CSE 256 Fall 2024, UCSD: PA1

Shang-Chun Tai
A69032339
shtai@ucsd.edu

Abstract

In this assignment, we explored three natural language processing techniques: Deep Average Network (DAN), Byte Pair Encoding (BPE), and Skip-Gram. The following discussion focuses on comparing the performance of DAN and BPE in sentiment classification tasks, while also providing insights into how the Skip-Gram model handles word embeddings.

1 Deep Averaging Network (DAN)

Deep Averaging Network (DAN) works by averaging the word embeddings of all words in a given sentence or document, creating a fixed-size vector representation, and then passing this averaged vector through a feedforward neural network for classification. Compared to traditional Bag-of-Words (BOW) models, DAN captures semantic meaning better due to the use of word embeddings. Moreover, DAN handles varying sentence lengths more naturally and can generalize well even with fewer parameters, offering better performance on longer sentences than BOW.

1.1 Implementations

First, assign labels to each word in the two pre-trained datasets `"glove.6B.50d-relativized.txt"` and `"glove.6B.300d-relativized.txt"` sequentially starting from index 2, with the aim of converting each word in the sentences to its corresponding index later. Starting the index at 2 serves two main purposes. First, since the number of words in each sentence varies, padding (denoted as PAD) is used to standardize the embedding dimensions. Second, some words may not be present in the pre-trained dataset, so these words are marked as unknown (denoted as UNK).

The second step involves extracting the training and testing datasets and sequentially retrieving each sentence. For every sentence, we convert each word into its corresponding index, which was mapped earlier. After converting the words in each sentence to their corresponding indices, we apply padding to ensure that each sentence has a uniform dimension.

Finally, we input the data into the Deep Averaging Network (DAN) model. First, we use `get initialized embedding layer` to obtain the torch.nn.Embedding layer, which initializes the embedding. Next, we calculate the average of all the word vectors [1] in each sentence, as demonstrated by the formula provided. This averaged vector is then passed through a two-layer fully connected network for further processing to produce the final output.

$$\frac{1}{n}\sum_{i=1}^{n} e(w_i)$$

Figure 1: Training Acc for Pretarined/Random

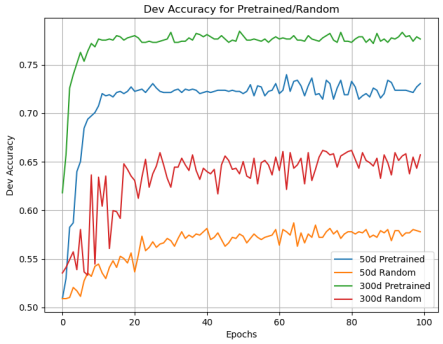

Figure 2: Dev Acc for Pretarined/Random

Table 1: Accuracy for Pretrained/Random

Embedding	Dimension	Dropout	Train Acc (%)	Dev Acc (%)
GloVe	50 / 300	0	74.0 / 82.8	72.9 / 77.8
GloVe	50 / 300	0.5	73.0 / 82.2	72.2 / 78.0
Random	50 / 300	0	60.1 / 68.3	60.1 / 68.0
Random	50 / 300	0.5	59.2 / 67.7	62.2 / 64.7

Table 2: Configuration

Layers	2
Batch size	16
Epoch	100
lr	0.0001
Opt	Adam
Loss	NLL
Act	Relu

1.2 Evaluation

Table 2 outlines the fixed parameter settings for the experiments. In the DAN model, using GloVe pretrained embeddings with a 300d word vector dimension resulted in the development set accuracy exceeding the assignment's baseline accuracy of 77%. Furthermore, I adjusted several hyperparameters to observe their effect on the model's performance.

Based on the results in Table 1, it is clear that hyperparameter variations significantly impact the predictive performance of the DAN model. For different embedding dimensions, 300d outperforms 50d, possibly because 300d can capture more complex relationships between words due to its higher capacity for representing detailed features. The use of pretrained embeddings greatly improves performance, as pretrained models often capture semantic and syntactic relationships in a language. Additionally, models with dropout outperform those without, likely because dropout helps prevent overfitting by regularizing the model's complexity. Furthermore, Table 3 indicates that increasing the hidden size improves model performance, likely due to the increase in parameters similar to the effects of higher-dimensional embeddings.

