Lab session 3: Word embedding

This lab covers word embedding as seen in the theory lectures (DL lecture 5).

General instructions:

- Complete the code where needed
- Provide answers to questions only in the cell where indicated
- **Do not alter the evaluation cells** (`## evaluation`) in any way as they are needed for the partly automated evaluation process

Embedding; the Steroids for NLP!

Pre-trained embedding have brought NLP a long way. Most of the recent methods include word embeddings into their pipeline to obtain state-of-the-art performance. `Word2vec` is among the most famous methods to efficiently create word embeddings and has been around since 2013. Word2Vec has two different model architectures, namely `Skip-gram` and `CBOW`. `Skip-gram` was explained in more detail in the theory lecture, and today we will play with `CBOW`. We will train our own little embeddings, and use them to visualize text corpora. In the last part, we will download and utilize other pretrained embeddings to build a Part-of-Speech tagging (PoS) model.


```
# import necessary packages
import random
import math
import numpy as np

from random import shuffle
from collections import Counter

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda
```

```
# for reproducibility

SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

1. Data preparation

As always, let's first prepare the data. We shall use the `text8` dataset, which offers cleaned English Wikipedia text. The data is clean UTF-8 and all characters are lower-cased with valid encodings.

```
!wget "http://mattmahoney.net/dc/text8.zip" -O text8.zip
!unzip -o text8.zip
!rm text8.zip
!head -c 1b text8 # print first bytes of text8 data
```

```
--2020-04-29 18:26:31--  http://mattmahoney.net/dc/text8.zip
Resolving mattmahoney.net (mattmahoney.net)... 67.195.197.75
Connecting to mattmahoney.net (mattmahoney.net)|67.195.197.75|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 31344016 (30M) [application/zip]
Saving to: ‘text8.zip’

text8.zip           100%[===================>]  29.89M   280KB/s    in 1m 54s

2020-04-29 18:28:25 (270 KB/s) - ‘text8.zip’ saved [31344016/31344016]

Archive:  text8.zip
  inflating: text8
 anarchism originated as a term of abuse first used against early working class radicals including the diggers of the english re
```

```
# read text8
with open('text8', 'r') as input_file:
    text = input_file.read()
```

Tokenization

We first chop our text into pieces using NLTK's `WordPuncTokenizer`:

```
from nltk.tokenize import WordPunctTokenizer

tknzr = WordPunctTokenizer()
tokenized_text = tknzr.tokenize(text)

print(tokenized_text[0:20])
```

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early', 'working', 'class', 'radicals
```

Build dictionary

In this step, we convert each word to a unique id. We can define our vocabulary trimming rules, which specify whether certain words should remain in the vocabulary, be trimmed away, or handled differently. In following, we limit our vocabulary size to `vocab_size` words and replace the remaining tokens with `UNK`:

```
def get_data(text, vocab_size = None):

    word_counts = Counter(text)

    sorted_token = sorted(word_counts, key=word_counts.get, reverse=True) # sort by frequency

    if vocab_size: # keep most frequent words
        sorted_token = sorted_token[:vocab_size-1]

    sorted_token.insert(0, 'UNK') # reserve 0 for UNK

    id_to_token = {k: w for k, w in enumerate(sorted_token)}
    token_to_id = {w: k for k, w in id_to_token.items()}

    # tokenize words in vocab and replace rest with UNK
    tokenized_ids = [token_to_id[w] if w in token_to_id else 0 for w in text]

    return tokenized_ids, id_to_token, token_to_id
```

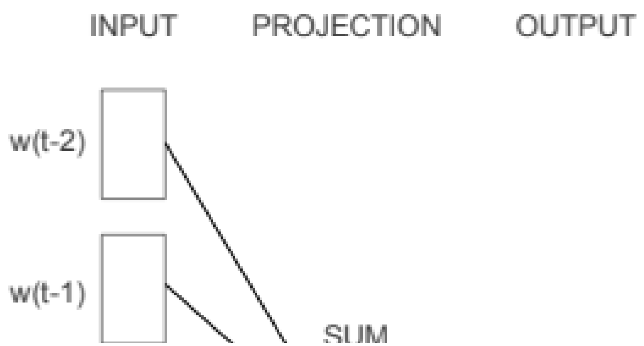
```
tokenized_ids, id_to_token, token_to_id = get_data(tokenized_text)
print('-' * 50)
print('Number of uniqe tokens: {}'.format(len(id_to_token)))
print('-' * 50)
print("tokenized text: {}".format(tokenized_text[0:20]))
print('-' * 50)
print("tokenized ids: {}".format(tokenized_ids[0:20]))
```

```
--------------------------------------------------
Number of uniqe tokens: 253855
--------------------------------------------------
tokenized text: ['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early', 'working', '
--------------------------------------------------
tokenized ids: [5234, 3081, 12, 6, 195, 2, 3134, 46, 59, 156, 128, 742, 477, 10572, 134, 1, 27350, 2, 1, 103]
```

Generate samples

The `CBOW` model architecture tries to predict the current target word (the center word) based on the source context words (surrounding words). The training data thus comprises pairs of `(context_window, target_word)`, for which the model should predict the `target_word` based on the `context_window` words.

Considering a simple sentence, **the quick brown fox jumps over the lazy dog**, with a `context_window` of size 1, we have examples like **([quick, fox], brown)**, **([the, brown], quick)**, **([the, dog], lazy)** and so on.



Now let us convert our tokenized text from `tokenized_ids` into `(context_window, target_word)` pairs.

You should loop over the `tokenized_ids` and build a **generator** which yields a target word of length 1 and surrounding context of length (2 $\times$ `window_size`) where we take `window_size` words before and after the target word in our corpus. Remember to pad context words with zeroes to a fixed length if needed.

```
def generate_sample(tknzd_ids, window_size = 5):
    for index, target in enumerate(tknzd_ids):
        ############### for student ################
        n_before = window_size
        n_after = window_size

        if index < window_size:
            n_before = index

        if index > len(tknzd_ids) - window_size - 1:
            n_after = len(tknzd_ids) - index - 1

        context_window = [0]*(window_size-n_before) + tknzd_ids[index-n_before:index] + tknzd_ids[index+1:index+n_after+1] + [0]*(wi
        ############################################
        yield context_window, target
```

```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY

dummy_gen = generate_sample([11, 12, 13, 14, 15], 2)
```

```
dummy_gen = generate_sample([11, 12, 13, 14, 15], 2)
dummy_example = list(dummy_gen)
print(dummy_example)

assert isinstance(dummy_example[0], tuple), "Is it a pair?"
assert len(dummy_example[0][0]) == 4, "Context length should be 2 * window_size"
assert dummy_example[0][1] == 11, "Did you return the correct target word?"
assert dummy_example[0][0][0] == dummy_example[0][0][1]==0, "Did you add 0 pads where needed?"
assert len(dummy_example[0]) == len(dummy_example[-1]), "Length of all instances should be the same due to the padding"
assert dummy_example[0][0] == [0, 0, 12, 13], "Did you consider contexts before and after the target word?"

print('Well done!')
```

```
[([0, 0, 12, 13], 11), ([0, 11, 13, 14], 12), ([11, 12, 14, 15], 13), ([12, 13, 15, 0], 14), ([13, 14, 0, 0], 15)]
Well done!
```

