MINING UNSTRUCTURED DATA
Document Structure and Language Detection

MUD Course Project

June 13, 2022

Daniel Arias
Mohana Fathollahi

Contents

1 Introduction

Our goal in this assignment is applying neural network models and tuning their parameters to increase accuracy for NERC and DDI labs.

2 NN-based NERC

In all the following experiments, we have used lower-cased words instead of words since that consistently gave us higher accuracy and two layer of LSTM instead of one layer. In the table 1 the performance of baseline model and hyper-parameters on validation set has been shown:

Baseline: Two Bi-LSTMs,embedding_dim=100, dropout=0.1; features: lower_cased word and suffix with length 5. Max-len=150

	tp	fp	fn	#pred	#exp	P	R	F1
brand	100	28	274	128	374	78.1%	26.7%	39.8%
drug	1600	118	317	1718	1917	93.1%	83.5%	88.0%
drug_n	9	91	37	100	46	9.0%	19.6%	12.3%
group	559	91	128	650	687	86.0%	81.4%	83.6%
M.avg	-	-	-	-	-	66.6%	52.8%	56.0%
m.avg	2268	328	756	2596	3024	87.4%	75.0%	80.7%
m.avg(no class)	2406	190	618	2596	3024	92.7%	79.6%	85.6%

Figure 1: Baseline

2.1 initialized word embeddings with pretrained models

We used “glove.6B.zip“ to initialize the ”word” feature embedding. It was trained on a dataset of one billion tokens (words) with a vocabulary of 400 thousand words. There are a few different embedding vector sizes, including 50, 100, 200 and 300 dimensions.

```
glove_Lword_emb = get_preTrained_embedding(glove_embeddings_index, emb_dim,
↪  codes.lc_word_index)
inptLW = Input(shape=(max_len,))
embLW = Embedding(input_dim=n_lc_words,
↪  output_dim=emb_dim,mask_zero=True,input_length=max_len,
↪  weights=[glove_Lword_emb])(inptLW)
```

In the table 2 the result of the baseline network after using pre-trained glove-embedding has been provided.As we can see accuracy after using glove increased to 60%.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	139	26	235	165	374	84.2%	37.2%	51.6%
drug	1620	105	297	1725	1917	93.9%	84.5%	89.0%
drug_n	9	50	37	59	46	15.3%	19.6%	17.1%
group	593	135	94	728	687	81.5%	86.3%	83.8%
M.avg	-	-	-	-	-	68.7%	56.9%	**60.4%**
m.avg	2361	316	663	2677	3024	88.2%	78.1%	82.8%
m.avg(no class)	2478	199	546	2677	3024	92.6%	81.9%	86.9%

Figure 2: Result of applying pretrained model

2.2 Apply different embedding dimension

In this part different values for embedding dimension have been tested, result of them are gathered in table 4 and we could get higher accuracy with 200 embedding dimension.

emb_dim		tp	fp	fn	#pred	#exp	P	R	F1
300	brand	145	16	229	161	374	90.1%	38.8%	54.2%
	drug	1644	115	273	1759	1917	93.5%	85.8%	89.4%
	drug_n	9	91	37	100	46	9.0%	19.6%	12.3%
	group	583	146	104	729	687	80.0%	84.9%	82.3%
	M.avg	-	-	-	-	-	68.1%	57.2%	59.6%
	m.avg	2381	368	643	2749	3024	86.6%	78.7%	82.5%
	m.avg(no class)	2534	215	490	2749	3024	92.2%	83.8%	87.8%
200	brand	240	114	134	354	374	67.8%	64.2%	65.9%
	drug	1692	167	225	1859	1917	91.0%	88.3%	89.6%
	drug_n	6	16	40	22	46	27.3%	13.0%	17.6%
	group	563	96	124	659	687	85.4%	82.0%	83.7%
	M.avg	-	-	-	-	-	**67.9%**	**61.9%**	**64.2%**
	m.avg	2501	393	523	2894	3024	86.4%	82.7%	84.5%
	m.avg(no class)	2638	256	386	2894	3024	91.2%	87.2%	89.2%
50	brand	140	34	234	174	374	80.5%	37.4%	51.1%
	drug	1642	116	275	1758	1917	93.4%	85.7%	89.4%
	drug_n	9	24	37	33	46	27.3%	19.6%	22.8%
	group	574	94	113	668	687	85.9%	83.6%	84.7%
	M.avg	-	-	-	-	-	71.8%	56.6%	62.0%
	m.avg	2365	268	659	2633	3024	89.8%	78.2%	83.6%
	m.avg(no class)	2459	174	565	2633	3024	93.4%	81.3%	86.9%

Figure 3: Result of different embeding dimension

2.3 Max length

The maximum sequence length in our training dataset is 168 and the mean is around 50 words. The maximum length of the sequence in the validation set is around 88. We selected 80 and 200 to check affect of them on accuracy. Based on result of table 4 we can see that when max_len is equal to 200 we can reach to higher accuracy that makes sense because we are covering sentences that has high length too.

Max_len	embed_dim=200; lstm units:200; Glove pre_weights; features: lower_case+ suffix@5								
200		tp	fp	fn	#pred	#exp	P	R	F1
	brand	236	121	138	357	374	66.1%	63.1%	64.6%
	drug	1676	150	241	1826	1917	91.8%	87.4%	89.6%
	drug_n	8	29	38	37	46	21.6%	17.4%	19.3%
	group	572	84	115	656	687	87.2%	83.3%	85.2%
	M.avg	-	-	-	-	-	**66.7%**	**62.8%**	**64.6%**
	m.avg	2492	384	532	2876	3024	86.6%	82.4%	84.5%
	m.avg(no class)	2631	245	393	2876	3024	91.5%	87.0%	89.2%
80		tp	fp	fn	#pred	#exp	P	R	F1
	brand	176	64	198	240	374	73.3%	47.1%	57.3%
	drug	1657	172	260	1829	1917	90.6%	86.4%	88.5%
	drug_n	7	16	39	23	46	30.4%	15.2%	20.3%
	group	555	98	132	653	687	85.0%	80.8%	82.8%
	M.avg	-	-	-	-	-	69.8%	57.4%	62.2%
	m.avg	2395	350	629	2745	3024	87.2%	79.2%	83.0%
	m.avg(no class)	2495	250	529	2745	3024	90.9%	82.5%	86.5%

Figure 4: Result of different max_len

2.4 Suffix length values

The baseline method has used suffix of length 5. In this table 5, we experimented with the suffix equal to 6, 4 and 3 and we could reach higher accuracy when we used suffix equel to 6.

suffix_len									
6		tp	fp	fn	#pred	#exp	P	R	F1
	brand	237	153	137	390	374	60.8%	63.4%	62.0%
	drug	1641	150	276	1791	1917	91.6%	85.6%	88.5%
	drug_n	3	30	43	33	46	9.1%	6.5%	7.6%
	group	550	81	137	631	687	87.2%	80.1%	83.5%
	M.avg	-	-	-	-	-	62.2%	58.9%	**60.4%**
	m.avg	2431	414	593	2845	3024	85.4%	80.4%	82.8%
	m.avg(no class)	2609	236	415	2845	3024	91.7%	86.3%	88.9%
4		tp	fp	fn	#pred	#exp	P	R	F1
	brand	179	50	195	229	374	78.2%	47.9%	59.4%
	drug	1684	182	233	1866	1917	90.2%	87.8%	89.0%
	drug_n	3	20	43	23	46	13.0%	6.5%	8.7%
	group	553	97	134	650	687	85.1%	80.5%	82.7%
	M.avg	-	-	-	-	-	66.6%	55.7%	60.0%
	m.avg	2419	349	605	2768	3024	87.4%	80.0%	83.5%
	m.avg(no class)	2545	223	479	2768	3024	91.9%	84.2%	87.9%
3		tp	fp	fn	#pred	#exp	P	R	F1
	brand	115	5	259	120	374	95.8%	30.7%	46.6%
	drug	1722	237	195	1959	1917	87.9%	89.8%	88.9%
	drug_n	3	5	43	8	46	37.5%	6.5%	11.1%
	group	550	91	137	641	687	85.8%	80.1%	82.8%
	M.avg	-	-	-	-	-	76.8%	51.8%	57.3%
	m.avg	2390	338	634	2728	3024	87.6%	79.0%	83.1%
	m.avg(no class)	2521	207	503	2728	3024	92.4%	83.4%	87.7%

Figure 5: Result of different values for suffix

2.5 LSTM Units

In the figure 6 different values for lstm units have been tested and 200 units of LSTM give us better accuracy compare to other values.

