UNIVERSITÀ DEGLI STUDI DI VERONA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Classification of programs as binary files with LSTM

Relatore

Ch.ma Prof. *Mila Dalla Preda*

Correlatore

Dott. *Niccolò Marastoni*

Laureando

Paolo D'Arienzo, VR424402

Anno Accademico 2019/2020

Sommario

In questo lavoro viene affrontato il problema della program similarity tramite classificazione di file binari con una rete neurale ricorrente. I file binari sono rappresentati come immagini e ad essi sono stati applicati diversi offuscamenti. Lo scopo del lavoro è espandere ulteriormente il lavoro anticipato da Marastoni et al. in cui viene affrontato il problema della program similarity con lo sviluppo di una rete neurale convoluzionale. In questo lavoro sono state esplorate diverse soluzioni, sia su modelli ricorrenti che sulla gestione delle immagini in input. I risultati del lavoro mostrano come un modello ricorrente sia in grado di affrontare con successo questo tipo di problema e che sia importante espandere la nostra conoscenza su come siano rappresentate le istruzioni di un binario in un'immagine.

Abstract

In this work the problem of program similarity is addressed through the classification of binary files with recurrent neural network. The binary files are represented as images and different obfuscations have been applied to them. The aim of the work is to further expand the work anticipated by Marastoni et al. in which they tackle the problem of program similarity with the development of a convolutional neural network. In this work different solutions were explored, both on recurrent models and on the management of input images. The results of the work show how a recurrent model is able to successfully approach this type of problem, and that it is important to expand our knowledge on how to represent the instructions of a binary in an image.

Contents

Chapter 1

Introduction

Code classification is a type of code analysis with multiple application fields, such as: program similarity, malware detection, code plagiarism detection and code search.

The number of programs is constantly increasing: new code is written everyday and many piece of code are being reused - with or without consent. In a context where there is a need for code analysis for optimization or for system security purposes, spending time and resources to analyze code that already exists but in other forms is ineffective. For example, in recent years there has been a large increase in malware; however, for the most part these malware are not new, but are slightly modified versions of already known and existing malware. Knowing how to recognize a malware by its behavior and not by the exact instructions that compose it in nearly real-time is essential both in critical and non-critical systems.

Code is the syntactic representation of the semantic properties of a program, which are impossible to formalize, according to Rice's theorem[1]. Code analysis can be divided into static and dynamic analysis of programs; static analysis involves code analysis as-it-is, that is not executed, therefore it cannot be possible to see how it interacts with the system, while dynamic analysis examine the code by inferring on its interaction with the system at run-time. Classification involves predicting which class an item belongs to. Some classifiers are binary, others are multi-class, able to categorize an item into one of several categories. With code classification we refer to the categorization of codes with the aid of some automatic tool.

Several state of the art methods on static analysis can be applied to achieve classification, like OpCode instruction analysis, often subject to the language used, or analysis through Natural Language Processing methods, and many more. All these techniques are subject to various limitations, for example, in real world scenarios what is available to analysis very often is binary code only, therefore all those techniques that work on source code can be hampered or cannot even be applied. If dynamic analysis were to be performed on programs, some degree of abstraction should always be taken into account; there are programs which are environment-aware, that is they show a different behavior when executed in a controlled environment, or they could have behaviors supported by randomness that cannot be analyzed with certainty.

Several papers show works in which AI techniques are applied, but often under more or less strong assumptions regarding the code. Sometimes, these techniques don't even take into consideration obfuscated code.

What prompted the origin of this work is the paper *A deep learning approach to program similarity*[2]. In the aforementioned paper which is further explained in 3.1, a novel approach is proposed for program similarity detection, and it resulted in the development of a deep learning architecture that classifies binary files represented as images, encouraged by the strong success achieved by neural network on visual recognition, achieving an accuracy of 88% after 20000 iterations. The purpose of this work is to expand the study on obfuscated binaries using a type of neural network that can achieve data correlation over time, i.e. a recurrent neural network, in particular the long short-term memory architecture.

Chapter 2

Background

In this section I will present an introduction to the concepts and technologies utilized in this work.

2.1 Obfuscation

Obfuscations are program transformations whose purpose is to prevent or at least to obstruct the comprehension of the programs on which are applied, in order to avoid intellectual theft, misuse or not licensed use. Formally,

Definition 2.1.1. [3] Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source program P into a target program P'.
$P \xrightarrow{\mathcal{T}} P'$ is an *obfuscating transformation* if P and P' have the same *observable behavior*. More precisely, in order for $P \xrightarrow{\mathcal{T}} P'$ to be a legal obfuscating transformation the following conditions must hold:

- if P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

Obfuscation transformations can be categorized based on the information they target[4]; what will be discussed in the paper is basically:

- Data obfuscation, that modifies the form in which data is stored in a program;
- Control obfuscation, that transforms the control flow of an executable into a (more) difficult to analyze one.

2.1.1 Tigress

Tigress is a diversifier/obfuscator tool for the C language[5]. It takes in input a C program and returns in output a semantically equivalent C program, obfuscated according to the chosen transformation. A subset of 8 transformations are being utilized in the creation of the dataset used for the training of the neural network. The transformations employed are:

- `Flatten`, implements the flattening of the control flow structures of the program by replacing cycles (`for` and `while`) with a switch statement.
- `Split`, performs a split of a specified function in different sub-functions that combined together have the same behaviour of the original function.
- `RandomFuns`, adds several random functions to the original code.
- `EncodeArithmetic`, replaces arithmetic expressions with more complex expressions; it is applied on the increment in the `for` cycle too.
- `EncodeLiterals`, replace literal integers and strings with less obvious expressions.
- `InitOpaque`, data structures with precise invariants are added; this transformation changes the initialization of data structures, initializing one value at a time.
- `InitEntropy`, creates variables that will generate entropy.
- `InitImplicitFlow`, initialized the signal handlers for other transformations that rely on implicit flow.

2.2 Development environment

2.2.1 Google Colab

The vast majority of this project is written in Python and within Google Colaboratory, in short Google Colab, a free-to-use platform developed by Google Research[6]. Google Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs and TPUs[7] and allowing students and researchers to experiment with machine learning and data analysis. Selecting the GPU as runtime mode, Google Colab assigns to the user an Intel Xeon CPU at 2.30GHz, 34GB of storage space and a

GPU Tesla K80 with 12GB of RAM. At the same time, Google imposes time limits on use that for policy are not declared, that should be around 12 hours of use and 30 mins of idle time max. To avoid monopolization of resources, Google Colab limits the user to 2 GPUs notebook to work at a time and retains the right to suspend the access to the service if intensive use is detected. The neural networks of this work are developed with TensorFlow and Keras.

2.2.2 TensorFlow

Figure 2.1: TensorFlow logo

TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

TensorFlow is an end-to-end open source platform that was developed by Google for internal use and became free to use under Apache 2.0 License in 2015[8, 9]. It's a collection of libraries that were meant to provide support to developers in the development and deployment of machine learning and ML-powered applications. TensorFlow provides stable Python and C++ APIs. In late 2019 TensorFlow 2.x was released; although all the neural networks presented in this work have been designed for TensorFlow 1.x, little to no changes should be done to upgrade them to TensorFlow 2.x.

2.2.3 Keras

Keras is an open-source deep learning library written in Python[10]. The project was started in 2015 by a Google engineer, François Chollet, and gained a lot of popularity due to the friendliness of use, becoming today one of the most appreciated deep learning libraries. Keras as a frontend is capable of running atop different machine learning backend, such as TensorFlow, Theano or R. The neural networks presented in this work utilize the Keras API integrated into TensorFlow (as `tensorflow.keras`) instead of standalone Keras.

2.3 Machine Learning and Deep Learning

Machine Learning is a subfield of Artificial Intelligence; it take roots in the late ’50s and it is credited to Arthur Samuel, who formulated the idea that rather than teaching computers what they need to know to carry out tasks, it might be possible to teach them to learn for themselves. In a first phase called *training*, a large quantity of data is provided to a machine learning algorithm, and the algorithm finds patterns and features in order to build a mathematical model that can make decisions and predictions based on new data[11].
In recent years, the amount of data available has increased dramatically: since ML needs lot of data, it is therefore understandable the popularity and explosion of use of such technologies today. Machine Learning are divided in 3 categories:

- Unsupervised learning
- Supervised learning
- Reinforcement learning

The main purpose of supervised learning consists in inferring a model, starting from labelled training data (i.e. each data has a label that reports its typology). In turn, two type of models can be distinguished:

- Classification model, where the model categorizes data never seen before in training (identifying subjects in drawings, photos etc);
- Regression model, where the model can forecast future values of a series on which the model has been trained (stock market forecast, weather forecast etc).

The difference between the two models is the fact that the dependent attribute is numerical for regression and categorical for classification.

In reinforcement learning a system, called agent, improves its performance while interacting with the environment. Since the information about the current state of the environment typically also includes a so-called *reward signal*, we can think of reinforcement learning as a field related to supervised learning. However, this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Some examples of application of reinforcement learning are self-driving cars, or robot manipulations.

In unsupervised learning, there are no labeled data or data with known structure; with unsupervised learning techniques, we are able to explore the structure of the data to extract meaningful information without the guidance of a known outcome variable or reward function[12].

2.3.1 Neural Network

Artificial Neural Networks are a paradigm of machine learning. Neural networks have been going in and out of fashion since the '60s; they were a major area of research in both neuroscience and computer science until 1969 when they have been shutdown to surge again in the '80s. The inspiration, and thus the name, comes from the behavior of the brain as a network of units called neurons. Modeled vaguely on the human brain, neural networks consists of thousands of simple processing nodes called neurons that are densely connected (figure 2.2).

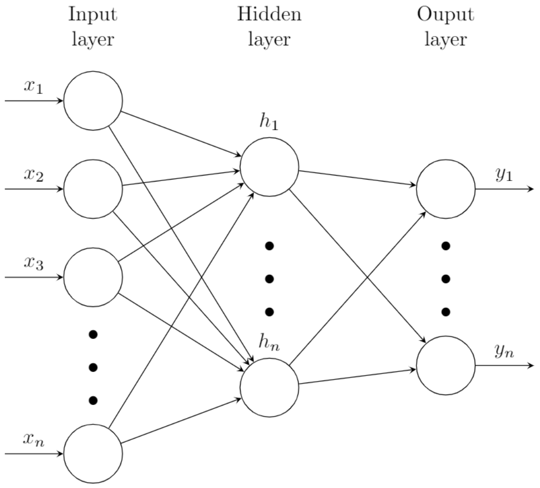


Figure 2.2: Representation of a neural network

These nodes are organized in layers, and they're *feed-forward*, meaning that the input is processed in only one direction, without cycles or loops. To each incoming connections, a node will assign a value known as *weight*; each node receives a different data item over each of its connections, it computes the weighted value by multiplying the data by the relative weight, then it adds the resulting products together for finally sending it as output (or to the next neuron)(figure 2.3)[13].

Deep Learning is a term that simply refers to a neural network that has more than one layer of neurons, and so this kind of networks are called *deep* to underline the presence of hidden layers.

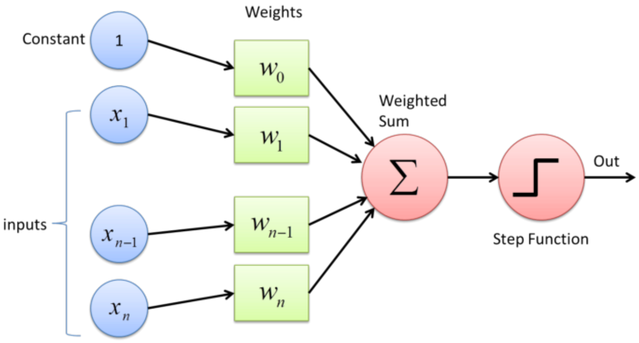


Figure 2.3: A perceptron, also called neuron[14]

2.3.2 Convolutional Neural Network

Convolutional neural networks (CNN) are a class of deep neural networks, mainly applied to visual analysis. As the name suggests, it is a type of neural network that employs convolution. Briefly, a convolution is a mathematical operation on two functions that produces a third function expressing how the shape of one is modified by the other. In image processing, the convolution operation plays an important role in edge detection and related processes.

Convolutional layer in a CNN convolve the input, that is an image as a matrix of pixel values, and pass its result to the next layer, thus identifying and extracting features from the input image.

2.4 Recurrent Neural Network

Recurrent neural network (RNN) are a class of deep artificial neural networks based on David Rumelhart's work in 1986[15] and have been an important focus of research and development during the '90s. In recent years, they reached huge success while applied to a variety of problems like language modeling, translation, image captioning etc. These networks are designed to be able to store information over time. A clarifying example should be an application of a RNN on text analysis: while reading a sentence, each word read gains meaning based on preceding words, whether they are verbs or subjects. RNNs manage to do this because they implement loops in them. As said in 2.3.1, neural networks process input without cycles, while RNNs have a feedback mechanism that enable them to remember previous values (figure 2.4[16]).

