Of Ciphers and Neurons
Detecting the Type of Ciphers Using Artificial Neural Networks

Nils Kopal
University of Siegen, Germany
nils.kopal@uni-siegen.de

Abstract

There are many (historical) unsolved ciphertexts from which we don't know the type of cipher which was used to encrypt these. A first step each cryptanalyst does is to try to identify their cipher types using different (statistical) methods. This can be difficult, since a multitude of cipher types exist. To help cryptanalysts, we developed a first version of an artificial neural network that is right now able to differentiate between five classical ciphers: simple monoalphabetic substitution, Vigenère, Playfair, Hill, and transposition. The network is based on Google's TensorFlow library as well as Keras. This paper presents the current progress in the research of using such networks for detecting the cipher type. We tried to classify all ciphers of a new MysteryTwister C3 challenge called "Cipher ID" created by Stamp in 2019. The network is able to classify about 90% of the ciphertexts of the challenge correctly. Furthermore, the paper presents the current state-of-the-art of cipher type detection. Finally, we present a method which shows that one can save about 54% computation time for classification of cipher types when using our artificial neural network instead of trying different solvers for all ciphertext messages of Stamp's challenge.

1 Introduction

Artificial neural networks (ANNs) experienced a renaissance over the past years. Supported by the development of easy-to-use software libraries, e.g. TensorFlow and Keras, as well as the wide range of new powerful hardware (especially graphic card processors and application-specific integrated circuits). ANNs found usages in a broad set of different applications and research fields. Their main purpose is fast filtering, classifying, and processing of (mostly) non-linear data, e.g. image processing, speech recognition, and language translation. Besides that, scientists were also able to "teach" ANNs to play games or to create paintings in the style of famous artists.

Inspired by the vast growth of ANNs, also cryptologists started to use them for different cryptographic and cryptanalytic problems. Examples are the learning of complex cryptographic algorithms, e.g. the Enigma machine, or the detection of the type of cipher used for encrypting a specific ciphertext.

In late 2019 Stamp published a challenge on the MysteryTwister C3 (MTC3) website called "Cipher ID". The goal of the challenge is to assign the type of cipher to each ciphertext out of a set of 500 ciphertexts, while 5 different types of ciphers were used to encrypt these ciphertexts using random keys. Each cipher type was used exactly 100 times and the different ciphertexts were shuffled then. While the intention of the author was to motivate people to start research in the field of machine learning and cipher type detection, all previous solvers solved the challenge by breaking the ciphertexts using solvers for the 5 different cipher types. Thus, after revealing the plaintext of each cipher, the participants knew which type of encryption algorithm was used.

We started to work on the cipher type detection problem in 2019 with the intention to detect the ciphers' types solely using ANNs. TensorFlow (Abadi et al., 2016) and Keras (Chollet, 2015) were used. TensorFlow is a free and open-source data flow and math library developed by Google written in Python, C++, and CUDA, and was publicly released in 2015. Keras is a free and open-source library for developing ANNs developed by Chollet and also written in

Python. In 2017 Google's TensorFlow team decided to support Keras in the TensorFlow core library. While working on the cipher type detection problem, Stamp's challenge was published. We then adapted our code and tools to the requirements of the challenge. Therefore, in this paper, we present our current progress of implementing a cipher type detection ANN with the help of the aforementioned libraries especially for the MTC3 challenge. At the time of writing this paper, we are able to classify the type of ciphers of the aforementioned challenge at a success rate of about 90%. Despite this relatively good detection rate it is still not good enough to solve the challenge on its own. Therefore, we also propose a first idea of a detection (and solving) method for ciphertexts with unknown cipher types.

The contributions and goals of this paper are:

1. First public ANN classifier for classical ciphers developed with TensorFlow and Keras.

2. Presentation of the basics of ANNs to the audience of HistoCrypt, who are from different research areas, e.g. history and linguistics (but mostly no computers scientists).

3. Example Python code which can be used to directly implement our methods in TensorFlow and Keras.

4. Overview of the existing work in the field of ANNs and cryptanalysis of classical/historical ciphers and cipher type detection.

5. Presentation of a first idea of a method which does both, cipher type detection and solving of classical ciphers.

The rest of this paper is structured as follows: Section 2 presents the related work in the field of machine learning and cryptanalysis with a focus on ANNs. Section 3 shows the foundation on which we created our methods. Here, firstly we discuss ANNs in general. Secondly, we briefly present TensorFlow as well as Keras. After that, Section 4 presents our cipher type detection approach based on the aforementioned libraries. Then, Section 5 discusses our first ideas for a cipher type detection and solving method. Finally, Section 6 briefly concludes the paper and gives an overview of planned future work with regards to ANNs and cryptology.

