UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Application of AI for ciphertext identification

Supervisor Prof. Stelvio CIMATO
Co-Supervisor Prof. Bernhard ESSLINGER

Author Stefano SALA
Matricola 12490A
Academic year 2024-2025

dedicato a mamma, papà, Francesco e Alice …

Astratto

In un mondo in cui l'intelligenza artificiale è sempre più in sviluppo, sia nella teoria che nella pratica, questa tesi svolge il compito di introdurvi al progetto NCID di CrypTool, il quale utilizza il Machine Learning per il rilevamento del cifrario dato un testo cifrato.

L'obiettivo della tesi è introdurre i framework Keras e PyTorch, con particolare attenzione alla traduzione dell'attuale implementazione del modello di allenamento e valutazione da Keras a PyTorch. Le differenze tra i due framework vengono analizzate nel dettaglio, insieme alle rispettive caratteristiche e strategie di implementazione.

Infine, la tesi si propone di confrontare i risultati ottenuti dalla nuova implementazione con quelli della versione originale, al fine di analizzarne le differenze e trarre delle conclusioni significative.

Abstract

In a world where artificial intelligence continues to evolve both theoretically and practically, this thesis presents the NCID project by CrypTool, which uses machine learning techniques to identify the type of cipher used in a given ciphertext.

This thesis focuses on the use of the Keras and PyTorch frameworks, with particular attention to the translation of the current model training and evaluation implementation from Keras to PyTorch. The differences between the two frameworks are examined in detail, together with an analysis of their characteristics and implementation strategies.

Finally, the thesis aims to compare the results obtained from the new implementation with those of the original version, in order to analyze the differences and draw meaningful conclusions.

Contents

Listings

List of Figures

1 Introduction

During my studies, I developed a strong interest in cryptography, and in particular in the breaking of encrypted texts and all the mathematical logic behind it. For this reason, I decided to become part of the CrypTool (CT) team. CrypTool[1] is an international open-source project which develops free e-learning software for illustrating cryptographic and cryptanalytic concepts. One of their outcomes is CrypTool-Online (CTO), a web-based tool that offers knowledge about cryptology and free tools to self-educate and to teach about cryptology. Part of CTO is the Neural Cipher Identifier (NCID) app[2] which can be found on GitHub [1].

The task assigned to me was to implement modifications to the code base that is used in the **Neural Cipher Identifier** (**NCID**) app. Specifically, the modifications consisted of implementing two different Machine Learning architectures among the five available, namely FFNN and LSTM (see section 2.2), migrating from the current Keras framework to the new PyTorch framework.

Therefore, in order to better understand the process carried out, we first need to introduce some aspects: What is NCID? How is it related to cryptography? What tools did I use to implement these modifications?

1.1 NCID app

The NCID project started as a master thesis supervised by the University of Applied Sciences Upper Austria, Hagenberg, and the CrypTool project. The NCID project is based on the 2021 paper by Nils Kopal *''Of Ciphers and Neurons – Detecting the Type of Ciphers Using Artificial Neural Networks''* [2]. Of course, this project has been further developed and improved over the years, and in the theory chapter we will see who implemented the various ciphers and architectures of the project (see section 2.2).

The NCID app, which is shown in fig. 1.1, allows the user to input an encrypted text string of at least 30 characters, and from that identify the cipher used for the encryption. For that, NCID uses several neural networks from which one or more can be selected. By doing so, it removes a part that can be considered complicated for a user who is approaching the world of cryptography for the first time.

[1] CrypTool: https://www.cryptool.org/en/
[2] NCID: https://www.cryptool.org/en/cto/ncid/

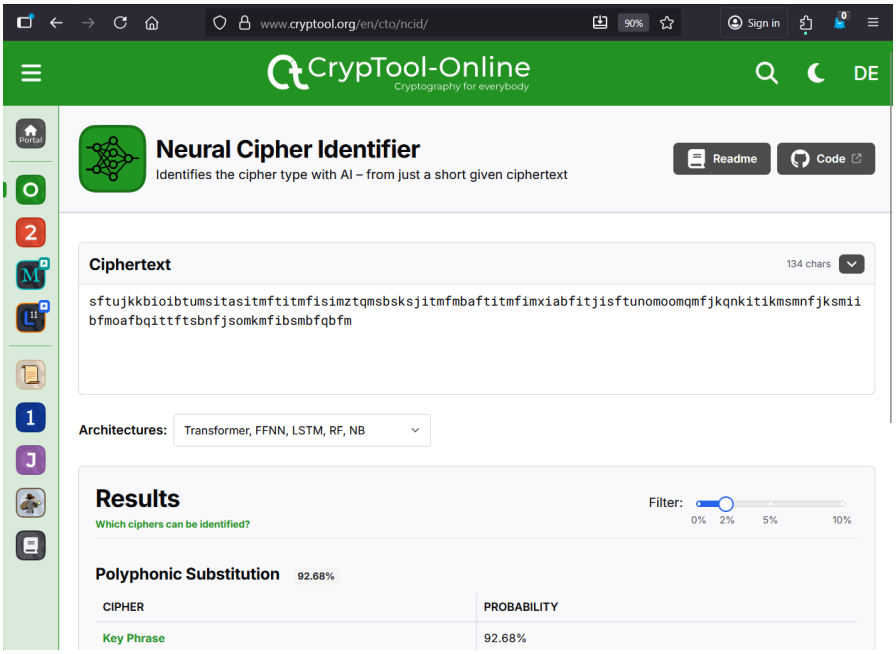

Figure 1.1: Screenshot of NCID app

In fact, in cryptanalysis, the identification of the cipher is the first step required to be able to decrypt an encrypted text. If you do not know which algorithm (cipher) was used to encrypt the text, you can hardly choose the correct attack technique or analysis method. For this reason, decryption often begins with statistical or structural observations on the ciphertext, which serve precisely to hypothesize which cipher has been used. In this regard, we will introduce in the theory chapter what are the statistical features used that allow the model to provide us with an accurate result, that is a correct result in predicting the type of cipher.

1.2 Cryptography aspects

ACA refers to the American Cryptogram Association, an organization founded in 1930 in the United States, dedicated to enthusiasts of classical cryptography and cryptogram puzzles. It is a community of amateur and professional cryptographers that for nearly a century has published the bulletin *The Cryptogram*, where classical ciphers (substitution, transposition, Vigenère, Playfair, etc.) are proposed and solved, using their own notation and standards.

The NCID project initially intended to solve a challenge based on identifying ACA cipher types[3]. Starting with a paper by Kopal [2] only the 56 ACA ciphers were originally planned. Later, the project was extended to recognize the five rotor ciphers — Enigma, M-209, Purple, SIGABA and Typex — also thanks to the work of Dalton and Stamp [3].

[3] See `https://mysterytwister.org/challenges?search=Cipher+ID` for more background.

The NCID app obviously does not aim at detecting modern ciphers, which require enormous computational power to break. First of all the practical problem: modern key spaces are gigantic. For example, a 128-bit key requires up to 2^{128} attempts for a brute-force attack. In addition, modern cryptography is not just simple encryption: it uses modes of operation, authentication, salting, password-iteration schemes, and more. Even if a 'partial' vulnerability was found, the combination of mechanisms often protects the overall communication.

1.3 Developer aspects

For the technical implementation of this project, I used Python, and more specifically I worked closely with the Keras and PyTorch frameworks. Keras is a high-level framework for building ML models, while PyTorch is lower-level and allows the programmer a much broader modification of the data flow (see section 2.5).

All the code provided to me comes from the project's GitHub repository[4], which I was able to fork and then modify and test on my local machine. In particular, we will discuss two files: `train.py` and `eval.py` (see appendices on pages 61 and 79). The first contains all the functions necessary for training the model, while the second includes everything needed for its evaluation. This aspect will then be further explored in the implementation chapter.

[4] See [1]

2 Theory and Foundations

This chapter focuses on outlining the foundational concepts necessary to address the topics covered in this thesis. We begin by examining the fundamentals of machine learning (section 2.1), after which we will shift the focus to the architectures that I modified in the project and explain their evolutionary process within it (section 2.2). We will then look at how the model training process works, highlighting some key concepts of Machine Learning and about the NCID project (section 2.3). We will continue by discussing what statistical features are (section 2.4), and finally, we will discuss the differences between the two frameworks adopted, Keras and PyTorch (section 2.5).

2.1 Machine Learning basics

> "A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**."

This is the definition that Tom Mitchell gives of Machine Learning in his book [4]. As we will notice throughout our work, this definition perfectly applies to our case. In fact, our *Task* consists in identifying the cipher, the *Performance* is measured through the accuracy of the trained model in identifying the correct cipher, and the *Experience* is determined by the dataset of pairs (x, y), where x is our input (that is, the strings encrypted with one of the 61 ciphers), and y are the labels, namely the corresponding correct cipher for each x.

But how does my model map the inputs x to our label y? To accomplish this task, a machine learning model can be trained in different ways: supervised, unsupervised, or reinforcement learning.

Supervised learning In supervised learning, the training set (named D) is composed of pairs (input, label).

$$D_{\text{supervised}} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

The goal is to learn a function f that maps the input x to its label y:

$$f : X \to Y \quad \text{such that} \quad f(x) \approx y$$

Once trained, the model must be able to predict y for new, unseen inputs x.

Unsupervised learning In this case, the dataset contains only inputs, and thus there are no labels y:

$$D_{\text{unsupervised}} = \{x_1, x_2, \ldots, x_n\}$$

The goal is to discover hidden structures, patterns, or relationships in the data, and in this case the model learns a function g that transforms or groups the data:

$$g : X \to Z \quad \text{where } Z \text{ is a space of representations, clusters, or rules}$$

Reinforcement learning Reinforcement learning is a type of machine learning in which an agent learns to make decisions by interacting with an environment, aiming to maximize a cumulative reward. It works without explicit labels for each action, but the agent learns through trial and error, observing the consequences of its actions.

An example could be a robot trying to exit a labyrinth, where the environment is the labyrinth, and the actions are moving up, down, left or right. The robot explores the maze randomly (exploration phase) and records what happens after each action. Over time, it learns that some moves lead to higher rewards than others (for example +10 if it exit and -1 for every step done), so it starts to exploit (exploitation phase) what it has learned.

Clearly, our project uses supervised learning, because as anticipated we provide our model with encrypted texts as input, and we also provide the corresponding labels. After sufficiently training the model, we test it with previously unseen encrypted sentences: the better the model performs at this task, the more accurate we consider it to be.

In particular, ours is a multiclass classification problem, that is, a supervised learning task in which each input $x \in X$ belongs to at least one class among a set of $k > 2$ possible classes. Formally:

$$f : X \to \{c_1, c_2, \ldots, c_k\}$$

where $k \geq 3$ and $f(x)$ returns the correct class c_i for each x.

Now let us focus on how the inputs are mapped, that is, how do we implement our function f?

2.2 Machine Learning architectures

> ”The choice of architecture inherently constrains the representational capacity of a model, determining not only what it can learn, but how efficiently it can transform inputs into useful outputs” [5]

Just as different vehicles serve distinct functions, a car for daily commuting, a tractor for agricultural work, and a truck for freight transport, different machine learning architectures are designed to address specific computational tasks. Certain models excel at image classification, while others specialize in sequential data prediction or clustering analysis. Some architectures demonstrate versatility across multiple domains, while others are finely tuned for specialized applications. This diversity in design reflects the need for tailored solutions in artificial intelligence, where the choice of architecture fundamentally shapes a model's capabilities and performance. This is precisely where our function lies: each architecture adopts a different way of processing and evaluating the inputs, yet they all share the common goal of mapping them into a meaningful output.

But which architectures does the NCID project employ? And who first introduced them?

The idea originates from the paper by Kopal [6], which employs the **Artificial Neural Network** (ANN) for cipher identification. In the paper Kopal uses the book *Make your own neural network* [7] to provide a good introduction to ANNs: neurons are interconnected through input and output links that carry signals, each associated with a specific weight. A neuron includes an activation function a, which determines its output based on the incoming inputs. Concretely, the neuron combines all incoming signals with their corresponding weights, adds a bias term b, and then applies the activation function to this aggregated result to produce the final output (see section 2.3).

A typical design in artificial neural networks is to arrange neurons into distinct layers. The input data are first provided to an input layer composed of n neurons. This layer is connected to one or more hidden layers, which process the information further. The final hidden layer is then linked to the output layer. In this structure, every neuron in one layer is connected to every neuron in the subsequent layer, this behavior is called fully connected layer. The learning, in general, is performed by adapting the weights of the connections between the neurons.

Subsequently, with the paper *A Massive Machine-Learning Approach for Classical Cipher Type Detection Using Feature Engineering*, architectures that computed features (see section 2.4) were introduced through feature engineering, namely Feedforward Neural Networks, Decision Trees, and Naive Bayes networks. But what does feature engineering actually mean?

In traditional machine learning, feature engineering plays a central role. It consists of manually designing and selecting features that best represent the data, often relying on domain expertise and statistical analysis. While this approach can yield effective results, it is limited by the knowledge and creativity of the researcher, and may not capture complex or hidden patterns in the data.

In contrast, feature learning, automatically extracts representations from raw data. Neural networks, for example, learn hierarchical features directly during the training process, reducing the need for manual intervention. This allows models to discover intricate structures that would be difficult to design by hand. However, feature learning requires larger datasets and higher computational resources compared to traditional feature engineering.

In fact, the paper *Detection of Classical Cipher Types with Feature-Learning Approaches* by Dalton and Stamp served as inspiration for the addition of rotor ciphers in the project, which was introduced by Maik Bastian.

It should be noted that Maik Bastian also had to incorporate an SVM classifier, since the main architectures, although able to distinguish between ACA ciphers and rotor ciphers, were not capable of effectively discriminating among the different types of rotor ciphers. The SVM was therefore specifically trained for this task and integrated as an additional module: whenever the primary model detected a rotor cipher, the final classification was performed by the SVM, resulting in higher accuracy in distinguishing between the five rotor ciphers considered.

To sum up, for cipher identification, NCID uses a combination of five different architectures: FFNN, LSTM, Transformer, RF and NBN. In the work I carried out, I had to translate from Keras to PyTorch the Feedforward Neural Network and Long Short-Term Memory, and we will now examine them in more detail.

2.2.1 FFNN

It is a subclass of ANN, in which the information flows only forward, from the input layer to the output layer, without cycles or connections that send the signal backward. It is the most classical and simple ANN model, sometimes also called MLP (Multilayer Perceptron). Figure 2.1 shows the structure of an FFNN. The aim of the training is to guide $f(x)$ to obtain the probability of belonging to a class (our label, y).

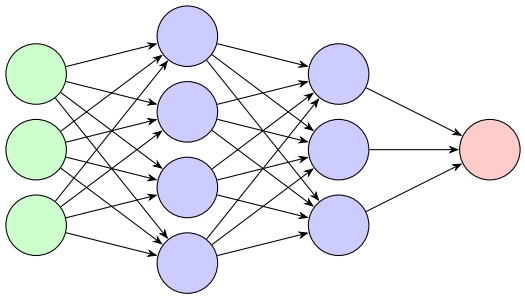

Figure 2.1: Structure of a feedforward neural network with two hidden layers (blue).

We can build deep networks with more than just two hidden layers; in fact, modern architectures often include over a hundred layers, each containing thousands of hidden layers. The term *width* refers to the number of hidden units within a single layer, while *depth* describes the number of hidden layers in the network. The total number of hidden units across all layers serves as an

indicator of the network's overall capacity.

If you are interested in exploring the topic further, I found useful consulting the paper by Goodfellow et al. [5]

2.2.2 LSTM

Long-Short Term Memory (LSTMs) is a recurrent neural network architecture designed by Sepp Hochreiter and Jürgen Schmidhuber in 1997 [8]. In contrast to FFNN, recurrent neural networks incorporate loops, enabling information from later stages of processing to be sent back to earlier stages. The LSTM has proven to be highly effective across a wide range of applications, including unconstrained handwriting recognition, speech recognition, handwriting generation, and many others [5].

To gain a clearer understanding of how the LSTM operates, its internal structure is illustrated in fig. 2.2. This graphic was designed by myself. Similar graphics can be found in Shi Yan's blog post "Understanding LSTM and its Diagrams" (2016) [9].

The LSTM architecture is built around a single core component called the memory unit, or LSTM cell. This unit comprises four neural networks, each containing an input layer and an output layer. Together, these networks form what are called the three gates: the Forget Gate, the Input Gate, and the Output Gate.

The Forget Gate determines how much of the long-term memory should be retained. This information is computed by combining the current input x_t with the previous hidden state h_{t-1}, which represents the short-term memory. A bias is then added, and the result is passed through a sigmoid activation function. The output of the Forget Gate is subsequently multiplied by the previous cell state c_{t-1}, which represents the long-term memory.

The Input Gate computes the candidate long-term memory to be added, along with the proportion of it that should be stored. This process consists of combining the current input x_t with the previous hidden state h_{t-1}, passing the result once through a sigmoid activation and once through a tanh activation. The two outputs are then multiplied element-wise, and the resulting vector is added to the cell state.

Lastly, the Output Gate updates the short-term memory by passing the combination of the current input x_t and the previous hidden state h_{t-1} through a sigmoid activation. This output is then multiplied element-wise with the tanh of the updated cell state, and the result forms the new hidden state h_t.

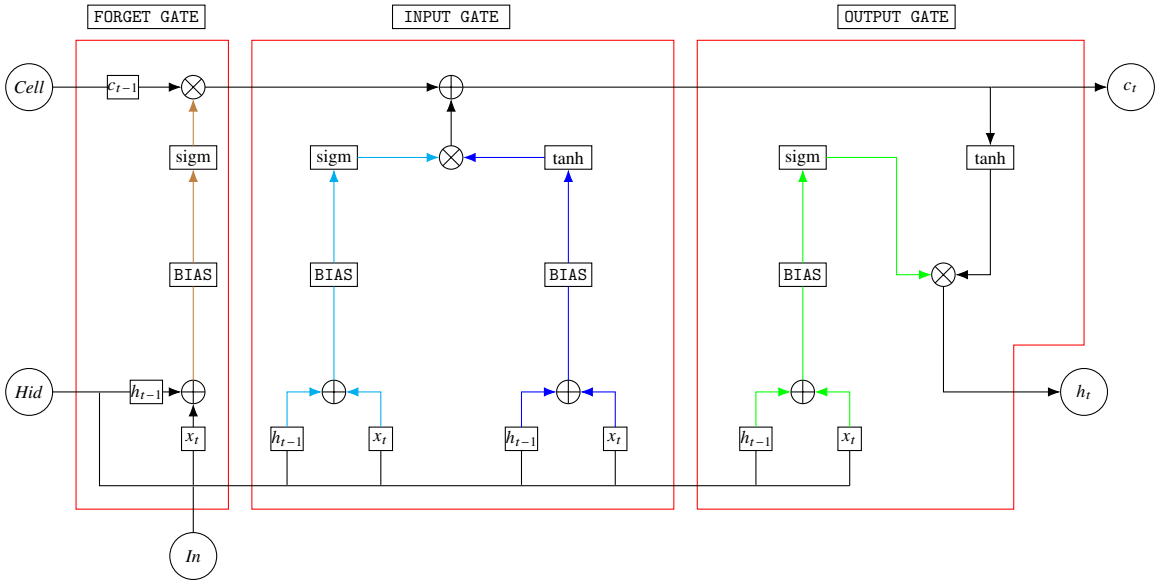

Figure 2.2: Internal structure of an LSTM unit

2.3 Training process

I will take this section to explain some fundamental concepts about ML and the project that will be useful multiple times throughout the work.

The path followed by a training process usually includes data setup, training and validation, and finally a final evaluation. In chapter 3, we will see in detail how this procedure is structured in NCID.

Let's now introduce some common Machine Learning terminology:

Weights The weights in a neural network are numerical parameters that determine how much each input influences the output. You can think of them as knobs that "weigh" the importance of each input. During training, the model searches for weight values that minimize the error between the predictions and the actual data. Therefore, the weights are not fixed: they are learned from the data.

Bias The bias in a neural network is an additional value that allows the model to shift the output independently of the inputs. It is similar to the intercept in a line $y = mx + q$, where the bias corresponds to that q. Without a bias, the output would always be constrained to pass through the origin (zero), which can limit the model's ability to fit the data. With a bias, even if all inputs are zero, the output can still take a value different from zero.

Linear layer A linear layer is used to transform the inputs into outputs through a linear combination of the values:

$$y = xW^T + b$$

where x is the input vector, i.e., the data entering the layer and W is the weight matrix. In practice, it is the basic building block of many neural networks. It serves to change the representation of the data so that the model can later recognize patterns or make predictions.

Rectified Linear Unit The activation function **ReLU** instead applies a non-linear, element-wise transformation, defined as:

$$\text{ReLU}(x) = \max(0, x)$$

In other words, negative values are replaced with zero, while positive values remain unchanged. The introduction of non-linearity is essential, as it enables the network to learn complex functions that could not be represented by linear transformations alone.

Optimizer The Adam optimizer (Adaptive Moment Estimation) is a widely used weight update algorithm in neural networks. It is a method based on computing the moments[1] of the gradient: Adam maintains an exponential estimate of the first moments (the mean of the gradients) and the second moments (the mean of the squared gradients), and uses these values to dynamically adjust the learning rate[2] of each parameter. The combination of these two moments allows for more stable and faster convergence compared to classical methods such as standard gradient descent.

Simply put: when the network makes a prediction, we compare the result with the correct answer and compute an error (loss). The optimizer 'looks' at this error and decides how to adjust the network's weights to reduce it in the next iteration.

Softmax Softmax is a mathematical function widely used in neural networks, particularly in the output layer of multi-class classification models. Its purpose is to transform a vector of real values (i.e., the logits) into a probability vector, meaning numbers between 0 and 1 that sum to 1. Therefore, softmax is used to interpret a network's outputs as probabilities, making it easier to select the most likely class.

[1] Momentum is an optimization technique that introduces a form of "inertia" in the weight update process, accumulating part of the past gradients' direction to make the descent more stable and faster.

[2] The learning rate is a hyperparameter that controls the step size during gradient descent, determining how much the model's parameters are updated in response to the estimated error at each iteration.

Loss function The loss function used in the NCID project is the **Cross-Entropy Loss**, typically employed for multiclass classification problems like ours. Cross-entropy measures the distance between the probability distribution predicted by the network and the true distribution of the labels. It performs the following steps:

1. Applies the softmax function to the logits (the raw output of the architectures) to transform them into probabilities.
2. Compares these probabilities with the true labels.
3. Returns the average log-loss[3] across all samples in the batch. The result of this computation is a scalar tensor (a vector that contains a single value), which when printed outputs a single value representing the loss.

Tokens Tokens are the smallest units of text into which a neural network splits a sentence in order to process it. In practice, tokenization transforms raw text into a sequence of numerical elements that the model can interpret. As we will see, these will be used in the LSTM (and more generally in sequential models) because they allow the network to process sequences of elements in order, preserving and learning the temporal or contextual dependencies between tokens.

Padding Padding is a technique that consists of adding values (typically zeros) to sequences of variable length to make them all the same size. This technique is used in LSTMs because each batch of data must have the same length since matrix operations require rectangular tensors.

Hyperparameters Finally, we would like to specify the parameters used as command line inputs. These will be particularly useful during the evaluation phase, since they must remain the same in order to compare the Keras models with the PyTorch ones. A quick reminder: We will talk about samples, but these depend on the context and the architectures used, as well as on the stage of the pipeline in which they are processed. For example, in the preprocessing phase, samples consist of plaintext sentences; after encryption, they become ciphered sentences, and later they are transformed into numerical features or representations suitable for the model.

The parameter `--batch_size` indicates the batch size, that is, how many samples are used in a single weight update step during training. Larger values make training more stable but more memory-intensive, while smaller values make the model noisier but allow more frequent updates.

`--train_dataset_size` specifies the total number of samples used in each fitting phase. It is important that this number is divisible by the amount of ciphers chosen with `--ciphers`, in order to obtain a dataset that has the same number of inputs for each cipher type.

`--dataset_workers` determines how many parallel processes are used to read the samples. More workers speed up the loading, which is useful with large datasets.

[3] For a deeper understanding the CrossEntropyLoss section of the PyTorch guide is available here [10]

`--epochs` defines how many times the same dataset is presented to the model during training. Increasing the number of epochs improves learning, but beyond a certain point it may lead to overfitting[4].

`--plaintext_input_directory` is the folder that contains the plaintexts used as a basis for creating ciphertexts. These plaintexts are used to generate the training data (see section 3.1).

`--model_name` defines the name of the final saved model file, which must have the `.h5` extension for Keras and `.pth` for PyTorch.

