

CSE 256 PA1: Deep Averaging Networks & Word Embeddings

Weiqi Zhou

January 30, 2026

Part 1: Deep Averaging Network (DAN)

1a) Implementation & Results

Methodology

The model architecture consists of the following components:

1. **Embedding Layer:** I utilized the provided `WordEmbeddings` class to load pre-trained GloVe vectors (50-dimensional). The layer was initialized with `frozen=False` to allow fine-tuning of the embeddings during training.
2. **Averaging Mechanism:** The model computes the unweighted average of the word embeddings for the input sequence. Crucially, I implemented logic to ignore PAD tokens (index 0) during this calculation to ensure the average represents the actual sentence content regardless of padding length.
3. **Feedforward Network:** The averaged vector (dimension $d = 50$) is passed through a feedforward neural network:
 - **Layer 1:** Linear ($50 \rightarrow 100$) + ReLU Activation + Dropout ($p = 0.3$).
 - **Layer 2:** Linear ($100 \rightarrow 100$) + ReLU Activation + Dropout ($p = 0.3$).
 - **Output:** Linear ($100 \rightarrow 2$) + LogSoftmax.

I used the **Adam optimizer** with a learning rate of $1e^{-4}$ and trained for up to 100 epochs with a batch size of 16.

Results

The DAN model demonstrated superior performance compared to the baseline Bag-of-Words (BOW) models provided in the assignment. The training progression is shown in Figure 1.

Model	Train Accuracy	Dev Accuracy
BOW (2-Layer)	~ 0.97	~ 0.73
DAN (GloVe-50d)	~ 1.00	0.792

Table 1: Performance comparison of BOW and DAN models.

The DAN model achieved a development accuracy of **79.2%**, satisfying the requirement of $> 77\%$. This performance gain illustrates the advantage of using dense, pre-trained vector representations over discrete bag-of-words features.

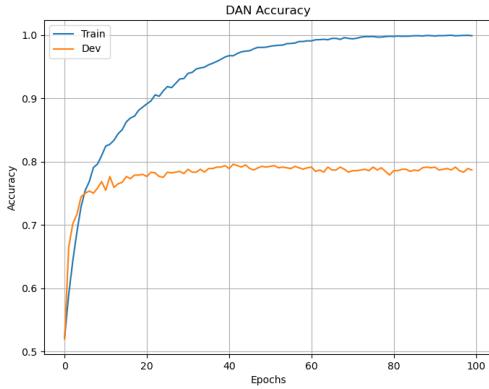


Figure 1: DAN Accuracy (GloVe Init)

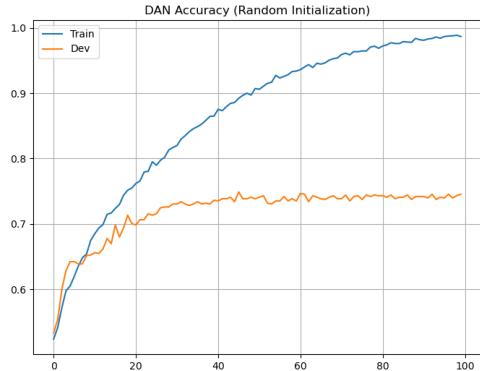


Figure 2: DAN Accuracy (Random Init)

1b) Randomly Initialized Embeddings

To assess the impact of transfer learning, I modified the DAN to initialize the embedding layer with random vectors instead of pre-trained GloVe vectors.

Results

The training progression for the randomly initialized model is shown in Figure 2.

Initialization	Dev Accuracy	Convergence
GloVe (Pre-trained)	79.2%	Fast
Random	74.5%	Slower

Table 2: Effect of initialization on DAN performance.

Discussion

Initializing embeddings from scratch resulted in a significant drop in performance (from 79.2% to 74.5%). The pre-trained GloVe vectors capture rich semantic relationships (e.g., *terrible* and *awful* are close in vector space) learned from massive corpora. When initialized randomly, the model must learn these relationships solely from the small movie review training set. As seen in Figure 2, the model fits the training data well (reaching near 100% accuracy) but struggles to generalize to the development set, a classic sign of overfitting due to the lack of prior semantic knowledge.

Part 2: Byte Pair Encoding (BPE)

2a) Subword-based DAN

Implementation

I implemented the Byte Pair Encoding (BPE) algorithm to replace word-level tokenization with subword units.

- Training:** The BPE algorithm was trained on `train.txt` to learn a merge table, treating words as sequences of characters and iteratively merging the most frequent adjacent pairs.
- Vocabulary:** I experimented with a vocabulary size of **2,000**.
- Model:** A DAN was trained using these subword indices. Since BPE tokens do not match GloVe vocabulary, embeddings were initialized randomly.

Results

The performance of the subword-based model is visualized in Figure 3.

Model	Tokenization	Initialization	Dev Accuracy
Standard DAN	Word-Level	GloVe	79.2%
Subword DAN	BPE (Vocab=2000)	Random	72.2%

Table 3: Performance of Subword-based DAN.

