Natural Language Processing
Word2Vec

Silin DU

Department of Management Science and Engineering, Tsinghua University
dsl21@mails.tsinghua.edu.cn

April 29, 2022

清華大學
Tsinghua University

Contents

1 Implementation of Word2Vec

1.1 Environments and Project Structure

We use PyTorch to implement the original version of Word2Vec .We only consider CBOW here. The required environments are as follows.

- torch-1.11.0
- torchtext-0.12.0
- web

Most of our experiments are conducted on GPUs, including NIVIDIA P100 and NIVIDIA RTX 3090. The structure of our project is as follows.

- /codes: contain all the codes.
  - main.py: the entrance of our project.
  - train.py: define the class to train the model.
  - model.py: define the model.
  - dataloader.py: load the corpus for training.
  - help_function.py: define some tools to parse the configurations.
  - evaluation.py: define functions to evaluate word embeddings.
  - config.yaml: store the configurations for model trianing.
- /weight: the directory to save models and training logs.
- /data: the directory of corpus used in training and evaluation.

There are several configurations need to be predefined in config.yaml, shown in Table 1.

Table 1: Configurations

Name	Description
model_name	The model used to train word embeddings, can only be CBOW.
dataset	The corpus used for training, chosen from WikiText103/WikiText2.
data_dir	The directory of corpus.
train_batch_size	The batch size of training.
val_batch_size	The batch size of validation.
shuffle	Whether to shuffle the dataset during training and validation.
optimizer	The name of optimizer, chosen from `torch.optim`.
learning_rate	Learning rate during training.
epochs	Total epochs of training.
embed_size	The size of word embeddings.
model_dir	The directory to save models.

To run our code, use the following command

```
python mian.py --config='config.yaml'
```

1.2 Details

In this subsection, we introduce the details of our models and experimental results.

The corpus mainly used in our project is WikiText103, which contains around 1.8M lines and 100M tokens in training set. Our codes also support a smaller version corpus, i.e. WiKiText2. The flow of preparing data for training is as follows.

- We use tokenizer provided in `torchtext.data.utils` to separate the sentence into words.
- Then we use `torchtext.vocab.build_vocab_from_iterator` to get the unique words/tokens in the corpus. We omit the words/tokens whose frequency are less than 50. Finally, we get a vocabulary containing around 50k words.
- To predict the middle word, we choose 4 context words before and after the middle word. Thus, we drop the sentence with less than 9(4+4+1) words. And we set the maximum length of each sentence to be 256.

We follow the original version of Word2Vec to implement our CBOW model. The model contains an embedding layer which transforms word IDs into embeddings. The embedding layer is followed by a fully-connected layer to predict the middle words, of which the output size is the size of vocabulary. The input of the model is the average embeddings of context words.

```
class CBOW_Model(nn.Module):
    def __init__(self, vocab_size: int, embed_size: int):
        super(CBOW_Model, self).__init__()
        self.embeddings = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embed_size,
            max_norm=EMBED_MAX_NORM,
        )
        self.linear = nn.Linear(
            in_features=embed_size,
            out_features=vocab_size,
        )
        self.embed_size = embed_size

    def forward(self, inputs_):
        x = self.embeddings(inputs_)
        x = x.mean(axis=1)
        x = self.linear(x)
        return x
```

We train our model on the training set with the batch size of 128, which might occupy around 11GB of the GPU memory. The initial learning rate is 0.025 and it will multiply by 0.5 after each 5 epochs. The model is trained through 25 epochs, and the one who performs

best on the validation set will be saved. We set the maximum norm of word embeddings to be 1.

We train three models with embedding sizes of 100, 200 and 300 respectively, and evaluate the word similarity through WordSim353 and Sim999 datasets. We try to use the evaluation tools in `web`, but this package can not be used directly since it was updated 4 years ago. Thus, we write some tools for evaluations, borrowing some codes from `web.evaluation`. Details can be found in evaluation.py .

