Report 3: NN-based NERC and DDI

Àlex Martorell i Locascio
Pim Schoolkate

April 2022

1 Introduction and report structure

This report aims to present the results and findings using neural approaches to a name entity recognition task, and a relation extraction task.

The report consists of two sections, which constitute the two different tasks, detailing the steps taken during all the process. The goal is not to show only the best results, but try to and list all the options that were tried and why did not produce good results. Observe that the structure of both sections is identical: First, the hyperparameters of the model as it are explored to see how to optimize them with regards to the base model. Second, the best structure of the model is explored, including mentions of methods that did not work. Only with a concrete understanding of the algorithm that the results of an experiment can be interpreted with maximum sense and efficacy. Lastly, these explanations are followed by a detailed description of the feature vector (which ones were tried and kept, and which ones were discarded). A section with the code of the function is also included. Finally, the results are detailed as well as problems arisen.

2 Task 4: Neural Net NERC

This task requires the classification of different drug types within a sentence, using information of the words in this sentence. Four different drug labels were classified: "drug", "group", "brand", and the elusive "drug_n".

2.1 Base Model

Before discussing what hyperparameters were tweaked, the base model is introduced first. As a visual representation, it can be seen in figure 2.1. The first layer of the model are the input layers, which retrieve the data and move it into the embedding layers. The embedding layers create vector representations of the data that is it is being fed (in this case, words and suffixes). These vector representations can be thought of as latent variable representations of the original data. For words specifically, this means that the embedding layers encode some higher level meaning of the words. The embedding layers are trainable, and thus the latent variables (or higher level representations) are tuned to the task at hand and will likely link to whether a word is a drug or not, and what label it could have.

Next, for each embedding layer, there is a drop out layer, which, during the training of the model, randomly deactivates 10% of the nodes in the layer per iteration in an epoch. For the embedding layers, this means that if we have 50 embedding columns, 5 of them will not be used during this training iteration. The benefit of this is two fold. First, it prevents overfitting on the training data as it not always gets the same information for the same sample. This is promotes a more general model that retains a high score on a validation, or even test set. Second, by deactivating nodes, the model cannot depend on only one set of nodes, meaning that more complex relations in the data are stimulated to be found.

After the drop out layers, the embedding layers are concatenated and fed into a bidirectional LSTM cell. The LSTM has quite a complex structure, and since it also used in Task 5, refer to Section **??** for more details.

The output of the LSTM if fed into a dense layer, which serves as the output of the model. The dense layer has four nodes, one for each label that needs to be classified.

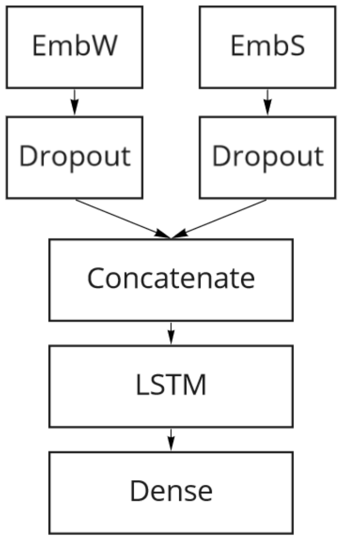

Figure 2.1: The base model

2.2 Hyperparameter tweaking

In all the upcoming tables, the titles or version of the model displays the change with regards to the base model. When hyperparameters, or the structure, are left unchanged, this is not noted in the version and should be assumed to be same as the starting model.

2.2.1 Embedding sizes

An obvious first step is to try and change the size of the embedding layers and the amount of units in the LSTM cells. As this was the first things tried in the project, it was done somewhat haphazardly and thus the order of which hyperparameters were tried is not solid. The results can be found in table 1, where textref helps to refer to the models in the upcoming text.

Increasing the size of the word embedding layer means that the model (B) is able to create more latent variables for each word in the vocabulary. In this case, the size was increased from 100 to 150. At the same time, the embedding size of the suffix was increased from 50 to 100. The resulting model had a slightly increased F1-score indicating that these hyperparameter tweaks might be worthwhile.

2.2.2 LSTM units

The second hyperparameter (C) tweak involved increasing the amount of LSTM units from 200 to 300, meaning that the size of the output of the LSTM is larger, which yielded another percentage of increase on the Macro F1-score. A reasoning behind this, is that the LSTM can represent its output in a larger space, which implies that the dense softmax layer has more information to work with. Because of the increase, it was decided to keep the change as is. Of course, more sizes could have been checked, but due to long training times and limited project time, it was opted to keep it for what it was.

textref	**Name**	**Macro F1**
A	Baseline	51.3%
B	EmbW size = 150, EmbS size = 100	52.2%
C	EmbW size = 150, EmbS size = 100, LSTM units = 300	53.8%
D	EmbW size = 150, LSTM units = 300	53.7%

Table 1: Macro F1 score for kept features on the **devel** dataset

2.2.3 More embedding sizes

In order to validate whether or not the increase in embedding layer of the suffixes made sense, model (D) was run, which reverted the changes to the suffix embedding layer and keeping the other changes. No difference in Macro F1-score between model C and model D could be found, and thus it was concluded that an increased size in suffixes does not result in a better model. The thought behind this is that suffixes tend not to have such a large latent variable space. Especially in this case, the model is only trying to predict whether a word is a drug or not, meaning that suffixes like ”nzine” or ”tamol” get a latent representation of drug (with possibly a distinction between the different drug labels), whereas suffixes like ”ation” or ”ining” get a latent representation of not a drug.

2.3 Combining hyperparameter tuning with model architecture

Although the introduction of this report specifically mentioned that hyperparamater tuning and model architecture would be discussed seperately, it often happens that these processes go hand in hand. Adding a layer requires to play around a bit with the hyperparameters in order to get the right fit (or not). Thus now, the addition of different layers, including the different hyperparameters that were changed for these layers are discussed together with their results. Please note that the ”base” model in this case refers to model (D) in table 1.

2.3.1 Fully Connected layer: the bad try

First, the inclusion of a fully connected dense layer between the embedding layers and the bi-directional LSTM cell is discussed, table 2. Both these model (E F) did not improve the Macro F1-score, but rather, decreased the performance. This makes a lot of sense, as, from a purely conceptual point of view, it does not make sense to use an LSTM cell on top of a fully connected dense layer. By doing so, the dense layer will remove the word per word representation in the embedding layer, and rather make something else out of it. Miraculously the F1-score did not drop to 0% which implies that some (if not almost all) information is retained in the layer. Yet, it does not make sense to keep this change as it will make the model more difficult to interpret and it not significantly increase the results.

textref	**Name**	**Macro F1**
D	Baseline	53.7%
E	Dense = 100, tanh	52.7%
F	Dense = 500, tanh	51.2%

