Assignment 3

Language Identification (Sentence Classification)

Course: Spoken and Written Language Processing

(POE-GCED)

Jordi Aguilar

Miquel Escobar

April 15, 2020

Contents

1 Introduction

The goal of this homework assignment is to classify a set of text lines to their corresponding language (also known as Language Identification or LI). In order to do so we have implemented and tested several models which are explained in detail in section **3. Models** and tested and analyzed in section 4. Models Performance Evaluation.

To test the variations of the model, we have modified the given baseline **Kaggle** notebook with the corresponding contributions. We have also proposed models with alternative architectures, in which we process words and phrases within the same text separately (see section **3.3**).

The performance report includes an analysis of the training (in terms of loss function evolution, training time and number of parameters) for each of the models, and a comparison of accuracy. These analysis comparisons often appear in the form of tables and visualizations in order to provide the reader an easier understanding of the obtained results.

2 Exploratory data analysis

To familiarize ourselves with the provided data we have performed a simple exploratory analysis. We are given 4 data files:

- **labels.csv**: contains information about all languages.
- **x_test.txt**: contains the test text lines to predict.
- **x_train.txt**: contains the training text lines to train the model, labeled by y_train.txt.
- **y_train.txt**: contains the labels of the training data in x_train.txt.

The core of the data is in both `x_train.txt` and `x_test.txt`, as they contain the text lines to classify. To gain a precise idea of the scope of the task, we can take a look at the basic statistics of the provided texts with respect to the number of characters:

dataset	#	Average size	Median size	Stdev.	Min size	Max size
test	117500	371.83	272	580.58	141	107301
train	117500	369.95	272	457.2	141	40579

We see that the amount of characters per text line is very similar (the median is, indeed, the exact same) in both datasets. It is also worth noting that there are significant outliers, as we can see with the maximum number of characters being well over the average number plues tens of times the standard deviation. Now let's take a look at the number of characters distribution [1]:

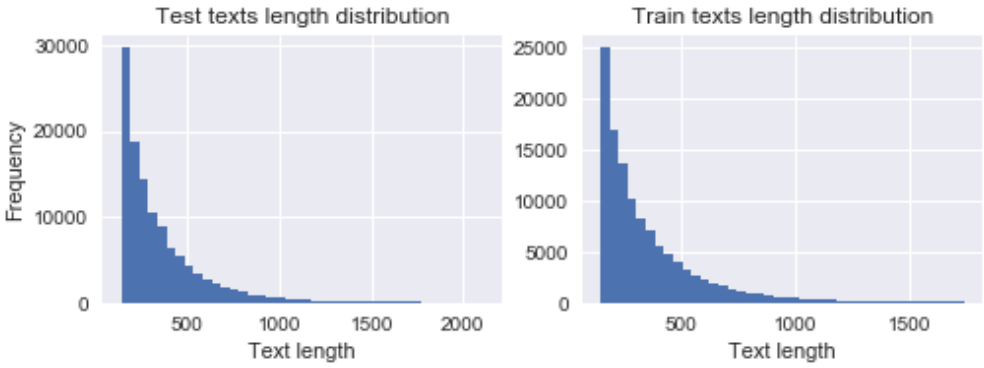

Figure 1: Texts' characters length distribution.

[1] In order to tackle the problem of outliers, we have omitted all those values such that $abs(x - mean) > 3 * stdev$.

3 Models

3.1 Base model

The base model that we have been given is a character level RNN. The goal is to predict the language of a given paragraph. The default input consists of paragraphs of different languages and lengths and these are the steps that it follows:

1. **Paragraphs to numbers**: First of all we must index every char of the sentence to embed them. We will also consider the blank space as a character and even an unknown char in case there is any character in the test set that we haven't seen in the training data. Then we split the paragraphs in batches of similar size and finally we pad the short paragraphs so they have the same size as the longer paragraph of the batch. Once we have this tensor of size $B \times T$ we are ready to train the neuronal network.

2. **Chars to embeddings**: an embedding is assigned to each char (initialized randomly with the use of `torch.nn.Embedding` class). The tensor size is now $B \times T \times E$.

3. **LSTM/GRU Layer**: Here we combine the embeddings of the sequence. We will try different parameters as making it bidirectional, adding layers or increasing the hidden size. The output has size $B \times T \times H$.

