PA1 Report

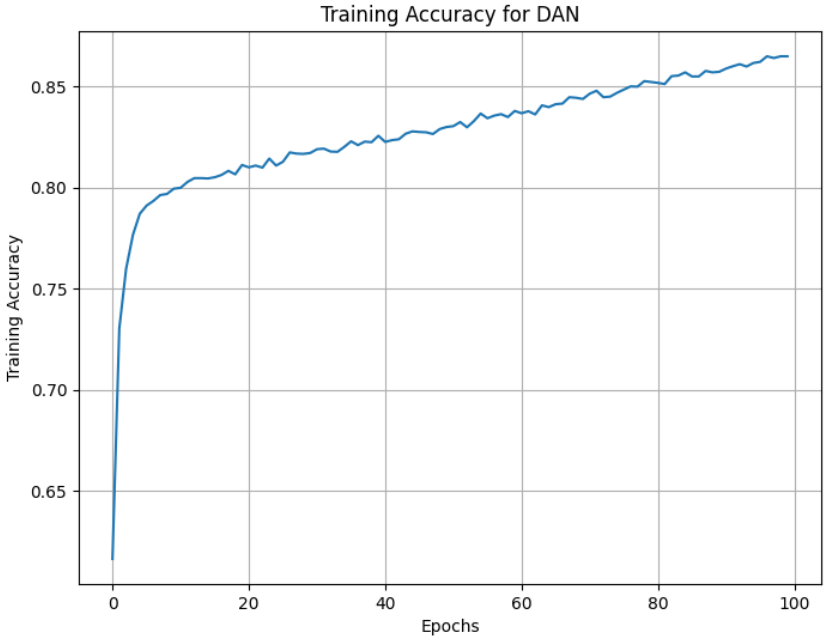
Kevin Lin

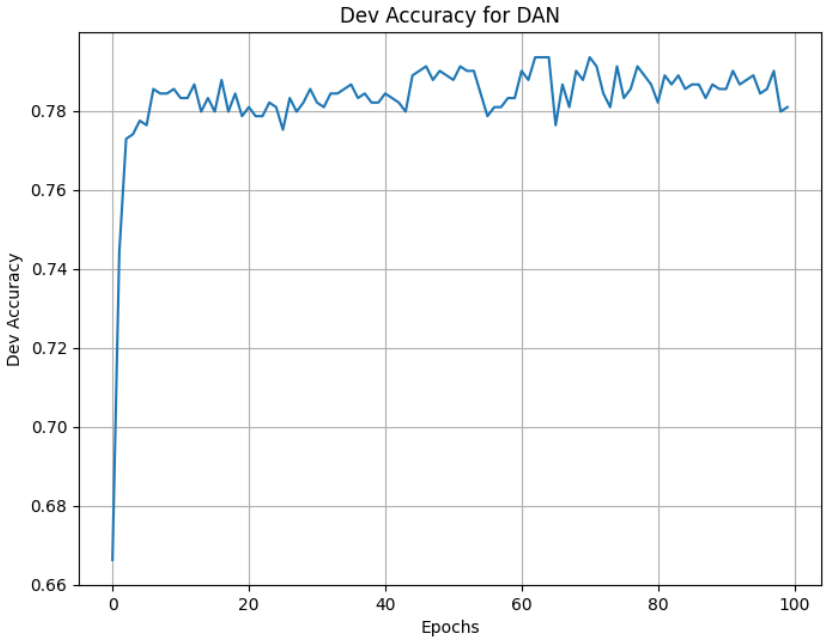
1/30/2025

Part 1: DAN

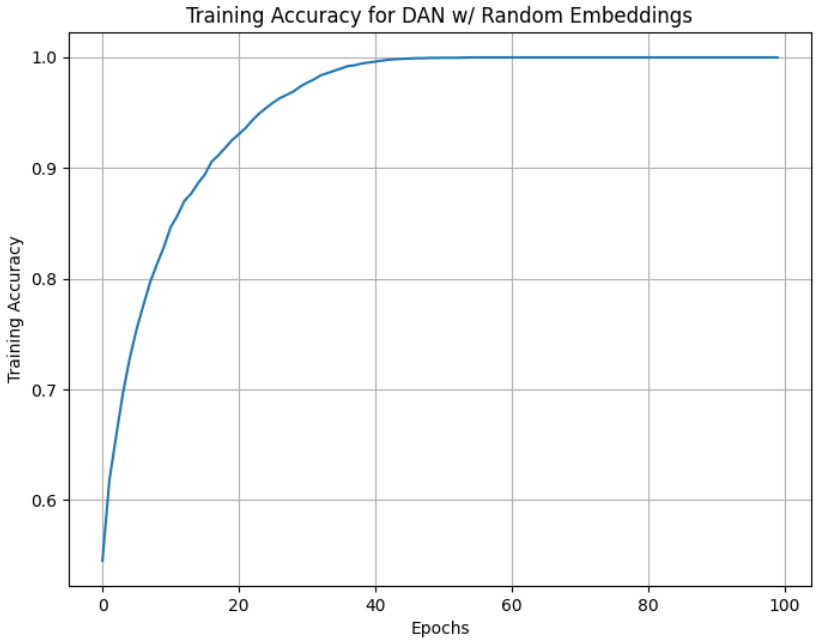
We implement the DAN as discussed in Iyyer et al. in `DANmodels.py`. The model follows a similar architecture as given in `BOWmodels.py`, with the addition of an averaging layer of the embeddings before passing to the feed-forward layers to effectively make it a deep averaging network. The primary difference lies in the dataset construction for the DAN model. This is handled by the `SentimentDatasetDAN` class implemented in `DANmodels.py`, which rather converts each sentence into a float vector of word embeddings like `SentimentDatasetBOW`, uses the given `WordEmbeddings` indexer class to convert words in each sentence to indices used in training. Unknown