Drug Recognition/Classification and Drug-Drug Interaction Detection with Neural Networks

Master in Data Science, FIB
Mining Unstructured Data

June 7th, 2023

Pau Comas Herrera - Odysseas Kyparissis

Table of Contents

Introduction

This report aims to examine and enhance several neural network (NN) architectures intended for the identification of drug entities - Named Entity Recognition Classification (NERC) - from medical corpora, as well as for the detection of drug-to-drug interactions (DDI). The general approach for NERC is built on top of the *B-I-O tagging* format[1]. The main objective of this assignment is to gain insights into the different steps involved in these approaches, including input embedding, model training using various neural network architectures, and result analysis. The continuous improvement of results is achieved through iterative experimentation with different options and endeavors to understand the workings of the sometimes intricate Neural Network architectures. During the input embedding phase, the goal is to extract significant syntactic and lexical information from each sentence, which is then converted into embeddings to be fed into the neural network. For model training, different Neural Network architectures are tested, briefly described in terms of their internal functions, and optimized by adjusting the hyperparameters. Finally, the performance of the model on unseen data is assessed to evaluate its generalization capability.

Moreover, within the mentioned sentences present in the corpora, there are four distinct categories of entities that require identification and subsequent classification: ***drug***, ***brand***, ***group***, and ***drug_n***. The *drug* category is utilized to denote human medications known by their generic names, while medicines referred to by their trade or brand names are categorized as *brand* entities. Additionally, given the frequent occurrence of descriptions of drug-drug interactions involving groups of medications in texts, the *group* category is employed to label such groups. Lastly, the entity type *drug_n* represents active substances that have not received authorization for human use, encompassing toxins or pesticides, among other substances.

On the other hand, in the case of the DDI corpus, which is a semantically annotated corpus of documents describing drug-drug entity interactions from the DrugBank database and MedLine abstracts, we face the goal of detecting when there is an interaction and when not, and classifying the existing drug-drug interactions into ***advice***, ***effect***, ***mechanism***, and ***int***. Hereafter we can read a brief description of the meaning of each one of them:

- **Advice**: DDI in which a recommendation or advice regarding the concomitant use of two drugs involved in them is described.
    - e.g.: “*Uroxatral* and *Methamphetamine* should not be used together”
- **Effect**: DDI in which the effect of the drug-drug interaction is described.
    - e.g.: “*Aspirin* and *Paracetamol* help decrease temperature.”
- **Mechanism**: The mechanism of interaction. Can be pharmacodynamic or pharmacokynetic.
    - e.g.: *Grepafloxacin*, like other *quinolones*, may inhibit the metabolism of *caffeine* and *theobromine*.

[1] https://en.wikipedia.org/wiki/Inside%E2%80%93outside%E2%80%93beginning_(tagging)

- **int**: the sentence simply states that an interaction occurs and does not provide any information about the interaction.
    - e.g.: The interaction of omeprazole and ketoconazole has been established.

The structure of the report includes two separate sections, namely Drug NERC with Neural Networks and Drug-Drug Interactions with Neural Networks, respectively. In both sections, the same structure has been followed, which incorporates a description of the architecture, followed by the input information, the final version of the code, experimentation and results and final conclusions per task.

Drug NERC with Neural Networks

Architecture

The neural network architecture used in this project is designed to process textual data from medical corpora and perform sequence labeling tasks. The architecture combines various input features and employs recurrent and dense layers to capture sequential dependencies and extract meaningful representations from the input data. The overall architecture of the neural network is defined using the Keras[2] functional API. The model is constructed by specifying the input layers and the subsequent flow of data through the embedding, bidirectional[3] LSTM[4] (BiLSTM), hidden, and output layers.

Compared to the traditional Machine Learning (ML) approach followed during the course's second laboratory session, this approach focuses on both using appropriate word embeddings to train the network in collaboration with new input features to improve results within the learning algorithm. Word embeddings are vector representations of words in a high-dimensional space, where each dimension captures a specific linguistic feature or semantic relationship. These embeddings are learned from large amounts of text data using techniques like Word2Vec[5], GloVe[6], or FastText[7]. By utilizing word embeddings, the network can capture meaningful relationships between words, enabling it to understand and generalize better.

