UMEÅ UNIVERSITY

MASTER THESIS

Extracting Information from Encrypted Data using Deep Neural Networks

Author:
Linus LAGERHJELM

Supervisor:
Kalle PROROK

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Interaction, Technology and Design

Department of Applied Physics and Electronics

January 21, 2019

UMEÅ UNIVERSITY

Abstract

Faculty of Science and Technology

Department of Applied Physics and Electronics

Master of Science in Interaction, Technology and Design

Extracting Information from Encrypted Data using Deep Neural Networks

by Linus LAGERHJELM

Keywords: Neural networks, Machine Learning, Cryptography, DES, LSTM, CNN, Cryptanalysis

In this paper we explore various approaches to using deep neural networks to perform cryptanalysis, with the ultimate goal of having a deep neural network decipher encrypted data. We use *long short-term memory* networks to try to decipher encrypted text and we use a convolutional neural network to perform classification tasks on encrypted MNIST images. We find that although the network is unable to decipher encrypted data, it is able to perform classification on encrypted data. We also find that the networks performance is depending on what key were used to encrypt the data. These findings could be valuable for further research into the topic of cryptanalysis using deep neural networks.

Acknowledgements

I would like to thank my supervisors and all involved employees at Omegapoint for the cooperation, for providing valuable insights and excellent guidance. Furthermore, I would like to thank everyone who have read this article during the course of the project for their efforts and for their help to make this paper better than it ever could have been otherwise.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**
1.1 Research questions 2

2 Theory **3**
2.1 Artificial Neural Networks 3
2.1.1 Multilayer perceptron 4
2.1.2 Convolutional Neural Networks 5
2.1.3 Long Short-Term Memory 6
2.2 Cryptography 7
2.3 Cryptanalysis 8
2.4 Cryptographic algorithms 9
2.4.1 DES 10
2.4.2 Modes of operation 11
2.5 Attacks on cryptographic systems 12
2.5.1 Brute force & Dictionary attacks 12
2.5.2 Frequency analysis 12
2.5.3 Chosen Plaintext Attack 13
2.5.4 Ciphertext-only attack 14
2.5.5 Differential Cryptanalysis 14
2.6 Neural Cryptanalysis 14

3 Method **16**
3.1 Hyperparameters 16
3.2 Model evaluation 16
3.3 Data representation 17
3.4 Our approach 17
3.5 Tools 18
3.6 DES Implementation 19

4 Experiments **20**
4.1 Experiment 0 20
4.2 Experiment 1 21
4.3 Experiment 2 22
4.4 Experiment 3 23
4.5 Experiment 4 24
4.6 Experiment 5 25
4.7 Experiment 6 26

5 Discussion **28**

5.1 Conclusions & Practical Uses 30
5.2 Future work 31

A Source code **36**

List of Abbreviations

AES	**A**dvanced **E**ncryption **S**tandard
ANN	**A**rtificial **N**eural **N**etworks
AWS	**A**mazon **W**eb **S**ervices
CBC	**C**ipher **B**lock **C**haining
CEC	**C**onstant **E**rror **C**arousel
CNN	**C**onvolutional **N**eural **N**etwork
COA	**C**iphertext **O**nly **A**ttack
CPA	**C**hosen-**P**laintext **A**ttack
DES	**D**ata **E**ncryption **S**tandard
DNN	**D**eep **N**eural **N**etworks
ECB	**E**lectronic **C**ode **B**ook
IV	**I**nitialization **V**ector
KPA	**K**nown **P**laintext **A**ttack
LSTM	**L**ong **S**hort-**T**erm **M**emory
MLP	**M**ulti **L**ayered **P**erceptron
MSE	**M**ean **S**quared **E**rror
RNN	**R**ecurrent **N**eural **N**etworks
RSA	**R**ivest **S**hamir **A**ldeman
XOR	e**X**clusive **OR**

Chapter 1

Introduction

Given the wide use of cryptographic applications throughout all different parts of society[1], it is of common interest that these algorithms remain as secure as they are set out to be. Verification of the algorithms integrity can only be done by continuously attempting to break them and be transparent about the success of these attempts, which is what is done in this study where we use neural networks to study encryption algorithms.

Informally an encryption algorithm is a mapping function between a plain text and a cipher text. Albeit a remarkably opaque function that is designed to be impenetrable, but a function nonetheless. This forms the foundation for this study, as Cybenko has proven [1] that a feed forward network can, simply put, approximate a continuous function with arbitrary precision (we will cover this in more detail in section 2.1.1). On basis of this theorem, and the fact that the required computing resources are now available, it seems likely that a neural network would be able to break encryption algorithms.

At a glance, using neural networks to break cryptographic algorithms has the theoretical qualifications to be successful, however, there has been few attempts to put it to the test. To the knowledge of the author at the time of writing, there are only a handful of papers that does the same thing. In one of these papers, Alani uses a neural network to attack the cryptographic algorithms DES and 3DES [2]. When using (on average) 2^{11} plain-text/cipher-text pairs, the mean squared error (MSE) for recreating the plain text from cipher text was successfully decreased below 10^{-2}.

In this project we perform a study that bear a striking resemblance to that of Alani as we also use neural networks to try to decrypt data that was encrypted with DES; however, differentiators between this study and that of Alani is that we explore different network architectures for the recreation of plain text, we also analyze the crypto in different ways, other than simple decryption. Finally, the main contribution imposed by this study is that we apply the cryptanalysis to real world data instead of sampling random bits which is what Alani did.

The ambition for this report is that it should be read by other machine learning researchers, however, the target audience is researchers within almost any theoretical field and thus, significant effort has been put into detailed explanation of relevant theory, both in the field of Machine Learning as well as Cryptography.

This research project was carried out during the period September 2018 - January 2019 in cooperation with Omegapoint. Omegapoint is a software security consultant company that specializes in software security and digital transformation [3].

[1]e.g., online banking, passwords for applications and military information

1.1 Research questions

During the work with this study we have been trying to answer a primary research question:

> Is it possible for Neural Networks to derive information about plain text that has been encrypted with DES by studying the cipher text? If yes, is it possible to derive enough information to make the network recreate the plain text?

To help us answer this question we break it down into three sub questions which is what we design our experiments to answer. When we have answered these questions, we can also answer the main question. The three questions read as follows:

- How to represent encrypted data to the network?
- Can a long short-term memory network (LSTM) recreate plain text that has been encrypted with DES?
- Can a convolutional neural network predict the class of the underlying data by looking at its encrypted correspondence?

Chapter 2

Theory

In this chapter we will cover the relevant theory for this paper. It starts with a deep dive into the theory of Artificial Neural Networks and the Universal approximation theorem as well as the two models used in this study, multilayer perceptron and Long Short-Term Memory (LSTM). Later in this section we will also cover the details about the cryptography and cryptanalysis.

2.1 Artificial Neural Networks

As most papers on the topic will tell you, artificial neural networks (ANN) is a set of algorithms which are inspired by the idea of how the brain works [4]. It usually builds upon small computational units that receives sensory inputs and by applying activation logic produces an output. Sensory input to an ANN is usually a vector of some sort and the output is one or more real values. The simplest kind of an ANN is the perceptron algorithm [5], which technically does not qualify as a network but since this paper is concerned with the multilayered perceptron (that indeed is a network) we will briefly cover it here.

A perceptron produces its output by computing the function:

$$y = f(\sum_{k=1}^{n} x_k \cdot W_k + \zeta) \tag{2.1}$$

where x_k is the k-th input, W_k is the k-th weight and f is an activation function. The ζ-term is known as bias and is trained the same way as the weights. While the weights determine the slope of the activation function, the bias term defines the shift of the function. The activation function can be any function but is usually chosen as the logistic function. The main goal of the ANN is that it should be able to *learn*.

Multiple methods exist for learning but the one that is used in this project is called *supervised learning*. The basic idea [6] of the supervised learning technique is to let the network produce an output given an input. After that, the difference $\widetilde{y} - y$ between the provided output and the desired output is used to update the weights of the network, in order to decrease this difference for the following inputs. This process of weight correction is what is referred to as *learning*.

There are a lot of different ANN models that are optimized for different use cases. In the remainder of this section we will cover the details of the three used in this study.

Weights and weight correction is a core concept in these models, in the above description as well as should be made obvious in later sections, we know what to correct the weights *to* but it fails to address what the weights should be updated *from*. At its core this question addresses how to initialize the weights. Research has been put into the question of how to select the best initial weights to a network [7] and a common way, that is also used exclusively in this project is *Xavier initialization* or *Glorot normal initialization* after the name of the author of the original paper.

To understand why a special initialization is needed, we must consider what will happen to the network if the weights are initialized too small on the one hand or on the other hand, are initialized to large. In the case where the weights are initialized too small, then the input signal to the network will also shrink and in the most extreme case, shrink too small to be useful. Similarly if the weights are initialized to large and we consider the most extreme case, the signal becomes too large to be useful. If we use a random initialization of the weights, there are no guarantee that we will not end up in, or arbitrarily close to, one of these cases. Instead we want to select weights to be right in between these cases and this is done using Xavier initialization.

In Xavier initialization the goal is to initialize the weights using the same distribution as the input data, typically Gaussian or uniform. The weights are randomly initialized, using the specified distribution, to have zero mean and the variance, as defined in the original paper:

$$Var[W^i] = \frac{2}{n_i + n_{i+1}} \tag{2.2}$$

WhereW^i refers to the weights of layer i, n_i is the number of hidden neurons in the current layer i and n_{i+1} is of course the number of hidden neurons in the next layer. By using this kind of weight initialization, we can initialize all layers at once before the training and thus eliminating the need to pretrain each layer separately with an unsupervised method.

2.1.1 Multilayer perceptron

Perceptrons are not applicable to most real world problems as they are only capable of binary linear classification [6]. To solve this problem the simple perceptron algorithm must be generalized to a model which supports multiple layers with multiple perceptrons in each layer.

When broken down, the learning rules in a multilayered perceptron network is the same as for the simple perceptron network as it too depends on the difference between the computed and actual output $\widetilde{y} - y$. However, updating the weights of the hidden layers is not as straight forward [6] as simply applying this difference. Introducing a method called *back propagation* which works by propagating the error from the output layer backwards through the network, updating each weight according to the formula:

$$w_{i,j} = w_{i,j} + \alpha \cdot \alpha_i \cdot \Delta_j \tag{2.3}$$

where Δ_j corresponds to an error that has been adjusted to the layer j. The variable α is a hyper parameter to the model called *learning rate* which determines how much the network should adjust in each correction. The factor α_i exists to allow each weight to have an individual learning rate.

By introducing the complexity of a hidden layer, we are able to represent any continuous function of n variables with arbitrary precision [1][6, p. 734].

A variation of this network type is called *Cascade-forward network*, which is the same [8] as the previously described multilayered perceptron except for the addition of an extra weight connection directly from the input layer to each succeeding layer.

2.1.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a variant of the multilayered perceptron. However, a CNN is typically deeper [9] than the usual MLP as it works by feature extraction, storing different kinds of features in different layers. In the experiments performed in this project, two different layer types were used that are specific to a CNN. A convolutional layer and a max pooling layer.

Formally, the convolutional layer is defined [10] as:

$$\sigma(b + \sum l = 1^p \sum m = 0^q w_{l,m} a_{j+l,k+m}) \tag{2.4}$$

Where $w_{l,m}$ is weights of a filter kernel of size $p \times q$ in which p and q may or may not have the same value but usually does. The other factor, $a_{j+l,k+m}$, represents the pixels of the image. On a higher level of abstraction this can be thought of as a filter kernel, a matrix of size $p \times q$, is placed on every unique position j, k of the image. In every placement of the kernel on the image, an element wise multiplication of the kernel and the sub-matrix $j + l, k + m$ of the image is performed and then the product of the multiplications are summarized to a single scalar. By doing this, the image is reduced from its original size to a new, smaller image which contains features from the previous image, because of this property, these layers are sometimes referred to as feature maps. Such features could for example be the edges between objects in the image. In order to give the network a reasonable chance of making a correct classification, multiple features have to be extracted and thus, multiple convolutional layers are stacked [10].

