CSE256 HomeWork1

Author: Zhecheng Li (PID: A69033467)

How to run the code?

Answer: Detailed instructions are available in the `README.md` file; simply modify the `config.yaml` file slightly, then run `python main.py`.

Part 1

1.1 Main changes to design our new DAN model

- I set `hidden_sizes` as a list to conveniently define the hidden dimensions for all layers in the MLP architecture across the entire DAN model.

```
layers = []
self.prev_size = self.input_size
for hidden_size in self.hidden_sizes:
    layers.append(nn.Linear(self.prev_size, hidden_size))
    layers.append(nn.ReLU())
    self.prev_size = hidden_size
self.hidden_layers = nn.ModuleList(layers)
```

- I set the final output dimension of the DAN model to `1` and applied the `sigmoid` function to obtain the final label.

```
self.final_layer = nn.Linear(self.output_size, self.num_classes) # Here
self.num_classes = 1
correct += ((torch.sigmoid(pred) > 0.5) == y).type(torch.float).sum().item()
```

- I rewrote the `SentimentDatasetDAN` class to load the training dataset using pre-trained GloVe embeddings. An `UNK` token is used for unknown words, and a `PAD` token is applied to pad all sentences to a length of either `32` or `64` (configurable in `config.yaml`), allowing for batch training with a `batch_size` greater than `1`.

```
for word in sentence:
    if embs.word_indexer.index_of(word) == -1:
        sentence_list.append(embs.word_indexer.index_of("UNK"))
    else:
        sentence_list.append(embs.word_indexer.index_of(word))
if len(sentence_list) > max_length:
    sentence_list = sentence_list[:max_length]
while len(sentence_list) < max_length:
    sentence_list.append(embs.word_indexer.index_of("PAD"))
```

- I modified the DAN model code to allow users to choose between randomly initialized embeddings and pre-trained GloVe embeddings, and added a parameter to enable or disable training of the embedding layer.

```
if self.use_random_embed:
    self.random_embedding = nn.Embedding(self.vocab_size, self.input_size)
    if self.freeze_embed:
        self.random_embedding.weight.requires_grad = False
    else:
        self.random_embedding.weight.requires_grad = True
else:
    self.pretrained_embedding =
read_word_embeddings(embeddings_file=self.embed_file).get_initialized_embedding_l
ayer(self.freeze_embed)
```

- I added a `dropout rate` to the DAN model to help prevent overfitting on the training data.

```
if self.use_dropout:
    layers.append(nn.Dropout(self.dropout_rate))
```

1.2 The experimental results with pre-trained GloVe embeddings.

To explore the relationship between various hyperparameters and final accuracy, I conducted numerous experiments with different parameter settings. The results are listed in the table below. (Note: each experiment was trained for up to `20` epochs, as the DAN model's performance typically converges by then, with a learning rate of `1.0e-4` throughout).

Exp Id	max_length	embed_file	freeze_embed	hidden_sizes	dropout_rate	Best Train Accuracy (epoch)	Best Dev Accuracy (epoch)
1	*32*	*glove.6B.50d-relativized.txt*	*False*	*[128, 128, 64]*	*0.4*	*0.883 (20)*	*0.791 (18)*
2	*32*	*glove.6B.300d-relativized.txt*	*False*	*[128, 128, 64]*	*0.4*	*0.983 (20)*	***0.821 (13)***
3	*32*	*glove.6B.50d-relativized.txt*	*True*	*[128, 128, 64]*	*0.4*	*0.731 (19)*	*0.741 (18)*
4	*32*	*glove.6B.300d-relativized.txt*	*True*	*[128, 128, 64]*	*0.4*	*0.830 (20)*	*0.790 (18)*
5	*32*	*glove.6B.300d-relativized.txt*	*False*	*[128, 64]*	*0.4*	*0.984 (20)*	*0.820 (10)*
6	*32*	*glove.6B.300d-relativized.txt*	*False*	*[128, 64]*	*0.0*	*0.989 (20)*	*0.817 (13)*
7	*64*	*glove.6B.300d-relativized.txt*	*False*	*[128, 64]*	*0.4*	*0.978 (20)*	*0.813 (10)*
8	*16*	*glove.6B.300d-relativized.txt*	*False*	*[128, 64]*	*0.4*	*0.974 (20)*	*0.790 (8)*

Here are some conclusions from the above experiments:

(1) **Max Length via Accuracy**. In experiments 5, 7, and 8, the only variable was `max_length`. Based on the model's accuracy, setting `max_length` to `32` yielded the best results. The poor performance of `max_length = 16` could be attributed to the loss of sentence information due to truncation, as many sentences exceed `16` tokens, leading to a drop in semantic content. On the other hand, `max_length = 64` performed slightly worse than `32`, likely because of the influence of numerous `PAD` tokens. Since the DAN model averages embeddings, an excess of `PAD` tokens may dilute the overall meaning of the sentence.