2 Related Work

In this section, we present different papers and articles, which deal with ANNs and cryptology. The usage of ANNs in the paper ranges between the emulation of ciphers, the detection of the cipher type, and the recovering of cryptographic keys. Also, there are papers where the authors worked with other techniques to detect the cipher type.

1. Ibrahem (Khalel Ibrahem Al-Ubaidy, 2004) presents two ideas: First, to determine the key from a given plaintext-ciphertext pair. He calls this the "cryptanalysis approach". Second, the emulation of an unknown cipher. He calls this the "emulation approach". He used an ANN with two hidden layers in his approach. For training his model he used Levenberg-Marquardt (LM). He successfully trains Vigenère cipher as well as two different stream ciphers (GEFFE and THRESHOLD, which are both linear feedback shift registers).

2. Chandra (Chandra et al., 2007) present their method of cipher type identification. They created different ANNs which are able to distinguish between different modern ciphers, e.g. RC6 and Serpent. Their ANN architecture is comparable small, consisting only of 2 hidden layers, where each layer has at most 25 neurons. They used different techniques to map from the ciphertext to 400 "input patterns", which they fed to their network.

3. Sivagurunathan (Sivagurunathan et al., 2010) created an ANN with one hidden layer to distinguish between Vigenère cipher, Hill cipher, and Playfair cipher. While their network was able to detect Playfair ciphers with an accuracy of 100%, the detection rate of Vigenère and Hill was between 69% and 85%, depending on their test scenarios.

4. The BION classifiers from BION's gadget website[1] are browser-based classifiers, integrated in two well working cipher type detection methods built in JavaScript. The first one works with random decision forests and the second one is based on a multitude of ANNs. The basic idea with the second classifier (ANN-based) is, that the different networks (different layers, activation functions,

[1] see `https://bionsgadgets.appspot.com/`

etc.) each have a "vote" for the cipher type. In the end, the votes are shown, and the correct cipher type probably has the most votes. The classifiers are able to detect the cipher types defined by the American cryptogram association (ACM).

5. Nuhn and Knight's (Nuhn and Knight, 2014) extensive work on cipher type detection used a support vector machine based on the libSVM toolkit (Chang and Lin, 2011). In their work, they used 58 different features to successfully classify 50 different cipher types out of 56 cipher types specified by the American cryptogram association (ACA).

6. Greydanus (Greydanus, 2017) used recurrent neural networks (RNN) to learn the Enigma. An RNN has connections going from successive hidden layer neurons to neurons in preceding layers. He showed that an RNN with a 3000-unit long short-term memory cell can learn the decryption function of an Enigma machine with three rotors, of a Vigenère cipher, and of a Vigenère Autokey cipher. Furthermore, he created an RNN network which was able to recover keys (length one to six) of Vigenère and Vigenère Autokey.

7. Focardi and Luccio (Focardi and Luccio, 2018) present their method of breaking Caesar and Vigenère ciphers with the help of neural networks. They used fairly simple neural networks having only one hidden layer. They were able to recover substitution keys with a success rate of about 93%, where at most 2 mappings in the keys were wrong.

8. Abd (Abd and Al-Janabi, 2019) developed three different classifiers based on neural networks. Their work is the closest related to our work. Their idea is to create three classifiers, each a single ANN, with different levels (1, 2, and 3), where each level increases the detection accuracy. The first level differentiates between natural language, substitution ciphers, transposition ciphers, and combined ciphers. Then, their second level differentiates between monoalphabetic, polyalphabetic, and polygraphic. Their last level differentiates between Playfair and different Hill ciphers. They state that their success rate is about 99.6%.

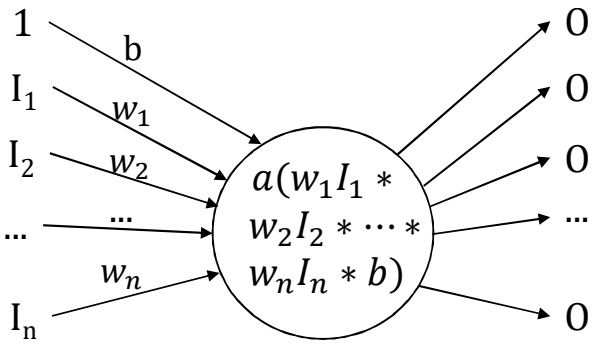

Figure 1: A single neuron of an ANN with inputs, outputs, bias, and activation function

3 Foundation

In this section, we describe the foundation used for our detection method. First, we discuss the ANN in general. Then, we give an introduction to TensorFlow and Keras and show some example Python code building an ANN.

