LAB 3

LANGUAGE IDENTIFICATION

Sentence Classification

POE

DATA SCIENCE AND ENGINEERING
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Adrián Cerezuela Hernández - 48222010A

Ramon Ventura Navarro - 21785256R

April 2024

Contents

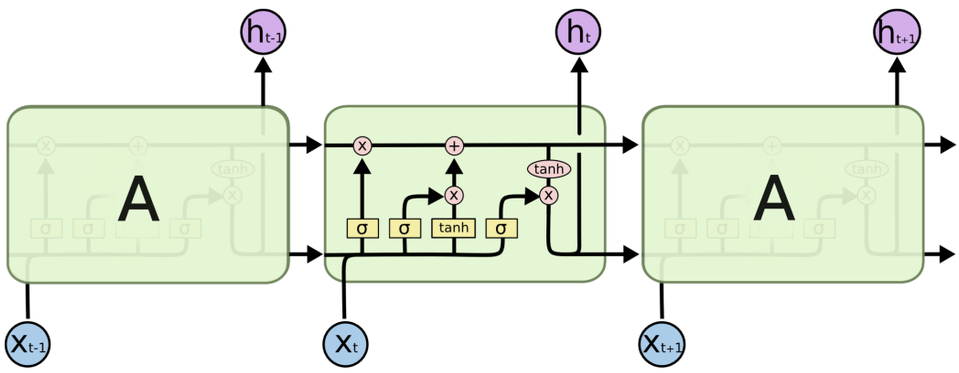
1 Introduction and baseline model

The task of determining the language of a given text is called language identification. This can range from recognizing the language of a single word to figuring out the language of an entire document or conversation. Sentence Classification is a particular form of Language Identification that entails sorting individual sentences into predetermined categories based on their content or other pertinent characteristics.

In this assignment we are asked to develop and compare different models with the aim of working on language identification using a database created from Wikipedia paragraphs. Our work will be based on a character-based RNN Baseline from which additional contributions will be attempted to the analysis, optimization, or comparative study of the proposed model and task.

The data used in this task is sourced from three primary datasets: x_train, y_train, and x_test. The x_train and x_test datasets comprise text samples from various sources written in diverse languages, while y_train contains the language labels corresponding to the x_train samples. We establish two dictionaries to store distinct characters and languages, utilizing indexes. It is important to note that index 0 is reserved for padding sequences and index 1 for any unidentified character.

The model consists of an architecture based on a Long Short-Term Memory (LSTM) as the recurrent layer, in addition to the preceding embedding and a subsequent linear layer. The structure of an LSTM layer is as follows:



With the goal to be able to compare the performance of the model and its modifications, a baseline model is established and given at the beginning of the assignment. Its metrics are the following ones:

Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
Train Accuracy (%)	Valid Accuracy (%)			
98.6	92.7	98.742	2766.1	1081835

Judging from the table, a slight overfitting can be observed as the training accuracy is higher than the validation accuracy. One of the primary objectives will be to correct this, in addition to attempting to increase the accuracy. Nonetheless, the initial results are very promising.

From this point onward, various experiments will be conducted based on modifications of the initial model, comparing their metrics to obtain the best possible model for solving the proposed task. These experiments will be based on the following points:

- Hyperparameter optimization
- Combining pooling layers
- Adding a dropout layer
- Comparing the performance with other RNNs or number of layers

2 Model Trials

2.1 Hyperparameter optimization

To start with the execution section, which will illustrate how modifications in the code impact the model's performance, tests will be conducted by changing the values of three hyperparameters: *Embedding size, Hidden layer size* and *Batch size*. Initially, these modifications will be carried out individually, keeping the other two hyperparameters distinct from the one being tested, set at their default values.

2.1.1 Embedding size

Firstly, we will increase the embedding size of the model. Its default value is *embedding_size = 64*. It is expected that the model will produce better results, although it may take longer to complete the training process.

