CS310 Natural Language Processing

Assignment 2. Word2vec Implementation

Total points: 50

Train a word2vec model using the **skip-gram** architecture and **negative sampling**.

You should roughly follow the structure of the notebook. Add additional cells if you feel needed.

You can (and you should) re-use the code from *Lab 4 (part 2): Data preparation for implementing word2vec*.

Make sure your code is readable and well-structured.

0. Import Necessary Libraries

In [1]:

```
from typing import List
from utils import CorpusReader
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import TruncatedSVD
from tqdm import tqdm
import numpy as np
from sklearn.decomposition import TruncatedSVD
import matplotlib.pyplot as plt
```

In [2]:

```
torch.cuda.is_available()
```

Out[2]:

```
True
```

1. Data Processing

The corpus data is in `lunyu_20chapters.txt`. Use the `CorpusReader` class in `utils.py` to help you.

In [3]:

```
# Read raw data in lunyu_20chapters.txt

### YOUR CODE HERE ###
corpus = CorpusReader(inputFileName="lunyu_20chapters.txt", min_count=1)
### END YOUR CODE ###
```

```
Total vocabulary: 1352
```

In [4]:

```
# Re-use the code from lab with necessary modifications

def generate_data(words: List[str], window_size: int, k: int, corpus: CorpusReader):
    """ Generate the training data for word2vec skip-gram model
    Args:
        text: the input text
        window_size: the size of the context window
        k: the number of negative samples
        corpus: the corpus object, providing utilities such as word2id, getNegatives, etc.
    """
    ### YOUR CODE HERE ###
    word_ids = []
    for word in words:
        word_ids.append(corpus.word2id[word])

    for i in range(len(word_ids)):
        center_word = word_ids[i]
        context_words = word_ids[max(0, i - window_size):i] + word_ids[i + 1:i + window_size + 1]
        for context_word in context_words:
            yield center_word, context_word, corpus.getNegatives(center_word, k)
    ### END YOUR CODE ###

def batchify(data: List, batch_size: int):
    """ Group a stream into batches and yield them as torch tensors.
    Args:
        data: a list of tuples
        batch_size: the batch size
    Yields:
        a tuple of three torch tensors: center, outside, negative
    """
    assert batch_size < len(data) # data should be long enough
    for i in range(0, len(data), batch_size):
        batch = data[i:i + batch_size]
        if i > len(data) - batch_size: # if the last batch is smaller than batch_size, pad it with the first few da
            batch = batch + data[:i + batch_size - len(data)]

        ### YOUR CODE HERE ###
        centers = []
        outsides = []
        negatives = []
        for center, outside, negative in batch:
            centers.append(center)
            outsides.append(outside)
            negatives.append(negative)
        centers = torch.tensor(centers)
        outsides = torch.tensor(outsides)
        negatives = torch.tensor(negatives)

        yield centers, outsides, negatives
        ### END YOUR CODE ###
```

2. Define the Model

In [ ]:

```
class SkipGram(nn.Module):
    def __init__(self, vocab_size, emb_size):
        super(SkipGram, self).__init__()
        self.vocab_size = vocab_size
        self.emb_size = emb_size
        self.emb_v = nn.Embedding(vocab_size, emb_size, sparse=True)
        self.emb_u = nn.Embedding(vocab_size, emb_size, sparse=True)

        initrange = 1.0 / self.emb_size # some experience passed down from generation to generation
        nn.init.uniform_(self.emb_v.weight.data, -initrange, initrange) # same outcome as self.emb_v.weight.data.un
        nn.init.constant_(self.emb_u.weight.data, 0) # same outcome as self.emb_u.weight.data.zero_()

    def forward(self, center, outside, negative):
        """
        Args:
            center: the center word indices (B, )
            outside: the outside word indices (B, )
            negative: the negative word indices (B, k)
        """
        v_c = self.emb_v(center)
        u_o = self.emb_u(outside)
        u_n = self.emb_u(negative)

        ### YOUR CODE HERE ###
        loss = None
        pos_score = torch.sum(v_c * u_o, dim=1)
        pos_score_clamped = torch.clamp(pos_score, min=-100, max=100)
        pos_loss = -F.logsigmoid(pos_score_clamped)

        neg_score = torch.sum(v_c.unsqueeze(1) * u_n, dim=2)
        neg_score_clamped = torch.clamp(neg_score, min=-100, max=100)
        neg_loss = -F.logsigmoid(-neg_score_clamped)
        neg_loss = torch.sum(neg_loss, dim=1)

        loss = pos_loss + neg_loss
        # Hint: torch.clamp the input to F.logsigmoid to avoid numerical underflow/overflow
        ### END YOUR CODE ###

        return loss

    def save_embedding(self, id2word, file_name):
        embedding = self.emb_v.weight.cpu().data.numpy()
        with open(file_name, 'w') as f:
            f.write('%d %d\n' % (len(id2word), self.emb_size))
            for wid, w in id2word.items():
                e = ' '.join(map(lambda x: str(x), embedding[wid]))
                f.write('%s %s\n' % (w, e))
```

