Evaluating Deep Learning Techniques for Known-Plaintext Attacks on the Complete Columnar Transposition Cipher

Nino Fürthauer[1*], Vasily Mikhalev[2], Nils Kopal[2], Bernhard Esslinger[2], Harald Lampesberger[1], Eckehard Hermann[1]
[1]University of Applied Sciences Upper Austria, Hagenberg, Austria
[2]University of Siegen, Germany
[*]nino.fuerth@gmail.com

Abstract

This paper examines whether deep neural networks (DNN) can learn known-plaintext attacks on plaintext-ciphertext-pairs, that were created by encrypting with complete columnar transposition. We propose a new algorithm that extends pure DNN-based prediction with additional post-processing steps to further enhance key prediction quality. Our approach is easily extensible and currently supports key lengths from 2 to 20 characters. Each key length has been empirically evaluated with plain-/ciphertext-pairs of different lengths. For plain- and ciphertexts with a length of five times the key length, our algorithm achieves a success rate of 96% which is, to the best of our knowledge, a new state of the art on deep-learning-based known-plaintext attacks against columnar transposition.

1 Introduction

The columnar transposition cipher is a historic manual cipher, where the order of the characters is changed instead of replacing characters by other characters or symbols as in substitution ciphers. It was one of the methods used by the Irish Republican Army (IRA) in the 1920s (Mahon and Gillogly, 2008). As it is a very simple cipher, it has also served as a building block for other more complex ciphers like double transposition cipher or the German ADFGVX cipher.

In the beginning of the 20th century, first known-plaintext attacks (KPAs) were successfully used by the British to break the German cipher machine Enigma. Here, so called "cribs", parts of known plaintext, were used in so-called Turing bombs to find configurations (keys) to finally decipher Enigma-encrypted ciphertexts. Today, modern cryptographic algorithms have to resist against both, ciphertext-only and known-plaintext attacks.

As machine learning and more specifically deep learning have made rapid advancements over the last years, the question arises if that technology can also be applied to cryptanalysis of classical and historic ciphers. (Greydanus, 2017) shows that recurrent neural networks are able to learn the decryption function of several substitution ciphers. For some of them, neural networks can support known-plaintext attacks, at least when short keys were used. Although there are several works focusing on substitution ciphers, there is only little research on applying deep learning for cryptanalysis of transposition ciphers.

The goal of this paper is to determine how modern deep learning architectures can assist in executing known-plaintext attacks on a transposition cipher. For testing and creating of a first prototype we started our initial research in this area by applying well-known deep learning algorithms on the complete columnar transposition cipher (all rows of the encryption rectangle have the same length (Lasry et al., 2016)). While we are fully aware that breaking a complete columnar transposition cipher in a known-plaintext scenario is a rather trivial problem, it serves as an ideal start for evaluating different machine learning methods for their suitability for attacking transposition ciphers. Our ongoing research and our here presented preliminary results will later be used to develop extended methods for attacking the incomplete columnar transposition, which is the much more difficult case. Furthermore, the presented research here builds the basis for attacking other classical and historical ciphers using deep learning techniques in planned future work.

The rest of this paper is structured as follows: Section 2 discusses the state-of-the-art of deep learning architectures for sequential data and its usage for cryptanalysis of classical and historic ci-

phers. Also, we present previous works relating to the columnar transposition cipher. Section 3 describes all steps of our proposed algorithm in detail. Section 4 summarizes our evaluation results, and Section 5 concludes this paper.

2 State of the Art and Related Work

Deep learning is a subdomain of artificial intelligence and more specifically of machine learning that focuses on the study of "deep" models (a model is the output of a training with a machine learning algorithm). Although the term is widely used, there is no consensus on what depth a model exactly requires to qualify as "deep" (Goodfellow et al., 2016). A recurrent neural network (RNN) is a neural network architecture used in deep learning for sequential data (LeCun et al., 2015). The term sequential data describes data where data points depend on other data points. An example would be a sentence where one word (= one data point) semantically depends on the surrounding words. (Hochreiter and Schmidhuber, 1997) introduce an improvement of vanilla RNN called long short-term memory (LSTM), that is able to learn faster and handle longer sequences. In (Cho et al., 2014), gated recurrent units (GRU) are proposed that, although similar to LSTM, require less parameters and are hence easier to compute. While newer attention-based architectures like Transformer (Vaswani et al., 2017) have success in many tasks involving sequential data, LSTM- as well as GRU-based RNN are still widely used. Furthermore, a combination of LSTM and GRU can outperform standalone LSTM and GRU networks on certain tasks, for example, (Ni and Cao, 2020) successfully apply that approach to sentiment analysis and (Islam and Hossain, 2021) predict exchange currency rates using a combined approach.