	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Baseline	98.6	92.7	98.742	2766.1s	1081835
embedding_size = 128	98.9	92.8	98.726	2826.2s	1839083
embedding_size = 256	98.6	92.9	98.240	3072.9s	3353579

Judging by the table, there is an improvement in both the train accuracy and the validation accuracy of the model with an embedding size of 128. Additionally, despite having over 800.000 more parameters, it takes only slightly longer than the base model. This, coupled with the fact that the model with an *embedding_size* of 256 yields worse results in terms of parameter count and training time, as the accuracy remains more or less the same, suggests that a model with an *embedding_size = 128* should be used instead of *64*.

2.1.2 RNN Hidden size

Next, the same test will be conducted, reverting back to *embedding_size=64* and modifying the *hidden_size*. Its default value is 256. The results, added to the previous ones for ease of comparison, are as shown in the following table:

	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Baseline	98.6	92.7	98.742	2766.1s	1081835
embedding_size = 128	98.9	92.8	98.726	2826.2s	1839083
embedding_size = 256	98.6	92.9	98.240	3072.9s	3353579
hidden_size = 512	99.9	93.9	99.888	5654.2s	1996011
hidden_size = 768	99.9	94.5	0.426	9240.6s	3434475

Firstly, it's worth noting that although the trend for the tests throughout the practice has been to double the parameter value, in this case, the execution couldn't be carried out for *hidden_size = 1024* due to memory constraints. Therefore, the value of *768* is tested instead. For this value, we observe a sharp decline in accuracy during model training, eventually reaching 0. This could be attributed to gradient saturation, hindering the model's ability to converge. Hence, this model is automatically ruled out.

On the other hand, the model with *hidden_size = 512* shows better results in terms of accuracy compared to the base model and the previously tested ones. It's prominent that as the accuracy increases, the model also utilizes a higher number of parameters and takes twice as long to train. In this case, considering our resources, balancing the trade-off between time and accuracy, a model with *hidden_size = 512* would be a suitable choice for this task.

2.1.3 Batch size

Continuing with hyperparameter optimization, different values of *batch_size* will be tested to potentially reduce overfitting. In this case, testing will include values both above and below *256*, the value defined in the baseline model. Similar to the previous point, the other two parameters will be reset

to their default values. The results, added to the previous ones for ease of comparison, are displayed in the following table:

	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Baseline	98.6	92.7	98.742	2766.1s	1081835
embedding_size = 128	98.9	92.8	98.726	2826.2s	1839083
embedding_size = 256	98.6	92.9	98.240	3072.9s	3353579
hidden_size = 512	99.9	93.9	99.888	5654.2s	1996011
hidden_size = 768	99.9	94.5	0.426	9240.6s	3434475
batch_size = 64	99.0	93.0	98.769	4309.4s	1081835
batch_size = 128	99.3	92.8	99.048	3490.9s	1081835
batch_size = 512	97.6	92.6	97.706	2273.1s	1081835
batch_size = 1024	96.8	92.6	97.031	2203.9s	1081835

As seen in the presented results, models with larger *batch_size* train faster but generate poorer results, whereas as the size is minimized, the opposite occurs: better metrics are obtained in terms of accuracy, even though with longer training times. However, these results do not significantly deviate from the previously observed ones. Therefore, it does not appear that modifying the *batch_size* parameter will lead to an improvement for the model chosen thus far.

2.1.4 A little bit of tuning

Once various values have been tested individually, models with multiple parameters changed will be executed. The results obtained are displayed in the table below. Once again, all tested models are presented together to facilitate comparison and subsequent selection.

	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Baseline	98.6	92.7	98.742	2766.1s	1081835
embedding_size = 128	98.9	92.8	98.726	2826.2s	1839083
embedding_size = 256	98.6	92.9	98.240	3072.9s	3353579
hidden_size = 512	99.9	93.9	99.888	5654.2s	1996011
hidden_size = 768	99.9	94.5	0.426	9240.6s	3434475
batch_size = 64	99.0	93.0	98.769	4309.4s	1081835
batch_size = 128	99.3	92.8	99.048	3490.9s	1081835
batch_size = 512	97.6	92.6	97.706	2273.1s	1081835
batch_size = 1024	96.8	92.6	97.031	2203.9s	1081835
embedding256 - hidden512 - batch64	99.7	93.7	99.586	9114.5s	4464363
embedding128 - hidden512 - batch64	99.8	94.0	99.725	8714.5s	2818795
embedding128 - hidden512 - batch256	**99.9**	**94.1**	**99.837**	**5819.4s**	**2818795**

