ΑΛΕΞΑΝΔΡΕΙΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΕΥΦΥΕΙΣ ΤΕΧΝΟΛΟΓΙΕΣ ΔΙΑΔΙΚΤΥΟΥ – WEB INTELLIGENCE

Power Side Channel Execution Monitoring using Convolution Neural Networks

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΙΩΑΝΝΗ ΧΡΙΣΤΟΥΔΗ

Επιβλέπων : **Κωνσταντίνος Διαμαντάρας**
Καθηγητής, ΑΤΕΙ Θεσσαλονίκη

Θεσσαλονίκη, Ιούνιος 2019

Η σελίδα αυτή είναι σκόπιμα λευκή.

ΑΛΕΞΑΝΔΡΕΙΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΘΕΣΣΑΛΟΝΙΚΗΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΕΥΦΥΕΙΣ ΤΕΧΝΟΛΟΓΙΕΣ ΔΙΑΔΙΚΤΥΟΥ – WEB INTELLIGENCE

Power Side Channel Execution Monitoring using Convolution Neural Networks

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΙΩΑΝΝΗ ΧΡΙΣΤΟΥΔΗ

Επιβλέπων : Κωνσταντίνος Διαμαντάρας
ΑΤΕΙΘ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις .

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Όνομα Επώνυμο	Όνομα Επώνυμο	'Όνομα Επώνυμο
Α.Τ.Ε.Ι.Θ.	Α.Τ.Ε.Ι.Θ.	Α.Τ.Ε.Ι.Θ.

Θεσσαλονίκη,

(Υπογραφή)

....

Ιωάννης Χριστούδης

Μηχανικός Πληροφορικής ΑΤΕΙΘ

© – Allrightsreserved

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέπων καθηγητή του ΑΤΕΙΘ Κωνσταντίνο Διαμαντάρα για την πολύτιμη βοήθειά του καθόλη τη διάρκεια της εκπόνησης της διπλωματικής μου εργασίας, την εμπιστοσύνη του στο πρόσωπό μου, αλλά και την υπομονή του. Οι ιδέες του είναι εξαιρετικές και άξιες προσοχής και εκτίμησης. Επίσης θα ήθελα να τον ευχαριστήσω για τη δυνατότητα που μου έδωσε να ζήσω μια μοναδική εμπειρία, εκπονώντας το ερευνητικό κομμάτι της διπλωματικής στο πανεπιστήμιο Rutgers στο New Jersey. Θα ήθελα να ευχαριστήσω θερμά την καθηγήτρια Αθηνά Πετροπούλου, του Rutgers πανεπιστημίου για την υποδοχή της και τη πολύτιμη συνεργασία μας. Επίσης θα ήθελα να ευχαριστήσω τη διοίκηση του ΑΤΕΙ και όσους ενεπλάκησαν προς τη μετακίνησή μου στην Αμερική. Τέλος θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα (PhD student) Yi Han του πανεπιστημίου Rutgers για την άψογη συνεργασία μας καθολη τη διάρκεια διαμονής μου εκεί, καθώς και για τη φιλία που κέρδισα.

Περίληψη

Οι ενσωματωμένες συσκευές, όπως οι προγραμματιζόμενοι λογικοί ελεγκτές (PLC) και συσκευές Internet-of Things (IoT), γίνονται στόχοι επιθέσεων κακόβουλου λογισμικού με αυξανόμενη συχνότητα και καταστροφικά αποτελέσματα. Η ανάλυση φυσικού πλευρικού καναλιού (SCA) είναι ένας τρόπος παρακολούθησης της συσκευής χωρίς πρόσβαση στο λογισμικό της, γεγονός που δεν προκαλεί επιβάρυνση πόρων στη συσκευή. Στην παρούσα εργασία παρουσιάσαμε έναν εναλλακτικό τρόπο χρήσης ανάλυσης πλευρικών καναλιών για την ανίχνευση ανωμαλιών σε ενσωματωμένες συσκευές κατά την εκτέλεση κώδικα. Χρησιμοποιήσαμε σήματα πλευρικού καναλιού κατανάλωσης ισχύος για να σχεδιάσουμε ένα σύστημα ανίχνευσης εισβολής βασισμένο σε συνελικτικά νευρωνικά δίκτυα (CNN). Χρησιμοποιήσαμε τον κατάλληλο εξοπλισμό για τη λήψη σημάτων που αντιπροσωπεύουν διαφορετικές διαδρομές του κώδικα εκτέλεσης. Λαμβάνοντας υπόψη αυτά τα μονοπάτια ως κλάσεις τροφοδοτήσαμε ένα συνελικτικό νευρωνικό δίκτυο που δημιουργήσαμε για να το εκπαιδεύσουμε για να προβλέψουμε πότε υπάρχει μια διείσδυση που οδηγεί σε μια ανωμαλία στην εκτέλεση κώδικα. Υπάρχει μια δημοσιευμένη σχετική εργασία σχετικά με την ανίχνευση εισβολών που βασίζεται στην ανάλυση καναλιού πλευράς EM. Στην παρούσα διπλωματική εργασία εστιάζουμε στα σήματα καναλιού ισχύος.

Λέξεις Κλειδιά: PLC, IoT, CNN, SCA, ανάλυση σήματος μέσω φυσικού καναλιού

Η σελίδα αυτή είναι σκόπιμα λευκή.

Abstract

Embedded devices, such as programmable logic controllers (PLC) and Internet-of Things (IoT) devices are becoming targets of malware attacks with increasing frequency and catastrophic results. Physical side channel analysis is one way to monitor the device without accessing its software, thus causing no resource overhead to the device. In this thesis we presented an alternative way of using side channel analysis for detecting anomalies in embedded devices during code execution. We used power consumption side channel signals to design an intrusion detection system based on convolutional neural networks. We used the proper equipment to capture signals representing different paths of the execution code. Considering these paths as classes we fed a convolutional neural network we created in order to train it to predict when there is an intrusion leading to an code execution abnormality. There is a previously published relative work on intrusion detection based on EM side channel analysis. In this thesis we focus on power side channel signals.

Keywords: Side channel analysis, Power consumption signals, Code execution monitoring, Machine learning, Deep learning, Convolutional neural networks, PLC, IoT

Η σελίδα αυτή είναι σκόπιμα λευκή.

Πίνακας περιεχομένων

Power Side Channel Execution Monitoring using Convolution Neural Networks............................1

Power Side Channel Execution Monitoring using Convolution Neural Networks............................3

1Introduction...8

1.1Code execution monitoring using side channel analysis..8

1.2Thesis topic...9

1.2.1Contribution..10

1.3Thesis structure..10

2Related Studies...12

2.1Execution monitoring using HMM..13

1.1.1Anomaly detection system based on HMM...14

1.1.2Program execution monitoring systems..14

2.2Execution monitoring using RNN...15

3Theoretical Background...17

3.1Power side channel signals..17

3.2Program analysis..18

3.3Machine learning...19

1.1.3Machine learning training...20

1.1.4Classification...21

1.1.5Artificial Neural Networks (ANNs)...22

1.1.6Deep learning...32

3.4Convolutional Neural Networks (CNNs)...38

1.1.7Architecture..39

4Power Side Channel Execution Monitoring using CNN..43

4.1Profiling Model Construction...43

1.1.8Program Analysis..43

1.1.9Signal acquisition...44

1.1.10Signal and data Pre-processing..47

4.2Execution Monitoring and Intrusion Detection...48

1.1.11Intrusion Detection..54

5 Evaluation..55

6 Conclusions..60

7 Βιβλιογραφία...61

1 Introduction

1.1 Code execution monitoring using side channel analysis

Programmable logic controllers (PLC) are embedded devices, used widely in industrial control systems (ICS), which connect and monitor critical infrastructure such as electricity grids, health-care, chemical production, oil and gas refinery [1]. Internet of Things (IoT) embedded devices can be found in various home appliance applications, including illumination control, entrance control and surveillance. The requirements for these devices in power consumption are really low and along with their small sizes and low cost make their on-chip processing capability limited.

Due to their popularity, PLC and IoT embedded devices are becoming targets for malicious attacks, leading to private information leakage [2], or even catastrophic system failures [3], [4]. A famous example of a malicious attack on a PLC is the Stuxnet malware, which, in 2010, by affecting the PLCs code damaged 20% of Iranian PLC-controlled centrifuges [5]. Other PLC attacks include Duqu [6] and Harvey [7]. Over the last few years, malware-infected IoT devices have been increasingly used for launching Distributed Denial of Service (DDoS) attacks, often without their owners knowing that their devices have been compromised. Recently, malicious actors have used botnets comprised of malware-infected IoT devices, such as Internet-connected appliances and home routers, to great effect. These devices are attractive targets for malware due to their lack of cryptographic encryption and

weak default authentication. Among attacks that have exploited IoT devices is a botnet malware family named Mirai, which brought the largest DDoS attack to date [8]. The botnets in these attacks primarily consisted of home devices such as home routers, web-cams, and printers [9], [10].

There has been an increasing body of work on embedded system security. Offline formal control logic analysis has been investigated in [11] through symbolic execution and model checking mechanisms. Solutions such as WeaselBoard [12] and CPAC [13] perform run-time PLC execution monitoring using control logic- and firmware-level reference monitor implementations. A summary of threats to IoT devices can be found in [8].

Although one could add an extra piece of code to monitor program execution of embedded devices and detect malicious attacks, such solution would impose on the device's limited I/O interfaces and constrained resources. Also, the required updates would introduce safety and cost concerns [14], [1]. Physical side channel signals have been investigated as a more desirable alternative. In [14] power consumption signals, collected during normal operation of a micro-controller were used to build a control flow model based on the hidden Markov model (HMM). The model was then used to track run-time code execution. In [15] local spectra of electromagnetic (EM) emanations of an ARM processor were used to detect deviations in program executions. EM emanations were also used in [1] for execution monitoring and intrusion detection in PLCs [1]. In particular, in [1], a long short-term memory (LSTM) neural network was trained to profile the legitimate execution status and detect anomalies when a mismatch between the EM signal and the model is encountered. Physical side channel has also been exploited in other security context due to its non-intrusive nature: attackers can eavesdrop the target without accessing its software, defenders can protect the target from a totally external system which does not cause any resource overhead. The works of [16], [17] use side channel leakage signals to extract the secret key of cryptographic implementations. There is also work about program activity (e.g. loops) profiling [18], which falls under the context of software testing.

1.2 Thesis topic

In this thesis, we demonstrate the use of power side channel signals for execution monitoring and intrusion detection on embedded devices using convolutional neural networks (CNN). CNNs are chosen here due to their ease of use in practice. CNN models better utilize the computing power of GPU [19], which significantly speeds up the training process. Moreover, due to the attention they have received from researchers and engineers, there is a lot of literature on architecture construction and tuning of parameters (e.g. initialization [20], architecture [21], [22], [23] and tuning [24]). Our system defends against control flow attacks,

launched by an adversary who uploads code to alter the control flow (execution pointer) of the program. Examples of such attacks include buffer overflow [25] and firmware modification [26], which inject malicious code into the software, and return oriented [27] and jump oriented programming [28], which use the software's own code in a different (designed) order for malicious purpose.

Our execution monitoring and intrusion detection system consists of two stages, namely, profiling and deployment. During the profiling stage, source code of the target program is analyzed to identify feasible paths and generate the program input, refered to as test cases, that will lead to those paths. The program is then exercised with these test cases while the corresponding power side channel signals are captured. These power side channel signals have unique signal patterns that represents the corresponding execution paths. The captured signals are reshaped into 2D matrices according to a predefined window length. These 2D matrices, referred to as power images contain discriminative visual features (e.g. edges, lines and shapes etc.) that are formed by the fluctuations in the power side channel signals. Our system uses these power images to construct a CNN classification model for profiling the legitimate execution status of the target program. For the deployment stage, run-time power side channel signals are captured and used to query the profiling model. This step determines whether the run-time execution path is legitimate, or it represents malicious code execution.

1.2.1 Contribution

The thesis contribution is summarized as follows

1. We captured power consumption signals from an embedded device (PLC)
2. We constructed 41 classes according to the execution paths and the power signal traces
3. We implemented a convolutional neural network based on existing ones to fulfill our purpose.
4. We conducted tests on unseen data
5. We were able to detect the intrusion
6. We discussed the challenges and future work

1.3 Thesis structure

In section 2 related studies to our work are presented. We tried to keep the number of these references limited as the literature has a lot of papers related on code execution monitoring using side channels. We focused on their approach and the nature of their challenges they had to deal with and overcome.

In section 3 a background overview is presented. Tools we used are analyzed in this section trying to provide details and references to the literature. A thorough review is done on the neural networks and the deep learning.

In section 4 the implementation of our model is presented.

In section 5 we evaluate our results.

2 Related Studies

The intrusion in embedded devices, like PLCs, is feasible by exploiting side channel signals and leakage leading to extracting sensitive data. This process is called side-channel analysis (SCA) and is really popular to cybersecurity. Power consumption, electomagnetic emanation (EM), timing information and sound can be exploited and used in a malicious way. In the research scientific community an important and significant number of studies has been conducted through the last 15-20 years. In their majority these studies are focused on crytpanalisis and cryptanalitic attacks in order to recover any secret related to their private keys, using template attacks, conventional machine learning and deep learning networks [16][29][30][31].

Detecting the intrusion is more challenging, as a consistent monitoring is needed. Signals are sensitive to noise and their monitoring is really hard to be implemented successfully. Using different techniques, research proved that this field seems to be promising resulting in positive outcomes. Hidden markov models (HMM) were till recently the most promising providing accurate results tracking code execution in normal and abnormal execution tracking [14][32]. Machine learning at the same time with its traditional algorithms (SVM) can be a really powerful tool [33]. Things changed with the rise of neural networks and especially deep learing networks giving the opportunity to the researcher to cover new fertile ground. Important studies had already been started a while ago [34] setting the neural networks to driving position in the research field. Its successors, deep learning networks are under the microscope last years and they have already started proving their leading position in intrusion detection [1][35].

It is mandatory to go through some of this important work and try to focus on the key points and concepts, as the implementation of our model, from profiling till the actual deep network , was based on these studies. The major purpose though driving this thesis' study was more intuitive and based on novelty approach. The literature lacks of monitoring CNN models in code execution.

2.1 Execution monitoring using HMM

The most popular techniques to monitor code execution in embedded devices is the use of HMM. Based on this method intrusion detection systems can be created based on execution traces. These models require a really long training time because of the traces' size.

HMM is actually based on Markov Chain. The latter one is a model which gives us information and values regarding the probabilities of states, sequences of variables. A given set of values is the set that these states are taking values from. So the Markov Chain says that the prediction of the next state of the sequence is depended on the current state. States that are coming before the current one don't have any impact on the future state, only via the current state [36]. In figure 3 the transitions between the states are assigned to a value indicating the probability for this transition to happen.

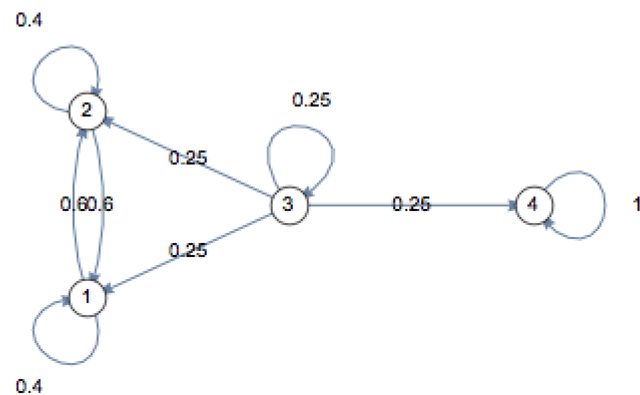


Figure 1. Markon Chain

A Markov Chain consists of **N** states, a transiiton probability matrix **A** and an initial probability over states **π**. The sum of probabilities leaving a state are equal to 1.