In recent years, deep learning approaches have been used for several tasks related to historic cryptography. (Greydanus, 2017) demonstrates that a single LSTM cell can learn the decryption function of the three polyalphabetic ciphers Vigenère, Autokey-Vigenère and Enigma. Furthermore, the author shows that RNN are able to perform basic known-plaintext attacks at least on Vigenère and Autokey-Vigenère ciphers, where accuracies exceeding 99% (Vigenère) and 95% (Autokey-Vigenère) have been achieved for keys ranging from 1 to 6 characters. (Focardi and Luccio, 2018) use neural networks for ciphertext-only attacks on simple classical ciphers (Caesar and Vigenère), while (Aldarrab and May, 2021) propose a multilingual Transformer model for decipherment of simple substitution ciphers. Deep learning approaches have also shown promising results in cipher type identification of classical ciphers based on feature engineering (Leierzopf et al., 2021a) as well as on feature learning (Leierzopf et al., 2021b).

When it comes to non-machine-learning-based attacks on columnar transposition, (Lasry et al., 2016) present a method based on two-staged hill climbing allowing the recovery of keys up to the length of 1 000 elements. This is the currently best known algorithm for attacking complete as well as incomplete columnar transposition ciphers.

3 Our Method

Our proposed method has several processing steps beside to the trained neural networks. A summary of the algorithm can be seen in Figure 1.

3.1 Problem Setup

We consider the encryption function $c = Enc(m,k)$ of a columnar transposition cipher[1] where c is the ciphertext resulting of the encryption of plaintext m with key k. As currently only complete columnar transposition is supported, $len(k)$ is a proper divisor of $len(m)$. The objective of neural network training now is to approximate a function $f(c,m) = k$, where f is in fact a known-plaintext attack. It should be noted that in columnar transposition alphabetical keys just act as kind of a mnemonic for the actual numerical key. For example, the key BLUE would evaluate to the numerical key [1,3,4,2], but so does DIME. As there are many alphabetical keys that can be mapped to that numerical key, it is impossible for a neural network to predict the exact alphabetical key. However, this is not a problem as only the numerical key is important for encryption.

3.2 Data Preparation

In a first attempt, an end-to-end network was implemented, where the whole plaintext concatenated with the ciphertext would act as an input to the neural network, but experiments showed that even short keys up to the length of 9 characters were not predicted reliably (accuracy was

