UNIVERSITY OF TORINO

M.Sc. in Stochastics and Data Science

Final dissertation

Thesis Title

Supervisor: Elena Cordero
Co-supervisor: Stefano Barbero

Candidate: Francesco Moraglio

ACADEMIC YEAR 2019/2020

Summary

Insert here a summary of your thesis

Acknowledgements

You can insert here possible thanks and acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction to Neural Networks

First chapter of this work is dedicated to its main ingredient: Artificial Neural Networks. These are models of computation based loosely on the way in which the brain is believed to work. Since their invention, biologists are interested in using such networks to model biological brains, but most of the impetus comes from their employment in applied sciences: NN's allow us to create machine that can perform "cognitive" tasks.

1.1 Biological Perspective

To start off, an overview on biological neural network is needed, as ANN's in principle attempt to simulate them. An extensive treatment of both principles and applications of neural networks can be found in Graupe (2007), to which I make reference for this chapter.
The biological neural network consists of interconnected nerve cells (called neurons), whose bodies are where most of the neural "computation" takes place. Neural activity passes from one neuron to another through electrical signals which travel from one cell to the following down the neuron's axon, that can be seen as a connection wire. However, the mechanism of signal transmission in not via electrical conduction, but via charge exchange that is transported by diffusion ions. At the terminal end of the axon, a synaptic gap takes place and an electro-chemical process allows communication with the other cell.
Since a given neuron may have many synapses, it can connect to many other cells. Similarly, since there are many dendrites (input connections) per each

neuron, a single nerve cell can receive signals from many other neurons. It is very important to notice that not all of such interconnections are equally weighted: some have a higher priority than others. Also some are inhibitory and some are excitory. These facts are also key feature of artificial neural networks, as I'll discuss in the following sections.

1.2 Principles

The theoretical principles of the artificial neural networks first appeared in the pioneering paper McCulloch and Pitts (1943). In this early work, five fundamental assumptions were formulated:

1. the activity of an artificial neuron is all-or-nothing;

2. a certain fixed number of synapses greater than one must be excited for a neuron to be excited;

3. the only significant delay within the neural system is the synaptic delay;

4. the activity of any inhibitory synapse absolutely prevents the excitation of the neuron at that time;

5. the structure of the interconnection network does not change over time.

Another widely applied principle is the so-called *Hebbian Rule* (or *Hebbian Learning Law*), that was first stated in Hebb (1949) as follows.
"When an axon of cell A is near-enough to excite cell B and when it repeatedly and persistently takes part in firing it, then some growth process or metabolic change takes place in one or both these cells such that the efficiency of cell A is increased".
Roughly speaking, such rule can be summarized as "cells that fire together wire together", that is the connections between two neurons might be strengthened (in terms of weights) if the neurons fire simultaneously.
Notice that above historical principles do not all apply to most modern neural network architectures. Following sections contain the description of the neural network models which have been employed (also in) neurocryptography.

1.3 The Perceptron

1.3.1 Basic structure

The first complete neural computation model, that is called Perceptron, first appeared in (Rosenblatt, 1958) and serves as a building block to most later models. The Perceptron posses the fundamental structure of a neural cell, of several weighted input connections and a single output channel. The input/output relations are defined to be

$$z = \sum_i w_i x_i \quad \text{(summation output)} \tag{1.1}$$

$$y = f_N(z) \quad \text{(cell output)}, \tag{1.2}$$

where w_i is the weight at the input x_i. Such weights, according to the biological model, need to be adjustable. y, instead, denotes the cell's output, that is a nonlinear activation function f_N (see Subsection 1.3.2 below) of the node output z.
It is customary to consider a network of n Perceptrons; in this case, by letting z_i be the summation output of the i-th Perceptron and its inputs $x_{i,j}$, the summation relation becomes

$$z_i = \sum_{j=1}^{m} w_{ij} x_{ij}, \quad i \in \{1, \ldots, n\}, \tag{1.3}$$

or, in vector form

$$z_i = \vec{w}_i^{\mathsf{T}} \vec{x}_i, \tag{1.4}$$

where

$$\vec{w}_i = [w_{i1}, \ldots, w_{in}]^{\mathsf{T}}$$
$$\vec{x}_i = [x_{i1}, \ldots, x_{in}]^{\mathsf{T}},$$

for every $i \in \{1, \ldots, n\}$.

1.3.2 Activation functions

The Perceptron's cell's output differs from the summation output 1.4 by the activation operation of the neuron's body, just as the output of the biological cell differs from the weighted sum of its inputs. Such operation is in terms of an activation function $f_N(z_i)$, which is a non-linear operator yielding i-th

neuron's output y_i to satisfy certain limiting properties. Different functions are in use, but the most common choice is the sigmoid function, that is

$$y_i = f_N(z_i) = \frac{1}{1+e^{-z_i}}, \quad i \in \{1, \dots, n\}. \tag{1.5}$$

Under this choice, the following relations are satisfied (see figure TO ADD!!).

$$\begin{aligned} z_i \to -\infty \quad &\Longleftrightarrow \quad y_i \to 0; \\ z_i = 0 \quad &\Longleftrightarrow \quad y_i = 0.5; \\ z_i \to \infty \quad &\Longleftrightarrow \quad y_i \to 1. \end{aligned}$$

Another popular activation function is

$$y_i = \frac{1+\tanh(z_i)}{2} = \frac{1}{1+e^{-2z_i}}, \tag{1.6}$$

whose shape is similar to the one of the previous function. The simplest activation one could choose is the Heaviside step function:

$$y_i = H(z_i) = \begin{cases} 1 & \text{if} \quad z_i \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Main advantage of the former two over the latter is that they are (mathematically) smooth.
In many applications the activation function's output is translated so that it ranges in $(-1, 1)$, rather than in $(0, 1)$. In particular, sigmoid function (1.5) becomes

$$y_i = \frac{2}{1+e^{-z_i}} - 1 = \tanh(z_i/2), \tag{1.7}$$

while activation (1.6) is transformed to

$$y_i = \tanh(z_i) = \frac{1-e^{-2z_i}}{1+e^{-2z_i}}. \tag{1.8}$$

We recall that in many neural network models, a bias term is added to the summation output, so that (1.1) becomes:

$$z = b + \sum_i w_i x_i, \tag{1.9}$$

where b is subject to the same training procedures as the the weights of the net.

Such training algorithms revealed themselves to be challenging for researchers, especially from the introduction of the so called Multi-Layer Perceptron (MLP). In fact, already in the late 1960s, limitations of single-layer architectures were shown (see Minsky and Papert (1969)), together with the possibility of solving more complicated problems using MLP's. However, finding efficient algorithms to adjust the weights of such nets was problematic and took several years. Some examples of solutions to these problems are presented in the following sections. All of such algorithms are thought to work in discrete time, while taking into account partial derivatives of the quantities of interest. Please read appendix A for further detail about Finite Difference Methods (FDMs).

1.3.3 Least Mean Square Training

We first discuss a simple algorithm to set weights of the so-called Adaline (ADAptive LInear NEuron) network. Such model first appeared in Widrow and Hoff (1960) and it has the basic structure of a bipolar perceptron. The nonlinear operator of equation (1.2) is here a simple threshold element, to yield the Adaline output y as

$$y = \text{sign}(z). \tag{1.10}$$

The training of such NN is according to the following procedure, that is called Least Mean Square (LMS). Given L training sets:

$$\vec{x}_1, \ldots, \vec{x}_L; \quad d_1, \ldots, d_L, \tag{1.11}$$

where

$$\vec{x}_i = [x_{i1}, \ldots, x_{in}]^\intercal, \quad i \in \{1, \ldots, L\}, \tag{1.12}$$

i denoting the i-th set, n being the number of inputs and d_i denoting the desired output of the neuron, we define a training cost, such that:

$$J(\vec{w}) = \mathbb{E}\left[e_k^2\right] \approx \frac{1}{L}\sum_{k=1}^{L} e_k^2; \tag{1.13}$$

$$\vec{w} = [w_1, \ldots, w_n]^\intercal, \tag{1.14}$$

where e_k denotes training error at the k-th set, namely

$$e_k = d_k - z_k, \quad k \in \{1, \ldots, L\}, \tag{1.15}$$

and z_k being the neuron's actual output.
Following the above notation we have that

$$\mathbb{E}\left[e_k^2\right] = \mathbb{E}\left[d_k^2\right] + \vec{w}^{\intercal}\mathbb{E}\left[\vec{x}_k\vec{x}_k^{\intercal}\right]\vec{w} - 2\vec{w}^{\intercal}\mathbb{E}\left[d_k\vec{x}_k\right], \tag{1.16}$$

with

$$\mathbb{E}\left[XX^{\intercal}\right] = R \tag{1.17}$$

$$\mathbb{E}\left[\vec{d}X\right] = \vec{p}, \tag{1.18}$$

where X denotes the input matrix, to yield the gradient of J such that

$$\nabla J = \frac{\partial J(\vec{w})}{\partial \vec{w}} = 2R\vec{w} - 2\vec{p}. \tag{1.19}$$

Hence, the optimal LMS setting of the weights becomes

$$\nabla J = 0, \tag{1.20}$$

which, by equation (1.19) satisfies the weight setting of

$$\vec{w}^{LMS} = R^{-1}\vec{p}. \tag{1.21}$$

Note that above LMS procedure employs expecting whereas the training data is limited to a small number of L sets, such that the sample averages will be inaccurate estimates of the true expectations.