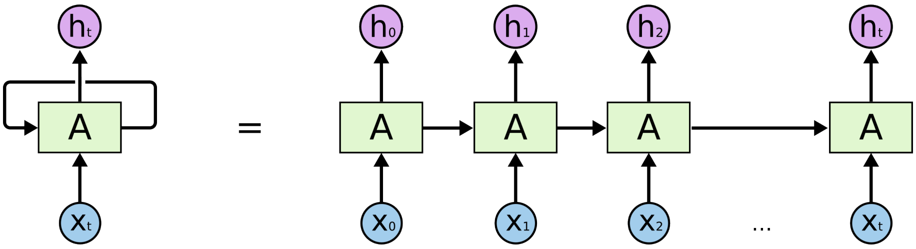


Figure 2.4: An unrolled recurrent neural network

A RNN is the perfect model to perform tasks in which the inputs should be correlated over time. But practical difficulties have been met when the inputs span on long intervals. These difficulties are related to the gradient descent, an optimization algorithm for finding a local minimum by iteratively moving in the direction of the steepest descent as defined by the negative of the gradient (the gradient of a function is the vector composed by partial derivatives of that function). One of these difficulties is known as *vanishing gradient problem*[17]. In particular, the vanishing gradient problem is encountered by neural network with gradient-based learning methods. The problem depends on the choice of the activation function - that is the function that defines the output of a neuron given a set of input -; in some cases, the gradient will be vanishingly small and thus preventing the update of the weight in a neuron, so that the RNN isn't learning anymore. Different architectures have been developed to overcome this problem, such as *Gated Recurrent Unit* (GRU) and *Long Short-Term Memory* (LSTM).

2.4.1 Long Short-Term Memory

Long Short-Term Memory is a kind of recurrent neural network capable of learning long-term dependencies. Introduced in 1997 by Hochreiter and Schmidhuber[18], they consist of a repeated module with a peculiar structure, shown in figure 2.5. In standard RNN, the module simply contains a $tanh$ layer, while the LSTM module is composed of a cell, an input gate, an output gate and a forget gate.

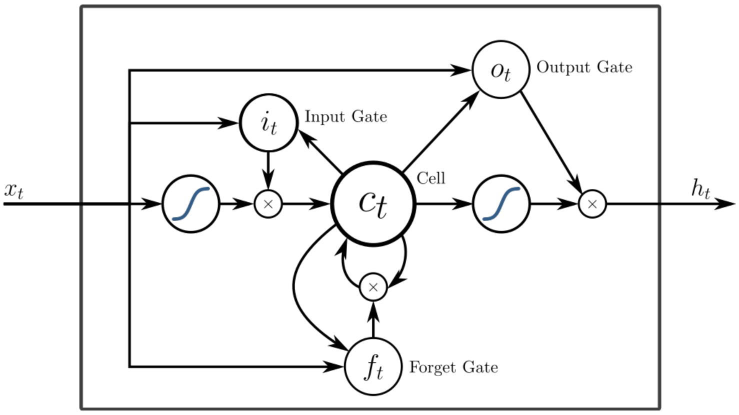


Figure 2.5: An LSTM unit

The LSTM has the ability to remove or add information to the cell state, where the information is stored, and this regulation is done by the structures called gates. Gates are often composed out of a logistic sigmoid function. The information received by an LSTM unit passes through the input gate that controls the extent to which a new value flows into the cell, afterwards it's combined to the value stored in the cell to then pass through the forget gate that will determine what will be stored in the cell. Finally, this value will pass through the output gate that controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit.

2.4.2 Bidirectional recurrent neural network

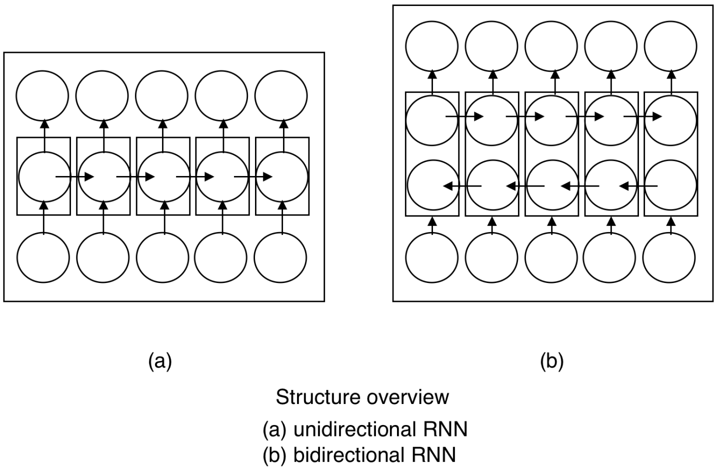


Figure 2.6: Structure of RNN and BRNN

Bidirectional recurrent neural networks were invented by Schuster et al. in 1997[19]; they make it possible to train recurrent neural networks in both time directions simultaneously, connecting two hidden layers of opposite directions to the same output (2.6). Different works benchmarked bidirectional RNN, LSTM and other kind of RNN and all came to conclusion that a bidirectional RNN outperform both in speed and in accuracy other architectures[20, 21].

2.5 Interpolation

Image interpolation occurs when an image is resized or distorted from one pixel grid to another. Interpolation works by using known data to estimate values at unknown points. It works in two directions, and tries to achieve a best approximation of a pixel's intensity based on the values at surrounding pixels. Common interpolation algorithms can be grouped into two categories: adaptive and non-adaptive. Adaptive methods change depending on what they are interpolating, whereas non-adaptive methods treat all pixels equally. Non-adaptive algorithms include: nearest neighbor, bilinear, bicubic, spline, sinc, lanczos and others[22].

The methods that were taken into consideration in this work are the methods made available in the OpenCV library, which are[23]:

- *inter_nearest* – a nearest-neighbor interpolation;
- *inter_linear* – a bilinear interpolation;
- *inter_area* – resampling using pixel area relation; it may be a preferred method for image decimation, as it gives moire'-free results, but when the image is zoomed, it is similar to the *inter_nearest* method.
- *inter_cubic* – a bicubic interpolation over 4×4 pixel neighborhood;
- *inter_lanczos4* – a lanczos interpolation over 8×8 pixel neighborhood.

Chapter 3

Related works

As explained in chapter 1, code analysis lends itself to different applications such as code suggestions and completion, program translation, program similarity, plagiarism detection, code optimization and much more. Since ’70s, a lot of tools have been introduced to measure the similarity of source code. Generally, these tools are classified into metric-based, text-based, token-based, tree-based, and graph-based approaches[24]. In this chapter I will present some related works on program similarity.

In the work "Program Similarity: Techniques and Applications", a PhD dissertation by L. Nichols[25], the author proposes a method to detect cross-language clones that, he claims, outperforms other methods of cross-language clone detection and can be successfully applied to single-language code detection. That paper analyzes the limits of other cross-language detection tools concerning the syntactic structure (i.e. parse tree) of the different languages involved, which can be very different but appear similar at source level. To work around the problem, other works enforce the syntactical structure for similar code with some solutions or they limit themselves to "nominal" analysis of clone detection, that is analysis of variable names and other user-defined abstractions. In the work of L. Nichols, he presents a method for enabling structural matching for cross-language clones, even in those cases where the syntactic structure is different, and presents a method for composing both structural and nominal matching into a singular matcher. The approach consists in the construction of a tool that extracts the parse tree from each function and simplifies it, abstract these obtained parse trees into a common representation and finally comparing the trees to present the pairwise similarity score to the user. The author then presents a method for plagiarism detection using the same algorithm that although it is

syntax-aware, it is not tied to the syntax of any particular language.
On semantic clone detection, usually program dependence graphs are used to expose non-syntactic similarities between code fragments. In the paper[25], a new approach to heuristic for finding similarities is presented, i.e. "semantically similar programs ask similar questions with similar frequencies", that computes probability distribution over paths in a given code fragment and then judge whether two code fragments are semantic clones by comparing their path distributions to see if they are "close enough".

In "A Novel Graph-Based Program Representation for Java Code Plagiarism Detection"[26], Cheers and Lin propose a novel graph-based representation of Java programs resilient to plagiarism-hiding transformations. They introduce the Program Interaction Dependency graph (PIDG), that represents the interaction and transformation of data within a program, and how these data interact with the system. They based the work on the assumption that if two plagiarized programs are the same, they will have similar execution behaviors and hence similar representations as a PIDG. The PIDG is constructed in two steps, firstly the program source code and all the required libraries are provided to a symbolic execution tool; then, the tool executes the program and generates a set of execution traces. Secondly, a distinct PIDG is built representing the behavior of each path. Finally, the similarity is measured based on the similarities between PIDGs of each program. This approach is clearly language-dependent; moreover, as noted in the paper, graph isomorphism on big graphs could reach high computational complexity, and they have developed a way to approach this problem that can be quickly summarized as a sub-graph isomorphism from particular starting points that they find to be semantically-significant.

In their paper, Sudhamani and Rangarajan[27] propose a code similarity detection method based on control statement and program features. The proposed model basically has five important stages: pre-processing, computation of features, dissimilarity matrix computation, similarity value computation and similar program detection through clustering. They reach a good accuracy value, although they work on the assumption of available source code and do not consider obfuscated code, that could easily nullify the efficiency of the algorithm.

A paper that takes obfuscation into account and that is more generic, so in the sense that it does not depend on a particular language or platform, is "Common program similarity metric method for anti-obfuscation"[28]. In this paper is presented the new concept of Reductive Instruction Dependence Graph (RIDG) that is platform independent and stable through most code obfuscation processes. In the paper is then proposed a four-level similarity schema based on RIDG for measuring program similarity: RIDGs on level 1, basic blocks on level 2, functions on level 3 and programs on level 4. RIDG is a graph that comes from the structure of the code, i.e. a transitive reduction of an instruction dependency graph. The similarity schema proposed makes it possible to evaluate the similarity between the various components of two programs. This approach is based on finding the maximum common subgraph between two RIDGs, which, however, is an NP-complete problem. Moreover, although the platform-independent method, they assert that finding dependency relations among assembly instructions is more challenging.

In [29], the approach is based on a neural network system for detection of malicious code that is trained on code as natural language. The architecture of the deep neural network involves convolutional and recurrent layers, and the input is the source code of programs treated as text in NLP (natural language processing) techniques.

In [30] is presented a hierarchical convolutional network for malware classification on assembly language source code. It is composed by two level of convolutional block (which in turn are composed of several convolutional layers): the input is processed by the first block that operates on mnemonic-level for feature extraction, and then its results are passed to the second convolutional block that works at function-level of the code, extracting then ngram like features from both levels. This unites syntactic analysis of code with register analysis and analysis of jump instructions and called functions, that could differ in position in semantically equivalent programs.

In [31], a deep neural network of stacked LSTM is implemented; the architecture consists of a mixed one of unsupervised pre-training and supervised training. Pre-training on training data is for the purpose of finding weight in a unsupervised manner and then fine-tuning the network in a supervised network for classifying malware. The training malware are analyzed as OpCode treated as pure text analysis with window length of 1024 bits. Moreover, it is used an embedding vector to represent datasets that contain system call or OpCodes of the samples.

In [32], a Restricted Boltzmann Machine and a Deep Belief Network (DBN) are used to classify malware of Android applications into families. A texture-fingerprint based approach is proposed to extract or detect the feature of malware content. A malware has a unique "image texture" in feature spatial relations. The method uses information on texture image extracted from malicious or benign code, which are mapped to uncompressed gray-scale according to the texture image-based approach. The approach works on executables that are converted into byteplot images that are later analyzed by the DBN. Their method is robust to code obfuscation and does not require unpacking or decryption, but the focus of the work is on Android executables (APK), there is a pre-training on the inputs and the focus of the research is only on 5 API calls.

Other approaches could regard graph analysis, like in [33], in which it is built a system that uses a deep graph convolutional neural network to embed structural information inherent in CFGs for malware classification. From each assembly code, a Control Flow Graph is constructed, then an attributed CFG is developed associating attributes of the code to the CFG, to be finally passed as input in the deep graph convolutional network for classification.

In [34], the study proposes a malware classification model over the Microsoft Malware Classification Challenge dataset. The model generates images from malware data: from larger malware data, more images are generated; all the images have a fixed size of 256x256. They used an unspecified method in order to adjust the images to the fixed size. The generated images are classified into one of the 9 families of malware by using a RNN and a CNN. In this study, there was a disadvantage regarding the impossibility of direct classification of the images, and an additional process of extracting feature points was needed. They experienced performance degradation due to the imbalance of samples in the datasets used. Moreover, the representation of the binary data as images is not very well explained.