Judging by the table, the two best models, considering both accuracy and training time along with parameter count, are the one that only modifies the *hidden_size = 512* compared to the baseline model and the one that additionally modifies the *embedding_size = 128*. Both fall within similar ranges for each metric, with the number of parameters being where they differ the most. Despite the latter having a higher number of parameters, it also provides a 0.2% higher validation accuracy. The decision is primarily based on the slight increase in accuracy, thus the selected model is the latter one, highlighted in bold in the table.

Finally, it has been tested by adding bidirectionality to the best performing model *embedding128 - hidden512 - batch256*, but due to the lack of memory in our machine it has not been possible. Therefore, we have implemented this functionality in a *embedding64 - hidden256 - batch128* model, which the machine is able to compute, and these are the metrics found.

	Bidirectional	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
		Train Accuracy (%)	Valid Accuracy (%)			
embedding128 - hidden512 - batch256	**No**	**99.9**	**94.1**	**99.837**	**5819.4s**	**2818795**
embedding64 - hidden256 - batch128	Yes	99.9	93.9	99.818	5587.7s	1471723

As seen above, adding the bidirectional property does not improve the model's validation accuracy and therefore seems not the best option so far. However, it's worth notating that both training time and parameters are reduced significantly when introducing bidirectionality, specially the latter. In conclusion, we will still take the *embedding128 - hidden512 - batch256* model as the best option so far.

2.1.5 Optimizer

Using the chosen model, different optimizers will be tested. The baseline model, as well as all those used in the experiments, utilize the *Adam* optimizer. Alternatives to Adam include *RMSProp* (Root Mean Square Propagation) and *SGD* (Stochastic Gradient Descent).

$$optimizer = torch.optim.RMSprop(model.parameters())$$

$$optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9)$$

While declaring *RMSProp* is sufficient with the model's parameters, for proper execution of *SGD*, it's necessary to specify values for both the learning rate and momentum. A common default choice for each is lr = 0.01 and momentum = 0.9. The results are as follows:

	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Chosen model (Adam Opt.)	**99.9**	**94.1**	**99.837**	**5819.4s**	**2818795**
Model with RMSProp Opt.	95.0	89.6	94.562	5844.4s	2818795
Model with SGD Opt.	88.9	87.8	89.422	5870.3s	2818795

As can be seen, the best metrics are obtained with the *Adam* optimizer. Therefore, it doesn't change from the model chosen in the previous section. From this point forward, we will refer to the unidirectional model previously identified as *embedding128 - hidden512 - batch256* as the *Best Model*. Additionally, we will also refer to the *embedding64 - hidden256 - batch128* model as the *Best Bidirectional Model*.

2.2 Combining pool layers

In the following section, we will study the effect of combining the output of the max-pool layer already existing in the baseline model with the output of a new mean-pool layer. Specifically, we'll explore the distinctions between combining them through concatenation and addition.

Even though *-inf* padding was obtained to compute the max-pool layer, it was necessary the usage of 0-padding in the output of the LSTM in order to avoid *-inf* values for the mean-pool layer computation.

Below are the corresponding code snippets, metrics tables, and lastly, the conclusions extracted for both concatenation and addition operations:

2.2.1 Concatenation

```python
class CharRNNClassifier(torch.nn.Module):

    def __init__(self, input_size, embedding_size, hidden_size, output_size, model="lstm", num_layers=1, bidirectional=False, pad_idx=0):
        super().__init__()
        self.model = model.lower()
        self.hidden_size = hidden_size
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        if self.model == "gru":
            self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        self.h2o = torch.nn.Linear(2 * hidden_size, output_size) # Incresase size to match combined_output size

    def forward(self, input, input_lengths):
        # T x B
        encoded = self.embed(input)
        # T x B x E
        packed = torch.nn.utils.rnn.pack_padded_sequence(encoded, input_lengths)
        # Packed T x B x E
        output, _ = self.rnn(packed)
        # Packed T x B x H
        # Important: you may need to replace '-inf' with the default zero padding for other pooling layers
        padded, _ = torch.nn.utils.rnn.pad_packed_sequence(output, padding_value=float('-inf'))
        padded_0, _ = torch.nn.utils.rnn.pad_packed_sequence(output) # Zero Padding
        # T x B x H
        max_pool_output, _ = padded.max(dim=0)  # Max-pooling
        mean_pool_output = padded_0.mean(dim=0)  # Mean-pooling
        combined_output = torch.cat((max_pool_output, mean_pool_output), dim=-1) # Output combination
        # B x H
        combined_output = self.h2o(combined_output)
        # B x O
        return combined_output
```

Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
Train Accuracy (%)	Valid Accuracy (%)			
99.9	93.7	99.843	7155.0	2939115

2.2.2 Addition

```python
class CharRNNClassifier(torch.nn.Module):

    def __init__(self, input_size, embedding_size, hidden_size, output_size, model="lstm", num_layers=1, bidirectional=False, pad_idx=0):
        super().__init__()
        self.model = model.lower()
        self.hidden_size = hidden_size
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        if self.model == "gru":
            self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        self.h2o = torch.nn.Linear(hidden_size, output_size)

    def forward(self, input, input_lengths):
        # T x B
        encoded = self.embed(input)
        # T x B x E
        packed = torch.nn.utils.rnn.pack_padded_sequence(encoded, input_lengths)
        # Packed T x B x E
        output, _ = self.rnn(packed)
        # Packed T x B x H
        # Important: you may need to replace '-inf' with the default zero padding for other pooling layers
        padded, _ = torch.nn.utils.rnn.pad_packed_sequence(output, padding_value=float('-inf'))
        padded_0, _ = torch.nn.utils.rnn.pad_packed_sequence(output) # Zero Padding
        # T x B x H
        max_pool_output, _ = padded.max(dim=0)  # Max-pooling
        mean_pool_output = padded_0.mean(dim=0)  # Mean-pooling
        combined_output = max_pool_output + mean_pool_output # Output combination
        # B x H
        combined_output = self.h2o(combined_output)
        # B x O
        return combined_output
```

Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
Train Accuracy (%)	Valid Accuracy (%)			
99.8	93.7	99.846	7181.1	2818795

2.2.3 Conclusions

When comparing both models with the best model so far, we first see that both of the two new methods perform essentially in the same way, even though the concatenation of outputs led to a slightly larger amount of parameters. We can then affirm there's not much difference between the concatenation and addition of outputs from the different pool layers. Moreover, it is also clear that they do not seem to improve the implementation with a single max-pool operation, making us discard the usage of a mean-pool layer from now on.

Additionally, we have implemented both approaches to the *Best Bidirectional Model* to see how the combination of pool layers would affect to the performance in a different configuration. It stands out that in this second case, this implementation led to a higher improvement of the validation accuracy than for the unidirectional LSTM, but still not significant enough to be considered the best model in any case.

	Bidirectional	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
		Train Accuracy (%)	Valid Accuracy (%)			
Best Model	**No**	**99.9**	**94.1**	**99.837**	**5819.4s**	**2818795**
Mean-Pool Layer Concatenation	No	99.9	93.7	99.843	7155.0	2939115
Mean-Pool Layer Addition	No	99.8	93.7	99.846	7181.1	2818795
Best Bidirectional Model	Yes	99.9	93.9	99.818	5587.7s	1471723
Mean-Pool Layer Concatenation	Yes	99.9	93.7	99.863	7009.0	1592043
Mean-Pool Layer Addition	Yes	99.9	93.8	99.878	7043.1	1471723

2.3 Adding a dropout layer

Moving forward, we will also examine the implementation of dropout layers within our best model, discussing how this enhancement contributes to our overall architecture. The aim is to decrease the overfitting by randomly dropping a proportion of neurons during each training iteration, helping the model learn more generalizable patterns.