1.3.4 Gradient Descent Training

The gradient descent training procedure, also known in literature as steepest descent procedure, attempts to provide weight-setting estimates from one training set to the next, starting with $L = n + 1$ training sets, where n is the number of inputs. Note that, from n weights, it is imperative that

$$L > n + 1 \tag{1.22}$$

Denoting a weights vector setting after the m-th training set, $m \in \{1, \ldots, L\}$, as $\vec{w}(m)$, then

$$\vec{w}(m+1) = \vec{w}(m) + \Delta\vec{w}(m), \tag{1.23}$$

where $\Delta\vec{w}$ is the variation in $\vec{w}(m)$, given by

$$\Delta\vec{w}(m) = \mu\nabla J\left(\vec{w}(m)\right). \tag{1.24}$$

In the above equation, μ is the rate parameter whose setting discussed below, and ∇J denotes the gradient of the training cost as defined in (1.13). With these conventions, the steepest descent procedure of (1.23) follows the steps below.

1. Apply input vector $\vec{x}_m$ and the desired output d_m for the m-th training set .

2. Determine e_m^2, that is

$$e_m^2 = [d_m - \vec{w}(m)^\intercal \vec{x}(m)]^2 = \tag{1.25}$$
$$= d_m^2 - 2d_m \vec{w}(m)^\intercal \vec{x}(m) + \vec{w}(m)^\intercal \vec{x}(m)\vec{x}(m)^\intercal \vec{w}(m) \tag{1.26}$$

3. Evaluate

$$\nabla J = \frac{\partial e_m^2}{\partial \vec{w}(m)} = 2\vec{x}(m)\vec{w}(m)^\intercal \vec{x}(m) - 2d_m \vec{x}(m) = \tag{1.27}$$
$$= -2\left[d_m - \vec{w}(m)^\intercal \vec{x}(m)\right]\vec{x}(m) = -2e_m \vec{x}(m), \tag{1.28}$$

   thus obtaining an approximation to ∇J by using e_m^2 as the approximate to J.

4. Update $\vec{w}(m+1)$ via equations (1.23) and (1.24), namely

$$\vec{w}(m+1) = \vec{w}(m) - 2\mu e_m \vec{x}(m). \tag{1.29}$$

   Here μ is chosen to satisfy

$$\frac{1}{\lambda_{max}} > \mu > 0, \tag{1.30}$$

   where λ_{max} is the maximum eigenvalue of matrix R of equation (1.17).

In the following sections we will discuss more advanced training algorithms, that are designed to be compatible with multiple-layer architectures.

1.4 Back Propagation Training Algorithm

Back Propagation (BP) algorithm was first proposed in Rumelhart, Hinton and Williams (1986), as a solution for setting weights (and hence for the training) of multi-layer perceptrons. The availability of a rigorous method to set intermediate weights, namely to train the hidden layers of neural networks, gave a major boost to the further development of such models.
In this section this algorithm is presented in detail.

1.4.1 The Algorithm

The BP algorithm starts with computing values of the output layer, which is by definition the only one whose desired outputs are available. Let ϵ be the the *error-energy* at the output layer, that is

$$\epsilon = \frac{1}{2}\sum_{k=1}^{N} e_k^2 = \frac{1}{2}\sum_{k=1}^{N} (d_k - y_k)^2 , \tag{1.31}$$

where N is the number of neurons in the output layer. We also recall d_k is the desired output and e_k the error at the k-th entry . Consider the gradient of ϵ:

$$\nabla\epsilon_k = \frac{\partial\epsilon}{\partial w_{kj}}. \tag{1.32}$$

Recall that w_{kj} denotes the weight of the j-th input to the k-th neuron. By the gradient descent procedure (see Subsection 1.3.4), we have that

$$w_{kj}(m+1) = w_{kj}(m) + \Delta w_{kj}(m), \tag{1.33}$$

where

$$\Delta w_{kj} = -\eta\frac{\partial\epsilon}{\partial w_{kj}} \tag{1.34}$$

and $\eta \in (0,1)$ is the rate parameter. Note that the minus sign indicates a down-hill direction towards a minimum.
With the notations of Subsection 1.3.1, we can now substitute

$$\frac{\partial\epsilon}{\partial w_{kj}} = \frac{\partial\epsilon}{\partial z_k}\frac{\partial z_k}{\partial w_{kj}} \tag{1.35}$$

and

$$\frac{\partial z_k}{\partial w_{kj}} = x_j(p) = y_j(p-1), \tag{1.36}$$

with p denoting the output layer. This way, equation (1.35) becomes

$$\frac{\partial \epsilon}{\partial w_{kj}} = \frac{\partial \epsilon}{\partial z_k} x_j(p) = \frac{\partial \epsilon}{\partial z_k} y_j(p-1). \tag{1.37}$$

We now have to define

$$\phi_k(p) = -\frac{\partial \epsilon}{\partial z_k}, \tag{1.38}$$

so that (1.37) yields

$$\frac{\partial \epsilon}{\partial w_{kj}} = -\phi_k(p) x_j(p) = -\phi_k(p) y_j(p-1). \tag{1.39}$$

So by this last equation and (1.34) we get

$$\Delta w_{kj} = \eta \phi_k(p) x_j(p) = \eta \phi_k(p) y_j(p-1). \tag{1.40}$$

Furthermore, by (1.38):

$$\phi_k = -\frac{\partial \epsilon}{\partial z_k} = -\frac{\partial \epsilon}{\partial y_k} \frac{\partial y_k}{\partial z_k}. \tag{1.41}$$

But, going back to error-energy definition in (1.31)

$$\frac{\partial \epsilon}{\partial y_k} = -\left(d_k - y_k\right) = y_k - d_k, \tag{1.42}$$

where we recall

$$y_k = f_N(z_k) = \frac{1}{1 + e^{-z_k}} \tag{1.43}$$

and so we have that

$$\frac{\partial y_k}{\partial z_k} = y_k(1 - y_k). \tag{1.44}$$

Consequently, by equations (1.41) and subsequent ones,

$$\phi_k = y_k(1 - y_k)(d_k - y_k), \tag{1.45}$$

such that, at the output layer, by (1.34) and (1.35)

$$\Delta w_{kj} = -\eta \frac{\partial \epsilon}{\partial w_{kj}} = -\eta \frac{\partial \epsilon}{\partial z_k} \frac{\partial z_k}{\partial w_{kj}}, \tag{1.46}$$

where, by (1.41)

$$\Delta w_{kj}(p) = \eta \phi_k(p) y_j(p-1), \tag{1.47}$$

with ϕ_k being as in (1.45), to complete the derivation of the setting of output layer's weights.
If we now consider, in general, the i-th input to the j-th neuron of the r-th hidden layer, we still have, as before

$$\Delta w_{ji} = -\eta \frac{\partial \epsilon}{\partial w_{ij}}, \quad j \in \{1, \dots, N\}; \quad i \in \{1, \dots, n\}. \tag{1.48}$$

Similarly to (1.35) it holds

$$\frac{\partial \epsilon}{\partial w_{ji}} = \frac{\partial \epsilon}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}} \tag{1.49}$$

and taking into account equation in (1.36) and the expression of ϕ in (1.41) we get

$$\Delta w_{ji} = -\eta \frac{\partial \epsilon}{\partial z_j} y_i(r-1) = \eta \phi_j(r) y_i(r-1). \tag{1.50}$$

This way, again by right-hand side of (1.41), we obtain

$$\Delta w_{ji} = -\eta \left[\frac{\partial \epsilon}{\partial y_j(r)} \frac{\partial y_j}{\partial z_j} \right] y_i(r-1), \tag{1.51}$$

where $\frac{\partial \epsilon}{\partial y_j}$ is inaccessible (just like $\phi_j(r)$ above).
Since we're considering backward propagation from the output, error-energy is only determined by upwards neurons:

$$\frac{\partial \epsilon}{\partial y_j(r)} = \sum_k \frac{\partial \epsilon}{\partial z_k(r+1)} \left[\frac{\partial z_k(r+1)}{\partial y_j(r)} \right] = \sum_k \frac{\partial \epsilon}{\partial z_k} \left[\frac{\partial}{\partial y_j(r)} \sum_m w_{km}(r+1) y_m(r) \right]. \tag{1.52}$$

We remark that, in above expression, summation over k is over the neurons of layer $r+1$ that connect to $y_j(r)$, while summation over m is performed over all inputs to each k-th neuron of the $(r+1)$-th layer. Keeping in mind the definition of ϕ, last equation yields

$$\frac{\partial \epsilon}{\partial y_j(r)} = \sum_k \frac{\partial \epsilon}{\partial z_k(r+1)} w_{kj} = -\sum_k \phi_k(r+1) w_{kj}(r+1), \tag{1.53}$$

since only $w_{kj}(r+1)$ is connected to $y_j(r)$. Consequently, by equations (1.41), (1.44) and (1.53)

$$\phi_j(r) = \frac{\partial y_j}{\partial z_j} \sum_k \phi_k(r+1) w_{kj}(r+1) = y_j(r) \left[1 - y_j(r)\right] \sum_k \phi_k(r+1) w_{kj}(r+1) \tag{1.54}$$

and finally, with some other substitutions, we obtain

$$\Delta w_{ji}(r) = \eta \phi_j(r) y_i(r-1), \tag{1.55}$$

that is a function of ϕ and of the weights of the $(r+1)$-th layer.
Back Propagation algorithm thus propagates up to the first layer ($r = 1$), iterating over a pre-fixed set of training vectors. Weights are initialized randomly, while the learning rate η should be adjusted stepwise for faster convergence.