To train our model faster, it is good idea to batchify our data. For your convenience, we implemented it for you:

```
def batch_gen(tknzd_ids, batch_size = 4,  window_size = 5):

    # shuffling all tokenized ids changes the context of each target word
    # shuffle(tknzd_ids) # shuffle is in place and does not return anything

    single_gen = generate_sample(tknzd_ids, window_size) # get sample generator

    while True:
        try:
            # The end of iterations is indicated by an exception
            context_batch = np.zeros([batch_size, window_size * 2], dtype=np.int32)
            target_batch = np.zeros([batch_size], dtype=np.int32)
            for index in range(batch_size):
                context_batch[index], target_batch[index] = next(single_gen)
            yield context_batch, target_batch
        except StopIteration:
            break
```

```
dummy_batches = batch_gen([11, 12, 13, 14, 15, 16, 17, 18], batch_size=4, window_size=2)
```

```
print("First batch:\n", next(dummy_batches))
print('-' * 50)
print("Second batch:\n", next(dummy_batches))
```

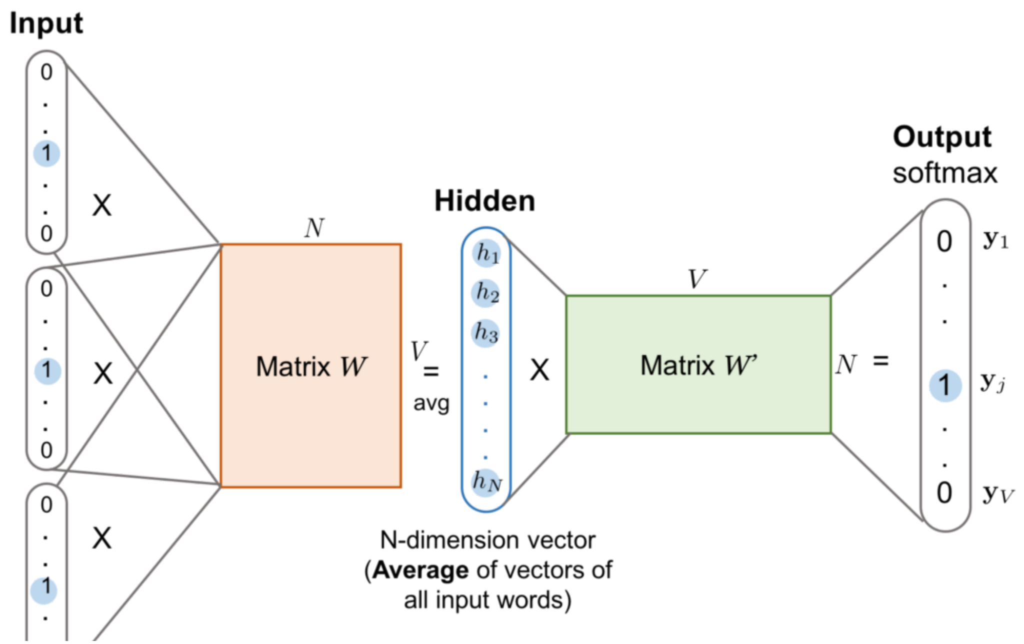
```
First batch:
 (array([[ 0,  0, 12, 13],
       [ 0, 11, 13, 14],
       [11, 12, 14, 15],
       [12, 13, 15, 16]], dtype=int32), array([11, 12, 13, 14], dtype=int32))
--------------------------------------------------
Second batch:
 (array([[13, 14, 16, 17],
       [14, 15, 17, 18],
       [15, 16, 18,  0],
       [16, 17,  0,  0]], dtype=int32), array([15, 16, 17, 18], dtype=int32))
```

2. CBOW Model

We now leverage pytorch to build our CBOW model. For this, our inputs will be our context words which are first converted into one-hot vectors, and next projected into a word-vector. Word-vectors will be obtained from an embedding-matrix (W) which represents the distributed feature vectors associated with each word in the vocabulary. This embedding-matrix is initialized with a normal distribution.

Next, the projected words are averaged out (hence we don't really consider the order or sequence in the context words when averaged) and then we multiply this averaged vector with another embedding matrix (W'), which defines so-called context embeddings to project the CBOW representation back to the one-hot space to match with the target word. (Note: in the theory, this is introduced as the linear output layer, with dimensions equal to the transposed of the embedding matrix.) We thus apply a log-softmax on the resulting context vectors, to predict the most probable target word given the input context.

We match the predicted word with the actual target word, compute the loss by leveraging the cross entropy loss and perform back-propagation with each iteration to update the embedding-matrix in the process.



Question-1

- How could we modify the `CBOW` architecture to consider the order and position of the context words?

By using max pooling instead of average pooling.

Now, complete the CBOW class below, following the instructions in the comments.

```
class CBOW(nn.Module):

    def __init__(self, embedding_dim=100, vocab_size=10000):
        super(CBOW, self).__init__()

        self.vocab_size = vocab_size

        # use nn.Parameter to define the two matrices W and W' from above,
        # thus one for word (W) and one for context (W') embeddings:
```

```
        # self.embed_in = ...  # word embedding
        # self.embed_out = ... # context embedding
        ############### for student ################
        self.embed_in = nn.Parameter(torch.zeros((embedding_dim, self.vocab_size)))
        self.embed_out = nn.Parameter(torch.zeros((self.vocab_size, embedding_dim)))
        ############################################

        self.reset_parameters()


    def reset_parameters(self):
        # Initialize parameters
        nn.init.kaiming_uniform_(self.embed_in, a=math.sqrt(5))
        nn.init.kaiming_uniform_(self.embed_out, a=math.sqrt(5))

    def get_word_embedding(self):
        return self.embed_in

    def get_context_embedding(self):
        return self.embed_out


    def forward(self, inps):
        """
        Convert given indices to log-probablities.
        Follow these steps:
        1) convert the inputs' word indices to one-hot vectors
        2) project the one-hot vectors to their embedding (use F.linear, do *NOT* use nn.Embedding)
        3) calculate the mean of the embedded vectors
        4) project back with the context embedding matrix
        5) calculate the log-probability (with F.log_softmax)

        :argument:
            inps (list): List of indices

        :return:
            log-probablity of words
        """
```

```
        ############### for student ################
        one_hot = F.one_hot(inps, self.vocab_size)
        word_embed = F.linear(one_hot.float(), self.embed_in)
        mean = word_embed.mean(dim=1)
        context_embed = F.linear(mean, self.embed_out)
        log_probs = F.log_softmax(context_embed)
        ############################################
        return log_probs
```

```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY

dummy_model = CBOW(20, 10)
dummy_inps1 = torch.tensor([[6, 7, 9, 0]], dtype=torch.long)
dummy_inps2 = torch.tensor([[6, 7, 9, 0], [1, 2, 3, 4]], dtype=torch.long)
dummy_pred1 = dummy_model(dummy_inps1)
dummy_pred2 = dummy_model(dummy_inps2)

assert isinstance(dummy_model.embed_in, nn.Parameter), "Use nn.Parameter for embed_in"
assert isinstance(dummy_model.embed_out, nn.Parameter), "Use nn.Parameter for embed_out"
assert dummy_model.embed_in.shape == torch.Size([20, 10]), "param_in shape is not correct"
assert dummy_model.embed_out.shape == torch.Size([10, 20]), "param_out shape is not correct"
assert dummy_pred1.shape == torch.Size([1,10]), "Prediction shape is not correct"
assert dummy_pred2.shape == torch.Size([2,10]), "Prediction shape is not correct"
assert dummy_pred1.grad_fn.__class__.__name__ == 'LogSoftmaxBackward', "softmax layer?"

print('Well done!')
```

```
Well done!
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:53: UserWarning: Implicit dimension choice for log_softmax has beer
```