In our case, we have tried 5 implementations of this new architecture with dropout probabilities: *0.05*, *0.1*, *0.2*, *0.4*, *0.6*. Below can be seen the code implementation for the latter dropout value and the metrics table for each of the models:

```python
class CharRNNClassifier(torch.nn.Module):

    def __init__(self, input_size, embedding_size, hidden_size, output_size, model="lstm", num_layers=1, bidirectional=False, pad_idx=0):
        super().__init__()
        self.model = model.lower()
        self.hidden_size = hidden_size
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        if self.model == "gru":
            self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        self.dropout = torch.nn.Dropout(p = 0.6)
        self.h2o = torch.nn.Linear(hidden_size, output_size)

    def forward(self, input, input_lengths):
        # T x B
        encoded = self.embed(input)
        # T x B x E
        packed = torch.nn.utils.rnn.pack_padded_sequence(encoded, input_lengths)
        # Packed T x B x E
        output, _ = self.rnn(packed)
        # Packed T x B x H
        # Important: you may need to replace '-inf' with the default zero padding for other pooling layers
        padded, _ = torch.nn.utils.rnn.pad_packed_sequence(output, padding_value=float('-inf'))
        # T x B x H
        output, _ = padded.max(dim=0)
        output = self.dropout(output)  # Applying dropout after pooling
        # B x H
        output = self.h2o(output)
        # B x O
        return output
```

	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Best Model	99.9	94.1	99.837	5819.4s	2818795
Dropout (p = 0.05)	99.8	94.1	99.723	5870.7	2818795
Dropout (p = 0.1)	**99.7**	**94.2**	**99.580**	**5825.8**	**2818795**
Dropout (p = 0.2)	99.3	94.2	99.216	5841.9	2818795
Dropout (p = 0.4)	98.0	94.2	97.832	5853.9	2818795
Dropout (p = 0.6)	96.2	94.1	95.805	5852.4	2818795

Among the tested dropout probabilities, we observe that dropout rates of *0.1*, *0.2*, and *0.4* demonstrate slightly higher validation accuracy. However, as the dropout rate increases, there is a corresponding decrease in training accuracy which mitigates overfitting (as already mentioned earlier). Therefore, we consider the model with dropout probability of *0.1* as the best performer, as it achieves a modest increase in validation accuracy without sacrificing significant training accuracy. So, the Best Model will be replaced by this one with dropout.

Before going to the next section, we have also executed a single **bidirectional** LSTM with dropout probability of *0.1* to keep tracking the effect of each of the methods studied in both unidirectional and bidirectional models. We see also a slight increase, but still under-performing when compared to the unidirectional model.

	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Best Bidirectional Model	99.9	93.9	99.818	5587.7s	1471723
Dropout (p = 0.1)	99.5	94.0	99.452	5564.5	1471723

2.4 Comparing the performance with other RNNs or number of layers

Finally, the last step to further improve our unidirectional model is to modify the layers of the network. Firstly, we will try changing the type of RNN to use. Up to now, we have been using an LSTM, but if we look at the *__init__()* method, there's an option to choose a GRU. For execution, we simply need to change the parameter *model="gru"* in the method. Below, you can find both the modified code section and the resulting metrics.

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

    def __init__(self, input_size, embedding_size, hidden_size, output_size, model="gru", num_layers=1, bidirectio
nal=False, pad_idx=0):
        super().__init__()
        self.model = model.lower()
        self.hidden_size = hidden_size
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        if self.model == "gru":
            self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        self.dropout = torch.nn.Dropout(p = 0.1)
        self.h2o = torch.nn.Linear(hidden_size, output_size)
```

	Cross Validation		Train accuracy (%)	Training time (s)	Parameters
	Train accuracy (%)	Valid accuracy (%)			
Best Model (model = LSTM)	**99.7**	**94.2**	**99.580**	**5825.8**	**2818795**
model = GRU	99.3	93.5	99.229	4777.2	2490091

Judging by the table, the metrics with an LSTM model are better in terms of accuracy compared to those obtained with a GRU model. While it's true that GRU is a type of neural network that trains faster and has fewer parameters than LSTM, the slight decrease in accuracy makes the model chosen in the previous section more suitable for our purposes.

Once we've determined the RNN type that suits us best, the last modification we'll attempt, or at least check, is the number of layers to use in our model. The *__init__()* method has a parameter called *num_layers*, and by changing its value, we can perform the relevant tests. We've attempted to execute with 2 and 3 layers. Below, you can see both the location of the modified parameter and the metrics table for each test.

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

    def __init__(self, input_size, embedding_size, hidden_size, output_size, model="lstm", num_layers=2, bidirecti
onal=False, pad_idx=0):
        super().__init__()
        self.model = model.lower()
        self.hidden_size = hidden_size
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        if self.model == "gru":
            self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        elif self.model == "lstm":
            self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        self.dropout = torch.nn.Dropout(p = 0.1)
        self.h2o = torch.nn.Linear(hidden_size, output_size)
```

