David Pérez Carrasco (NIA: 241375)

Guillem Escriba Molto (NIA: 242123)

Arnau Solans Vázquez (NIA: 216530)

<u>Report P1</u>

Ex1:

At the beginning of the lab, we were expected to load and visualize the training and testing data. This was already done in the given code cell with the test data, but we also did it with the training data, showing how it is distributed. As we see in both the training and testing visualizations, we have 6 classes of the 210 training sequences and 90 testing sequences of 40 dimensions each one.

Once we started with the implementation of the first exercise, we basically needed to use the example code cells as a baseline which we needed to adapt for our actual purpose. The example code was designed for a binary classification, while the lab code is about a multiclass classification, which means that we needed to adapt the whole process of the classification, from the model architecture to the training and testing.

First, we know from the theory that the loss functions are quite different for binary classification and for multiclass classification. For binary classification, the Binary Cross Entropy (BCE) loss function is appropriate, while for multiclass the cross entropy loss is required.

This is why BCE measures the dissimilarity between predicted probabilities and the true labels, encouraging the model to output high probabilities for the correct class and low for the incorrect, and this is well-suited for binary classification where there are only 2 classes to distinguish.

However, when it comes to Cross Entropy loss, it calculates the average dissimilarities between predicted class probabilities and the real labels, considering the entire probability distribution over all classes and allowing the model to learn distinctions and make decisions over multiple classes, so it is suitable for multiclass classification, since it takes into account the relative probabilities of all classes. Having said all that, we just needed to change the loss function from the BCE to the CrossEntropy function to adapt the loss function.

When it comes to the SequenceClassifier model, we required several small changes that completely changed the orientation of the model from binary to multiclass classification. First, in the last linear we needed to put n_classes instead of 1 as input of nn.Linear due to a dimensionality aspect. In binary classification, we only need the output in 1 dimension, since we are only considering two exclusive classes, but in multiclass we need the output to correspond with the number of classes our dataset contains, otherwise we would not be able to distinguish among all the possible classes. The nn:linear layer with hidden_size and n_classes as output performs a linear transformation, mapping the learned features from the hidden layers to the corresponding logits for each class, which will be used with the CrossEntropyLoss to compute the loss and perform backpropagation to train the model. If we correctly set n_classes, the model will be able to properly learn and make predictions for all the possible classes.

Another important change in the SequenceClassifier was the activation function. In binary classification, we apply the sigmoid activation function to obtain a probability estimate for the positive class, mapping any real value to a probability value, which is the best way to classify binary variables, since the variables with probabilities closer to 1 are more likely to be from class 1 (or the positive class). However, for multiclass classification we do not have only 2 classes, we have more, hence classifying the variables by the closer or further probabilities from 1 would not work here. But since we are already using the CrossEntropyloss function, we don't need any activation function, since the Cross-Entropy Loss function handles the normalization of class probabilities using the softmax function, which ensures that the sum of all the probabilities sum up to 1.

An important note here is that we have also tried to use the softmax function with the output in the SequenceClassifier, thinking that it may help normalizing the probabilities and getting more accurate predictions. The truth is that it indeed did not work bad, getting test accuracy values close to those models not using the softmax for the output, but it worked slightly better without it, due to the fact that we just commented about the softmax being already applied inside the loss function, so we left the sequence classifier without the softmax and with the 2 changes commented.

Having the model correctly implemented, we had to focus on the training and testing. The training was quite easy, since no changes were required, despite the ones regarding the dimensionality due to the new loss function. However, the testing was the trickiest part.

In Binary classification the test works as already commented before, assigning the higher values than a probability threshold to a class and the others to another. However, with multiple classes it cannot work like this. Another important reason why the test algorithm needed to be changed is the classifier output. Since we have changed the output dimensionality (before it was 1, now it's n_classes), we need to change the test algorithm to make it work with all the classes probabilities. In order to fix all this, we needed to use something that takes the bigger probability, instead of just a threshold, so we used argmax. Torch.argmax function is used to determine the predicted label for each sample by selecting the class index with the highest probability from the output tensor, so it returned the exact class we wanted to assign, since it had the highest probability. Now, we just needed to compute the correct predictions, with the comparison

```
correct_predictions = (predicted_labels == Y_test).sum().item()
```

and then divide it by the total number of predictions to obtain the test accuracy.

In order to visualize the confusion matrix, we could do it in several ways, from just printing it in the terminal to visualizing with a heat map from sns library or using sklearnmetrics to visualize. We could also include that visualization inside the same testing function or outside, but we decided to let it inside. It was a bit annoying if we visualized each time we tested a model when looking for the best hyperparameters values, but since there is no need to iterate through any model from now, only execute the chosen model, it will work to include that code inside the same function and visualize the confusion matrix once the accuracy has been computed and printed. We eventually decided to use sklearn.metrics to directly use the confusion_matrix function and then we visualize it using matplotlib.pyplot.

In order to explain the visualization we first must recall what a confusion matrix is. The confusion matrix is a table that allows us to visualize the performance of the classification model, by displaying the counts of true positive (the correctly classified samples), true negative (the correctly discarded samples for that class), false positive (the incorrectly classified samples for that class) and false negative (the incorrectly discarded samples that are actually from that cñass) predictions for each class.

Therefore, we use a visualization in form as a heatmap, where each cell corresponds to a combination of an actual class (y-axis) and a predicted class (x-axis). The color shows the proportion of samples belonging to a particular class, in a way that darker colors indicate higher values. In this sense, the most optimal confusion matrix would be the one with the dark colors in the diagonal and totally light colors in the other cells (representing 0 misclassified samples). We considered this visualization the most accurate and profitable since it allows us to quickly identify patterns and we can easily gain insights into the strengths and weaknesses of our model's predictions and performance.

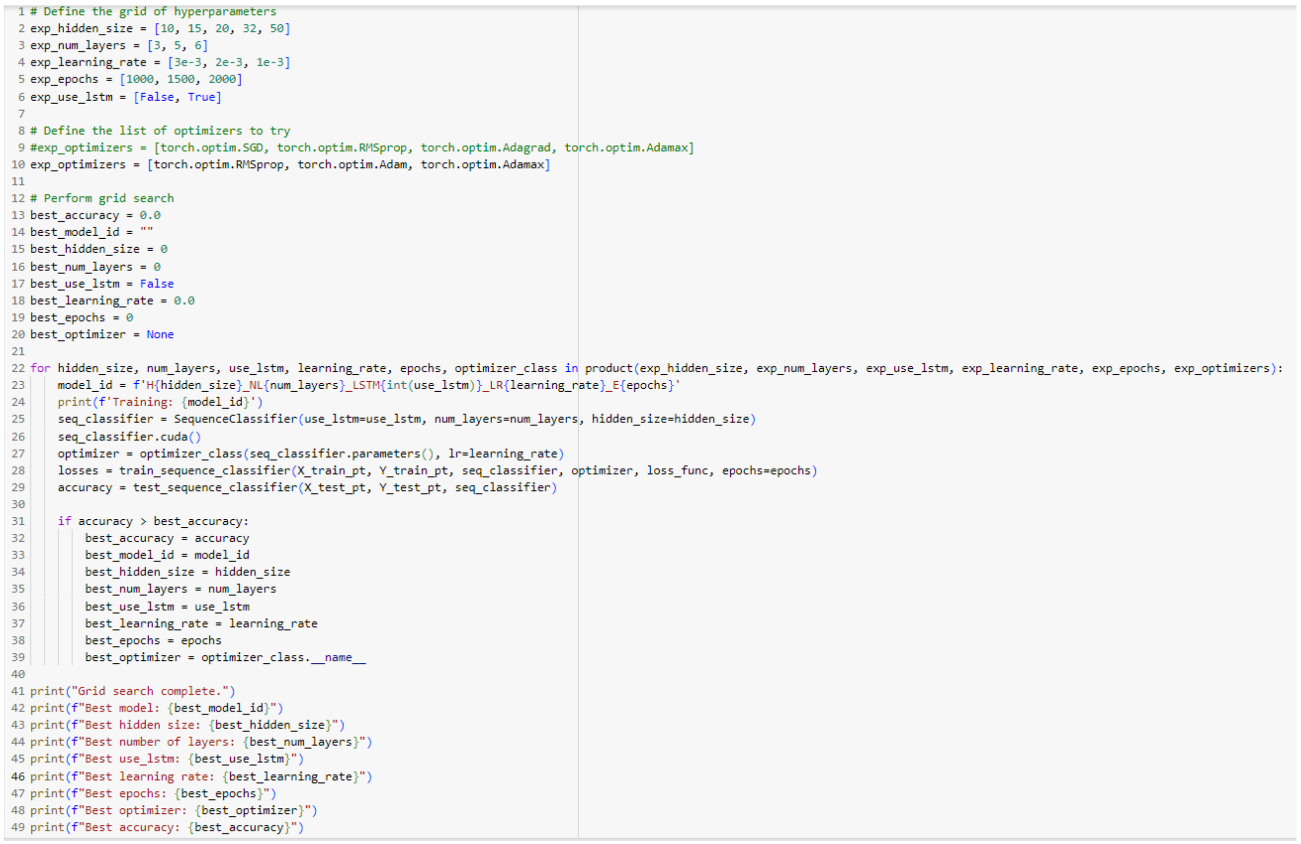
A final change that we had to make was about the pytorch variables dimensions, since we needed to delete the "squeeze()" call for the Y variables when transforming the dataset to PyTorch format, since now we did not want to remove dimensions from them. This is because in binary classification, the target variable usually has a shape of [batch_size, 1], so we remove that 1 to only work with [batch_size] which allows for direct coimparison with the target labels.

