2024 FALL CSE-256 PA1 Submission

Tang Sheng tasheng@ucsd.edu

October 17, 2024

Abstract

This report presents the implementation and analysis of CSE-256 PA1 questions. The content includes Deep Averaging Network (DAN), pretrained GloVe word embedding, randomly intialized embedding, BPE tokenization technique, and Skip-Gram embedding training. In this report, I will also discuss the experiments of model configurations and hyperparameters to optimize performance.

Contents

1 Deep Averaging Network (DAN) with GloVe Pretrained Embedding

1.1 Model Architecture

The implemented DAN consists of an embedding layer followed by averaging over word embeddings and one or more feedforward layers leading to the output layer. Below is the code implementation for the GloVe-based DAN model, which deploys pretrained GloVe word embedding, a 2-hidden-layer design and a hidden states dimension of 100.

```
class DAN2Model_GloVe(nn.Module):
    def __init__(self, embedding_dim, hidden_size=100):
        super().__init__()
        self.embedding_dim = embedding_dim
        embeddings_file = f'data/glove.6B.{embedding_dim}d-relativized.txt'
        self.embedding_class = read_word_embeddings(embeddings_file)
        self.embedding_layer = self.embedding_class.get_initialized_embedding_layer
        self.output_size = 2
        self.activation = nn.ReLU()
        self.layer1 = nn.Linear(embedding_dim, hidden_size)
        self.layer2 = nn.Linear(hidden_size, self.output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = x.int()
        embeddings = self.embedding_layer(x)
        # Compute the average embedding
        hidden_states = torch.mean(embeddings, dim=1)
        hidden_states = self.layer1(hidden_states)
        hidden_states = self.activation(hidden_states)
        hidden_states = self.layer2(hidden_states)
        hidden_states = self.activation(hidden_states)
        return self.softmax(hidden_states)
```

The DAN2Model_GloVe class utilizes GloVe word embeddings, with a customizable embedding dimension, followed by two linear layers with ReLU activations. The embeddings are averaged before being passed through the network. A LogSoftmax layer is used to produce the output probabilities.

In addition, a new Dataset class is implemented as below demonstration. The important detail in implementing this customized class lies in the ”padding” technique.

1.1.1 Variable Experimental Factors

- **Number of Layers:** Experiments were conducted with 2 and 3 hidden layers.
- **Embedding Sources:** Both 50-dimensional and 300-dimensional GloVe embeddings were used.

1.1.2 Constant Experimental Factors

- **Optimizer:** The Adam optimizer was used with default parameters.
- **Nonlinearity:** ReLU activation functions were applied after hidden layers.

- **Hyperparameters** A learning rate of 5e-5 is deployed. Total number of training epochs is set to be 100.

1.2 Results

Table 1: Performance of DAN with Different Configurations

Configuration	Train Accuracy (%)	Dev Accuracy (%)
2 layers, 50d embeddings	95.4	79.6
3 layers, 50d embeddings	93.4	79.6
2 layers, 300d embeddings	94.1	**81.7**
3 layers, 300d embeddings	96.2	80.6

