

CSE 256 PA1

Sentiment Analysis Model Report

Chuhan Wang

0. Introduction

This report presents a sentiment analysis project using the Deep Averaging Network (DAN) model. The main goal is to classify the sentiment of text data using different word embedding techniques and model architectures. The dataset used for this analysis contains text labeled with sentiments, and we aim to evaluate the performance of various configurations of the DAN model to find the optimal setup.

1. Deep Averaging Network (DAN)

1. Introduction

To perform sentiment analysis using a DAN, we follow three main steps:

(1) Data Preprocessing

(a) Convert words into pretrained embeddings.

I. This step transforms words into numerical representations, enabling us to perform computations using DAN in the following steps.

- I used GloVe embeddings, available in dimensions of 50 or 300.
- **GloVe** is a popular unsupervised learning algorithm designed to generate word embeddings by analyzing word co-occurrence statistics in large corpora. Each word is mapped to a high-dimensional vector, where similar words are positioned closely in the vector space. These embeddings can significantly improve downstream NLP tasks, as they encode semantic meaning in a compact form.

II. I used the *WordEmbeddings* class to facilitate the embedding process.

Implementation details:

- I used the function *read_word_embeddings()* to read word embeddings from the file *glove.6B.50d-relativized.txt*. I used *read_word_embeddings()* to read from the embedding file *glove.6B.50d-relativized.txt*. The function *read_word_embeddings()* reads the embeddings from the file *glove.6B.50d-relativized.txt*. This function initializes an *Indexer* object -- *word_indexer* to store the index of each word and a list *vectors* to store the corresponding embedding vectors.
- During the file-reading process, *word_indexer* and *vectors* are continuously updated and passed to the *WordEmbeddings* object. This allows us to convert each word in the sentences to its corresponding embedding when processing the training data.

(b) Convert words into randomly initialized embeddings.

In cases where no pretrained embeddings are provided, we can initialize the embeddings randomly. To do this, I used *nn.Embedding(vocab_size, embedding_dim)* to generate a randomly initialized embedding layer.

(2) Build a DAN model.

The core of DAN algorithm is to average the word embeddings to form a single, representative embedding for the entire sentence. This average embedding is then passed into a neural network (with 2, 3, or 4 layers) for further processing, eventually producing a two-dimensional vector

through a two-headed classifier to perform sentiment analysis.

(3) Train and evaluate the model.

- For the loss function, I used *nn.NLLLoss()* and *nn.CrossEntropyLoss()* for different experiments.
- The model was optimized using the *torch.optim.Adam* method.

2. Experiment Configurations and Results

Configurations:

(1) Default Configurations.	(2) Change single configuration.
<ul style="list-style-type: none"> • Data Preprocessing: Embedding Pretrained: True Embedding file: <i>glove.6B.50d-relativized.txt</i>. Embedding dimension: 50 • DAN model: Number of layers: 3 Hidden layer size: 100 • Model training: Optimizer: Adam Dropout layer: None (0) Loss function: NLL Loss Learning rate: 0.0001 	<ul style="list-style-type: none"> • Embedding Pretrained: False 1(b) • Embedding dimension: 300 • Number of layers: 2, 4 • Hidden layer size: 50; 200 • Loss function: CrossEntropyLoss • Learning rate: 0.001

Results Comparison:

