

APPLICATION OF DEEP-LEARNING TO COMPILER-BASED GRAPHS

by

Tristan Vanderbruggen

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer and Information Science

Winter 2018

© 2018 Tristan Vanderbruggen
All Rights Reserved

**APPLICATION OF DEEP-LEARNING TO COMPILER-BASED
GRAPHS**

by

Tristan Vanderbruggen

Approved: _____
Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Science

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____
Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
John Cavazos, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Daniel Quinlan, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Sunita Chandrasekaran, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Stephen Siegel, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. John Cavazos for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me throughout the research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Stephen Siegel, Prof. Sunita Chandrasekaran, and Dr. Daniel Quinlan, for their encouragement, insightful comments, and challenging questions.

My sincere thanks also goes to Dr. Daniel Quinlan, Dr. Chunhua Liao, Dr. Markus Schordan, and Justin Too, for offering me the opportunity to work with them on the ROSE Compiler. Working with them provided the theoretical and technical knowledge required to perform research on compilers. I wish to thank Prof. Christophe Wolinski who introduced me to Dr. Daniel Quinlan.

I wish to thank my labmates at University of Delaware. Dr. Sameer Kulkarny, Dr. Eunjung Park, William Killian, and Robert Searles contributed advice, critiques, and encouragement, which made my work on performance optimization possible. Similarly, my work on malware classification would not have been possible without the support of Sean Kilgallon, Leonardo De La Rosa, and Ian Lantzy.

My internship at the Lawrence Livermore National Laboratory has lead me to meet many great people. I wish to thanks Tony Baylis who provided us with memorable networking opportunities. I must mention Michael Driscoll, George Vulov, and Sriram Aananthakrishnan who were not afraid of my broken English during my first internship. Our work together lead to lasting friendships.

The baristas from my favorite coffee shop, Saxbys, must be acknowledge as well. Without their coffee and welcoming attitude, I would not have been able to finish writing this dissertation.

Last but not the least, I would like to thank my family. My mom, brother, and sister for their support even as it meant being far away from home. My fiancée for keeping me together when the pressure threatened to break me.

TABLE OF CONTENTS

LIST OF TABLES	xiii
LIST OF FIGURES	xiv
ABSTRACT	xvii
 Chapter	
1 INTRODUCTION	1
2 MACHINE LEARNING BACKGROUND	5
2.1 Introduction	5
2.2 Deep Learning	5
2.2.1 Artificial Neural Networks	5
2.2.2 Training Neural Networks	8
2.2.3 Deep Neural Networks	10
2.2.4 Implementation	11
2.3 Random Forest	12
2.4 Support Vector Machine	12
2.5 Comparing Models	14
2.5.1 Cross Validation	14
2.5.2 Accuracy	15
2.5.2.1 Error Rate	16
2.5.2.2 Confusion Matrix	16
2.5.2.3 Receiver Operating Characteristic and Area Under the Curve	17
2.5.3 In practice	17
2.6 Conclusion	18

3	MACHINE LEARNING FOR GRAPHS	19
3.1	Introduction	19
3.2	Graph Formalism	19
3.3	Graph Kernels for Support Vector Machine	20
3.4	Graph Spectral Analysis	22
3.5	Graph Spectral Features	23
3.6	Conclusion	24
4	BASIC NEURAL NETWORKS FOR MALWARE CLASSIFICATION	25
4.1	Introduction	25
4.2	Malware Datasets and Characterization	26
4.2.1	Dataset	26
4.2.2	Bytes-Entropy Histogram	27
4.2.3	Malware’s Executable Code	31
4.2.3.1	Disassembly & Analysis	32
4.2.3.2	Code Features	34
4.2.4	Summary of the Features	36
4.3	Machine Learning in the Cloud	36
4.3.1	Training and Consensus	37
4.3.2	Cloud Infrastructure	39
4.4	Malware Classification Results	41
4.4.1	Models	41
4.4.1.1	Multilayer Perceptrons	41

4.4.1.2	Random Forests	42
4.4.2	Accuracy and Training Time	43
4.5	Related Work	47
4.5.1	Malware Characterization	47
4.5.2	Feature Graphs	49
4.6	Conclusion	49
5	ADVANCED NEURAL NETWORKS FOR MALWARE CLASSIFICATION	51
5.1	Introduction	51
5.2	Feature Sets	52
5.2.1	Three Types of Features	52
5.2.1.1	Hashes Histograms	53
5.2.1.2	Bytes-Entropy Histograms	54
5.2.1.3	Spectral Features of Assembly Graphs	54
5.2.2	Transformations	54
5.2.3	Feature Sets Summary	55
5.3	Dataset	56
5.3.1	Relevant Files From Our Database	56
5.3.2	Composition of the Dataset	56
5.3.3	Streaming of Samples	57
5.4	Neural Network Engineering	58
5.4.1	Training Procedure	58
5.4.1.1	High Level Features	58