However, for multiclass classification, the target variable is typically represented as a one-hot encoded matrix or tensor of shape [batch_size, num_classes], as we saw in the lectures, so the predicted output needs to be of the same size, hence we do not need to remove any dimension as there are no singleton dimensions now.

Once we have completely adapted the implementation for the multiclass classification, we had to do the longest and trickiest part, to experiment with the hyper-parameters. In order to do that we performed different rounds. The first one was a massive iteration in which we computed the loss and the test accuracy of the models for any reasonable value for any hyper-parameter. We worked with the hidden_size values of 10,15,20,32 and 50 (we also considered other values such as 5, 7, 25 and 40 but the computational cost was too high and there were no relevant differences), with num_layers values of 3,5 and 6 (though again, we massively tried several values, from 1 to 15, but the most optimal interval was between 3 and 6), with learning rate values of 0.003, 0.002 and 0.001 (again, we tried from 0.1 to 0.0001, but the most optimal interval was between the 3 final values), with epoch values of 1000, 1500 and 200 (also tried 500 and 2500), though this was a controversial point, since we realized that it added no relevant information to the hyper-parameter grid, since we must fix the epoch value and then try the best other hyper-parameter values for that epoch value, and that's exactly what we finally did with epoch=2000, since the other also returned good models, but for the most of the models it was too quick to reach the maximum accuracy,. We also tried using LST and not using it, though we quickly discovered that the LST worked pretty better than the RNN, so we discarded the RNNs to avoid higher computational cost.

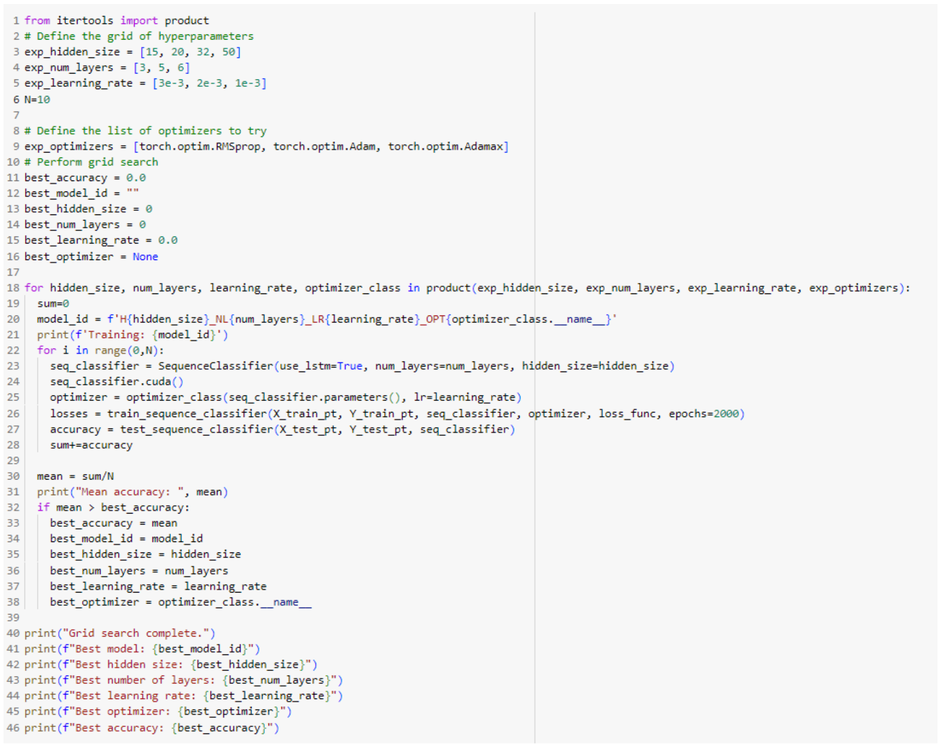
Finally we also tried different optimizers, RMSprop, Adam, Adamax and Adagrad. The last one resulted in working slightly worse than the others so we discarded it and the first two were generally the best, especially the RMSprop, but we tried with the three of them anyway. Finally an important consideration here is that we did not use a for loop with zip as in the examples, we used a grid search through itertools.product iterative function, in order to test any possible combination between the contemplated values. It was obviously so far away in computational cost, but we wanted to make sure of the best possible models to select them and then reduce the grid search. Since there is no need of executing that code anymore we are providing it through the following screenshot and fully commenting it in the notebook.

Code from the first grid search of hyper-parameters values:



```python
# Define the grid of hyperparameters
exp_hidden_size = [10, 15, 20, 32, 50]
exp_num_layers = [3, 5, 6]
exp_learning_rate = [3e-3, 2e-3, 1e-3]
exp_epochs = [1000, 1500, 2000]
exp_use_lstm = [False, True]

# Define the list of optimizers to try
#exp_optimizers = [torch.optim.SGD, torch.optim.RMSprop, torch.optim.Adagrad, torch.optim.Adamax]
exp_optimizers = [torch.optim.RMSprop, torch.optim.Adam, torch.optim.Adamax]

# Perform grid search
best_accuracy = 0.0
best_model_id = ""
best_hidden_size = 0
best_num_layers = 0
best_use_lstm = False
best_learning_rate = 0.0
best_epochs = 0
best_optimizer = None

for hidden_size, num_layers, use_lstm, learning_rate, epochs, optimizer_class in product(exp_hidden_size, exp_num_layers, exp_use_lstm, exp_learning_rate, exp_epochs, exp_optimizers):
    model_id = f'H{hidden_size}_NL{num_layers}_LSTM{int(use_lstm)}_LR{learning_rate}_E{epochs}'
    print(f'Training: {model_id}')
    seq_classifier = SequenceClassifier(use_lstm=use_lstm, num_layers=num_layers, hidden_size=hidden_size)
    seq_classifier.cuda()
    optimizer = optimizer_class(seq_classifier.parameters(), lr=learning_rate)
    losses = train_sequence_classifier(X_train_pt, Y_train_pt, seq_classifier, optimizer, loss_func, epochs=epochs)
    accuracy = test_sequence_classifier(X_test_pt, Y_test_pt, seq_classifier)

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_model_id = model_id
        best_hidden_size = hidden_size
        best_num_layers = num_layers
        best_use_lstm = use_lstm
        best_learning_rate = learning_rate
        best_epochs = epochs
        best_optimizer = optimizer_class.__name__

print("Grid search complete.")
print(f"Best model: {best_model_id}")
print(f"Best hidden size: {best_hidden_size}")
print(f"Best number of layers: {best_num_layers}")
print(f"Best use_lstm: {best_use_lstm}")
print(f"Best learning rate: {best_learning_rate}")
print(f"Best epochs: {best_epochs}")
print(f"Best optimizer: {best_optimizer}")
print(f"Best accuracy: {best_accuracy}")
```