A HMM comes to cover the case of states, observations that we currently can't see directly. These states are called hidden. Given the above a HMM has in addition a sequence of observations O and a sequence of observation likelihoods. The latter ones are called emission probabilities.

1.1.1 Anomaly detection system based on HMM

There are four basic steps they need to be considered to implement a HMM model for detecting abnormalities during the execution [32].

- Indicating the HMM system. In this stage the number of states must be declared. The states are fully-connected, meaning that transitions are from any state can go to any state. The probabilities are randomly initialized and the Baurn-Welch (BW) algorithm is used for the training with four steps. Iterations have to be several and the training stops when the likelihood of the model producing a second set of normal traces does not improve.
- Using Viterbi algorithm the normal state transition sequence is obtained, in order to finde the sequence that mazimize the corresponding transition probability. Viterbi algorith has 4 steps to successfully obtain this sequence.
- Using Viterbi algorithm to obtain the test state transition sequence.
- The normal and the test state transition sequence are fed to the detection module.

1.1.2 Program execution monitoring systems

In [14] power consumption signals are being captured, measured and analyzed in order to track code execution of a micro-controller. To protect data processed by micro controller the system has to defend senarios like:

- key extraction attack. This can be happened by recovering the instruction flow during execution.
- Control flow graph (CFG) can be hijacked and malicious functions are implemented, such as code injection attack and firmware modification attack

Power side channel reflects the power consumption of the micro-controller during code execution because different set instruction can create different power traces. Power traces are leaking information about the instruction being executed without modifying the software or hardware.

This approach is absolutely challening as the noise can affect those traces. This noise is created by other parts of the embedded device, making difficult the extraction of useful information. Researchers found that the accuracy is affected significantly, as at some cases they achieved 60% accuracy.

Considering the above, researchers tried to overcome two major problems by answering two major questions.

- Which instruction instance in code is being executed at a given momentum

- Wheter the actual execution flow include abnormal execution deviating from the CFG

The first one is a result of a normal execution tracking while the second one a result of an abnormal execution tracking. Using HMM answers can be given to both queries. For a given power trace a recover of the most probable instruction sequence is feasible and at the same time by checking the likelihood of the reported instructions in the reported sequence can indicate an abnormal activity.

Signal acquisition was obtained using an STC89C52 micro-controller with a clock rate of 11 Mhz and a sampling rate of 1.25 GHz (more than 100 times of the clock rate) to ensure good performance. However, higher sampling rate means higher computation burden and latency. The signal is segmented according to the clock cycle, so that each segment represents one instruction. When determining the window length, there is a trade off between temporal and frequency resolution. Extracting useful features from signal segment is done by eliminating all the factors besides the intruction type from the power side channel.

The proposed HMM model is customized by the CFG of the program. The transitions between the instruction blocks in the CFG are represented as state transitions in the HMM, while emission probabilities as signal templates. The signal segments that correspond to each instruction type are fitted to a Gaussian model.

2.2 Execution monitoring using RNN

In [1] Zeus is presented, a contactless embedded controller security monitoring model. As in [14] control flow intergity is monitored during the PLC program execution by leveraging the electromagnetic emission of PLC circuit.

During the PLC code execution there is a change on the electric current in the PLC circuitry. This causes to electromagnetic signals which are captured with an antenna. Zeus is using this to capture leakage information related to the program executed on the device. These EM signal traces have unique local characteristics, depending each time on the control flow.

Because of its contactless nature, the system suffers from signal noise which can lead to bad results. Signal to Noise ratio is too low and the researcher dealt with that inspired by the speech recognition research. They are looking at frequency representation of signal segments within a local sliding window. Those segments are extracted and a power spectral desnity is computed based on their consecutive instructions.

The collected signal traces are used to train the classifier and Zues, during the runtime, collects the emanations and classify them. This results in a probability distribution computed by the model over all the classes.

The classifier is constructed based on a LSTM network. LSTM is a RNN and is covered in the next section.

In [1], since the input sequences corresponded to all possible execution paths of the target program, program control flow is implicitly embedded in the profiling model. This is different from [14] where CFG is explicitly used when constructing the profiling model. During deployment, query signals are matched with the program control flow, so that their corresponding execution paths can be predicted.

Both RNN and HMM capture signal transitions in a kind of memory. HMM models signal segments as a state machine, therefore the transition matrix which describes the probabilities of transitions between signal segments is their memory. HMM has a 1st-order dependency assumption, i.e., the transition matrix modeling is only between two adjacent input signal segments. However, capturing long-term dependency is essential in execution monitoring, since branch conditions (e.g. variables be checked in an IF statement) could be correlated to lines of code far away. RNN solves the long-term dependency problem by relaxing the first order dependency assumption. Instead of computing a transition probability matrix connecting adjacent input signal segments, RNN carries useful information through a hidden state vector. The hidden state vector can hold longer dependencies in the input signal segment sequence. For RNN the hidden state vector can be viewed as the internal 'memory', i.e. when processing each input signal segment, RNN refers to the hidden state vector for useful information of past signal segments. However, RNN suffers from the vanishing gradient problem, making it hard to train. LSTM and the gated recurrent unit (GRU) mitigate this problem by replacing the traditional linear transform & nonlinear activation combination with a set of switch gates, which helps the model deliberately 'forget' useless information and thus remember longer. Moreover, when the size of the target program gets large (i.e. large number of execution paths), more computing power and more memory are needed to construct and deploy the model. This is because a large neural network (large number of weights) is needed to hold all the paths. In some resource limited application scenarios this might be infeasible

3 Theoretical Background

Physical side channel signals, as power consumption, can reflect the execution of a set of particular instructions inside the processor of an embedded PLC, thus can be leveraged for monitoring program execution. Program analysis can be used in addition to side channel signal analysis to enable finer granularity of execution monitoring [1], [14], [15]. In this section we provide some background on how physical side channel signals are correlated with program execution, and also introduce basic concepts and useful techniques in program analysis. Basic and mandatory concepts of machine learning field are introduced. An overview of neural networks and convolutional neural networks is presented as well, tools which are used in order to classify and, eventually, detect any possible intrusion.

3.1 Power side channel signals

Electronic devices consist of circuits with a large number of CMOS components. Turning on and off these components leads to varying currents and voltages. The latter ones can be captured and measures without interacting with the device. Power consumption provides useful information for the corresponding device. Voltage fluctuations can be captured by measuring the voltage at the VCC pins of a digital chip.

In PLCs, as in every other electronic device, code execution causes the components to turn on and off giving rise to the voltage and the power consumption. Several factors affect the shape of side channel signals, namely, the type of instructions executed, the operands of each instruction, the order of executed instructions, and also interference from other components of the device and ambient noise. The effects of these factors can be understood as follows.

Execution of different instructions utilizes processor resources in different ways, and different operands imply different data transmitted and processed. Therefore, both of the aforementioned factors gives rise to unique signal patterns. Processors use a pipeline mode [38], via which the next instruction is fetched during the execution of the current instruction. The effect of this pipeline mechanism can be seen in Figure 2a, where each clock cycle contains two peaks, corresponding to the “fetch” and “execute” instructions of the AVR ATmega328p micro-controller. The pipeline mechanism causes the signal pattern of the currently executed instruction to interfere with that of the next instruction. Finally, environmental noise, such as noise from the external power supply, or from unpredictable program events (e.g., cache misses, interrupts) and interference from electronic components in the circuits also contribute to the signal pattern. Power signals tend to have very small variability between the different runs, with the minor differences caused by noise.

3.2 Program analysis

In order to accurately track program execution through a side channel signal, some information about the program structure is required. Such information can be obtained by analyzing the target program via a “static approach”, or a “dynamic approach”. Static analysis extracts the information by directly inspecting the code. Via static analysis, a control flow graph (CFG) can be derived. Source code may be available for some devices. The binary code of certain IoT devices firmware can be downloaded from the product website. If neither source nor binary code are available, it might be possible for certain embedded devices to extract the binary code through a JTAG connection [39]. JTAG is a hardware interface found on most embedded devices, which allows users to debug or download the program from the device. CFG is a directed graph, where each node, referred to as basic block, represents a code snippet between two consecutive conditional statements. Basic blocks are connected according to the logic in the conditional statements at their end, e.g., continue execution, or jump back to the beginning of a WHILE loop. Figure 5 shows a piece of code and its corresponding CFG. Depending on the variables associated with those conditional statements, the program will go through different paths. The term path condition refers to the set of conditions that lead the program execution to a particular path. Identification of all feasible execution paths of a program is achieved via “dynamic analysis,” e.g. symbolic execution and the Satisfiability Modulo Theory (SMT) solver [40]. Symbolic execution is a means of analyzing a program to determine what inputs cause each path to be executed. Inputs are replaced with symbolic values and the program is executed. The conditions of the conditional statements for each possible outcome during execution are aggregated. In this way, path conditions for all execution paths are obtained together with the program outputs in terms of

the input symbols and variables in the program. The SMT solver takes as input a path condition and returns a set of test cases for that path. These test cases are used as inputs to the target program, e.g., traffic packet data sent to a network router.

3.3 Machine learning

Machine Learning classifiers can also be used to classify side channel signals as “normal” or “abnormal”. The advantage of such models is that they can improve their performance through training without requiring the understanding of the physical model that relates the side signals with the underlying instruction code execution. For these reasons, various ML models have found applications in side channel analysis, including Support Vector Machines (SVM), Random Forests (RF), and Self Organizing Maps (SOM) [41], [42].

More recently, Artificial Neural Networks attracted significant attention for side channel analysis thanks mostly to their considerable success in many other fields. There are generally two types of neural network models: stateless and stateful. Stateless models, have no memory, ie. the current output y(t) depends only on the current input x(t). A typical stateless Neural Network is the Multi-layer Perceptron. The use of MLP’s for side channel analysis has been proposed by various authors, for example, [43].

In a stateful model, on the other hand, the output depends on the recent history of the input patterns for a time-window of length T. Depending on the value of T, we say that the model has short- or long-term memory. Recurrent Neural Networks (RNN) are stateful models with infinite memory implementing a nonlinear IIR system, through an output-to-input feedback loop.

Unfolding an RNN in time we realize it looks like a feed-forward model with infinite depth. Cleverly designed recurrent units can be trained efficiently, by allowing the propagation of the gradients very deep into the architecture. The Long-Short-Term-Memory (LSTM) unit [44] is such an example which has gained immense popularity due to its successful application in many sequence classification tasks. The state, s(t), of the LSTM unit is a dynamic variable which is fed-back on to itself with a gain factor γ. With $\gamma = 1$ the unit can back-propagate the gradients without decay for any number of time instances bypassing the vanishing gradients problem. Moreover, the unit contains two gates: the in-gate in, which controls how much of the input affects the state and the out-gate g out, which controls what percentage of the state goes to the output. Since the in- and out-gates are trained, the model can adaptively determine the length of its own memory by opening the gates for a sufficient number of time instances. Another alternative recurrent unit proposed recently is the Gated Recurrent Unit (GRU) [45] which is quite similar to the LSTM unit but simpler. In some examples, the GRU has demonstrated improved performance, especially in smaller datasets.

Currently there is no mature and systematic rules to determining the hyper-parameters of an deep neural network, i.e. the number of layers, the number of neurons at each layer etc. Researchers usually empirically derive a hyper-parameter set based on trails and errors. Fortunately there is an emerging research trend on automatically determining the hyper-parameters of a deep neural network [46], [47].

An RNN, in general, can be used to capture relationships among data samples in a sequence. It is flexible enough to find relationships even if the samples are further away from each other. The execution of a program can naturally be represented by a sequence of states along the execution, e.g. instructions executed, functions called, or most related to this paper, physical side channel signals emitted. Dependencies in the program control flow (e.g. the conditions of an IF...ELSE... statement might be correlated with several lines of code before) are reflected in the sequence of execution states. These dependencies are essential for tracking which section of the code are executed. Therefore, RNN is very a suitable tool for modeling program execution. Although RNNs have not demonstrated the best possible performance in the context of side channel attacks [48] as we will see, they are very suitable for anomaly detection.

1.1.3 Machine learning training

There are three ways the machine learning uses to train models [49].

Supervised learning is when during the learning process there are labels present on data, both input and the desired output. Training data for supervised learning consist of pairs of input and output data described using predictive variables or features, and the target variable.

It is used in problems such as

- Classification
- Regression
- Interpretation

Unsupervised learning is when there are no labels present. It is the process of uncovering patterns and structures from unlabeled data. It is usually met on grouping data (clustering) and associate analysis.

Reinforcement learning is based on software agents interacting with an environment. These agents are able to automatically figure out how to optimize their behavior, given a system of rewards and punishments. Reinforcement Learning draws inspiration from behavioral psychology, and has applications in many fields, such as economics, genetics, as well as game playing.

In our case supervised learning is used as the problem we are trying to solve is categorized as classification

1.1.4 Classification

Classification is the method or process which classifies a given set of data to a specific number of classes [50]. Classes can be 2 or more and their values are not numerical but categorical. That means the values could be more than a simple “Yes” or “No” like, for example, in the case of an ISP categorizing emails as spams or no spams. Classes are called sometimes as targets, labels or categories.

Classification’s prediction model is based on supervised learning and it is the function that maps the input variables X to discrete output variables Y. Data are fed to the machine learning classifier and a prediction is made from the latter one. Data can be an image, some text from an email etc. These data are fed into a machine learning classifier. In machine learning there is a table of data from which to learn. Along with these data there are the target data, the correct answers as well. The classifier learns to make correct predictions by providing lots of examples of inputs and their target labels in to some algorithm which learns to identify this pattern. There are two classification problems. Binary and multi-class classification.

The case of the binary classification lies in the process in which given a set of standards Xi = {x1, x2, ..., xv} and a set of two classes C = {C0, C1}, it must be determined in which of the two classes belongs to each of the X models. The above procedure is based on the finding of a target function f, which represents each set of values of an object X in one of the two predetermined classes, so that it is possible the classification of future inputs.

For example, spam detection in email service providers can be identified as a classification problem. This is s binary classification since there are only 2 classes as spam and not spam. A classifier utilizes some training data to understand how given input variables relate to the class. In this case, known spam and non-spam emails have to be used as the training data. When the classifier is trained accurately, it can be used to detect an unknown email.

$$y=f(x;w) \tag{1}$$

There are actual three stages in the classification process. The training process, the validation (or testing) of the predictive model and the application to new incoming data. In the first stage, supervised learning is applied with the training data and the corresponding labels being entered. Training data are analyzed, discovering this way the relationships that link each class to the corresponding inputs. This process shows the corresponding classification model for the training data.

In the second stage the classification model is tested using the test data as input, specifying the class to which each input belongs. Comparing the predicted class with the actual class to which it belongs it calculates the accuracy of the model's performance.

The third stage is the actual application of the model to new data, not seen during the training or testing process. The goal of the classifier is to have the ability to generalize.

Mutliclass classification on the other hand is when there are mutliple classes instead of two. Each training data belongs to one of N different classes. The goal remains the same, constructing a classification model which, given new (unseen) data, will correctly predict the class to which one of these data point belongs.

1.1.5 Artificial Neural Networks (ANNs)

Neural networks or, as they are mostly known, artificial neural networks, are communication and information processing systems, inspired by the structure and functionality of the human brain and more precisely by its biological neurons [50].

The most usual Artificial Neural Networks use simple neural models, thus keeping the basics

characteristics of biological neurons. Yet even these models can create interesting networks if they meet two basic characteristics [51]

- Neurons have adjustable parameters
- The network are successfully processing and distributing information through a large density of neurons

Neurons are the basic processing element of one ANN. They often appear in the bibliography as nodes (nodes). The stimuli that the neuron receives from its entrances (dendrites), is the information that is transported to its outputs (synapses), penetrating its axon, to another neuron. The synaptic link and how strong it is, is determined by a numeric value called weight. So when this value it's taken under consideration, while information is transferring to the neuron's entrances, and the sum of these entrances exceeds a threshold, then the axon fires, meaning it transfers the information. Otherwise the neuron is inactive.