(2) **Embed Dimension via Accuracy**. When comparing experiments 1 with 2 and 3 with 4, the only difference is the `embed_file`. It's evident that higher-dimensional embeddings yield better prediction accuracy, as the `300-dimensional` embeddings significantly outperform the `50-dimensional` ones. This improvement likely results from the higher-dimensional embeddings storing more semantic information per token, leading to more accurate token representations.

(3) **Num Layers vis Accuracy**. When comparing experiment IDs 2 and 5, the only difference lies in the number of layers in the MLP architecture of the DAN model. We observe that the final accuracy shows almost no difference. However, this does not imply that the number of layers does not impact the model's capabilities. The model's complexity and the training dataset's length mean that the overall number of parameters has a minimal effect. Unfortunately, due to computational resource limitations, I was unable to set the hidden layers to `10` to investigate the potential outcomes.

(4) **Dropout via Accuracy**. Based on experiments IDs 5 and 6, the use of dropout layers in the DAN model does not significantly affect the final results. While the training accuracy without dropout layers is slightly higher than with them, this does not guarantee better results when using dropout layers. However, dropout layers do provide some benefit in preventing overfitting. The minimal impact observed may be attributed to the total number of layers in the model; increasing both the total layers and the number of dropout layers could enhance this effect.

(5) The results of all experiments indicate significant overfitting on the training dataset. The training accuracy approaches `1.00` even after `20` epochs, while the validation accuracy remains around `0.810` to `0.820`.

Overall, achieving an accuracy of 0.77 using pre-trained GloVe embeddings is relatively straightforward.

1.3 How about randomly initialized embeddings?

To illustrate the impact of pre-trained embeddings on the overall prediction model, we conducted experiments using randomly initialized embeddings. We set the maximum length to `32`, the learning rate to `3.0e-4`, and the number of epochs to `50`, as we needed to train the embeddings from scratch.

Exp Id	freeze_embed	embed_size (input_size)	hidden_sizes	dropout_rate	Best Train Accuracy (epoch)	Best Dev Accuracy (epoch)
1	*True*	*128*	*[128, 128, 64]*	*0.4*	*0.733 (50)*	*0.617 (20)*
2	*True*	*128*	*[128, 128, 64]*	*0.0*	*0.861 (50)*	*0.619 (6)*
3	*False*	*128*	*[128, 128, 64]*	*0.4*	*0.996 (50)*	*0.769 (34)*
4	*True*	*256*	*[128, 128, 64]*	*0.4*	*0.769 (48)*	*0.650 (12)*
5	*True*	*256*	*[128, 128, 64]*	*0.0*	*0.878 (50)*	*0.654 (32)*
6	*False*	*256*	*[128, 128, 64]*	*0.4*	*0.992 (46)*	***0.790 (34)***

Here are some conclusions from the above experiments:

(1) **Dropout via Accuracy**. Comparing the results of experiment IDs 1 and 2, we observe that setting the dropout rate effectively helps the model reduce overfitting on the training dataset. However, this does not yield any improvement in validation accuracy.

(2) **Pre-trained vs. Random**. Comparing experiment ID 3 with the results in the table above, we find that training a `128-dimensional` embedding from scratch yields worse results than using the pre-trained `50-dimensional` GloVe embedding (`0.769` vs. `0.791`). This difference is due to the GloVe embedding being trained on a large corpus, allowing it to more accurately represent word meanings. Relying solely on the training dataset is insufficient for fully capturing word semantics.

(3) **Random Embed Size via Accuracy**. Comparing experiment IDs 3 and 6, we observe that even when using randomly initialized embeddings, the higher-dimensional embeddings achieve significantly better accuracy after the same number of training epochs. This improvement is due to higher dimensions allowing for more information to represent each word.

Achieving an accuracy of 0.77 is still relatively straightforward by training a random embedding solely based on the training data.

Part 2

2.1 Where is the code to train a BPE tokenizer?

The complete code can be found in the `bpe_trainer.py` file.

2.2 The steps for training a BPE tokenizer?

- Gather a large corpus of text data relevant.
- Start by creating a vocabulary with all unique characters in the text.
- Convert the text into sequences of character tokens.
- Identify and count all adjacent character pairs in the text, find the most frequent pair in the entire corpus.
- Merge the most common pair of characters into a single token. For example, if "t" and "h" are most frequent, replace "th" with a new token.
- Add the new merged token to your vocabulary.
- Continue merging pairs, updating the vocabulary with each merge, until reaching a specified vocabulary size or number of merges.

2.3 Experimental results with self-trained BPE tokenizer.

To assess the impact and effectiveness of the self-trained BPE tokenizer, we conducted several experiments, primarily varying the BPE vocabulary size and embedding dimension. For these experiments, we set the learning rate to `2.0e-4`, the maximum length to `32`, and the number of epochs to `50`.