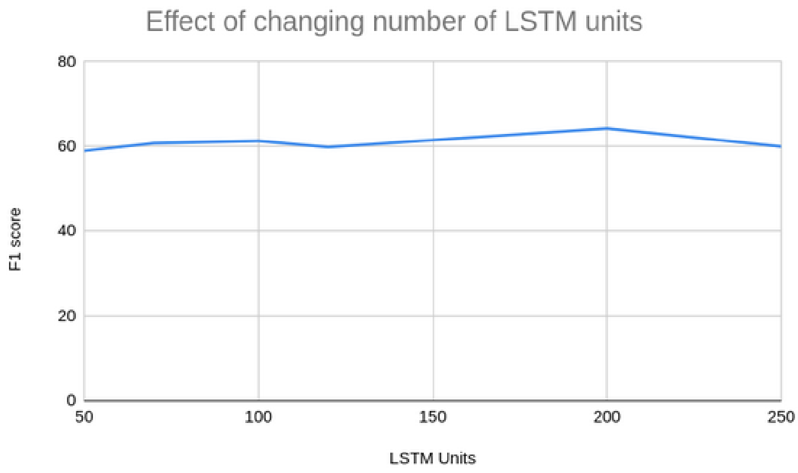


Figure 6: LSTM units

2.6 Optimizer

There are some tudies that used rmsprop optimizer in Recurrent neural networks, therefore we applied this optimizer to compare its performance with adam optimizer. In the table 7 we can see performance of rmsprop optimizer that based on its result rmsprop has lower accuracy than adam.

	tp	fp	fn	#pred	#exp	P	R	F1
brand	168	53	206	221	374	76.0%	44.9%	56.5%
drug	1639	118	278	1757	1917	93.3%	85.5%	89.2%
drug_n	7	80	39	87	46	8.0%	15.2%	10.5%
group	544	102	143	646	687	84.2%	79.2%	81.6%
M.avg	-	-	-	-	-	65.4%	56.2%	59.5%
m.avg	2358	353	666	2711	3024	87.0%	78.0%	82.2%
m.avg(no class)	2499	212	525	2711	3024	92.2%	82.6%	87.1%

Figure 7: Accuracy after applying rmsprop optimizer

2.7 Network structure

- kind of layers
  GRU is a unit in the Recurrent network. It is similar to LSTM, but it does not have to use a memory unit to control the flow of information like the LSTM unit. It can directly make use of all hidden states without any control. GRUs have fewer parameters and thus may train a bit faster or need less data to generalize. In our experiment, we observed that GRU training time of GRU network is shorter than similar network with LSTM unit and its F1 score on validation set was slightly higher. In the table 8 comparison between GRU and LSTM has been provided

Time	GRU	LSTM
Training	real 16m11.840s user 35m52.914s sys 3m20.907s	real 17m39.038s user 40m9.337s sys 3m52.765s
Inference	real 0m29.894s user 0m28.436s sys 0m1.810s	real 0m25.511s user 0m23.926s sys 0m1.880s
F1 score/val	57.2%	52.4%

Figure 8: Comparing GRU and LSTM

- Dense fully connected layer after LSTM layers

```
    bilstm_1 = Bidirectional(LSTM(units=lstm_units, recurrent_dropout=LSTM_DROPOUT,
    ↪   return_sequences=True))(drops)
 bilstm_2 = Bidirectional(LSTM(units=lstm_units, recurrent_dropout=LSTM_DROPOUT,
 ↪   return_sequences=True,))(bilstm_1)                         timeDistributed =
 ↪   TimeDistributed(Dense(fc_units, activation="relu"))(bilstm_2)
out = TimeDistributed(Dense(n_labels, activation="softmax"))(bilstm_2)#(timeDistributed)
```

fully connected units									
100		tp	fp	fn	#pred	#exp	P	R	F1
	brand	201	95	173	296	374	67.9%	53.7%	60.0%
	drug	1669	143	248	1812	1917	92.1%	87.1%	89.5%
	drug_n	2	28	44	30	46	6.7%	4.3%	5.3%
	group	548	90	139	638	687	85.9%	79.8%	82.7%
	M.avg	-	-	-	-	-	63.1%	56.2%	59.4%
	m.avg	2420	356	604	2776	3024	87.2%	80.0%	83.4%
	m.avg(no class)	2562	214	462	2776	3024	92.3%	84.7%	88.3%

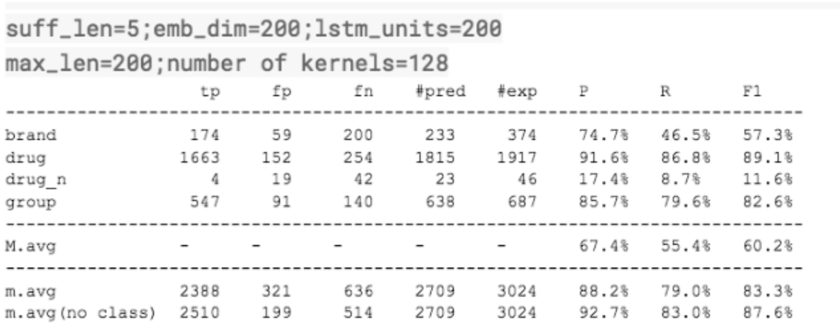
Figure 9: Accuracy after adding fully connected layer

- conv1D after LSTM layers

```
    bilstm_1 = Bidirectional(LSTM(units=lstm_units, recurrent_dropout=LSTM_DROPOUT,
    ↪   return_sequences=True))(drops)

 bilstm_2 = Bidirectional(LSTM(units=lstm_units, recurrent_dropout=LSTM_DROPOUT,
 ↪   return_sequences=True,))(bilstm_1)

  out_conv1D = Conv1D(fc_dim, kernel_size = 3, padding = "same", kernel_initializer =
  ↪   "glorot_uniform")(bilstm_2)
```



suff_len=5;emb_dim=200;lstm_units=200
max_len=200;number of kernels=128

	tp	fp	fn	#pred	#exp	P	R	F1
brand	174	59	200	233	374	74.7%	46.5%	57.3%
drug	1663	152	254	1815	1917	91.6%	86.8%	89.1%
drug_n	4	19	42	23	46	17.4%	8.7%	11.6%
group	547	91	140	638	687	85.7%	79.6%	82.6%
M.avg	-	-	-	-	-	67.4%	55.4%	60.2%
m.avg	2388	321	636	2709	3024	88.2%	79.0%	83.3%
m.avg(no class)	2510	199	514	2709	3024	92.7%	83.0%	87.6%

Figure 10: Accuracy after adding convectional layer

Based on results that we get these two approach did not improve accuray and we used just 2 lstm layers.

2.8 Different combination of prefix and pos

Based on the results in the table 11, we believe adding POS improve F1 score. Prefix with length of 3 characters seems to hear the performance. Therefore, in the last line of the following table, we ran another experiment with a prefix length of 5, but at the end using just Pos has better accuracy.