To begin with, the following techniques will be tested to further enhance the performance of the network: Adding custom input embedding layers with lowercase words, their POS[8] tags, their lemmas, suffixes and prefixes of length 3 to 6, words that contain one capital letter, multiple capital letters, digits, and punctuation. Moreover, four additional input layers have been tested, which contain words that are included in the external sources provided (*DrugBank.txt, HSDB.txt)*, as well as a frozen layer containing pre-trained embeddings from Stanford (GloVe). For more detailed information, the following input and embedding layers are the ones kept in the final solution (the effect of all the tried inputs mentioned above are presented in the section Experiments and Results):

Input Layers

- *inptW*: layer for word indices representing the main words in the sentences.
- *inptS*: layer for suffix of length 5 indices representing the suffixes of the words.
- *inptLc*: lowercase word indices, which represent the lowercase versions of the words.

[2] https://keras.io/
[3] https://keras.io/api/layers/recurrent_layers/bidirectional/
[4] https://keras.io/api/layers/recurrent_layers/lstm/
[5] https://en.wikipedia.org/wiki/Word2vec
[6] https://nlp.stanford.edu/projects/glove/
[7] https://fasttext.cc/
[8] https://en.wikipedia.org/wiki/Part-of-speech_tagging

- *inptS3*, *inptS4*, *inptS6*: layers for suffix indices of different lengths (3, 4, and 6 characters), capturing variations in word endings.
- *inptPos*: part-of-speech tag indices representing the grammatical category of the words.
- *inptCs*: capitalization input layer indicating whether each word contains more than one capitalized letter or not.
- *inptSC*: single capitalization indices indicating whether each word contains a single capital letter.
- *inptPU*: punctuation input layer representing the presence or absence of punctuation marks in the words.
- *inptD*: digit indices layer indicating the presence or absence of digits in the words.
- *inptDr*: indices layer representing the presence or absence of drug-related terms in the words.
- *inptBr*: representing the presence or absence of brand-related terms in the words.
- *inptGr*: include the presence or absence of group-related terms in the words.
- *inptHs*: indices that include the presence or absence of HSDB (Hazardous Substances Data Bank) terms in the words.

Embedding Layers

For each input layer, an embedding layer is applied to transform the discrete indices into continuous vector representations. The embedding layers capture the semantic and syntactic relationships between words and their respective features. The dimensions of the embedding layers vary depending on the specific input:

- *embW*: embeddings for word indices with an output dimension of 100.
- *embS*: embedding layer for suffix indices with an output dimension of 50.
- *embLc*: lowercase word indices with an output dimension of 100.
- *embS3*, *embS4*, *embS6*: suffixes of different lengths (3, 4, and 6 characters) with an output dimension of 50.
- *embPos*: part-of-speech tag embedding layer with an output dimension of 50.
- *embCs*: capitalization embedding layer indicating whether each word contains more than one capitalized letter or not with an output dimension of 50.
- *embSC*: embeddings for single capitalization indices with an output dimension of 50.
- *embPU*: punctuation indices with an output dimension of 50.
- *embD*: embedding layer for digit indices with an output dimension of 50.
- *embDr*: drug-related term indices embeddings with an output dimension of 50.
- *embBr*: brand-related term indices with an output dimension of 50.
- *embGr*: embeddings for group-related term indices with an output dimension of 50.
- *embHs*: HSDB term indices embedding layer with an output dimension of 50.

Pre-trained GloVe Word Embeddings

Furthermore, the network utilizes pre-trained word embeddings from the GloVe (Global Vectors for Word Representation) model. The GloVe word embeddings capture semantic relationships between words based on their co-occurrence statistics in a large corpus of text. The

embeddings are loaded from the ***glove.6B.300d.txt*** file, which contains word vectors of dimensionality 300.

Finally, to initialize the word embedding matrix, each word in the vocabulary is matched with its corresponding pre-trained GloVe vector. Words absent in the pre-trained embeddings are initialized with random values. The embedding matrix is then used as weights in the frozen embedding layer *frozen_embW* for the word indices.

BiLSTM Layer

After embedding the input features, a BiLSTM layer is employed to capture contextual information and sequential dependencies within the text. The BiLSTM layer consists of 300 hidden units and is set to return sequences. Additionally, as an activation function was not explicitly specified in the initial script, the usage of the default activation function for LSTM layers in Keras, meaning the hyperbolic tangent[9] function (tanh) was used. Although ReLU[10] activation function was tested as well in this part of the architecture but the results were worse compared to Tanh.