5.4.1.2	Classifiers	59
5.4.2	Neural Network Architectures	60
5.4.2.1	Input Convolutional Layers	60
5.4.2.2	Hidden Convolutional Layers	63
5.4.2.3	Reference Architectures	63
5.5	Results	64
5.5.1	Experimental Setup	64
5.5.2	Accuracy Results	66
5.5.3	Computational Performances	66
5.6	Related Work	69
5.6.1	Dataset Augmentation	69
5.6.2	Auto-encoder and Convolutional Architectures	70
5.7	Conclusion	71
6	EXPLORATION AND CHARACTERIZATION OF COMPILER TUNING SPACE	73
6.1	Introduction	73
6.2	TileK	75
6.2.1	TileK language	75
6.2.2	Iteration Domain	76
6.2.3	Distributed Kernels	77
6.2.3.1	TileK Threads	77
6.2.3.2	TileK Accelerator	79
6.3	Generating Tiled Kernels	80
6.3.1	LoopTrees	80
6.3.2	Generated Kernel	82

6.3.3	From Loop Bounds to Tile Bounds	84
6.4	Generating SPMD Kernels	84
6.4.1	Kernel Index to Tile Iteration	85
6.4.2	Threads	86
6.4.3	Accelerator	86
6.5	Optimization Space Exploration	87
6.5.1	Thread Experiments	92
6.5.2	Accelerator Experiments	94
6.6	Related Work	96
6.7	Conclusion	97
7	PERFORMANCE PREDICTION FOR COMPUTATION	
	KERNEL TUNING	98
7.1	Introduction	98
7.2	Dataset	99
7.2.1	Feature Graph from TileTree	99
7.2.2	Targets	100
7.3	Models	102
7.3.1	Neural Networks	103
7.3.1.1	TileTree Representations	103
7.3.1.2	Architectures	104
7.3.2	Support Vector Machine	104
7.3.3	Evaluation	104
7.4	Results	105
7.4.1	Effect of Complex Neural Networks	106
7.4.2	Performance Milestone	106
7.5	Related Work	108
7.6	Conclusion & Future Work	109

8 CONCLUSION	112
8.1 Results	112
8.2 Insights & Future Work	114
BIBLIOGRAPHY	115
Appendix	
A MAGIC FRAMEWORK	124
A.1 Design	124
A.1.1 Data Storage	125
A.1.2 Datasets	125
A.1.2.1 Feature Description	125
A.1.2.2 Building a Dataset	127
A.1.3 Sessions	129
A.1.3.1 Models	129
A.1.3.2 Usage	131
A.2 WebUI	132
A.3 Milestones	133
A.3.1 Machine Learning Backends	133
A.3.2 Learning Patterns	133
A.3.3 HPC support	134
A.3.4 Control from Wizard	135
A.3.5 Data Sources	135
A.3.6 Virtual Segmentation	136
B ROSE COMPILER	137
B.1 Abstract Syntax Tree	137
B.2 Visualization	140

C	PARSING COMPILER DIRECTIVES	142
C.1	Directive-based Language Extension	142
C.1.1	Directive Format	142
C.1.1.1	Structure	142
C.1.1.2	Clause Arguments	143
C.1.1.3	Relations between directive and AST nodes	143
C.1.1.4	Relations between directives	143
C.1.1.5	How to Parse Directives	143
C.2	Generic Parser	144
C.2.1	Implementation of DLX	144
C.2.1.1	Class and Method Factory	144
C.2.1.2	Pattern in DLX	145
C.2.1.3	Automation	146

LIST OF TABLES

2.1	Confusion Matrix	16
4.1	Malware families	27
4.2	Radare2 operation categories	34
4.3	Features summary	37
4.4	Size and shape of the MLPs for each feature set	42
4.5	Error-rates random-forests	46
5.1	Feature Sets before and after preprocessing	55
5.2	Auto-encoder Architectures & Layers Size	65
5.3	Groups of Feature Sets	65
5.4	Average test accuracy of the best model in each fold	67
6.1	Input space and Flops	91
6.2	Thread versions	92
6.3	Accelerator versions	95
7.1	Number of evaluations needed to reach performance milestones. The speedup compared to random search is shown between parenthesis.	107
C.1	The different components of the directives in Listing C.1.	142

LIST OF FIGURES

2.1	Artificial neuron	6
2.2	MLP with single hidden layer	6
2.3	Convolutional neuron network	11
2.4	SVM: separating hyperplanes	13
2.5	SVM: kernel trick	14
4.1	Distribution of metrics in dataset	28
4.2	Flow of the bytes-entropy histogram extraction	29
4.3	Bytes-entropy histograms	30
4.4	Assembly graph construction	32
4.5	Code Nested Graphs	33
4.6	Statistic accumulation across graph granularity levels	34
4.7	Structure of three blocks level graphs	35
4.8	Cross-validation and Consensus	38
4.9	Deep learning on AWS cloud	40
4.10	Error rates vs training time	44
4.11	Area under the Curve	45
5.1	List of Strings to Fixed Size Vector	53
5.2	Composition of the dataset	57

