CSE 256 PA1 Report

1 Part 1: Deep Averaging Network (DAN)

1.1 DAN with GloVe Embeddings

For this part, I built a Deep Averaging Network following the instructions. The model architecture is pretty straightforward. First, it takes each word in a sentence and looks up its 300-dimensional GloVe vector. Then it averages all these vectors together to get a single representation of the whole sentence. I made sure to use the WordEmbeddings class that was provided and ignored padding tokens (index 0) when calculating the average so they wouldn't mess things up.

After averaging, the sentence vector goes through a feedforward network with two hidden layers. I used ReLU activation functions and added dropout layers to help prevent overfitting. The final output layer gives scores for the two classes (positive and negative sentiment).

For training, I used the Adam optimizer which worked really well. I set the learning rate to 0.005 and used a batch size of 16. The model trained for 30 epochs total. Training was surprisingly fast - the model learned pretty quickly right from the start.

Results: My best dev set accuracy was **80.3%** at epoch 1. This easily beats the required 77% threshold. The whole training process took about 160 seconds on my CPU. Compared to the Bag-of-Words baseline (which gets around low 70s), this is a big improvement. I think this shows that averaging pre-trained word vectors does a much better job at capturing semantic meaning than just counting how many times each word appears.

1.2 DAN with Random Embeddings

For this part, I kept the exact same network architecture but removed the pre-trained GloVe vectors. Instead, I initialized the embedding layer with random values and let the model learn them from scratch during training.

The results were noticeably different. The random embeddings did eventually work, but they started off much worse and took longer to reach good performance. My best dev accuracy was **78.6%** at epoch 4, which is about 2 points lower than what I got with GloVe (80.3%).

One thing I noticed was that the training accuracy climbed really high (close to 99%), but the dev accuracy was more unstable and jumped around between epochs. This suggests the model was starting to overfit the training data. When you start with random vectors, the model doesn't have any prior knowledge about word meanings, so it seems more prone to just memorizing patterns in the training set rather than learning generalizable features.

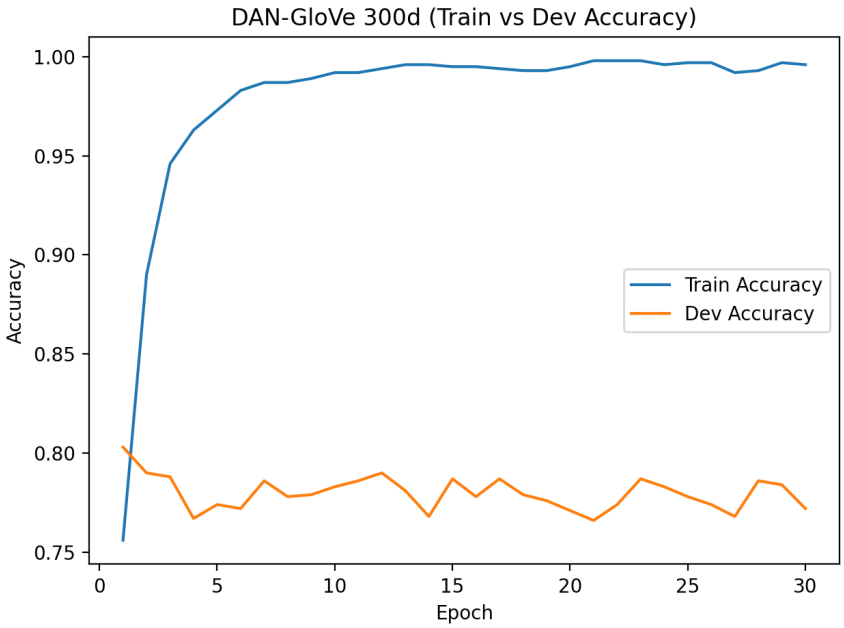

Figure 1: Training progress for DAN with GloVe embeddings

Analysis: Starting with pre-trained embeddings like GloVe gives the model a huge head start. The vectors already encode semantic relationships between words that were learned from massive amounts of text. When you use random initialization, the model has to learn everything from scratch using just the small training set we have, which is much harder.

