Assignment 1: Word Embeddings

Alan Sheu (E1436561)

November 8, 2024

The code can be found at https://github.com/alansheu2004/Word-Embeddings.

1 Introduction

The way that computers traditionally represent language data is through strings of characters, which are effective in conveying spelling and orthography information but don't shed any light on its meaning. In natural language processing, the goal of a word embedding is to capture the semantics of any given word by representing it as a vector.

The bases of this vector space should encode some meaningful aspect of the word, allowing us to do the usual vector operations on them. If we let e_x represent the embedding of word x, then we can measure the similarity of words by their dot product as

$$s(e_1, e_2) = \frac{e_1 \cdot e_2}{\|e_1\|\|e_2\|} = \cos\theta$$

.

This is called cosine similarity since it represents the cosine of the angle between the embeddings. Synonyms should have similarity closer to 1 but closer to -1 for antonyms.

Vectors can also work by analogy through addition. A classic example is

$$e_{\text{king}} - e_{\text{man}} + e_{\text{woman}} \approx e_{\text{queen}}$$

2 How Embeddings Work

2.1 One-Hot Encoding

The most naive approach would be to have each dimension of the vector represent the word itself. In a vocabulary of v words, the ith word would be represented as

$$o_i = \begin{bmatrix} \underset{1}{0} & \cdots & \underset{i-1}{0} & \underset{i}{1} & \underset{i+1}{0} & \cdots & \underset{v}{0} \end{bmatrix}.$$

While this is fast to compute, it is inefficient in space, needing a v-dimensional vector to make a very sparse encoding. It is also not a true embedding as it does not represent the semantics in any way. Still it is a useful representation for identifying words.

2.2 TF-IDF

To compare against an example that does not use machine learning, we turn to Term Frequency Inverse Document Frequency (TF-IDF). Suppose our training corpus contains D documents. Then the term frequency (TF) of word i can be given by a D-dimensional vector where

$$(TF_i)_d = \frac{\#\text{ of word } i \text{ in document } d}{\#\text{ of words in document } d}.$$

Words that co-occur will tend to have similar values, but since the distribution of words does not matter, TF is akin to a bag of words. To balance out varying word frequencies, the scalar inverse document frequency is given as

$$IDF_i = \log\left(\frac{D}{\#\text{ of documents containing word } i}\right).$$

Combining this yields the embedding

$$e_i = TF_i \times IDF_i \in \mathbb{R}^D$$

Unfortunately, the size of the vector is tied to the number of documents, and the embedding is sensitive to document length and relatedness. As such, TF-IDF is most used for broad genre classification rather than encoding specific words.

2.3 Word2Vec

Word2Vec is a neural network based algorithm to generate these word embeddings. The main idea of Word2Vec is that given a good embedding of a word, you should be able to derive how related other words are.

$$\underset{1\times V}{o_a} \xrightarrow[V\times n]{E} \underset{1\times n}{e_a} \xrightarrow[n\times V]{\Theta} \underset{1\times V}{p^*} \xrightarrow{\text{softmax}} \underset{1\times V}{p}$$

Beginning with a one-hot representation of input word a, multiplying by some matrix E will yield its embedding, reducing it to a smaller dense representation. Each row of E is effectively the embedding of each word in the vocabulary, so we can just look up the row in E in practice rather than multiplying. The matrix Θ encodes how likely an embedding is related each word in the vocab as represented by each column.

$$E = \begin{bmatrix} — & e_1 & — \\ — & e_2 & — \\ & \vdots & \\ — & e_V & — \end{bmatrix}, \quad \Theta = \begin{bmatrix} | & | & & | \\ \theta_1 & \theta_2 & \cdots & \theta_V \\ | & | & & | \end{bmatrix}$$

This yields $p^* \in \mathbb{R}^{1\times V}$ with a scalar for each word in the vocabulary. We take the softmax to turn these values into probabilities that add to 1:

$$\text{softmax}(p^*) = \frac{\exp(p^*)}{\sum_{i=1}^{V} \exp(p_i^*)}$$

where $\exp(p^*)$ is element-wise. Ideally, the resulting p should be similar to the one-hot vector of our related output word b. The cross-entropy loss we want to minimize then becomes

$$L = o_b \cdot \log(p)$$

where o_j is the one-hot representation we want to get. This is our objective function for gradient descent.

2.4 CBOW

The Word2Vec paper lays out two methods for getting training data for the model. The first is continuous bag of words (CBOW), which takes the surrounding context of a word in the document like BOW, but only on a limited sliding window around it. The model tries to take the surrounding context to predict the target word in the middle.