words are consequently given a special `<UNK>` token with index 1, while additional padding tokens `<PAD>` with index 0 are added to ensure uniform length across all sentences in a batch (`DAN_collate_fn` function, called during data loading). These are then passed to the DAN model. The averaging layer in the DAN model averages the embeddings of all words in the sentence, excluding any padding tokens to not skew the average. Training and dev accuracy testing utilize the same optimizer and loss function as the BOW models (Adam, NLL loss), and the training loop is similar as well (100 epochs, batch size 16, lr 0.0001).

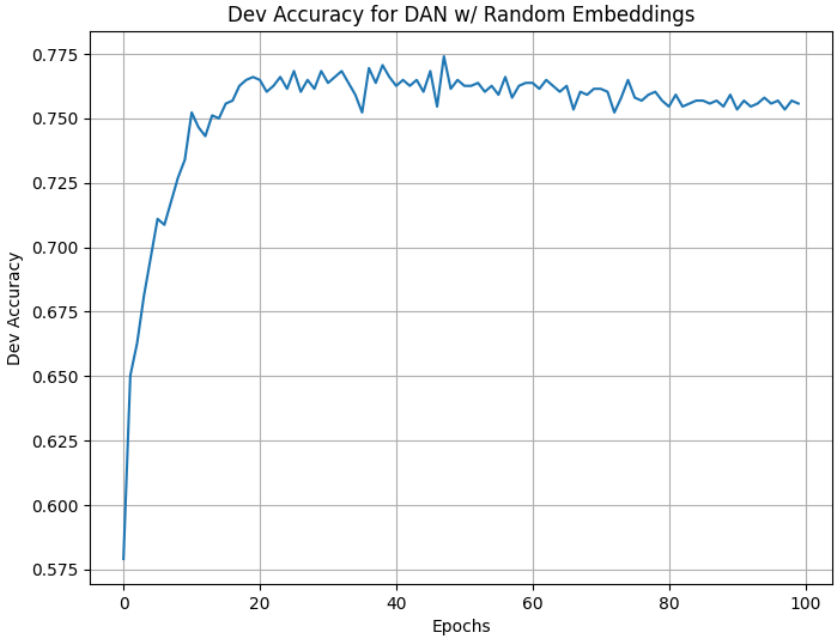
Testing with GloVe embeddings, `300d` performed significantly better than `50d`, achieving a dev accuracy of 0.781 and training accuracy of 0.865. See graphs:


Training Accuracy for DAN
Training Accuracy
0.85
0.80
0.75
0.70
0.65
0
20
40
60
80
100
Epochs


Dev Accuracy for DAN
Dev Accuracy
0.78
0.76
0.74
0.72
0.70
0.68
0.66
0
20
40
60
80
100
Epochs

Initializing the DAN model with random embeddings instead of GloVe embeddings simply required using `nn.Embedding` instead of the given `WordEmbeddings` class to create the embedding layer with the same dimensions. This model achieved a dev accuracy of 0.756, however by epoch 50 already had hit a training accuracy of 1.0, indicating overfitting to the training data. The slightly lower dev accuracy is expected due to the lack of pretraining on a large corpus like GloVe provides, as well as the overfitting observed. See graphs:

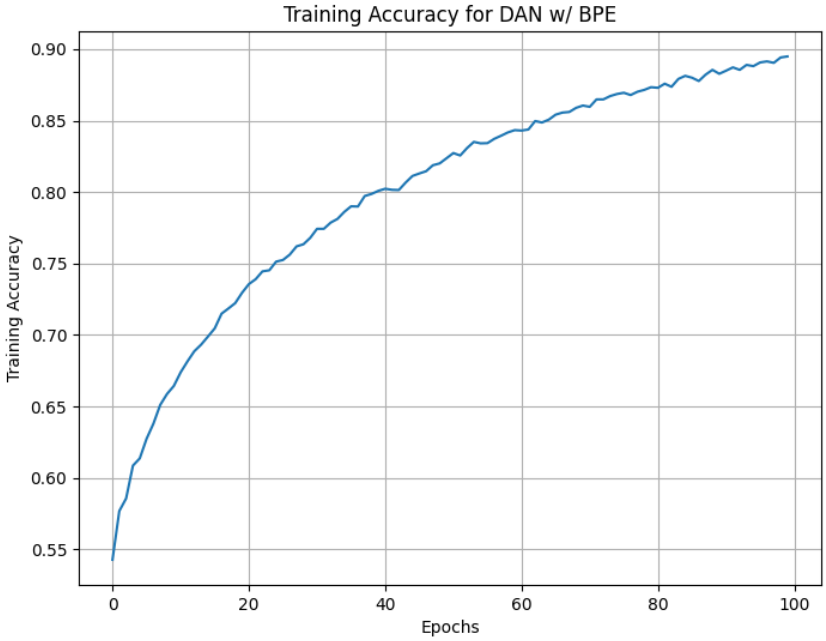


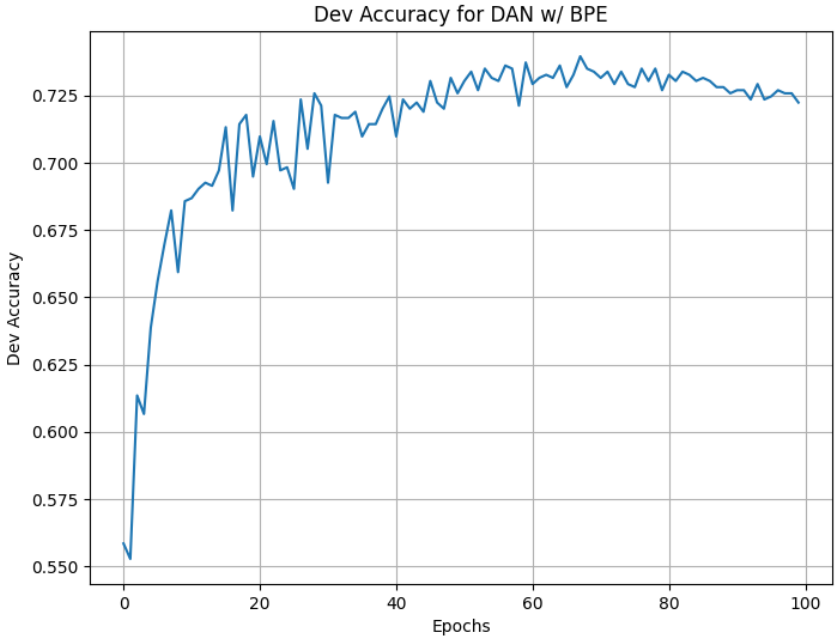


Part 2: BPE

We implement the BPE algorithm a discussed by Senrich et al. in `bpe.py`. The implementation follows the steps outlined in the paper, starting with splitting each sentence into its respective subwords and using the `BPETokenizer` class to handle the BPE operations. The `get_stats` function computes the frequency of each adjacent symbol pair in the corpus, while the `merge_vocab` function merges the most frequent pair into a new symbol. We again use the `<PAD>` and `<UNK>` tokens to handle padding and unknown subwords respectively, with the same indices. The BPE tokenizer is then used in the `SentimentDatasetBPE` class to convert sentences into sequences of subword indices for training the `DANBPE` model, which is similar to the DAN model but uses the BPE tokenizer and embeddings. Here, we use a vocabulary size of 5000 and embedding dimension of 100. Training and dev accuracy testing utilize the same optimizer and loss function as the previous models (Adam, NLL loss), and