3.1 A deep learning approach to program similarity

In "A deep learning approach to program similarity", by N. Marastoni, R. Giacobazzi and M. Dalla Preda[2], the authors present a method to achieve program similarity through deep learning techniques.
The dataset used in the work is composed of 47 programs: 32 simple programs and 15 programs extracted from the *Google Code Jam* competition. The simple dataset is listed below:

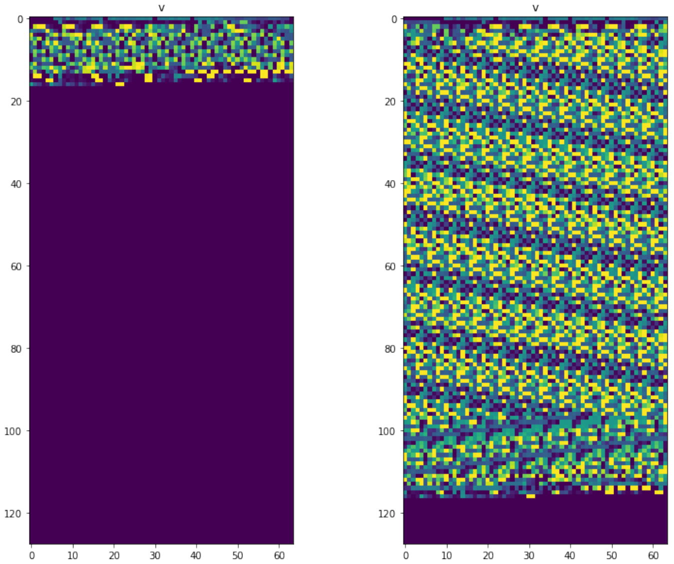
1. armstrong_n.c
2. calculator.c
3. char_frequency.c
4. count_digits.c
5. count_vowels.c
6. factorial.c
7. factorial_rec.c
8. factors.c
9. fib_1.c
10. fib_2.c
11. gcd.c
12. gcd_rec.c
13. hello_world.c
14. lcm.c
15. leap_year.c
16. n_is_palindrome.c
17. n_is_prime.c
18. n_is_sum_of_primes.c
19. positive_or_negative.c
20. power_n.c
21. prime_n_intervals.c
22. pyramid.c
23. quotient_reminder.c
24. remove_char.c
25. reverse_integer.c
26. store_struct.c
27. strcat.c
28. strcpy.c
29. stringsort.c
30. strlen.c
31. sum.c
32. times_table.c

The 15 samples from solutions submitted to the *Google Code Jam* between 2008 and 2011 are here listed:

33. alien_language.c
34. bot_trust.c
35. candy_splitting.c
36. fire_warning.c
37. fly_swatter.c
38. magicka.c
39. minimum_scalar_product.c
40. multibase_happiness.c
41. rotate.c
42. saving_the_universe.c
43. snapper_chain.c
44. theme_park.c
45. train_time_table.c
46. watersheds.c
47. welcome_to_codejam.c

for a total of 47 programs.

This initial dataset has been augmented by creating semantically equivalent copies of each program that however differ syntactically. To do so, 8 different obfuscation transformations are applied through Tigress (presented in 2.1.1). All the possible combinations of the obfuscations without repetitions are $\binom{8}{k}$, and up to 3 obfuscations per program could be applied, based on the size of the desired augmented dataset. Each obfuscated file is then compiled and the binary obtained is stripped of its header. Each binary is afterwards transformed in a `numpy` array through which it was possible to represent them as images (figure 3.1).



(a) Sum.c InitImplicitFlow-EncodeLiterals

(b) Positive_or_negative.c Flatten-InitOpaque-EncodeArithmetic

Figure 3.1: 2 examples of binary files as images

Each program has different size thus resulting in a different matrix size: different approaches have been taken into consideration, and the best model of normalization of image dimensions has been images of size 596x64, with 0-padding of smaller images.

The neural network developed is a convolutional neural network with 2 convolutional layers, each followed by a max-pooling layer. the final accuracy achieved was 88% over 20000 iterations.

Chapter 4

Work overview

What prompted the origin of this work is the paper *A deep learning approach to program similarity*[2]. In the aforementioned paper which is explained in 3.1, a novel approach is proposed for program similarity detection, and it resulted in the development of a deep learning architecture that classifies binary files represented as images, achieving an accuracy of 88% after 20000 iterations.
This work is the ideological prosecution of the paper by Marastoni et al. In this project I firstly approached the neural network architectures, testing a simple neural network and a convolutional neural network on two simple databases, MNIST and FASHION MNIST, obtaining results that I used as a reference to understand the achievements of the subsequent developed recurrent networks.
During the work, I developed a recurring LSTM model and a recurrent convolutional model, testing them on the MNIST and FASHION MNIST databases. Then I moved on to the development and the research of the best models capable of working on the binaries as images. The time and memory limits to which I was obliged by Google Colab pushed me to adopt some solutions rather than others. Finally, an analysis of the results obtained from the models was performed to understand which programs or obfuscations slowed down the classification.

Chapter 5

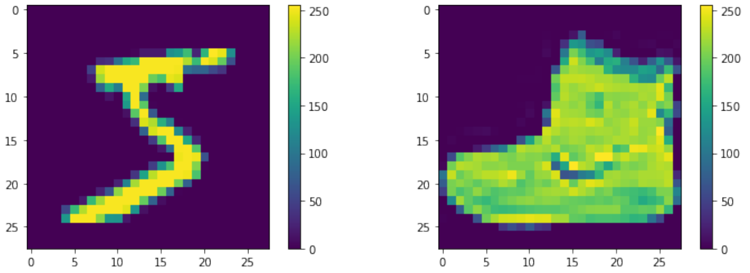
Implementation of recurrent models

The implementation of all the neural networks are coded in Python within Google Colab. All the work done is publicly available on my GitHub Repository[35].

Before tackling the problem of binary classification, the neural network technologies were studied on simpler databases, specifically on:

- MNIST database[36], a ready-to-use database for handwritten digits, with 60000 28x28 images in the training set and 10000 images in the test set, with their respective labels. Each images is composed of values in range 0-255 while the labels denote an integer between 0 and 9;

- FASHION MNIST database[37], with the same structure as the MNIST database, but composed of 10 items of clothing.



(a) MNIST sample (5)

(b) FASHION MNIST sample (boot)

Figure 5.1: 2 samples from MNIST and FASHION MNIST

On these databases were tested a shallow neural network with 2 hidden layers and a convolutional neural network with an hidden convolutional layer and an hidden max-pooling layer, obtaining results that served as reference values for comparison with the recurrent neural network models.

RNN time steps Before going into the explanation of the RNN models, it is necessary to understand how the parameters of a RNN are identified. We must indicate to the RNN model the number of *time steps*, i.e. the number of times the input is cyclically passed to a cell, as shown in figure 2.4. The number of time steps is associated with the number of elements we want to be remembered, and for each time step it is necessary to specify the number of features that composes a single time step. Since MNIST images are 28x28 in size, the most natural choice is wanting to remember one row at a time, therefore 28 should be the number of features composing each time step, bringing the number of time steps to be, obviously, 28; in this way in each time step a whole row is read, and at the 28th time step, the whole image has been processed.

Number of hidden layers and of nodes There isn't a methodical approach for selecting the number of hidden layers and the number of nodes that compose a layer. As a general rule, a model without hidden layers is only capable of representing linear separable functions or decisions; with one hidden layer, it can approximate any function that contains a continuous mapping from one finite space to another; with two hidden layers, it can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy[38]. Regarding the number of nodes[38], it is a common procedure to follow a schema such as the following:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be $2/3$ the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

LSTM MNIST parameters The number of layers for the LSTM model on MNIST has been selected to be 3, the first layer and the first hidden layer as LSTM layer and the last layer as a fully connected one; the number of nodes for the first layer was selected to be equal to the batch size, 128, progressively reduced until the third layer with the number of nodes as the number of classes, 10, and it was trained for only 5 epochs. With such a ridiculously little amount of epochs, the accuracy reached by the LSTM model was able to rival the accuracy achieved by the simple neural network, i.e. around 98%. Allowing the LSTM network to use more epochs for training, while still trying to avoid overfitting, could easily allow the classification of the MNIST database to reach over 99%.

Convolutional LSTM model Before speaking about the convolutional LSTM model, we need to introduce how the input of a convolutional recurrent model should be. The input of an LSTM cell consists of a set of data through time, namely a 3-dimensional vector in the shape `[samples, time_steps, features]`; the input of a convolutional layer is a 4-dimensional tensor `[samples, rows, columns, channels]`, where `(rows, columns, channels)` are the descriptor of an image. So the input of a convolutional LSTM is a 5-dimensional tensor in the shape `[samples, time_steps, rows, columns, channels]`, i.e. in each time step is selected an image. In the MNIST, as well as in the binary dataset, there are no images intended over time, as they were frames of a video. So we have to design the sub-images that we want to be remembered, that constitute the image in input.
At the state of the art, there are no known works that use binary as images in a recurrent convolutional neural network model. We have little knowledge of the structure of an obfuscated binary as image, hence the idea of a neural network. In the paper in 3.1, the study showed that the best width of the image should be 64, mainly because it represents the maximum bit size that an instruction can take in a 64 bit system.

CONVLSTM MNIST parameters the CONVLSTM model is composed of a convolutional LSTM layer with 32 nodes with a kernel mask 3x3, followed by a flatten layer, that is just an input manipulation layer for the third and last layer, and then a fully connected layer of 10 nodes. As said previously, we need to identify the sub-images that constitute an input image; for the dimensions of these, different measures could be chosen. Reasoning on the number of time steps, if we decide that the image is composed of 4 sub-images, this means that we choose 4 as time steps and, wanting to keep the sub-images squared, we can indicate the sub-images as 14x14x1. The architecture is robust enough to perform well in any chosen set of parameters. Similar to the LSTM MNIST model, the CONVLSTM MNIST model is limited to only 5 epochs and it can still reach an accuracy of over 96%.

CuDNN With CuDNN[39] we refer to the NVIDIA CuDNN library, a GPU-accelerated library of deep neural network primitives developed by NVIDIA which makes use of the CUDA technology. Keras provides CuDNN API that guarantees the same accuracy of LSTM API but in astonishingly less time. In fact, in TensorFlow 2.x CuDNN became the standard implementation for LSTM API. All the LSTM neural networks that have been developed in this work utilize the CuDNN API. There isn't a CuDNN implementation yet for the convolutional LSTM API in Keras.

Activation functions The activation function of all the layers except the output one is the rectified linear unit (ReLU), while for the output layer I chose the softmax function.

5.1 Dataset and pre-processing of data

The database is composed of 47 programs, augmented through the application of 8 obfuscation transformations (described in 2.1.1). These transformations can be stacked: depending on the limit of transformations indicated during the database generation, it is possible to obtain databases of different sizes. In this way, the 3 databases on which the models were trained were generated, with a limit of 200, 300 and 400 obfuscations, i.e. with 9400, 14100 and 18800 samples.
The binaries are saved on file as `numpy arrays` of integers between 0 and 65535. The database is loaded in the notebooks image by image with `numpy.load(filename)`. At this point, each image is resized accordingly to the chosen parameters, as better explained in 5.1.1. The database is subsequently randomly shuffled and divided into 3 datasets: training set, test set, validation set[40]. In the literature there are some confusions about the nomenclature of test set and validation set; in this case, the test set is the one employed in the progress analysis of each epoch, while the validation set is the unbiased set that the model has never seen before, and all the accuracy here reported were generated through this set. The database is shuffled randomly for each model run. Images have been then normalized in range 0-1 and saved as `numpy array`. The labels, on the other hand, have been coded (through scikit-learn `LabelEncoder` API) with indexes between 0 and 46. In theory, it's not necessary to normalize data but in practice, if numeric data is not normalized and the magnitudes of two predictors are far apart, then a change in the value of a neural network weight has far more relative influence on the x-value with larger magnitudes[41, 42]. To not lose precision, the images are `numpy arrays` of `float32` values.

5.1.1 Images pre-processing

The database is composed of 47 several programs that differ in size and in complexity of implementation, in particular the database ranges from a program that prints a single string to programs that make an advanced use of data structures. The application of some obfuscations, moreover, affects the length of the binary files in different ways. The width of the images - the number of columns of the matrix - is fixed at 64 based on the conclusions of the work in 3.1. Therefore, while this dimension is constant, the length of the binary files can vary a lot. I performed an analysis on the structure of the databases and I noticed that the average size of the file length is around 420, but only 4 classes, specifically *alien_language*, *fair_warning*, *magicka* and *watersheds*, far exceed the average value. As it can be noted in figure A.3 in appendix A, most of the files have a modest length, below 128. Since the neural model needs inputs of the same size, all the images should be reshaped to a common size; due to memory limits it was not possible to 0-pad all the images to the biggest dimension: the best solution that respects both memory and time constraints is 512. Furthermore, most of the files do not exceed 128 in length, so the developed models have been tested with images of dimensions 512x64 and 128x64. Images exceeding the chosen size are truncated, and in fact some information is lost. It can be noted that all the values considered for the images are all values of the power of 2.
Another solution has been devised in order to be able to take into account to some extent all the information of an image, i.e. the interpolation. Although, as it will be shown, this approach has been found to be effective in some respects, it introduces some degree of abstraction, since the images are quite far from the size they are interpolated to; for example, if the common size chosen is 512x64, these values are far from the dimensions of a 64x64 image, and equally, if not more so, for larger images such as 4096x64 of size.