`--min_train_len` and `--max_train_len` set the minimum and maximum length, respectively, of plaintexts used for training. If the value is set to `-1`, no constraint is applied. The same logic applies to `--min_test_len` and `--max_test_len`, which refer instead to the data for the evaluation.

`--architecture` defines the model architecture to be used. Combinations are also possible, such as `[FFNN,NB]` or `[DT,ET,RF,SVM,kNN]`, as well as a special case for rotor ciphers (`SVM-Rotor`).

2.4 Statistical features

Features are numerical measures that can be computed from the ciphertext, highlighting certain regularities. These regularities arise from the fact that the encrypted message retains statistical correlations with the original language (English, Italian, etc.). Essentially, statistical features are tools to quantify how much a ciphertext deviates from random text and how much it "resembles" natural language text. These differences correspond to the known weaknesses of the ACA ciphers and allow them to be identified and attacked.

One of the most widely used features in cryptanalysis is the Index of Coincidence (IoC), which is the probability that two randomly chosen letters from a text are the same. In a natural language like English, where some letters are much more frequent than others, the IoC takes on a characteristic value of around 0.066. In completely random text, the IoC is much lower, approximately 0.038.

Using substitution or transposition ciphers, this value remains unchanged, because the letters are simply rearranged. Consequently, an IoC that remains close to that of natural language is a strong indication that the cipher belongs to one of these families.

In contrast, for ciphers like the Bacon cipher, the frequency distribution changes drastically. Here, the IoC no longer reflects that of the original language and may appear anomalous or much higher. Therefore, if the model observes an IoC value that is particularly high or low compared to the typical English range, it can immediately rule out a simple transposition or substitution. In this sense, the IoC becomes a discriminative feature: it allows distinguishing between cipher families

[4] Overfitting is a phenomenon that occurs when a machine learning model learns the training data too well, memorizing its noise and specific details instead of capturing general patterns.

that preserve the statistical structure of the language and others that alter it significantly. It is through this type of analysis that the model can narrow down the possible ciphers used.

The project, as cited in the paper by Kopal et al. [6], initially implemented 28 statistical features, divided into groups: distribution statistics, frequency statistics, binary features, and cipher-specific features. After a selection phase, not all 28 features proved useful: only 20 were actually used in the final models, because some did not improve accuracy or introduced noise, meaning that a feature adds no new information and is redundant relative to others already present, leading to incorrect classification. If you are interested in seeing all the features used in the project, they can be found in [6].

2.5 Keras vs PyTorch

In the Machine Learning landscape, selecting the right framework is crucial for achieving desired results. Keras [11] and PyTorch [12] are the most popular open-source frameworks (along with TensorFlow, used for large-scale machine learning and deep learning applications; Scikit-learn, used for traditional machine learning algorithms and data preprocessing; and NumPy, used for efficient numerical computations and array manipulation), both used to train and evaluate neural networks. Despite sharing a common goal, they differ in their design, making them suitable for different use cases. This section aims to underline the main differences between Keras (introduced in 2015 by François Chollet) and PyTorch (released in 2016 by Meta AI, formerly Facebook AI Research) by analyzing their principal functionalities and their respective strengths and weaknesses. In the implementation chapter, we examine the actual translation from the Keras version to the PyTorch one.

So, why translating a code from Keras to PyTorch?

There are several reasons that make the conversion of a model from Keras to PyTorch an essential modification. Firstly, there is the desire to experiment with a new framework or the need to adapt to the demands of the job market. In fact, Keras has declined in popularity in the last few years among working environments, since PyTorch offers more flexibility for coding training loops. The key differences between Keras and PyTorch reside in their architectures, how they define and train models, and the level of control they provide to the coder over those processes.

Keras presents itself as a modular library designed for fast training and experimentation with deep neural networks. Its architecture favors quick and easy prototyping, offering straightforward "building blocks". It is considered a "plug-and-play" framework that allows the coder to quickly build, train, and evaluate models, simplifying the development process.

PyTorch, on the contrary, is a low-level framework. It offers a more flexible approach and an imperative programming style, where network behavior is defined directly in Python code.

But how do we train and evaluate a model with those frameworks?

In Keras, training and evaluating a model is fairly easy, since the whole training process is managed by the `fit` method. For example, in the Keras implementation the FFNN was implemented like this:

```
history = model.fit(statistics, labels,
                                    batch_size=args.batch_size,
                                    validation_data=(val_data, val_labels),
                                    epochs=args.epochs,
                                    callbacks=[early_stopping_callback,
                                            tensorboard_callback,
                                            custom_step_decay_lrate_callback,
                                            checkpoint_callback])
```

Listing 2.1: Keras fit implementation

In PyTorch, instead, the architecture is divided into two parts: the definition of the states and the definition of the path that the data has to follow during the training. Training and evaluation require manual writing of complete training and test loops. This must include the upload of data inside batches, the passage of data through the network, loss calculation, backpropagation, and weight updates. [13]

Without delving into much theory, in pseudo code a neural network training loop looks like this: [14]

Algorithm 1 Neural network training loop

```
for each epoch do
    for each batch do
        Load data
        Pass data through network
        Calculate losses
        Calculate gradients
        Adjust network weights through backpropagation
    end for
end for
```

At this point you are probably wondering: Does PyTorch really expect me to write all that code? The answer is yes, but as we said, it comes with benefits for the programmer who wants to experiment with Machine Learning.

In summary, the decision between Keras and PyTorch should be driven by your project goals and your familiarity with deep-learning concepts. If you need to prototype quickly or teach fundamental models, Keras is likely the better fit; if you require research-grade flexibility, and fine-grained control, PyTorch will better meet those demands. If you are eager to know what can PyTorch accomplish I suggest you to read the implementation section 3.3.3 about the validation steps.

3 Implementation

In this chapter, we examine the code modifications that were introduced, explaining both their purpose and the rationale behind each change. Figure 3.1 shows the training process of the cipher classification model. These are the fundamental steps performed by the code to create a new model, starting from the extraction of texts from the Gutenberg Library and ending with the evaluation of the trained model. In the following paragraphs, each of these steps will be analyzed in detail, with particular emphasis on the `fit` (i.e., the actual training) and validation stages, as they required the most significant modifications.

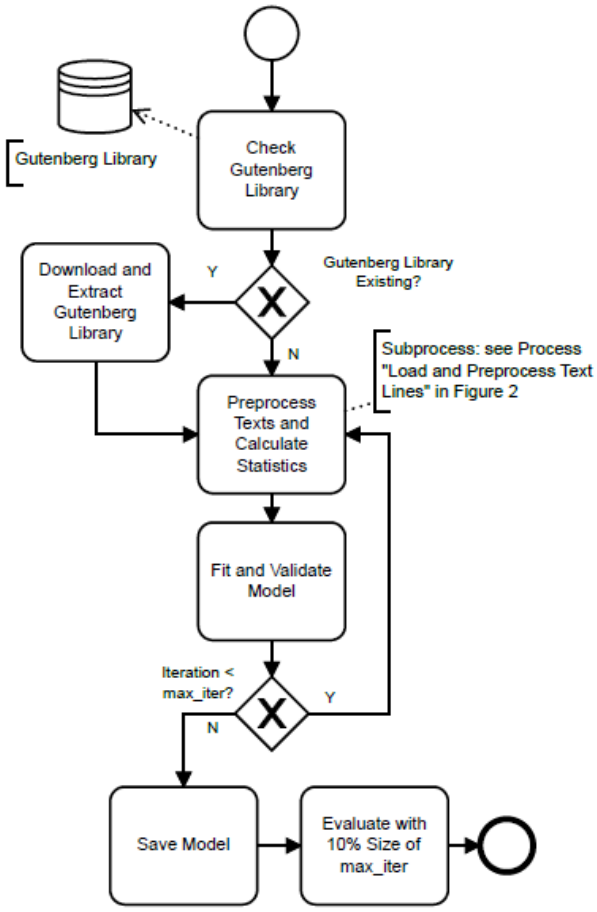

Figure 3.1: Training process [6]

3.1 Gutenberg library

The initial step involves extracting plaintext. For this purpose, we utilize the Gutenberg Library, an open-source collection of public domain texts, which provides a diverse set of literary works suitable for training and testing our models. These texts serve as the basis for generating the encrypted input used in our supervised learning tasks.

In the previous Keras-based implementation, the procedure for handling the Gutenberg plaintext dataset was the same as in the PyTorch version. The dataset was downloaded from the same public Google Drive link, automatically extracted into a temporary directory, and then relocated to the project's designated plaintext folder (`data/gutenberg_en`). After the extraction, auxiliary directories created by the download manager were removed, leaving only the necessary dataset in the target directory.

This design choice was not altered during the transition from Keras to PyTorch because the handling of plaintext data is a preprocessing step independent of the training framework. Both frameworks rely on the same input format: plaintext files are required to generate ciphertext samples through encryption with the implemented classical ciphers, which are then converted into statistical feature vectors or character sequences for model training. Consequently, the data pipeline, from downloading and extracting the Gutenberg texts to splitting them into training and testing subsets, remained unchanged.

3.2 Preprocess texts and calculate statistics

In this phase, the plaintexts are first encrypted using different cryptographic algorithms and then preprocessed to be transformed into a numerical form suitable for training machine learning models. The preprocessing consists of text normalization (for example, converting all letters to lowercase), the removal or replacement of unrecognized symbols, and the conversion of character sequences into numerical vectors. In summary, this phase makes it possible to translate the encrypted texts into standardized numerical representations, making them comparable to one another and enabling the models to learn the distinctive patterns of the different ciphers.

Like the previous plaintext extraction phase, this step did not require any changes, since the data are preprocessed in the same way for both Keras and PyTorch. The only precaution is to provide them, during training, in the appropriate format: tensors for PyTorch and NumPy arrays for Keras.

3.3 Fit and validate model

Let's now move on to the major changes.

Initially, I tried to understand how the GitHub repository was formed [1]. The file that we will examine now is called `train.py`, so every modification from now on (page 25 to 43) has been

done on that file. I was first focused on understanding how the training sessions worked, which I did by testing and running some basic training scripts on my local machine. These experiments were mainly aimed at grasping the overall workflow of the code.

Once I understood the overall functioning, I began working on the changes related to the training and validation of the model. In order to translate the code from Keras to PyTorch there are several logical steps to follow.

3.3.1 Imports

The first step consisted of importing the `PyTorch` library along with all the necessary modules required to run the code.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchinfo import summary
import numpy as np
from torch.utils.data import TensorDataset, DataLoader
```

Listing 3.1: Imports for PyTorch

These include essential components such as `torch.nn` for defining neural networks, `torch.optim` for optimization algorithms, and `torch.utils.data` for managing datasets and data loaders. Additionally, `torchinfo` was imported to provide a summary of the model's structure, and `NumPy` was used for numerical operations and compatibility with existing data structures.

We will now delve into the details of the individual architectures.

3.3.2 FFNN PyTorch class

As anticipated in the theory chapter I personally worked on and translated the FFNN and LSTM (see section 2.2), and in the following sections we will examine in detail the code modifications that were introduced in order to use them with the new framework.

Let's start from the FFNN class.

```
class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
        ↪ num_hidden_layers):
        super().__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_hidden_layers = num_hidden_layers

        layers = [nn.Linear(input_size, hidden_size), nn.ReLU()]
        for _ in range(num_hidden_layers - 1):
```

```
            layers += [nn.Linear(hidden_size, hidden_size), nn.ReLU()]
        layers.append(nn.Linear(hidden_size, output_size))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

Listing 3.2: Definition of the TorchFFNN class

To gain an intuitive understanding of how the network works, let us consider a simplified example. Suppose we have an FFNN model with three input neurons, two hidden layers of five neurons, and an output layer with two possible classes (for instance, distinguishing whether a text belongs to cipher A or cipher B, so a binary classification).

As starting data, we take a small batch consisting of four vectors, each with three numerical features:

```
model = FFNN(input_size=3, hidden_size=5, output_size=2, num_hidden_layers=2)

Input:
tensor([[ 0.1000,  0.2000,  0.3000],
        [ 1.0000, -1.0000,  0.5000],
        [ 0.0000,  0.0000,  0.0000],
        [ 2.0000,  1.0000, -1.0000]])
```

Listing 3.3: Example of input

At this stage, the input is multiplied by the weight matrix of the first layer and added to a bias term. The result is a linear transformation that projects the data into the space of the hidden layer:

```
tensor([[ 0.0317,  0.2515, -0.1857,  0.0923, -0.0142],
        [ 0.6049, -0.7834,  0.1521, -0.4418,  0.2357],
        [ 0.0005,  0.0003, -0.0007,  0.0002, -0.0001],
        [ 0.3672,  0.8550, -0.6538,  0.3214, -0.1579]])
```

Listing 3.4: First linear layer (Linear input → hidden)

ReLU (see section 2.3) replaces negative values with zero:

```
tensor([[0.0317, 0.2515, 0.0000, 0.0923, 0.0000],
        [0.6049, 0.0000, 0.1521, 0.0000, 0.2357],
        [0.0005, 0.0003, 0.0000, 0.0002, 0.0000],
        [0.3672, 0.8550, 0.0000, 0.3214, 0.0000]])
```

Listing 3.5: ReLU activation function

The linear transformation and the ReLU are then applied again until the final output state is obtained. The last layer maps the data into a vector that has as many entries as the number of classes:

```
tensor([[-0.1613,  0.0144],
        [ 0.0568, -0.1409],
        [-0.1585,  0.0267],
```

```
4         [-0.0652, -0.0819]])
```

Listing 3.6: Example of output

These values are not probabilities, but logits. In other words, they are the ”raw” outputs produced by the last linear layer of your FFNN before any transformation that normalizes them between 0 and 1. Each row corresponds to a batch example, and each column corresponds to a class. These logits can be positive or negative and are not constrained to sum to 1. When you apply the softmax function to each row, you obtain a probability vector that sums to 1, which is useful for interpreting the model and for computing the cross-entropy loss (in PyTorch, cross-entropy loss can accept logits directly, see section 2.3).

This simple example demonstrates how an initial set of numerical data, through a sequence of linear and non-linear transformations, is projected into a space that allows separation between different types of ciphers. In practice, the same logic is extended to more complex features and a larger number of classes, enabling NCID to recognize a wide range of ciphers.

An important aspect of the network implementation concerns the construction of the sequence of layers. In the code, the various layers of the network are initially stored in a Python list called `layers`. This list contains, in order: the linear transformation from the input to the hidden layer, the ReLU activation function, any additional pairs of linear layer and ReLU, and finally the output layer.

To transform this list into a proper PyTorch model, the `nn.Sequential` class is used, which allows multiple modules (`nn.Module`) to be concatenated into a single structure. In this way, the object `self.net` represents the entire neural network as an ordered sequence of operations.

The `*layers` syntax serves to “unpack” the list and pass its individual elements as arguments to `nn.Sequential`.
For example, if `layers = [A, B, C]`, then the statement `nn.Sequential(*layers)` is equivalent to `nn.Sequential(A, B, C)`.

3.3.3 FFNN train function

In this section, we will dive into the analysis of the lines of code that allow the neural network to learn, starting from setting the optimizer and the loss function, up to performing an evaluation.

Setting optimizer and loss function Let’s go through a detailed examination:

```
1 def train_torch_ffnn(model, args, train_ds):
2     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3     model.to(device)
4
5     optimizer = optim.Adam(
6         model.parameters(),
7         lr=config.learning_rate,
```

```
        betas=(config.beta_1, config.beta_2),
        eps=config.epsilon,
        amsgrad=config.amsgrad
    )
    criterion = nn.CrossEntropyLoss()
```

Listing 3.7: Adam optimizer and Loss function

After defining the optimizer and the loss function (see section 2.3), the code sets the model to training mode using the command `model.train()`.

This instruction is crucial because it tells PyTorch that the model should behave as it does during training: in particular, certain layers such as *Dropout* (see the following paragraph) and *Batch Normalization* (see chapter 4) behave differently depending on whether the model is in training or validation/testing mode.

Dropout for example randomly deactivates, with a given probability (for example 50%), certain neurons at each forward pass during training to reduce overfitting. This prevents the network from relying too heavily on specific connections.

Without the `model.train()` call, the model would use the parameters fixed for evaluation mode, which could compromise the training process.

Immediately before the training loops, several control variables are also initialized:

- `best_val_acc = 0`: stores the highest accuracy achieved on the validation set so far, necessary for implementing *early stopping* (we will see it later in this section).
- `patience_counter = 0` and `patience_limit = 250`: used to count how many consecutive iterations do not improve performance. If the limit is exceeded, training is stopped to avoid wasting computational resources.
- `train_iter = 0`: keeps track of the total number of samples processed during training.
- `train_epoch = 0`: counts the number of completed epochs.
- `start_time = time.time()`: allows measuring the total duration of the training process.
- `val_data_created = False` together with `x_val` and `y_val`: ensures that the validation dataset is created only once, at the first iteration, and then reused without generating random splits at each cycle.

These assignments therefore serve to initialize the training state, so that the model can be trained, monitored, and evaluated correctly.

Training loop The code then enters the actual training phase (see listing 3.8). The outer loop `for epoch in range(args.epochs)` iterates over the number of epochs (see section 2.3). Inside, the loop `while train_ds.iteration < args.max_iter` ensures that training continues until the maximum number of iterations is reached.

The command `training_batches = next(train_ds)` retrieves a new batch of data from the generator `train_ds`. Each batch is organized as pairs of `(statistics, labels)`, where `statistics` contains the numerical features extracted from the ciphered texts i.e., the values that will be fed as input to the model while `labels` represents the corresponding class labels.

The statement `statistics, labels = training_batch.items()` separates the data from the labels. With `stats_np = statistics.numpy()` and `labels_np = labels.numpy()`, the PyTorch tensors are converted into NumPy arrays, which is useful for later splitting the dataset into training and validation sets using `train_test_split`.

For clarification, consider a numerical example: suppose the generator `train_ds` provides a batch with 4 samples, each described by 3 statistical features, along with their respective class labels. We would then have:

$$\texttt{statistics} = \begin{bmatrix} 0.2 & 0.5 & 0.7 \\ 0.1 & 0.4 & 0.3 \\ 0.9 & 0.8 & 0.6 \\ 0.0 & 0.2 & 0.1 \end{bmatrix}, \quad \texttt{labels} = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 1 \end{bmatrix}$$

After conversion with `.numpy()`, we obtain the arrays

$$\texttt{stats\_np} = \begin{bmatrix} 0.2 & 0.5 & 0.7 \\ 0.1 & 0.4 & 0.3 \\ 0.9 & 0.8 & 0.6 \\ 0.0 & 0.2 & 0.1 \end{bmatrix}, \quad \texttt{labels\_np} = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 1 \end{bmatrix}$$

Although they appear identical, these arrays can now be used either to create the PyTorch tensors required for training or to be split into a *training set* and a *validation set*.

In this way, the model receives as input the statistics and the corresponding labels, which allows the loss function to be computed and the weights to be updated via backpropagation.

```
    for epoch in range(args.epochs):
        while train_ds.iteration < args.max_iter:
            training_batches = next(train_ds)
            for training_batch in training_batches:
                statistics, labels = training_batch.items()
                stats_np = statistics.numpy()
                labels_np = labels.numpy()
```

Listing 3.8: Loop structure, code lines from train.py, see appendix from 155 to 161

Splitting dataset In the following lines of `train.py`, the division of data into training and validation sets is handled. At the first pass, the variable `val_data_created` is set to `False`, so the function `train_test_split(stats_np, labels_np, test_size=0.3)` is executed, which splits the data into two parts: 70% for training (`x_train_np`, `y_train_np`) and 30% for validation (`x_val_np`, `y_val_np`).

Subsequently, the validation data are converted into PyTorch tensors and moved to the selected device (CPU or GPU).

After this initial split, the variable `val_data_created` is set to `True`, so in subsequent iterations the split is not repeated and the validation data remain fixed, while the set `x_train_np`, `y_train_np` is updated with the next batches provided by the dataset.

Suppose that the initial batch consists of four samples, each with three features, along with their corresponding labels:

```
stats_np = [[0.2, 0.5, 0.7],
            [0.1, 0.4, 0.3],
            [0.9, 0.8, 0.6],
            [0.0, 0.2, 0.1]]

labels_np =[1, 0, 2, 1]
```

Listing 3.9: Batches before being splitted

By splitting with `test_size=0.3`, a possible outcome could be:

```
x_train_np = [[0.2, 0.5, 0.7],
            [0.1, 0.4, 0.3],
            [0.0, 0.2, 0.1]]

y_train_np = [1, 0, 1]

x_val_np = [[0.9, 0.8, 0.6]]

y_val_np = [2]
```

Listing 3.10: Batches after being splitted

In this example, three samples are used for training and one for validation. After conversion to PyTorch tensors, `x_val` and `y_val` will remain unchanged until the end of training, ensuring that the model's performance is always evaluated on the same reference data.

```
if not val_data_created:
    x_train_np, x_val_np, y_train_np, y_val_np = train_test_split(stats_np,
        ↪ labels_np, test_size=0.3)
    x_val = torch.tensor(x_val_np, dtype=torch.float32).to(device)
    y_val = torch.tensor(y_val_np, dtype=torch.long).to(device)
    val_data_created = True
else:
    x_train_np = stats_np
```

```
8     y_train_np = labels_np
```

Listing 3.11: Splitting the data between trainin and validation

Optimization of the gradients Now we move on to a very important part of the training, the optimization of the gradients that is the process of updating the weights of a neural network during training. The step we are going to analyze was initially implemented differently and produced poorly performing models, but thanks to the help of Maik Bastian, I was able to solve it. Below is a snippet of the code I had first implemented in listing 3.12:

```
1
2 optimizer.zero_grad()
3 outputs = model(x_train)
4 loss = criterion(outputs, y_train)
5 loss.backward()
6 optimizer.step()
```

Listing 3.12: Wrong implementation

The issue with these lines is that, regardless of the dataset size, the optimization is performed only once, whereas in the Keras implementation it was applied for every batch. The problem of calling the optimizer only once for the entire dataset (i.e., after processing all the samples of that batch) instead of after each batch is that the model performs too few weight updates. For example, if you have a dataset of 10k samples and process it as a single batch, the optimizer is updated only once every 10,000 samples. In practice, the network executes just one averaged backpropagation over the whole block, causing the model to learn much more slowly. This problem is known as a *larger effective batch size*.

To achieve behavior similar to Keras (mini-batches and more frequent updates), you should manually split x_{train} and y_{train} into mini-batches of size `args.batch_size`, as illustrated in listing 3.14.

In listing 3.13 is shown an example of the flow of data with the right implementation where we have a binary classification task. I need to emphasize to the reader that this example is not meant to follow precise calculations, but rather to understand the correct way to split the data. For this reason, the input data are represented by randomly chosen integers, as are the outputs of the loss function:

```
1
2 x_train_np = [
3     [1, 2],
4     [2, 1],
5     [3, 4],
6     [4, 3],
7     [5, 6],
8     [6, 5]
9 ]
10
11 y_train_np = [0, 0, 1, 1, 0, 1]
12
13 # Batch 1
```

```
x_batch1 = [[2, 1], [4, 3]]
y_batch1 = [0, 1]

# Batch 2
x_batch2 = [[5, 6], [1, 2]]
y_batch2 = [0, 0]

# Batch 3
x_batch3 = [[3, 4], [6, 5]]
y_batch3 = [1, 1]

outputs_batch1 = [[1.0, 0.5], [0.2, 1.2]]
outputs_batch2 = [[0.8, 0.4], [1.1, 0.3]]
outputs_batch3 = [[0.3, 0.7], [0.1, 1.0]]

loss_batch1 = 0.6
loss_batch2 = 0.5
loss_batch3 = 0.4

batch_losses = [loss_batch1, loss_batch2, loss_batch3]

epoch_loss = sum(batch_losses) / len(batch_losses)
print("Average Loss:", epoch_loss)  # Output: 0.5
```

Listing 3.13: Example of right implementation

With the wrong implementation (see listing 3.12, `optimizer.step()` would update all the weights only once, using the gradients computed over all 6 samples together. Thus, to achieve this goal, it is not necessary to change the previous steps where optimization was performed, but simply to add a for loop that does it for each `x_batch` and `y_batch` contained in the `train_loader`. The lines before the for loop are used to prepare the data as PyTorch tensors and to create a PyTorch dataset that associates each input with its corresponding label:

```

x_train = torch.tensor(x_train_np, dtype=torch.float32)
y_train = torch.tensor(y_train_np, dtype=torch.long)

train_dataset = TensorDataset(x_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=
    ↪ True)

batch_losses = []

for x_batch, y_batch in train_loader:
    x_batch = x_batch.to(device)
    y_batch = y_batch.to(device)

    optimizer.zero_grad()
    outputs = model(x_batch)
    loss = criterion(outputs, y_batch)
    loss.backward()
    optimizer.step()

```

```
20      batch_losses.append(loss.item())
21      train_iter += len(y_batch)
22
23  epoch_loss = sum(batch_losses) / len(batch_losses)
```

Listing 3.14: Right implementation, see lines from 175 to 194 of the appendix

Evaluation The next step of our training loop consists of computing predictions using the model that we are training, see listing 3.15

```
199
200 model.eval()
201 with torch.no_grad():
202     val_outputs = model(x_val)
203     val_loss = criterion(val_outputs, y_val)
204     val_pred = torch.argmax(val_outputs, dim=1)
205     val_acc = (val_pred == y_val).float().mean().item()
206
207     top3 = torch.topk(val_outputs, k=3, dim=1).indices
208     y_val_exp = y_val.unsqueeze(1).expand_as(top3)
209     val_k3 = (top3 == y_val_exp).any(dim=1).float().mean().item()
```

Listing 3.15: Evaluation 8.1

The command `model.eval` in PyTorch is used to set the model into evaluation mode. By switching to `model.eval`, we inform PyTorch that training is no longer being performed, but rather validation or testing, and that the learned parameters should be used while disabling stochastic components. By stochastic components, we refer to mechanisms that introduce randomness during training, such as Dropout. In evaluation mode, Dropout is disabled and all neurons remain active. The context manager `torch.no_grad` in PyTorch specifies that, within its block of code, gradients should neither be computed nor stored. During training, PyTorch keeps track of all operations performed on tensors by building a computational graph, since at the end gradients are required through the `loss.backward()` operation. This process, however, consumes memory and involves additional computations. During validation or inference (that is, when using `model.eval`), gradients are not needed, as only the model predictions are of interest.

