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.

↳ 2 cells hidden

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
```

```
--2020-04-28 14:51:47--  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   900KB/s    in 34s

2020-04-28 14:52:22 (888 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`:

↳ 1 cell hidden

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
```

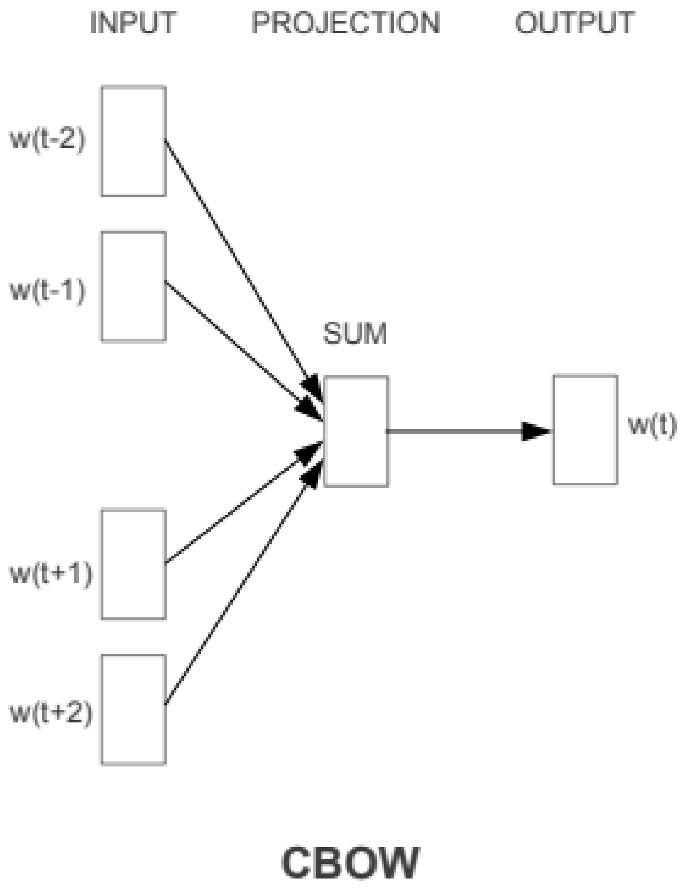
```
tokenized_ids, id_to_token, token_to_id = get_data(tokenized_text)
print('-' * 50)
print('Number of uniqe tokens: {}'.format(len(id_to_token)))
print('-' * 50)
print("tokenized text: {}".format(tokenized_text[0:20]))
print('-' * 50)
print("tokenized ids: {}".format(tokenized_ids[0:20]))
```

```
--------------------------------------------------
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.



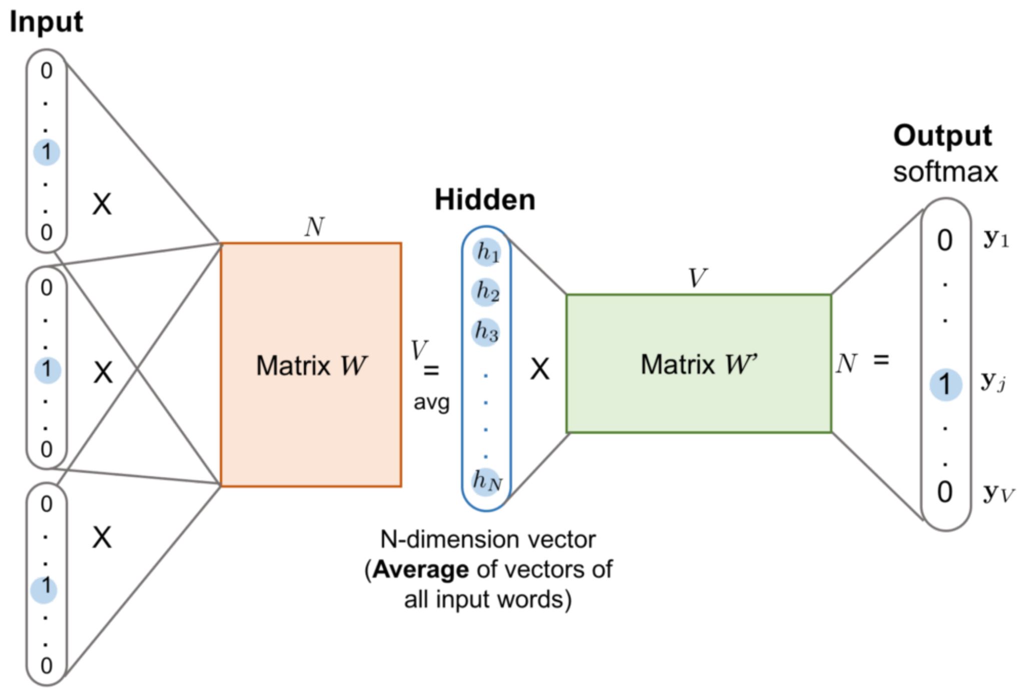
↳ *6 cells hidden*

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.