	Training accuracy	Developing accuracy
Default Configurations	A line graph titled 'Training Accuracy for DAN Networks' showing training accuracy over 100 epochs. The y-axis ranges from 0.6 to 1.0. The accuracy starts at approximately 0.65 and rises sharply to about 0.95 by epoch 20, then continues to rise more gradually, reaching nearly 1.0 by epoch 100.	A line graph titled 'Dev Accuracy for DAN Networks' showing developing accuracy over 100 epochs. The y-axis ranges from 0.55 to 0.80. The accuracy starts at approximately 0.55, rises sharply to about 0.78 by epoch 20, and then fluctuates between 0.75 and 0.80 for the remainder of the training.
Embedding Pretrained: False 1(b)	A line graph titled 'Training Accuracy for DAN Networks' showing training accuracy over 100 epochs. The y-axis ranges from 0.5 to 1.0. The accuracy starts at approximately 0.55 and rises steadily to about 0.95 by epoch 100.	A line graph titled 'Dev Accuracy for DAN Networks' showing developing accuracy over 100 epochs. The y-axis ranges from 0.50 to 0.75. The accuracy starts at approximately 0.50, rises to about 0.70 by epoch 20, and then fluctuates between 0.70 and 0.75 for the remainder of the training.
Embedding dimension: 300	A line graph titled 'Training Accuracy for DAN Networks' showing training accuracy over 100 epochs. The y-axis ranges from 0.6 to 1.0. The accuracy starts at approximately 0.65 and rises sharply to about 0.95 by epoch 20, then continues to rise more gradually, reaching nearly 1.0 by epoch 100.	A line graph titled 'Dev Accuracy for DAN Networks' showing developing accuracy over 100 epochs. The y-axis ranges from 0.650 to 0.825. The accuracy starts at approximately 0.650, rises to about 0.825 by epoch 20, and then fluctuates between 0.750 and 0.825 for the remainder of the training.

Number of layers: 2	A line graph showing training accuracy for a 2-layer DAN network over 100 epochs. The accuracy starts at approximately 0.65 and rises steadily to reach nearly 1.0 by epoch 100.	A line graph showing development accuracy for a 2-layer DAN network over 100 epochs. The accuracy starts at approximately 0.65, peaks at about 0.80 around epoch 20, and then fluctuates between 0.75 and 0.80.
Number of layers: 4	A line graph showing training accuracy for a 4-layer DAN network over 100 epochs. The accuracy starts at approximately 0.65 and rises to reach nearly 1.0 by epoch 100.	A line graph showing development accuracy for a 4-layer DAN network over 100 epochs. The accuracy starts at approximately 0.65, peaks at about 0.80 around epoch 20, and then fluctuates between 0.75 and 0.80.
Hidden layer size: 50	A line graph showing training accuracy for a DAN network with 50 hidden units over 100 epochs. The accuracy starts at approximately 0.65 and rises to reach nearly 1.0 by epoch 100.	A line graph showing development accuracy for a DAN network with 50 hidden units over 100 epochs. The accuracy starts at approximately 0.65, peaks at about 0.80 around epoch 20, and then fluctuates between 0.75 and 0.80.
Hidden layer size: 200	A line graph showing training accuracy for a DAN network with 200 hidden units over 100 epochs. The accuracy starts at approximately 0.65 and rises to reach nearly 1.0 by epoch 100.	A line graph showing development accuracy for a DAN network with 200 hidden units over 100 epochs. The accuracy starts at approximately 0.65, peaks at about 0.80 around epoch 20, and then fluctuates between 0.75 and 0.80.
Loss function: CrossEntropyLoss	A line graph showing training accuracy for a DAN network using CrossEntropyLoss over 100 epochs. The accuracy starts at approximately 0.65 and rises to reach nearly 1.0 by epoch 100.	A line graph showing development accuracy for a DAN network using CrossEntropyLoss over 100 epochs. The accuracy starts at approximately 0.65, peaks at about 0.80 around epoch 20, and then fluctuates between 0.75 and 0.80.
Learning rate: 0.001	A line graph showing training accuracy for a DAN network with a learning rate of 0.001 over 100 epochs. The accuracy starts at approximately 0.65 and rises to reach nearly 1.0 by epoch 100.	A line graph showing development accuracy for a DAN network with a learning rate of 0.001 over 100 epochs. The accuracy starts at approximately 0.65, peaks at about 0.80 around epoch 20, and then fluctuates between 0.75 and 0.80.