For a window that reaches to the left and right by l from in the documents, CBOW takes the mean of these context words as the input to the model and tries to minimize the loss with the target word d_l.

$$o_a = \frac{1}{2m}\left(\sum_{i=l-m}^{l-1} o_{d_i} + \sum_{j=l+1}^{l+m} o_{d_j}\right), \quad o_b = o_{d_l}$$

This means that we have the size of our training data is the size of our text corpus.

2.5 Skipgram

An alternative formation is a skipgram, where given the target word in the middle, the model predicts what the surrounding context words are. In this case,

$$o_a = o_{d_l}, \quad o_b = o_{d_{l\pm j}}$$

where $j = 1, 2, \ldots, m$. This means that the size of our training data is the size of the corpus multiplied by $2m$ for each context word. This means that skipgram takes more time to train than CBOW.

2.6 Negative Sampling

One issue with this formulation of Word2Vec is the computational cost of softmax, where we must sum across the entire vocabulary in the demoninator each iteration. We can just randomly choose k negative samples to train against instead, labeling the positive samples with $q = 1$ and negative examples with $q = -1$. Now instead of getting probabilities across the entire vocabulary, we have a logistic sigmoid function for each sample

$$p = \sigma(e_a \cdot \theta_b) = \frac{1}{1 + \exp(-e_a \cdot \theta_b)}.$$

The final logistic loss is then

$$L = -\sigma((e_a \cdot \theta_b)q)$$

3 Implementation

A total of three model architectures were tested in Google CoLab: a non-neural TF-IDF model, two CBOW models with window size 2 and 5 each, and two skipgram models with window size 2 and 5 each. Both neural models used negative sampling, which was found to be faster than softmax despite the efficiency of GPUs.

3.1 Data Collection

Corpus data was gathered from the *Super Mario Wiki* to test on domain-specific terms for this project. Using the *Media Wiki* API, articles were traversed through random hyperlinks and each article's plain text was collected. Template articles that did not meet a threshold of 100 words were discarded.

Each article was then split into sentences and then tokenized by words to be tabulated for the vocabulary. To reduce the amount of noisy data on extremely rare words, any words that appeared 3 or fewer times were dropped from the documents and vocabulary for subsampling.

In total, there were 1,173 documents and 56,865 sentences for processing. The total size of the corpus was 1,149,615 words with $V = 10,761$ valid vocab words.

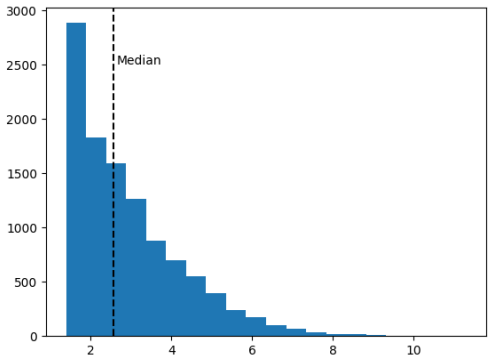

Fig: Distribution of Log Frequency in Vocabulary

Overall, this is a relatively small corpus, but hopefully some meaningful embeddings relationships can still be gathered.

3.2 Subsampling

In the Word2Vec models, before processing any data, certain words were dropped out of the data for subsampling with probability given by

$$P(\text{drop word } i) = \left(1 + \sqrt{\frac{\text{freq. of word } i}{s}}\right) \frac{s}{\text{freq. of word } i}$$

Where $s = 0.0001$ was a subsampling rate. Then when selecting negative samples, rather than sampling from the vocabulary uniformly, they were selected with

$$P(\text{word } i \text{ selected}) = \frac{(\text{freq. of word } i)^{3/4}}{\sum_{j=1}^{V}(\text{freq. of word } j)^{3/4}}$$

It was found in the literature empirically that subsampling could focus on the training of rarer words while suppressing more common words like "the" or "and" that aren't helpful in training other words while having plenty of representation in the data to be trained.

3.3 Descent Algorithm

An embedding size of $n = 200$ was chosen with Lecun normal weight initialization given by $\mathcal{N}\left(0, \sqrt{\frac{1}{V}}\right)$ for both $E, \Theta \in \mathbb{R}^{V,n}$. A negative sample size was chosen as $k = 7$ since the corpus was relatively small.

The algorithm then followed with stochastic gradient descent with larger batches of size 512 to make use of the GPU's RAM. In addition, an ADAM optimizer was used with exponential rates of 0.9 for the gradient and 0.999 to handle non-convexity.