3.1 Artificial Neural Network

Artificial neural networks (ANNs) are computing models (organized as graphs) that are in principle inspired by the human brain. The book "Make your own neural network" from (Rashid, 2016) gives a good introduction into ANNs. Different **neurons** are connected via input and output connections, providing **signals**, having different **weights** assigned to them. A neuron itself contains an **activation function** a, which fires the neuron's outputs based on the neuron's input values. For example, all the values of the input connections are combined with their respective weight values. Then, all resulting values are combined and a bias value b is also added to the result. After that, an activation function is computed using the result of the combined values. Figure 1 depicts an example of one neuron with different input connections, a bias input connection, an activation function a, and output connections. Usually, the value of the bias input connection is set to 1.

A common practice in ANNs is to organize neurons in so-called **layers**. The input data is given to an **input layer** consisting of n different neurons. The input layer is then connected to one or more **hidden layers**. Finally, the last hidden layer is connected to an **output layer**. Each neuron of the previous layer is connected to each neuron of the following layer. Figure 2 depicts an example of an ANN with only a single hidden layer. In general,

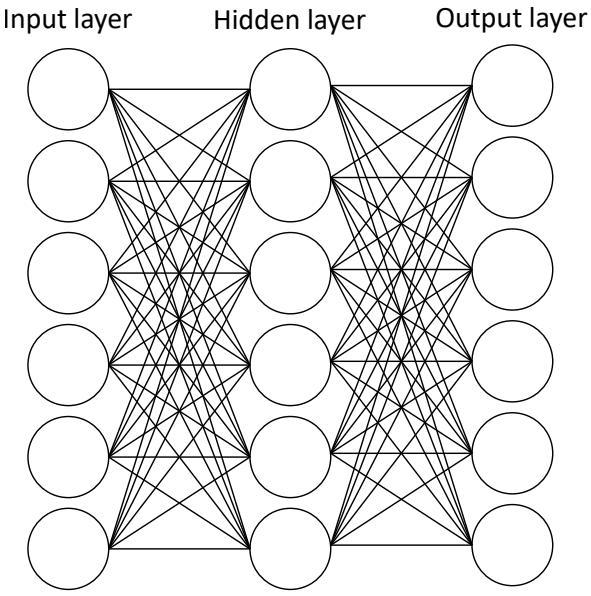

Figure 2: An ANN with input, hidden, and output layers

when working with ANNs having several hidden layers, researchers refer to the term of **deep learning** (Wartala, 2018).

The learning, in general, is performed by adapting the weights of the connections between the neurons. There exist different methods for learning, e.g. supervised and unsupervised learning. Here, we focus on supervised learning, which is suited well for classification tasks. The input data is given as a so called **feature vector** x from the input space X and the output is a **label** y from the output space Y. A label, in general, clusters a set of similar input values, i.e. each of the input values of the same **cluster** is mapped to the same label. The goal is to find a function $f : X \rightarrow Y$ that maps each element of the input space correctly to the labels of the output space.

As a basic idea, the ANN's connection weights are initialized with random values. Then, a set of data (inputs and desired labels) is feeded to the network. While doing so, the actual output labels as well as the desired labels are compared using a **loss function**. Using **back propagation** the error is propagated in the reverse order through the network and the weight values are changed for each neuron of each layer accordingly.

Different parameters and attributes of the ANN and the learning process influence the success rate of the learning: e.g. the quality and quantity of the input data and labels, the number of hidden layers of the ANN, the number of neurons of each layer, the types of used activation functions of the neurons, the used loss function, and the number of times the input data is feeded to the network.

Usually, the input data and their respective labels are divided into two different sets: **training data** and **test data**. For the actual learning, the training data is used. Then, to measure the quality of the ANN the test data is used. In the best case, after training the ANN is able to classify the test data correctly. In the worst case, the ANN only learned the training data (perfectly), but fails in classifying the test data. In this case, researcher refer to the term **overfitting**.