1.1.5.1 Artificial Neural Networks history

In 1943, Warren McCulloch, a neurophysiologist, and a young mathematician, Walter Pitts, wrote a paper [52] on how neurons might work. They implemented a highly simplified model of a neuron trying to understand how the brain could produce complex patterns by using many cells that are connected togerther.

In 1949, Donald Hebb, a psychologist, tried to understand how the function of neurons were contributing to processes of psychological nature, such as learning. In his book [52] "The

Organization of Behavior", he pointed out that neural pathways are strengthened each time they are used.

In 1955, Nathanial Rochester from the IBM research laboratories led the first effort to simulate a neural network.

In 1956 the Dartmouth Summer Research Project on Artificial Intelligence [53] provided a boost to both artificial intelligence and neural networks. Considerable theoretical and experimental work had been done and this stimulated research in AI and in the much lower level neural processing part of the brain.

In 1958, Frank Rosenblatt, a neuro-biologist of Cornell, began work on the Perceptron [54]. The Perceptron, which resulted from this research, was in fact a major improvement over the MCP model (McCulloch and Pitts). It was built in hardware and is the oldest neural network still in use today. This invention granted him international recognition. A single-layer perceptron was found to be useful in classifying a continuous-valued set of inputs into one of two classes. The perceptron, using supervised learning, was able to figure out the correct weights by itself directly from the training data. It computes a weighted sum of the inputs, subtracts a threshold, and passes one of two possible values out as the result.

In 1959, Bernard Widrow and Marcian Hoff of Stanford implemented the models ADALINE and MADALINE. These models were named for their use of Multiple ADAptive LINear Elements. MADALINE was actually a multi-ADALINE model. MADALINE was the first neural network to be applied to a real-world problem. It is an adaptive filter which eliminates echoes on phone lines. This neural network is still in commercial use.

Until 1981 there were no progress on neural network research as funding was stopped due to a big wave of criticism towards the scientific teams and their results.

In 1997 a recurrent neural network framework, Long Short-Term Memory (LSTM) was proposed by Schmidhuber & Hochreiter [44].

In 1998, Yann LeCun published Gradient-Based Learning Applied to Document Recognition [55].

There was a small reference in the introduction of what exactly is ANN. Below there is a more detailed description and the best way for a fully understanding of the components and how neural networks work is to go through the simplest model there ever was, the perceptron.

1.1.5.2 Perceptron

Perceptron is a single layer neural network [56]. As mentioned previously, perceptron was introduced in 1958 and it was based on the MCP model. It is actually consisted of one neuron

and it is the still used today as well (Figure 2). Given the fact that this neuron is simulating the biological neuron function it is better to explain what the components are in comparison.

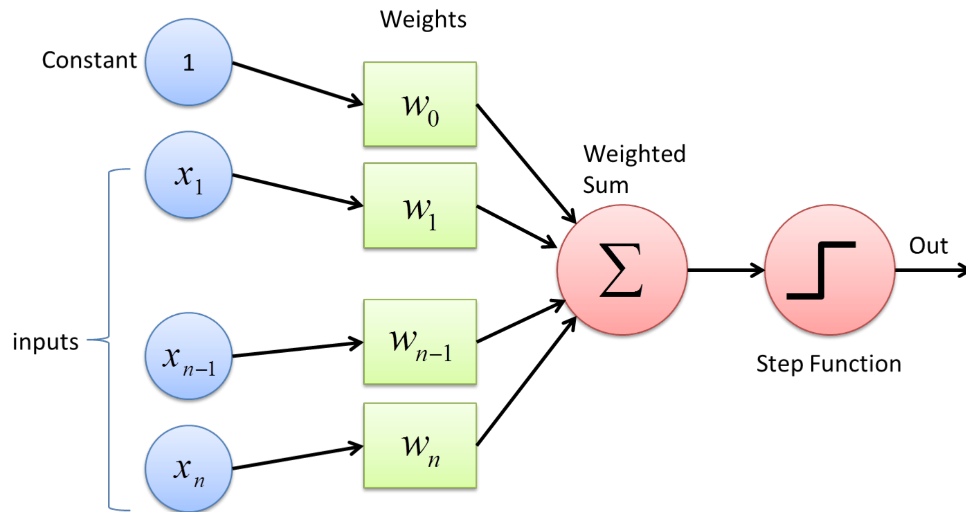


Figure 2. Perceptron

In biological neurons dendrites are receiving electric signals from other neurons through the latter ones' synapses. Synapses are modulating these signals in various amounts. In perceptron these signals are numerical values, inputs X and the corresponding synapses are the weights W. The biological neuron action is modeled by multiplying each input by the corresponding weight.

$$w_i \times x_i \tag{2}$$

The biological neuron fires only when the total amount of the input signals exceeds a certain threshold (θ). In perceptron this action is modeled by calculating the weighted sum of the inputs

$$u = \sum_{i=1}^{n} w_i \times x_i \tag{3}$$

The biological neuron the feeds its output to the next neuron. Applying a step function (activation function) in the perceptron the neuron can determine its output which is fed to other perceptrons. The activation function in a non linear function and especially in the Perceptron it can be:

$$f(u) = \begin{cases} 0, if\ u > 0 \\ 1, if\ u \le 0 \end{cases} \tag{4}$$

or

$$f(u)=\begin{cases} 1, if\ u>0 \\ -1, if\ u\leq 0 \end{cases} \quad (5)$$

Perceptron is used for binary classification (figure 3) and is been trained using supervised learning. The problem with the perceptron is that it can't be used to solve problems which don't have a linear solution. Because of that Multi-layer perceptron was introduced, to solve non linear problems.

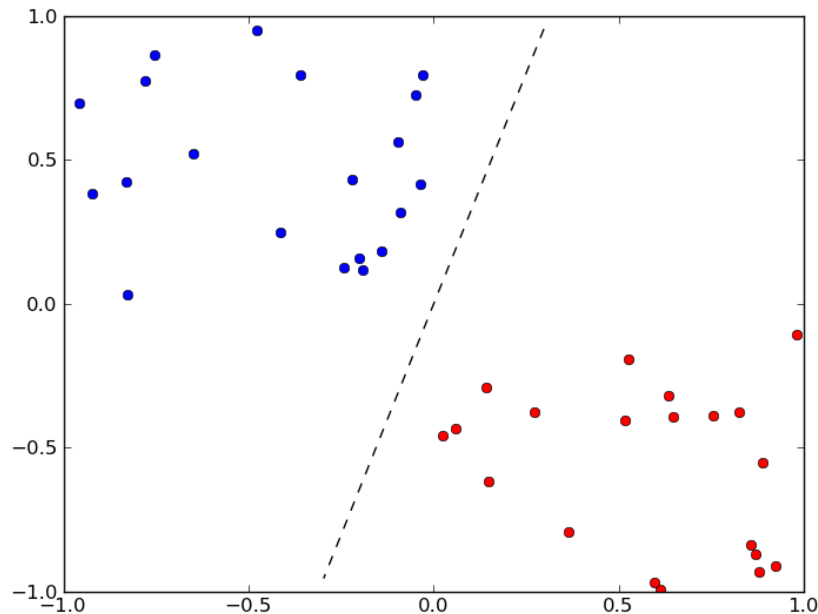


Figure 3. *Linear Binary Classifier*

1.1.5.3 Multi-layer Perceptron (MLP)

The restriction imposed by the use of Perceptron is that it can represent only two dimensions due to the single neuron it has (figure 4). If another neuron is added then the representation of more than two dimensions is possible. It is possible to implement functions that do not can be implemented through a simple Perceptron network. Networks that are made up of various perceptrons are called Multi-Layer Perceptron [57]. The main feature of these networks is that the neurons of any layer are fed exclusively from the neurons of the previous layers and at the same time feed exclusively the neurons of the next layer.

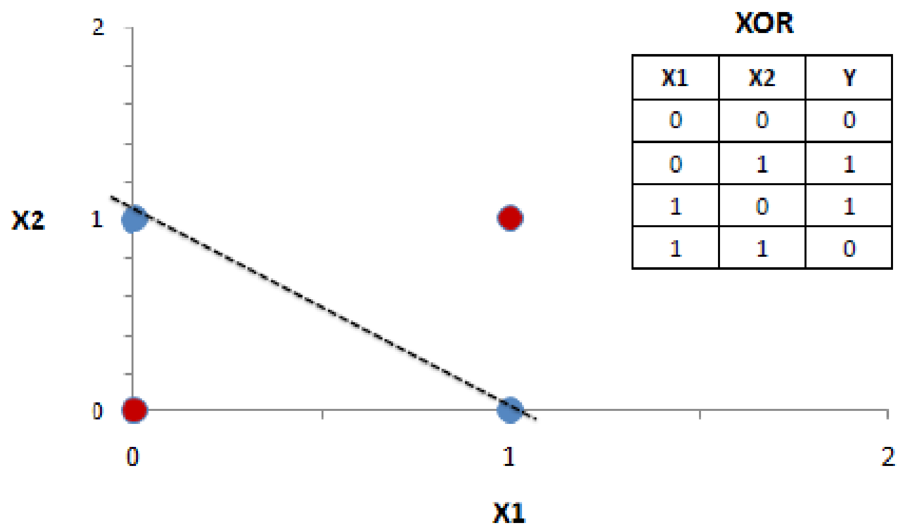


Figure 4*. Inability of perceptron to solve problems like XOR*

MLP is the fundamental principle of neural networks and mostly of Deep Learning. A MLP is a deep, artificial neural network and they are composed of an input layer for receiving the signal, an output layer for making the decisions or predictions and a number of hidden layers between them. These hidden layers are the heart of MLP and deep learning itself.

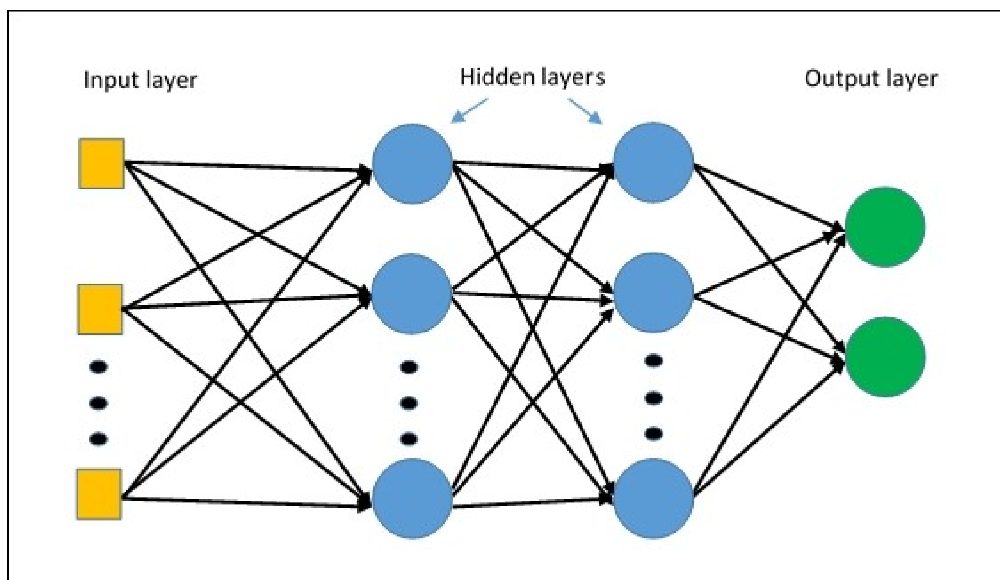


Figure 5*. Multi-layer Perceptron*

MLP is a feed forward network, which means that the input data move towards the output through the hidden layer(s) and there are no loops or so ever. The training of this model is based on the backpropagation algorithm with weights correctness and error minimizing.

1.1.5.4 Activation functions

Activation functions are really important component of the ANNs as they are actually mapping the output to a given input. They are, usually, non-linear functions and they convert the input signal of a neuron to an output signal which will feed the following neurons.

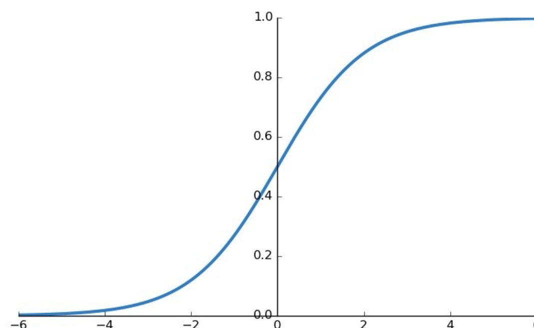
The essence of the activation functions' nature must be non-linear in order for the ANN to learn more complicated things. A linear equation is much easier to be solved but they are limited in applied to really simple models as linear regression and classification models. The world is built on complicated concepts and is nature is non-linear. ANNs must learn to compute and represent data as images, audio etc where their architecture is not simple. These networks have to become more powerful by adding the ability of learning something complex and more complicated. Thus non-linear activation functions are able to generate non-linear mappings from inputs to outputs.

Finally non-linear activation functions are differentiable, something that is mandatory to perform training algorithms such as back-propagation optimization algorithms.

Some of the most popular and used activation functions are:

- Sigmoid or Logistic

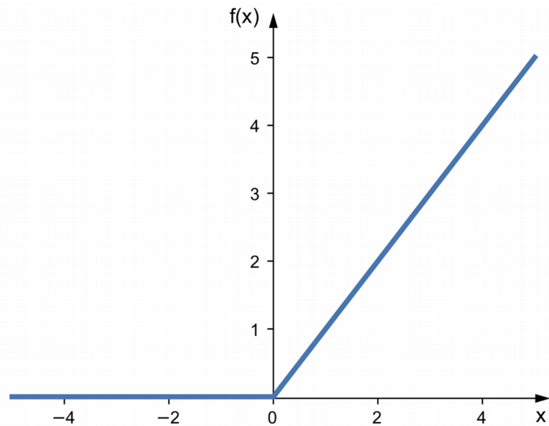
$$f(x)=\frac{1}{1+e^{-x}} \tag{6}$$



Sigmoid function is a non-linear continuously differentiable function and that exactly is its biggest advantage against linear function. The range is from 0-1 and that is a disadvantage as it is not symmetric around the origin. Besides that the gradients can be really small after a certain point.

- ReLU

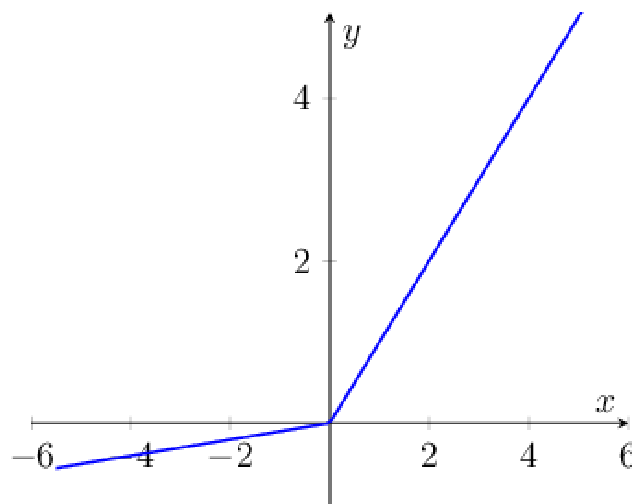
$$f(x)=max(0,x) \tag{7}$$



ReLU function is the most popular and most widely used activation function today. It is not linear function, which means it can be easily differentiated. One of ReLU's disadvantages is that the negative side of the graph results in gradients equal to zero, meaning that the weights are not updated during the back-propagation.

- Leaky ReLU

$$f(u)=\begin{cases} ax, if\ x<0 \\ x, if\ x\geq 0 \end{cases} \tag{8}$$



Leaky ReLU function is an improved version of the ReLU function. It can actually delivers successfully the solution to the gradient equal to zero issue of the ReLU for $x < 0$, which makes the neuron's activation zero.

