Variable Embedding

Brendon Eby BEW1949
Harkirat Gill HSG7760
Justin Chae JHC2944
Igor Ryzhkov IRX3805

Abstract

In the field of Natural Language Processing (NLP), advances in neural probabilistic language such as those pioneered by (Yoshua Bengio, 2003) have been combined with a method of storing word context as vectors in embedding layers to improve performance in NLP tasks (Tomas Mikolov, 2013). In practice, researchers can use sequence models to learn an embedding or leverage pre-trained models to build context, but in either case, traditional deep learning architectures often rely on just a single embedding for each word. As a result, although sequence models such as the Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNN) are capable of producing state of the art results, they are constrained to understanding only a single meaning for a word when, in fact, words can have multiple meanings.

We find the constraint of singular embeddings to be unsatisfactory and in this paper, we present the results of an experiment wherein we construct language models with an LSTM RNN architecture and learn variable embeddings based on context. The purpose is to leverage the benefits of embeddings, but for multiple senses of a word, i.e., those words characterized by the concepts of polysemy, homonymy, or word sense disambiguation (Philip Edmonds, 2008). The goal is to optimize the perplexity of a language model without the assistance of pretrained embeddings.

For the experiment, we evaluated the Wikitext-2 corpus across four configurations of an LSTM RNN having various combinations of attention- and residual-like mechanisms and feed forward networks. For results, we measured performance in terms of perplexity and found that at least three out of four methods [1] of learning variable embeddings produce results that are statistically insignificant to a default architecture [2]. Additionally, we tested a fourth method with variable embedding that performs substantially worse than the default method [3]. Contrary to our initial motivation, our experiments show that a standard LSTM RNN with a single embedding layer performs just as well or better than architectures that learn variable embeddings. However, although our methods do not demonstrate marked improvement, we are intrigued as to the reasons why they fail to achieve performance results over the default.

***Keywords*—** statistical language modeling, artificial neural networks, variable word embedding, word sense disambiguation

1 Introduction

Advances in deep learning architectures, namely the combination of LSTM RNNs with word embeddings, have addressed two major challenges with sequence modeling. Firstly, LSTM RNNs enable networks to capture long-term dependencies in sequences that "standard" RNNs cannot achieve without modification. Secondly, when LSTM RNNs are configured to learn word embeddings, language modeling is greatly improved because the network considers both (1) context within the current sequence and (2) representative meaning relative to the rest of the corpus vocabulary.

However, for NLP tasks, the traditional LSTM RNN and other deep learning architectures are often deployed with a single embedding. With a single embedding model, the same embedding vector is considered for a word that may have multiple meanings or "senses". As a result, it may be the case that language model perplexity can be reduced with a properly configured variable embedding.

To illustrate the motivation for variable word embedding, consider a word within the context of two or more different sequences of words, or sentences. For instance, the word "bat" may refer to either (1) a wooden stick meant for hitting baseballs or (2) a flying mammal that lives in caves. Intuitively, predicting the next sequence of words after the word "bat" may turn on which embedding is used.

On the one hand, if the meaning of "bat" is meant to represent the wooden stick, then perhaps the next sequence of

[1] See section A.2 for detailed experiment results.

[2] Experimental configurations of att-emb, res-att-emb, and res-ff-emb perform worse on average as compared to default LSTM RNN with embedding layer (default) and fail one-tail test at 95% confidence interval.

[3] Experimental configuration of ff-emb performed worse than default model by 500%.

words should be related to the American sport of baseball.

> During baseball practice, the player swung the bat [and hit the ball]...

On the other hand, if the meaning of "bat" is meant to the flying mammal, then perhaps the next word should be "cave" as in a "bat cave."

> While investigating virus origins in the jungles and mountains, we explored a bat [cave]...

In either case, if a language model learns with both context and variable word meanings, then it may be able to understand multiple meanings of "bat" for NLP tasks. We should note, it is not lost on us that context alone in the "bat" example may be good enough for the task; however, the goal is to explicitly learn and use both embeddings.

2 Related Works

2.1 Sense2Vec

(Andrew Trask, 2015) recognized the shortcomings of relying on a single word embedding and introduced the sense2vec model as the solution. The sense2vec model is based on the work of (Eric H. Huang, 2012) which introduces the wang2vec model, a method of training "multi-sense" embedding with unsupervised clustering.