	Cross Validation		Train accuracy (%)	Training time (s)	Parameters
	Train accuracy (%)	Valid accuracy (%)			
Best Model (num_layers = 1)	**99.7**	**94.2**	**99.580**	**5825.8**	**2818795**
num_layers = 2	99.8	94.0	99.756	10371.8	4920043
num_layers = 3	99.7	93.7	99.569	14371.3	7021291

As observed, as recurrent layers are added, logically, both the training time and the number of parameters increase considerably. Moreover, considering that the values of each metric fall within similar ranges, meaning that we don't achieve any significant improvement and even lose validation accuracy, we can conclude that increasing the number of layers will not provide anything that could enhance the results of our task in the case of unidirectionals LSTMs.

Before ending this section, we have also made a study on how the addition of layers and changing the type of RNN affects the *embedding64 - hidden256 - batch128* **bidirectional** model. Note that due to memory capacity limitations 3 layers executions were aborted, although they seemed interesting for the GRU architectures. Moreover, for the LSTM we have only considered the models counting with a dropout mechanism, as it has been proven that they perform better given that they address the overfitting. The models executed are the following:

RNN	No. Layers	Dropout (p = 0.1)	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
			Train Accuracy (%)	Valid Accuracy (%)			
LSTM	1	Yes	99.5	94.0	99.452	5564.5s	1471723
LSTM	2	Yes	99.7	93.9	99.735	11091.1s	3048683
GRU	1	No	99.2	93.4	98.978	5239.7s	1306859
GRU	1	Yes	99.0	93.6	98.755	5218.5s	1306859
GRU	**2**	**No**	**99.8**	**94.4**	**99.625**	**9792.4s**	**2489579**
GRU	2	Yes	99.5	94.4	99.417	9769.7s	2489579

Therefore, after these last implementations, the bidirectional approach appears to significantly improve the performance of the best unidirectional model. We see an increase of a 2% in the validation accuracy, reaching a 94.4% of accuracy against the previous 94,2%, and it becomes the final best model of this first part of the study.

Hyperparameters	*Bidirectional*	*Optimizer*	*Combining Pooling Layers*	*Dropout*	*Model*	*Layers*
embedding_size = 64 hidden_size = 256 batch_size = 128	Yes	Adam	No	No	GRU	2

3 Extra Model Architectures

3.1 Comparative analysis with other DNN architectures

In this extra section we're turning our attention to CNN models, following our examination of RNN models. While RNN models have been our primary focus due to their suitability for sequential data tasks like text processing, we're intrigued by the potential of other architectures in this context, such as CNNs.

Although CNNs are typically associated with image processing tasks, their ability to extract spatial features makes them worthy contenders for text recognition as well and we aim to gain a comprehensive understanding of their effectiveness in this specific task.

We have tried different configurations by tuning the number of filters, embedding size and adding dropout (p = 0.1). The convolutional layers maintain a consistent structure, employing filters of sizes 2, 3, 4, and 5, consecutively. These are the implemented code and results:

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

    def __init__(self, input_size, embedding_size, num_filters, filter_sizes, output_size, pad_idx=0):
        super().__init__()
        self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        self.dropout = torch.nn.Dropout(0.1)
        self.convs = torch.nn.ModuleList([
            torch.nn.Conv1d(in_channels=embedding_size, out_channels=num_filters,
                            kernel_size=fs)
            for fs in filter_sizes
        ])
        self.fc = torch.nn.Linear(len(filter_sizes) * num_filters, output_size)

    def forward(self, input, input_lengths):
        # Ensure input tensor is on the same device as the model's parameters
        input = input.to(self.embed.weight.device)