Several learning rates were tried on small subsets of the data, and stopping after a certain number of epochs:

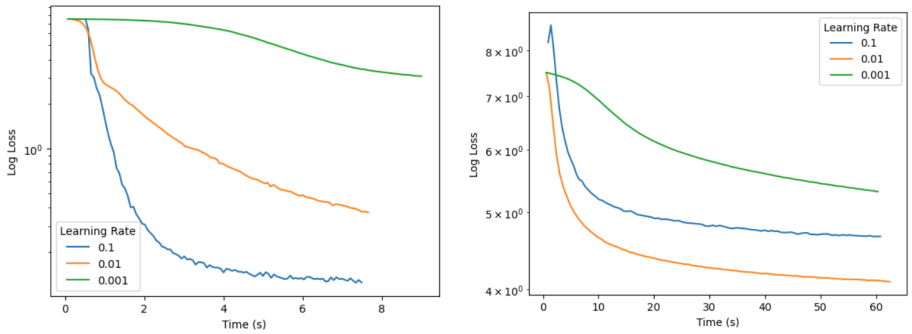

Fig: Log-Loss over time for CBOW (left) and skipgram (right) on toy datasets

Note the difference in timing, where skipgram needs to train for longer with its larger training set. Interestingly, the optimal learning rate for CBOW at 0.1 seems to be higher than skipgram's 0.01 despite the identical model architecture. This makes sense as CBOW takes the average of several context words and likely "dilutes" the gradient closer to 0 whereas the skipgrams directly train on each pair of words which can be more volatile with SGD.

3.4 Numerical Stability

In testing the model, there were often issues with numerical stability, where loss values and gradients could diverge.

One issue was with loss functions, where logarithms needed a small ϵ term added to prevent instability. In addition, reformulations of

$$\text{softmax}(p^*) = \frac{\exp(p^* - \max(p^*))}{\sum_{i=1}^{V} \exp(p_i^* - \max(p^*))}$$

and

$$\sigma(p^*) = \begin{cases} \frac{1}{1+\exp(-p^*)} & p^* \geq 0 \\ \frac{\exp(p^*)}{1+\exp(p^*)} & p^* < 0 \end{cases}$$

were needed to prevent an overflow in the exponential term.

Other techniques were needed to help with keeping the gradient in control. Gradient clipping was used to cap the the norm of a gradient from becoming too large. It was also necessary to reshuffle the training data for SGD if any mini-batches contained only negative samples as this could cause some volatility.

4 Results

In the final pass, the data was trained until it reached passed through 15 epochs, or ran out of computing time.

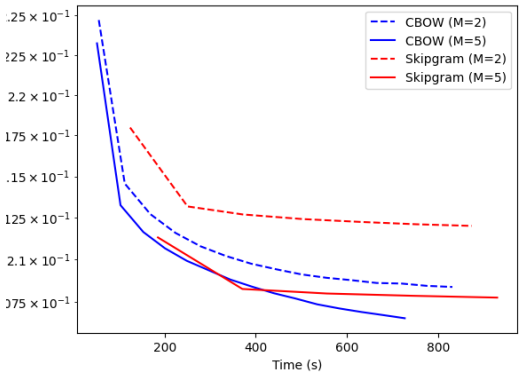

Fig: Log-loss over time for final pass

As expected, larger windows led to a smaller loss as there was more data to train off of with skipgrams and a "wider net" with CBOW, but there is likely a limit to this. Note that CBOW went through several more epochs with smaller training time.

4.1 Validation

We can confirm that our model worked as intended by running example targets and contexts through the neural network and check that the results are coherent. Inputting "mario" (the titular main character) into the network should return the highest probability for words that are found clustered near "mario":

Table: Top 5 predicted words given "mario" as input by probability

CBOW				Skipgram			
$M = 2$		$M = 5$		$M = 2$		$M = 5$	
kart	0.9993	kart	0.9999	super	0.6865	in	0.5994
super	0.9978	paper	0.9999	in	0.6325	super	0.5957
luigi	0.9968	super	0.9999	kart	0.6232	and	0.5553
paper	0.9934	enemy	0.9999	and	0.5819	the	0.5390
party	0.9922	party	0.9998	bros	0.5591	mario	0.4905

As expected, common predictions are recurrent names of mario games such as "Mario Kart" or "Super Mario". A larger window in CBOW means higher probabilities as more words are included in the context while skipgrams have lower probabilities because each context-target pair is handled separately and thus occurs less. It also seems that frequent words like "in" or "and" are more frequently predicted with skipgrams where their influence would be diluted with CBOW.