Interpolation method It is not an easy task to identify an optimal interpolation method over others, as each of the different methods can perform better or worse if the image is reshaped bigger or smaller. Also, these methods were studied and optimized to work on real images, in which there are defined borders and so on. The choice of the final interpolation method, given the very similar results obtained by models on different interpolation methodologies which can be seen in figure A.1 in the appendix A (on both image size of 128x64 and 512x64), fell on the *inter_cubic* method.

5.2 Implementation of models

As said in 2.4.2, a bidirectional model outperforms an unidirectional one. A comparison benchmark was performed on an LSTM architecture with binary as images and it confirmed the superiority of the bidirectional model (figure A.4). Keras supports different merge mode of the forward and backward outputs; the merge mode selected for the bidirectional networks, after an empirical research, was the `concat` method.

5.2.1 LSTM

In this section I will describe the implementation of the bidirectional LSTM model.

Time steps

In the choice of the size of the time steps, and consequently the number of them, it was used the same approach for the choice of the size of time steps of the model trained on the MNIST database, i.e. relative to the width of the images. All the binary images have a width of 64, therefore the first choice of that parameter was simply 64. All the possibilities for the number of time steps were then explored, from 1 to 128 as power of 2. Let an image be of size $MxN = V$; if we indicate the number of time steps as V, it means processing the image one pixel at a time; this approach is very heavy computationally and isn't effective, because a single pixel doesn't represent the size of an instruction; moreover, the RNN, despite the presence of the forget gate, is forced to remember too many values. On the contrary, having a time step of 1 means processing the image at an almost equal level to a shallow neural network, without remembering anything. The best values for the number of time steps were found to be 32, 64 and 128. Among these values, the most performing parameter in terms of memory and time was found to be 64, with good performance on both 128x64 and 512x64 image dimensions (figure A.2).

LSTM model

The LSTM model, as said in previous sections, is a bidirectional CuDNN model. It is composed of a first LSTM layer with 141 nodes, an hidden layer with 94 nodes, and lastly an output layer with 47 nodes. The number of nodes is in accordance with one of the schemes presented in 5, and these values where chosen because they are multiple of 47. Both LSTM layers have the parameter `unit_forge_bias` set to `True`. This parameter, when true, adds 1 to the bias of the forget gate, and set the `bias_initializer` to zero, the parameter that initialize bias vector. Having `unit_forge_bias` to true should allows a faster learning, as reported in [43].

5.2.2 Convolutional LSTM

In this section I will describe the implementation of the bidirectional convolutional LSTM model. Unlike the LSTM model, whose input where images of 2 kinds of dimensions, 512x64 e 128x64, with this convolutional model only the 128x64 size is taken into consideration, in order to respect the time constraints imposed by Google Colab.

Time steps

For convolutional recurrent models, there is a particular image manipulation needed: since the convolutional LSTM needs to analyze images related to time, these should necessarily be derived from a single input image. That is, for each image input, we should delimit the sub-images that compose it, on which the convolution related to time will be performed. We can keep the third parameter of the images, the number of channels, equal to 1 as in black and white images, or else we can take advantage of it to reshape each image in a 3D tensor. All the dimensions tested have been designed to be square matrices. Different dimensions of the sub-images were tested and were identified two promising settings: with sub-images of the dimension (16 16, 1) and of the dimension (32, 32, 1). Using the channel with a different value than 1, a promising setting identified was (8, 8, 8).

Convolutional LSTM model

The convolutional LSTM consists of a convolutional LSTM layer (`CONVLSTM2D`) with 141 nodes and a kernel of dimension 3x3, with `unit_forget_bias` set to `true`, followed by a flatten layer, that is a data manipulation layer only, and lastly a fully connected layer with 47 nodes.

Chapter 6

Results

In this chapter I will list the final results of each model, with a focus on the analysis of the results of the best model.

The developed models are fundamentally 2, LSTM and convolutional LSTM, and each model was trained on images of different size, with different parameters. The results achieved are consistent across the databases of the three dimensions mentioned before (in 5.1), but the bigger the size of the database, the better the accuracy, with a difference of up to 4 percentage points between the smallest and the biggest database. All the displayed results refer to the database of size 18800. Although the accuracy between the different models varies, the learning difficulties of each model remain stable, so the analysis of classification errors was more focused on the model with the highest accuracy value.

LSTM	CONV-LSTM
0-PADDING	***0-PADDING***
(512, 64)	(16, 8, 8, 8)
(128, 64)	(32, 16, 16, 1)
(64, 64)	(8, 32, 32, 1)
INTERPOLATION	***INTERPOLATION***
(512, 64)	(16, 8, 8, 8)
(128, 64)	(32, 16, 16, 1)
(64, 64)	(8, 32, 32, 1)

Table I: All models implemented

In table I are indicated all the models implemented. The LSTM models with images $(64, 64)$ were employed as a yardstick to be able to notice the effectiveness in the employment of larger images.

6.1 Convolutional LSTM

Given the use of images, the recurrent convolutional model seemed to be the most suitable choice. This model, however, requires a research to find the meaningful sub-images. In addition to the search for the best size of the sub-images, the best representation for that sub-images should be sought too.

In table II and III I report the results of a first research. Each model utilize images in the shape $(128, 64)$; in column *features size* I reported the number of pixels that compose each time steps and in which shape, that is none other than the shape of the sub-images, while the last column shows the sequence of obfuscations, applied in any order, where multiple classification errors were found.

Image size	**Time steps**	**Features size**	**Validation accuracy**	**Top obf errors**
128x64	16	(8, 8, 8)	92,75%	Flatten-RandomFuns-Split EncodeArithmetic-Flatten-RandomFuns
128x64	32	(16, 16, 1)	89,13%	EncodeArithmetic-Flatten-RandomFuns Flatten-RandomFuns-Split
128x64	6	(32, 32, 1)	90,04%	Flatten-RandomFuns-Split EncodeArithmetic-RandomFuns-Split

Table II: Results of convolutional LSTM - 0-padding

Image size	**Time steps**	**Features size**	**Validation accuracy**	**Top obf errors**
128x64	16	(8, 8, 8)	89,59%	EncodeArithmetic-Flatten-Randomfuns EncodeLiterals-RandomFuns-Split
128x64	32	(16, 16, 1)	87,90%	EncodeLiterals-InitOpaque-Split Flatten-RandomFuns-Split
128x64	6	(32, 32, 1)	89,43%	Flatten-RandomFuns-Split EncodeArithmetic-Flatten-RandomFuns

Table III: Results of convolutional LSTM - interpolation

6.1.1 Convolutional LSTM - results analysis

From the validation accuracy value, we can notice a better reliability of the 0-padded model over the interpolated one. This denotes that with interpolation to size $(128, 64)$, the information obtained obstruct the learning rate of the model. Regarding the most difficult obfuscations, we can notice the sequence *Flatten-RandomFuns-Split* and *EncodeArithmetic-Flatten-RandomFuns*.

Reshaping the sub-images as $(8, 8, 8)$ means that the sub-images are a 3D tensor of 512 pixels. This means that the first 512 pixels of the image will form a time step, so with $(8, 8, 8)$, the first 8 rows of the images are analyzed (512 pixels divided by

64 width). What differs between the 3 convolutional models is the shape of the sub-images and the number of rows each sub-image represents: if with $(8, 8, 8)$ the first 8 rows are analyzed, with $(16, 16, 1)$ the first 4 rows of the image are analyzed, while with $(32, 32, 1)$ are analyzed the first 16 rows. The representation of a program as an image, and thus all the work, is based on the assumption that each instruction isn't represented in more than 64 pixels, i.e. a row.

It should be noted that the use of a 3D tensor speeds up the learning of the model, which is much slower for convolutional models than the LSTM counterparts.

6.2 LSTM

As mentioned in the previous chapter, the images were both interpolated and 0-padded, and the results for each models are listed in tables IV and V; the *N° features* column denotes the number of pixels that composes each time step, while the last column denotes the sequence of obfuscations, applied in any order, that causes the most misclassifications.

Image size	Time steps	N° features	Validation accuracy	Top obf errors
512x64	64	512	92,62%	EncodeArithmetic-Flatten-RandomFuns Flatten-RandomFuns-Split
128x64	64	128	93,52%	Flatten-RandomFuns-Split EncodeArithmetic-Flatten-RandomFuns
64x64	64	64	84,41%	Flatten-RandomFuns-Split EncodeArithmetic-Flatten-RandomFuns

Table IV: Results of LSTM - 0-padding

Image size	Time steps	N° features	Validation accuracy	Top obf errors
512x64	64	512	92,15%	Flatten-RandomFuns-Split EncodeArithmetic-Flatten-RandomFuns
128x64	64	128	91,66%	Flatten-RandomFuns-Split EncodeArithmetic-Flatten-RandomFuns
64x64	64	64	91,02%	Flatten-RandomFuns-Split EncodeArithmetic-Flatten-RandomFuns

Table V: Results of LSTM - interpolation

6.2.1 LSTM - results analysis

As we can deduce from the validation accuracy, the 0-padding method provides better results. We can also see how, with interpolation, the model with $(64, 64)$ images has a much better performance compared to the 0-padding model; this means that cropping the images in shape $(64, 64)$ results in a loss of information that the neural network utilizes in the image classification. On the contrary, with interpolation the lost information is instead taken into consideration. But this advantage doesn't reflect on bigger images, because the vast majority of the images is interpolated to an excessive size, which makes it not suitable. Not losing information on big images results in a bigger loss of precision on smaller images, which are far more numerous.

The sequence of obfuscations that hinders the classification the most, as in convolutional LSTM model, is *Flatten-RandomFuns-Split*.

6.3 Error analysis

There isn't a class, i.e. a program, which is particularly difficult to classify, but instead there are classes that often achieve an high rate of correct classification, over 98%. These classes, however, are very different from each other; for example, there is *hello_world*, one the most simple program and that generates small files, and there are programs like *watersheds*, which belongs to the dataset generated from the *Google Code Jam* competition, which generates some of the biggest files. These classes are the simplest to classify on both LSTM model and convolutional LSTM model.
I calculated the statistics of error and of accuracy of each class with a mean of 20 runs, and I calculated the standard deviation of the accuracy value for each class across the 20 runs. In figure 6.1, I reported the mean accuracy of each class; the classes whose standard deviation was above the average value are colored in pink, while the classes whose standard deviation was under the average, are blue. We can notice that the classes with the mean accuracy under the average value are the most difficult classes to identify, and are mostly programs that come from the simple dataset. This is attributable to the fact that the programs coming from *Google Code Jam*, being more specific, will present unique features that the model is able to extract more easily.

I have generated a graph that reports the percentage of occurrence of a specific obfuscation among the wrongly classified samples, with the aim of looking for a specific obfuscation that was more difficult to classify. As represented in figure 6.2, no more complex obfuscations than others appear. Instead, when *InitImplicitFlow* is applied, it results in a very low error rate, suggesting that when applied, the classification is facilitated. Browsing in the dataset, we can find some samples with the obfuscation label *base*; it denotes the original file, without any obfuscation applied, and is 1 file out of 400, for each class; it's normal that it is the most different, since every obfuscation, even the simplest, adds and initializes structures in the code, and this results in the original code as the most alien one; for this reason, the *base* obfuscation is removed from the graph 6.2.