Train Model

Before jumping into the training part, we need to define some hyper-parameters:

```
# embedding hyper-parameters

EMBED_DIM = 100
WINDOW_SIZE = 5
BATCH_SIZE = 128
VOCAB_SIZE = 10_000

EPOCHS = 1 # to make things faster in this basic setup
interval = 1000
```

```
# get data

tokenized_ids, id_to_token, _ = get_data(tokenized_text, VOCAB_SIZE)
```

Now we define our main training loop. Please implement the typical steps for training:

- Reset all gradients
- Compute output and loss value
- Perform back-propagation
- Update the network's parameters

```
model = CBOW(EMBED_DIM, VOCAB_SIZE)
model = model.to(device)

criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters())

loss_history = []

for e in range(EPOCHS):

    batches = batch_gen(tokenized_ids, batch_size=BATCH_SIZE, window_size=WINDOW_SIZE)
    total_loss = 0.0
```

```
for iteration, (context, target) in enumerate(batches):

    # Step 1. Prepare the inputs to be passed to the model (wrap integer indices in tensors)
    # Step 2. Recall that torch *accumulates* gradients. Before passing a
    #         new instance, you need to zero out the gradients from the old instance
    # Step 3. Run the forward pass, getting predicted target words log probabilities
    # Step 4. Compute your loss function.
    # Step 5. Do the backward pass and update the gradient

    ############### for student ################
    context = torch.LongTensor(context).to(device)
    target = torch.LongTensor(target).to(device)

    optimizer.zero_grad()

    log_probs = model.forward(context)
    loss = criterion(log_probs, target)

    loss.backward()
    optimizer.step()
    ############################################

    total_loss += loss.item()

    if iteration % interval == 0:
        print('Epoch:{}/{},\tIteration:{},\tLoss:{}'.format(e, EPOCHS, iteration, total_loss / interval))#, end = "\r", flush = T
        loss_history.append(total_loss / interval)
        total_loss = 0.0
```

```
Epoch:0/1,      Iteration:75000,        Loss:5.647324280977249
Epoch:0/1,      Iteration:76000,        Loss:5.503534972667694
Epoch:0/1,      Iteration:77000,        Loss:5.527165543317794
Epoch:0/1,      Iteration:78000,        Loss:5.58760048365593
Epoch:0/1,      Iteration:79000,        Loss:5.623143346309662
Epoch:0/1,      Iteration:80000,        Loss:5.546127943992615
Epoch:0/1,      Iteration:81000,        Loss:5.588176884174347
Epoch:0/1,      Iteration:82000,        Loss:5.473987120866775
Epoch:0/1,      Iteration:83000,        Loss:5.485766774177551
Epoch:0/1,      Iteration:84000,        Loss:5.54457437825203
Epoch:0/1,      Iteration:85000,        Loss:5.617064198255539
Epoch:0/1,      Iteration:86000,        Loss:5.540236554861068
Epoch:0/1,      Iteration:87000,        Loss:5.501413831949234
Epoch:0/1,      Iteration:88000,        Loss:5.60458944773674
Epoch:0/1,      Iteration:89000,        Loss:5.525987426757813
Epoch:0/1,      Iteration:90000,        Loss:5.548740146160125
Epoch:0/1,      Iteration:91000,        Loss:5.573115302801132
Epoch:0/1,      Iteration:92000,        Loss:5.51930584359169
Epoch:0/1,      Iteration:93000,        Loss:5.45489479637146
Epoch:0/1,      Iteration:94000,        Loss:5.597678895950318
Epoch:0/1,      Iteration:95000,        Loss:5.641539286136627
Epoch:0/1,      Iteration:96000,        Loss:5.44974497961998
Epoch:0/1,      Iteration:97000,        Loss:5.57084781050682
Epoch:0/1,      Iteration:98000,        Loss:5.512481476306915
Epoch:0/1,      Iteration:99000,        Loss:5.5466942312717435
Epoch:0/1,      Iteration:100000,       Loss:4.9509973757267
Epoch:0/1,      Iteration:101000,       Loss:4.993693380117416
Epoch:0/1,      Iteration:102000,       Loss:5.157804739713669
Epoch:0/1,      Iteration:103000,       Loss:5.2345543820858005
Epoch:0/1,      Iteration:104000,       Loss:5.387575642585754
Epoch:0/1,      Iteration:105000,       Loss:5.440636324882507
Epoch:0/1,      Iteration:106000,       Loss:5.501149267911911
Epoch:0/1,      Iteration:107000,       Loss:5.470418267965317
Epoch:0/1,      Iteration:108000,       Loss:5.422294055461884
Epoch:0/1,      Iteration:109000,       Loss:5.421575326919555
Epoch:0/1,      Iteration:110000,       Loss:5.396064746141434
Epoch:0/1,      Iteration:111000,       Loss:5.627514048576355
Epoch:0/1,      Iteration:112000,       Loss:5.466992191314697
Epoch:0/1,      Iteration:113000,       Loss:5.370510748207569
Epoch:0/1,      Iteration:114000,       Loss:5.508462253570556
Epoch:0/1,      Iteration:115000,       Loss:5.447392021656037
Epoch:0/1,      Iteration:116000,       Loss:5.444370402097702
```