The experimental results are shown in Table 2.

Table 2: Experimental results of CBOW

Model	WordSim353	Sim999
CBOW(dim=100)	0.497	0.273
CBOW(dim=200)	0.521	0.289
CBOW(dim=300)	0.543	0.302
Glove[1](dim=300)	0.522	0.371
Glove(dim=100)	0.451	0.298

Meanwhile, we also evaluate word embeddings through analogy, using the Google analogy test set, which contains 19544 question pairs in 14 types of relationships. We borrow some codes from `web.analogy` to write an evaluation function. Details can be found in the evaluation.py . The evaluation results are shown in Table 3.

Table 3: Analogy Test of CBOW

Model	Accuracy	Semantic	Syntactic
CBOW(dim=100)	16.37	7.26	23.94
CBOW(dim=200)	20.58	7.50	31.46
CBOW(dim=300)	22.91	12.58	31.49
CBOW[2](dim=50)	50.66	44.43	55.83

We also visualize some word embeddings (dim=300), shown in Figure 1.

1.3 Discussion and Summary

From Table 2, we observe that

- With the increase of dimensions of word embeddings, the evaluation results become better on two datasets.

[1]Evaluation results of Glove are from: https://github.com/kudkudak/word-embeddings-benchmarks/wiki

[2]Evaluation results of CBOW are from: https://aclweb.org/aclwiki/Google_analogy_test_set_(State_of_the_art)

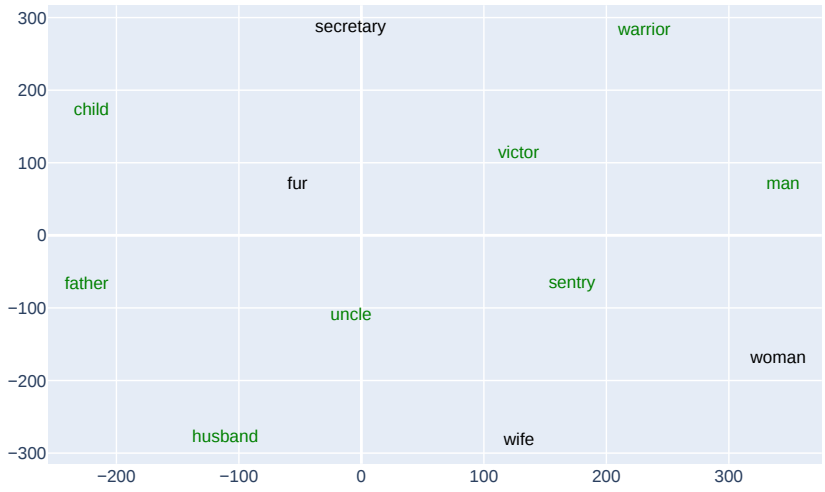

Figure 1: Visualization of Some Word Embeddings

- Compared with Glove(dim=300), the embeddings from our CBOW(dim=300) model get a comparable score on WordSim353. On Sim999, our model are comparable with Glove(dim=100).
- With the increase of dimensions, the superiority of larger embedding sizes decreases.

From Table 3, we observe that

- With the increase of dimensions of word embeddings, the evaluation results become better in the analogy test.
- Our pretrained embeddings cannot get a comparable score in the analogy test. The main reason is that **there are 4589 missing words, which is out of the vocabulary of our corpus**.

2 Improvement of Word2Vec

2.1 Environments and Project Structure

In this section, we use WordNet from `nltk` to improve the pretrained word embeddings from Section 1.

The structure of this project is roughly the same. But it contains some additional codes in the code directory.

- model_finetune.py: define the model to fine-tune the embeddings.
- train_finetune.py: define the function to train the fine-tune model.

- data_finetune.py: prepare data for training the fine-tune model.
- main_finetune.py: the entrance to fine-tune the word embeddings.

