Introduction to Task 5

The objective of the task is to detect drug-drug interactions using Neural Networks. For this, throughout the task we tried improving the general architecture of the model, experimenting with convolutional layers, Long-Short Term Memory (LSTM) cells, etc.

We observed that the same model configurations can produce significantly different F1 scores in different training and testing runs. This is why, in order to obtain more reliable results, we decided to run the different models at least three times and average the F1 scores.

Base Model

As can be seen in Figure 1, the base model consists of the following layers: Input, Embedding, Conv1D, Flatten and Dense.

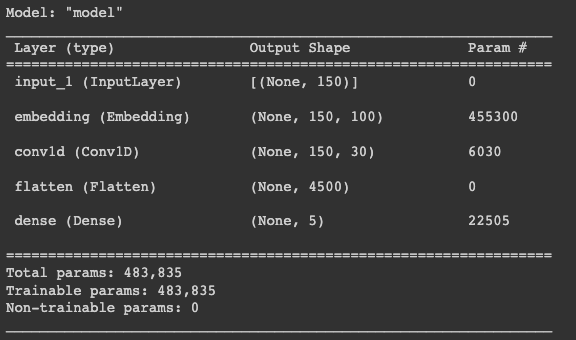


Figure 1. Base Model Layers

The embedding layer is responsible for converting words into numerical vectors. If the embedding dimension is set to 100, it means that the output of this layer is a tensor with dimensions [(None, 150, 100)]. Each word in the tensor is now represented by a vector with 100 components. This embedded tensor is then fed into a 1-dimensional convolutional layer, The output dimension of this convolutional layer is (None, 150, 30), where the last dimension represents the number of filters per word.

The Flatten layer serves as a reshaping layer that reduces the current dimensions to one by sequentially arranging the vectors. For the base case, this results in an output dimension of (None, 4500).

The Dense layer represents the final step in the model and is a standard feed-forward neural network. The final output layer consists of four neurons, and the softmax activation function

is used to provide probability distributions across the output classes, enabling the model to make predictions.

When running the base model we got a Macro F1 score of 52.3%. So, this will be our base to improve.

**Embeddings**

As a first step we tried changing the embedding dimensions and the data that is fed into the embeddings.

From the different results it can be seen that lemmatization and PoS tagging do not contribute significant additional information to the world embeddings.

| Embeddings | Total Embedding dimension | F1 score |
| --- | --- | --- |
| Pos | 100 | 31.93% |
| Words | 200 | 53.68% |
| Words | 300 | 51.1% |
| Words + Lemmas | 400 | 53.2% |
| Words + Pos | 400 | 41.2% |
| Words + Lowercase | 400 | 53.16% |
| Words + LowerCase + Lemmas | 600 | 50.2% |
| Words + LowerCase + Pos | 600 | 53.03% |

**Filter and kernel size**

Looking at the base code we can see that filters=30 and kernel\_size=2. This indicates that the convolutional layer will utilize 30 different filters, each responsible for detecting specific features in the input data and that the filters will have a size of 2, meaning that they will analyze two adjacent elements at a time along the sequence.

Increasing the number of filters and using larger kernel sizes can increase the model´s expressive power, enabling it to capture more intricate features.

We tried different configurations for these parameters.

| Conv1D Configurations | F1 score |
| --- | --- |
| filters = 50 | 52.9% |
| filters = 100 | 51.3% |
| filters = 10 | 50.97% |
| filter = 30, kernel\_size=3 | 48.4% |
| filter = 50, kernel\_size=3 | 51.67% |
| filter = 30, kernel\_size=4 | 55% |
| filter = 50, kernel\_size=5 | 55.07% |

**Dense and Dropout layers**

Both Dense and Dropout layers are commonly used together in deep learning models. The former enables the network to learn complex representations and capture intricate relationships in the data while the latter helps prevent overfitting and improves generalization.