Table 3: Accuracy for different Hidden Size

Hidden Size	Train Acc (%)	Dev Acc (%)
16	80.3	78.7
64	82.3	77.2
256	85.2	78.4
1024	92.4	79.8

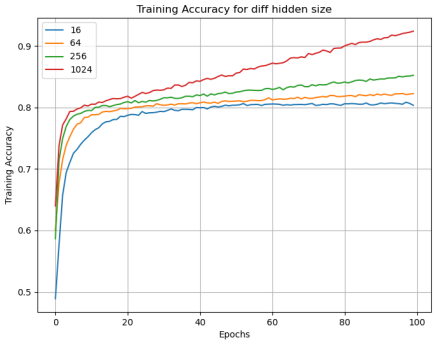

Figure 3: Training Acc for diff hidden size

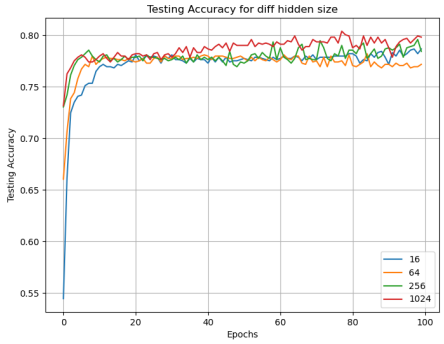

Figure 4: Dev Acc for diff hidden size

2 Byte Pair Encoding (BPE)

The tokens in a DAN model are usually words and punctuations separated by spaces, such as ["i", "went", "to", "new", "york", "last", "week", "."]. However, this has some drawbacks. For example, if the model learned the relationship between “old”, “older”, and “oldest”, it does not tell the model anything about the relationship between “smart”, “smarter”, and “smartest”. If we use some subtokens such as “er” and “est”, and the model learned the relationship between “old”, “older”, and “oldest”, it will tell the model some information about the relationship between “smart”, “smarter”, and “smartest”. Therefore, subtokens sound a better tokenization method, and Byte Pair Encoding (BPE) [2] is an easy implementation way.

2.1 Implementations

The first step is to append the end-of-word stop token </w> to every word in the `train.txt` file. Stop token is important. Without “</w>”, say if there is a token “st”, this token could be in the word “st ar”, or the word “wide st”, however, the meanings of them are quite different. With “</w>”, if there is a token “st</w>”, the model immediately knows that it is the token for the word “wide st</w>” but not “st ar</w>”.

Next, we insert a separator (space) between each character of the word to prepare for calculating the frequency of subword occurrences. We then pair adjacent characters in each word to determine the frequency of character pairs. The more frequently two characters appear next to each other, the more likely it is that this pair of characters represents a meaningful subword. By identifying high-frequency pairs, we can iteratively merge them into subword units, which helps efficiently compress or tokenize text for language models.

Finally, we map each subword to an index similar to the way it’s done in a DAN model. We then split each sentence in the file into subword indices. These indices are fed into the BPEModel using torch.nn.Embedding, where the subwords are converted into embeddings for further processing in the model.

2.2 Evaluation

Different merge times Subwords allow the model to capture the meanings of certain tokens, and the number of subword merges affects the model’s accuracy. In this analysis, we examine the impact of different merge counts on model accuracy. The results are shown in Table 1. In Table 4, we observed that the more merge steps performed, the better the model’s predictive performance (e.g., accuracy increased from 70.2% with 500 merges to 75.5% with 2000 merges). I believe this is because more characters are merged into meaningful subwords, providing the model with better features to learn from. However, as we increased the number of epochs, training accuracy improved, while development accuracy initially increased and then decreased. This suggests that the model was overfitting during later epochs.