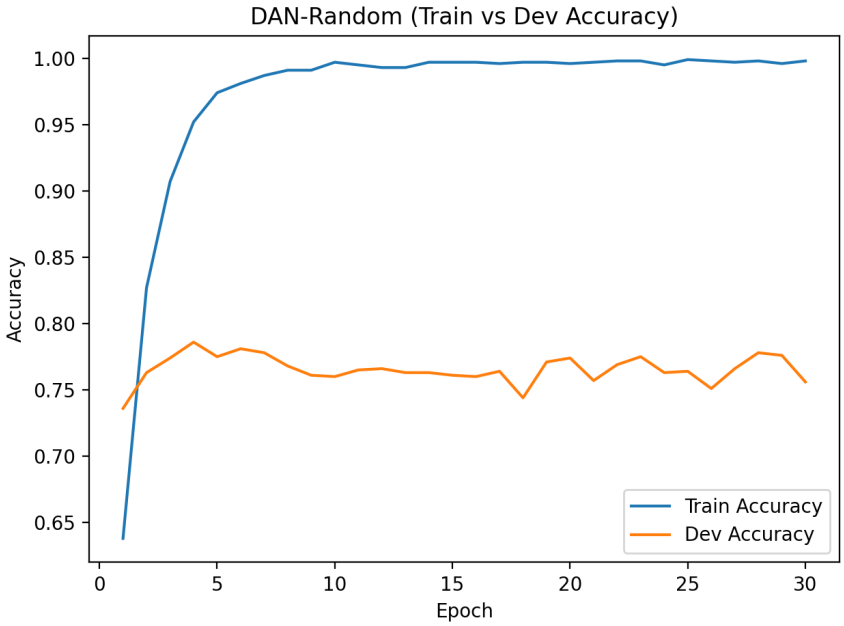

Figure 2: Comparison of training curves for GloVe vs random embeddings

2 Part 2: Byte Pair Encoding (BPE)

2.1 Subword Tokenization

For this section, I modified the DAN to use subword tokens instead of full words. I implemented the Byte Pair Encoding algorithm and trained it on the training data. Since BPE creates new subword units that don't exist in GloVe's vocabulary, I had to initialize the embeddings randomly and train them from scratch (similar to Part 1b).

I experimented with four different vocabulary sizes to see how it affects performance:

Vocabulary Size	Best Dev Accuracy
BPE 1000	72.6%
BPE 2000	75.1%
BPE 5000	75.2%
BPE 10000	75.5%

Table 1: Performance of BPE-based models with different vocabulary sizes

Analysis: There's a clear trend here - bigger vocabulary sizes lead to better accuracy. With only 1000 tokens, the model performed worst at 72.6%. I think this is because the vocabulary is so small that words get split into really tiny pieces that don't carry much meaning on their own. With 10,000 tokens (the largest size I tried), the model did best at 75.5% because it can keep more common words intact.

However, even my best subword model (75.5%) didn't beat the word-level model with random embeddings from Part 1b (78.6%). This suggests that for this particular sentiment classification task, using complete words works better than breaking them down into subwords. I also noticed that larger vocabularies made training slower - going from about 17 seconds with vocab size 1000 to about 37 seconds with vocab size 10,000. This makes sense because the embedding matrix gets bigger.

When BPE might help: Even though BPE didn't win here, I think it could be really useful for other scenarios. For example, if you have a dataset with lots of rare words or typos, BPE can handle them by breaking them into familiar subword pieces. It's also good for languages with complex morphology where words have lots of different forms.

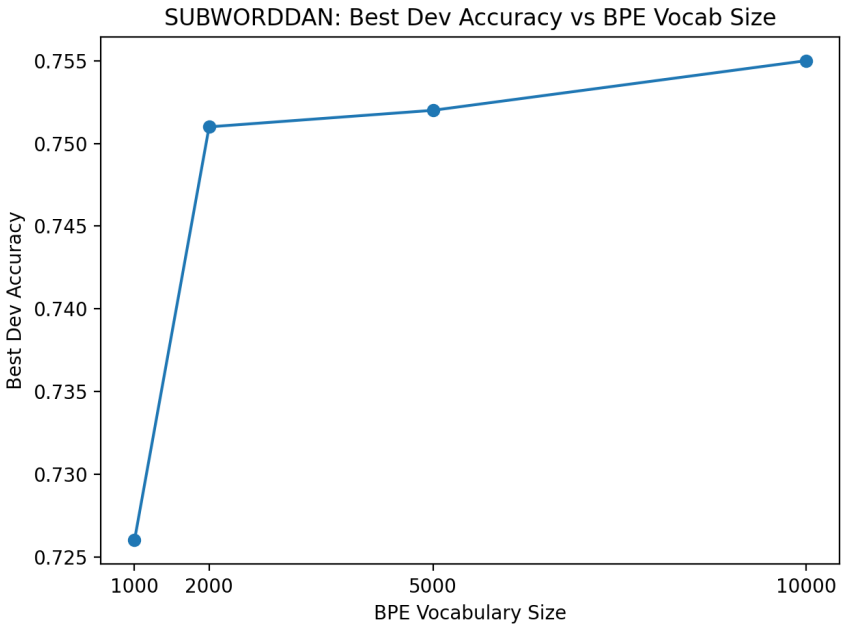

Figure 3: Effect of vocabulary size on BPE model performance

3 Part 3: Understanding Skip-Gram

3.1 Q1: Skip-Gram with Three Sentences

Given sentences:

- the dog
- the cat
- a dog

With window size $k = 1$, we look at words right next to each other.

3.1.1 3a) Maximum Likelihood Probabilities

First, I need to find all the training pairs where "the" is the center word.