The second layer type used in our experiments that are unique for the CNN is a *max pooling layer* which is somewhat similar to the convolutional layer [10]. Much like the feature map, a max pooling layer consists of a filter kernel that is convoluted over the input matrix. The difference between the two types is that while the feature map extracts features from an image, the max pooling layer works like a contractor of the mapped features. It consists of a smaller kernel than the feature map and in every step returns the largest number within the region. One way to imagine this is that the network does not need to know the exact pixel that contains a given feature, only the rough location.

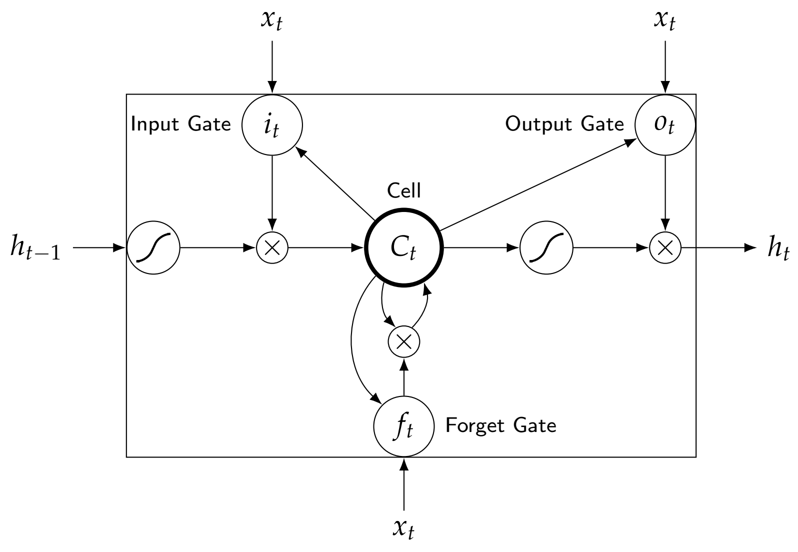


FIGURE 2.1: An illustration of a LSTM cell

2.1.3 Long Short-Term Memory

Although effective in some areas, the multilayer perceptron fails to capture contextual information which is crucial to some types of applications e.g. speak recognition, music composition or written prose [11]. This problem is addressed by designing a new architecture type for the network, Recurrent Neural Network (RNN). In an RNN, data that is outputted from a cell is in the next step fed back into it together with new input. This mechanism allows an RNN to remember information it has been exposed to in the past.

This is great in theory, however, RNNs are difficult to train in practice as they suffer from the vanishing gradient problem [12][13] and fails to capture long term dependencies, i.e. it only has a relatively short memory [11].

To address this problem, Horchreiter (1997) proposed a specialization of the traditional RNN architecture [14], Long Short-Term memory or LSTM as they will be referred to from hereon. The LSTM does not suffer from the vanishing gradient problem nor does it struggle to learn long term dependencies.

Figure 2.1 shows an illustration of an LSTM cell. In the figure, x_t denotes the input to the cell at time t. The cell consists of a few important parts. The most central is the *Constant Error Carousel* (CEC) denoted c_t in the figure. The CEC can be thought of as a stream of information flowing through the cell and it consists solely of its own output from the previous step, i.e. h_{t-1}.

Aside from the CEC, an LSTM-cell consist of three *gates*. Input, output and forget. These gates conceptually resembles floodgates and inflows on the flow that is the CRC. The *forget gate* (f_t) is a function that acts as a floodgate and it determines how much of the previous information (h_{t-1}) that should be kept. It does so by looking at h_{t-1} and x_t and produce a real value in range $(0, 1)$, where everything is kept at 1 and everything is discarded at 0. Formally, the forget gate (as well as all the other gates) is defined by Olah [15] as:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.5}$$

where σ is the logistic function, W_f is the weights and b_f is the bias. Inflow to the stream is done via the *input gate* (i_t) which determines how much of the new input should be added to the information flow. Input to the information stream really is a matter of replacing old information with new information which is why two equations are needed. In the below equations i_t denotes the values to update and $\widetilde{C}_t$ the new values:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.6}$$

$$\widetilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.7}$$

i_t and $\widetilde{C}_t$ are then multiplied to get the new value and combined with the information left after the forget gate to build the new internal state (C_t):

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{2.8}$$

At this stage, the CEC is updated and the cell is all but ready to produce an output. C_t is not the final output, however, it is filtered through the *output gate*. The output gate looks almost identical to the two other gates:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.9}$$

Finally the output from the cell is defined as:

$$h_t = o_t * tanh(C_t) \tag{2.10}$$

There exist multiple variations [16] of the traditional LSTM model, none of which have been shown to significantly improve the standard model in terms of accuracy.

2.2 Cryptography

It is likely that even the uninitiated reader already have an idea about what *cryptography* is. It is also likely that this idea resembles the definition provided by Oxford Dictionary (2006), namely that cryptography is *"the art of writing or solving codes"*. At the same time as this definition proves a basic idea about the nature of encryption it fails to provide enough rigor to be of actual use in this context. Instead we borrow our definition of cryptography from Menezes et. al [17]:

> "**Cryptography** is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication"

Using this definition it is clear that cryptography is a formal science rather than a form of art as suggested by Oxford Dictionary. The most fundamental goal of cryptography is the creation of *encryption schemes* (sometimes also referred to as *crypto*).

[18] Such a scheme must at least satisfy the property:

$$D_k(E_k(m)) = m \tag{2.11}$$

Where D is the decryption function, E is the encryption function, k is the key and m is the message which is a string of text of arbitrary length. Put simply, the above equation states that given an encryption algorithm E and a key k, there exists a decryption algorithm D such that, given the same key k, the original message can be retrieved.

In order for equation 2.11 to be of any practical use, we must include some additional requirements. A principle that is common within cryptography is coined by Auguste Kerckhoff [19] and, when translated to English, reads as follows:

> "A cryptosystem should be secure even if everything about the system, except the key, is public knowledge."

This means that we must choose E_k in a way that m can not be retrieved unless k is known. Given this, we need just one more piece for our definition of cryptography to be complete and this piece deals with the secrecy of E_k. In 1949 Shannon [20] proved that an encryption scheme can exist such that it is impossible to retrieve m without knowing k. This encryption scheme requires a random sampling of the key bits and also that every key bit is unique for every bit of the message. Or in other words: perfect secrecy requires [18] the key to be as long as the message to encrypt. Further more, the key must not be reused.

Although the method mentioned above will yield an unbreakable crypto, it is not put to use in practice, instead a shorter key is used. While this destroys the perfect secrecy, it is supposed to still provide enough secrecy for practical use. I.e., it would take a very long time for a computer to decipher the message.

Lastly, there is another concept that must be addressed in a cryptographic system which is the *avalanche effect* [21], that stipulates that if the input changes slightly, the output is changed significantly. In the strictest case it means that if a single bit is flipped in the input then 50% of the output bits are changed.

2.3 Cryptanalysis

In the previous section we learned that although it is possible to achieve perfect secrecy this is not something that is put to use in practice [18]. Perfect secrecy requires keys as long as the message they are encrypting to be used. Furthermore it requires that keys are only ever used once.

The main practical problem is the key exchange as today, data is transmitted with high frequency, in large quantities and usually one party communicates with multiple different parties separately. This means that one would, continuously, have to distribute several gigabytes (if not more) worth of keys to multiple counter parties. Additionally, this distribution has to be done over a perfectly secure channel or the security of the system is no better than the security of this channel.

Common practice is instead to introduce the notion of *practical secrecy*. This notion is derived from another one of Kerckhoff's principles [19] which states: "*The [cryptographic] system must be practically, if not mathematically, indecipherable*". What this means is that when a crypto is used, it need not be perfectly secure. It just needs to be secure enough to fulfil the requirements of difficulty [18]. The book *Introduction to modern cryptography* goes into much detail about what is required from a cryptographic scheme in order to be practically secure [18, pp. 47 - 60]. The key takeaway from the discussion is that the algorithm should be breakable by a super computer with no more than probability ϵ if allowed to run for no more than time t. Examples of values for ϵ and t may be 10^{-30} and 200 years. To make the algorithms resilient to increase of computing power, a cost parameter n is introduced which determines the computational cost of breaking the scheme. This parameter n is usually analogous to the key size.

This is the background on which modern day *cryptanalysis* is based. *Handbook of applied cryptography* [17, p. 15] defines *cryptanalysis* as: "*the study of mathematical techniques for attempting to defeat cryptographic techniques*". In less formal terms it could be said that someone who practice cryptanalysis works with the goal of finding a way to reveal the secrets introduced by a cryptographic schema. In this paper we have studied the case where the attacker is interested in retrieving a plain text message given the corresponding cipher text. A goal that is achieved using a method known as *known plaintext attack* or *chosen plaintext attack* which is explained in section 2.5.3.

2.4 Cryptographic algorithms

In this section we will explain the cryptographic algorithm used in this study, DES, as well as covering some details about block ciphers. Before explaining how the DES algorithm works, however, it is necessary to define some basic concepts of cryptography which is done below.

Block cipher is a way of doing encryption [22] that specifies a function that accepts a plain text of size n, a key k and outputs a cipher text of size n. Examples of sizes of n and k is 64 and 128, all units are given in bits. In practice, messages that are to be encrypted are of course longer than 64 bits, thus requiring the message to be split up into blocks. In some cases special padding is also needed to make the text fill the entire block. It should be noted that there exists an alternative to block cipher known as *stream cipher*.

Key in cryptography is used in much the same way as the word is used in everyday speech and thus should provide an intuitive understanding. For the sake of clarity; however, it would seem appropriate to also provide the exact definition used when discussing the concepts in this report. A key is a piece of information [18] that is used to encrypt messages. In the case of private key encryption with symmetric keys the key is shared among all parties of the encrypted communication and secret to everyone else (private). The same key is also used for encryption and decryption (symmetric).

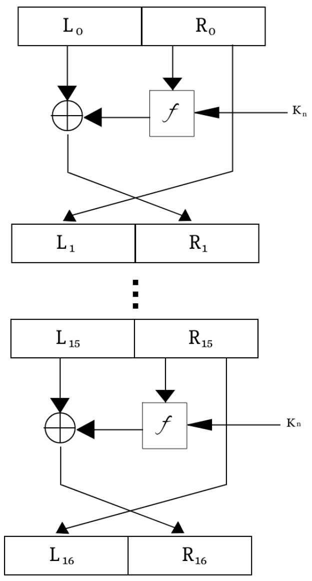


FIGURE 2.2: An illustration of the rounds in DES

2.4.1 DES

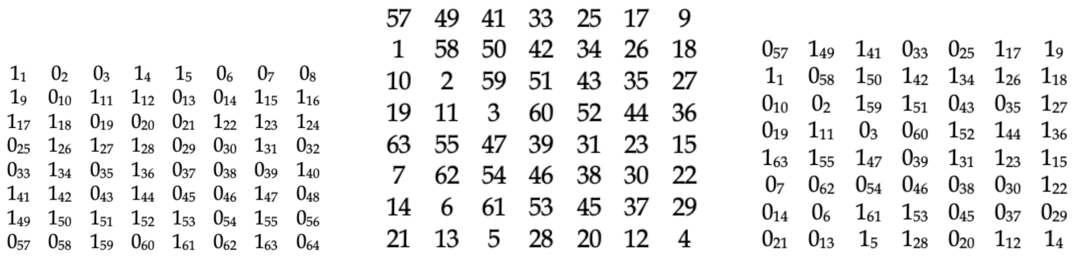
Data Encryption Standard (DES) is a federal encryption standard [23]. It describes a symmetric block cipher that uses a block and key size of 64 bits. Encryption in terms of DES is done in 16 rounds, meaning that to fully encrypt a block, the encryption operations are applied 16 times.

Before the rounds are applied to the message, an initial permutation of the block is applied. The inverse of this permutation is applied after all rounds. Scrambling the bits using permutation tables is an important concept of the DES algorithm and an example of how this works is depicted in figure 2.3, which shows how the original key bits are permuted using such a table. A permutation table are read left to right, top to bottom and consists of indices such that the first element in the permutation table determines what index from the original matrix to put as the first index in the output matrix.