Table 4: Accuracy for different merge times

Merge times	Epochs	Train Acc (%)	Dev Acc (%)
500	50 / 100	72.8 / 75.6	71.3 / 70.2
1000	50 / 100	77.8 / 81.2	74.2 / 74.1
2000	50 / 100	83.6 / 88.3	76.4 / 75.5

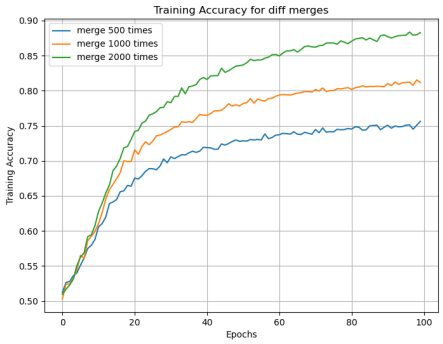

Figure 5: Training accuracy for different merges

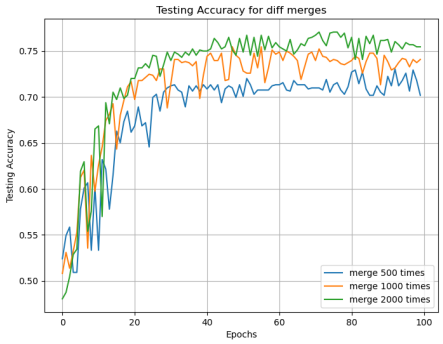

Figure 6: Dev accuracy for different merges

Different layers Table 6 shows that the increase in the number of layers in a model leads to a greater ability to learn complex features due to the additional parameters. Comparing the performance of 2-layer and 3-layer models, we observed that under 500 merges, the 3-layer model performed better (72.6%) than the 2-layer model(71.9%). However, with 2000 merges, the 3-layer model underperformed (74.5%) compared to the 2-layer model (76.0%). This further confirms that the 2-layer model at 2000 merges is already overfitting, and increasing layers exacerbates the overfitting issue.

DAN vs BPE Byte Pair Encoding (BPE) was primarily designed to address the limitations of models like DAN, and theoretically, it should offer better predictive performance. However, our experimental results showed that this was not the case. I believe this discrepancy stems from the effectiveness of the pretrained dataset used with DAN. When the pretrained embedding vectors were replaced with randomly initialized embeddings, DAN's performance dropped to around 66.6%, demonstrating that its success relies heavily on the pretrained data.

3 Understanding Skip-Gram

3.1 Q1 Answers

3a with a window size $k = 1$, the training examples are:

$$(x = \text{“the"}, y = \text{“dog"}), \quad (x = \text{“dog"}, y = \text{“the"})$$
$$(x = \text{"the"}, y = \text{"cat"}), \quad (x = \text{"cat"}, y = \text{"the"})$$
$$(x = \text{"a"}, y = \text{"dog"}), \quad (x = \text{"dog"}, y = \text{"a"})$$

Table 5: Accuracy for different merge times

Merge times	Dropout (Yes/No)	Train Acc (%)	Dev Acc (%)
500	0 / 0.3	75.6 / 75.2	70.2 / 72.5
1000	0 / 0.3	77.8 / 81.2	74.2 / 74.1
2000	0 / 0.3	88.3 / 88.2	75.5 / 75.0

Table 6: Accuracy for different merge times

Merge times	Layers	Train Acc (%)	Dev Acc (%)
500	2 / 3	75.8 / 76.6	71.9 / 72.6
2000	2 / 3	88.6 / 91.7	76.0 / 74.5

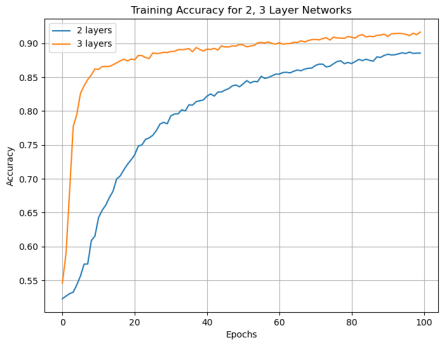

Figure 7: Training accuracy for different layers

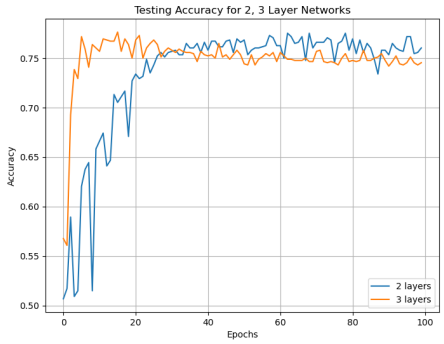

Figure 8: Dev accuracy for different layers

Suppose that the embedding dimension $d = 2$, and the context embeddings are as follows:

$$c_{\text{dog}} = (0, 1), \quad c_{\text{cat}} = (0, 1), \quad c_{\text{a}} = (1, 0), \quad c_{\text{the}} = (1, 0)$$