This massive search grid lasted for 2 hours and a half, but gave us really good insights about the performance of our models. We took notes from the best models and about the values that worked better in general for each hyper-parameter, but we also noticed that it was still too imprecise since at each execution the results may vary. For instance, we could get 0.96 of accuracy with a hidden size of 15, 6 layers and 0.003 of learning rate with RMSprop optimizer, but when using this model again we got values from 0.8 to 0.96, which clearly showed us that the random initialization plays an important role here. Due to this, we implemented a second round, where we directly selected epoch=2000 and used a LSTM, as well as reducing the hyper-parameter values, with the only difference that now we executed each model up to 10 times and computed the mean, in a way that we could know the real and accurate performance of each model. The code was the following:



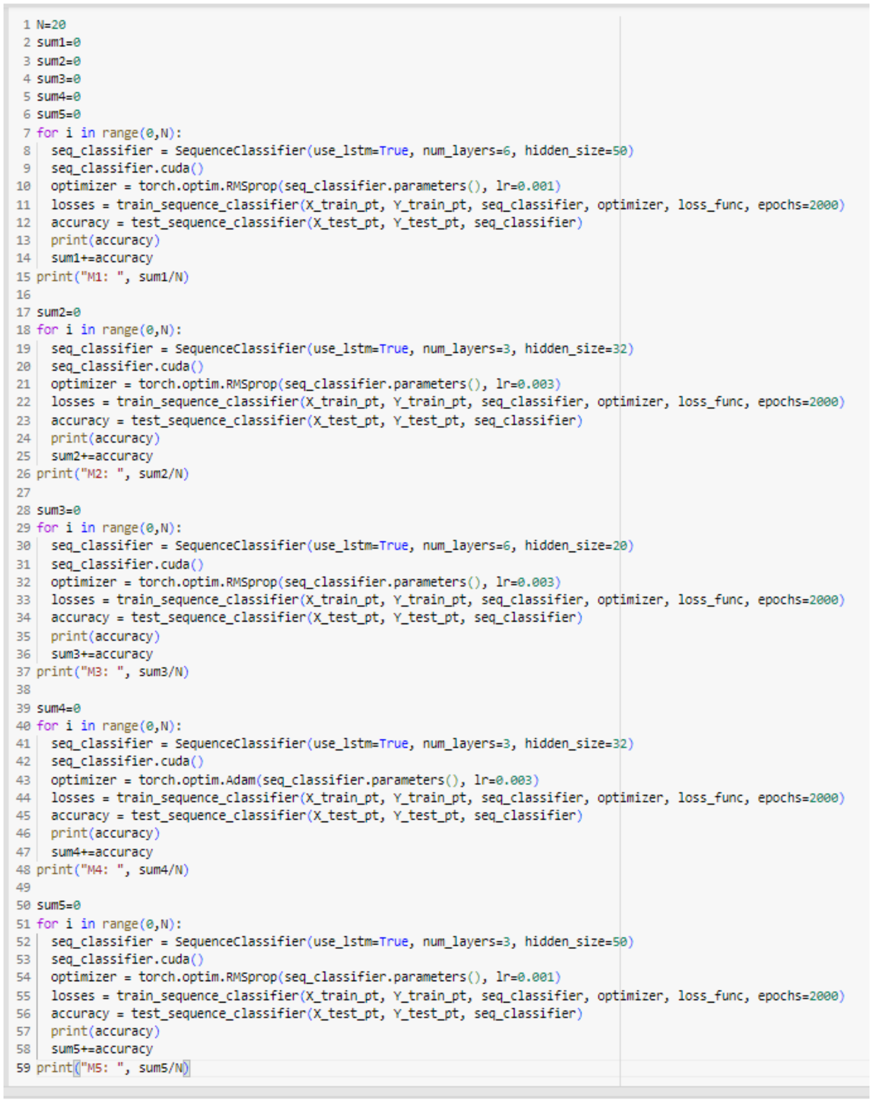
```python
from itertools import product
# Define the grid of hyperparameters
exp_hidden_size = [15, 20, 32, 50]
exp_num_layers = [3, 5, 6]
exp_learning_rate = [3e-3, 2e-3, 1e-3]
N=10

# Define the list of optimizers to try
exp_optimizers = [torch.optim.RMSprop, torch.optim.Adam, torch.optim.Adamax]
# Perform grid search
best_accuracy = 0.0
best_model_id = ""
best_hidden_size = 0
best_num_layers = 0
best_learning_rate = 0.0
best_optimizer = None

for hidden_size, num_layers, learning_rate, optimizer_class in product(exp_hidden_size, exp_num_layers, exp_learning_rate, exp_optimizers):
  sum=0
  model_id = f'H{hidden_size}_NL{num_layers}_LR{learning_rate}_OPT{optimizer_class.__name__}'
  print(f'Training: {model_id}')
  for i in range(0,N):
    seq_classifier = SequenceClassifier(use_lstm=True, num_layers=num_layers, hidden_size=hidden_size)
    seq_classifier.cuda()
    optimizer = optimizer_class(seq_classifier.parameters(), lr=learning_rate)
    losses = train_sequence_classifier(X_train_pt, Y_train_pt, seq_classifier, optimizer, loss_func, epochs=2000)
    accuracy = test_sequence_classifier(X_test_pt, Y_test_pt, seq_classifier)
    sum+=accuracy

  mean = sum/N
  print("Mean accuracy: ", mean)
  if mean > best_accuracy:
    best_accuracy = mean
    best_model_id = model_id
    best_hidden_size = hidden_size
    best_num_layers = num_layers
    best_learning_rate = learning_rate
    best_optimizer = optimizer_class.__name__

print("Grid search complete.")
print(f"Best model: {best_model_id}")
print(f"Best hidden size: {best_hidden_size}")
print(f"Best number of layers: {best_num_layers}")
print(f"Best learning rate: {best_learning_rate}")
print(f"Best optimizer: {best_optimizer}")
print(f"Best accuracy: {best_accuracy}")
```

This also lasted almost 2 hours, but it gave us the last insights we needed to understand what were the best hyper-parameter values to choose. Another note here is that we are conscious that we did not choose the nicer or best programming technique to compute the mean, since we could have used a dictionary to store the losses and then compute the mean, but in this way we could easily experiment with all the hyper-parameters in a greedy search. Also, we used N=20 which is clearly too high but we used this high value to be more sure of the good performance of the chosen models, since for N=3, for instance, the randomness played a higher role.

At this point, we observed that with a hidden size of 50 we get clearly better performance with the model, and that RMSprop was the most accurate model in general with Adam so close. However, we were conscious that 50 may be too much, which may not be any problem in this lab, but could imply some problems if we extended this implementation for further study, so for a most optimal solution, also in ter4ms of complexity and computational cost, we explored different alternatives with a hidden size of 32, and they were indeed highly competitive with the previous best models.

