Advanced Human Languages Technologies
NN-based NERC and DDI

Denaldo Lapi and Francesco Aristei

November 5, 2022

1 Introduction

This report contains some possible solutions based on Neural Network (NN) architectures of the tasks 9.1 and 9.2 of the *SemEval-2013 challenge*[1], concerning respectively the named entity recognition and classification of drugs (NERC), and the detection and classification of drug-drug interactions between pairs of drugs (DDI).

We were provided with a dataset consisting of XML files containing sentences and entities representing various types of classes (drug, brand, group and drug_n), together with the type of interactions between entities (mechanism, effect, advise, interaction). The data is already splitted in three subsets (folders): Train, Devel and Test.

We were also provided with evaluation scripts to compute metrics like precision, recall and F1 score.

The aim of this report is to explain how we used NN architectures to solve the 2 tasks, describing the main aspects, architectures and decisions taken in order to optimize the performance in each task.

2 NN-based NERC

The NN approach to solve the NERC task consists on the following main steps:

- Properly encode training and validation data to fed as input to a NN
- Train a NN model on the Training data with the obtained input encoding
- Perform hyperparameter selection by using the Devel dataset
- Evaluate the final obtained model on the Test dataset

[1] Segura-Bedmar, Isabel. "Semeval-2013 task 9: Extraction of drug-drug interactions from biomedical texts (ddiextraction 2013)." Proceedings of the 7th International Workshop on Semantic Evaluation (SemEval 2013). 341-350.

2.1 Architecture

At the beginning we focused our attention on trying several NN architectures for the NERC task, without touching the already provided input encoding which takes into account words and suffixes. We evaluated the performance of each architecture on the Devel dataset by focusing on the F1 score. Starting from the initially provided architecture, which gives a F1 score of 55%, we tried the following architectures, in order:

- We removed the recurrent dropout from the bilstm and used instead a dropout value of 0.2. The reason for this choice is due to the fact that the training of a bilstm layer with recurrent dropout doesn't allow to exploit the GPU for training. Therefore, in all our experiments we used the dropout instead of the recurrent dropout. This increased the F1 score to 55.5%.

- We tried several values for the embedding dimension of the words: 150, 200, 100. No improvement was obtained.

- We then modified the baseline model by adding an additional biltsm layer, in particular we used a stack of 2 bilstm of 200 units each and this allowed us to push the F1 score to 63.5%

- We then tried to optimize the hyperparameters of the architecture described above by trying the following:

  - change number of units of each bilstm layer: 250, 150, 300, 100
  - several dropout values for each bilstm: 0.1, 0.2, 0.3, 0.4, 0.5
  - change *max_len* parameter representing the length of each input sequence (initially set to 150): tried values of 100 and 200

  But no one of the above allowed us to further improve the F1 score.

- We then decided to add an additional dense layer after the stack of bilstm. We tried to tune the parameters of this layer, i.e. number of units, type of activation function and we also tried to add a dropout layer after it. This layer didn't allowed us to obtain any improvement.

  As a further step, we also tried to add a residual connection (skip connection) between this dense layer and the first bilstm layer, but still no improvement was obtained, we therefore removed this additional dense layer.

- We modified our architecture by adding a third bilstm to the stack of bilstms after the concatenation of the embeddings: no improvement was obtained.

- We tried to add the Early Stopping callback, useful to prevent overfitting: the callback simply monitors the validation loss during training and stops the training early when the loss stops to decrease. We tried several values for the *patience* parameter and we also tried to monitor the *validation accuracy*, stopping the training when it stops to increase. Early Stopping didn't give any improvement in our model's performance in terms of F1 score on the Devel dataset.

- We also tried another callback for reducing the learning rate during training whenever the validation loss stops improving. This callback was tried together with the previous one and also alone, but no improvement was seen on the F1 score.

- We then decided to try another loss function instead of the sparse categorical crossentropy: we tried the sparse categorical focal loss function[2], by trying several values for the *gamma* parameter (1, 0.5, 1.5, 2.5), but no improvement was seen.
- We then tried to replace the second bilstm of the stack with a simple LSTM (of 200 units), but this didn't gave any improvement.
- we tried the *Adamax* optimizer which, as suggested also by Keras[3], 'is sometimes superior to adam, specially in models with embeddings'. We tried it with several values of the learning rate and beta parameters, but we didn't obtain improvements.
- In a next step, we tried to change the sizes of the words and suffixes embeddings together by trying several combinations of values, but the best result is the one obtained with a size of 100 for words and 50 for suffixes.
- We also tried to change the batch size, moving it from 16 to 32, but this worsened the Devel F1 score.
- As a following step, we decided to use a pre-trained *GloVe* embedding[4], by trying pre-trained models with different embedding dimensions (200, 300, 50). We tried also to modify the bilstm layers parameters when using GloVe (we tried the GloVe embedding with 6B tokens), but we were not able to obtain any kind of improvement and therefore we discarded the option of using a pre-trained embedding.

2.2 Input Information

After trying in several ways to improve our model's performance by considering the already provided input information containing word and suffixes, we decided to work more on the input information to provide to our model and to how to properly encode and format it.

The model we used to test these new input information was the one composed by a stack of 2 bilstm layers of 200 units each. Starting from an initial F1 score of around 63%, we tried to modify the architecture by taking into account additional input information, besides words and suffixes:

- As a first trial, we decided to create an additional feature vector to provide as input to our model (see *codemaps.py*). We started by using a binary vector encoding the presence or absence of a specific feature in each word of a sentence. The features we considered were taken from the *Assignment 1* and they encode whether a word is a stop-word, whether the word contains only characters from the alphabet, if it is in upper-case, if it contains a dash, if it contains only digits, if it contains any digit, if it contains both upper-case and lower-case characters. We also added to this vector of features a further feature encoding the length of the word.

