University of Paris-Saclay - UFR Sciences

Lab 1 report - Sentence classification

Diyun LU - Marwan MASHRA - Yihan ZHONG

November 22, 2022

1 Introduction

1.1 Continuous Bag of Words (CBOW)

Firstly, the bag of words(BOW) model assumes that for a document, its word order, grammar, syntax and other elements are ignored. It is only regarded as a collection of several words. The appearance of each word in the document is independent and does not depend on other words' appearance.

On the other hand, CBOW can be utilised to train Word Embedding. The difference between BOW and CBOW is that CBOW is always with respect to a word in the sequence. We will consider other words in the sentence according to their position relative to the target word. The training input of the CBOW model is the word embedding corresponding to the context-related words of a certain feature word, and the output is the word embedding of this target word.

Normally, CBOW is used as one of the architectures (the other one is Skip-gram Model) to create the word embeddings in Word2vec model.

1.2 Convolutional Neural Network(CNN)

As we all know, CNN is often used in computer vision (CV), but unlike the image pixels input of CV, the input of NLP is a sentence or document.

In order to apply CNN to text classification, the sentences or documents we input will be converted to a words embedding through CBOW model when inputting. Each row represents a word, the total number of rows is the length of the sentence, and the total number of columns is the dimension of the embedding. For example, a sentence containing ten words uses a 128-dimensional embedding, and we will have a matrix with an input of $10 * 128$.

In CV, filters slide over the entire image in the form of a patch, so we can use any length with any width. However, in NLP, we need to cover all dimensions of word embeddings. Thus, the convolution will be over words with a fixed size sliding window, so the shape will be $[filterSize, embeddingSize]$.

After the convolution layer, we will obtain a sequence of extracted features. Then with the maximum pooling operation, we will take out the maximum value in each column to get a one-dimensional vector. Passing it to an activation function, we can finally obtain a non-linear fixed size sentence hidden representation.

2 Implementation

2.1 Data preprocessing

2.1.1 Tokenization

Tokenization can split the given documents into tokens, these tokens represent the minimal meaningful units in the documents. Certain characters, such as punctuation, may be abandoned. Tokenization allows stopwords to be removed. Stopwords are numbers and words that are not informative, such as the prepositions and conjunctions.

2.1.2 Converting data to tensor

In order to convert data to PyTorch tensor, we firstly used a dictionary to map each unique word to an integer using their indices. We then map the sentences into a list of integers with the indices of the words. In the case of unseen words in the dev and test set, we give them the index 0. Furthermore, in order to create minibatches as a two-dimensional tensor, we need to make all sentences the same length. Therefore, we pad each sentence with 0 to the maximum length of all the sentences. We then map the "positive" label to 1 and the "negative" label to 0. However, we converted both x and y tensors into *LongTensor* form, as it is required in the PyTorch.

```python
# get all unique words
WORDS = {}
for sentence in txt_train:
    for word in sentence:
        WORDS[word] = 0
for index, word in enumerate(WORDS):
    WORDS[word] = index+1

# max length
max_len = max(map(len,txt_train))

# create functions to map
def word_to_integer(word):
    return WORDS[word] if word in WORDS else 0

def sentence_to_integer(sentence):
    return list(map(word_to_integer, sentence))
def list_to_padded_tensor(list):
    pad = max_len-len(list)
    return F.pad(th.tensor(list, dtype=th.long, requires_grad=False), (0,pad))

def process_data(txt, label):
    label_int = list(map(lambda x: 1 if x=="pos" else 0, label))
    y = th.tensor(label_int, dtype=th.float32, requires_grad=False)

    data = list(map(sentence_to_integer, txt))
    data_tensor = list(map(list_to_padded_tensor, data))
    x = th.stack(data_tensor,dim=0) # from a list of tensors to a two-dimensional tensor.

    return  x, y

x_train, y_train = process_data(txt_train, label_train)
x_dev, y_dev = process_data(txt_dev, label_dev)
x_test, y_test = process_data(txt_test, label_test)

```

2.2 Word embedding

Normally, text is unstructured data information, which cannot be directly calculated. Thus, text representation is used to convert these unstructured information into structured information.

Word embedding is one of the methods for text representation. It has the same purpose as one-hot encoding and integer encoding, but it has more advantages :

1. It's more compact than one-hot encoding because each word is mapped into a vector in a continuous space instead of binary space.