There are several activation functions to be used, like Sigmoid, Tanh, Relu, LeakyRelu, Softamx, etc.

We choose to employ the activation function LeakyRelu. It helps address the dying ReLu problem where certain neurons can become inactive during training.

We tested different Dropout and Dense layers, like adding a Dense(100), Dropout(0.1) after the Dense, Dropout(0.2) after Embedding, Dropout(0.2) after Embedding and after Dense among others. None of the results improved the F1 score.

**Max Pooling**

Max Pooling allows the network to progressively reduce the spatial dimensions while retaining the most important features for subsequent processing and learning.

There are two common types of pooling operations used in CNNs: maximum pooling and average pooling.

Maximum pooling selects the maximum value within each pooling window. It retains the most prominent or activated features within each region.

Average pooling computes the average value within each pooling window. It calculates the mean activation across the region. It provides a more smoothed representation of the input features.

| Pooling Configuration | F1 Score |
| --- | --- |
| MaxPooling: pool\_size= 2, kernek\_size=4 | 55.9% |
| MaxPooling: pool\_size= 3, kernek\_size=3 | 56.2% |
| AveragePooling: pool\_size= 2, kernek\_size=4 | 55.3% |

**Pre-Trained Embeddings**

There are pre-trained embedding layers available that can be incorporated into a model to enhance its performance. As a consequence of using them the model does not need to extensively train the embedding layer, it just needs to slightly fine tune it or use it as it is.

We decided to use GloVe. For this we adapted a suggested GloVe model to get the initial embedding matrix from this repository https://github.com/suriak/sentence-classification-cnn/blob/master/sentence%20classifier.py

By using the pre-trained embedding and adding an extra dense layer and a Dropout we were able to achieve a F1 score of 57.9%.

**Different architectures**

All the architectures tried are with pre-trained embeddings.

The first architecture we tried was LSTM which is designed to overcome the limitations of traditional RNNs in capturing long-term dependencies in sequential data. They are particularly effective in tasks that involve sequential data, such as natural language processing.

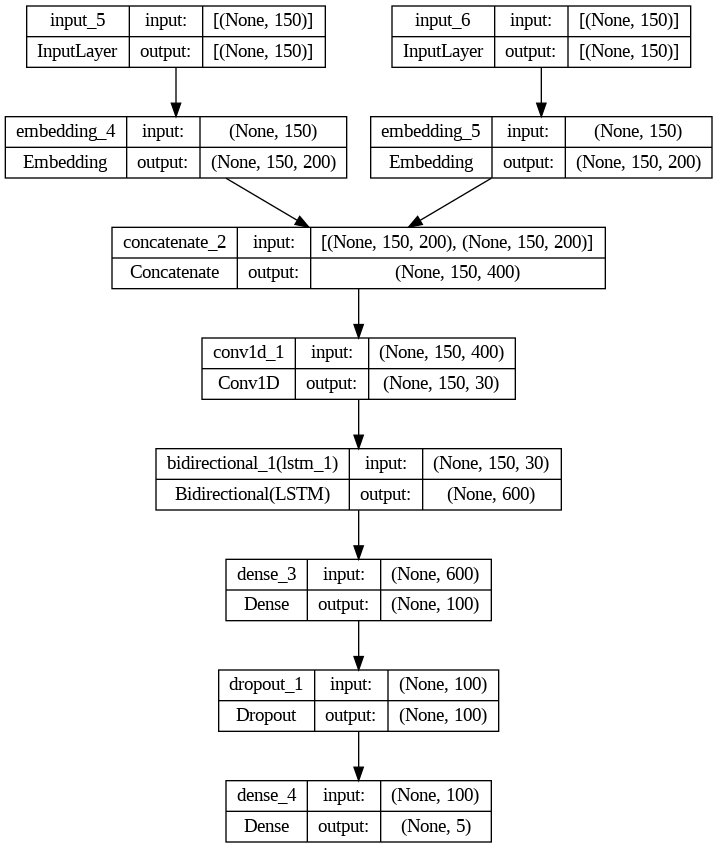
The fundamental concept behind LSTM is to incorporate memory cells that have the ability to retain information across extended sequences, enabling the network to comprehend and retain significant patterns or dependencies. These memory cells are designed with specialized gates that control the flow of information, including the input gate, forget gate, and output gate. The input gate determines the amount of new information to be stored in the memory cell, while the forget gate determines which information should be discarded. Finally, the output gate regulates the amount of information to be propagated to subsequent time steps or the final output. By integrating these gating mechanisms, LSTM networks can effectively capture and retain long-term dependencies, addressing challenges such as vanishing or exploding gradients and mitigating information loss over time commonly encountered in traditional recurrent neural networks.