We can now examine how predictions are computed. For this purpose, we will consider an example with `train_dataset_size = 10` (see section 2.3), a total of 7 features, and only 5 cipher classes. The command `model(x_val)` executes the forward pass, where `x_val` is the input batch, a tensor containing the calculated feature values (see section 2.4) of the validation set for each sample given.

In this example, the input tensor has the shape `x_val.shape = (3, 7)`. It should be emphasized, however, that in the actual project a total of 724 feature values are employed. To see where in the code this is defined, refer to the appendix at line 504. As you can notice, the number 724 is assigned to the variable `input_layer_size`. This is because each neuron in the input layer corresponds to one feature.

An example of `x_val` and the corresponding `val_outputs` after the forward pass is shown below:

```
x_val =
tensor([[0.12, 0.05, 0.33, 0.21, 0.08, 0.17, 0.02],
        [0.07, 0.18, 0.22, 0.10, 0.14, 0.05, 0.04],
        [0.30, 0.11, 0.09, 0.25, 0.16, 0.08, 0.12]])
x_val.shape = torch.Size([3, 7])
```

The result of the forward pass is a tensor called `val_outputs` with shape (3, 5), since in this example 5 ciphers are used, while in the full project there are 61 of which 56 were implemented in the paper "A Massive Machine-Learning Approach For Classical Cipher Type Detection Using Feature Engineering" [15], and the 5 rotor ciphers were later implemented thanks to the contribution of Dalton and Stamp's paper [3].

```
val_outputs =
tensor([[ 0.80,  1.90, -0.30,  0.20,  0.50],
        [ 2.20, -0.10,  0.70,  1.50, -0.40],
        [-0.20,  0.30,  2.00,  0.10,  1.20]])
val_outputs.shape = torch.Size([3, 5])
```

This tensor will then be employed to compute both the predictions and the loss. The loss is calculated by applying the cross-entropy loss function. The result of this computation is a scalar tensor, `val_loss`, which when printed outputs a single value representing the loss.

As for the predictions, the `val_outputs` tensor is used again, this time combined with the `argmax` function to obtain the index of the maximum value along dimension 1:

```
val_pred = tensor([1, 0, 2])
```

Once the predictions are obtained, they can be compared with the true labels in order to compute the accuracy, which is stored in the variable `val_acc`. This value indicates how well the model is classifying the validation data. The process is as follows:

1. `(val_pred == y_val)` compares the predictions with the true labels (`y_val`), returning a vector of booleans (True if correct, False if incorrect).
2. `.float().mean()` converts the booleans to numerical values (1.0 for correct, 0.0 for incorrect) and computes the mean.
3. `.item()` extracts the value as a standard Python number.

For example, if `y_val = tensor([1, 3, 2])`, then `val_acc` will be 0.67, since only 2 out of 3 samples were predicted correctly. Thus, the model achieves an accuracy of 67%.

To compute the top-3 accuracy, which measures the percentage of times the correct class appears among the three highest predictions (rather than only the top-1), the `topk` function selects the three highest logits for each sample, producing a tensor with shape `(batch_size, 3)`, which contains the indices of the three most probable classes:

```
top3 =
[[1, 0, 4],
 [0, 3, 2],
 [2, 4, 1]]
```

After that, the command `y_val_exp = y_val.unsqueeze(1).expand_as(top3)` is used to make the dimensions of the true labels (`y_val`) and the multiple predictions comparable, since while `y_val` has shape (3,), `top3` has shape (3, 3). The expansion produces the following tensor:

```
y_val_exp =
tensor([[1, 1, 1],
        [3, 3, 3],
        [2, 2, 2]])
```

Finally, similarly to `val_acc`, the predictions are compared with the true labels using:

```
val_k3 = (top3 == y_val_exp).any(dim=1).float().mean().item()
```

which results in:

```
tensor([[ True, False, False],
        [False,  True, False],
        [ True, False, False]])
```

Therefore, in this example `val_k3 = 1`, corresponding to 100%.

In this example, one can observe the advantage of PyTorch over Keras. For instance, the line `val_loss = criterion(val_outputs, y_val)` shows how PyTorch allows for full flexibility in defining the loss function (here named `criterion`), not only by choosing among the standard implementations (such as `CrossEntropyLoss`) but also by creating entirely customized loss functions and directly integrating them into the training loop. This feature makes it possible to adapt the model to highly specific and complex scenarios, where a traditional error metric would not be sufficient. In Keras, by contrast, the choice of the loss function is typically limited to a predefined set of options (such as `Poisson`, `binary_crossentropy` for binary classification, or `mean_squared_error` for regression problems) or otherwise constrained by the high-level APIs.

Early stopping check The last addition was simply an early stopping check. In Keras, these types of checks are already available through the `callbacks` module, such as `EarlyStopping`, which can monitor a specified metric and stop training when it ceases to improve. In PyTorch, however, we need to implement them manually. This particular check is used to prevent overfitting by stopping the training process once the model's performance on the validation set stops improving, thus saving computational resources.

```
# --- Early stopping check ---
if val_acc > best_val_acc:
    best_val_acc = val_acc
    patience_counter = 0
else:
    patience_counter += 1
    if patience_counter >= patience_limit:
        print("Early stopping triggered.")
        elapsed = time.time() - start_time
        t = time.gmtime(elapsed)
        print(f"Finished training in {t.tm_yday - 1} days {t.tm_hour} hours {
            ↪ t.tm_min} minutes {t.tm_sec} seconds with {train_iter}
            ↪ iterations.")
        class DummyEarlyStopping:
            stop_training = True
        return DummyEarlyStopping(), train_iter, f"Early stopped at epoch {
            ↪ epoch+1}"
```

Listing 3.16: Early stopping check

Although functional, we preferred not to use this check during the runs on the DGX machine (see chapter 4), in order to assess the model's true limits. By allowing the training to continue without early stopping, we could observe the maximum performance the model can achieve and better understand how it behaves when trained for extended periods.

And with that we end the training loop.

3.3.4 FFNN prediction function

After training and saving a model, we need to evaluate whether it accurately recognizes our ciphers. To this end, I implemented the prediction function, which evaluates the fully trained model on a dedicated test dataset, without updating the weights anymore.

Similar to the snippet observed in the evaluation section (see section 3.3.3), this function also employs `model.eval()` and `with torch.no_grad()` to disable dropout and gradient computation, ensuring stable predictions without additional memory costs. Indeed, this function also makes use of Cross Entropy Loss (see section 2.3), `torch.argmax`, `torch.topk()`, etc.

However, we are now in a different stage, namely the final testing phase which is computed by the FFNN prediction function (see lines from 355 to 393 of appendix). In the previous case, after validation, the model was switched back to `train()` mode to continue learning, while in this function the model remains in `eval()` mode for the entire test process.

Moreover, an iterative test dataset (`test_ds`) is used, which provides batches until `args.max_iter` is reached, instead of a fixed validation set created by splitting the training data. In this context, an essential aspect is that, in addition to loss and metrics, all predictions (`all_preds`) and all labels (`all_labels`) are stored and then returned by the function for subsequent calculations (e.g., confusion matrix and F1-score, which will be discussed later in chapter 4).

```
def predict_torch_ffnn(model, test_ds, args):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()
    model.to(device)
    criterion = nn.CrossEntropyLoss()

    all_preds = []
    all_labels = []

    with torch.no_grad():
        while test_ds.iteration < args.max_iter:
            testing_batches = next(test_ds)
            for testing_batch in testing_batches:
                statistics, labels = testing_batch.items()
                stats_np = statistics.numpy()

                x = torch.tensor(stats_np, dtype=torch.float32).to(device)
                y = torch.tensor(labels.numpy(), dtype=torch.long).to(device)

                outputs = model(x)
                loss = criterion(outputs, y)

                pred_top1 = torch.argmax(outputs, dim=1)
                acc = (pred_top1 == y).float().mean().item()

                top3 = torch.topk(outputs, k=3, dim=1).indices
                y_expanded = y.unsqueeze(1).expand_as(top3)
                k3_acc = (top3 == y_expanded).any(dim=1).float().mean().item
                    ↪ ()

                print(f"Eval -> Loss: {loss.item():.4f}, Accuracy: {acc:.4f},
                    ↪  Top-3 Accuracy: {k3_acc:.4f}")

                preds = torch.softmax(outputs, dim=1).cpu().numpy()
                all_preds.append(preds)
                all_labels.append(labels.numpy())

    all_preds = np.concatenate(all_preds, axis=0)
    all_labels = np.concatenate(all_labels, axis=0)

    return all_preds, all_labels
```

Listing 3.17: Custom predict function for FFNN

3.3.5 LSTM PyTorch class

It is now time to discuss our second architecture, namely the LSTM. The main difference between the two architectures lies in the class definition, that is, how the function maps inputs to outputs. In fact, the structure of the LSTM class is very different from that of the FFNN, while the training process, as we will see, is identical for both.

Let us start by examining the inputs received by the class:

The `LSTM` class defines a recurrent neural network model based on LSTM. The parameter `vocab_size` indicates the size of the vocabulary, i.e., the total number of distinct tokens (see section 2.3) present in the input data, and it is used to build the initial *embedding layer*. The embedding dimension is specified by `embed_dim`, which determines how rich the representation of each token will be: larger embeddings can capture more detailed information but also increase the model's complexity. The parameter `hidden_size` represents the number of hidden units in the LSTM and defines the network's capacity to store long-term information. `output_size` indicates the number of neurons in the final layer, corresponding to the number of classes in the classification problem or the desired output dimension. The parameter `num_layers` establishes the number of stacked LSTM layers, while `dropout` specifies the dropout probability between LSTM layers, useful to reduce overfitting when multiple layers are used.

```
class LSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size, output_size,
        ↪ num_layers=1, dropout=0.0):
        super().__init__()

        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers
        self.dropout = dropout

        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embed_dim,
            padding_idx=0
        )
```

Listing 3.18: Input LSTM class

Within the class, the LSTM layer itself is defined using `nn.LSTM`. This PyTorch function is used to define the structure of our architecture, to which the layers and parameters such as dropout are passed. After the LSTM layer, a linear layer (`self.fc`) is defined, which takes the hidden state of the LSTM as input and produces the final output of dimension `output_size`, corresponding to the classes of the classification problem or the desired output dimension.

```
        self.lstm = nn.LSTM(
            input_size=embed_dim,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0.0
        )
        self.fc = nn.Linear(hidden_size, output_size)
```

Listing 3.19: Input LSTM class

The `forward` method defines the flow of data through the LSTM model. Initially, the dimension of the input tensor `x` is checked. If `x` has three dimensions and the last dimension is equal to 1, it is removed using `x.squeeze(2)` to ensure that the input has the correct shape for the embedding layer.

Next, the input tokens are converted into dense vectors, that is a vector where every element contains a significant information, through the embedding layer defined as `self.embedding(x)`, producing the tensor `emb`. This tensor is then passed to the LSTM layer, `self.lstm(emb)`, which returns two main values: `output`, containing the hidden states for each time step, and `hidden`, representing the last hidden state of each LSTM layer.

To obtain a compact representation of the entire sequence, the last hidden state of the final layer is selected, `last_hidden = hidden[-1]`. This vector summarizes the sequential information learned by the LSTM. Finally, `last_hidden` is passed through the linear layer `self.fc` to obtain the `logits`.

```
    def forward(self, x):
        if x.dim() == 3 and x.size(2) == 1:
            x = x.squeeze(2)

        emb = self.embedding(x)
        output, (hidden, _) = self.lstm(emb)
        last_hidden = hidden[-1]
        logits = self.fc(last_hidden)

        return logits
```

Listing 3.20: Forward method LSTM class

3.3.6 LSTM train function

The LSTM training function in `train.py` is generic with respect to the type of model: for both FFNN and LSTM, the training loop handles taking data batches, computing the loss, backpropagating the gradient, and updating the weights via the optimizer. In other words, the training loop does not "know" whether it is working on an FFNN or an LSTM: it applies the same optimization procedure. The substantial difference lies in how each model transforms inputs into outputs, meaning the input to output mapping is architecturally different, but the weight update mechanism remains the same.

The only note to make is the following. During the conversion of data into Torch tensors, the LSTM performs the following conversion:

```
stats_np = statistics.numpy().astype(int)
```

Listing 3.21: LSTM tensor conversion

In contrast, as we saw in section 3.3.3 the FFNN simply performs:

```
stats_np = statistics.numpy()
```

Listing 3.22: FFNN tensor conversion

The difference between the two cases is not arbitrary, but depends on the type of data each architecture expects as input. In `train.py`, when working with an FFNN, `statistics` are already continuous numerical features (floats), derived from the computation of statistics on the encrypted text, so it is sufficient to convert them into a float tensor using `.numpy()`. This is consistent with the linear operations and ReLU activations in the FFNN, which operate on `float32` values.

In the case of the LSTM, however, input sequences are often treated as discrete indices or integer values representing tokens or symbols. For this reason, a different conversion is applied. This way, the values are integers, which can then be transformed into embeddings or processed as sequences.

3.3.7 LSTM prediction function

3.3.8 Other changes

Additionally, I introduced two minor but necessary modifications to improve the compatibility between the legacy script and the newly implemented PyTorch-based architecture. These changes involve printing a model summary and saving the model with the appropriate file extension.

Model summary To visualize the architecture of the model, I used the `torchsummary` library. If the model does not implement its own `summary()` method, the code falls back to calling `summary()` with a predefined input size. This is particularly helpful when debugging or validating the model structure.

```
if architecture in ("FFNN", "CNN", "LSTM", "Transformer") and extend_model is
    ↪ None:
    if hasattr(model, "summary"):
        model.summary()
    else:
        summary(model, input_size=(1, 724))
```

Listing 3.23: Printing the model summary

Here the visualization of the summary for the FFNN architecture:

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
FFNN                                     [1, 61]                   --
Sequential: 1-1                         [1, 61]                  --
    Linear: 2-1                         [1, 543]                 393,675
    ReLU: 2-2                           [1, 543]                 --
    Linear: 2-3                         [1, 543]                 295,392
```

```
    ReLU: 2-4                                [1, 543]                  --
    Linear: 2-5                              [1, 543]                  295,392
    ReLU: 2-6                                [1, 543]                  --
    Linear: 2-7                              [1, 61]                   33,184
==========================================================================
Total params: 1,017,643
Trainable params: 1,017,643
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 1.02
==========================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 4.07
Estimated Total Size (MB): 4.09
==========================================================================
```

Listing 3.24: Printing the model summary of FFNN

And here is the summary of LSTM:

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
Embedding: 1-1                           [1, 1000, 64]             3,584
LSTM: 1-2                                [1, 1000, 500]            1,132,000
Linear: 1-3                              [1, 61]                   30,561
==========================================================================
Total params: 1,166,145
Trainable params: 1,166,145
Non-trainable params: 0
Total mult-adds (G): 1.13
==========================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 4.51
Params size (MB): 4.66
Estimated Total Size (MB): 9.18
==========================================================================
```

Listing 3.25: Printing the model summary of LSTM

Saving model Originally, models were saved using the `.h5` extension. However, for the PyTorch-based models, it was necessary to switch to the `.pth` format. To avoid overwriting existing files, the function `save_model` (see appendix lines from 1218 to 1294) dynamically appends an incremental number to the file name. Moreover, if a specific name is provided via `--model_name` (see section 2.3).

Another important modification involved the `torch.save` function. Unlike Keras' `model.save`, which stores both the weights and the architecture (such as input size, output size, and hidden layer configuration), PyTorch's `torch.save` saves only the model's weights via the `state_dict()`. To reproduce the same behavior, it was necessary to manually include the structural parameters

when saving the model. This was achieved by using the `FFNN` class of PyTorch (see section 3.3.2) to pass all required values.

```
    if args.model_name == 'm.h5':
        i = 1
        base_name = args.model_name.split('.')[0]
        extension = '.pth' if architecture == "FFNN" else '.h5'
        while os.path.exists(os.path.join(args.save_directory, base_name +
            ↪ str(i) + extension)):
            i += 1
        model_name = base_name + str(i) + extension
    else:
        model_name = args.model_name
        if architecture == "FFNN":
            model_name = model_name.replace('.h5', '.pth')
```

Listing 3.26: Saving the model with correct extension

```
    if architecture in ("FFNN", "LSTM"):
        state_dict = {
            'model_state_dict': model.state_dict(),
            'hidden_size': model.hidden_size,
            'output_size': model.output_size,
        }

        if architecture == "FFNN":
            state_dict['input_size'] = model.input_size
            state_dict['num_hidden_layers'] = model.num_hidden_layers
        elif architecture == "LSTM":
            state_dict['vocab_size'] = model.vocab_size
            state_dict['embed_dim'] = model.embed_dim
            state_dict['num_layers'] = model.num_layers
            state_dict['dropout'] = model.dropout

        torch.save(state_dict, model_path)
```

Listing 3.27: Saving the model with correct extension

Input validation Last but not least, the input validation logic was modified. Previously, the script accepted only filenames with the `.h5` extension, consistent with Keras models. However, as previously discussed, PyTorch models are conventionally saved using the `.pth` extension. Therefore, the validation step was updated to enforce this requirement, ensuring that the file extension matches the framework being used.

```
if os.path.splitext(args.model_name)[1] not in ('.h5', '.pth'):
    print('ERROR: The model must have extension ".h5" (for Keras) or ".pth" (
        ↪ for PyTorch).',file=sys.stderr)
    sys.exit(1)
```

Listing 3.28: Newer input validation

3.4 Evaluation code of the model

After porting `train.py` to PyTorch and successfully training models, I proceeded to adapt the `eval.py` script accordingly, see pages 61 and 79. Unlike `train.py`, I chose to retain the original function flow already present in `eval.py`, modifying only the conditional statements responsible for selecting the correct evaluation logic for the specified architecture.

3.4.1 Benchmark function

The benchmark function was adapted to evaluate PyTorch models using tensors. The evaluation is now performed as seen in listing 3.29:

```
if architecture == ("FFNN", "LSTM"):
    if hasattr(model, "evaluate"):  # Keras model
        results.append(model.evaluate(statistics, labels, batch_size=args.
            ↪ batch_size, verbose=1))
    else:  # PyTorch model
        x = torch.tensor(statistics.numpy(), dtype=torch.float32)
        y = torch.tensor(labels.numpy(), dtype=torch.long)
        with torch.no_grad():
            outputs = model(x)
            loss = F.cross_entropy(outputs, y)
            top1 = torch.argmax(outputs, dim=1)
            acc = (top1 == y).float().mean()

            top3 = torch.topk(outputs, k=3, dim=1).indices
            y_expanded = y.unsqueeze(1).expand_as(top3)
            k3_acc = (top3 == y_expanded).any(dim=1).float().mean()
```

Listing 3.29: New evaluation for `FFNN` and `LSTM` architecture

As can be observed, the evaluation method is identical to the one carried out by the `predict` function in `train.py` (see section 3.3.4). This is because the evaluations must be consistent for both cases, meaning that they should yield more or less the same result.

3.4.2 Model loading

The `load_model` function was updated to support PyTorch `.pth` models. Parameters previously saved using `torch.save` are now correctly reloaded with `torch.load`. listing 3.30 allow the checkpoint saved on disk with `torch.save()` to be restored, and depending on the chosen architecture (FFNN or LSTM), the appropriate class is initialized by passing the stored parameters (input dimensions, hidden size, number of layers, etc.). Subsequently, the model weights are loaded with `model.load_state_dict(...)` and the model is set to `eval()` mode in order to be used exclusively for prediction, without further training.

```
if architecture in ("FFNN", "LSTM") and model_path.endswith(".pth"):
    if architecture == "FFNN":
        from cipherTypeDetection.train import TorchFFNN
    elif architecture == "LSTM":
        from cipherTypeDetection.train import TorchLSTM

    checkpoint = torch.load(model_path, map_location=torch.device("cpu"))

    if architecture == "FFNN":
        model = TorchFFNN(
            input_size=checkpoint['input_size'],
            hidden_size=checkpoint['hidden_size'],
            output_size=checkpoint['output_size'],
            num_hidden_layers=checkpoint['num_hidden_layers']
        )
    elif architecture == "LSTM":
        model = LSTM(
            vocab_size=checkpoint['vocab_size'],
            embed_dim=checkpoint['embed_dim'],
            hidden_size=checkpoint['hidden_size'],
            output_size=checkpoint['output_size'],
            num_layers=checkpoint['num_layers'],
            dropout=checkpoint['dropout']
        )

    model.load_state_dict(checkpoint['model_state_dict'])
    model.eval()

    config.FEATURE_ENGINEERING = (architecture == "FFNN")
    config.PAD_INPUT = (architecture == "LSTM")

    return model
```

Listing 3.30: Setting model parameters

In addition, in this way, if an FFNN is loaded, `FEATURE_ENGINEERING=True` and `PAD_INPUT=False` are set (see section 2.4 and section 2.3).
Conversely, if an LSTM is loaded, `FEATURE_ENGINEERING=False` and `PAD_INPUT=True`, so that the model receives tokenized ciphertexts instead of numerical features.

In addition, I changed the previous logic that always evaluated models as ensembles. Now, the model is only wrapped in a `RotorDifferentiationEnsemble` when rotor ciphers are actually present. This modification was introduced to avoid unnecessary overhead when evaluating models that do not involve rotor ciphers, ensuring that the ensemble mechanism is only applied when it is actually required:

```
has_rotor_ciphers = any(c in config.ROTOR_CIPHER_TYPES for c in cipher_types)

if has_rotor_ciphers:
    rotor_only_model_path = args.rotor_only_model
    if not os.path.exists(rotor_only_model_path):
        raise FileNotFoundError(f"Rotor-only model is required but not found
            ↪ at {rotor_only_model_path}")
```

```
    with open(rotor_only_model_path, "rb") as f:
        rotor_only_model = pickle.load(f)
    return RotorDifferentiationEnsemble(architecture, model, rotor_only_model
        ↪ )

return model
```

Listing 3.31: Changes to Ensemble architecture

3.4.3 Input validation

Finally, the input validation in the `main` function was extended to support models saved with the `.pth` extension:

```
if os.path.splitext(args.model)[1] not in ('.h5', '.pth'):
    print('ERROR: The model must have extension ".h5" (for Keras) or ".pth" (
        ↪ for PyTorch FFNN and LSTM).', file=sys.stderr)
    sys.exit(1)
```

Listing 3.32: Newer input validation

4 Evaluation and results

As I have already discussed in section 3.4, the final part of this work consists of the evaluation phase, that is, testing the model with new data in order to obtain meaningful results such as accuracy. In section 3.4, the goal was to correctly implement the code that enables this phase, whereas in this chapter we will compare the obtained results, aiming to contrast the Keras and PyTorch implementations.

Before analyzing the content of this chapter, I need to clarify the purpose of these evaluations. The goal of this thesis is not to improve the performance of the app; therefore, as we will see, I will generally aim to replicate the results achieved by the previous Keras implementation. This is because I do not have the exhaustive knowledge required to enhance a learning model like NCID, but I have worked to make the two implementations as similar as possible.