Each round in DES uses a different key which is sampled from the original key in the following way: An initial permutation is applied, this is the permutation described in figure 2.3. It is worth noting that in this step, the permutation table only has 56 entries which means that the resulting key will also have 56 entries and every eight bit is skipped. This means that the effective key length in DES is only 56 bits and not 64 bits as is suggested by the call for a 64 bit key. Instead the last 8 bits are used as parity bits. When the key is permuted from a 64 bit to a 56 bit key it is split into a left and a right half, L & R. Each of the two halves are left shifted, one or two places, 16 times and after each shift, L and R are concatenated and LR becomes the key for the corresponding round.

In each round of the DES encryption algorithm, the block is split into the halves L and R as can be seen in figure 2.2. The right block is simply moved to become the left block for the next round while the left block becomes the right block through the transformation:

$$R_n = L_{n-1} \oplus f(R_{n-1}, K_n \tag{2.12}$$



1_1	0_2	0_3	1_4	1_5	0_6	0_7	0_8
1_9	0_{10}	1_{11}	1_{12}	0_{13}	0_{14}	1_{15}	1_{16}
1_{17}	1_{18}	0_{19}	0_{20}	0_{21}	1_{22}	1_{23}	1_{24}
0_{25}	1_{26}	1_{27}	1_{28}	0_{29}	0_{30}	1_{31}	0_{32}
0_{33}	1_{34}	0_{35}	1_{36}	0_{37}	0_{38}	0_{39}	1_{40}
1_{41}	1_{42}	0_{43}	1_{44}	0_{45}	0_{46}	1_{47}	0_{48}
1_{49}	1_{50}	1_{51}	1_{52}	1_{53}	0_{54}	1_{55}	0_{56}
0_{57}	0_{58}	1_{59}	0_{60}	1_{61}	0_{62}	1_{63}	0_{64}

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

0_{57}	1_{49}	1_{41}	0_{33}	0_{25}	1_{17}	1_9
1_1	0_{58}	1_{50}	1_{42}	1_{34}	1_{26}	1_{18}
0_{10}	0_2	1_{59}	1_{51}	0_{43}	0_{35}	1_{27}
0_{19}	1_{11}	0_3	0_{60}	1_{52}	1_{44}	1_{36}
1_{63}	1_{55}	1_{47}	0_{39}	1_{31}	1_{23}	1_{15}
0_7	0_{62}	0_{54}	0_{46}	0_{38}	0_{30}	1_{22}
0_{14}	0_6	1_{61}	1_{53}	0_{45}	0_{37}	0_{29}
0_{21}	0_{13}	1_5	1_{28}	0_{20}	1_{12}	1_4

FIGURE 2.3: (1) original key, (2) permutation table, (3) permuted key

Where $\oplus$ is the bitwise XOR operation and f is a function that can be said to be the heart of the algorithm. It starts by expanding the block from 32 to 48 bits, by duplicating some bits, followed by xor:ing the 48 bit block with the n:th key. The result of this xor is then split into 8 blocks of 6 bits each. From the 6 bit block, a 4 bit block is selected using a precomputed table. After the selection function, the eight 4 bit blocks are concatenated into a 32 bit vector that is permuted using another permutation table.

After the completion of each round, the process starts again 15 times and after the completion of the 16th round, the encryption of the block is complete. The algorithm as it is described above assumes that each block is exactly 64 bits and that there are only one block, assumptions that rarely holds true in practice. Different techniques to circumvent some of the problems is discussed in section 2.4.2.

2.4.2 Modes of operation

DES is a block cipher algorithm and block ciphers operates, as previously mentioned, on blocks of a fixed size, typically 64 bit. 64 bit is not a lot of data and messages that are sent in practice are typically longer. To circumvent this limitation, the message is split into blocks of 64 bit, adding padding to the last block if the message length is not divisible by 64. In this report, two ways of encrypting continuous blocks of data is used: ECB and CBC.

Under ECB mode, data is split into blocks of the correct size and encrypted, block by block [17]. Advantages of this method is that it is easy to implement and also resilient to bit errors, as an erroneous bit in a block only affects that one block. However, ECB mode is not particularly secure as identical blocks yield identical cipher text. This has the drawback that patterns within the data are not obfuscated and it also allows an attacker to insert new data by replacing encrypted blocks.

CBC mode addresses the weaknesses of ECB mode by introducing a concept called initialization vector (IV) [17] which is used in the encryption of the first block. Before the block is fed to the encryption algorithm, it is *XOR*:d with this initialization vector. For each succeeding block, XOR with the cipher text of the previous block. This makes each block dependent on all the previous blocks of message as well as the key and IV, thus preventing identical block from having identical cipher text. It also prevents an attacker from switching cipher text blocks. IV does not need to be kept secret the same way as the key, however, it should be random and unique.

2.5 Attacks on cryptographic systems

When performing cryptanalysis, it is done with the goal to retrieve information about the plain text given a cipher text, something that can be done in various different ways. Throughout this paper we shall be referring to these methods of extracting information as *attacks*. In this section we will cover some of the current attacks against crypto systems that are used in practice today in order to provide the reader with a basic understanding of the current landscape of cryptanalysis, how our methodology relates to current methodologies within the field and, where our findings may fit into the picture.

2.5.1 Brute force & Dictionary attacks

Both of these approaches are attacks which applies different keys to encrypted data. Within cryptography, the term *brute force* means that each possible key is tried [24]. If it for example is know that the key consist of six numbers 0-9[1] then, during a brute force attack an attacker would try each combination: 000000, 000001, 000002, ..., 999998, 999999.

A brute force attack is, by definition, guaranteed to find the correct key, however it may require significant time and computing power to do so. In the above example, there are only 10^6 possible passwords to try so a brute force attack can be performed without much effort on modern hardware. However, in a case where every printable ascii character is allowed and the password can be of arbitrary length, such as in the case for most passwords in e.g., web applications, a brute force attack may take billions of years to complete.

Similarly to the brute force attack, a dictionary attack tries to guess the correct key by applying multiple different keys to the cipher text. However, a search heuristic in the form of a dictionary is used, i.e., only keys that exists in the dictionary is applied [25]. The benefit of this is that the attacker does not need to check nearly as many keys as in the brute force case, on the downside however, it is not guaranteed to find the correct key. If the key does not exist in the dictionary, the attack will be unsuccessful.

A dictionary may be as simple as a list of English words however, the dictionaries used in practice usually contain a large variety of real passwords that were exposed in password leaks. Modern dictionary tools can also easily perform rule based expansion of these lists to find variations of passwords. For example replacing letters, such as **A** and **O** with the numbers 4 and 0 to catch the case where the password "p4ssw0rd" has been used instead of "password". Similar tricks applied by users, such as appending or prepending years or numbers to a password can also quite easily be countered by such rules.

2.5.2 Frequency analysis

This kind of attack is one of the oldest attacks against a cryptographic system and was first used against the Caesar cipher. A Caesar cipher works by shifting the alphabet *n* places [26] and then replacing each letter of the message with the n:th

[1]Which happens to be the default setup of the screen lock password on iOS year 2018

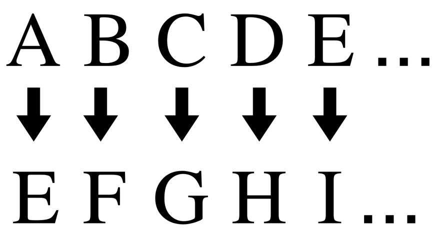


FIGURE 2.4: Shows how the letters of the alphabet are shifted in a Caesar cipher using a shift of four.

following letter. Figure 2.4 shows a Caesar cipher where the alphabet is shifted by n = 4, meaning that each letter **A** in the plain text is replaced by **E** in the new message, each letter **B** with the letter **F**, etc. Caesar ciphers are quite simple and thus it is possible that a brute force attack would have been more efficient than applying frequency analysis. There are however other ciphers that are also vulnerable to this kind of attack where the brute force approach is impractical.

This process will yield a cipher text which, at a glance, looks like gibberish. A fundamental flaw of the Caesar cipher, however, is what makes the frequency analysis attack possible. Each letter of the alphabet is always substituted for the same letter for the same key (in this case n = 4) which fails to hide some properties of the text, namely the distribution of letters. In the English language, some letters, e.g., **E** is used more frequently than others. An attacker could thus count the occurrences of each letter in the encrypted message and replace the most frequent letter with the most frequent letter in English (**E**) and so on where the probably of being correct increases with the length of the message.

2.5.3 Chosen Plaintext Attack

Chosen Plaintext Attack (CPA), or something that closely resembles a CPA is what we use in this paper as it is highly compatible with the supervised learning method described in previous sections.

In the setup of a CPA [18, p. 81] the attacker A has the ability to ask an encryption black box $E_k(\cdot)$ to encrypt message m to produce cipher text c, i.e., $E_k(m) = c$. Letting E_k produce numerous different cipher texts from messages that A can choose freely, may eventually provide A with enough information to deduce the key k and thus be able to read every other cipher text that was outputted by $E_k(\cdot)$. As an example of how a CPA might look like, consider the following:

Suppose that we have an 8 bit block of data $m_c = 11010111$ that we, for some reason, know has been encrypted using the XOR function and we want to decrypt it. If we do not have access to the key, this is impossible as there is no way of knowing which of the bits were flipped and which were not. But if we somehow could gain access to the encryption function, i.e., the XOR with the same key as were used to encrypt m_c, and we could provide it with plain text of our own that it would encrypt. Then, in such a scenario, we could ask it to encrypt the message $m_{pa} = 10000001$ to get the output $m_{ca} = 00001000$.

Now, if we simply XOR m_{pa} with m_{ca} we get the key that were used: $k = 10001001$. And when we have the key we can easily retrieve the plain text version of m_c by XOR between k and m_c resulting in the plain text: 01011110 and we have successfully decrypted the message. What more is that we now know the key and we will thus be able to decrypt every message that were ever encrypted with this function.

A Chosen plaintext attack is somewhat similar to another attack called *Known plaintext attack* (KPA). They differ however on a very important point which is that in a known plaintext attack, the attacker does not get to pick their own input for the known input/output pairs which makes CPA a much more powerful attack and they should not be confused.

2.5.4 Ciphertext-only attack

As suggested by the name, a ciphertext-only attack (COA) is the opposite of chosen plaintext attack. In this attack scenario, the attacker only have access to cipher text [27] and not to the plain text that produces the given cipher text. Furthermore, in this scenario, the attack is successful if the plain text can be recreated and although it, in some cases, may be possible to retrieve the key, this is not the main goal as it is in the CPA case.

For this attack to be successful, the attacker must have access to multiple cipher text samples encrypted with the same algorithm and key and it requires some information of the underlying plain text. Information such as the language of the plain text or the format and a key factor in this scenario is that there are repeated information in the data. For example, http headers contains much the same information for each request and it has a known format.

2.5.5 Differential Cryptanalysis

Differential Cryptanalysis is described by Biham and Shamir [28] as they perform it on DES. It can be said to be similar to a CPA or a KPA as it depends on the attackers ability to know the corresponding cipher text to a specific plain text. At its core, differential cryptanalysis is the practice of comparing pairs of plain text with the corresponding pairs of cipher text and compare the difference in the resulting cipher text when the plaintext is changed. By doing this it is possible to assign probabilities to different keys and make an educated guess about what key were used.

At a glance, this might seem like a weaker attack than CPA as we saw in the example in section 2.5.3 that it was possible to conclusively determine what key were used while in the differential cryptanalysis case, we have to make do with probabilities. This is however not the case as most cryptos are more complicated than a simple XOR meaning that a CPA in a lot of cases is insufficient for determine the key and in such cases, differential cryptanalysis is a powerful alternative.

2.6 Neural Cryptanalysis

In lack of a well established term we will settle for this one to refer to the practice of applying various ANN approaches to cryptanalysis problems. In this section we will cover some of the, limited, previous works that is of relevance to this study.

If we expand our view and look at research that uses general machine learning techniques to perform cryptanalysis there are quite a few where Prajapat & Thakur [29] have done great work in broadly covering them. A majority of which did not use ANNs and formulated the task as a classification problem of determine which cipher that were used. Maghrebi et. al [30] uses a deep neural network to recover the key from a cipher, similarly to what we have done. However in their paper they used a side channel attack rather than attacking the plain text-cipher text pair directly.