1.4.2 Refinements and Extensions of BP

A series of modifications to the Back Propagation algorithm were introduced by scientists in order to improve the training procedure. Most important extensions include the introduction of bias into neural networks, smoothing the weight adjustment function and rescaling the activation function.
If we want to consider bias in a model, we can regard it as a trainable quantity, just as is any other weight. In general, the bias term b_i at the input to the i-th neuron is realized in term of a constant input B with its relative weight:

$$b_i = w_i^b B, \quad i \in \{1, \dots, N\}. \tag{1.56}$$

This algorithm sometimes tends to go through instability issues when computing weights and several smoothing techniques were proposed to deal with such problems. For instance, in Sejnowski and Rosenberg (1987), a smoothing term to modify equation (1.40) is given:

$$\begin{aligned} \Delta w_{ij}^{(m)} &= \alpha \Delta w_{ij}^{(m-1)} + (1-\alpha)\phi_i(r) y_j(r-1) \\ w_{ij}^{(m+1)} &= w_{ij}^{(m)} + \eta \Delta w_{ij}^{(m)}, \end{aligned} \tag{1.57}$$

where $\alpha \in (0,1)$. Last, modifying the range of the sigmoid function, typically to $(-0.5,0.5)$ seems to improve the convergence rate of BP algorithm.

1.5 Adam

In this section we present Adam, a recent method for stochastic optimization that first appeared in Kingma and Ba (2015). This method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name Adam is derived from adaptive moment estimation. Such algorithm is well-suited for the training of neural networks and indeed it is applied to most modern architectures; following chapters of this work include examples of such NNs.

1.5.1 The Algorithm

Here the theory of *Adam* optimizer is presented according to the original publication where it was first proposed. Pseudocode for this algorithm is reported below.

Algorithm 1: *Adam* stochastic optimization algorithm. g_t^2 indicates the elementwise square $g_t \cdot g_t$.

Require: α: Stepsize;
Require: $\beta_1, \beta_2 \in [0,1)$: Exponential decay rates;
Require: $f(\theta)$: Stochastic objective function with parameters θ;
Require: θ_0: Initial parameter vector;
$m_0 \leftarrow 0$ (Initialize first moment vector);
$v_0 \leftarrow 0$ (Initialize second moment vector);
$t \leftarrow 0$ (Initialize time step);
while θ_t *not converged* **do**
  $t \leftarrow t + 1$;
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients);
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ (Biased first moment est.);
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ (Biased second moment est.);
  $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ (Bias-corrected first moment est.);
  $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ (Bias-corrected second moment est.);
  $\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ (Update parameters);
end
return θ_t

Let $f(\theta)$ be a noisy objective function: a stochastic scalar function that is differentiable with respect to parameters θ. The stochasticity might come from the evaluation at random subsamples (minibatches)of datapoints, or arise from inherent function noise.

We are interested in minimizing the expected value of such function, that is $\mathbb{E}[f(\theta)]$. Denote by $f_1(\theta), \ldots, f_T(\theta)$ the realizations of the function at subsequent time steps $1, \ldots, T$. Let $g_t = \nabla_\theta f_t(\theta)$ be the gradient, that is the vector of partial derivatives of f (w.r.t. θ) evaluated at time step t.
The algorithms updates exponential moving averages of the gradient and of the squared gradient; denote these estimates by m_t and v_t, respectively. Exponential decay rates of these moving averages are controlled by the hyper parameters $\beta_1, \beta_2 \in [0,1)$. The moving average themselves are estimates of the mean and the uncentered variance (the second moment) of the gradient. Since these estimates are initialized as vectors of zeros, they are biased towards zero, especially during the initial time steps and when the decay rates are small (i.e. $\beta_1, \beta_2 \approx 1$). This initialization bias can however be easily counteracted, resulting in bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$, as described in the following.
Recall that $g_1, \ldots, g_T$ denote the gradients at subsequent time steps; each can be seen as a draw from an underlying gradient distribution $p(g_t)$. Consider the second moment estimate. As can be seen in pseudocode 1, the exponential moving average is initialized as $v_0 = 0$. First note that the update at time step t, $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$, can be written as a function of the gradients at all previous time steps:

$$v_t = (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2. \tag{1.58}$$

We wish to know how $\mathbb{E}[v_t]$, the expected value of the exponential moving average at time step t, relates to the the true second moment $\mathbb{E}\left[g_t^2\right]$, so we can correct for discrepancy between the two. Taking expectations of both sides of equation (1.58), we have

$$\mathbb{E}[v_t] = \mathbb{E}\left[(1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2\right] = \tag{1.59}$$

$$= \mathbb{E}\left[g_t^2\right] (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} + \zeta = \tag{1.60}$$

$$= \mathbb{E}\left[g_t^2\right] (1-\beta_2^t) \ + \zeta, \tag{1.61}$$

where $\zeta = 0$ if the true second moment $\mathbb{E}\left[g_t^2\right]$ is stationary. If not, ζ can be kept small since the exponential decay rate β_2 can (and should) be chosen such that the exponential moving average assigns small weights to gradients far in the past. What is left is the term $(1-\beta_2^t)$ which is caused by initializing

the running average with zero. In algorithm 1 we therefore divide by this term to correct the initialization bias.
Going back to *Adam*'s update rule, we have to focus on its careful choice of stepsizes. Assuming $\epsilon = 0$, the effective step taken in the parameter space at time step t is

$$\Delta_t = \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}}. \tag{1.62}$$

The effective step size has two upper bounds:

$$\begin{cases} |\Delta_t| \leq \alpha \frac{1-\beta_1}{\sqrt{1-\beta_2}} & \text{if} \quad (1-\beta_1) > \sqrt{1-\beta_2}; \\ |\Delta_t| \leq \alpha & \text{otherwise.} \end{cases} \tag{1.63}$$

The first case only happens in the most severe case of sparsity: when the gradient has been zero at all time steps except at the current time step. Otherwise, the effective step size will be smaller. In more common scenarios, we will have that

$$\frac{\hat{m}_t}{\sqrt{v_t}} \approx \pm 1,$$

since

$$\frac{|\mathbb{E}\left[g\right]|}{\sqrt{\mathbb{E}\left[g^2\right]}} \leq 1.$$

Hence the effective magnitude of the steps taken in the parameter space at each time step are approximately bounded by the stepsize setting α.
The effective step size Δ_t is also invariant to the scale of the gradients; rescaling the gradients g with factor c won't lead to changes:

$$\frac{c\hat{m}_t}{\sqrt{c^2\hat{v}_t}} = \frac{\hat{m}_t}{\sqrt{\hat{v}_t}}. \tag{1.64}$$

Chapter 2

Genetic Algorithms: a brief overview

Genetic Algorithms were invented by John Holland in the 1960s and can be considered a key building block of artificial intelligence. His aims were that of formally studying the phenomenon of natural adaptation and that of importing such adaptation mechanisms into computer systems. In Holland (1975), Genetic Algorithms (GA's) are presented as abstractions of biological evolution. Moreover, a theoretical framework for adaptation under GA's is described.
Holland's GA is a method for evolving from one population of "chromosomes" to a new population by using an imitation of natural selection together with the genetic-inspired operators of crossover and mutation. In this chapter we briefly discuss how GA's works and why. Main reference for this part is Mitchell (1998), whose author was one of Holland's students at MIT.

2.1 Definitions and Setting

The genome of the evolutionary model is represented by a multiset of size N:

$$\mathbf{P}(t) = \{\vec{v}_1, \ldots, \vec{v}_N\}, \tag{2.1}$$

whose elements (that are not necessarily distinct) are called chromosomes. Time variable $t \in \mathbb{N}$ denotes the t-th generation of the population. Each chromosome $\vec{v}_i$ is a bit vector of fixed length n:

$$\vec{v}_i = \left[v_i^{(1)}, \ldots, v_i^{(n)}\right], \quad i \in \{1, \ldots, N\}, \tag{2.2}$$

where $v_i^{(j)} \in \{0,1\}$, $\forall j = 1, \ldots, n$. We recall that there exist more advanced ways to represent chromosomes, but here we are limiting to the binary representation for simplicity.
We now need to define some probabilities which are involved in the GA. Selection is based on the definition of a fitness function over the population:

$$f : \mathbf{P}(t) \longrightarrow \mathbb{R}_0^+. \tag{2.3}$$

We're now allowed to define total fitness and average fitness, namely

$$F = \sum_{\vec{v} \in \mathbf{P}(t)} f(\vec{v}) \quad \text{and} \tag{2.4}$$

$$\overline{F} = \frac{F}{N}, \tag{2.5}$$

respectively. We can hence define the selection probability $P_S(\vec{v})$, for each chromosome $\vec{v}$:

$$P_S(\vec{v}) = \frac{f(v)}{F}. \tag{2.6}$$

Note that (2.5), despite not being explicitly computed in the algorithm, has great theoretical relevance (see Subsection 2.3.2).
The other probabilistic quantities involved are crossover probability P_C and mutation probability P_M. The former can also be regarded as the fraction of elements of $\mathbf{P}(t)$ that will go through reproduction, while the latter can be thought of as the percentage of bits (of all chromosomes) that will change due to random mutations. Both values are fixed and typically we have $P_C >> P_M$.

2.2 The Algorithm

Each execution of the Genetic Algorithm produces an offspring of chromosomes: it moves from population $\mathbf{P}(t)$ to $\mathbf{P}(t+1)$. This evolution process happens in three main phases:

1. selection via *Wheel of Fortune* model;
2. reproduction via crossover;
3. mutation.