  This vector of features was fed to the NN into an additional input layer, which was concatenated to the embeddings of the words and suffixes before being fed to the bilstm. This additional input information allowed us to reach a F1 score of 66.6% on the Devel dataset, which was obtained after increasing the dropout of the bilstm layers to 0.3

[2] https://focal-loss.readthedocs.io/en/latest/generated/focal_loss.SparseCategoricalFocalLoss.html
[3] https://keras.io/api/optimizers/adamax/
[4] https://nlp.stanford.edu/projects/glove/

- We then decided to use also the external information contained in the provided *HSDB.txt* and *DrugBank.txt* files. We extended our feature vector containing features for each word by adding 3 additional binary features indicating whether the word is present in the external files and the type of drug represented by the word. This allowed to further boost the performance of our model to 67.4% of F1 score on the Devel set.

- With the new added input information, we slightly changed our NN architecture, by considering only 1 bilstm layer of 200 units, and this pushed the performance to 68.3% of F1.

- We then added as an additional input the lower-case words, using an additional embedding of size 100 for them in our architecture. The 3 embeddings (suffixes, word and lower-case word) and the feature vector were concatenated before being fed to the bilstm. The best performance, in this case, was obtained with a stack of 2 bilstm of 200 units each and with a dropout value of 0.3 for each bilstm, which allowed to reach an F1 score of 71.4% on the Devel set.

- We then used as additional input information the POS tag of each word (after indexing it as done for words, suffixes and lower-case words). We used an additional input layer in our NN model and we used an embedding also for the POS tags, by trying several embedding sizes. However, this additional information didn't gave any improvement.

- Then we tried to fed the Neural Network using more vectors encoding the information of the suffixes, using different lengths. We continued with this approach, focusing instead on the prefixes. So we created different vectors encoding the information on the prefixes of the words, varying the length of such prefixes. We tried several configurations, having only one vector of suffixes and one vector of prefixes for each word, or using two or more vector of suffixes and no vector of prefixes and viceversa.

  What we observed is that adding more features encoding different information regarding the prefixes is not beneficial for the training of the NN. Indeed, the best performances with our model defined above (stack of 2 bilstm), have been obtained using two vectors encoding the information regarding the suffixes, with a length equal to 4 and 7. The performances of the model with this feature vector, together with the binary features and the lower-case words defined above, reached an F1 score of 71.9% on the Devel dataset. Adding more information regarding the prefixes, deteriorated the performances significantly. For example, we tried to add more feature vectors encoding suffixes of length 5 and 7 together with prefixes by using the same lengths (5 and 7), but we observed a drop in the performance to 67.8% on the Devel dataset.

- The next step we performed was trying to take into account more information regarding the words, like the Lemma to which the words belong. Therefore, we added for each word a vector representing the lemma (we indexed in the same way we did for words, suffixes and lower-case words). In order to extract the Lemma for a given word, we decided to rely on the *WordNetLemmatizer*, provided by the nltk.stem, using a simple method that, taken the word, returns its lemmatized form (see *codemaps.py*). However the performances didn't really improved with respect to the best performing model defined above.

- Another approach we tried to follow has been to modify the way the POS tags features are encoded for each word. As described above, initially, the way we encoded the POS tags was the same performed for words, suffixes and so on. In this case, we tried to apply the same approach defined above for the features extracted from the *Assignment 1.* Specifically,

using the *nltk.help.upenn_tagset()* we listed all the possible POS tags defined in the *Penn Treebank Tag Set* which is the corpus used by the *nltk.pos_tags()* function. After that, we proceeded in the same way as seen in the binary-encoding described previously. Once defined this binary-encoding for the POS tags, we fed this vector of features to the NN using an additional input layer, which was concatenated to the embedding of the normal features (words, lower_case_words, lemma etc) together with the binary ones before being fed to the bilstm.

2.3 Code

We include here the code of the *build_network* function, which defines the best model we decided to use, together with the methods this function calls. Specifically, we include the *encode_words()* method of the *codemaps.py* file, together with the initialization needed to perform the encodings.

```
# rename packages
tfk = tf.keras
tfkl = tf.keras.layers

def build_network(codes) :

   # sizes
    n_words = codes.get_n_words()
    n_lc_words = codes.get_n_lc_words()
    n_sufs1  = codes.get_n_sufs1()
    n_sufs1  = codes.get_n_sufs2()
    n_labels = codes.get_n_labels()
    max_len = codes.maxlen  # 150

    inptW = tfk.Input(shape=(max_len,)) # word input layer & embeddings
    embW = tfkl.Embedding(input_dim=n_words, output_dim=100,
                    input_length=max_len, mask_zero=True)(inptW)

    inptLC = tfk.Input(shape=(max_len,)) # lc_word input layer & embeddings
    embLC = tfkl.Embedding(input_dim=n_lc_words, output_dim=100,
                    input_length=max_len, mask_zero=True)(inptLC)

    inptS1 = tfk.Input(shape=(max_len,))  # suf input layer & embeddings
    embS1 = tfkl.Embedding(input_dim=n_sufs1, output_dim=50,
                    input_length=max_len, mask_zero=True)(inptS1)

    inptS2 = tfk.Input(shape=(max_len,))  # suf input layer & embeddings
    embS2 = tfkl.Embedding(input_dim=n_sufs2, output_dim=50,
                    input_length=max_len, mask_zero=True)(inptS2)

    # dropout applied to each embedding
    dropW = tfkl.Dropout(0.2)(embW)
    dropLC = tfkl.Dropout(0.2)(embLC)
    dropS1 = tfkl.Dropout(0.2)(embS1)
    dropS2 = tfkl.Dropout(0.2)(embS2)

    # additional input layer for numerical features
    input_3 = tfk.Input(shape=(max_len, 11))
```

```
39
40      # concatenate embeddings
41      drops = tfkl.Concatenate(axis=2)([dropW, dropLC, dropS1, dropS2, input_3])
42
43      # stack of biLSTMs
44      bilstm1 = tfkl.Bidirectional(tfkl.LSTM(units=200, return_sequences=True,
45                                  dropout=0.3))(drops)
46
47      bilstm2 = tfkl.Bidirectional(tfkl.LSTM(units=200, return_sequences=True,
48                                  dropout=0.3))(bilstm1)
49
50      # output softmax layer
51      out = tfkl.TimeDistributed(tfkl.Dense(n_labels, activation="softmax"))(bilstm2)
52
53      # build and compile model
54      model = tfk.models.Model([inptW, inptLC, inptS1, inptS2, input_3], out)
55      model.compile(optimizer="adam",
56                    loss="sparse_categorical_crossentropy",
57                    metrics=["accuracy"])
58      return
```