↳ 3 cells hidden

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
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:53: UserWarning: Implicit dimension choice for log_softmax has beer
Epoch:0/1,	Iteration:0,	Loss:0.009210083961486816
Epoch:0/1,	Iteration:1000, Loss:6.98003680229187
Epoch:0/1,	Iteration:2000, Loss:6.41237203502655
Epoch:0/1,	Iteration:3000, Loss:6.355920181751252
Epoch:0/1,	Iteration:4000, Loss:6.268236405968666
Epoch:0/1,	Iteration:5000, Loss:6.011583807945251
Epoch:0/1,	Iteration:6000, Loss:6.0881432962417605
Epoch:0/1,	Iteration:7000, Loss:5.935342714190483
Epoch:0/1,	Iteration:8000, Loss:5.704964931488037
Epoch:0/1,	Iteration:9000, Loss:5.688196800708771
Epoch:0/1,	Iteration:10000,	Loss:6.082110327959061
Epoch:0/1,	Iteration:11000,	Loss:6.02968112373352
Epoch:0/1,	Iteration:12000,	Loss:5.972457123041153
Epoch:0/1,	Iteration:13000,	Loss:5.8920275294780735
Epoch:0/1,	Iteration:14000,	Loss:5.933533553361893
Epoch:0/1,	Iteration:15000,	Loss:5.758544758796692
Epoch:0/1,	Iteration:16000,	Loss:6.000870287895203
Epoch:0/1,	Iteration:17000,	Loss:5.954390692472458
Epoch:0/1,	Iteration:18000,	Loss:5.969454768419266
Epoch:0/1,	Iteration:19000,	Loss:5.980127104520798
Epoch:0/1,	Iteration:20000,	Loss:5.868254191160202
Epoch:0/1,	Iteration:21000,	Loss:5.960954417705536
Epoch:0/1,	Iteration:22000,	Loss:5.928487191200256
Epoch:0/1,	Iteration:23000,	Loss:5.927696528673172
Epoch:0/1,	Iteration:24000,	Loss:5.881724512100219
Epoch:0/1,	Iteration:25000,	Loss:5.850472261190414
Epoch:0/1,	Iteration:26000,	Loss:5.772154316663742
Epoch:0/1,	Iteration:27000,	Loss:5.739269967079163
Epoch:0/1,	Iteration:28000,	Loss:5.897262148857116
Epoch:0/1,	Iteration:29000,	Loss:5.876156377077103
Epoch:0/1,	Iteration:30000,	Loss:5.830652189254761
Epoch:0/1,	Iteration:31000,	Loss:5.952709870576858
Epoch:0/1,	Iteration:32000,	Loss:5.9373462994098665
Epoch:0/1,	Iteration:33000,	Loss:5.5956826369762425
Epoch:0/1,	Iteration:34000,	Loss:5.930290914773941
Epoch:0/1,	Iteration:35000,	Loss:5.815020031929016
Epoch:0/1,	Iteration:36000,	Loss:5.8009558248519895
Epoch:0/1,	Iteration:37000,	Loss:5.89508767080307
Epoch:0/1,	Iteration:38000,	Loss:5.767864404201507
Epoch:0/1,	Iteration:39000,	Loss:5.761582095384598
Epoch:0/1,	Iteration:40000,	Loss:5.622309210538864
```