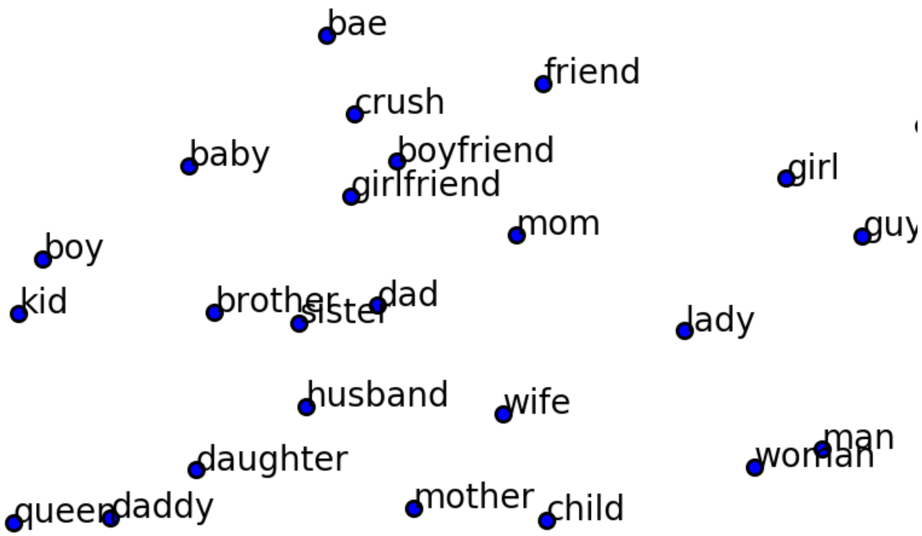
```
Epoch:0/1,      Iteration:117000,       Loss:5.410220887899399
Epoch:0/1,      Iteration:118000,       Loss:5.585351082324982
Epoch:0/1,      Iteration:119000,       Loss:5.5671480720043185
Epoch:0/1,      Iteration:120000,       Loss:5.5660786802768705
Epoch:0/1,      Iteration:121000,       Loss:5.286805953264237
Epoch:0/1,      Iteration:122000,       Loss:5.185703769922257
Epoch:0/1,      Iteration:123000,       Loss:5.424082554578781
Epoch:0/1,      Iteration:124000,       Loss:5.493606802225113
Epoch:0/1,      Iteration:125000,       Loss:5.247191852092743
Epoch:0/1,      Iteration:126000,       Loss:5.439170330762863
Epoch:0/1,      Iteration:127000,       Loss:5.518364743232727
Epoch:0/1,      Iteration:128000,       Loss:5.443316512584686
Epoch:0/1,      Iteration:129000,       Loss:5.28316717171669
Epoch:0/1,      Iteration:130000,       Loss:5.401509357690811
Epoch:0/1,      Iteration:131000,       Loss:5.354206557750702
Epoch:0/1,      Iteration:132000,       Loss:5.472748785257339
```

```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY

assert loss_history[-1] < 6.5

print('Well done!')
```

```
Well done!
```

Nearest words

So far, we trained the **CBOW** successfully, now it is time to explore it more. In this part, we want to find the k nearest word to a given word, i.e., nearby in the vector space.



Define a helper function to retrieve the corresponding vector for a given word:

```
# be sure jupyter session is not terminated!
# use token_to_id to retrieve the index

def get_vector(embedding, word):
    """
    :argument:
        embedding (matrix): embedding matrix
        word (str): The given input
    :return:
        word-vector for a given word
    """
    ############### for student ################
    index = token_to_id[word]
    word_vector = embedding[:,index].unsqueeze(1)
    return word_vector
    ############################################
```

```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY
```

```
embedding = model.embed_in.data

assert get_vector(embedding, 'the').shape == torch.Size([100, 1]), "vector size should be (embed_dim, 1)"
assert np.allclose(embedding[:,(0,)].data.cpu().numpy(), get_vector(embedding, 'UNK').data.cpu().numpy()), "Do you retrieve correct v
print('Well done!')
```

```
Well done!
```

Define a function to return the list of k most similar words, e.g., based on `cosine-similarity`, to a given word:

```
def most_similar_words(embedding, word, k=1):
    """
    return k similar (based on cosine similarity) items
    :argument:
        embedding (matrix): embedding matrix
        word (str): The given input
        k (int): The number of similar items
    :return:
        list of k similar items
    """
    x = get_vector(embedding, word) # 100, 1
    ############### for student ################
    distances = F.cosine_similarity(embedding.T, x.T)
    ids = torch.argsort(distances, descending=True)[1:k+1]
    most_similar = [id_to_token[id.item()] for id in ids]
    ############################################
    return most_similar
```

```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY

embedding = model.embed_in.data

dummy_list = most_similar_words(embedding, "mutual", 3)
s1 = F.cosine_similarity(get_vector(embedding, dummy_list[0]).T, get_vector(embedding, "mutual").T)
```

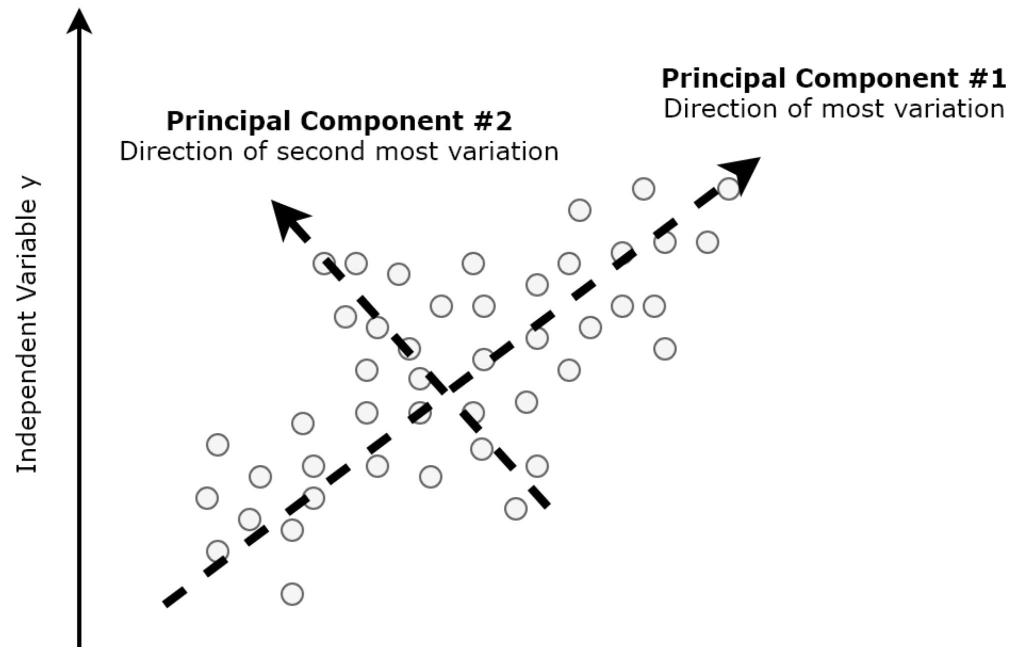
```
s2 = F.cosine_similarity(get_vector(embedding, dummy_list[1]).T, get_vector(embedding, "mutual").T)
s3 = F.cosine_similarity(get_vector(embedding, dummy_list[2]).T, get_vector(embedding, "mutual").T)

assert len(dummy_list) == 3, "return k nearest words"
assert s1.data.cpu().numpy()[0] >= s2.data.cpu().numpy()[0], "first item should have higher probablity to the given word"
assert s2.data.cpu().numpy()[0] >= s3.data.cpu().numpy()[0], "second item should have higher probability"
assert s1.data.cpu().numpy()[0] != 1 , "Similarity score of one means you return the word itself"

print('Well done!')
```

```
Well done!
```

Linear projection

The simplest linear dimensionality reduction method is **P**rincipial **C**omponent **A**nalysis.

