# Leveraging Data Augmentation to Evaluate Semantic Sentence Similarity

**Vikram Shetty** [1 2] **Arun Ravi** [1 2]

## 1. Introduction

One of the key challenges for any complex machine learning task is to gather sufficient good quality representative data. This problem is magnified for natural language applications due to the amount of human time that is required to annotate and process data. One way to circumvent this problem is to use data augmentation to artificially increase the size of the available dataset for better test case generalization. Data Augmentation in vision applications is widely known to perform extremely well, but the same isn't as trivial to implement in NLP.

In this project we will be comparing the efficacy of various data augmentation techniques while evaluating semantic sentence similarity. We expect primitive data augmentation techniques like synonym replacement, random insertions and deletions to perform poorly as compared to Mixup for Text.

This report is divided into 5 sections, the section on Background gives a detailed description on the technology and models we have employed in our projects - this includes Word2Vec, the MaLSTM model and data augmentation techniques. The next section on Experimental setup talks about training details and how we have implemented the sub-modules in our project. The following section on Results discusses how good the generated word embeddings are, how well the Siamese Architecture performs and how effective data augmentation is. The final section provides the overall conclusion for the project.

## 2. Background

### 2.1. Word2Vec

As humans we understand lexical and semantic differences between different words in the English language. However, making a computer understand such intricacies is a challenging task. A very simple and naive solution would be to represent every word in the vocabulary as a one hot encoded vector, but this encoding scheme wouldn't capture semantic similarities or differences between words. Another reason for the failure of such a scheme is the length of the word representations. For example, if we had 3 million unique words in our text corpora, each word would be represented with a vector of length 3 million!

We can establish that we need word encodings that are lower dimensional, so its feasible to operate on such vectors in large scale NLP applications. We also want these vectors to capture relationships between words. For example, embeddings of similar words should be closely inclined with one another, while embeddings of antonyms should be orthogonal to one another. This is where Word2Vec has proven to be extremely useful (2).

Word2Vec is a two-layer neural network that processes text by vectorizing words. The input to the network is a text corpus and the output is a set of feature vectors that represent words in that corpus. While Word2Vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand. These vectors constructed by Word2Vec have two very desirable properties - a) we can encapsulate vocabularies with as many as 3 million words into only 300 dimensional word embeddings b) the cosine similarity between similar words is high and that between dissimilar words is low. Word2Vec can be implemented in two distinct ways - The Skip-Gram Model or the Continuous Bag of Words (CBoW) model. In this project, we have used the Skip Gram Model.

The Skip-Gram Model is a single layer neural network that predicts the context word given an input word. However, for generating word embeddings, we are not actually interested in the output of the model, rather we are interested in the weight matrix between the input and hidden layer. This weight matrix has $V$ rows and $d$ columns, where $V$ is the total number of words in the vocabulary and $d$ is the number of units in the hidden layer (also the dimensions of our word vectors). The $i^{th}$ row in this weight matrix is a $d$ - dimensional word embedding corresponding to the $i^{th}$ word in the vocabulary.

### 2.2. Manhattan LSTM Model

The Manhattan LSTM (MaLSTM) model (10) is outlined in *Figure 1* (10). The model consists of two LSTMs - $\text{LSTM}_a$ and $\text{LSTM}_b$ - each of which process one sentence in an input

[1]Equal Contribution [2]University of Wisconsin-Madison. Correspondence to: Vikram Shetty <vikram.shetty@wisc.edu>, Arun Ravi <arun.ravi@wisc.edu>.

*CS 760 Project Report, Fall 21*,

pair. In this network, the pair of LSTMs have tied weights such that $\text{LSTM}_a = \text{LSTM}_b$. Each sentence is tokenized into words and the word embeddings are sent into their respective LSTMs. The final hidden state generated by the LSTM is a vector representation for the sentence. Subsequently, the similarity between these representations is used as a predictor of semantic similarity.

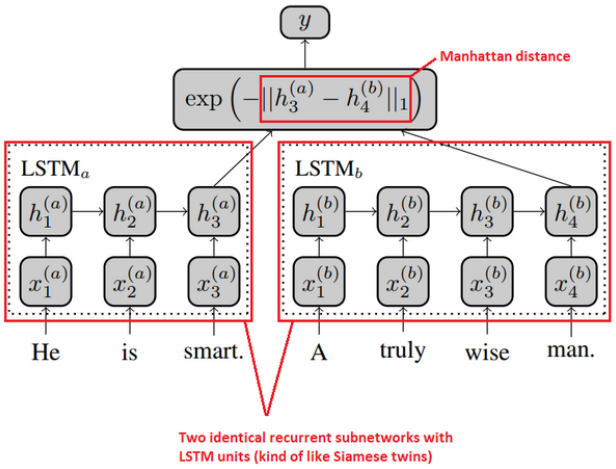


*Figure 1.* The Siamese Network

The LSTM learns a mapping from a $d_{in}$ - dimensional vectorspace into $R^{d_{rep}}$, where $d_{in}$ is the length of the word embedding, and $d_{rep}$ is the length of the sentence embeddings ($d_{in}$ = 300, $d_{rep}$ = 50 in this work). Each sentence (represented as a sequence of word vectors) is passed to the LSTM, which updates its hidden state at each sequence-index. The final representation of the sentence is encoded by the last hidden state of the model. For a given pair of sentences, the model applies a similarity function that maps from $R^{d_{rep}}$ to $R$. This similarity function is used to infer the sentences' underlying semantic similarity (10).