In [6]:

```
# corpus = CorpusReader('lunyu_20chapters.txt', 1)
vocab_size = corpus.vocab_size
```

3. Train and Evaluate

In [ ]:

```python
def train(model, dataloader, optimizer, scheduler, epochs):
    # Write your own code for this train function
    # You don't need exactly the same arguments

    ### YOUR CODE HERE ###

    model.train()
    device = "cpu" if not torch.cuda.is_available() else "cuda"
    model.to(device)
    print(f"device: {device}")
    epoch_loss = np.zeros(epochs)
    total_steps = 0
    for epoch in tqdm(range(epochs)):
        epoch_loss[epoch] = 0
        epoch_loss_temp = []
        for i, (center_word, context_word, negative_words) in enumerate(dataloader):
            center_word = center_word.to(device)
            context_word = context_word.to(device)
            negative_words = negative_words.to(device)

            loss = model(center_word, context_word, negative_words).mean()
            # print(f"loss ,{loss}")
            epoch_loss_temp.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            scheduler.step()
            total_steps += 1

            # if (i+1) % 5 == 0:
            #     print(f"Epoch {epoch+1}, Step {i+1}, Loss: {loss.item():.4f}")
        epoch_loss[epoch] = np.mean(epoch_loss_temp)
        if (epoch+1) % 5 == 0:
            print(f"[Epoch {epoch+1}, Loss: {epoch_loss[epoch]:.4f}]")
    return epoch_loss

    ### END YOUR CODE ###
```

In [ ]:

```python
from torch.utils.data import DataLoader
# Suggested hyperparameters
initial_lr = 0.025
batch_size = 16
emb_size = 50
window_size = 5
k = 10 # the number of negative samples, change with your own choice for better embedding performance
min_count = 1 # because our data is small. If min_count > 1, you should filter out those unknown words from the dat
# optimizer = torch.optim.Adam(model.parameters(), lr=initial_lr) # or torch.optim.SparseAdam()
# scheduler = torch.optim.lr_scheduler.CosineAnnealingLR() # or torch.optim.lr_scheduler.StepLR()


with open("lunyu_20chapters.txt", "r", encoding="utf-8") as f:
    text = f.read()
text=text.replace("\n", "")
text=text.replace(" ", "")
words = list(text)
### Hints: ###
# - If you have cuda-supported GPUs, you can run the training faster by
#   `device = torch.device("cuda" if self.use_cuda else "cpu")`
#   `model.cuda()`
#   You also need to move all tensor data to the same device
# - If you find Inf or NaN in the loss, you can try to clip the gradient usning `torch.nn.utils.clip_grad_norm_`
# - Remember to save the embeddings when training is done
```

In [9]:

```python
model1 = SkipGram(vocab_size, emb_size=50)
data= list(generate_data(words, window_size=1, k=2, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
optimizer = torch.optim.SparseAdam(model1.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb50_k2_win1= train(model1, train_data, optimizer,scheduler,epochs=20)
```

```
/tmp/ipykernel_1784905/2220791497.py:47: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely s
low. Please consider converting the list to a single numpy.ndarray with numpy.array() before converting to a tensor.
(Triggered internally at ../torch/csrc/utils/tensor_new.cpp:278.)
  negatives = torch.tensor(negatives)
```

```
device: cuda
 25%|██▌       | 5/20 [00:42<02:05,  8.37s/it]
[Epoch 5, Loss: 1.0547]
 50%|█████     | 10/20 [01:23<01:23,  8.40s/it]
[Epoch 10, Loss: 0.9555]
 75%|███████▌  | 15/20 [02:06<00:42,  8.54s/it]
[Epoch 15, Loss: 0.9362]
100%|██████████| 20/20 [02:49<00:00,  8.46s/it]
[Epoch 20, Loss: 0.9284]
```