Setting		tp	fp	fn	#pred	#exp	P	R	F1
Prefix=False @3 Pos=False	brand	83	0	291	83	374	100.0%	22.2%	36.3%
	drug	1686	181	231	1867	1917	90.3%	87.9%	89.1%
	drug_n	7	7	39	14	46	50.0%	15.2%	23.3%
	group	573	83	114	656	687	87.3%	83.4%	85.3%
	M.avg	-	-	-	-	-	81.9%	52.2%	58.5%
	m.avg	2349	271	675	2620	3024	89.7%	77.7%	83.2%
	m.avg(no class)	2442	178	582	2620	3024	93.2%	80.8%	86.5%
Prefix=True @3 Pos=False	brand	83	1	291	84	374	98.8%	22.2%	36.2%
	drug	1679	199	238	1878	1917	89.4%	87.6%	88.5%
	drug_n	4	1	42	5	46	80.0%	8.7%	15.7%
	group	582	145	105	727	687	80.1%	84.7%	82.3%
	M.avg	-	-	-	-	-	87.1%	50.8%	55.7%
	m.avg	2348	346	676	2694	3024	87.2%	77.6%	82.1%
	m.avg(no class)	2461	233	563	2694	3024	91.4%	81.4%	86.1%
Prefix=False pos = True	**brand**	**172**	**27**	**202**	**199**	**374**	**86.4%**	**46.0%**	**60.0%**
	drug	**1713**	**221**	**204**	**1934**	**1917**	**88.6%**	**89.4%**	**89.0%**
	drug_n	**7**	**8**	**39**	**15**	**46**	**46.7%**	**15.2%**	**23.0%**
	group	**581**	**83**	**106**	**664**	**687**	**87.5%**	**84.6%**	**86.0%**
	M.avg	-	-	-	-	-	**77.3%**	**58.8%**	**64.5%**
	m.avg	**2473**	**339**	**551**	**2812**	**3024**	**87.9%**	**81.8%**	**84.7%**
	m.avg(no class)	**2599**	**213**	**425**	**2812**	**3024**	**92.4%**	**85.9%**	**89.1%**
Prefix=True @3 Pos=True	brand	199	54	175	253	374	78.7%	53.2%	63.5%
	drug	1695	184	222	1879	1917	90.2%	88.4%	89.3%
	drug_n	6	8	40	14	46	42.9%	13.0%	20.0%
	group	572	102	115	674	687	84.9%	83.3%	84.1%
	M.avg	-	-	-	-	-	74.1%	59.5%	64.2%
	m.avg	2472	348	552	2820	3024	87.7%	81.7%	84.6%
	m.avg(no class)	2574	246	450	2820	3024	91.3%	85.1%	88.1%
Prefix=True @5 Pos=True	brand	116	11	258	127	374	91.3%	31.0%	46.3%
	drug	1646	132	271	1778	1917	92.6%	85.9%	89.1%
	drug_n	6	6	40	12	46	50.0%	13.0%	20.7%
	group	577	118	110	695	687	83.0%	84.0%	83.5%
	M.avg	-	-	-	-	-	79.2%	53.5%	59.9%
	m.avg	2345	267	679	2612	3024	89.8%	77.5%	83.2%
	m.avg(no class)	2410	202	614	2612	3024	92.3%	79.7%	85.5%

Figure 11: Combination of prefix and pos

2.9 Casing features

This is a combination of one-hot encoded features. For example if the word consists of all numbers the first dim will be active; if there is a '-' in the word the last dimension will be active:
case2Idx = {'numeric': 0, 'allLower':1, 'allUpper':2, 'initialUpper':3, 'other':4,'mainly_numeric':5, 'contains_digit': 6, 'PADDING_TOKEN':7, 'contains_dash':8}.
After applying this feature, based on table 12 we could reach to accuracy 70% that in validation and 64% in test set.

Two Bi-LSTM layers with dropout 0.3 and 0.1 word_emb_dim=100; suffix_emb_dim=100; lstm_units=100; max_len=200 features: lower case word + casing + suffix @5									
Set		tp	fp	fn	#pred	#exp	P	R	F1
Train	brand	1143	11	15	1154	1158	99.0%	98.7%	98.9%
	drug	6952	76	293	7028	7245	98.9%	96.0%	97.4%
	drug_n	423	52	103	475	526	89.1%	80.4%	84.5%
	group	2515	126	171	2641	2686	95.2%	93.6%	94.4%
	M.avg	-	-	-	-	-	95.6%	92.2%	93.8%
	m.avg	11033	265	582	11298	11615	97.7%	95.0%	96.3%
	m.avg(no class)	11076	222	539	11298	11615	98.0%	95.4%	96.7%
Validation	**brand**	**281**	**61**	**93**	**342**	**374**	**82.2%**	**75.1%**	**78.5%**
	drug	**1718**	**133**	**199**	**1851**	**1917**	**92.8%**	**89.6%**	**91.2%**
	drug_n	**8**	**8**	**38**	**16**	**46**	**50.0%**	**17.4%**	**25.8%**
	group	**594**	**91**	**93**	**685**	**687**	**86.7%**	**86.5%**	**86.6%**
	M.avg	**-**	**-**	**-**	**-**	**-**	**77.9%**	**67.2%**	**70.5%**
	m.avg	**2601**	**293**	**423**	**2894**	**3024**	**89.9%**	**86.0%**	**87.9%**
	m.avg(no class)	**2667**	**227**	**357**	**2894**	**3024**	**92.2%**	**88.2%**	**90.1%**
Test	brand	228	109	46	337	274	67.7%	83.2%	74.6%
	drug	1833	130	294	1963	2127	93.4%	86.2%	89.6%
	drug_n	3	15	69	18	72	16.7%	4.2%	6.7%
	group	596	142	97	738	693	80.8%	86.0%	83.3%
	M.avg	-	-	-	-	-	64.6%	64.9%	63.6%
	m.avg	2660	396	506	3056	3166	87.0%	84.0%	85.5%
	m.avg(no class)	2803	253	363	3056	3166	91.7%	88.5%	90.1%

Figure 12: Best Result

2.10 Conclusion

We introduced a new feature called casing which gave us the greatest boost among all other model, hyper parameters and features variation. Additionally, using pre-trained glove weights and embedding dimensions gave the considerable boost.
In all of our experiments, the low-population class ”durg-n” was the culprit on the gap between validation and training set performance. Even though we tried increasing the dropout or reducing the model complexity, they didn’t help much.
We experimented with several features: lowercase word, lemmatization, Pos, different length of suffix and prefix. Although we only tried a handful of suffixes and prefix length. Some of these features such as prefix hurt the performance and some marginally improved the results.

2.11 Future direction

All the hyperparameter and network architectures that we experimented with was using suffix and lowercase word. We can add the new ”casing” feature and re-run all the previous experiments.

3 NN-based DDI

3.1 Parameters

In the different Neural Networks architectures created we have these parameters to take into account. We run all the experiments with the same configuration of the parameters in order to have a better approach to compare them.

- maxlen = 100
- batch_size = 32
- filters = 32
- kernel_size = 2
- epochs = 10
- n_words = 1000
- vocab_size = n_words
- hidden_dims = 250
- embeddings_dims = 300
- activation = 'relu'
  The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time. This means that the neurons will only be deactivated if the output of the linear transformation is less than 0.
- padding = 'same'
  When padding="same" and strides=1, the output has the same size as the input.
- kernel_regularizer_l2 = 0.001
  Regularizers allow us to apply penalties on layer parameters or layer activity during optimization. These penalties are summed into the loss function that the network optimizes.

3.2 Architectures

In order to classify the different labels of drug-drug interaction we created different Neural Networks architectures. The goal was to identify which architecture performs better, taking to account the dataset has more 'no interaction' labels than the others 'advise','effect','int', and 'mechanism'.