In geometric terms, PCA tries to find axes along which most of the variance occurs. The "natural" axes, if you wish.



Under the hood, it attempts to decompose an object-feature matrix X into two smaller matrices: W and $\hat{W}$ minimizing the *mean squared error*:

$$\min_{W,\hat{W}} \; \|(XW)\hat{W} - X\|_2^2$$

with

- $X \in \mathbb{R}^{n \times m}$ - object matrix (**centered**);
- $W \in \mathbb{R}^{m \times d}$ - matrix of direct transformation;
- $\hat{W} \in \mathbb{R}^{d \times m}$ - matrix of reverse transformation;
- n samples, m original dimensions and d target dimensions;

```
from sklearn.decomposition import PCA

# Map word vectors onto a 2D plane with PCA. Use the good old sklearn API (fit, transform).
# Finally, normalize the mapped vectors, to make sure they have zero mean and unit variance

############### for student ################
word_vectors = embedding.data.cpu().numpy()
```

```
pca = PCA(n_components=2, whiten=True)
word_vectors_pca = pca.fit_transform(word_vectors)
############################################
```

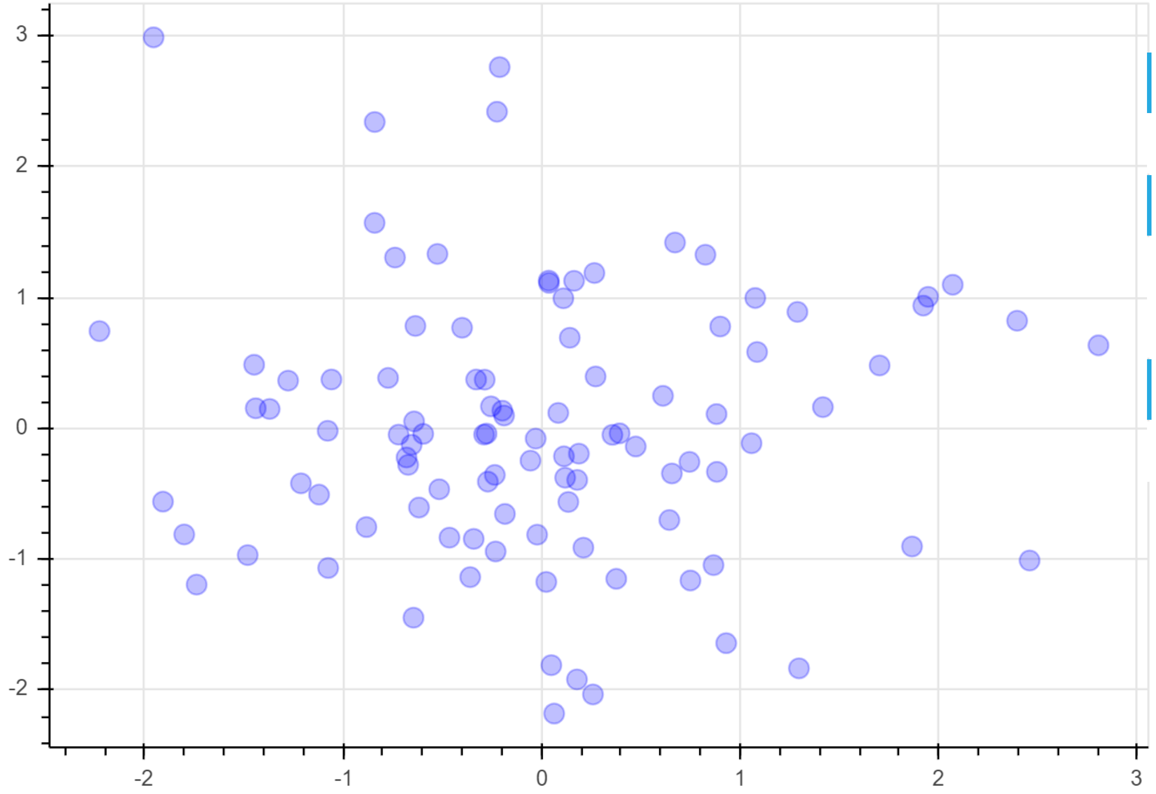
```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY

assert word_vectors_pca.shape == (len(word_vectors), 2), "there must be a 2D vector for each word"
assert max(abs(word_vectors_pca.mean(0))) < 1e-5, "points must be zero-centered"
assert max(abs(1.0 - word_vectors_pca.std(0))) < 1e-2, "points must have unit variance"

print('Well done')
```

```
Well done
```

```
# !pip install bokeh

import bokeh.models as bm, bokeh.plotting as pl
from bokeh.io import output_notebook

output_notebook()

def draw_vectors(x, y, radius=10, alpha=0.25, color='blue',
                 width=600, height=400, show=True, **kwargs):
    """ draws an interactive plot for data points with auxiliary info on hover """
    if isinstance(color, str): color = [color] * len(x)
    data_source = bm.ColumnDataSource({ 'x' : x, 'y' : y, 'color': color, **kwargs })

    fig = pl.figure(active_scroll='wheel_zoom', width=width, height=height)
    fig.scatter('x', 'y', size=radius, color='color', alpha=alpha, source=data_source)

    fig.add_tools(bm.HoverTool(tooltips=[(key, "@" + key) for key in kwargs.keys()]))
    if show: pl.show(fig)
    return fig
```

```
draw_vectors(word_vectors_pca[:, 0], word_vectors_pca[:, 1], token=list(id_to_token.values()))
```

```
BokehUserWarning: ColumnDataSource's columns must be of the same length. Current lengths: ('color', 100), ('token', 10000), ('x
```



Figure(id = '1002', ...)

Visualizing neighbors with t-SNE

PCA is nice but it's strictly linear and thus only able to capture coarse high-level structure of the data.

If we instead want to focus on keeping neighboring points near, we could use TSNE, which is itself an embedding method. Here you can read more on TSNE.