Table 2: Macro F1 score for kept features on the **devel** dataset

2.3.2 Fully Connected layer: the good try

$$\ln(k) = C_0 + \frac{1}{C_1}T + C_2\mathrm{pH} + C_3\mathrm{pH}^2$$

Second, the inclusion of a fully connected layer after the LSTM and before the output dense layer was considered. The rationale behind this inclusion is that an extra dense layer after the LSTM can help interpret the output of the LSTM, which before, was only done by four nodes in the output layer. By doing so, it is possible to ”better” prepare the data for the output layer and choosing an adequate activation function which can help facilitate this. The results for the inclusion of this layer, including the different combinations of layer size and activation function can be found in table 3.

textref	Name	Macro F1
D	Baseline	53.7%
G	Dense = 100, tanh	54.1%
H	Dense = 500, tanh	52.4%
I	Dense = 100, relu	54.5%
J	Dense = 500, relu	51.9%
K	Dense = 100, sigmoid	47.6%
L	Dense = 100, LeakyReLU	54.1%

Table 3: Macro F1 score for kept features on the **devel** dataset

2.3.3 Fully Connected layer: Activation functions

Comparing model D with model G, it is already apparent that the inclusion of a Dense layer at the end of the model has benefits for its F1-score. In terms of size, it is clear that a smaller size of the dense layer is preferred over the large size (please note that these sizes were chosen arbitrarily, and that, when given more time more sizes would have been considered). What is left, is the choice between the different activation functions. Clearly, the sigmoid activation function is not an improvement. Although this is one of the more original activation functions, literature has shown that activation functions such as relu and LeakyReLU are way more effective, which explains the under-performance of the sigmoid function.

From the remaining models (G, I, L), it is difficult to decide on the activation function based on the results, since all of them more or less perform as good. Since the relu gave slightly better results, but due to the problems it has with non-existing gradients on the left side of the curve, it was opted to go for LeakyReLU as this activation combats this gradient problem.

2.3.4 Dropout Layer

Thirdly, a dropout was added on top of the dense layer in order to promote a more generalized layer and prevent overfitting (as discussed above). At the same time, different sizes of the layer were tested, combined with different drop out rates in order to find an optimal combination. The results of these experiments can be found in table 4. In this model, the "base" model refers to model L in table 3.

textref	Name	Macro F1
L	Baseline	54.1%
M	Dense = 100, dropout = 0.1	57.5%
N	Dense = 500, dropout = 0.2	56.6%
O	Dense = 200, dropout = 0.2	56.9%
P	Dense = 50, dropout = 0.1	52.8%

Table 4: Macro F1 score for kept features on the **devel** dataset

An exploration was performed to find the best combination of layer size and dropout rate. It was theorized that with a higher dropout rate, a larger layer would yield better results. However, this turned out not to be the case. Despite this, the dropout layer does seem to help the model to generalize better on the validation set, and thus it was opted to continue with model G as a baseline model.

2.3.5 Tweaking LeakyReLU

Another small change was made to the dense layer, after the inclusion of the dropout layer. Namely, setting `alpha` of the LeakyReLU to 0.1 (previously 0.3). `alpha` refers to the negative slope coefficient, meaning that reducing its value, it resembles a relu activation more. This was found to be more effective, as can be seen in table 5.

textref	**Name**	**Macro F1**
M	Baseline	57.5%
Q	LeakyReLU(alpha = 0.1)	59.7%

Table 5: Macro F1 score for kept features on the **devel** dataset

2.3.6 Optimizers

At this stage, different optimizers were tried, but unfortunately, these results were not properly archived and thus have to be discussed without evidence. The Adam optimizer is widely known to be the most optimal and best optimizer for training deep neural networks. However, some suggest that, especially for networks that include a LSTM cell, RMSprop might improve the learning. In the case of this project, it was found that the Adam optimizer performed best. Tweaking β_1 and β_2 could have been interesting to try, but due to the scope of this project and the limited time at hand, it was decided to not further investigate the optimizers.

2.3.7 Pre-trained Glove Embeddings

As discussed before, the embedding layers learn a way to represent the words in different latent variables, that the model finds itself. However, pre-trained embedding layers already exist, which can be integrated in a model for better performance. This has the main benefit that the model does not need to train this layer much, but rather, only needs to tweak it slightly, or even simply use it as is, and interpret latent variables produced by the pre-trained embedding. One downside of the Pre-trained embedding is that most drug names are not part of the vocabulary of the embedding, meaning that for these, no preexisting embeddings exist. Thus, for these drugs, the model has to find its own embedding. This however should not be a problem, as the non-drug words will be rich in embedding and thus the model can learn to ignore these words.

textref	**Name**	**Macro F1**
Q	Baseline	59.7%
R	Glove = 50	58.2%
S	Glove = 100	62.4%
T	Glove = 200	61.6%
U	Glove = 300	58.6%

Table 6: Macro F1 score for kept features on the **devel** dataset

In this project, Glove embeddings were used as pre-trained embeddings. Glove provides four possible output dimension shapes: 50, 100, 200, and 300. All of these were tested to see which one gave the best result, of which the results can be seen in table 6. Clearly, the addition of pre-trained glove embeddings with dimension size one hundred improves the model significantly.

2.3.8 L2 regularization

Although the model does not seem to overfit (untill now), the addition of regularization was tried to see if this would improve results. Regularization works by penalizing large weights, which means that all weights will more or less be in the same range. Including regularization often implies longer training time, and thus it is expected that the results after 10 epochs are not going to be as good as the previous best model (S). For this project, only L2 regularization was used (L1 and L2 roughly do the same, with the difference being that L2 squares the weight, meaning that the penalty is larger). The results can be found in table 7.

textref	Name	Macro F1
S	Baseline	62.4%
V	L2 regularization, 10 epochs	30%
W	L2 regularization, 30 epochs	37.6%

Table 7: Macro F1 score for kept features on the **devel** dataset

As expected, the time it takes to train the model is increased. However, it was not anticipated that it would have such a heavy effect, as even after 30 epochs, the model is nowhere near the previous results. Thus L2 regularization is not applied to this model.

2.4 Adding Features

Now that the model structure and the effect on its effectiveness has been studies, the attention will be diverted to adding different features. From the previous project, it was already clear that the inclusion of Part-of-Speech (pos) tags could help improve the model, and so naturally, this was the first thing that was tested.

2.4.1 POS-tags

It was opted to introduce the pos-tags to the model by means of an embedding layer. This would help the model to classify what pos-tags are more commonly associated with drugs, but also give the model opportunity to learn different interactions between the pos-tags, which, as was shown during lectures and previous project, can have a significant effect on the ability to classify words by any model.

textref	Name	Macro F1
S	Baseline	62.4%
X	pos-tag embP = 100	66.9%
Y	pos-tag embP = 50	66.3%

Table 8: Macro F1 score for kept features on the **devel** dataset

The results of the addition of the pos-tags can be found in table 8. As can be seen, the improvement of the model goes up by roughly 4% indicating that the addition indeed is a good choice. The larger embedding size yields a slightly better result, and thus it was opted to continue with this one.

2.4.2 Different feature architectures

Next, together with the lower case word form, all other features were added at once (they will later be discussed one by one). It was done like this, because of three possible architecture options for the extra features. The lower case word forms were added with their own input and embedding layer.