The research that most closely resembles the work in this article is that of Alani [2] and Greydanus [31] who both use neural networks and known plain text analysis to attack ciphers. Alani, in their paper, use randomized binary vectors and a cascade forward network to attack DES and Triple DES and reaches good results. Although similar, our approach is slightly more realistic as it uses real English text that could be read and understood by a human.

The approach used by Greydanus differs from that of Alani as it uses regular text samples when training the network. To circumvent the representation problem, Greydanus use one hot encoding of the word vectors when showing them to the network. This approach were also quite well fitted to the question discussed in the paper as it examines ciphers that works by substituting letters of the message to other letters of the alphabet. Vigenère and Enigma, where the substitution in Vigenère is done by hand and in Enigma it is done mechanically. Enigma is the cipher that the Germans used during the second world war.

Chapter 3

Method

The first few sections in this chapter will outline the general procedure of a machine learning project as this is the method used in this project. The later sections relates more directly to the study performed and lays out the details about what were done and how it work towards answering the research questions.

3.1 Hyperparameters

Within the field of machine learning, *hyperparameters* is a term that is commonly used to refer to the parameters to the model. Parameters that are set by the user. This is to discriminate them from the parameters that are learned by the model itself which are simply referred to as *parameters*.

A lot of the complexity when working with ANNs is due to the high number of hyperparameters and the fact that small changes to them might have a large impact on the result. Examples of hyperparameters are: number and size of layers, layer type, learning rate, number of epochs (one complete iteration of the dataset), activation function, loss function etc.

3.2 Model evaluation

Although performance of a network could possibly be measured in many ways, common practice is to use two metrics: *loss* and *accuracy*. There exists multiple choices for both of these metrics and the choice will greatly impact the results when training the model. During training, the algorithm tries to minimize the value of the loss function by adjusting the weights. In the simplest case, the loss function is the difference between the expected and actual output. Some other, more complex loss functions are for example: Mean squared error (MSE), L1, L2, binary and categorical cross entropy. Categorical cross entropy is the one that is most relevant to this study and is defined as:

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C}1_{c\in C_c} log p_{model}[y_i \in C_c] \tag{3.1}$$

where N is the number of observations, C the number of classes. $p_{model}[y_i \in C_c]$ is the probability computed by the model that observation i belongs to category c.

Although loss is the metrics that is minimized during training, accuracy is the metric that most people are interested in, as it represents the practical capabilities of the network. However, getting the accuracy metric right is not as straight forward as it may seem at first sight.

A common training method, that is also used in this project, is called *supervised learning* and it consists of two phases: training and evaluation. During the training phase, the network is shown samples of the data and makes a prediction, it is then told the correct answer and the weights are updated to decrease loss (usually against the gradient of the loss function). During the test phase, the networks performance is evaluated and it is during this phase the true value for accuracy is acquired. During the training phase, the network sees the same data samples again and again while in the test phase, the network is exposed to data it have never seen before. Thus the accuracy in the test phase becomes a measure of how well the network generalize the training to new data. No weight correction is applied during this phase.

If the training were not done this way, one might end up in a situation where the network became really good at classifying the training data but did not generalize to new data which, in turn would defeat the purpose of the network. This scenario is known as *overfitting*.

Overfitting can happen even if the above mentioned methodology is applied; if the data is too small and shown to the network too many times. One method to circumvent this [10] is *dropout* which is a process in which random neurons in the network is dropped for a small amount of iterations. When the neurons are dropped, it means that they do not contribute to the activation during this period and does not get its weights updated. Neurons to be dropped are randomly chosen and after they are restored, another few neurons are dropped at random. The amount of neurons to be dropped in each step is another hyper parameter to the model.

3.3 Data representation

In general, a neural network can only operate on numeric input. Since data may be of any format (text, categorical, etc.), there is often a need to encode the data in a numeric way so that the network can understand it. One such encoding is called *one hot encoding* and it is used in machine learning to represent categorical input or output. It works by introducing vectors of the same length as number of categories in the data. Each entry in the vector is zero except for one entry which is one. The first category is then represented as a binary vector with a 1 at index 1 and then zeros. The second category as the vector with a 1 in the second index etc. One alternative to this method would be to represent each class as an integer in range 1–*number of categories*. However, as the logistic function ranges between (0,1) the one hot encoding is preferable. One hot encoding is an encoding that is used in multiple of the experiments in section 4.

3.4 Our approach

As mentioned in the introduction to this chapter, our approach follows that of a traditional machine learning project more or less to the point. More specifically: to answer the research questions, we design a set of experiments (see chapter 4) that follows the proposed structure from above. Each of these experiments is intended to address one, or part of one, of the research questions and this is done by providing it with data and see how it behaves.

A CPA shares many similarities between supervised learning as in both cases, both the input and output is known and the goal is to find the mapping function from one to the other. CPA is an attack against a crypto system that is used in practice and by basing our trails on this type of attack, we can make sure that any result can be translated into real world scenarios. In the setup of our experiments we encrypt the original data and provides it as input to the network and the original data is provided as target data for the network.

Our main research question is concerned with whether a neural network can learn anything about the relation between the cipher text and plain text and to determine this we use validation accuracy to quantify how much the network have learned. If the validation accuracy reaches above that would be expected from random guessing, the network must have been able to learn something about the mapping between input and target data. This means that it is important to remember that although a higher validation accuracy in general is better than a lower one, it must be adjusted for the number of classes in the task. In conclusion, if the validation accuracy is increased above this limit for any experiment we have positively answered our research question.

3.5 Tools

Parts of the study were implemented in MATLAB 2018b. However, the majority of the experiments were implemented in Python 3 using either Tensorflow 1.10 directly or via Keras [32] which is a high level API on top of Tensorflow. The preparation of data were done with small, handwritten Python scripts and the analysis of the data, including statistical analysis and graph generation were done using Python in a jupyter notebook[1].

As for hardware, there were three levels of hardware available, where good performing models were escalated to the next level. The basic level where all experiments were run initially is the computer that were used to develop the experiments, which is a MacBook Pro Late 2013 with a 2,8 GHz Intel Core i7 processor and 16GB RAM, using macOS as operating system. For the second step a machine provided by Umeå University were used which has four 3.7GHz Intel Xeon CPU Processors, 39.9GB RAM and Debian as operating system. The third step was an Amazon web services (AWS) instance, which had slightly different configurations between each experiment. Most notably the difference between the AWS machines and the others is the utilization of a Tesla K80 GPU. The operating system on these machines were Ubuntu.

[1] `https://jupyter.org/`

3.6 DES Implementation

Neither MATLAB nor Python comes included with a default implementation of the DES algorithm. For this reason we have been forced to look for stand alone implementations, something that could potentially be a source of error. In order to remove this source of error, we went to great lengths to verify that the implementations were correct according to the standard [23].

For MATLAB, we tried two different implementations[2,3] and for Python we used one third party implementation[4]. All of these implementations were written by different people and they were checked against each other in order to make sure that they yielded the same output for the same input.

Additionally, we wrote our own reference implementation (see appendix A) of DES according to the standard and verified the output from the third party implementations against our own. Lastly we verified everything using an online tool that perform DES encryption[5]. As all of the implementations yielded the exact same output for each input, we can be fairly certain that the implementations were correct according to the standard.

[2]https://www.mathworks.com/matlabcentral/fileexchange/53768-des-str-key-mode
[3]https://www.mathworks.com/matlabcentral/fileexchange/37847-data-encryption-standard-des
[4]https://github.com/twhiteman/pyDes
[5]http://extranet.cryptomathic.com/descalc/index

Chapter 4

Experiments

In this chapter we provide the results of the study. As this study has an exploratory nature where multiple different experiments were implemented, this chapter present the setup, results and a shorter discussion for each experiment. This is done in order to, in a structured way, present how we went about answering the sub questions of this research project.

The first experiment, experiment 0, presents the result of trying to reproduce previous studies, experiment 1 tries on a novel approach to representing data to the network, experiment 2 and 3 uses an LSTM network and an MLP to try to decrypt encrypted bits. Experiment 4 & 5 uses a CNN to perform classification on encrypted images and so does experiment 6 but this experiment investigates whether the classification performance is effect by what key were used to encrypt the data.

4.1 Experiment 0

As a stepping stone into this project, we began by trying to reproduce those studies from within the same field that most closely resembles what we are doing in this study, Alani [2] and Greydanus [31].

Beginning with Alani, we followed the instructions from the paper and used a cascade forward neural network. An implementation of that kind of network were easily available in MATLAB, which is the main reason we chose that. MATLAB was also the program of choice in the original article. The versions differ as Alani uses the 2008 version and we use version 2018b and the deep learning toolbox. In the paper, Alani provides the result of multiple different network layouts but in our replication attempts we choose to only apply the layout that provided the lowest MSE; which was the network: 128-256-256-128, i.e., a network with four layers consisting of 128, 256, etc. neurons respectively. For other parameters such as learning function, error function, maximum epochs, and other stopping conditions we followed the example of Alani.

Using this configuration our network was not able to achieve a MSE lower than 0.22, a result that can not be said to be in parity with the results presented in the original study. When that did not work we also tried to recreate the results by implementing the experiment in keras with all the same setup except that we had to use a regular MLP instead of a skip forward network. Switching the implementation environment to python also meant that we had a different implementation of the DES encryption algorithm. However, this did not yield any different result than when implementing it in MATLAB.

It is not entirely clear why this is the case as much effort were put into the recreation attempts. One possible source of error is the implementation of DES used in the original study and the replication attempts. MATLAB does not include a native implementation of the algorithm and Alani does not mention which implementation that were used. Therefore it has not been possible to verify that the DES implementation used in the original paper is correctly implemented.

Replication of Greydanus were done by reimplementing the original model in Keras, which were done for practical reasons as we have more experience of working with Keras than Tensorflow. The task for the model is to reproduce plain text messages from cipher text that had been encrypted using the Vigenere cipher. The model uses the same hyper parameters as the original study, i.e., the network consists of a single LSTM-cell with 512 hidden units. It uses softmax as activation function, adam as optimizer and categorical crossentropy as loss function. Weight initialization is done using Xavier initialization [7], which is referred to as `glorot_normal` in Keras. Even the data generation followed that of Greydanus and were randomly generated on the fly. Loss reached 1.1921×10^{-7} and accuracy reached 100%.

These results confirms the results presented by Greydanus in the original study as they also managed to reach 100% for deciphering the Vigenere cipher.

4.2 Experiment 1

Introduction: Operations in the cryptographic algorithms are described for bit vectors. Bit level granularity is not directly available to the programmer on a software level, forcing the implementations of the cryptographic algorithms to operate on bytes. Introducing a translation problem where the data has to be translated to a byte array, divided into blocks of size 8, and then sent through the encryption algorithm as a block of bytes, although they are actually supposed to be individual bits. The purpose of this experiment is to address our first sub question *How to repersent encrypted data to the network?*

Method: Because a neural network can only operate on numeric input, this experiment explores a way of numerically encoding the data to the network. In this setup the byte array that represents the input and output blocks were interpreted as an integer number. As 8 bytes (64 bits) can represent quite large numbers ($2^{63} - 1$ or 9,223,372,036,854,775,807 for a signed integer), these numbers could not be fed directly into the network as that most certainly would have caused paralysis in the network. Therefore, the numbers were linearly transformed into the range (0,1) resulting in numbers with quite a lot of decimals.

In this experiment we trained two different multilayered perceptron models both of which had a single hidden layer which in turn had 1024 and 2048 hidden units respectively. Two different LSTM networks were also used in this experiment which, similarly to the MLP models used a single hidden layer. In the first model, this hidden layer consisted of 1024 hidden units for the hidden layer and the second model used 2048 hidden units for the hidden layer. For all of the network configurations, both sigmoid and softmax were used as activation function throughout different runs. All of the networks were trained on the first hardware level for the experiments, i.e., the Macbook and were allowed to train for 1000 epochs.