Hidden Layer

A dense hidden layer follows the BiLSTM layer. This layer serves as a non-linear transformation to further extract and encode relevant features from the contextual representations provided by the BiLSTM. The intention of adding a hidden layer is to allow the network to learn more complex patterns included in the data. However, the difference in the final results is not so important. The hidden layer was tested with different configurations, including tries of 50 and 200 units (neurons) and usage of tanh and ReLU activation functions.

Output Layer

Finally, a time-distributed dense layer is applied to produce the output predictions for each position in the input sequences. The number of units in this layer corresponds to the number of unique labels in the dataset. The activation function used in the output layer is a softmax function, which yields a probability distribution among the target labels, enabling the classification of each word in the input sequence.

To conclude, the aforementioned tries involved several different preprocessing steps of the datasets before feeding them into the network. In that way, the network is trained to recognize more complex patterns that are included in the semantics and linguistic features of the training dataset, and thereby the result is the increasing ability to predict the class of the entities for the specific task.

[9] https://www.sciencedirect.com/topics/mathematics/hyperbolic-tangent-function
[10] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

By exploring these alternative approaches, we aim to identify the most effective methodology for training the network and achieving optimal results. The final results and experimentations for the different architectures and inputs tried are presented in the section Experiments and Results.

Code

In this section of the report, the updated version of the scripts ***codemaps.py*** and ***train.py*** (collab notebook) are presented. For space-saving reasons, only important information is included in this report, while the complete files can be found in the shared links of the text below.

To begin with, the same functions that were used in the second laboratory session of the course were applied here as well, in order to extract information from the external resources (*DrugBank.txt, HSDB.txt)* and save them into four separate lists named: *DB_drug_list*, *DB_brand_list*, *DB_group_list,* and *hsdb_list* respectively. Those two functions are called *ext_drug_bank* and *ext_HSDB* and can be found inside the updated Google Colab Notebook that was shared for the NERC task.

Moreover, in the following block one can find the updates of the file ***codemaps.py*** *(in this link is the complete python file).* The main changes included the update of the functions ***__create_indexs****,* and ***encode_words****,* as well as the addition of ***get_number*** functions for the new indexes created. Those changes were necessary in order to produce the current solutions and results. In order to present the logic followed for adding new input layers and their embeddings in the network, only the code used to include the capitalization indices and its embeddings (*inptCs*, *embCs*) is going to be presented. The rest of the code can be found in the shared links.

```python
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
nltk.download('averaged_perceptron_tagger', quiet=True)
nltk.download('wordnet')

def __create_indexs(self, data, maxlen, suflen, DB_drug_list, DB_brand_list, DB_group_list,
hsdb_list):

        lemmatizer = WordNetLemmatizer()
        self.maxlen = maxlen
        self.suflen = suflen
        ...
        caps = set([])
        ...
        for s in data.sentences() :
                for t in s :
                        if re.search(r'[A-Z]{2,}', t['form']):
                           caps.add(t['form'])
        self.caps_index = {cap: i + 2 for i, cap in enumerate(list(caps))}
        self.caps_index['PAD'] = 0  # Padding
```

```
        self.caps_index['UNK'] = 1  # Unknown capital letter patterns

def encode_words(self, data):
        ...
        # encode and pad caps
        Xcs = [[self.caps_index[w['form']] if w['form'] in self.caps_index
                        else self.caps_index['UNK'] for w in s] for s in data.sentences()]
        Xcs = pad_sequences(maxlen=self.maxlen, sequences=Xcs, padding="post",
                        value=self.caps_index['PAD'])
        ...
        return [Xw,Xs,Xlc,Xs3,Xs4,Xs6,Xpos, Xcs, Xsc, Xpu, Xd, Xdr, Xbs, Xgs, Xhsdb]

def get_n_caps(self) :
       return len(self.caps_index)
```

The same idea and logic were followed in order to create the total number of input layers and their indexes for the training, validation and test sets.