[1] operation mode was write by rows, read by columns

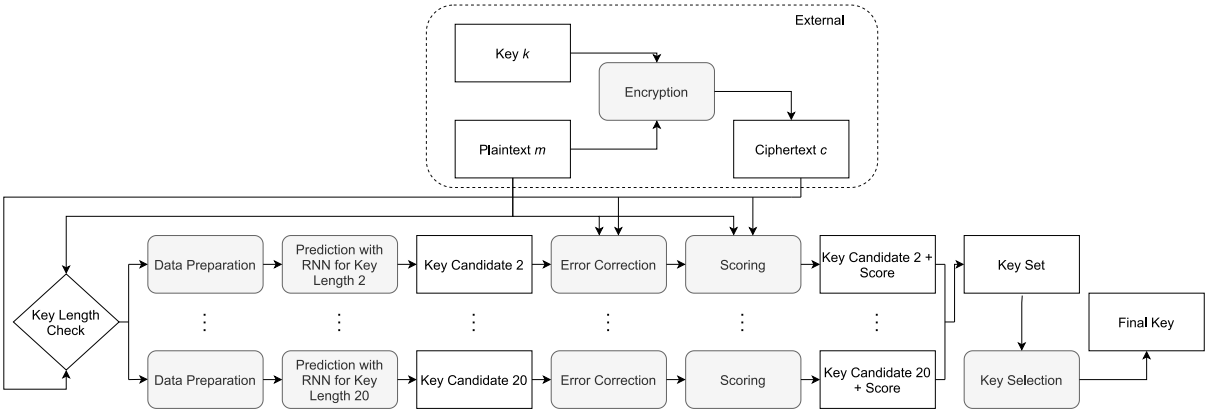

Figure 1: Flowchart of our algorithm. Key length check determines possible key lengths and for each of those one prediction iteration is executed. Finally, the key with the highest score is selected out of the proposed key set.

only 65%). Additionally, it became increasingly difficult for the model to predict correct keys with increasing plain- and ciphertext lengths. As a result, we focused on slicing the plain- and the ciphertext into blocks of equal length l, where l corresponds to the key length. Row x of the already resorted encryption rectangle is just a permutation of the x^{th} l-characters long substring of m effectively leading to $\frac{len(m)}{l}$ permutation pairs of plaintext-substrings and encryption rows. All these pairs share the same key for transposing the plaintext-substring into the ciphertext part and hence key prediction can be done on all of them. Figure 2 shows a simple example with an encryption rectangle for m = MYSECRETTEXT and k = BLUE ([1,3,4,2]). The resulting ciphertext c is MCTETTYRESEX. According to the described slicing method the three pairs [MYSE, MEYS], [CRET, CTRE] and [TEXT, TTEX] are used for key prediction. Finally, before feeding this set of pairs into the neural network, character-wise one-hot encoding is applied and both vector sequences are concatenated. For each time step t the input vector represents a concatenation of the character at position t of the plaintext-substring and the character at position t of the corresponding encryption row part.

3.3 Neural Networks

This section describes the details of the deep learning part of the algorithm.

Architecture Empirical evaluations of several architectures suitable for handling sequential data showed that a bidirectional GRU-LSTM-RNN

B	L	U	E
M	Y	S	E
C	R	E	T
T	E	X	T

→

B	E	L	U
M	E	Y	S
C	T	R	E
T	T	E	X

Figure 2: Shown above is an example of an encryption rectangle described in Section 3.2. The pair [MYSE, MEYS] is colored (blue = plaintext part of the pair, orange = ciphertext part) as this pair is used as an example input in Figure 3 and the colors should make it more clear how the input was created.

achieved the best accuracy with relative few required training samples. Other approaches tested were standalone (stacked) LSTM, standalone (stacked) GRU, encoder-decoder architectures with and without attention and Transformer, but all of those resulted in less accuracy and/or more required samples. Figure 3 shows the structure of our used model including a four time step long data example (the arrows show the flow of information). Additionally, Listing 1 shows the used Keras code, providing details of the employed architecture.

Training Data For training we generated datasets of 2 million samples of randomly generated plaintexts and keys. All characters were drawn from uppercase Latin alphabet with uniform probability. Corresponding ciphertexts were created by encrypting the plaintext with the key. Length of plaintext (and thus ciphertext) was always equal to key length l as a model's objective

Listing 1: Used network (in this case for the model for 20 character long keys).

```
# number of units of GRU, LSTM and Dense depends on key length
model = Sequential()
model.add(Input(shape=(20,54)))
model.add(Bidirectional(GRU(units=4096,return_sequences=True)))
model.add(Bidirectional(LSTM(units=4096,return_sequences=True)))
model.add(Dense(20,activation="softmax"))
model.compile(loss="categorical_crossentropy",
              optimizer=Adam(learning_rate=0.005),
              metrics=["accuracy"])
```