- Softmax

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for j=1, …, K} \tag{9}$$

Softmax function is a type of sigmoid function but is really useful when it comes to handle multiple classes. It is squeezing the outputs for each class to a value between 0 and 1 and also dividing them by the sum of the outputs. Essentially this gives the probability of each input belonging to a specific class.

1.1.5.5 Back-propagation

In the Back-Propagation algorithm the learning process is supervised, thus we have the input vector, output vector and targets (x, y, d) and the corresponding training data sets in pairs.

$$x^{(k)} = \begin{bmatrix} x_1^{(k)} \\ \ldots \\ x_m^{(k)} \end{bmatrix} \quad \text{input vector } \mathbf{x} \tag{10}$$

$$y^{(k)} = \begin{bmatrix} y_1^{(k)} \\ \ldots \\ y_n^{(k)} \end{bmatrix} \quad \text{output vector } \mathbf{y} \tag{11}$$

$$d^{(k)} = \begin{bmatrix} d_1^{(k)} \\ \ldots \\ d_n^{(k)} \end{bmatrix} \quad \text{target vector } \mathbf{d} \tag{12}$$

A neural network captures its learning ability and its knowledge on the weights. Weights can successfully transform the signal to a really good decision or the opposite, depending on the training process. As already mentioned, neural networks are feed-forward networks and once the prediction is made the output data distance from the target or ground truth can be measured. This is the error of the particular model. A really popular cost function which is being used often is the mean square error

$$E = \frac{1}{K} \sum_{k=1}^{K} \| d^{(k)} - y^{(k)} \|^2 \tag{13}$$

The algorithm uses the cost criterion (function) to minimize the error based on the model outputs and the desired values of the targets [58]. The optimization method used by the back-propagation algorithm is the gradient descent method which finds the minimum of the cost function.

There is an obvious separation of the training process in two major steps, forward and back-propagation.

During the forward-propagation:

- an input vector **x** is fed to the network and its input neurons. The weights (including biases) are given by the vector **w**
- an output vector **y** is produced
- weights don't change at all at this stage

During the back-propagation:

- the error is calculated based on the output and the target (equation 15)
- using gradient descent the error is fed back to the network backwards.

  The total error **E** is affected by the weights as the input doesn't change. Increasing or decreasing the weights the value of the total error becomes smaller or bigger. The desirable value of the error has to be close to zero, the minimum. Considering a graph of the cost function the global loss minimum is the value we are looking for.

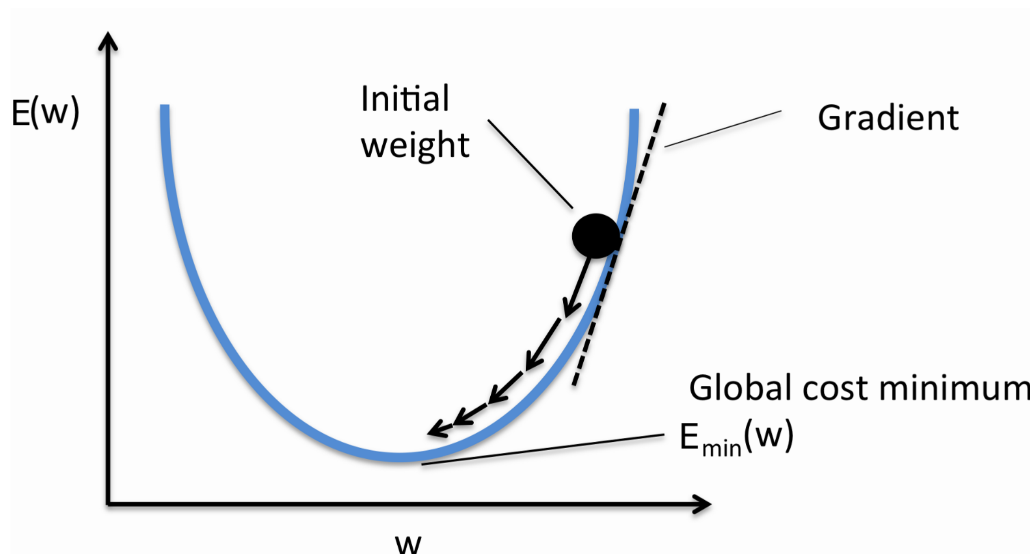


***Figure 5**. Gradient descent*

  This can be written by its mathematical form, the partial derivative of **E** with respect to the $\mathbf{w_{ij}}$.

$$\frac{\partial E}{\partial w_{ij}} \tag{14}$$

  Applying the chain rule it is possible to calculate all the partial derivatives with respect to each weight. This is done by the optimization functions.

- weights adjust their values. The update on each weight is being calculated in a reverse way, from the output towards the input and there is a learning rate ***a*** which is a really small positive number.

$$W_{ij}^{(t+1)} = W_{ij}^{(t)} - a \frac{\partial E}{\partial w_{ij}} \tag{15}$$

The whole training process using the back-propagation algorithm is really easy to be implemented and there are few parameters need to be adjusted. It might be slow and it can be terminated when:

- the cost for one cycle (age) of the algorithm is smaller than a specified threshold
- the error in two consecutive epochs is not diminished significantly
- the difference between two consecutive updated weight values are significant small

1.1.5.6 Loss function

Loss function is the function that evaluates the output, the solution of an optimization algorithm. Depending on the solution it might be searching the best possible output that has the highest or the lower score. It is an indicator of how well the predictive model performs and it helps adjusting the weights towards the best possible solution.

In neural networks the loss function has an important job to do. It has to calculate the error of the model during the optimization process and that might be a really challenging task, as it must captures the properties of the corresponding problem [59]. The choice of a loss function is not as simple as it seems but there are some ways to point the right one with respect to the particular problem or model is being constructed. It is much more preferable the use of function where the solutions are on a smooth landscape allowing the optimizer to navigate through in a better-smoothy way.

Especially in the case of classification a really popular and most common loss function is the cross entropy loss.

1.1.5.7 Optimization methods

Optimization methods are used for finding the minimum of a function. In the case of the neural networks this function is the loss function and it helps in updating the weights through back-propagation. Most of the optimization methods Deep learning is using come from the Stochastic Gradient Descent (SGD).

The problem with SGD is that it updates a parameter for each training example. That can be a good thing, as it helps to discover more, and possibly better, local minima, because of the frequent fluctuations with high diversion. However, at the same time, the procedure becomes more complicated and once the convergence to the minimum is done it will keep going on. To overcome this issue many other alternative methods were created based on SGD. Some of the most used ones are

- Gradient descent with momentum
- AdaGrad

- Adadelta
- Adam
- AdaMax

1.1.6 Deep learning

Deep learning is a neural network with many layers (hidden layers). Its concept is based on multi layers of single neural networks. Visible layers are only the input and the output [59].

Traditional machine learning works well on datasets which are consisted of a few hundred features. An unstructured dataset however, like the ones from images, has a really big number of features making the process challenging and completely unfeasible. Deep learning models are using their many layers to find the best possible way to learn from raw data and extract useful features. They learn progressively as data goes through each neural network layer. In the image example, as data goes through the first layers, model is learning how to detect features like edges and is moving even deeper to the next layers transferring these features. Those layers help as well to extract other features, adding more details. The higher layers build higher levels of abstraction based on things that the lower layers are learning. That way the model could be able to distinguish objects and parts in the image like a face, an arm, a car or a tree. It allows the computer to learn complicated concepts by building them out of simpler ones.

The concept of Deep learning is not something new. As mentioned previously neural networks' history starts way back in the 50's and 60's. However, only in 00's there was a scientific explosion with researches related to deep networks. There is a reason for that and that would be the enormous amount of data and the incredible computational power of today's systems along with the appearance of multi-cores GPUs.

Back in the days when Deep learning concept was introduced, there were no enough resources to provide the necessary and, at the same time, large number of data to train such models. Gathering data was an exhaustive and, totally, not pleasant job at all. Besides that those models demand a huge computational power, making the deep learning models not a good fit for the standards of that time.

Today, in the time of Big Data, the sources deep learning models use can provide data more than enough to help them in the training process and of course to evaluate the final outcome. Deep learning requires large amounts of labeled data. It requires also computational power that didn't exist till recently. GPU-accelerated frameworks are used to train deep learning networks more flexible and provide speed and accuracy at the same time. GPUs can have a

parallel architecture which is efficient for deep learning models. Training process duration can be reduced from days and weeks to hours or even less.

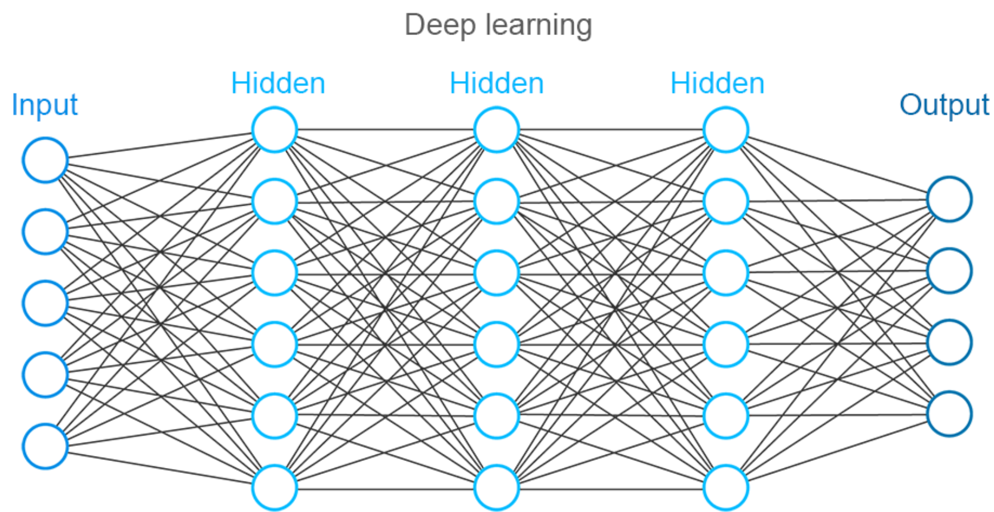


***Figure 6**. Deep learning network*

There are several deep learning architectures in the scientific community, others are used in commercial applications and others are still in the research stage, providing some promising results though.

1.1.6.1 Deep Belief Networks

Deep Belief Networks (DBNs) are based on Restricted Boltzmann Machines. To fully understand the architecture of DBNs an introduction of RBMs is mandatory.

An RBM is a shallow two layer net, the first layer is known as the visible layer and the second is called an hidden layer. Each node in the visible layer is connected to every node in the hidden layer and RBM is considered restricted because no two nodes in the same layer share a connection. RBM, in the forward pass takes the inputs and translates them into a set of numbers that encode the inputs and in the backward pass it takes this set of numbers and translates them back to form the initial inputs. RBM was introduced to replace the traditional way the neural networks were trained, the back-propagation algorithm and the vanishing gradient problem [60].

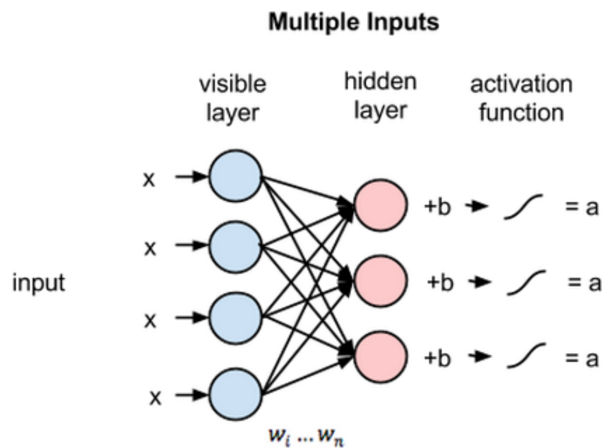


Figure 7. Restricted Boltzmann Machine

An interesting aspect of an RBM is that the data does not need to be labeled. This turns out to be very important for real world data sets like photos, videos, voices and sensor data, all of which tend to be unlabeled.

A well trained RBM will be able to perform the backwards translation with a high degree of accuracy. In both steps, the weights and biases have a very important role. They allow the RBM to correlate and, at the same time, find relationship patterns among the input features. This way they help RBM to decide which features are the most important when detecting patterns [61].

After several forward and backward passes RBM is trained to reconstruct the input data. Three steps are repeated over and over through the training process. With a forward pass, every input is combined with an individual weight and one overall bias and the result is passed to the hidden layer, which may or may not activate. Next, in a backward pass, each activation is combined with an individual weight and an overall bias and the result is passed to the visible layer for reconstruction. At the visible layer, the reconstruction is compared against the original input to determine the quality of the result.

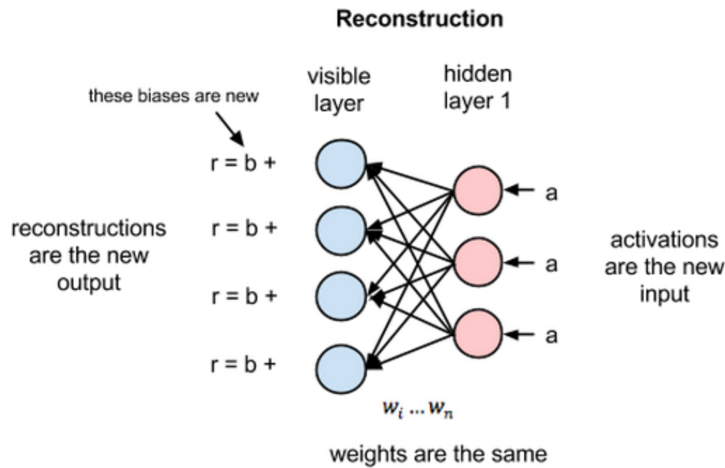


Figure 8. *Restricted Boltzmann Machines - Reconstruction*

Rather than having people manually label the data and introduce errors, RBM automatically sorts through the data. By properly adjusting the weights and biases RBM is able to extract the important features and reconstruct the input. An important note is that an RBM is actually making decisions about which input features are important and how they should be combined to form patterns.

RBM is part of a family of feature extractor neural nets, which are all designed to recognize inherent patterns in data. These nets are also called autoencoders because in a way, they have to encode their own structure.

An RBM can extract features and reconstruct inputs. By combining RBMs together and introducing a clever training method, a powerful new model can be obtained, the Deep Belief Network (DBN). Just like the RBM, deep belief nets were also conceived by Geoff Hinton as an alternative to back propagation.

In terms of network structure, a DBN is identical to an MLP but when it comes to training, they are entirely different. In fact, the difference in training methods is the key factor that enables DBNs to outperform their shallow counterparts. A deep belief network can be viewed as a stack of RBMs, where the hidden layer of one RBM is the visible layer of the one above it [62].

A DBN is trained as follows. The first RBM is trained to reconstruct its input as accurately as possible. The hidden layer of the first RBM is treated as the visible layer for the second and the second RBM is trained using the outputs from the first RMB. This process is repeated until every layer in the network is trained. A DBN works globally by fine tuning the entire input in succession as the model slowly improves. Kind of like a camera lens slowly focusing a picture. The reason that a DBN works so well is highly technical, but it would suffice to say that a stack of RBMs will outperform a single unit, just like a multi layer perceptron was able to outperform a single perceptron working alone. After this initial training, the RBMs have

created a model that can detect inherent patterns in the data. To finish training, a process of introducing labels to the patterns is needed and fine tune the net using supervised learning. To do this, a very small set of labeled samples is needed so that the features and patterns can be associated with a name. The weights and biases are altered slightly, resulting in a small change in the perception of the patterns and often a small increase in the total accuracy. Fortunately, the set of label data can be small relative to the original data set, which is extremely helpful in real world applications.