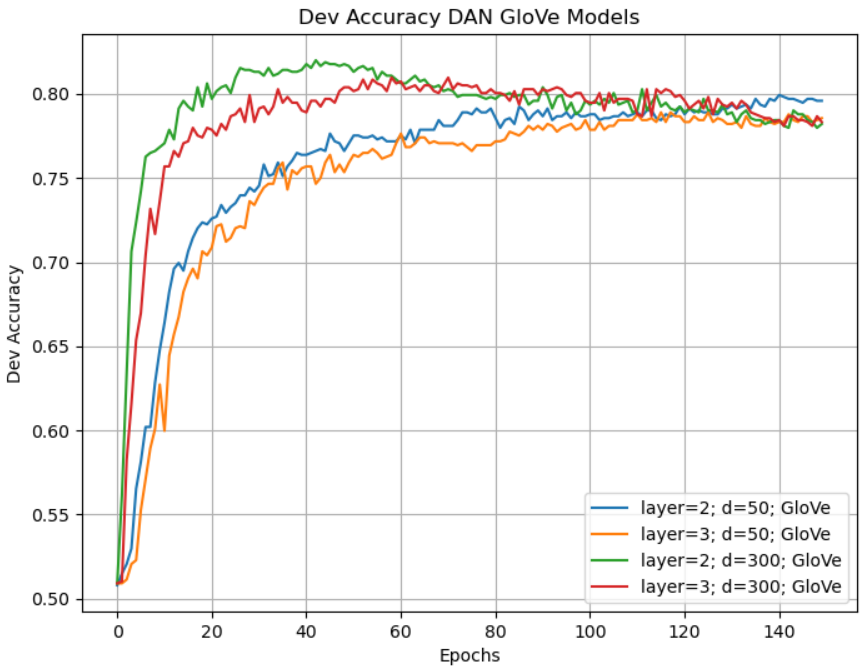

Figure 1: Architecture of the Deep Averaging Network (DAN) with GloVe embeddings.

1.3 Discussion

The result in the table suggests that while richer embeddings capture more semantic information and boost performance, simply adding more layers may not yield better results and need more experiments to justify the relationship between the number of layers and the ability of model generalization. Therefore, a 2-layer DAN with 300-dimensional embeddings provides an optimal balance between model complexity and performance.

2 DAN with Randomly Initialized Embeddings

2.1 Implementation Details

The embedding layer was modified to initialize embeddings randomly instead of using pre-trained GloVe embeddings. These embeddings were updated during training. For implementation details, see below code example. In the below code snippet, "..." represents content that are exactly the same as or quite similar with the code snippet in section 1.1.

```
class DAN2Model_Random(nn.Module):
def __init__(self, embedding_dim, hidden_size=100):
    ...
    self.vocab_size = self.embedding_class.vectors.shape[0]
    self.embedding_layer = nn.Embedding(self.vocab_size, self.embedding_dim)
    ...
def forward(self, x):
    ...
```

2.2 Results and Comparison

Table 2: Comparison between GloVe and Random Embeddings

# Layers	Embedding	Train Acc(%)	Dev Acc(%)
2	GloVe 300d	94.1	**81.7**
2	Random 300d	97.1	77.5
3	GloVe 300d	96.2	**80.6**
3	Random 300d	97.5	76

2.3 Discussion

The above table presents a comparison between GloVe and Random embeddings in terms of training and development accuracy when using two and three-layer models. The GloVe embedding demonstrates consistently higher development accuracy across different layer settings, which suggests that pre-trained embeddings help models generalize better compared to randomly initialized embeddings. Future experiments may explore the performance across more embedding dimensions or various layer depths to assess the scalability of these findings.

3 BPE-based DAN Model

3.1 Model Description and implementation details

A Byte Pair Encoding (BPE) tokenizer was used to split words into subword units. The DAN model was then trained using these subword tokenization and randomly initialized subword embeddings. I designed and implement a BPE_Tokenizer class to train a subword-level tokenizer and to tokenize any given sentences accordingly. Below code snippet demonstrate the basic skeleton of the tokenizer class. For the detailed implementation please refer to the attached codes.

```
class BPE_Tokenizer:
def __init__(self, dataset, vocab_size_limit):
    ...
    self.train()

def get_vocab(self):
    ...

def get_stats(self):
    ...

def merge_vocab(self, pair):
    ...

def train(self):
    while len(self.vocab) < self.vocab_size_limit:
        pairs = self.get_stats()
        best = max(pairs, key=pairs.get)
        self.merge_vocab(best)
        self.merges[best] = ''.join(best)

def get_vocab_dict(self):
    ...

def build_index_vocab(self):
    ...

def tokenize(self, list_of_words):
    ...
    return tokenized_indices
```

3.2 Experiments with different vocabulary sizes and embedding dimensions

Experiments were conducted with vocabulary sizes of 2,500, 5,000, 7,500, and all of them are deployed with a randomly initialized word embedding of dimension size of 50.

3.3 Results

Table 3: Performance of Subword-based DAN with Different Vocabulary Sizes

Vocabulary Size	**Train Accuracy(%)**	**Dev Accuracy (%)**
2000 tokens	93.6	**75.5**
5000 tokens	93.7	74.7
7,500 tokens	88.6	73.6

3.4 Comparison with Word-level DAN

When comparing the subword-based DAN to the word-level DAN implemented earlier, we observe that the word-level model outperforms the subword-based model in terms of development accuracy. The word-level DAN achieved a development accuracy of up to 81.7% with 300-dimensional embeddings, whereas the best subword-based model with a vocabulary size of 2,000 tokens reached a development accuracy of 75.5%. This indicates that the word-level model is more effective for this particular task and dataset.

