CSE 256 Fall 2024, UCSD: PA1

Vedant Yogesh Deshpande
PID: A69032161

Part 1 : Deep Averaging Networks

This code implements a Deep Averaging Network (DAN) for sentiment analysis using PyTorch. The approach combines word embeddings with a neural network architecture to classify text sentiment.

Dataset Preparation : A custom dataset class that prepares the sentiment data for training. This class extends torch.utils.data.Dataset and is used to preprocess and serve sentiment data to the neural network.

1. It reads sentiment examples and GloVe word embeddings.
2. Sentences are tokenized and converted to word indices.
3. Labels are extracted from the examples.
4. Sentences are padded or truncated to a fixed length (default 50 words).
5. Unknown words are handled by replacing them with an "UNK" token.
6. Shorter sentences are padded with a "PAD" token.
7. Word indices and labels are converted to PyTorch tensors.
8. In the __getitem__() function, individual word indices and labels are used by the DataLoader during training and evaluation.

Model Architecture : The DAN class defines the neural network architecture:

1. Embedding Layer : Uses either pre-trained GloVe embeddings or randomly initialized embeddings. For pre-trained embeddings, it loads and initializes the embedding layer. For random initialization, it creates an embedding layer with specified input size and 50-dimensional embeddings.

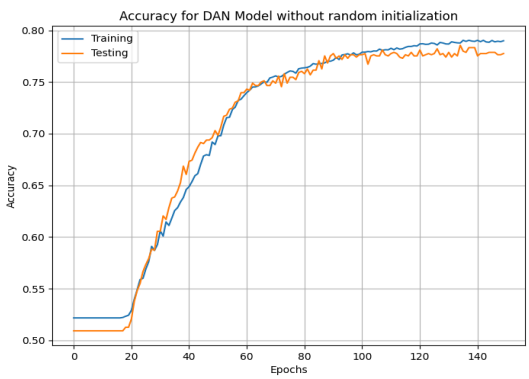
2. Neural Network Layers:
    I. Input Layer: Word embeddings (pre-trained or random)
    II. Hidden Layers:
        A. First fully connected layer (FC1): Maps from embedding dimension to hidden size
        B. Second fully connected layer (FC2): Hidden size to hidden size
        C. Third fully connected layer (FC3): Hidden size to 2 (output classes)
    III. Output Layer: Log softmax for classification probabilities

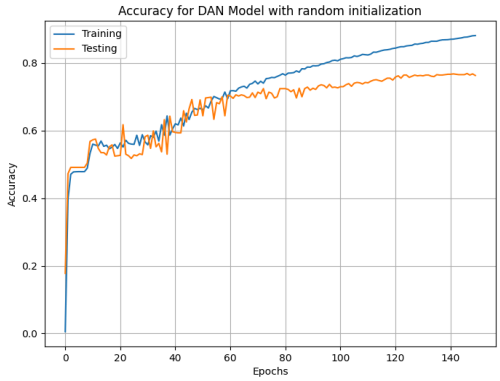
Results:

Table 1 - Hyperparameter Experimentation for DAN

Embedding Dimensions (GloVe size)	Hidden Size	Number of Layers	Train Accuracy	Dev Accuracy
50	10	3	70.3	71.9
50	10	4	71.1	72.8
50	100	3	73.36	73.27
50	100	4	73.41	73.96
300	10	3	77.90	77.01
300	10	4	75.88	75.40
300	100	3	80.86	78.32
300	**100**	**4**	**84.36**	**79.85**
300	100	5	85.08	79.37

Graph 1 - Q1A: Accuracy for DAN Model without random initialization
Graph 2 - Q1B: Accuracy for DAN Model with random initialization





Observations: The experiment revealed several insights into the model's performance based on hyperparameter tuning:

1. Embedding Dimension: Increasing the embedding dimension from 50 to 300 consistently improved both training and development accuracy (from 71.9% to 79.85% on the dev data), indicating that richer embeddings enable the model to capture more meaningful information.

2. Hidden Size and Architecture Depth: Larger hidden sizes and deeper architectures (more layers) generally enhance performance, with noticeable gains in accuracy up to a certain point.
3. Complexity Limitations: After reaching higher complexities (e.g. 5 layers), improvements in accuracy began to taper off, suggesting that additional capacity may not significantly benefit the model for this particular task.

Apart from the embedding dimensions, hidden size and number of layers, the following hyperparameters provided the optimal results.

1. Optimizer: Adam optimizer was utilized.
2. Loss Function: NLLLoss was applied.
3. Batch Size: A batch size of 64 was employed for training.
4. Learning Rate: The learning rate was set to 0.0001.
5. Epochs: The model was trained for a total of 150 epochs.