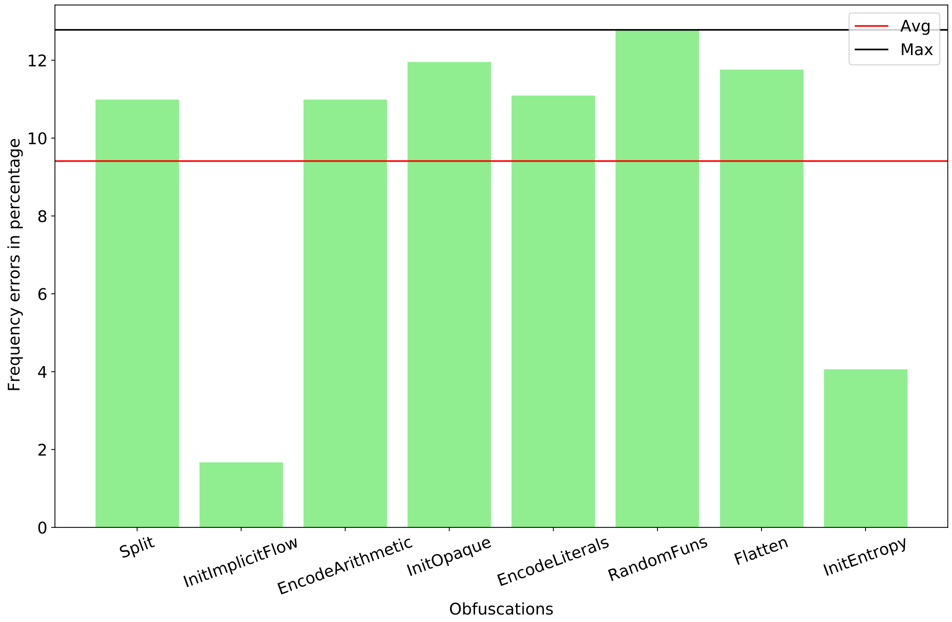


Figure 6.2: Error percentage per obfuscation

Before proceeding to the explanation of how each obfuscation affects the classification, it is good to remember that the 8 obfuscations applied are different:

- `InitImplicitFlow`, `InitEntropy` and `InitOpaque` are initialization transformations;
- `EncodeLiterals` and `EncodeArithmetic` are encoding transformations with specific target;
- `Flatten`, `RandomFuns` and `Split` are transformations that impact the CFG.

For the error analysis, the size of the validation set has been set to 40% of the total number of samples; to compensate for the smaller number of samples in the training, the number of epochs and the *patience* parameter have been increased; the accuracy of the models was not affected. The database consists of 400 samples for each class; to generate the validation set, an equal number of samples were randomly extracted for each class, i.e .: $(18800 * 40\%)/47 = x$, where x is the number of samples from each class. In this way, there is a random but equally distributed number of samples from each class in the validation set.

Furthermore, the graphs are generated by analyzing the wrongly classified samples; this means that whenever I report a percentage of a certain obfuscation, this value refers to the wrongly classified samples over the total samples in the validation set: if a certain class has an high accuracy, the wrongly classified samples will be low; it is therefore not possible to make a 1:1 comparison between the error rates of different classes.

6.3.1 InitImplicitFlow

In figure 6.2 we can notice how each obfuscation is near the mean value, while *InitImplicitFlow* is much below the average. I explored the changes that are made to the code by it, to understand why it made the classification easier, and it turned out that it is an obfuscation that does not apply changes to the control flow of the code, but adds some dummy functions and initializes some data structures. Furthermore, if in a sequence of obfuscation the *InitImplicitFlow* is applied as the last one, it doesn't change the binary in the slighest.

Namely, let i be the *InitImplicitFlow* obfuscation, let a, b be every other obfuscation, let P be a program, then: $P_{ai} = P_a$ and $P_{abi} = P_{ab}$.

Moreover, if we have $P_{aib} = P'$, $P_{ab} = P''$, P' is just marginally different from P''.

This is reflected in a simpler classification when *InitImplicitFlow* is applied, because those are files that the neural network already knows and on which it was already trained.

6.3.2 InitOpaque

While the initialization transformations in figure 6.2 are low, i.e. *InitImplicitFlow* and *InitEntropy*, *InitOpaque* is instead near average.

For each class, I calculated the percentage of presence of each obfuscation among the wrongly classified samples out of the total obfuscations present in the validation set (for that class). The procedure is similar to the one used to generate the figure 6.2, but it refers to each single class rather than to all classes. It turns out that *InitOpaque* is a transformation that, alone, is one of the most hindering in the classification for every class. But, from the analysis of the applied transformations that most hinder the classification, *InitOpaque* does not emerge (see tables II, III, IV and V). This is because other transformations manage to hinder more when stacked than *InitOpaque* alone, which remains an initialization function after all; it is, however, among the most complex obfuscations for any class. For example, in figure 6.3 are represented *RandomFuns*, *Split* and *InitOpaque*; the first two are obfuscations that affect deeply the code and are always applicable, in fact they are among the most difficult transformations; *Initopaque*, is often on par with the other two single transformations, or even more complex.

InitOpaque alters the code by initializing data structures one value at a time; the compiler can't optimize them, and so the binaries are transformed into more complex files. For this reason, *initOpaque* turns out to be harder to classify.

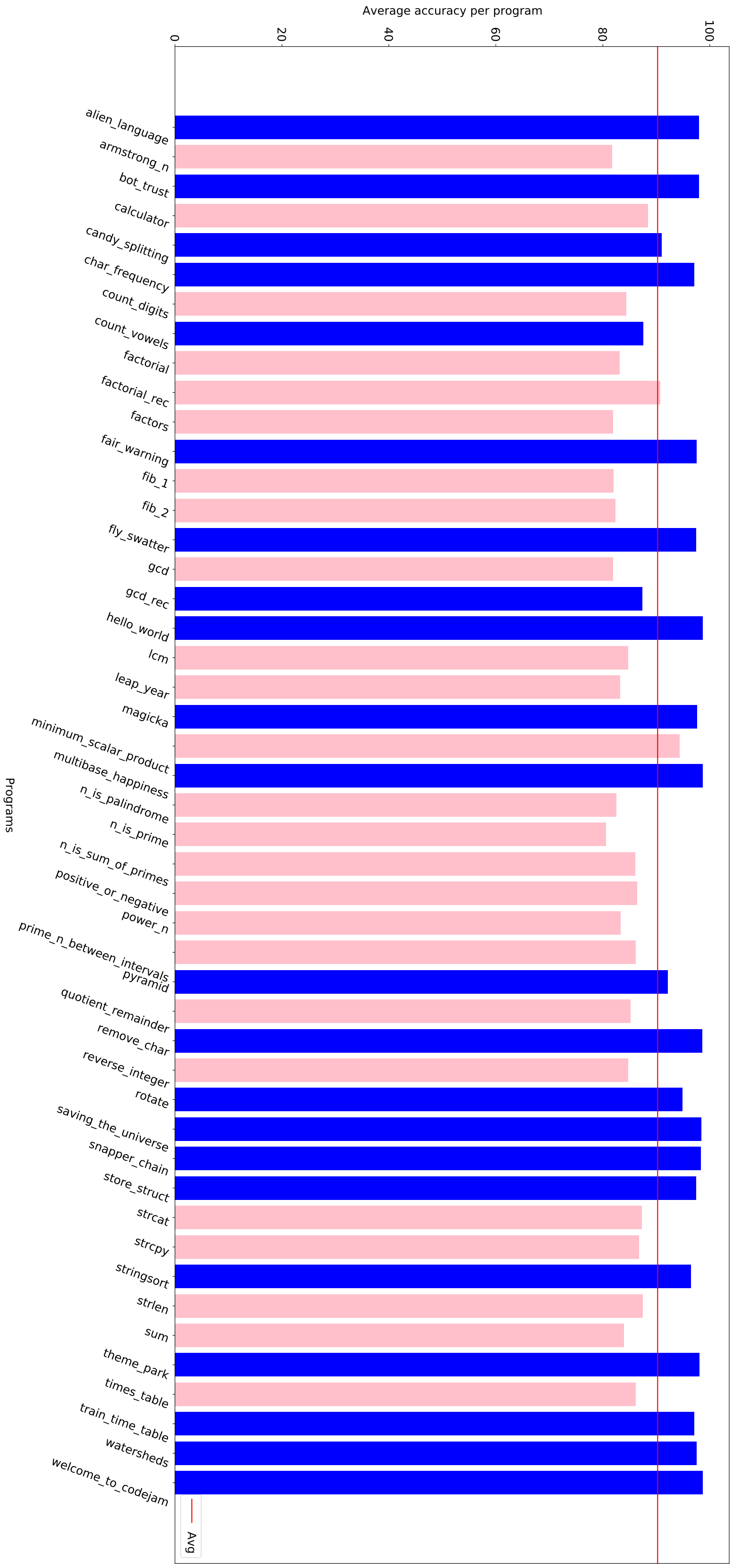


Figure 6.1: Average accuracy of each program; in pink, programs which standard deviation is over the average standard deviation value

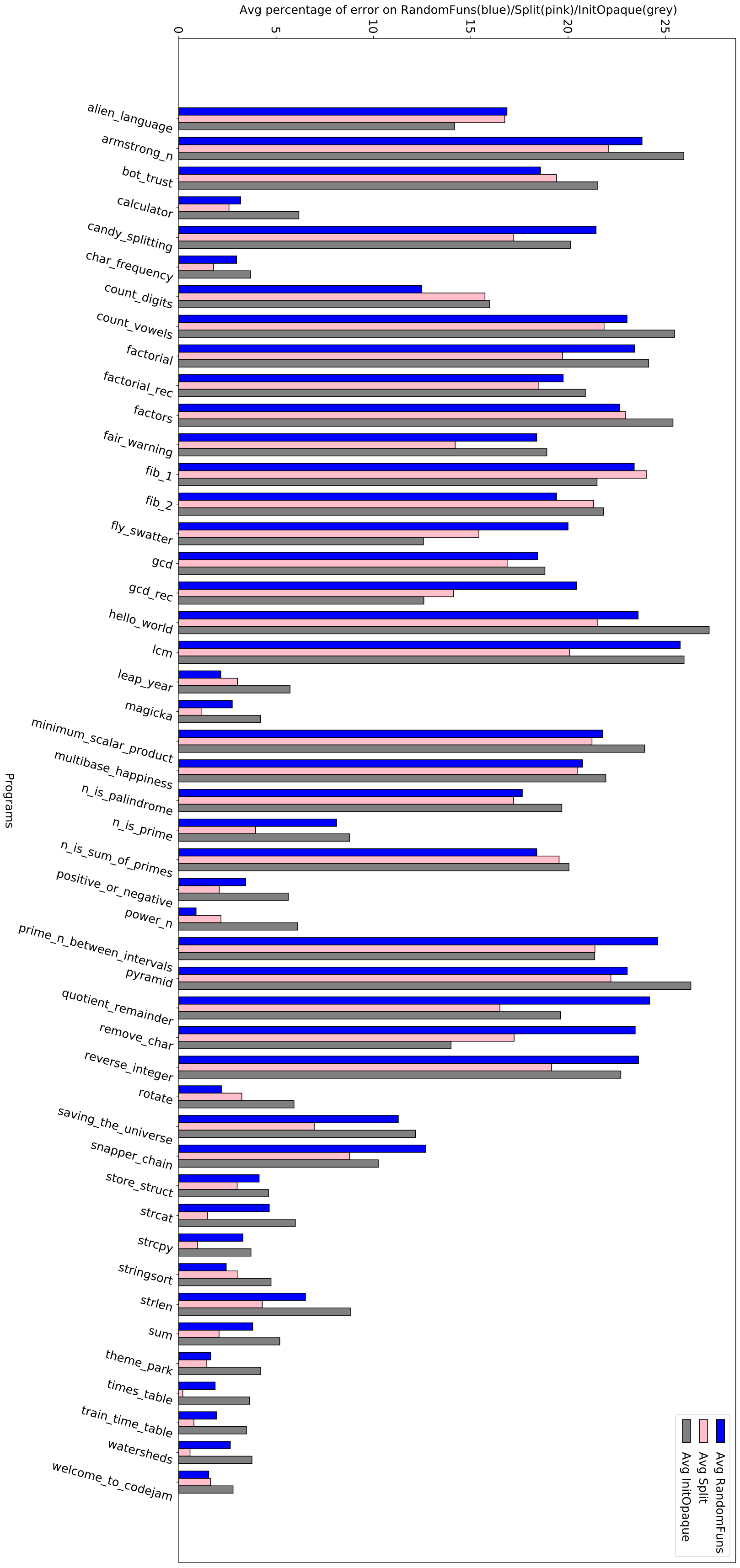


Figure 6.3: Error percentage of RandomFuns, Split, InitOpaque

6.3.3 EncodeLiterals and EncodeArithmetic

EncodeLiterals and *EncodeArithmetic* are transformations that have a precise target, respectively literal expressions and arithmetic expressions. Figure 6.4 shows the percentage of wrongly classified samples that contain *EncodeLiterals* and *EncodeArithmetic*.

EncodeArithmetic

EncodeArithmetic encodes arithmetic expressions, including increments inside `for` loops. As we can see in figure 6.5, the distribution of difficulties of *EncodeArithmetic* ranges a lot. Programs with high difficulty are *fib_1*, *armstrong*, *power_n*. Looking at the source code of these programs, we can see how they have different arithmetic instructions, especially in short-length codes. For example, *fib_1* is a program with less than 20 lines of code, and it has several assignments and two arithmetic expressions. Looking at the less difficult programs, we can see *hello_world* and *store_struct*, which are in fact programs that don't have *any* arithmetic expression. In *alien_language* instead, there are some arithmetic expressions, but they are few in a code longer than the average; also, *alien_language* is a program with an accuracy of over 98%.