3.5 Discussion

The subword-based model underperforms compared to the word-level DAN, which achieved higher development accuracy. Possible reasons include the dataset favoring whole-word representations, the subword model requiring architectural adjustments to match the word-level model's capacity, and potential information loss due to subword tokenization. Future experiments could involve tweaking hyperparameters, trying different subword segmentation techniques, or combining word-level and subword-level embeddings to improve performance.

4 Skip-Gram Model Analysis

4.1 Q1 - (a)

Let's find empirical probability distribution. We have (x = the, y = dog) once, and (x = the, y = cat) once. So there are only two observations for the events (x = the). Thus we have

$$P(\text{dog}|\text{the}) = \frac{1}{2}, \quad P(\text{cat}|\text{the}) = \frac{1}{2}$$

4.2 Q1 - (b)

Answer: we can have word vector for "the" to be (0,k), where k is an arbitrarily large number.
Proof: We have formula

$$P(y|x) = \frac{\exp(v_x \cdot c_y)}{\sum_{y'} \exp(v_x \cdot c_{y'})}$$

where x here represents the embedding for the center word "the", and y here represents the embedding for the context word around it. From the problem, we have dimension d = 2, the context embedding vectors w for dog and cat are both (0, 1), and the context embedding vectors w for a and the are (1, 0). Therefore, setting $v_{\text{the}} = (0, k)$ will lead the formular to be

$$P(cat|the) = P(dog|the) = \frac{\exp(k)}{\exp(k) + \exp(k) + \exp(0) + \exp(0)} = \frac{\exp(k)}{2\exp(k) + 2} \approx 1/2$$

Thus, a very large k will make satisfy the conditions.

4.3 Q2 - (a)

With a window size of $k = 1$, the skip-gram model considers the neighbors of a word as the words directly on either side. Given the sentences, the complete set of training examples is:

$$\begin{aligned}\{&(x = \text{the}, y = \text{dog}), (x = \text{dog}, y = \text{the}),\\ &(x = \text{the}, y = \text{cat}), (x = \text{cat}, y = \text{the}),\\ &(x = \text{a}, y = \text{dog}), (x = \text{dog}, y = \text{a}),\\ &(x = \text{a}, y = \text{cat}), (x = \text{cat}, y = \text{a})\}\end{aligned}$$

4.4 Q2 - (b)

Answer:
context embedding

$$u_{dog} = (1, 0), u_{cat} = (1, 0), u_{the} = (0, 1), u_a = (0, 1)$$

word embedding

$$v_{dog} = (0, 5), v_{cat} = (0, 5), v_{the} = (5, 0), v_a = (5, 0)$$

Proof:
We compute empirical probabilities, and for center word "the", we have

$$P(dog|the) = 0.5, P(cat|the) = 0.5, P(a|the) = 0, P(the|the) = 0$$

For other center words, this is similar. This means, center word "the" desires "dog" or "cat" near it, and don't desire "a" or "the" near it. Similar situations can be deducted for other center words. So we set context vectors as

$$u_{dog} = (1, 0), u_{cat} = (1, 0), u_{the} = (0, 1), u_a = (0, 1)$$

And we define embedding vectors as

$$v_{dog} = (0, a), v_{cat} = (0, b), v_{the} = (c, 0), v_a = (d, 0)$$

Now lets compute

$$a, b, c, d$$

Notice

$$P(dog|the) = \frac{\exp(c)}{2\exp(c) + 2} \approx 1/2$$

So c should be very large. Similar for other center words, a, b, c, d should be both very large such that the given probabilities within 0.01 of the optimum. To achieve $\frac{\exp(x)}{2\exp(x)+2} > 0.49$, we will have $2 + 2/\exp(x) < 0.49/1$, and thus get $x > \ln 49$, and it's safe to have $x >= 5$. So the final result can be

$$u_{dog} = (1, 0), u_{cat} = (1, 0), u_{the} = (0, 1), u_a = (0, 1)$$

and

$$v_{dog} = (0, 5), v_{cat} = (0, 5), v_{the} = (5, 0), v_a = (5, 0)$$