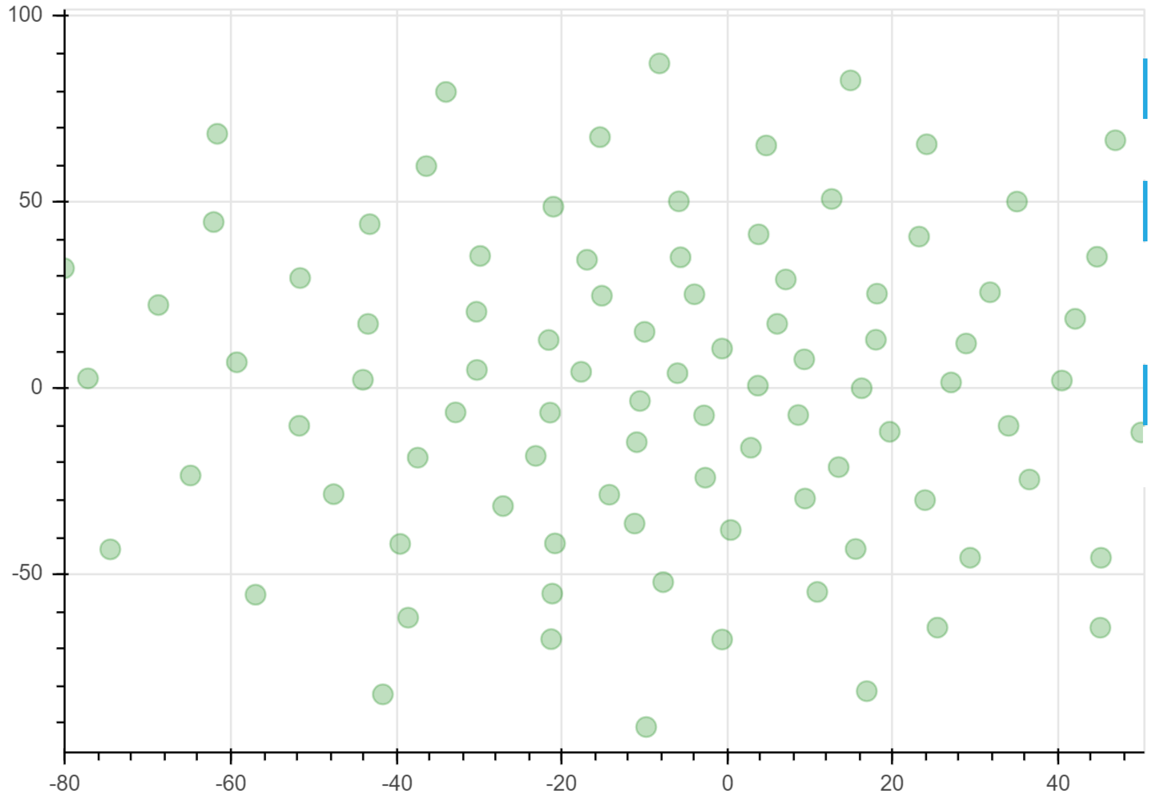
```
from sklearn.manifold import TSNE

# Map word vectors onto a 2d plane with TSNE. (Hint: use verbose=100 to see what it's doing.)
# Normalize them just like with PCA into word_tsne
```

```
# Normalize them just like with PCA into word_tsne

############### for student ################
tsne = TSNE(n_components=2,verbose=100)
word_tsne = tsne.fit_transform(word_vectors)
############################################
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 100 samples in 0.001s...
[t-SNE] Computed neighbors for 100 samples in 0.019s...
[t-SNE] Computed conditional probabilities for sample 100 / 100
[t-SNE] Mean sigma: 17.458120
[t-SNE] Computed conditional probabilities in 0.019s
[t-SNE] Iteration 50: error = 78.1549301, gradient norm = 0.4270932 (50 iterations in 0.921s)
[t-SNE] Iteration 100: error = 84.7751923, gradient norm = 0.3823479 (50 iterations in 0.744s)
[t-SNE] Iteration 150: error = 87.7745285, gradient norm = 0.3684431 (50 iterations in 0.467s)
[t-SNE] Iteration 200: error = 86.1695480, gradient norm = 0.3078792 (50 iterations in 0.905s)
[t-SNE] Iteration 250: error = 79.3507080, gradient norm = 0.4334438 (50 iterations in 0.502s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 79.350708
[t-SNE] Iteration 300: error = 1.5316188, gradient norm = 0.0038714 (50 iterations in 0.316s)
[t-SNE] Iteration 350: error = 1.1856136, gradient norm = 0.0016147 (50 iterations in 0.508s)
[t-SNE] Iteration 400: error = 1.0662470, gradient norm = 0.0005160 (50 iterations in 0.308s)
[t-SNE] Iteration 450: error = 1.0214061, gradient norm = 0.0005548 (50 iterations in 0.258s)
[t-SNE] Iteration 500: error = 0.9990393, gradient norm = 0.0002731 (50 iterations in 0.229s)
[t-SNE] Iteration 550: error = 0.9829387, gradient norm = 0.0002624 (50 iterations in 0.518s)
[t-SNE] Iteration 600: error = 0.9645217, gradient norm = 0.0002733 (50 iterations in 0.220s)
[t-SNE] Iteration 650: error = 0.9362703, gradient norm = 0.0004367 (50 iterations in 0.241s)
[t-SNE] Iteration 700: error = 0.9142976, gradient norm = 0.0001829 (50 iterations in 0.214s)
[t-SNE] Iteration 750: error = 0.9070136, gradient norm = 0.0001754 (50 iterations in 0.501s)
[t-SNE] Iteration 800: error = 0.8955668, gradient norm = 0.0002331 (50 iterations in 0.258s)
[t-SNE] Iteration 850: error = 0.8810040, gradient norm = 0.0003621 (50 iterations in 0.268s)
[t-SNE] Iteration 900: error = 0.8740257, gradient norm = 0.0001722 (50 iterations in 0.255s)
[t-SNE] Iteration 950: error = 0.8659576, gradient norm = 0.0001889 (50 iterations in 0.219s)
[t-SNE] Iteration 1000: error = 0.8587109, gradient norm = 0.0004381 (50 iterations in 0.227s)
[t-SNE] KL divergence after 1000 iterations: 0.858711
```

```
draw_vectors(word_tsne[:, 0], word_tsne[:, 1], color='green', token=list(id_to_token.values()))
```

```
BokehUserWarning: ColumnDataSource's columns must be of the same length. Current lengths: ('color', 100), ('token', 10000), ('x
```



3. POS tagging task

The embeddings by themselves are nice to have, but the main objective of course is to solve a particular (NLP) task. Further, so far we have trained our own embedding from a given corpus, but often it is beneficial to use existing word embeddings.

Now, let's use embeddings to train a simple Part of Speech (PoS) tagging model, using pretrained word embeddings. We shall use [50d glove word vectors]() for the rest of this section.

Before jumping into our neural POS tagger, it is better to set up a baseline to give us an intuition how the neural model performs compared to other models. The baseline model is the [Conditional-Random-Field (CRF)](https://en.wikipedia.org/wiki/Conditional_random_field, also discussed in lecture `NLP_03_PoS_tagging_and_NER_20201`) which is a discriminative sequence labelling model. The evaluation is done on a 10% sample of the Penn Treebank (which is offered through NLTK)

Download data from `nltk` repository and split it into test (20%) and training (80%) sets:

```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
stopwords = set(stopwords.words('english'))

# download necessary packages from nltk
nltk.download('treebank')
nltk.download('universal_tagset')

tagged_sentence = nltk.corpus.treebank.tagged_sents(tagset='universal')
print("Number of Tagged Sentences ", len(tagged_sentence))
print(tagged_sentence[0])
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Unzipping corpora/treebank.zip.
[nltk_data] Downloading package universal_tagset to /root/nltk_data...
[nltk_data]   Unzipping taggers/universal_tagset.zip.
Number of Tagged Sentences  3914
[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), (',', '.'), ('will', 'VEI
```

```
from sklearn.model_selection import train_test_split

train, test = train_test_split(tagged_sentence, test_size=0.20, random_state=42)

print("Train size: {}".format(len(train)))
print("Test size: {}".format(len(test)))
```

```
Train size: 3131
Test size: 783
```