Exp Id	bpe_vocab_size	freeze_embed	embed_size (input_size)	hidden_sizes	dropout_rate	Best Train Accuracy (epoch)	Best Dev Accuracy (epoch)
1	*20,000*	*False*	*64*	*[128, 128, 64]*	*0.4*	*0.980 (49)*	*0.759 (50)*
2	*20,000*	*False*	*128*	*[128, 128, 64]*	*0.4*	*0.985 (50)*	*0.761 (29)*
3	*20,000*	*True*	*64*	*[128, 128, 64]*	*0.4*	*0.687 (49)*	*0.619 (27)*
4	*20,000*	*True*	*128*	*[128, 128, 64]*	*0.4*	*0.766 (49)*	*0.651 (39)*
5	*25,000*	*False*	*64*	*[128, 128, 64]*	*0.4*	*0.979 (49)*	*0.756 (38)*
6	*25,000*	*False*	*128*	*[128, 128, 64]*	*0.4*	*0.987 (48)*	*0.753 (29)*
7	*15,000*	*False*	*64*	*[128, 128, 64]*	*0.4*	*0.984 (50)*	***0.778 (34)***

Exp Id	bpe_vocab_size	freeze_embed	embed_size (input_size)	hidden_sizes	dropout_rate	Best Train Accuracy (epoch)	Best Dev Accuracy (epoch)
8	*15,000*	*False*	*128*	*[128, 128, 64]*	*0.4*	*0.989 (47)*	*0.767 (44)*
9	*15,000*	*False*	*50*	*[128, 128, 64]*	*0.4*	*0.967 (47)*	*0.750 (41)*
10	*15,000*	*False*	*300*	*[128, 128, 64]*	*0.4*	*0.990 (47)*	*0.758 (45)*

Here are some conclusions from the above experiments:

(1) **Embedding Training via Accuracy**. Comparing experiment IDs (1, 3) and (2, 4), we find that training embeddings yields significantly better accuracy, which is an obvious result.

(2) **BPE Vocab Size via Accuracy**. Based on experiment IDs (1, 5, 7) and (2, 6, 8), we observe that setting the BPE vocabulary size to `15,000` yields the best validation accuracy compared to `20,000` and `25,000`. This improvement may be attributed to the lower vocabulary size, which results in each token containing more characters on average. With a constant maximum length of `32`, this means we can represent more words when the vocabulary size is `15,000`, potentially enhancing accuracy.

(3) **BPE Tokenizer vs. GloVe Embedding**. To effectively compare the performance of the trained BPE tokenizer with pre-trained GloVe embeddings, I conducted experiments 9 and 10. We found that even when setting the embedding size to match that of the two types of GloVe embeddings, the validation accuracy of our BPE tokenizer still falls short of the GloVe embeddings (`0.750` vs. `0.791` and `0.758` vs. `0.821`). However, we cannot conclude that the BPE tokenizer's method of splitting sentences is inferior to word embeddings. This discrepancy arises because we used randomly initialized embeddings for the BPE tokenizer, which significantly differ from the pre-trained embeddings, as indicated in the previous tables.

Part 3

(3a) **Question:** If we treat the above data as our training set, what is the set of probabilities P (y | the) that maximize the data likelihood? Note: this question is asking you about what the optimal probabilities are independent of any particular setting of the skip-gram vectors.

Answer:

In the sentence "the dog", (x = the, y = dog) appears once; similarly, in "the cat", (x = the, y = cat) appears once, with no other occurrences of x = the. Therefore, the probabilities that maximize the likelihood are: **P (dog|the) = 1 / (1 + 1) = 1 / 2, P (cat|the) = 1 / (1 + 1) = 1 / 2.**

(3b) **Question:** Can these probabilities be nearly achieved with a particular setting of the word vector for `the`? Give a nearly optimal vector (returns probabilities within 0.01 of the optimum) and a description of why it is optimal.

Answer: The vectors for both "dog" and "cat" are (0, 1). From question (3a), we know that P(dog|the) = P(cat|the) = 1 / 2. Therefore, we need to satisfy the following requirement:

$$\exp(v_{\text{the}} \cdot c_{\text{dog}}) \approx \exp(v_{\text{the}} \cdot c_{\text{cat}})$$

It's easy to figure out that we could set the vector of word `the` to be like (0, T) (T is a very large constant). In this case, we have:

$$v_{\text{the}} \cdot c_{\text{dog}} = v_{\text{the}} \cdot c_{\text{cat}} = (0, T) \cdot (0, 1) = T.$$

Thus, the answer is (0, T), where T is a sufficiently large constant.

(3c) **Question:** The training examples derived from these sentences.

Answer:

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

(3d) **Question:** A set of both word and context vectors that nearly optimizes the skip-gram objective. These vectors should give probabilities within 0.01 of the optimum.

Answer: Based on the training examples, it is evident that for every possible group (X, y), P(X|y) = 1 / 2. Using the same reasoning as in (3b) and following the problem's given settings, we can define all embeddings as follows:

$$Word Embeddings$$
$$v_{\text{the}} = (1, 0)$$
$$v_{\text{a}} = (1, 0)$$
$$v_{\text{dog}} = (0, 1)$$
$$v_{\text{cat}} = (0, 1)$$

$$Context Embeddings:$$
$$c_{\text{the}} = (1, 0)$$
$$c_{\text{a}} = (1, 0)$$
$$c_{\text{dog}} = (0, 1)$$
$$c_{\text{cat}} = (0, 1)$$