2. Some word embeddings, like Word2Vec, can encode the semantic relationships between words, so that Words with similar semantics will be mapped to similar vector spaces. However, this is not the case of word embeddings that we use here.

3. It is highly versatile and can be used in different NLP tasks.

We can create word embeddings in PyTorch as following :

```
emb_table = nn.Embedding(vocab_size, embedding_dim)
```

vocab_size is the number of unique words in the data set. *embedding_dim* is a hyperparameter that can be adjusted. Then we can retrieve embeddings of the input words using the embedding table.

```
word_emb = emb_table(inputs)
# inputs.size() = (batch_size, max_num_words)
# word_emb.size() = (batch_size, max_num_words, embedding_dim)
```

2.3 Masking

One important task before using the word embedding to compute the sentence hidden representation, we need to mask the embeddings to get rid of the embeddings of the padding. We have padded our sentences with *zero* to make all of them the same length. However, those zeros correspond to a non-zero vector in the embedding table. We don't want these paddings to effect our final results. We start by create a binary mask for each sentence, 1 if it's a word and 0 if padding. We then expand the dimension of each element of the mask to the dimension of the embedding representation, which allows us to preform element wise multiplication canceling the embedding values of paddings.

```
word_emb = emb_table(inputs)
mask = (inputs!=0).to(th.int).unsqueeze(dim=2).expand(word_emb.size())
word_emb *= mask
```

2.4 Sentence Hidden representation

In this lab, we have implemented two different methods for computing sentences hidden representations to pass to the classifier: CBOW and CNN.

2.4.1 CBOW

The general idea of CBOW (Continuous Bag-of-Words) is to compute the sentence representation (sentence embedding) by aggregating the words embeddings, for example taking the sum or the mean of all words. The sum/mean of the word embedding vectors will produce the sentence embedding. The sentence embedding thus has the same dimension as the word embedding.

```
sentence_emb = word_emb.mean(dim=1)
```

2.4.2 CNN

By applying a convolution using a sliding window over the words' embeddings, we can keep the connections between words as well as information of the position and the context. The sliding window is used to extract the local feature. In this lab, we implemented the CNN as following :

1. we use a sliding window of size 2, which gives us a Bi-gram model. Each bi-gram representation is the concatenation of two consecutive words' embedding. Therefore, we get a tensor z of dimension : (batch_size, max_num_words-window_size+1, window_size*embedding_dim)

2. we pass z through a linear projection layer

$$z = A * z + b$$

3. we apply max pooling on z to reduce from a matrix per sentence to a vector per sentence.
4. we apply an activation function (ReLU) to add non-linearity:

$$z = max(0, z)$$

5. we apply a dropout because we notice that the model overfits a lot. To describe this mathematically, let a tensor of the dimension of z and $\rho \in [0, 1]$ be the parameter that controls the dropout, and :

$$\beta \sim \mathbb{B}(1 - \rho)$$
$$z = z * \beta$$

Here is our implementation of the CNN :

```python
batch_size, max_num_words, embedding_dim = word_emb.size()
z = th.zeros((batch_size, max_num_words-self.window_size+1, self.window_size*self.embedding_dim))
for step in range(max_num_words-self.window_size+1):
    z[:,step,:] = word_emb[:,step:self.window_size+step,:].reshape(z[:,step,:].size())
z = self.proj(z)          # linear projection
z, _ = th.max(z, dim=1)   # max pooling
z = nn.ReLU()(z)          # activation function
z = self.dropout(z)       # dropout
```

2.5 Classifier

The classifier is same for both models, CBOW and CNN. It's an MLP that takes the hidden representation of the sentences as input. We have experimented with several variations:

- Shallow (linear) classifier Without hidden layers
- MLP with one or more hidden layers
- MLP with dropout rate of 0.1, 0.3 and 0.5

The output of the classifier is the logits, where negative values means the negative class and positive values means the positive class.

2.6 Loss function

For the loss function, we use BCEWithLogitsLoss, which takes as input the logits and passes them through the sigmoid before computing the BCELoss.

```python
loss_function = nn.BCEWithLogitsLoss()
logits = model(x_train[i:i+batch_size])
loss = loss_function(logits, y_train[i:i+batch_size])
```

2.7 Training

In the training part, we have applied a strategy of early stopping to avoid overfitting. The idea is to stop the training when we notice that the validation loss have been increasing for several consecutive epochs. We also use the clip normalization of gradient to avoid gradient explosion.