2.2.1 Selection

As anticipated just above, in the first phase the algorithm mimics the behavior of wheel of fortune. First, a sequence $\{q_i\}_{i=0}^{N}$ is computed recursively in the following way:

$$\begin{aligned} q_0 &= 0, \\ q_1 &= P_S(\vec{v}_1), \\ q_2 &= P_S(\vec{v}_1) + P_S(\vec{v}_2), \\ &\vdots \\ q_i &= \sum_{j=1}^{i} P_S(\vec{v}_j), \\ &\vdots \\ q_N &= \sum_{j=1}^{N} P_S(\vec{v}_j) = 1. \end{aligned} \tag{2.7}$$

Such sequence determines a partition of the interval [0,1] and the algorithm proceeds by "spinning" the wheel of fortune N times . This means, a random number $r \in [0,1]$ is generated; if $r \leq 1$, then $\vec{v}_1$ is chosen; else if $q_{j-1} < r \leq q_j$ the chosen chromosome is $\vec{v}_j$. Notice that, with this procedure, a single chromosome may be selected several times.

2.2.2 Reproduction

Second phase simulates biological crossover, thanks to which genetic recombination is guaranteed at every generation. For each chromosome $\vec{v}$ selected in previous phase

- a random number $r \in [0,1]$ is generated;
- if $r < P_C$, then $\vec{v}$ is chosen for crossover.

Recall that P_C is a fixed selection probability. Chromosomes chosen this way are then randomly coupled. Whenever their number is odd, another random chromosome can be added (or removed from the reproduction subset).
For each couple $\vec{v}_A = \left[v_A^{(1)}, \ldots, v_A^{(n)}\right]$ and $\vec{v}_B = \left[v_B^{(1)}, \ldots, v_B^{(n)}\right]$, a random integer $c \in \{1, 2, \ldots, n-1\}$ is computed and the chromosomes are split in

the following way:

$$\vec{v}_A = \left[v_A^{(1)}, \ldots, v_A^{(c)} | v_A^{(c+1)}, \ldots, v_A^{(n)}\right];$$
$$\vec{v}_B = \left[v_B^{(1)}, \ldots, v_B^{(c)} | v_B^{(c+1)}, \ldots, v_B^{(n)}\right]. \tag{2.8}$$

After that, offspring chromosomes $\vec{w}_A$ and $\vec{w}_B$ are generated by swapping the blocks found above:

$$\vec{w}_A = \left[v_A^{(1)}, \ldots, v_A^{(c)} | v_B^{(c+1)}, \ldots, v_B^{(n)}\right];$$
$$\vec{w}_B = \left[v_B^{(1)}, \ldots, v_B^{(c)} | v_A^{(c+1)}, \ldots, v_A^{(n)}\right]. \tag{2.9}$$

In other words

$$\vec{w}_A = \begin{cases} v_A^{(i)} & \text{if} \quad 1 \leq i \leq c \\ v_B^{(i)} & \text{if} \quad c+1 \leq i \leq n; \end{cases} \tag{2.10}$$

and

$$\vec{w}_B = \begin{cases} v_B^{(i)} & \text{if} \quad 1 \leq i \leq c \\ v_A^{(i)} & \text{if} \quad c+1 \leq i \leq n. \end{cases} \tag{2.11}$$

2.2.3 Mutation

For each chromosome $\vec{v}_i$ we consider its every bit $v_i^{(j)}$, $\forall j = 1, \ldots, n$ and the algorithm generates a random $r_j \in [0,1]$. If $r_j < P_M$, then the corresponding bit is changed to its two's complement (otherwise it's left unchanged), where we recall that P_M denotes the mutation probability. After these three phases we obtain a new population $\mathbf{P}(t+1)$ with

$$|\mathbf{P}(t+1)| = |\mathbf{P}(t)| = N. \tag{2.12}$$

2.3 Holland Theorem

Despite GA's are quite simple to describe and implement, understanding their behavior appears to be quite complicated. According to Holland's theory, such algorithms work by discovering, emphasizing and recombining good "building blocks" (taken from chromosomes with high fitness). Such building blocks are formally named "schemas".

2.3.1 Definitions

A schema is a set of bit strings that can be described as a template made up of the symbols 0, 1 and $*$, which denotes a "free entry". For example, the schema

$$K = 1 * *1 \tag{2.13}$$

represents the set of all four-bit strings that begin and end with 1. Vectors belonging to one of such sets are named instances of that schema.
For a scheme H, we denote $o(H)$ the order of H, that is its number of defined bits ($\neq *$) and we let $d(H)$ be the schema's length (the distance between its outermost defined bits). In example before (2.13), we have $o(K) = 2$ and $d(K) = 3$.

2.3.2 Behavior of Genetic Algorithms

After this few premises, we can discuss how the GA processes schemas.
Any given bit string of length l in an instance of 2^l different schemas. Thus, given a population of N chromosomes, it contains instances of k different schemes, with $2^l \leq k \leq N2^l$. This means that, at a given generation t, while the GA is explicitly evaluating the fitness of the N chromosomes, it is actually implicitly estimating the average fitness of a much larger number of schemas (where the average fitness of a schema is defined to be the average fitness of all possible instances of that schema). Obviously, the estimates of schema average fitness are not calculated or stored by the GA, However, the algorithm's behavior can be described as though it was actually calculating and storing these averages.
Let H be a schema with at least one instance present in $\mathbf{P}(t)$ and let $m(H, t)$ the number of instances of H at time t. Moreover let $\hat{U}(H, t)$ be the observed average fitness. We want to compute

$$\mathbb{E}\left[m(H, t+1)\right], \tag{2.14}$$

the expected number of instances of H at time $t+1$. Assume that selection is carried out as described in previous section; the expected number of offspring of a chromosome $\vec{v}$ is equal to

$$\frac{f(\vec{v})}{\overline{F}}. \tag{2.15}$$

We will write $\vec{v} \in H$ to denote "$\vec{v}$ is an instance of H".
Ignoring (only for now) the effects of crossover and mutation, we have

$$\mathbb{E}\left[m(H, t+1)\right] = \sum_{\vec{v} \in H} \frac{f(\vec{v})}{\overline{F}} = \frac{\hat{U}(H,t)}{\overline{F}} m(H,t) \tag{2.16}$$

by definition, since

$$\hat{U}(H,t) = \frac{\sum_{\vec{v} \in H} f(\vec{v})}{m(H,t)}. \tag{2.17}$$

Thus even though the GA does not compute $\hat{U}(H,t)$ explicitly, the propagation of schema instances in the population depends on this quantity.
Crossover and mutation can both destroy and create instances of H, but we limit to considering the disruptive effects. We say schema H "survives" under crossover if at least one instance of such schema is present in the next generation. Letting $S_C(H)$ be the probability of such survival, with the customary notations we have

$$S_C(H) \geq 1 - P_C \left(\frac{d(H)}{n-1} \right). \tag{2.18}$$

In short, the probability of survival under crossover is higher for short schemas.
The effects of mutations can be quantified as follows. Let $S_M(H)$ be the probability that schema H will survive under mutation. It can be expressed as

$$S_M(H) = (1 - P_M)^{o(H)}, \tag{2.19}$$

where we recall that $o(H)$ is the number of defined bits in the schema. This formula holds because mutation acts independently on each bit of each chromosome. In this case, longer schemes appear to be more resistant.
These disruptive effects can be used to amend equation (2.16), in order to get a lower bound for $\mathbb{E}\left[m(H, t+1)\right]$. Hence we get

$$\mathbb{E}\left[m(H, t+1)\right] \geq \frac{\hat{U}(H,t)}{\overline{F}} m(H,t) \left(1 - P_C \left(\frac{d(H)}{n-1} \right)\right) \left[(1 - P_M)^{o(H)}\right]. \tag{2.20}$$

Last result is known as Holland Theorem (or Schemata Theorem). It implies that short, high-average-fitness schemas will see their instances increase exponentially over time.

Chapter 3

Neural Cryptography: History

3.1 Pioneers of Neural Cryptography

3.2 Neural Networks in Cryptography: an interesting attempt

This section describes one of the first attempts in designing a neural network to be practically used in both cryptography and cryptanalysis. It must be said that the results attained in Volná (2000), the paper to which I refer, are still quite rough. However, its importance lies in influencing subsequent works. (CITATIONS NEEDED!!!!!).