2.4.3 2-dimensional convolutoinal layers

Firstly, all features could be added as one vector, which would add an extra dimension to the input and would have to be reduced in order to combine it with the rest of the data. In order to achieve this, a series of 2D-convolutional layers was used to reduce the shape of the input from (x, emb, max_len, extras) to (x, emb, max_len) (where x refers to the amount of training data, emb refers to output length of the embedding layer, max_len is the max length of the sentences, and extras is the amount of features added (at that moment)). The convolutional layers were configured to have a stride of 1 (we do not want to skip a feature), a kernel size of 3, padding set to "same", and a LeakyReLU activation function. The filters would be decreased each layer and thus for the 3 convolutional layers, the first one had 5 filters, the second 3, and the third had 1 filter. After, the output was reshaped to the compatible shape of the other embedding layers, concatenated with these, and fed into the LSTM cells just like the other data.

Clearly, a few major mistakes were overlooked. Firstly, by using a 2 dimensional convolutional layer and a kernel size of 3, only features that are next to each other get combined. Although they all, get combined to one single value, it is still meaningless, as the filters would not be able to detect any sort of pattern (which is what convolutional layers are good at). Secondly, and even worse, is that already in the convolutional layer, the features of all words, influence all other features directly when next to each other, and indirectly in the following layers. Thirdly, by setting padding to "same" in order to keep the length of the sentences constant, the features of words at the start and at the end of the sentence might be weakened, which is not desirable either. Despite these errors and shortcomings, the model still improves by 2% as can be seen in table 9.

textref	**Name**	**Macro F1**
X	Baseline	66.9%
Z	Conv 2D	68.9%
A2	Conv 1D	67.3%
B2	Separate inputs	71.4%

Table 9: Macro F1 score for kept features on the **devel** dataset

2.4.4 1-dimensional convolutional layers

Secondly in the list of ways to try to add the extra features, the approach was the same, but now with a 1-dimensional convolutional layer. The sequence of layers, similarly to the 2-dimensional variant, had a kernel size of 3, a stride of 1, padding set to "same", and the LeakyReLU as activation function. The filters also followed the 5, 3, 1 pattern. This solved the first two problems described for the 2-dimensional convolutional layer, which is expected to have a great effect on the performance of the model. Weirdly enough, the 1D approach actually performs worse than the 2D approach. It is not exactly clear why this would happen, but it surely prompted to try the last method of including all features.

2.4.5 Each feature separately

The third and last method was, on hindsight, the most obvious one; add each feature as a separate input layer. This was a considerably more work to code and put together compared to the other 2 methods (which might explain why these were tried first), but when completed resulted in a far better result than the convolutional approaches, as can be seen in table 9. Each feature was given its own embedding layer with appropriate sizes for that feature (i.e. the length of a word might need 2 instead of a 100 embedding dimensions). Clearly, it makes more sense to represent each feature on its own, instead of one value representing all, and thus the better results were not surprising.

With these results, it can be concluded that model B2 is the best model, and, since the project statement asked for a F1-score of roughly 70%, the project can be considered done. However, since it is not satisfactory to not know what features improve the model, and which do not, each feature was tested on its own. In here, we will compare the results with those of the previous assignment on the same task. The results of both can be found in table 10. In this table, the number behind the feature indicates the embedding dimension size.

textref	**Name**	**Macro F1**
X	Baseline - No extra features	66.9%
B2	All features	71.4%
C2	Lower Case words(100)	68.4%
D2	Lower Case words + Word Length(5)	69.6%
E2	Lower Case words + Word Length + Upper/Lower cases(2)	66.9%
F2	Lower Case words + Word Length + Has Dash(2)	70.2%
G2	Lower Case words + Word Length + Has Dash + Syllables(5)	68.8%
H2	Lower Case words + Word Length + Has Dash + Word Shape(50)	70.0%
I2	Lower Case words + Word Length + Has Dash + Is plural(2)	68.0%
J2	Lower Case words + Word Length + Has Dash + Lookup(2)	70.6%

Table 10: Macro F1 score for kept features on the **devel** dataset

2.4.6 Lower Case words

Compared to model (X) the baseline model where no features are added, the inclusion of the lower case words seems like good choice, improving the model by 1.5%. It should be noted that the embedding for this layer was initialized using the glove pre-trained embeddings. The reasoning for the lower case words to improve the model is simply that it removes the difference between words that should be the same word. A word that starts a sentence is written with a capital letter, and thus is different from the same word in the middle of a sentence. Doing this reduces the vocabulary from 9687 words with capitals to 8350 without capitals. Because the inclusion of lower case words improved the model, they are used in the upcoming models too.

2.4.7 Word length

Next, the word length seems to improve the models performance too. This is logical as there are drugs like *(2R)-2-[(1R)-1-[(2Z)-2-(2-Amino-1,3-thiazol-4-yl)-2-(methoxyimino)acetyl]amino-2-oxoethyl]-5-methylene-5,6-dihydro-2H-1,3-thiazine-4-carboxylic* which are frankly hard to miss based on their lenght alone. Including the length thus makes sense, and, supported by the better model performance, was kept in the model.

2.4.8 Upper vs lower casing

Another feature consisted of checking if the word is upper or lower case, or a combination of the two. This could help identify brands like ZYPREXA which are written completely in upper cases. From the previous project, this inclusion was found to bring significant improvement to the model. However, in this project, this improvement was not found. Rather, the model performed worse, which does not entirely make sense. The most obvious hypothesis would be that all words with upper or lower, or a combination were already predicted correctly, or that the model did not yet converge. The training and validation loss were also considered, but no overfitting on the training model was found. Thus the upper and lower case feature was not included.

2.4.9 Has dash?

Now, the "Has Dash" feature is considered. This feature simply states whether or not the word includes a "-". The rationale behind this is simply that drugs, like the long one described above, often include dashes. The model, with this feature included, improves by 0.6%.

2.4.10 syllables

The amount of syllables in a word can also identify a word as a drug. Many drugs have a rather large amount of syllables, which might be of use to the model. Unfortunately, this does not increase, but decrease the performance of the model. Likely, this is because the amount of syllables is highly correlated with the length of the word, and thus syllables does not provide new information for the model.

2.4.11 Word shape

The shape of the word might also help identify drugs. The word shape recreates the word by replacing each number with "0", each upper case letter with "A", each lower case letter with "a", and every other character like "*&#!" with "O". Certain shapes might be reoccurring for certain drugs and thus it might help identify them. In total, 718 different shapes were found for the vocabulary. Adding the shapes of the words, neither improved or worsened the model, and thus it was opted to exclude the word shape.

2.4.12 Plurality

It was hypothesized that whether a word is plural or not could give a weak hint towards the word being a drug group. However, as many other words can be plural it was not expected that this would improve something. This was apparent from the results as the model performed worse with this input.