The error signal that needs to be backpropagated through this network has to be a function of the final hidden state representations of the two LSTM networks. We denote these final hidden states as $h_{T_a}^{(a)}$ and $h_{T_b}^{(b)}$. The authors in (10) use the Manhattan Norm in the exponent of the similarity function -

$$g(h_{T_a}^{(a)}, h_{T_b}^{(b)}) = exp(-||h_{T_a}^{(a)} - h_{T_a}^{(a)}||_1) \quad (1)$$

The authors in (4) point out that using the $L_2$ norm can end up causing undesirable plateaus in the objective function, which isn't the case with the $L_1$ norm. The authors explain that this happens because an $L_2$ based model is unable to correct errors where the network erroneously classifies semantically disparate sentences as identical due to vanishing gradients caused by the $L_2$ norm. Hence, the authors in (10) have also preferred the $L_1$ over the $L_2$ norm and further claim that it outperforms other reasonable similarity metrics like the cosine distance.

### 2.3. Data Augmentation

Data augmentation is a technique used to increase the amount of data available for machine learning models to train on. Data augmentation helps reduce overfitting during training by acting as a regularizer. It can be achieved in two ways -

1. Inserting modified pre-existing data into the train set.
2. Creating synthetic data samples from existing data.

Data augmentation is also very useful if we already have a large quantity of data. In this case, data augmentation will increase the number of relevant data samples in your dataset. For example, say we have a large dataset with images from two classes of cars - A and B. Suppose, all images of car A face to the left, and all images of car B face towards the right. Now if we input a test image of car A facing towards the right, then our machine learning model will have a high chance of predicting car B. In this case, if we augment our train set with flipped versions of all images, this error can be mitigated.

Data augmentation techniques for computer vision applications are intuitive and well established (flipping, cropping, rotation, zooming and random noise injections). However, the same cannot be said for natural language processing tasks. One major reason for this is that in natural language tasks, the grammatical structure of data needs to be preserved. Thus data needs to be manipulated such that inherent semantics do not change. In this project we focus on three paradigms for NLP data augmentation - Easy Data Augmentation (EDA), Back-Translation and Mixup.

#### 2.3.1. Easy Data Augmentation

Random Deletion - This is the process of randomly deleting a word from a sentence and inserting the new sentence into the dataset.

Random Insertion - This is the process of randomly inserting a word into a sentence and inserting the new sentence into the dataset. Random Insertion can seen as the equivalent to random noise insertion in computer vision applications.

Synonym Replacement - This is the process of randomly replacing a word from a sentence with a synonym and inserting the new sentence into the dataset. Since we are replacing a word with its synonym, the context of the sentence will not change.

#### 2.3.2. BACK-TRANSLATION

Here we translate the sentence into a different language, and this translated sentence is back-translated into the original language. This back-translated sentence is added to the dataset. An example of back-translation is presented in *Figure 9* (3).

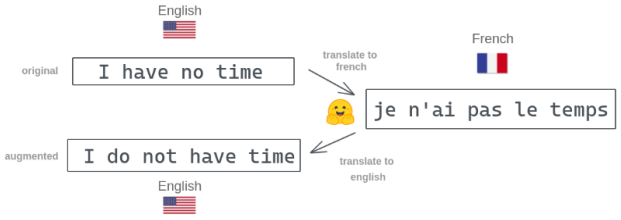


*Figure 2.* Example of Back-Translation

#### 2.3.3. MIXUP

The idea of using Mixup for data augmentation in an image classification task was first proposed in (12). The idea was to generate a synthetic image by linearly combining the pixel values of two random images and adding the generated image to the dataset. Consider two images $(x_i, Y_i)$ and $(x_j, Y_j)$, where $x$ is the input and $Y$ is the output. Then the new data sample obtained on using mixup is as follows -

$$\bar{x} = \lambda x_i + (1 - \lambda) x_j \tag{2}$$

$$\bar{Y} = \lambda Y_i + (1 - \lambda) Y_j \tag{3}$$

where $\lambda$ is the mixing policy for the two data samples. $\lambda$ is generated from a beta distribution with parameter $\alpha$.

While this method works well for images, it cannot directly be applied to a sequence of words or sentences. In NLP tasks which deal with sentences, a sentence is first converted into a sequence of word embeddings, and then fed into a sentence encoder. The sentence encoder outputs a sentence embedding which can be used for any subsequent tasks involving the sentence. Here to use mixup, we need to work with either the word embeddings (wordMixup) or sentence embeddings (senMixup) (6).

While using wordMixup, all sentences are zero padded to the same length. Consider two sentences $S_1$ and $S_2$. Then the wordMixup is formulated as -

$$S^i_{new} = \lambda S^i_1 + (1 - \lambda) S^i_2 \tag{4}$$

$$Y_{new} = \lambda Y_1 + (1 - \lambda) Y_2 \tag{5}$$

where $S^i$ is the word embedding of the $i^{th}$ word in the sentence, $\lambda$ is the mixup-ratio, and $Y$ is the corresponding label to the sentence (6).