Analysis and Conclusion from Results:

- **Without pretrained embeddings**, the dev accuracy is lower compared to the model with default configurations. It takes about 40 epochs to reach 75% accuracy, while the default configuration achieves the same in just 3 or 4 epochs.
- **Embedding dimension of 300** leads to rapid convergence in training accuracy, reaching nearly 1 very quickly. However, the development accuracy declines significantly after peaking at 82% around epoch 6, which is a clear indication of overfitting.
- **DAN models with hidden layer sizes of 50, 100, and 200** show that the model with a hidden

layer size of 100 has the most stable and best performance on development accuracy.

- **DAN models with 2, 3, and 4 layers** demonstrate that the model with 3 layers consistently provides the most stable and highest development accuracy.

- **CrossEntropyLoss and NLLoss** exhibit similar performance over 100 epochs, both in terms of training and development accuracy.

- **Learning rate** is a crucial hyperparameter. If it is too high, the model may converge too quickly, leading to instability and overfitting. If it is too low, convergence becomes excessively slow, preventing the model from reaching optimal accuracy. Based on comparisons, a learning rate of 0.0001 proves to be more effective than 0.001 for stable and accurate performance.

2. Byte Pair Encoding (BPE)

1. Introduction

In this section, we replace word-level tokenization in the DAN with subword tokenization using the Byte Pair Encoding (BPE) algorithm, based on Sennrich et al. [3]. We trained BPE on the `'train.txt'` dataset to generate subword units, which are then used in the DAN model instead of the original word-level vocabulary.

2. Steps

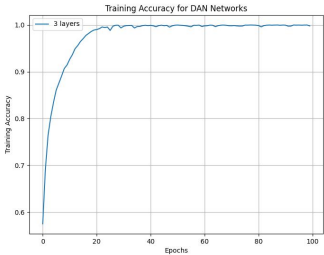
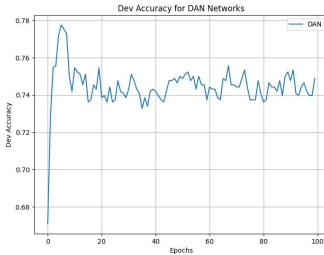
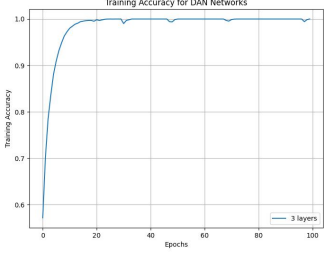
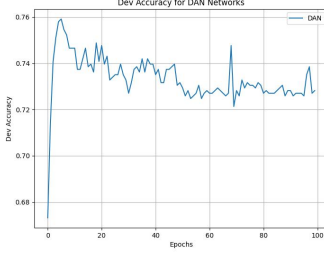
- First, **generating subword dictionary**. Code module `'BPE.py'` to generate a file `'subword_vocab.txt'`, aiming to merge characters into subwords. In this section, vocabulary size (merge_size) can be changed.

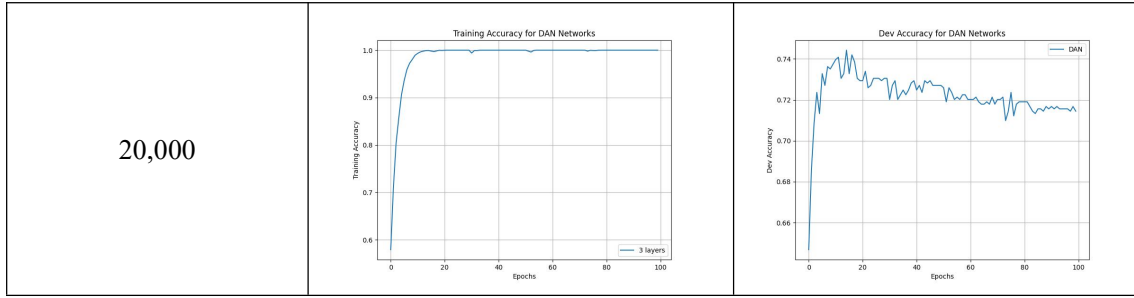
- Second, **preprocessing data**. Build a new class `SentimentDatasetSubwordDAN(Dataset)` in module `'DANmodels.py'`,

aiming to turn words into a list with indexes as elements; the indexes are subword indexes according to the subword dictionary.