2.4.13 Look up table

Lastly, a lookup table of known drugs was used and words which were found in this lookup table were tagged with being in the lookup table. This addition seemed to make a minor improvement, although one would expect the improvement to be much higher. One reason

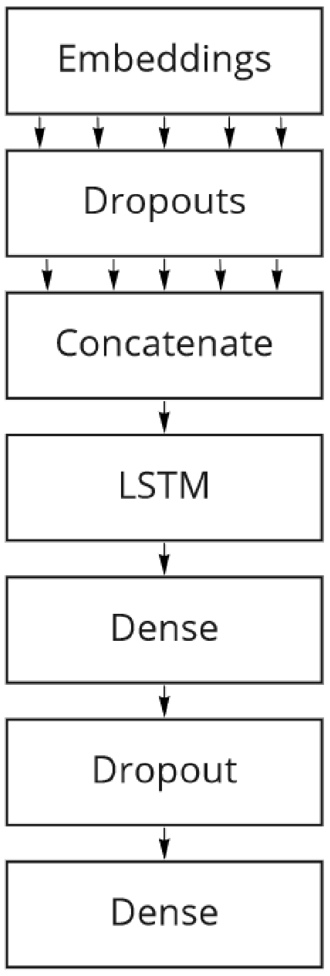

Figure 2.2: The final model

for this could be the fact that all drugs that are found in the lookup table were already predicted correctly, or that the model did not have enough training time to train the embeddings in order to recognize the "is in lookup" feature as a indicator that it is a drug. A last possibility could be that some drugs that are in the lookup table are not in fact drugs in the training set, meaning that the model gets confused. This is more of a data-issue that would be easily solved by checking both data sources.

2.4.14 Everything together

When looking at table 10 it becomes clear that all features combined still produces the best model with an F1-score of 71.4%. It can thus be assumed that the model is able to combine some features discarded in the elimination process to come to more accurate predictions. A better process would have been to one by one eliminate the features, seeing which ones can be excluded and which one not. However, this was not done and due to the limited time was not repeated.

2.5 Conclusion

This project successfully managed to produce an neural network, together with a set of features, that is able to classify which words in a sentence belong to either one of the four lables, described in the introduction. The final model layout can be seen in figure 2.2.

3 Task 5: Neural Net DDI

3.1 Introduction to Task 5

Task 5 is an extension of Task 3, as the aim is to try to solve the same problem but with newer state-of-the-art Neural Network (NN) techniques. The objective is to detect drug-to-drug interactions. These are labeled either as: “Effect”, “Advise”, “Int” or “mechanism”.

The main difference with respect to Task 3 is the techniques used. Feature extraction was one of the main objectives of Task 3, which are low level syntactical or semantical features that are considered for the classification model. Some examples of these features were Lowest Common Subsumer, Syntactic Paths… among many others. For this project, the emphasis will be mostly on the NN in contrast with the emphasis on feature extraction in task 3. Therefore, it is weakly assumed that the information contained in the provided features is sufficient for the model to reach a F1-score of 65% or higher. Thus, regarding the input of the model, toying with the embedding layers seemed sufficient.

The different steps in this task were mostly dedicates to improving the general architecture of the model. Experimentation was done with convolutional layers, Long-Short Term Memory (LSTM) cells, and transformers.

A technical note must be pointed out: In absence of functional code, the XML files were previously parsed into a “.pck” file which can already be fed into the Codemaps class. Observe that this limits the number of embeddings that can be used: The only available are the ones that have been previously coded. This is because the Stanford Core NLP Parser is used, and some its potential is not put to practice in the final “.pck” file. Although this was later fixed, it was not deemed necessary to change the approach of the above mentioned due to the results obtained.

The Macro F1 score will function as an assessment of the global accuracy of the model. As back-propagation, the random initialization of the weights, and the random order of the data, adds randomness to the model, and such F1 score will be different each time the whole program is executed. Although a seed could be set, this would still add ambiguity as some model configuration might work better on this particular seeds than others. Thus, the model is run 3 times and the F1-score is averaged to get a more truthful result, in essence performing a small Monte Carlo simulation to get the true mean. This is justified as it was observed that the same model configurations could yield substantially different F1-scores on different training and testing runs.

3.2 Base Model

The base case only considers the following layers: It consists of an embedding of each word in a sentence. Note that in the class `Dataset`, indexes are created for each sentence: Vectors of size 150 are the output of this coding process, meaning that if the sentences do not reach this max length of 150, padding is added by setting the components of the vector to 0.

It is worth to take a close look at the dimensions of outputs of the different layers to gain some familiarity with the concept of tensor. First, the summary of the model prints the following dimension for the input layer: [(None, 150)]. Note that the in the model each of the outputs have this first "None" dimension. This refers to the batch size, which is specified as the model is run. The **embedding** layer embeds words into numeric vectors. If the Embedding dimension is set to 100, This means that the output of this layer is a tensor [(None, 150, 100)] , meaning that each word in a tensor now has a vector of 100 components representing it. The embedded tensor is fed into a 1-dimensional **convolutional layer**. The in `Tensorflow` denoted `Conv1D` [1] function in Keras which accepts several parameters: The number of outputted filters, the kernel size, the stride and the activation function. These parameters will be discussed later. The output dimension is: (None, 150, 30). The last dimension is the number of filters per word. The filters have dimension (kernel size, embedding dimension, Output dimension of the convolution) meaning that they are a tensor. Observe that there is also a `padding` parameter, which is set equal to *same*. This is not related to the padding added to the end of the input sentences. This is strictly so that the size of the input is preserved after the convolutional layer. There are two options: "valid" and "same". The first one means that padding is added if the vector is not divisible by the kernel size. In the latter one, the vector remains untouched and the last word will not considered.

The **Flatten** layer is a reshaping layer. It reduces the current dimensions to 1, by setting one vector after the other. This means that the output dimensions are (None, 4500) for the base case, since $150 \times 30 = 4500$, meaning that one long vector is left. This should be one of the last steps, since then it is ready to be fed into a final Dense layer.

The **Dense** layer is the final step in the model, a standard feed-forward NN. Similar to the NERC task, the `n_labels` is the output dimension of the NN. In this task, the objective is to predict interactions, 4 kinds of them. Hence, the final output has just 4 neurons with a softmax activation function.

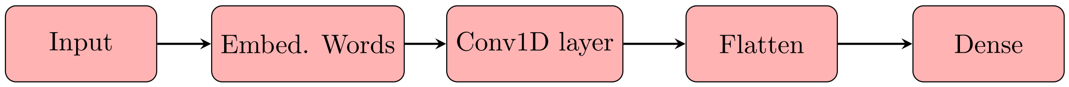

Figure 3.1: Layers in the Base Model

Observe that the **base model** gives an average Macro F1 score of 52.83%.

3.3 Hyperparameters, Model architecture, New Layers

This section is devoted to studying in depth the different options tried to improve the overall prediction score of DDI task (Macro F1). If needed, theoretical remarks will be made to clarify reasoning behind the choices made. To summarize, the following will be the focus of this report: embeddings, number

[1] A lot of more information can be found for Conv2D, which are more popular since they are used for visual recognition. The main difference is that output filters are matrices instead of vectors.

and kind of layers (Dense, Drop, Pooling...), CNN and LSTM architectures (and a combination of both), pretrained embeddings, and transformer attention blocks.