After implementing our architectures, as discussed in the previous chapter, it is now time to test the models with large amounts of data. Until now, I had been testing all modifications locally with parameters that allowed for simple execution, mainly to highlight potential issues in the data flow. However, these runs had no significance in terms of the actual evaluation of the model and the study of its accuracy.

To perform more meaningful evaluations, I was able to use a machine at the Universität der Bundeswehr München, namely the NVIDIA DGX H100, with the assistance of Doris Behrendt and Maik Bastian. For details on the specifications of the machine used, you can consult the following website [16]. A small clarification: I did not use the full power of the DGX machine; I was only able to train the model on a single GPU.

In order to run the modifications I implemented, I had to create Dockerfiles which, once unpacked on the cluster, allowed the execution of the code from my personal GitHub repository. A Dockerfile is a simple text file that contains the instructions to build a Docker container, that is, an isolated environment in which to run an application in exactly the same way on any machine.

In listing 4.1 depicts the command line used to run the dockerfile previously built. I won't share the build command since it contains reserved informations.

```
1 docker run -v $(pwd)/output:/app/ncid/cipherTypeDetection/output -it --gpus
    ↪ all --name ncid-keras-benchmark ncid-keras-benchmark
```

Listing 4.1: Docker run command

That said, let's start analyzing the results obtained.

4.1 First runs

Result from Keras Initially, I used the Keras implementation to perform a small training run with the following input data (see section 2.3):

- `train_dataset_size = 1000`
- `batch_size = 256`
- `dataset_workers = 4`
- `min_train_len = 100`
- `max_train_len = 1000`
- `min_test_len = 100`
- `max_test_len = 1000`
- `max_iter = 10000`

The results training the FFNN were the following:

```
Accuracy: 0.1984 k3-accuracy: 0.3783
```

Here, **accuracy** indicates the probability that the cipher was correctly identified, and **k3-accuracy** represents the probability that the correct cipher was among the top 3 most probable predictions.

So now we have a clear goal: to get as close as possible to the results of the Keras implementation.

Result from PyTorch The code has undergone several modifications over the months; indeed, the first training run with the same inputs yielded these results.

```
Accuracy: 0.042523, k3-accuracy: 0.075315
```

Obviously, we were still far from a good implementation. For example, at the beginning I added an unnecessary data normalization step. This happened because I initially thought that the architecture class did not receive already normalized inputs. In fact, as we have seen in section 3.2, the data were already normalized, and therefore performing this operation a second time significantly worsened the model's accuracy. In listing 4.2 depicts the redundant normalization step that was added.

```
1 mean = stats_np.mean(axis=0)
2 std = stats_np.std(axis=0) + 1e-8
3 stats_np = (stats_np - mean) / std
```

Listing 4.2: Redundant normalization

So, after discussing it with Maik Bastian, I removed the redundant normalization. Then I obtained the following results:

```
Accuracy: 0.076720, k3-accuracy: 0.151515
```

As can be seen, both the single accuracy and the k3 accuracy improved, although only slightly. This is where the modification I mentioned in the implementation chapter, which Maik Bastian pointed out to me (see section 3.3.3), comes into play. After fixing the implementation as discussed earlier, I finally obtained results similar to those of Keras:

```
Accuracy: 0.165479, k3-accuracy: 0.364742
```

Target acquired!

4.2 DGX runs

It was then time to test our models on a large scale using the DGX machine. Below are the command line hyperparameters:

- `train_dataset_size = 976`
- `batch_size = 64`
- `dataset_workers = 16`
- `min_train_len = 100`
- `max_train_len = 1000`
- `min_test_len = 100`
- `max_test_len = 1000`
- `max_iter = 10000000`

To compare the two implementations, I prepared two shell scripts. The one shown in listing 4.3 is the script used on the DGX machine to test the PyTorch implementation. Of course, I also created a similar one for Keras that I won't add here:

```
BASE_DIR=output
mkdir -p "$BASE_DIR"

DATE=$(date "+%Y%m%d")

COMMON_ARGS="--download_dataset=False \
    --plaintext_input_directory=../data/gutenberg_en \
    --rotor_input_directory=../data/rotor_ciphertexts \
    --train_dataset_size=976 \
    --dataset_workers=16 \
    --batch_size=64 \
    --max_iter=10000000 \
    --min_train_len=100 \
    --max_train_len=1000 \
    --min_test_len=100 \
    --max_test_len=1000 \
    --epochs=1 \
    --ciphers=all"

run_benchmark () {
  ARCH=$1
  for i in {1..3}; do
    RUN_DIR="$BASE_DIR/${ARCH}_run_$i"
    mkdir -p "$RUN_DIR"
    echo "Launching $ARCH run $i..."
    python train.py \
      --architecture=$ARCH \
      $COMMON_ARGS \
      --save_directory="$RUN_DIR" \
      --model_name="${ARCH}_run_$i.pth" \
      > "$RUN_DIR/${ARCH}_var_10000000_run_${i}_${DATE}.txt" \
      2> "$RUN_DIR/err_${ARCH}_var_10000000_run_${i}_${DATE}.txt"
  done
}

# FFNN
run_benchmark FFNN

# LSTM
run_benchmark LSTM
```

Listing 4.3: Shell script for PyTorch

The goal is to test the FFNN architecture implementation three times and the LSTM implementation three times, for both frameworks. This process was carried out using the machine reserved for these tests, in order to ensure the most accurate results possible.

Several tests were performed to verify whether the implementation was correct, whether the models were being saved properly, and to address other potential issues. From this point on, I will report only the results of one result for both Keras and PyTorch. A quick note: the PyTorch results date back to the test performed on August 8, while the Keras results refer to the test conducted on October 6. The Keras results were repeated later because some anomalies had

been found in the model saving process, which prevented a correct evaluation.

4.2.1 Results from Keras

Once again, we will use the results obtained as a benchmark to assess whether we have successfully implemented our architectures. Moreover, it is fundamental introducing a the concept of *Precision*, *Recall* and *F1-score*. Precision indicates the percentage of correct positive predictions, that is, how often the model is right when it predicts a positive class. Recall measures the model's ability to identify all actual positive instances, that is, how many of the real positive classes have been correctly recognized. Precision and recall differ in focus: precision emphasizes the accuracy of positive predictions, while recall emphasizes the completeness of capturing all positive instances. The F1-score is a metric used to evaluate the performance of a classification model, especially in cases where the data are imbalanced, meaning that some classes have significantly more samples than others.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.2}$$

where TP stands for true positive, FP for false positive and FN for false negative. All three variables (TP, FP and FN) are non-negative integer counts that represent the number of samples in each category. The F1 score is defined as the harmonic mean between precision and recall:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

This value ranges from 0 to 1, where 1 indicates perfect performance (both precision and recall are equals to 1), and 0 indicates the worst performance.

Now we will discuss the numerical results of the training sessions. To provide a direct reference, I include a list of the ciphers in their order of appearance, sorted alphabetically. This is the same order that appears when we obtain the results during the model evaluation.

1. amsco	**2.** autokey	**3.** baconian	**4.** bazeries	**5.** beaufort	**6.** bifid
7. cadenus	**8.** checkerboard	**9.** columnar_transposition	**10.** condi	**11.** cmbifid	**12.** digrafid
13. foursquare	**14.** fractionated_morse	**15.** grandpre	**16.** grille	**17.** gromark	**18.** gronsfeld
19. headlines	**20.** homophonic	**21.** key_phrase	**22.** monome_dinome	**23.** morbit	**24.** myszkowski
25. nicodemus	**26.** nihilist_substitution	**27.** nihilist_transposition	**28.** null	**29.** numbered_key	**30.** periodic_gromark
31. phillips	**32.** phillips_rc	**33.** plaintext	**34.** playfair	**35.** pollux	**36.** porta
37. portax	**38.** progressive_key	**39.** quagmire1	**40.** quagmire2	**41.** quagmire3	**42.** quagmire4
43. ragbaby	**44.** railfence	**45.** redefence	**46.** route_transposition	**47.** running_key	**48.** seriated_playfair
49. slidefair	**50.** swagman	**51.** tridigital	**52.** trifid	**53.** tri_square	**54.** two_square
55. variant	**56.** vigenere	**57.** enigma	**58.** m209	**59.** purple	**60.** sigaba
61. typex					

If you wish to skip directly to the comparison, see section 4.3.

FFNN results With the input hyperparameter specified in section 4.2, we obtained the following results for the Keras FFNN:

```
Total accuracy: 0.7427164374276443

Total k3: 0.9042634659180658

precision: [0.84247601 0.70099323 1.         0.99434685 0.23120063 0.76061568
 0.87966872 0.95546632 0.93239781 0.88377353 0.80099023 0.77054351
 0.93703211 0.99554579 0.94213309 0.9879969  0.89974178 0.7988629
 0.99993887 0.8506699  0.99640551 0.99546737 0.92647763 0.63775634
 0.85385154 0.99027014 0.73630802 1.         0.99939062 0.97517683
 0.86251165 0.7487211  0.99750623 0.96016622 0.98180602 0.47141996
 0.99668019 0.39556921 0.16037532 0.20766575 0.22230205 0.19764465
 1.         0.41750766 0.43044917 0.99993903 0.49032965 0.84550003
 0.48359369 0.99441375 0.77116174 0.8177442  0.95524203 0.89872368
 0.2066785  0.20256524 0.31359946 0.46392964 0.34882021 0.29313037
 0.32087629]

recall: [0.92615854 0.78323171 1.         0.99743902 0.23378049 0.86176829
 0.66605189 0.9720122  0.97640244 0.73859756 0.75957317 0.83420732
 0.81664634 0.99487805 0.99969512 0.93353659 0.97731707 0.97670732
 0.99743902 0.98335366 0.9972561  0.7097561  0.94432927 0.4797561
 0.96719512 0.99914634 0.92469512 1.         1.         0.89109756
 0.73329268 0.89243902 1.         0.98621951 0.895      0.71560976
 0.98853659 0.73817073 0.09067073 0.18335366 0.09432927 0.14121951
 0.99963415 0.34054878 0.57908537 1.         0.76365854 0.8552439
 0.75128049 0.97689024 0.87103659 0.74579268 0.95780488 0.85871951
 0.11359756 0.22341463 0.16957317 0.6979878  0.20371951 0.2747561
 0.12146341]

f1: [0.88233756 0.73983412 1.         0.99589054 0.2324834  0.80803865
 0.7580995  0.96366824 0.95389289 0.80469009 0.7797321  0.80111258
 0.87270713 0.99521181 0.97006094 0.95999498 0.9369264  0.87887849
 0.99868738 0.91221223 0.99683062 0.82867618 0.93531828 0.54758673
 0.90699603 0.99468844 0.81981836 1.         0.99969521 0.93124323
 0.79267047 0.8142873  0.99875156 0.9730185  0.93639553 0.56839811
 0.99259169 0.5151051  0.11584606 0.19475389 0.1324543  0.16473433
 0.99981704 0.37512174 0.49382523 0.99996951 0.59720566 0.85034405
 0.58842352 0.98557411 0.81806208 0.78011289 0.95652174 0.87826629
 0.1466121  0.2124797  0.22012031 0.55738423 0.25721765 0.28364598
 0.17622081]
```

LSTM results In this paragraph the Keras LSTM results are shown:

```
Total accuracy: 0.699571922430185
```

```
Total k3: 0.8750897243388391

precision: [0.87641498 0.65051067 0.99993903 0.97511339 0.23158169 0.62077346
 0.73410459 0.84212268 0.6456672  0.32411558 0.62267438 0.79025262
 0.92058659 0.97330374 0.98561677 0.99636811 0.99944673 0.85276796
 1.         0.99115871 0.94371099 0.99415133 0.98101886 0.63922907
 0.66583541 0.9971741  0.65088876 1.         1.         0.99129173
 0.60972544 0.62101725 0.9995122  0.62416934 0.97185539 0.61542203
 0.93694264 0.30687831 0.16879706 0.22280193 0.20307135 0.18400538
 0.99939036 0.40662154 0.37781341 0.99993902 0.53409208 0.6180449
 0.7126617  0.96941424 0.96679606 0.76808586 0.77763496 0.76023877
 0.1938861  0.21226216 0.25221621 0.42009865 0.21202813 0.21203855
 0.24711656]

recall: [0.91585366 0.65634146 1.         0.99628049 0.21658537 0.5295122
 0.47924598 0.77603659 0.74054878 0.36591463 0.54487805 0.75536585
 0.90335366 0.98926829 0.98609756 0.98695122 0.99134146 0.97121951
 0.99695122 0.97067073 0.95890244 0.98463415 0.97695122 0.38829268
 0.71634146 0.9897561  0.98689024 1.         0.99987805 0.9995122
 0.76914634 0.51817073 0.9995122  0.86481707 0.99170732 0.84871951
 0.9929878  0.51280488 0.05878049 0.14060976 0.12981707 0.08347561
 0.99957317 0.43585366 0.37359756 0.99987805 0.6295122  0.5404878
 0.88682927 0.95664634 0.98713415 0.78981707 0.70091463 0.83871951
 0.33878049 0.24487805 0.19603659 0.65432927 0.15993902 0.07109756
 0.18420732]

f1: [0.8957004  0.65341306 0.99996951 0.9855833  0.22383263 0.57152259
 0.57990935 0.80773014 0.68986083 0.3437491  0.58118435 0.77241551
 0.91188871 0.98122108 0.98585711 0.99163731 0.9953776  0.90814756
 0.99847328 0.98080774 0.95124607 0.98936985 0.97898081 0.48311964
 0.69016567 0.99345125 0.78442301 1.         0.99993902 0.99538499
 0.68022002 0.56495147 0.9995122  0.72504665 0.981681   0.71348387
 0.96415145 0.38397443 0.08719642 0.17241121 0.15838417 0.11484899
 0.99948175 0.42073045 0.37569366 0.99990853 0.57788973 0.57667035
 0.79026299 0.96298797 0.97685925 0.7787999  0.73728433 0.79755313
 0.24662642 0.22740657 0.22060589 0.51168224 0.18233638 0.10648888
 0.21107424]
```

4.2.2 Results from PyTorch

Now we present the PyTorch results, after which we will draw some conclusions.

FFNN results Starting with the FFNN:

```
Total accuracy: 0.7313708694494867
```

```
Total k3: 0.9020696031909635

precision: [0.77108553 0.64081574 1.         0.99379977 0.22407726 0.77174341
 0.94204352 0.99584505 0.88031958 0.75089452 0.83762434 0.80873805
 0.89755428 0.9911354  0.92796322 0.99900431 0.88510873 0.80968893
 0.99987772 0.64407725 0.9967045  0.90825141 0.86191933 0.46486261
 0.95426792 0.98740129 0.69856062 1.         0.99266889 0.97904397
 0.76588775 0.86504006 0.99350689 0.96091244 1.         0.31089329
 0.99572101 0.3690516  0.16369407 0.19653499 0.20846609 0.16659684
 1.         0.35294118 0.43524855 1.         0.57956381 0.83109792
 0.67200429 0.99152857 0.86308309 0.79863337 0.9533808  0.82372935
 0.19591978 0.19693404 0.32494279 0.53625992 0.43583976 0.27847226
 0.31505273]

recall: [0.9486084  0.8170166  1.         0.99786377 0.16711426 0.83782959
 0.57701757 0.9508667  0.98858643 0.90942383 0.69897461 0.78973389
 0.87805176 0.99633789 0.9977417  0.9185791  0.98132324 0.98339844
 0.99816895 0.91595459 0.99682617 0.93048096 0.94561768 0.47601318
 0.95391846 0.99975586 0.97454834 1.         1.         0.87255859
 0.85620117 0.72491455 0.99926758 0.98730469 0.49353027 0.73370361
 0.99420166 0.80847168 0.06610107 0.19525146 0.0838623  0.14562988
 0.99981689 0.02783203 0.70648193 1.         0.70068359 0.85473633
 0.53533936 0.97869873 0.74871826 0.80609131 0.95611572 0.94665527
 0.1729126  0.18896484 0.09533691 0.65985107 0.24835205 0.29504395
 0.10028076]

f1: [0.85068418 0.71826792 1.         0.99582762 0.19144845 0.80342981
 0.71567321 0.97283627 0.93131702 0.82259089 0.76204418 0.799123
 0.88769592 0.99372984 0.96158824 0.95710515 0.93073606 0.88812943
 0.9990226  0.75632497 0.99676533 0.91923181 0.90183067 0.47037182
 0.95409316 0.99354017 0.8137917  1.         0.99632096 0.9227393
 0.80853026 0.78880255 0.99637891 0.9739298  0.66089089 0.43673025
 0.99496075 0.50677175 0.09417391 0.19589112 0.11960827 0.15540937
 0.99990844 0.05159538 0.53864768 1.         0.63439434 0.8427514
 0.59593695 0.98507188 0.80184332 0.80234501 0.95474631 0.88092466
 0.18369861 0.19286715 0.14742107 0.59167032 0.31640747 0.28651869
 0.15213667]
```

LSTM results

```
Total accuracy: 0.7012079733647427

Total k3: 0.8742761016214723

precision: [0.85796652 0.65652963 0.99987794 0.99353659 0.19896735 0.68323991
 0.56204133 0.85257582 0.8650025  0.27295937 0.64737512 0.83648348
 0.85027119 0.98911391 0.99039579 0.99307159 0.99993889 0.90409117
 0.99993883 0.99721172 0.91260593 0.99652397 0.98542363 0.71317752
```

```
 0.48079802 0.99211117 0.58891286 0.99987794 0.99993897 0.99871982
 0.67685637 0.63413706 0.99987786 0.8504757  0.98306821 0.68676212
 0.97486797 0.32358401 0.13672073 0.19156903 0.18345945 0.18534671
 1.         0.35910812 0.35320563 0.99993895 0.52338435 0.66906094
 0.76714031 0.92055668 0.99588022 0.71154232 0.72936094 0.81924149
 0.20497916 0.19357802 0.23506053 0.4480726  0.22592295 0.201594
 0.28153913]

recall: [0.94458008 0.66278076 1.         0.99450684 0.16699219 0.51330566
 0.41035906 0.82525635 0.52874756 0.45025635 0.55847168 0.6428833
 0.94726562 0.99267578 0.98815918 0.99731445 0.99865723 0.94415283
 0.99768066 0.9822998  0.94647217 0.99737549 0.99029541 0.43768311
 0.84283447 0.99786377 0.88311768 1.         1.         0.99993896
 0.76220703 0.60998535 0.99926758 0.80749512 0.99578857 0.83435059
 0.98016357 0.43621826 0.06585693 0.19360352 0.08435059 0.02593994
 0.99981689 0.52099609 0.33087158 0.99969482 0.60107422 0.69360352
 0.86663818 0.96893311 0.98852539 0.8661499  0.8449707  0.87939453
 0.22814941 0.32159424 0.22399902 0.58459473 0.13745117 0.07873535
 0.19873047]

f1: [0.89919238 0.65964038 0.99993897 0.99402147 0.18158288 0.58620569
 0.47436993 0.83869367 0.65631274 0.3398756  0.59964611 0.72701546
 0.89615151 0.99089164 0.98927622 0.9951885  0.99929765 0.92368782
 0.99880847 0.9896996  0.92923058 0.99694955 0.98785351 0.54245622
 0.61230462 0.99497916 0.7066149  0.99993897 0.99996948 0.99932902
 0.71700063 0.62182678 0.99957262 0.8284283  0.98938751 0.75339635
 0.9775086  0.3715526  0.08889438 0.1925809  0.11556633 0.04551052
 0.99990844 0.42516312 0.34167402 0.99981687 0.55954545 0.68111121
 0.81385951 0.94412561 0.99218918 0.78127065 0.78292097 0.84825292
 0.21594454 0.24168062 0.22939651 0.50730932 0.17091682 0.11324233
 0.23299581]
```

4.3 Comparison

The reported evaluations show very satisfactory results regarding the achievement of our goal, which, I would like to recall once again, was not to improve our previous implementation in terms of model accuracy, but rather to obtain results with PyTorch that were consistent with those achieved using Keras. At first glance, the two implementations behave in a very similar way, as can be seen in table 4.1 and table 4.2. It should also be reminded that both implementations were trained with identical hyperparameters, in order to ensure a fair evaluation.

Starting from the FFNN results, the Keras model achieved a total accuracy of 74.27% and a top-3 accuracy of 90.04%. The same architecture implemented in PyTorch reached a total accuracy of 73.13% and a top-3 accuracy of 90.02%. The total accuracy result is perfectly in line with our expectations, while the total accuracy is slightly lower. These discrepancies are not concerning, but they often result from differences between the two frameworks, which use slightly different

	Keras	PyTorch
FFNN	74.27%	73.13%
LSTM	69.95%	70.12%

Table 4.1: Table of total accuracies

	Keras	PyTorch
FFNN	90.04%.	90.02%
LSTM	87.50%	87.42%

Table 4.2: Table of top 3 accuracies

weight initialization or floating-point rounding mechanisms, leading to minor variations in the results.

Similarly, for the LSTM architecture, the Keras implementation achieved a total accuracy of 69.95% and a top-3 accuracy of 87.50%, while the PyTorch version obtained a total accuracy of 70.12% and a top-3 accuracy of 87.42%. As with the FFNN, these small discrepancies are due to operational differences between the frameworks.

Let's now take a look to the evaluation metrics. A more detailed analysis of these highlights that both implementations perform consistently across most classes, with minor variations in Precision, Recall, and F1-score. Overall, the Keras model shows slightly higher values for these metrics.

The Precision values indicate that both models are generally reliable when predicting a positive class, although Keras tends to produce fewer false positives in several cipher categories. The Recall values are also comparable, suggesting that both models are capable of identifying the majority of the true instances. However, in a few specific classes, PyTorch shows lower Recall, meaning it occasionally fails to recognize some correct samples that Keras is able to detect.

The F1-score, which combines Precision and Recall into a single measure, reflects this balance. The Keras implementation achieves slightly higher F1-scores overall, confirming its better equilibrium between accurate and complete predictions.

5 Conclusion

As we have seen, the project aimed at the development and evaluation of FFNN and LSTM architectures, now familiar to you, for the automatic classification of classical ciphers, with particular attention to the comparison between implementations in Keras and PyTorch.

We started from the NCID application, analyzing what it is, how it works, its connections with cryptography, and the technologies it relies on. This provided the necessary foundation for understanding the work carried out. We began by reviewing the fundamentals of Machine Learning, including the concept of statistical features, and comparing the two frameworks Keras and PyTorch.

All of this served as preparation for the core part of the project: the implementation. In this phase, we analyzed the data flow within the NCID app and examined each stage: the extraction of plaintexts and their encryption, the preprocessing of texts and the calculation of statistical features, the training and validation of the models, and finally their evaluation. We highlighted the technical differences between the two frameworks and solved some technical issues. The most significant issues encountered were the incorrect implementation of data normalization and the improper optimization of gradients during model training, which, however, were resolved as discussed in chapter 3.

We concluded the work by comparing the results obtained from the Keras and PyTorch implementations. As already mentioned, the goal was not to outperform the previous implementation, but to replicate it accurately, thus providing a solid starting point for anyone working on this project. As shown in chapter 4, this goal was successfully achieved for both the FFNN and LSTM architectures.

Looking ahead, the work could be extended by implementing in PyTorch the remaining architectures, such as Transformer and Naive Bayes, as well as by making more in-depth modifications to the training and evaluation cycles. I hope that the project has provided a concrete contribution to the development of the NCID app, offering both significant experimental results and a useful foundation for future developments.

6 Acknowledgments

For the realization of this thesis, I would like to sincerely thank the entire CrypTool team for their support in the development of this work and for guiding me throughout the project. In particular, I would like to thank Dr. Doris Behrendt and Maik Bastian, who were always available to answer my questions and provided valuable guidance with their advice throughout the entire process. I would also like to thank Professor Bernhard Esslinger for giving me the opportunity to work on this project, which I truly appreciated and in which I fully immersed myself, and that has been fundamental in shaping my choice of master's degree.

Stefano Sala, October 13, 2025

7 Bibliography

[1] Ernst Leierzopf. *NCID*. 2021. URL: https://github.com/ernstleierzopf/ncid.

[2] Nils Kopal. "Of Ciphers and Neurons – Detecting the Type of Ciphers Using Artificial Neural Networks". In: *Proceedings of the 3rd International Conference on Historical Cryptology (HistoCrypt 2020)*. Vol. 171. Linköping University Electronic Press, May 2020, pp. 77–86. DOI: 10.3384/ecp2020171011. URL: https://ep.liu.se/en/conference-article.aspx?series=ecp&issue=171&Article_No=11.