A relevant discovery here is that we could get up to 1 of accuracy with a couple of models, which means 100% of well-classified sequences. It was so rare since the randomness is constant in each execution, but getting that meant that we were working in the right direction since we had the chance to obtain the most accurate model for our test data. However, we did not want to choose the model whose maximum accuracy found was the highest, but the model whose accuracy average was the highest, since it showed a good general performance for our data.

Following this criteria, we selected 5 different models, with hidden size of 50, 32 or 20, learning rates of 0.001 or 0.003 and using either Adam or RMSprop optimizer and 2000 epoch and LSTM. The hyper-parameter selection was not random at all, we selected the 4 best combinations observed and tested 20 times, to observe which average accuracy really performed best. Beside the top 4 models, we also select the best model with a hidden size of 20, in order to see if the increase in size was worth it in terms of average accuracy. This is the code for the top 5 models selected:



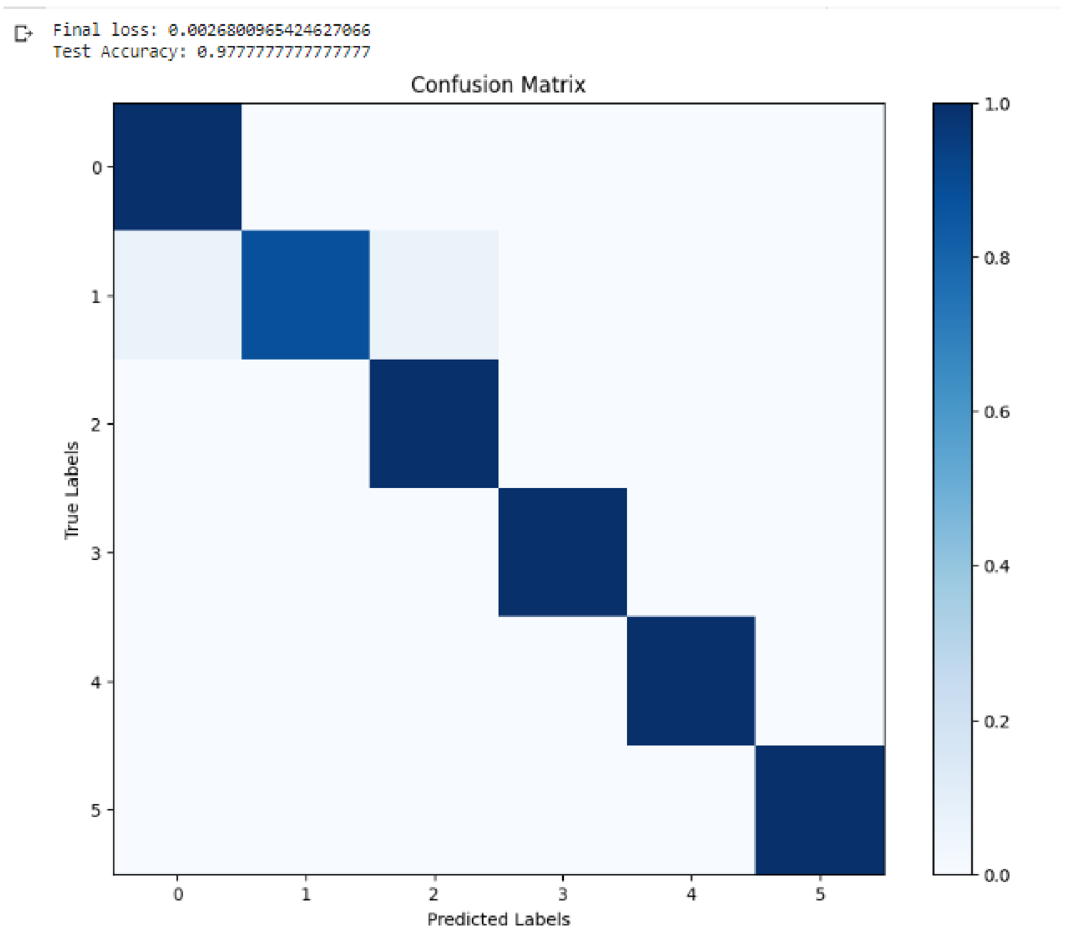
```
N=20
sum1=0
sum2=0
sum3=0
sum4=0
sum5=0
for i in range(0,N):
  seq_classifier = SequenceClassifier(use_lstm=True, num_layers=6, hidden_size=50)
  seq_classifier.cuda()
  optimizer = torch.optim.RMSprop(seq_classifier.parameters(), lr=0.001)
  losses = train_sequence_classifier(X_train_pt, Y_train_pt, seq_classifier, optimizer, loss_func, epochs=2000)
  accuracy = test_sequence_classifier(X_test_pt, Y_test_pt, seq_classifier)
  print(accuracy)
  sum1+=accuracy
print("M1: ", sum1/N)

sum2=0
for i in range(0,N):
  seq_classifier = SequenceClassifier(use_lstm=True, num_layers=3, hidden_size=32)
  seq_classifier.cuda()
  optimizer = torch.optim.RMSprop(seq_classifier.parameters(), lr=0.003)
  losses = train_sequence_classifier(X_train_pt, Y_train_pt, seq_classifier, optimizer, loss_func, epochs=2000)
  accuracy = test_sequence_classifier(X_test_pt, Y_test_pt, seq_classifier)
  print(accuracy)
  sum2+=accuracy
print("M2: ", sum2/N)

sum3=0
for i in range(0,N):
  seq_classifier = SequenceClassifier(use_lstm=True, num_layers=6, hidden_size=20)
  seq_classifier.cuda()
  optimizer = torch.optim.RMSprop(seq_classifier.parameters(), lr=0.003)
  losses = train_sequence_classifier(X_train_pt, Y_train_pt, seq_classifier, optimizer, loss_func, epochs=2000)
  accuracy = test_sequence_classifier(X_test_pt, Y_test_pt, seq_classifier)
  print(accuracy)
  sum3+=accuracy
print("M3: ", sum3/N)

sum4=0
for i in range(0,N):
  seq_classifier = SequenceClassifier(use_lstm=True, num_layers=3, hidden_size=32)
  seq_classifier.cuda()
  optimizer = torch.optim.Adam(seq_classifier.parameters(), lr=0.003)
  losses = train_sequence_classifier(X_train_pt, Y_train_pt, seq_classifier, optimizer, loss_func, epochs=2000)
  accuracy = test_sequence_classifier(X_test_pt, Y_test_pt, seq_classifier)
  print(accuracy)
  sum4+=accuracy
print("M4: ", sum4/N)

sum5=0
for i in range(0,N):
  seq_classifier = SequenceClassifier(use_lstm=True, num_layers=3, hidden_size=50)
  seq_classifier.cuda()
  optimizer = torch.optim.RMSprop(seq_classifier.parameters(), lr=0.001)
  losses = train_sequence_classifier(X_train_pt, Y_train_pt, seq_classifier, optimizer, loss_func, epochs=2000)
  accuracy = test_sequence_classifier(X_test_pt, Y_test_pt, seq_classifier)
  print(accuracy)
  sum5+=accuracy
print("M5: ", sum5/N)
```

After the execution of this code, we got results higher than 0.91 in all the cases except from the model with 20 as hidden size, which helped us conclude that an architecture of 32 or 50 was more accurate than one with 20. All the values were pretty close, but the second model performed slightly better in some cases, so we chose it following the complexity and computational criteria too, since the accuracy mean was almost the same as the models with 50 of hidden size but the architecture dimensions were quite different. To be precise, the first model had an accuracy average of 0.93, the second 0.92, the third 0.85, the fourth 0.91 and the fifth 0.91.

Therefore, we finally selected the network architecture of hidden_size=32, num_layers=3 and use_lstm=True, using a learning rate of 0.003 and 2000 epoch. An important note here is that we are conscious that these optimization algorithms are known by its technique of changing the learning rate value at each iteration selecting the most proper one, so we could have omitted the learning rate value searching in the massive grid. However, the differences between the models with different initial learning rates were slightly significant, that’s why we finally chose a learning rate of 0.003, because in general we got better accuracy with this. Again, we know that we did not choose the nicest way to program the model's iterations to compute the average, but we selected this manner since it illustrated the differences within the code in an easier way.