3.3.1 Embeddings

The first thing that was tried, is changing the embedding dimensions, as well as what kind of data is fed into the embeddings.

In [2] it says that the dimensionality of an embedding is not a well understood hyper-parameter and remains an open problem. It affects the quality of the word vectors as a small dimensionality is not capable of expressing the information contained in all the words of a sentence, yet, high dimension embeddings contradicts the own concept (reducing the amount of variables) and may also suffer from overfitting. The base case sets an embedding dimension of 100 for words. Different dimensions are tried, as well as different embedding vectors (lowercase words, Part of Speech (POS) tagging and lemmas). The scores are summarized in Table 11.

Embeddings	Dim. per Embedding	Total Embedding dim.	Macro F1 average
Words	200	200	53,77%
Words	300	300	50.00%
Words, PoS	200	400	40.7%
Words, Lemmas	200	400	53.7%
Words and Lowercase	200	400	53.57%
Words, Lowercase, Lemmas	200	600	51.30%
Words, Lowercase, PoS	200	600	44.13%

Table 11: Different scores for Embeddings and sizes.

Observe that just Word and Word + Lowercase , Word + Lemmas give good embedding results. It is clear that lemmatization and PoS Tagging do not add further information to the word embedding. A possible reason for this might be that lemmas PoS tags do not introduce new information, but rather contradict some of the information, making it hard for the model to find a pattern. Another reason could be that the model simply does not have enough time to converge, since the complexity of the model has increased. This, however, was not checked and thus remains somewhat of a mystery or black box.

3.3.2 Filter and Kernel size

Note. There is a common terminology confusion between the output of the filter and the actual filter applied in the convolutional operation. In this report, the filter is the output and the parameters involved in the convolutional operation are called ”filter parameters”.

The CNN layer has several parameters that can be altered. One of the main ones is the number of filters. Recall that in the base model the number of filters is set to 30 As stated before, the output of the CNN is (None, 150, 30). For each word in a sentence, there are 30 filters computed. The filter size is determined by the kernel size parameter, set to 2 by default. Note that the filter is actually a tensor and its dimensions are (kernel size, word embedding dimension, number of filters). In other words, the base model outputs the following tensor:

$$\text{Sentences} \times \text{words} = 150 \times \text{embedding size} = 200 \times \text{filters} = 30 \times \text{filter size} = 2$$

In a Conv1D network, the intuition of the filters is as follows: given a 150 size vector with 200-length embedding size per word, the filters are vectors of weights of size (200×1) (given that the kernel size is 1). Each weight gets multiplied with its corresponding input, after which the output is summed and transformed using an activation function which results in a single value. When choosing more than one filter (e.g. 30) the output will be that many values for each word in the sentence. When setting kernel size to 3, the filter is also applied to the embeddings for the words around it, which allows for interaction between words to be captured.

Let $i = 1, N$ be the number of words in a sentence. Let h be the kernel size and k the embedding dimension (For the base case this means $i = 150$, $h = 2$ and $k = 200$). Now if $w \in \mathbb{R}^{hk}$ is a vector with the parameters of the filter (sometimes also called a feature map), the filter output (i.e. the number of is *in general* $c = [c_1, \ldots, c_{N-h+1}]$ (This is one filter, remember that 30 are required in the base case), for which each c_i:

$$c_i = f(w \cdot x_{i:i+h-1} + b) \quad \forall i$$

where f is the activation function, and b is the intercept. In the whole of Task 5, the activation function is set to ReLU ($f(x) = \max(0, x)$). ReLU works especially well to increases non-linearity in the filter/feature map.Recent studies have demon-strated that ReLU performs better than sigmoid or tanh when overcoming the Vanishing Gradient Problem. Note that this is mathematically coherent since $w \times x_{i:i+h-1}$ is the inner product. To sum it up, Figure 3.2 summarizes graphically the convolutional operation. [2]

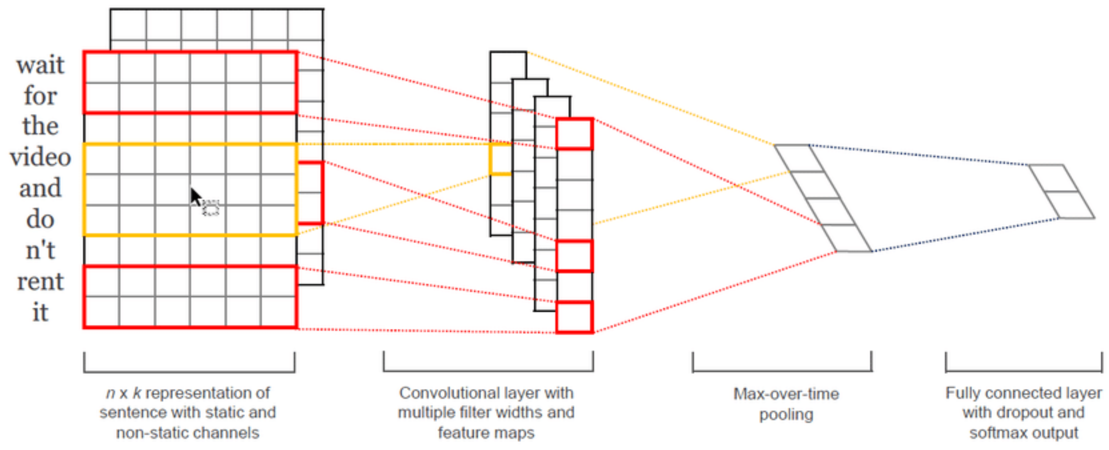
wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 3.2: The Convolutional Operation

Observe that the stride = 1 and should be kept to that. Stride > 1 would mean that some components of a vector are skipped, which is a very bad idea for a drug-drug interaction task.

Configurations	Macro F1
filters = 50	50.30%
filters = 10	51.47%
filter = 30 and kernel size = 3	55.90%
filter = 30 and kernel size = 4	54.07%

Table 12: Conv1D Parameter configurations

[2]Image retrieved from: https://richliao.github.io/supervised/classification/2016/11/26/textclassifier-convolutional/

3.3.3 Dense and Dropout layers

Dropout and Dense layers are tested here. For more information on the concepts behind these layers, refer to **??**. Table 13 show the results for these features.

Layers	Macro F1
Dense(100)	52.40%
Dropout(0.1) after Dense(100)	53.00%
Dropout(0.1) after Embedding and Dense	52.80%
Dropout(0.2) after Embedding and after Dense	53.63%

Table 13: Dense and Dropout layers

Observe that all the results do not really improve the Macro F1 score, but rather worsen it. However, they will be absolutely necessary when we complicate the architecture with more layers (i.e. LSTM, LSTM + CNN).