From "the dog": the pair is (center="the", context="dog")

From "the cat": the pair is (center="the", context="cat")

So "the" appears as a center word 2 times total. The context word "dog" appears once and "cat" appears once.

To maximize the log-likelihood objective $\sum_{(x,y)\in D} \log P(y|x)$, the model parameters must be set such that the predicted distribution matches the empirical distribution of the training data. Specifically, for a center word x, the optimal probability $P(y|x)$ is given by:

$$P(y|x) = \frac{\text{count}(x, y)}{\sum_{y'} \text{count}(x, y')}$$

Maximum likelihood probabilities:

$$P(\text{dog} \mid \text{the}) = \frac{1}{2} = 0.5$$
$$P(\text{cat} \mid \text{the}) = \frac{1}{2} = 0.5$$
$$P(\text{a} \mid \text{the}) = 0$$
$$P(\text{the} \mid \text{the}) = 0$$

The last two are 0 because we never see those pairs in the data.

3.1.2 3b) Nearly Optimal Vector

We're given these fixed context vectors:

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

I need to find a word vector v_{the} that gives probabilities close to the optimal ones above.

In skip-gram, the score for each context word is calculated using the dot product: $\text{score}(y) = v_{\text{the}} \cdot c_y$. Then we use softmax to turn scores into probabilities.

To make $P(\text{dog} \mid \text{the})$ high and $P(\text{a} \mid \text{the})$ low, I need the score for "dog" to be much bigger than the score for "a".

Let me try $v_{\text{the}} = (-10, 10)$.

Calculating scores:

$$\text{Score(dog)} = (-10, 10) \cdot (0, 1) = 0 + 10 = 10$$
$$\text{Score(cat)} = (-10, 10) \cdot (0, 1) = 0 + 10 = 10$$
$$\text{Score(a)} = (-10, 10) \cdot (1, 0) = -10 + 0 = -10$$
$$\text{Score(the)} = (-10, 10) \cdot (1, 0) = -10 + 0 = -10$$

Now applying softmax (probabilities are proportional to e^{score}):

The scores for "dog" and "cat" are both 10, so they'll have equal probability. The scores for "a" and "the" are both -10, which is way smaller, so their probabilities will be nearly zero.

More precisely:

$$P(\text{dog} \mid \text{the}) = \frac{e^{10}}{e^{10} + e^{10} + e^{-10} + e^{-10}} = \frac{e^{10}}{2e^{10}} = 0.5$$

This matches the optimal probability exactly! Same logic works for the other probabilities. So $v_{\text{the}} = (-10, 10)$ is a good choice.

3.2 Q2: Skip-Gram with Four Sentences

Given sentences:

- the dog
- the cat
- a dog
- a cat

3.2.1 3c) Training Examples

With window size $k = 1$, each pair of neighboring words creates two training examples (one in each direction).

From "the dog": (the, dog) and (dog, the)
From "the cat": (the, cat) and (cat, the)
From "a dog": (a, dog) and (dog, a)
From "a cat": (a, cat) and (cat, a)
So we have 8 training pairs total.

3.2.2 3d) Nearly Optimal Vectors

Now I need to find both word vectors and context vectors that work well for all these training pairs.

Looking at the pattern: determiners ("the" and "a") should predict nouns ("dog" and "cat"), and nouns should predict determiners. Each should split probability 50-50 between the two valid options.

I'll use these vectors:

Context vectors:

$$c_{\text{dog}} = c_{\text{cat}} = (3, -7)$$
$$c_{\text{a}} = c_{\text{the}} = (-4, 2)$$

Word vectors:

$$v_{\text{the}} = v_{\text{a}} = (14, -6)$$
$$v_{\text{dog}} = v_{\text{cat}} = (-8, 11)$$

Why this works:

When "the" is the center word (determiner), it needs to predict "dog" or "cat":

$$\text{Score(dog)} = (14, -6) \cdot (3, -7) = 42 + 42 = 84 \quad \text{(high)}$$
$$\text{Score(a)} = (14, -6) \cdot (-4, 2) = -56 - 12 = -68 \quad \text{(low)}$$

Since 84 is way bigger than -68, the probability mass goes to "dog" and "cat" (which have the same score).

When "dog" is the center word (noun), it needs to predict "a" or "the":

$$\text{Score(a)} = (-8, 11) \cdot (-4, 2) = 32 + 22 = 54 \quad \text{(high)}$$
$$\text{Score(dog)} = (-8, 11) \cdot (3, -7) = -24 - 77 = -101 \quad \text{(low)}$$

Again, since 54 is way bigger than -101, the probability goes to "a" and "the".

The same logic applies when "a" or "cat" is the center word because they share vectors with "the" and "dog" respectively.

These vectors give probabilities within 0.01 of the optimal values (which is 0.5 for valid pairs and 0 for invalid pairs).

References

Used Claude for code correction (after completing my working draft) and improved writing (enhancement of my draft).