3 Results of our experiments

3.1 Training and evaluation loss

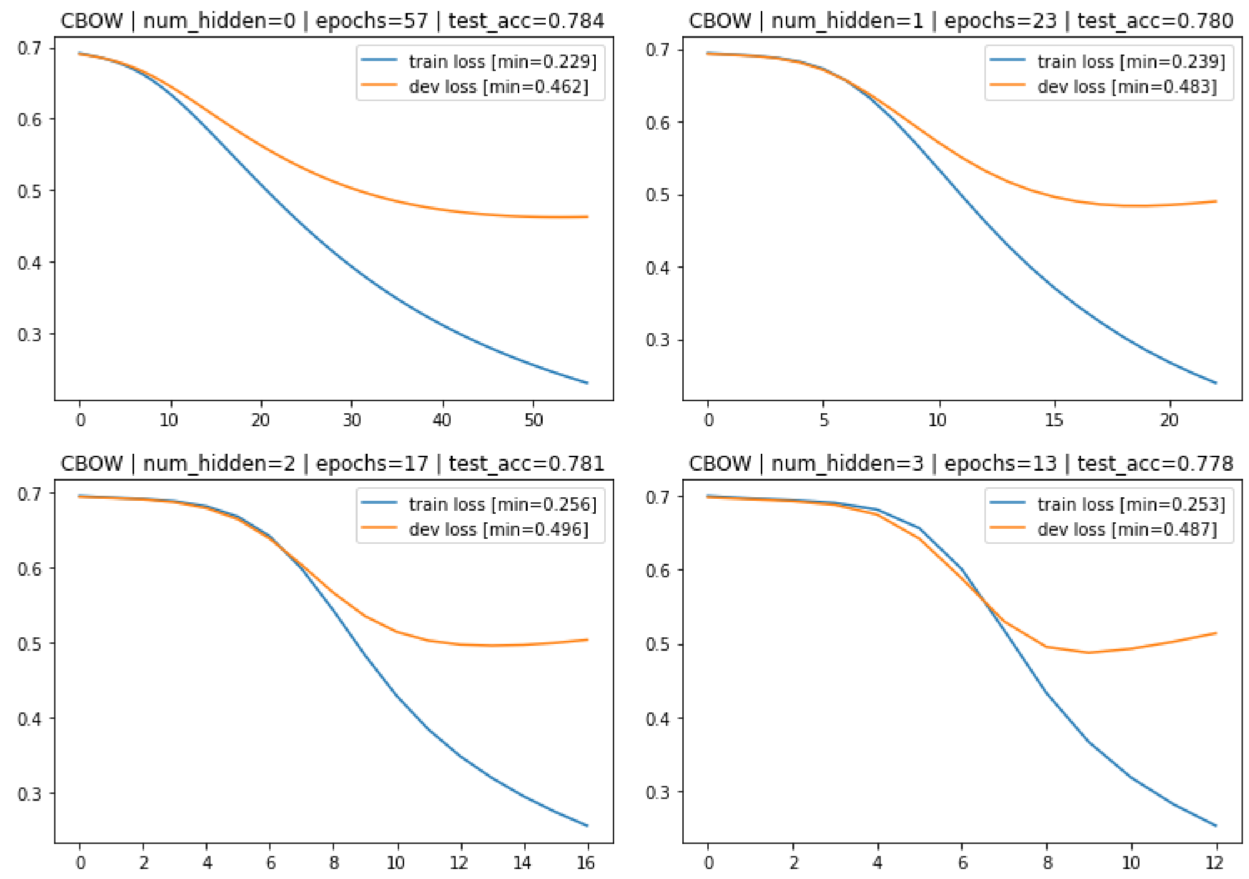

Figure 1: training of CBOW models with different number of hidden layers

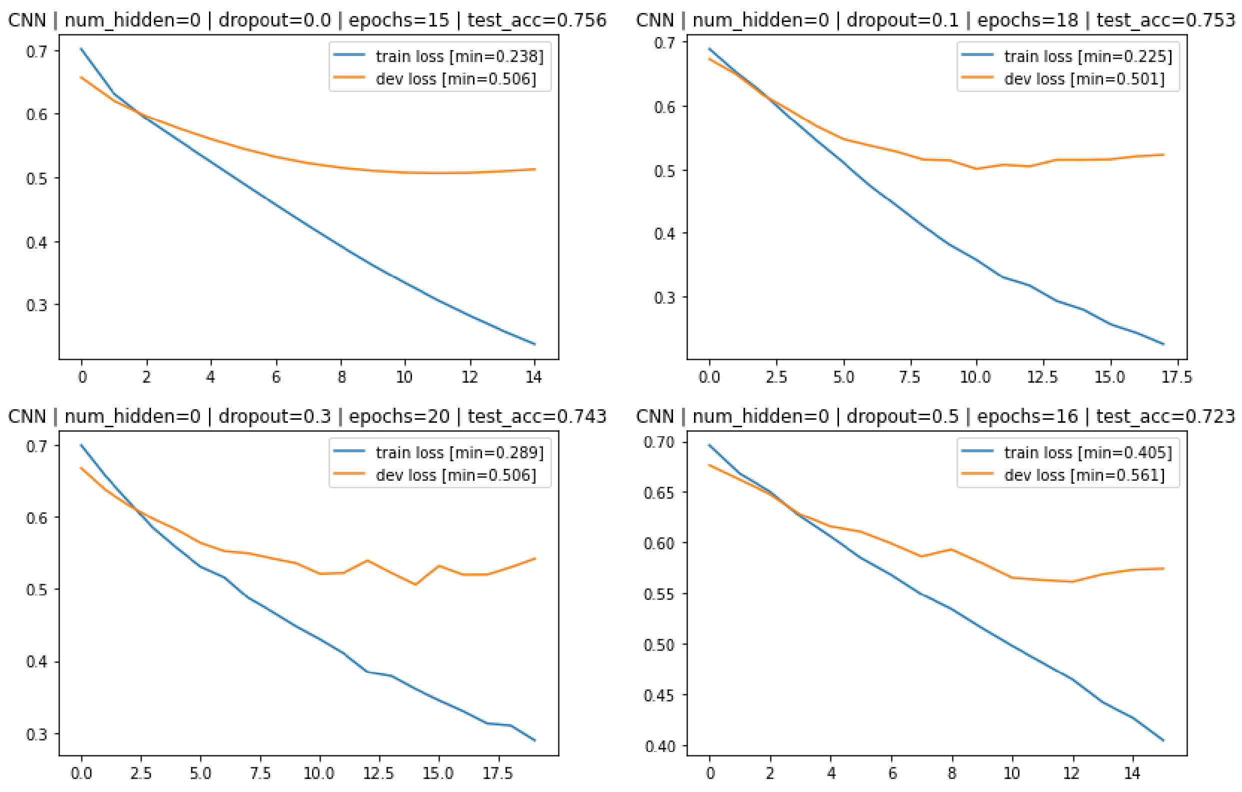

Figure 2: training of CNN models with shallow classifier and different dropouts

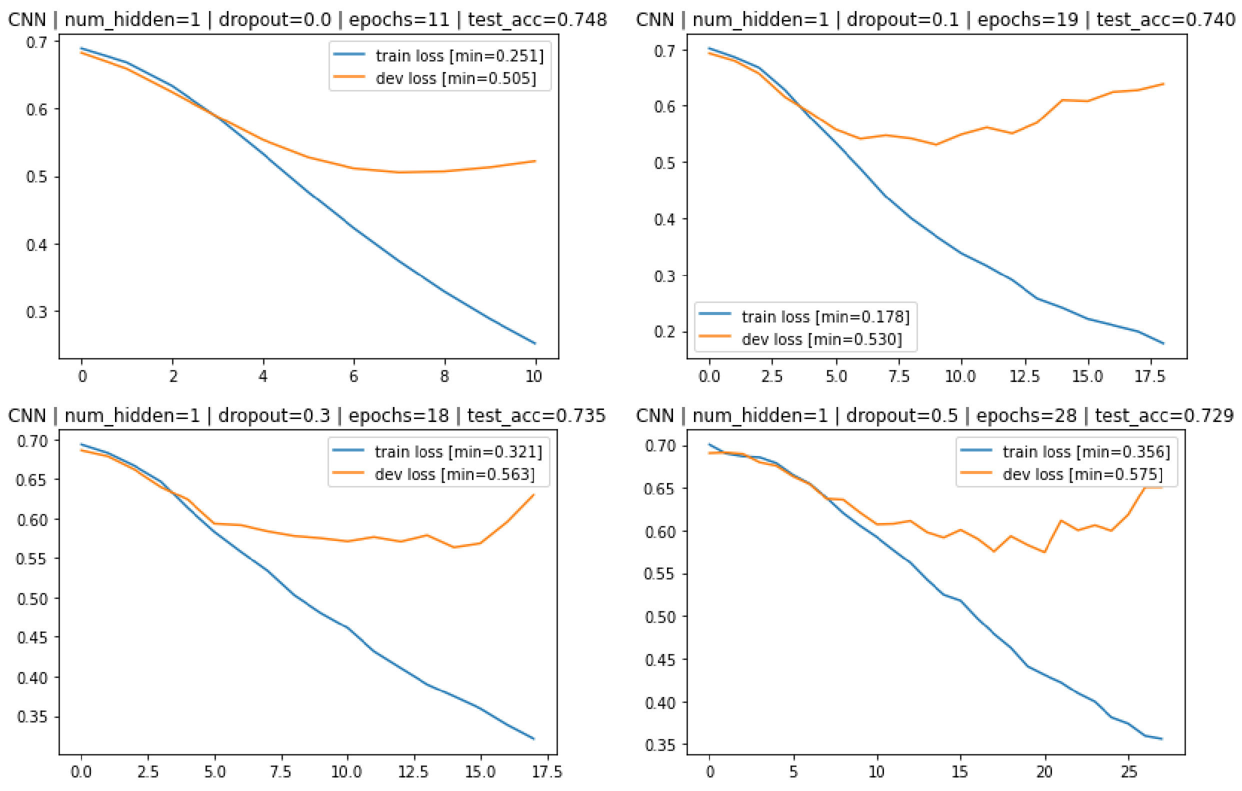

Figure 3: training of CNN models with deep classifier (1 hidden layer) and different dropouts

3.2 Comparision of two models

3.2.1 Testset accuracy

For the different test accuracy with different parameters please check the images above. In addition, we tested the total accuracy and the training time of test dataset for different models. The test ratio was 0.2, so total test sentences are $6000 * 0.2 = 300$. The test accuracy is almost similar of two models. Thus, we assume that using CNN to extract text features before the last classifier will not make much difference.

In addition, we applied dropout on our model, the accuracy slighted decreased with the dropout ratio increased.

	CBOW		CNN	
	dropout=0	dropout=0	dropout=0.1	dropout=0.3
accuracy	0.772	0.77	0.756	0.743