Result: Although multiple different configurations were attempted none of them managed to reach a validation accuracy above zero according to the log output of the monitoring code, although small values for validation accuracy were observed during training.

Discussion: It should be noted that the final validation accuracy of zero most likely is the result of rounding within the software computing the value and not that zero is the true value of the validation accuracy. Possible explanations to this result could be that the numbers require high decimal precision to be represented correctly, making it vulnerable to rounding errors when doing the data transformations before showing it to the network. Because the output space in this particular experiment is another decimal number, it is infinitely large. Although this is not automatically a bad thing, the same is true for other experiments in this report as well, it could explain how the validation accuracy could be so close to zero. Another possible explanation is that any patterns that may be present within the data is lost to the network when converted to a single input.

4.3 Experiment 2

Introduction: The second experiment uses a format that closely resembles that of Alani [2]. We used source text that were split into blocks of 8 bytes and encrypted. Each block were then converted into a binary vector corresponding to the 64 bits that makes up the 8 bytes. The encrypted block was used as input data to the network and the plain text block as target values. This emulates the scenario that the network are operating directly on the bit vectors described by the cryptographic algorithms while in practice each of the entries in this new vector corresponds to 8 bits. This question addresses sub question two (*Can a long short-term memory network recreate plain text that has been encrypted with DES?*), it investigates whether a LSTM network can recreate plain text from cipher text and to do that, a new way of representing data to the network is investigated, addressing the first sub question (*How to represent encrypted data to the network?*) again.

Method: This experiment were run using two different network configurations, both the multilayered perceptron and an LSTM network. For the multilayered perceptron, the same configuration as when trying to replicate Alani were used, i.e., four layers with 128, 256, 256, and 128 hidden units respectively. For the LSTM network we tried multiple different configurations, each of which used one hidden layer. The first one with 512 hidden units and the second one with 1024 hidden units. Loss function was *binary cross entropy* and *RMSprop* were chosen as optimizer. Training were allowed for 1000 epochs. Two types of data were used in this experiment, both randomly generated data as well as data generated from the complete works of Shakespeare.

Result: When using randomly generated data, the network only produced a validation accuracy of 50%, which is the result that is to be expected to achieve when applying random guessing. This was the case for both network types and all different configurations. However, when using the classical literature, both network types reached a validation accuracy of just above 73%.

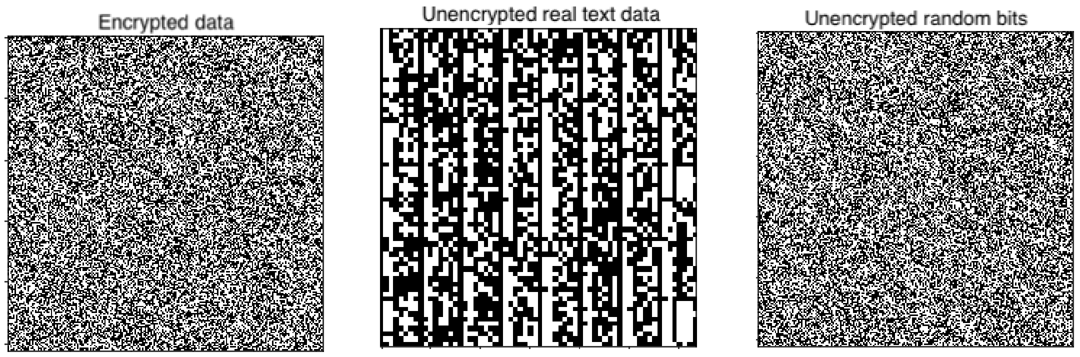


FIGURE 4.1: Visual comparison between random and structured training data to the network.

Discussion: The result might find its explanation by looking at figure 4.1 where we can see that the real text data is not completely random, compared to the randomly generated data which of course is random. Reaching a validation accuracy that is above 50% for the literature data implies that the network has been able to exploit the patterns within the data to some extent or, in other words, these results can be interpreted as the network has discovered frequency analysis and applied that attack to the data. Although these results are interesting, the kind of data representation used in this experiment makes them less practically applicable. As the letters in the data is represented as its bit correspondence, the network should have needed a validation accuracy much close to 100% to be of any use. This is of course because if the network gets even one out of the eight bits wrong, the interpretation is a completely different letter than the original one. In addition to this, the error may occur anywhere and everywhere within the 8 bit block and since there is no way to know about this error, we are not able to take any correcting measures.

4.4 Experiment 3

Introduction: This experiment aims to address only the second sub question: *Can a long short-term memory network recreate plain text that has been encrypted with DES?*. If the validation accuracy can be increased, it indicates that the network is able to recreate some of the plain text.

Method: For this experiment we modified the source code [1] of Greydanus to operate on data encrypted using the DES algorithm. All hyper parameters were kept the same as in the original experiment except that we used 2048 hidden units compared to the much smaller network of 512 hidden units used in the original study. We choose 2048 for our number of hidden units as that was the largest number that were able to run on the hardware without causing an out of memory exception. Because DES is a much more complicated cipher than enigma, it seems reasonable that a bigger model is needed to learn the function.

Result: In the original paper, about 50k iterations were needed before the loss started to go down and validation accuracy started to go up. After about 100k iterations the validation accuracy is starting to level off and not increase as drastically as between

[1]https://github.com/greydanus/crypto-rnn

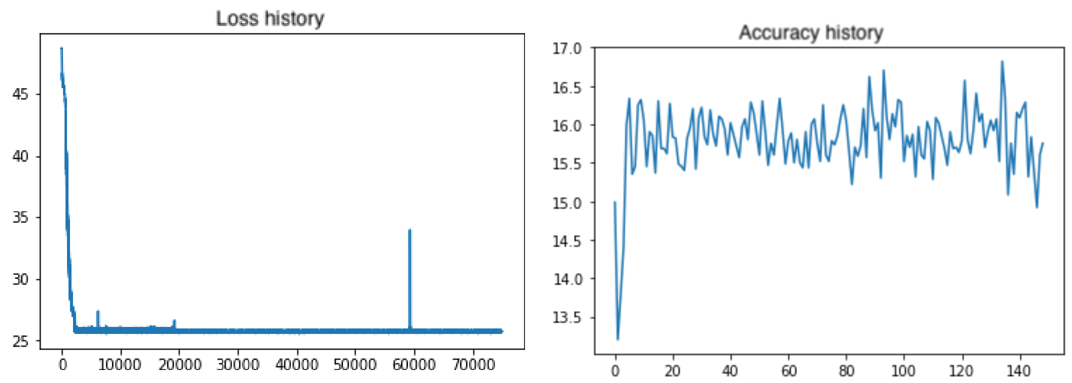


FIGURE 4.2: Loss history and accuracy history for the LSTM-network that tried to decrypt encrypted text data.

50-100k. Because of this we also allowed our model to run for 100k iterations. Depicted in figure 4.2 is a graph of the loss history and validation accuracy history respectively. As can bee seen in the loss history graph, the loss initially decreases rapidly until it reaches around 25 and then stagnates aside from some occasional outliers. The validation accuracy is measured only every 1000th step, hence the smaller number of data points. Visual inspection of the graph implies that the validation accuracy is much more variant than the loss, however, almost all values are within the rather narrow interval 15–16% thus forcing us to consider it as quite stable.

Discussion: To some extent, these values are not surprising. Especially the loss graph looks quite similar to the loss graph provided in the original study, which also did a huge drop in the beginning followed by a long time of no progress. Unfortunately our model did not start to converge despite the significant training effort. Possible reasons for this could be that it would need an even larger network than was possible for us to train due to the complexity of DES.

4.5 Experiment 4

Introduction: MNIST is the name of a classical dataset that contains bitmap images of hand written digits. All images are 28×28 pixels in size where each pixel is represented by an integer number in range 0–255. In this experiment we use the MNIST images as input to the network instead of text. Each image were encrypted using DES and converted to an integer value using the integer function in python. This experiment addresses, our first sub question (*How to represent encrypted data to the network?*) by introducing a new way of representing data to the network but it also investigates the third question (*Can a convolutional neural network predict the class of the underlying data by looking at its encrypted correspondence?*) quite directly as an increase of validation accuracy in this setup would mean that the answer to the question is yes.

Method: The rest of the setup follows that of a traditional image recognition task where the network were shown the image data, encrypted in this case, and predicts a class label for the data. In case of the MNIST data there are 10 classes where each one corresponds to the digit depicted in the image.

Layer #	Layer type	Hidden units	Dropout	Activation
1	Convolutional 2D	64	-	ReLu
2	Convolutional 2D	128	-	ReLu
3	MaxPooling 2D	-	-	-
4	Dropout	-	0.25	-
5	Flatten	-	-	-
6	Dense	256	-	ReLu
7	Dropout	-	0.5	-
6	Dense	10	-	Softmax

TABLE 4.1: Network architecture during experiment 4

For comparison, the same network layout were trained on the same classification task for both encrypted and unencrypted data. This layout consisted of a convolutional neural network whose architecture is illustrated in table 4.1. Not present in the table but still relevant is that the convolutional layers (layer 1–3) use a kernel/pool size of 3×3.

Result: For the network that trained on regular images the highest validation accuracy achieved were 99,3% after about 100 epochs. For the network that trained on the encrypted data, however, the validation accuracy got stuck at 10,22%.

Discussion: An accuracy of 10% corresponds to random guesses as there were 10 different classes for the network to guess from. This indicates that the network were unable to find and exploit any patterns within the data.

4.6 Experiment 5

Introduction: Experiment 5 is almost identical to experiment 4 in its setup, both in terms of data, network architecture and hyperparameters. What distinguishes this experiment from experiment 4 is that in this experiment, we change the mode of operation from CBC (which has been used in the previous experiments) to ECB mode. There are a few motivations behind the use of ECB mode instead of the stronger CBC mode, firstly: completion. We are interesting in finding any indication that a neural network could be useful in cryptanalysis so we want to exhaust all possibilities to reach good results in order to find something to dig deeper into. Secondly: ECB is the mode of operation that were used by Alani to reach good results which might be an indicator that it is possible to reach better results under this mode of operation. The third sub question (*How to represent encrypted data to the network?*) is the main question being investigated in this experiment but indirectly, the first sub question (*Can a convolutional neural network predict the class of the underlying data by looking at its encrypted correspondence?*) is also addressed.

Method: Everything was identical to experiment 4 except that the data were encrypted using ECB mode instead of CBC mode. It should be noted that no padding were used, as the length of the input data in the MNIST dataset is $28 \times 28 = 784$ which is a multiple of eight which is our byte block size.

Result: When the mode of operation were kept at CBC the validation accuracy of the network remained at 10%, i.e., the network failed to learn anything from the data.

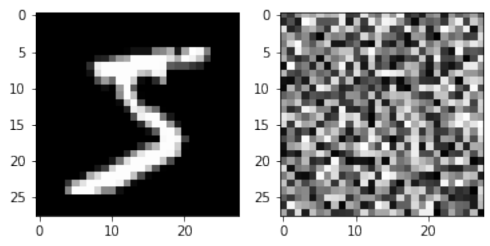


FIGURE 4.3: Unencrypted and encrypted image when using DES in CBC mode.

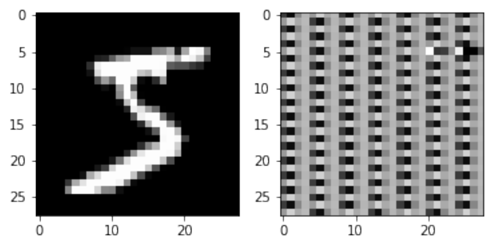


FIGURE 4.4: Unencrypted and encrypted image when using DES in ECB mode.

However, when the mode were switched to ECB, the validation accuracy achieved by the network reached 42% after about 120 epochs.

Discussion: ECB mode is known for being weaker than CBC, which can be seen while comparing figure 4.3 and 4.4. When looking at the figures it is clear that the data encrypted with CBC mode is much more random than the same data encrypted with ECB mode. If one looks closely in the upper right corner of the encrypted image in figure 4.4 there are some anomalies, similar anomalies were present for all of the images in the dataset. It is likely that it is from these anomalies that the network were able to deduce enough information to perform better than random guessing.

