

# Deep Learning

## Laboratory 3

**Q1. Implement RNN and LSTM for the binary to decimal conversion problem in the ipython notebook : 3-1\_2-RNN\_LSTM\_Pytorch\_Question.ipynb (20 Points).**

In this first task we are asked to implement a Recurrent Neural Network and a Long Short-term memory Neural Network to study the sequences behind a Binary to Decimal conversion. We are given a dataset in two files: *binary.py* and *decimal.py*. The first one contains an array of arrays, each array represents a number in binary and is of size 8. The second one contains an array of arrays but each array contains the decimal conversion of each binary number in the *binary.py*.

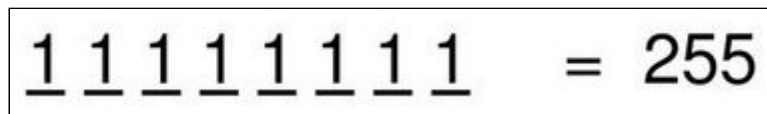

$$\underline{1} \underline{1} \underline{1} \underline{1} \underline{1} \underline{1} \underline{1} \underline{1} = 255$$

Figure 1

What we tried at first was using the standard RNN that PyTorch has to offer, but we were getting very big numbers of the Loss during the training. We were getting numbers close to 2000, and it converged in 2117.5. After trying to tweak the learning rate, trying to add momentum to the optimizer and trying to add weight\_decay (regularization), we were not able to lower the loss. So we decided to implement our own RNN class. An it ended up looking like can be seen in Figure 2.

```
class RNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNN, self).__init__()
        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

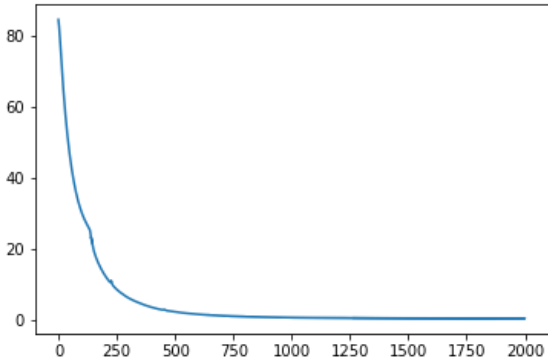
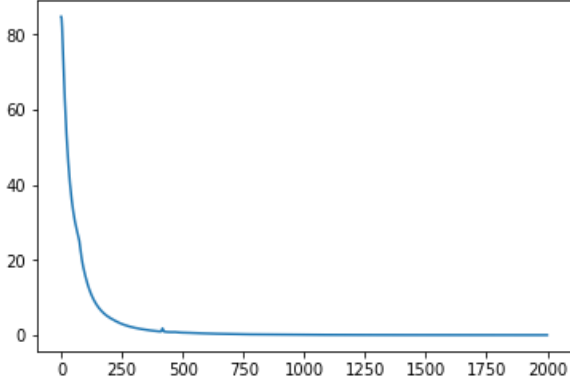
        # Number of hidden layers
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.ReLU = nn.ReLU()

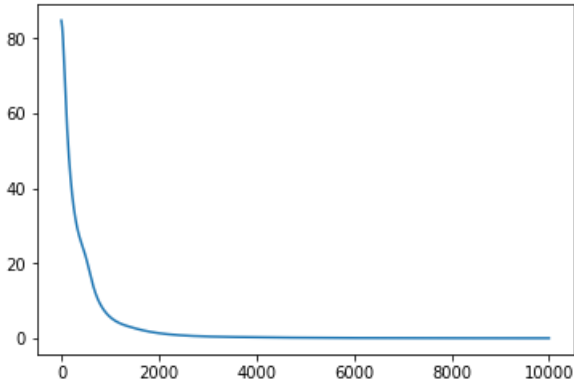
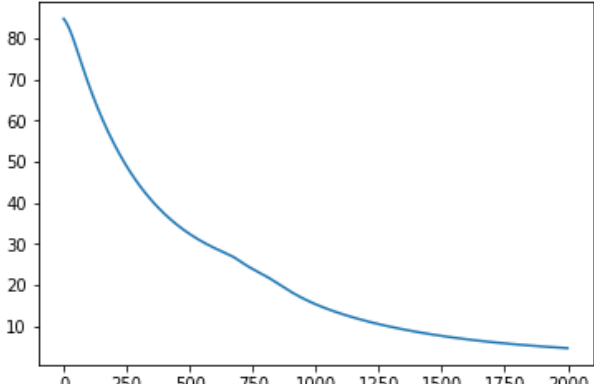
    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(1, x.size(0), self.hidden_dim)).to(device)

        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out)
        out = self.ReLU(out)
        return out
```

Figure 2

We decided to connect the RNN to a fully connected network and passing the output to a ReLU function, to change the range of the values. With this new RNN class and tweaking a bit the learning rate we were getting much better results. Then we decided to experiment with the size of the hidden layer to see how it impacted the results.