- Lastly, **training and evaluating DAN model**. Send the preprocessed data into DAN model, which has a randomly initialized embedding layer, and we can get the performance of training and developing accuracy.

3. Experiment Configurations and Results

Vocabulary size	Training accuracy	Developing accuracy
5,000		
10,000		



Analysis and Conclusion from Results:

- **Subword-based DAN models with vocabulary sizes of 5000, 10,000, and 20,000** show that as the vocabulary size increases, training accuracy improves, but development accuracy decreases. The trends in development accuracy are similar across all sizes: they reach their peak (around 78%, 76%, and 74%, respectively) around epoch 10 and then decline with fluctuations.
- **Compared to the word-level DAN model**, the subword-based DAN exhibits better training accuracy, reaching nearly 1 before epoch 20. However, the development accuracy is consistently lower, failing to surpass 80%.

3. Understanding Skip-Gram

Q1:

3a) Given the training set in Q1, the word ‘the’ appear twice: ‘the dog’ and ‘the cat’ respectively. Thus, the empirical probabilities are:

$$\begin{cases} P(\text{dog} \mid \text{the}) = \frac{1}{2} \\ P(\text{cat} \mid \text{the}) = \frac{1}{2} \end{cases}$$

The set of probabilities $P(y \mid \text{the})$ that maximizes the data likelihood are:

$$\begin{cases} P(\text{dog} \mid \text{the}) = \frac{1}{2} \\ P(\text{cat} \mid \text{the}) = \frac{1}{2} \end{cases}$$

3b) Assume the word vector for ‘the’ is (t_1, t_2) .

According to the skip-gram model:

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

$$P(\text{context} = \text{dog} \mid \text{word} = \text{the}) = P(\text{context} = \text{cat} \mid \text{word} = \text{the}) = \frac{e^{t_2}}{2(e^{t_1} + e^{t_2})} = \frac{1}{2} \pm 0.01$$

$(t_1, t_2) = (0, 5)$ satisfy the above requirements.

Thus, the word vector for ‘the’ can be $(0, 5)$.

Q2:

3c) Training examples derived from Q2 samples:

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

3d) Based on the samples, ‘dog’ and ‘cat’ share the same word vector and context vector; ‘a’ and ‘the’ also share the same word and context vector.

Therefore, context embedding vectors for ‘dog’ and ‘cat’ can be both (0, 1), and the context embedding vectors for ‘a’ and ‘the’ can be both (1, 0).

Same to the computation in **3b)**, the word vectors can be easily derived from skip-gram models.

Word embedding vectors for ‘dog’ and ‘cat’ can be both (5, 0), and word embedding vectors for ‘a’ and ‘the’ can be both (0, 5).

	Word embedding vector	Context embedding vector
dog	(5, 0)	(0, 1)
cat	(5, 0)	(0, 1)
a	(0, 5)	(1, 0)
the	(0, 5)	(1, 0)

4. References

- [1] Mohit Iyyer, Varun Manjunatha, Jordan L. Boyd-Graber, and Hal Daum’e III. Deep unordered composition rivals syntactic methods for text classification. In *ACL*, pages 1681–1691. The Association for Computer Linguistics, 2015.
- [2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *EMNLP*, pages 1532–1543. ACL, 2014.
- [3] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith, editors, *ACL*, pages 1715–1725, 2016.
- [4] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a senti ment treebank. In *ACL*, pages 1631–1642. ACL, 2013.