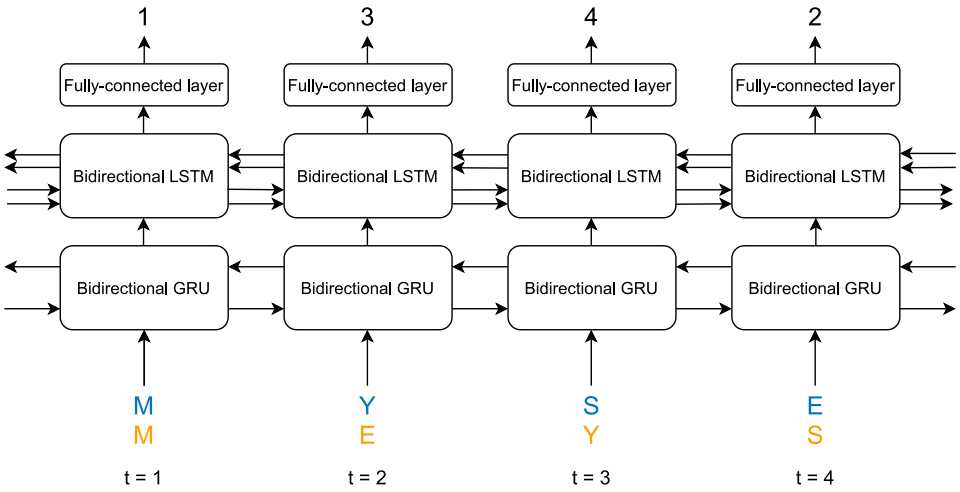

Figure 3: Architecture of used RNN including a four time step long example from Figure 2. Blue letters correspond to the same color plaintext substring in Figure 2, orange letters to the marked ciphertext characters.

is to predict keys on the string pairs described in Section 3.2 and not on a whole plain-/ciphertext combination. Training on sequences of characters randomly drawn from a uniform distribution has the advantage that resulting models cannot learn some kind of a language model and hence work regardless of the language used in the plaintext (as long as the used language is based on the Latin alphabet). The models therefore have to focus solely on learning the KPA function. If instead real-world text samples with their highly skewed frequency distributions were used, it could not be ruled out that the models additionally learned the statistical distributions of the used language, instead of merely the KPA function. It is important for the chosen approach, that the objective of our work is to analyze if it is technically possible that neural networks learn the KPA function of columnar transposition itself. As unwanted behavior could distort our results, the possibility of such has to be ruled out. This should make the result more robust, but on the other hand, this requirement gives away some information which was used in classical attack scenarios like in (Lasry et al., 2016).

Hyperparameters Unit sizes of GRU and LSTM cells were always equal but varied depending on key length l ($2 \leq l < 4$: 512 units, $4 \leq l < 6$: 1 024 units, $6 \leq l < 10$: 2 048 units and for $l \geq 10$: 4 096 units). In general it is advisable to keep unit size as low as possible as this decreases training time, memory consumption and storage space[2], but if a unit size is too small the model does not converge. In training we employ mini-batch stochastic gradient descent with a batch size of 100 and Adam, which is a very commonly used optimization algorithm, with a learning rate η of $1 \cdot 10^{-3}$. Categorical cross entropy is chosen as loss function.

Models Trials of a single universal model supporting all key lengths from 2 to 20 characters first showed promising results (accuracy was at ca. 78%), but upon closer examination it became

[2]512 units need 96.6 MB, 1 024 units 382 MB, 2 048 units 1.52 GB and 4 096 units 6.06 GB disk space to store the weights.

Model	Accuracy	Model	Accuracy
2	92.83%	12	82.58%
3	96.07%	13	80.27%
4	90.40%	14	79.71%
5	91.40%	15	78.01%
6	90.03%	16	76.20%
7	89.55%	17	74.68%
8	87.67%	18	74.76%
9	86.40%	19	72.32%
10	84.60%	20	70.53%
11	82.88%		

Table 1: Individual evaluations of models. The number on the left corresponds to the key length supported by the respective model.

clear that only key predictions of shorter keys were good enough for using them in our algorithm. The longer the key became, the worse was the prediction and hence we switched to key-length-specific models. Even though individual per model evaluations[3] show that accuracy still is decreasing for longer keys (see Table 1), final evaluations of the whole algorithm (see Section 4) proof that predictions are reliable enough for the following post-processing steps to calculate the correct key with high probability. Another drawback of a universal model for all key lengths is that it is not as easily extensible. If support for longer keys should be added, the universal model has to be retrained from scratch, while with specific models further models can be added at any time. In total, the algorithm in its current state employs 19 specific models, which requires about 74 GB disk space to store the weights. All models were trained on three Nvidia A100 40 GB GPUs resulting in training times of up to three hours per model.