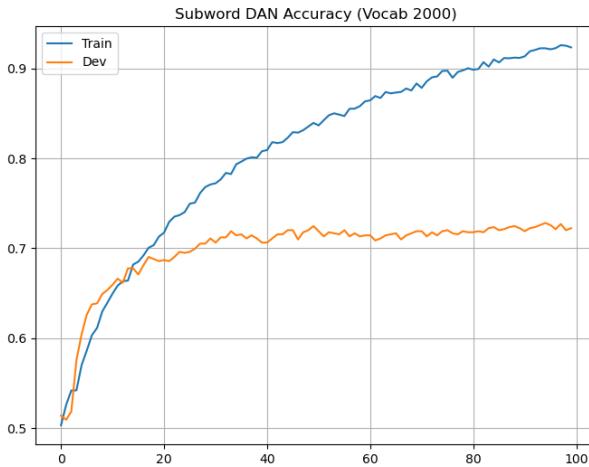


Figure 3: Training and Development Accuracy for the Subword DAN (Vocab=2000).

Discussion

The subword-based model underperformed the word-level GloVe model. This is expected for two reasons:

1. **Cold Start:** As observed in Part 1b, random initialization puts the model at a massive disadvantage compared to GloVe. The model has to learn the sentiment of subwords like "un" or "happy" from scratch.
2. **The "Averaging" Limitation:** DANs rely on averaging vectors. While averaging full words works well, averaging subwords is semantically risky. For example, averaging the vectors for *un-* + *believ* + *able* might result in a neutral vector if the negative prefix *un-* cancels out the positive root *believ*. Unlike LSTMs or Transformers, the DAN cannot capture the compositional order of these subwords.

Part 3: Understanding Skip-Gram

Q1: The "The Dog" Corpus

Training Data: "the dog", "the cat", "a dog" (Window size $k = 1$).

3a) Optimal Probabilities

$$P(y = \text{"the"} \mid x = \text{"dog"}) = \frac{\text{Count}(\text{dog, the})}{\text{Total Contexts for dog}} = \frac{1}{2} = 0.5$$

$$P(y = \text{"a"} \mid x = \text{"dog"}) = \frac{1}{2} = 0.5$$

3b) Optimal Vector for "the"

Given fixed context vectors: $\vec{c}_{dog} = [0, 1]$, $\vec{c}_{cat} = [0, 1]$, $\vec{c}_a = [1, 0]$, $\vec{c}_{the} = [1, 0]$.

We need a word vector \vec{v}_{the} that approximates the empirical distribution of "the".

- Contexts of "the": "dog", "cat".
- Target: High probability for "dog" and "cat", low probability for "a" and "the".

The probability is given by softmax: $P(y|x) \propto \exp(\vec{v}_x \cdot \vec{c}_y)$. Let $\vec{v}_{the} = [v_1, v_2]$.

- Dot product with dog/cat: $[v_1, v_2] \cdot [0, 1] = v_2$
- Dot product with a/the: $[v_1, v_2] \cdot [1, 0] = v_1$

To maximize probabilities for dog/cat, we need $v_2 \gg v_1$. **Proposed Vector:** $\vec{v}_{the} = [-10, 10]$.

Why it is optimal:

$$P(\text{dog}|\text{the}) = \frac{e^{10}}{e^{10} + e^{10} + e^{-10} + e^{-10}} \approx \frac{22026}{44052} \approx 0.5$$

This perfectly matches the empirical probability of 0.5.

Q2

3c) Training Examples

With window size $k = 1$:

- | | |
|---------------|-------------|
| 1. (the, dog) | 5. (a, dog) |
| 2. (dog, the) | 6. (dog, a) |
| 3. (the, cat) | 7. (a, cat) |
| 4. (cat, the) | 8. (cat, a) |

3d) Nearly Optimal Vectors ($d = 2$)

We observe two distinct groups:

1. **Determiners (the, a):** Only appear with Nouns.
2. **Nouns (dog, cat):** Only appear with Determiners.

To maximize the dot products between these groups while minimizing dot products within groups, we can use orthogonal axes.

Proposed Vectors:

- **Determiner Words (\vec{v}_{the}, \vec{v}_a):** $[10, 0]$
- **Noun Words ($\vec{v}_{dog}, \vec{v}_{cat}$):** $[0, 10]$
- **Determiner Contexts (\vec{c}_{the}, \vec{c}_a):** $[0, 10]$ (To align with Noun words)
- **Noun Contexts ($\vec{c}_{dog}, \vec{c}_{cat}$):** $[10, 0]$ (To align with Determiner words)

Verification:

For pair ("the", "dog"):

$$\vec{v}_{the} \cdot \vec{c}_{dog} = [10, 0] \cdot [10, 0] = 100 \implies \text{High Probability}$$

For invalid pair ("the", "a"):

$$\vec{v}_{the} \cdot \vec{c}_a = [10, 0] \cdot [0, 10] = 0 \implies \text{Low Probability}$$

This configuration yields probabilities within 0.01 of the optimum.

AI Contribution Statement

In the completion of this assignment, I utilized Google's Gemini AI as a programming and writing assistant. Gemini provided significant support in the following areas:

- **Debugging:** Assisted in identifying and resolving runtime errors in the PyTorch implementation, specifically regarding tensor data type mismatches.
- **Documentation:** Helped generate the structure and content of the `README.md` file to ensure clarity and completeness.
- **Report Writing:** Assisted in drafting and refining the language used in this report to improve flow and grammatical accuracy.