[3] Brooke Dalton and Mark Stamp. *Classifying World War II Era Ciphers with Machine Learning*. 2023. arXiv: 2307.00501 [cs.LG]. URL: https://arxiv.org/abs/2307.00501.

[4] Tom M. Mitchell. *Machine Learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[6] Ernst Leierzopf et al. "A Massive Machine-Learning Approach For Classical Cipher Type Detection Using Feature Engineering". English. In: *Proceedings of the 4th International Conference on Historical Cryptology*. Aug. 2021, pp. 111–120. DOI: 10.3384/ecp183164.

[7] Tariq Rashid. *Make Your Own Neural Network*. 1st. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2016. ISBN: 1530826608.

[8] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[9] Amrit Singh Bist. *Comprehensive Understanding of LSTM Architecture: A Deep Dive into Long Short-Term Memory Networks*. Accessed: 2025-07-17. 2023. URL: https://medium.com/@amritsinghbist/comprehensive-understanding-of-lstm-architecture-a-deep-dive-into-long-short-term-memory-networks-5405f97afc6b.

[10] PyTorch Contributors. *torch.nn.CrossEntropyLoss*. 2025. URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

[11] Keras Team. *Keras API Reference*. 2024. URL: https://keras.io/api/.

[12] PyTorch Team. *PyTorch: An Open Source Machine Learning Framework*. 2024. URL: https://pytorch.org/.

[13] Luca Pietro Giovanni Antiga, Eli Stevens, and Thomas Viehmann. *Deep learning with PyTorch*. Simon and Schuster, 2020.

[14] Pavan Patel. *From Keras to PyTorch*. 2020. URL: https://medium.com/analytics-vidhya/from-keras-to-pytorch-722fa3b65cce.

[15] Ernst Leierzopf et al. “Detection of Classical Cipher Types with Feature-Learning Approaches”. In: *Data Mining*. Ed. by Yue Xu et al. Singapore: Springer Singapore, Dec. 2021, pp. 152–164. ISBN: 978-981-16-8531-6. DOI: 10.1007/978-981-16-8531-6_11. URL: https://link.springer.com/chapter/10.1007/978-981-16-8531-6_11.

[16] NVIDIA Corporation. *Introduction to NVIDIA DGX H100/H200*. 2025. URL: https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html.

8 Appendix: Full Python code

Here is listed all the code I wrote during the internship.

8.1 Full listing of `train.py`

```
import multiprocessing
from pathlib import Path

import argparse
import sys
import time
import shutil
from sklearn.model_selection import train_test_split
import os
import math
import pickle
import functools

# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim
from torchinfo import summary
import numpy as np
from torch.utils.data import TensorDataset, DataLoader

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
from datetime import datetime

# This environ variable must be set before all tensorflow imports!
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
from tensorflow.keras.metrics import SparseTopKCategoricalAccuracy
from tensorflow.keras.optimizers import Adam  # , Adamax
import tensorflow_datasets as tfds
sys.path.append("../")
from cipherTypeDetection.nullDistributionStrategy import NullDistributionStrategy
import cipherTypeDetection.config as config
from cipherTypeDetection.trainingBatch import TrainingBatch
from cipherTypeDetection.cipherStatisticsDataset import RotorCiphertextsDatasetParameters, PlaintextPathsDatasetParameters,
    ↪ CipherStatisticsDataset
from cipherTypeDetection.predictionPerformanceMetrics import PredictionPerformanceMetrics
from cipherTypeDetection.miniBatchEarlyStoppingCallback import MiniBatchEarlyStopping
from cipherTypeDetection.transformer import TransformerBlock, TokenAndPositionEmbedding
from cipherTypeDetection.learningRateSchedulers import TimeBasedDecayLearningRateScheduler, CustomStepDecayLearningRateScheduler
tf.debugging.set_log_device_placement(enabled=False)
# always flush after print as some architectures like RF need very long time before printing anything.
print = functools.partial(print, flush=True)
for device in tf.config.list_physical_devices('GPU'):
    tf.config.experimental.set_memory_growth(device, True)

class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_hidden_layers):
        super().__init__()

        # saves parameters so that they can be saved and loaded later
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_hidden_layers = num_hidden_layers

```

```
        layers = [nn.Linear(input_size, hidden_size), nn.ReLU()]
        for _ in range(num_hidden_layers - 1):
            layers += [nn.Linear(hidden_size, hidden_size), nn.ReLU()]
        layers.append(nn.Linear(hidden_size, output_size))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)


class LSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size, output_size, num_layers=1, dropout=0.0):
        super().__init__()

        # saves parameters so that they can be saved and loaded later
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers
        self.dropout = dropout

        # Layers
        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=embed_dim,
            padding_idx=0
        )
        self.lstm = nn.LSTM(
            input_size=embed_dim,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0.0
        )
        self.fc = nn.Linear(hidden_size, output_size)

    # B: Batch size -           number of sequences processed in parallel
    # L: Sequence length -      number of time steps (tokens) in each sequence
    # D: Embedding dimension -  size of each 'tokens embedding vector
    # H: Hidden size -          number of features in the LSTM hidden state
    # C: Number of classes -    dimensionality of the output logits

    def forward(self, x):
        # x: LongTensor of shape [B, L] or [B, L, 1]
        if x.dim() == 3 and x.size(2) == 1:
            x = x.squeeze(2)                    # remove channel dimension -> [B, L]

        emb = self.embedding(x)                 # embeddings -> [B, L, D]

        # LSTM returns:
        # - output: hidden state at each time step -> [B, L, H]
        # - hidden: final hidden state for each layer -> [num_layers, B, H]
        # not used as we only need the last hidden state, but can be useful for debugging
        output, (hidden, _) = self.lstm(emb)

        # hidden[-1] selects the final hidden state of the top (last) layer
        # at the last time step -> [B, H]
        last_hidden = hidden[-1]

        # apply the fully-connected layer to get logits -> [B, C]
        logits = self.fc(last_hidden)

        return logits



def train_torch_ffnn(model, args, train_ds):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    optimizer = optim.Adam(
        model.parameters(),
        lr=config.learning_rate,
        betas=(config.beta_1, config.beta_2),
        eps=config.epsilon,
        amsgrad=config.amsgrad
    )
    criterion = nn.CrossEntropyLoss()
    model.train()

    best_val_acc = 0
    patience_counter = 0
    patience_limit = 250

    train_iter = 0
    train_epoch = 0
    start_time = time.time()

```

```
    val_data_created = False
    x_val = y_val = None

    for epoch in range(args.epochs):
        while train_ds.iteration < args.max_iter:
            training_batches = next(train_ds)
            for training_batch in training_batches:
                statistics, labels = training_batch.items()
                stats_np = statistics.numpy()
                labels_np = labels.numpy()

                if not val_data_created:
                    x_train_np, x_val_np, y_train_np, y_val_np = train_test_split(
                        stats_np, labels_np, test_size=0.3
                    )
                    x_val = torch.tensor(x_val_np, dtype=torch.float32).to(device)
                    y_val = torch.tensor(y_val_np, dtype=torch.long).to(device)
                    val_data_created = True
                else:
                    x_train_np = stats_np
                    y_train_np = labels_np

                # Use DataLoader for creating minibatch
                x_train = torch.tensor(x_train_np, dtype=torch.float32)
                y_train = torch.tensor(y_train_np, dtype=torch.long)

                train_dataset = TensorDataset(x_train, y_train)
                train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)

                batch_losses = []

                for x_batch, y_batch in train_loader:
                    x_batch = x_batch.to(device)
                    y_batch = y_batch.to(device)

                    optimizer.zero_grad()
                    outputs = model(x_batch)
                    loss = criterion(outputs, y_batch)
                    loss.backward()
                    optimizer.step()

                    batch_losses.append(loss.item())
                    train_iter += len(y_batch)

                epoch_loss = sum(batch_losses) / len(batch_losses)

                # --- Validation step ---
                model.eval()
                with torch.no_grad():
                    val_outputs = model(x_val)
                    val_loss = criterion(val_outputs, y_val)
                    val_pred = torch.argmax(val_outputs, dim=1)
                    val_acc = (val_pred == y_val).float().mean().item()

                    top3 = torch.topk(val_outputs, k=3, dim=1).indices
                    y_val_exp = y_val.unsqueeze(1).expand_as(top3)
                    val_k3 = (top3 == y_val_exp).any(dim=1).float().mean().item()

                print(f"Epoch: {epoch+1}, Iteration: {train_iter}, "
                      f"Train Loss: {epoch_loss:.4f}, Val Loss: {val_loss.item():.4f}, "
                      f"Val Acc: {val_acc:.4f}, Val Top-3 Acc: {val_k3:.4f}")
                model.train()

                """
                # --- Early stopping check ---
                if val_acc > best_val_acc:
                    best_val_acc = val_acc
                    patience_counter = 0
                else:
                    patience_counter += 1
                    if patience_counter >= patience_limit:
                        print("Early stopping triggered.")
                        elapsed = time.time() - start_time
                        t = time.gmtime(elapsed)
                        print(f"Finished training in {t.tm_yday - 1} days {t.tm_hour} hours {t.tm_min} minutes {t.tm_sec} seconds with {
                            ↪ train_iter} iterations.")
                        class DummyEarlyStopping: stop_training = True
                        return DummyEarlyStopping(), train_iter, f"Early stopped at epoch {epoch+1}"
                """

                if train_iter >= args.max_iter:
                    break
            if train_iter >= args.max_iter:
                break
        train_epoch += 1

    elapsed = time.time() - start_time
    t = time.gmtime(elapsed)
    print(f"Finished training in {t.tm_yday - 1} days {t.tm_hour} hours {t.tm_min} minutes {t.tm_sec} seconds with {train_iter} iterations.
        ↪ ")
```

```
    class DummyEarlyStopping: stop_training = False
    return DummyEarlyStopping(), train_iter, f"Trained for {train_epoch} epochs"


def train_torch_lstm(model, args, train_ds):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    optimizer = optim.Adam(
        model.parameters(),
        lr=config.learning_rate,
        betas=(config.beta_1, config.beta_2),
        eps=config.epsilon,
        amsgrad=config.amsgrad
    )
    criterion = nn.CrossEntropyLoss()
    model.train()

    best_val_acc = 0
    patience_counter = 0
    patience_limit = 250

    train_iter = 0
    train_epoch = 0
    start_time = time.time()

    val_data_created = False
    x_val = y_val = None

    for epoch in range(args.epochs):
        while train_ds.iteration < args.max_iter:
            training_batches = next(train_ds)
            for training_batch in training_batches:
                statistics, labels = training_batch.items()
                stats_np = statistics.numpy().astype(int)
                labels_np = labels.numpy()

                if not val_data_created:
                    x_train_np, x_val_np, y_train_np, y_val_np = train_test_split(stats_np, labels_np, test_size=0.3)
                    x_val = torch.tensor(x_val_np, dtype=torch.long).to(device)
                    y_val = torch.tensor(y_val_np, dtype=torch.long).to(device)
                    val_data_created = True
                else:
                    x_train_np = stats_np
                    y_train_np = labels_np

                x_train = torch.tensor(x_train_np, dtype=torch.long)
                y_train = torch.tensor(y_train_np, dtype=torch.long)

                train_dataset = TensorDataset(x_train, y_train)
                train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)

                batch_losses = []
                for x_batch, y_batch in train_loader:
                    x_batch = x_batch.to(device)
                    y_batch = y_batch.to(device)

                    optimizer.zero_grad()
                    outputs = model(x_batch)
                    loss = criterion(outputs, y_batch)
                    loss.backward()
                    optimizer.step()

                    batch_losses.append(loss.item())
                    train_iter += len(y_batch)

                epoch_loss = sum(batch_losses) / len(batch_losses)

                # --- Validation step ---
                model.eval()
                with torch.no_grad():
                    val_outputs = model(x_val)
                    val_loss = criterion(val_outputs, y_val)
                    val_pred = torch.argmax(val_outputs, dim=1)
                    val_acc = (val_pred == y_val).float().mean().item()

                    top3 = torch.topk(val_outputs, k=3, dim=1).indices
                    y_val_exp = y_val.unsqueeze(1).expand_as(top3)
                    val_k3 = (top3 == y_val_exp).any(dim=1).float().mean().item()

                print(f"Epoch: {epoch+1}, Iteration: {train_iter}, "
                      f"Train Loss: {epoch_loss:.4f}, Val Loss: {val_loss.item():.4f}, "
                      f"Val Acc: {val_acc:.4f}, Val Top-3 Acc: {val_k3:.4f}")
                model.train()

                """
                # --- Early stopping check ---
                if val_acc > best_val_acc:
                    best_val_acc = val_acc
```

```
                    patience_counter = 0
                else:
                    patience_counter += 1
                    if patience_counter >= patience_limit:
                        print("Early stopping triggered.")
                        elapsed = time.time() - start_time
                        t = time.gmtime(elapsed)
                        print(f"Finished training in {t.tm_yday - 1} days {t.tm_hour} hours {t.tm_min} minutes {t.tm_sec} seconds with {
                            ↪ train_iter} iterations.")
                        class DummyEarlyStopping: stop_training = True
                        return DummyEarlyStopping(), train_iter, f"Early stopped at epoch {epoch+1}"
                """

                if train_iter >= args.max_iter:
                    break
            if train_iter >= args.max_iter:
                break
        train_epoch += 1

    elapsed = time.time() - start_time
    t = time.gmtime(elapsed)
    print(f"Finished {train_epoch} epochs in {t.tm_hour}h {t.tm_min}m {t.tm_sec}s")
    class DummyEarlyStopping: stop_training = False
    return DummyEarlyStopping(), train_iter, f"Trained for {train_epoch} epochs"



def predict_torch_ffnn(model, test_ds, args):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()
    model.to(device)
    criterion = nn.CrossEntropyLoss()

    all_preds = []
    all_labels = []

    with torch.no_grad():
        while test_ds.iteration < args.max_iter:
            testing_batches = next(test_ds)
            for testing_batch in testing_batches:
                statistics, labels = testing_batch.items()
                stats_np = statistics.numpy()

                x = torch.tensor(stats_np, dtype=torch.float32).to(device)
                y = torch.tensor(labels.numpy(), dtype=torch.long).to(device)

                outputs = model(x)
                loss = criterion(outputs, y)

                pred_top1 = torch.argmax(outputs, dim=1)
                acc = (pred_top1 == y).float().mean().item()

                top3 = torch.topk(outputs, k=3, dim=1).indices
                y_expanded = y.unsqueeze(1).expand_as(top3)
                k3_acc = (top3 == y_expanded).any(dim=1).float().mean().item()

                print(f"Eval → Loss: {loss.item():.4f}, Accuracy: {acc:.4f}, Top-3 Accuracy: {k3_acc:.4f}")

                preds = torch.softmax(outputs, dim=1).cpu().numpy()
                all_preds.append(preds)
                all_labels.append(labels.numpy())

    all_preds = np.concatenate(all_preds, axis=0)
    all_labels = np.concatenate(all_labels, axis=0)

    return all_preds, all_labels



def predict_torch_lstm(model, test_ds, args):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()
    model.to(device)
    criterion = nn.CrossEntropyLoss()

    all_preds = []
    all_labels = []

    with torch.no_grad():
        while test_ds.iteration < args.max_iter:
            testing_batches = next(test_ds)
            for testing_batch in testing_batches:
                statistics, labels = testing_batch.items()

                stats_np = statistics.numpy().astype(int)  # input tokenizzati
                x = torch.tensor(stats_np, dtype=torch.long).to(device)
                y = torch.tensor(labels.numpy(), dtype=torch.long).to(device)

                outputs = model(x)
```

```
                loss = criterion(outputs, y)

                preds = torch.softmax(outputs, dim=1).cpu().numpy()
                all_preds.append(preds)
                all_labels.append(labels.numpy())

    all_preds = np.concatenate(all_preds, axis=0)
    all_labels = np.concatenate(all_labels, axis=0)

    return all_preds, all_labels



def str2bool(v):
    return v.lower() in ("yes", "true", "t", "1")


def create_model_with_distribution_strategy(architecture, extend_model, output_layer_size, max_train_len):
    """Creates models depending on the GPU count and on extend_model"""
    print('Creating model...')

    strategy = None
    gpu_count = (len(tf.config.list_physical_devices('GPU')) +
        len(tf.config.list_physical_devices('XLA_GPU')))
    if gpu_count > 1:
        print("Multiple GPUs found.")
        strategy = tf.distribute.MirroredStrategy()
        print(f"Number of mirrored devices: {strategy.num_replicas_in_sync}.")
        with strategy.scope():
            if extend_model is not None:
                extend_model = tf.keras.models.load_model(extend_model, compile=False)
            model = create_model(architecture, extend_model, output_layer_size, max_train_len)
        if architecture in ("FFNN", "CNN", "LSTM", "Transformer") and extend_model is None:
            if hasattr(model, "summary"):
                model.summary()
            else:
                # for LSTM use a LongTensor dummy input of shape (1, max_train_len)
                if architecture == "LSTM":
                    summary(model, input_size=(1, max_train_len), dtypes=[torch.long])
                else:
                    summary(model, input_size=(1, 724))

    else:
        print("Only one GPU found.")
        strategy = NullDistributionStrategy()
        if extend_model is not None:
            extend_model = tf.keras.models.load_model(extend_model, compile=False)
        model = create_model(architecture, extend_model, output_layer_size, max_train_len)
        if architecture in ("FFNN", "CNN", "LSTM", "Transformer") and extend_model is None:
            if hasattr(model, "summary"):
                model.summary()
            else:
                # for LSTM use a LongTensor dummy input of shape (1, max_train_len)
                if architecture == "LSTM":
                    summary(model, input_size=(1, max_train_len), dtypes=[torch.long])
                else:
                    summary(model, input_size=(1, 724))


    print('Model created.\n')
    return model, strategy

def create_model(architecture, extend_model, output_layer_size, max_train_len):
    """
    Creates an un-trained model to use in the training process.

    The kind of model that is returned, depends on the provided architecture and
    the `extend_model` flag.

    Parameters
    ----------
    architecture : str
        The architecture of the model to create.
    extend_model
        When `extend_model` is not None and architecure in ('FFNN', 'CNN', 'LSTM'),
        the `extend_model` will be further trained.
    output_layer_size : int
        Defines the size of the output layer of the neural networks

    """
    optimizer = Adam(
        learning_rate=config.learning_rate, beta_1=config.beta_1, beta_2=config.beta_2,
        epsilon=config.epsilon, amsgrad=config.amsgrad)

    # Depends on the number of features returned by `calculate_statistics()` in
    # `featureCalculations.py`.
    input_layer_size = 724
    hidden_layer_size = int(2 * (input_layer_size / 3) + output_layer_size)