In senMixup, the sentence embeddings are first generated using a sentence encoder, and a linear combination of these embeddings are derived. Consider two sentences $S_1$ and $S_2$. Then the senMixup is formulated as -

$$S^k_{new} = \lambda f(S_1)^k + (1 - \lambda) f(S_2)^k \tag{6}$$

$$Y_{new} = \lambda Y_1 + (1 - \lambda) Y_2 \tag{7}$$

where $k$ is the dimension of the sentence embedding, $\lambda$ is the mixup-ratio, $Y$ is the corresponding label to the sentence, and function $f$ is the sentence encoder. In our project we only explore wordMixup (6).

## 3. Experimental Setup

### 3.1. word2vec Training

The skip-gram network (11) is trained by feeding it word pairs found in the training corpus. *Figure 3* (1) shows an example of a training sample that is fed into the network. The input is generally called the target word, while the output is generally called the context. The trained skip-gram network will predict the context given the target word.

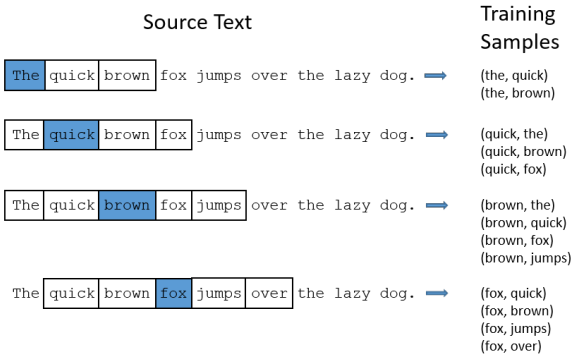


*Figure 3.* Generating Training Data

For our project we have three distinct variants of word2vec - our own implementation from scratch, using Gensim's skip-gram training module, and using pre-trained word vectors from the google-news dataset. Our implementation of word2vec was trained on the SICK dataset. The details of our own implementation of word2vec are presented below.

#### 3.1.1. SKIP-GRAM TRAINING

*Figure 4* (11) shows the Skip Gram Model. Let $x$ denote the one-hot encoded version of the input word passed to the Skip-Gram network. Then $h = W^T x = v^T_{W_I}$ gives us the corresponding word embedding of the input word. Note that $W$ here is the input→hidden weight matrix and $v_{W_I}$ is the input vector representation of the word $w_I$, which is also

the row indexed by $x$ in $W$. Since $x$ is a one hot encoded vector, $h$ is simply copying (and transposing) a row of the input→hidden weight matrix, $W$, associated with the input word $w_I$.

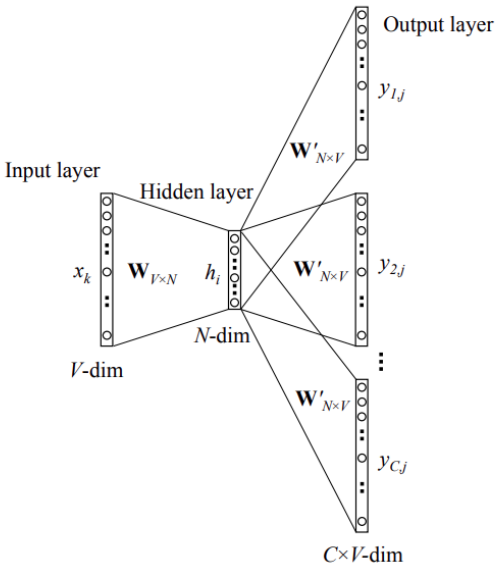


*Figure 4.* The Skip Gram Model

The output of the network consists of $C$ multinomial distributions ($V$ - dimensional), where $C$ is the number of context words in our input sample, each computed using the same hidden→output matrix:

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{exp(u_{c,j})}{\sum_{j'=1}^{V} exp(u_{j'})} \quad (8)$$

where $w_{c,j}$ is the word corresponding to the $j^{th}$ unit on the $c^{th}$ panel of the output layer, $w_{O,c}$ is the $c^{th}$ word in the set of output context words, $w_I$ is the input word; $y_{c,j}$ is the output of the $j^{th}$ unit on the $c^{th}$ panel of the output layer; $u_{c,j}$ is the net input of the $j^{th}$ unit on the $c^{th}$ panel of the output layer.

Since the output layer panels share the same weights -

$$u_{c,j} = u_j = v_{w_j}^{'T} * h \; for \; c = 1, 2, ....C \quad (9)$$

where $v_{w_j}^{'}$ is the output vector of the $j^{th}$ word in the vocabulary, and also $v_{w_j}^{'}$ is taken from a column of the hidden→output weight matrix, $W^{'}$.