Learning Rate = 0.001	
Hidden size: 4	 <p>Final loss: 0.46295374631881714</p>
Hidden size: 8	 <p>Final loss: 0.03239273279905319</p>

Hidden size: 16	 <p>Final loss: <b>0.004384198691695929</b></p>
Learning Rate = 0.0001	
Hidden size: 8	 <p>Final loss: <b>4.5771074295043945</b></p>

As we can see in the table above what performs best is having a learning rate of 0.001 and a hidden layer size of 16. Finally to test how well the model works we decided to pass an input of 8 ones as binary, which in decimal is 255. The RNN returned the value of: **246.0029**.

Then we did the same process and experiment for the LSTM. We defined the LSTM as follows.

```
class LSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(LSTM, self).__init__()
        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

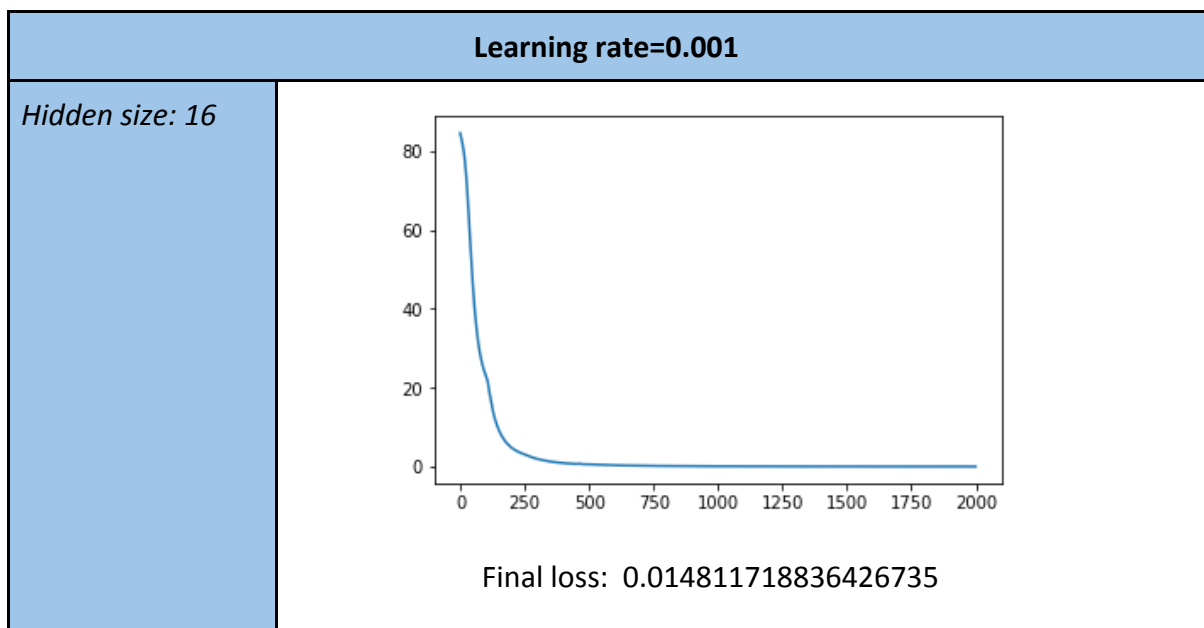
        # Number of hidden layers
        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.ReLU = nn.ReLU()

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(1, x.size(0), self.hidden_dim)).to(device)
        c0 = Variable(torch.zeros(1, x.size(0), self.hidden_dim)).to(device)

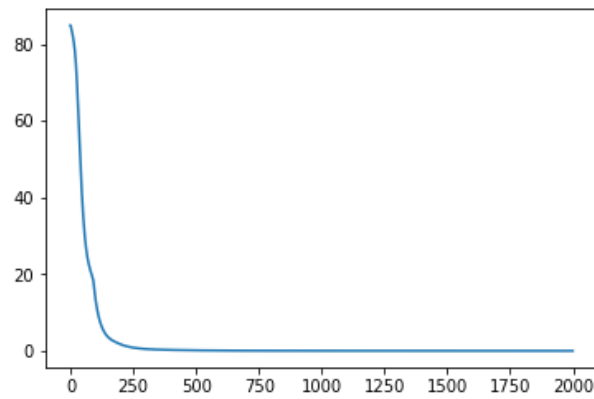
        # One time step
        out, hn = self.lstm(x, (h0, c0))
        out = self.fc(out)
        out = self.ReLU(out)
        return out
```

Figure 3

After having created the model of the LSTM we did the experiment to see which was the best learning rate and best hidden size for the network, as we previously did with RNN.

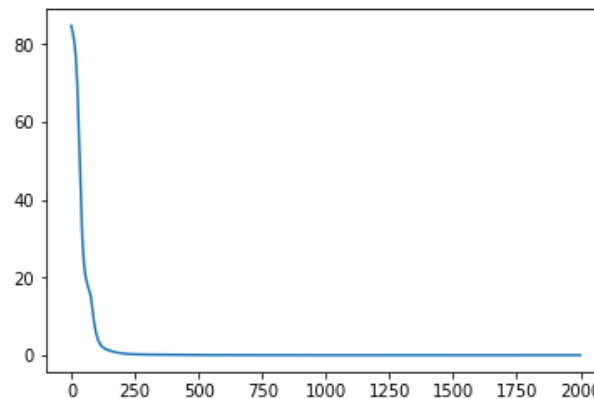


Hidden size: 32



Final loss: 0.0035577192902565002

Hidden size: 64



Final loss: **0.000834997626952827**

Regarding this experiments, we just tried with learning rate of 0.001 because we saw that the loss already converged, and as for the hidden size is concerned the best one is 64 with a loss of 0.000834997626952827. Finally, as we previously did with the RNN, to test how well the model works we decided to pass an input of 8 ones as binary, which in decimal is 255. The RNN returned the value of: **252.4994**.