Whereas wang2vec requires multiple training passes, a resource-intensive process relative to a single pass, sense2vec asserts an efficiency improvement by leveraging supervised learning to eliminate both multiple training passes and clustering. Importantly, the sense2vec model demonstrates that modeling for multi-sense embedding, or variable embedding as we call it, can improve performance when evaluated in terms of error reduction.

2.2 Concatenate Multiple Pre-trained Embeddings

Whereas sense2vec and word2vec originate from either a supervised or unsupervised labeling approach, (Brian Lester, 2020) introduced the concept of "representational diversity" to solve the multi-sense problem. According to (Brian Lester, 2020), representational diversity results in an architecture that has both context from the LSTM and variable embedding from concatenated embeddings.

Importantly, the concatenation method demonstrates that having two or more pre-trained word embeddings with low similarity, i.e. "a high degree of coverage" can improve performance of a language model in terms of several types of tasks that include part-of-speech (POS) tagging and slot-filling.

3 Experiments

In our approach to the multi-sense problem, we seek to improve the perplexity of a language model without the use of pre-trained word embeddings. Instead, we use variations of the LSTM RNN and learn variable word embeddings during training. Although (Brian Lester, 2020) demonstrate that pre-trained embeddings work, multiple embeddings must be carefully selected to provide both (1) coverage over the corpus and (2) sufficient dissimilarity across the ensemble to yield performance improvements.

To overcome these shortcomings, our approach to variable embedding is intended to achieve the performance improvement that multiple word embeddings are capable of creating while customizing coverage for a given corpus. Furthermore, our approach focuses solely on perplexity as a measure of performance instead of error reduction by (Andrew Trask, 2015) or task-oriented performance by (Brian Lester, 2020).

3.1 Overview

To evaluate variable embeddings, we first establish a baseline understanding of how a traditional architecture performs with a singular embedding architecture. After establishing a baseline, we then evaluate varied embedding architectures while maintaining all other variables constant. The sole difference between models lies in the embedding transformation component before the input reaches the LSTM cell.

To evaluate the performance result of each variable architecture against the baseline, we measure performance with perplexity, Equation 1. While perplexity is not the only tool to measure performance, it is both a common metric and something that provides an easy measure of relative performance for the purposes of this experiment. To implement perplexity, we compute loss with the PyTorch cross-entropy loss function and then calculate the exponential of the mean of the losses (Contributors, 2019).

$$perplexity(LM) = exp\left(\sum_{n=0}^{N} \log(p_n)\right) \tag{1}$$

3.2 LSTM Architecture Baseline

All of our models for this experiment rely on the LSTM RNN which is based on RNN Regularization, a concept pioneered by (Wojciech Zaremba, 2015). In an LSTM, two context vectors are maintained between each token, a "short term memory" vector $\vec{h_n}$ that reflects recent information, and a "long term memory" vector $\vec{c_n}$ that maintains states for longer periods.

The "short term" memory works as it would in a regular RNN, where a hidden state is updated after each token is passed through the network. Afterwards, the output is passed as a parameter to the model for the next token. The "long term" portion of the LSTM maintains a state that has information added and removed via "gates" after each pass.

Our LSTM architecture includes an embedding layer followed by two stacked LSTM cells, each with dropout applied. Our prediction for the next word in the sequence is formed by taking the dot product of the output from the second LSTM cell and the initial embedding space of our vocabulary. These core configurations remain consistent through each variation of our model.

For simplification, from here on we will refer to the 2 LSTM layers a single layer, and assume that the vectors $\vec{h_n}$ and $\vec{c_n}$ include the context states from both LSTM cells. Our predictive model will therefore have an equation like:

$$model(t_n) = LSTM(u(t_n), \vec{h}_{n-1}, \vec{c}_{n-1}) \cdot v(t_n) \tag{2}$$

where

$$v(t_n) = Embedding(Token_n) \tag{3}$$

$$u(t_n) = VariableEmbedding(Token_n) \tag{4}$$

Each model variation in this section will use the architecture from equation (2), with an distinct definition for u.

3.3 Default Model

In the baseline model (default), we simply pass our word embeddings directly into the LSTM cells, such that:

$$u(t_n) = v(t_n) \tag{5}$$

The results of the default model are used to evaluate architecture adjustments in each subsequent model. The default model has the architecture seen in figure 1.