Table 2 - Comparison of DAN models without and with randomly initialized embeddings

DAN Model	Training Accuracy	Dev Accuracy
Without randomly initialized embeddings	84.36	79.85
With randomly initialized embeddings	88.09	76.38

As shown in Table 2, the development accuracies of the two DAN models are quite similar (79.65% and 76.38%), likely because the model effectively learns from the training data, even with randomly initialized embeddings. However, the discrepancy between the training accuracies offers a different perspective. The model with randomly initialized embeddings has lower development accuracy than the one with pre-trained embeddings, despite higher training accuracy. Additionally, as illustrated in Graph 2, the model with randomly initialized embeddings shows signs of overfitting, while the pre-trained model does not. This may be because random initialization forces the model to learn representations from scratch, fitting the training data more closely and struggling to generalize.

Part 2: Byte Pair Encoding

This code implements a Subword Deep Averaging Network (DAN) for sentiment analysis using PyTorch. The approach combines byte-pair encoding (BPE) with a neural network architecture to process and classify text data.

Dataset Preparation: A custom dataset class that prepares the sentiment data for training. This class extends torch.utils.data.Dataset and is used to preprocess and serve sentiment data to the neural network.

1. Uses BPE to build a subword vocabulary, allowing better handling of out-of-vocabulary and rare words.
2. Converts sentences into sequences of subword indices based on the vocabulary.
3. Pads or truncates sequences to a specified fixed length, ensuring consistency in input size for the neural network.

Model Architecture: The SubwordDAN class defines the neural network architecture:

1. Embedding Layer: Converts subword indices to dense vectors.
2. Neural Network Layers:
    a. Multiple fully connected layers with ReLU activation.
    b. Dropout layers for regularization.
3. Output layer with log softmax activation for binary classification.

Approach:

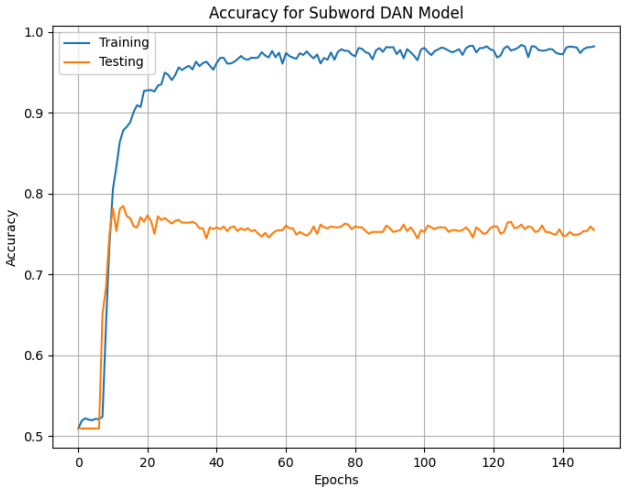
1. Tokenizing input text at the character level.
2. Applying BPE to construct a subword vocabulary.
3. Encoding sentences into sequences of subword indices using this vocabulary.
4. Padding or truncating the sequences to match a fixed length for uniform input.
5. The network processes these sequences through the defined layers to make predictions.

Results:

Table 3 - Comparison of number of merges and accuracy

Number of merges	Train accuracy	Dev accuracy
500	74.78	68.80
1000	87.41	73.05
2000	88.06	75.68
4000	**94.22**	**76.16**

Graph 3 - Accuracy for Subword DAN Model



Observations:

The data reflects a clear upward trend in performance metrics as the input size increases. Specifically, as the number of samples grows from 500 to 4000, the dev accuracy shows consistent improvement (from 68.6% for 500 merges to 76.16% for 4000 merges), suggesting that the model becomes more effective with a larger vocabulary. The gradual increase in performance metrics also suggests that the model benefits from more information, leading to better sentiment classification accuracy. However, as the number of merges increases, the model shows signs of overfitting.

Comparison of Subword DAN Model with the Word-level DAN Model:

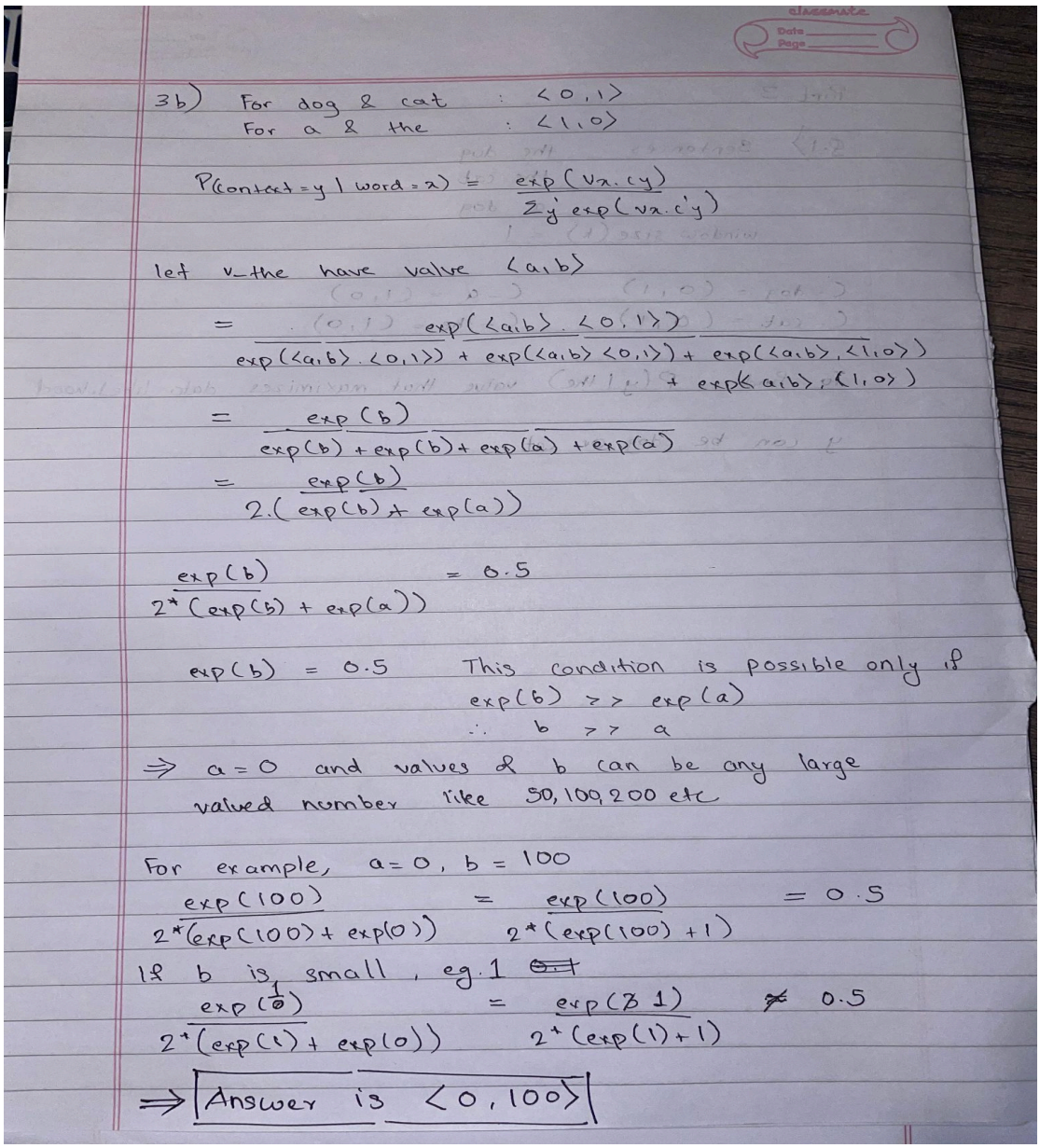
The accuracy of the DAN model (79.85%) for this sentiment analysis task is observed to be slightly higher than that of the Subword DAN model (76.16%). This is because the DAN model utilizes pre-trained embeddings during training, which provides an advantage. While the Subword DAN can approach the accuracy of the DAN model, it requires a significantly high number of merges, which is computationally and time-intensive. Given the same dataset, hyperparameters, and resources, the DAN model will most likely outperform the Subword DAN model for this task.

Part 3 : Skip-Gram

3A)

If we consider the provided data as our training set, we have two instances where we can fix the center word "the" and predict the corresponding context words. In the training data, there is one occurrence where the context word is "dog" with the center word "the," and another occurrence where the context word is "cat" for the same center word. Therefore, the probabilities are **P(cat | the) = 1/2** and **P(dog | the) = 1/2**

3B)



3C) The training examples derived from the given sentences are:

1. (x=the, y=dog)
2. (x=the, y=cat)
3. (x=a, y=dog)
4. (x=the, y=cat)
5. (x=dog, y=the)
6. (x=cat y=the)
7. (x=dog, y=a)
8. (x=cat, y=a)

3D)

References:

1. Let's build the GPT Tokenizer (https://www.youtube.com/watch?v=zduSFxRajkE)
2. Distributed Representations of Words and Phrases and their Compositionality, arXiv (https://arxiv.org/pdf/1310.4546)
3. PyTorch Documentation (https://pytorch.org/docs/stable/index.html)
4. ChatGPT for conceptual understanding
5. PPT provided in class