        # B x T
        embedded = self.embed(input)
        embedded = embedded.permute(1, 2, 0)
        # B x E x T
        conv_outs = []
        for conv in self.convs:
            conv_out = conv(embedded)
            pool_out = torch.nn.functional.max_pool1d(conv_out,
                                                      kernel_size=conv_out.size(2))
            pool_out = pool_out.squeeze(-1)
            conv_outs.append(pool_out)
        # B x (F * len(filter_sizes))
        output = torch.cat(conv_outs, dim=1)
        output = self.dropout(output)
        # B x O
        output = self.fc(output)
        return output
```

```
num_filters = 512
filter_sizes = (2, 3, 4, 5)
def get_model():
    model = CharCNNClassifier(ntokens, embedding_size, num_filters, filter_sizes, nlabels).to
(device)
    optimizer = torch.optim.Adam(model.parameters())
    return model, optimizer
```

Filters	Emb. Size	Dropout	Cross Validation		Train Accuracy (%)	Training Time (s)	Parameters
			Train Accuracy (%)	Valid Accuracy (%)			
32	64	Yes	96.9	92.6	96.702	442.60	750827
32	128	No	99.2	91.9	99.089	640.19	1471211
64	64	Yes	99.1	93.6	98.979	1313.77	809707
64	128	No	99.7	93.4	99.683	1842.04	1558763
100	128	No	99.8	94.2	99.707	958.78	1657259
100	128	Yes	99.4	94.2	99.372	972.40	1657259
128	64	Yes	99.6	94.3	99.544	748.66	927467
128	128	No	99.8	94.3	99.726	1073.08	1733867
256	64	No	99.8	94.3	99.817	1410.45	1162987
256	64	Yes	99.7	94.7	99.633	1385.30	1162987
256	128	No	99.7	94.5	99.666	1883.06	2084075
512	64	No	99.8	94.6	99.654	2590.28	1634027
512	64	Yes	99.6	94.7	99.56	2585.77	1634027
512	128	No	99.6	94.7	99.590	3458.43	2784491
512	128	Yes	99.5	94.5	99.455	3486.88	2784491
768	**64**	**No**	**99.6**	**95.0**	**99.673**	**3775.45**	**2105067**
768	64	Yes	99.6	94.8	99.568	3722.90	2105067
768	128	No	99.5	94.5	99.515	5004.51	3484907
768	128	Yes	99.5	94.4	99.428	5010.37	3484907

After looking at the results, we clearly see that dropout does in general improve the results of the models. However, once we reach a certain amount of filters, dropout seems to not have such a positive impact, worsening the implementations for 768 filters. This last amount of filters is the highest we have tried, as due to memory limitation higher values are not possible to implement.

From all the different configurations tried, we have found that using 768 filters, embedding size 64 and dropout with p = 0.1 is the optimal. This model reaches a 95.0% of validation accuracy and becomes the best overall model of this whole study, improving the best LSTM configuration found earlier in all three validation accuracy, time execution and total parameters. In conclusion, the implementation of Convolutional Neural Networks has been found useful to highlight the efficacy of this kind of DNN in language recognition, besides from the traditional RNNs.

3.2 Other input features

With the aim of enhancing our best model, various input feature alternatives will be considered. Up to now, the model was fed input character by character. In this section, two alternatives will be tested: passing n-grams or passing character counts as input.

Note that these results are based on a model that was considered the best available during our study (unidirectional LSTM with dropout), though it ended up not being the best overall. Due to the constraints on computational resources, we have been unable to rerun all experiments with the latest bidirectional model configurations.

3.2.1 Character counts as input feature

Once the *Dictionary* class is created to map tokens into consecutive integer indexes, the construction of both the vocabulary and these indexes is carried out. The modified part of the code for this purpose, followed by the code snippet defining the *CharRNNClassifier* class, is presented below.