the training loop is similar as well (100 epochs, batch size 16, lr 0.0001). The BPE tokenizer on average is trained in about 2 minutes. The model achieves a dev accuracy of 0.722 and training accuracy of 0.895. Both `DAN_random` and `DANBPE` models perform worse than the original `DAN` model with GloVe embeddings, likely due to the lack of pretrained word embeddings on the given dataset like GloVe provides. Additionally, random embeddings are more prone to overfitting, as seen in the `DAN_random`

model, and subwords may not capture the full semantic meaning of words as effectively as full word embeddings. See graphs:





Part 3: Skip-Gram

Q1

3a) We are given the skip-gram model defined as:

$$P(\text{context} = y|\text{word} = x) = \frac{\exp(\mathbf{v}_x \cdot \mathbf{c}_y)}{\sum_{y'} \exp(\mathbf{v}_x \cdot \mathbf{c}_{y'})}$$

where x is the "center word", y is a "context word" being predicted, and $\mathbf{v}_x$ and $\mathbf{c}_y$ are d-dimensional vectors corresponding to words and contexts respecitvely. Each word has independent vectors for each, thus each word has two embeddings.
Given the sentences:

```
the dog
the cat
a dog
```

window size of $k = 1$, we get the training examples: $(x = \text{the}, y = \text{dog})$, $(x = \text{dog}, y = \text{the})$. Consequently, the skip-gram objective, log-likelihood is $\sum_{(x,y)} \log P(y|x)$. With word and context embeddings of dimension $d = 2$, the context embedding vectors w for *dog* and *cat* are

both $(0, 1)$, and the embeddings vectors w for *a* and *the* are $(1, 0)$. Thus, the set of probabilities $P(y|\text{the})$ that maximize the log-likelihood are:

$$P(y|\text{the}) = \begin{cases} \frac{1}{2} & y = \text{dog} \\ \frac{1}{2} & y = \text{cat} \\ 0 & y = \text{a} \\ 0 & y = \text{the} \end{cases}$$