Moreover, in the following section, the updated version of the ***build_network*** function is included. Again for space-saving reasons, a simple example including only the input layers and embeddings of the words themselves, the capitalization indices (*inptCs*, *embCs*), the GloVe, BiLSTM and Hidden layers are being presented. The complete file can be found here:

Python

```
def build_network(codes) :

   # sizes
   n_words = codes.get_n_words()
   n_labels = codes.get_n_labels()
   n_caps = codes.get_n_caps()
   ...
   max_len = codes.maxlen

   ###################################
   # Load the GloVe word embeddings
   glove_file = utilsdir + "/glove.6B.300d.txt"
   embedding_dim = 300  # Adjust the dimension based on the downloaded file
   embeddings_index = {}
   with open(glove_file, encoding="utf8") as f:
       for line in f:
           values = line.split()
           word = values[0]
           coefs = np.asarray(values[1:], dtype="float32")
           embeddings_index[word] = coefs
    # Initialize the embedding matrix with random values
    embedding_matrix = np.random.random((n_words, embedding_dim))

    # Assign the pre-trained GloVe word embeddings to the corresponding words
```

```
#in the embedding matrix
for word, i in codes.word_index.items():
   if word in embeddings_index:
       embedding_matrix[i] = embeddings_index[word]

inptW = Input(shape=(max_len,))
inptCs = Input(shape=(max_len,))  # caps input layer & embeddings
embW = Embedding(input_dim=n_words, output_dim=100,
                 input_length=max_len, mask_zero=False)(inptW)
embCs = Embedding(input_dim=n_caps, output_dim=50,
                 input_length=max_len, mask_zero=True)(inptCs)
# Create the frozen embedding layer using the embedding matrix
frozen_embW = Embedding(input_dim=n_words, output_dim=embedding_dim,
                    input_length=max_len,weights=[embedding_matrix],
                    trainable=False)(inptW)

dropW = Dropout(0.1)(embW)
dropCs = Dropout(0.1)(embCs)
drops = concatenate([dropW,dropCs,frozen_embW])

bilstm = Bidirectional(LSTM(units=200, return_sequences=True))(drops)
# Create the hidden layer
hidden_layer = Dense(100, activation="relu")(bilstm)
out = TimeDistributed(Dense(n_labels, activation=Softmax()))(hidden_layer)

model = Model([inptW,inptCs], out)

return model
```

Experiments and Results

In this section, the total number of experiments for the NERC task is being presented. The idea of experimentation followed the gradual increase of the input layers and their embeddings, in order to understand which feature embeddings are the most important ones. Once this work was completed, the addition of the hidden layer and the frozen pre-trained word embeddings were added and tested. When the final architecture was concluded, then the hyper parametrization of the model took place. Since, the available configurations that can be tuned in order to improve the performance of the model are extremely-high in cardinality, only the most important and interesting tests are being presented. It is important to state here that the comparisons between the different architectures and approaches were completed by using the devel set for calculating the accuracy and the f1-score of the network. The final model was also tested on the test set in order to make sure that the model generalizes and it avoids overfitting. The experimentations and their results can be found in Table 1.

Table 1. Experiments and Results

#	Model	Total Parameters	Trainable Parameters	Non Trainable Parameters	Precision	Recall	F1 score
1.	Baseline	1,781,060	1,781,060	0	71.30%	47.80%	50.40%
2.	1.+ lowercase	2,774,960	2,774,960	0	76.1%	50.1%	56.2%
3.	2. +suffixes (3,4,6)	3,590,860	3,590,860	0	72.8%	52.1%	57.8%
4.	3. + POS	3,671,160	3,671,160	0	78.9%	53.8%	60.4%
5.	4. + lemma	4,190,910	4,190,910	0	78.2%	53.3%	59.3%
6.	5. + prefixes (3,4,5,6)	5,387,860	5,387,860	0	79.1%	51.5%	57.7%
7.	6. + more than one capital letter	5,501,460	5,501,460	0	81.0%	53.2%	59.7%
8.	7. + one capital letter, digits, punctuation	5,864,560	5,864,560	0	80.8%	54.5%	61.5%
9.	8. + external resources	6,252,160	6,252,160	0	87.1%	54.4%	61.2%
10.	9. + GloVe	9,634,960	6,732,160	2,902,800	81.5%	57.9%	**65.0%**
11.	10. - lemmas, prefixes (3,4,5,6) + hidden layer (150 units, ReLU), LSTM (300 units, tanh)	9,265,610	6,362,810	2,902,800	62.6%	65.6%	63.7%
12.	Same as 11. with RMSprop instead of Adam	9,265,610	6,362,810	2,902,800	75.3%	47.8%	52.5%
13.	Same as 11. with tanh activation for hidden layer	9,265,610	6,362,810	2,902,800	65.0%	61.3%	59.8%
14.	**Same as 11. but with 200 LSTM units**	**7,975,910**	**5,073,110**	**2,902,800**	**67.2%**	**65.4%**	**65.3%**