4. **Reduction and Prediction**: Finally, to predict the final language we can use a FC layer. The problem arises when we have different sequence lengths T for each batch. To overcome this problem, we can obtain a tensor of shape $B \times H$ by reducing the dimension of T by taking the maximum, the average, adding it... Once we have homogeneous tensors of size $B \times H$, we use a FC layer to predict the language of each paragraph.

3.2 Base model variations

A way to improve the performance of our model is by simply doing little changes in the architecture and testing the behaviour of different hyper-parameters.

3.2.1 Changing the RNN

We will try two different RNN

- **LSTM**: For each element of the sequence we compute its output and the "contribution" that the previous element of the sequence has on its output. This result will also go through the next element of the sequence as well as a "Cell State". This layer called Cell State that runs straight down the entire sequences keeps information of all the previous cells. The forgetting, updating and input of each cell is done by three gates.
- **GRU**: The main difference between the LSTM is that it doesn't has a Cell State, and it just exposes the full hidden content of the previous cell without any control. It only has 2 gates and less parameters than the LSTM. In practice they have similar performance but GRU trains faster.

These RNN computes the hidden states from left to right, but we would like that a character had the context from the whole word, not only the left side. For this reason we can also add the possibility of bidirectional RNN, where we basically perform a second RNN from right to left, doubling the number of parameters and the size of the output.

3.2.2 The Pooling layer

The step between the recurrent layer and the fully connected layer is significant, we come from a tensor of shape $B \times T \times H$ and want a tensor of shape $B \times H$. The methods that we are going to try are:

- Max-pool layer
- Mean-pool layer
- Combination of max-pool and mean-pool trough concatenation and addition
- Apply an activation function like *ReLu*. (We will do this implicitly, padding the output of the LSTM with 0 and taking the maximum).

3.2.3 Words instead of characters

Another interesting approach that we have tried is to embed each word instead of each character. We know that there are a lot of unique words in a language rather than characters, so we will evaluate this option.

3.2.4 Early stopping

An ideal model must generalize the data, not only perform properly in the training data. Given that this task requires little time and we perform a lot of *epochs*, early stopping avoids overtraining the model too much and interrupts the training when the validation loss does not improve. This is done by setting a *patience*: an integer that limits the number of epochs without an improvement on the validation loss. It keeps the best model. The code to implement this method has been extracted from this repository, but it is extremely intuitive.

```
class EarlyStopping:
    """Early stops the training if validation loss doesn't improve after a
    given patience."""
    def __init__(self, patience=7, verbose=False, delta=0):

        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta

    def __call__(self, val_loss, model):

        score = -val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta: #Check if -val_loss has
    improved
            self.counter += 1
            print(f'EarlyStopping counter: {self.counter} out of {self.
    patience}')
```

```
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        '''Saves model when validation loss decrease.'''
        if self.verbose:
            print(f'Validation loss decreased ({self.val_loss_min:.6f} --> {
    val_loss:.6f}).  Saving model ...')
        torch.save(model.state_dict(), 'checkpoint.pt')
        self.val_loss_min = val_loss
```

Listing 1: Class `Early_stopping`

Befor entering the the `for` loop where we perform each epoch (train and validate the model), we initialize our `Early_Stopping` object with the desired `patience`, and inside the `for` loop we add this few lines:

```
early_stopping(val_loss, model)
if early_stopping.early_stop:
    print(f'Early stopping after {epoch} epochs')
    break
```

Listing 2: Checking if we have drained our patience

Once we have breaked or completed the `for` loop, we load the model with better validation loss. The main drawbacks of this method is that we can't re-train the model with the entire data, because the early stopping would have been useless.

3.3 Processing words and phrases separately

We have thought that the baseline model could improve with more attention on the words. This is why we have decided to change a little bit the architecture. First of all our model is going to focus on the characters of the words separately through a LSTM, then we are going to extract a single vector of each word, put together the words of the same paragraph and input them in another LSTM. [2]

We have also experimented changing the first LSTM for a CNN, inspired by the paper *Character-Aware Neural Language Models*, and both LSTM for a couple of CNN. The filters of the CNN are as wide as the last dimension of the input (the `embedding_size` in the case of the first CNN and the `hiden_size1` in the case of the second), this means that the last dimension of the output will be 1.