```
Epoch:0/1,      Iteration:41000,        Loss:5.752723930835724
Epoch:0/1,      Iteration:42000,        Loss:5.800832928180695
Epoch:0/1,      Iteration:43000,        Loss:5.7503017930984495
Epoch:0/1,      Iteration:44000,        Loss:5.788173782348633
Epoch:0/1,      Iteration:45000,        Loss:5.7320178942680355
Epoch:0/1,      Iteration:46000,        Loss:5.7810084304809575
Epoch:0/1,      Iteration:47000,        Loss:5.717383912086487
Epoch:0/1,      Iteration:48000,        Loss:5.790286029100418
Epoch:0/1,      Iteration:49000,        Loss:5.723027593374252
Epoch:0/1,      Iteration:50000,        Loss:5.64465229010582
Epoch:0/1,      Iteration:51000,        Loss:5.6001872014999385
Epoch:0/1,      Iteration:52000,        Loss:5.670156351566314
Epoch:0/1,      Iteration:53000,        Loss:5.665253391742707
Epoch:0/1,      Iteration:54000,        Loss:5.63490158200264
Epoch:0/1,      Iteration:55000,        Loss:5.625751535892487
Epoch:0/1,      Iteration:56000,        Loss:5.738576672554016
Epoch:0/1,      Iteration:57000,        Loss:5.733219198465347
Epoch:0/1,      Iteration:58000,        Loss:5.571284301519394
Epoch:0/1,      Iteration:59000,        Loss:5.64305598282814
Epoch:0/1,      Iteration:60000,        Loss:5.58204007768631
Epoch:0/1,      Iteration:61000,        Loss:5.670817386388778
Epoch:0/1,      Iteration:62000,        Loss:5.670162817716599
Epoch:0/1,      Iteration:63000,        Loss:5.5971675368547436
Epoch:0/1,      Iteration:64000,        Loss:5.624200838088989
Epoch:0/1,      Iteration:65000,        Loss:5.5601652076244354
Epoch:0/1,      Iteration:66000,        Loss:5.581682207345962
Epoch:0/1,      Iteration:67000,        Loss:5.681835259437561
Epoch:0/1,      Iteration:68000,        Loss:5.475357088088989
Epoch:0/1,      Iteration:69000,        Loss:5.550533980846405
Epoch:0/1,      Iteration:70000,        Loss:5.625025253772735
Epoch:0/1,      Iteration:71000,        Loss:5.514824985027313
Epoch:0/1,      Iteration:72000,        Loss:5.609921064138413
Epoch:0/1,      Iteration:73000,        Loss:5.6210317995548245
Epoch:0/1,      Iteration:74000,        Loss:5.633709595441818
Epoch:0/1,      Iteration:75000,        Loss:5.647324303388595
Epoch:0/1,      Iteration:76000,        Loss:5.503534961938858
Epoch:0/1,      Iteration:77000,        Loss:5.527165512323379
Epoch:0/1,      Iteration:78000,        Loss:5.587600441217423
Epoch:0/1,      Iteration:79000,        Loss:5.623143375635147
Epoch:0/1,      Iteration:80000,        Loss:5.546127947330475
Epoch:0/1,      Iteration:81000,        Loss:5.5881768581867215
Epoch:0/1,      Iteration:82000,        Loss:5.473987106800079
```

```
Epoch:0/1,      Iteration:83000,        Loss:5.4857667939662935
Epoch:0/1,      Iteration:84000,        Loss:5.54457437825203
Epoch:0/1,      Iteration:85000,        Loss:5.6170641911029815
Epoch:0/1,      Iteration:86000,        Loss:5.540236571788788
Epoch:0/1,      Iteration:87000,        Loss:5.501413816690445
Epoch:0/1,      Iteration:88000,        Loss:5.6045894391536715
Epoch:0/1,      Iteration:89000,        Loss:5.525987417221069
Epoch:0/1,      Iteration:90000,        Loss:5.548740170478821
Epoch:0/1,      Iteration:91000,        Loss:5.573115305185318
Epoch:0/1,      Iteration:92000,        Loss:5.519305857658386
Epoch:0/1,      Iteration:93000,        Loss:5.454894805908203
Epoch:0/1,      Iteration:94000,        Loss:5.597678880691529
Epoch:0/1,      Iteration:95000,        Loss:5.641539328575134
Epoch:0/1,      Iteration:96000,        Loss:5.449745023012161
Epoch:0/1,      Iteration:97000,        Loss:5.5708477778434755
Epoch:0/1,      Iteration:98000,        Loss:5.512481455564499
Epoch:0/1,      Iteration:99000,        Loss:5.5466942653656
Epoch:0/1,      Iteration:100000,       Loss:4.950997400283813
Epoch:0/1,      Iteration:101000,       Loss:4.993693469762802
Epoch:0/1,      Iteration:102000,       Loss:5.157804750919342
Epoch:0/1,      Iteration:103000,       Loss:5.2345544228553775
Epoch:0/1,      Iteration:104000,       Loss:5.387575657367706
Epoch:0/1,      Iteration:105000,       Loss:5.4406364030838015
Epoch:0/1,      Iteration:106000,       Loss:5.501149290800095
Epoch:0/1,      Iteration:107000,       Loss:5.470418320417404
Epoch:0/1,      Iteration:108000,       Loss:5.422294109106064
Epoch:0/1,      Iteration:109000,       Loss:5.421575304985047
Epoch:0/1,      Iteration:110000,       Loss:5.3960647859573365
Epoch:0/1,      Iteration:111000,       Loss:5.627514034271241
Epoch:0/1,      Iteration:112000,       Loss:5.46699217414856
Epoch:0/1,      Iteration:113000,       Loss:5.370510830402374
Epoch:0/1,      Iteration:114000,       Loss:5.508462259531021
Epoch:0/1,      Iteration:115000,       Loss:5.447392034053802
Epoch:0/1,      Iteration:116000,       Loss:5.444370437383652
Epoch:0/1,      Iteration:117000,       Loss:5.410220898866654
Epoch:0/1,      Iteration:118000,       Loss:5.585351110219955
Epoch:0/1,      Iteration:119000,       Loss:5.567148096323013
Epoch:0/1,      Iteration:120000,       Loss:5.566078731060028
Epoch:0/1,      Iteration:121000,       Loss:5.286806033372879
Epoch:0/1,      Iteration:122000,       Loss:5.185704100608826
Epoch:0/1,      Iteration:123000,       Loss:5.424082656860351
Epoch:0/1,      Iteration:124000,       Loss:5.493606863498687
```

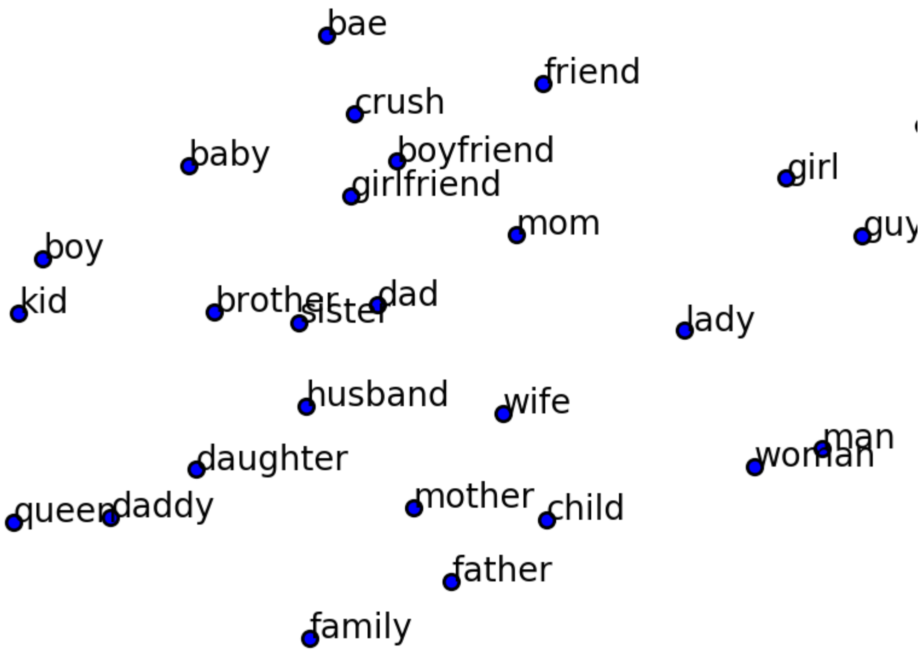
```
Epoch:0/1,      Iteration:125000,       Loss:5.2471918568611144
Epoch:0/1,      Iteration:126000,       Loss:5.439170346736908
Epoch:0/1,      Iteration:127000,       Loss:5.518364778757095
Epoch:0/1,      Iteration:128000,       Loss:5.443316507577896
Epoch:0/1,      Iteration:129000,       Loss:5.283167228460312
Epoch:0/1,      Iteration:130000,       Loss:5.401509329080581
Epoch:0/1,      Iteration:131000,       Loss:5.354206582784653
Epoch:0/1,      Iteration:132000,       Loss:5.472748783826828
```

```
## 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.



↳ 6 cells hidden

Linear projection

↳ 6 cells hidden

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**.

↳ 2 cells hidden

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

↳ 1 cell hidden

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,
 'length': 6,
 'next_word': 'Vinken',
 'prev_word': '',
 'word': 'Pierre'}
```

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

```
!pip install sklearn-crfsuite
```

```
Collecting sklearn-crfsuite
  Downloading https://files.pythonhosted.org/packages/25/74/5b7befa513482e6dee1f3dd68171a6c9dfc14c0eaa00f885ffeba54fe9b0/sklearn
Requirement already satisfied: tqdm>=2.0 in /usr/local/lib/python3.6/dist-packages (from sklearn-crfsuite) (4.38.0)
Collecting python-crfsuite>=0.8.3
  Downloading https://files.pythonhosted.org/packages/95/99/869dde6dbf3e0d07a013c8eebfb0a3d30776334e0097f8432b631a9a3a19/python_
     |████████████████████████████████| 747kB 6.6MB/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)
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
```

```
--2020-04-28 15:00:15--  https://www.dropbox.com/s/lc3yjhmovq7nyp5/glove6b50dtxt.zip?dl=1
Resolving www.dropbox.com (www.dropbox.com)... 162.125.1.1, 2620:100:601b:1::a27d:801
Connecting to www.dropbox.com (www.dropbox.com)|162.125.1.1|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /s/dl/lc3yjhmovq7nyp5/glove6b50dtxt.zip [following]
--2020-04-28 15:00:15--  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://uce8e2bab1bcb0c1e081ebfa5ae5.dl.dropboxusercontent.com/cd/0/get/A2uEQEbAaGLBVmsjN2eYewGO8yQneTKHb4oiOLXfrBVeOI
--2020-04-28 15:00:15--  https://uce8e2bab1bcb0c1e081ebfa5ae5.dl.dropboxusercontent.com/cd/0/get/A2uEQEbAaGLBVmsjN2eYewGO8yQneTI
Resolving uce8e2bab1bcb0c1e081ebfa5ae5.dl.dropboxusercontent.com (uce8e2bab1bcb0c1e081ebfa5ae5.dl.dropboxusercontent.com)... 162
Connecting to uce8e2bab1bcb0c1e081ebfa5ae5.dl.dropboxusercontent.com (uce8e2bab1bcb0c1e081ebfa5ae5.dl.dropboxusercontent.com)|16
HTTP request sent, awaiting response... 200 OK
Length: 70948798 (68M) [application/binary]
Saving to: ‘glove6b50dtxt.zip’

glove6b50dtxt.zip   100%[===================>]  67.66M  56.3MB/s    in 1.2s

2020-04-28 15:00:17 (56.3 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)
        ############################################

    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.

Complete the forward path of the POSTagger model:

```
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!
/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
```

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 been
  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!')
```

```
/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
Accuracy: 0.9592359209121568
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-47-8c593e1c810a> in <module>()
      2 print("Accuracy:", score)
      3
----> 4 assert score > 0.96, "accuracy should be above 96%"
      5 assert score < 1.00, "accuracy should be less than 100!%"
      6

AssertionError: accuracy should be above 96%
```

SEARCH STACK OVERFLOW

Question-3

- Whether or not to fine-tune the pre-trained embeddings, the number of epochs you need (whether or not to use 'early stopping'), to apply regularization... are hyperparameters that should be properly tuned on a validation set. We did not do this here. It is therefore hard to make strong claims about the model at this point. However, as a quick test, please train the POS model with the same settings, but with a standard randomly initialized embedding layer instead of the pretrained embeddings. What do you observe compared to the CRF baseline / compared to the GloVe initialization? (Note: for your final code in `POSTagger`, please make sure it again loads the pretrained embeddings).

Results:

- CRF baseline: 94.17 %
- POSTagger random: 91.05 %
- POSTagger GloVe: 95.68 %

Without the pre-trained GloVe embeddings, the model clearly performs worse than with pre-trained weights. The model with pre-trained GloVe embeddings performs better than the CRF baseline. After tuning some hyperparameters, it should be possible to outperform CRF.

Acknowledgment

If you received help or feedback from fellow students, please acknowledge that here. We count on your academic honesty:

I did not receive any help from fellow students, besides Jarne Verhaeghe, who found a small bug in reshaping the inputs of the LSTM. This is explained into more detail on the forum on Ufora.