4.7 Experiment 6

Introduction: This experiment uses the same setup as the previous two experiments when it comes to data and design choices for the network. However, in this experiment we wanted to more closely investigate a phenomenon of which we saw glimpses when performing the other experiments, namely, that the performance on the classification task for the network yields different results when different keys are applied. In this experiment, none of the sub questions are addressed, instead the main question is addressed directly, should there be a difference in performance for the different keys, it would mean that information about the key could be derived which in turn would result in decryption, or full recreation, of the encrypted data.

Method: As previously mentioned, nothing in this setup was changed from the previous experiments and thus, for details, please consult section 4.5. The experiment were conducted by using two different keys to encrypt the MNIST dataset each of which, the network were allowed to train for 100 epochs. Further more, in order to increase the reliability of the experiment, the network were trained 10 times on each

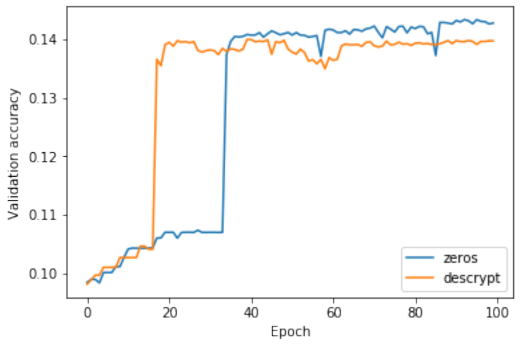


FIGURE 4.5: The average of validation accuracies over the ten runs using the different keys.

of the keys, resetting the weights in between each run. The numbers 100 epochs and 10 repetitions is due to resource constraints. Running through the 2000 epochs (100 epochs during 10 cycles for two different keys) on an AWS instance with 4 Tesla V1000 GPUs still took about 55h and longer time could not be spent on the experiment. One of the keys consists of all zeros and the other consists of the 64 bits that makes up the string "descrypt". The zeros key were chosen to try to induce as little randomization of the data as possible while the descrypt string were chosen as a possible incarnation of what a key used in the real world could look like.

Result: The different runs for each key were averaged into a single dataset consisting of 100 points which is the validation accuracy for that epoch. A plot of these datasets can be found in figure 4.5 the blue line represents the validation accuracy on the data that were encrypted using an all zeros key while the orange line is the "descrypt" bit key. As can be seen in the figure, the curves have more or less the same shape except for one notable difference which is that the orange graph on average converges faster than the blue curve.

To determine whether there were a significant difference between the datasets or not, a two sample t-test were performed which showed that the validation accuracy of descrypt were higher with $p = 0.1$, i.e., the network achieves, with 90% probability, higher validation accuracy when trained on data that was encrypted using the key descrypt than the all zeros key.

Discussion: When looking at the values of the y-axis in the graph 4.5 we see that the range is only between 0–14% and not 42% as in the previous experiment; this is because not all of the 10 runs converged to 42%. In this experiment we have observed a difference in the networks performance when trying to classify encrypted images, depending on what key were used. Our significance test shows that the validation accuracy, with 90% accuracy, can be expected to vary depending on the key choice. 90% confidence is somewhat lower than the 95% that is perhaps the most common confidence level, however, within a field (cryptanalysis) that relies heavily on heuristics, it can be considered reasonably high.

Knowing that the network behaves differently depending on the key used to encrypt the data may be used to derive information about what key were used. As a final note, it should be pointed out that in this experiment, we used the special key consisting of all zeros. This key is special as XOR between a block B and a block of zeros will result in B and thus, it is not entirely certain that the difference will exist or be this large for arbitrary keys.

Chapter 5

Discussion

In this chapter we will discuss the results of the experiments in general and how these results could be understood, in section 5.1 we will try to formulate some conclusions from the results and possible implications and in section 5.2 we will discuss where to go next and where to continue the research.

But first of all we would like to remind the reader of the research questions asked in the beginning of the report. Remember that there were three sub questions to the main research question:

- How to represent encrypted data to the network?
- Can a long short-term memory network recreate plain text that has been encrypted with DES?
- Can a convolutional neural network predict the class of the underlying data by looking at its encrypted correspondence?

Starting with the first question, from our findings we have some suggestions on how to represent encrypted data to a network. As noted in the experiments section, bytes can not be presented directly to the network and thus a direct representation is out of question. A number of methods for representing data to the network has been explored, some of them yielded better results than other. Interpreting the bytes as a floating point numbers did not work very well, which perhaps were to be expected, and neither did representing bytes as their binary decomposition in a list of ones and zeros. One reasonable explanation for why the binary decomposition did not work is because of the size of the output space, and the fact that there were only one sample input per target class.

Using a higher level of representation is probably a better approach; in our experiments we tried one-hot-encoding to represent the text input to the network which did not yield the best results either which is probably due to some similar reason as the one just mentioned. When we formulate the data as a pure classification problem, in the case with the encrypted MNIST images, the network reached higher validation accuracy than random guessing, indicating that it managed to learn something.

For answering the first question this means that the best way to represent encrypted data to the network is to find some way to translate the problem into some higher order of abstraction to turn it into a more traditional classification problem. Of course, this limits the practical usage of these findings as it would require quite a lot of knowledge about the kind of plaintext.

The second question is addressed in experiments 2 and 3 where we made different attempts at getting an LSTM to recreate plain text, also with various results. In experiment 3, the results were not convincing to say the least, probably due to the fact that the algorithms are designed with contradicting goals. An LSTM works by finding long term relations within the data, such dependencies that one may find in written prose, however, a good encryption algorithm should be designed with the sole purpose of removing such long term dependencies. However, experiment 2, when using classical literature, shows that the network managed to learn at least something as it reached a validation accuracy of 75%. When looking at the data there was a small imbalance between ones and zeros, although not enough to explain all the 75%. If we look at image 4.1, we can see that the target data contains patterns and it is likely that the network actually learned the structure of the target data rather than the mapping between input and output.

Thus, for the second question, we have not been able to recreate plain text from cipher text using an LSTM network. Although our study does not provide enough evidence to completely rule out that possibility, we did not find any real indication that recreating plaintext from cipher text using an LSTM network would be possible.

Lastly, we ask the question about if a convolutional neural network can perform image classification on encrypted data, which is perhaps where we had the most success, as our convolutional network managed to reach a validation accuracy as high as 42% when classifying the images. As was mentioned in the description of experiment 5, random guessing would have resulted in 10% validation accuracy so obviously, the network managed to learn something in this setup. While this is very interesting, it should once again be noted that this increase in accuracy was only achieved when the data were encrypted with ECB mode, when CBC mode were used, the network showed no increase in performance compared to random guessing. Nonetheless, an encryption algorithm that allows an attacker to correctly guess the data in more than 40% of the cases would not be considered particularly secure, especially when considering the avalanche effect, that similar but not equal inputs should have completely different outputs. If we then look at the example in figure 4.4 of what the encrypted data look like, it is also easy to imagine that the network had a higher accuracy than a human would have had. It should also be noted that 42%, which is the highest accuracy reached in our experiments, is probably not the highest possible accuracy for this task, the accuracy could probably have been increased with more data, a larger network and by utilizing ensemble techniques such as bagging and boosting.

This means that the answer to the last question is that during the right circumstances, a convolutional neural network is indeed able to perform classification tasks, even if the data is encrypted. Although it requires a set of specific circumstances that makes it less suitable for practical use, it is an interesting theoretical finding that could potentially lay a foundation for future research and open up new areas of cryptanalysis.

Further more, remember from the introduction that the main research questions were:

> Is it possible for Neural Networks to derive information about plain text that has been encrypted with DES by studying the cipher text? If yes, is it possible to derive enough information to make the network recreate the plain text?

In the work presented in this report, there are nothing that indicates that a neural network would be able to recreate plain text from cipher text. However, we can conclude that a neural network can derive information about the plain text just by studying the cipher text, we have learned that if the problem is formulated as a classification problem to the network, it may be able to make correct classifications of the cipher text. Furthermore we have learned that the key that were used to encrypt the data will effect the performance of the network which, in practice, means that the network is able to learn information about the key. Thus it is indeed possible for a network to derive information about the plain text when the data has been encrypted with DES if ECB mode is used.

We would like to spend the rest of this section with a discussion about the findings of previous studies and especially the study previously mentioned [2] in this article. In our work we spent a significant amount of effort reproducing the results of that study which claimed to have been able to recreate the plain text bits from the cipher text bits. Although we followed the precise description provided in the paper and used two different implementations of DES algorithm we did not find results anywhere in line with those presented in the original study. There are at least two reasonable explanations to this, the first one is the ambiguity of the implementation of the DES algorithm in the original study which is unknown and could perhaps be faulty in some way. Assuming however that this is not the case, there is the possibility that the description of the original experiment setup were not precise enough to allow for an exact replication. We reached out to Alani and asked to have a look at the source code of the experiments to be able to perform a better replication attempt but we were not able to get it as it apparently were not available anymore.

5.1 Conclusions & Practical Uses

In short, the conclusions of this paper is that a neural network is still able to perform classification tasks correctly, to some extent, if the data is encrypted. We also conclude that the networks performance when performing classification is affected by the key that were used to encrypt the data. Lastly we conclude that although the network is able to learn things about encrypted data, we have not found anything that would suggest that neural networks could be used to do a downright recreation of the plaintext of encrypted data.

Although a neural network has all these abilities in a controlled lab environment, the practical applications of these findings are more limited. However, there are a few cases in which this could be of use, below follows a selection of possible use cases:

Offline attacks: In the event that an attacker have come across a known set of data that happens to be encrypted, they could apply these techniques in order to extract information from the data. For example, if an attacker had gotten a hold of encrypted emails of a company, they would perhaps be able to train a network to classify sender and receiver of an email and thus map out communication structures of that company.

Verification: As we have seen, the network managed to learn when the data were encrypted using ECB mode and not while doing CBC mode, i.e., if an encryption algorithm is designed and used properly, the network should not be able to learn anything. This insight could then be exploited to use neural networks to detect erroneous usage of cryptographic schemata or as a way to do automatic testing of the security of a new cryptographic algorithms.

Key extraction: Given that the performance of the network is affected by what key was used to encrypt the data, it is possible for an attacker to make key guessing more efficient for example by reducing the size of dictionaries needed for a brute force attack. In the same spirit as the previous point, the attacker could also use the network to detect whether it has succeeded in finding the correct key.

5.2 Future work

This study has been exploring multiple approaches to how neural networks can be used within the field of cryptanalysis to learn information about the plain text. Although a broad range of topics were covered in this paper, they were mostly touched upon and thus, this study should be thought of as a spring board for future research. We would like to highlight the positive results of this study as good candidates for future research, i.e., the fact that ANNs can perform classification tasks on encrypted data. It would be interesting to see if the accuracy can be improved above 42% and whether the ability to perform classification persists for other kinds of data, e.g., text classification. It would also be valuable to determine if an ANN is able to differentiate between different kinds of encrypted data. If a network could be able to correctly classify the cipher text as an encrypted image or encrypted text, it would drastically reduce the search space when the work of the attacker continues.

Another potential for future work is the finding that different keys yield different performance of the networks, for this discovery to have real practical value, further research has to be done in order to map out how differences in results are affected by the changes of the keys. Furthermore it is interesting to notice that although one key caused better network performance than the other, it was not the all zeros key which we expected, instead the more complex key caused significantly better performance, the reason for this is also something that would be of great value to further research.

Bibliography

[1] G. Cybenko
"Approximation by superpositions of a sigmoidal function"
in: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314
ISSN: 1435-568X
DOI: 10.1007/BF02551274
URL: https://doi.org/10.1007/BF02551274.

[2] Mohammed M. Alani
"Neuro-Cryptanalysis of DES and Triple-DES"
in: *Neural Information Processing*
ed. by Tingwen Huang et al.
Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 637–646
ISBN: 978-3-642-34500-5.

[3] Omegapoint
About us
https://omegapoint.se/om-oss-2/
[Online; accessed 4-Sept-2018].