3b) We want a setting $\mathbf{v}_{\text{the}}$ where $P(\text{dog}|\text{the}) \approx 0.5$, $P(\text{cat}|\text{the}) \approx 0.5$, and $P(\text{a}|\text{the}), P(\text{the}|\text{the}) \approx 0$. We know we can calculate $P(y|\text{the})$ as:

$$P(y|\text{the}) = \frac{\exp(\mathbf{v}_{\text{the}} \cdot \mathbf{c}_y)}{\sum_{y'} \exp(\mathbf{v}_{\text{the}} \cdot \mathbf{c}_{y'})}$$
$$= \frac{\exp(\mathbf{v}_{\text{the}} \cdot \mathbf{c}_y)}{\exp(\mathbf{v}_{\text{the}} \cdot \mathbf{c}_{\text{dog}}) + \exp(\mathbf{v}_{\text{the}} \cdot \mathbf{c}_{\text{cat}}) + \exp(\mathbf{v}_{\text{the}} \cdot \mathbf{c}_{\text{a}}) + \exp(\mathbf{v}_{\text{the}} \cdot \mathbf{c}_{\text{the}})}$$

We know that $\mathbf{c}_{\text{dog}} = \mathbf{c}_{\text{cat}} = (0, 1)$ and $\mathbf{c}_{\text{a}} = \mathbf{c}_{\text{the}} = (1, 0)$. Thus, we want the dot products of cat and dog to be very large and positive, while minimizing the dot products of a and the. Therefore, we can look for settings of $\mathbf{v}_{\text{the}}$ in a $(-C, C)$ format. We search for such values using a Python script `part3.py` by iterating through values from 1 upwards and calculating the probabilities until we find that a nearly optimal vector within 0.01 of the optimum, which is $\mathbf{v}_{\text{the}} = (-2, 2)$. Increasing the values further brings the probabilities closer to the target.

Q2

3c) With a word embedding space $d = 2$, the training examples derived from the sentences:

```
the dog
the cat
a dog
a cat
```

with window size $k = 1$ are:

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

3c) We see that each center word appears equally often with each context word. Thus, the optimal probabilities for each context word given a center word are:

$$P(\text{the}|\text{dog}) = P(\text{a}|\text{dog}) = 0.5$$
$$P(\text{the}|\text{cat}) = P(\text{a}|\text{cat}) = 0.5$$
$$P(\text{dog}|\text{the}) = P(\text{cat}|\text{the}) = 0.5$$
$$P(\text{dog}|\text{a}) = P(\text{cat}|\text{a}) = 0.5$$

and all other context words have probability 0. With context vectors:

$$\mathbf{c}_{\text{dog}} = \mathbf{c}_{\text{cat}} = (0, 1)$$
$$\mathbf{c}_{\text{a}} = \mathbf{c}_{\text{the}} = (1, 0)$$

we can find nearly optimal word vectors $\mathbf{v}_w$ for each word w using similar logic as in 3b. However, we also need to ensure that $\mathbf{v}_{\text{dog}}$ and $\mathbf{v}_{\text{cat}}$ this time give equal probabilities, and thus should follow $(C, -C)$ format instead. From our value in 3b, we then obtain the following vectors:

$$\mathbf{v}_{\text{the}} = (-2, 2)$$
$$\mathbf{v}_{\text{a}} = (-2, 2)$$
$$\mathbf{v}_{\text{dog}} = (2, -2)$$
$$\mathbf{v}_{\text{cat}} = (2, -2)$$

Running `part3.py` confirms that these vectors yield probabilities within 0.01 of the optimum.

LLM Usage: All work was done by myself in VSCode with GitHub Copilot integration. The integration "provides code suggestions, explanations, and automated implementations based on natural language prompts and existing code context," and also offers autonomous coding and an in-IDE chat interface that is able to interact with the current codebase. Only the Copilot provided automatic inline suggestions for both LaTex and Python in `.tex` and `.py` files respectively were taken into account / used.