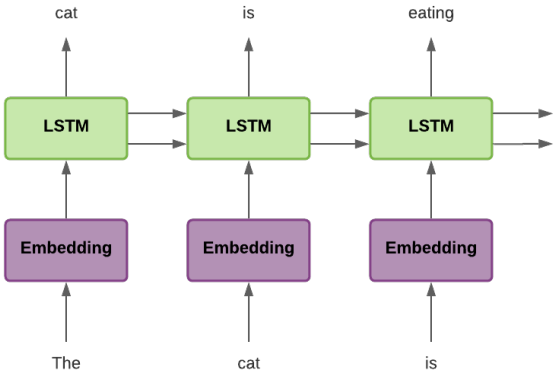

Figure 1: LSTM with embedding (default).

3.4 LSTM with Embedding and Feed Forward

In the feed forward with embedding model (ff-emb), instead of feeding the output of our embedding layer directly into the LSTM, we first pass it through a simple feed forward network F which concatenates the word embedding with the $\vec{h_n}$ and $\vec{c_n}$ LSTM outputs from the previous token in our sequence. The output of the feed forward network is the same size as the output from our embedding layer, see equation 6.

$$v(t_n) = F(\left(u(t_n),\ \vec{h}_{n-1},\ \vec{c}_{n-1}\right)) \tag{6}$$

The general idea with a feed forward network is that the network should learn to extract more appropriate meanings from a word embedding, when combined with the context in which a word appears. As a result, for the multi-sense problem, this architecture should be especially beneficial for words that have multiple meanings.

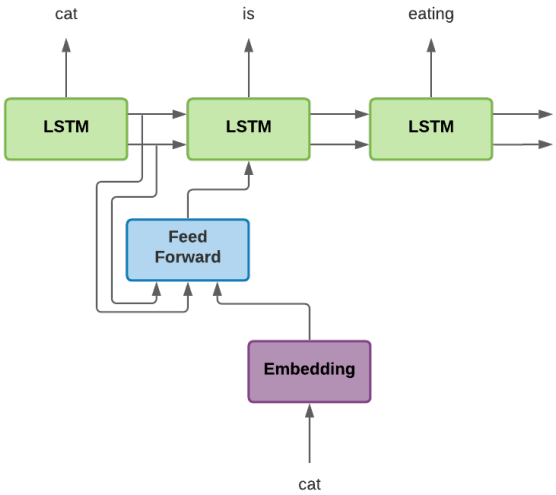

Figure 2: Feed forward model with embedding (ff-emb).

As the saying goes, the correct meaning of a word should be discernible by "the company it keeps" (acl, 2021). Moreover, according to a commonly accepted notion, this is how humans tell the difference between a word's possible interpretations as well (Justin Garten, 2019). The ff-emb architecture is given by figure 2.

An important distinction for this ff-emb architecture is that it is the only model that forces all tokens through the variable embedding layer. All other models in our experiment allow some form of bypass.

3.5 LSTM with Attention and Embedding

In this model, we introduce an attention-like mechanism as an alternative way of accounting for multiple senses of a word. The idea behind att-emb is that instead of generating new embeddings directly from the feed forward layer, we can instead use that output to choose what parts of a word's existing embedding are most relevant, given their context.

In this configuration, the variable encoding function is given by equation 7.

$$v(t_n) = F(\left(u(t_n),\ \vec{h}_{n-1},\ \vec{c}_{n-1}\right)) \cdot u(t_n) \tag{7}$$

The output the our feed forward network with attention and embedding uses a tanh activation function, ensuring that the resulting vector values are all between -1 and 1. We interpret the result of the activation function as a weighting factor for each embedding dimension. As a result, what we feed into the LSTM is the dot product of this weighting factor output and our initial word embedding, see figure 3.

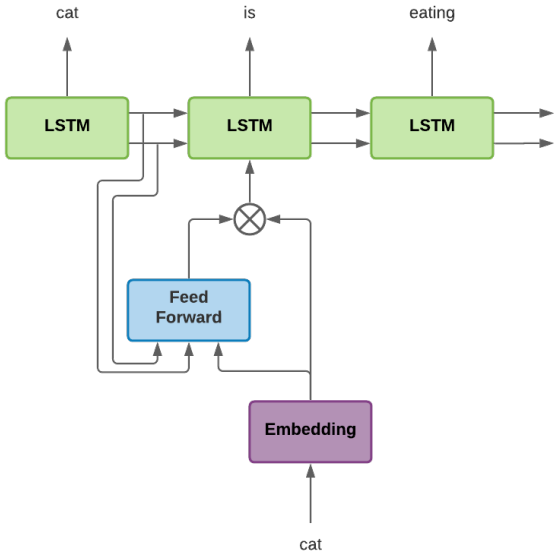

Figure 3: Attention model with embedding (att-emb).