As part of every Neural Network Architecture, the embedding is the first layer for the inputs. We train with a combination of inputs for the initial architecture. Thus, in order to see which of them gives better results.

As the *codemaps* could retrieve the pad sequence for different features of the whole sentence like 'form', 'lower case', 'Part of the speech', and the 'Lemma' we did a combination of experiments for each of the Neural Networks were in the inputs it receives the 'lower case' or the whole pad sequence of all the features without the 'form' feature that is the raw form of the words in the sentence. We did not take this 'form' due to another part of these combinations of inputs we included the vector representations for words provided by GloVe in several dimensions and those representations were taken from the lower case word in the English vocabulary. In order to us ethe GloVe vectors depending on the number of dimensions we were testing we use the *lc_word_index* provided by the *Codemaps* class.

```python
def load_glove_embedding(EMBEDDING_DIM, word2index: dict):
    glove_dir_path = f'{glove_dir}/glove.6B.{EMBEDDING_DIM}d.txt'
    n_words = len(word2index)
    embedding_matrix = np.zeros((n_words, EMBEDDING_DIM))
    with open(glove_dir_path, "r") as f:
        for _line in f:
            line = _line.split()
            word = line[0]
            if word in word2index:
                idx = word2index[word]
                embedding_vector = np.array(line[1:], dtype=np.float32)
                embedding_matrix[idx] = embedding_vector
    return Embedding(n_words, EMBEDDING_DIM, weights=[embedding_matrix], trainable=False)
```

Meaning this, we run each the following set of inputs combinations:

- Lower case pad sequence.
- GloVe matrix created from the words found in the corpus
- Lower case, Part of the speech and Lemma pad sequence.
- GloVe matrix created from the words found in the corpus alongside Part of the speech and Lemma pad sequence.

```diff
def encode_words(self, data):
...
+       return [Xlw]
-       return [Xlw,Xl,Xp]
```

With this, we would be able to compare the F1 score given the inputs and recognize which one is the combinations are better.

First we compare the effect of using the 'Lower case' pad sequences against the use of the GloVe vector embedding.

develconvolutional_initial_lw.stats

results > devel > stats > develconvolutional_initial_lw.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	78	57	63	135	141	57.8%	55.3%	56.5%
effect	123	67	189	190	312	64.7%	39.4%	49.0%
int	9	1	19	10	28	90.0%	32.1%	47.4%
mechanism	87	107	174	194	261	44.8%	33.3%	38.2%
M.avg	-	-	-	-	-	64.3%	40.1%	47.8%
m.avg	297	232	445	529	742	56.1%	40.0%	46.7%
m.avg(no class)	322	207	420	529	742	60.9%	43.4%	50.7%

develconvolutional_initial_GloVe.stats

results > devel > stats > develconvolutional_initial_GloVe.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	82	142	59	224	141	36.6%	58.2%	44.9%
effect	50	53	262	103	312	48.5%	16.0%	24.1%
int	0	1	28	1	28	0.0%	0.0%	0.0%
mechanism	37	89	224	126	261	29.4%	14.2%	19.1%
M.avg	-	-	-	-	-	28.6%	22.1%	22.0%
m.avg	169	285	573	454	742	37.2%	22.8%	28.3%
m.avg(no class)	186	268	556	454	742	41.0%	25.1%	31.1%

testconvolutional_initial_lw.stats

results > test > stats > testconvolutional_initial_lw.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	75	50	134	125	209	60.0%	35.9%	44.9%
effect	123	66	163	189	286	65.1%	43.0%	51.8%
int	0	3	25	3	25	0.0%	0.0%	0.0%
mechanism	129	144	211	273	340	47.3%	37.9%	42.1%
M.avg	-	-	-	-	-	43.1%	29.2%	34.7%
m.avg	327	263	533	590	860	55.4%	38.0%	45.1%
m.avg(no class)	375	215	485	590	860	63.6%	43.6%	51.7%

testconvolutional_initial_GloVe.stats

results > test > stats > testconvolutional_initial_GloVe.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	67	243	142	310	209	21.6%	32.1%	25.8%
effect	53	38	233	91	286	58.2%	18.5%	28.1%
int	0	0	25	0	25	0.0%	0.0%	0.0%
mechanism	86	233	254	319	340	27.0%	25.3%	26.1%
M.avg	-	-	-	-	-	26.7%	19.0%	20.0%
m.avg	206	514	654	720	860	28.6%	24.0%	26.1%
m.avg(no class)	223	497	637	720	860	31.0%	25.9%	28.2%

Figure 13: train test Lower Case vs GloVe results

Image fig. 13 depicted the results from the different inputs alongside the train and test datasets, giving a F1 Score for 47.8% and 34.7% for the *Lower Case* and 22% and 20% for GloVe Embedding layer. These due to the fact that Gloval Vectors in this case that the context plays an important role having a global representation does not help at the time of finding interactions.

develconvolutional_initial_lw_pos_lemma.stats

results > devel > stats > develconvolutional_initial_lw_pos_lemma.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	92	64	49	156	141	59.0%	65.2%	62.0%
effect	139	70	173	209	312	66.5%	44.6%	53.4%
int	16	1	12	17	28	94.1%	57.1%	71.1%
mechanism	95	64	166	159	261	59.7%	36.4%	45.2%
M.avg	-	-	-	-	-	69.8%	50.8%	57.9%
m.avg	342	199	400	541	742	63.2%	46.1%	53.3%
m.avg(no class)	396	145	346	541	742	73.2%	53.4%	61.7%

develconvolutional_initial_GloVe_pos_lemma.stats

results > devel > stats > develconvolutional_initial_GloVe_pos_lemma.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	88	53	53	141	141	62.4%	62.4%	62.4%
effect	141	52	171	193	312	73.1%	45.2%	55.8%
int	16	1	12	17	28	94.1%	57.1%	71.1%
mechanism	128	88	133	216	261	59.3%	49.0%	53.7%
M.avg	-	-	-	-	-	72.2%	53.4%	60.8%
m.avg	373	194	369	567	742	65.8%	50.3%	57.0%
m.avg(no class)	401	166	341	567	742	70.7%	54.0%	61.3%

testconvolutional_initial_lw_pos_lemma.stats

results > test > stats > testconvolutional_initial_lw_pos_lemma.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	91	38	118	129	209	70.5%	43.5%	53.8%
effect	137	59	149	196	286	69.9%	47.9%	56.8%
int	16	1	9	17	25	94.1%	64.0%	76.2%
mechanism	140	104	200	244	340	57.4%	41.2%	47.9%
M.avg	-	-	-	-	-	73.0%	49.2%	58.7%
m.avg	384	202	476	586	860	65.5%	44.7%	53.1%
m.avg(no class)	424	162	436	586	860	72.4%	49.3%	58.6%

testconvolutional_initial_GloVe_pos_lemma.stats

results > test > stats > testconvolutional_initial_GloVe_pos_lemma.stats

	tp	fp	fn	#pred	#exp	P	R	F1
advise	92	35	117	127	209	72.4%	44.0%	54.8%
effect	139	39	147	178	286	78.1%	48.6%	59.9%
int	4	2	21	6	25	66.7%	16.0%	25.8%
mechanism	154	105	186	259	340	59.5%	45.3%	51.4%
M.avg	-	-	-	-	-	69.2%	38.5%	48.0%
m.avg	389	181	471	570	860	68.2%	45.2%	54.4%
m.avg(no class)	419	151	441	570	860	73.5%	48.7%	58.6%

Figure 14: train test Lower Case vs GloVe with PoS and Lemma

In fig. 14 we can observe that the best results are the ones whose input is with *Lower Case* pad sequences and that the use of other pad sequences inputs such as the Part of the speech and the Lemma of the words considerably improve the F1 score, increasing almost a 10% in training and test dataset for the *Lower Case* and almost a 40% for training and 30% in test dataset using the GloVe embedding layer. Given this, all the other different architectures were been done using the three inputs without the use of GloVe embedding layer.