Once we had everything ready for the model choosing, we came up with the following question: What if other optimizer was better with this network architecture, since we only considered the best models and the best model for this architecture was with RMSprop, but it may vary using more executions of models to compute the mean. In order to try that, we copied the second model but using Adam instead of RMSprop, and surprisingly we got slightly better results. Specifically, we got 0.929 for this new model and 0.915 for the same model with RMSprop. Even if the difference is so small and it may vary depending on the executions, we preferred to choose the model with Adam, since it is an optimizer we have studied in class, so we understood better the results and were able to make conclusions not only in practical terms but also in theoretical terms. Having said that, if we were not sure enough of our selection, we executed the code of the 20 iterations for the two models even 2 times more, to be completely sure of our choosing, and the new model with Adam indeed returned an average closer to 0.93 than the same model with RMSprop, that rounded 0.92, while the model M! from the previous code screenshot, which performed the best, with hidden size of 50, also returned around 0.93, so we finally chose this new model with Adam optimizer, hidden_size=32, num_layers=3, use_lstm=True, lr=0.003 and epoch=2000.

Finally, we included the code for the chosen model, which is the only part not commented about the executions of the models. Executing that cell we get the test accuracy, the final loss and the confusion matrix visualization as a heat map, as already explained. In our execution we got a final loss of 0.97 with the following confusion matrix:

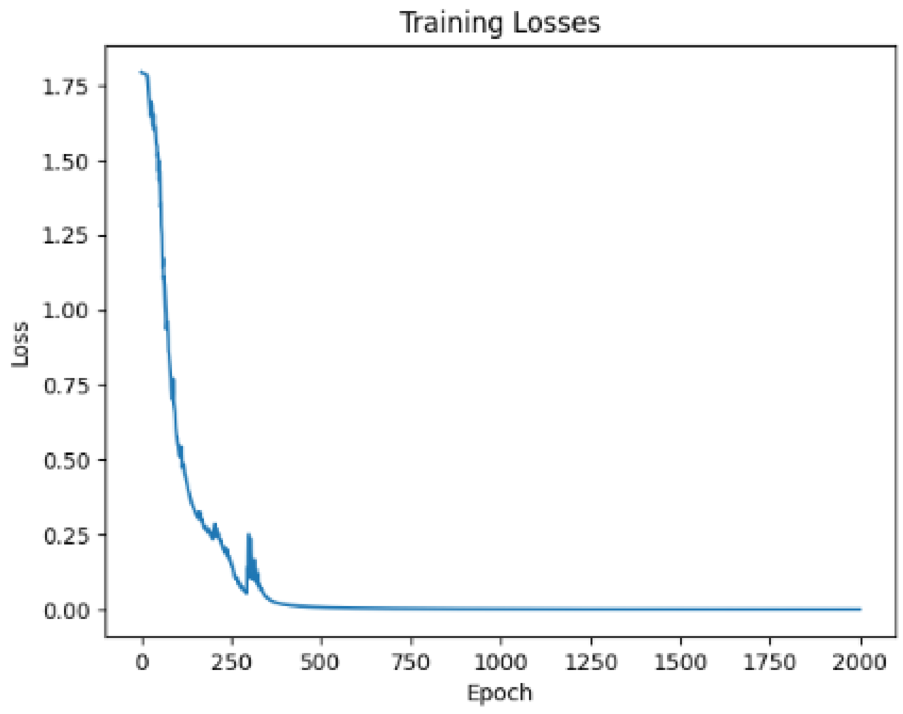


```
Final loss: 0.0026800965424627066
Test Accuracy: 0.9777777777777777
```

As we can see in the confusion matrix, the diagonal is almost perfectly fitted which means that the prediction is almost perfect. We observe that the cell corresponding to the predicted and true label 1 is slightly lighter than the rest, and this is because some samples from class 1 have been misclassified. If we observe at the same row, the cells corresponding to the predicted label 0 and 2 are slightly colored, which means that the few samples misclassified from class 1 were assigned to class 0 and class 2. This shows how useful our confusion matrix visualization is, since we can perfectly detect where the model is failing, and that is with classifying the samples from class 2, which can be due to the similarities of this class to the other two classes.

An important consideration here is that this is only one execution of the model, and that means that, when executing it, you can get any accuracy value from 0.85 to 0.99 approximately. We have considered the first execution of this cell, but if this code is executed when checking our conclusions, it must be recalled that the randomness plays an important role, as two executions of the same code can show, so an accuracy of 0.85 does not contradict anything from our report, as neither it does an accuracy of 0.99. If a small accuracy value is obtained, such as 0.85, it is recommended to execute again the code cell in order to get some values closer to what we have discussed in this report, since these higher values should be more likely to happen than the smaller ones.

Finally, in order to optimize our study, we have decided to also include a plot of the loss of our model execution, even though it was not asked, and it looks like the following screenshot:



It clearly shows an abrupt decrease in the loss value, which gets really close to 0 from the 400th epoch more or less. In some executions we have observed an abrupt increase and decrease again in the loss around the 1000th epoch, which could have happened due to overfitting given those parameters, but since we have selected epoch=2000 on purpose, arriving at this point there is no loss with those high values.

To sum up, after adapting the network architecture, loss function and test function for a multi class classification, we have found that the generally best model for our data is an architecture of 32 hidden size, 3 layers, using LSTM and considering a learning rate of 0.003, 2000 epoch and the Adam optimizer.

Ex2:

In the second exercise we were asked to create a neural network able to decrypt a set of 32-length encrypted sentences including 4 corrupted chars randomly distributed. The first thing we have done is essentially obtaining the decryption keyword of the Vigenère encryption just comparing both sentences, the ciphertext and the plaintext, which is called Known-Plaintext Attack. Doing simple operations we finally obtain that the 7-length keyword is: "`LCHMJYT`", we have double-check it by decrypting two different ciphertext using a function called *vigenere_decrypt* that we have created, but this is not relevant. Once we have checked that both recovered plaintexts were the same as the ones we already had, we were able to start creating the network.

But before starting, we need to redefine the alphabet since now we have an additional symbol that represents the corruption that is “-”. So the new alphabet would be “ABCDE…XYZ-”.

The network that we will use has the same architecture as the one in the examples since the problem, inputs and outputs are practically the same. The only difference is that our test data is corrupted and this will reduce the performance of that network. Unlike test data, the train dataset is not corrupted making the training of the network even more difficult. Firstly, we tried to execute the network as it was, without changing anything. The accuracy obtained was always below 50% so it was not efficient. After a while changing hyperparameters, we realized that increasing the number of layers increases the performance of the network obtaining an accuracy of about 87%. It would be a good result if this percentage were not the exact amount of non-corrupted characters of the network. So we essentially were decrypting everything that was not corrupted but we were not able to handle the corruption.

Once we have identified the problem, the first thing we need to do is design a strategy to handle the corrupted and uncorrupted data during the network training. Since we cannot revert the corruption of the test data, the best, and only, option to train a network able to process and decrypt corrupted data is with more corrupted data. At this point our main goal is to transform the uncorrupted train dataset into a corrupted data like test. To do so we have created some functions to introduce in the train encrypted sentences corruption, i.e. “-”. Firstly, we have created a function called *corrupt_tensors* that essentially, with a given probability, introduces into train encrypted tensors an additional char “-”.