1.1.6.2

Recurrent Neural Networks (RNNs) are really useful for processing sequential data such as time series, sound or text processing. This deep learning model has a simple structure with a built in feedback loop, allowing it to act as a forecasting engine. In a feed forward neural network signals flowing only one direction from input to output one layer at a time in a recurrent that the output of a layer is added to the next input and fed back into the same layer, which is typically the only layer in the entire network. At t=1, the net takes the output of time t=0, and sends it back into the net along with the next input. The net repeats this for t=2, t=3, and so on. Unlike feedforward, recurrent nets can receive a sequence of values as input, and it can also produce a sequence of values as output. The ability to operate with sequences opens up these nets to a wide variety of applications, for example, when the input is singular, and the output is a sequence the potential application is image captioning. A sequence of inputs with a single output can be used for document classification. When both the input and output are sequences, these nodes can classify videos frame by frame. As seen previously with other deep learning models, by stacking RNNs on top of each other, you can form a net capable of more complex output than a single RNN working alone.

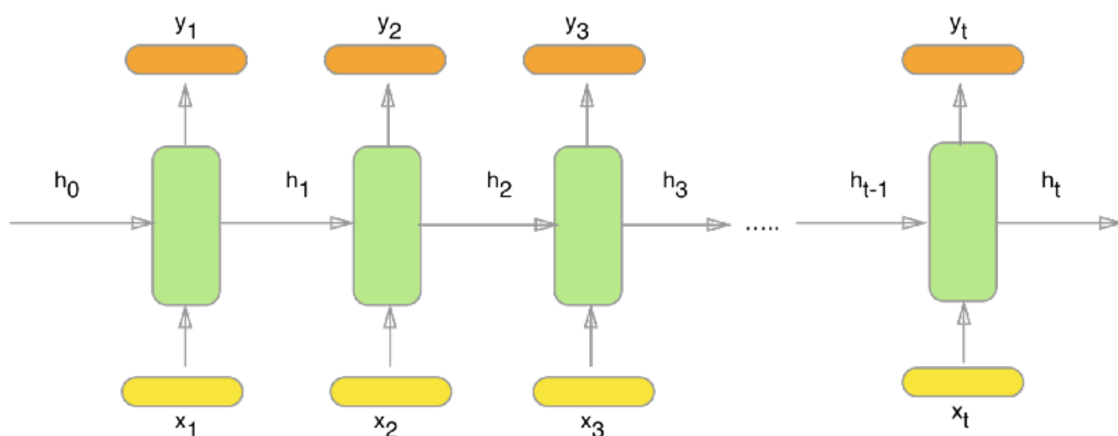


Figure 9. *RNN*

An RNN is an extremely difficult net to train. Since these nets use back propagation, the problem of the vanishing gradient is present. Unfortunately, the vanishing gradient is

exponentially worse for an RNN. The reason for this is that each time the step is the equivalent of an entire layer in a feedforward network. So training an RNN for 100 time steps is like treating a 100-layer feedforward net. This leads to exponentially small gradients and decay of information through time.

There are several ways to address this problem, the most popular which is gating. Gating is a technique that helps the net decide when to forget the current input and when to remember it for future time steps. The most popular gating types today are GRU and LSTM. When it comes to training a recurrent net GPUs are an obvious choice over an ordinary CPU. GPUs are able to train RNNs 250 times faster.

1.1.6.2.1 Long Short-Term Memory (LSTM)

In RNN there is an obvious flaw caused by the vanishing gradient problem. It doesn't contribute to learning too much. The earlier layers usually don't learn at all. They are suffering from sort memory. This is exactly the reason that LSTM networks were created, as a solution to the sort memory problem [44].

LSTM's most known feature is the gates. These are internal mechanisms which are used to regulate the flow of data. These gates can learn in time which data in a sequence is important to keep and which are not. By doing that it learns to use relevant information to make predictions. LSTMs can be found in speech recognition, in synthesis softwares speech to text etc.

RNN works pretty good for a short sequences of data. The computational cost is significant smaller, having just a few operations to go through, internally. LSTM is pretty much similar to RNN. Data flow is close to the latter's one. The differences are located internally in the cells operations. They are using the gating technique.

LSTMs fundamental unit is the memory block, located in the recurrent hidden layer. They are formed by the memory cells and the gates. Cells are connecting to gates with the latter ones being responsible to control the data flow. These gates are the input, output and forget [63]. All three gates contain sigmoid functions for squishing the values between 0 and 1. Some of them contain tanh to help regulate the network. Information is coming through the input gate which decides how important this information is. Value closer to zero means not important. Information coming through the forget gate gives an output within this time range. Values that are closer to 0 indicate that the corresponding information has to be thrown away, has to be forgotten. Values closer to 1 are kept and passed through the next layer. Cell state is updating to new values relevant to the network and the output gate decides what the next hidden state is.

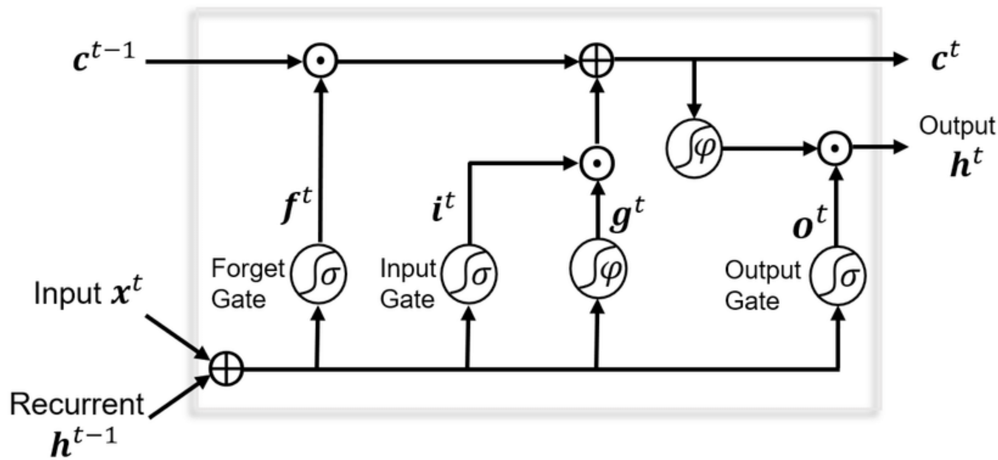


Figure 10. LSTM

3.4 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are deep neural networks and their main use is on image processing. They primarily are used for image classification in the field of computer vision [64][65]. They are the most popular deep networks since 2012 when they were first used on a challenging and well known competition, the ImageNet. The winner, a CNN model (AlexNet) succeed into an unexpected 15% classification error, dropping the record by 11 units (26% was till that time) [66]. Scientific community welcomed this astounding improvement and after that there was an explosion in research papers, trying to imitate and improve CNN models based on the first successor. All the next competitions had remarkable models improving even more the classification error, setting the CNN a mandatory network when it comes to image classification.

CNNs are used in several other occasions, besides the traditional image classification. Companies are using these networks for photo searching, grouping them by similarity (clustering). Google does that and the results are impressive. Along with photo clustering they are incredible reliable to perform object and face recognition. Facebook is using CNNs in their tagging algorithms. Amazon, Instagram, Pinterest are some other companies that are using CNNs for their services.

CNNs are making the assumption that data has the form of images. The input is images, a bunch of pixels, numbers indicating the values referring to RGB. The output is nothing more than numbers describing the probability of the image belonging to a specific class. Making the forward function more efficient to implement, CNN architecture is reducing at the same time the amount of parameters in the network.

1.1.7 Architecture

Traditional neural networks can not scale well to full images. The amount of weights that have to handle is, from a certain point and on, quite big. The huge number of parameters would quickly lead the model to overfitting. CNN’s architecture can be more flexible and it can be constrained in a more meaningful way. Its layers are consisted of neurons set in 3 dimensions, for example a colorful image which has width, height and depth.

CNN’s architecture is a little bit different than the traditional neural networks. The basic components are the Convolutional layer, the activation function (non-linear), the pooling layer (downsampling) and the Fully-connected layer [55]. The common components with respect to neural network are the activation function and the Fully-connected layer. Their functionality is exactly the same on both networks. The input on a CNN model is actual an array of pixel values, representing an image. The dimensions of the array as mentioned before is, mostly, 3D. The output can be a single class (or a probability of classes) describing the image in the best possible way. It is easy and correct to say that the CNN architecture mainly is consisted of two parts. The first part, the convolutional, is responsible for extracting features and the second part is the classifier, the Fully-connected layer.

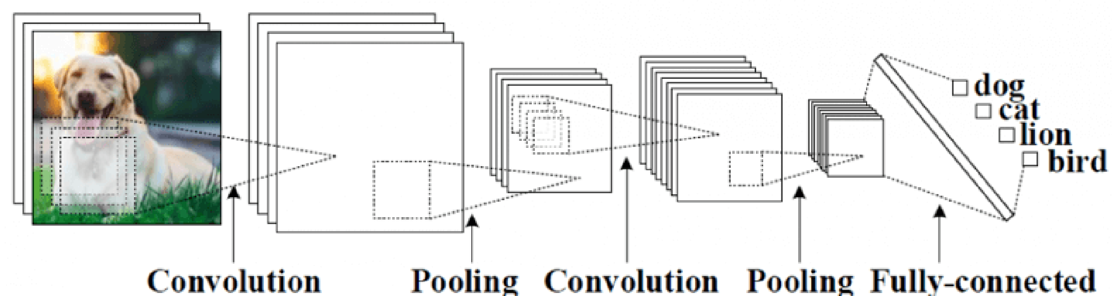


Figure 11. *CNN high layer architecture*

1.1.7.1 Convolution layer

It is always the first layer in a CNN model. The parameters of a Conv layer are actually part of a group of filters. Filters which are learnng while they are scanning the image. Each filter convolves across the width and height of the input volume, computing element wise multiplications between the pixel values of the image and the values in the filter. Convolution is coming from the signal processing field and is a mathematical way to express the combination of two signals to a third one. Mathematically during the convolution the multiplications are all summed up and the output is a single number.

Filters have the same depth with the input so if, for example, the input has size 32x32x3 then filters might be 5x5x3. The numbers forming the filters are called weights or parameters. Depth in this case is the number 3 and the area of the image through which filters are passing

is called receptive field. As the filter keeps sliding over the image more single numbers are produced. These numbers are moved each time to a two dimension array, the activation or feature map. Each filter produces one activation map and the activation maps from the input layer, stacked together, are becoming the input to the second Conv layer.

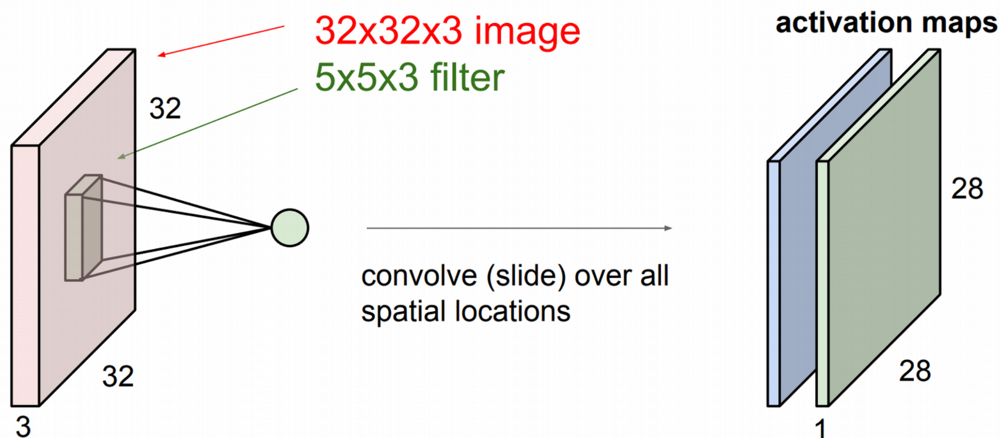


Figure 12. *Convolution layer*

The intelligent nature of CNNs is hidden in these layers. Filters are providing useful information (features) regarding the image and its context. Starting from the first Conv layer the features being extracted are low level features. They might be edges, lines, or curves etc. Each filter is totally different so the extracted feature are a bunch of different lines, curves. Going through the next Conv layers the features become more complex, high level features.

When it comes to CNNs the calculations in Conv layers demand the basic arithmetic knowledge to be known. Besides that there are some parameters to set and other layers to go through before the output of the previous Conv layer becomes the input to the next Conv layer. These are the activation function (usually ReLU) and the pooling layer.

There are three hyperparameters being used in CNNs in order to maintain the spatial nature of the initial input. The size of the output volume is based on these parameters and is a key point in the training process:

- Depth, is the number of the filters being applied to the input. It is not a steady number and can be changed on each Conv layer.
- Stride which is going to be used for sliding the filter, for example, if stride is 1 then the filter is being moving one pixet at a time.
- Zero-padding, is the number which indicates the size of padding the input volume with zeros around the border. Zero-padding maintains the spatial size of the original volume and controls it.

Let’s assume we have images of size 32x32x3 and we want to use 4 filters of 5x5x3 size with stride 1 and no padding.

Assuming that W is the width, H is the height and D the dimension then we will have the following values:

$W_1 = 32$, $H_1 = 32$ and $D_1 = 3$

F is the spatial size of the filters and K is their actual number

F = 5, K = 4, S = 1, P = 0

The output is going to be a volume of size W_2 x H_2 x D_2

$$W_2=\frac{(W_1-F+2P)}{S}+1 \quad (16)$$

$$H_2=\frac{(H_1-F+2P)}{S}+1 \quad (17)$$

$$D_2=K \quad (18)$$

$W_2 = 28$, $H_2 = 28$ and $D_2 = 4$

The pooling layer is responsible for downsampling, reducing the spatial size of the volume. The reason is to reduce the amount of parameters and computation in the network, so the network doesn’t overfit. The pooling layer has filters as well and based on the size of these filters along with type of pooling, the downsampling changes the output size without changing the data representing the initial pixels. Calculating the hyperparameters the only difference with the filters’ applications is that there is no padding and $D_2 = D_1$

1.1.7.2 Fully Connected layer

The convolutional layers helps the model to extract meaningful features from the initial input (images). The final output (in the Convolution part) is a set of features (weights) that need to be classified in order to provide any information that indicates the relation of the output to the ground output. The output of all neurons are joined and flattened to a vector of N dimesnion, where N is the number of the classes we want to classify the original input. The difference between the fully connected layer and the convolution layer is that the latter one can share parameters through its neurons, not connecting all the activations of each layer to the next one’s. On the other site, fully connected layers have connections to all activations in the previous layer. Each number in this vector represents the probability of a specific class (out of the N classes).

1.1.7.3 Overfitting in CNN

Overfitting and undefitting are presents in CNN as well. However, there are introduced some methods which are not applied to traditional machine learning alrgorithms. It is well known that a model is overfitting when its training procedure makes it fitting too well to the training set. Generalization in this case is difficult to established, as the model learn to recognise specific images instead of patterns.

There are some steps to avoid overfitting and amongst those the most popular and recent one is the dropout method [67]. This method, during the training process, sets activations to zero, in a random way. In the prediction process this method does not apply but a reduce to the activations number is taking place. This way dropout can make a network to learn more useful features. Along with dropout method data augmentation can contribute in reducing overfitting as well. Similar to dropout data augmentation happens only in training set and it includes images coming from the original rotating it, adding color filter etc.

4 Power Side Channel Execution Monitoring using CNN

Studying the related work and analyzing the advantages and disadvantages of several models, it is obvious the need of a much better model overall. Here we demonstrate the use of power side channel signals for execution monitoring and intrusion detection using CNNs. Based on the literature the steps following this section would be the profiling model construction and the execution monitoring intrusion detection.