3.3.4 Max Pooling

Pooling is specific to Convolution Neural Network architectures. It performs a downsampling of a given input to reduce the number of parameters. There are two basic kinds of pooling layers: Max Pooling and Average Pooling. Max Pooling (in a 1D case) accepts `pool _size` and `strides=None` parameters and takes the maximum value over a window of fixed size (Average Pooling takes the average over this same window). For example, if the following vector $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ should be the input and the window size is 2 the result would be:

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) \xrightarrow{\text{MaxPooling}} (2, 4, 6, 8, 10)$$
$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) \xrightarrow{\text{AveragePooling}} (1.5, 3.5, 5.5, 7.5, 9.5)$$

Note that the mere technicality: padding = "valid" and padding = "same" can produce slightly different outputs (as explained before). What is the intuition behind a Pooling layer, especially a MaxPoolingLayer ? Recall that the CNN layer has produced a feature vector or a filter c. The main idea of $\tilde{c} = \max(c)$ is to **capture and retain** special features (high values). Observe that if the window is of size 2, what is being done is generating $\tilde{c}$ of dimension 75 (Down from 150 words). If the decision is to actually take $\tilde{c} = \max(c)$ this is called Global Average Pooling, and leaves only 1 value.

Table 14 below lists Maximum and Average Pooling layer results with different kernel combinations. They are considered together with the kernel_size because they are somewhat dependent on it. (The kernel size determines the feature vector).

Layer configuration	Macro F1
MaxPooling(pool_size = 2) and kernel_size = 4	56,17%
MaxPooling(pool_size = 2) and kernel_size = 3	55,30%
MaxPooling(pool_size = 3) and kernel_size = 3	51,03%
AveragePooling(pool_size = 2) and kernel_size = 3	55,43%
AveragePooling(pool_size = 2) and kernel_size = 4	55,97%

Table 14: Pooling configurations

Note that most configurations produce similar results while slightly improving the models from Section 3.3.2.

3.3.5 Pre-Trained Embeddings

Similarly to Task 4, Pre Trained Embeddings are tried in order to improve the performance of the model.

PreTrained Embeddings	Macro F1
Without Extra Dense Layer	54.4%
With Extra Dense and subsequent Dropout	58.2 %

Table 15: Pre-trained embedding results

Notice that an Extra Dense Layer now does make a significant difference. This shall be kept for further configurations, as seen in the next section.

3.3.6 CNN, LSTM, CNN+(Bi)LSTM architecture comparison

Task 4 uses an LSTM (c.f. Section 2.2.2) for NERC. This is because NERC could have some long-term dependencies that might escape standard RNN models. Also, the standard RNN has trouble with small gradients, which occur when backpropagating to determine parameters.

The structure of LSTM has been covered in depth in Theory Lectures, so just a summary is provided here. It consists of complex cells that has a hidden state $h^{(t)}$ and a cell state $c^{(t)}$. [1] shows that recent applications have used the CNN + LSTM combination for a variety of applications. It is worth a try.

Observe in the code that the only parameter to tweak is the number of units. This is the dimensionality of the output space. By default, it is set to 300. (i.e. the output will be vector of dimension 300).

In Task 4, the Keras `LSTM()` function outputs a vector for every input word. In Task 5, the interest is just the final vector, since this a sentence classification task. That is why the parameter `return_sequences = True` is removed from the LSTM method.

Observe that since it produced good results, an intermediate Dense layer is considered between the CNN or LSTM layer and the final Dense layer. A Dropout after the intermediate Dense layer is also part of the architecture. All the results are shown in Table 16.

Architecture tried	Macro F1 average
Best CNN configuration	58.2%
LSTM (PreTrained)	61.50%
CNN + LSTM (No pretrained)	57.07 %
BiLSTM (with Pretrained Embeddings)	68.50%
CNN + BiLSTM (Pretrained Embeddings)	69.63 %

Table 16: Comparison between CNN, LSTM architectures

Figure 3.3 shows the different layers in the final model (CNN + BiLSTM). Clearly, the LSTM with pre-trained embeddings outperforms the convolutional architecture (which also had pre-trained embeddings). Apparently, some representation of the sentence that indicates the type of interaction is propagated through the LSTM, which is not captured by the CNN, only relating words that are directly next to each other. Obviously, and as discussed and explained why in task 4, the models without pre-trained embeddings perform worse.

Curiously, the BiLSTM outperforms both the convolutional and the single LSTM approach. Likely, the interaction between the drugs can sometimes only be found when approaching the sentence from the back to the front. Another explanation might be that the model finds evidence for one type of interaction when going from left to right, but finds a different interaction when going from right to left. With extra evidence, the model might thus be able to better distinguish between different interactions.

Furthermore, when combining the BiLSTM with the convolutional architecture, an even better result is obtained. It is hypothesized that the CNN part helps the BiLSTM part better distinguish between interactions by providing more evidence, as described above.

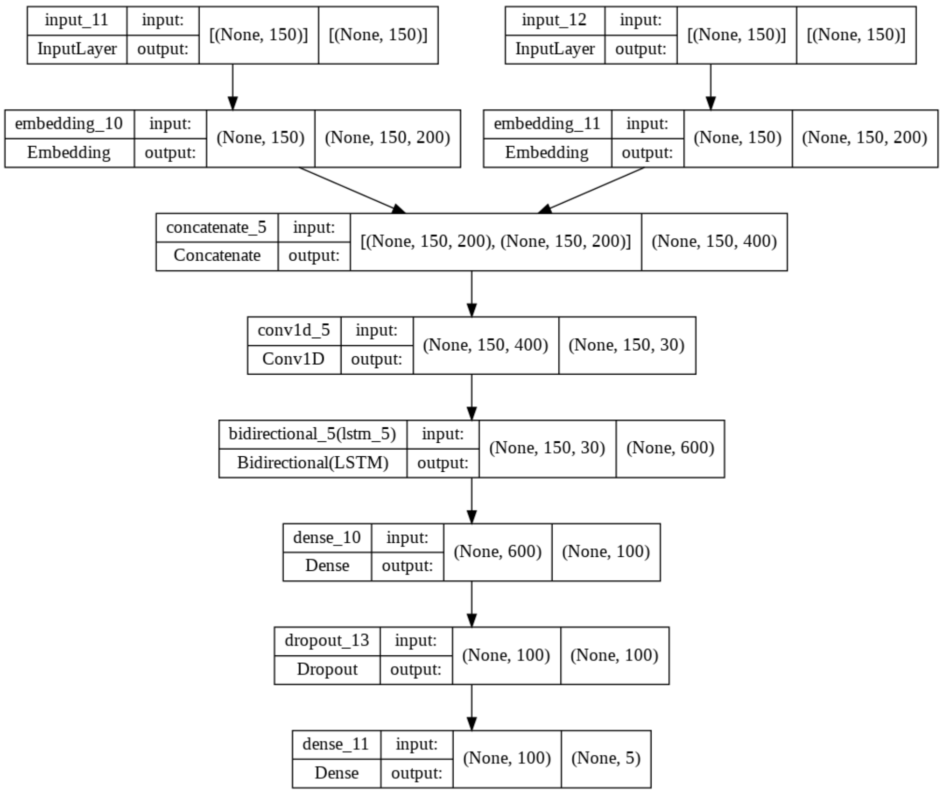
input_11 | InputLayer | input: | output: | [(None, 150)] | [(None, 150)]
input_12 | InputLayer | input: | output: | [(None, 150)] | [(None, 150)]
embedding_10 | Embedding | input: | output: | (None, 150) | (None, 150, 200)
embedding_11 | Embedding | input: | output: | (None, 150) | (None, 150, 200)
concatenate_5 | Concatenate | input: | output: | [(None, 150, 200), (None, 150, 200)] | (None, 150, 400)
conv1d_5 | Conv1D | input: | output: | (None, 150, 400) | (None, 150, 30)
bidirectional_5(lstm_5) | Bidirectional(LSTM) | input: | output: | (None, 150, 30) | (None, 600)
dense_10 | Dense | input: | output: | (None, 600) | (None, 100)
dropout_13 | Dropout | input: | output: | (None, 100) | (None, 100)
dense_11 | Dense | input: | output: | (None, 100) | (None, 5)