3.6 LSTM with Feed Forward, Embedding, and Residuals

In the ff-res-emb model, we introduce "residuals" as a variation of the LSTM with attention and embedding. The intent with residuals is to provide the LSTM cells with both our context-modified feed forward embeddings and our initial embedding vector for a given word.

By accounting for residuals, we allow the LSTM to use both information contained in the original embedding and any additional information gained from context. Additionally, residuals allow the feed forward network to learn only what

additional information can be gained from context instead of needing to include information contained in the original embeddings.

For the ff-res-emb model, before passing the output into the LSTM, the output from the feed forward layer is concatenated to the output from the embedding layer, see equation 8.

$$v(t_n) = \Big(F(\ \big(u(t_n),\ \vec{h}_{n-1},\ \vec{c}_{n-1}\big)\ ), u(t_n)\Big) \quad (8)$$

As both output vectors have a dimension equal to the size of the embedding space, the LSTM input will therefore have a size twice that of the embedding space, see figure 4.

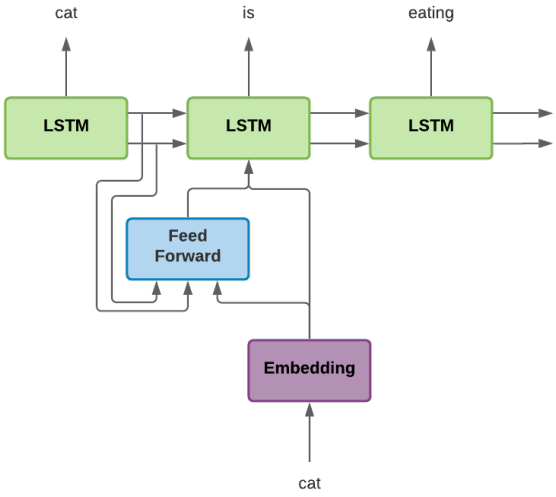

Figure 4: Feed forward model with embedding and residuals (ff-res-emb).

3.7 LSTM with Attention, Embedding, and Residuals

In the res-att-emb model, we combine all of the concepts introduced in previous models to include the feed forward network, attention, and residuals. The intent of the res-att-emb model is to provide the LSTM with the information contained in the original embedding along with those embeddings scaled along each embedding direction based on context.

$$v(t_n) = \Big(F(\ \big(u(t_n),\ \vec{h}_{n-1},\ \vec{c}_{n-1}\big)\ ) \cdot u(t_n), u(t_n)\Big) \quad (9)$$

The embedding outputs are fed through the feed forward network along with the context vectors to provide values -1 to 1 along each dimension. The dot product of the context vectors and the original embedding vector forms the attention output which is concatenated to the original embedding vector and passed into the LSTM, as shown in figure 5. As in the previous model, the LSTM input has a size of twice that of the embedding space.

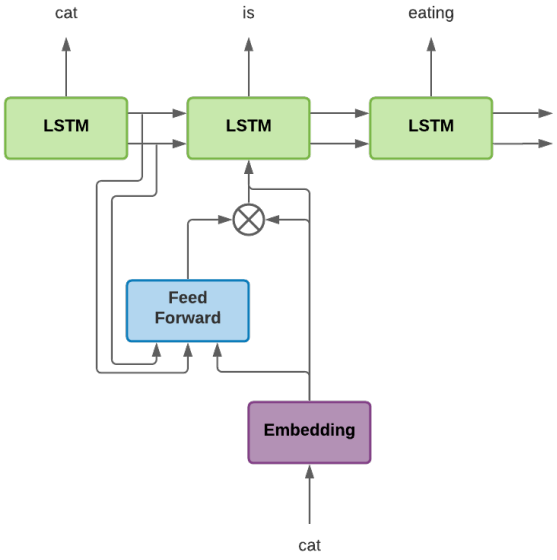

Figure 5: Attention model with embedding, attention, and residuals (att-res-emb).