The training objective (for one training sample) is to maximize the conditional probability of observing the actual output word $w_O$ (denote its index in the output layer as $j^*$) given the input word $w_I$. The loss function is given by

$$E = -logp(w_{O,1}, w_{O,2}, ...., w_{O,C}|w_I) \quad (10)$$

$$E = -log \prod_{c=1}^{C} \frac{u_{c,j_{c^*}}}{\sum_{j'=1}^{V} exp(u_{j'})} \quad (11)$$

$$E = -\sum_{c=1}^{C} u_{c,j_{c^*}} + Clog \sum_{j'=1}^{V} exp(u_{j'}) \quad (12)$$

Taking the derivative of $E$ with regard to the net input of every unit on every panel of the output layer, $u_{c,j}$, we obtain

$$\frac{\partial E}{\partial u_{c,j}} = y_{c,j} - t_{c,j} := e_{c,j} \quad (13)$$

which is the prediction error on the unit.

Lets define a $V$ dimensional vector $EI = \{EI_1, EI_2, ...., EI_V\}$ as the sum of prediction errors over the entire context - $EI_j = \sum_{c=1}^{C} e_{c,j}$. Next, we differentiate $E$ w.r.t to the hidden→output matrix $W^{'}$ and using the chain rule we can easily see that

$$\frac{\partial E}{\partial w_{ij}^{'}} = \frac{\partial E}{\partial u_{c,j}} \frac{\partial u_{c,j}}{\partial w_{ij}^{'}} = EI_j * h_i \quad (14)$$

Thus, we finally obtain the update equation for the hidden→output matrix as -

$$w_{ij}^{'(new)} = w_{ij}^{'(old)} - \eta * EI_j * h_i \quad (15)$$

and consequently -

$$v_{w_j}^{'(new)} = v_{w_j}^{'(old)} - \eta * EI_j * h \quad (16)$$

The derivation of the update equation for the input→hidden matrix is identical to the steps taken in 14 to 16. The update equation is given by -

$$v_{wI}^{new} = v_{wI}^{old} - \eta * EH^T \quad (17)$$

where $EH$ is an $N$ dimensional vector, each component of which is defined as -

$$EH_i = \sum_{j=1}^{V} EI_j * w_{ij}^{'} \quad (18)$$

#### 3.1.2. Optimizing Computational Efficiency

While training word2vec for the first few times, we realized the process was taking a prohibitive amount of time, even on medium sized datasets. We realized this is because, for the Skip-Gram model we discussed above, there exist two vector representations for every word in the vocabulary: the input vector $v_w$ and the output vector $v^{'}_w$. Learning the input vectors is cheap, but learning the output vectors turns out to be computationally very expensive. From update equation 15 we can see that in order to update $v^{'}_w$, we need to iterate through every word in our vocabulary, compute the net input $u_j$, probability prediction $(y_{c,j})$, their prediction error $(EI_j)$ and then finally update the output vector $v^{'}_j$.

While trying to optimize our network we came across (9) which proposed two strategies to reduce training time for Word2Vec and also claimed to generate superior word embeddings - Subsampling and Negative Sampling - both of which we have implemented.

#### 3.1.3. Subsampling of Frequent Words

If we take a look at *Figure 3* again, we see the problems with common stop words like "the" -

1. When looking at word pairs ("fox", the"), "the" doesn't tell us much about the meaning of "fox". "the" appears in the context of pretty much every word.
2. We will have many more samples of ("the",...) than required to learn a good vector representation for "the".

Word2Vec implements a "subsampling" scheme to address this. For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we cut the word is related to the word's frequency. If we have a window size of 10, and we remove a specific instance of "the" from the current window -

1. None of the other words within the window will have "the" in their contexts.
2. We will have 10 fewer training samples where "the" is the input word.

We thus calculate the probability of retaining every word in our corpus before generating training instances. Let $z(w_i)$ represent the fraction of total words in the corpus that are the word $w_i$. The authors in (9) have calculated the probability of retaining a word as -

$$P(w_i) = (\sqrt{\frac{z(w_i)}{0.001}} + 1) * \frac{0.001}{z(w_i)} \tag{19}$$

The plot of *Equation 19* is as shown in *Figure 5*. We can make out the following interesting facts from the plot -

- $P(w_i) = 1.0$ (100% chance of being kept) when $z(w_i) <= 0.0026$. This means that only words which represent more than 0.26% of the total words will be subsampled.
- $P(w_i) = 0.5$ (50% chance of being kept) when $z(w_i) = 0.00746$.
- $P(w_i) = 0.033$ (3.3% chance of being kept) when $z(w_i) = 1.0$. That is, if the corpus consisted entirely of word $w_i$, which is purposeless.

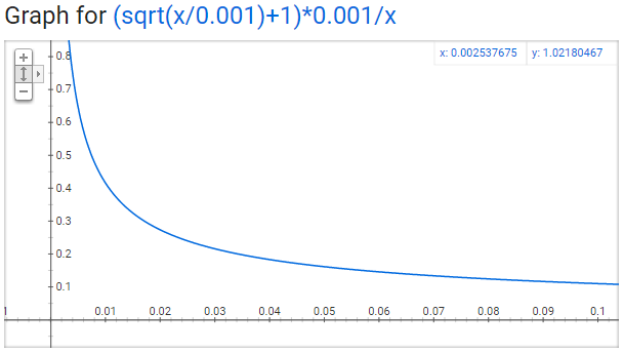


*Figure 5.* Subsampling Commonly Occurring Words *(8)*

#### 3.1.4. Negative Sampling

The idea behind negative sampling is simple - in order to to deal with the difficulty of having to update too many output vectors per iteration, we update only a tiny subset of them (9). The output vectors corresponding to the output word (i.e., the ground truth, or positive sample) along with the negative samples generated are the only ones updated. The probabilistic distribution needed to generate these negative samples is called the noise distribution denoted by $P_n(w)$. The authors in (9) argue that using the following simplified objective for training produces high quality word vectors -