Figure 3.3: Keras sketch of the final model with dimensions

3.3.7 Transformers

One of the main differences with Transformers is that it calculates the attention (An inner product) between all word combinations and the position of the word is lost. The `TokenAndPositionEmbedding()` creates two embedding layers, one for the words and one with the position of each word. The method

In the encoder part of the Transformer operation one can distinguish 2 main parts: The Attention scores and the Feed Forward NN. Unfortunately, is way over the scope of this project to try and understand how the attention scores are computed exactly and comprehend all the different layers in the complex architecture.

The architecture tried is simple and standard : The two transformer blocks, and a GlobalAveragePooling layer at the end with a Dropout(0.1). Transformer results are summarized in Table 17.

Funnily, transformers performed exceptionally bad. It was hypothesized that as the transformer is a highly complex and difficult to train model, it had not yet converged. Thus, increasing the epochs and looking at the training history is a straightforward step and is visible in figure 3.4. As expected,

Parameters	Macro F1 average
embed_dim=100, ffn_size = 64 , num_heads = 4	44.1%
embed_dim=200, ffn_size = 64 , num_heads = 4	29.8%
embed_dim=200, ffn_size = emb_dim*2 , num_heads = 4	36.3%

Table 17: Transformer results

the training loss and accuracy are still improving considerably, confirming the hypothesis that more training might be needed. However, more curiously, the validation accuracy and loss does not seem to be effected by the training. It is not very clear why this is the case, but one idea might be that the transformer is so powerful that it immediately overfits on the training data. Therefore, it is stipulated that to train a transformer from scratch a massive amount of data, time, energy, and computing power is needed to get a meaningful model. Such example is the BERT model, which would yield amazing results, but was not tried due to time constraints.

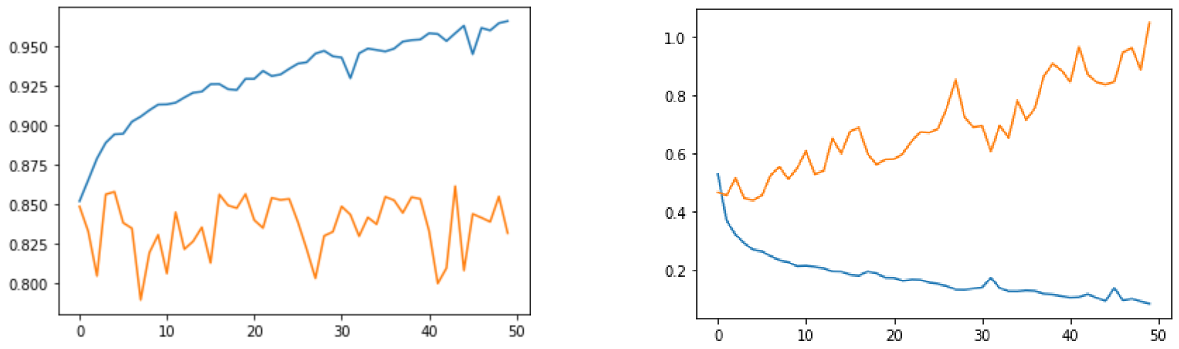

Figure 3.4: Loss (right) and accuracy (left) of the transformer during training. Training: Blue, Validation: Orange

3.4 Final model and conclusion

With a substantial difference, the CNN + BiLSTM with an intermediate Dense layer is the best option. It is clear that CNN works well and many different parameter combinations (number of filters, kernel...) do improve the base model. It was observed that LSTM and especially BiLSTM (which keep previous and later state information) increased the overall score almost to 70%. Also, an intermediate Dense Layer is necessary as most classification tasks perform better since it is to process information in different steps as the number of output neurons is reduced.

References

[1] Md. Zabirul Islam, Md. Milon Islam, and Amanullah Asraf. A combined deep cnn-lstm network for the detection of novel coronavirus (covid-19) using x-ray images. 2020.

[2] Zi Yin and Yuyuan Shen. On the dimensionality of word embedding. 2018.

3.5 Code for Task 4

We made a lot of changes to get everything working and to try out different things. Lost of things are commented out, which reflect the different things tried and played with. The codemaps.py and dataset.py were also altered, but just extended the already existing architecture with more features and thus are not shown here.

3.5.1 train.py

```python
#! /usr/bin/python3

import sys
from contextlib import redirect_stdout

from tensorflow.keras import Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import LSTM, Embedding, Dense, TimeDistributed, Dropout,
    Bidirectional, concatenate, Lambda, Constant, LeakyReLU

import numpy as np

from dataset import *
from codemaps import *

def create_embeddings(word_index, emb_dim):
  embeddings_index = {}
  with open(BASE_DIR + f"/Embeddings/glove.6B.{emb_dim}d.txt") as f:
      for line in f:
          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 = len(word_index)
  hits = 0
  misses = 0

  # Prepare embedding matrix
  embedding_matrix = np.zeros((num_tokens, emb_dim))
  for word, i in 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.
          # This includes the representation for "padding" and "OOV"
          embedding_matrix[i] = embedding_vector
          hits += 1
      else:
          misses += 1
  print("Converted %d words (%d misses)" % (hits, misses))
  return embedding_matrix