[4] Adam H. Marblestone, Greg Wayne, and Konrad P. Kording
"Toward an Integration of Deep Learning and Neuroscience"
in: *Frontiers in Computational Neuroscience* 10 (2016), p. 94
ISSN: 1662-5188
DOI: 10.3389/fncom.2016.00094
URL: https://www.frontiersin.org/article/10.3389/fncom.2016.00094.

[5] Yoav Freund and Robert E. Schapire
"Large Margin Classification Using the Perceptron Algorithm"
in: *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*
COLT' 98
New York, NY, USA: ACM, 1998, pp. 209–217
ISBN: 1-58113-057-0
DOI: 10.1145/279943.279985
URL: http://doi.acm.org/10.1145/279943.279985.

[6] Stuart Russell and Peter Norvig
Artificial Intelligence: A Modern Approach
3rd
Upper Saddle River, NJ, USA: Prentice Hall Press, 2009
ISBN: 9780136042594.

[7] Xavier Glorot and Yoshua Bengio
"Understanding the difficulty of training deep feedforward neural networks"

in: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*
2010.

[8] Omaima N Ahmad Al-allaf
"Cascade-forward vs. function fitting neural network for improving image quality and learning time in image compression system"
in: *Proceedings of the world congress on engineering*
vol. 2
2012, pp. 4–6.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton
"ImageNet Classification with Deep Convolutional Neural Networks"
in: *Advances in Neural Information Processing Systems 25*
ed. by F. Pereira et al.
Curran Associates, Inc., 2012, pp. 1097–1105
URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[10] Michael A. Nielsen
Neural Networks and Deep Learning
[Online; accessed 30-November-2018]
Determination Press, 2015
URL: `http://neuralnetworksanddeeplearning.com/chap6.html`.

[11] Felix A. Gers and Juergen Schmidhuber
Recurrent Nets That Time and Count
tech. rep.
2000.

[12] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever
"An Empirical Exploration of Recurrent Network Architectures"
in: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*
ICML'15
Lille, France: JMLR.org, 2015, pp. 2342–2350
URL: `http://dl.acm.org/citation.cfm?id=3045118.3045367`.

[13] S. Hochreiter
Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München
1991.

[14] Sepp Hochreiter and Jürgen Schmidhuber
"Long Short-Term Memory"
in: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780
ISSN: 0899-7667
DOI: `10.1162/neco.1997.9.8.1735`
URL: `http://dx.doi.org/10.1162/neco.1997.9.8.1735`.

[15] C. Olah
Understanding LSTM networks
Online; accessed 5-Sept-2018
Aug. 2015
URL: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[16] Klaus Greff et al.
"LSTM: A Search Space Odyssey"
in: *CoRR* abs/1503.04069 (2015)
arXiv: 1503.04069
URL: http://arxiv.org/abs/1503.04069.

[17] Paul C. van Oorschot Alfred J. Menezes and Scott A. Vanstone
Handbook of applied cryptography
Boca Raton London New York Washington, D.C., 1996.

[18] Y Katz J. & Lindell
Introduction to modern cryptography
Boca Raton London New York Washington, D.C.: CRC PRESS, 2008.

[19] Auguste Kerckhoffs
"La cryptographie militaire"
in: *Journal des sciences militaires* IX (Jan. 1883), pp. 5–83.

[20] C. Shannon
"Communication Theory of Secrecy Systems"
in: *Bell System Technical Journal, Vol 28, pp. 656–715* (Oct. 1949).

[21] A. F. Webster and S. E. Tavares
"On the Design of S-Boxes"
in: *Advances in Cryptology — CRYPTO '85 Proceedings*
ed. by Hugh C. Williams
Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 523–534
ISBN: 978-3-540-39799-1.

[22] Henk C. A. Tilborg and Sushil Jajodia
Encyclopedia of Cryptography and Security
2nd
Springer Publishing Company, Incorporated, 2011
ISBN: 9781441959065.

[23] Des
"Data Encryption Standard"
in: *In FIPS PUB 46, Federal Information Processing Standards Publication*
1977, pp. 46–2.

[24] Learn cryptography
Brute Force Attack
[Online; accessed 03-January-2019]
URL: https://learncryptography.com/attack-vectors/brute-force-attack.

[25] Learn cryptography
Dictionary Attack
[Online; accessed 03-January-2019]
URL: https://learncryptography.com/attack-vectors/dictionary-attack.

[26] Damico T. M.
A Brief History of Cryptography
[Online; accessed 03-January-2019]
2009
URL: http://www.inquiriesjournal.com/a?id=1698.

[27] Alex Biryukov and Eyal Kushilevitz
"From Differential Cryptanalysis to Ciphertext-Only Attacks"
in: *Lecture Notes in Computer Science* (Aug. 2001)
DOI: 10.1007/BFb0055721.

[28] Eli Biham and Adi Shamir
"Differential Cryptanalysis of DES-like Cryptosystems"
in: *Advances in Cryptology-CRYPTO' 90*
ed. by Alfred J. Menezes and Scott A. Vanstone
Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 2–21
ISBN: 978-3-540-38424-3.

[29] Shaligram Prajapat and Ramjeevan Singh Thakur
Various Approaches towards Cryptanalysis.

[30] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff
"Breaking Cryptographic Implementations Using Deep Learning Techniques"
in: (Dec. 2016), pp. 3–26.

[31] Sam Greydanus
"Learning the Enigma with Recurrent Neural Networks"
in: *CoRR* abs/1708.07576 (2017)
arXiv: 1708.07576
URL: http://arxiv.org/abs/1708.07576.

[32] François Chollet et al.
Keras
https://keras.io
2015.

Appendix A

Source code

MNIST on encrypted data

```
#!/usr/bin/env python


# Import all the dependencies
import sys
import keras
from keras.layers import *
from keras.models import Sequential
from scipy import stats
from matplotlib import pyplot as plt
import numpy as np
from crypto.pyDes import *
import array

# Declare all the variables
batch_size = 128
num_classes = 10
epochs = 1000
img_rows = 28
img_cols = 28
input_shape = (img_rows, img_cols, 1)

# Load and normalize the data
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
```

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Define the encryption function
def enc(img, mode):
    from sklearn.preprocessing import scale

    if mode == CBC:
        c = des(
            b'descrypt',
            CBC,
            b'\0\0\0\0\0\0\0\0',
            pad=None,
            padmode=PAD_NORMAL
        )
    else:
        c = des(b'\0\0\0\0\0\0\0\0', ECB)

    flat_img = np.reshape(img, (img_rows*img_cols,))
    img_bytes = array.array('B', flat_img).tobytes()
    a = np.array(list(c.encrypt(img_bytes))[0:img_rows*img_cols])
    b = np.interp(a, (a.min(), a.max()), (0, 255))
    return np.reshape(b.astype(float), (img_rows, img_cols))

img = x_train[0]
img_cbc = enc(img, CBC)
img_ecb = enc(img, ECB)

plt.figure()
plt.gray()
_, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=False)
ax1.imshow(np.reshape(img, (img_rows, img_cols)))
ax2.imshow(np.reshape(img_cbc, (img_rows, img_cols)))
ax3.imshow(np.reshape(img_ecb, (img_rows, img_cols)))


x = enc(x_train[0])

# Encrypt all the training data
x_train_short = x_train
x_test_short = x_test

x_train_enc = [enc(x) for x in x_train_short]
x_test_enc = [enc(x) for x in x_test_short]

x_train_enc_expanded = x_train_enc
x_test_enc_expanded = x_test_enc

# Add another dimension to the data
```

```
x_train_enc_expanded = np.array(
    [np.expand_dims(x, axis=2) for x in x_train_enc]
)
x_test_enc_expanded = np.array(
    [np.expand_dims(x, axis=2) for x in x_test_enc]
)

print(x_train_enc_expanded.shape)
print(x_test_enc_expanded.shape)

# One-hot encode the labels
from keras.utils import to_categorical
y_train_oh = to_categorical(y_train)
y_test_oh = to_categorical(y_test)

model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(28,28,1)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train_enc_expanded, y_train_oh,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          shuffle=True,
          validation_data=(x_test_enc_expanded, y_test_oh))
```

Different keys experiment

```
import json

import keras
from keras.layers import *
from keras.models import Sequential
from keras.datasets import mnist
from keras.utils import to_categorical
import numpy as np
```

```
def load_encrypted_dataset(key_name):
    train_file = 'train_' + key_name + '.gz'
    test_file = 'test_' + key_name + '.gz'

    train = np.loadtxt(train_file)
    test = np.loadtxt(test_file)

    train2d = [np.reshape(img.astype(float), (28, 28)) for img in train]
    test2d = [np.reshape(img.astype(float), (28, 28)) for img in test]

    return train2d, test2d


def run_experiment(key_name, number):
    batch_size = 128
    num_classes = 10
    epochs = 100
    img_rows = 28
    img_cols = 28
    input_shape = (img_rows, img_cols, 1)

    x_train_enc, x_test_enc = load_encrypted_dataset(key_name)
    print("Loaded data")

    x_train_enc_expanded = np.array(
        [np.expand_dims(x, axis=2) for x in x_train_enc]
    )

    x_test_enc_expanded = np.array(
        [np.expand_dims(x, axis=2) for x in x_test_enc]
    )

    y_train_oh = to_categorical(y_train)
    y_test_oh = to_categorical(y_test)

    print("Running experiment: %s" % key_name)
    model = Sequential()
    model.add(Conv2D(64, kernel_size=(3, 3),
                     activation='relu',
                     input_shape=input_shape))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))

    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adadelta(),
                  metrics=['accuracy'])
    history = model.fit(x_train_enc_expanded, y_train_oh,
                        batch_size=batch_size,
                        epochs=epochs,
                        verbose=1,
                        shuffle=True,
                        validation_data=(x_test_enc_expanded, y_test_oh))

    output_file = 'results_' + key_name + '_' + str(number) + '.json'
    with open(output_file, 'w') as f:
        json.dump(history.history, f, ensure_ascii=False)


if __name__ == '__main__':

    for i in range(10):
        run_experiment('descrypt', i)
        run_experiment('zeros', i)
```

DES Implementation

```
import warnings
import numpy as np
from bitstring import Bits


def _byte_array_to_bit_list(arr):
    """Converts a byte array into a list of the corresponding bits

    Example:
        b'123' -> [0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1,
        0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1]

    Args:
        arr (bytes): The bytes to convert

    Returns:
        Numpy array of bits
    """
    length = (len(arr) * 8)
    bitstring = Bits(bytes=arr, length=length)
    return np.array([int(i) for i in list(bitstring)])


def _bit_list_to_byte_array(arr):
    """Converts a list of bits into a bytes object
    of the corresponding bytes

    Example:
        [0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1,
        0, 0, 0, 1, 1, 0, 0, 1, 1] -> b'123'

    Args:
        arr (ndarray): List of bits to convert to bytes

    Returns:
        bytes: the corresponding bytes
    """
    bitstring = ''.join(map(str, map(int, arr)))
    return int(bitstring, 2).to_bytes(len(bitstring) // 8, byteorder='big')


def _perm(msg, p):
    """Permutes the bit array using the provided permutation table

    Expects the both lists to be one dimensional

    Args:
        msg (ndarray): The message bits.
        p (ndarray): The permutation table.