4.2 Evaluation

To actually evaluate our embeddings, we can look at the cosine similarity between words. Here, we list the most similar words to "mushroom" (a power-up in the games) aside from "mushroom" itself:

Table: Top 5 similar words to "mushroom" by cosine similarity

TF-IDF		CBOW				Skipgram			
		$M = 2$		$M = 5$		$M = 2$		$M = 5$	
mushrooms	0.534	star	0.304	between	0.289	features	0.413	nintendo	0.429
capped	0.397	planet	0.290	flower	0.279	certain	0.406	fire	0.415
specialty	0.353	toadette	0.280	spinys	0.269	now	0.386	koopa	0.394
occupation	0.351	goomba	0.268	small	0.269	lakitu	0.375	a	0.382
refined	0.351	chance	0.263	star	0.265	of	0.364	pipes	0.366

There are a couple semantically related terms such as "capped" or "goomba" (mushroom-shaped enemy) as well as a couple syntactical related words with other power-ups such as "[fire] flower" or "[super] star". However it is clear from the top words that the embeddings are not very robust.

We can instead look at how each model measures the similarity between prototypical pairs of words:

Table: Cosine similarity between different word pairs

Word Pairs		TF-IDF	CBOW		Skipgram	
			$M = 2$	$M = 5$	$M = 2$	$M = 5$
"mario"	"luigi"	0.464	-0.006	-0.065	0.31	0.419
"peach"	"daisy"	0.309	0.144	0.205	0.369	-0.040
"koopa"	"goomba"	0.084	0.065	0.041	-0.085	0.296
"hat"	"cap"	0.079	0.011	0.011	0.063	0.063
"big"	"mega"	0.127	-0.027	0.003	-0.145	-0.019
"man"	"woman"	0.024	-0.067	-0.067	-0.081	-0.081
"mario"	"bowser"	0.309	0.124	0.112	0.236	0.292
"good"	"bad"	0.081	0.076	0.076	-0.064	-0.064
"mega"	"mini"	0.027	0.036	-0.01	-0.059	-0.139
"live"	"die"	0.053	-0.092	-0.092	-0.090	-0.090

For the top section of the table, we would expect to have mostly positive results and the bottom mostly negative as prototypical examples. Instead, characters like "mario", "luigi", and "bowser" have relatively significant deviations from zero while the rest, even complete synonyms and antonyms, are close to zero and are inconsistent in sign across models.

What is also interesting is that skipgrams seem to deal a fair bit better with distinguishing between these pairings. This makes sense as it trains on the actual word pairs rather than an averaged context, but takes longer to train. Window size seems to have little observable

4.3 Conclusion

It seems that instead of encoding semantic information per se, the generated embeddings for the *Mario Wiki* instead capture more of the co-occurence and frequency of terms regardless of how semantically related they are.

There are several potential reasons for this discrepancy. One is almost surely the size and origin of the text corpus. A million words is not very much to be able to contain enough information even for just a vocabulary of size 10,000. In addition, a wiki might not be very representative as it is inherently templatic in nature and lacks the context variation to fully describe the semantics. Extensions of this project may want to use more pages for data and vary the sourcing.

Another reason is the linguistics. Many words are ambiguous in their word meaning, and the model makes no distinction. "Mario" is the main character of the games, but is also just a general label in the franchise. Word2Vec tends to just average these meanings, but a more sophisticated model could try to distinguish them. Otherwise, they may need to be tagged manually. The algorithm also makes no inference on the morphology, so "mushroom" and "mushrooms" are considered different words. Linking them together could not only increase the amount of data used to train each word stem but also provide syntactic information.

Overall, this project serves as a good proof of concept, but more data and more training would be required to get higher quality embeddings.

5 References

Below are a list of references for both information and code:

https://jaketae.github.io/study/word2vec/
https://www.geeksforgeeks.org/word-embeddings-in-nlp/
https://www.youtube.com/playlist?list=PLhWB2ZsrULv-wEM8JDKA1zk8_2Lc88I-s
https://towardsdatascience.com/skip-gram-neural-network-from-scratch-485f2e688238
https://stackoverflow.com/questions/4576077/how-can-i-split-a-text-into-sentences
https://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/
https://stackoverflow.com/questions/4452102/how-to-get-plain-text-out-of-wikipedia