Following the *encode_words()* method:

```
1       ## --------- encode X from given data -----------
2       def encode_words(self, data) :
3
4           # encode and pad sentence words
5           Xw = [[self.word_index[w['form']] if w['form'] in self.word_index else
            ↪  self.word_index['UNK'] for w in s] for s in data.sentences()]
6           Xw = pad_sequences(maxlen=self.maxlen, sequences=Xw, padding="post",
            ↪  value=self.word_index['PAD'])
7
8           # encode and pad sentence lc_words
9           # the dictionary self.lc_index is built in the same way as self.word_index
10          Xlc = [[self.lc_index[w['lc_form']] if w['lc_form'] in self.lc_index else
            ↪  self.lc_index['UNK'] for w in s] for s in data.sentences()]
11          Xlc = pad_sequences(maxlen=self.maxlen, sequences=Xlc, padding="post",
            ↪  value=self.lc_index['PAD'])
12
13
14          # encode and pad suffixes
15          # the dictionary self.suf_index1 is built in the same way as
            ↪  self.word_index: suffixes are keys and values are the obtained
            ↪  indicization
16          Xs1 = [[self.suf_index1[w['lc_form'][-self.suflen1:]] if
            ↪  w['lc_form'][-self.suflen1:] in self.suf_index1 else
            ↪  self.suf_index1['UNK'] for w in s] for s in data.sentences()]
17          Xs1 = pad_sequences(maxlen=self.maxlen, sequences=Xs1, padding="post",
            ↪  value=self.suf_index1['PAD'])
18
19
20          # encode and pad suffixes (with different length)
```

```
        Xs2 = [[self.suf_index2[w['lc_form'][-self.suflen2:]] if w['lc_form'][-self.suflen2:] in self.suf_index2 else self.suf_index2['UNK'] for w in s] for s in data.sentences()]
        Xs2 = pad_sequences(maxlen=self.maxlen, sequences=Xs2, padding="post", value=self.suf_index2['PAD'])


        # encode and pad prefixes
        Xp1 = [[self.pref_index1[w['lc_form'][:self.preflen1]] if w['lc_form'][:self.preflen1] in self.pref_index1 else self.pref_index1['UNK'] for w in s] for s in data.sentences()]
        Xp1 = pad_sequences(maxlen=self.maxlen, sequences=Xp1, padding="post", value=self.pref_index1['PAD'])


        # encode and pad prefixes with different length
        Xp2 = [[self.pref_index2[w['lc_form'][:self.preflen2]] if w['lc_form'][:self.preflen2] in self.pref_index2 else self.pref_index2['UNK'] for w in s] for s in data.sentences()]
        Xp2 = pad_sequences(maxlen=self.maxlen, sequences=Xp2, padding="post", value=self.pref_index2['PAD'])


        # encode and pad pos tags
        Xpos = [[self.pos_index[self.pos_tag(w['form'])] if self.pos_tag(w['form']) in self.pos_index else self.pos_index['UNK'] for w in s] for s in data.sentences()]
        Xpos = pad_sequences(maxlen=self.maxlen, sequences=Xpos, padding="post", value=self.pos_index['PAD'])

        # encode and pad lemmas
        Xl = [[self.lemma_index[self.lemma(w['form'])] if self.lemma(w['form']) in self.lemma_index else self.lemma_index['UNK'] for w in s] for s in data.sentences()]
        Xl = pad_sequences(maxlen=self.maxlen, sequences=Xl, padding="post", value=self.lemma_index['PAD'])

        # one-hot encoding of POS
        pos_list = ["\$", "''", "(", ")", "," , "--", "." , ":", "CC", "CD", "DT", "EX", "FW", "IN", "JJ", "JJR", "JJS", "LS", "MD", "NN", "NNP", "NNPS", "NNS", "PDT", "POS","PRP",          "PRP\$", "RB", "RBR", "RBS", "RP", "SYM", "TO", "UH", "VB", "VBD", "VBG", "VBN", "VBP", "VBZ", "WDT", "WP", "WP\$", "WRB", "``"]
        X_pos_features = []

        # build pos one-hot encoding
        for s in data.sentences():
            sentence_pos = []
            for w in s:
                f = self.pos_tag(w['form'])
                feat_length = 0
                word_pos_features = []
```

```
54
55                  for pos in pos_list:
56                      word_pos_features.append(1 if pos == f else 0)
57                      feat_length += 1
58
59                  sentence_pos.append(word_pos_features)
60                  if len(sentence_pos) >=self.maxlen:
61                      break
62
63              # padding with zeros
64              while len(sentence_pos) < self.maxlen:
65                  sentence_pos.append([0 for i in range(feat_length)])
66
67              X_pos_features.append(sentence_pos)
68
69
70          # build feature vector (features from Assignment 1)
71          X_features = []
72
73          for s in data.sentences():
74
75              # list of features for each word of the sentence
76              sentence_features = []
77              for w in s:
78                  f = w['form']
79                  feat_length = 0     # length of the feature list of each word
80                  word_features = []      # features of each word
81
82                  # is stopword
83                  word_features.append(1 if f.lower() in stop_words else 0)
84                  feat_length +=1
85
86                  # is alpha
87                  word_features.append(1 if f.isalpha() else 0)
88                  feat_length +=1
89
90                  # is upper
91                  word_features.append(1 if f.isupper() else 0)
92                  feat_length +=1
93
94                  # dash
95                  word_features.append(1 if '-' in f else 0)
96                  feat_length +=1
97
98                  # contains any digit
99                  word_features.append(1 if any(char.isdigit() for char in f) else 0)
100                 feat_length +=1
101
102                 # contains only digits
103                 word_features.append(1 if all(char.isdigit() for char in f) else 0)
104                 feat_length +=1
105
```

```
                # is upper and lower
                word_features.append(1 if lower.match(f) and upper.match(f) else 0)
                feat_length +=1

                # length
                word_features.append(len(f))
                feat_length +=1

                # external resources: we have only the types 'drug', 'group',
                ↪  'brand'
                word_features.append(1 if f.lower() in external and
                ↪  external[f.lower()] == 'drug' else 0)
                word_features.append(1 if f.lower() in external and
                ↪  external[f.lower()] == 'group' else 0)
                word_features.append(1 if f.lower() in external and
                ↪  external[f.lower()] == 'brand' else 0)
                feat_length +=3


                sentence_features.append(word_features)

                # check max len
                if len(sentence_features) >=self.maxlen:
                    break

            # padding with zeros
            while len(sentence_features) < self.maxlen:
                sentence_features.append([0 for i in range(feat_length)])

            X_features.append(sentence_features)

        # return encoded sequences
        return [Xw, Xlc, Xs1, Xs2, np.array(X_features)]


    ## -------- get word index size ---------
    def get_n_words(self) :
        return len(self.word_index)

    ## -------- get lc_word index size ---------
    def get_n_lc_words(self) :
        return len(self.lc_index)

    ## -------- get suf1 index size ---------
    def get_n_sufs1(self) :
        return len(self.suf_index1)

    ## -------- get suf2 index size ---------
    def get_n_sufs2(self) :
        return len(self.suf_index2)

    ## -------- get label index size ---------
```

```
    def get_n_labels(self) :
        return len(self.label_index)
```

Finally, additional code, needed to understand some of the encodings defined above, like the one using external information extracted from the *DrugBank.txt* or the one using the lemma and pos _tag of the words.

```
# instantiate lemmatizer
lemmatizer = WordNetLemmatizer()

# dictionary containing information from external knowledge resources
external = {}

# build the dictionary
path = "TaskData/resources/HSDB.txt" # read the HSDB file
with open(path, encoding="utf8") as h :
    for x in h.readlines() :
        external[x.strip().lower()] = "drug" # add drug into the dictionary

path = "TaskData/resources/DrugBank.txt" # read the DrugBank file
with open(path, encoding="utf8") as h :
    for x in h.readlines() :
        (n,t) = x.strip().lower().split("|")
        external[n] = t # add drug into the dictionary with the corresponding type

# obtain the list of english stopwords
stop_words = stopwords.words('english')

# build regular expression patterns for lower and upper chars
lower = re.compile(r'.*[a-z]+')
upper = re.compile(r'.*[A-Z]+')

# auxiliary function to get POS tags
def pos_tag(self, tkn):
  pos_list = pos_tag(list(tkn)) # it returns a list made by one element which is a
  ↪  tuple [(word, pos)]
  pos = pos_list[0][1] # accessing the pos inside the tuple inside the list
  return pos

# auxiliary function to lemmatize
def lemma(self, tkn):
  lemma = lemmatizer.lemmatize(tkn)
  return lemma
```

2.4 Experiments and Results

In this section, we present the best results obtained for what regard our best performing model, with the combination of input features that improve the most the training of the NN. Specifically, we report in Figure 1 the statistics obtained from this model on the Devel and Test datasets.

What we can observe from the performances obtained in the *test dataset* is that the model is not overfitting the Devel, therefore it has good generalization capabilities. Besides that, the model has also a pretty high Recall on the Test dataset. An interesting observation is the low

	tp	fp	fn	#pred	#exp	P	R	F1
brand	301	14	73	315	374	95.6%	80.5%	87.4%
drug	1654	104	252	1758	1906	94.1%	86.8%	90.3%
drug_n	10	23	35	33	45	30.3%	22.2%	25.6%
group	576	102	111	678	687	85.0%	83.8%	84.4%
M.avg	-	-	-	-	-	76.2%	68.3%	71.9%
m.avg	2541	243	471	2784	3012	91.3%	84.4%	87.7%
m.avg(no class)	2601	183	411	2784	3012	93.4%	86.4%	89.8%

	tp	fp	fn	#pred	#exp	P	R	F1
brand	248	34	26	282	274	87.9%	90.5%	89.2%
drug	1749	69	378	1818	2127	96.2%	82.2%	88.7%
drug_n	24	204	48	228	72	10.5%	33.3%	16.0%
group	591	144	102	735	693	80.4%	85.3%	82.8%
M.avg	-	-	-	-	-	68.8%	72.8%	69.2%
m.avg	2612	451	554	3063	3166	85.3%	82.5%	83.9%
m.avg(no class)	2801	262	365	3063	3166	91.4%	88.5%	89.9%

Figure 1: Best NN model performances on the Devel set (on the left) and on the Test set (on the right

F1 score regarding the 'drug_n' class: this is due to the fact that there are very few samples belonging to that group, and this makes it difficult for our NN classifier to learn to identify samples in the class. Moreover, the drop in F1 score between Devel and Test is pretty high for the 'drug_n' class (almost 10%), while for the other groups the performance is above 80% of F1 both in Devel and in Test with no significant drop.

3 NN-based DDI

The NN approach to solve the DDI task consists on the same steps seen for the NERC task, except that in this case we have 1 single output for each sentence and pair of entities, while in the NERC we have 1 output for each token. Again, we tried several architectures and input information and we evaluated the results on the Devel dataset.

3.1 Architecture

As a first step, we focused our attention on trying several NN architectures for the DDI task, without modifying the already provided input encoding which takes into account only the words properly encoded. We evaluated the performance of each architecture on the Devel dataset by focusing on the F1 score. Starting from the initially provided architecture which gives a F1 score of 50%, we tried the following architectures, in order:

- We modified the initially provided architecture by increasing the size of the word embeddings. A value of 150 gave an F1 of 50.6%, while a value of 200 gave us an F1 score of 54.1%. We tried to increase the embedding size even more, but the best result was the one obtained with an embedding size of 200.
- We then tried to modify the parameters of the convolutional (conv) layer, in particular we tried several numbers of filters, by changing also the embedding size in order to find the best configuration. We obtained the best result by using 100 filters and a word embedding size of 200, which reached an F1 score of 56.6% on the Devel.
- We then tried to add some dropout layers, useful to reduce the overfitting issue, and we tried to add it after the word embedding and after the flatten layer, but that didn't improve the performance.
- We decided to add additional dense layers after the flatten layer, but no improvement was seen.
- Then, we tuned the stride and the kernel size of the conv layer, but the best performance was still obtained with the already provided values of size=2 and stride=1.

- We then replaced the flatten layer with a global average pooling layer (GAP), but this didn't gave any improvement, even after adding other dense layers between the gap and the output layer.
- We then tried to use a stack of 2 conv layers followed by a flatten layer or by a GAP layer, and eventually also by an additional dense layer. No improvement was obtained.
- We changed the batch size, trying the values of 64 and 16, but no improvement was obtained, therefore we continued to use the size of 32.
- We replaced the flatten layer with a max pooling layer, but this didn't work, even after adding an additional dense layer after the pooling.
- Then, we tried to change optimizer, by using Adamax, SGD and Adadelta with different values of their parameters, but no improvement was seen.
- We continued to try other conv1D-based architectures: in particular we built some models based on an architecture composed by a stack of (Conv1D + MaxPool) blocks followed by a GAP layer, and then a dense classifier before the output layer, which is a typical architecture in convolutional neural networks:
  - by using an architecture of the type *Conv1D + MaxPool + Conv1D + GAP + Dropout + Dense + Output dense* after the embedding, we obtained a good improvement of the F1 score which reached a value of 58%, obtained after trying several parameters for the conv layers, i.e. different number of filters, filter sizes and strides (best result obtained with 100 filters for each conv, kernel size set to 2, and stride set to 1).
  - We then tried to add an additional block composed by Conv1D + MaxPool before the final Conv1D , but we didn't improve the performance, while, after removing the additional Dense layer after the GAP we reached an F1 score of 60%. Again the result was obtained after trying several filter sizes, strides, number of neurons in the dense layers. We also tried to use an additional dropout layer after the dense, but this didn't gave any improvement.
- We then moved into architectures composed by LSTM layers instead of convolutions. We started by using simple LSTM (not bidirectional) stacked one on top of the other (with *return_sequences* parameter set to True except for the last LSTM and followed by the output layer but we didn't obtain any improvement, even after trying several parameters and number of LSTM layers.
- As a subsequent step, we substituted the LSTMs with bidirectional LSTMs (bilstm), which are much more useful in a case in which the entire input sequence is available, since they allow to capture both the left and the right context of a word. Starting from the initially provided architecture, characterized by a conv layer followed by a flatten and then by the output layer, we simply substituted the conv and the flatten with a bilstm. In this phase, we tried several parameters for that layer (number of units, dropout) and we also tried with different embedding dimensions. The best result was obtained with a bilstm of 200 units with a dropout of 0.2 and with an embedding size of 200, which allowed us to reach an F1 score of 62.4%. We also tried to add an additional dense layer before the final output layer, but still we had no improvement.
- We then tried to use a stack of 2 bilstms in sequence, but this kind of architecture didn't improved the F1.

- From the training of these last models based on bidirectional layers, we saw that the validation accuracy was continuing to improve through the training epochs. Therefore, we decided to increase the number of epochs to train the model to 20, and we used the Early Stopping callback, monitoring the validation accuracy and stopping the training whenever the validation accuracy stopped to increase. We set the *patience* parameter to 5 epochs and the *restore_best_weights* to True. We then tried several architectures based on bilstms, by changing the number of bilstms, their units, dropout, by adding an additional dense layer at the end. The best result was obtained with a stack of 2 bilstms with 100 units a dropout of 0.2 each, followed by an additional dense layer of 100 units with 'relu' activation function. This model reached an F1 score on the Devel dataset of 64.9%. So, we decided to keep the Early Stopping callback for the next architectures we tried.

- We then moved into architectures combining convolutional layers with bilstm layers. Again, starting from the initial provided network composed by only a conv and a flatten layer, we replaced them with bilstms and convolutional layers. At first, we tried models composed by bilstms followed by conv layers:

  - we tried at first a model that adds the following layers after the word embeddings: *bilstm (units=100, dropout=0.2) + Conv1D + MaxPool + Conv1D + GAP + Dense(100, 'relu') + Dense(50, 'relu') + output dense.* We used conv layers with 100 filters each, kernel size of 2 and stride set to 1.

    This model reached an F1 score of 62.9%.

  - A better performance was obtained by adding an additional (Conv1D + MaxPool) block and obtaining the following architecture (after the embedding): *bilstm(units=100, dropout=0.2) + Conv1D(filters=100) + MaxPool + Conv1D(units=200) + MaxPool + Conv1D(units=300) + GAP + Dense (100) + output.*

    In this case, we obtained our best performance by using a kernel size of 6, with a stride of 2. The obtained F1 score is 64.0%

  - We tried to modify the above architecture by adding a stack of bilstms before the conv layers, and we reached and F1 score of 64.1% with the following architecture: *2 bilstms (100 units and 0.2 dropout each) + Conv1D(filters=100, size=2, stride=1)+ MaxPool + Conv1D(filters=200, size=2, stride=1) + MaxPool+ Conv1D(filters=300, size=2, stride=1) + GAP + dense(units=80, activation='relu') + output layer.*

  - We tried to simplify the previous model, in order to reduce the number of parameters and, after several trials, we were able to improve the performance of our model after adding a dropout of 0.2 to the word embedding layer, and by attaching the following layers to the embedding dropout: *bilstm(units=100, dropout=0.2) + Conv1D(units=100, size=2, stride=1) + GAP + output.*

    This model reached and F1 score of 65.1% on the Devel.

- With the model built by composing bilstm and conv layers, we decided to use the pre-trained *GloVe* embedding for the words embedding layer. At first, we downloaded and used the pre-trained GloVe with 6B tokens and we build an embedding matrix for the words appearing in our dataset. This matrix was then used to initialize our word embedding Keras layer.

  At the beginning we used the *trainable* parameter set to False and we didn't obtain any improvement on the F1 score. Then, we set the *trainable* parameter to True and we obtained a final F1 score of 67.1% on the Devel dataset, by using an embedding size of 200. Then, we repeated the same procedure by using the pre-trained GloVe with an embedding

size of 100, but the performance worsened. While, with the pre-trained GloVe with an embedding size of 300 (with *trainable* set to True and *mask* set to False), we obtained a very nice improvement of the F1 score which reached a value of 70.1% on the Devel dataset. We tried this model also on the Test dataset to understand whether we were overfitting on the Devel, and we obtained a score of around 65% F1 on the Test, which is a quite good result.

- As a further step, we downloaded the GloVe embedding with 42B tokens, since a lot of words in our dataset were not already present in the GloVe 6B. We repeated the training with the same parameters as before and we obtained 70.8% F1 score on the Devel, but less that 60% F1 score on the Test. Therefore, being the drop in performance larger than 10% between Devel and Test in this last case, we decided to use the GloVe 6B embedding, with an embedding size of 300.

- We then tried the architecture described in `https://www.kaggle.com/code/hamishdickson/cnn-for-sentence-classification-by-yoon-kim/notebook`, which is composed by 2D convolutions. We tried the model by using the GloVe embedding for words explained before, and by trying with different parameters, i.e. with different numbers of convolutional layers, by trying the Adadelta optimizer, but the model obtained very poor results on our task.

- We tried also the model described in `https://github.com/suriak/sentence-classification-cnn/blob/master/sentence%20classifier.py`, which reached an F1 score of 60% on the Devel, still pretty low with respect to our best model so far.

- As a next step, we decided to try another architecture, characterized by convolutional layers followed by bilstms, i.e. the so-called CNN-LSTM architecture, where the initial CNN is used for feature extraction on the input data, while the LSTM allows to perform the classification, by building a state upon the extracted features. Again, by starting from the initially provided model composed by a convolutional layer following the word embedding, we replaced it with a network with the following structure (after the word embeddings): *Conv1D(units=100, size=2, stride=1) + MaxPool + bilstm(units=100, dropout=0.1)*. We didn't obtain any improvement.

- We tried to work more on the previous architecture, by trying to use a stack of (Conv1D + MP) before the bilstm, with increasing number of filters in the conv layers (32, 64, 128, ...). This didn't improve our best F1 score, even after trying with several values for the strides, kernel sizes and number of kernels.

- We also tried to add a stack of bilstms after the convolutional layers, but no improvement was seen. Our best model with this kind of architecture is composed by 1 initial Conv1D layer with 100 filters with size 2 and stride set to 1, followed by a stack of 2 bilstms with 100 units and 0.1 dropout each: this model reached 67% F1 score on the devel.

- As a final step, we decided to build a model based on the attention mechanism, build upon the states of a bilstm layer. The model was build according to the structure shown in `https://colab.research.google.com/drive/16hleozlJZb00nX2Lyq8XKyg9ud0bY98N?usp=sharing#scrollTo=createclass`. This model allowed us to reach a very nice F1 score of 74.9% on the Devel dataset but, when trying on the Test data, its score was of only 55%. We therefore tried several values of the model's parameters, by changing the number of attention layers, the parameters of the bilstm, but we always observed a significant drop between the Devel and the Test performances.

After performing this trials, we took our best performing model so far (the one composed simply by a bilstm followed by a conv and a GAP before the final output layer) and we applied the additional input information described above, by adding the information regarding the lower-case words, prefixes, suffixes, POS and so on. The best combination of input information with this model is described in the following.

3.2 Input Information

Also in this task, after trying several ways to improve our model performance by considering the already provided input, we decided to work more on the input information to provide to our model and to how to properly encode and format it. The model we used was the one composed by a stack of 2 bilstm layers of 100 units each. Starting from an initial F1 score of around 63.6%, we tried to modify the architecture by taking into account additional input information besides words and lower case words.

- At first, we tried to fed the NN taking into account also the information regarding the prefixes. Just using a feature vector encoding the prefixes of length 5 for each word, together with the encoding of the words and lower case words, we saw an improvement in the performances of 4%, bringing the F1 score on the Devel to 67.4%.

- Then we tried to fed the Neural Network using more vectors encoding the information of the suffixes, using different lengths. We continued with this approach, focusing instead on the prefixes. So we created different vectors encoding the information on the prefixes of the words, varying the length of such prefixes. We tried several configurations, having only one vector of suffixes and one vector of prefixes for each word, or using two or more vector of suffixes and no vector of prefixes and viceversa. What we observed in this case is that adding more features encoding different information regarding the prefixes and suffixes is not beneficial for the training of the NN. Indeed in all the configurations attempted, the performances have worsen with respect to the one of the model encoding just suffixes of length 5.

- We then used as additional input information the POS tag of each word. We used an additional input layer in our NN model and an embedding also for the POS tags, by trying several embedding sizes. We combined such feature together with the encoding of the words, lower case words and suffixes of length 5. We observed a worsening in the performances in the Devel, which reached 65.7%.

- The next step we performed, was trying to take into account more information regarding the words, like the Lemma to which the words belong. Therefore, we added for each word, a vector representing the lemma. Training the model with features representing the lemma for each word gave us better results. Indeed, we trained the model using as features the encoding of the words, lower case words, suffixes of length 5 and lemma, obtaining an improvement with respect of the best configuration found before of 1.4%. In fact we obtained a F1 score on the Devel of 68.8%.

- We finally tried to apply the GloVe embeddings defined in the *Architecture* section to the words and lower-case words. Therefore, we combined these embeddings with different suffixes of different lengths, together with feature vectors describing the lemma and pos tag of each word. The best result obtained, was the one considering as feature vectors, only the GloVe embeddings of the words together with the lower case words. Indeed we reached a performance on the Devel dataset of 71.2. Therefore, we decided to continue

with this configuration also on the best performing model, obtained after all the trials in the *Architecture* section.

3.3 Code

We include here the code of the *build_network()* function, which defines the best model we decided to use, together with the code needed to implement the embedding using GloVe.

```
# rename packages
tfk = tf.keras
tfkl = tf.keras.layers

# downlowad pre-trained Glove embedding
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip -q glove.6B.zip

embeddings_index = {}
with open("glove.6B.300d.txt", encoding='utf-8') as f:
    for line in f:
        #print(line)
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print("Found %s word vectors." % len(embeddings_index))

num_tokens = codes.get_n_words()

embedding_dim = 300
hits = 0
misses = 0

# Prepare embedding matrix
embedding_matrix = np.zeros((num_tokens, embedding_dim))
for word, i in codes.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # Words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))

# GloVe embedding for lower case words
num_tokens = codes.get_n_lc_words()

embedding_dim = 300
hits = 0
misses = 0

# Prepare embedding matrix for lc words
```

```
embedding_matrix_lc = np.zeros((num_tokens, embedding_dim))
for word, i in codes.lc_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # Words not found in embedding index will be all-zeros.
        embedding_matrix_lc[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))

def build_network(codes) :
    # sizes
    n_words = codes.get_n_words()
    max_len = codes.maxlen
    n_labels = codes.get_n_labels()
    n_lc = codes.get_n_lc_words()

    # word input layer & embeddings
    inptW = tfk.Input(shape=(max_len,))
    embW = tfkl.Embedding(
                        input_dim=n_words,
                        output_dim=embedding_dim,
                        input_length=max_len,
                        embeddings_initializer=
                        tfk.initializers.Constant(embedding_matrix),
                        trainable=True,
                        mask_zero=False)(inptW)

    # lower-case word input layer & embeddings
    inptLC = tfk.Input(shape=(max_len,))
    embLC = tfkl.Embedding(
                    input_dim=n_lc,
                    output_dim=embedding_dim,
                    input_length=max_len,
                    embeddings_initializer=
                    tfk.initializers.Constant(embedding_matrix_lc),
                    trainable=True,
                    mask_zero=False)(inptLC)

    # apply dropout to embeddings
    dropW = tfkl.Dropout(0.5)(embW)
    dropLC = tfkl.Dropout(0.5)(embLC)

    # concatenate embeddings
    drops = tfkl.Concatenate(axis=2)([dropW, dropLC])

    bilstm = tfkl.Bidirectional(tfkl.LSTM(units=100, dropout=0.2,
    ↪  return_sequences=True))(drops)
    cnn = tfkl.Conv1D(filters=100, kernel_size=2, strides=1, activation='relu',
    ↪  padding='same')(bilstm)
    gap = tfkl.GlobalAveragePooling1D()(cnn)
```

```
95
96      out = tfkl.Dense(n_labels, activation='softmax')(gap)
97
98      # build and compile model
99      model = tfk.models.Model([inptW, inptLC], out)
100     model.compile(loss='categorical_crossentropy', optimizer='adam',
        ↪  metrics=['accuracy'])
101
102     return model
```

Finally, the code of the *encode_words()* and the other auxiliary methods in the *code_maps.py* needed to build the features vectors.

```
1       ## --------- encode X from given data -----------
2       def encode_words(self, data) :
3
4           # encode and pad sentence words
5           Xw = self.__encode_and_pad(data, self.word_index, 'form')
6           # encode and pad sentence lc_words
7           Xlw = self.__encode_and_pad(data, self.lc_word_index, 'lc_form')
8           # encode and pad lemmas
9           Xl = self.__encode_and_pad(data, self.lemma_index, 'lemma')
10          # encode and pad PoS
11          Xp = self.__encode_and_pad(data, self.pos_index, 'pos')
12
13          # encode and pad suffixes
14          Xs1 = [[self.suf_index1[w['lc_form'][-self.suflen1:]] if
            ↪  w['lc_form'][-self.suflen1:] in self.suf_index1 else
            ↪  self.suf_index1['UNK'] for w in s['sent']] for s in data.sentences()]
15          Xs1 = pad_sequences(maxlen=self.maxlen, sequences=Xs1, padding="post",
            ↪  value=self.suf_index1['PAD'])
16
17
18          # encode and pad suffixes (with different length)
19          Xs2 = [[self.suf_index2[w['lc_form'][-self.suflen2:]] if
            ↪  w['lc_form'][-self.suflen2:] in self.suf_index2 else
            ↪  self.suf_index2['UNK'] for w in s['sent']] for s in data.sentences()]
20          Xs2 = pad_sequences(maxlen=self.maxlen, sequences=Xs2, padding="post",
            ↪  value=self.suf_index2['PAD'])
21
22
23          # encode and pad prefixes
24          Xp1 = [[self.pref_index1[w['lc_form'][:self.preflen1]] if
            ↪  w['lc_form'][:self.preflen1] in self.pref_index1 else
            ↪  self.pref_index1['UNK'] for w in s['sent']] for s in data.sentences()]
25          Xp1 = pad_sequences(maxlen=self.maxlen, sequences=Xp1, padding="post",
            ↪  value=self.pref_index1['PAD'])
26
27
28          # encode and pad prefixes (with different length)
```

```
        Xp2 = [[self.pref_index2[w['lc_form'][:self.preflen2]] if
        ↪  w['lc_form'][:self.preflen2] in self.pref_index2 else
        ↪  self.pref_index2['UNK'] for w in s['sent']] for s in data.sentences()]
        Xp2 = pad_sequences(maxlen=self.maxlen, sequences=Xp2, padding="post",
        ↪  value=self.pref_index2['PAD'])

        # return encoded sequences
        return [Xw,Xlw]

    ## --------- encode Y from given data -----------
    def encode_labels(self, data) :
        # encode and pad sentence labels
        Y = [self.label_index[s['type']] for s in data.sentences()]
        Y = [to_categorical(i, num_classes=self.get_n_labels()) for i in Y]
        return np.array(Y)

    ## -------- get word index size ---------
    def get_n_words(self) :
        return len(self.word_index)
    ## -------- get word index size ---------
    def get_n_lc_words(self) :
        return len(self.lc_word_index)
    ## -------- get label index size ---------
    def get_n_labels(self) :
        return len(self.label_index)
```

3.4 Experiments and Results

In this section, we present the best results obtained, combining the most performing model together with the input features that improve the most the training of the NN. The results were obtained with the model reported in the above code snippet. Specifically, we report in Figure 2 the statistics obtained from this model on the Devel and Test datasets.

	tp	fp	fn	#pred	#exp	P	R	F1
advise	94	51	47	145	141	64.8%	66.7%	65.7%
effect	216	65	96	281	312	76.9%	69.2%	72.8%
int	21	3	7	24	28	87.5%	75.0%	80.8%
mechanism	159	50	102	209	261	76.1%	60.9%	67.7%
M.avg	-	-	-	-	-	76.3%	68.0%	71.8%
m.avg	490	169	252	659	742	74.4%	66.0%	70.0%
m.avg(no class)	533	126	209	659	742	80.9%	71.8%	76.1%

	tp	fp	fn	#pred	#exp	P	R	F1
advise	114	36	95	150	209	76.0%	54.5%	63.5%
effect	179	65	107	244	286	73.4%	62.6%	67.5%
int	16	6	9	22	25	72.7%	64.0%	68.1%
mechanism	207	117	133	324	340	63.9%	60.9%	62.3%
M.avg	-	-	-	-	-	71.5%	60.5%	65.4%
m.avg	516	224	344	740	860	69.7%	60.0%	64.5%
m.avg(no class)	579	161	281	740	860	78.2%	67.3%	72.4%

Figure 2: Best NN model performances on the Devel set (on the left) and on the Test set (on the right

What we can observe is that the model presents good performances also in the Test set, which proofs us the robustness of the model and its capabilities to generalize, indeed we have a small drop of 6% in the F1 score. In the case of the 'int' class, we can see a more significant drop between Devel and Test: this is due to the fact that the dataset is highly unbalanced, and there are very few samples in that class which makes it difficult for our NN classifier to correctly detect samples in the class. For what regard the other classes, we have comparable performances betweeen Devel and Test.

4 Conclusions

During this assignment we solved the DDI and the NERC task by using an approach based on Neural Networks. We had the possibility to experiment with different architectures and input information, and we observed how it is difficult to improve the performances, while keeping a good trade-off between Devel and Test. Indeed NN are very prone to overfitting, since their huge number of parameters may lead to great Train and Devel performances, but very bad ones on the Test data. Moreover, NN approaches generally needs much greater volumes of data with respect to ML approaches, to provide reliable results that don't overfit the training. As expected NN provides better results with respect to the standard ML approach we used in the previous assignments in both DDI and NERC task. The power of a Neural Network approach to such problems, is the ability of the model to capture the relationships among the words in a sentence, thanks to LSTMs and CNN architectures. Moreover, with respect to machine learning, a neural network approach, typically needs less ongoing human intervention, once the model has been built.