4.1 Profiling Model Construction

1.1.8 Program Analysis

The flow of a program execution is controlled by conditional statements, such as IF THEN, WHILE and FOR loops. Depending on the variables associated with those conditional statements the program will go through different execution paths. The term path condition refers to the set of conditions that lead the program execution of a particular path. For profiling, we need to identify all feasible paths. This is achieved by applying symbolic execution and the satisfiability modulo theory (SMT) solver [40] on the source code. Symbolic execution is a means of analyzing a program to determine what inputs cause each path of a program to be executed. It replaces concrete inputs with symbolic values and then executes the program. The conditions of the conditional statements are aggregated for each possible outcome during execution. Therefore, path conditions for all execution paths can be achieved together with the program outputs in terms of the input symbols and variables in the

program. The SMT solver takes as input a path condition and outputs a set of concrete value sets. These value sets are called test cases for that path and are used as inputs to the program.

Program analysis was performed using the symbolic testing tool KLEE. For our preliminary results, the micro-controller was running a Dijkstra shortest path algorithm im- plementation. We collected 41 execution paths and generated 100 test cases for each path. We transferred the test cases onto the Arduino, through the serial communication interface of Arduino (controlled by its MATLAB API).

1.1.9 Signal acquisition

For power signals, the quality depends on the voltage regulator. The voltage fluctuation with respect to the ground (GND) at the VCC pin can be measured by an oscilloscope probe connecting to an oscilloscope. Resistors and capasitors can be added between the power supply and the VCC pin of the processor in order to get more stabilized measuremens. This is called a differential measurement. This has the advantage of masking unpredictated events such as random pertubations. However, when the target device and the oscilloscope are plugged in the same outlet, can damage the oscilloscope in some cases. To avoid somethiing like that it might not power the target device from the outlet, but instead use the USB port of a battery powered laptop, or simply use battery. One can also measure the voltage fluctuation on each end of the resistor separately, and do the differentiation in the oscilloscope if a multi-channel oscilloscope is available. Finally, a differential probe will also solve the problem

The corresponding signal was captured on the oscilloscope using the oscilloscope's MATLAB API through the Ethernet connection from a computing station. One power signal was captured for each test case. Captured signals were also transferred through the Ethernet connection to the computing station for future processing. Power signals were reshaped with a window size of 500.

```
function dataCollection1(num_tc, num_tr, data_folder, testcases)
% test using random testcases

% connect to Arduino serial port
arduino = serial('/dev/ttyACM0', 'BaudRate', 9600);

% connect to oscilloscope
if 1
    % Create a TCPIP object.
    interfaceObj = instrfind('Type', 'tcpip', 'RemoteHost', '192.168.137.1', 'RemotePort', 1861,
'Tag', '');

    % Create the TCPIP object if it does not exist
    % otherwise use the object that was found.
    if isempty(interfaceObj)
        interfaceObj = tcpip('192.168.137.1', 1861);
    else
        fclose(interfaceObj);
        interfaceObj = interfaceObj(1);
    end

    % Create a device object.
    deviceObj = icdevice('lecroy_basic_driver.mdd', interfaceObj);
    set(interfaceObj, 'InputBufferSize', 2000000);

    % Connect device object to hardware.
    connect(deviceObj);

    % Get utility device for sending commands
    util = get(deviceObj, 'Util');
end

% test case generation
% load(fullfile('data', testcases_file), 'testcases')
```

```
if 1
    tc = randperm(length(testcases));
end

tested_cases = tc(1:num_tc);

if ~exist(fullfile('data', data_folder), 'dir')
    mkdir(fullfile('data', data_folder))
end

save(fullfile('data', 'tested', data_folder), 'tested_cases')


% data collection
fprintf('testcases:'), display(tested_cases)
if 1
    for i = 1:num_tc
        % keep the serial port open will cause strange pattern in the power trace
        fopen(arduino);
        pause(3);
        fprintf(arduino, '%s', testcases(tested_cases(i)));
        pause(5)
        fclose(arduino);
        pause(1);
        % send testcase
        disp(['collecting #', num2str(i),  ' test case, #', num2str(tc(i))])

%         checkCurrentDir(util)
%         % create data folder
%         CMD = ['DIR DISK,HDD,ACTION,CREATE,', num2str(i)];
%         invoke(util, 'sendcommand', CMD);
%         % switch to data folder
%         CMD = ['DIR DISK,HDD,ACTION,SWITCH,', num2str(i)];
%         invoke(util, 'sendcommand', CMD);
%         dd = checkCurrentDir(util)
%         while ~isequal(dd(21), '\')
%             dd = checkCurrentDir(util)
%         end
       % data collection
        CMD = 'STST C1,HDD,MATLAB';
        invoke(util, 'sendcommand', CMD);
        for num_traces = 1:num_tr
            disp(['-----collecting #', num2str(num_traces), ' sample...'])
            invoke(util, 'sendcommand', 'STO');
            pause(2)
        end
        % trigger collection
        CMD = 'STST C2,HDD,MATLAB';
        invoke(util, 'sendcommand', CMD);
        invoke(util, 'sendcommand', 'STO');
        pause(5);
```

```
%       % switch directory back
%       invoke(util, 'sendcommand', 'DIR DISK,HDD,ACTION,SWITCH,..');
%       checkCurrentDir(util)
    end
end

movefile('/mnt/osc/testing/*', fullfile('data', data_folder));

delete(deviceObj)
delete(interfaceObj)
delete(arduino)
```

1.1.10 Signal and data Pre-processing

Power side channel signals are reshaped into 2D matrices by splitting each signal according to a window length and stacking these signal segments together. Power side channel signals have several components that contribute to its discriminative signal pattern, e.g., a fundamental periodic signal component that corresponds to the processor's clock rate; modulation signal components that correspond to the loops in the program structure; signal peaks of various amplitude that correspond to different instructions as well as different orders of instructions along the execution; large spikes due to I/O read and write. By reorganizing the signal points into a 2D matrix, these components form fundamental image elements such as edges, lines and shapes that will be utilized by our CNN model to recognize signals of different execution paths.

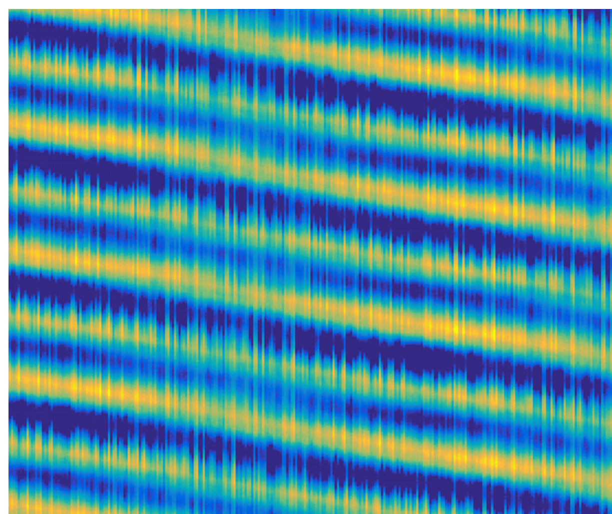


Figure 12. Power Image

```
"""
rename subfolders and files
"""

import os


#data folder
folder_path = '../sig2img_new2'

#creating list with the folders/classes
folders_list = os.listdir(folder_path)                         # ['class32', 'class15', 'class39', ....]

for folder_name in folders_list:
    folders_path = os.path.join(folder_path, folder_name)              # '../sig2img/class32'
    new_folder_name = 'class' + folder_name
    new_folder_path = os.path.join(folder_path, new_folder_name)
    os.rename(folders_path, new_folder_path)
    files_list = os.listdir(new_folder_path)                         # ['93.mat', '78.mat',
'65.mat', .......]
    for each_file in files_list:
        files_path = os.path.join(new_folder_path, each_file)
        new_filename = new_folder_name + '_' + each_file
        new_filename_path = os.path.join(new_folder_path, new_filename)
        os.rename(files_path, new_filename_path)
```

4.2 Execution Monitoring and Intrusion Detection

The execution profiling model is essentially a classification model, where execution paths are treated as classes. Power signals generated by test cases that belong to one specific path are members of that class.

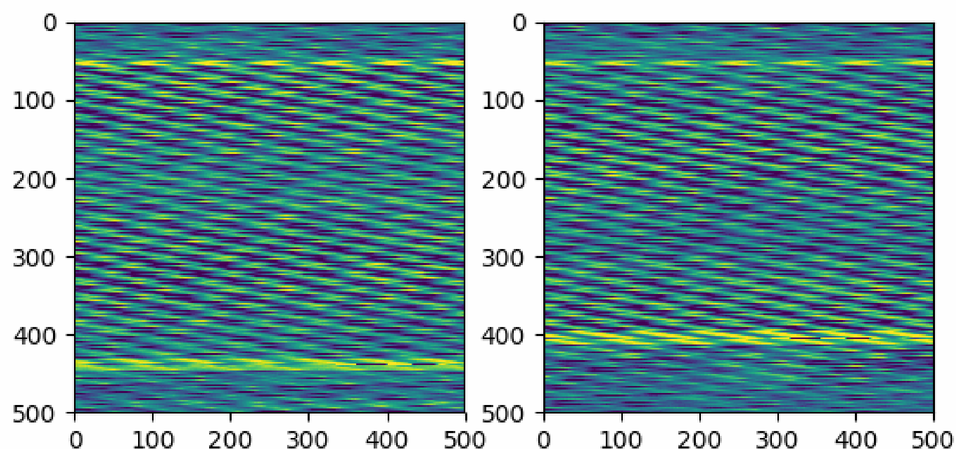


Figure 13. *Two different classes – two paths*

```
"""
rename subfolders and files
"""

import os


#data folder
folder_path = '../sig2img_new2'

#creating list with the folders/classes
folders_list = os.listdir(folder_path)                                  # ['class32', 'class15', 'class39', ....]

for folder_name in folders_list:
    folders_path = os.path.join(folder_path, folder_name)               # '../sig2img/class32'
    new_folder_name = 'class' + folder_name
    new_folder_path = os.path.join(folder_path, new_folder_name)
    os.rename(folders_path, new_folder_path)
    files_list = os.listdir(new_folder_path)                            # ['93.mat', '78.mat',
'65.mat', .......]
    for each_file in files_list:
        files_path = os.path.join(new_folder_path, each_file)
        new_filename = new_folder_name + '_' + each_file
        new_filename_path = os.path.join(new_folder_path, new_filename)
        os.rename(files_path, new_filename_path)
```

```
"""
seperates files into two folders with the same subfolders
"""

import os
import shutil
import random

folder_path = '../sig2img_new'
folders_list = os.listdir(folder_path)                                  # ['32', '15', '39', ....]
dest_folder_path = '../sig2img_new2'

for folder_name in folders_list:
    folders_path = os.path.join(folder_path, folder_name)               # '../sig2img/32'
    dest_folder_name = os.path.join(dest_folder_path, folder_name)      # '../sig2img2/32'
    if not os.path.exists(dest_folder_name):
        os.makedirs(dest_folder_name)
    files_list = os.listdir(folders_path)                               # ['93.mat', '78.mat', '65.mat', .......]
100 in total
    random_list = random.sample(files_list, 50)                         # ['65.mat', '35.mat', ......] 50
in total random
    for each_file in random_list:
        files_path = os.path.join(folders_path, each_file)              # '../sig2img/32/93.mat'
        dest_files_path = os.path.join(dest_folder_name, each_file)     #
'../sig2img2/32/93.mat'
        shutil.move(files_path, dest_files_path)
```

We use CNN architectures to construct our classification model. For different execution paths, various code portions are exercised. This produces unique local patterns in the power signals, and therefore discriminative local features in the power images. Our model utilizes these local features to distinguish among different execution paths and thus profiling the execution status. During the training process, the most significant stage of the CNN model is the application of filters to the input of each layer which allows the model to learn useful features and feed them into the classifier. Filters vary in size depending on the features we want to extract [67]. Since the features we want our model to extract are not distinct enough from each other, we apply small size filters stacked in order to increase the depth of our network and allow it to learn more complex features. The final output of our first set of layers before the fully connected layer is a vector, holding a specific amount of features. The features are passed to a set of fully connected layers together with a softmax function to produce class scores. This is essentially a softmax classifier and the output class scores are the probability mass function (PMF) over all classes. The loss function used in our model is the cross-entropy loss. Using back propagation, our model's optimization algorithm adjusts the weights with respect to the loss function. The trained model is used to monitor the execution. Power signals captured at run-time are fed into the model. Class labels are computed for each query signal as their predicted execution path.

```
from __future__ import print_function
from scipy import io
import os
import numpy as np
import torch
import torch.utils.data as Data
import torch.nn as nn
from torch.autograd import Variable
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
#import scikitplot as skplt
import pandas as pd
import seaborn as sn
class CustomDataset(Data.Dataset):

    def __init__(self, dataset_dir):
        self.classes_path = dataset_dir                                    # initialize the path (dataset_dir)
of our dataset directory
        self.classes_list = os.listdir(self.classes_path)                  # ['class32', 'class15',
'class39', 'class4',.........]  total 41
        self.trace_list = []                                               # ['class18_53.mat', class18_12.mat,....]
 total 2050
        self.label_list = []                                               # ['class18', 'class18',....]            total
2050
        self.label_dict = {}                                               # {0: 'class36', 1: 'class18', ....}
total 41
        for i, each_class in enumerate(self.classes_list):
            self.label_dict[i] = each_class
            files_path = os.path.join(self.classes_path, each_class)
            files_list = os.listdir(files_path)
            self.trace_list += files_list
            for each_file in files_list:
                self.label_list.append(each_class)

    def __getitem__(self, item):
        label_path = os.path.join(self.classes_path, self.label_list[item])
        trace_path = os.path.join(label_path, self.trace_list[item])

        trace_array = (io.loadmat(trace_path))['img']
        trace = torch.from_numpy(np.expand_dims(trace_array, axis=0)).float()

        for key, values in self.label_dict.items():
            if values == self.label_list[item]:
                label_key = [key]
        label = torch.LongTensor(label_key)

        return trace, label

    def __len__(self):
        return len(self.trace_list)
```

```
train_data = '../sig2img_new'
train_dataset = CustomDataset(train_data)

test_data = '../sig2img_new2'
test_dataset = CustomDataset(test_data)

epoch_number = 20
batchSize = 32
learn_rate = 0.00003

train_loader = Data.DataLoader(train_dataset, batch_size = batchSize, shuffle= True)
test_loader = Data.DataLoader(test_dataset, batch_size = batchSize, shuffle= False)

class CNN_Model(nn.Module):
    def __init__(self):
        super(CNN_Model, self).__init__()
        self.features = nn.Sequential(
                nn.Conv2d(1, 32, kernel_size = 5, stride = 5),
                nn.ReLU(inplace = True),
                nn.Conv2d(32, 64, kernel_size = 4, stride = 2),
                nn.ReLU(inplace = True),
                nn.Conv2d(64, 128, kernel_size = 3, stride = 2),
                nn.ReLU(inplace = True),
                nn.Conv2d(128, 256, kernel_size = 3),
                nn.ReLU(inplace = True),
                nn.Conv2d(256, 256, kernel_size = 3),
                nn.ReLU(inplace = True),
                nn.MaxPool2d(kernel_size = 2, stride = 2)
                )
        self.classifier = nn.Sequential(
                #nn.Dropout(0.5),
                nn.Linear(256*10*10, 1024),
                nn.ReLU(inplace = True),
                #nn.Dropout(0.5),
                nn.Linear(1024, 512),
                nn.ReLU(inplace = True),
                nn.Linear(512, 41))
    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 256*10*10)
        x = self.classifier(x)
        return x


model = CNN_Model()
#model.load_state_dict(torch.load('../CNN_model'))
model.cuda()
print(model)

def train_model(train_loader):
    optimizer = torch.optim.Adam(model.parameters(), lr=learn_rate)
    criterion = nn.CrossEntropyLoss()
    total_loss = []
```

```
for epoch in range(epoch_number):
        instance_loss = 0.0
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            inputs, labels = inputs.cuda(), labels.cuda()
            inputs, labels = Variable(inputs), Variable(labels)
            labels = labels.view(-1)
            optimizer.zero_grad()
            output = model(inputs)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()
            instance_loss += loss.item()
            print('loss data', loss.item())
        print('instance', instance_loss)
        print('i', i + 1)
        print('Epoch: ', epoch, '| train loss: %.4f' % (instance_loss / (i + 1)))
        total_loss.append(instance_loss / (i+1))

    torch.save(model.state_dict(), '../CNN_model')

def test_model(test_loader):
    correct = 0
    total = 0
    y_pred = np.array([])
    y_true = np.array([])
    for data in test_loader:
        signals, labels = data
        signals, labels = signals.cuda(), labels.cuda()
        signals = Variable(signals)
        output = model(signals)
        labels = labels.view(-1)
        _, predicted = torch.max(output.data, 1)
        total += labels.size(0)
        y_pred = np.concatenate((y_pred, predicted.cpu().numpy()))
        y_true = np.concatenate((y_true, labels.cpu().numpy()))
        correct += (predicted == labels).sum()
    print(correct, total)
    print('Test Accuracy: %.2f' % (100 * float(correct) / float(total)))
    print('pred: ', y_pred)
    print('true: ', y_true)
    conf_matrix = confusion_matrix(y_true, y_pred)
    print(conf_matrix, conf_matrix.shape)
    for i in range(conf_matrix[0].size):
        for j in range(conf_matrix[1].size):
            conf_matrix[i, j] = (conf_matrix[i, j] / 50.0) * 100
    #skplt.metrics.plot_confusion_matrix(y_true, y_pred)
    cm = pd.DataFrame(conf_matrix, index=[i + 1 for i in range(41)], columns=[i + 1 for i in
range(41)])
    plt.figure(figsize=(10, 7))
    sn.set(font_scale=1.7)
    sn.heatmap(cm, annot=False, annot_kws={"size": 16})
    plt.ylabel('True class')
    plt.xlabel('Predicted class')
    plt.show()
```