3.2 TensorFlow and Keras

TensorFlow (Abadi et al., 2016) is a software library developed by Google and firstly released in 2015. Its name is based on the term "tensor", which describes a mathematical function that maps a specific number of input vectors to output vectors, and on the term "flow", the idea of different tensors flowing as data streams through a dataflow graph. Keras (Chollet, 2015) is an open-source deep learning Python library and since 2017 also included in TensorFlow.

Working with TensorFlow and Keras (with ANNs), in general, consists of the following five steps:

1. Loading and preparing training and test data
2. Creating a model
3. Training the model
4. Testing and optimizing the model
5. Persisting the model

In the following, we describe the above steps involved in the creation, training, and usage of a Keras model. TensorFlow models work on multi-dimensional Python numpy arrays.

Step 1) First, the data has to be loaded and then split into a test and a training data set. In the following example, we split a data set of 5000 test data and their according labels (each label corresponds to one output class) into two disjunct sets of training and test data and labels:

```
# data is a set of data
# labels is a set of labels
# here, we split both into
# two different sets
```

```
train_data = data[0:4500]
train_labels = labels[0:4500]
test_data = data[4500:5000]
test_labels = labels[4500:5000]
```

Step 2) The second step is the creation of a Keras model. TensorFlow and Keras offer different methods of creating a model. The easiest method is to use the sequential model, which creates a multi-layered ANN. An example call of creating a simple ANN with an input layer, a single hidden layer, and an output layer is the following:

```
# create model:
m = keras.Sequential()
# create and add input layer:
m.add(Flatten(input_shape=(100,)))
# create and add hidden layer:
m.add(Dense(100,
 activation='relu',
 use_bias=True))
# create and add output layer:
m.add(Dense(5,
 activation='softmax'))
m.compile(optimizer="adam",
 loss='sparse_categorical_
 crossentropy',
 metrics=['accuracy'])
```

The first call creates a sequential Keras model. With the add-function, layers are added to the model. We add an input layer with 100 neurons (or features), a hidden layer with 100 neurons, and an output layer with 5 neurons. Each neuron of the next layer is automatically connected to each neuron of the previous layer, as shown in Figure 2. In this example, we classify some data with 100 features into 5 different output classes. Some remarks on the parameters: the activation function of the hidden layer is set to rectified linear unit ('relu'), wich is defined as $y = max(0,x)$. The activation function of the output layer is set to 'softmax', which is also known as a normalized exponential function. It maps an input vector to a probability distribution consisting, in our case, of 5 different probabilities. Each probability corresponds to one of five classes, in which we classify the input vectors. The last call is the actual creation of the model using the compile-function. Different loss-functions, optimizers, and metrics can be used. In our example we use the 'sparse_categorical_crossentropy' loss function, and as a metric the accuracy. The Adam optimizer is an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. (For details on Adam, see (Kingma and Ba, 2014)).

Step 3) The next step is to train the newly created model using the prepared test data and labels:

```
m.fit(train_data, train_labels,
 epochs=20,
 batch_size=32)
```

Calling the fit-function starts the training. In our case we use the train_data and train_labels to train the model. Epochs define how many times the model should be trained using the data set. The data is always given in a different ordering to the model. The batch_size is the amount of samples which are feeded to the ANN in a single training step.

Step 4) After training, the test data is used for testing the accuracy of the model:

```
# predict the test data
prediction = m.predict(test_data)
# we count the correct predictions
correct = 0.0
# do the counting
for i in range(0, len(prediction)):
  if test_labels[i] ==
  np.argmax(prediction[i]):
    correct = correct + 1
print('Correct:', 100.0 * correct /
  len(prediction))
```

First, we call the predict function on the model to predict labels of the test_data. After that, to check how accurate the prediction with the trained model is, we count how many times the prediction equals the correct label and calculate the correctness as percentage value. In the end, we output the value to the console.

Step 5) In the last step, we persist the model by storing it in the hierarchical data format (.h5).

```
# save the model to the hard drive
m.save("mymodel.h5")
# delete the model
del m
# load model from hard drive
m = load_model("mymodel.h5")
```

After persisting the model, it can be deleted from memory and later be loaded from the hard drive using the 'load_model' function.

4 Our Cipher Type Detection Approach

In this section, we present our cipher type detection approach. First, we give a short overview of the MysteryTwister C3 challenge created by Stamp. Then, we discuss the cipher ID problem as a classification problem. After that, we present our cipher detection ANN in detail (input/hidden/output-layers, features, training and test data).