3.3.1 Baseline architecture

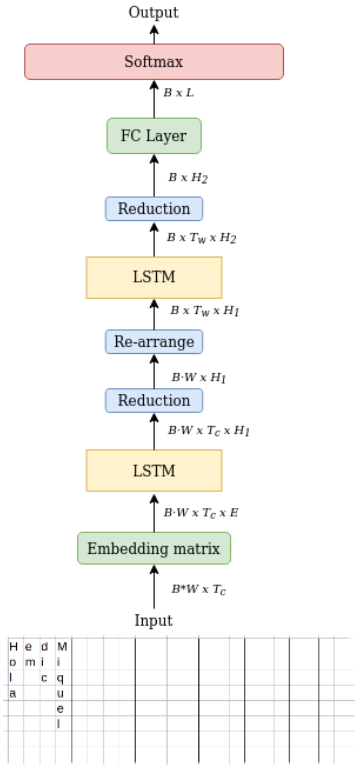

Figure 2: Architecture of the character-word neuronal network

[2] In order to avoid exceeding the GPU memory, we select words up to the 25th character and sentences up to the 100th word.

3.3.2 Modified code

The code below includes has a CNN followed by a RNN.

```
class CharWordRNNClassifier(torch.nn.Module):

    def __init__(self, input_size, embedding_size, hidden_size1, hidden_size2,
     output_size, model="lstm", num_layers=1, dropout = 0.5, bidirectional=
    False):
        super().__init__()
        self.model = model.lower()
        self.hidden_size1 = hidden_size1
        self.hidden_size2 = hidden_size2
        self.embed = torch.nn.Embedding(input_size, embedding_size,
    padding_idx=pad_idx)

        self.rnn1 = RNN(embedding_size, hidden_size1, model = model,
    bidirectional = bidirectional)
        self.rnn2 = RNN(hidden_size1*2, hidden_size2, model = model,
    bidirectional = bidirectional)

        #self.cnn1 = CNN(embedding_size, hidden_size1)
        #self.cnn2 = CNN(hidden_size1, hidden_size2)

        self.h2o = torch.nn.Linear(hidden_size2*2, output_size)
        self.dropout1 = torch.nn.Dropout(0.3)
        self.dropout2 = torch.nn.Dropout(dropout)


    def forward(self, input, w_lengths):
        # (B*W) x T_c

        words = self.embed(input).unsqueeze(1)
        # (B*W) x 1 x T_c x E

        words, _ = self.cnn(words).max(-1)
        # (B*W) x H1

        words = words.split(w_lengths)
        # tuple(W_1 x H1, ... , W_B * H1)

```

```
        words = torch.nn.utils.rnn.pad_sequence(words, batch_first = True)
        # # B x T_w x H1

        w_lengths = torch.tensor(w_lengths, dtype=torch.long, device=device)
        words = self.rnn(words, w_lengths)
        # B x T_w x H2

        words = self.dropout2(words)
        # B x H2
        words = self.h2o(words)
        # B x L
        return words
```

Listing 3: Architecture of the character-word neuronal network implemented using `PyTorch`

```
class RNN(torch.nn.Module):
    def __init__(self, input_size, hidden_size, model="lstm", num_layers=1,
    dropout = 0.3, bidirectional=False):
        super().__init__()
        self.model = model.lower()
        self.hidden_size = hidden_size
        if self.model == "gru":
            self.rnn = torch.nn.GRU(input_size, hidden_size, num_layers,
    bidirectional=bidirectional, batch_first = True)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(input_size, hidden_size, num_layers,
    bidirectional=bidirectional, batch_first = True)

    def forward(self, encoded, c_lengths):

        # B x T x E
        encoded, _ = torch.nn.utils.rnn.pack_padded_sequence(encoded,
    c_lengths, batch_first = True, enforce_sorted=False))
        # Packed B x T x E
        encoded, _ = self.rnn(packed)
        # Packed B x T x H
        encoded, _ = torch.nn.utils.rnn.pad_packed_sequence(encoded,
    padding_value=float(0), batch_first = True)
        # W x T x H

```

```
        return encoded
```

Listing 4: RNN class implemented in PyTorch

```
class CNN(torch.nn.Module):
    def __init__(self, embedding_size, num_out_fmaps, width = 5, bias=True):
        super().__init__()

        self.conv = torch.nn.Conv2d(1, num_out_fmaps, [width, embedding_size],
     padding = [width//2, 0], bias = bias)
        self.tanh = torch.nn.Tanh()

    def forward(self, input):
        # B x 1 x T x E

        output = self.tanh(self.conv(input))
        # B x C x T x 1

        output = output.squeeze(-1)
        # B x C x T

        return output
```

Listing 5: CNN class implemented in PyTorch

4 Models performance evaluation

In this section we show the performance of each of the models by means of tables and visualizations. For visualization and aesthetic purposes, we have "encoded" the model names into a more compressed form. Below are the compressed names and main characteristics of all the tested models:

Variations of the baseline model

Name	Embed. size	RNN h. size	Batch size	Epochs	Pooling	Pool. value	Comments
base	64	256	256	25	max	$-\infty$	-
V1	128	256	256	25	mean	0	-
V2	128	256	256	25	max	0	-
V3	128	256	256	25	max	$-\infty$	-
V4	128	256	256	25	max&mean	0	Final addition
V5	128	256	256	25	max&mean	0	Final concatenation
V6	128	256	256	25	max	0	Dropout 0.15. Bidirectional
V7	128	256	256	25	max	0	GRU

W1	Words instead of characters. Dropout = 0.15.
W2	Words instead of characters. Dropout = 0.5.
ES	Early stopping.

Models processing words and phrases separately

Name	Embed. size	H. size 1	H. size 2	Words	Phrases	Batch size	Epochs
S1	64	64	128	LSTM	LSTM	32	25
S2	64	128	256	CNN	LSTM	128	25
S3	64	256	512	CNN	CNN	128	25

The plots and tables have been generated by the notebook `models-metrics.ipynb`. The data has been stored during the training of the models. It is important to note that the granularity of the data depends on the metric as for instance, the training accuracy is available for all generated batches while the validation accuracy only for each of the epochs.

4.1 Training time

The training time helps to identify the computational costs and scalability of the model and its training. A model might perform well with predictions, but might have a huge training duration, which would imply it is not the best fit for a production environment.

In this case, each of the models was trained only once (as we are in a testing/development environment), reason for which it is not a critical metric to choose the best overall-performing model, but it is always interesting to observe what relation there is between the accuracy, performance and training time.

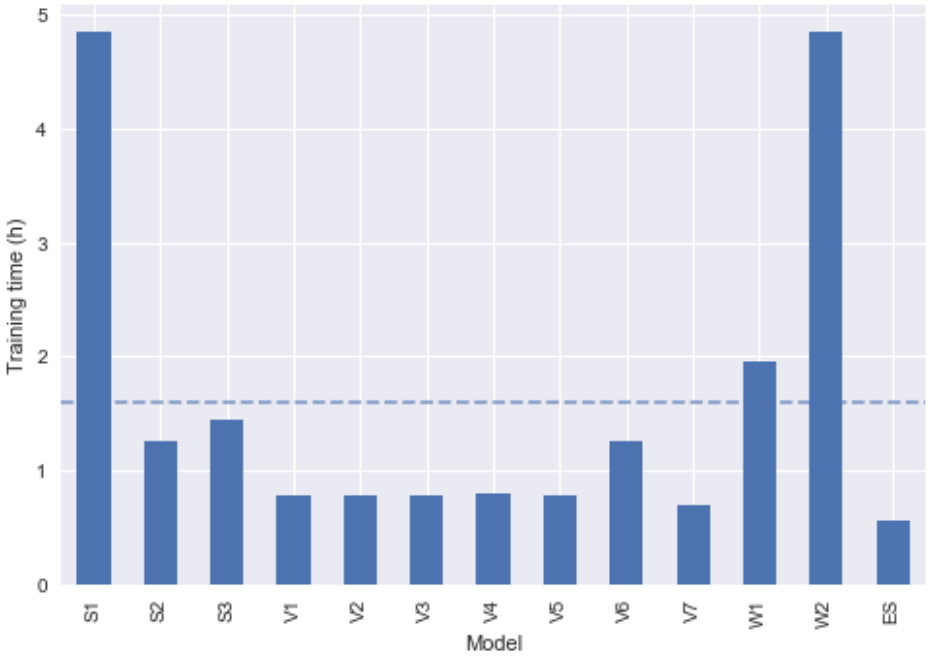

Figure 3: Training time for each of the models. The dashed blue line indicates the average.

We observe that the model with a longer training duration is, indeed, the second best in terms of accuracy (**S1**). We also observe how the models that are variations to the first model (section **3.1**) are trained in a small amount of time in comparision to the other proposed architectures, with the exception of the model **ES** (Early Stopping) which, as expected, was the fastest to train.

It is also interesting to see how the models based on LSTM (as **S1**) take much longer to train than those based on GRU (as **S3**). On the other hand, the models in which we use words instead of characters (**W1** and **W2**) are also slow to train, as we could expect given the fact that the embeddings are made on thousands of words instead of on hundreds of characters.