3.2.1 Genetic Algorithms in Neural Network Design

Main element in this research are feedforward neural nets with Back Propagation, but the most interesting characteristic of Volna's approach is that it relies on EP (Evolutionary Programming): genetic algorithms are used for optimization of the designed NN topology. This is based on a previous work of the same author, that is Volná (1998).
The criterion of choice is the minimization of the sum of square of deviation of output from neural network. At first, the maximal architecture of the nets is proposed, then, at each step, to optimize the population it is necessary to solve the cryptographic problems of interest. Thereafter the process of genetic algorithms is applied. An optimal population is found either when it achieves the maximal generation or when fitness function achieves the maximal defined value.
At this point, it is required to complete the "best" architecture by adapting weights and hence three digits are generated for every connection coming out from a unit. If the connection does not exist, three zeroes are assigned, else weights are computed this way:

$$w_{ij,kl} = \eta[e_2(e_1 2^1 + e_0 2^0)]; \quad \begin{array}{r} i,k \in \{1,\dots,L\}, \\ j \in \{1,\dots,n_i\}, \\ l \in \{1,\dots,n_k\}, \end{array} \tag{3.1}$$

where $w_{ij,kl} = w(x_{ij}, x_{kl})$ is the weight value between the j-th unit in the i-th layer and the l-th unit in the k-th layer and

$$\begin{aligned} \eta &= \text{learning parameter}; \quad \eta \in (0,1) \\ e_0, e_1 &= \text{random digits} \\ e_2 &= \text{sign bit.} \end{aligned}$$

L denotes the total number of layers, while n_i and n_k are the number of units in layers i and k, respectively. Recall that these values are determined by the execution of the genetic algorithm.
Error between the desired and the real output is then computed and stored in the vector $\vec{E}$. On the basis of it, the algorithm computes the fitness precursor value $f_i^\star$, for each individual $i = 1, \ldots, N$, that is

$$f_i^\star = k_1(E_i)^2 + k_2(U_i)^2 + k_3(L_i)^2, \tag{3.2}$$

where k_j, j = 1,2,3 are fixed constants and

$$\begin{aligned} E_i &= \text{error for network } i \\ U_i &= \text{number of hidden units} \\ L_i &= \text{number of hidden layers.} \end{aligned}$$

The general fitness function f is then calculated as follows:

$$f_i = \begin{cases} k - (f_i^\star + k_5) & \text{if} \quad E_i > k_4 \\ k - f_i^\star & \text{otherwise.} \end{cases}$$

In the above expressions, k, k_4 and k_5 also denote constants. The genetic algorithm used by Volná makes use of standard crossover and mutation procedures, as the ones described in the specific chapter. Here we omit details.
Adaptation of the best found network architecture is finished with Back Propagation.

3.2.2 Volná's experiment

In this work, the parameters of the adapted neural network become the key of an encryption/decryption algorithm. Topology of such NN clearly depends on the training set that, in Volná's case, is represented in table 3.1, while the chain of chars of the plain text is equivalent to a binary value, that is 96 less than its ASCII code. The cipher text is a randomly generated chain of bits. Thus, the decrypting neural network has six input units and five output ones, with an unspecified number of hidden units. Viceversa, the net that performs encryption has five input neurons and six output ones.
This encryption scheme is symmetric: it uses a single key for both encryption and decryption. It is interesting to notice that Volná, in his publication, thought that this feature was very bad for his encryption system, due to the

Plaintext			**Cyphertext**
Char	*ASCII Code*	*Bit String Representation*	*Bit String Representation*
a	97	00001	000010
b	982	00010	100110
c	99	00011	001011
d	100	00100	011010
e	101	00101	100000
f	102	00110	001110
g	103	00111	100101
h	104	01000	010010
i	105	01001	001000
j	106	01010	011110
k	107	01011	001001
l	108	01100	010110
m	109	01101	011000
n	110	01110	011100
o	111	01111	101000
p	112	10000	001010
q	113	10001	010011
r	114	10010	010111
s	115	10011	100111
t	116	10100	001111
u	117	10101	010100
v	118	10110	001100
w	119	10111	100100
x	120	11000	011011
y	121	11001	010001
z	122	11010	001101

Table 3.1: The training set.

popularity and goodness of asymmetric, non-neural cryptography. In fact, this model has many limits, but we'll see in next chapters that most modern (and secure) neurocryptographic protocols still are symmetric. Leaving aside asymmetric protocols is indeed one of the main strengths of this new approach to cryptography.
Going back to the protocol, the key will include the adapted neural network parameters; that is its topology (architecture) and its configuration (the weight values on connections). Uniquely identifying the NN is hence equivalent to uniquely characterizing the encryption/decryption function.

3.3 The KKK Key Exchange Protocol

We now deal with the first complete cryptosystem based on neural networks. It has been later referred to as *KKK*, from the surnames of its inventors. It first appeared in Kanter, Kinzel and Kanter (2001), a year later than Volná's work.
Here, a new concept appears in neurocryptography: synchronization of two nets to build a secure communication channel.

3.3.1 The Protocol

Object of the above cited work is a key-exchange protocol based on a learning process of feedforward neural networks. The two NN's participating in the communication start from private key vectors $E_k(0)$ and $D_k(0)$. Mutual learning from the exchange of public information leads the two nets to develop a common, time dependent key: $E_k(t) = -D_k(t)$. This is then used for both encryption and decryption.
This phenomenon, known as synchronization of synaptic weights, has the core feature of speed. In fact, experiments of the authors show that such synchronizing is faster than the process of tracking the weights of one of the networks by an eavesdropper. It must be said that the inventors weren't able to find a mathematical proof of this, that instead was published little later by other cryptographers in Klimov, Mityagin and Shamir (2002). In this same work, all of the limitations of *KKK* are also shown, but we'll deal with this in the following subsections (Synchronization: 3.3.2; Attacks: 3.3.3).
Going back to the model, the architecture used by both sender and recipient is a tree parity machine, a two-layered perceptron with K hidden units, $K \times N$ input neurons and a single output. Input units take binary values $x_{kj} = \pm 1$, $k = 1, \dots, K$ and $j = 1, \dots, N$. The K binary hidden units are denoted by y_k, $k = 1, \dots, K$, while the integer weight from the j-th input unit from the k-th hidden unit is denoted $w_{kj} \in \{-L, \dots, L\}$. Output O is the product of the state of the hidden neurons. (ADD FIGURE!!!)
Fix for simplicity $K = 3$ and let w^S_{kj}, w^R_{kj} be the secret information of sender and recipient, respectively (that is, the initial values for the weights). Hence this consists of $3N$ integer numbers for each of the two participants. Note that, from now on, superscript S will denote sender, while R will denote recipient.
Each network is then trained with the output of its partner. At each step,

both for synchronization and for encryption/decryption steps, a new common public input vector is needed. Given $\vec{x}_{kj}$, output is computed in two steps. In the first one, states of hidden units are computed as

$$y_k^{S/R} = \text{sign}\left(\sum_{j=1}^{N} w_{kj}^{S/R} x_{kj}\right), \tag{3.3}$$

with the convention $y_k^S = 1$ and $y_k^R = -1$ whenever argument of the sign function is zero. In second step, output is computed as the product of the hidden units:

$$O^{S/R} = \prod_{k=1}^{3} y_k^{S/R}. \tag{3.4}$$

Sender and recipient send their outputs to each other and in case they do not agree on them (if $O^S O^R < 0$), weight are updated according to the following Hebbian rule:

$$\begin{aligned} \text{if} \quad \left(O^{S/R} y_k^{S/R} > 0\right) \quad \text{then} \quad w_{kj}^{S/R} \leftarrow w_{kj}^{S/R} - O^{S/R} x_{kj}; \\ \text{if} \quad \left(|w_{kj}^{S/R}| > L\right) \quad \text{then} \quad w_{kj}^{S/R} \leftarrow \text{sign}\left(w_{kj}^{S/R}\right) L. \end{aligned} \tag{3.5}$$

Note that this algorithm only updates weights belonging to the hidden units which are in the same state as that of their output unit.
Synchronizing time depends on the choice of the parameters, but this learning rule implies that, as soon as the two networks are synchronized, so they stay forever. Moreover, it was already clear for these experimenter that those times were relatively short compared to the task of externally intercepting weights of one of the two participants (using the same net structure). As soon as the weights are antiparallel, the initialization of the cryptosystem is completed and the secure communication may start. Here we have two possibilities: either use a conventional algorithm, for example a stream cipher, or use the parity machine itself. In the first case, we can build the seed for a pseudo-random number generator basing on the weight vector after synchronization. In the other case, we directly use the output bit of the net for a stream cipher. Note that, using this approach, the complexity of encryption/decryption is linear.

3.3.2 More details on Synchronization

In Kanter's work, many experiments are reported to show the effectiveness of synchronization process. However, I decided to omit these details since,

as I anticipated in previous section, the "sacred cow" of cryptography Adi Shamir provided us with a mathematical proof of it.
The full treatment of this problem found in Klimov, Mityagin and Shamir (2002) relies on the properties of random walks in bounded domains. Another assumption: learning rule (3.5) would complicate the notation, as it forces the mutually learning NN's into anti-parallel states. Instead, a dual scheme in which the two parties eventually become identical is proposed. Given input vectors $\vec{x}_{kj}$, hidden outputs are calculated as

$$y_k^{S/R} = \operatorname{sign}\left(\sum_{j=1}^{N} w_{kj}^{S/R} x_{kj}\right), \tag{3.6}$$

but with the standard convention

$$\operatorname{sign}(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{3.7}$$

Final output is computed in the same way as before, that is $O^{S/R} = \Pi_{k=1}^{3} y_k^{S/R}$. In this version of the algorithm, after the output exchange, each party updates its weights only if $O^S = O^R = O$ and in this case

$$\text{if}\left(y_k^{S/R} = O\right) \quad \text{then} \quad w_{kj}^{S/R} \leftarrow B_{-L,L}\left(w_{kj}^{S/R} - y_k^{S/R} x_{kj}\right), \tag{3.8}$$

where

$$B_{-L,L}(w) = \begin{cases} w & \text{if} \quad |w| < L \\ L & \text{if} \quad w > L \\ -L & \text{otherwise.} \end{cases} \tag{3.9}$$

In other words, above function rescales weights to the prefixed bounds whenever any of them exceeds the allowed values.
After these premises, we can show how the two participants converge and why a third net cannot converge to the same parameters by following the same learning procedure.
We start off by considering an oversimplified model of a single Perceptron with a single weight. Let $w_t^{S/R}$ be the current weights and denote inputs $x_t \in \{-1,1\}$. The update process of each weight can be described as a random walk with absorbing boundaries $-L, L$, starting from a random point $w_0^{S/R} \in \{-L, \ldots, L\}$.
At each round t, sender and receiver decide either to move their respective weights in the same direction (determined by x_t), or not to vary them. In

first case, if any of $w_t^{S/R}$ tries to step beyond the fixed boundaries, it remains stuck at it (due to (3.9)), while the other one gets nearer. If none of the the weights is absorbed, we have $|w_{t+1}^S - w_{t+1}^R| = |w_t^S - w_t^R|$, else their distance is reduced by one. Since a random walk is expected to hit its boundaries infinitely often, the paths of the weights admit a mixing time.
A simple generalization: consider the case of a single perceptron with multiple weights. Here, the two weight vectors move in the same direction determined by inputs in a multidimensional rectangle, and along each coordinate the distance is either preserved or reduced by one. When all these distances are reduced to zero, the two random walks mix.
The general case of neural networks with multiple Perceptrons is more complicated, since the two parties may update different subsets of their perceptrons in each round. The complete treatment of such problem can be found in Klimov, Mityagin and Shamir (2002) but I decided to omit these details.