```

```
    # Create a model based on an existing one for further trainings
    if extend_model is not None:
        # remove the last layer
        model = tf.keras.Sequential()
        for layer in extend_model.layers[:-1]:
            model.add(layer)
        model.add(tf.keras.layers.Dense(output_layer_size, activation='softmax', name="output"))
        model.compile(optimizer=optimizer, loss="sparse_categorical_crossentropy",
                      metrics=["accuracy", SparseTopKCategoricalAccuracy(k=3, name="k3_accuracy")])
        return model

    # Create new model based on architecture
    if architecture == "FFNN":
        # Use PyTorch for FFNN
        model = FFNN(
            input_size=input_layer_size,
            hidden_size=hidden_layer_size,
            output_size=output_layer_size,
            num_hidden_layers=config.hidden_layers
        )
        return model

    elif architecture == "CNN":
        config.FEATURE_ENGINEERING = False
        config.PAD_INPUT = True
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Conv1D(
                filters=config.filters, kernel_size=config.kernel_size,
                input_shape=(max_train_len, 1), activation='relu'))
        for _ in range(config.layers - 1):
            model.add(tf.keras.layers.Conv1D(filters=config.filters, kernel_size=config.kernel_size, activation='relu'))
        # model_.add(tf.keras.layers.Dropout(0.2))
        model.add(tf.keras.layers.MaxPooling1D(pool_size=2))
        model.add(tf.keras.layers.Flatten())
        model.add(tf.keras.layers.Dense(output_layer_size, activation='softmax'))
        model.compile(optimizer=optimizer, loss="sparse_categorical_crossentropy",
                      metrics=["accuracy", SparseTopKCategoricalAccuracy(k=3, name="k3_accuracy")])
        return model

    elif architecture == "LSTM":
        config.FEATURE_ENGINEERING = False
        config.PAD_INPUT = True
        model = LSTM(
            vocab_size=56,
            embed_dim=64,
            hidden_size=config.lstm_units,
            output_size=output_layer_size,
            num_layers=1,
            dropout=0.0
        )
        return model


    elif architecture == "DT":
        return DecisionTreeClassifier(criterion=config.criterion, ccp_alpha=config.ccp_alpha)

    elif architecture == "NB":
        return MultinomialNB(alpha=config.alpha, fit_prior=config.fit_prior)

    elif architecture == "RF":
        return RandomForestClassifier(n_estimators=config.n_estimators, criterion=config.criterion,
                                      bootstrap=config.bootstrap, n_jobs=30,
                                      max_features=config.max_features, max_depth=30,
                                      min_samples_split=config.min_samples_split,
                                      min_samples_leaf=config.min_samples_leaf)

    elif architecture == "ET":
        return ExtraTreesClassifier(n_estimators=config.n_estimators, criterion=config.criterion,
                                    bootstrap=config.bootstrap, n_jobs=30,
                                    max_features=config.max_features, max_depth=30,
                                    min_samples_split=config.min_samples_split,
                                    min_samples_leaf=config.min_samples_leaf)

    elif architecture == "Transformer":
        config.FEATURE_ENGINEERING = False
        config.PAD_INPUT = True
        vocab_size = config.vocab_size
        maxlen = max_train_len
        embed_dim = config.embed_dim  # Embedding size for each token
        num_heads = config.num_heads  # Number of attention heads
        ff_dim = config.ff_dim  # Hidden layer size in feed forward network inside transformer

        inputs = tf.keras.layers.Input(shape=(maxlen,))
        embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
        x = embedding_layer(inputs)
        transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
        x = transformer_block(x)
        x = tf.keras.layers.GlobalAveragePooling1D()(x)
        outputs = tf.keras.layers.Dense(output_layer_size, activation="softmax")(x)
```

```
596
597         model = tf.keras.Model(inputs=inputs, outputs=outputs)
598         model.compile(optimizer=optimizer, loss="sparse_categorical_crossentropy",
599                       metrics=["accuracy", SparseTopKCategoricalAccuracy(k=3, name="k3_accuracy")])
600         return model
601
602     elif architecture == "SVM":
603         return SVC(probability=True, C=1, gamma=0.001, kernel="linear")
604
605     elif architecture == "SVM-Rotor":
606         pipe = Pipeline([
607             ('scale', StandardScaler()),
608             ('clf', SVC(probability=True, C=10, gamma=0.001, kernel="rbf"))])
609
610         return pipe
611
612     elif architecture == "kNN":
613         return KNeighborsClassifier(90, weights="distance", metric="euclidean")
614
615     elif architecture == "[FFNN,NB]":
616         model_ffnn = tf.keras.Sequential()
617         model_ffnn.add(tf.keras.layers.Input(shape=(input_layer_size,)))
618         for _ in range(config.hidden_layers):
619             model_ffnn.add(tf.keras.layers.Dense(hidden_layer_size, activation='relu', use_bias=True))
620         model_ffnn.add(tf.keras.layers.Dense(output_layer_size, activation='softmax'))
621         model_ffnn.compile(optimizer=optimizer, loss="sparse_categorical_crossentropy",
622                       metrics=["accuracy", SparseTopKCategoricalAccuracy(k=3, name="k3_accuracy")])
623         model_nb = MultinomialNB(alpha=config.alpha, fit_prior=config.fit_prior)
624         return [model_ffnn, model_nb]
625
626     elif architecture == "[DT,ET,RF,SVM,kNN]":
627         dt = DecisionTreeClassifier(criterion=config.criterion, ccp_alpha=config.ccp_alpha)
628         et = ExtraTreesClassifier(n_estimators=config.n_estimators, criterion=config.criterion,
629                                   bootstrap=config.bootstrap, n_jobs=30,
630                                   max_features=config.max_features, max_depth=30,
631                                   min_samples_split=config.min_samples_split,
632                                   min_samples_leaf=config.min_samples_leaf)
633         rf = RandomForestClassifier(n_estimators=config.n_estimators, criterion=config.criterion,
634                                     bootstrap=config.bootstrap, n_jobs=30,
635                                     max_features=config.max_features, max_depth=30,
636                                     min_samples_split=config.min_samples_split,
637                                     min_samples_leaf=config.min_samples_leaf)
638         svm = SVC(probability=True, C=1, gamma=0.001, kernel="linear")
639         knn = KNeighborsClassifier(90, weights="distance", metric="euclidean")
640         return [dt, et, rf, svm, knn]
641
642     else:
643         raise Exception(f"Could not create model. Unknown architecture '{architecture}'.")
644
645 def parse_arguments():
646     parser = argparse.ArgumentParser(
647         description='CANN Ciphertype Detection Neuronal Network Training Script',
648         formatter_class=argparse.RawTextHelpFormatter)
649     parser.add_argument('--batch_size', default=128, type=int,
650                         help='Batch size for training.')
651     parser.add_argument('--train_dataset_size', default=16000, type=int,
652                         help='Dataset size per fit. This argument should be dividable \n'
653                              'by the amount of --ciphers.')
654     parser.add_argument('--dataset_workers', default=1, type=int,
655                         help='The number of parallel workers for reading the \ninput files.')
656     parser.add_argument('--epochs', default=1, type=int,
657                         help='Defines how many times the same data is used to fit the model.')
658     parser.add_argument('--plaintext_input_directory', default='../data/gutenberg_en', type=str,
659                         help='Input directory of the plaintexts for training the aca ciphers.')
660     parser.add_argument('--rotor_input_directory', default='../data/rotor_ciphertexts', type=str,
661                         help='Input directory of the rotor ciphertexts.')
662     parser.add_argument('--download_dataset', default=True, type=str2bool,
663                         help='Download the dataset automatically.')
664     parser.add_argument('--save_directory', default='../data/models/',
665                         help='Directory for saving generated models. \n'
666                              'When interrupting, the current model is \n'
667                              'saved as interrupted_...')
668     parser.add_argument('--model_name', default='m.h5', type=str,
669                         help='Name of the output model file. The file must \nhave the .h5 extension.')
670     parser.add_argument('--ciphers', default='all', type=str,
671                         help='A comma seperated list of the ciphers to be created.\n'
672                              'Be careful to not use spaces or use \' to define the string.\n'
673                              'Possible values are:\n'
674                              '- mtc3 (contains the ciphers Monoalphabetic Substitution, Vigenere,\n'
675                              '        Columnar Transposition, Plaifair and Hill)\n'
676                              '- aca (contains all currently implemented ciphers from \n'
677                              '       https://www.cryptogram.org/resource-area/cipher-types/)\n'
678                              '- rotor (contains Enigma, M209, Purple, Sigaba and Typex ciphers)'
679                              '- all (contains aca and rotor ciphers)'
680                              '- all aca ciphers in lower case'
681                              '- simple_substitution\n'
682                              '- vigenere\n'
683                              '- columnar_transposition\n'
684                              '- playfair\n'
```

```
                            '- hill\n')
    parser.add_argument('--keep_unknown_symbols', default=False, type=str2bool,
                        help='Keep unknown symbols in the plaintexts. Known \n'
                            'symbols are defined in the alphabet of the cipher.')
    parser.add_argument('--max_iter', default=1000000, type=int,
                        help='the maximal number of iterations before stopping training.')
    parser.add_argument('--min_train_len', default=50, type=int,
                        help='The minimum length of a plaintext to be encrypted in training. \n'
                            'If this argument is set to -1 no lower limit is used.')
    parser.add_argument('--min_test_len', default=50, type=int,
                        help='The minimum length of a plaintext to be encrypted in testing. \n'
                            'If this argument is set to -1 no lower limit is used.')
    parser.add_argument('--max_train_len', default=-1, type=int,
                        help='The maximum length of a plaintext to be encrypted in training. \n'
                            'If this argument is set to -1 no upper limit is used.')
    parser.add_argument('--max_test_len', default=-1, type=int,
                        help='The maximum length of a plaintext to be encrypted in testing. \n'
                            'If this argument is set to -1 no upper limit is used.')
    parser.add_argument('--architecture', default='FFNN', type=str,
                        choices=['FFNN', 'CNN', 'LSTM', 'DT', 'NB', 'RF', 'ET', 'Transformer',
                                 'SVM', 'kNN', '[FFNN,NB]', '[DT,ET,RF,SVM,kNN]', 'SVM-Rotor'],
                        help='The architecture to be used for training. \n'
                            'Possible values are:\n'
                            '- FFNN\n'
                            '- CNN\n'
                            '- LSTM\n'
                            '- DT\n'
                            '- NB\n'
                            '- RF\n'
                            '- ET\n'
                            '- Transformer\n'
                            '- SVM\n'
                            '- kNN\n'
                            '- [FFNN,NB]\n'
                            '- [DT,ET,RF,SVM,kNN]'
                            '- SVM-Rotor'
                            )
    parser.add_argument('--extend_model', default=None, type=str,
                        help='Load a trained model from a file and use it as basis for the new training.')

    return parser.parse_args()

def should_download_plaintext_datasets(args):
    """Determines if the plaintext datasets should be loaded"""
    return (args.download_dataset and
            not os.path.exists(args.plaintext_input_directory) and
            args.plaintext_input_directory == os.path.abspath('../data/gutenberg_en'))

def download_plaintext_datasets(args):
    """Downloads plaintexts and saves them in the plaintext_input_directory"""
    print("Downloading Datsets...")
    checksums_dir = '../data/checksums/'
    if not Path(checksums_dir).exists():
        os.mkdir(checksums_dir)
    tfds.download.add_checksums_dir(checksums_dir)

    download_manager = tfds.download.download_manager.DownloadManager(download_dir='../data/',
                                                                      extract_dir=args.plaintext_input_directory)
    data_url = ('https://drive.google.com/uc?id=1bF5sSVjxTxa3DB-P5wxn87nxWndRhK_V&export=download' +
        '&confirm=t&uuid=afbc362d-9d52-472a-832b-c2af331a8d5b')
    try:
        download_manager.download_and_extract(data_url)
    except Exception as e:
        print("Download of datasets failed. If this issues persists, try downloading the dataset yourself "
              "from: https://drive.google.com/file/d/1bF5sSVjxTxa3DB-P5wxn87nxWndRhK_V/view."
              "(For more information see the README.md of this project.)")
        print("Underlying error:")
        print(e)
        sys.exit(1)

    path = os.path.join(args.plaintext_input_directory,
                        'ZIP.ucid_1bF5sSVjxTx-P5wxn87nxWn_V_export_downloadR9Cwhunev5CvJ-ic__'
                        'HawxhTtGOlSdcCrro4fxfEI8A',
                        os.path.basename(args.plaintext_input_directory))
    dir_name = os.listdir(path)
    for name in dir_name:
        p = Path(os.path.join(path, name))
        parent_dir = p.parents[2]
        p.rename(parent_dir / p.name)
    os.rmdir(path)
    os.rmdir(os.path.dirname(path))
    print("Datasets Downloaded.")

def load_rotor_ciphertext_datasets_from_disk(args, batch_size):
    """
    Load ciphertext input data for rotor ciphertexts from disk.

    This method (in contrast to `load_plaintext_datasets_from_disk`) loads the data
    immediately from disk. Currently this does not take too long, since the input files
```

```
    and the number of ciphers that need ciphertexts as input for the feature extraction
    are limited. (If this method should lazily load ciphertexts, `RotorCiphertextsDataset`
    needs to be adapted.)

    Parameters
    ----------
    args :
        The arguments parsed by the `ArgumentParser`. See `parse_arguments()`.
    batch_size : int
            The number of samples and labels per batch.

    Returns
    -------
    tuple[list, RotorCiphertextsDatasetParameters, list, RotorCiphertextsDatasetParameters]
        A tuple with the training and testing ciphertexts as well as their
        `RotorCiphertextsDatasetParameters` that are both provided to the
        `CipherStatisticsDataset`s for training and testing.
    """

    def validate_ciphertext_path(ciphertext_path, cipher_types):
        """Check if the filename of the file at `ciphertext_path` matches the
        names inside `cipher_types`."""
        file_name = Path(ciphertext_path).stem.lower()
        if not file_name in cipher_types:
            raise Exception(f"Filename must equal one of the expected cipher types. "
                            f"Expected cipher types are: {cipher_types}. Current "
                            f"filename is '{file_name}'.")


    # Filter cipher_types to exclude non-ciphertext ciphers

    # Filter cipher_types to exclude non-ciphertext ciphers
    rotor_cipher_types = [config.CIPHER_TYPES[i] for i in range(56, 61)]

    # Load all ciphertexts in the train and dev folders in `args.rotor_input_directory`
    rotor_cipher_dir = args.rotor_input_directory

    def find_ciphertext_paths_in_dir(folder_path):
        """Loads all .txt files in the given folder and checks that their names match the
        known cipher types."""
        file_names = os.listdir(folder_path)
        result = []
        for name in file_names:
            path = os.path.join(folder_path, name)
            file_name, file_type = os.path.splitext(name)
            if os.path.isfile(path) and file_name.lower() in rotor_cipher_types and file_type == ".txt":
                validate_ciphertext_path(path, config.ROTOR_CIPHER_TYPES)

                result.append(path)
        return result

    train_dir = os.path.join(rotor_cipher_dir, "train")
    test_dir = os.path.join(rotor_cipher_dir, "dev")
    train_rotor_ciphertext_paths = find_ciphertext_paths_in_dir(train_dir)
    test_rotor_cipherext_paths = find_ciphertext_paths_in_dir(test_dir)


    # Return empty lists and parameters if no requested ciphers were found on disk
    if len(train_rotor_ciphertext_paths) == 0 or len(test_rotor_cipherext_paths) == 0:
        empty_params = RotorCiphertextsDatasetParameters(config.ROTOR_CIPHER_TYPES,
                                                         0,
                                                         args.dataset_workers,
                                                         args.min_train_len,
                                                         args.max_train_len,
                                                         generate_evalutation_data=False)
        return ([], empty_params, [], empty_params)

    # Create dataset parameters, which will be used for creating a `CipherStatisticsDataset`.
    # This class will provide an iterator in `train_model` to convert the plaintext files
    # (applying the provided options) into statistics (features) used for training.
    train_rotor_ciphertexts_parameters = RotorCiphertextsDatasetParameters(config.ROTOR_CIPHER_TYPES,
                                                           batch_size,
                                                           args.dataset_workers,
                                                           args.min_train_len,
                                                           args.max_train_len,
                                                           generate_evalutation_data=False)
    test_rotor_ciphertexts_parameters = RotorCiphertextsDatasetParameters(config.ROTOR_CIPHER_TYPES,
                                                           batch_size,
                                                           args.dataset_workers,
                                                           args.min_test_len,
                                                           args.max_test_len,
                                                           generate_evalutation_data=False)

    # Return the tuples of training and testing rotor_ciphertexts as well as the parameter
    # for initializing the `CipherStatisticsDataset`s.
    return (train_rotor_ciphertext_paths, train_rotor_ciphertexts_parameters,
            test_rotor_cipherext_paths, test_rotor_ciphertexts_parameters)

def load_plaintext_datasets_from_disk(args, requested_cipher_types, batch_size):
```

```
    """
    Gets all plaintext paths found in `args.plaintext_input_directory`, and converts
    them into training and testing list and parameters, used to create
    `CipherStatisticsDataset`s.

    This method does not load the contents of the plaintext files. This is done
    lazily by the `CipherStatisticsDataset`.

    Parameters
    ----------
        args :
            The arguments parsed by the `ArgumentParser`. See `parse_arguments()`.
        requested_cipher_types : list
            A list of cipher types to provide as parameters to the `CipherStatisticsDataset`.
            The list is filtered and only ACA ciphers are used as parameters, since the features
            of rotor ciphers currently have to be extracted from ciphertext files.
        batch_size : int
            The number of samples and labels per batch.

    Returns
    -------
    tuple[list, PlaintextPathsDatasetParameters, list, PlaintextPathsDatasetParameters]
        A tuple with the training and testing plaintext paths as well as their
        `PlaintextPathsDatasetParameters` that are both provided to the
        `CipherStatisticsDataset`s for training and testing.
    """
    # Filter cipher_types to exclude non-plaintext ciphers
    aca_cipher_types = [config.CIPHER_TYPES[i] for i in range(56)]
    cipher_types = [type for type in requested_cipher_types if type in aca_cipher_types]

    # Get all paths to plaintext files in the `plaintext_input_directory`
    plaintext_files = []
    dir_name = os.listdir(args.plaintext_input_directory)
    for name in dir_name:
        path = os.path.join(args.plaintext_input_directory, name)
        if os.path.isfile(path):
            plaintext_files.append(path)

    # Use some plaintext for training and others for testing
    train_plaintexts, test_plaintexts = train_test_split(plaintext_files, test_size=0.05,
                                                         random_state=42, shuffle=True)

    # Create dataset parameters, which will be used for creating a `CipherStatisticsDataset`.
    # This class will provide an iterator in `train_model` to convert the plaintext files
    # (applying the provided options) into statistics (features) used for training.
    train_plaintext_parameters = PlaintextPathsDatasetParameters(cipher_types, batch_size,
                                            args.min_train_len, args.max_train_len,
                                            args.keep_unknown_symbols, args.dataset_workers)
    test_plaintext_parameters = PlaintextPathsDatasetParameters(cipher_types, batch_size,
                                           args.min_test_len, args.max_test_len,
                                           args.keep_unknown_symbols, args.dataset_workers)

    # Return the training and testing plaintexts as well as their parameters
    return (train_plaintexts, train_plaintext_parameters, test_plaintexts, test_plaintext_parameters)

def load_datasets_from_disk(args, requested_cipher_types):
    """
    Loads training and testing data from the file system.

    In case of the ACA ciphers the datasets are plaintext files that need to be
    encrypted before the features can be extracted. In case of the rotor ciphers
    there are already encrypted ciphertext files that can directly be used to
    extract the features.
    To simplify the training code, both kinds of input data are returned in
    `CipherStatisticsDataset`s that provide an iterator interface, returning
    `TrainingBatch`es of the requested size on each `next()` call.
    Plaintext input is loaded lazily, while ciphertexts currently are loaded
    immediately.

    Parameters
    ----------
    args :
        The parsed commandline arguments. See also `parse_arguments()`.
    requested_cipher_types : list
        A list of the requested cipher types. These are provided as parameters to
        the returned `CipherStatisticsDataset`s as well as for selection of input
        files.
    max_rotor_lines : int
        Limits the amount of input lines loaded from ciphertext files.

    Returns
    -------
    tuple[CipherStatisticsDataset]
        Training and testing `CipherStatisticsDataset` that lazily calculate the
        features for the input data on `next()` calls.
    """

    print("Loading Datasets...")