4 Data

Because the original intention behind allowing for variable embeddings was so the language model can better consider polysemy and homonymy, we expected to see the benefit of that approach in texts where there is a higher occurrence of this phenomenon. However, since we did not find any corpus where we believed there would be a higher occurrence of polysemy and homonymy, we used the generic "Wikitext-2" corpus, expecting to still see some improvement reflected in the perplexity.

The version of Wikitext-2 that we used retained grammatical punctuation and numbers, but already had rare words mapped to the `<unk>` token. We further process the data in the text cleaning part of our pipeline in

`stage_wikipedia_text_cleaning.py`.

The script converts everything to lowercase, removes unusual characters including non-ascii, and applies lemmatization. After processing we get a total of 2,585,666 tokens, with 25,452 unique tokens from 753 articles. The following table shows these statistics for the train, validation, and test set.

Data	Fraction	Total	Unique	Articles
Train	.8	2.1m	25K	631
Valid	.1	220K	10K	60
Test	.1	250K	11K	62

Table 1: Summary statistics of evaluation data for train, validation, and test data; "TOTAL" and "UNIQUE" refer to number of tokens; "Articles" refer to number of articles.

5 Experimentation

We experimented with different configurations to see if there was any effect on the relative performance of the various models. Some things we tried included (1) lemmatizing our

vocabulary or not, (2) adjusting the embedding size, (3) changing the number of hidden layers in our feed forward network, (4) different numbers of LSTM cells, (5) using a feed forward network for outputs vs using the embedding space, (6) different step sizes, (7) smaller learning rates, longer training times, and different learning rate decays.

Although we noticed a change in overall perplexity with those different configurations, we saw the same distribution of performances between the models. Our final model includes the configuration that shows the best overall performance, after experimentation.

5.1 Model Configuration

Model	Model Parameters
batch size	100
embedding size	100
max norm	4
max init param	0.1
num layers	2
sequence length	30
dropout probability	0.1

Table 2: Model Parameters

5.2 Hardware

To run variable embedding experiments, we cloned the project repository from GitHub to a Google Colab notebook where we used either a Tesla P100-PCIE-16GB or Tesla V100-SXM2-16GB. For more see appendix section A.1 for reproducibility.

6 Discussion, Results, and Analyses

Although we expected improvement in perplexity with some of these architectures, all our experimental models performed about the same or worse than the default model. In the case of the ff-emb model, perplexity was significantly worse than the default by more than 500%. In addition, training the default model was much faster than any of our other models. The performance pattern emerged in all of the experimental configurations described in section 5.

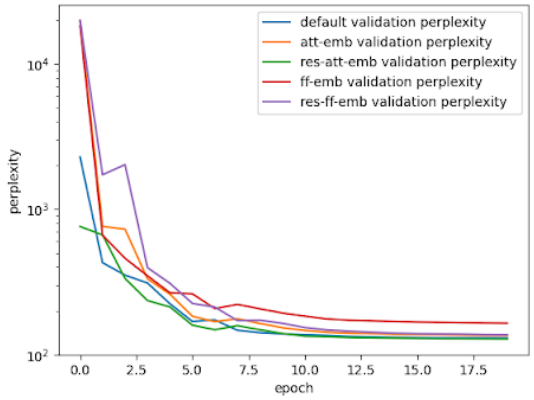

Figure 6: Consolidated perplexity results.

As to why there was no improvement in perplexity from variable embeddings, we have two ideas.

Our preliminary conclusion is that the experimental models are simply learning to ignore the comparatively noisy variable embeddings when possible and instead rely exclusively on the initial fixed embeddings for each token. In the cases where we use the attention-like mechanism, the weights are being adjusted to let fixed embeddings go through un-modified. For the models using residuals, the model learns to rely only on those values, and ignore the variable component. For ff-emb, this is more difficult as the entire network must learn to let only the initial embeddings through. We believe this its inability to quickly learn to bypass our variable embeddings is the reason that ff-emb has such poor performance.

Table 3 Results for test perplexity.[4]

Model	Avg Test Perplexity
default lstm	104.77
att-emb	105.52
att-res-emb	105.86
ff-emb	680.47
ff-res-emb	104.88

Table 3: Test perplexity for different models.