4.1 The MTC3 Cipher ID Challenge

MysteryTwister C3 (MTC3) is an online platform for publishing cryptographic riddles (= challenges). In 2019, Stamp published a cipher type detection challenge[2] on MTC3, named "Cipher ID – Part 1". The detection of the cipher type of an unsolved ciphertext is a difficult problem, since a multitude of different (classical as well as modern) ciphers exist. E.g. in the DECODE database (Megyesi et al., 2019), there is a huge collection of (historical) ciphertexts of which we don't know the (exact) type of cipher. Without knowing the type, breaking of such texts is impossible. Thus, a first cryptanalysis step is always to determine the cipher type. Different metrics, like text frequency analysis and the index of coincidence are helpful tools and indicators for the type of the cipher.

The MTC3 challenge is based on the aforementioned problem of often not knowing the type of ciphers of historic encrypted texts. The term "Cipher ID" refers to the type of used algorithm, or its "identifier". In the challenge the participants have to identify different ciphers that were used for encryption of a given dataset of 500 ciphertexts, where each is 100 characters long. The goal is to determine the type of cipher used to encrypt each message. The following ciphers were used exactlay 100 times each: simple monoalphabetic substitution cipher, Vigenére cipher, columnar transposition cipher, Playfair cipher, and the Hill cipher. The English plaintexts are randomly taken from the Brown University Standard Corpus[3]. The set of provided ciphertexts is shuffled.

4.2 Cipher ID as a Classification Problem

The general idea is to treat the detection of the cipher type as a classification problem. Each type of cipher is regarded as a disjunct class, hence, there is a monoalphabetic substitution class, a Vigenère class, etc. Figure 3 depicts the general idea. In the figure, two feature dimensions (A and B) are shown. Based on the cipher's characteristics, features have stronger or weaker influence on the output. Examples for features are the frequency of the letter 'A' or the index of coincidence. The colored dots (red, green, and blue) represent different ciphertexts. The dots are surrounded by a line showing the classes (or ciphers) each ciphertext belongs to.

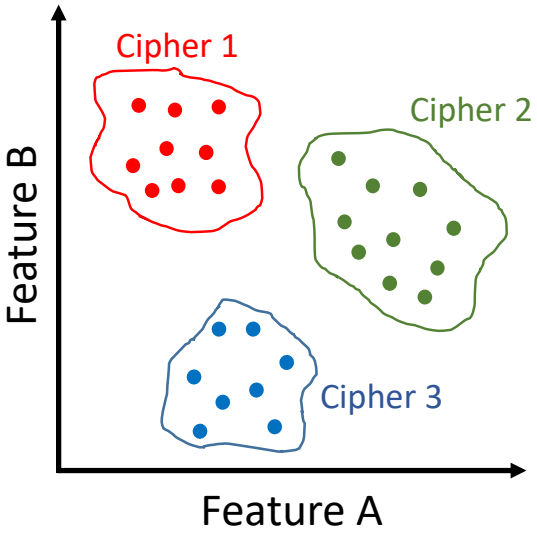

Figure 3: Ciphertexts (dots) in a multidimensional feature space. Classified into three cipher classes (red, green, blue)

With Stamp's challenge, we have 5 different classes, one for each cipher type. The ciphertexts' features are given as input vectors to an ANN which then classifies the text into one of the aforementioned classes. As output, the ANN then returns the ID of the detected cipher.

4.3 A Cipher ID Detection ANN

In the following we discuss the development of a cipher ID detection ANN based on the steps introduced in Section 3.2. Since it is a trivial step, we omit the persisting step (Step 5):

Step 1: Loading/preparing training/test data To train an ANN a sufficient amount of training and test data is needed. In the case of the cipher ID detection ANN, ciphertexts of the types which should be detected are needed. Therefore, we first implemented all 5 ciphers in Python. We also created a Python script which extracts random texts

[2] https://www.mysterytwisterc3.org/en/challenges/level-2/cipher-id-part-1

[3] Brown University Standard Corpus of Present-Day American English, available for download at http://www.cs.toronto.edu/~gpenn/csc401/a1res.html

from a local copy of the Gutenberg library. Using this script, we can create an arbitrary amount of different (English) plaintexts of a specific length. After extracting a sufficient amount of plaintexts of length 100 each, we encrypted these with the ciphers – always using randomly generated keys. We created different sets of ciphertext files with different amounts of ciphertexts for each cipher (1 000, 5 000, 50 000, 100 000, and 250 000). Thus, the total amount of ciphertexts provided to the ANN is a multiple of 5 of those numbers.