EncodeLiterals

The target of this obfuscation are literal expressions, and very few programs don't have at least one literal expression (figure 6.6). Looking at the codes, there are programs that are very extensive but with small literal expressions: for example, *alien_language* is a long code that has 4 strings, 3 of which totally built at runtime, and very short. Similar to *alien_language*, there are *bot_trust* and *train_time_table*. In fact, these three programs are not harder to classify when they are transformed by *EncodeLiterals*. *n_is_prime*, the program with the most errors caused by *EncodeLiterals*, is a clarifying example on how this obfuscation can become an obstacle: that program is composed of few code lines, but has several literals expressions, relatively long compared to the code.
EncodeLiterals obfuscates literal expressions by introducing functions that store each char (composing the literals) in a data structure, to then build the encoded string at runtime: this means that a lot of instructions are added in programs as *n_is_prime*, thus explaining their difficulties.

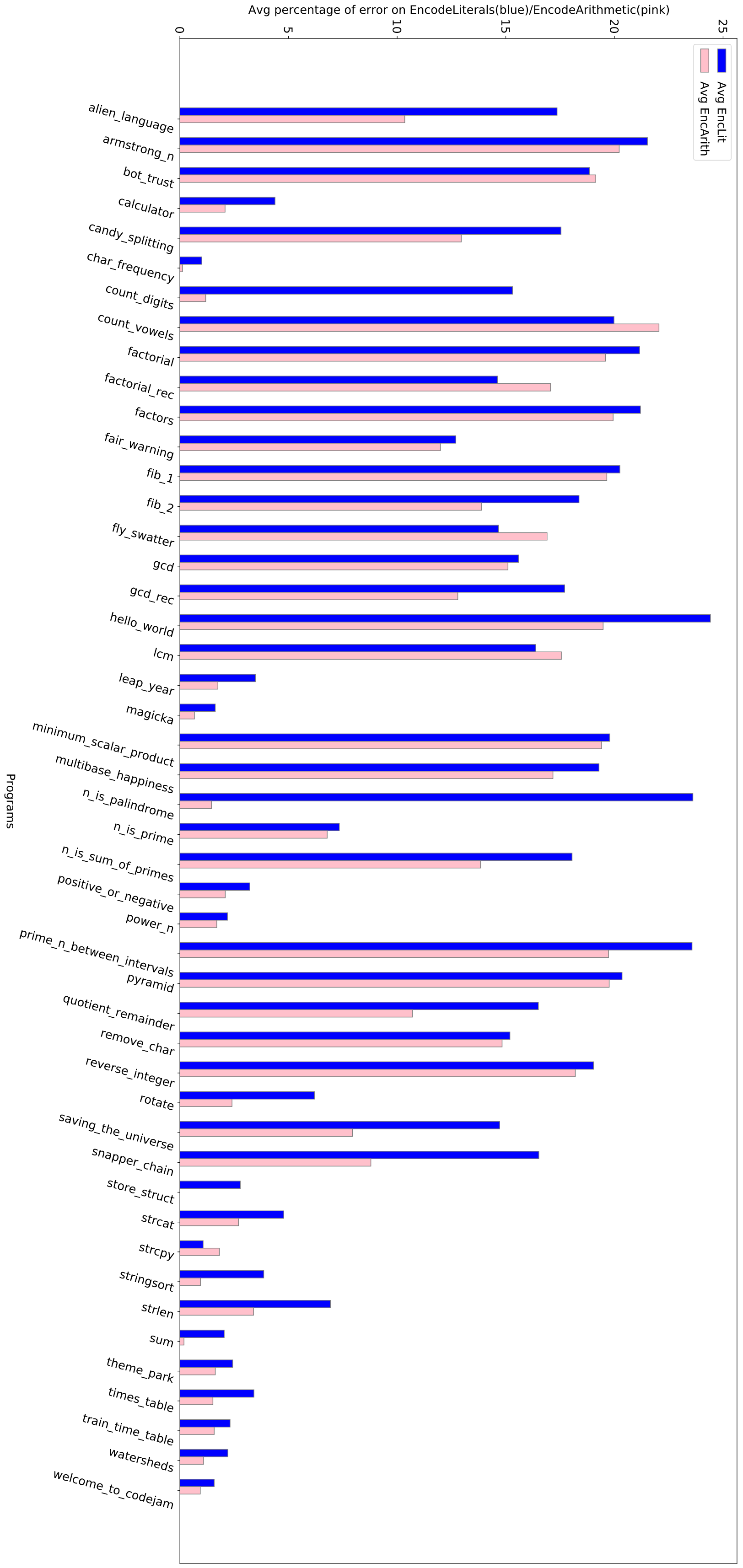


Figure 6.4: Error percentage of EncodeLiterals and EncodeArithmetic

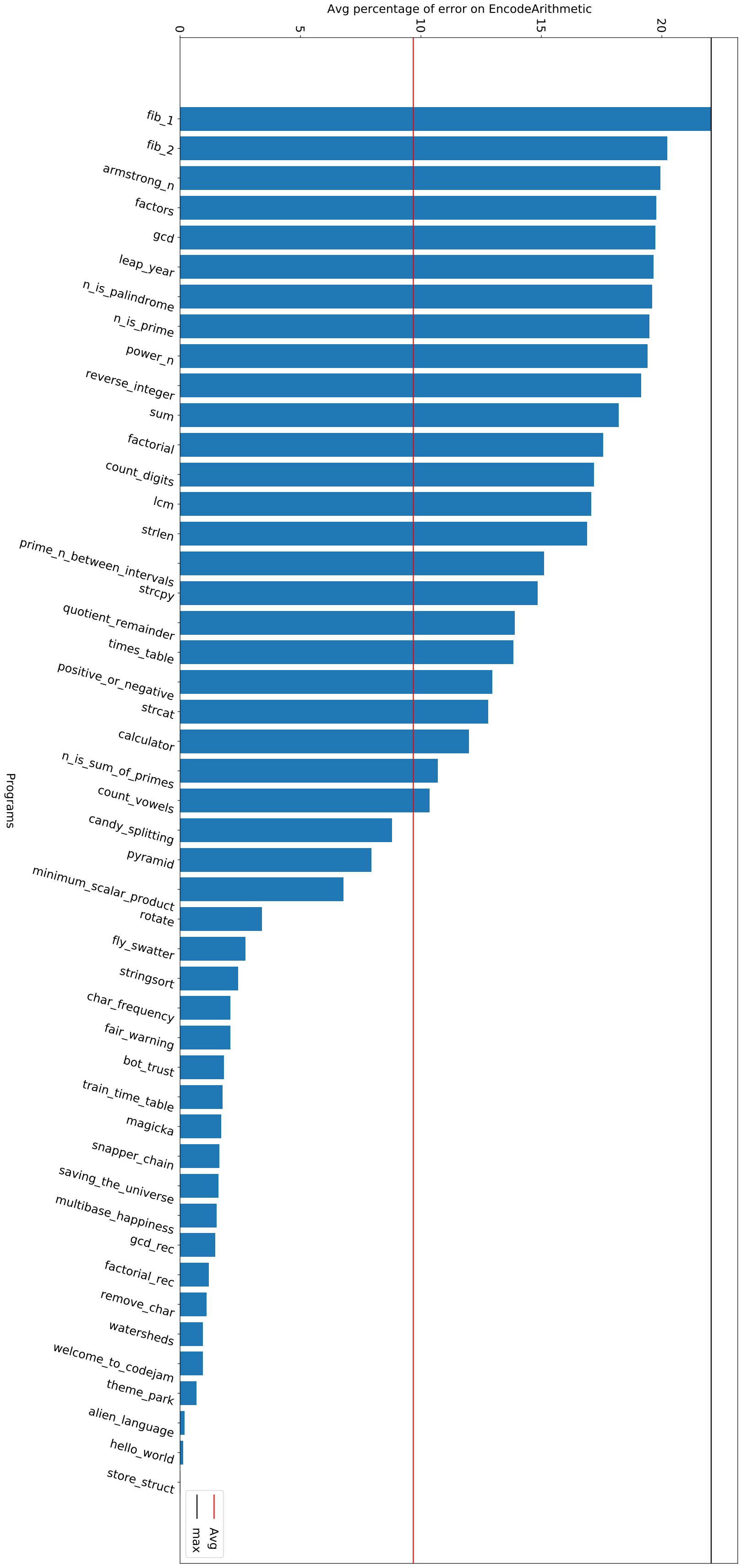


Figure 6.5: Error percentage of EncodeArithmetic

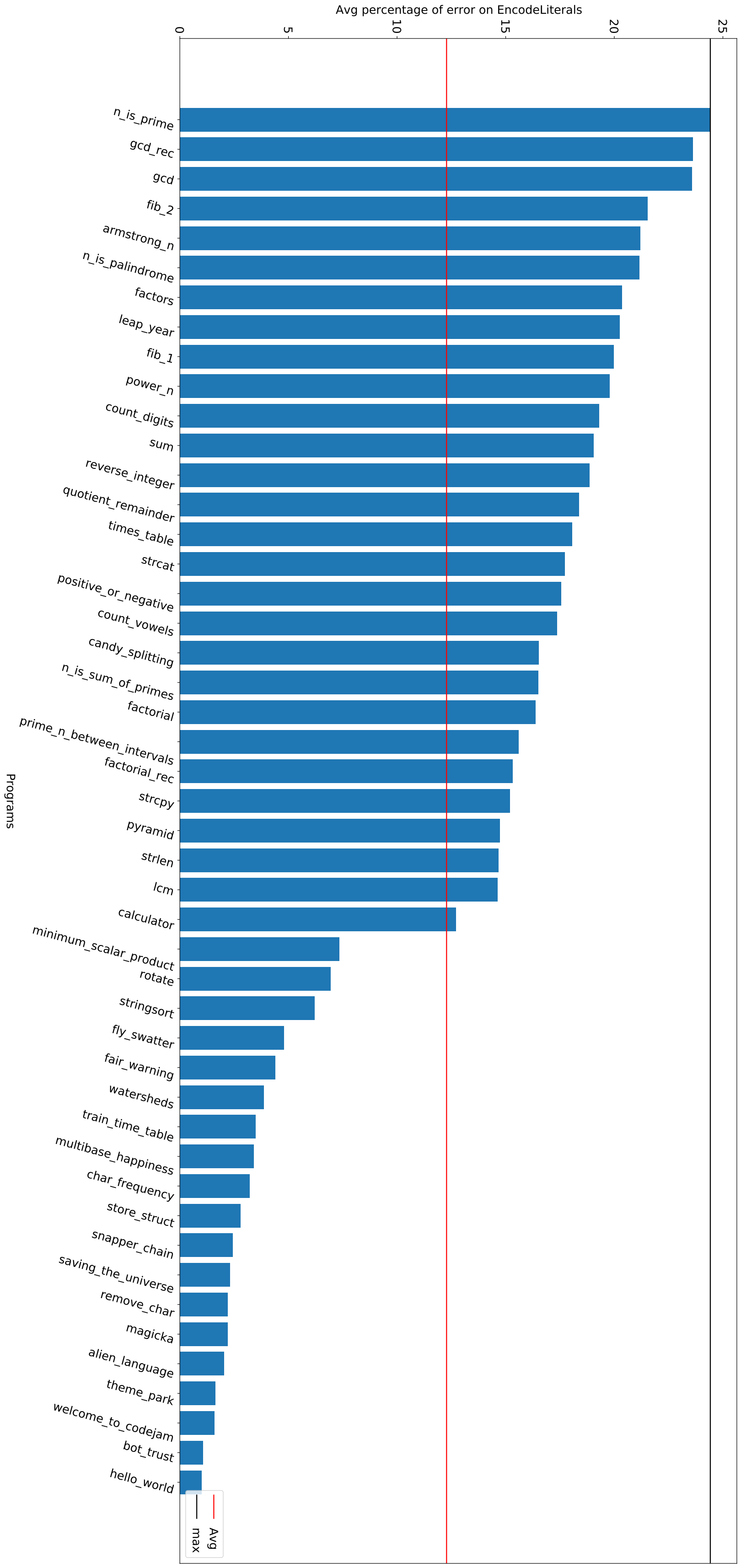


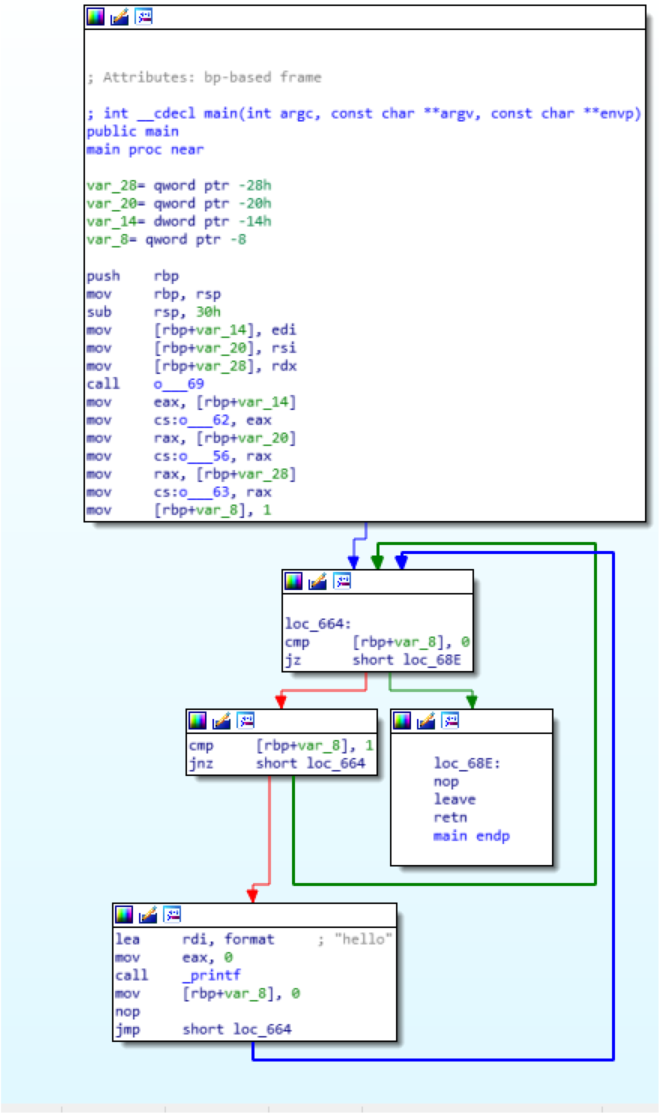
Figure 6.6: Error percentage of EncodeLiterals

6.3.4 Flatten, RandomFuns and Split

When combined, these three transformations appear to be the greatest obstacle to classification. *RandomFuns* adds dummy functions, while *Split* separates functions into sub-functions. Both transformations increase the number of code functions, and therefore the number of jump instructions (`JMP`) in assembly. Splitting after a *RandomFuns* only increases the number of functions in a code. This also impacts the binary code about the proximity of related instructions, and adding *Flatten* helps disrupt the control flow even more.

Flatten

Flatten flattens the CFG by adding `switch cases`, even in programs that have no control instructions, such as *hello_world*: in this case, it builds a `switch case` with two cases, one empty and the other one with the original code. Although the case is empty, the compiler fails to optimize it; as it can be seen from the CFG built by IDA in figure 6.7, the case dummy has a NoOp and a return only.



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_28= qword ptr -28h
var_20= qword ptr -20h
var_14= dword ptr -14h
var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     [rbp+var_14], edi
mov     [rbp+var_20], rsi
mov     [rbp+var_28], rdx
call    o___69
mov     eax, [rbp+var_14]
mov     cs:o___62, eax
mov     rax, [rbp+var_20]
mov     cs:o___56, rax
mov     rax, [rbp+var_28]
mov     cs:o___63, rax
mov     [rbp+var_8], 1
```