Setup a baseline

```
def features(sentence, index):
    """
    Return hand designed features for a given word
    :argument:
        sentence: tokenized sentence [w1, w2, ...]
        index: index of the word
    :return:
        a feature set for given word
    """

    return {
        'word': sentence[index],
        'is_first': index == 0,
        'is_last': index == len(sentence) - 1,
        'is_capitalized': sentence[index][0].upper() == sentence[index][0],
        'prev_word': '' if index == 0 else sentence[index - 1],
        'next_word': '' if index == len(sentence) - 1 else sentence[index + 1],
        ############### for student ################
        'length': len(sentence[index]),
        # 'sentence_length':len(sentence),
        # 'index': index,
        'is_number': sentence[index].isdigit(),
        # 'is_stopword': sentence[index] in stopwords,
        # 'prev_prev_word': '' if index <= 1 else sentence[index - 2],
        ############################################
    }
```

Question-2

- Suggest about 6 more features that you could improve the above feature-set and add them to the code above. After running the model with these features: which features worked best, and how much did your new features help in improving the model?

results:

- none: 93.76 %
- length: 93.92 %
- sentense_length: 93.41 %
- index: 93.69 %
- is_number: 94.09 %
- is_stopword: 93.66 %
- prev_prev_word: 93.77 %
- all: 93.98 %
- is_number and length: 94.17 %

The new features do not have a big influence on the accuracy of the model. sentence_length, index, and is_stopword even makes the model perform worse. The best features are is_number and length. Using those together results in the best model.

```
def transform2feature_label(tagged_sentence):
    X, y = [], []

    for tagged in tagged_sentence:
        X.append([features([w for w, t in tagged], i) for i in range(len(tagged))])
        y.append([tagged[i][1] for i in range(len(tagged))])

    return X,y
```

```
X_train, y_train = transform2feature_label(train)
X_test, y_test = transform2feature_label(test)
```

```
X_train[0][0]
```

```
{'is_capitalized': True,
 'is_first': True,
 'is_last': False,
 'is_number': False,
```

```
# install crf-classifier
!pip install sklearn-crfsuite
```

```
Collecting sklearn-crfsuite
  Downloading https://files.pythonhosted.org/packages/25/74/5b7befa513482e6dee1f3dd68171a6c9dfc14c0eaa00f885ffeba54fe9b0/sklearn
Collecting python-crfsuite>=0.8.3
  Downloading https://files.pythonhosted.org/packages/95/99/869dde6dbf3e0d07a013c8eebfb0a3d30776334e0097f8432b631a9a3a19/python_
     |████████████████████████████████| 747kB 10.6MB/s
Requirement already satisfied: tabulate in /usr/local/lib/python3.6/dist-packages (from sklearn-crfsuite) (0.8.7)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from sklearn-crfsuite) (1.12.0)
Requirement already satisfied: tqdm>=2.0 in /usr/local/lib/python3.6/dist-packages (from sklearn-crfsuite) (4.38.0)
Installing collected packages: python-crfsuite, sklearn-crfsuite
Successfully installed python-crfsuite-0.9.7 sklearn-crfsuite-0.3.6
```

```
import sklearn_crfsuite

# fit crfsuite classifier on train data
############### for student ################
crf = sklearn_crfsuite.CRF()
crf.fit(X_train, y_train)
############################################

print ("Accuracy:", crf.score(X_test, y_test))
```

```
Accuracy: 0.9417003260499295
```

Build neural model

Now it's time to build our Neural PoS-tagger. The model we want to play with is a bi-directional LSTM on top of pretrained word embeddings. First, we prepare the embedding part and then go into the model itself:

```
# download glove 50d
!wget "https://www.dropbox.com/s/lc3yjhmovq7nyp5/glove6b50dtxt.zip?dl=1" -O glove6b50dtxt.zip
!unzip -o glove6b50dtxt.zip
!rm glove6b50dtxt.zip
```

```
--2020-04-29 18:44:48--  https://www.dropbox.com/s/lc3yjhmovq7nyp5/glove6b50dtxt.zip?dl=1
Resolving www.dropbox.com (www.dropbox.com)... 162.125.81.1, 2620:100:601b:1::a27d:801
Connecting to www.dropbox.com (www.dropbox.com)|162.125.81.1|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /s/dl/lc3yjhmovq7nyp5/glove6b50dtxt.zip [following]
--2020-04-29 18:44:49--  https://www.dropbox.com/s/dl/lc3yjhmovq7nyp5/glove6b50dtxt.zip
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://uc5f5644f890d9c9c2ab8939fff5.dl.dropboxusercontent.com/cd/0/get/A2y8wWY7eZMdj2XTmN3p_YzOcGnlFMwaceydQ4N6fH7Lv
--2020-04-29 18:44:49--  https://uc5f5644f890d9c9c2ab8939fff5.dl.dropboxusercontent.com/cd/0/get/A2y8wWY7eZMdj2XTmN3p_YzOcGnlFM
Resolving uc5f5644f890d9c9c2ab8939fff5.dl.dropboxusercontent.com (uc5f5644f890d9c9c2ab8939fff5.dl.dropboxusercontent.com)... 16
Connecting to uc5f5644f890d9c9c2ab8939fff5.dl.dropboxusercontent.com (uc5f5644f890d9c9c2ab8939fff5.dl.dropboxusercontent.com)|1
HTTP request sent, awaiting response... 200 OK
Length: 70948798 (68M) [application/binary]
Saving to: ‘glove6b50dtxt.zip’

glove6b50dtxt.zip   100%[===================>]  67.66M  21.9MB/s    in 3.1s

2020-04-29 18:44:53 (21.9 MB/s) - ‘glove6b50dtxt.zip’ saved [70948798/70948798]

Archive:  glove6b50dtxt.zip
  inflating: glove.6B.50d.txt
```

```
GLOVE_PATH = 'glove.6B.50d.txt'
```

We build two dictionaries for mapping words and tags to uniqe ids, which we need later on:

```
word_to_id = {}
tag_to_id = {}
```

```
for sentence in tagged_sentence:
    for word, pos_tag in sentence:
        if word not in word_to_id.keys():
            word_to_id[word] = len(word_to_id)
        if pos_tag not in tag_to_id.keys():
            tag_to_id[pos_tag] = len(tag_to_id)

word_vocab_size = len(word_to_id)
tag_vocab_size = len(tag_to_id)

print("Unique words: {}".format(word_vocab_size))
print("Unique tags: {}".format(tag_vocab_size))
```

```
Unique words: 12408
Unique tags: 12
```

We created a wrapper for the embedding module to encapsulate it from the other parts. This module aims to load word vectors from file and assign the weights into the corresponding embedding.