Inference In deep learning, inference describes using a trained model to make predictions on live data. In inference, prediction is done on CPU so that the models are loaded into main memory instead of GPU VRAM. This is necessary as there is usually not enough VRAM to accommodate all models. For every string pair of the set provided by the data preparation step the model for the current key length makes one key prediction which leads to $\frac{len(m)}{l}$ key proposals. For example, the plain- and ciphertext from Figure 2 would lead to three key proposals. This set of proposals is then further processed in the following post-processing steps.

[3]For each model the evaluation was done with 1500 samples of corresponding length. The samples were created with the same method that was used for creating the training datasets mentioned in the paragraph "Training Data".

3.4 Majority Decision

To compress the set of key proposals for a particular key length into one key candidate, a column-wise majority decision is applied. Due to the possible multiple occurrences of a character in a plaintext-substring, it is often the case that there are multiple fitting keys. The longer the plain-/ciphertext becomes, the smaller is the number of those possible keys as more rows become available to the majority decision which means that divergent integers of single rows can be corrected better. Evaluations have shown that this procedure significantly improves the probability of getting the correct key (see Section 4 for details). An exception is the case where only one key proposal is returned by the neural network (which happens when the length of m equals l). In such a case there is no need for compressing multiple key proposals into one key and the majority decision is skipped.

3.5 Error Correction

As a result of the majority decision predicted key, integers may occur multiple times which engenders invalid keys. To correct such errors an error correction mechanism (ECM) has been introduced. It works as follows:

1. As for every multiple occurrence of an integer another one has to be missing, create an internal list with all error-causing and missing integers.

2. Distribute the elements of that list across positions where multiple occurring integers were found. The result is a valid key.

3. Calculate the ciphertext similarity score (CSS) for that key (see Section 3.6). If score is 1.0, return the found key, otherwise try a permutation of the list, and start again at step 2.

If no correct key with a CSS of 1.0 is found, the key with highest CSS is returned.

ECM is able to correct up to ε errors (i.e. missing integers). If more errors occur, the key candidate is considered to be predicted incorrectly. The choice of ε decides how many permutations are

tried and hence is a decisive factor in algorithm runtime. At worst the evaluation of key candidates requires $O(\varepsilon!)$ time, possibly making a single key prediction very slow if ε is too high. At the same time a rather big ε increases the chance of finding a key with a CSS of 1.0. We found that a value of 5 is a good trade-off between speed and accuracy.

3.6 Ciphertext Similarity Score

As mentioned above, a metric for estimating key quality is needed for selecting the best fitting key in ECM. Additionally, this selection also applies to the end of the algorithm when multiple key lengths are possible for a given plaintext m. We propose a simple score called ciphertext similarity score (CSS) that is calculated for a key candidate $\hat{k}$ according to the following equations:

$$\hat{c} = Enc(m, \hat{k}) \tag{1}$$

$$match(c_i, \hat{c}_i) = \begin{cases} 1 & \text{if } c_i = \hat{c}_i \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$CSS(c, \hat{c}) = \frac{\sum_{i=1}^{len(c)} match(c_i, \hat{c}_i)}{len(c)}. \tag{3}$$

In Equation 1 the key candidate is used for encrypting the plaintext m which results in an alternative ciphertext $\hat{c}$. c and $\hat{c}$ are then compared character-wise for equal characters (Equations 2 and 3). Finally, the number of equal characters is divided by the length of c resulting in a ratio of how many positions match between c and $\hat{c}$ (Equation 3). A CSS of 1.0 means that c and $\hat{c}$ are completely the same which indicates that a correct key has been found. Even if CSS is high but below 1.0, the key candidate might be useful for the cryptanalyst as a manual transposition of a few columns could be enough to produce the right key.