After several tries and errors, we decided to create another corruption function, *randomized_corrupt_tensors* whose functionality is the same as the previous one but with the difference that we state instead of the corruption probability, the probability of changing that corruption probability. We will explain it later during the training of the network. These are the two functions that we have implemented, they work similarly, but essentially they iterate through each letter of each sentence (tensor) of the dataset and with probability *probability* they change that letter by the corruption.

```
# Corrupt with equal probability all sentences
def corrupt_tensors(tensors, probability):
  corrupted_tensors = []
  for tensor in tensors:
    corrupted_tensor = []
    corrupted_tensor.append(tensor[0].clone())
    corrupted_tensor.append(tensor[1].clone())
    for i in range(len(tensor[0])):
      if random.random() < probability:
        corrupted_tensor[0][i] = 26  # Index 26 represents "-"
    corrupted_tensors.append(corrupted_tensor)
  return corrupted_tensors

# Corrupt with variable probability each tensor
def randomized_corrupt_tensors(tensors, swap_probability = 0.05):
  corrupted_tensors = []
  probability = 0.125
  for tensor in tensors:
    corrupted_tensor = []
    corrupted_tensor.append(tensor[0].clone())
    corrupted_tensor.append(tensor[1].clone())
        if (random.random() < swap_probability): probability =
random.uniform(0,0.6)  # Probability of change the current corruption
probability
    for i in range(len(tensor[0])):
      if random.random() < probability:
        corrupted_tensor[0][i] = 26  # Index 26 represents "-"
    corrupted_tensors.append(corrupted_tensor)
  return corrupted_tensors
```

Once we had a corrupted training set, we started to test with our current model with the hyperparameters we have used previously, 2 layers and 16 hidden neurons per layer, now the performance was slightly better, going from 87% to 89% approximately. This would seem anything compared to the effort done, but this is because we are seeing the increment of accuracy as an absolute value to the overall accuracy and we need to see the accuracy relative to the corruption. To do so we modified both, the training function to return 3 accuracies, the overall accuracy, the corrupted accuracy and the uncorrupted accuracy and the evaluation, where we included new plots of each accuracy and we print the top N best decrypted words to see a practical result. The meaning of each accuracy is different, while the overall accuracy is an indicator to see how well is performing the network globally, the corrupted accuracy is essentially telling us how much contributes the corruption decryption to that accuracy. This second accuracy is part of the overall accuracy and several times smaller than the uncorrupted accuracy (the other part of the overall accuracy). But even if it is smaller, it is the one that we are looking to improve our model. In the evaluation, we considered relevant to print some sentences, decrypted with the neural network to see in practical terms how efficient it was with real examples. To do so we have ordered by each accuracy each sample and printed the N ones with the highest accuracy.

Now that we have the training and the evaluation functions, we can start trying to improve the performance of the neural network. Firstly we had an accuracy of 89% at most, but we saw that the corruption correction was not enough. After trying with different parameters and corruption probabilities, we realized that it may be better to train the model with less uniform corruption, not a single probability. So to make the network more robust against corruption we implemented the second corruption function, the *randomized_corrupt_tensors* to make more unpredictable the corruption and train the model with that, essentially instead of changing the *probability* itself we change the rate of variation of that *probabiliy*, ie with a probability *swap_probability* we change the corruption probability *probability*:

```
if(random.random()< swap_probability): probability = random.uniform(0,0.6)
```

At the same time we considered doing data augmentation. We had considered several alternatives, such as creating more data, performing transformations to our existent data, reorganizing our data,... But the ones we have thought were better were essentially to apply transformations and change the corruption of our data. To do so we had created a dataset called *augmented_train* that after some tries we decided to make it combining already existent datasets: 1 time the whole dataset of train without corruption, 1 time the whole dataset of train with a fixed corruption of 12’5% using *corrupt_tensors* (the same corruption as test), 2 times train with randomized corruption with variability rates of 5% and 10% using *randomized_corrupt_tensors*. To summarize, this new dataset is 4 times train and includes essentially, 25% of data without corruption, to see the relations between sentence words, 25% of data like test, to bias it to test-like data and 50% of random corrupted data that can goes from 0% to 60% of corruption to make it more robust against corruption wherever it is.

Even if this is 4 times the train data size, it barely is equivalent to 1.5 times increment of data since all the time we are reusing the same base sentences and it does not add any kind of variance. Although, it is the best option we have found since randomly reordering the current data may break completely the patterns that the neural network is training and it may lead to meaningless sentences. Another good option could be creating more samples, but since we don't know which kind of sentences are used and with which constraints we thought that it may bias or overfit the model since we might add sentences that are not possible in this context.

Now, with this new training set, we obtained about 90% of overall accuracy and about 2'5% of corrupted accuracy that is a 20% of corruption correction that even if it is not a high rate, we need to consider that the corruption problem is not easy to solve, since it does not follow any pattern to predict.

After all this process, we only need to change the hyperparameters, and other functions to try to optimize the error the most that we can. To do so, we have created a new function that essentially iterates over possible hyperparameters, loss functions and architectures to see which is the best option for our model. After several iterations, we have obtained that the best architecture and hyperparameters for our network are the following:

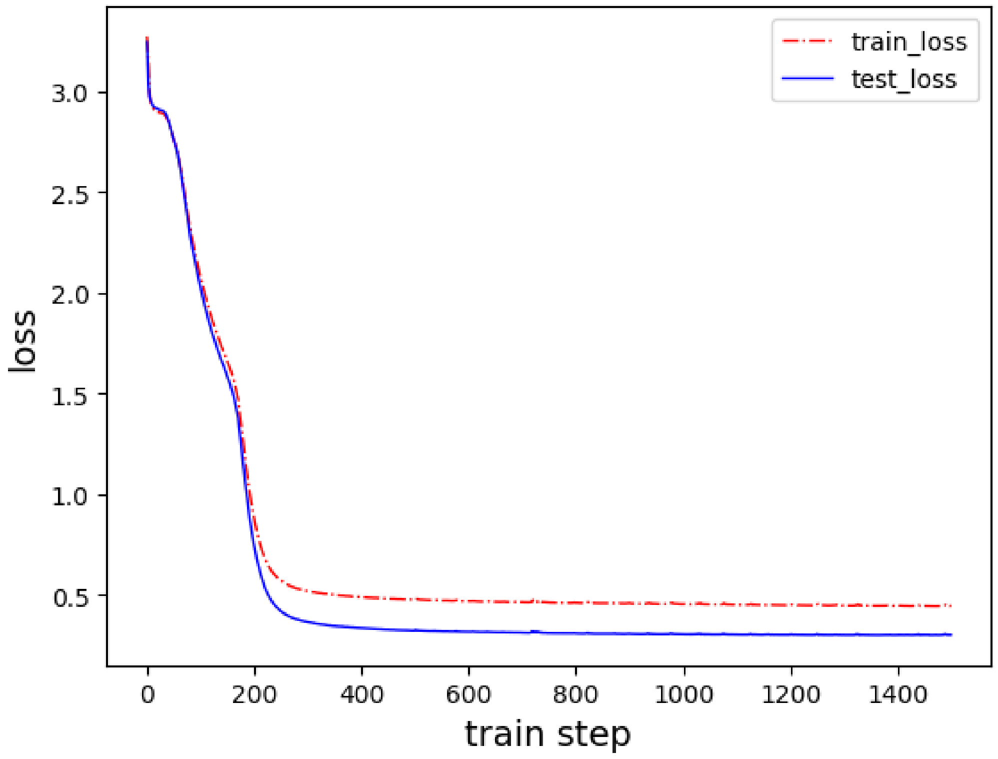
1. *Use_LSTM = True*: Even if we have tried it without using LSTM, it works several times better with it, even if it takes more time.
2. *Use_CUDA = True*: It does not affect the results, just the execution time.
3. *letters_embedding_size = 32*: After train and error we have found that with a bigger embedding, the neural network performs better, since it can make more relations between letters and its positions and it is translated with better accuracy.