After selecting **model 14.** It was tested on the **test set** and the final results indicated a value of **62% for M.avg F1-score**, which can lead to the conclusion that the model is not overfitted and its generalization power is quite good since the **F1-score** on **devel set** was **65.3%**. Final conclusions about the experiments and results are included in the section Conclusion.

Drug-Drug Interactions with Neural Networks

Architecture

Baseline

The baseline model consisted of an input index vector over the word space, which is connected to the embedding matrix of the words and that leads to a one-dimensional CNN layer, which is then flattened and densely connected to a layer of 5 neurons, which represent the classification alternatives: ***advice***, ***effect***, ***mechanism***, and ***int***. As expected, the loss function is categorical cross-entropy and the final layer activation is a softmax.

The total number of different words in our training corpora is 4553, which is the horizontal dimensionality of the word embedding matrix, with every embedding consisting of 100 numbers. The maximum sentence size is 150, and we discarded testing with it given that it seems a proper choice since only 1485 out of 23148 sentences have more than 150 words.

CNN and BiLSTM

The first improvement that appeared obvious to us was to enhance the context retrieval of the CNN one-dimensional layer, which was limited to direct neighbors of every word (kernel size=2). After trying different kernel size values, we found that with 10, the CNN was much more able to recognize the relationships between words the best in a relatively close but informative distance, and the final F1-score was clearly benefited by that even though the increment in trainable parameters of the model was problematic. Bigger kernel sizes than 10 over the baseline model provided the opposite effect: more complexity and too much scope of analysis for the profit of extending the window of the kernel, and worse F1 scores.

Convolutional layers are powerful for learning local patterns in data, while Bidirectional LSTMs are effective at capturing long-term dependencies in sequential data, and in both directions. Because of that, there are many ways of attempting to combine them to obtain enhanced results. We acknowledged the three alternatives:

- CNN's output as the input to the BiLSTM. It may allow the BiLSTM to learn features from the input data that have been learned by the CNN.
- LSTM's output as the input to the CNN . It may allow the BiLSTM to learn features from the input data that have been learned by the CNN.
- The CNN and BiLSTM operate on the input data independently, and their outputs are concatenated and passed to maybe a fully connected layer.

The tests we undertook were focused on the first two options, to see which one could be more suitable for our specific problem. We saw that by a small margin, the LSTM block should be after the 1D convolutional layer.

Early stopping

We realized that the model did overfit way before arriving at the default number of 10 epochs in some tests. That is, the validation loss and validation accuracy started to get worse than expected. Our solution for that was to stop training always a fixed 10 epochs and instead implement an early stopping mechanism with the patience of 5 epochs over the validation loss reduction.

Input and embedding layers

We've seen in the NERC task's results how adding different input index vectors and embedding matrices (which enrich the available information about the words) can boost our model capabilities and potential. For that reason, here as well we decided to try adding PoS, lemma, and lowercase spaces of our sentence words, and given the premise that it will potentially lead to better future results, add complexity to the model as needed from that point onwards. The immediate impact of their addition was low, but we believe that in effect it led to future improvements since most of the time we added further complexity, the results improved steadily. Every embedding trainable instance has a size of 100.

Transformer and BERT

Transformers can be suitable for drug-drug interaction classification in sentences because they effectively capture contextual information and long-range dependencies within the text. Also, as opposed to RNN-based approaches, parallelization of processes is not an issue and can lead to the creation of more complex models. Putting apart temporarily our CNN and LSTM basic model, we decided to check the behavior of a model with only the word embeddings and a default transformer design, with multi-attention heads of size 2, embedding size 100, and the use of positional encoding integrated. From this model, we decided to add trainable parameters complexity and other embeddings to see how the results evolved. We tried with the embedding size of 100,200 and 512 and for different head and dense layers dimensionality values. Results for different transformers' complexity were quite similar, with F1 scores far from the best ones throughout the project. We believe that it is due to the lack of a bigger dataset or the ability to properly parallelize the pipeline, which with our limited resources we couldn't take advantage of, compared to CNN + LSTM approaches.