3.3.3 Attacking KKK

The creators of *KKK* claimed its security basing on their experiments. In fact, simulations shown clearly that it was computationally unfeasible for an attacker to intercept any of the parameters related to the two parties, if such eventual attacker relied on the same neural network structure as the ones used in communication.
In Shamir's paper there is also a mathematical proof of this impossibility, but most interesting content of such work is in its last section. This is dedicated to the cryptanalysis of *KKK*, which can indeed be broken by using other types of attack.
For example, many successful cryptanalytic applications of Genetic Algorithms can be found in literature.
In this case, a large population of NN's with the same structure as the two participants is simulated. These nets are then trained with the same inputs of the two communicant ones. Networks whose outputs mimic those of the two parties breed and multiply, while unsuccessful networks die.
Shortly after sender S and recipient R synchronize, they can be sure of this fact (since their outputs keep coinciding) and the same can be checked for any of the attacker's neural networks.
An interesting experimental result: in the majority of tests led by the authors, at least one of the attacking nets became synchronized with S even before S and R became fully synchronized.

Other examples of successful attacks to the *KKK* protocol are the one based on geometrical reasoning and the probabilistic case. The former exploits the geometrical representation of inputs and weight in an algorithm that guesses hidden outputs in the two parties'nets. The latter, instead, is again based on the properties of generalized random walks: these allow us to compute conditional probabilities for the values of hidden states.

Chapter 4

Neural Cryptanalysis

Not only neural networks reveal themselves capable of learning how to communicate securely. In fact, recent studies show that NN's can be employed to perform efficient cryptanalysis over lightweight ciphers. The present chapter is dedicated to discussing an example of such studies. Before this, it is important to recall a previous study of interest, that is Alani (2012). In this work, Neural Networks, together with the Known Plaintext Attack (KPA) methodology, are efficiently employed to break Data Encryption Standard. A similar combination, that is NN's together with the Chosen Plaintext Attack (CPA), will be later used to build extremely strong cryptosystems (see Subsection 5.2.1).

4.1 Cryptanalysis of Simon Cipher using Neural Networks

The paper I am considering in this section is Jayachandiran (2018), in which the author describes a novel, complete approach of using neural networks in cryptanalysis. The cryptosystem of interest belongs to the family of Simon block ciphers, publicly released by NSA in 2013. (ADD CIT.)

4.1.1 Simon Cipher

Simon cipher can be consider a lightweight cipher; as such it generally finds its use in devices that have very restricted hardware resources. Hence we remark it cannot be considered a top-security algorithm, but it is good for research purpose; that is, experimenting the capabilities of neural networks

in cryptanalysis.
Simon has has input block size of length 32, with a key size of 64 bits in length. In this study, Simon cipher is round-reduced: it encrypts the plaintexts with first a single round of the Feistel network and then with two rounds; same is for decryption. (ADD PICTURE)
More precisely, the 32 bits long plaintext x is divided into two blocks of 16 bits in length, namely x_i and x_{i+1}, respectively. The former half is made to go through several left circular shift units, with S^j, $j = 1,2,8$, representing shift by one, two and eight bits respectively. A bitwise AND is performed on the results of S^1 and S^8, followed by a bitwise XOR with the second half x_{i+1}. The result of previous operation is again XOR-ed with the result from S^2.
In general, Simon creates a list of key words $k_0, k_1, \ldots, k_T$, where T is the number of rounds. Since in this case the algorithms is simplified to one round and two rounds of the cipher, the key schedules generate only one and two key words respectively. The corresponding key word from the key schedule is then XOR-ed with the result of the last performed operation. In the final step of the round, the result from the last step is swapped with the input block x_{i+1} and then passed on to the next round.

4.1.2 Neural Network Design

The architecture of choice is the multi-layer perceptron, with some extensions that we discuss in the following.
Each neuron in the input layer corresponds to each bit of the plaintext and ciphertext pairs; that is 64 input neurons. Same size for the output layer, in which each cell predicts whether the key bit is 0 or 1. The number of neurons in hidden layers, instead, varies up to 1024. According to the author of Jayachandiran (2018), this decision helped increase the accuracy of the model. For the same reason, all layers are fully-connected (FC).
A few words about the activation function. It takes into account a bias assigned to the neuron and it's known in literature as Linear Rectifier, that is

$$y = f(z) = z + b = \sum_j w_j x_j + b, \tag{4.1}$$

where the x_j's are the inputs with the corresponding weights w_j; and b denotes the bias. Weights of the neurons are initialized randomly according to a uniform distribution over the interval (0,1) and are updated after each iteration; same for the biases.

4.1.3 Data and Results

In the experiment described in Jayachandiran (2018), training data consists in a large set of records, each of which is a 32-bit, randomly generated plaintext, with the corresponding 32-bit long ciphertext. Around 5 million records were generated with 1000 keys (where each key encrypted 5000 plaintexts).
The neural nets describes in previous section are implemented using *Keras*, configured to work on top of *Tensorflow*. The project required the installation of the GPU versions of such softwares. In fact, not being run on a Cloud, the training of such networks clearly appears to be computationally expensive.
(ADD TABLE WITH SAMPLE TRAINING DATA).

The trained neural network model is then used to predict the key of the plaintext-ciphertext pairs. The accuracy of the network is measured by identifying the number of bits predicted by the neural network that are the same as the key bits in the original key. Better results were attained when considering a single round of the Speck cipher. In this case, after a considerable number of training epochs, an accuracy around 70% was reached.
A powerful extension of this project could be a design based on fuzzy classifiers that could be used along with the neural networks to yield probabilities for zeros and ones, rather than hard predictions that this model does.
For example, another sensible improvement in using NN's to attack this kind of ciphers can be found in (Gohr, 2019). In this paper, an efficient Chosen-Plaintext-Attack to Speck (the hardware version of Simon) is implemented by making use of convolutional neural networks. In this case, the attack reveals to be effective even against ciphertext created eight or nine rounds of the cipher. (ADD MORE DETAILS? GOHR'S WORK IS VERY COMPLICATED).

Chapter 5

Adversarial Neural Cryptography

After a long period without any substantial contribution to neural cryptography, a new, revolutionary approach made its appearance in 2016, thanks to Google's researchers. That is, neural networks can learn to protect the secrecy of their data from other neural networks: they discover forms of encryption and decryption, without being taught specific algorithms for these purposes. This new approach to cryptography has been named ANC (Adversarial Neural Cryptography).

5.1 Google's Model

The research I'm considering is Abadi and Andersen (2016). In this paper, a new cryptosystem is proposed. At its core there are adversarial neural networks and in this section I'm examining this protocol.

5.1.1 Cryptosystem Organization

Let's start with some notation. In this scenario, we consider the problem of the secure communication between Alice and Bob, denote them $\mathcal{A}$ and $\mathcal{B}$, respectively. Consider also a third participant, Eve ($\mathcal{E}$)), who wishes to eavesdrop on their communication. Such adversary is passive, that means it can only intercept communications.
In this scenario, Alice wishes to send a private message P ("Plaintext") to Bob. Such message must be regarded as an input to $\mathcal{A}$, together with a key K. This key is also known by $\mathcal{B}$, that uses it to decipher $\mathcal{A}$'s output; denote it C ("Ciphertext"). Also Eve knows C and tries to recover P from it, but