    Returns:
        ndarray: A new list with the bits of msg in
        the order prescribed by p

    """

    return np.reshape(np.fromiter(map(lambda x: msg[x-1], p), int), p.shape)


def _xor(arr1, arr2):
    """Computes element wise xor

    Computes element wise xor for each element in the
    lists and returns a new list containing the results.
    Expects the lists to be of the same length and
    only contain logical values (True/False/1/0)

    Example:
        a = [0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0,
        1, 0, 0, 0, 1, 1, 0, 0, 1, 1]
        b = [1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0,
        1, 1, 1, 0, 0, 1, 0, 1, 1, 1]

        _xor(a, b) = _xor(b, a) = [1, 0, 0, 0, 0, 1, 0,
        1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0]

    Args:
        arr1 (ndarray): The first list to xor.
        arr2 (ndarray): The second list to xor.
    Returns:
        ndarray: A new list of the same length as
        arr1 & arr2 containing the result of the xor

    """

    return np.array(list(map(lambda x, y: x ^ y, arr1, arr2)))


def _KS(key):
    """Performs key sampling as described in DES standard

    Accepts a block of 64 bits, performs all necessary
    operations to produce the 16 sub-keys

    Args:
         key (ndarray): The key to sample

    Returns:
         ndarray: An array of shape (16, 48) where row i is the i:th key

    """
    left_shifts = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]
    pc1 = np.array([
        57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,

        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4
    ])
    pc2 = np.array([
        14, 17, 11, 24, 1, 5,
        3, 28, 15, 6, 21, 10,
        23, 19, 12, 4, 26, 8,
        16, 7, 27, 20, 13, 2,
        41, 52, 31, 37, 47, 55,
        30, 40, 51, 45, 33, 48,
        44, 49, 39, 56, 34, 53,
        46, 42, 50, 36, 29, 32
    ])

    C, D = np.split(_perm(key, pc1), 2)

    keys = np.empty(16, object)
    for i in range(16):
        C = np.roll(C, -left_shifts[i])
        D = np.roll(D, -left_shifts[i])
        keys[i] = _perm(np.concatenate((C, D)), pc2)

    return keys


def _S(n, block):
    """Implements the selection function from the DES paper

    Accepts the round and the block (6 bit) for this round
    and computes the value to retrieve
    from the appropriate S—function as described in the standard.
    Returns the 4 bytes as a list of integers 1/0

    Args:
        n (int): Integer that corresponds to the current round.
        block (ndarray): The key bits for this round as a list of 1/0

    Returns:
        ndarray: The appropriate bits for this round
        as a list of integers 1/0

    """
    s = [
        [
            [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
            [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
            [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
            [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]
        ],
        [
            [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
            [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
            [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
            [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]
        ],
        [
            [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
            [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
            [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
            [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]
        ],
        [
            [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
            [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
            [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
            [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]
        ],
        [
            [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
            [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
            [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
            [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]
        ],
        [
            [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
            [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
            [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
            [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]
        ],
        [
            [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
            [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
            [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
            [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]
        ],
        [
            [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
            [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
            [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
            [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]
        ]
    ]

    i = int(str(block[0]) + str(block[5]), 2)
    j = int(''.join(map(str, block[1:5])), 2)

    bin_str = bin(s[n][i][j])[2:]

    return np.array(([0] * (4-len(bin_str))) + list(map(int, bin_str)))


def _E(bits):
    """Implements the E function from the DES paper

    Accepts a block of 32 bits and produces a block of 48 bits

    Args:
        bits (ndarray): The block of 32 bits to expand

    Returns:
        ndarray: The expanded block consisting of 48 bits.

    """
    e = np.array([
        32, 1, 2, 3, 4, 5,
        4, 5, 6, 7, 8, 9,
        8, 9, 10, 11, 12, 13,
        12, 13, 14, 15, 16, 17,
        16, 17, 18, 19, 20, 21,
        20, 21, 22, 23, 24, 25,
        24, 25, 26, 27, 28, 29,
        28, 29, 30, 31, 32, 1
    ])

    return _perm(bits, e)


def _P(L):
    """Implements the P function from the DES paper

    Accepts a block of 32 bits and outputs a block of 32 bits,
    permuted using table P

    Args:
        L (ndarray): The block of 32 bits to permute

    Returns:
        ndarray: The contracted block consisting of 48 bits.

    """
    table = np.array([
        16, 7, 20, 21,
        29, 12, 28, 17,
        1, 15, 23, 26,
        5, 18, 31, 10,
        2, 8, 24, 14,
        32, 27, 3, 9,
        19, 13, 30, 6,
        22, 11, 4, 25
    ])

    new_shape = (32, 1)
    return _perm(np.reshape(L, new_shape), table)


def _f(R, K):
    """The f function from the DES paper

    It accepts a 32 bit block to encrypt and a 48 bit key
    to use for encryption

    Args:
        R (ndarray): The block to encrypt (32 bit).
        K (ndarray): The key to use in encryption (48 bit).

    Returns:
        ndarray: a permuted version of R (32 bit)

    """
    blocks = np.split(_xor(K, _E(R)), 8)
    L = [_S(i, block) for block, i in zip(blocks, range(len(blocks)))]
    return _P(L)
```

```
def _round(key, message):
    """Performs an encryption round of the DES algorithm

    Accepts an entire 64 bit block and performs an
    encryption round on the block

    Args:
        key (ndarray): A 48 bit key to use in this round.
        message (ndarray): A 64 bit message to encrypt.

    Returns:
        ndarray: the encrypted message (64 bit).

    """
    # pylint: disable=unbalanced-tuple-unpacking
    L0, R0 = np.split(message, 2)
    L1 = R0
    R1 = _xor(L0, _f(R0, key))
    return np.concatenate((L1, R1))


def _encrypt_block(key_n, block):
    """Performs the complete encryption of a block using
    the provided keys

    Accepts the 16 key samples to use when performing
    the 16 rounds of encryption of the block.
    This function performs all necessary operations to
    completely encrypt the provided block.

    Args:
        key_n (ndarray): List of shape (16, 48)
            containing the different keys to use for each round
        block (ndarray): 64 bit block to encrypt

    Returns:
        ndarray: The encrypted block (64 bits)

    """
    block = _perm(block, __ip)

    for key in key_n:
        block = _round(key, block)

    L, R = np.split(block, 2)
    swapped = np.concatenate((R, L))
    return _perm(swapped, __ip_inv)


def __encrypt_ecb(key_n, blocks):
    """Encrypts the provided blocks using the provided keys using ECB mode

    Accepts the 16 key samples to use when performing
    the 16 rounds of encryption of the block.

    Args:
        key_n (ndarray): List of shape (16, 48) containing
        the different keys to use for each round
        blocks (ndarray): n x 64 bit array representing
        the blocks to encrypt

    Returns:
        ndarray: The data encrypted with ECB mode

    """
    return [_encrypt_block(key_n, block) for block in blocks]


def __encrypt_cbc(key_n, blocks, iv):
    """Encrypts the provided blocks using the provided
    keys using CBC mode

    Accepts the 16 key samples to use when performing
    the 16 rounds of encryption of the block.

    Args:
        key_n (ndarray): List of shape (16, 48) containing
        the different keys to use for each round
        blocks (ndarray): n x 64 bit array representing
        the blocks to encrypt

    Returns:
        ndarray: The data encrypted with CBC mode

    """
    encrypted_blocks = []
    previous_block = iv

    for block in blocks:
        block = _xor(block, previous_block)

        previous_block = _encrypt_block(key_n, block)
        encrypted_blocks.append(previous_block)

    return encrypted_blocks


def __decrypt_cbc(key_n, blocks, iv):
    """Decrypts data that were previously encrypted using CBC mode

    Accepts the 16 key samples to use when performing
    the 16 rounds of decryption of a block.

    Args:
        key_n (ndarray): List of shape (16, 48) containing
        the different keys to use for each round
        blocks (ndarray): n x 64 bit array representing
        the blocks to decrypt

    Returns:
        ndarray: Decrypted data

    """
    blocks = [iv] + blocks
    decrypted_blocks = []
    i = 1
    while i < len(blocks):
        block = blocks[-i]
        block = _encrypt_block(key_n, block)
        block = _xor(block, blocks[-(i + 1)])

        decrypted_blocks = [block] + decrypted_blocks
        i += 1

    return decrypted_blocks


def _pad(block, n=8):
    """Pads the block to a multiple of n

    Accepts an arbitrary sized block and pads it to be a multiple of n.
    Padding is done using the PKCS5 method, i.e., the block is padded with
    the same byte as the number of bytes to add.

    Args:
        block (bytes): The block to pad, may be arbitrary large.
        n (int): The resulting bytes object will be padded to a
        multiple of this number

    Returns:
        bytes: a bytes object created from the input string

    """

    pad_len = n - (len(block) % n)
    block += bytes([pad_len] * pad_len)
    return block


def __unpad(string):
    """Removes padding from the provided string
    This function is the inverse of _pad such that unpad(pad(m)) = m
    Args:
        string (bytes): The block to unpad, may be arbitrary large.

    Returns:
        bytes: a bytes object created from the input string

    """
    pad_len = string[-1]
    return string[:-pad_len]


def __make_sure_bytes(input_str):
    """Makes sure that the input string is of type bytes

    Tries to convert type to byte and throws
    an error if that is not possible

    Args:
        input_str (bytes): The input string to validate.

    Returns:
        bytes: a bytes object created from the input string

    """
    if not isinstance(input_str, bytes):
        raise TypeError('Argument must be of type string or bytes')


def __enforce_key_length(block):
    if len(block) % 8 != 0:
        raise ValueError('Expected key
        to be exactly 8 bytes, got {}'.format(len(block)))


def __validate_iv(mode, iv):
    """Makes sure that the initialization vector is correct

    Throws an error if the iv does not follow the specification

    Args:
        mode (str): The mode of operation that is used in this encryption.
        iv (bytes): The initialization vector to validate.

    """
    if mode == CBC:
        if not iv:
            raise TypeError('Initialization vector
            must be provided if mode is CBC')

        else:
            __make_sure_bytes(iv)

        if len(iv) != 8:
            raise ValueError('Initialization vector must be 8 bytes long')

    if mode == ECB and iv:
        warnings.warn('Initialization vector is not used in ECB mode. \
        The IV is ignored but this might indicate an error in your program')


def __validate_input(block, key, mode, iv):
    """Validates the input parameters

    Makes sure that the parameter values and combinations
    are valid enough to perform a correct encryption.
    If something is wrong, this function will throw an
    appropriate error with useful error information.

    Args:
        block (bytes): The input string to validate.
        key (bytes): The key to validate.
        mode (str): The mode to validate.
        iv (bytes): The initialization vector to validate

    """
    __make_sure_bytes(block)
    __make_sure_bytes(key)
    __enforce_key_length(key)
    __validate_iv(mode, iv)


__ip = np.array([
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
])

__ip_inv = np.array([
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25
])

CBC = 'CBC'
ECB = 'ECB'


def encrypt(block, key, mode=CBC, iv=None):
    """Encrypts the provided block using the provided key

    Accepts the block to encrypt as well as the key to use.
    The key MUST be exactly 8 bytes long and both the key
    and the block have to be of type bytes. Will not do a
    parity bit check of the key.
    If the size of block is not a multiple of 8, it will
    be padded using the PKCS5 method.

    Args:
        block (bytes): The input string to encrypt.
        key (bytes): The key to use for encryption.
        mode (str): One of CBC or ECB, defaults to CBC.
        iv (bytes): The initialization vector to use
        for CBC mode, should probably not be provided in ECB mode.

    Returns:
        bytes: The block encrypted with the provided key.

    """
    __validate_input(block, key, mode, iv)

    key = _byte_array_to_bit_list(key)
    key_n = _KS(key)

    bits = _byte_array_to_bit_list(_pad(block))
    blocks = np.split(bits, int(len(bits) / 64))

    if mode == CBC:
        encrypted_blocks = __encrypt_cbc(
            key_n,
            blocks,
            _byte_array_to_bit_list(iv)
        )
    else:
        encrypted_blocks = __encrypt_ecb(key_n, blocks)

    return _bit_list_to_byte_array(np.concatenate(encrypted_blocks))


def decrypt(block, key, mode=CBC, iv=None):
    """Decrypts the provided block that were previously
    encrypted with the provided key
    Args:
        block (bytes): The input string to decrypt.
        key (bytes): The key to use for decryption.
        mode (str): One of CBC or ECB, defaults to CBC.
        iv (bytes): The initialization vector used for CBC mode,
        should probably not be provided in ECB mode.
    """
    __validate_input(block, key, mode, iv)

    key = _byte_array_to_bit_list(key)
    key_n = list(reversed(_KS(key)))

    bits = _byte_array_to_bit_list(block)
    blocks = np.split(bits, int(len(bits) / 64))

    if mode == ECB:
        decrypted_blocks = __encrypt_ecb(key_n, blocks)

    else:
        iv = _byte_array_to_bit_list(iv)
        decrypted_blocks = __decrypt_cbc(key_n, blocks, iv)

    decrypted = _bit_list_to_byte_array(np.concatenate(decrypted_blocks))
    return __unpad(decrypted)

# end
```