4.2 Accuracy

The accuracy indicates the proportion of good predictions made by the model. Of course, as the training advances we expect the accuracy to improve. This is what we observe in all models (for both training and validation accuracy).

All models seem to converge pretty fast onto their final accuracy limit. In fact, the Early Stopping model only stops at epoch 24, out of the maximum of 25 set for all models.

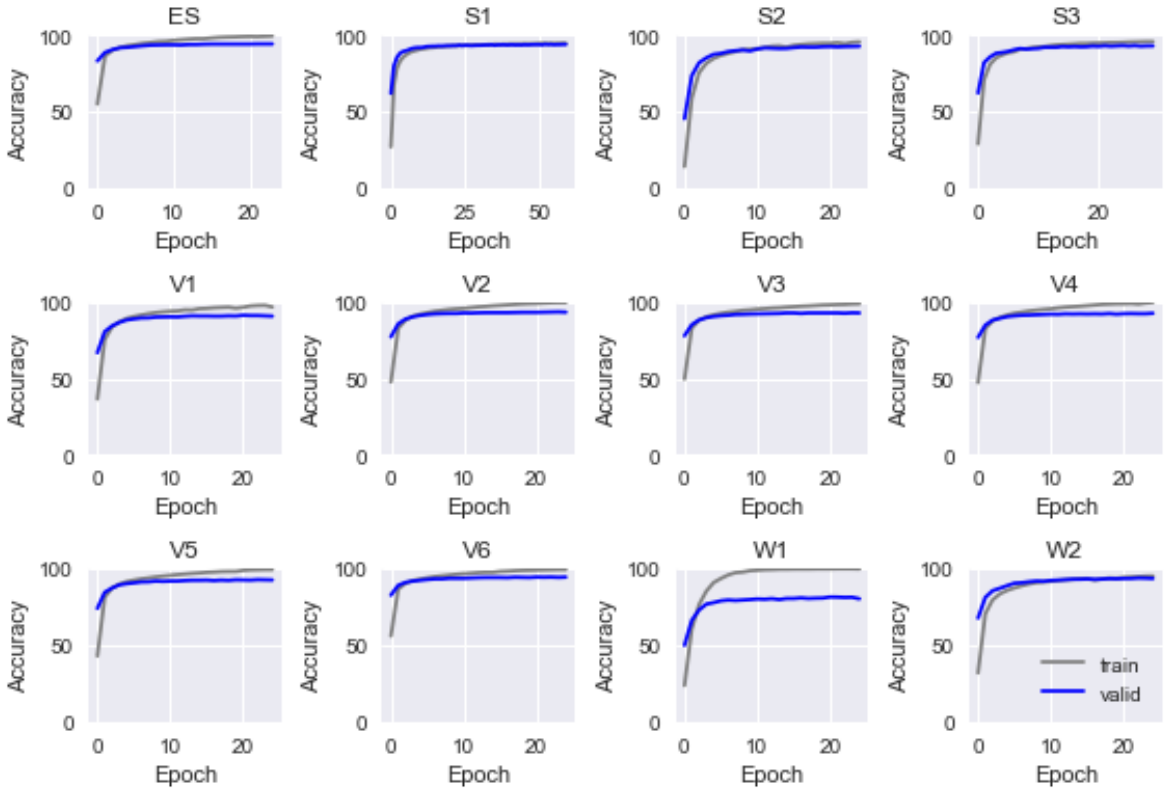

Figure 4: Accuracy value evolution during training for each of the models.

It is worth noting that model **W1** training clearly overfits the training data: the training accuracy is almost 100% while the validation accuracy is pretty low.

After training the models, we obtained the test accuracy for each of the models (by submitting their predicitons to the *Kaggle* competition). Based on this metric, the best performing model is **V6**, and only $\frac{5}{13}$ models improve the baseline (corresponding to the given model).