```

```
952     # Filter cipher_types to exclude non-ciphertext ciphers
953     rotor_cipher_types = [config.CIPHER_TYPES[i] for i in range(56, 61)]
954     non_rotor_ciphers = [type for type in requested_cipher_types if type not in rotor_cipher_types]
955     rotor_cipher_types = [type for type in requested_cipher_types if type in rotor_cipher_types]
956 
957     # Calculate batch size for rotor ciphers. If both aca and rotor ciphers are requested,
958     # the amount of samples of each rotor cipher per batch should be equal to the
959     # amount of samples of each aca cipher per loaded batch.
960     number_of_rotor_ciphers = len(rotor_cipher_types)
961     number_of_aca_ciphers = len(non_rotor_ciphers)
962     if number_of_aca_ciphers <= 0:
963         rotor_dataset_batch_size = args.train_dataset_size
964         aca_dataset_batch_size = 0
965     else:
966         amount_of_samples_per_cipher = args.train_dataset_size // (number_of_aca_ciphers + number_of_rotor_ciphers)
967         rotor_dataset_batch_size = amount_of_samples_per_cipher * number_of_rotor_ciphers
968         aca_dataset_batch_size = amount_of_samples_per_cipher * number_of_aca_ciphers
969 
970     # Load the plaintext file paths and the rotor ciphertexts from disk.
971     (train_plaintexts,
972      train_plaintext_parameters,
973      test_plaintexts,
974      test_plaintext_parameters) = load_plaintext_datasets_from_disk(args,
975                                                                     requested_cipher_types,
976                                                                     aca_dataset_batch_size)
977     (train_rotor_ciphertext_paths,
978      train_rotor_ciphertexts_parameters,
979      test_rotor_ciphertext_paths,
980      test_rotor_ciphertexts_parameters) = load_rotor_ciphertext_datasets_from_disk(args,
981                                                                                   rotor_dataset_batch_size)
982 
983     # Convert the training and testing ciphertexts and plaintexts, as well as
984     # their parameters into `CipherStatisticsDataset`s.
985     train_ds = CipherStatisticsDataset(train_plaintexts, train_plaintext_parameters, train_rotor_ciphertext_paths,
            ↪ train_rotor_ciphertexts_parameters)
986     test_ds = CipherStatisticsDataset(test_plaintexts, test_plaintext_parameters, test_rotor_ciphertext_paths,
            ↪ test_rotor_ciphertexts_parameters)
987 
988     if train_ds.key_lengths_count > 0 and args.train_dataset_size % train_ds.key_lengths_count != 0:
989         print("WARNING: the --train_dataset_size parameter must be dividable by the amount of --ciphers  and the length configured "
990               "KEY_LENGTHS in config.py. The current key_lengths_count is %d" %
991                   train_ds.key_lengths_count, file=sys.stderr)
992 
993     print("Datasets loaded.\n")
994 
995     # Return `CipherStatisticsDataset`s for training and testing.
996     return train_ds, test_ds
997 
998 def train_model(model, strategy, args, train_ds):
999     """
1000    Trains the model with the given training dataset.
1001 
1002    Depending on the value of `args.architecture` a different approach is
1003    taken to train the model. Some architectures need to be trained in one
1004    iteration, while others can be trained with multiple input batches.
1005    While the training is in progress, status messages are logged to
1006    stdout to indicate the amount of seen input data as well as the current
1007    accuracy of the trained model.
1008 
1009    Parameters
1010    ----------
1011    model :
1012        The model that will be trained. Needs to match `args.architecture`.
1013    strategy :
1014        A distribution strategy (of the `Tensorflow` library) to distribute
1015        the `fit` calls to multiple devices. Could also be a `NullStrategy`
1016        if no GPU devices are found on the system.
1017    args :
1018        Commandline arguments entered by the user. See also: `parse_arguments()`.
1019    train_ds :
1020        A `CipherStatisticsDataset` providing the features to use for training.
1021 
1022    Returns
1023    -------
1024    tuple
1025    """
1026 
1027    checkpoints_dir = Path('../data/checkpoints')
1028    def delete_previous_checkpoints():
1029        shutil.rmtree(checkpoints_dir)
1030 
1031    def create_checkpoint_callback():
1032        """Provides a `keras` `ModelCheckpoint` used to periodically save a model in training"""
1033        if not checkpoints_dir.exists():
1034            os.mkdir(checkpoints_dir)
1035        checkpoint_file_path = os.path.join(checkpoints_dir,
1036                                            "epoch_{epoch:02d}-"
1037                                            "acc_{accuracy:.2f}.h5")
1038 
1039        return tf.keras.callbacks.ModelCheckpoint(
```

```
            filepath=checkpoint_file_path,
            save_weights_only=False,
            monitor='val_accuracy',
            mode='max',
            save_best_only=False,
            save_freq=100)

    print('Training model...')

    delete_previous_checkpoints()

    # Create callbacks for tensorflow models
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir='../data/logs',
                                                          update_freq='epoch')
    early_stopping_callback = MiniBatchEarlyStopping(min_delta=1e-5,
                                                     patience=250,
                                                     monitor='accuracy',
                                                     mode='max',
                                                     restore_best_weights=True)
    custom_step_decay_lrate_callback = CustomStepDecayLearningRateScheduler(early_stopping_callback)
    checkpoint_callback = create_checkpoint_callback()

    # Initialize variables
    architecture = args.architecture
    start_time = time.time()
    train_iter = 0
    train_epoch = 0
    val_data = None
    val_labels = None
    training_batches = None
    combined_batch = TrainingBatch("mixed", [], [])
    classes = list(range(len(config.CIPHER_TYPES)))
    should_create_validation_data = True

    if args.architecture == "FFNN" and isinstance(model, FFNN):
        return train_torch_ffnn(model, args, train_ds)

    elif args.architecture == "LSTM" and isinstance(model, LSTM):
        return train_torch_lstm(model, args, train_ds)


    # Perform main training loop while the iterations don't exceed the user provided max_iter
    while train_ds.iteration < args.max_iter:
        training_batches = next(train_ds)

        # For architectures that only support one fit call: Sample all batches into one large batch.
        if architecture in ("DT", "RF", "ET", "SVM", "kNN", "SVM-Rotor", "[DT,ET,RF,SVM,kNN]"):
            for training_batch in training_batches:
                combined_batch.extend(training_batch)
            if train_ds.iteration < args.max_iter:
                print("Loaded %d ciphertexts." % train_ds.iteration)
                continue
            train_ds.stop_outstanding_tasks()
            print("Loaded %d ciphertexts." % train_ds.iteration)
            training_batches = [combined_batch]

        for index, training_batch in enumerate(training_batches):
            statistics, labels = training_batch.items()
            train_iter = train_ds.iteration - len(training_batch) * (len(training_batches) - index - 1)

            # Create small validation dataset on first iteration
            if should_create_validation_data:
                statistics, val_data, labels, val_labels = train_test_split(statistics.numpy(),
                                                                            labels.numpy(),
                                                                            test_size=0.3)
                statistics = tf.convert_to_tensor(statistics)
                val_data = tf.convert_to_tensor(val_data)
                labels = tf.convert_to_tensor(labels)
                val_labels = tf.convert_to_tensor(val_labels)
                should_create_validation_data = False
                train_iter -= len(training_batch) * 0.3

            # scikit-learn architectures:
            if architecture in ("DT", "RF", "ET", "SVM", "kNN", "SVM-Rotor"):
                train_iter = len(labels) * 0.7
                print(f"Start training the {architecture}.")
                if architecture == "kNN":
                    history = model.fit(statistics, labels)
                elif architecture == "SVM-Rotor":
                    history = model.fit(list(statistics), list(labels))
                    # print(f"RFE-support: \n{list(model.support_)}\n")
                    # print(f"RFE-rank: \n{list(model.ranking_)}\n")
                    # print()
                else:
                    history = model.fit(statistics, labels)
                if architecture == "DT":
                    plt.gcf().set_size_inches(25, 25 / math.sqrt(2))
                    print("Plotting tree.")
                    plot_tree(model, max_depth=3, fontsize=6, filled=True)
```

```
                plt.savefig(args.model_name.split('.')[0] + '_decision_tree.svg',
                            dpi=200, bbox_inches='tight', pad_inches=0.1)

        # Naive Bayes training
        elif architecture == "NB":
            history = model.partial_fit(statistics,
                                        labels,
                                        classes=classes)

        # Ensemble: [FFNN,NB]
        elif architecture == "[FFNN,NB]":
            with strategy.scope():
                history = model[0].fit(statistics,
                                       labels,
                                       batch_size=args.batch_size,
                                       validation_data=(val_data, val_labels),
                                       epochs=args.epochs,
                                       callbacks=[early_stopping_callback,
                                                  tensorboard_callback,
                                                  custom_step_decay_lrate_callback,
                                                  checkpoint_callback])
            history = model[1].partial_fit(statistics,
                                           labels,
                                           classes=classes)

        # Ensemble: [DT,ET,RF,SVM,kNN]
        elif architecture == "[DT,ET,RF,SVM,kNN]":
            print(f"Start training the {architecture}.")
            dt, et, rf, svm, knn = model
            for index, m in enumerate([dt, et, rf, svm]):
                m.fit(statistics, labels)
                print(f"Trained model {index + 1} of {len(model)}")
            knn.fit(statistics, labels)
            print(f"Trained model {len(model)} of {len(model)}")

        else:
            with strategy.scope():
                history = model.fit(statistics, labels,
                                    batch_size=args.batch_size,
                                    validation_data=(val_data, val_labels),
                                    epochs=args.epochs,
                                    callbacks=[early_stopping_callback,
                                               tensorboard_callback,
                                               custom_step_decay_lrate_callback,
                                               checkpoint_callback])

        # print for Decision Tree, Naive Bayes and Random Forests
        if architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN", "SVM-Rotor"):
            val_score = model.score(val_data, val_labels)
            train_score = model.score(statistics, labels)
            print("train accuracy: %f, validation accuracy: %f" % (train_score, val_score))

        if architecture == "[FFNN,NB]":
            val_score = model[1].score(val_data, val_labels)
            train_score = model[1].score(statistics, labels)
            print("train accuracy: %f, validation accuracy: %f" % (train_score, val_score))

        if architecture == "[DT,ET,RF,SVM,kNN]":
            for m in model:
                val_score = m.score(val_data, val_labels)
                train_score = m.score(statistics, labels)
                print(f"{type(m).__name__}: train accuracy: {train_score}, "
                      f"validation accuracy: {val_score}")

        if train_ds.epoch > 0:
            train_epoch = (train_ds.iteration
                           // ((train_iter + train_ds.batch_size * train_ds.dataset_workers)
                           // train_ds.epoch))

        print("Epoch: %d, Iteration: %d" % (train_epoch, train_iter))
        if train_iter >= args.max_iter or early_stopping_callback.stop_training:
            break

    if train_ds.iteration >= args.max_iter or early_stopping_callback.stop_training:
        train_ds.stop_outstanding_tasks()
        break

elapsed_training_time = datetime.fromtimestamp(time.time()) - datetime.fromtimestamp(start_time)
training_stats = ("Finished training in %d days %d hours %d minutes %d seconds "
                  "with %d iterations and %d epochs.\n"
                  % (elapsed_training_time.days,
                     elapsed_training_time.seconds // 3600,
                     (elapsed_training_time.seconds // 60) % 60,
                     elapsed_training_time.seconds % 60,
                     train_iter,
                     train_epoch))
print(training_stats)
return early_stopping_callback, train_iter, training_stats
```

```
def save_model(model, args):
    """Writes the model and the commandline arguments to disk."""
    print('Saving model...')
    architecture = args.architecture

    if not os.path.exists(args.save_directory):
        os.mkdir(args.save_directory)

    # Gestione nome modello
    if args.model_name == 'm.h5':
        i = 1
        base_name = args.model_name.split('.')[0]
        extension = '.pth' if architecture == "FFNN" else '.h5'
        while os.path.exists(os.path.join(args.save_directory, base_name + str(i) + extension)):
            i += 1
        model_name = base_name + str(i) + extension
    else:
        model_name = args.model_name
        if architecture == "FFNN":
            model_name = model_name.replace('.h5', '.pth')

    model_path = os.path.join(args.save_directory, model_name)

    if architecture in ("FFNN", "LSTM"):
        state_dict = {
            'model_state_dict': model.state_dict(),
            'hidden_size': model.hidden_size,
            'output_size': model.output_size,
        }

        if architecture == "FFNN":
            state_dict['input_size'] = model.input_size
            state_dict['num_hidden_layers'] = model.num_hidden_layers
        elif architecture == "LSTM":
            state_dict['vocab_size'] = model.vocab_size
            state_dict['embed_dim'] = model.embed_dim
            state_dict['num_layers'] = model.num_layers
            state_dict['dropout'] = model.dropout

        torch.save(state_dict, model_path)


    elif architecture in ("CNN", "Transformer"):
        model.save(model_path)

    elif architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN", "SVM-Rotor"):
        with open(model_path, "wb") as f:
            pickle.dump(model, f)

    elif architecture == "[FFNN,NB]":
        model[0].save('../data/models/' + model_path.split('.')[0] + "_ffnn.h5")
        with open('../data/models/' + model_path.split('.')[0] + "_nb.h5", "wb") as f:
            pickle.dump(model[1], f)

    elif architecture == "[DT,ET,RF,SVM,kNN]":
        for index, name in enumerate(["dt", "et", "rf", "svm", "knn"]):
            with open('../data/models/' + model_path.split('.')[0] + f"_{name}.h5", "wb") as f:
                pickle.dump(model[index], f)

    """
    # Saving parameters
    with open('../data/' + model_path.split('.')[0] + '_parameters.txt', 'w') as f:
        for arg in vars(args):
            f.write("{:23s}= {:s}\n".format(arg, str(getattr(args, arg))))

    # Managing logs
    if architecture in ("FFNN", "CNN", "LSTM", "Transformer"):
        logs_destination = '../data/' + model_name.split('.')[0] + '_tensorboard_logs'
        try:
            if os.path.exists('../data/logs'):
                if os.path.exists(logs_destination):
                    shutil.rmtree(logs_destination)
                shutil.move('../data/logs', logs_destination)
        except Exception:
            print(f"Could not move logs from '../data/logs' to '{logs_destination}'.")
    """
    print('Model saved.\n')



def predict_test_data(test_ds, model, args, early_stopping_callback, train_iter):
    """
    Testing the predictions of the model.

    The trained model is used to predict the data in `test_ds` and the results
    are evaluated in regard to accuracy, precision, recall, etc. The calculated
    metrics are printed to stdout.

    Parameters
```

```
    ----------
    test_ds : CipherStatisticsDataset
        The dataset used for prediction.
    model :
        The trained model to evaluate.
    args :
        The commandline arguments provided by the user.
    early_stopping_callback :
        Indicates whether the training was stopped before `args.max_iter` was
        reached. Used together with `train_iter` and `args.max_iter` to control
        the number of prediction iterations.
    train_iter : int
        The number of iterations used until the model converged. Used together with
        `early_stopping_callback` and `args.max_iter` to control the number of
        prediction iterations.

    Returns
    -------
    str
        The statistics of this prediction run.
    """

    print('Predicting test data...\n')

    architecture = args.architecture
    start_time = time.time()
    total_len_prediction = 0
    cntr = 0
    test_iter = 0
    test_epoch = 0

    # Determine the number of iterations to use for evaluating the model
    prediction_dataset_factor = 10
    if early_stopping_callback.stop_training:
        while test_ds.dataset_workers * test_ds.batch_size > train_iter / prediction_dataset_factor and prediction_dataset_factor > 1:
            prediction_dataset_factor -= 1
        args.max_iter = int(train_iter / prediction_dataset_factor)
    else:
        while test_ds.dataset_workers * test_ds.batch_size > args.max_iter / prediction_dataset_factor and prediction_dataset_factor > 1:
            prediction_dataset_factor -= 1
        args.max_iter /= prediction_dataset_factor

    # Initialize `PredictionPerformanceMetrics` instances for all classifiers. These
    # are used to save and evaluate the batched prediction results of the models.
    prediction_metrics = {}
    if architecture == "[FFNN,NB]":
        prediction_metrics = {"FFNN": PredictionPerformanceMetrics(model_name="FFNN"),
                              "NB": PredictionPerformanceMetrics(model_name="NB")}
    elif architecture == "[DT,ET,RF,SVM,kNN]":
        prediction_metrics = {"DT": PredictionPerformanceMetrics(model_name="DT"),
                              "ET": PredictionPerformanceMetrics(model_name="ET"),
                              "RF": PredictionPerformanceMetrics(model_name="RF"),
                              "SVM": PredictionPerformanceMetrics(model_name="SVM"),
                              "kNN": PredictionPerformanceMetrics(model_name="kNN"),}
    else:
        prediction_metrics = {architecture: PredictionPerformanceMetrics(model_name=architecture)}

    combined_batch = TrainingBatch("mixed", [], [])
    while test_ds.iteration < args.max_iter:
        testing_batches = next(test_ds)

        # For architectures that only support one fit call: Sample all batches into one large batch.
        if architecture in ("DT", "RF", "ET", "SVM", "kNN", "SVM-Rotor", "[DT,ET,RF,SVM,kNN]"):
            for testing_batch in testing_batches:
                combined_batch.extend(testing_batch)
            if test_ds.iteration < args.max_iter:
                print("Loaded %d ciphertexts." % test_ds.iteration)
                continue
            test_ds.stop_outstanding_tasks()
            print("Loaded %d ciphertexts." % test_ds.iteration)
            testing_batches = [combined_batch]

        for testing_batch in testing_batches:
            statistics, labels = testing_batch.items()

            # Decision Tree, Naive Bayes prediction
            if architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN"):
                prediction = model.predict_proba(statistics)
                prediction_metrics[architecture].add_predictions(labels, prediction)
            elif architecture == "SVM-Rotor":
                prediction = model.predict_proba(statistics)
                # add probability 0 to all aca labels that are missing in the prediction
                padded_prediction = []
                for p in list(prediction):
                    padded = [0] * 56 + list(p)
                    padded_prediction.append(padded)
                prediction_metrics[architecture].add_predictions(labels, padded_prediction)
            elif architecture == "[FFNN,NB]":
                prediction = model[0].predict(statistics, batch_size=args.batch_size, verbose=1)
```

```
                nb_prediction = model[1].predict_proba(statistics)
                prediction_metrics["FFNN"].add_predictions(labels, prediction)
                prediction_metrics["NB"].add_predictions(labels, nb_prediction)
            elif architecture == "[DT,ET,RF,SVM,kNN]":
                prediction = model[0].predict_proba(statistics)
                prediction_metrics["DT"].add_predictions(labels, prediction)
                prediction_metrics["ET"].add_predictions(labels, model[1].predict_proba(statistics))
                prediction_metrics["RF"].add_predictions(labels, model[2].predict_proba(statistics))
                prediction_metrics["SVM"].add_predictions(labels, model[3].predict_proba(statistics))
                prediction_metrics["kNN"].add_predictions(labels, model[4].predict_proba(statistics))

            elif architecture == "FFNN" and isinstance(model, FFNN):
                preds, labels = predict_torch_ffnn(model, test_ds, args)
                # You may want to adapt this to your PredictionPerformanceMetrics usage:
                prediction_metrics = {architecture: PredictionPerformanceMetrics(model_name=architecture)}
                prediction_metrics[architecture].add_predictions(labels, preds)
                for metrics in prediction_metrics.values():
                    metrics.print_evaluation()
                elapsed_prediction_time = datetime.fromtimestamp(time.time()) - datetime.fromtimestamp(start_time)
                prediction_stats = 'Prediction time: %d days %d hours %d minutes %d seconds.' % (
                    elapsed_prediction_time.days, elapsed_prediction_time.seconds // 3600,
                    (elapsed_prediction_time.seconds // 60) % 60,
                    elapsed_prediction_time.seconds % 60)
                return prediction_stats

            elif architecture == "LSTM" and isinstance(model, LSTM):
                preds, labels = predict_torch_lstm(model, test_ds, args)
                prediction_metrics = {architecture: PredictionPerformanceMetrics(model_name=architecture)}
                prediction_metrics[architecture].add_predictions(labels, preds)
                for metrics in prediction_metrics.values():
                    metrics.print_evaluation()
                elapsed_prediction_time = datetime.fromtimestamp(time.time()) - datetime.fromtimestamp(start_time)
                prediction_stats = 'Prediction time: %d days %d hours %d minutes %d seconds.' % (
                    elapsed_prediction_time.days, elapsed_prediction_time.seconds // 3600,
                    (elapsed_prediction_time.seconds // 60) % 60,
                    elapsed_prediction_time.seconds % 60)
                return prediction_stats
            else:
                prediction = model.predict(statistics, batch_size=args.batch_size, verbose=1)
                prediction_metrics[architecture].add_predictions(labels, prediction)

            total_len_prediction += len(prediction)
            cntr += 1
            test_iter = args.train_dataset_size * cntr
            test_epoch = test_ds.epoch
            if test_epoch > 0:
                test_epoch = test_iter // ((test_ds.iteration + test_ds.batch_size * test_ds.dataset_workers) // test_ds.epoch)
            print("Prediction Epoch: %d, Iteration: %d / %d" % (test_epoch, test_iter, args.max_iter))
            if test_iter >= args.max_iter:
                break
        if test_ds.iteration >= args.max_iter:
            break

    test_ds.stop_outstanding_tasks()
    elapsed_prediction_time = datetime.fromtimestamp(time.time()) - datetime.fromtimestamp(start_time)

    if total_len_prediction > args.train_dataset_size:
        total_len_prediction -= total_len_prediction % args.train_dataset_size
    print('\ntest data predicted: %d ciphertexts' % total_len_prediction)

    # print prediction metrics
    for metrics in prediction_metrics.values():
        metrics.print_evaluation()

    # print selected feature again
    if architecture == "SVM-Rotor":
        # print(f"RFE-support: \n{list(model.support_)}\n")
        # print(f"Support names: {convert_rfe_support_to_names(model.support_)}")
        # print(f"RFE-rank: \n{list(model.ranking_)}\n")
        print()

    # print("GridSearchCV:")
    # print(f"Best score: {model.best_score_}")
    # print(f"Best params: {model.best_params_}")

    prediction_stats = 'Prediction time: %d days %d hours %d minutes %d seconds with %d iterations and %d epochs.' % (
        elapsed_prediction_time.days, elapsed_prediction_time.seconds // 3600,
        (elapsed_prediction_time.seconds // 60) % 60,
        elapsed_prediction_time.seconds % 60, test_iter, test_epoch)

    return prediction_stats

def expand_cipher_groups(cipher_types):
    """Turn cipher group identifiers (ACA, MTC3, ROTOR, ALL) into a list of their ciphers."""
    expanded = cipher_types
    if config.MTC3 in expanded:
        del expanded[expanded.index(config.MTC3)]
        for i in range(5):
            expanded.append(config.CIPHER_TYPES[i])
```

```
    elif config.ACA in expanded:
        del expanded[expanded.index(config.ACA)]
        for i in range(56):
            expanded.append(config.CIPHER_TYPES[i])
    elif config.ROTOR in expanded:
        del expanded[expanded.index(config.ROTOR)]
        for i in range(56, 61):
            expanded.append(config.CIPHER_TYPES[i])
    elif config.ALL in expanded:
        del expanded[expanded.index(config.ALL)]
        for i in range(61):
            expanded.append(config.CIPHER_TYPES[i])
    return expanded

def main():
    # Don't fork processes to keep memory footprint low.
    multiprocessing.set_start_method("spawn")

    args = parse_arguments()

    cpu_count = os.cpu_count()
    if cpu_count and cpu_count < args.dataset_workers:
        print("WARNING: More dataset_workers set than CPUs available.")

    # Print arguments
    for arg in vars(args):
        print("{:23s}= {:s}".format(arg, str(getattr(args, arg))))

    args.plaintext_input_directory = os.path.abspath(args.plaintext_input_directory)
    args.rotor_input_directory = os.path.abspath(args.rotor_input_directory)
    args.ciphers = args.ciphers.lower()
    cipher_types = args.ciphers.split(',')
    architecture = args.architecture
    extend_model = args.extend_model

    # Validate inputs
    if os.path.splitext(args.model_name)[1] not in ('.h5', '.pth'):
        print('ERROR: The model must have extension ".h5" (for Keras) or ".pth" (for PyTorch FFNN).', file=sys.stderr)
        sys.exit(1)

    if extend_model is not None:
        if architecture not in ('FFNN', 'CNN', 'LSTM'):
            print('ERROR: Models with the architecture %s can not be extended!' % architecture,
                  file=sys.stderr)
            sys.exit(1)
        if len(os.path.splitext(extend_model)) != 2 or os.path.splitext(extend_model)[1] != '.h5':
            print('ERROR: The extended model name must have the ".h5" extension!', file=sys.stderr)
            sys.exit(1)

    if architecture == "SVM-Rotor" and cipher_types[0] != "rotor":
        print(f"When training rotor-only model, the argument `ciphers` "
              f"should equal 'rotor'. Selected ciphers are: '{cipher_types}'.")
        sys.exit(1)

    if args.train_dataset_size * args.dataset_workers > args.max_iter:
        print("ERROR: --train_dataset_size * --dataset_workers must not be bigger than --max_iter. "
              "In this case it was %d > %d" %
                  (args.train_dataset_size * args.dataset_workers, args.max_iter),
              file=sys.stderr)
        sys.exit(1)

    # Convert commandline cipher argument (all, aca, mtc3, rotor, etc.) to list of
    # all ciphers contained in the provided group. E.g. 'rotor' gets expanded
    # into 'enigma', 'm209', etc.
    cipher_types = expand_cipher_groups(cipher_types)

    # Ensure plaintext dataset is available at `args.plaintext_input_directory`.
    if should_download_plaintext_datasets(args):
        download_plaintext_datasets(args)

    # Load the datasets for the requested cipher types. If aca and rotor cipher types
    # are contained in `cipher_types`, both plaintext and ciphertext datasets are loaded.
    train_ds, test_ds = load_datasets_from_disk(args, cipher_types)

    # Get the number of cipher classes to predict. Since the label numbers are fixed,
    # it must be ensured that the output_layer_size of the neural networks contain
    # enough nodes upto the higest wanted class label.
    output_layer_size = max([config.CIPHER_TYPES.index(type) for type in cipher_types]) + 1

    # Create a model and allow for distributed training on multi-GPU machines
    model, strategy = create_model_with_distribution_strategy(
    architecture, extend_model, output_layer_size=output_layer_size, max_train_len=args.max_train_len)


    early_stopping_callback, train_iter, training_stats = train_model(model, strategy,
                                                                      args, train_ds)
    save_model(model, args)
    prediction_stats = predict_test_data(test_ds, model, args, early_stopping_callback, train_iter)

```

```
    print(training_stats)
    print(prediction_stats)

if __name__ == "__main__":
    main()
```