In [10]:

```python
output_file = 'emb50_k2_win1_embeddings.txt'
weights = model1.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [11]:

```python
model2 = SkipGram(vocab_size, emb_size=50)
data= list(generate_data(words, window_size=1, k=5, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
optimizer = torch.optim.SparseAdam(model2.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb50_k5_win1= train(model2, train_data, optimizer,scheduler,epochs=20)
```

```
device: cuda
 25%|██▌       | 5/20 [00:42<02:07,  8.50s/it]
[Epoch 5, Loss: 1.7580]
 50%|█████     | 10/20 [01:25<01:25,  8.53s/it]
[Epoch 10, Loss: 1.5612]
 75%|███████▌  | 15/20 [02:08<00:43,  8.63s/it]
[Epoch 15, Loss: 1.5224]
100%|██████████| 20/20 [02:51<00:00,  8.56s/it]
[Epoch 20, Loss: 1.5107]
```

In [12]:

```python
output_file = 'emb50_k5_win1_embeddings.txt'
weights = model2.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [13]:

```python
model3 = SkipGram(vocab_size, emb_size=50)
data= list(generate_data(words, window_size=3, k=2, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
optimizer = torch.optim.SparseAdam(model3.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb50_k2_win3= train(model3, train_data, optimizer,scheduler,epochs=20)
```

```
device: cuda
 25%|██▌       | 5/20 [02:06<06:13, 24.89s/it]
[Epoch 5, Loss: 1.4750]
 50%|█████     | 10/20 [04:13<04:13, 25.36s/it]
[Epoch 10, Loss: 1.3740]
 75%|███████▌  | 15/20 [06:21<02:07, 25.57s/it]
[Epoch 15, Loss: 1.3446]
100%|██████████| 20/20 [08:28<00:00, 25.42s/it]
[Epoch 20, Loss: 1.3313]
```

In [14]:

```python
output_file = 'emb50_k2_win3_embeddings.txt'
weights = model3.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [15]:

```python
model4 = SkipGram(vocab_size, emb_size=50)
data= list(generate_data(words, window_size=3, k=5, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
```

```
optimizer = torch.optim.SparseAdam(model4.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb50_k5_win3= train(model4, train_data, optimizer,scheduler,epochs=20)
```

```
device: cuda
 25%|██▌       | 5/20 [02:09<06:26, 25.76s/it]
[Epoch 5, Loss: 2.2641]
 50%|█████     | 10/20 [04:17<04:15, 25.60s/it]
[Epoch 10, Loss: 2.1160]
 75%|███████▌  | 15/20 [06:24<02:07, 25.52s/it]
[Epoch 15, Loss: 2.0712]
100%|██████████| 20/20 [08:32<00:00, 25.63s/it]
[Epoch 20, Loss: 2.0505]
```

In [16]:

```
output_file = 'emb50_k5_win3_embeddings.txt'
weights = model4.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [ ]:

```
emb_size=100
model5 = SkipGram(vocab_size, emb_size=100)
data= list(generate_data(words, window_size=1, k=2, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
optimizer = torch.optim.SparseAdam(model5.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb100_k2_win1= train(model5, train_data, optimizer,scheduler,epochs=20)
```

```
device: cuda
 25%|██▌       | 5/20 [00:42<02:06,  8.44s/it]
[Epoch 5, Loss: 1.0396]
 50%|█████     | 10/20 [01:24<01:23,  8.33s/it]
[Epoch 10, Loss: 0.9731]
 75%|███████▌  | 15/20 [02:05<00:41,  8.34s/it]
[Epoch 15, Loss: 0.9578]
100%|██████████| 20/20 [02:47<00:00,  8.37s/it]
[Epoch 20, Loss: 0.9541]
```

In [18]:

```
output_file = 'emb100_k2_win1_embeddings.txt'
weights = model5.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [19]:

```
model6 = SkipGram(vocab_size, emb_size=100)
data= list(generate_data(words, window_size=1, k=5, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
optimizer = torch.optim.SparseAdam(model6.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb100_k5_win1= train(model6, train_data, optimizer,scheduler,epochs=20)
```

```
device: cuda
 25%|██▌       | 5/20 [00:42<02:08,  8.57s/it]
[Epoch 5, Loss: 1.7091]
 50%|█████     | 10/20 [01:26<01:27,  8.70s/it]
[Epoch 10, Loss: 1.5734]
 75%|███████▌  | 15/20 [02:09<00:43,  8.66s/it]
[Epoch 15, Loss: 1.5488]
100%|██████████| 20/20 [02:52<00:00,  8.61s/it]
[Epoch 20, Loss: 1.5425]
```

In [20]:

```
output_file = 'emb100_k5_win1_embeddings.txt'
weights = model6.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [21]:

```
model7 = SkipGram(vocab_size, emb_size=100)
data= list(generate_data(words, window_size=3, k=2, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
optimizer = torch.optim.SparseAdam(model7.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb100_k2_win3= train(model7, train_data, optimizer,scheduler,epochs=20)
```

```
device: cuda
 25%|██▌       | 5/20 [02:15<07:03, 28.23s/it]
[Epoch 5, Loss: 1.4177]
 50%|█████     | 10/20 [04:24<04:19, 25.97s/it]
[Epoch 10, Loss: 1.3435]
 75%|███████▌  | 15/20 [06:32<02:08, 25.78s/it]
[Epoch 15, Loss: 1.3276]
100%|██████████| 20/20 [08:38<00:00, 25.91s/it]
[Epoch 20, Loss: 1.3213]
```

In [22]:

```
output_file = 'emb100_k2_win3_embeddings.txt'
weights = model7.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [23]:

```
model8 = SkipGram(vocab_size, emb_size=100)
data= list(generate_data(words, window_size=3, k=5, corpus=corpus))
train_data = list(batchify(data, batch_size=batch_size))
dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
optimizer = torch.optim.SparseAdam(model8.parameters(), lr=initial_lr)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
# Train the model
loss_emb100_k5_win3= train(model8, train_data, optimizer,scheduler,epochs=20)
```

```
device: cuda
 25%|██▌       | 5/20 [02:09<06:28, 25.93s/it]
[Epoch 5, Loss: 2.1932]
 50%|█████     | 10/20 [04:16<04:14, 25.50s/it]
[Epoch 10, Loss: 2.0623]
 75%|███████▌  | 15/20 [06:23<02:07, 25.40s/it]
[Epoch 15, Loss: 2.0353]
100%|██████████| 20/20 [08:30<00:00, 25.52s/it]
[Epoch 20, Loss: 2.0245]
```

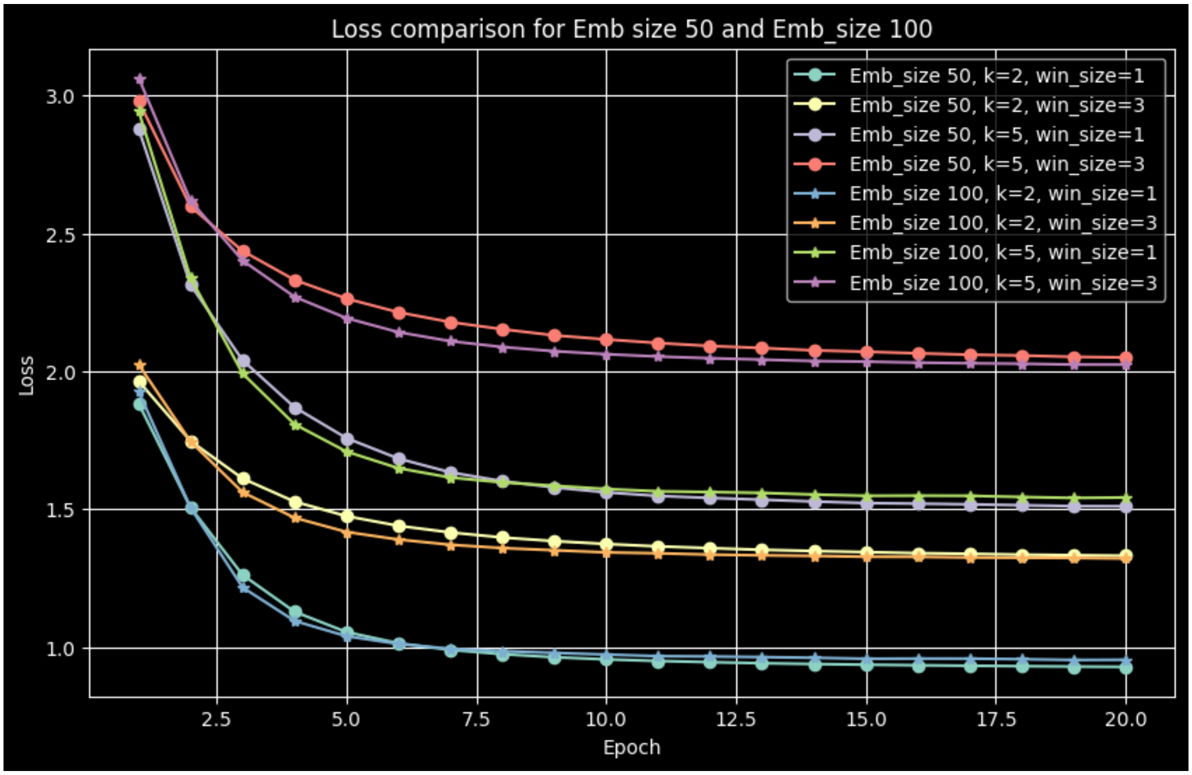
In [24]:

```
output_file = 'emb100_k5_win3_embeddings.txt'
weights = model8.emb_v.weight.cpu().data.numpy()
with open(output_file, "w") as f:
    f.write(f"{vocab_size} {emb_size}\n")  # First line: vocab size and vector dimension
    for idx, vector in enumerate(weights):
        vector_str = " ".join(map(str, vector))
        f.write(f"{corpus.id2word[idx]} {vector_str}\n")
```

In [25]:

```
import matplotlib.pyplot as plt

# 创建两个列表用于存储x和y轴数据
epochs = list(range(1, 21))  # 1到20的epoch数


# 绘制折线图
plt.figure(figsize=(10, 6))
plt.plot(epochs, loss_emb50_k2_win1, marker='o', label='Emb_size 50, k=2, win_size=1')
plt.plot(epochs, loss_emb50_k2_win3, marker='o', label='Emb_size 50, k=2, win_size=3')
plt.plot(epochs, loss_emb50_k5_win1, marker='o', label='Emb_size 50, k=5, win_size=1')
plt.plot(epochs, loss_emb50_k5_win3, marker='o', label='Emb_size 50, k=5, win_size=3')

plt.plot(epochs, loss_emb100_k2_win1, marker='*', label='Emb_size 100, k=2, win_size=1')
plt.plot(epochs, loss_emb100_k2_win3, marker='*', label='Emb_size 100, k=2, win_size=3')
plt.plot(epochs, loss_emb100_k5_win1, marker='*', label='Emb_size 100, k=5, win_size=1')
plt.plot(epochs, loss_emb100_k5_win3, marker='*', label='Emb_size 100, k=5, win_size=3')

# 设置图表标题和轴标签
plt.title('Loss comparison for Emb size 50 and Emb_size 100')
```

```python
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:

```python

```

4. Save the Embeddings

Save the embeddings into a `gensim` compatible format.

In [ ]:

```python
# The embeddings has saved above.
```

5. Plot and Compare Embeddings

Use `sklearn.decomposition.TruncatedSVD` to reduce the dimensionality of the obtained embeddings to 2 and plot the selected words in 2D space.

Hint:

- Obtain the embeddings into a numpy array by `model.emb_v.cpu().data.numpy()`
- The word2id dictionary is in `model.word2id`
- If you are trying to load from a saved embedding file, you can use the APIs from `gensim`.
    - For exmaple, `model = gensim.models.KeyedVectors.load_word2vec_format('path/to/file')`
    - Check out the documentation for more details: https://radimrehurek.com/gensim/models/keyedvectors.html

In [ ]:

```python
# Because the gensim uses numpy version is conflict with the numpy version my torch needed,
# so I use another notebook plot.py to plot the embeddings.
```