```
loc_664:
cmp     [rbp+var_8], 0
jz      short loc_68E
```

```
cmp     [rbp+var_8], 1
jnz     short loc_664
```

```
loc_68E:
nop
leave
retn
main endp
```

```
lea     rdi, format     ; "hello"
mov     eax, 0
call    _printf
mov     [rbp+var_8], 0
nop
jmp     short loc_664
```

Figure 6.7: Hello_world - Flatten

Knowing that *Flatten* also acts on codes that do not have control structures, in programs that struggle with such obfuscation the most (figure 6.8), it is not surprising to find conditional branches inside loops. Surprisingly, *fly_swatter* is not among the most complex programs, despite the source code having up to 4 `for` nested loops, with different branches; analyzing its obfuscated code, I noticed that *Flatten* fails to make changes on such complex codes.

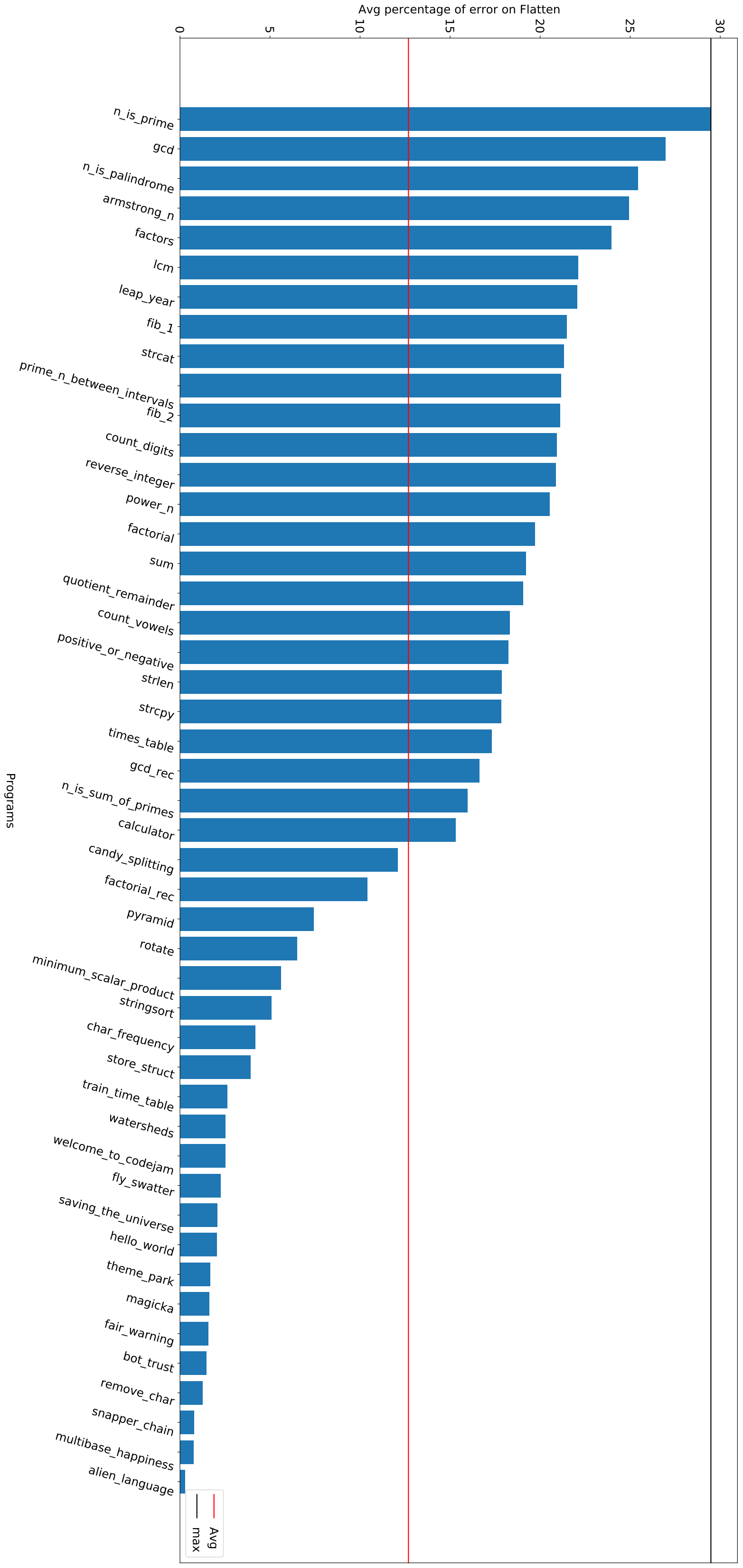


Figure 6.8: Error percentage of Flatten

6.4 Conclusions

From the analysis of the results, it emerged that the hypothesis of using images at $(128, 64)$, being the programs generally below this length, is more effective than using larger sizes, especially because the larger files come from the *Google Code Jam* competition, and are among the most easily classified programs despite the chosen size. Furthermore, the idea of using interpolation has been discarded, since the information that is added is an approximation that achieves a maximum of 90% accuracy, while with 0-padding it is possible to overcome this value.

It was then seen that, despite the time limits imposed by the development platform, the recurrent architecture has some advantages over the convolutional model presented in 3.1. Among the LSTM and convolutional LSTM models, the former was better, not only for the training time needed, but also as it required a search for only the most suitable size for the time steps, while a much more thorough search should be done to find the best parameters for images in the convolutional LSTM model.

From the error analysis, it is clear that the obfuscations that cripple more the learning process, are those that have a target on which to operate, and especially all those transformations that increase the size of the program, with the addition of dummy functions that could also slow down the correlation between instructions that the LSTM is trying to learn.

In addition, the complex obfuscations in this model, such as *InitOpaque* or *RandomFuns-Split*, are the ones that were found to be harder in the convolutional model in 3.1, underlining how the learning difficulty is more closely related to obfuscations than to the architecture used.

Chapter 7

Limitations and future work

Among the limits, the most burdensome were the time limits that Google Colab imposes. If the memory usage can be controlled through the batch size, the number of epochs for training the models and the size of the images must comply with time limits. Moreover, Google Colab imposes a much lower limit to idle time, which forced me to focus on one task at a time.

As for the future work, resolving the time limits issue could enable a further research of the best parameters for the sub-images in the convolutional LSTM model. As well, understanding how each instruction in the binary file is represented in the image should help to define better time steps size. More obfuscations could be taken into account and furthermore, the models presented in this work could be tested on a real database, like the database proposed for the Microsoft Malware Classification Challenge in 2015, a huge dataset of nearly 0.5 terabytes, consisting of disassembly and bytecode of more than 20K malware samples.

Continuing to work on images, exploring how to embed semantic information within images, moving from a pure binary representation to an hybrid one, could help and speed up the models in identifying similarities between programs.

Appendix A

Other figures

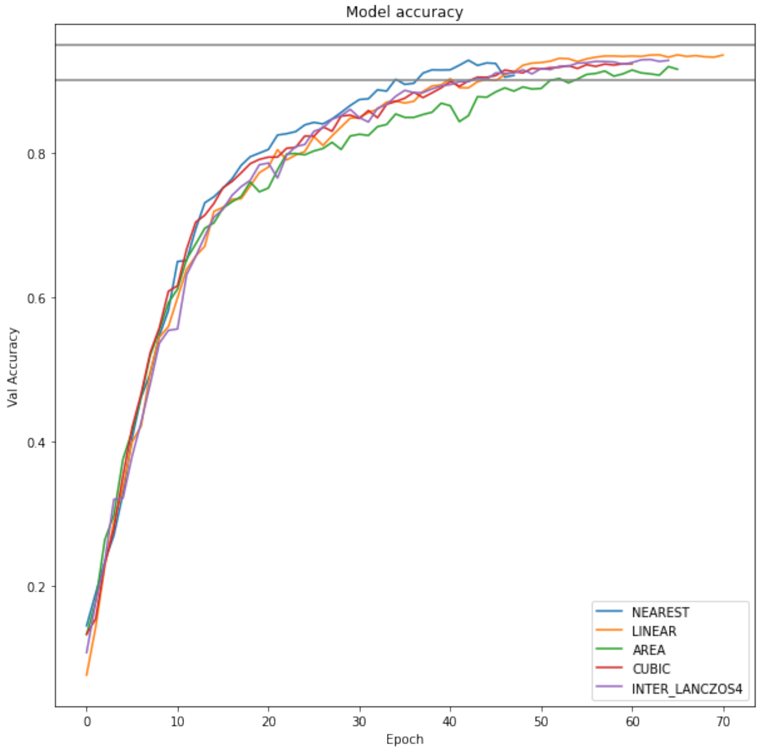


Figure A.1: Model accuracy with different interpolation methods

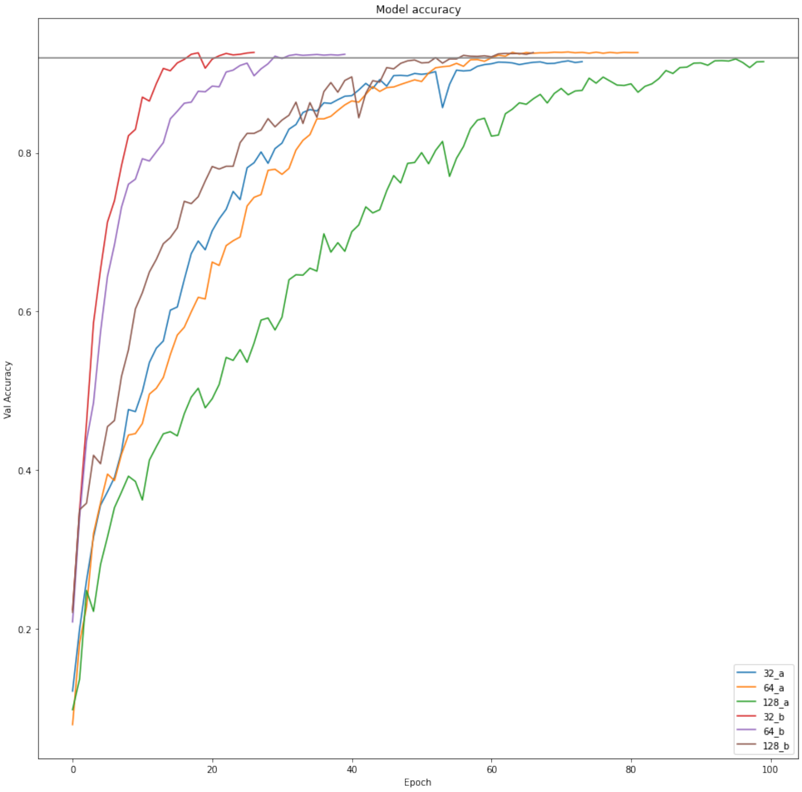


Figure A.2: Different time steps; *a* refers to 512x64 images, *b* refers to 128x64 images

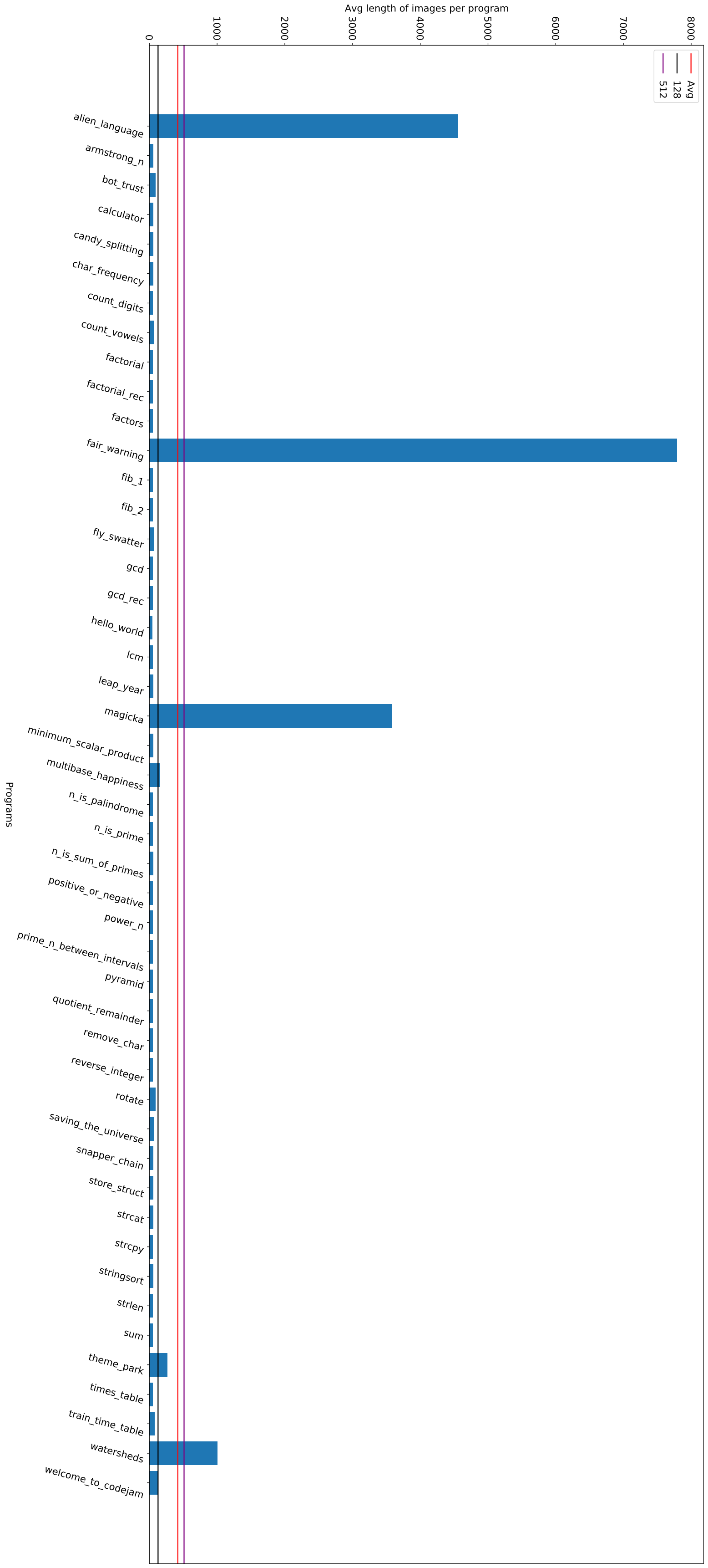


Figure A.3: Distribution of average image length per class

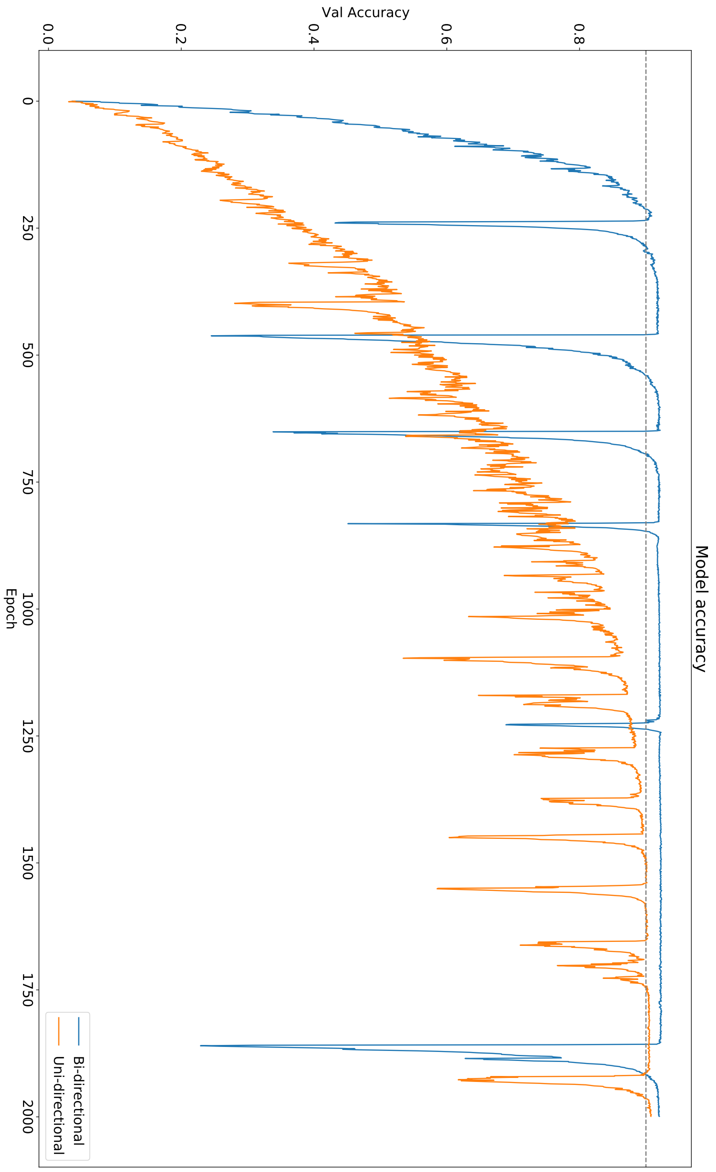


Figure A.4: Accuracy of bidirectional and unidirectional models

Bibliography

[1] H. G. Rice. Classes of recursively enumerable sets and their decision problem. *Trans. Amer. Math. Soc.*, 74, 1953.

[2] Niccolò Marastoni, Roberto Giacobazzi, and Mila Dalla Preda. A deep learning approach to program similarity. *MASES 2018: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, page 26–35, September 2018.

[3] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '98*, page 184–196, 1998.

[4] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[5] Christian Collberg. the tigress C obfuscator. `https://tigress.wtf/index.html`.