To compute the probabilities $P(y|\text{the})$ that maximize the data likelihood, we need to calculate the maximum likelihood estimate for $P(y|\text{the})$. The word "the" appears twice in the training examples. In these examples, the word "dog" follows "the" once, and the word "cat" also follows "the" once. Therefore, the maximum likelihood estimates for these probabilities are:

$$P(\text{dog}|\text{the}) = \frac{1}{2}, \quad P(\text{cat}|\text{the}) = \frac{1}{2}$$

This is because both "dog" and "cat" appear an equal number of times in the context of "the."

3b We use the Skip-Gram model's probability formula:

$$P(y|x) = \frac{\exp(v_x \cdot c_y)}{\sum_{y'} \exp(v_x \cdot c_{y'})},$$

where:

- v_x is the embedding vector for the center word x,
- c_y is the embedding vector for the context word y,

The probabilities $P(\text{dog}|\text{the})$ and $P(\text{cat}|\text{the})$ should both be 0.5, implying that:

$$v_{\text{the}} \cdot c_{\text{dog}} = v_{\text{the}} \cdot c_{\text{cat}}.$$

Since both $c_{\text{dog}} = (0, 1)$ and $c_{\text{cat}} = (0, 1)$, we need the second element of v_{the} to determine the dot product. To ensure equal dot products, we can set v_{the} as:

$$v_{\text{the}} = (0, v),$$

where v is a constant. This results in:

$$v_{\text{the}} \cdot c_{\text{dog}} = (0, v) \cdot (0, 1) = v,$$

$$v_{\text{the}} \cdot c_{\text{cat}} = (0, v) \cdot (0, 1) = v.$$

Thus, the dot products are equal, leading to equal probabilities for both "dog" and "cat."

To achieve probabilities $P(\text{dog}|\text{the}) = 0.5$ and $P(\text{cat}|\text{the}) = 0.5$, we need the dot products to be large enough so that the softmax function outputs probabilities close to 0.5. By setting $v_{\text{the}} = (0, v)$ with v being a large positive constant, we can make the softmax output approximately 0.5 for both context words.

Therefore, a near-optimal choice for v_{the} is:

$$v_{\text{the}} = (0, v),$$

where v is a large constant. This ensures that the probabilities for "dog" and "cat" are approximately 0.5, within 0.01 of the optimal value.

3.2 Q2 Answers

3c : For each sentence, with a window size $k = 1$, the training examples are as follows:

- For "the dog":

$$(x = \text{"the"}, y = \text{"dog"}), \quad (x = \text{"dog"}, y = \text{"the"})$$

- For "the cat":

$$(x = \text{"the"}, y = \text{"cat"}), \quad (x = \text{"cat"}, y = \text{"the"})$$

- For "a dog":

$$(x = \text{"a"}, y = \text{"dog"}), \quad (x = \text{"dog"}, y = \text{"a"})$$

- For "a cat":

$$(x = \text{"a"}, y = \text{"cat"}), \quad (x = \text{"cat"}, y = \text{"a"})$$

So the complete set of training examples is: (the, dog), (dog, the), (the, cat), (cat, the), (a, dog), (dog, a), (a, cat), (cat, a)

3d We want to set the word vectors v_x and context vectors c_y such that the dot products give probabilities close to the empirical distribution.

A possible set of vectors that achieves this is:

- **Word Vectors**:

$$v_{\text{the}} = (1, 0), \quad v_{\text{a}} = (1, 0), \quad v_{\text{dog}} = (0, 1), \quad v_{\text{cat}} = (0, 1)$$

- **Context Vectors**:

$$c_{\text{dog}} = (0, 1), \quad c_{\text{cat}} = (0, 1), \quad c_{\text{the}} = (1, 0), \quad c_{\text{a}} = (1, 0)$$

This setup ensures that the dot products between the word vectors and context vectors give probabilities close to 0.5 for the observed training examples, which matches the empirical distribution and satisfies the skip-gram objective.

References

[1] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. In Chengqing Zong and Michael Strube, editors, *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1681–1691, Beijing, China, July 2015. Association for Computational Linguistics.

[2] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics.