Deep Neural Networks for NER and DDI

Zofia Tarant, Marcel Pons

June 13, 2021

In the previous reports, rule-based implementations and machine learning approaches were used for the first and second tasks of the SemEval 2013 competition. In this report, deep learning models are implemented to handle both tasks, assessing whether these approaches accomplish significant improvements on the predictions.

As a reminder, both tasks involved analysing biomedical text from the Drugank and Medline abstracts on the subject of drug-drug interactions (DDI corpus). The first task consisted on the named entity recognition (NER) of the drugs appearing in the abstracts, classifying them into either *drug*, *group*, *brand* and *drug_n*. On the other hand, the second task involved the identification and extraction of the interactions between pairs of drugs (DDI) in each sentence, classifying their relationship into either *mechanism*, *effect*, *advise* and *int*.

As it has been done in the previous reports, in both NERC and DDI python programs that parse all the XML files of the DDI corpus, preprocess the text and perform the given task are developed. More specifically, for each task two programs are deployed: a `learner.py` which takes the training data to optimize all the deep learning model parameters, being saved to be used by the `classifier.py`, which is used to classify unseen text and ultimately evaluate the model performance by means of the `evaluator.pyc` script provided by the instructors. The evaluator compares the results with a ground truth and returns classification metrics (precision, recall , *F1*, etc.) for each type of interaction. Besides, a microaverage and a macroaverage are also provided. The macroaverage is the metric used to evaluate the rules implemented and the final performance of the models.

Both NERC and DDI programs have a similar structure and share some functions (with slight modifications). The principal difference between them lies in the neural network architectures, especially due to the fact that the NER task is a many-to-many situation, taking as input a sequence of tokens and returning as output a sequence of tags, whereas the DDI task is a many-to-one situation in which a unique output is returned given an input sequence.

1 Named Entity Recognition & Classification

Preprocessing functions

In the NER-NN model, both `learner` and `classifier` start by loading the data. The `load_data()` functions reads through the XML files of the given input directory, tokenizes each sentence and extracts the start-end offset and ground truth BIO tag for each token. The function returns a dictionary with the sentences id (`sid`) as keys and a list conformed by tuples (`word, start end, tag`) as items.

Since the neural network models only accept numeric data as input, the text sentences and labels have to be converted into vectors. To do so, first an index of all the words and labels in the data used to train the models is created with the function `create_index()`, which returns a dictionary where each key is an index name (e.g. "words", "labels"), and the value is a dictionary mapping each word/label to a number. For the words dictionary, numbers 0 and 1 correspond to `<PAD>` and `<UNK>`, respectively. On the other hand, only `<PAD>` is considered for the labels dictionary. `<UNK>` referes to an unknown word and `<PAD>` referes to a phantom padding word, as all sentences have been standardized to be represented by a fixed number of words, truncating some sentences and extending (padding) others.

The index created is used by the `encode_words()` function, which encodes the words in each sentence that was formed by lists of tokens in `load_data()` into lists of integer indexes suitable for the neural network input. Since all the vectors have to be of the same length, sentences are post-padded with a maximum length of `max_len`. `Max_len` is a potential parameter that influences the final performance of the NER classification. We tried several lengths, being 20, which corresponds to the average length (see Annex for length distribution), the one that gives the best results. Large values of `max_len`, around 40, lead to a high validation accuracy of the network training, but very low actual results of the evaluator, as most "words" in the input were actually the auxiliary `<PAD>` tokens.

In the train data, all words have their corresponding index number (since the index dictionary was build from these data), however, in both devel and test data, new words appear, thus implying not having an index in the dictionary. For this reason, they are mapped to the `<UNK>` index.

Similar functions, `encode_suffixes()` and `encode_pos_tags()` were coded. They also take the lists of tokens from `load_data()`, but they process the words so that they create an index representing the last 4 letters or the POS tag of the word, respectively.

The B-I-O tags are encoded in the same way as `encode_labels()`, where all labels are encoded to their corresponding integer values, this time in all datasets since the labels are predefined to appear in all of them.

`learner`

The final network consists of three types of input: the words themselves, suffixes and POS tags. Each input is fed into respective embedding layers. These layers include pre-trained embedding weights from Stanford's global vectors (`GloVe`). Subsequently, each of those embeddings is fed into a bidirectional LSTM layer (128 neurons for the word embeddings and 64 for the other two types of embeddings; dropout and recurrent dropout of 0.1). Then, those layers are concatenated. One more BiLSTM layer (with 64 neurons and 0.2 dropout values) is next and after that comes a bidirectional simple RNN with 32 neurons. The output layer is a dense layer with a softmax activation function. The architecture is visualized in Figure 1.

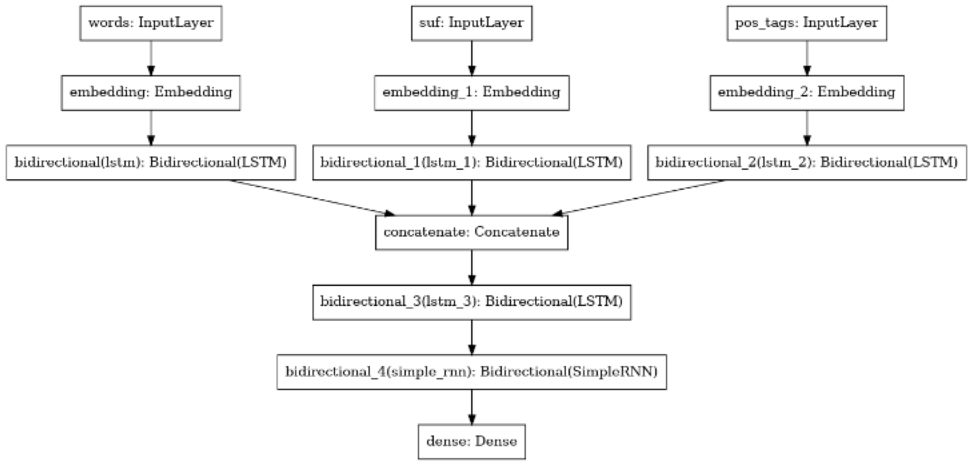

Figure 1: NERC neural network architecture

We found that the inclusion of POS tag embeddings brought the most significant improvement to the quality of the network; the F-measure produced by the evaluator was increased by around 10 percentage points. We used the Adam optimizer with a 0.005 learning rate, and calculated loss using categorical cross-entropy.

The network is trained using batches of size 16 and 3 epochs. It has a 93% training accuracy and 94% validation accuracy. We found that running the learning process for large numbers of epochs led to overfitting.

```python
def build_network(idx):
    '''
    Task: Create network for the learner.
    Input:
        idx: index dictionary with word/labels codes, plus maximum sentence length.
    Output: Returns a compiled Keras neural network with the specified layers
    '''

    #sizes
    n_words = len(idx['words'])
    n_labels = len(idx['labels'])#+1
    max_len = idx['maxlen']
    suffix_dict = create_suffix_index(idx)
    n_suf = len(suffix_dict)
    # create network layers
    inp = Input(shape=(max_len,), name="words")
    inp_suf = Input(shape=(max_len,), name="suf")
    inp_pos = Input(shape=(max_len,), name="pos_tags")
    model1 = Embedding(input_dim=n_words+1, output_dim=128, weights=[embedding_matrix], input_length=max_len)(inp)
    model2 = Embedding(input_dim =n_suf+1, output_dim=64, weights=[suffix_embedding_matrix], input_length=max_len)(inp_suf)
    model3 = Embedding(input_dim =n_suf+1, output_dim=64, input_length=max_len)(inp_pos)
```

```
    model1 = Bidirectional(LSTM(units=256, return_sequences=True, recurrent_dropout=0.1, dropout=0.1))(model1)
    model2 = Bidirectional(LSTM(units=128, return_sequences=True, recurrent_dropout=0.1, dropout=0.1))(model2)
    model3 = Bidirectional(LSTM(units=128, return_sequences=True, recurrent_dropout=0.1, dropout=0.1))(model3)
    model = Concatenate()([model1, model2, model3])
    model = Bidirectional(LSTM(units=64, return_sequences=True, recurrent_dropout=0.2, dropout=0.2))(model)
    model = Bidirectional(SimpleRNN(units=32, return_sequences=True))(model)
    model = Dense(n_labels, activation="softmax")(model)
    model = Model(inputs=[inp, inp_suf, inp_pos], outputs=model)

    optimiz = Adam(lr=0.005, amsgrad=True, epsilon=1e-7)
    model.compile(optimizer=optimiz, loss='categorical_crossentropy', metrics=["accuracy"])

    return model
```

`classifier`

The classifier program starts by loading the model and the indexes with `load_data()`. Then it loads the test data with `load_data()` and encodes the words with `encode_words()`. Next, `model.predict()` is called to classify the interactions on the unseen test data. The predictions are the probabilities of the type of interaction that exists between pairs, hence by using the `arg.max` function to find the label that is the most probable and then assigning the label to the corresponding the final classification is obtained. Finally, the classifications are passed to `output_interactions()` function to make them compatible for the evaluator.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	203	87	157	290	360	70.0%	56.4%	62.5%
drug	987	217	926	1204	1913	82.0%	51.6%	63.3%
drug_n	4	63	41	67	45	6.0%	8.9%	7.1%
group	215	348	466	563	681	38.2%	31.6%	34.6%
M.avg	-	-	-	-	-	49.0%	37.1%	41.9%
m.avg	1409	715	1590	2124	2999	66.3%	47.0%	55.0%
m.avg(no class)	1545	574	1454	2119	2999	72.9%	51.5%	60.4%

Figure 2: Results of development dataset

	tp	fp	fn	#pred	#exp	P	R	F1
brand	158	122	130	280	288	56.4%	54.9%	55.6%
drug	1048	318	1072	1366	2120	76.7%	49.4%	60.1%
drug_n	9	45	63	54	72	16.7%	12.5%	14.3%
group	230	359	469	589	699	39.0%	32.9%	35.7%
M.avg	-	-	-	-	-	47.2%	37.4%	41.4%
m.avg	1445	844	1734	2289	3179	63.1%	45.5%	52.9%
m.avg(no class)	1615	668	1564	2283	3179	70.7%	50.8%	59.1%

Figure 3: Results of test dataset

Tried and discarded architectures

We experimented with different architectures and types of layers. For instance, we found that adding an additional BiLSTM layer decreased the final F-measure by around 5 percentage points. We tried networks without pre-trained embedding weights, but they were significantly less accurate. We also experimented with different hyperparameters and dropout values for BiLSTM layers, but larger dropout rates decreased the accuracy by at least 3 percentage points.

We decided not to include a CRF layer after hours of experimenting with it. We read the Tensorflow documentation of CRF, some tutorials using it as well as GitHub issues forums and we found that many people struggle to use it as well. When we added the CRF layer from the Tensorflow addons package to our existing model, we found that the weights were not updating because the optimizer wasn't able to recognize the gradients of the layers before the CRF layer. Other implementations that we found either required Tensorflow 1 or did not support the serialization of the model. After all those hours, we decided that we did not have time to manually implement a serializable CRF layer.

Below are some of the attempted models:

- A network with only word embeddings, a single BiLSTM layer with 64 neurons and a dense layer with softmax activation F1 of 25.9% for the devel dataset and 28.9% for the test dataset
- The final network, but with only word embeddings: F1 of 34.5% for the devel dataset and 34.0% for the test dataset
- The final network with only word and suffix embeddings: F1 of 35.9% for the devel dataset and 33.5% for the test dataset
- A network with word, suffix and POS tag embedings with three BiLSTM layers + a Dense layer with softmax activation: F1 of 28.1% for the devel dataset and 29.6% for the test dataset

2 Drug-Drug Interaction

Preprocessing functions

For the DDI task the same preprocessing of the data is carried out in order to make the data compatible for neural network models. Nevertheless, the `load_data()` function is different from the NER task. In this case, the function reads through the XML files of the given directory, and for each sentence, it creates a list composed by the sentence id (`sid`), the entities identifications that make a pair (`eid1`, `eid2`), the interaction type and a list of the sentence tokens, including word form, lemma and stem. Therefore, in a sentence different pair combinations can appear, making the output of the function bigger than that of NER. Furthermore, on each pair instance, the drugs conforming the pair are changed to `<DRUG1>` and `<DRUG2>`, whereas any other drug appearing in the sentence that is outside the pair relationship is changed to `<OTHERDRUG>`. Moreover, the stopwords are converted to `<SW>` and the punctuation to `<PUNCT>`. We do so because these tokens do not make an impact when predicting drug-drug interactions. Also, the dimensions of the word index that is built afterwards are smaller.

The function `create_index()` creates a dictionary for the different words, 4-word suffixes and prefixes and part of speech tags. The encoding functions work in the same way than the used in the NER task. On the contrary, `encode_labels()` is different because now there is only a label to encode for each pair, therefore no padding is required.

For all the encodings except from the labels, different `max_length` were tried. We started by using length 20, which is the average length of the sentences (see Annex for length distribution), but the final results were not remarkable. Then we increased gradually the parameter, assessing how it influenced the performance. In the end, `100` was the `max_length` that gave better results.

`learner`

Several deep learning architectures have been tried in order to find the one that best predicts the interactions. Among those, the one that yields the better performance is surprisingly a single LSTM layer with 64 units followed by a dense layer of 24 neurons with `relu` activation function and an output dense layer with `n_labels` neurons with `softmax` activation function. The only input of the model is the encoded words, which are embedded into global vectors (GloVe) within 16 embedding dimensions. The architecture is visualized in Figure 4.

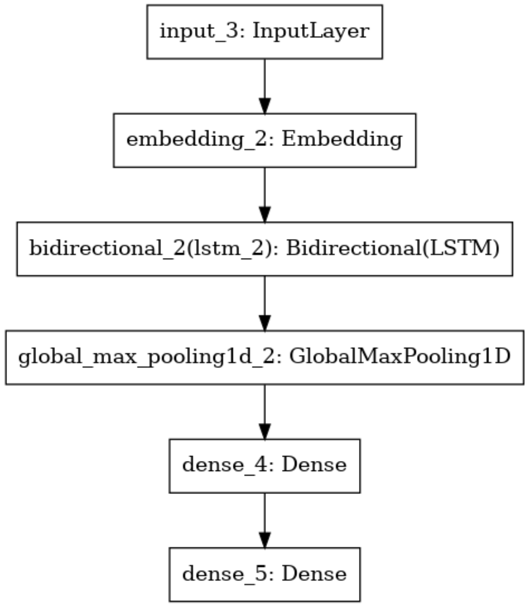

Figure 4: DDI neural network architecture

The Adam optimizer is chosen as the parameter optimization algorithm, with a learning rate of 0.01 and decay 10^{-5}. RMSProp was also used but the models performed worse. Finally, categorical cross-entropy is used as loss function.

```python
def build_network(idx, embedding_dim, embedding_matrix):

    # Sizes
    n_words = len(idx['words'])
    n_labels = len(idx['labels'])
    max_len = idx['maxlen']

    # Create network layers
    inp = Input(shape=(max_len,))
    emb_layer = Embedding(input_dim=n_words+1, output_dim=embedding_dim, weights=[embedding_matrix], input_length=max_len)(inp)
    model = Bidirectional(LSTM(units=64, return_sequences=True, recurrent_dropout=0.4))(emb_layer)
    model = GlobalMaxPooling1D()(model)
    model = Dense(24, activation='relu')(model)
    out = Dense(n_labels, activation='softmax')(model)

    # Create and compile model
    model = Model(inp, out)

    optimiz = Adam(lr=0.01, decay=1e-6)
    model.compile(optimizer=optimiz, loss="categorical_crossentropy", metrics=["accuracy"])

    return model
```

The model is trained with the training data with a batch size of 32 and 5 epochs. Also, the development data is used to obtain a validation accuracy. The scores are used as an orientative measure of how the models are working, since most of the pair interactions have `null` type, misleading the real classification performance. The aforementioned model showed a 94.2% training accuracy and a 87.7% validation accuracy.

Once the model is trained by learning from the training data, it is saved together with the index dictionary using the `save_model_and_indexs()` function.

`classifier`

The classifier works analogically to the NER learner described above.

	tp	fp	fn	#pred	#exp	P	R	F1
advise	64	40	74	104	138	61.5%	46.4%	52.9%
effect	154	114	161	268	315	57.5%	48.9%	52.8%
int	15	2	20	17	35	88.2%	42.9%	57.7%
mechanism	130	117	134	247	264	52.6%	49.2%	50.9%
M.avg	-	-	-	-	-	65.0%	46.8%	53.6%
m.avg	363	4306	389	4669	752	7.8%	48.3%	13.4%
m.avg(no class)	752	3917	0	4669	752	16.1%	100.0%	27.7%

Figure 5: Results of development dataset

	tp	fp	fn	#pred	#exp	P	R	F1
advise	88	52	124	140	212	62.9%	41.5%	50.0%
effect	152	107	131	259	283	58.7%	53.7%	56.1%
int	6	4	12	10	18	60.0%	33.3%	42.9%
mechanism	206	198	131	404	337	51.0%	61.1%	55.6%
M.avg	-	-	-	-	-	58.1%	47.4%	51.1%
m.avg	452	5239	398	5691	850	7.9%	53.2%	13.8%
m.avg(no class)	850	4841	0	5691	850	14.9%	100.0%	26.0%

Figure 6: Results of test dataset

Tried and discarded architectures

As mentioned above, other architectures were tried consisting in using different layers, optimizers and learning rates, several inputs and different embedding dimensions. We expected that by increasing the complexity of the architectures better classification results could have been obtained, but that was not the case. Here below are some of the models attempted:

- Convolutional Neural Network (CNN) with 128 filters and kernel size 5 performed with a 50.6% F1 in devel and 41.9% F1 in test.

- Bidirectional LSTM with multiple inputs (word, suffix and prefix[1]). We first tried the model

[1] PoS tags and stems were tried but the results were worse considering them.

with 128 units in each layer, classifying devel with 61.5% F1 and test with 46.3% F1. By relaxing the complexity to 64 units in each layer, we increased test F1 to 49.5% F1 (devel decreased to 56.9%).

- Bidirectional LSTM + CNN with multiple inputs. It performed not as expected with 53.4% F1 in devel data and 40.1% F1 in test data.

3 Conclusions

Different neural network architectures were tried for both the NER and DDI tasks of the SemEval competition. Initially, since NN are the preferred models which accomplish the best results in many classification problems, we thought that by using these models the accuracies in both tasks would easily overcome the ones obtained by using Machine Learning models. Unfortunately we have not been able to find architectures that were significantly better, accomplishing considerably good results for the deployment on biomedical drug related texts.

One of the takeaways that apply to both parts of this assignment, is that the pre-trained GloVe embedding weights significantly improved the results. We also learned the importance of choosing the right fixed length of the input layer. While we effectively lose some information by cutting off the endings of long sentences, ultimately a large size of the input layer leads to a low F-measure caused by the over-representation of padding tags.

With respect to the NER task, the Machine Learning approach based on CRF, developed as a part of the previous laboratory assignment, surpassed the best neural network model achieved. As previously mentioned, many NN architectures were tried to improve these results, but none of them performed as desired. We think that a CRF layer on top of the LSTM and SimpleRNN layers could have been a key point on the performance of the model, but we have not been able to make the `tfa.layers.CRF` work due to incompatibilities with the other layers. Besides, `keras_contrib.layers.CRF` was deprecated and did not work with our Tensorflow version, while other available implementations did not include all the desired functionalities.

We discovered that the key factor in improving the performance of our NN was the inclusion of suffix and POS tag embeddings. While they increased the complexity of the network, they also provided vital information.

Approach	F1 devel	F1 test
Rule-Based	27.60%	39.95%
ML-CRF	61.1%	63.1%
NN	41.9%	41.4%

Table 1: NER comparisons

Regarding the DDI task, while not meeting the expectations, the obtained results were better than the other models implemented in previous works. One of the architectures which we ultimately discarded, but had some potential, included biLSTM multi-input. It initially seemed to perform quite well in the development dataset, with a 61.5% F-measure and 93.2% validation accuracy while training the model, however the results on the test data were not good enough. We think that should we have kept trying architectures following that line, better models could have been obtained. Nevertheless, among the models that we were able to analyze, we were surprised to discover that a very simple model displayed a *good enough* performance.

Approach	F1 devel	F1 test
Rule-Based	31.0%	27.5%
ML-MEM	53.4%	42.2%
NN	53.6%	51.1%

Table 2: DDI comparisons

Throughout the semester, we have learned the nuances of named entity recognition and the detection of interactions between entities. The experiments that we conducted improved our knowledge in the area of human language technologies. We also faced challenges related to available tools and the compatibility of the solutions with the provided `evaluator` function (e.g. the offsets of entities). Although we aimed to obtain significantly better results for the neural network approach and did not quite meet that expectation, we still managed to achieve the best (out of the three approaches) result for the DDI part. We believe that if we had had more time and experience with neural networks, we would have succeeded in creating better solutions.

4 Annex

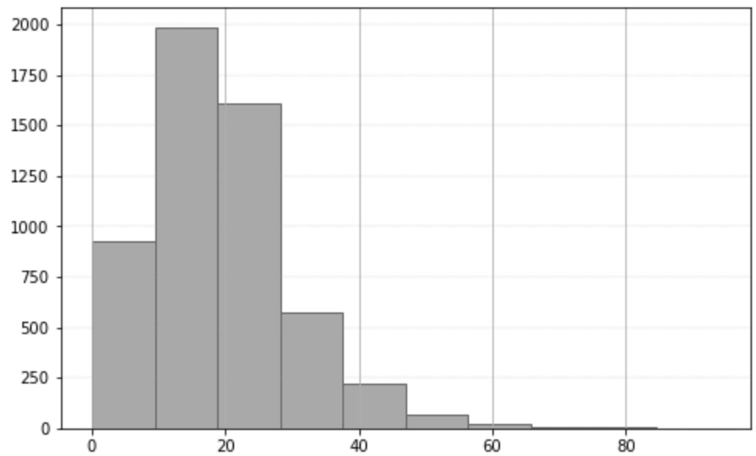

Figure 7: Distribution of sentence length in the training dataset

Code - NERC NN

```
def encode_words(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    for key, item in dataset.items():
        aux = []
        for t in item:
            w = str(t[0]).lower() # When using lower case words
            if w in idx['words']:
                i = idx['words'][w]
            else:
                i = idx['words']['<UNK>']
            aux.append(i)
        seq.append(aux)
    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')

    return seq_padded

def encode_labels(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    for key, item in dataset.items():
        aux = []
        for t in item:
            w = t[3]
            i = idx['labels'][w]
            aux.append(i)
        seq.append(aux)
    seq_padded = [x  for x in pad_sequences(maxlen = max_length, sequences = seq, padding =␣
↪'post', truncating="post")]
```

```
    seq_categ = [to_categorical(i, num_classes = 10) for i in seq_padded]  # 9 classes + 1 PAD

    return seq_padded, seq_categ
```

```
def create_suffix_index(word_index):
    suffix_dict = {}
    i = 0
    for word in word_index['words']:
        suf = word[-4:]
        if suf not in suffix_dict:
            suffix_dict[suf] = i
            i+=1
    return suffix_dict

suffix_index = create_suffix_index(idx)

def encode_suffixes(dataset, idx, suf_index):
    max_length = idx['maxlen']
    seq = []
    for key, item in dataset.items():
        aux = []
        for t in item:
            w = str(t[0]).lower()[-4:]
            if w in suf_index:
                i = suf_index[w]
            else:
                i = suf_index['UNK>']
            aux.append(i)
        seq.append(aux)

    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')

    return seq_padded
```

Note: the `create_suffix_index()` function iterates through the word index to create a new index with just 4-letter suffixes of all words in the train dataset.

```
pos_tags = ['LS', 'TO', 'VBN', "''", 'WP', 'UH', 'VBG', 'JJ', 'VBZ', '--', 'VBP', 'NN', 'DT',␣
↪'PRP', ':', 'WP$', 'NNPS', 'PRP$', 'WDT', '(', ')', '.', ',', '``', '$', 'RB', 'RBR', 'RBS',␣
↪'VBD', 'IN', 'FW', 'RP', 'JJR', 'JJS', 'PDT', 'MD', 'VB', 'WRB', 'NNP', 'EX', 'NNS', 'SYM',␣
↪'CC', 'CD', 'POS']
def create_pos_index():
    pos_dict = {'<UNK>': 0}
    i = 1
    for t in pos_tags:
        pos_dict[t] = i
        i+=1
    return pos_dict

pos_index = create_pos_index()
```

```python
def encode_pos_tags(dataset, idx, pos_tag_index):
    max_length = idx['maxlen']
    seq = []
    for key, item in dataset.items():
        aux = []
        sentence = [t[0] for t in item]
        pos_tags = nltk.pos_tag(sentence)
        for w, tag in pos_tags:
             # When using lower case words
            if tag in pos_tag_index:
                i = pos_tag_index[tag]
            else:
                i = pos_tag_index['<UNK>']
            aux.append(i)
        seq.append(aux)

    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')

    return seq_padded
```

```python
def learner(traindir, validationdir):#, modelname):
    # load train and validation data in a suitable form
    train_dataset = load_data(traindir)
    val_dataset = load_data(validationdir)

    # create indexes from trainindg data
    max_len = 20
    idx = create_index(train_dataset, max_len)

    # build network
    model = build_network(idx)

    # encode datasets
    Xtrain = encode_words(train_dataset, idx)
    Y, Ytrain = encode_labels(train_dataset, idx)
    Xval = encode_words(val_dataset, idx)
    Yv, Yval = encode_labels(val_dataset, idx)

    X_train_suf = encode_suffixes(train_dataset, idx, suffix_index)
    X_train_pos = encode_pos_tags(train_dataset, idx, pos_index)
    X_val_suf = encode_suffixes(val_dataset, idx, suffix_index)
    X_val_pos = encode_pos_tags(val_dataset, idx, pos_index)

    model.fit({"words": Xtrain, "suf": X_train_suf, "pos_tags": X_train_pos}, np.array(Y_train),
                    batch_size=16,
                    epochs=2,
                    verbose=1,
                    validation_data=({"words": Xval, "suf": X_val_suf, "pos_tags": X_val_pos}, np.
 ↪array(Y_dev)))
```

```
    # save model and indexs, for later use in prediction
    save_model_and_indexes(model, idx)
```

```
def predict(datadir, outfile):
    # load model and associated encoding data
    model, idx = load_model_and_indexs()

    # load data to annotate
    testdata = load_data(datadir)

    # encode dataset
    X = encode_words(testdata, idx)
    X_suf = encode_suffixes(testdata, idx, suffix_index)
    X_pos = encode_pos_tags(testdata, idx, pos_index)

    # tag sentences in dataset
    Y = model.predict({"words": X, "suf": X_suf, "pos_tags": X_pos})
    Y = [[find_label(idx, np.argmax(y)) for y in s] for s in Y]

    # extract entities and dump them to output file
    output_entities(testdata, Y, outfile)

    # evaluate using official evaluator
    evaluate("NER", datadir, outfile)
```

```
def output_entities(dataset, preds, outfile):
    outf = open(outfile, 'w')
    for sentence, pred in zip(dataset.items(), preds):
#         print(sentence, pred)
        sid = sentence[0]
        tokens = sentence[1]
        for i in range(min(len(tokens), len(pred))):
            token = tokens[i]
            label = pred[i]
            if label[0] == 'B':
                offset_from = str(token[1])
                offset_to = str(token[2])
                tag_name = label[2:]
                entity = token[0]
                j = i+1
                while j < len(tokens) and len(tokens[j]) >=3 and j>len(pred):
                    token_next = tokens[j]
                    word_next = token_next[0]
                    offset_from_next = str(token_next[1])
                    offset_to_next = str(token_next[2])
                    tag_next = pred[j]
                    j += 1
                    if int(offset_from_next) - int(offset_to) > 3 or tag_next[0] != 'I':
                        break
```

```
                    if tag_next[2:] == tag_name:
                        entity = entity + ' ' + word_next
                        offset_to = offset_to_next
                outf.write(sid + "|" + offset_from + '-' + offset_to + "|" + entity + "|" +␣
↪tag_name+'\n')
                print(sid + "|" + offset_from + '-' + offset_to + "|" + entity + "|" + tag_name)
```

Code - DDI NN

```
def create_index(dataset, max_length):
    index_words = {'<PAD>':0, '<UNK>':1}
    i = 2

    index_stems = {'<PAD>':0, '<UNK>':1}
    z = 2

    index_labels = {}
    j = 0

    index_suf = {'<PAD>':0, '<UNK>':1}
    ii = 2

    index_pref = {'<PAD>':0, '<UNK>':1}
    iii = 2

    pos_tags = ['LS', 'TO', 'VBN', "''", 'WP', 'UH', 'VBG', 'JJ', 'VBZ', '--', 'VBP', 'NN',
                'DT', 'PRP', ':', 'WP$', 'NNPS', 'PRP$', 'WDT', '(', ')', '.', ',', '``',
                '$', 'RB', 'RBR', 'RBS', 'VBD', 'IN', 'FW', 'RP', 'JJR', 'JJS', 'PDT', 'MD',
                'VB', 'WRB', 'NNP', 'EX', 'NNS', 'SYM', 'CC', 'CD', 'POS']
    index_pos = {'<PAD>':0, '<UNK>':1}
    y = 2

    for t in pos_tags:
        index_pos[t] = y
        y += 1
    for s in dataset:
        words = s[4]
        label = s[3]
        if label not in index_labels:
            index_labels[label] = j
            j += 1
        for tup in words:
            w = tup[0]
            suff = w[-4:]
            pref = w[:4]
            s = tup[1]
            if w not in index_words:
                index_words[w] = i
                i += 1
            if suff not in index_suf:
```

```
                index_suf[suff] = ii
                ii += 1
            if pref not in index_pref:
                index_pref[pref] = iii
                iii += 1
            if s not in index_stems:
                index_stems[s] = z
                z += 1

    idx = {'words': index_words, 'suffixes': index_suf, 'prefixes':index_pref, 'stems':
↪index_stems, 'PoS':index_pos, 'labels':index_labels, 'maxlen':max_length}

    return idx
```

```
def encode_words(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    for s in dataset:
        words = s[4]
        aux = []
        for tup in words:
            w = tup[0]
            if w in idx['words']:
                i = idx['words'][w]
            else:
                i = idx['words']['<UNK>']
            aux.append(i)
        seq.append(aux)

    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')

    return seq_padded

def encode_suffixes(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    for s in dataset:
        words = s[4]
        aux = []
        for tup in words:
            w = tup[0]
            suff = w[-4:]
            if suff in idx['suffixes']:
                i = idx['suffixes'][suff]
            else:
                i = idx['suffixes']['<UNK>']
            aux.append(i)
        seq.append(aux)

    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')
```

```
    return seq_padded

def encode_prefixes(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    for s in dataset:
        words = s[4]
        aux = []
        for tup in words:
            w = tup[0]
            pref = w[:4]
            if pref in idx['prefixes']:
                i = idx['prefixes'][pref]
            else:
                i = idx['prefixes']['<UNK>']
            aux.append(i)
        seq.append(aux)

    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')

    return seq_padded

def encode_stems(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    for s in dataset:
        words = s[4]
        aux = []
        for tup in words:
            w = tup[1]
            if w in idx['stems']:
                i = idx['stems'][w]
            else:
                i = idx['stems']['<UNK>']
            aux.append(i)
        seq.append(aux)

    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')

    return seq_padded

def encode_tags(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    for s in dataset:
        words = s[4]
        aux = []
        for tup in words:
            w = tup[0]
```

```
            pos = nltk.pos_tag(w)
            if w in idx['PoS']:
                i = idx['PoS'][w]
            else:
                i = idx['PoS']['<UNK>']
            aux.append(i)
        seq.append(aux)

    seq_padded = pad_sequences(maxlen = max_length, sequences = seq, padding = 'post')

    return seq_padded

def encode_labels(dataset, idx):
    max_length = idx['maxlen']
    seq = []
    seq = [idx['labels'][s[3]] for s in dataset]
    Y = [to_categorical(i, num_classes=5) for i in seq]
    Y = np.array(Y)

    return Y
```

```
def learner(traindir, validationdir):
    # load train and validation data in a suitable form
    traindata = load_data(traindir)
    valdata = load_data(validationdir)

    # create indexes from trainindg data
    max_len = 100
    idx = create_index(traindata, max_len)
    embedding_matrix = create_embedding_matrix(glove_path, idx, 16)

    # build network
    model = build_network(idx, 16, embedding_matrix)

    # encode datasets
    Xtrain = encode_words(traindata, idx)
    Ytrain = encode_labels(traindata, idx)
    Xval = encode_words(valdata, idx)
    Yval = encode_labels(valdata, idx)

    # train model
    model.fit(Xtrain, Ytrain,
              batch_size=16,
              verbose=1,
              epochs=3,
              validation_data=(Xval, Yval))

    save_model_and_indexes(model, idx)
```

```python
def output_interactions(dataset, preds, outf):
    length = len(dataset)
    for i in range(length):
        sid = dataset[i][0]
        id_e1 = dataset[i][1]
        id_e2 = dataset[i][2]
        ddi_type = preds[i]
        outf.write(str(sid) +"|"+ str(id_e1) +"|"+ str(id_e2) +"|"+ str(ddi_type))
        outf.write("\n")
```

```python
def predict(datadir, outfile):
    model, idx = load_model_and_indexs()

    testdata = load_data(datadir)

    X = encode_words(testdata, idx)
    X_suff = encode_suffixes(testdata, idx)
    X_pref = encode_prefixes(testdata, idx)

    Y = model.predict({"words": X, "suf": X_suff, "pref": X_pref})
    preds = []
    for s in Y:
        it = np.argmax(s)
        for key, item in idx['labels'].items():
            if item == it:
                preds.append(key)

    outf = open(outfile, "w")
    output_interactions(testdata, preds, outf)
    outf.close()

    evaluate("DDI", datadir, outfile)
```