Create an embedding layer (this time use `nn.Embedding`), and assign the pretrained embeddings to its `weight` field. In this exercise, you can continue to finetune the embeddings while training the end task; no need to freeze them: this means the pre-trained embeddings serve as a smart initialization of the embedding layer.

```
class PretrainedEmbeddings(nn.Module):
    def __init__(self, filename, word_to_id, dim_embedding):
        super(PretrainedEmbeddings, self).__init__()

        wordvectors = self.load_word_vectors(filename, word_to_id, dim_embedding)
        ############### for student ################
        self.embed = nn.Embedding(num_embeddings=len(word_to_id), embedding_dim=dim_embedding)
        self.embed.weight = nn.Parameter(wordvectors)
        self.embed.weight.requires_grad = True
        ############################################

    def forward(self, inputs):
```

```
        return self.embed(inputs)

    def load_word_vectors(self, filename, word_to_id, dim_embedding):
        wordvectors = torch.zeros(len(word_to_id), dim_embedding)
        with open(filename, 'r') as file:
            for line in file.readlines():
                data = line.split(' ')
                word = data[0]
                vector = data[1:]
                if word in word_to_id.keys():
                    wordvectors[word_to_id[word],:] = torch.Tensor([float(x) for x in vector])

        return wordvectors
```

```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY

dummy_model = PretrainedEmbeddings(GLOVE_PATH, word_to_id, 50)
dummy_inps = torch.tensor([0, 4, 3, 5, 9], dtype=torch.long)

assert dummy_model.embed.weight.shape == torch.Size([word_vocab_size, 50]), "embedding shape is not correct"
assert dummy_model(dummy_inps).shape == torch.Size([5, 50]), "word embedding shape is not correct"
assert np.allclose(dummy_model.embed.weight.detach().numpy()[0], [0] * 50), "Load weights from glove?"
assert np.allclose(dummy_model.embed.weight.detach().numpy()[714], [0] * 50), "Are you sure you load from glove correctly?"

print('Well done')
```

```
Well done
```

Let’s now define the model. Here’s what we need:

- We’ll need an embedding layer that computes a word vector for each word in a given sentence
- We’ll need a bidirectional-LSTM layer to incorporate context from both directions (reshape the embedding since `nn.LSTM` needs 3-dimensional inputs)

- After the LSTM Layer we need a Linear layer that picks the appropriate POS tag (note that this layer is applied to each element of the sequence).
- Apply the LogSoftmax to calculate the log probabilities from the resulting scores.

```
class POSTagger(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, word_to_id, tag_to_id, embedding_file_path):
        super(POSTagger, self).__init__()

        self.embed = PretrainedEmbeddings(embedding_file_path, word_to_id, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, bidirectional=True)
        self.hidden2tag = nn.Linear(hidden_dim * 2, len(tag_to_id))
        self.logsoftmax = nn.LogSoftmax()

    def forward(self, sentence):
        ############### for student ################
        embeddings = self.embed(sentence)
        hidden, _ = self.lstm(embeddings.unsqueeze(1))
        tag = self.hidden2tag(hidden).squeeze(1)
        tag_scores = self.logsoftmax(tag)
        ############################################
        return tag_scores
```

```
## evaluation
## DON'T CHANGE THIS CELL IN ANY WAY

dummy_model = POSTagger(50, 50, word_to_id, tag_to_id, GLOVE_PATH)
dummy_inps = torch.tensor([0, 4, 3, 5, 9], dtype=torch.long)

assert dummy_model(dummy_inps).grad_fn.__class__.__name__ == 'LogSoftmaxBackward', "softmax layer?"
assert dummy_model(dummy_inps).shape == torch.Size([5, len(tag_to_id)]), "The output has wrong shape! Probably you need some reshapi

print("Well done!")
```

```
Well done!
```

Perfect! Now train your model:

```
# Training start
model = POSTagger(50, 64, word_to_id, tag_to_id, GLOVE_PATH)
model = model.to(device)
criterion = nn.NLLLoss()
optimizer = optim.AdamW(model.parameters())

accuracy_list = []
loss_list = []

interval = round(len(train) / 100.)
EPOCHS = 6
e_interval = round(EPOCHS / 10.)

for e in range(EPOCHS):
    acc = 0
    loss = 0

    model.train()

    for i, sentence_tag in enumerate(train):

        sentence = [word_to_id[s[0]] for s in sentence_tag]
        sentence = torch.tensor(sentence, dtype=torch.long)
        sentence = sentence.to(device)
        targets = [tag_to_id[s[1]] for s in sentence_tag]
        targets = torch.tensor(targets, dtype=torch.long)
        targets = targets.to(device)

        model.zero_grad()

        tag_scores = model(sentence)

        loss = criterion(tag_scores, targets)
```

```
        loss.backward()

        optimizer.step()

        loss += loss.item()

        _, indices = torch.max(tag_scores, 1)

        acc += torch.mean((targets == indices).float())

        if i % interval == 0:
            print("Epoch {} Running;\t{}% Complete".format(e + 1, i / interval), end = "\r", flush = True)

    loss = loss / len(train)
    acc = acc / len(train)
    loss_list.append(float(loss))
    accuracy_list.append(float(acc))

    if (e + 1) % e_interval == 0:
        print("Epoch {} Completed,\tLoss {}\tAccuracy: {}".format(e + 1, np.mean(loss_list[-e_interval:]), np.mean(accuracy_list[-e_i
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:15: UserWarning: Implicit dimension choice for log_softmax has beer
  from ipykernel import kernelapp as app
Epoch 1 Completed,      Loss 3.27258967445232e-05       Accuracy: 0.8677334785461426
Epoch 2 Completed,      Loss 1.4633681530540343e-05     Accuracy: 0.9653012156486511
Epoch 3 Completed,      Loss 1.3041941201663576e-05     Accuracy: 0.9827902913093567
Epoch 4 Completed,      Loss 1.2006984434265178e-05     Accuracy: 0.9901929497718811
Epoch 5 Completed,      Loss 4.038249244331382e-06      Accuracy: 0.994438648223877
Epoch 6 Completed,      Loss 3.2298216865456197e-06     Accuracy: 0.996758222579956
```

So far, so good! It's time to test our classifier. Complete the evaluation part. Compute accuracy on the test data:

```
def evaluate(model, data):
```

```
    model.eval()

    acc = 0.0

    # calculate accuracy based on predictions
    ############### for student ################
    for i, sentence_tag in enumerate(data):
        sentence = [word_to_id[s[0]] for s in sentence_tag]
        sentence = torch.LongTensor(sentence).to(device)
        targets = [tag_to_id[s[1]] for s in sentence_tag]
        targets = torch.LongTensor(targets).to(device)

        tag_scores = model(sentence)
        _, indices = torch.max(tag_scores, 1)
        acc += torch.mean((targets == indices).float())

    score = acc.item() / len(data)
    ############################################
    return score
```

```
score = evaluate(model, test)
print("Accuracy:", score)

assert score > 0.96, "accuracy should be above 96%"
assert score < 1.00, "accuracy should be less than 100!%"

print('Well done!')
```