Another test we did is to add pre-trained word embeddings to our model. BERT (Bidirectional Encoder Representations from Transformers) is a powerful pre-trained contextual representation system built on bi-directionality. We brought the BERT system to obtain the embeddings, but even with deactivated training for the BERT parameters, it was too much for our Collab environment to handle. The alternative was to use the applicable GloVe embeddings that we had applied in the NERC task here as well and see the impact.

Dense layers

Finally, mention that we attempted different model designs with more or less dense layers, choosing the ReLu activation function for all the hidden layers not corresponding to the last one,

which has softmax. Including max-pooling or global average pooling transformations between was not thoroughly tested and can be something interesting for further iterations.

Code

As specified in the requirements, the code appended is corresponding to the final ***build_network*** optimal model creation function and a ***build_network_transformer*** attempt.

Python

```
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()
   n_lemmas = codes.get_n_lemmas()
   n_pos = codes.get_n_pos()

      ####################################
   # Load the GloVe word embeddings
   glove_file = utilsdir + "/glove.6B.300d.txt"  # Provide the path to the downloaded GloVe
file
   embedding_dim = 300  # Adjust the dimension based on the downloaded file
   embeddings_index = {}
   with open(glove_file, encoding="utf8") as f:
       for line in f:
           values = line.split()
           word = values[0]
           coefs = np.asarray(values[1:], dtype="float32")
           embeddings_index[word] = coefs

    # Initialize the embedding matrix with random values
   embedding_matrix = np.random.random((n_words, embedding_dim))

   # Assign the pre-trained GloVe word embeddings to the corresponding words in the embedding
matrix
   for word, i in codes.word_index.items():
       if word in embeddings_index:
           embedding_matrix[i] = embeddings_index[word]

   # word input layer & embeddings
   inptW = Input(shape=(max_len,))
```

```
    inptLC = Input(shape=(max_len,))
    inptL = Input(shape=(max_len,))
    inptPoS = Input(shape=(max_len,))
    embW = Embedding(input_dim=n_words, output_dim=100,
                     input_length=max_len, mask_zero=False)(inptW)
    embLC = Embedding(input_dim=n_lc, output_dim=100,
                     input_length=max_len, mask_zero=False)(inptLC)
    embL = Embedding(input_dim=n_lemmas, output_dim=100,
                     input_length=max_len, mask_zero=False)(inptL)
    embPOS = Embedding(input_dim=n_pos, output_dim=100,
                     input_length=max_len, mask_zero=False)(inptPoS)
    frozen_embW = Embedding(input_dim=n_words, output_dim=embedding_dim, input_length=max_len,
                            weights=[embedding_matrix], trainable=False)(inptW)

    model = concatenate([embW,embLC, embL,embPOS, frozen_embW])

    conv = Conv1D(filters=30, kernel_size=10, strides=1, activation='relu',
padding='same')(model)
    dropout = Dropout(0.1)(conv)  # Adjust the dropout rate as needed

    lstm = Bidirectional(LSTM(200, return_sequences=True))(dropout)

    flat= Flatten()(lstm)
    ag_ppol = AveragePooling1D(pool_size=10)(flat)
    pre2 = Dense(100, activation='relu')(ag_ppol)
    out = Dense(n_labels, activation='softmax')(pre2)

    model = Model([inptW,inptLC,inptL,inptPoS], out)
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

from transformer import *

def build_network_transformer(codes) :

  vocab_size = codes.get_n_words()
  max_len = codes.maxlen
  n_labels = codes.get_n_labels()

  embed_dim = 512 # Embedding size for each token
  num_heads = 4  # Number of attention heads
  ff_dim = 64 # Hidden layer size in feed forward network inside transformer

  inptW = Input(shape=(max_len,))
```

```
embedding_layer = TokenAndPositionEmbedding(max_len, vocab_size, embed_dim)
x = embedding_layer(inptW)
transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
x = transformer_block(x)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Dropout(0.1)(x)
x = layers.Dense(40, activation="relu")(x)
x = layers.Dropout(0.1)(x)
outputs = layers.Dense(n_labels, activation="softmax")(x)

model = keras.Model(inputs=inptW, outputs=outputs)
return model
```

Experiments and Results

Here are the most relevant numerical/tangible representations of model results we obtained throughout our project experimentation. Because of our early stopping mechanism, the number of optimal epochs and training time varies among the different models, but we do not include it here as a meaningful metric because all models ended in a reasonably short and manageable time.

