Report 3 : NN
AHLT - Advanced Human Languages Technologies

Lackmann Simon
Piotrowski Maciej

1 Introduction

This document is a report for the NERC an DDI using neural networks project of the AHLT - Advanced Human Languages Technologies course. It consists of the implementation of 2 python programs that parses all XML files in the folder given as argument and :

1. recognizes and classifies drug names
2. classifies drug-drug interactions between pairs of drugs

Both using Neural Networks (NN). It's inspired by SemEval 2013 Task 9: DDIExtraction. We received some documents extracted from DrugBank (Drug description leaflets database) and MedLine (abstracts of medical papers) as the initial documentation. We also received three directories containing data files that we had to use as training, validation (devel) and test sets for our experiments. For both parts we received an initial structure with some initial rules that we had to improve. Basic F1 scores were 55% for the first part and 50% for the second part. The goal was to reach respectively 70% and 65%.

2 RNN-based NERC

In this section, we describe the experiments we did with a NN for NERC. We tried different architectures and hyper-parameters and different input informations. We include our code and the results we obtained on the devel and test datasets with our best model. We also describe all the other models we used that we think are relevant and the according results.

2.1 Architecture

We tried several different architecture approaches. As training needs sometimes a very long time, not all possible combinations can be tested. Therefore, we tried to maximize the results of our model by testing playing with:

- Embedding dimensions
- Using GloVe
- Different max_len values, suffixes and prefixes length
- Number of LSTM units
- Differents optimizers
- Number of layers, tanh and relu activation function
- Additional input layers
- Different dropout values
- Adding a BiLSTM

At the end, we obtained best results using the following architecture:

- No GloVe
- 1 input layer per input information, each with an output dimension of 50

- Applying a 10% dropout for every embedding
- 1 BiLSTM with 32 units and a recurrent dropout of 32
- An RMSprop optimizer with a learning rate of 0.001
- max_len of 100, suffixes length of 5 and prefixes length of 4

2.2 Input information

As input information, we added information about :

- Sentence words
- Suffixes
- Prefixes
- Lemma of the word
- Some specific features we used during Lab1:
  - Presence in external sources
  - Presence of some suffixes
  - Presence of some sub-strings
- Lower case form of words

For this purpose, we needed to slightly change the codemaps.py and therefore we added it to the code section. We also tried to add an input information about the word position but it didn't increase the score in a noticeable manner. We also tried to enrich the specific features with information about capitalization and dashes but it didn't improve the score.

2.3 Code

```
#! /usr/bin/python3

import code
import sys
from contextlib import redirect_stdout

from tensorflow.keras import Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import LSTM, Embedding, Dense, TimeDistributed, Dropout, Bidirectional, co
from keras.optimizers import RMSprop

from dataset import *
from codemaps import *

def build_network(codes) :

   # sizes
   n_words = codes.get_n_words()
   n_sufs = codes.get_n_sufs()
   n_prefs = codes.get_n_prefs()
```

```
n_labels = codes.get_n_labels()
max_len = codes.maxlen
n_lem = codes.get_n_lem()
n_lab1 = codes.get_n_lab1()
n_lc = codes.get_n_lc()

inptW = Input(shape=(max_len,)) # word input layer & embeddings
embW = Embedding(input_dim=n_words, output_dim=50,
                 input_length=max_len, mask_zero=True)(inptW)

inptS = Input(shape=(max_len,))  # suf input layer & embeddings
embS = Embedding(input_dim=n_sufs, output_dim=50,
                 input_length=max_len, mask_zero=True)(inptS)

inptP = Input(shape=(max_len,))  # pref input layer & embeddings
embP = Embedding(input_dim=n_prefs, output_dim=50,
                 input_length=max_len, mask_zero=True)(inptP)

inptL = Input(shape=(max_len,))  # lemma input layer & embeddings
embL = Embedding(input_dim=n_lem, output_dim=50,
                 input_length=max_len, mask_zero=True)(inptL)

inptLab1 = Input(shape=(max_len,))  # specific features input layer & embeddings
embLab1 = Embedding(input_dim=n_lab1, output_dim=50,
                 input_length=max_len, mask_zero=True)(inptLab1)

inptLC = Input(shape=(max_len,))  # lowercase input layer & embeddings
embLC = Embedding(input_dim=n_lc, output_dim=50,
                 input_length=max_len, mask_zero=True)(inptLC)


dropW = Dropout(0.1)(embW)
dropS = Dropout(0.1)(embS)
dropP = Dropout(0.1)(embP)
dropL = Dropout(0.1)(embL)
dropLab1 = Dropout(0.1)(embLab1)
dropLC = Dropout(0.1)(embLC)
drops = concatenate([dropW, dropS,dropP,dropL,dropLab1,dropLC])

# biLSTM
bilstm = Bidirectional(LSTM(units=32, return_sequences=True,
                            recurrent_dropout=0.1))(drops)
# output softmax layer
out = TimeDistributed(Dense(n_labels, activation="softmax"))(bilstm)

# build and compile model
model = Model([inptW,inptS,inptP,inptL,inptLab1,inptLC], out)
model.compile(optimizer=RMSprop(lr=0.001, epsilon=None, decay=0.0),
              loss="sparse_categorical_crossentropy", metrics=["accuracy"])
```

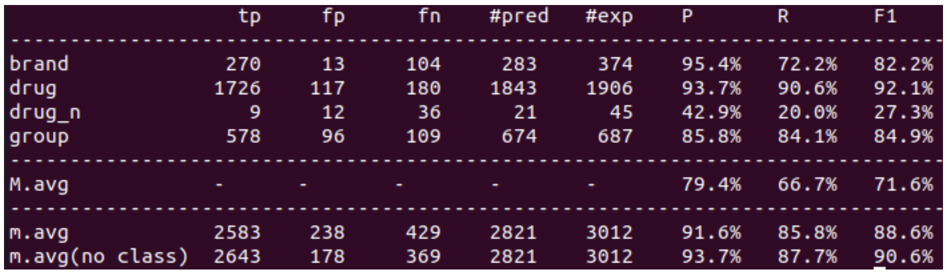
```
   return model



## --------- MAIN PROGRAM -----------
## --
## -- Usage:  train.py ../data/Train ../data/Devel  modelname
## --

# directory with files to process
traindir = sys.argv[1]
validationdir = sys.argv[2]
modelname = sys.argv[3]

# load train and validation data
traindata = Dataset(traindir)
valdata = Dataset(validationdir)

# create indexes from training data
max_len = 100
suf_len = 5
pref_len= 4
codes  = Codemaps(traindata, max_len, suf_len, pref_len)

# build network
model = build_network(codes)
with redirect_stdout(sys.stderr) :
   model.summary()

# encode datasets
Xt = codes.encode_words(traindata)
Yt = codes.encode_labels(traindata)
Xv = codes.encode_words(valdata)
Yv = codes.encode_labels(valdata)

# train model
with redirect_stdout(sys.stderr) :
   model.fit(Xt, Yt, batch_size=32, epochs=10, validation_data=(Xv,Yv), verbose=1)

# save model and indexs
model.save(modelname)
codes.save(modelname)


import string
import re

import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences
import nltk
```

```
nltk.download('wordnet')
nltk.download('omw-1.4')
from nltk.stem import WordNetLemmatizer

# Lemma
lemmatizer = WordNetLemmatizer()

# Dictionnary with external sources
external = {}
with open("resources/HSDB.txt") as h :
    for x in h.readlines() :
        external[x.strip().lower()] = "drug"
with open("resources/DrugBank.txt") as h :
    for x in h.readlines() :
        (n,t) = x.strip().lower().split("|")
        external[n] = t

# Suffixes lists
suffixes = ['azole','mycin']
suffixesGroup = ['otics','tives','sants','ators']

from dataset import *

class Codemaps :
    # --- constructor, create mapper either from training data, or
    # --- loading codemaps from given file
    def __init__(self, data, maxlen=None, suflen=None, preflen=None) :

        if isinstance(data,Dataset) and maxlen is not None and suflen is not None and preflen is not No
            self.__create_indexs(data, maxlen, suflen, preflen)

        elif type(data) == str and maxlen is None and suflen is None and preflen is None:
            self.__load(data)

        else:
            print('codemaps: Invalid or missing parameters in constructor')
            exit()


    # --------- Create indexs from training data
    # Extract all words and labels in given sentences and
    # create indexes to encode them as numbers when needed
    def __create_indexs(self, data, maxlen, suflen, preflen) :

        self.maxlen = maxlen
        self.suflen = suflen
        self.preflen = preflen
        words = set([])
        sufs = set([])
        prefs = set([])
        lem = set([])
        labels = set([])
        lab1 = set([])
        lc = set([])

        for s in data.sentences() :
            for t in s :
                words.add(t['form'])
                sufs.add(t['lc_form'][-self.suflen:])
                prefs.add(t['lc_form'][0:self.preflen])
                labels.add(t['tag'])
                lem.add(lemmatizer.lemmatize(t['lc_form']))
                lab1.add(t['form'])
                lc.add(t['lc_form'])

        self.word_index = {w: i+2 for i,w in enumerate(list(words))}
        self.word_index['PAD'] = 0 # Padding
        self.word_index['UNK'] = 1 # Unknown words

        self.suf_index = {s: i+2 for i,s in enumerate(list(sufs))}
        self.suf_index['PAD'] = 0  # Padding
        self.suf_index['UNK'] = 1  # Unknown suffixes

        self.pref_index = {s: i+2 for i,s in enumerate(list(prefs))}
        self.pref_index['PAD'] = 0  # Padding
        self.pref_index['UNK'] = 1  # Unknown prefixes

        self.lem_index = {s: i+2 for i,s in enumerate(list(lem))}
        self.lem_index['PAD'] = 0  # Padding
        self.lem_index['UNK'] = 1  # Unknown lemmas

        self.lab1_index = {w: i+2 for i,w in enumerate(list(lab1))}
        self.lab1_index['PAD'] = 0 # Padding
        # Drug types
        self.lab1_index['brand'] = 1
        self.lab1_index['drug'] = 2
        self.lab1_index['drug_n'] = 3
        self.lab1_index['group'] = 4
        self.lab1_index['NONE'] = 5

        self.lc_index = {w: i+2 for i,w in enumerate(list(lc))}
        self.lc_index['PAD'] = 0 # Padding
        self.lc_index['UNK'] = 1 # Unknown lower case form words

        self.label_index = {t: i+1 for i,t in enumerate(list(labels))}
        self.label_index['PAD'] = 0 # Padding

    ## --------- load indexs -----------
    def __load(self, name) :
        self.maxlen = 0
        self.suflen = 0
        self.word_index = {}
        self.suf_index = {}
        self.pref_index = {}
        self.label_index = {}
        self.lem_index = {}
        self.lab1_index = {}
        self.lc_index = {}

        with open(name+".idx") as f :
            for line in f.readlines():
                (t,k,i) = line.split()
                if t == 'MAXLEN' : self.maxlen = int(k)
                elif t == 'SUFLEN' : self.suflen = int(k)
                elif t == 'PREFLEN' : self.preflen = int(k)
                elif t == 'WORD': self.word_index[k] = int(i)
                elif t == 'SUF': self.suf_index[k] = int(i)
                elif t == 'PREF': self.pref_index[k] = int(i)
                elif t == 'LABEL': self.label_index[k] = int(i)
                elif t == 'LEM': self.lem_index[k] = int(i)
                elif t == 'LAB1': self.lab1_index[k] = int(i)
                elif t == 'LC': self.lc_index[k] = int(i)

    ## ---------- Save model and indexs ---------------
    def save(self, name) :
        # save indexes
        with open(name+".idx","w") as f :
            print ('MAXLEN', self.maxlen, "-", file=f)
            print ('SUFLEN', self.suflen, "-", file=f)
            print ('PREFLEN', self.preflen, "-", file=f)
            for key in self.label_index : print('LABEL', key, self.label_index[key], file=f)
            for key in self.word_index : print('WORD', key, self.word_index[key], file=f)
            for key in self.suf_index : print('SUF', key, self.suf_index[key], file=f)
            for key in self.pref_index : print('PREF', key, self.pref_index[key], file=f)
            for key in self.lem_index : print('LEM', key, self.lem_index[key], file=f)
            for key in self.lab1_index : print('LAB1', key, self.lab1_index[key], file=f)
            for key in self.lc_index : print('LC', key, self.lc_index[key], file=f)


    ## --------- encode X from given data -----------
    def encode_words(self, data) :
        # encode and pad sentence words
        Xw = [[self.word_index[w['form']] if w['form'] in self.word_index else self.word_index['UNK'] i
        Xw = pad_sequences(maxlen=self.maxlen, sequences=Xw, padding="post", value=self.word_index['PA

        # encode and pad suffixes
        Xs = [[self.suf_index[w['lc_form'][-self.suflen:]] if w['lc_form'][-self.suflen:] in self.suf_
        Xs = pad_sequences(maxlen=self.maxlen, sequences=Xs, padding="post", value=self.suf_index['PAD
        # encode and pad prefixes
        Xp = [[self.pref_index[w['lc_form'][0:self.preflen]] if w['lc_form'][0:self.preflen] in self.pr
        Xp = pad_sequences(maxlen=self.maxlen, sequences=Xp, padding="post", value=self.pref_index['PA

        # encode and pad lemmas
        Xl = [[self.lem_index[lemmatizer.lemmatize(w['lc_form'])] if lemmatizer.lemmatize(w['lc_form']
        Xl = pad_sequences(maxlen=self.maxlen, sequences=Xl, padding="post", value=self.lem_index['PAD

        # encode and pad specific features extracted from lab1
        Xlab1=[]
        for s in data.sentences():
            sent=[]
            for w in s:
                # Presence in an external source
                if w['lc_form'] in external :
                    sent.append(self.lab1_index[external[w['lc_form']]])
                # Suffixe present in our lists
                elif w['form'][-5:] in suffixes : sent.append(self.lab1_index["drug"])
                elif w['form'][-5:] in suffixesGroup : sent.append(self.lab1_index["group"])
                # Presence of a substring
                elif "methyl" in w['form'] : sent.append(self.lab1_index["drug_n"])
                elif "acids" in w['form'] : sent.append(self.lab1_index["group"])
                elif "blocker" in w['form'] : sent.append(self.lab1_index["group"])
                elif "toxin" in w['form'] : sent.append(self.lab1_index["drug_n"])
                elif "gaine" in w['form'] : sent.append(self.lab1_index["drug_n"])
                else : sent.append(self.lab1_index["NONE"])
            Xlab1.append(sent)
        Xlab1 = pad_sequences(maxlen=self.maxlen, sequences=Xlab1, padding="post", value=self.lab1_inde

        # encode and pad lower case words form
        Xlc = [[self.lc_index[w['lc_form']] if w['lc_form'] in self.lc_index else self.lc_index['UNK']
        Xlc = pad_sequences(maxlen=self.maxlen, sequences=Xlc, padding="post", value=self.lc_index['PA

        # return encoded sequences
        return [Xw,Xs,Xp,Xl,Xlab1,Xlc]


    ## --------- encode Y from given data -----------
    def encode_labels(self, data) :
        # encode and pad sentence labels
        Y = [[self.label_index[w['tag']] for w in s] for s in data.sentences()]
        Y = pad_sequences(maxlen=self.maxlen, sequences=Y, padding="post", value=self.label_index["PAD"
        return np.array(Y)

    ## -------- get word index size ---------
    def get_n_words(self) :
        return len(self.word_index)
    ## -------- get suf index size ---------
    def get_n_sufs(self) :
        return len(self.suf_index)
    ## -------- get pref index size ---------
    def get_n_prefs(self) :
        return len(self.pref_index)
    ## -------- get label index size ---------
    def get_n_labels(self) :
        return len(self.label_index)
    ## -------- get lemma index size ---------
    def get_n_lem(self) :
        return len(self.lem_index)
    ## -------- get specific features index size ---------
    def get_n_lab1(self) :
        return len(self.lab1_index)
    ## -------- get lower case words index size ---------
    def get_n_lc(self) :
        return len(self.lc_index)
    ## -------- get index for given word ---------
    def word2idx(self, w) :
        return self.word_index[w]
    ## -------- get index for given suffix --------
    def suff2idx(self, s) :
        return self.suf_index[s]
    ## -------- get index for given prefix --------
    def preff2idx(self, p) :
        return self.pref_index[p]
    ## -------- get index for given label --------
    def label2idx(self, l) :
        return self.label_index[l]
    ## -------- get index for given lemma --------
    def lem2idx(self, l) :
        return self.lem_index[l]
    ## -------- get index for given specific feature --------
    def lab12idx(self, l) :
        return self.lab1_index[l]
    ## -------- get index for given lowercase --------
    def lc2idx(self, l) :
        return self.lc_index[l]
    ## -------- get label name for given index --------
    def idx2label(self, i) :
        for l in self.label_index :
            if self.label_index[l] == i:
                return l
        raise KeyError
```

2.4 Experiments and results

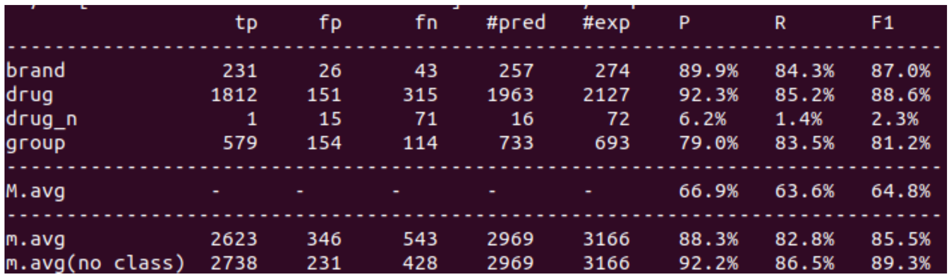
In this section we describe the results of our experiments. Important is that the usage of neural networks can produce very different results. Because of that, we should have ran the model - for good working architectures and parameters - multiple times in order to get averaged results. But due to limited resources and very long training time, we decided base our trust on a single run, in order to test as much as possible.

With the code included in the previous section , we managed to reach a F1-Score on the devel data of 71.6% and 64.8% on the test data which is a quite good result because we are not overfitting too much.



```
                   tp     fp      fn    #pred    #exp    P       R       F1
------------------------------------------------------------------------------
brand             270     13     104     283      374    95.4%   72.2%   82.2%
drug             1726    117     180    1843     1906    93.7%   90.6%   92.1%
drug_n              9     12      36      21       45    42.9%   20.0%   27.3%
group             578     96     109     674      687    85.8%   84.1%   84.9%
------------------------------------------------------------------------------
M.avg               -      -       -       -        -    79.4%   66.7%   71.6%
------------------------------------------------------------------------------
m.avg            2583    238     429    2821     3012    91.6%   85.8%   88.6%
m.avg(no class)  2643    178     369    2821     3012    93.7%   87.7%   90.6%
```

Figure 1: Best score on the devel data



```
                   tp     fp      fn    #pred    #exp    P       R       F1
------------------------------------------------------------------------------
brand             231     26      43     257      274    89.9%   84.3%   87.0%
drug             1812    151     315    1963     2127    92.3%   85.2%   88.6%
drug_n              1     15      71      16       72    6.2%    1.4%    2.3%
group             579    154     114     733      693    79.0%   83.5%   81.2%
------------------------------------------------------------------------------
M.avg               -      -       -       -        -    66.9%   63.6%   64.8%
------------------------------------------------------------------------------
m.avg            2623    346     543    2969     3166    88.3%   82.8%   85.5%
m.avg(no class)  2738    231     428    2969     3166    92.2%   86.5%   89.3%
```

Figure 2: Best score on the test data

We tried many other combinations as well but unfortunately we couldn't reach a better score with them. It only decreased it. Here some of scores that we obtained playing with some architectures and hyper-parameters:

Architecture/Parameter	Devel-Set	Test-Set
Basic	55.2%	53.2%
Decreasing out_dim of first input layer to 50 and the BiLSTM units to 32	53.2%	54.4%
Decreasing max_len to 100	51.9%	53.8%
Adding prefix of 2 input	53.9%	54.4%
Modifying prefix length to 4 input	57.2%	54.5%
Adding lemma input	57.6%	54.3%
Modifying the optimizer to RMSprop	55.6%	57.7%
Adding specific features input	65.4%	62.7%
Adding lowercase input	68.1%	62.6%
Doubling the dropout	66.1%	61.0%

As the built models are not deterministic, it's really difficult to say with certainty if some combinations are really better than the others if the score is really similar. As we can see from the table, a more complex model not always mean better results. We can for sure say that adding our specific features input had the most impact on the score and that's, in our opinion, thanks to the presence checking of drugs in external sources. Our best model isn't the most complex network we could build (we didn't include GloVe embeddings, the size of the embeddings and number of units isn't really high) but despite this we reach a really decent score. We wanted to play even more with the architectures (tanh and relu activation function) and GloVe but unfortunately the computations were too intensive for our computer and took way too much time. As we reached the score of 71.6, we decided not to spend too much time on trying more various hyper-parameters/architectures and decided to jump to work on the final lab session.

3 NN-based DDI

3.1 Architecture

We tried several different architecture approaches with the help of the suggested websites. As training needs sometimes a very long time, not all possible combinations can be tested. Therefore, we tried to maximize the results of our model by testing the following with consideration of the suggested models on the different websites: Use of only CNNs and of multiple CNN+MaxPool layers. Using only (Birectional) LSTM with different values for the dropout. Combining CNNs with LSTMs and also Dropout layers. Additionally, dense layers with different activation functions like *relu* and *tanh*. Moreover, we tested values for the adam optimizer, as we didn't see any superior optimizer suggested anywhere (only adadelta, but it was similar with longer training time). Furthermore, we had some difficulties with integrating a BERT model as first layers. But we managed to utilize the GloVe embedding matrix initialisation for the Embedding layer. Additionally, we examined the influence of trainable embeddings. This has the advantage, that randomly chosen unknown values of the GloVe embedding matrix can be optimized during training. Furthermore, we investigated different dimensions everywhere it was possible, e.g. for the embedding output, for the CNN kernel filters and for the units of the LSTM model. Moreover, we tested different batch sizes for fitting the model in combination with different numbers of epochs. We also tested different values for the *max_len* when creating indexes from training data.

3.2 Input information

As input information, we added information about PoS, lemmas and lc words concatenated to the input words. Hence, we tested different combinations in order to distinguish between their influence on the model. For this purpose, we needed to slightly change the codemaps.py and therefore we added it to the code section.

3.3 Code

Here we present the code for our best performing model. For this purpose, we depict the *build_network* function and all methods it calls.

```
#! /usr/bin/python3

import sys
import random
from contextlib import redirect_stdout
from tensorflow.keras import regularizers, Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Embedding, Dense, Dropout, Conv1D, MaxPool1D, Reshape, Concatenate
from tensorflow.keras.layers import concatenate, GlobalMaxPooling1D
from tensorflow.keras.optimizers import Adam
from dataset import *
from codemaps import *
import tensorflow as tf
import numpy as np

def build_network(idx) :
   # get sizes of words, lc words and PoS tags
   n_words = codes.get_n_words()
   n_labels = codes.get_n_labels()
   n_lc = codes.get_n_lc_words()
   n_pos = codes.get_n_pos()
```

```
# define maximum length of used words
max_len = codes.maxlen

# we adapted a suggested GloVe model to get initial embedding matrix
# from https://github.com/suriak/sentence-classification-cnn/blob/master/sentence%20classifier.py
def load_glove():
    """
    Load pre-trained glove vectors which is downloaded from http://nlp.stanford.edu/data/glove.6B.
    EMBED_DIR.
    :return: A dictionary with key as word and value as vectors.
    """
    embeddings_index = {}
    try:
            f = open('/content/glove.6B.100d.txt', encoding='utf-8')
    except FileNotFoundError:
            print("GloVe vectors missing. You can download from http://nlp.stanford.edu/data/glove
            sys.exit()
    for line in f:
        values = line.rstrip().rsplit(' ')
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
    f.close()
    print("\tNumber of Tokens from GloVe: %s" % len(embeddings_index))
    return embeddings_index

def glove_embedding_matrix(embeddings_index):
    """
    Creates an embedding matrix for all the words(vocab) in the training data. An embedding matrix
    (vocab * EMBEDDING_DIM), where each row represents a word in the training data and its correspo
    in GloVe.
    Any word that is not present in GloVe but present in training data is represented by a random
    between -0.25 and + 0.25
    :param embeddings_index: The dictionary of words and its corresponding vector representation.
    :return: A matrix of shape (vocab * EMBEDDING_DIM)
    """
    words_not_found = []
    vocab = len(codes.word_index)# + 1
    # embedding_matrix = np.zeros((vocab, self.EMBEDDING_DIM))
    embedding_matrix = np.random.uniform(-0.25, 0.25, size=(vocab, 100))  # 0.25 is chosen so
    # the unknown vectors have (approximately) same variance as pre-trained ones
    print("codes:")
    print(codes)
    for word, i in codes.word_index.items():
        if i >= vocab:
            continue
        embedding_vector = embeddings_index.get(word)
        if (embedding_vector is not None) and len(embedding_vector) > 0:
            embedding_matrix[i] = embedding_vector
        else:
                words_not_found.append(word)
        # print('Number of null word embeddings: %d' % np.sum(np.sum(embedding_matrix, axis=1) == 0))
        print("\tShape of embedding matrix: %s" % str(embedding_matrix.shape))
        print("\tNo. of words not found in GloVe: ", len(words_not_found))
        return embedding_matrix

# load downloaded GloVe
emb_ind = load_glove()
# Get the embedding matrix with matching size
embedding_matrix = glove_embedding_matrix(emb_ind)

# set the dimension for Embedding layers
out_dim = 100

# Define the Input-layers, which are afterwards concatenated
# Only lc words and PoS tags helped to improve the model
inptW = Input(shape=(max_len,))
embW = Embedding(input_dim=n_words, output_dim=out_dim,
   weights=[embedding_matrix],
   input_length=max_len, mask_zero=False,trainable=True)(inptW)
inptLC = Input(shape=(max_len,))
embLC = Embedding(input_dim=n_lc, output_dim=out_dim,
   input_length=max_len, mask_zero=False,trainable=True)(inptLC)
inptPos = Input(shape=(max_len,))
embPos = Embedding(input_dim=n_lemmas, output_dim=out_dim,
   input_length=max_len, mask_zero=False,trainable=True)(inptPos)

embs = concatenate([embW,embLC,embPos])

# Define the CNN and MaxPooling layers with different filter sizes
# As in https://towardsdatascience.com/cnn-sentiment-analysis-1d16b7c5a0e7
filter_sizes = [2,3,4,5,10]
convs = []
for filter_size in filter_sizes:
 l_conv = Conv1D(filters=100,
                 kernel_size=filter_size,
                 activation='relu')(embs)
 l_pool = MaxPool1D()(l_conv)
 convs.append(l_pool)
l_merge = Concatenate(axis=1)(convs)

# Define the LSTM layer
# Recurrent dropout set to 0.0, as otherwise GPU can't be utilized properly
lstm = Bidirectional(LSTM(units=50, return_sequences=False,
 dropout=0.3,recurrent_dropout=0.0))(l_merge)

# add a dropout layer
x = Dropout(0.2)(lstm)

# Define the output layer
```

```
   out = Dense(n_labels, activation='softmax')(x)
   # Define the model with matching input and output
   model = Model([inptW,inptLC,inptPos], out)
   # Use a dedicated optimizer to test parameters of learning rate
   adam = Adam(learning_rate=0.001)
   # compile the model and return it
   model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])
   return model

## --------- MAIN PROGRAM -----------
## --
## -- Usage:  train.py ../data/Train ../data/Devel  modelname
## --

## --------- MAIN PROGRAM -----------
## --
## -- Usage:  train.py ../data/Train ../data/Devel  modelname
## --

# directory with files to process
trainfile = sys.argv[1]
validationfile = sys.argv[2]
modelname = sys.argv[3]

# load train and validation data
traindata = Dataset(trainfile)
valdata = Dataset(validationfile)

# create indexes from training data
max_len = 150
codes = Codemaps(traindata, max_len)

# build network
model = build_network(codes)
with redirect_stdout(sys.stderr) :
   model.summary()

# encode datasets
Xt = codes.encode_words(traindata)
Yt = codes.encode_labels(traindata)
Xv = codes.encode_words(valdata)
Yv = codes.encode_labels(valdata)

# train model
with redirect_stdout(sys.stderr) :
   model.fit(Xt, Yt, batch_size=32, epochs=12, validation_data=(Xv,Yv), verbose=1)

# save model and indexs
model.save(modelname)
codes.save(modelname)
```

Now, the function changed in codemaps.py follows:

```
## --------- encode X from given data -----------
def encode_words(self, data) :
    # encode and pad sentence words
    Xw = self.__encode_and_pad(data, self.word_index, 'form')
    # encode and pad sentence lc_words
    Xlw = self.__encode_and_pad(data, self.lc_word_index, 'lc_form')
    # encode and pad lemmas
    Xl = self.__encode_and_pad(data, self.lemma_index, 'lemma')
    # encode and pad PoS
    Xp = self.__encode_and_pad(data, self.pos_index, 'pos')
    # return encoded sequences
    # return [Xw,Xlw,Xl,Xp] (or just the subset expected by the NN inputs)
    return [Xw,Xlw,Xp]
```

3.4 Experiments and results

In this section we describe the results of our experiments. Important is that the usage of neural networks can produce very different results. Because of that, we ran the model - for good working architectures and parameters - multiple times in order to get averaged results. Due to limited resources and very long training time, we decided to conduct every experiment three times, in order to test as much as possible. As results in our case vary up to 7 or 8%, it would be necessary to run experiments more often than three times in order to get more meaningful results.
At first we want to comment on less effective parameters, which did not really help to improve the results (or it was not visible by conducting only three experiments). For example changing the batch size in order to have faster training did decrease the results, even when increasing the number of epochs. Furthermore, adding dense layers with activation functions *relu* and *tanh* didn't increase the performance. Moreover, changing the embedding dimensions for word embeddings did not help.
The following changes improved the performance. We tested different combinations of CNN, LSTM and CNN+LSTM. Here, CNN is a concatenation of multiple CNN and Max Pooling layers with different kernel sizes. Both CNN and LSTM deliver good results, but the combination of a stacked CNN + a Bidirectional LSTM performed best. We got the best result by combining multiple layers of CNN and MaxPooling with different filters and a bidirectional LSTM with only 50 units. Furthermore we utilized additional input concatenated to the words, namely the lc words and PoS tags. Moreover, we were using a GloVe embedding matrix for initialisation of the Embedding. Here, we allowed the embedding to be updated while training. We did this, as over 1200 words of the training data were not found in GloVe and so these vectors were set randomly by GloVe (in a range between -0.25 and 0.25). To see the influence of most important changes, see the table below:

Architecture/Parameter	Devel-Set	Test-Set
CNN	60.5%	49.2%
LSTM	61.2%	54.3%
CNN+Concat inputs	64.1%	50.2%
LSTM+Concat inputs	63.0%	59.3%
CNN+Concat inputs+GloVe(trainable)	63.1%	51.3%
LSTM+Concat inputs+GloVe(trainable)	64.2%	56.4%
CNN+LSTM+Concat inputs+GloVe(trainable)	64.2%	63.0%

As we can see in the table, more and more complex networks helped to slightly increase the results on the Devel set. But when we examine the test set values, we can see that some models might overfit due to higher complexity.

We were not able to use recurrent dropout for the LSTM, as otherwise training time exploded for training, as the GPU can't be utilized properly (see https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM). As already mentioned, there is another caveat in these experiments. Only 3 models per experiment setup were trained, so the averaged values in the table above have to be considered carefully, because model performance between the three experiments varied up to 8%. However, we think that our results give interesting insights and can lead into the general direction of a good performing model. Also, the best of the three runs of our best model was able to achieve 69.3%.

4 Conclusions

With the neural network setting, the results have to be interpreteted always with caution, as through the non-deterministic behavior the results for single runs can vary heavily. However, we reached overall decent results. We tried a lot of different things to see the impact on our model. Here, we saw that it is again very important to not only test on devel but also on the test set, as nn-based algorithms can tend to overfit if they are too complex. For both tasks, the evaluating of model performance is different, as in the first task not the most complex model was the best. But at all, even when it was not possible to test every possible parameter/architecture combination, we reached the expected F1-Scores on our models. Of course, there's still a possibility to improve even more the models to obtain a better F1-Score (by tuning even more the inputs/architectures) but we are happy with the scores we reached and we considered that it's more interesting to experience different combinations and their advantages/disadvantages instead of focusing to reach the best possible score with only few tried combinations.