The concept of CSS is directly related to the Hamming distance (Hamming, 1950), which is defined as

$$Hamming(c, \hat{c}) = \sum_{i=1}^{len(c)} (1 - match(c_i, \hat{c}_i)). \tag{4}$$

The more characters that do not match, the bigger the Hamming distance becomes, which is why a Hamming distance of 0 means that a correct key has been found. We chose CSS over Hamming distance because it can also be interpreted as a percentage, which makes interpretation of the metric easier. Nonetheless, CSS can easily be converted to Hamming distance:

$$Hamming = len(c) - len(c) \cdot CSS. \tag{5}$$

3.7 Handling Unknown Key Lengths

Beginning with the first step of the algorithm, the data preparation, the unknown key length is needed. We experimented with a two-staged approach where an additional neural network is used to predict the key length, but only 60% accuracy, even with a limited key length range of up to 12 characters, was too low to support the remaining algorithm. Consequently, key lengths are tried exhaustively, although some lengths can be skipped as only complete columnar transposition is supported at the moment. All key lengths that are not proper divisors of $len(m)$ can be skipped which speeds up key prediction.

To sum up, a concatenation of a plaintext m and a ciphertext c acts as the input for our algorithm. Depending on $len(m)$ the steps of Sections 3.2 to 3.6 are repeated for every possible key length. The output of each iteration is a single key candidate with an associated CSS. Out of these candidates the key with the highest CSS is selected as the final result of the KPA.

4 Evaluation

Five evaluations with different plain- and ciphertext lengths were conducted for each supported key length (2 to 20 characters) using the whole proposed algorithm. In every iteration the length of the plain- and ciphertext was increased by key length l resulting in text lengths of l to $5 \cdot l$. As every increase of l characters means an additional row for the majority decision we therefore tested the algorithm with inputs ranging from 1 to 5 input pairs. Every evaluation dataset consisted of 100 samples with randomly generated plaintexts and keys, the corresponding ciphertexts were again created by encrypting the plaintexts with the keys. The advantage of individual datasets for each key length in comparison to one big mixed dataset is that it allows us to monitor the algorithm's performance in more detail. As a result, the influence of longer keys on the algorithm becomes well visible.

As a target metric we again used CSS, but because scores slightly below 1.0 can also indicate key proposals where only small manual changes can lead to the correct key, we introduce three key

Keylength	1 row			2 rows			3 rows			4 rows			5 rows		
	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.
2	96%	96%	96%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
3	70%	70%	70%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
4	94%	94%	94%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
5	98%	98%	98%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
6	100%	100%	100%	100%	100%	100%	100%	100%	100%	99%	99%	100%	100%	100%	100%
7	100%	100%	100%	100%	100%	100%	99%	100%	100%	99%	99%	100%	100%	100%	100%
8	100%	100%	100%	99%	99%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
9	100%	100%	100%	99%	99%	100%	97%	99%	100%	97%	97%	100%	100%	100%	100%
10	100%	100%	100%	95%	98%	100%	96%	97%	100%	100%	100%	100%	100%	100%	100%
11	100%	100%	100%	65%	95%	98%	94%	96%	100%	99%	99%	100%	99%	99%	100%
12	100%	100%	100%	62%	93%	99%	91%	94%	100%	99%	100%	100%	99%	100%	100%
13	100%	100%	100%	68%	93%	99%	88%	91%	100%	97%	100%	100%	100%	100%	100%
14	99%	99%	100%	44%	86%	99%	90%	99%	100%	97%	99%	100%	100%	100%	100%
15	100%	100%	100%	51%	90%	99%	89%	100%	100%	96%	100%	100%	100%	100%	100%
16	100%	100%	100%	40%	87%	98%	89%	99%	100%	94%	99%	100%	100%	100%	100%
17	96%	96%	96%	52%	87%	97%	77%	99%	100%	92%	100%	100%	98%	100%	100%
18	98%	98%	98%	31%	74%	97%	74%	99%	100%	89%	98%	100%	97%	100%	100%
19	95%	95%	95%	29%	61%	95%	75%	95%	99%	85%	100%	100%	97%	100%	100%
20	83%	87%	87%	23%	71%	85%	75%	95%	98%	90%	99%	100%	96%	100%	100%

Table 2: Results of evaluations with ciphertext similarity score (CSS)

Keylength	1 row			2 rows			3 rows			4 rows			5 rows		
	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.
2	95%	95%	95%	97%	97%	97%	100%	100%	100%	98%	98%	98%	100%	100%	100%
3	65%	65%	65%	63%	63%	63%	69%	69%	69%	66%	66%	66%	66%	66%	66%
4	79%	79%	79%	95%	95%	95%	99%	99%	99%	100%	100%	100%	100%	100%	100%
5	82%	82%	82%	90%	90%	90%	100%	100%	100%	100%	100%	100%	100%	100%	100%
6	67%	67%	67%	85%	85%	85%	99%	99%	99%	99%	99%	99%	100%	100%	100%
7	59%	59%	59%	85%	85%	85%	99%	99%	99%	99%	99%	99%	100%	100%	100%
8	56%	56%	56%	82%	82%	82%	100%	100%	100%	100%	100%	100%	100%	100%	100%
9	57%	57%	57%	69%	69%	69%	97%	97%	97%	97%	97%	97%	100%	100%	100%
10	41%	41%	83%	70%	70%	78%	96%	96%	99%	100%	100%	100%	100%	100%	100%
11	33%	33%	71%	64%	64%	95%	94%	94%	100%	99%	99%	100%	99%	99%	100%
12	30%	30%	75%	58%	58%	93%	91%	91%	100%	99%	99%	100%	99%	99%	100%
13	20%	20%	63%	62%	62%	92%	88%	88%	99%	97%	97%	100%	100%	100%	100%
14	22%	22%	51%	42%	42%	83%	90%	90%	99%	97%	97%	100%	100%	100%	100%
15	8%	8%	41%	48%	48%	88%	89%	89%	100%	96%	96%	100%	100%	100%	100%
16	10%	10%	35%	37%	37%	83%	88%	88%	99%	94%	94%	99%	100%	100%	100%
17	7%	7%	26%	44%	44%	84%	77%	77%	99%	92%	92%	100%	98%	98%	100%
18	6%	6%	30%	26%	26%	72%	74%	74%	99%	89%	89%	98%	97%	97%	100%
19	1%	1%	21%	27%	27%	57%	75%	75%	95%	85%	85%	100%	97%	97%	100%
20	2%	11%	28%	20%	48%	74%	75%	95%	98%	90%	99%	100%	96%	100%	100%

Table 3: Results of evaluations with key similarity score (KSS)

quality categories:

- Perfect: Score = 1.0
- Good: Score ≥ 0.9
- Acceptable: Score ≥ 0.8

Table 2 shows the results of the evaluation. It can be seen clearly that with increasing key length, 2 rows become too few to reliably predict a correct ("perfect") key. This is not surprising as there can be no majority decision based on two values. If only one row is added, the number of predicted keys with CSS 1.0 increases significantly. In general, with increasing plain- and ciphertext lengths, the algorithm works better. For example, when 5 rows are provided, probability for predictions with a CSS of 1.0 is at least 96%, even for long keys like 20-character long ones.

In the case where the ciphertext had the same size as the used key (columns "1 row") there were some unexpected results. Partially (mainly for short keys) having higher perfect values than plain- and ciphertexts of four times the length, we hypothesized that for such short plain-/ciphertext-combinations there are multiple fitting keys thus the transposition is not unique. To test this, we introduced another metric called key similarity score (KSS). This score is virtually the same as CSS,

only that this time not two ciphertexts are compared for similarity, but the predicted and the real key. KSS therefore measures how many positions of the predicted key match with the key that was used for creating the ciphertext. To illustrate the difference between CSS and KSS, we consider m = HELL and c = EHLL. The originally used key is k = [2,1,4,3] . When the algorithm predicts the also suitable key $\hat{k}$ = [2,1,3,4], CSS is 1.0 although it is not the original key. However, KSS is as low as 0.5 as only half of the key integers match between k and $\hat{k}$. The evaluation described above was repeated with this new metric and then results were compared (see Table 3). For reasons of efficiency, the three categories remained the same, but the perfect classification was of special interest for this experiment. The idea was that if fewer key predictions with score 1.0 occur with KSS than with CSS, there have to be multiple fitting keys. The comparison of Table 2 and Table 3 clearly shows that with increasing key length only a decreasing fraction of the predicted keys with score 1.0 equals to the actually used keys, which confirms our hypothesis of multiple fitting keys.