```
x_train_idx = []
for line in x_train_full:
    vector = [char_vocab.token2idx[c] for c in line]
    count_vector = np.bincount (vector, minlength=len(char_vocab)).astype('float32')
    x_train_idx.append(count_vector)
y_train_idx = np.array([lang_vocab.token2idx[lang] for lang in y_train_full])


class CharRNNClassifier(torch.nn.Module):

    def __init__(self, input_size, embedding_size, hidden_size, output_size, model="lstm", num_layers=1, bidirectional=Fal
se, pad_idx=0):
        super().__init__()
        self.model = model.lower()
        self.hidden_size = hidden_size

        #self.embed = torch.nn.Embedding(input_size, embedding_size, padding_idx=pad_idx)
        #if self.model == "gru":
            #self.rnn = torch.nn.GRU(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)
        #elif self.model == "lstm":
            #self.rnn = torch.nn.LSTM(embedding_size, hidden_size, num_layers, bidirectional=bidirectional)

        self.i2h = torch.nn.Linear(input_size, hidden_size)
        self.h2o = torch.nn.Linear(hidden_size, output_size)

    def forward(self, input, input_lengths):
        # T x B
        input = input.permute(1,0)
        output = self.i2h(input)
        # B x H
        output = self.h2o(output)
        # B x O
        return output
```

With this input configuration, a train accuracy of 88.4% and a validation accuracy of 85.7% are achieved in the cross-validation process, along with a train accuracy of 88.346% after 25 epochs of training. Without even considering the time it takes or the number of parameters it encompasses, this model worsens the metrics compared to those obtained previously. Therefore, the option of using character counts as input feature is discarded.

3.2.2 N-grams as input feature

Next, we will attempt to pass n-grams as input features to the model. Similar to the previous section, the vocabulary and indexes are constructed. Below, the modified code section will be displayed, where

the n-grams are constructed, with n being a parameter to determine, and subsequently, the variables *x_train_idx* and *y_train_idx* are built.

```
import nltk
for sentence in x_train_full:
    tokenized_sentence = list(sentence)
    bigrams = nltk.ngrams(tokenized_sentence,2)

    # Once we have the bigram, add it to the vocab.
    for bg in bigrams:
        char_vocab.add_token(''.join(bg))

n = 2

def build_ngrams(text):
    return list(nltk.ngrams(text,n))

x_train_ngrams = [build_ngrams(line) for line in x_train_full]
x_train_idx = [np.array([char_vocab.token2idx[''.join(pair)] for pair in ngram]) for ngram in x_train_ngrams]
y_train_idx = np.array([lang_vocab.token2idx[lang] for lang in y_train_full])
```

It has been tested for *n=2* and *n=3*, thus constructing bigrams and trigrams respectively. With this subdivision of words as inputs, we obtain the following results for the best RNN model from the mandatory tasks section:

	Cross-Validation		Train Accuracy (%)	Training time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Best RNN Model	99.7	94.2	99.580	5825.8	2818795
Bigrams (n=2)	99.9	94.6	99.936	6094.5	42631019
Trigrams (n=3)	100.0	94.0	99.959	6509.5	157531499

As shown in the table, using bigrams as inputs enhances the model's performance in terms of accuracy, achieving a validation accuracy of 94.6%. The training time, as well as the number of parameters, are increased in the construction of these, but it's a trade-off that, given the improvement, compensates. We conclude, therefore, that the best-performing model performs better when receiving bigrams as input compared to its original performance in this task.

Once it's confirmed that bigrams provide an improvement, we will also try passing them as input features to our best convolutional model, as seen in the previous section. The obtained result are the following:

	Cross-Validation		Train Accuracy (%)	Training time (s)	Parameters
	Train Accuracy (%)	Valid Accuracy (%)			
Best CNN Model	99.6	95.0	99.673	3775.5	2105067
Bigrams (n=2)	99.8	94.6	99.786	3864.3	22011179

In this case, for our convolutional layer-based model, bigrams are not the optimal choice for input features. As observed in the table, they decrease the validation accuracy while increasing the train accuracy. This indicates an increase in the model's overfitting. Furthermore, the number of parameters is significantly higher. Therefore, in the convolutional case, we will stick with the input feature used in the model from the previous section, the one originally provided in our task.