$$E = -log\sigma({v^{'}_{wO}}^T * h) - \sum_{w_j \epsilon W_{neg}} log\sigma(-{v^{'}_{w_j}}^T * h) \tag{20}$$

where $W_{neg} = \{w_j | j = 1, 2, ..., K\}$ is the set of words sampled from the noise distribution $(P_n(w))$ i.e. the negative samples, and the remaining symbols are the same as described in *section* 3.1.1

The authors in (9) claim the following noise distribution is ideal to sample negative samples -

$$P_n(w_i) \sim \frac{c(w_i)^{\frac{3}{4}}}{\sum_{j=1}^{n} c(w_j)^{\frac{3}{4}}} \tag{21}$$

where $c(w_i)$ denotes the unigram count of a word $w_i$ in our corpus.

To obtain the update equations of the word vectors under negative sampling, we first take the derivative of $E$ with regard to the net input of the output unit $w_j$ :

$$\frac{\partial E}{\partial {v'_{w_j}}^T h} = \begin{cases} \sigma({v'_{w_j}}^T) - 1 \; if \; w_j = w_O \\ \sigma({v'_{w_j}}^T) \; if \; w_j \; \epsilon \; W_{neg} \end{cases}$$

$$\frac{\partial E}{\partial {v'_{w_j}}^T h} = \sigma({v'_{w_j}}^T h) - t_j \tag{22}$$

where "$t_j$" is the label of the word $w_j$ - $t = 1$ when $w_j$ is a positive sample, otherwise $t = 0$. Now we take the derivative of $E$ w.r.t the output vector of the word $w_j$,

$$\frac{\partial E}{\partial v'_{w_j}} = \frac{\partial E}{\partial {v'_{w_j}}^T h} \frac{\partial {v'_{w_j}}^T h}{\partial v'_{w_j}} = (\sigma(v'_{w_j} h) - t_j)h \tag{23}$$

which results in the following equation for the output vector

$$v'^{new}_{w_j} = v'^{old}_{w_j} - \eta(\sigma(v'_{w_j} h) - t_j)h \tag{24}$$

which only needs to be applied to $w_j \; \epsilon \; w_O \cup W_{neg}$ instead of every word in the vocabulary. This shows why we save a significant amount of computational effort every iteration. *Equation 24* needs to be applied one context word at a time for the Skip Gram model.

To backpropagate the error to the hidden layer and thus update the input vectors, we need to differentiate $E$ w.r.t $h$ -

$$\frac{\partial E}{\partial h} = \sum_{w_j \epsilon \{w_O\} \cup W_{neg}} \frac{\partial E}{\partial {v'_{w_j}}^T h} \frac{\partial {v'_{w_j}}^T h}{\partial h} \tag{25}$$

Simplifying this further, we have

$$\frac{\partial E}{\partial h} = \sum_{w_j \epsilon \{w_O\} \cup W_{neg}} (\sigma({v'_{w_j}}^T h) - t_j) {v'_{w_j}}^T := EH \tag{26}$$

For the Skip Gram model, we need to calculate a $EH$ value for each word in the Skip Gram context, and plug the sum of the $EH$ values into 17 to obtain the update equation for the input vector.

### 3.2. Data Augmentation for Sentence Similarity

The canonical data set used for comparing semantic sentence similarity is the SICK data set (7). The dataset consists of 9840 sentence pairs, and each pair of sentences is accompanied with a similarity score (between 1 and 5) corresponding to the average similarity judged by 10 different individuals. We are dividing the data into a 80-20 train test split.

Since LSTM's consist of a large number of parameters, we require a very large amount of data to make sure the LSTM is trained fully. Thus it becomes a necessity to augment the SICK data set with more data (we are only augmenting the training data, and are not modifying the test data). The results of the various data augmentations are given below.

#### 3.2.1. Easy Data Augmentation

In all the EDA techniques, we have generated 3 new data samples for every data point in the SICK train set. Say we have a data point from the train set - ($S_1$, $S_2$, $y$). To generate the first augmented data sample, we keep $S_2$ constant and modify $S_1$. To generate the second augmented data sample, we keep $S_1$ constant and modify $S_2$. Finally, to generate the third augmented data sample, we modify both $S_1$ and $S_2$.