Table 1: The accuracy of test dataset for different models when epoch=20

3.2.2 Training time

On the other hand, we found out that the training time are very different between CBOW and CNN. After we added CNN, the training became very slow comparing with the training without it.

Also, with the increase of hidden layers and decrease of dropout ratio, the training gets slightly slower, which is reasonable as we have more layers and parameters to train. However, the differences were still very insignificant in our experiment as our dataset is small.

n hidden	CBOW		CNN	
	dropout=0	dropout=0	dropout=0.1	dropout=0.3
0	2.8	36.7	36.7	36.1
1	2.9	36.8	36.7	36.2
2	2.9	43.7	40.9	39.9
3	3.1	49.2	43.8	42.6

Table 2: The training time (seconds) for different models when epoch=20

3.3 Data Analysis: sentences that are always fail

We tested different models, found about 50 sentences that all models have failed to classify correctly. We showed the typical ones as follows. The label is the predicted ones by our model; true labels are the opposite. (All the sentences can be seen in the code script:)

Positive	Negative
a waste of some good actors	not bad !
amazingly stupid	simply superb
could have been a good movie	sad and still relevant
quantum physics , yeah right	alas , poor julie
potted beatle bio	not liz 's cleopatra

Table 3: The sentences that are always fail

There are some possible reasons:

1. The sentences are too short, so the information that the models can use are relatively less.
2. There are very strong positive or negative words in the sentences. Such as ”good”, ”bad”, they will significantly affect the models. As in the CBOW model, when we consider context words, the order of these context words is ignored. Even though the whole sentence is ”not bad” or ”a waste of some good actors”, the model would misunderstand them.
3. There are proper nouns, such as ”liz”, ”julie” and etc. As our dataset is small, the model could not get enough information about these words as the frequency of them are very low in the dataset.

All in all, CBOW and CNN are still very simple models for NLP, if we want to solve above problems, we can use pre-trained models such as Bert, Roberta and GPT etc, which are more powerful nowadays.