8.2 Full listing of `eval.py`

```
import multiprocessing
from pathlib import Path
import argparse
import random
import sys
import os
import pickle
import functools
import numpy as np
from datetime import datetime

import torch
import torch.nn.functional as F
import torch.optim as optim

# This environ variable must be set before all tensorflow imports!
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import SparseTopKCategoricalAccuracy
sys.path.append("../")
from util.utils import map_text_into_numberspace
from util.utils import print_progress
import cipherTypeDetection.config as config
from cipherTypeDetection.cipherStatisticsDataset import CipherStatisticsDataset, PlaintextPathsDatasetParameters,
    ↪ RotorCiphertextsDatasetParameters, calculate_statistics, pad_sequences
from cipherTypeDetection.predictionPerformanceMetrics import PredictionPerformanceMetrics
from cipherTypeDetection.rotorDifferentiationEnsemble import RotorDifferentiationEnsemble
from cipherTypeDetection.ensembleModel import EnsembleModel
from cipherTypeDetection.transformer import MultiHeadSelfAttention, TransformerBlock, TokenAndPositionEmbedding
from util.utils import get_model_input_length
from cipherImplementations.cipher import OUTPUT_ALPHABET, UNKNOWN_SYMBOL_NUMBER
tf.debugging.set_log_device_placement(enabled=False)
# always flush after print as some architectures like RF need very long time before printing anything.
print = functools.partial(print, flush=True)

for device in tf.config.list_physical_devices('GPU'):
    tf.config.experimental.set_memory_growth(device, True)

def str2bool(v):
    return v.lower() in ("yes", "true", "t", "1")


def benchmark(args, model, architecture):
    cipher_types = args.ciphers
    args.plaintext_folder = os.path.abspath(args.plaintext_folder)
    if args.dataset_size * args.dataset_workers > args.max_iter:
        print("ERROR: --dataset_size * --dataset_workers must not be bigger than --max_iter. In this case it was %d > %d" % (
            args.dataset_size * args.dataset_workers, args.max_iter), file=sys.stderr)
        sys.exit(1)
    if args.download_dataset and not os.path.exists(args.plaintext_folder) and args.plaintext_folder == os.path.abspath(
            '../data/gutenberg_en'):
        print("Downloading Datsets...")
        tfds.download.add_checksums_dir('../data/checksums/')
        download_manager = tfds.download.download_manager.DownloadManager(download_dir='../data/', extract_dir=args.plaintext_folder)
        download_manager.download_and_extract(
            'https://drive.google.com/uc?id=1bF5sSVjxTxa3DB-P5wxn87nxWndRhK_V&export=download')
        path = os.path.join(args.plaintext_folder, 'ZIP.ucid_1bF5sSVjxTx-P5wxn87nxWn_V_export_downloadR9Cwhunev5CvJ-ic__'
                                                   'HawxhTtGOlSdcCrro4fxfEI8A', os.path.basename(args.plaintext_folder))
        dir_nam = os.listdir(path)
        for name in dir_nam:
            p = Path(os.path.join(path, name))
            parent_dir = p.parents[2]
            p.rename(parent_dir / p.name)
        os.rmdir(path)
        os.rmdir(os.path.dirname(path))
        print("Datasets Downloaded.")

    print("Loading Datasets...")
    def validate_ciphertext_path(ciphertext_path, cipher_types):
        file_name = Path(ciphertext_path).stem.lower()
        if not file_name in cipher_types:
            raise Exception(f"Filename must equal one of the expected cipher types. Expected cipher types are: {cipher_types}. Current
                ↪ filename is '{file_name}'.")

    plaintext_files = []
    dir_nam = os.listdir(args.plaintext_folder)
```

```
    for name in dir_nam:
        path = os.path.join(args.plaintext_folder, name)
        if os.path.isfile(path):
            plaintext_files.append(path)

    def find_ciphertext_paths_in_dir(folder_path):
        """Loads all .txt files in the given folder and checks that their names match the
        known cipher types."""
        file_names = os.listdir(folder_path)
        result = []
        for name in file_names:
            path = os.path.join(folder_path, name)
            file_name, file_type = os.path.splitext(name)
            if os.path.isfile(path) and file_name.lower() in cipher_types and file_type == ".txt":
                validate_ciphertext_path(path, config.ROTOR_CIPHER_TYPES)

                result.append(path)
        return result

    eval_rotor_ciphertext_paths = find_ciphertext_paths_in_dir(args.rotor_ciphertext_folder)

    # Calculate batch size for rotor ciphers. The amount of samples per rotor cipher should be
    # equal to the amount of samples per aca cipher.
    number_of_rotor_ciphers = len(config.ROTOR_CIPHER_TYPES)
    number_of_aca_ciphers = len(config.CIPHER_TYPES) - number_of_rotor_ciphers
    amount_of_samples_per_cipher = args.dataset_size // number_of_aca_ciphers
    rotor_train_dataset_size = amount_of_samples_per_cipher * number_of_rotor_ciphers

    plaintext_dataset_params = PlaintextPathsDatasetParameters(cipher_types[:56],
                                                               args.dataset_size,
                                                               args.min_text_len,
                                                               args.max_text_len,
                                                               args.keep_unknown_symbols,
                                                               args.dataset_workers,
                                                               generate_evaluation_data=True)
    rotor_dataset_params = RotorCiphertextsDatasetParameters(config.ROTOR_CIPHER_TYPES,
                                                             rotor_train_dataset_size,
                                                             args.dataset_workers,
                                                             args.min_text_len,
                                                             args.max_text_len,
                                                             generate_evalutation_data=True)
    dataset = CipherStatisticsDataset(plaintext_files, plaintext_dataset_params, eval_rotor_ciphertext_paths,
                                      rotor_dataset_params, generate_evaluation_data=True)

    if args.dataset_size % dataset.key_lengths_count != 0:
        print("WARNING: the --dataset_size parameter must be dividable by the amount of --ciphers  and the length configured KEY_LENGTHS in "
              " config.py. The current key_lengths_count is %d" % dataset.key_lengths_count, file=sys.stderr)
    print("Datasets loaded.\n")

    print('Evaluating model...')

    import time
    start_time = time.time()
    iteration = 0
    epoch = 0
    results = []
    prediction_metrics = PredictionPerformanceMetrics(model_name=architecture)

    while dataset.iteration < args.max_iter:
        batches = next(dataset)

        for index, batch in enumerate(batches):
            statistics, labels, ciphertexts = batch.items()

            if architecture == "FFNN":
                if hasattr(model, "evaluate"):  # Keras model
                    results.append(model.evaluate(statistics, labels, batch_size=args.batch_size, verbose=1))
                else:  # PyTorch model
                    stats_np = statistics.numpy()

                    x = torch.tensor(stats_np, dtype=torch.float32)
                    y = torch.tensor(labels.numpy(), dtype=torch.long)

                    with torch.no_grad():
                        outputs = model(x)
                        loss = F.cross_entropy(outputs, y)
                        top1 = torch.argmax(outputs, dim=1)
                        acc = (top1 == y).float().mean()

                        # Calc top-3
                        top3 = torch.topk(outputs, k=3, dim=1).indices
                        y_expanded = y.unsqueeze(1).expand_as(top3)
                        k3_acc = (top3 == y_expanded).any(dim=1).float().mean()

                        results.append((loss.item(), acc.item(), k3_acc.item()))


            elif architecture in ("CNN", "LSTM", "Transformer"):
                results.append(model.evaluate(ciphertexts, labels, batch_size=args.batch_size, verbose=1))
```

```
            elif architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN"):
                results.append(model.score(statistics, labels))
                print("accuracy: %f" % (results[-1]))
            elif architecture == "Ensemble":
                results.append(model.evaluate(statistics, ciphertexts, labels, args.batch_size, prediction_metrics, verbose=1))

            iteration = dataset.iteration - len(batch) * (len(batches) - index - 1)
            epoch = dataset.epoch
            if epoch > 0:
                epoch = iteration // (dataset.iteration // dataset.epoch)
            print("Epoch: %d, Iteration: %d" % (epoch, iteration))
            if iteration >= args.max_iter:
                break

        if dataset.iteration >= args.max_iter:
            break

    elapsed_evaluation_time = datetime.fromtimestamp(time.time()) - datetime.fromtimestamp(start_time)
    print('Finished evaluation in %d days %d hours %d minutes %d seconds with %d iterations and %d epochs.\n' % (
        elapsed_evaluation_time.days, elapsed_evaluation_time.seconds // 3600, (elapsed_evaluation_time.seconds // 60) % 60,
        elapsed_evaluation_time.seconds % 60, iteration, epoch))

    if architecture in ("FFNN", "CNN", "LSTM", "Transformer"):
        avg_loss = 0
        avg_acc = 0
        avg_k3_acc = 0
        for loss, acc_pred, k3_acc in results:
            avg_loss += loss
            avg_acc += acc_pred
            avg_k3_acc += k3_acc
        avg_loss = avg_loss / len(results)
        avg_acc = avg_acc / len(results)
        avg_k3_acc = avg_k3_acc / len(results)
        print("Average evaluation results: loss: %f, accuracy: %f, k3_accuracy: %f\n" % (avg_loss, avg_acc, avg_k3_acc))
    elif architecture in ("DT", "NB", "RF", "ET", "Ensemble", "SVM", "kNN"):
        avg_test_acc = 0
        avg_k3_acc = 0
        for acc, k3_acc in results:
            avg_test_acc += acc
            avg_k3_acc += k3_acc
        avg_test_acc = avg_test_acc / len(results)
        avg_k3_acc = avg_k3_acc / len(results)
        print("Average evaluation results from %d iterations: avg_test_acc=%f, k3_accuracy: %f\n" % (iteration, avg_test_acc, avg_k3_acc))

        print("Detailed results:")
        prediction_metrics.print_evaluation()


def evaluate(args, model, architecture):
    results_list = []
    dir_name = os.listdir(args.data_folder)
    dir_name.sort()
    cntr = 0
    iterations = 0
    for name in dir_name:
        if iterations > args.max_iter:
            break
        path = os.path.join(args.data_folder, name)
        if os.path.isfile(path):
            if iterations > args.max_iter:
                break
            batch = []
            batch_ciphertexts = []
            labels = []
            results = []
            dataset_cnt = 0
            input_length = get_model_input_length(model, args.architecture)
            with open(path, "rb") as fd:
                lines = fd.readlines()
            for line in lines:
                # remove newline
                line = line.strip(b'\n').decode()
                if line == '':
                    continue
                split_line = line.split(' ')
                labels.append(int(split_line[0]))
                statistics = [float(f) for f in split_line[1].split(',')]
                batch.append(statistics)
                ciphertext = [int(j) for j in split_line[2].split(',')]
                if input_length is not None:
                    if len(ciphertext) < input_length:
                        ciphertext = pad_sequences([ciphertext], maxlen=input_length)[0]
                    # if the length its too high, we need to strip it..
                    elif len(ciphertext) > input_length:
                        ciphertext = ciphertext[:input_length]
                batch_ciphertexts.append(ciphertext)
                iterations += 1
                if iterations == args.max_iter:
                    break
```

```
255                 if len(labels) == args.dataset_size:
256                     if architecture == "FFNN":
257                         results.append(model.evaluate(tf.convert_to_tensor(batch), tf.convert_to_tensor(labels), args.batch_size, verbose
                                ↪ =0))
258                     elif architecture in ("CNN", "LSTM", "Transformer"):
259                         results.append(model.evaluate(tf.convert_to_tensor(batch_ciphertexts), tf.convert_to_tensor(labels),
260                                                       args.batch_size, verbose=0))
261                     elif architecture == "Ensemble":
262                         results.append(model.evaluate(tf.convert_to_tensor(batch), tf.convert_to_tensor(batch_ciphertexts), tf.
                                ↪ convert_to_tensor(labels),
263                                                       args.batch_size, verbose=0))
264                     elif architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN"):
265                         results.append(model.score(batch, tf.convert_to_tensor(labels)))
266                     batch = []
267                     batch_ciphertexts = []
268                     labels = []
269                     dataset_cnt += 1
270             if len(labels) > 0:
271                 if architecture == "FFNN":
272                     results.append(model.evaluate(tf.convert_to_tensor(batch), tf.convert_to_tensor(labels), args.batch_size, verbose=0))
273                 elif architecture in ("CNN", "LSTM", "Transformer"):
274                     results.append(
275                         model.evaluate(tf.convert_to_tensor(batch_ciphertexts), tf.convert_to_tensor(labels), args.batch_size, verbose=0))
276                 elif architecture == "Ensemble":
277                     results.append(
278                         model.evaluate(tf.convert_to_tensor(batch), tf.convert_to_tensor(batch_ciphertexts), tf.convert_to_tensor(labels),
279                                        args.batch_size, verbose=0))
280                 elif architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN"):
281                     results.append(model.score(batch, tf.convert_to_tensor(labels)))
282             if architecture in ("FFNN", "CNN", "LSTM", "Transformer"):
283                 avg_loss = 0
284                 avg_acc = 0
285                 avg_k3_acc = 0
286                 for loss, acc_pred, k3_acc in results:
287                     avg_loss += loss
288                     avg_acc += acc_pred
289                     avg_k3_acc += k3_acc
290                 result = [avg_loss / len(results), avg_acc / len(results), avg_k3_acc / len(results)]
291             elif architecture in ("DT", "NB", "RF", "ET", "Ensemble", "SVM", "kNN"):
292                 avg_test_acc = 0
293                 for acc in results:
294                     avg_test_acc += acc
295                 result = avg_test_acc / len(results)
296             results_list.append(result)
297             cntr += 1
298             if args.evaluation_mode == 'per_file':
299                 if architecture in ("FFNN", "CNN", "LSTM", "Transformer"):
300                     print("%s (%d lines) test_loss: %f, test_accuracy: %f, test_k3_accuracy: %f (progress: %d%%)" % (
301                         os.path.basename(path), len(batch) + dataset_cnt * args.dataset_size, result[0], result[1], result[2], max(
302                             int(cntr / len(dir_name) * 100), int(iterations / args.max_iter) * 100)))
303                 elif architecture in ("DT", "NB", "RF", "ET", "Ensemble", "SVM", "kNN"):
304                     print("%s (%d lines) test_accuracy: %f (progress: %d%%)" % (
305                         os.path.basename(path), len(batch) + dataset_cnt * args.dataset_size, result,
306                         max(int(cntr / len(dir_name) * 100), int(iterations / args.max_iter) * 100)))
307             else:
308                 print_progress("Evaluating files: ", cntr, len(dir_name), factor=5)
309             if iterations == args.max_iter:
310                 break
311 
312     if architecture in ("FFNN", "CNN", "LSTM", "Transformer"):
313         avg_test_loss = 0
314         avg_test_acc = 0
315         avg_test_acc_k3 = 0
316         for loss, acc, acc_k3 in results_list:
317             avg_test_loss += loss
318             avg_test_acc += acc
319             avg_test_acc_k3 += acc_k3
320         avg_test_loss = avg_test_loss / len(results_list)
321         avg_test_acc = avg_test_acc / len(results_list)
322         avg_test_acc_k3 = avg_test_acc_k3 / len(results_list)
323         print("\n\nAverage evaluation results from %d iterations: avg_test_loss=%f, avg_test_acc=%f, avg_test_acc_k3=%f" % (
324             iterations, avg_test_loss, avg_test_acc, avg_test_acc_k3))
325     elif architecture in ("DT", "NB", "RF", "ET", "Ensemble", "SVM", "kNN"):
326         avg_test_acc = 0
327         for acc in results_list:
328             avg_test_acc += acc
329         avg_test_acc = avg_test_acc / len(results_list)
330         print("\n\nAverage evaluation results from %d iterations: avg_test_acc=%f" % (iterations, avg_test_acc))
331 
332 
333 def predict_single_line(args, model, architecture):
334     cipher_id_result = ''
335     ciphertexts = []
336     result = []
337     if args.ciphertext is not None:
338         ciphertexts.append(args.ciphertext.encode())
339     else:
340         ciphertexts = open(args.file, 'rb')
341 
342     print()
```

```
343     for line in ciphertexts:
344         # remove newline
345         line = line.strip(b'\n')
346         if line == b'':
347             continue
348         # evaluate aca features file
349         # label = line.split(b' ')[0]
350         # statistics = ast.literal_eval(line.split(b' ')[1].decode())
351         # ciphertext = ast.literal_eval(line.split(b' ')[2].decode())
352         # print(config.CIPHER_TYPES[int(label.decode())], "length: %d" % len(ciphertext))
353 
354         # Append ciphertext to itself. This improves the reliablity of the results.
355         while len(line) < 1000:
356             line = line + line
357         # Limit line to at most 1000 characters to limit the execution time
358         line = line[:1000]
359 
360         print(line)
361         ciphertext = map_text_into_numberspace(line, OUTPUT_ALPHABET, UNKNOWN_SYMBOL_NUMBER)
362         try:
363             statistics = calculate_statistics(ciphertext)
364         except ZeroDivisionError:
365             print("\n")
366             continue
367         results = None
368         if architecture == "FFNN":
369             result = model.predict(tf.convert_to_tensor([statistics]), args.batch_size, verbose=0)
370         elif architecture in ("CNN", "LSTM", "Transformer"):
371             input_length = get_model_input_length(model, architecture)
372             if len(ciphertext) < input_length:
373                 ciphertext = pad_sequences([list(ciphertext)], maxlen=input_length)[0]
374             split_ciphertext = [ciphertext[input_length*j:input_length*(j+1)] for j in range(len(ciphertext) // input_length)]
375             results = []
376             if architecture in ("LSTM", "Transformer"):
377                 for ct in split_ciphertext:
378                     results.append(model.predict(tf.convert_to_tensor([ct]), args.batch_size, verbose=0))
379             elif architecture == "CNN":
380                 for ct in split_ciphertext:
381                     results.append(
382                         model.predict(tf.reshape(tf.convert_to_tensor([ct]), (1, input_length, 1)), args.batch_size, verbose=0))
383             result = results[0]
384             for res in results[1:]:
385                 result = np.add(result, res)
386             result = np.divide(result, len(results))
387         elif architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN"):
388             result = model.predict_proba(tf.convert_to_tensor([statistics]))
389         elif architecture == "Ensemble":
390             result = model.predict(tf.convert_to_tensor([statistics]), [ciphertext], args.batch_size, verbose=0)
391 
392         if isinstance(result, list):
393             result_list = list(result[0])
394         else:
395             result_list = result[0].tolist()
396         if results is not None and architecture not in ('Ensemble', 'LSTM', 'Transformer', 'CNN'):
397             for j in range(len(result_list)):
398                 result_list[j] /= len(results)
399         if args.verbose:
400             for cipher in args.ciphers:
401                 print("{:23s} {:f}%".format(cipher, result_list[config.CIPHER_TYPES.index(cipher)]*100))
402             max_val = max(result_list)
403             cipher = config.CIPHER_TYPES[result_list.index(max_val)]
404         else:
405             max_val = max(result_list)
406             cipher = config.CIPHER_TYPES[result_list.index(max_val)]
407             print("{:s} {:f}%".format(cipher, max_val * 100))
408         print()
409         cipher_id_result += cipher[0].upper()
410 
411     if args.file is not None:
412         ciphertexts.close()
413 
414     # return a list of probabilities (does only return the last one in case a file is used)
415     res_dict = {}
416     if len(result) != 0:
417         for j, val in enumerate(result[0]):
418             res_dict[args.ciphers[j]] = val * 100
419     return res_dict
420 
421 
422 def load_model(architecture, args, model_path, cipher_types):
423     strategy = args.strategy
424     model_list = args.models
425     architecture_list = args.architectures
426 
427     model = None
428 
429     if architecture == "FFNN" and model_path.endswith(".pth"):
430         from cipherTypeDetection.train import FFNN
431 
```

```
        checkpoint = torch.load(model_path, map_location=torch.device("cpu"))

        model = FFNN(
            input_size=checkpoint['input_size'],
            hidden_size=checkpoint['hidden_size'],
            output_size=checkpoint['output_size'],
            num_hidden_layers=checkpoint['num_hidden_layers']
        )
        model.load_state_dict(checkpoint['model_state_dict'])
        model.eval()

        config.FEATURE_ENGINEERING = True
        config.PAD_INPUT = False

        return model

    elif architecture in ("FFNN", "CNN", "LSTM", "Transformer"):
        if architecture == 'Transformer':
            if not hasattr(config, "maxlen"):
                raise ValueError("maxlen must be defined in the config when loading a Transformer model!")
            model = tf.keras.models.load_model(args.model, custom_objects={
                'TokenAndPositionEmbedding': TokenAndPositionEmbedding,
                'TransformerBlock': TransformerBlock})
        else:
            model = tf.keras.models.load_model(args.model)
        if architecture in ("CNN", "LSTM", "Transformer"):
            config.FEATURE_ENGINEERING = False
            config.PAD_INPUT = True
        else:
            config.FEATURE_ENGINEERING = True
            config.PAD_INPUT = False
        optimizer = Adam(learning_rate=config.learning_rate, beta_1=config.beta_1, beta_2=config.beta_2, epsilon=config.epsilon,
                         amsgrad=config.amsgrad)
        model.compile(optimizer=optimizer, loss="sparse_categorical_crossentropy",
                      metrics=["accuracy", SparseTopKCategoricalAccuracy(k=3, name="k3_accuracy")])
    elif architecture in ("DT", "NB", "RF", "ET", "SVM", "kNN"):
        config.FEATURE_ENGINEERING = True
        config.PAD_INPUT = False
        with open(model_path, "rb") as f:
            model = pickle.load(f)
    elif architecture == 'Ensemble':
        cipher_indices = []
        for cipher_type in cipher_types:
            cipher_indices.append(config.CIPHER_TYPES.index(cipher_type))
        model = EnsembleModel(model_list, architecture_list, strategy, cipher_indices)
    else:
        raise ValueError("Unknown architecture: %s" % architecture)

    # Controlla se ci sono cifrari rotor tra quelli richiesti
    has_rotor_ciphers = any(c in config.ROTOR_CIPHER_TYPES for c in cipher_types)

    # Se ci sono cifrari rotor, carica anche il modello rotor_only
    if has_rotor_ciphers:
        rotor_only_model_path = args.rotor_only_model
        if not os.path.exists(rotor_only_model_path):
            raise FileNotFoundError(f"Rotor-only model is required but not found at {rotor_only_model_path}")
        with open(rotor_only_model_path, "rb") as f:
            rotor_only_model = pickle.load(f)
        return RotorDifferentiationEnsemble(architecture, model, rotor_only_model)

    # Se non ci sono cifrari rotor:
    # - se è un ensemble, restituisci direttamente l'ensemble
    # - altrimenti restituisci il modello normale
    return model


def expand_cipher_groups(cipher_types):
    """Turn cipher group identifiers (ACA, MTC3) into a list of their ciphers"""
    expanded = cipher_types
    if config.MTC3 in expanded:
        del expanded[expanded.index(config.MTC3)]
        for i in range(5):
            expanded.append(config.CIPHER_TYPES[i])
    elif config.ACA in expanded:
        del expanded[expanded.index(config.ACA)]
        for i in range(56):
            expanded.append(config.CIPHER_TYPES[i])
    elif "aca+rotor" in expanded:
        del expanded[expanded.index("aca+rotor")]
        for i in range(61):
            expanded.append(config.CIPHER_TYPES[i])
    return expanded

def parse_arguments():
    parser = argparse.ArgumentParser(
        description='CANN Ciphertype Detection Neuronal Network Evaluation Script', formatter_class=argparse.RawTextHelpFormatter)
    sp = parser.add_subparsers()
    bench_parser = sp.add_parser('benchmark',
                                 help='Use this argument to create ciphertexts on the fly, \nlike in training mode, and evaluate them with
```

```
                                    ↪ '
                                'the model. \nThis option is optimized for large throughput to test the model.')
    eval_parser = sp.add_parser('evaluate', help='Use this argument to evaluate cipher types for single files or directories.')
    single_line_parser = sp.add_parser('single_line', help='Use this argument to predict a single line of ciphertext.')

    parser.add_argument('--batch_size', default=128, type=int,
                        help='Batch size for training.')
    parser.add_argument('--max_iter', default=1000000000, type=int,
                        help='the maximal number of iterations before stopping evaluation.')
    parser.add_argument('--model', default='../data/models/ml.h5', type=str,
                        help='Path to the model file. The file must have the .h5 extension.')
    parser.add_argument('--rotor_only_model', default='../data/models/svm_rotor_only.h5', type=str,
                        help='Path to rotor only model. This model is used in conjunction '
                        'with the normal model in an ensemble to improve the recogintion '
                        'of rotor ciphers. The file must have the .h5 extension.')
    parser.add_argument('--architecture', default='FFNN', type=str, choices=[
        'FFNN', 'CNN', 'LSTM', 'DT', 'NB', 'RF', 'ET', 'Transformer', 'SVM', 'kNN', 'Ensemble'],
        help='The architecture to be used for training. \n'
             'Possible values are:\n'
             '- FFNN\n'
             '- CNN\n'
             '- LSTM\n'
             '- DT\n'
             '- NB\n'
             '- RF\n'
             '- ET\n'
             '- Transformer\n'
             '- SVM\n'
             '- kNN\n'
             '- Ensemble')
    parser.add_argument('--ciphers', '--ciphers', default='aca', type=str,
                        help='A comma seperated list of the ciphers to be created.\n'
                             'Be careful to not use spaces or use \' to define the string.\n'
                             'Possible values are:\n'
                             '- mtc3 (contains the ciphers Monoalphabetic Substitution, Vigenere,\n'
                             '        Columnar Transposition, Plaifair and Hill)\n'
                             '- aca (contains all currently implemented ciphers from \n'
                             '       https://www.cryptogram.org/resource-area/cipher-types/)\n'
                             '- aca+rotor\n'
                             '- simple_substitution\n'
                             '- vigenere\n'
                             '- columnar_transposition\n'
                             '- playfair\n'
                             '- hill\n')

    parser.add_argument('--models', action='append', default=None,
                        help='A list of models to be used in the ensemble model. The length of the list must be the same like the one in '
                             'the --architectures argument.')
    parser.add_argument('--architectures', action='append', default=None,
                        help='A list of the architectures to be used in the ensemble model. The length of the list must be the same like '
                             'the one in the --models argument.')
    parser.add_argument('--strategy', default='weighted', type=str, choices=['mean', 'weighted'],
                        help='The algorithm used for decisions.\n- Mean voting adds the probabilities from every class and returns the mean
                            ↪ '
                             ' value of it. The highest value wins.\n- Weighted voting uses pre-calculated statistics, like for example '
                             'precision, to weight the output of a specific model for a specific class.')
    parser.add_argument('--dataset_size', default=16000, type=int,
                        help='Dataset size per evaluation. This argument should be dividable \nby the amount of --ciphers.')

    bench_parser.add_argument('--download_dataset', default=True, type=str2bool)
    bench_parser.add_argument('--dataset_workers', default=1, type=int)
    bench_parser.add_argument('--plaintext_folder', default='../data/gutenberg_en', type=str)
    bench_parser.add_argument('--rotor_ciphertext_folder', default='../data/rotor_ciphertexts', type=str)
    bench_parser.add_argument('--keep_unknown_symbols', default=False, type=str2bool)
    bench_parser.add_argument('--min_text_len', default=50, type=int)
    bench_parser.add_argument('--max_text_len', default=-1, type=int)

    bench_group = parser.add_argument_group('benchmark')
    bench_group.add_argument('--download_dataset', help='Download the dataset automatically.')
    bench_group.add_argument('--dataset_workers', help='The number of parallel workers for reading the input files.')
    bench_group.add_argument('--plaintext_folder', help='Input folder of the plaintexts.')
    bench_group.add_argument('--keep_unknown_symbols', help='Keep unknown symbols in the plaintexts. Known \n'
                                                            'symbols are defined in the alphabet of the cipher.')
    bench_group.add_argument('--min_text_len', help='The minimum length of a plaintext to be encrypted in the evaluation process.\n'
                             'If this argument is set to -1 no lower limit is used.')
    bench_group.add_argument('--max_text_len', help='The maximum length of a plaintext to be encrypted in the evaluation process.\n'
                             'If this argument is set to -1 no upper limit is used.')

    eval_parser.add_argument('--evaluation_mode', nargs='?', choices=('summarized', 'per_file'), default='summarized', type=str)
    eval_parser.add_argument('--data_folder', default='../data/gutenberg_en', type=str)

    eval_group = parser.add_argument_group('evaluate')
    eval_group.add_argument('--evaluation_mode',
                            help='- To create an single evaluation result over all iterated data files use the \'summarized\' option.'
                                 '\n  This option is to be preferred over the benchmark option, if the tests should be reproducable.\n'
                                 '- To create an evaluation for every file use \'per_file\' option. This mode allows the \n'
                                 '  calculation of the \n  - average value of the prediction \n'
                                 '  - lower quartile - value at the position of 25 percent of the sorted predictions\n'
                                 '  - median - value at the position of 50 percent of the sorted predictions\n'
```

```
                                '  - upper quartile - value at the position of 75 percent of the sorted predictions\n'
                                '  With these statistics an expert can classify a ciphertext document to a specific cipher.')
    eval_group.add_argument('--data_folder', help='Input folder of the data files with labels and calculated features.')

    single_line_parser.add_argument('--verbose', default=True, type=str2bool)
    data = single_line_parser.add_mutually_exclusive_group(required=True)
    data.add_argument('--ciphertext', default=None, type=str)
    data.add_argument('--file', default=None, type=str)

    single_line_group = parser.add_argument_group('single_line')
    single_line_group.add_argument('--ciphertext', help='A single line of ciphertext to be predicted by the model.')
    single_line_group.add_argument('--file', help='A file with lines of ciphertext to be predicted line by line by the model.')
    single_line_group.add_argument('--verbose', help='If true all predicted ciphers are printed. \n'
                                                     'If false only the most accurate prediction is printed.')

    return parser.parse_args()

def main():
    multiprocessing.set_start_method("spawn")

    args = parse_arguments()

    for arg in vars(args):
        print("{:23s}= {:s}".format(arg, str(getattr(args, arg))))
    m = os.path.splitext(args.model)
    if os.path.splitext(args.model)[1] not in ('.h5', '.pth'):
        print('ERROR: The model must have extension ".h5" (for Keras) or ".pth" (for PyTorch FFNN).', file=sys.stderr)
        sys.exit(1)

    architecture = args.architecture
    model_path = args.model
    args.ciphers = args.ciphers.lower()
    cipher_types = args.ciphers.split(',')
    args.ciphers = expand_cipher_groups(cipher_types)
    if architecture == 'Ensemble':
        if not hasattr(args, 'models') or not hasattr(args, 'architectures'):
            raise ValueError("Please use the 'ensemble' subroutine if specifying the ensemble architecture.")
        if len(args.models) != len(args.architectures):
            raise ValueError("The length of --models must be the same like the length of --architectures.")
        models = []
        for i in range(len(args.models)):
            model = args.models[i]
            arch = args.architectures[i]
            if not os.path.exists(os.path.abspath(model)):
                raise ValueError("Model in %s does not exist." % os.path.abspath(model))
            if arch not in ('FFNN', 'CNN', 'LSTM', 'DT', 'NB', 'RF', 'ET', 'Transformer', 'SVM', 'kNN'):
                raise ValueError("Unallowed architecture %s" % arch)
            if arch in ('FFNN', 'CNN', 'LSTM', 'Transformer') and not os.path.abspath(model).endswith('.h5'):
                raise ValueError("Model names of the types %s must have the .h5 extension." % ['FFNN', 'CNN', 'LSTM', 'Transformer'])
    elif args.models is not None or args.architectures is not None:
        raise ValueError("It is only allowed to use the --models and --architectures with the Ensemble architecture.")

    print("Loading Model...")
    # There are some problems regarding the loading of models on multiple GPU's.
    # gpu_count = len(tf.config.list_physical_devices('GPU'))
    # if gpu_count > 1:
    #     strat = tf.distribute.MirroredStrategy()
    #     with strat.scope():
    #         model = load_model()
    # else:
    #     model = load_model()
    model = load_model(architecture, args, model_path, cipher_types)
    print("Model Loaded.")

    # Model is now always an ensemble
    #architecture = "Ensemble"

    # the program was started as in benchmark mode.
    if args.download_dataset is not None:
        benchmark(args, model, architecture)
    # the program was started in single_line mode.
    elif args.ciphertext is not None or args.file is not None:
        predict_single_line(args, model, architecture)
    # the program was started in prediction mode.
    else:
        evaluate(args, model, architecture)

if __name__ == "__main__":
    main()
```