Random Deletion - Randomly deleting words from sentences might have a detrimental effect as there is no reliable way to compute the new similarity scores between modified sentences. Thus, we limit ourselves to only deleting random stop words. Stop words are a list of low information words and add little context to the sentence. Thus, deleting one of these stop words will not change the meaning of the sentence, and consequently will not change the similarity score between two sentences. We use the standard list of stop words as got from the Natural Language Toolkit (NLTK). *Figure 6* shows an example of how random deletion is used to augment data.

| | Original Data | Augmented Data 1 | Augmented Data 2 | Augmented Data 3 |
|---|---|---|---|---|
| Sentence 1 | a dog is running towards a ball. | dog is running towards a ball. | a dog is running towards a ball. | dog is running towards a ball. |
| Sentence 2 | a dog is running towards a toy. | a dog is running towards a toy. | a dog running towards a toy. | a dog running towards a toy. |
| Similarity Score | 4.3 | 4.3 | 4.3 | 4.3 |

*Figure 6.* Example of Random Deletion

Random Insertion - Similar to Random Deletion, randomly inserting words into sentences might not have the desired effect as there will be no reliable way to compute the new similarity scores between the modified sentences. Thus we limit ourselves to only inserting random stop words. Since stop words don't add too much additional information to sentences, the similarity scores will remain the same. *Figure 7* shows an example of how random insertion is used to augment data.

| | Original Data | Augmented Data 1 | Augmented Data 2 | Augmented Data 3 |
|---|---|---|---|---|
| Sentence 1 | a man is sitting in a field. | a man is sitting in a our field. | a man is sitting in a field. | a man is the sitting in a field. |
| Sentence 2 | a man is running in a field. | a man is running in a field. | a the man is running in a field. | a man he is running in a field. |
| Similarity Score | 2.6 | 2.6 | 2.6 | 2.6 |

*Figure 7.* Example of Random Insertion

Synonym Replacement - We use the WordNet (5) corpus imported from NLTK to generate synonyms of words. While randomly choosing words, we avoid stop words as replacing them provides no additional information and hence may not produce good quality augmentations. *Figure 8* shows an example of how synonym replacement is used to augment data.

| | Original Data | Augmented Data 1 | Augmented Data 2 | Augmented Data 3 |
|---|---|---|---|---|
| Sentence 1 | a dog with a ball is being chased in the grass. | a dog with a ball is being pursued in the grass. | a dog with a ball is being chased in the grass. | a dog with a ball is being pursued in the grass. |
| Sentence 2 | a dog is chasing a ball on the grass. | a dog is chasing a ball on the grass. | a dog is chase a ball on the grass. | a dog is chase a ball on the grass. |
| Similarity Score | 3.6 | 3.6 | 3.6 | 3.6 |

*Figure 8.* Example of Synonym Replacement

#### 3.2.2. BACK-TRANSLATION

We have used the Marian Machine Translation model from the HuggingFace library to first translate a sentence into French and then translate it back into English. *Figure 9* shows an example of how Back-Translation is used to augment data.

| | Original Sentence | Translated Sentence (French) | Back - Translated Sentence (English) |
|---|---|---|---|
| Sentence 1 | A group of kids is playing in a yard and an old man is standing in the background. | Un groupe d'enfants joue dans une cour et un vieil homme se tient en arrière-plan. | A group of children play in a yard and an old man stands in the background. |
| Sentence 2 | A group of boys in a yard is playing and a man is standing in the background. | Un groupe de garçons dans une cour joue et un homme se tient en arrière-plan. | A group of boys in a playground plays and a man stands in the background. |

*Figure 9.* Example of Back-Translation

However, generating this one instance took us 30 seconds. Thus generating a large quantity of data to see a visible difference was prohibitive. For this reason, we do not further explore how back-translation affects semantic sentence similarity.

#### 3.2.3. MIXUP

Algorithm 1 describes how mixup is performed given two sentences. Algorithm 2 details how mixup data augmentation is applied to an instance of the training set.

**Algorithm 1** Generating Mixedup Sentences

**Input:** sentence $S_1$, sentence $S_2$
$\alpha \leftarrow 0.4$
$\lambda \leftarrow Beta(\alpha, \alpha)$
**for** i = 0 **to** $max\_len$ **do**
  $w_1^i \leftarrow$ wordEmbedding($S_1[i]$)
  $w_2^i \leftarrow$ wordEmbedding($S_2[i]$)
  s.append($\lambda w_1^i + (1 - \lambda) w_2^i$)
**end forreturn** s

**Algorithm 2** Mixup

**Input:** sentence $S_1^i$, sentence $S_2^i$, label $y_i$, train_set_size
**for** $i = 1$ **to** $4$ **do**
  j ←randomInt(0,train_set_size)
  data.append(generateMixedupSentences($S_1^i$, $S_1^j$))
  data.append(generateMixedupSentences($S_2^i$, $S_2^j$))
  data.append($\lambda y_i + (1 - \lambda) y_j$)
**end forreturn** data

#### 3.2.4. MIXING DATA AUGMENTATIONS

In this work we have tested the following combinations -

1. Mixup + Random Deletion (MX + RD)
2. Mixup + Random Insertion (MX + RI)
3. Mixup + Synonym Replacement (MX + SR)
4. Synonym Replacement + Random Deletion (SR + RD)
5. Synonym Replacement + Random Insertion (SR + RI)
6. Random Deletion + Random Insertion (RD + RI)

### 3.3. Siamese Training

After using the above augmentation techniques detailed above on the SICK train set, we end up with exactly $31,488$ sentence pairs in our training set as opposed to the original $7,872$. We still have our original test set of $1,968$ sentence pairs intact with us. Since sentences may differ in length, we compute the length of the largest sequence in the dataset and pad other sentences with 300 - dimensional zero tensors.