As the Neural Networks takes this combinations of inputs there are two possible ways for each architecture to handle each of the embedding layers.

One of them is to take all the inputs and create the embedding layer for each of them each layer (ConvID, LSTM, Bidirectional LSTM, or Hybrid) will take care of each of the embeddings, and the output from these layers after going to a *MaxPooling* layer is concatenated to go as input in the final *Dense* layer. On the other hand, we create a model from each of the inputs and concatenated the outputs to be the input of the last *Dense* layer.

```
+   concatenate = Concatenate(axis=1, name='Concatenate_MaxPool')(
+       [model1.output, model2.output, model3.output])
-   concatenate = Concatenate(axis=1,name='Concatenate_MaxPool')(
-       [maxpool_0,maxpool_1,maxpool_2,maxpool_01,maxpool_11,maxpool_21,maxpool_02,maxpool_12,maxpool_22])

+   model = Model([model1.input, model2.input, model3.input], outputs=final)
-   model = Model(inputs=[inptW, inptL, inptP], outputs=final)
```

3.2.1 Accuracy

For all of our experiments we save the history, with these we can see how the accuracy increase and decrease for each epoch in the train and the validation dataset.

3.2.2 Loss

As all of our models are performing a multi-class classification task, we use the *Categorical Crossentropy.* which is a loss function, that is used for the prediction error of Neural Network in order to calculate the gradients. The best model is the one with less loss.

3.2.3 Convolutional Networks

The first architecture consist on having just one *Conv1D*.

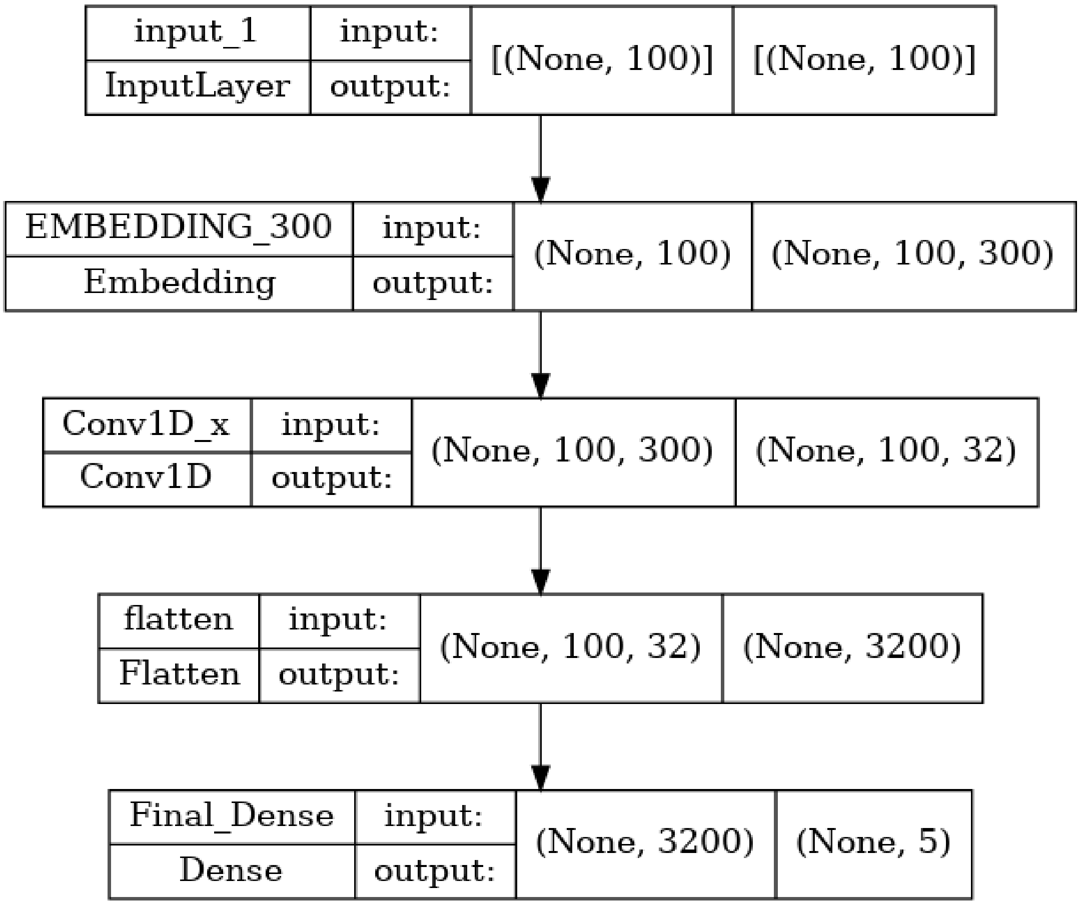


Figure 15: CNN model

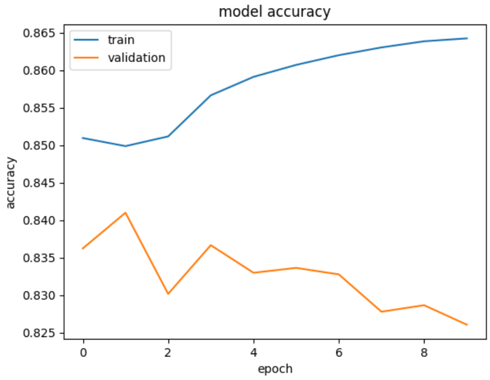


Figure 16: Accuracy

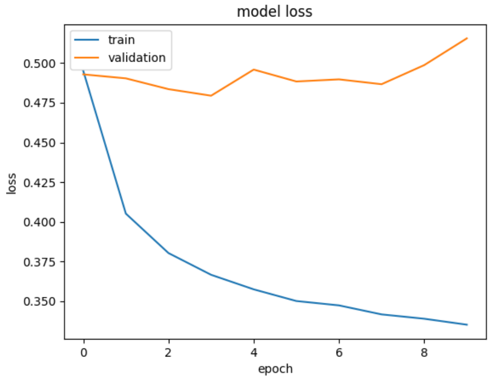


Figure 17: Loss

	tp	fp	fn	#pred	#exp	P	R	F1
advise	96	73	45	169	141	56.8%	68.1%	61.9%
effect	127	46	185	173	312	73.4%	40.7%	52.4%
int	16	0	12	16	28	100.0%	57.1%	72.7%
mechanism	87	27	174	114	261	76.3%	33.3%	46.4%
M.avg	-	-	-	-	-	76.6%	49.8%	58.4%
m.avg	326	146	416	472	742	69.1%	43.9%	53.7%
m.avg(no class)	364	108	378	472	742	77.1%	49.1%	60.0%

Figure 18: CNN Train

	tp	fp	fn	#pred	#exp	P	R	F1
advise	119	107	90	226	209	52.7%	56.9%	54.7%
effect	124	42	162	166	286	74.7%	43.4%	54.9%
int	1	2	24	3	25	33.3%	4.0%	7.1%
mechanism	148	64	192	212	340	69.8%	43.5%	53.6%
M.avg	-	-	-	-	-	57.6%	37.0%	42.6%
m.avg	392	215	468	607	860	64.6%	45.6%	53.4%
m.avg(no class)	432	175	428	607	860	71.2%	50.2%	58.9%

Figure 19: CNN Test

3.2.4 Deep Convolutional Networks

In this one we add more *Conv1D* layers where *MaxPooling* layer takes the max of the outputs and then we concatenate them as the final input of the *Dense* layer.