```

```
def build_network(codes):
    # sizes
    n_words = codes.get_n_words()
    n_sufs = codes.get_n_sufs()
    n_labels = codes.get_n_labels()
    n_pos = codes.get_n_pos()
    n_lc_words = codes.get_n_lc_words()
    #  n_extra = codes.get_n_extra()
    max_len = codes.maxlen
    emb_dim = 100

    inptW = Input(shape=(max_len,))  # word input layer & embeddings
    embW = Embedding(input_dim=n_words,
                     output_dim=emb_dim,
                     input_length=max_len,
                     embeddings_initializer=Constant(create_embeddings(codes.
    word_index, emb_dim)),
                     mask_zero=True)(inptW)

    inptLW = Input(shape=(max_len,))  ## Lower case word input layer & embeddings
    embLW = Embedding(input_dim=n_lc_words,
                      output_dim=emb_dim,
                      input_length=max_len,
                      embeddings_initializer=Constant(create_embeddings(codes.
    lc_word_index, emb_dim)),
                      mask_zero=True)(inptLW)

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

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

    ## ADDING EXTRA FEATURES STARTS HERE...
    ## They are all commented out in order to test different configurations..
    ## could be more optimal with gridsearch config...

    # Lenght of the word
    #  inptWordLength = Input(shape=(max_len,))
    #  embWordLength = Embedding(input_dim=codes.get_n_WordLength(),
    #                            output_dim=5,
    #                            input_length=max_len,
    #                            mask_zero=True)(inptWordLength)
    #  dropWordLength = Dropout(0.1)(embWordLength)

    #  # If the word is in UPPER cases, lower cases, or a CoMbInAtIoN
    #  inptUpperLower = Input(shape=(max_len,))
    #  embUpperLower = Embedding(input_dim=codes.get_n_UpperLower(),
    #                            output_dim=2,
    #                            input_length=max_len,
    #                            mask_zero=True)(inptUpperLower)
    #  dropUpperLower = Dropout(0.1)(embUpperLower)

    #  ## If the first letter of the word is in UPPER case
    #  inptFirstCap = Input(shape=(max_len,))
    #  embFirstCap = Embedding(input_dim=codes.get_n_FirstCap(),
    #                          output_dim=2,
    #                          input_length=max_len,
    #                          mask_zero=True)(inptFirstCap)
    #  dropFirstCap = Dropout(0.1)(embFirstCap)

    ## If the word contains a dash
    #  inptHasDash = Input(shape=(max_len,))
    #  embHasDash = Embedding(input_dim=codes.get_n_HasDash(),
    #                          output_dim=2,
    #                          input_length=max_len,
    #                          mask_zero=True)(inptHasDash)
    #  dropHasDash = Dropout(0.1)(embHasDash)

    ## How many syllables the word has (Aprrox)
    #  inptSyllables = Input(shape=(max_len,))
    #  embSyllables = Embedding(input_dim=codes.get_n_Syllables(),
    #                            output_dim=5,
    #                            input_length=max_len,
    #                            mask_zero=True)(inptSyllables)
    #  dropSyllables = Dropout(0.1)(embSyllables)

    #  ## The shape of the word
    #  inptWordShape = Input(shape=(max_len,))
    #  embWordShape = Embedding(input_dim=codes.get_n_WordShape(),
    #                            output_dim=50,
    #                            input_length=max_len,
    #                            mask_zero=True)(inptWordShape)
    #  dropWordShape = Dropout(0.1)(embWordShape)

    #  ## If the word is a plurl (only checking the 's')
    #  inptIsPlural = Input(shape=(max_len,))
    #  embIsPlural = Embedding(input_dim=codes.get_n_IsPlural(),
    #                           output_dim=2,
    #                           input_length=max_len,
    #                           mask_zero=True)(inptIsPlural)
    #  dropIsPlural = Dropout(0.1)(embIsPlural)

    ## If the word is in the drug-bank file
    #  inptIsInLookup = Input(shape=(max_len,))
    #  embIsInLookup = Embedding(input_dim=codes.get_n_IsInLookup(),
    #                             output_dim=2,
    #                             input_length=max_len,
    #                             mask_zero=True)(inptIsInLookup)
    #  dropIsInLookup = Dropout(0.1)(embIsInLookup)

    # Concatenate all of them together here
    #  dropExtra = concatenate([dropWordLength,
    #                            dropWordShape,
    #                            dropFirstCap,
    #                            dropUpperLower,
    #                            dropHasDash,
    #                            dropSyllables,
    #                            dropIsPlural,
    #                            dropIsInLookup])

    ## Convolutional Solution to the extra parameters
    ## REMNANTS OF THE CONVOLUTIONAL IMPLEMENTATION

    #  inptE = Input(shape=(max_len,7,))
    #  embE = Embedding(input_dim=n_extra, output_dim=50, input_length=max_len,
    #                   mask_zero=True)(inptE)
    #  reshE = Reshape((150,50,7))(embE)
    #  convE = Conv1D(filters=5,kernel_size=3, padding='same',
    #                 activation=LeakyReLU(alpha=0.1))(reshE)
    #  convE = Conv1D(filters=3,kernel_size=3, padding='same',
    #                 activation=LeakyReLU(alpha=0.1))(convE)
    #  convE = Conv1D(filters=1,kernel_size=3, padding='same',
    #                 activation=LeakyReLU(alpha=0.1))(convE)
    #  reshE = Reshape((150, 50))(convE)

    dropW = Dropout(0.1)(embW)
    dropLW = Dropout(0.1)(embLW)
    dropS = Dropout(0.1)(embS)
    dropP = Dropout(0.1)(embP)
    #  dropE = Dropout(0.1)(reshE)
    drops = concatenate([dropW, dropLW, dropS, dropP,
                         # dropExtra
                         ])

    # biLSTM
    bilstm = Bidirectional(LSTM(units=300,
                                return_sequences=True))(drops)
    # output softmax layer

    dense1 = Dense(100, activation=LeakyReLU(alpha=0.1))(bilstm)
    drop1 = Dropout(0.1)(dense1)

    out = TimeDistributed(Dense(n_labels, activation="softmax"))(drop1)

    # build and compile model

    model = Model([inptW, inptLW, inptS, inptP,
                   # inptWordLength,
                   # inptUpperLower,
                   # inptFirstCap,
                   # inptHasDash,
                   # inptSyllables,
                   # inptWordShape,
                   # inptIsPlural,
                   # inptIsInLookup
                   ], out)
    model.compile(optimizer='adam',
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])

    return model


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

# directory with files to process
traindir = TRAIN_DIR
validationdir = VALIDATION_DIR

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

# create indexes from training data
max_len = 150
suf_len = 6
codes = Codemaps(traindata, max_len, suf_len)


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

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

```

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

# save model and indexs
model.save(MODELS_DIR + MODEL_NAME)
codes.save(MODELS_DIR + MODEL_NAME)
```

3.6 Code for Task 5

```python
def build_network(idx) :

    # sizes
    n_words = codes.get_n_words()
    n_lc_words = codes.get_n_lc_words()
    max_len = codes.maxlen
    n_labels = codes.get_n_labels()
    n_pos = codes.get_n_pos()

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

    emb_dim = 200
    embW = Embedding(input_dim=n_words, output_dim=emb_dim,
                        input_length=max_len, embeddings_initializer=Constant(create_embeddings(codes.word_index, emb_dim)), mask_zero=False)(inptW)
    embLW = Embedding(input_dim=n_lc_words, output_dim=emb_dim,
                        input_length=max_len, embeddings_initializer=Constant(create_embeddings(codes.lc_word_index, emb_dim)), mask_zero=False)(inptLW)
    conc = concatenate([embW, embLW])

    #Convolutional Operation
    conv = Conv1D(filters=30, kernel_size=3, strides=1, activation='relu', padding='same')(conc)

    # Dense Layer
    lstm = Bidirectional(LSTM(units=300))(conv)

    dense1 = Dense(100, activation=LeakyReLU(alpha=0.1))(lstm)
    drop = Dropout(0.2)(dense1)

    out = Dense(n_labels, activation="softmax")(drop)

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

    return model
```