We can observe how we incrementally improved the model (remind that these are just representative results, not all of them), we tried a transformer approach to compare and didn't result in a direct improvement, and how we fine-tuned the last model, which is available in the Code section.

#	Model	Total Parameters	Trainable Parameters	Non-Trainable Parameters	Precision	Recall	F1 score
1.	Baseline	483,835	483,835	0	60.5%	45.6%	51.7%
2.	1D CNN kernel size 10 (optimal)	507,835	507,835	0	66.8%	50.2%	56.2%
3.	2. + Lowercase, PoS, lemma input, embeds	1,137,985	1,137,985	0	77.1%	43.7%	57.2%
4.	3. + BiLSTM + dropout before	1,969,585	1,969,585	0	74.5%	50.3%	59.3%
5.	3. + dropout +	1,785,085	1,785,085		72.4%	53.8%	61.2%

	BiLSTM after CNN						
6.	Transformer (default)	560,057	560,057	0	44.0%	34.4%	36.6%
7.	Transformer (added complexity)	1,279,957	1,279,957	0	52.7%	29.9%	36.5%
8.	Transformer (word embeds=512, 2 heads, ffn of 64 neurons)	6,777,854	6,777,854	0	52.8%	30.1%	36.6%
9.	BERT pre-trained embeddings over baseline	109,550,855	68,615	109,482,240	–	–	–
10.	4. + non-trainable GloVe embeddings and added intermediate dense layer (in code)	3,763,335	2,397,435	1,365,900	74.2%	59.4%	**65.9%**

Applying our final model to the **test** set retrieved an F1 score of 63.5%, 2.4% lower than the best value of the **devel** set.

Conclusion

In conclusion, the experiments conducted for the NERC task revealed several key findings. Firstly, incorporating lowercase information improved the performance, as it allows the model to capture case-specific patterns and variations in drug substance classification. Secondly, the inclusion of suffixes proved beneficial, as they provide additional morphological information that aids in distinguishing different drug substances. Thirdly, the integration of POS tags enhanced precision, recall, and F1 score by enabling the model to consider syntactic contexts and relationships.

However, the addition of lemmas as well as the impact of prefixes on classification accuracy was found to be limited, indicating that they may not provide significant discriminative power in this task. Moreover, the consideration of capitalization patterns, digits, and punctuation marks led to enhanced performance, as these features convey important structural and contextual cues in drug substance identification. Also, leveraging external resources proved highly beneficial, as it provided the model with access to additional knowledge and information, leading to improved results.

Furthermore, the utilization of pre-trained GloVe embeddings significantly enhanced classification accuracy by leveraging semantic representations derived from a large corpus. Architectural modifications, such as the addition of a bidirectional LSTM and a hidden layer with ReLU activation, played a crucial role in achieving a balanced trade-off between complexity and performance.

Finally, the selected model demonstrated good generalization capabilities and avoided overfitting, as evidenced by its satisfactory performance on both the development and test sets. Overall, these findings highlight the importance of feature selection and architectural design in optimizing neural network models for drug substance classification tasks.

One of the overall conclusions is that most of the NERC findings could be translated into the DDI use case. The addition of new input indices and embeddings completing the information available around the words allowed for further improvements of the model, along with adding complexity to the layer's parameters. It was important to amplify the context for the one-dimensional convolutional layer and stagger the hidden dense layer dimensionalities until the softmax layer.

Our transformer approaches didn’t work as well and as fast as expected. We believe one of the reasons may be the fact that they’re designed to overcome traditional recurrent methods in bigger and more extensive projects, with more training corpora available and as many parameters as having a handful of GPUs and proper parallelization advantages would give to us. Nonetheless, also without being able to properly add BERT to our finished trials, we have achieved our initial goal of a 65% in F1-score with the CNN + BiLSTM combination and the complete embedding space joint with an extra ReLu-activated hidden dense layer before the classification layer.