Since the similarity function described above in *Equation 1* always outputs a score which lies between 0 & 1, the labels in the SICK training set are also normalized to lie between

0 & 1 by simply dividing by 5. The output is then scaled back to lie between 0 & 5 by multiplying by 5 again.

The Siamese Network is trained using Backpropagation Through Time (BPTT) under the Mean Squared Error (MSE) Loss function. The authors in (10) used Adadelta over Adam, but we found Adam optimization to actually work better for our use case. We used a batch size of 64 and trained the Siamese Network for 50 epochs, because we found the loss plateauing thereafter and only marginal improvements in the loss. We also didn't train further to avoid overfitting the model on the training set. All model variants were trained on Google Colab, with the unaugmented variant taking around 6 minutes to train, while the augmented variants each took around 25 minutes to train.

## 4. Results

### 4.1. word2vec

*Table 1* shows the cosine similarities for various word pairs for three different version of word2vec we have used - our own implementation (OE), pre-trained vectors from google-news dataset (GN), and gensim's skip-gram implementation (GS).

*Table 1.* word2vec cosine similarities

| WORD 1 | WORD 2 | OE | GN | GS |
|---|---|---|---|---|
| BOY | CHILD | 0.80 | 0.59 | 0.62 |
| BOY | MAN | 0.79 | 0.68 | 0.11 |
| BOY | GIRL | 0.80 | 0.85 | 0.35 |
| GIRL | WOMAN | 0.79 | 0.74 | 0.20 |
| WOMAN | MUSHROOM | 0.49 | 0.08 | 0.19 |
| BASKETBALL | GAME | 0.37 | 0.39 | 0.51 |
| COWBOY | HORSE | 0.44 | 0.40 | 0.43 |
| SEA | BEACH | 0.36 | 0.40 | 0.21 |
| TREE | BRANCH | 0.32 | 0.33 | 0.25 |
| EYES | SUNGLASSES | 0.41 | 0.36 | 0.15 |
| SEA | FISH | 0.33 | 0.32 | 0.43 |
| VEGETABLE | BROCCOLI | 0.12 | 0.61 | 0.58 |

In *Table 1*, we see that OE produces very similar cosine similarities as GN does. In some cases, where the words in the pair are highly similar, the OE model outputs even higher cosine similarities as compared to GN. This is seen in the word pair - (boy, child). However, there are other cases like (woman, mushroom) and (vegetable, broccoli) where we would expect low and high similarity scores respectively. However, the converse is observed when using OE, that is, the model trained from scratch outputs a higher similarity for (woman, mushroom) as compared to GN, while outputting a lower similarity for (vegetable, broccoli) than GN.

We believe this discrepancy arises from the number of times the word is encountered in the dataset. In general, the more times the skip-gram model encounters a particular word, the better the embedding generated tends to be. Thus, if certain words are only encountered only a handful of times, we will not be able to produce good word embeddings. This is clearly seen in the table, as words like - boy, child and man - are encountered multiple times in the dataset, and hence have good word vectors, however the word broccoli only appears a few times in the dataset, thus the word vector generated is not truly representative of the word's context.

### 4.2. Manhattan LSTM

*Figure 10* shows the learning curve of the Manhattan LSTM on the base SICK dataset.

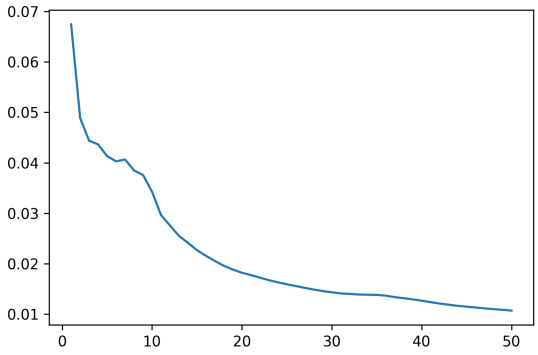


*Figure 10.* Learning Curve for MaLSTM Model

In the first sentence pair in *Figure 11*, we see that the word "little" is replaced by "young" and the MaLSTM Model predicts a high sentence similarity score nevertheless, which is indicative of its robustness to synonym replacements. Again in the second sentence pair, we see that the sentence construction has slightly been altered, but the MaLSTM Model recognizes this and yet again predicts a high similarity score.

| Sentence 1 | Sentence 2 | Similarity |
|---|---|---|
| A little girl is looking at a woman in costume. | A young girl is looking at a woman in costume. | 4.7152 |
| A married couple is walking down the aisle | A couple who have just got married are walking down the aisle | 4.8881 |
| A classroom is full of students | A classroom is empty | 1.9926 |
| A man is playing a guitar | A guitar is being played by a man | 4.6162 |
| A girl is playing the piano | The piano is being played by the girl | 4.7027 |
| A girl is playing the piano | There is no girl playing the piano | 3.5951 |

*Figure 11.* Good Predictions made by the MaLSTM Model

The third sentence is an example of a contradiction. Even though these two sentences share a couple of words, our model is able to distinguish these two in meaning by predicting a much lower score. The fourth and fifth examples delineate our model's ability to map active and passive voices to a similar hidden representation, thus, resulting in a higher similarity score.