**Q2. Implement the LSTM version of standard RNN on the ipython notebook: 3-2\_1-RNN-Overfit-Question.ipynb and analyse the performance differences (20 points).**

In this second task we were first asked to implement the LSTM equivalent of the given RNN network in the notebook. Our LSTM implementation ended up looking like this.

```
class SimpleLSTM(nn.Module):
    #!Task one, complete the model with internal three layer LSTM
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleLSTM, self).__init__()
        self.lstm1 = nn.LSTM(input_size, hidden_size)
        self.lstm2 = nn.LSTM(hidden_size, hidden_size)
        self.lstm3 = nn.LSTM(hidden_size, output_size)

    def forward(self, input, hidden=None):
        output, states = self.lstm1(input)
        output, states = self.lstm2(output)
        output, states = self.lstm3(output)
        return output, states

rnn = SimpleRNN(input_size = 1, hidden_size = 100, output_size = 1).to(device)
#!Task two, instantiate the lstm
lstm = SimpleLSTM(input_size = 1, hidden_size = 100, output_size = 1).to(device)
```

Figure 4

Then we had to create an optimizer for the LSTM, the Loss function and initialize the lists for the losses and the of both the LSTM and RNN. For our implementation see Figure 5.

```
optimizerRNN = torch.optim.SGD(rnn.parameters(), lr=0.9)
optimizerLSTM = torch.optim.SGD(rnn.parameters(), lr=0.9)

criterionRNN = nn.MSELoss()
criterionLSTM = nn.MSELoss()

# Hyper Par
number_or_epoch = 100

l_rnn, l_lstm = [], []
y_rnn, y_lstm = [], []
```

Figure 5

Finally, we had to implement training part for the LSTM as well as the addition of the loss values and the y values to their corresponding lists.

```
#!/Task three, perform the forward and optimization following RNN operation
#!/above for LSTM NN. Record the loss.
optimizerLSTM.zero_grad()
yHatAll,states = lstm(x)
outputLSTM = yHatAll[:,2,:]

lossLSTM = criterionLSTM(outputLSTM,y)
lossLSTM.backward()
optimizerLSTM.step()

l_lstm.append(lossLSTM.item())
y_lstm.append(y)
```

Figure 6

In the end, we obtain two different plots of both LSTM (orange) and RNN (blue) losses as we can see in Figure 7.

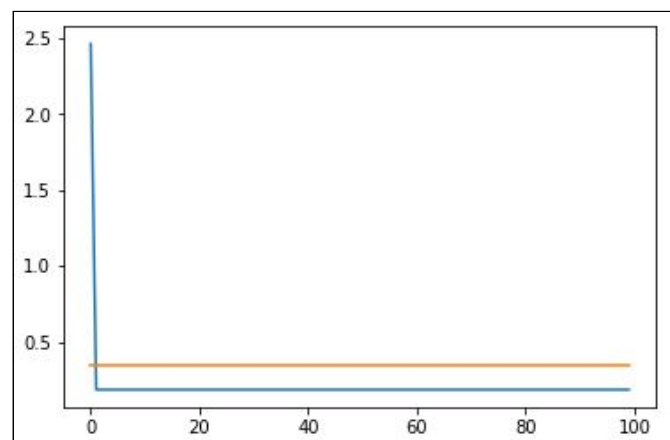


Figure 7

Just by looking to Figure 7 we can see that both LSTM and RNN converged to a number very close to 0 and if we take a look to the training trace printed we see that they do it quite fast: LSTM at the first epoch and RNN at the second epoch (see Figure 8). By looking at Figure 7 this loss could make us think that both Networks adapt very well to training data, and probably the model won't generalize too well. As we have seen in class, it is worth to highlight that the LSTM converged faster than the RNN, as can be seen in Figure 8, even though it converged to a quite higher loss compared to the one of the RNN.

```
0 lossRNN : 2.461430788040161 lossLSTM : 0.3476763665676117
1 lossRNN : 0.18518516421318054 lossLSTM : 0.3476763665676117
```

Figure 8

**Q3. Load the Sales Store CSV data (try using the mean data first), then implement your RNN to predict the immediate store value given series of past store data in the ipython notebook : 3-1\_3-TimeSeriesRNNAnswer.ipynb (30 Points).**

For the third question, we were asked to implement a RNN to try to predict time series data. In particular we were asked to predict Sales of a Store given a series of past data of the named Store.

First we loaded the data from the CSV and decided to plot it to see more or less the graph it describes (see Figure 9).