Since the ANN is not able to work on text directly, the data has to be transformed into a numerical representation. Our first idea was to directly give each letter as a number to the network, thus, having a feature vector of 100 float values. As this lead to a poor performance of our network we began experimenting with different other features, i.e. statistical values of the ciphertext. The next step shows our features and the overall ANN.

Step 2: Creating a model We experimented with different features as input values as well as with different amounts of hidden layers, widths of hidden layers, activation functions, optimizers, etc. We here now present the final ANN setup which performed best in our tests.

We use the following features:

- 1 neuron: index of coincidence (unigrams)
- 1 neuron: index of coincidence (bigrams)
- 26 neurons: text frequency distribution of unigrams
- 676 neurons: text frequency distribution of bigrams

Thus, the ANN has an input layer consisting of a total of 704 input neurons. After that, we create 5 hidden layers, where each layer has a total of

$$\left\lfloor \frac{2}{3} \cdot inputSize + outputSize \right\rfloor = \left\lfloor \frac{2}{3} * 704 + 5 \right\rfloor = 474 \quad (1)$$

neurons. Since we have 5 classes of cipher types, the output layer consists of five output neurons, each one for a specific cipher type. In Python, we created the network with the following code:

```
# sizes of layers
inputSize =  704
outputSize = 5
hiddenSize = 2 * (inputSize / 3) +
  outputSize
# create ANN model with Keras
model = keras.Sequential()
# create input layer
model.add(keras.layers.Flatten(
  input_shape=(inputSize,)))
# create five hidden layers
for i in range(0, 5) :
    model.add(keras.layers.Dense(
    (int(hiddenSize)),
    activation="relu",
    use_bias=True))
# create output layer
model.add(keras.layers.Dense(
  outputSize,
  activation='softmax'))
```

The type of the hidden layer's activation function is 'relu' and the output layer's activation function is 'softmax' (see Section 3.1).

Step 3: Training the model We trained different configurations of our model with different amounts of ciphertexts. We used different sizes of training data sets and obtained the following results (output of our test program) with our best model:

```
Training data: 4,500 ciphertexts
Test data: 500 ciphertexts
– Simple Substitution: 87%
– Vigenere: 75%
– Columnar Transposition: 100%
– Playfair: 80%
– Hill: 32%
Total correct: 74%

Training data: 24,500 ciphertexts
Test data: 500 ciphertexts
– Simple Substitution: 88%
– Vigenere: 54%
– Columnar Transposition: 100%
– Playfair: 93%
– Hill: 64%
Total correct: 79%

Training data: 249,500 ciphertexts:
Test data: 500 ciphertexts
– Simple Substitution: 97%
– Vigenere: 63%
– Columnar Transposition: 100%
– Playfair: 99%
– Hill: 70%
```

```
Total correct: 86%

Training data: 499,500 ciphertexts:
Test data: 500 ciphertexts
– Simple Substitution: 99%
– Vigenere: 63%
– Columnar Transposition: 100%
– Playfair: 97%
– Hill: 67%
Total correct: 87%

Train. data: 1,249,500 ciphertexts:
Test data: 500 ciphertexts
– Simple Substitution: 100%
– Vigenere: 69%
– Columnar Transposition: 100%
– Playfair: 99%
– Hill: 78%
Total correct: 90%
```

The first two training runs were done in a few minutes. The third test already took about an hour on an AMD FX8350 with 8 cores. The last two tests took several hours to run. Since there is a problem with the CUDA support of TensorFlow with the newest Nvidia driver in Microsoft Windows, we could only work with the CPU and not with the GPU, making the test runs quite slow.

During our tests, we saw that with increasing the size of our training data, we could also increase the quality of our detection ANN. Nevertheless, the detection rate of the Vigenère cipher and the Hill cipher is too low (between 60% and 80%). In our first experiment, ciphertexts encrypted with the Hill cipher were only correctly detected by 32% and Vigenère was only 75%. We assume, that there is a problem for our ANN to differentiate between those two ciphers, since their statistical values (text frequencies, index of coincidence) are similar.

Step 4: Testing and optimizing the model For optimizing our model (with respect to detection performance), we tested other additional features provided to the ANN. Those features are:

- Text frequency distribution of trigrams
- Contains double letters
- Contains letter J
- Chi square
- Pattern repetitions
- Entropy
- Auto correlation

The text frequencies of trigrams had no noticeable influence on the detection rate, but made the training phase much slower, since $26^3 = 17576$ additional input neurons were needed. Also, an equivalent number of neurons in the hidden layers were needed. Thus, we removed the trigrams from our experiment.