4. *num_layers = 3*: Here we had the most important part, since changing between the number of layers completely changes the results, with just 1 layer, the accuracy is lower, specially in the corrupted chars, the same happens with more than 5 due to overfitting. We have observed that the best performance is between 2 and 4 layers, with the best of them in 3,4 but since the result was practically the same, we decided to choose 3 layers since it is significantly faster than 4 layers.
5. *num_letters = len(vocabulary)*: It cannot be changed.
6. *hidden_size = 32:* This also was an important parameter which also changed the performance of the network, it varies a lot between distinct values. It is less relevant than the number of layers, but the best performance was essentially with 16, 32 and 64. Even if with 64 sometimes it provides better results than with the other two, it also takes more time and sometimes produces overfitting, for this reason we have chosen 32 because it is faster than 64, does not produce overfitting apparently and has better results than 16.
7. *optimizer = torch.optim.Adam(...)*: Even if we tried other optimizers such as Adagrad, Adamax, Adadelta and RMSprop, the better performance was in Adam. Adadelta did not increase the accuracy more than 11%, Adagrad until 56%, Adamax until 90% and RMSprop slightly smaller than Adam.
8. *loss = torch.nn.NLLLoss()*: We essentially tried two loss functions, the first one the Cross Entropy and the second one Negative Log-Likelihood, even if the results were similar, the second one provides slightly better performance, and since both are multiclass classifiers it does not care a lot which to use.

9. *epochs = 1500*: The number of epochs just increases the iterations that the model does. We have observed that with bigger epochs it increases the accuracy, but after 1200 the accuracy does not increase a lot since it is closer to the convergence point.
10. *lr = 1e-2*: It is not too relevant since we are using Adam and it automatically updates the value of the learning rate, but this starting *lr* seems to work correctly with our model.

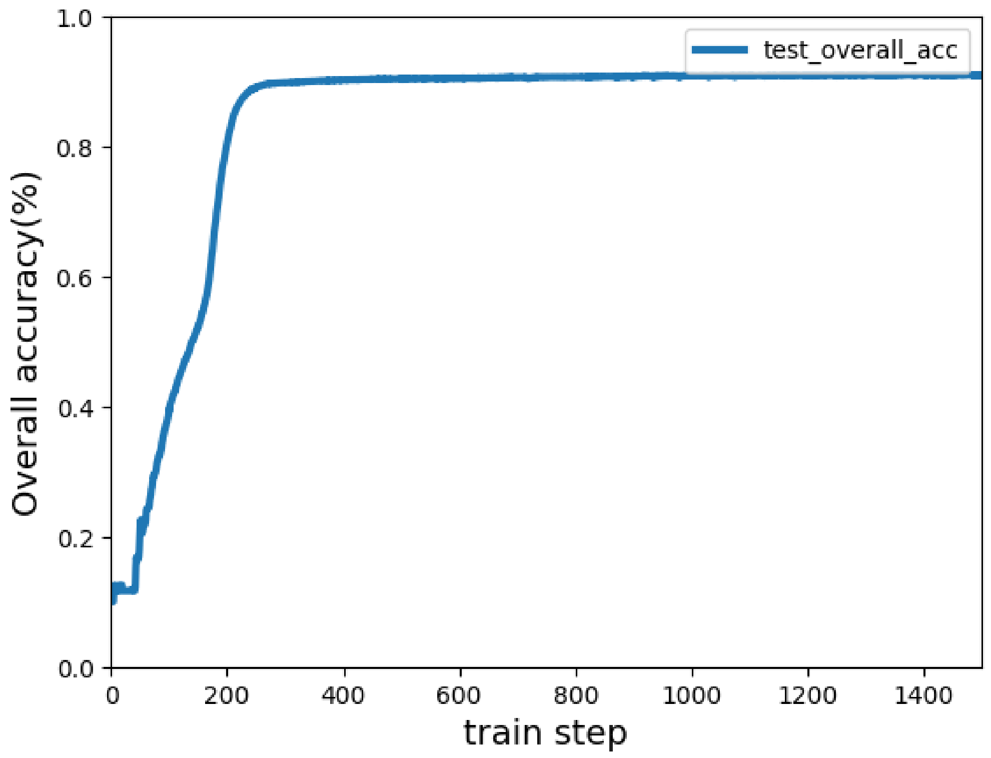
So with all these chosen hyperparameters we have obtained a Training Loss of 0.459, a Test Loss of 0.303, a Overall Accuracy of 90.9, a Corrupted Accuracy of 3'61% and a Uncorrupted Accuracy of 87.4% .

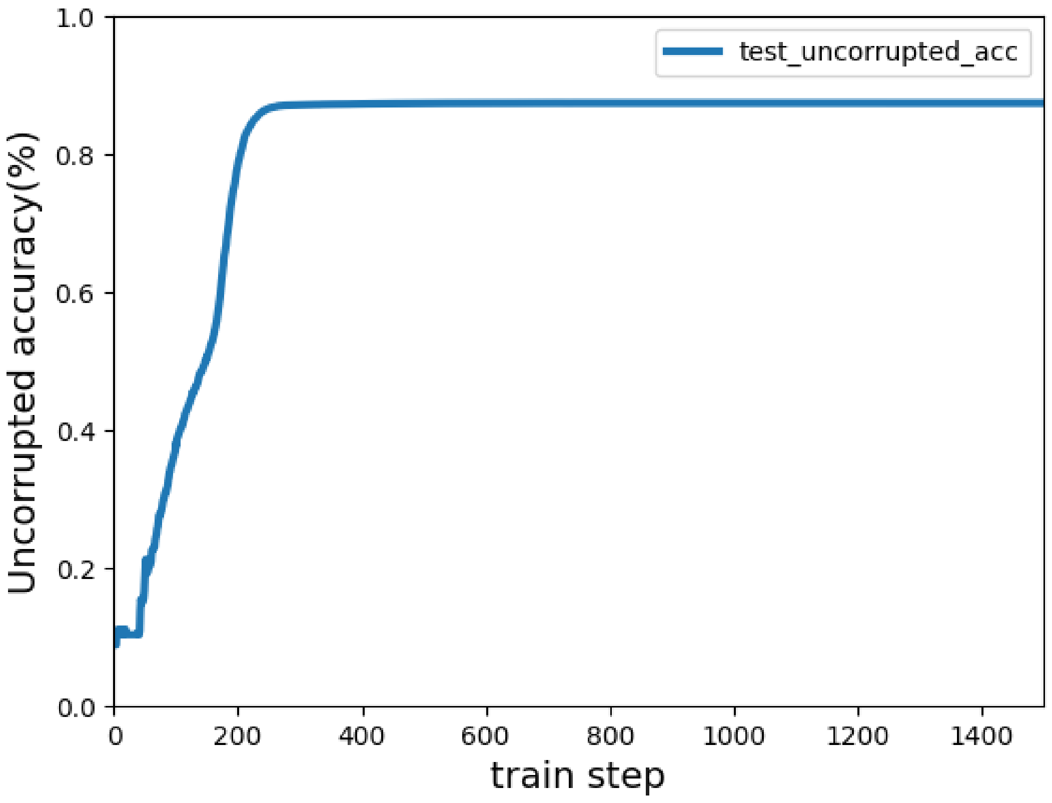
As we can see, these are the best values we have obtained.