1.1.11 Intrusion Detection

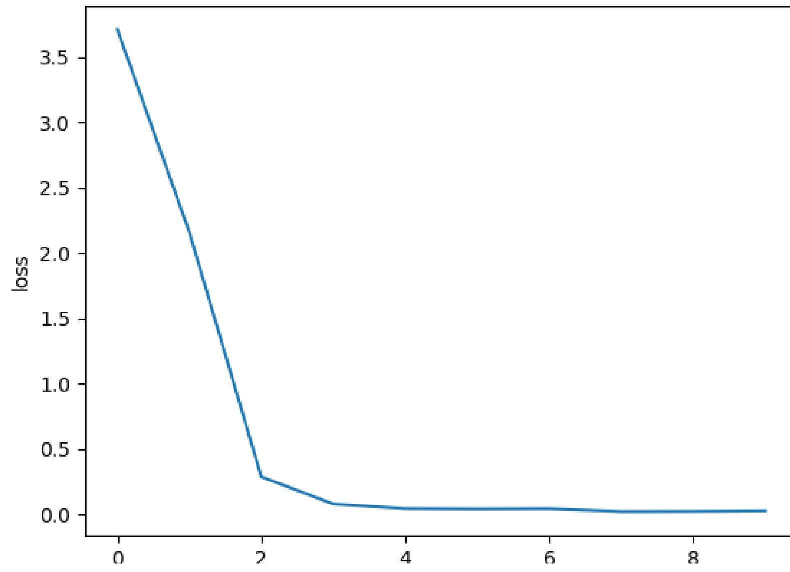
A PMF is also computed for each query signal. Legitimate executions will have power signals that match the model well, resulting in output PMFs that are significantly biased towards the actual execution path. We take the maximum probability as the likelihood score forthat signal. Anomalous executions deviate from the legitimate state (section III), causing some local patterns of the power signals to change, thus affecting the local features of the power images. This makes the anomalous power images not good matches for the model, thus the likelihood scores are smaller. By setting a threshold on the likelihood score, anomalous executions can be identified.

We also tested our system regarding its anomaly detection accuracy. We generated three type of anomalies: (i) Code replacement, which entirely replaces the legitimate code. In this case, we replaced the Dijkstra code with a new program that manipulates the I/O interfaces; (ii) Control flow deviation, which replaces the code from some point in the control flow. This is to produce the effect of a buffer overflow attack where adversaries inject malicious shellcode and direct the program execution pointer to it. Note that for control flow deviation attacks, program execution will not return to normal state after the malicious code execution. Again we injected shellcode that manipulates the I/O interfaces; (iii) Code injection, which injects shellcode to the code but does not terminate the legitimate code execution. Sensor data logging is an example of such attack. We injected a piece of code that constantly saves sensor data onto the SD card. These three types of anomalies represent three levels of changes on the power signals, from total replacement to only small changes. The detection difficulty is supposed to increase when the amount of changes is smaller. We collected 100 signals for each anomaly, accompanied by 100 legitimate signals.

5 Evaluation

We adopted the Alexnet image classification model as our execution profiling model. Specifically, we changed the input and output dimensions of the first few convolutional layers, since our input power images are of only one channel instead of three for natural images. We adjusted the filter size of the first few convolutional layers to be smaller in order to capture the fine-grained details in the power images. All pooling layers except the last one are removed for the same purpose. The model architecture is shown in Figure 3. On each box, we denote the type of layer. In case of Convolution layers (Conv), the filter's size is shown, followed by the stride (s5) and the filter's number. In case of Fully connected layer (FC), the number of neurons is shown. The activation function (ReLU) is applied after each of the two above layers. In case of pooling layer (Max Pooling) the size of kernel and stride are given.

To evaluate the execution monitoring performance of our system we used 2050 power images (50 for each execution path) as the training set and the same amount of data for testing. The entire training set is passed to the model 20 times. The experiments along with the research had been conducted at Rutgers University in New Jersey, in the Communications and Signal Processing Laboratory (CSPL). Equipment for the aqcuisition and the measurement of the power consumption signal was used frequently.



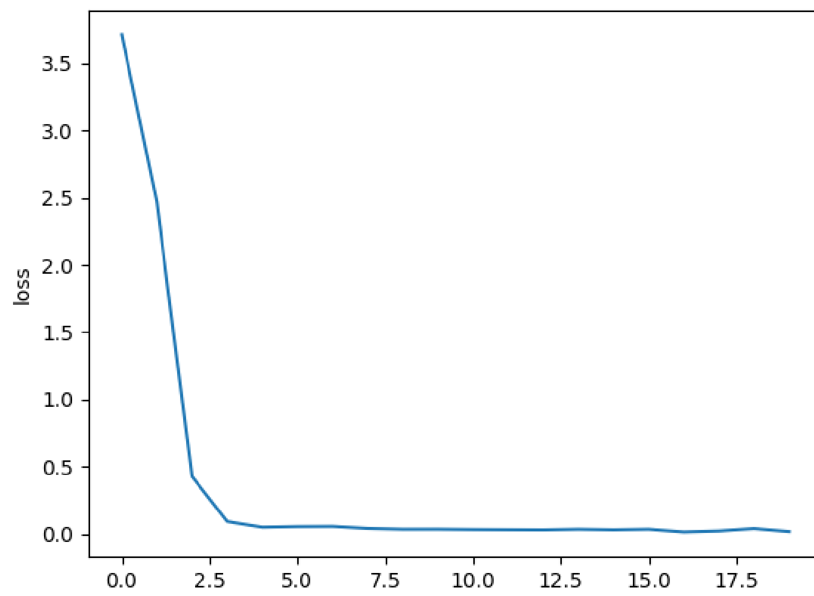


Figure 16. *Learning rate = 0.0003. Epochs 10 and 20*

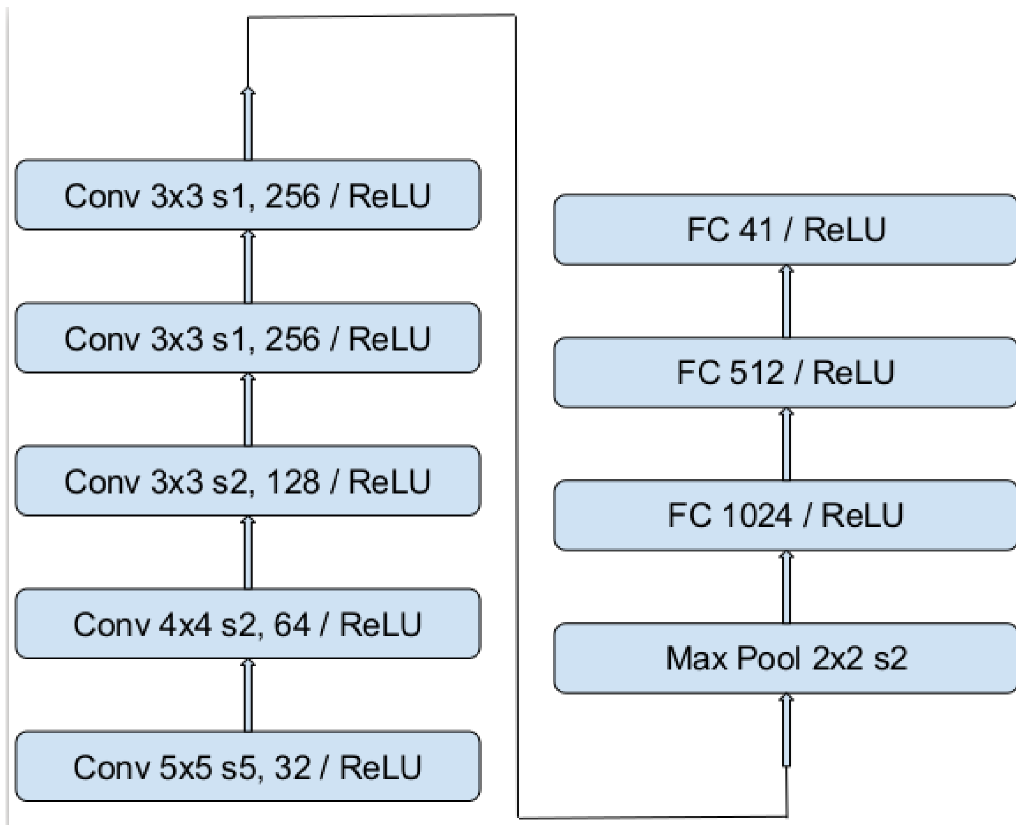
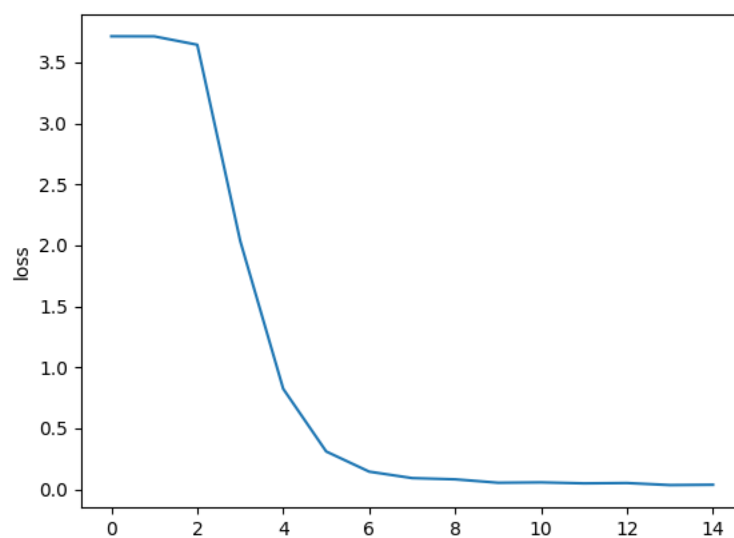
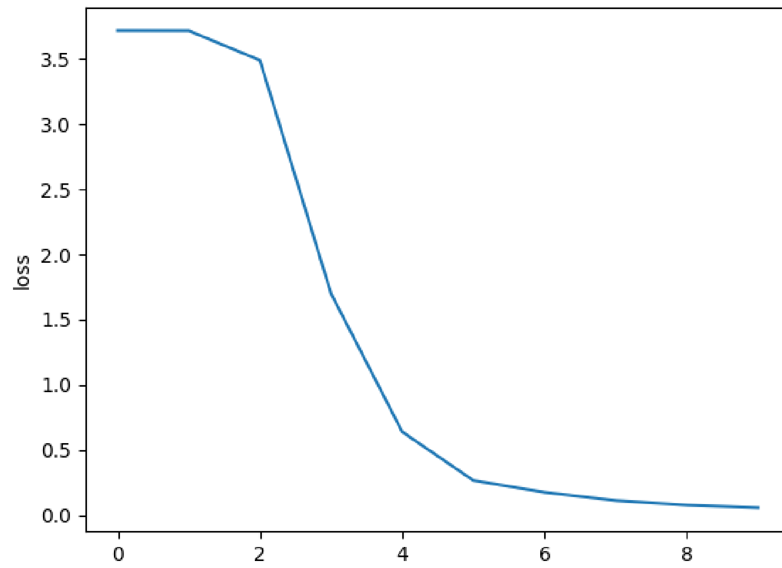


Figure 14. *CNN proposed architecture*

After several experiments based on this architecture we focused on tuning the hyper-parameters of the CNN. This is something that is based on testing and evaluation, without ensuring that the results will be good. The number of epochs, the size of the batch, the optimizer along with the learing rate were tested with different values combined all together.





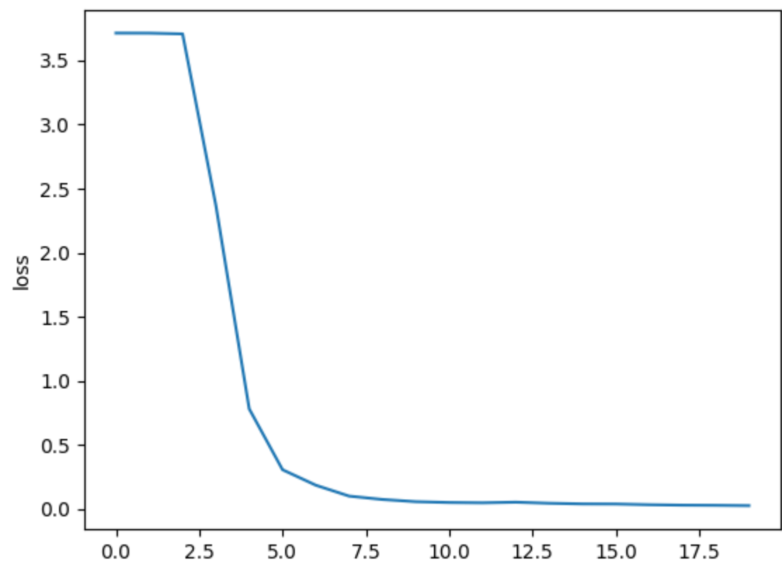


Figure 15. Learning rate = 0.0001. Epochs 10, 15, 20

The suggested number of epochs was 20 and it was enough for this amount of data we had to use for training. Learning rate was the main factor towards better results along with the batch size. The latter one helped us understand how important is during the training process to choose a proper value for it.

ROC curves of the detection results, where one can see that for all three anomalies the detection performance is good. Also, it is interesting to see that although the code injection only changes a very small portion of the power signal, perfect detection is still achieved. However, control flow deviation, which changes larger portion of the code, achieves a worse detection performance. This is because the decision boundaries of our model for different classes is so nonlinear that a tiny perturbation could drift the power image away from its true class thus result in a low likelihood score. Such phenomenon has been investigated in the topic of adversarial machine learning. Corresponding to the worse performance of control flow deviations, the likelihood scores indicate that signals are mispredicted as the 29th class with high confidence. The reason is that these signals happen to resemble legitimate power signals of that class.