E cannot know K. Hence let P_{Bob} and P_{Eve} be the respective outputs.
Alice, Bob, and Eve are represented by, guess what, neural networks, based on a "Mix and Trasform" architecture. This structure has a first fully-connected (FC) layer, where the number of outputs is equal to the number of inputs. P and K are fed into this layer, that enables mixing between the bits of these two vectors. Such layer is followed by a sequence of convolutional layers, the last of which produces an output of a size suitable for a plaintext or ciphertext. These neural networks have parameters, which we write $\theta_{\mathcal{A}}$, $\theta_{\mathcal{B}}$ and $\theta_{\mathcal{E}}$. Moreover, P, C, K, P_{Bob} and P_{Eve} are vectors of float numbers; in particular values are allowed to range in $(-1, 1)$.
Eve's objective is to reconstruct P, that is, to minimize the distance between P and P_{Eve}. Alice and Bob want to communicate clearly (to minimize the error between P and P_{Bob}), but also to hide their communication from $\mathcal{E}$. $\mathcal{A}$ and $\mathcal{B}$ are hence trained jointly to communicate securely and to defeat Eve: here lies the biggest innovation in ANC.
Since we want Alice and Bob to be trained against the best possible version of Eve, we need to assume a probability distribution on plaintexts and keys. After that, we can rephrase the parties' objectives in terms of expectation. Denote Alice and Bob's output as $A(\theta_{\mathcal{A}}, P, K)$ and $B(\theta_{\mathcal{B}}, C, K)$, respectively. Similarly, write $E(\theta_{\mathcal{E}}, C)$ for Eve's output. As anticipated above, we need to choose a distance function on the space of plaintexts. In this model, assuming such P's have length N, we take

$$d(P, P') = \sum_{i=0}^{N} |P_i - P'_i|, \tag{5.1}$$

that is, the L^1 distance. Note that this choice is not crucial. After defining an example-specific loss function for Eve,

$$L_{\mathcal{E}}\left(\theta_{\mathcal{A}}, \theta_{\mathcal{E}}, P, K\right) = d\left(P, E(\theta_{\mathcal{E}}, A(\theta_{\mathcal{A}}, P, K))\right),$$

we can define the "true" loss function for $\mathcal{E}$ by taking expectations over the distributions of the P's and K's:

$$L_{\mathcal{E}}\left(\theta_{\mathcal{A}}, \theta_{\mathcal{E}}\right) = \mathbb{E}_{P,K}\left[d\left(P, E(\theta_{\mathcal{E}}, A(\theta_{\mathcal{A}}, P, K))\right)\right]. \tag{5.2}$$

The optimal configuration for Eve, $O_{\mathcal{E}}$, can hence be found by minimizing the above loss (given a configuration of $\mathcal{A}$) .

$$O_{\mathcal{E}}\left(\theta_{\mathcal{A}}\right) = \text{argmin}_{\theta_{\mathcal{E}}} L_{\mathcal{E}}\left(\theta_{\mathcal{A}}, \theta_{\mathcal{E}}\right) \tag{5.3}$$

Similarly, a per-example loss for Bob is defined and then extended to the distributions of plaintexts and keys:

$$\begin{aligned} L_{\mathcal{B}}\left(\theta_{\mathcal{A}}, \theta_{\mathcal{B}}, P, K\right) &= d\left(P, B(\theta_{\mathcal{B}}, A(\theta_{\mathcal{A}}, P, K), K)\right) \\ L_{\mathcal{B}}\left(\theta_{\mathcal{A}}, \theta_{\mathcal{B}}\right) &= \mathbb{E}_{P,K}\left[d\left(P, B(\theta_{\mathcal{B}}, A(\theta_{\mathcal{A}}, P, K), K)\right)\right]. \end{aligned} \tag{5.4}$$

We now need to define a joint loss function for Alice and Bob, which takes into account $\mathcal{B}$'s loss and the optimal value of Eve's loss (by maximizing $\mathcal{E}$'s reconstruction error). This is:

$$L_{\mathcal{AB}}(\theta_{\mathcal{A}}, \theta_{\mathcal{B}}) = L_{\mathcal{B}}\left(\theta_{\mathcal{A}}, \theta_{\mathcal{B}}\right) - L_{\mathcal{E}}(\theta_{\mathcal{A}}, O_{\mathcal{E}}\left(\theta_{\mathcal{A}}\right)). \tag{5.5}$$

Note that also this choice may be subject to variations. Optimal configurations for Alice and Bob can hence by found as

$$(O_{\mathcal{A}}, O_{\mathcal{B}}) = \text{argmin}_{(\theta_{\mathcal{A}}, \theta_{\mathcal{B}})} L_{\mathcal{AB}}(\theta_{\mathcal{A}}, \theta_{\mathcal{B}}) \tag{5.6}$$

It must be said that such optimal configurations are not necessarily unique. A few words about training. Such procedure is based upon stochastic gradient descent and on estimated values calculated over examples (not on expected values over a distribution). In addition, $O_{\mathcal{E}}$ is not computed for given values of $\theta_{\mathcal{A}}$, but simply approximated. This is done by alternating the training of Eve with that of Alice and Bob.
Roughly speaking, training proceeds follows. At first, $\mathcal{A}$'s outputs may be totally incomprehensible for both $\mathcal{B}$ and $\mathcal{E}$. After a few steps, Bob could discover a way to decrypt Alice's messages at least partially, without Eve being able to do the same. Later, however, $\mathcal{E}$ may start to break this code. Alice and Bob should then find improvements to their security, in a way such that Eve would find impossible to adjust to those ciphers. Note that this kind of alternation is typical in game theory.

5.1.2 ANC and selective protection

Authors of ANC described an experiment to study to test this cryptosystem over selective protection: the question of whether neural networks can learn what information to protect. For example, a plaintext may be made up of several components, of which only few of them are required to be kept secret to the adversary. In this case, selective protection would mean a maximization in utility.
Consider a dataset consisting of vectors of for values, namely (A, B, C, D). The target is to build and train a system that, given as inputs the first three

values, outputs two predictions of D: $\hat{D}$ and D_{public}. The former is most accurate possible estimate, while the latter is defined as the best possible estimate of D that does not reveal any information about the value of C.
As before, Alice and Bob share a key and both $\mathcal{B}$ and Eve have access to $\mathcal{A}$'s outputs, that are D_{public} and a ciphertext T. Her input is (A, B, C). Bob produces $\hat{D}$, while Eve tries to recover C. Using strongly correlated values for (A, B, C) and D, the authors were able to show that the adversarial training permits approximating D without revealing C.

5.2 Improvements: CPA-ANC

This section is dedicated to an extension of Abadi and Andersen's model. In fact, also this last model has some limitations, that are shown in Coutinho et al. (2018). Authors of this paper elaborated an extension to ANC, that is discusses in the following.

5.2.1 Chosen-Plaintext Attack ANC

Main limitation to the security of original ANC model lies in Eve's job, that appears to be too hard. It must decrypt a random message having access only to the ciphertext. The consequence is that, under this methodology, Alice and Bob learn to protect against a weak adversary. It is however possible to strengthen Eve by letting it mount a CPA.
Eve chooses two plaintext messages, namely P_0 and P_1, and sends them to Alice. $\mathcal{A}$ chooses one of the messages randomly, encrypts it to C and sends it to $\mathcal{B}$ and $\mathcal{E}$. The former decrypts the message using a NN, while the latter only outputs 0 if it believes P_0 was encrypted or 1 if it believes P_1 was encrypted.

5.2.2 Neural Network Architecture

A specific network architecture for this model is proposed, based on the continuous extension of logic operator XOR, built to learn an OTP algorithm.
Consider mapping bit 0 to angle 0 and bit 1 to angle π (XOR can be seen as the sum of the angles). Generalizing bits to a continuous space, the following defines the mapping of a bit b into an angle:

$$f(b) = \arccos(1 - 2b) \tag{5.7}$$

Then consider its inverse:

$$f^{-1}(a) = \frac{1 - \cos(a)}{2} \tag{5.8}$$

The new net, named *CryptoNet*, takes as input P and K and, for each bit, applies (5.7). Next step is a weight matrix (write W) multiplication, followed by the inverse transformation (5.8); output is C. Note bits $C_i \in (0,1)$. *CryptoNet* will be denote mathematically as the function

$$C = \zeta_n(W, P, K), \tag{5.9}$$

where we recall n is the size of vectors P and K.

5.2.3 Method

Let $E_{\mathcal{A}}(\cdot, K)$ be Alice's encryption function and let $D_{\mathcal{B}}(\cdot, K)$ be Bob's decryption function. In this setup, these maps are defined to be the same *CryptoNet*:

$$E_{\mathcal{A}}(\cdot, K) = D_{\mathcal{B}}(\cdot, K) = \zeta_n(W, \cdot, K). \tag{5.10}$$

This means, such network must be the inverse of itself. More details on the training procedure can be found in the original paper.

Chapter 6

Chapter title

6.1 Another section

Tex of the section with example of theorem

Theorem 6.1.1. *Example of theorem*

Proof. proof of the theorem □

to be referenced like Theorem 6.1.1.
Same for proposition

Proposition 6.1.2. *Example of proposition*

or lemma

Definition 6.1.3. Example of definition □

Remark 6.1.4. Example of remark □

Lemma 6.1.5. *Example of lemma*

Appendix A

Finite Differences Methods

In numerical analysis, Finite Difference Methods (FDMs) are discretizations used for solving differential equations by approximating them with difference equations that finite-difference approximate the derivatives of interest. In the first chapter of this work, some algorithms for neural network training are presented which rely on the solution of partial differential equations (PDEs).
In the present appendix we discuss the theoretical bases of finite difference methods applied to PDEs. The results presented here are taken from the second chapter of Grossmann, Roos and Stynes (2005), where a much wider treatment of these methods can be found.