The ”Contains double letters” feature did also have no influence. We additionally realized that the double letters are also detected by the bigram frequencies. Thus, we also removed this feature. Same applies to the ”Contains letter J” feature. The idea here was, that the Playfair cipher has $I = J$, thus, there is no J in the ciphertext.

The chi square feature also had no influence on the detection rate.

With pattern repetitions, we aimed at giving the network an “idea” of the repetitive character of Vigenère ciphertexts. Unfortunately it did not help to increase the detection rate.

Entropy and auto correlation of the ciphertext were also given as features. Also no influence on the detection rate was realized.

Finally, we kept only index of coincidence on unigrams and bigrams as well as letter frequencies of unigrams and bigrams.

5 Cipher Type Detection and Solving Method for Stamp’s Challenge

To actually solve Stamp’s challenge this method brings together the following parts:

- **Cipher type detection ANN**
- Monoalphabetic substitution solver
- Vigenère solver
- Transposition solver
- Playfair solver

The method consists of the cipher type detection ANN and of solvers for each cipher despite the Hill cipher. The basic idea is the following: First, the set of ciphers is classified by the cipher detection ANN. After that, each cipher has been assigned a cipher ID. Since we know that only about 90% of the cipher types is classified correctly, we have to check each cipher type for correctness, in

order to reach a overall classification correctness of 100%. Thus, each ciphertext is then tested in a first run using its corresponding solver, despite the ciphertexts marked as Hill cipher. Hill cipher, especially in the case of a 4x4-matrix and ciphertext-only is a hard to solve cipher.

After that, all ciphertexts that could be successfully solved using the solvers are marked as "correctly classified". The remaining ciphertexts, that could not be solved using the assigned cipher type, are then tested using the three other solvers. In the end, there should only be a set of 100 ciphertexts (in the case of the Stamp challenge), which cannot be solved with the four solvers. In that case, these 100 remaining ciphertexts must be encrypted by the Hill cipher. Since there is no good solver available for Hill ciphers, which performs much better than brute-force in the ciphertext-only case, this is very time consuming or nearly impractical for the Hill cipher.

Execution time for classification with additional help of solvers Let S be the time a single solver needs to test a given ciphertext, and this time is the same for all solvers. After S time is elapsed, the solver either produced a correct result or we stop it, since we assume that the solver is the wrong one for the specific ciphertext. In the case that we do not use the cipher detection ANN, we would need an overall of $4 \cdot 500 \cdot S = 2000 \cdot S$ amount of time to test each ciphertext with 4 different solvers. If after executing all solvers exactly 100 unsolved ciphertexts remain, these are most probably texts encrypted using the Hill cipher. In that case, we solved Stamp's challenge.

Now, lets assume that testing a ciphertext using the ANN takes only a fraction of S, i.e. the classification time for a single ciphertext is T where $T \ll S$. In the real world, this is true since testing the 500 ciphertexts using our ANN only takes less than a second to be done. Generally, applying (testing) an ANN is much faster than training it. Since we know that the classification is only correct by about 90%, we have to test each ciphertext using the classified cipher type despite those classified as Hill cipher-encrypted. Lets assume that about 100 texts are classified as hill cipher, thus about 400 ciphertexts remain to be analyzed. Since we know that 90% of those 400 texts are already classified correctly, 10% of those texts remain unsolved. These 10% plus the 100 hill-cipher classified texts have now to be analyzed using all 4 solvers (this can be further optimized by only testing the remaining 10% with the three unused solvers). This leads to the following total amount of time needed for classification:

$500 \cdot T + 400 \cdot S + 40 \cdot 3 \cdot S + 100 \cdot 4 \cdot S$

which is $920 \cdot S$ is so small that it can be left out of the calculation since $T \ll S$. Thus, we have a total execution time saving of about $100\% - 100\% \cdot \frac{920 \cdot S}{2{,}000 \cdot S} = 54\%$ for the classification of the ciphertexts of Stamp's challenge.

If we assume that a solver needs about one minute to successfully solve a ciphertext, using all solvers for testing would take about $2{,}000$ minutes (about 33h). Using the ANN to reduce the amount of needed solvers, this time would now be 920 minutes (about 15h). Clearly, in the case of the ANN the time for training the network has also to be considered, which can also take several hours. Nevertheless, this time is only needed once, since the resulting ANN can be reused for classification tasks. The solvers could be executed in parallel, which further reduces the overall elapsed time.