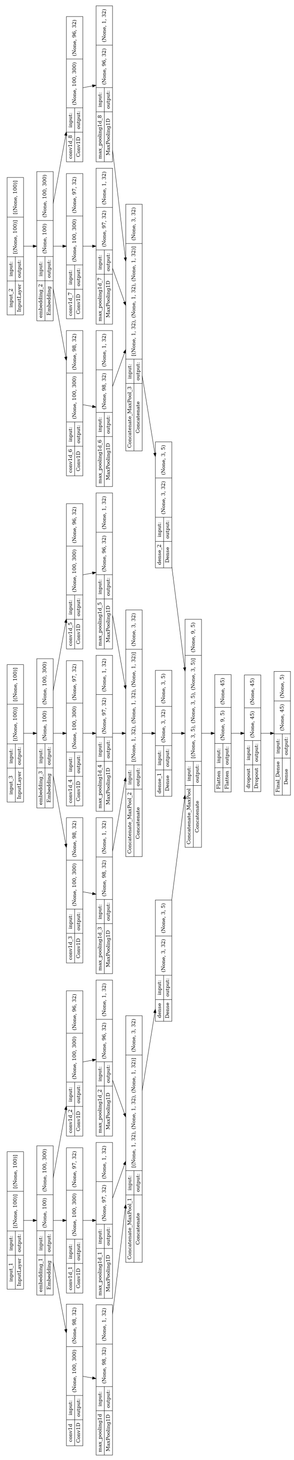


Figure 20: Deep CNN Model

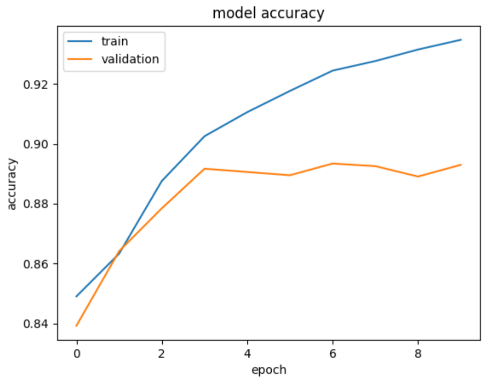


Figure 21: Accuracy

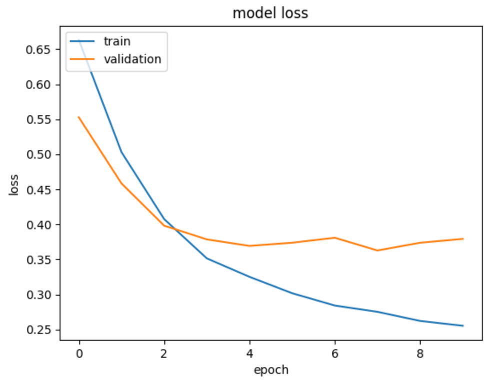


Figure 22: Loss

	tp	fp	fn	#pred	#exp	P	R	F1
advise	96	73	45	169	141	56.8%	68.1%	61.9%
effect	127	46	185	173	312	73.4%	40.7%	52.4%
int	16	0	12	16	28	100.0%	57.1%	72.7%
mechanism	87	27	174	114	261	76.3%	33.3%	46.4%
M.avg	-	-	-	-	-	76.6%	49.8%	58.4%
m.avg	326	146	416	472	742	69.1%	43.9%	53.7%
m.avg(no class)	364	108	378	472	742	77.1%	49.1%	60.0%

Figure 23: Deep CNN Train

	tp	fp	fn	#pred	#exp	P	R	F1
advise	119	107	90	226	209	52.7%	56.9%	54.7%
effect	124	42	162	166	286	74.7%	43.4%	54.9%
int	1	2	24	3	25	33.3%	4.0%	7.1%
mechanism	148	64	192	212	340	69.8%	43.5%	53.6%
M.avg	-	-	-	-	-	57.6%	37.0%	42.6%
m.avg	392	215	468	607	860	64.6%	45.6%	53.4%
m.avg(no class)	432	175	428	607	860	71.2%	50.2%	58.9%

Figure 24: Deep CNN Test

3.2.5 Classification LSTMs

As we saw in the previous section in the NER classification the use of the *LSTM* layer we make use of it in this architecture instead of the *Conv1D*.

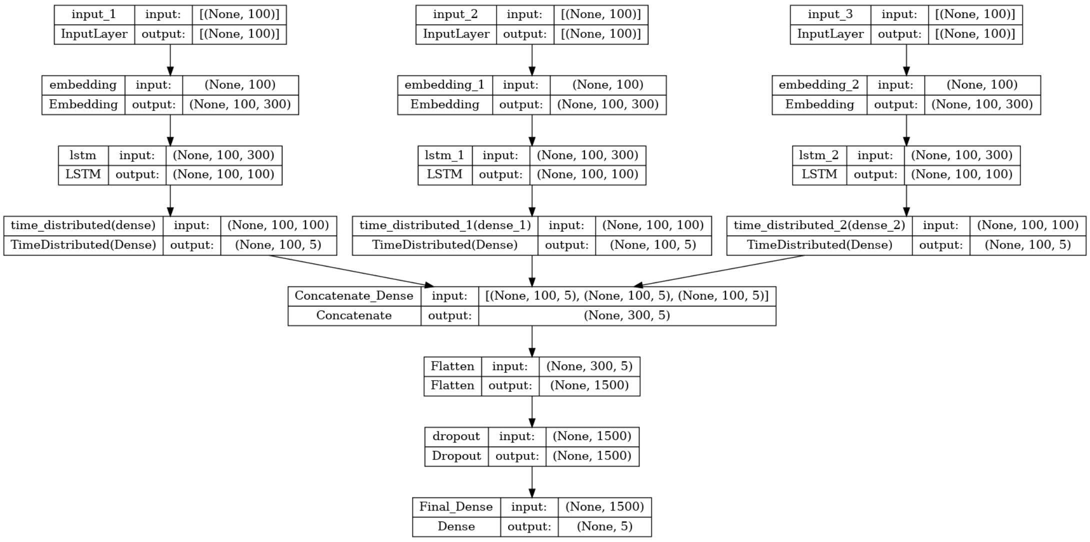


Figure 25: LSTM Model

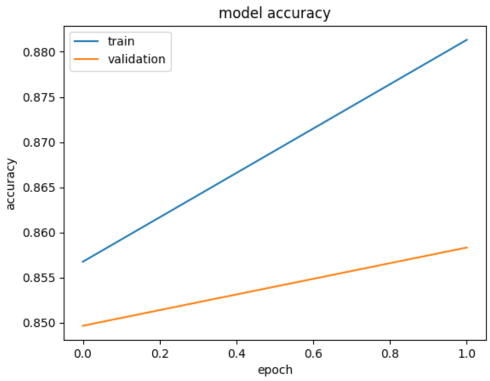


Figure 26: Accuracy

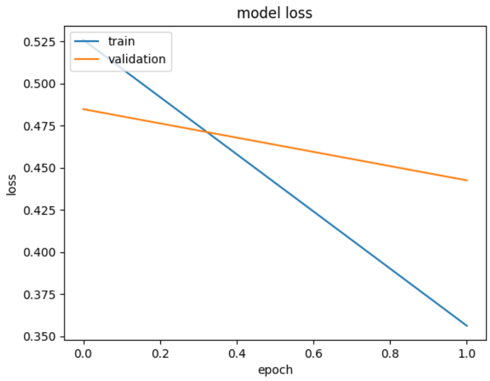


Figure 27: Loss

	tp	fp	fn	#pred	#exp	P	R	F1
advise	97	119	44	216	141	44.9%	68.8%	54.3%
effect	167	155	145	322	312	51.9%	53.5%	52.7%
int	12	0	16	12	28	100.0%	42.9%	60.0%
mechanism	100	92	161	192	261	52.1%	38.3%	44.2%
M.avg	-	-	-	-	-	62.2%	50.9%	52.8%
m.avg	376	366	366	742	742	50.7%	50.7%	50.7%
m.avg(no class)	454	288	288	742	742	61.2%	61.2%	61.2%