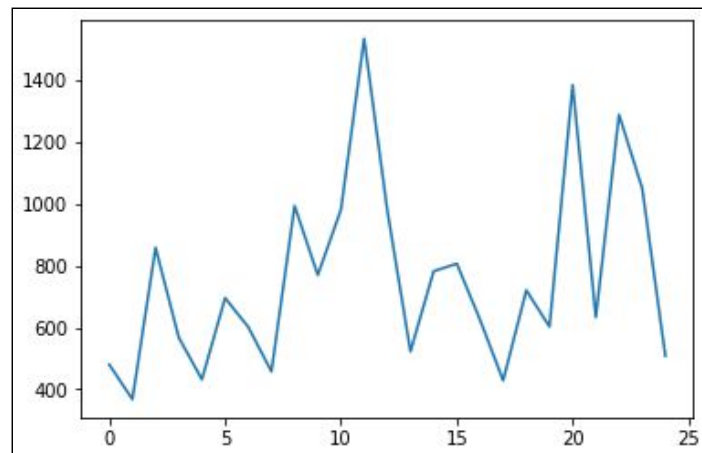


Figure 9

After that we loaded the data to tensors and to the GPU to be able to work with it. Then we defined both LSTM and RNN implementations to be able to use both and compare which fits best to the data. Our implementations of LSTM and RNN can be seen in the following Figures 10 and 11.

```
class LSTM_FCN(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(LSTM_FCN, self).__init__()
        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.layer_dim = layer_dim

        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.ReLU = nn.ReLU()

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_dim)).to(device)
        c0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_dim)).to(device)

        # One time step
        out, hn = self.lstm(x, (h0, c0))
        out = self.fc(out)
        out = self.ReLU(out)
        return out
```

Figure 10



```
class RNN_FCN(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNN_FCN, self).__init__()
        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.layer_dim = layer_dim

        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.ReLU = nn.ReLU()

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_dim)).to(device)

        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out)
        out = self.ReLU(out)
        return out
```

Figure 11

Following those definitions and after trying out some RNN and some LSTM models we decided to use an LSTM model with the following sizes: 32 for the input, 256 for one hidden layer and one as output. Also we used the Mean Squared Error loss as loss function, a learning rate of 0.01 and a Stochastic Gradient Descent as optimizer method.

The most important of the past values is the input size. At first, we had the model working but it was not able to learn. We tried to change the learning rate, the size of the hidden dimensions and even the number of recurrent blocks but it was still not working. We then realized that the problem was that the amount of information that we were giving to the model at each step was too low (we were trying to train the model with an input size of 8). We observed improvements with 16 and with 32 the results were good enough.

```
loss_list = []
samples_per_step = input_dim
for epoch in range(num_epoch):
    for i in range(0, data.shape[-1]):
        # Clear gradients
        optimizer.zero_grad()

        if i+samples_per_step+1 >= data.shape[-1]:
            continue

        x = data[:, :, i:i+samples_per_step]
        y = data[:, :, i+samples_per_step+1]

        # Forward propagation
        outputs = model(x)
        #print("INPUT", x)
        #print(y)
        #print(outputs)

        # Calculate softmax and MSE loss
        loss = error(outputs, y)

        # Calculating gradients
        loss.backward()

        loss_list.append(loss.item())

        # Update parameters
        optimizer.step()

    print(f'loss : {loss.item()}')
```

In the left picture we can see the resulting code of our training process.

Finally is time to evaluate the model and try to predict the Sales of the Store. So when predicting the future Sales we get the following graph.

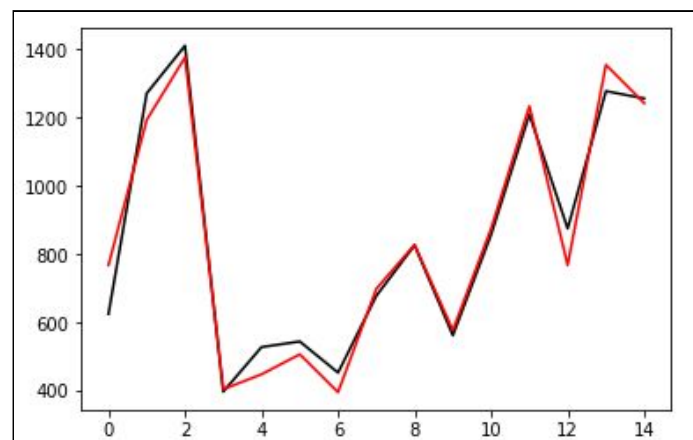


Figure 12

**Q4. Convert the available RNN model or create RNN model to use standard pytorch RNN libraries. Also extend with using LSTM version of the network to see the impact in a character modelling names classification by origin: 3-1\_4-CharacterRNNQuestion.ipynb (30 Points).**

In this last exercise we tested three models, a RNN implemented with Linear layers, the pytorch RNN implementation and the pytorch LSTM implementation. You can reproduce our results uncommenting each one of this models in the cell right after the definition of our last model:

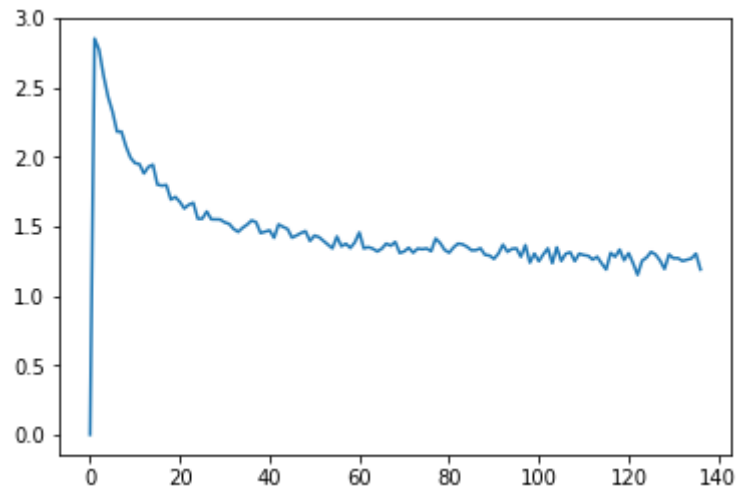
```
n_hidden = 256

# model = RNN(n_letters, n_hidden, n_categories).to(device)
# model = RNN_FC(n_letters, n_hidden, n_categories).to(device)
model = LSTM_FC(n_letters, n_hidden, n_categories).to(device)
```

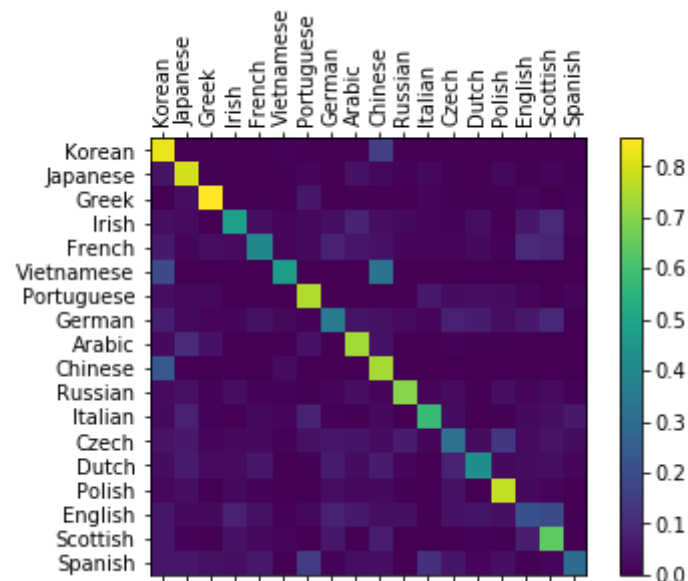
To start we decided to implement our own RNN with linear layers and see how it was performing. At first, using a hidden size of 128 we tried different learning rates. We saw how we were getting NaNs in the output if it was too big (0.01 for example) and that it was not changing at all if we used too small numbers (like 0.0001). At the end we found optimal the value of 0.8. Then we saw that although it seemed to learn something, it needed more parameters to understand such patterns, so we decided to use a hidden dimension of 256. With this parameters we got, surprisingly, pretty nice results. First we show how our training outputs and our loss curve:

```
from IPython.display import clear_output as clear

0 0% (0m 0s) 2.9400 Ungaretti / French x (Italian)
5000 2% (0m 12s) 2.5932 Shula / Japanese x (Czech)
10000 5% (0m 24s) 1.7576 Melo / Portuguese ✓
15000 7% (0m 36s) 1.3672 Ma / Korean x (Vietnamese)
20000 10% (0m 47s) 0.1954 Shon / Korean ✓
25000 12% (0m 58s) 0.8103 Ferreiro / Portuguese ✓
30000 15% (1m 9s) 0.2708 Ponikarovskiy / Russian ✓
35000 17% (1m 20s) 0.9721 Kosmas / Greek ✓
40000 20% (1m 31s) 0.6709 Jeon / Korean ✓
45000 22% (1m 42s) 1.0165 Cabello / Italian x (Spanish)
50000 25% (1m 54s) 2.5679 Pritchard / Scottish x (English)
55000 27% (2m 6s) 3.2080 Klineberg / German x (Czech)
60000 30% (2m 17s) 2.5208 Chantler / French x (English)
65000 32% (2m 28s) 0.7057 Rousses / Greek ✓
70000 35% (2m 39s) 0.0263 Karahalios / Greek ✓
75000 37% (2m 50s) 0.9613 Maria / Portuguese ✓
80000 40% (3m 2s) 1.3777 Pak / Korean ✓
85000 42% (3m 14s) 0.9320 Yun / Chinese x (Korean)
90000 45% (3m 25s) 1.8735 Jurbin / English x (Russian)
95000 47% (3m 37s) 0.3032 Tagawa / Japanese ✓
100000 50% (3m 48s) 1.0578 Shen / Korean x (Chinese)
105000 52% (3m 59s) 0.4494 Sobol / Polish ✓
110000 55% (4m 10s) 0.4178 Thian / Chinese ✓
115000 57% (4m 21s) 0.8028 Kijek / Polish ✓
120000 60% (4m 33s) 0.1586 Sunada / Japanese ✓
125000 62% (4m 44s) 2.1236 Seviens / Spanish x (Dutch)
130000 65% (4m 55s) 1.7511 Navratil / French x (Czech)
135000 67% (5m 6s) 2.6161 Belmonte / French x (Spanish)
```



As we can see it doesn't converge to 0 or a value too low, but it seems to perform well in practice. We can confirm this performance looking at the next confusion matrix:



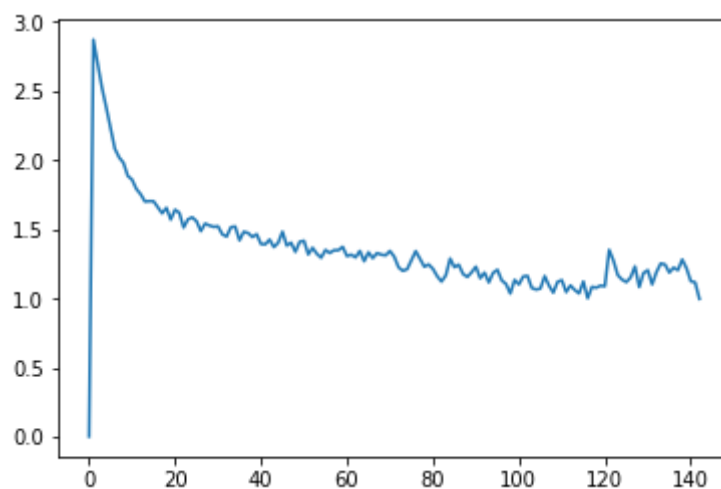
Then, we did the same experiment with the RNN class given by pytorch. In this case we used the same parameters as before and it seemed to train well with them. First, we show how our training outputs and our loss curve:

Víctor Pérez 183950

Diego Vincent 174458

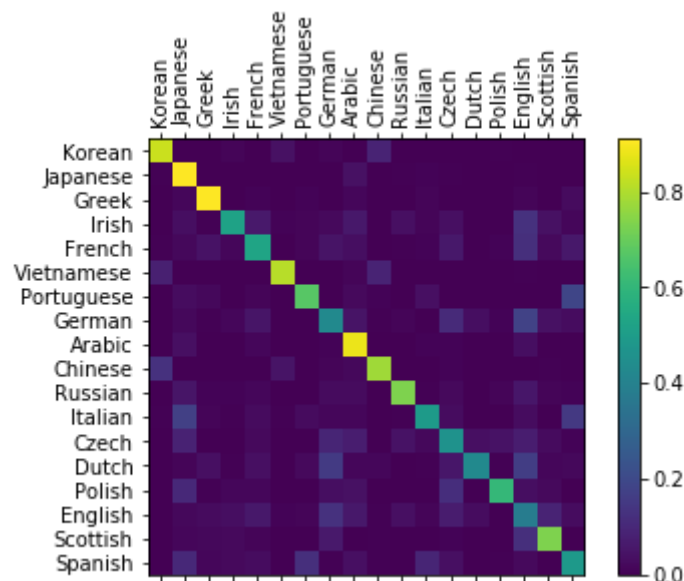
Óscar Font 183826

```
0 0% (0m 0s) 2.8321 Thuy / Japanese x (Vietnamese)
5000 2% (0m 23s) 1.8451 Salvatici / Polish x (Italian)
10000 5% (0m 45s) 1.8899 Pape / English x (French)
15000 7% (1m 7s) 1.2706 Okura / Japanese ✓
20000 10% (1m 29s) 1.1075 Gil / Korean ✓
25000 12% (1m 52s) 1.4717 Otake / Japanese ✓
30000 15% (2m 15s) 0.0969 Shamoun / Arabic ✓
35000 17% (2m 37s) 0.9122 Cui / Chinese ✓
40000 20% (3m 0s) 0.6116 Rorris / Greek ✓
45000 22% (3m 22s) 1.8131 Toru / Vietnamese x (Japanese)
50000 25% (3m 44s) 0.0687 Ziemniak / Polish ✓
55000 27% (4m 6s) 1.8028 Taka / Polish x (Japanese)
60000 30% (4m 29s) 2.3583 Merckx / Russian x (Dutch)
65000 32% (4m 51s) 4.5070 Okenfuss / Greek x (Czech)
70000 35% (5m 13s) 0.3248 Issa / Arabic ✓
75000 37% (5m 36s) 2.6357 Arendonk / Czech x (Dutch)
80000 40% (5m 58s) 2.9725 Han / Chinese x (Korean)
85000 42% (6m 20s) 1.3078 Perreault / Russian x (French)
90000 45% (6m 42s) 1.5847 Yeo / Chinese x (Korean)
95000 47% (7m 5s) 0.4558 Donnell / Irish ✓
100000 50% (7m 27s) 1.9030 Laberenz / Italian x (German)
105000 52% (7m 49s) 0.2109 Shu / Chinese ✓
110000 55% (8m 12s) 0.0262 Rios / Portuguese ✓
115000 57% (8m 34s) 0.3621 Escarcega / Spanish ✓
120000 60% (8m 56s) 0.1655 Lauwers / Dutch ✓
125000 62% (9m 18s) 0.9344 Deeb / Dutch x (Arabic)
130000 65% (9m 43s) 0.1781 Jeong / Korean ✓
135000 67% (10m 5s) 0.0519 Mitchell / Scottish ✓
```



What we can extract from this images is first, how the training time increases considerably with this model. We can clearly see that is not converged after 10 min of training while the Linear model converged in 5. The results are better though, we can see how the performance is better than the last model and how the error is smaller. We can confirm this in the following confusion matrix, where we can see more yellow colors in the diagonal:

Víctor Pérez 183950  
 Diego Vincent 174458  
 Óscar Font 183826



When we started testing the LSTM model we saw that with these same parameters was not able to learn as we can see in the following image:

```
0 0% (0m 0s) 2.8496 Kada / Irish x (Japanese)
5000 2% (0m 26s) 2.9216 Opp / Dutch x (Czech)
10000 5% (0m 52s) 2.8691 Ryu / Dutch x (Korean)
15000 7% (1m 18s) 2.8256 Offermans / Dutch x
20000 10% (1m 44s) 2.9602 Bellerose / Russian x (French)
25000 12% (2m 8s) 2.8765 Espino / Irish x (Spanish)
30000 15% (2m 33s) 2.8425 Doyle / Russian x (Irish)
35000 17% (2m 59s) 2.8794 Solo / Irish x (Spanish)
40000 20% (3m 25s) 2.9393 Zdunowski / Irish x (Polish)
45000 22% (3m 49s) 2.8334 Cathain / Russian x (Irish)
50000 25% (4m 15s) 2.8265 Cassidy / Dutch x (Irish)
55000 27% (4m 40s) 2.8924 Truong / Irish x (Vietnamese)
60000 30% (5m 5s) 2.8869 Chu / Dutch x (Vietnamese)
```

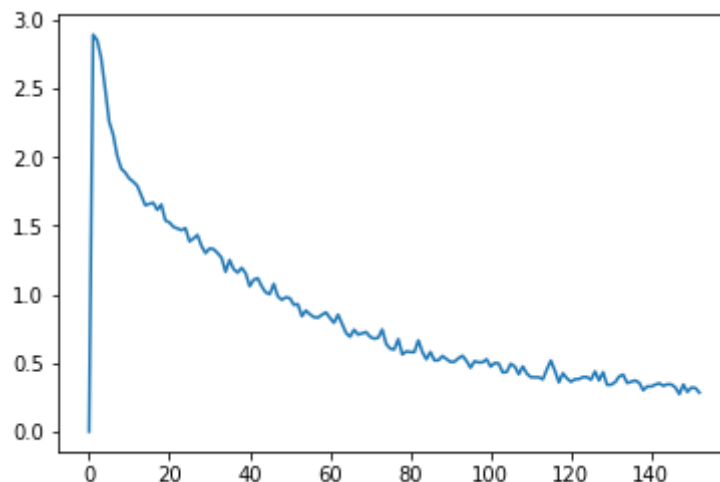
As the values were too static we decided to increase the learning rate to 0.05 and the results were amazing. Here we show our training outputs and our loss curve:

Víctor Pérez 183950

Diego Vincent 174458

Óscar Font 183826

```
0 0% (0m 0s) 2.9511 Taverna / Spanish x (Italian)
5000 2% (0m 25s) 2.8022 Hase / Arabic x (German)
10000 5% (0m 51s) 0.9978 Ta / Vietnamese ✓
15000 7% (1m 16s) 1.8844 Kiefer / Dutch x (German)
20000 10% (1m 41s) 0.0408 Nervetti / Italian ✓
25000 12% (2m 7s) 2.6655 Murata / Spanish x (Japanese)
30000 15% (2m 31s) 0.6975 Baudin / French ✓
35000 17% (2m 56s) 0.6109 Shan / Chinese ✓
40000 20% (3m 21s) 4.8772 Meier / German x (Czech)
45000 22% (3m 47s) 0.0028 Shibasawa / Japanese ✓
50000 25% (4m 11s) 1.2620 Kennedy / English x (Scottish)
55000 27% (4m 36s) 5.2026 Nagel / Polish x (Dutch)
60000 30% (5m 3s) 0.2944 Dubhan / Irish ✓
65000 32% (5m 28s) 1.7520 Pitterman / Russian x (Czech)
70000 35% (5m 53s) 0.4965 Sitta / Czech ✓
75000 37% (6m 19s) 0.0132 Cheng / Chinese ✓
80000 40% (6m 44s) 1.6784 Solos / Polish x (Spanish)
85000 42% (7m 9s) 1.6462 Gorman / English x (Irish)
90000 45% (7m 36s) 1.3969 Tang / Vietnamese x (Chinese)
95000 47% (8m 1s) 0.0440 Doan / Vietnamese ✓
100000 50% (8m 26s) 0.0265 Agrikoff / Russian ✓
105000 52% (8m 52s) 0.0543 Ridley / English ✓
110000 55% (9m 16s) 0.0252 Loong / Chinese ✓
115000 57% (9m 42s) 0.0635 Krol / Polish ✓
120000 60% (10m 9s) 1.0407 Mochan / Irish ✓
125000 62% (10m 34s) 0.0148 Haik / Arabic ✓
130000 65% (10m 59s) 0.0006 Meeuwessen / Dutch ✓
135000 67% (11m 25s) 0.0293 Boulous / Arabic ✓
140000 70% (11m 50s) 0.0020 Yu / Korean ✓
145000 72% (12m 16s) 0.0517 Dickson / Scottish ✓
150000 75% (12m 41s) 0.0421 Ferro / Portuguese ✓
```