Afterwards, we tried a Bidirectional LSTM (BiLSTM) which is an LSTM variant that processes the input sequence in both forward and backward directions. We tried different parameters for the units but the best result was obtained using 300, which means that the LSTM layer will have 300 hidden units.

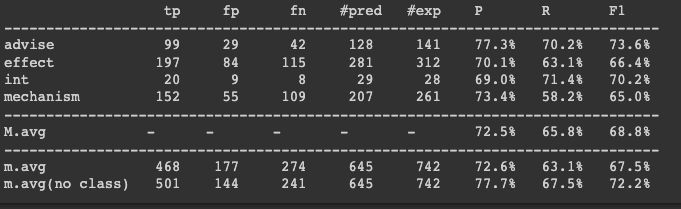
Then, we tried combining the CNN and the BiLSTM and this is where we got the best result. We think that the presence of the CNN enhances the ability of the BiLSTM to better differentiate between interactions by offering additional supporting evidence.

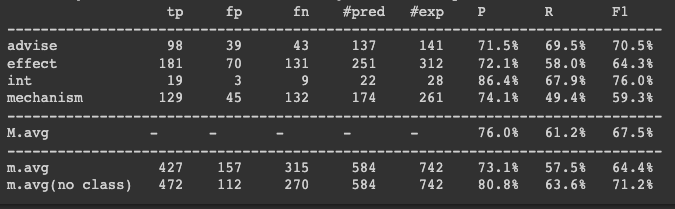
Architectures tried:

| Architecture | F1 score |
| --- | --- |
| LSTM | 61.03% (best obtained was 63.6%) |
| CNN + LSTM | 58.1% |
| BiLSTM | 66% |
| CNN + BiLSTM | 67.47% |

****

**Figure2: Final model**

****

****

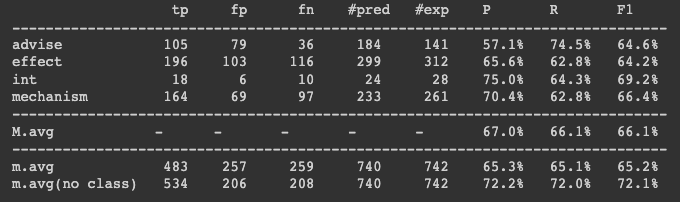
****

Figure 3: Best results architecture CNN + BiLSTM

**Conclusions**

The CNN + BiLSTM architecture, complemented with an intermediate Dense layer, stands out as the most optimal choice. We believe that CNN assists the BiLSTM in effectively discerning interactions.

The combination of the convolutional and recurrent components exploits the strengths of both architectures. The CNN excels in capturing local patterns and extracting relevant features from the input, while the BiLSTM comprehends long-range dependencies and encodes sequential information. By integrating these two components, the model benefits from the CNN's ability to highlight discriminative features, which in turn enhances the BiLSTM's capacity to distinguish meaningful interactions within the data.

The CNN has proven to be effective and by twitching some parameters we were able to improve the base model.