For the sixth sentence pair, we replace the second sentence in the fifth example with its negative and although these two sentences will now share a lot of common words, we see that the similarity score has now plummeted significantly. This example along with the third example are indicative of our model's neat ability to pick out contradictions as well.

| Sentence 1 | Sentence 2 | Similarity |
|---|---|---|
| someone is boiling okra in a pot | the man is not playing the drums | 2.684 |
| a brown and white coat is floating across shallow water near a trotting dog. | a woman is brushing her feet. | 2.741 |
| a cluster of four brown dogs are playing in a field of brown grass | four dogs are playing in a area covered by grass. | 3.6095 |

*Figure 12.* Bad Predictions made by the MaLSTM Model

In the first two sentence pairs in *Figure 12*, we see that the sentences have absolutely no relation, yet our model gives them "medium" scores. We would have expected scores below 1 for these. We suspect this is the case because of an imbalanced training set. Most of the training instances in the SICK dataset have a similarity score of atleast 2.5 and there are only a handful of occassions when this score dropped below this threshold.

In the third sentence pair, we see that the Sentence 1 has a bunch of additional words/phrases like "a cluster of" & "brown" which don't really alter the meaning of the sentence all that much, but our model isn't able to recognize this and penalizes this sentence pair with a lower score.

### 4.3. Data Augmentation

Since random insertion and synonym replacement introduce new words into our dataset, we could neither use our own embeddings nor pre-trained vectors from google-news dataset. Thus, when we use either our own embeddings or pre-trained vectors from google-news dataset, we stick to only experimenting on the 4 datasets as shown in *Table 2* and *Table 3*.

*Table 2.* Siamese Training - Own Embeddings

| DATASET | TRAIN ERROR | TEST ERROR |
|---|---|---|
| BASE | 0.0107 | 1.156 |
| RD | 0.004 | 1.066 |
| MX | 0.003 | 0.917 |
| MX + RD | 0.004 | 1.009 |

*Table 2* gives the train and test errors when we used embeddings generated by our own model written from scratch. Here we see that all the data augmented datasets produce lower test errors as compared to the base dataset. Amongst the data augmentation techniques, we see that mixup (MX) performs the best as compared to random deletion (RD).

*Table 3.* Siamese Training - Google News

| DATASET | TRAIN ERROR | TEST ERROR |
|---|---|---|
| BASE | 0.004 | 0.86 |
| RD | 0.002 | 0.86 |
| MX | 0.001 | 0.96 |
| MX + RD | 0.001 | 0.83 |

*Table 3* gives the train and test errors when we used pre-trained embeddings generated from google-news dataset. Here we see that the mixture of data augmentations on the dataset produces lower test error as compared to the base dataset and standalone data augmentations. We also see that mixup has a higher test error as compared to other datasets.

*Table 4.* Siamese Training - Gensim Skip Gram Implementation

| DATASET | TRAIN ERROR | TEST ERROR |
|---|---|---|
| BASE | 0.004 | 1.89 |
| RD | 0.002 | 1.71 |
| RI | 0.001 | 1.87 |
| SR | 0.001 | 2.03 |
| MX | 0.001 | 2.02 |
| MX + RD | 0.002 | 1.87 |
| MX + SR | 0.002 | 2.10 |
| MX + RI | 0.001 | 2.25 |
| RD + RI | 0.002 | 0.96 |
| RD + SR | 0.002 | 0.97 |
| SR + RI | 0.001 | 0.98 |

*Table 4* gives the train and test errors when we used embeddings generated by gensim's skip-gram model. Here we see that a combination of the simple data augmentation techniques significantly outperforms more involved techniques like mixup.

## 5. Conclusion

We see that augmenting data for Semantic Sentence Similarity leads to an improved system, that is, test errors produced are either better than, or in the worst case, as good as the errors obtained from training on the non-augmented version of the dataset. However, comparing the efficacy of mixing multiple data augmentations is non-trivial and needs to explored further.

## References


[1] Word2vec tutorial - the skip-gram model, 2016.

[2] A beginner's guide to word2vec and neural word embeddings, 2021.

[3] Amit Chaudhary. Text data augmentation with marianmt, 2021.

[4] Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 539–546. IEEE, 2005.

[5] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Cambridge, MA: MIT Press., 1998.

[6] Hongyu Guo, Yongyi Mao, and Richong Zhang. Augmenting data with mixup for sentence classification: An empirical study. *CoRR*, abs/1905.08941, 2019.

[7] Marco Marelli, Stefano Menini, Marco Baroni, Luisa Bentivogli, Raffaella Bernardi, Roberto Zamparelli, et al. A sick cure for the evaluation of compositional distributional semantic models. In *Lrec*, pages 216–223. Reykjavik, 2014.

[8] Chris McCormick. Word2vec tutorial part 2 - negative sampling, 2017.

[9] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[10] Jonas Mueller and Aditya Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[11] Xin Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.

[12] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *CoRR*, abs/1710.09412, 2017.