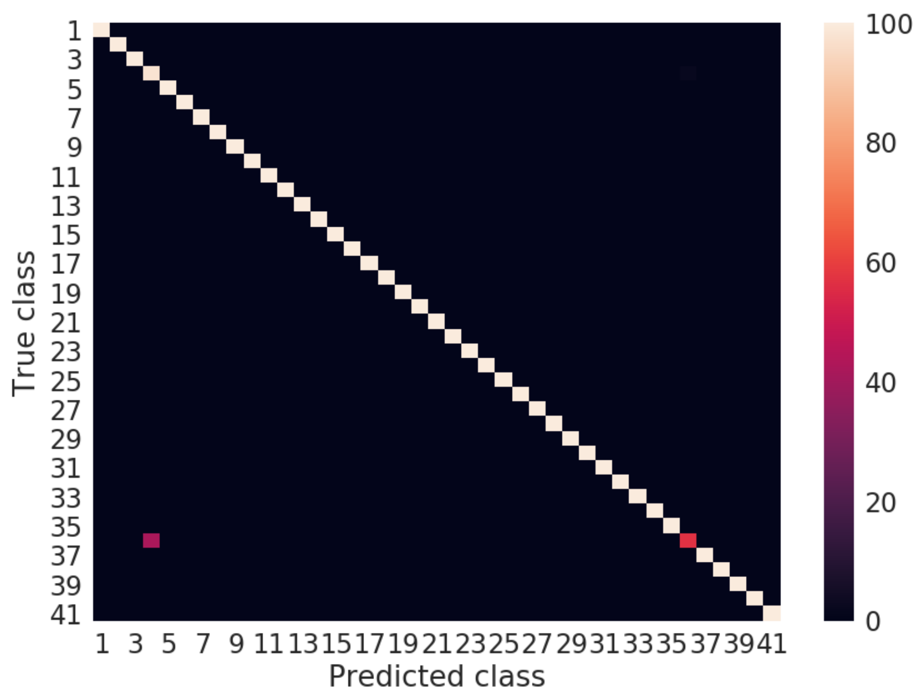


Figure 17. Confusion matrix

6 *Conclusions*

In this thesis we presented a deep learning model (CNN) based on which a high score classification on different code execution paths was delivered. PLCs and embedded devices in industry are lack of a proper monitoring when it comes to security against malicious threats. We saw how dangerous is to leave those devices unmonitored and what are the challenges to implement and deploy a proper monitoring system.

Using side channel analysis for monitoring it is more efficient in terms of cost and overhead but at the same time the signals themselves are hard to be captured without ovecoming some important issues like noise. Power consumption signals are promising in this field against other side channel signals and more easy to be captured without a major interference.

Creating a monitoring system using deep learning networks is becoming a reality as the last years CNNs have become really popular. We proved that CNNs can behave as good as other solutions. In fact in some cases they can perform even better.

Of course there are challenges when it comes to implementing a new or tuning an existing model. These limitations might be blocking more researchers to investigate time in this field.

In the future a proper tutorial on using deep learning methods for side channel analysis can be delivered and attract more scientits and researchers.

7 *Βιβλιογραφία*

[1] Han, Y., Etigowni, S., Liu, H., Zonouz, S., & Petropulu, A. (2017). Watch Me, but Dont Touch Me! Contactless Control Flow Monitoring via Electromagnetic Emanations. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS 17*. doi:10.1145/3133956.3134081

[2] Dale, B., & Dale, B. (2015, July 16). Eight Internet of Things Security Fails. Retrieved from http://observer.com/2015/07/eight-internet-of-things-security-fails

[3] Leyden, J. (2008, January 11). Polish teen derails tram after hacking train network. Retrieved from https://www.theregister.co.uk/2008/01/11/tram_hack/

[4] Slay, J., & Miller, M. (n.d.). Lessons Learned from the Maroochy Water Breach. IFIP International Federation for Information Processing Critical Infrastructure Protection,73-82. doi:10.1007/978-0-387-75462-8_6

[5] N. Falliere, L. O. Murchu, and E. Chien, “W32. stuxnet dossier,” White paper, Symantec Corp., Security Response, vol. 5, no. 6, p. 29, 2011.

[6] E. Chien, L. O’Murchu, and N. Falliere, “W32. duqu: The precursor to the next stuxnet.” in LEET, 2012.

[7] Garcia, L. A., Brasser, F., Cintuglu, M. H., Sadeghi, A., Mohammed, O., & Zonouz, S. A. (2017). Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. Proceedings 2017 Network and

Distributed System Security Symposium. doi:10.14722/ndss.2017.23313
[8] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, F. Zong, P. Qian, X. Wang, K. Chen, Y. Tian et al., "Understanding iot security through the data crystal ball: Where we are now and where we are going to be," arXiv preprint arXiv:1703.09809, 2017.
[9] Bertino, E., & Islam, N. (2017). Botnets and Internet of Things Security. Computer,50(2), 76-79. doi:10.1109/mc.2017.62
[10] Krebs, B. (2016). Who makes the iot things under attack. Krebs on Security.
[11] Mclaughlin, S., Zonouz, S., Pohly, D., & Mcdaniel, P. (2014). A Trusted Safety Verifier for Process Controller Code. Proceedings 2014 Network and Distributed System Security Symposium. doi:10.14722/ndss.2014.23043
[12] J. Mulder, M. Schwartz, M. Berg, J. R. Van Houten, J. Mario, M. A. K. Urrea, A. A. Clements, and J. Jacob, "Weaselboard: zero-day exploit detection for programmable logic controllers," Sandia report SAND2013-8274, Sandia national laboratories Google Scholar, 2013.
[13] Etigowni, S., Tian, D. (., Hernandez, G., Zonouz, S., & Butler, K. (2016). Cpac. Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC 16. doi:10.1145/2991079.2991126
[14] Liu, Y., Wei, L., Zhou, Z., Zhang, K., Xu, W., & Xu, Q. (2016). On Code Execution Tracking via Power Side-Channel. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS16. doi:10.1145/2976749.2978299
[15] Nazari, A., Sehatbakhsh, N., Alam, M., Zajic, A., & Prvulovic, M. (2017). Eddie. Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA 17. doi:10.1145/3079856.3080223
[16] Genkin, D., Pipman, I., & Tromer, E. (2015). Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. Journal of Cryptographic Engineering,5(2), 95-112. doi:10.1007/s13389-015-0100-7
[17] Cagli, E., Dumas, C., & Prouff, E. (2017). Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures. Lecture Notes in Computer Science Cryptographic Hardware and Embedded Systems – CHES 2017,45-68. doi:10.1007/978-3-319-66787-4_3
[18] Sehatbakhsh, N., Nazari, A., Zajic, A., & Prvulovic, M. (2016). Spectral profiling: Observer-effect-free profiling by monitoring EM emanations. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). doi:10.1109/micro.2016.7783762
[19] /@smallfishbigsea. (2017, October 10). Basic Concepts in GPU Computing - Hao Gao. Retrieved from https://medium.com/@smallfishbigsea/basic-

concepts-in-gpu-computing-3388710e9239

https://medium.com/@smallfishbigsea/basic-concepts-in-gpu-computing-3388710e9239, 2017.

[20] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/cvpr.2016.90

[21] Chatfield, K., Simonyan, K., Vedaldi, A., & Zisserman, A. (2014). Return of the Devil in the Details: Delving Deep into Convolutional Nets. Proceedings of the British Machine Vision Conference 2014. doi:10.5244/c.28.6

[22] Huang, L., Liu, X., Liu, Y., Lang, B., & Tao, D. (2017). Centered Weight Normalization in Accelerating Training of Deep Neural Networks. 2017 IEEE International Conference on Computer Vision (ICCV). doi:10.1109/iccv.2017.305

[23] Poernomo, A., & Kang, D. (2018). Biased Dropout and Crossmap Dropout: Learning towards effective Dropout regularization in convolutional neural network. Neural Networks,104, 60-67. doi:10.1016/j.neunet.2018.03.016CS231n: Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved from http://cs231n.stanford.edu/

[24] F. Li, J. Johnson, and S. Yeung, “Stanford cs231n: Convolutional neural networks for visual recognition,” http://cs231n.stanford.edu/, 2015.CS231n: Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved from http://cs231n.stanford.edu/

[25] Smashing The Stack For Fun And Profit Aleph One. (n.d.). Retrieved from http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

[26] A. Cui, M. Costello, and S. J. Stolfo, “When firmware modifications attack: A case study of embedded exploitation.” in NDSS, 2013.

[27] N. Carlini and D. Wagner, “Rop is still dangerous: Breaking modern defenses.” in USENIX Security Symposium, 2014, pp. 385–399.

[28] Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z. (2011). Jump-oriented programming. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS 11*. doi:10.1145/1966913.1966919

[29] Lerman, L., Medeiros, S. F., Bontempi, G., & Markowitch, O. (2014). A Machine Learning Approach Against a Masked AES. *Smart Card Research and Advanced Applications Lecture Notes in Computer Science*,61-75. doi:10.1007/978-3-319-14123-7_5

[30] Maghrebi, H., Portigliatti, T., & Prouff, E. (2016). Breaking Cryptographic

Implementations Using Deep Learning Techniques. *Security, Privacy, and Applied Cryptography Engineering Lecture Notes in Computer Science,*3-26. doi:10.1007/978-3-319-49445-6_1

[31] Bartkewitz, T., & Lemke-Rust, K. (2013). Efficient Template Attacks Based on Probabilistic Multi-class Support Vector Machines. *Smart Card Research and Advanced Applications Lecture Notes in Computer Science*,263-276. doi:10.1007/978-3-642-37288-9_18

[32] Qiao, Y., Xin, X., Bin, Y., & Ge, S. (2002). Anomaly intrusion detection method based on HMM. *Electronics Letters,38*(13), 663. doi:10.1049/el:20020467

[33] Apap, F., Honig, A., Hershkop, S., Eskin, E., & Stolfo, S. (2002). Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. *Lecture Notes in Computer Science Recent Advances in Intrusion Detection,*36-53. doi:10.1007/3-540-36084-0_3

[34] J. Ryan, M.-J. Lin, and R. Miikkulainen, “Intrusion detection with neural networks,” in Advances in neural information processing systems, 1998, pp. 943–949.

[35] Martinasek, Z., Malina, L., & Trasy, K. (2015). Profiling Power Analysis Attack Based on Multi-layer Perceptron Network. *Lecture Notes in Electrical Engineering Computational Problems in Science and Engineering,*317-339. doi:10.1007/978-3-319-15765-8_18

[36] Rabiner, L. R. (1990). A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Readings in Speech Recognition,*267-296. doi:10.1016/b978-0-08-051584-7.50027-9

[37] Xu, K., Yao, D. D., Ryder, B. G., & Tian, K. (2015). Probabilistic Program Modeling for High-Precision Anomaly Classification. *2015 IEEE 28th Computer Security Foundations Symposium*. doi:10.1109/csf.2015.37

[38] Session 3A on-chip networks-I. (2009). *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. doi:10.1109/hpca.2009.4798249

[39] JTAG Explained (finally!): Why "IoT" Makers, Software Security Folks, and Device Manufacturers Should Care. (n.d.). Retrieved from https://blog.senr.io/blog/jtag-explained

[40] Moura, L. D., & Bjørner, N. (2008). Z3: An Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science,337-340. doi:10.1007/978-3-540-78800-3_24

[41] Hospodar, G., Gierlichs, B., Mulder, E. D., Verbauwhede, I., & Vandewalle, J. (2011). Machine learning in side-channel analysis: A first study. Journal of Cryptographic Engineering,1(4), 293-302. doi:10.1007/s13389-011-

0023-x

[42] Lerman, L., Bontempi, G., & Markowitch, O. (2014). Power analysis attack: An approach based on machine learning. International Journal of Applied Cryptography,3(2), 97. doi:10.1504/ijact.2014.062722

[43] Martinasek, Z., Malina, L., & Trasy, K. (2015). Profiling Power Analysis Attack Based on Multi-layer Perceptron Network. Lecture Notes in Electrical Engineering Computational Problems in Science and Engineering,317-339. doi:10.1007/978-3-319-15765-8_18

[44] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation,9(8), 1735-1780. doi:10.1162/neco.1997.9.8.1735

[45] Cho, K., Merrienboer, B. V., Bahdanau, D., & Bengio, Y. (2014). On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation. doi:10.3115/v1/w14-4012

[46] Neary, P. (2018). Automatic Hyperparameter Tuning in Deep Convolutional Neural Networks Using Asynchronous Reinforcement Learning. 2018 IEEE International Conference on Cognitive Computing (ICCC). doi:10.1109/iccc.2018.00017

[47] Shibata, K. (2011). Emergence of Intelligence through Reinforcement Learning with a Neural Network. Advances in Reinforcement Learning. doi:10.5772/13443

[48] Maghrebi, H., Portigliatti, T., & Prouff, E. (2016). Breaking Cryptographic Implementations Using Deep Learning Techniques. Security, Privacy, and Applied Cryptography Engineering Lecture Notes in Computer Science,3-26. doi:10.1007/978-3-319-49445-6_1

[49] BISHOP, C. M. (2016). PATTERN RECOGNITION AND MACHINE LEARNING. Place of publication not identified: SPRINGER-VERLAG NEW YORK.

[50] Dougherty, G. (2013). Pattern Recognition and Classification: An introduction. New York, NY: Springer.

[51] Fausett, L. V. (1998). Fundamental of neural networks. Prentice Hall.

[52] Hebb, D. O. (2012). The organization of behavior: A neuropsychological theory. New York: Routledge, Taylor & Francis Group.

[53] Dartmouth College Summer Session. (1916). School Science and Mathematics,16(4), 372-372. doi:10.1111/j.1949-8594.1916.tb01644.x

[54] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review,65(6), 386-408. doi:10.1037/h0042519

[55] Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based

learning applied to document recognition. Proceedings of the IEEE,86(11), 2278-2324. doi:10.1109/5.726791

[56] Reviews of Books and Papers in the Computer Field. (1968). IEEE Transactions on Computers,C-17(1), 97-98. doi:10.1109/tc.1968.5008883

[57] Haykin, S. S. (2009). Neural networks and learning machines. Upper Saddle River: Prentice Hall.

[58] Hirose, Y., Yamashita, K., & Hijiya, S. (1991). Back-propagation algorithm which varies the number of hidden units. Neural Networks,4(1), 61-66. doi:10.1016/0893-6080(91)90032-z

[59] Goodfellow, I., Bengio, Y., & Courville, A. (2017). Deep learning. Cambridge, MA: MIT Press.

[60] Hinton, G. E. (2006). Reducing the Dimensionality of Data with Neural Networks. Science,313(5786), 504-507. doi:10.1126/science.1127647

[61] AHinton, G. E. (2012). A Practical Guide to Training Restricted Boltzmann Machines. Lecture Notes in Computer Science Neural Networks: Tricks of the Trade,599-619. doi:10.1007/978-3-642-35289-8_32

[62] Hinton, G. E., Osindero, S., & Teh, Y. (2006). A Fast Learning Algorithm for Deep Belief Nets. Neural Computation,18(7), 1527-1554. doi:10.1162/neco.2006.18.7.1527

[63] Bleiweiss, A. (2019). LSTM Neural Networks for Transfer Learning in Online Moderation of Abuse Context. Proceedings of the 11th International Conference on Agents and Artificial Intelligence. doi:10.5220/0007358701120122

[64] Mane, D. T., & Kulkarni, U. V. (2017). A Survey on Supervised Convolutional Neural Network and Its Major Applications. International Journal of Rough Sets and Data Analysis,4(3), 71-82. doi:10.4018/ijrsda.2017070105

[65] Fleet, D., Pajdla, T., Schiele, B., & Tuytelaars, T. (2014). Computer vision - ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014: Proceedings. Cham: Springer.

[66] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM,60(6), 84-90. doi:10.1145/3065386

[67] Aghdam, H. H., & Heravi, E. J. (2017). Visualizing Neural Networks.Guide to Convolutional Neural Networks,247-258. doi:10.1007/978-3-319-57550-6_7

[68] Carlini, N., & Wagner, D. (2017). Towards Evaluating the Robustness of Neural Networks. 2017 IEEE Symposium on Security and Privacy (SP). doi:10.1109/sp.2017.49