Even though the outcomes vary significantly across individual runs, we were able to achieve satisfactory overall results. Throughout the laboratory we tried a significant amount of different things in order to observe the impact on our model. It was quite time consuming since it takes a while for the model to run and also we tried every different combination at least three times due to the variability in the results mentioned above.

It is worth mentioning that we did try different activation functions and different parameters but as they did not improve the score we left them out of the document.

**Code**

Build\_network:

def build\_network(codes) :

# sizes

n\_words = codes.get\_n\_words()

max\_len = codes.maxlen

n\_labels = codes.get\_n\_labels()

n\_lc\_words = codes.get\_n\_lc\_words()

n\_pos = codes.get\_n\_pos()

n\_lemmas = codes.get\_n\_lemmas()

emb\_dim = 200

emb\_ind = load\_glove\_embeddings(emb\_dim)

inptW = Input(shape=(max\_len,))

inptLW = Input(shape=(max\_len,))

embW = Embedding(input\_dim=n\_words, output\_dim=emb\_dim, input\_length=max\_len, embeddings\_initializer=Constant(glove\_emb\_matrix(emb\_ind, 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(glove\_emb\_matrix(emb\_ind, codes.lc\_word\_index, emb\_dim)), mask\_zero=False)(inptLW)

#embW = Embedding(input\_dim=n\_words, output\_dim=emb\_dim,

# input\_length=max\_len, mask\_zero=False)(inptW)

#embLW = Embedding(input\_dim=n\_lc\_words, output\_dim=emb\_dim,

# input\_length=max\_len, mask\_zero=False)(inptLW)

#embLcW = Embedding(input\_dim=n\_lc\_words, output\_dim=emb\_dim,

#input\_length=max\_len, mask\_zero=False)(inptLcW)

#dropW = Dropout(0.2)(embW)

#dropLW = Dropout(0.2)(embLW)

#embs = concatenate([dropW, dropLW])

embs = concatenate([embW,embLW])

conv = Conv1D(filters=30, kernel\_size=4, strides=1, activation='relu', padding='same')(embs)

#flat= Flatten()(conv)

#dense1 = Dense(100, activation=LeakyReLU(alpha=0.1))(flat)

#dense\_drop = Dropout(0.1)(dense1)

#pooling = MaxPooling1D(pool\_size=3, strides=1, padding='same')(conv)

#pooling = AveragePooling1D(pool\_size=3, strides=1, padding='same')(conv)

#conv2 = Conv1D(filters=30, kernel\_size=3, strides=1, activation='relu', padding='same')(pooling)

#pooling2 = MaxPooling1D(pool\_size=2, strides=1, padding='same')(conv2)

# drop = Dropout(0.1)(dense1)

# drop = Dropout(0.2)(dense1)

#flat = Flatten()(pooling)

#lstm = LSTM(units=300)(conv)

lstm = Bidirectional(LSTM(units=300))(conv)

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

#flat = Flatten()(conv)

#activation = Activation('relu')(lstm)

drop = Dropout(0.2)(dense1)

#flat = Flatten()(drop)

out = Dense(n\_labels, activation='softmax')(drop)

model = Model(inputs=[inptW, inptLW], outputs=out)

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

return model



Glove embedding:

def load\_glove\_embeddings(emb\_dim):

embeddings\_index = {}

try:

f = open(utilsdir + f"/Embeddings/glove.6B.{emb\_dim}d.txt")

except FileNotFoundError:

print("GloVe missing. Download it from http://nlp.stanford.edu/data/glove.6B.zip")

sys.exit()

with open(utilsdir + 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))

return embeddings\_index

def glove\_emb\_matrix(embeddings\_index, word\_index, emb\_dim):

num\_tokens = len(word\_index)

found = 0

not\_found = 0

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:

embedding\_matrix[i] = embedding\_vector

found += 1

else:

not\_found += 1

print("\tShape of embedding matrix: %s" % str(embedding\_matrix.shape))

print("\tNo. of words not found in GloVe: ", not\_found)

return embedding\_matrix