There may still be value to learning variable embeddings, but the gradient signal may just be too noisy in our configuration for that to be effectively learned. Traditional static embeddings are adjusted during training every time a word appears, but here we are trying to simultaneously learn both the static embedding for each word, and every variation of that embedding for every context in which it appears. Essentially, learning embeddings+context may be a goal several orders of magnitude more complex than learning embeddings alone. That may simply be too granular of a task, or too much to learn simultaneously for this configuration.

It is possible that a much larger corpus and longer training time could result in better performance. Or beginning with a pre-trained embedding space and/or LSTM and isolating the variable embedding layer could allow it to learn variable representations more effectively. Those are things that could be interesting to explore in a future project.

7 Conclusion and Future Work

We presented four ways to learn variable embeddings with an LSTM RNN as a method to tackle the multi-sense word problem. While traditional approaches combine LSTM RNNs with a single embedding, our approach with variable embeddings attempts to account for situations where a word has multiple meanings, i.e., polysemy or homonymy. Moreover, we attempt to fill a void left by other approaches to the multi-sense problem by learning the embeddings during training instead of using pre-trained word embeddings.

We started this investigation with a theory that having multiple embeddings should improve model performance in terms of language model perplexity; however, our results do not support the theory. Instead, according to a one-tailed test, all configurations returned perplexity results that were statistically insignificant to the default model. The only exception was our ff-emb configuration which performed markedly worse than the default by over 500%, and was also the only model we tested that lacked an easy way to revert to a static embedding representation. Said in another way, LSTM RNNs with so-called "vanilla" embedding layers are still relatively state of the art, or close to it, in language modeling.

[4] Average results across 10 runs.

There are several distinct areas open for investigation. First, we are intrigued as to why these particular methods failed and seek to develop hybrid methods of embedding. We are also interested in exploring different training paradigms and corpuses, and whether beginning with pretrained embedding space may help with our training process. Finally, we are interested in applying the idea of variable embeddings to a deep learning architecture composed of transformers instead of LSTM RNNs.

References

2021. Distributional hypothesis, association for computational linguistics wiki.

John Liu Andrew Trask, Phil Michalak. 2015. sense2vec - a fast and accurate method for word sense disambiguation in neural word embeddings. arXiv:1511.06388 [cs.CL].

Amy Hemmeter Sagnik Ray Choudhury Srinivas Bangalore Brian Lester, Daniel Pressel. 2020. Multiple word embeddings for increased diversity of representation. arXiv:2009.14394 [cs.CL].

Torch Contributors. 2019. Crossentropyloss.

Christopher D. Manning Andrew Y. Ng Eric H. Huang, Richard Socher. 2012. Improving word representations via global context and multiple word prototypes. volume 1, Pages 873–882.

Kenji Sagae Morteza Dehghani Justin Garten, Brendan Kennedy. 2019. *Measuring the importance of context when modeling language comprehension*.

Eneko Agirre Philip Edmonds. 2008. Word sense disambiguation.

Greg Corrado Jeffrey Dean Tomas Mikolov, Kai Chen. 2013. Efficient estimation of word representations in vector space. arXiv:1301.3781v3 [cs.CL], url = https://arxiv.org/abs/1301.3781,.

Oriol Vinyals Wojciech Zaremba, Ilya Sutskever. 2015. Recurrent neural network regularization. arXiv:1409.2329v5 [cs.NE].

Pascal Vincent Christian Jauvin Yoshua Bengio, Réjean Ducharme. 2003. A neural probabilistic language model. *JMLR*, Volume 3.

A Appendices

A.1 Reproducibility

We have made this project available in its entirety. To reproduce the experiments, there are two options.

The first option is to clone our GitHub repository from the following URL: https://github.com/iryzhkov/nlp-pipeline/tree/nn_embedding.

```
git checkout <URL>
```

Our project on variable embedding is contained in a branch of the main project called "nn_emebdding". To switch to variable embedding:

```
git checkout nn_embedding
```

After cloning the project to your local computer and selecting the branch, install dependencies and run the project.

```
make install
make model-given-data
```

The second option is to run this project through a web browser with Google Colab at the following URL: https://colab.research.google.com/drive/13EOfZHtKrgWWwHnF8rGUOewVtwND9Ns-?usp=sharing.

A.2 Detailed Experiment Results

A full listing of the experiment results and statistical significance test can be found in a Goolge Sheets worksheet at the following URL: https://docs.google.com/spreadsheets/d/1xWR_LA1LlpdEApbcNU5qe34wGCjjPweQTGbTHs4IIjg/edit?usp=sharing