5 Conclusion

This paper proposes a deep-learning-based algorithm for known-plaintext attacks on complete columnar transposition. Using a GRU-LSTM-hybrid architecture and additional post-processing steps, we achieved a high accuracy of at least 96% in predicting keys up to the length of 20 characters when plain-/ciphertexts of length $5 \cdot l$, where l is the used key length, are given. For shorter plain-/ciphertexts the accuracy is lower, but even then, predicted keys, if at all, suffer from few mistakes in the predicted key that can be corrected by an experienced cryptanalyst. To take full advantage of all capabilities of our algorithm, the provided plain-/ciphertext length should be at least $3 \cdot l$ as this results in the minimum of required rows to make a majority decision.

Currently our prototype is limited to a maximum key length of 20 characters, but the algorithm itself is easily extensible as only more trained models have to be added to cover longer keys. Capabilities of current RNN architectures are the only limiting factor when it comes to key length. We have also tested our architecture for keys of size 25, but accuracy was too poor for using the predictions for the further algorithm. In future work it should be evaluated if it is possible to tweak the GRU-LSTM-hybrid architecture so that it can also be applied to 20+-character long keys or if an even better architecture can be used. Advancements in deep learning have been tremendous over the last years, so we think that it is not unlikely that with more powerful networks and hardware our approach can be extended to far longer keys in the years to come. The main drawback of loading more models into the algorithm is that memory requirements increase even more. This could be avoided if the key length predictor mentioned in Section 3.7 can be refined so that it achieves very high accuracy. Almost perfect accuracy is critical in such a multi-staged approach as a mistake in the first stage renders the rest of the algorithm useless.

This evaluation worked only with random plaintext. Using normal text from different languages would train models for real-world examples. It would be interesting how much this improves the accuracy. Another constraint is the limitation on complete columnar transposition. Further work should aim to overcome this limitation to make the algorithm more applicable in real-world text samples which are encrypted using the incomplete columnar transposition cipher.

Acknowledgments

This work has been supported by the Swedish Research Council (grant 2018-06074, DECRYPT - Decryption of historical manuscripts) and the University of Applied Sciences Upper Austria for providing access to a Nvidia DGX A100 server.

References

Nada Aldarrab and Jonathan May. 2021. Can Sequence-to-Sequence Models Crack Substitution Ciphers? *arXiv preprint arXiv:2012.15229*.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078*.

Riccardo Focardi and Flaminia L. Luccio. 2018. Neural Cryptanalysis of Classical Ciphers. In *ICTCS*.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT press.

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

Richard W. Hamming. 1950. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural computation*, 9(8):1735–1780.

Md Saiful Islam and Emam Hossain. 2021. Foreign Exchange Currency Rate Prediction Using a GRU-LSTM Hybrid Network. *Soft Computing Letters*, 3:100009.

George Lasry, Nils Kopal, and Arno Wacker. 2016. Cryptanalysis of columnar transposition cipher with long keys. *Cryptologia*, 40(4):374–398.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *nature*, 521(7553):436–444.

Ernst Leierzopf, Nils Kopal, Bernhard Esslinger, Harald Lampesberger, and Eckehard Hermann. 2021a. A Massive Machine-Learning Approach for Classical Cipher Type Detection Using Feature Engineering. In *International Conference on Historical Cryptology*, pages 111–120.

Ernst Leierzopf, Vasily Mikhalev, Nils Kopal, Bernhard Esslinger, Harald Lampesberger, and Eckehard Hermann. 2021b. Detection of Classical Cipher Types with Feature-Learning Approaches. In *Australasian Conference on Data Mining*, pages 152–164. Springer.

Thomas Mahon and James Gillogly. 2008. *Decoding the IRA*. Mercier Press Ltd.

Ru Ni and Huan Cao. 2020. Sentiment Analysis Based on GloVe and LSTM-GRU. In *2020 39th Chinese Control Conference (CCC)*, pages 7492–7497. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.