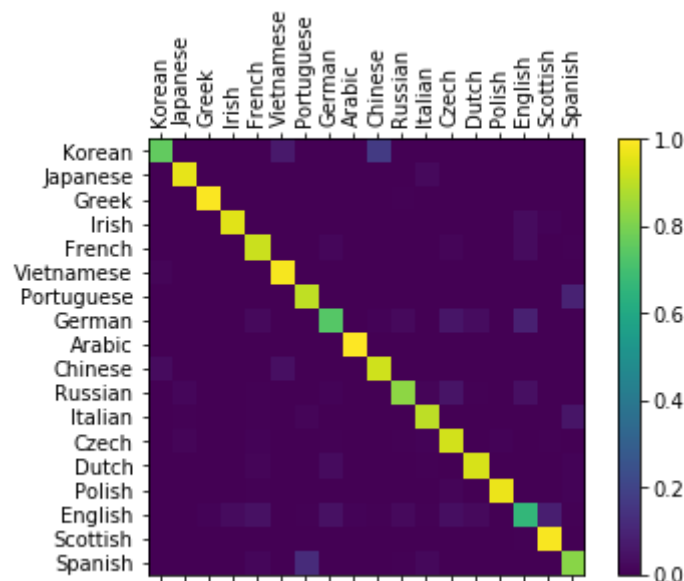
With the last images we can see how now the error is way more smaller than with the previous models. We can also see that is still not converged after 12 mins, so it could be even better! Here, we show the resulting confusion matrix:



Víctor Pérez 183950

Diego Vincent 174458

Óscar Font 183826



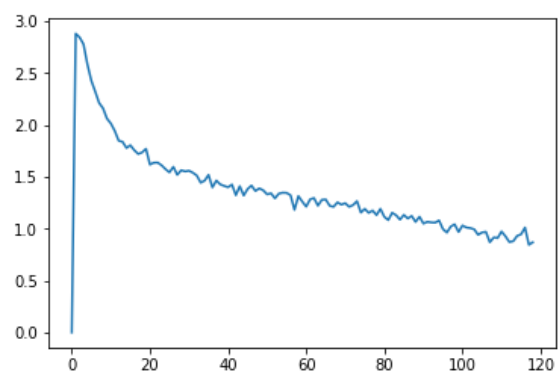
As we can see almost all the values are in the diagonal, this confirms the good performance of this model.

To finish, we wanted to give a last chance to the RNN model with these last parameters to perform a fair comparison between this models but the gradients exploded:

```
0 0% (0m 0s) 2.9471 Papageorge / Japanese x (Greek)
5000 2% (0m 22s) 2.3854 Ma / Chinese x (Korean)
10000 5% (0m 46s) 0.0038 Sokolof / Polish ✓
15000 7% (1m 8s) 6.0797 Pereira / Arabic x (Portuguese)
20000 10% (1m 30s) 6.2360 Mcmillan / Polish x (Scottish)
25000 12% (1m 53s) 13.8954 Igarashi / Portuguese x (Japanese)
30000 15% (2m 17s) 17.6203 Bukowski / English x (Polish)
```

It didn't work neither with a learning rate of 0.01, it was so unstable, but when we tried with 0.005 we got the following results:

```
0 0% (0m 0s) 2.8592 Turchi / English x (Italian)
5000 2% (0m 22s) 2.2828 Jebson / Irish x (English)
10000 5% (0m 45s) 2.6464 Bonner / German x (French)
15000 7% (1m 7s) 3.3102 Dziedzic / Russian x (Polish)
20000 10% (1m 30s) 0.7587 Chau / Vietnamese ✓
25000 12% (1m 54s) 3.4357 Aitken / Dutch x (Scottish)
30000 15% (2m 16s) 0.9517 Davlertgareev / Russian ✓
35000 17% (2m 38s) 1.2229 Caiazza / Portuguese x (Italian)
40000 20% (3m 0s) 1.6354 Duchamps / English x (French)
45000 22% (3m 24s) 1.2653 Ron / Korean ✓
50000 25% (3m 46s) 0.5497 Pena / Spanish ✓
55000 27% (4m 8s) 0.8065 Barros / Portuguese ✓
60000 30% (4m 31s) 0.4016 Pereira / Portuguese ✓
65000 32% (4m 53s) 0.0204 Winogrodzki / Polish ✓
70000 35% (5m 17s) 0.0077 Emohonov / Russian ✓
75000 37% (5m 39s) 0.6543 Almeida / Portuguese ✓
80000 40% (6m 3s) 1.5160 Sarkis / Greek x (Arabic)
85000 42% (6m 25s) 0.1363 Wojda / Polish ✓
90000 45% (6m 47s) 0.3880 Mackay / Scottish ✓
95000 47% (7m 11s) 0.1107 Shim / Korean ✓
100000 50% (7m 33s) 0.0864 Szweda / Polish ✓
105000 52% (7m 56s) 4.3882 Brooker / Dutch x (English)
110000 55% (8m 18s) 0.1980 Ueshima / Japanese ✓
115000 57% (8m 41s) 0.0736 McIntosh / Scottish ✓
```





*Víctor Pérez* 183950

*Diego Vincent* 174458

*Óscar Font* 183826

As we can see, now is learning so much better and getting close to the results that we obtained with the LSTM.

So our conclusion is that LSTM, in this case, outperforms the other models and it's interesting how each one performs better with different learning rates, it seems like the more parameters that a model has, the larger learning rate we need to use (in this case).