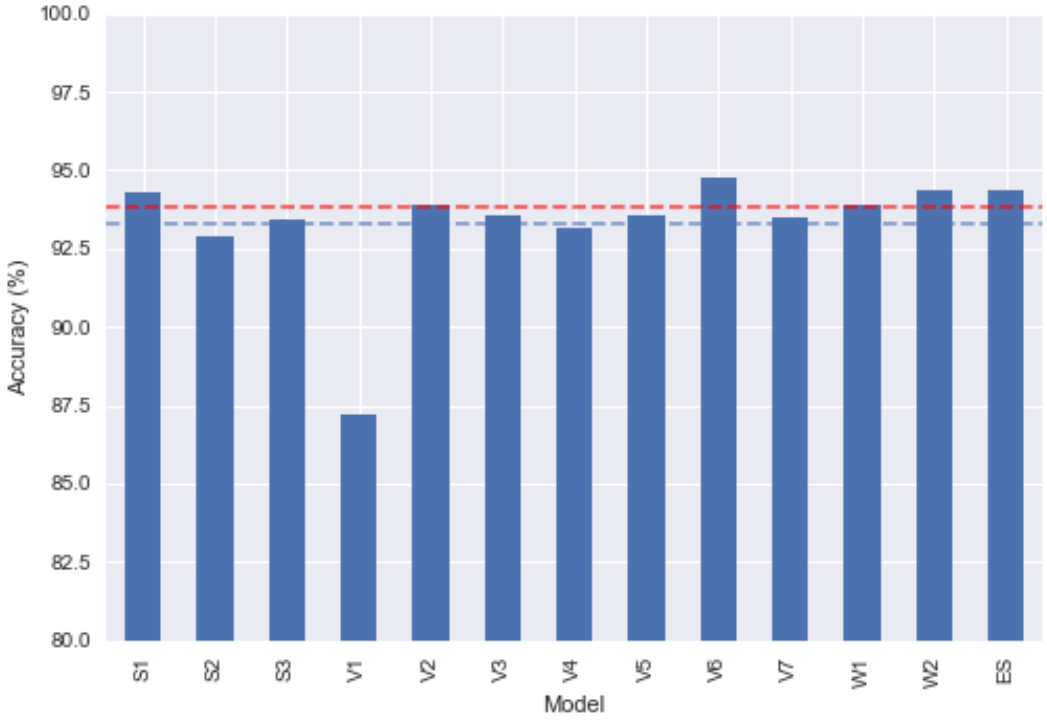

Figure 5: Test accuracy value for each of the models. The dashed blue line indicates the average. The red dashed line indicates the baseline accuracy.

After analyzing all the results, from our perspective the best model and the one we would choose for a production environment is model **V6**. Not only it is the one with highest test accuracy, but also has a reasonably low training time - slightly higher than variations with less training parameters, and incredibly lower than the second best model in terms of accuracy (**S1**).

4.3 Summary table

After a detailed exploration into each of the collected metrics, we have summarized the models into the table below. This helps to identify the best performing models taking into account all available information.

	Embed_dim	RNN size	Pooling layer	Pool. value	Comments	Train Acc.	Test Acc.
baseline	64	256	max	$-\infty$	-	98.92	93.86
	128	256	mean	0	-	89.66	87.20
	128	256	max	0	-	99.47	93.90
	128	256	max	$-\infty$	-	98.71	93.55
	128	256	max&mean	0	Final addition	99.67	93.14
	128	256	max&mean	0	Final concatenation	99.47	93.54
	128	256	max	0	Dropout 0.15. Bidirectional	99.47	94.78
	”	”	”	”	+ Early Stopping (16 epochs)	98.08	94.45
	128	256	max	0	GRU	98.08	93.46
				Character-Word RNN architecture			
	Embed_dim	Hidden_size1	Hidden_size2		Layers		
	64	64	128		LSTM - LSTM	94.8	94.3
	64	128	256		CNN - LSTM	95.7	92.9
	64	256	512		CNN - LSTM	97.7	93.4

It is also worth mentioning that the models we have thought of have been following the goal of reducing overfitting - a problem we have bumped into more than we would have liked. That is also one of the main reasons for which in some networks architectures we have not performed hyperparameter tuning, but instead we have jumped directly into new designs expecting this reduction of overfitting.

5 Conclusions

To wrap up this report, we can say that we now understand more the different approaches to Language Identification problems. It has probably been the assignment with more flexibility in terms of designing and implementing the models, because of the multiple effective manners to solve the problem.

One of the positive notes of this assignment is the relatively small training time for the implemented models. Given that we have to predict one output out of 235 possibilities (and not thousands as in previous assignments), and also given that the amount of training data was not exaggerately large, the implemented models did not take that long to train and this allowed us to train more models than usual, which was great for learning.

We can also conclude that Language Identification is a much easier task than, for instance, missing word predicitons, at least in terms of accuracy. The reason we can obtain accuracies near the 100% mark (in the previous assignment we were around 30-35%) has a lot to do, again, with the fact that the number of output possibilities (235) is relatively low.

In general, it has been a great assignment to increase the knowledge acquired in the previous lab and expand our abilities in the world of Natural Language Processing.