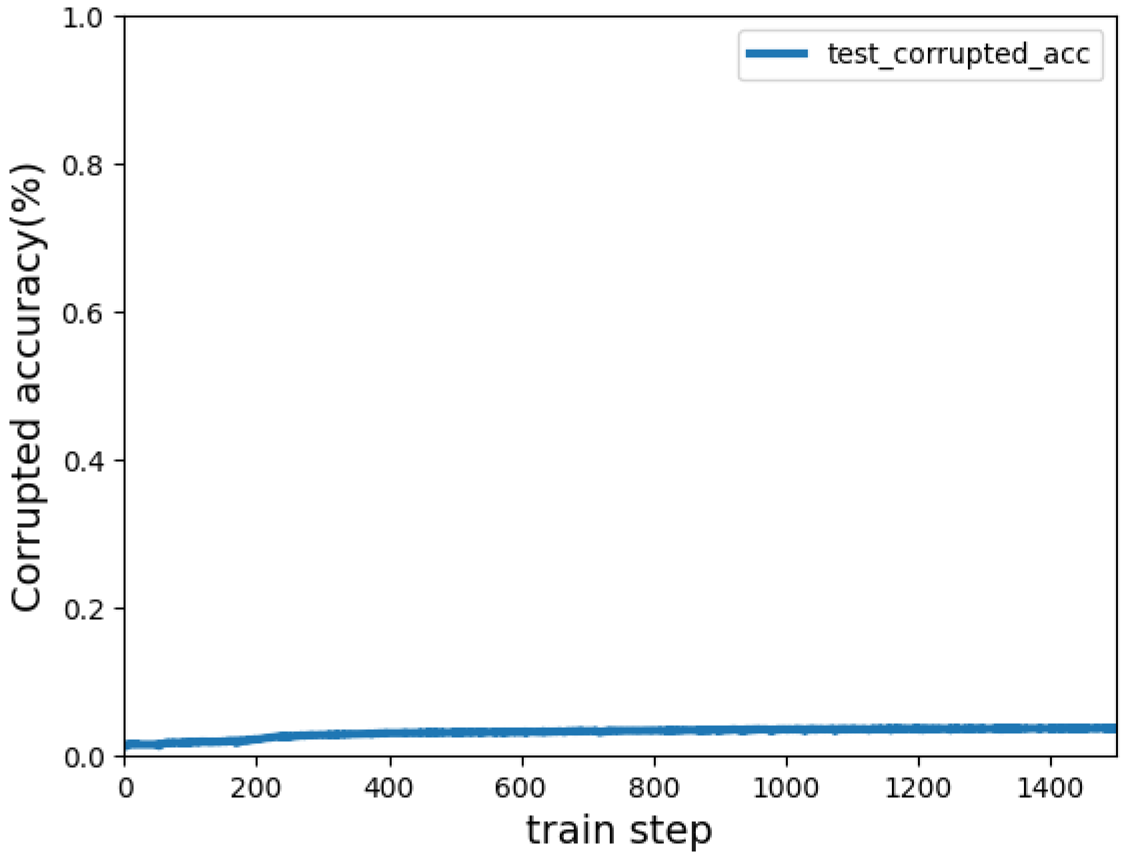
It may seem strange that the test loss is smaller than the train loss, but this makes sense if we look at the construction of the train data since it may have bigger corruption than the test data and this makes it more difficult for the model to correct that corruption. Also we can say that it does not have overfitting since in that case, the test loss would be much bigger than the train loss and that is not the case. In the plot of above, we can clearly see that how the test loss decreases similarly to the train loss until we reach the 200 epochs, then the train loss stops decreasing while test loss decreases slightly more.

About the accuracies, we can clearly see that the uncorrupted characters are almost all correctly decrypted since it has an accuracy of 87’4% and there are an 87’5% of uncorrupted chars so it is almost a 100% of uncorrupted accuracy. Respect the uncorrupted accuracy, it is almost a 30% of correction of corrupted chars, even if it is not the best option, it still being a good correction, taking into account that we have a small train set and the corruption problem is a too complex to solve it easily.





As we can see in these plots, how the accuracy grows is more or less the same since uncorrupted chars are almost the 90% of the train data so its contribution to the overall accuracy is much bigger. Note that the while the overall accuracy stills growing slowly after the 200 epochs, the uncorrupted accuracy remains constant, this is because after that point the model is able to predict about 100% of uncorrupted chars. Then, from where the overall accuracy still growing is from the corrupted accuracy.



As we can see here, even if the contribution of corruption accuracy to the overall accuracy is much smaller than the uncorrupted one, it remains growing slowly but constantly, even after the 200 epochs where the uncorrupted stopped. In fact, it goes faster, this is because when the model is already able to predict all decryption of the uncorrupted characters, it is ready to start to predict how to correct the corruption based on the patterns and weights that the network has learned.

In conclusion, we have seen that with that architecture and those hyperparameters, we can partially correct the corruption of the model, it may not be perfect, but we can still making optimizations and create more complex architectures to solve this problem. Now we will show some of the decryptions that the model has predicted:

```
Overall accuracy top examples:
```

```
Original Message encrypted: XBHBKMUD-PVVVO-PMCDYPA-LK-OVGLHH
Message decrypted: IDONTKNOWWHETHERTOMWILLSWIMORNOT
Prediction Message decrypted: IDONTKNOWWHETHERTOMWILLSWIMORNOT
Prediction Message Accuracy : 1.0
Original Message encrypted: NCLHVTKPWBGROV--RSJ-KPWMVRPADBT-
Message decrypted: YESTERDAYISAMOREYESTDAYTHANTODAY
Prediction Message decrypted: YESTERDAYISAMOREYESTDAYTHANTODAY
Prediction Message Accuracy : 1.0
Original Message encrypted: XBHBKVOXLD-JJV-JWOKVLCBM-RVLK-GH
Message decrypted: IDONTTHINKISHOULDATTENDTHATEVENT
Prediction Message decrypted: IDONTTHINKISHOULDATTENDTHATEVENT
Prediction Message Accuracy : 1.0

Corrupted accuracy top examples:

Original Message encrypted: XBHBKMUD-PVVVO-PMCDYPA-LK-OVGLHH
Message decrypted: IDONTKNOWWHETHERTOMWILLSWIMORNOT
Prediction Message decrypted: IDONTKNOWWHETHERTOMWILLSWIMORNOT
Prediction Message Corrupted Accuracy : 0.12
Original Message encrypted: NCLHVTKPWBGROV--RSJ-KPWMVRPADBT-
Message decrypted: YESTERDAYISAMOREYESTDAYTHANTODAY
Prediction Message decrypted: YESTERDAYISAMOREYESTDAYTHANTODAY
Prediction Message Corrupted  Accuracy : 0.12
Original Message encrypted: XBHBKVOXLD-JJV-JWOKVLCBM-RVLK-GH
Message decrypted: IDONTTHINKISHOULDATTENDTHATEVENT
Prediction Message decrypted: IDONTTHINKISHOULDATTENDTHATEVENT
Prediction Message Corrupted Accuracy : 0.12

Uncorrupted accuracy top examples:

Original Message encrypted: XFT-VHHXR-WEGH-FTBUGCT-RCEGVUWHI
Message decrypted: IHAVEFAITHINEACHANDEVERYONEOFYOU
Prediction Message decrypted: IHAVEFAITHINEATHANDEVERYONEOFYOU
Prediction Message Uncorrupted Accuracy : 0.88
Original Message encrypted: XFT--CMG-XBUYODFTGRJVJQ-WEDVHRHB
Message decrypted: IHAVEAFRIENDWHOHASAHOUSEINBOSTON
Prediction Message decrypted: IHAVEAFROENDWHOHASAHOUSEINBOSTON
Prediction Message Uncorrupted Accuracy : 0.88
Original Message encrypted: -FTR-QPSCTMFWDTP-GFIVDBTHTQVZG-U
Message decrypted: IHADNOIDEAYOUWERESOGOODATCOOKING
Prediction Message decrypted: THADIOIDEAYOUWERTSOGOODATCOOKING
Prediction Message Uncorrupted Accuracy : 0.88
```

As we can observe, there are 100% accuracy of the 3 types, (100%, 12% and 88% respectively). The corrupted accuracy in some cases is able to correct the whole sentence as we can see in the Corrupted accuracy example where the sentences are correctly decrypted entirely. So we can conclude that our network is able to efficiently correct some of the corrupted sentences of the dataset.