6 Conclusion

This paper shows the current progress of our work in the area of artificial neural networks (ANN) used to detect the cipher types of ciphertexts encrypted with five different classical ciphers: simple monoalphabetic substitution, columnar transposition, Vigenère, Hill, and Playfair. For creation and training of an ANN consisting of five hidden layers, we used Google's TensorFlow library and Keras. The goal of our initial research was to solve Stamp's challenge (see Section 4.1), which required to determine the cipher type of 500 encrypted using the aforementioned five classical ciphers. The network was able to detect about 90% of the ciphers correctly. Detection rates for Playfair and Hill were too low to solve the challenge completely. Besides the creation of the ANN we also proposed a method (see Section 4) for solving the challenge using the ANN as well as different solvers, e.g. from CrypTool 2 (Kopal et al., 2014). Examples, how the solvers of CrypTool 2 can be used are shown in (Kopal, 2018). With the method, described in Section 5, about 54% execution time could be saved for solving Stamp's challenge. Another part of this paper is a survey of the related work with respect to ANN and cryptanalysis of classic ciphers (see Section 2) and an intro-

duction into the topic for the HistoCrypt audience (see Section 3).

In future work, we want to extend our network (e.g. by using different ANN architectures) and method (e.g. by finding better features)in order to detect more different and difficult cipher types. We also want to use the methods in the DECRYPT research project (Megyesi et al., 2020) to further identify unkown types of several ciphers currently stored in the DECODE database.

Acknowledgments

This work has been supported by the Swedish Research Council, grant 2018-06074, DECRYPT – Decryption of historical manuscripts.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.

Ahmed J Abd and Sufyan Al-Janabi. 2019. Classification and Identification of Classical Cipher Type Using Artificial Neural Networks. *Journal of Engineering and Applied Sciences*, 14(11):3549–3556.

B Chandra, P Paul Varghese, Pramod K Saxena, and Shri Kant. 2007. Neural Networks for Identification of Crypto Systems. In *IICAI*, pages 402–411.

Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27.

François Chollet. 2015. Keras: Deep learning library for theano and tensorflow. *URL: https://keras.io/*, 7(8):T1.

Riccardo Focardi and Flaminia L Luccio. 2018. Neural Cryptanalysis of Classical Ciphers. In *ICTCS*, pages 104–115.

Sam Greydanus. 2017. Learning the Enigma with Recurrent Neural Networks. *arXiv preprint arXiv:1708.07576*.

Mahmood Khalel Ibrahem Al-Ubaidy. 2004. Black-box attack using neuro-identifier. *Cryptologia*, 28(4):358–372.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

Nils Kopal, Olga Kieselmann, Arno Wacker, and Bernhard Esslinger. 2014. CrypTool 2.0. *Datenschutz und Datensicherheit-DuD*, 38(10):701–708.

Nils Kopal. 2018. Solving Classical Ciphers with CrypTool 2. In *Proceedings of the 1st International Conference on Historical Cryptology HistoCrypt 2018*, number 149, pages 29–38. Linköping University Electronic Press.

Beáta Megyesi, Nils Blomqvist, and Eva Pettersson. 2019. The DECODE Database: Collection of Historical Ciphers and Keys. In *The 2nd International Conference on Historical Cryptology, HistoCrypt 2019, June 23-26 2019, Mons, Belgium*, pages 69–78.

Beáta Megyesi, Bernhard Esslinger, Alicia Fornés, Nils Kopal, Benedek Láng, George Lasry, Karl de Leeuw, Eva Pettersson, Arno Wacker, and Michelle Waldispühl. 2020. Decryption of historical manuscripts: the decrypt project. *Cryptologia*, pages 1–15.

Malte Nuhn and Kevin Knight. 2014. Cipher Type Detection. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1769–1773.

Tariq Rashid. 2016. *Make your own neural network*. CreateSpace Independent Publishing Platform.

G Sivagurunathan, V Rajendran, and T Purusothaman. 2010. Classification of Substitution Ciphers using Neural Networks. *International Journal of computer science and network Security*, 10(3):274–279.

Ramon Wartala. 2018. *Praxiseinstieg Deep Learning: Mit Python, Caffe, TensorFlow und Spark eigene Deep-Learning-Anwendungen erstellen*. O'Reilly.