2.2 Details

We try to use the knowledge in WordNet to improve word embeddings. WordNet can provide synonyms for nouns, adjectives and adverbs. Thus, we implement a simple idea that **the embeddings of two synonyms should be similar**.

Specifically we add a small embeddings at the end of pretrained word embeddings from Section 1. The dimension of small embeddings is 5% of the original dimension. To train the new embeddings, we use the data prepared through the following process.

- Query all the synonyms of words in our vocabulary.
- Filter out the synonyms which are out of our vocabulary.
- Organize word-synonym pairs like

$$\{(\text{word1}, \text{a synonym of word1}, 1), (\text{word1}, \text{negative word}, 0)\}$$

  where negative words are randomly sampled from our vocabulary for each word-synonym pairs.
- For each word who has more than 2 synonyms in our vocabulary, we will randomly sample one synonym of this word and put this pair into the validation set.

Now we can perform our fine-tune task which is to predict whether two words are a word-synonym pair or not.

The model takes two words as input and predicts a binary label. We first define two embedding layers. The first one loads its parameters from pretrained word embeddings and is fixed during training. The second one represents the small embeddings and is initialized as 0 for all words. Two emebddings of a word will be concatenated together for model training. We then concatenate the embeddings of the word-pair, and feed them into a fully-connected layer to get the output.

```
class Finetune_Model(nn.Module):

    def __init__(self,
                 vocab_size: int,
                 embed_size_1: int,
                 embed_size_2: int,
                 dropout: float,
                 max_norm: float,
                 embedding):

        super(Finetune_Model, self).__init__()
        self.embeddings_1 = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embed_size_1,
```

```
            max_norm=1
        )
        self.embeddings_1.weight = nn.Parameter(embedding)
        self.embeddings_1.weight.requires_grad = False

        self.embeddings_2 = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embed_size_2,
            max_norm=max_norm
        )
        nn.init.constant_(self.embeddings_2.weight, 0)
        self.classifier = nn.Sequential(nn.Dropout(dropout),
            nn.Linear(2*(embed_size_1+embed_size_2), 1), nn.Sigmoid())

    def forward(self, word1, word2):
        x1 = torch.cat((self.embeddings_1(word1), self.embeddings_2(word1)), dim=1)
        x2 = torch.cat((self.embeddings_1(word2), self.embeddings_2(word2)), dim=1)

        x = torch.cat((x1,x2),dim=1)
        x = self.classifier(x)

        return x.squeeze()
```

To train this model, we use a batch size of 256, which might occupy around 3GB of the GPU memory. The initial learning rate is 0.025 and it will multiply by 0.5 after every 5 epochs. The model is trained through 20 epochs, and the one who performs best on the validation set will be saved.

The experimental results of improved CBOW on several datasets are shown in Table 4.

Table 4: Experimental results of Improved CBOW

Model	WordSim353	Sim999
CBOW(dim=100)	0.497	0.273
CBOW(dim=200)	0.521	0.289
CBOW(dim=300)	0.543	0.302
CBOW(dim=105)	0.513	0.292
CBOW(dim=210)	0.532	0.301
CBOW(dim=315)	0.559	0.314

2.3 Discussion and Summary

We fixed the pretrained word embeddings during model training for the following reasons.

- Word2Vec is trained by the co-occurance of words. While the synonymity is different forms of knowledge.
- The dataset used in this section is too small for tuning the entire embeddings.

Also for these reasons, we set the maximum norm of the small embeddings to be 0.2, which will limit the ranges of parameters. Note that for those words who do not have synonyms, we set their small embeddings as zeros.

We can observe some improvements from our fine-tuning in Table 4, but is relatively small ($2 \sim 3\%$). We do not observe obvious improvements in the analogy test, since this modification does not solve the missing words issue mentioned in Section 1.

We here only present a way to improve Word2Vec using external knowledge, but we don't solve the ambiguity problem and either don't change the context-free manner of Word2Vec. In the future, we might try to jonintly learn word embeddings and sense embeddings or conduct experiments on a larger corpus.