Figure 28: LSTM Train

	tp	fp	fn	#pred	#exp	P	R	F1
advise	132	97	77	229	209	57.6%	63.2%	60.3%
effect	162	174	124	336	286	48.2%	56.6%	52.1%
int	0	0	25	0	25	0.0%	0.0%	0.0%
mechanism	118	109	222	227	340	52.0%	34.7%	41.6%
M.avg	-	-	-	-	-	39.5%	38.6%	38.5%
m.avg	412	380	448	792	860	52.0%	47.9%	49.9%
m.avg(no class)	484	308	376	792	860	61.1%	56.3%	58.6%

Figure 29: LSTM Test

3.2.6 Bidirectional LSTM

As the interaction between drugs in the sentences did not always appear in the first max length of the sentence, in order to add these potential information we make a *Bidirectional LSTM* layer that could capture these interactions. However it is strongly dependent on the filter, batch, kernel and dimensions size.

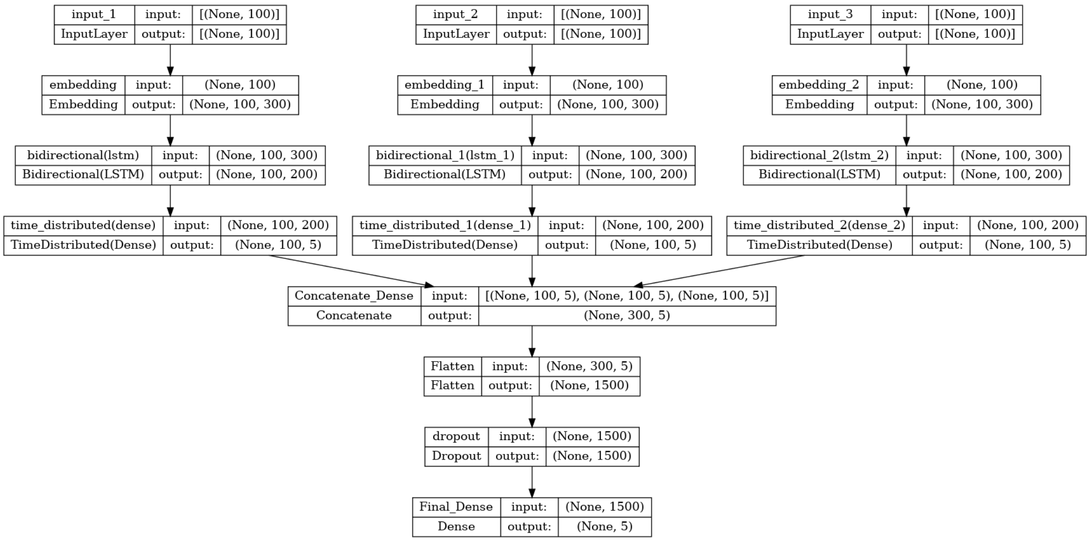


Figure 30: Bidirectional LSTM Model

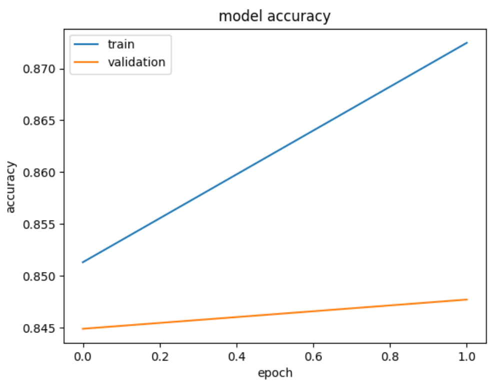


Figure 31: Accuracy

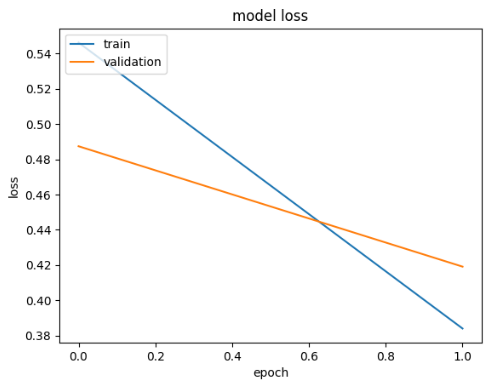


Figure 32: Loss

	tp	fp	fn	#pred	#exp	P	R	F1
advise	92	116	49	208	141	44.2%	65.2%	52.7%
effect	180	205	132	385	312	46.8%	57.7%	51.6%
int	13	0	15	13	28	100.0%	46.4%	63.4%
mechanism	89	64	172	153	261	58.2%	34.1%	43.0%
M.avg	-	-	-	-	-	62.3%	50.9%	52.7%
m.avg	374	385	368	759	742	49.3%	50.4%	49.8%
m.avg(no class)	424	335	318	759	742	55.9%	57.1%	56.5%

Figure 33: Bidirectional LSTM Train

	tp	fp	fn	#pred	#exp	P	R	F1
advise	126	159	83	285	209	44.2%	60.3%	51.0%
effect	160	179	126	339	286	47.2%	55.9%	51.2%
int	0	0	25	0	25	0.0%	0.0%	0.0%
mechanism	136	109	204	245	340	55.5%	40.0%	46.5%
M.avg	-	-	-	-	-	36.7%	39.1%	37.2%
m.avg	422	447	438	869	860	48.6%	49.1%	48.8%
m.avg(no class)	461	408	399	869	860	53.0%	53.6%	53.3%

Figure 34: Bidirectional LSTM Test

3.2.7 CNN/LSTM Hybrid

Finally, after trying with different type of layers one of each by separated, we make the final architecture combining all of them. Therefore, we create a *Conv1D* and a *Bidirectional LSTM* layer whose outputs go in the *Concatenate* to get the *Dense* as final output.