A.1 An introductory example

When a finite difference method (FDM) is used to treat numerically a partial differential equation, the differentiable solution is approximated by some grid function, in example, by a function that is defined only at a finite number of grid points that lie in the underlying domain and its boundary. Each derivative that appears in the partial differential equation has to be replaced by a suitable divided difference of function values at the chosen grid points. Such approximations of derivatives by difference formulas can be generated in various ways; in case they are generated by a Taylor expansions, the approach is generally known in literature as finite difference method.
As an introduction to FDMs, consider the following example. We are interested in computing an approximation to a sufficiently smooth function u that for given f satisfies Poisson's equation in the unit square and vanishes

on its boundary:

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega := (0,1)^2 \subset \mathbb{R}^2, \\ u &= 0 \quad \text{on} \quad \Gamma := \partial\Omega. \end{aligned} \tag{A.1}$$

FDMs provide values $u_{i,j}$ that approximate the desired function values $U(x_{i,j})$ at the grid points $\{x_{i,j}\}$. Let the grid points in our example be

$$x_{i,j} = (ih, jh)^\intercal \in \mathbb{R}^2, \quad i,j \in \{0,1,\ldots,N\}. \tag{A.2}$$

Here $h = \frac{1}{N}$, $N \in \mathbb{N}$, is the mesh size of the grid.
At grid points lying on the boundary Γ the given function values (which here are homogeneous) can be immediately taken as the point values of the grid functions. All derivatives in problem (A.1) have however to be approximated by difference quotients. From Taylor's theorem we obtain

$$\frac{\partial^2 u}{\partial x_1^2}(x_{i,j}) \approx \frac{1}{h^2}\left(u(x_{i-1,j}) - 2u(x_{i,j}) + u(x_{i+1,j})\right), \tag{A.3}$$

$$\frac{\partial^2 u}{\partial x_2^2}(x_{i,j}) \approx \frac{1}{h^2}\left(u(x_{i,j-1}) - 2u(x_{i,j}) + u(x_{i,j+1})\right). \tag{A.4}$$

If these formulas are used to replace the partial derivatives at the inner grid points and the given boundary values are taken into account, then an approximate description of the original boundary value problem (A.1) is given by the system of linear equations

$$\begin{aligned} &4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f(x_{i,j}), \\ &u_{0,j} = u_{N,j} = u_{i,0} = u_{i,N} = 0; \quad i,j \in \{1,\ldots,N-1\}. \end{aligned} \tag{A.5}$$

For any $N \in \mathbb{N}$ this linear system has a unique solution $u_{i,j}$ Under certain smoothness assumptions on the desired solution u of the original problem one has $u_{i,j} \approx u(x_{i,j})$.

A.2 Method of Finite Differences applied to PDEs: Basic Concepts

Let us consider the very simple domain $\Omega := (0,1)^n \subset \mathbb{R}^n$. Denote its closure by $\overline{\Omega}$. For the discretization of $\overline{\Omega}$ a set $\overline{\Omega}_h$ of grid points has to be selected, e.g., we may chose an equidistant grid that is defined by the points of intersection obtained when one translates the coordinate axes through

consecutive equidistant steps with step size $h = \frac{1}{N}$. Here $N \in \mathbb{N}$ denotes the number of shifted grid lines in each coordinate direction.
We distinguish between those grid points lying in the domain Ω and those at the boundary Γ by setting

$$\Omega_h := \overline{\Omega}_h \cap \Omega \quad \text{and} \quad \Gamma_h := \overline{\Omega}_h \cap \Gamma. \tag{A.6}$$

Unlike the continuous problem, whose solution u is defined on all of $\overline{\Omega}$, the discretization leads to a discrete solution

$$u_h : \overline{\Omega}_h \longrightarrow \mathbb{R} \tag{A.7}$$

that is defined only at a finite number of grid points. Such mappings are called grid functions.
To deal properly with grid functions we introduce the following function spaces:

$$U_h = \{u_h : \overline{\Omega}_h \longrightarrow \mathbb{R}\}, \tag{A.8}$$

$$U_h^0 = \{u_h \in U_h : u_h|_{\Gamma_h} = 0\}, \tag{A.9}$$

$$V_h = \{v_h : \Omega_h \longrightarrow \mathbb{R}\}. \tag{A.10}$$

To shorten the writing of formulas for difference quotients, let us define the following difference operators where the discretization step size is $h > 0$:

$$(D_j^+ u)(x) := \frac{1}{h}\Big(u(x + he^j) - u(x)\Big); \quad \text{(forward difference quotient)}$$

$$(D_j^- u)(x) := \frac{1}{h}\Big(u(x) - u(x - he^j)\Big); \quad \text{(backward difference quotient)}$$

$$D_j^0 := \frac{1}{2}\Big(D_j^+ + D_j^-\Big); \quad \text{(central difference quotient).}$$

In the above formulas e^j denotes the unit vector in the positive direction of the j-th coordinate axis.
How can difference approximations be generated? For sufficiently smooth functions $u : \mathbb{R}^n \longrightarrow \mathbb{R}$, by a Taylor expansion one has

$$u(x + z) = \sum_{k=0}^{m} \frac{1}{k!} \left(\sum_{j=1}^{n} z_j \frac{\partial}{\partial x_j} \right)^k u(x) + R_m(x, z). \tag{A.11}$$

Here the remainder $R_m(x, z)$ can be written in the Lagrange form, namely

$$R_m(x, z) = \frac{1}{(m+1)!} \left(\sum_{j=1}^{n} z_j \frac{\partial}{\partial x_j} \right)^{(m+1)} \times u(x + \theta z); \quad \exists \theta = \theta(x, z) \in (0,1). \tag{A.12}$$

If derivatives are replaced by approximating difference formulas that are derived from (A.11), one can deduce estimates for the consequent error. Further detail can be found in Grossmann, Roos and Stynes (2005).
Going back to difference approximation of derivatives, Taylor theorem provides a systematic tool for the generation of such approximations, for example one could show

$$\frac{\partial u}{\partial x_j}(x) = \frac{1}{2h}\left(-3u(x) + 4u(x + 4u(x + he^j) - u(x + 2he^j)\right) + O(h^2). \tag{A.13}$$

In general, any given partial differential equation problem, including its boundary and/or initial conditions, can be expressed as an abstract operator equation

$$Fu = f, \tag{A.14}$$

with appropriately chosen function spaces U and V, a mapping $F : U \longrightarrow V$ and $f \in V$. The related discrete problem can be stated analogously as

$$F_h u_h = f_h, \tag{A.15}$$

with $F_h : U_h \longrightarrow V_h$, $f_h \in V_h$, and discrete spaces U_h, V_h.

Bibliography

Ethier, S.N. and Kurtz, T.G. (1981). The infinitely-many-neutral-alleles diffusion model. *Adv. Appl. Probab.* **13**, 429–452.

Ethier, S.N. and Kurtz, T.G. (1986). *Markov processes: characterization and convergence*. Wiley, New York.

Graupe, D. (2007). *Principles of Artificial Neural Networks (2nd Edition)*. Word Scientific Publishing, Singapore.

Anthony, M. (2001). *Discrete Mathematics of Neural Networks*. Society for Industrial and Applied Mathematics, Philadelphia.

McCulloch, W. and Pitts, W. (1943). *A logical calculus of the ideas immanent in nervous activity*. Bulletin of Mathematical Biophysics 5, 115-133.

Hebb, D. O. (1949). *The organization of behavior; a neuropsychological theory*. Wiley, New York.

Rosenblatt, F. (1958). *The perceptron: A probabilistic model for information storage and organization in the brain*. Psychological Review, 65.

Widrow, B. and Hoff, M. E. (1960). *Adaptive Switching Circuits*. IRE WESCON Convention Record, 96-104.

MInsky, M. and Papert, S. A. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

Rumelhart, D., Hinton, G. and Williams, R. (1986). *Learning representations by back-propagating errors*. Nature 323, 533–536.

Sejnowski, T. J. and Rosenberg, C. R. (1987). *Parallel Networks that Learn to Pronounce English Text*. Complex Systems, 1.

Kingma, D.P. and Lei Ba, J. (2015). *Adam: a method for stochatic optimization.* CoRR.

Mitchell, M. (1998). *An introduction to Genetic Algorithms.* MIT Press.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems.* MIT Press.

Volnà, E. (2000). *Using Neural Network in Cryptography.* University of Ostrava.

Volnà, E. (1998).*Learning algorithm which learns both architectures and weights of feedforward neural networks.* Neural Network World. Int. Journal on Neural and Mass-Parallel Compo and Inf. Systems.

Kanter, I., Kinzel, W. and Kanter, E. (2001). *Secure exchange of information by synchronization of neural networks.* Bar Ilan University.

Klimov, A., Mityagin, A. and Shamir, A. (2002). *Analysis of Neural Cryptography.* Weizmann Institute.

Abadi, M. and Andersen, D. G. (2016). *Learning to protect communications with Adversarial Neural Cryptography.* Google Brain.

Coutinho, M., Robson de Oliveira Albuquerque, R., Borges, F. , Villalba, L. J. G. and Kim T. H. (2018). *Learning Perfectly Secure Cryptography to Protect Communications with Adversarial Neural Cryptography.* University of Brasília.

Jayachandiran, K. (2018). *A Machine Learning Approach for Cryptanalysis.* Rochester Institute of Technology.

Alani, M. M. (2012). *Neuro-cryptanalysis of DES.* World Congress on Internet Security (WorldCIS-2012)

Gohr, A. (2019). *Improving Attacks on Round-Reduced Speck32/64 Using Deep Learning.* Bundesamt für Sicherheit in der Informationstechnik (BSI).

Nielsen, M. A. and Chuang, I. L. (2000). *Quantum Computation and Quantum Information.* Cambridge University Press.

Bernstein, D. J., Buchmann, J. and Dahmen, E. (2009). *Post-Quantum Cryptography.* Springer.

Shi, J., Chen, S., Lu, Y., Feng, Y., Shi, R., Yang, Y. and Li, J. (2020). *An Approach to Cryptography Based on Continuous-Variable Quantum Neural Network*. Nature.

Grossmann, C., Roos H. and Stynes, M. (2005). *Numerical Treatment of Partial Differential Equations (3rd Ed.)*. Springer.