[6] Google. Google Colaboratory intro. `https://colab.research.google.com/notebooks/intro.ipynb`.

[7] Google. Google Colaboratory FAQ. `https://research.google.com/colaboratory/faq.html`.

[8] Google. TensorFlow GitHub repository. `https://github.com/tensorflow`.

[9] Google. TensorFlow. `https://www.tensorflow.org/`.

[10] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[11] IBM Cloud Education. Machine learning, July 2020. `https://www.ibm.com/cloud/learn/machine-learning`.

[12] Sebastian Raschka and Vahid Mirjalili. *Python Machine Learning*. Packt Publishing Ltd, ISBN 978-1-78995-575-0, 2019.

[13] Larry Hardesty. Explained: Neural networks. *MIT News Office*, April 2017. `https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414`.

[14] Sagar Sharma. What the hell is perceptron?, September 2017. `https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53`.

[15] Ronald J. Williams, Geoffrey E. Hinton, and David E. Rumelhart. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[16] Christopher Olah. Understanding lstm networks, August 2015. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[17] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5:157–166, March 1994.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:157–166, November 1998.

[19] Schuster Mike and Paliwal Kuldip K. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45:2673–2681, November 1997.

[20] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18:602–610, July 2005.

[21] Z. Cui, R. Ke, Z. Pu, and Y. Wang. Stacked bidirectional and unidirectional lstm recurrent neural network for network-wide traffic speed prediction. *arXiv preprint arXiv:1801.02143*, 2018.

[22] Gholamreza Anbarjafari. Digital image processing. University of Tartu, `https://sisu.ut.ee/imageprocessing/book/3`, 2014.

[23] Chul Hyun Kwag. cv2 resize interpolation methods, November 2018. `https://chadrick-kwag.net/cv2-resize-interpolation-methods/`.

[24] Ragkhitwetsagul C., Krinke J., and Clark D. A comparison of code similarity analysers. *Empirical Software Engineering*, 32:2464–2519, August 2018.

[25] Lawton Higgins Nichols. *Program Similarity Techniques and Applications*. PhD thesis, University of California Santa Barbara, June 2020.

[26] Cheers H. and Lin Y. A novel graph-based program representation for java code plagiarism detection. *Proceedings of the 3rd International Conference on Software Engineering and Information Management*, page 115–122, January 2020.

[27] Sudhamani M. and Rangarajan L. Code similarity detection through control statement and program features. *Expert Systems with Applications*, 132:63–75, October 2019.

[28] Zhang X., . Pang J, and X. Liu. Common program similarity metric method for Anti-Obfuscation. *IEEE Access*, 6:47557–47565, 2018.

[29] C. Smyth, C. Fong, Y. C. Lui, and Y. Cao. Systems and methods for malicious code detection, 2020. US Patent 10,685,284 B2.

[30] Gibert D., Mateu C., and Planes J. A hierarchical convolutional neural network for malware classification. *2019 International Joint Conference on Neural Networks (IJCNN)*, page 1–8, July 2019.

[31] A. N. Jahromi, S. Hashemi, A. Dehghantanha, R. M. Parizi, and K.-K. R. Choo. An enhanced stacked lstm method with no random initialization for malware threat hunting in safety and time-critical systems. *IEEE Trans. Emerg. Top. Comput. Intell.*, page 1–11, 2020.

[32] Shiqi Luo, Shengwei Tian, Long Yu, Yu Jiong, and Sun Hua. Android malicious code classification using deep belief network. *KSII TIIS*, 12:1–11, 2018.

[33] J. Yan adn G. Yan and D. Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, page 52–63, 2019.

[34] Youngbok Cho. Dynamic rnn-cnn based malware classifier for deep learning algorithm. *29th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6, November 2019.

[35] Paolo D'Arienzo. GitHub repository. `https://github.com/PaoloDarienzo/`.

[36] Mnist database. `http://yann.lecun.com/exdb/mnist/`.

[37] Zalando. Fashion mnist database. `https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/`.

[38] Heaton Jeff. Heaton research: The number of hidden layers, June 2017. `https://www.heatonresearch.com/2017/06/01/hidden-layers.html`.

[39] NVIDIA. Nvidia cudnn. `https://developer.nvidia.com/cudnn`.

[40] Y. Xu and R. Goodacre. On splitting training and validation set: A comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning. *Journal of Analysis and Testing*, 2:249–262, 2018.

[41] Timo Stöttner. "why data should be normalized before training a neural network". Article published on `https://towardsdatascience.com/`.

[42] James McCaffrey. How to standardize data for neural networks. `https://visualstudiomagazine.com/articles/2014/01/01/how-to-standardize-data-for-neural-networks.aspx`.

[43] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. *ICML'15: Proceedings of the 32nd International Conference on International Conference on Machine Learning*, 37, 2015.