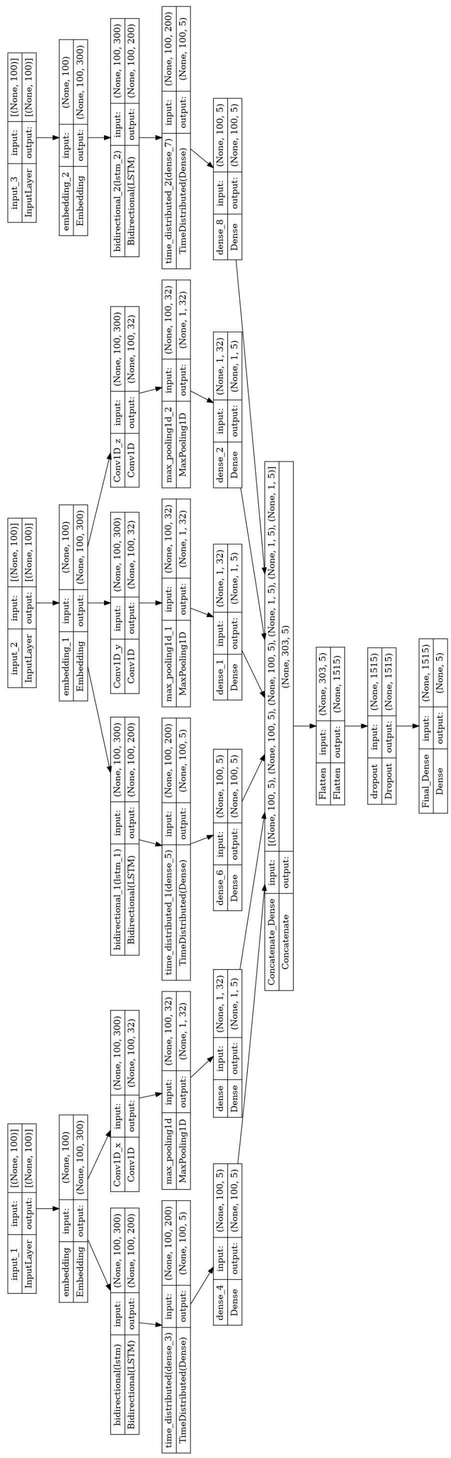


Figure 35: Hybrid Model

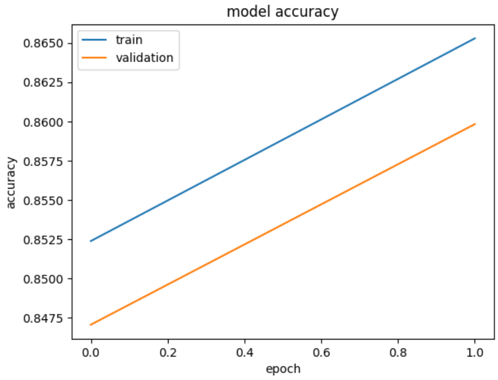


Figure 36: Accuracy

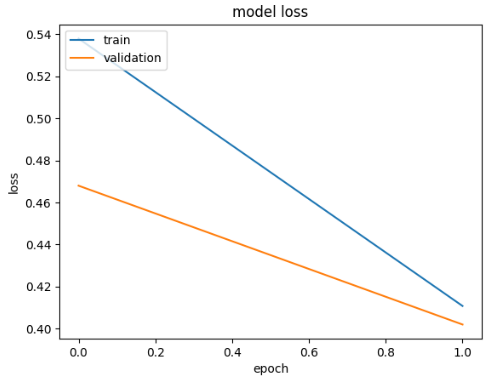


Figure 37: Loss

	tp	fp	fn	#pred	#exp	P	R	F1
advise	71	29	70	100	141	71.0%	50.4%	58.9%
effect	133	171	179	304	312	43.8%	42.6%	43.2%
int	0	0	28	0	28	0.0%	0.0%	0.0%
mechanism	0	0	261	0	261	0.0%	0.0%	0.0%
M.avg	-	-	-	-	-	28.7%	23.2%	25.5%
m.avg	204	200	538	404	742	50.5%	27.5%	35.6%
m.avg(no class)	295	109	447	404	742	73.0%	39.8%	51.5%

Figure 38: Hybrid Train

	tp	fp	fn	#pred	#exp	P	R	F1
advise	70	28	139	98	209	71.4%	33.5%	45.6%
effect	143	200	143	343	286	41.7%	50.0%	45.5%
int	0	0	25	0	25	0.0%	0.0%	0.0%
mechanism	0	0	340	0	340	0.0%	0.0%	0.0%
M.avg	-	-	-	-	-	28.3%	20.9%	22.8%
m.avg	213	228	647	441	860	48.3%	24.8%	32.7%
m.avg(no class)	330	111	530	441	860	74.8%	38.4%	50.7%

Figure 39: Hybrid Test

3.3 Parameter Tuning

After selecting the best architecture trained with the same parameters we can proceed to search for the best parameters in order to improve the F1 score. For this we make use of the *GridSearchCV* API provided by the *sklearn*.

First we find the best the best Neuron Activation Function where we included: ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'elu']. Then we adjust the grid the best Dropout Regularization we consider the following values [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]. Finally we find the best Batch Size and Number of Epochs we take into account these values:

- batch_size = [8, 16, 32, 64]
- epochs = [2, 4, 6, 8, 10]

It could also be done in only one execution of the *GridSearchCV* but the computational cost would have been too much to get the results. Therefore, we implemented in these way trying to find the best parameters from the most inside in the Neural Network parameters.

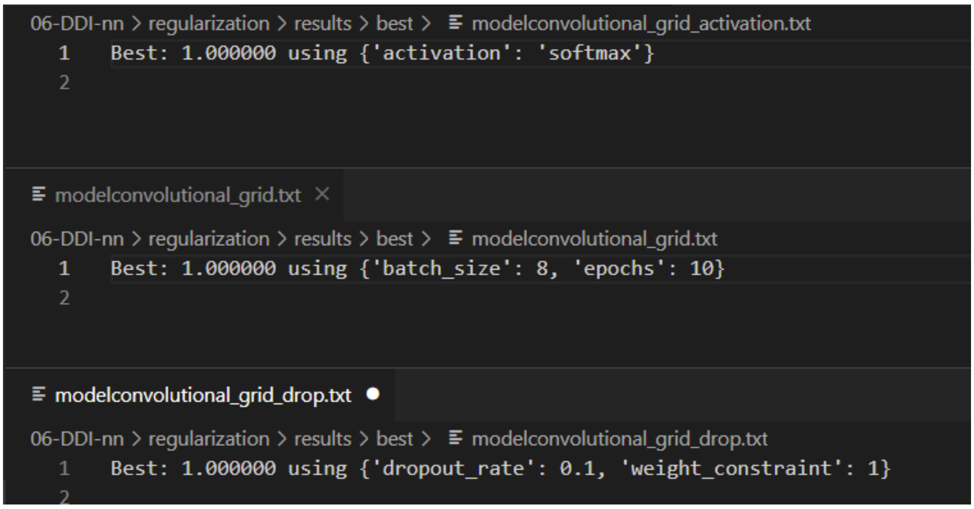


Figure 40: Best Parameters

The best parameter from the Neuron Activation Function is *'Softmax'*, this could be due to the multiple labels that we are trying to classify. In case of the dropout the best parameter was 0.1. Finally for the batch size and the number of epochs the grid search returned a 10 for the number of epochs and a batch size of 8.

We proceed to run our best model with the best hyper-parameters.

3.4 Final Architecture

The best architecture was the deep CNN concatenating the outputs from the *MaxPooling* layer. Then we update the parameters and run our model.

	tp	fp	fn	#pred	#exp	P	R	F1
advise	92	64	49	156	141	59.0%	65.2%	62.0%
effect	139	70	173	209	312	66.5%	44.6%	53.4%
int	16	1	12	17	28	94.1%	57.1%	71.1%
mechanism	95	64	166	159	261	59.7%	36.4%	45.2%
M.avg	-	-	-	-	-	69.8%	50.8%	57.9%
m.avg	342	199	400	541	742	63.2%	46.1%	53.3%
m.avg(no class)	396	145	346	541	742	73.2%	53.4%	61.7%

Figure 41: Train

	tp	fp	fn	#pred	#exp	P	R	F1
advise	91	38	118	129	209	70.5%	43.5%	53.8%
effect	137	59	149	196	286	69.9%	47.9%	56.8%
int	16	1	9	17	25	94.1%	64.0%	76.2%
mechanism	140	104	200	244	340	57.4%	41.2%	47.9%
M.avg	-	-	-	-	-	73.0%	49.2%	58.7%
m.avg	384	202	476	586	860	65.5%	44.7%	53.1%
m.avg(no class)	424	162	436	586	860	72.4%	49.3%	58.6%

Figure 42: Test

3.5 Conclusions

In summarising, from all the different architectures we tried, training Neural Networks in special, Convolutional Neural Networks, we encountered that the most important part is the design of the different layers to use, CNN offered very good and very fast results, however, it not that straightforward to debug or realize in which step the coefficients are not meeting the expected results and every change in the parameters makes a significant difference in the results. Augmenting the number of epochs could improve the train statistics but generally, it is over-fitting and not learning to generalize, this was illustrated when we test our models with the test dataset.

The implementation of Functional API Models for Neural Networks allows working with the layers in multiple ways creating branches and then unifying the results, something that Sequential models do not allow.