5.3	Convolutional Layer for Hashes Histogram	61
5.4	Convolutional Layer for Bytes-Entropy Histogram	62
5.5	Convolutional Layer for Graph Spectral Features	62
5.6	Hidden Convolutional Layers of the Stacked Auto-encoders	63
5.7	Compare Accuracy across Feature Sets and Architectures	67
5.8	Comparing the Number of Parameters in the Models	68
5.9	Training and Testing Time per Model	68
6.1	TileK's compilation flow	74
6.2	Iteration domain tiled with TileK	77
6.3	Thread spawning	78
6.4	Thread tiling	78
6.5	TileK accelerator machine model	79
6.6	TileK's LoopTree	81
6.7	TileK's TileTree	81
6.8	Performance results for optimization and input spaces exploration .	93
7.1	Construction of a Feature Graph from TileK's IR	101
7.2	Guiding Iterative Compilation	102
7.3	Two stage ranking model: NN and SVM	102
7.4	Average Performance vs Evaluated Samples	105
7.5	Comparing the overfitting for the six models	107
7.6	Search Acceleration when using Perceptrons	108
7.7	Deep Learning for Iterative Compilation	111

A.1	Building a dataset with MAGIC	128
A.2	Create and Run Sessions in MAGIC	132
B.1	AST from ROSE Compiler	139
B.2	AST with different formating	141

ABSTRACT

Graph-structured data is used in many domains to represent complex objects, such as the molecular structure of chemicals or interactions between members of a social network. However, extracting meaningful information from these graphs is a difficult task, which is often undertaken on a case by case basis. Devising automated methods to mine information from graphs has become increasingly important as the use of graphs becomes more prevalent. Techniques have been developed that adapt algorithms, like support vector machine, to extract information from graphs with minimal preprocessing. Unfortunately, none of these techniques permit the use of deep neural networks (DNNs) to learn from graphs. Given the potential of DNNs to learn from large amounts of data, this has become an important area of interest. Recently, a technique based on graph spectral analysis was proposed to characterize graphs in a way that allows them to be used as input by DNNs.

We used this technique to apply DNNs to two different systems problems, i.e., 1) classifying malicious applications based on graph-structured representations of executable code and 2) developing prediction models that assist in iterative compilation to optimize and parallelize scientific code. Our results on malicious application classification show that graph-based characterizations increase the ability of DNN to distinguish malware from different families. We performed a detailed evaluation of deep learning applied to state-of-the-art and graph-based malware characterizations. The graph-based characterizations are obtained by reverse engineering potentially malicious applications. For performance prediction, the graphs represent versions of optimized code. We use machine learning to rank these versions and inform an iterative compilation process. The models are trained using only five percent of the search space.

Our work shows that graph structured data can be used to build powerful deep learning models. The techniques developed for this dissertation shows great potential in a diverse pair of systems.

Chapter 1

INTRODUCTION

Over the past several decades, machine-learning has revolutionized our lives. From individually targeted advertisements to self-driving cars, machines are getting better at solving tasks once reserved for humans. In the last few years, deep learning has significantly advanced. Last year, Kaiming He et al trained the first deep neural network (DNN) model to surpass humans at an image classification task [He et al., 2015]. The authors note that their research was not proof that computer image identification, in general, was better than humans. However, this work showed that deep learning excelled at specific “fine-grained recognition” problems. Where humans see a dog or a bird, this model can distinguish their species with greater specificity than humans.

The algorithms used for deep learning have been known for many years. However, it is only recently that these algorithms have shown great promise due to the large amounts of data and computation they require to be trained. The kind of computation capability needed to construct large models with millions of training examples has only been widely available for a few years.

This dissertation pertains to using compiler internal representations (IRs) and machine learning algorithms to solve hard system problems. Compiler IRs are typically directed graphs used to represent code of a program. Learning from graphs such as these requires special techniques to handle their size and non uniformity. Not all techniques to handle graphs can scale to tens or hundreds of millions of graphs. Deep learning often performs better with large datasets, which creates a trade off between the size of the dataset and the computational complexity of the technique used to handle graphs. Also, techniques enabling deep learning to use graph-structured data

are often limited to domain specific transformations that can not be applied in the general case.

Machine learning plays an important role in improving compiler technologies. As compilers have to handle an ever growing base of codes, there are many challenges to be addressed, some of which are related to computational performance or malware classification, specifically addressed in this proposal. However, many other problems involving code can be represented as graphs. For example, we could inspect the source code of applications to detect duplicated code, stolen intellectual properties, or vulnerable programming patterns. These problems can be solved by analyzing the abstract syntax tree (AST) representing the source code. Unfortunately, with the current state-of-the-art, applying machine learning to solve these problems requires definition of domain specific transformations of the AST. Enabling the widespread application of machine learning to compiler technologies requires automated methods to characterize graphs. The doctoral research presented in this dissertation presents general purpose techniques to use neural networks, especially deep learning, with graph-structured data.

We evaluate this technique on various compiler graphs. Using graph spectral analysis, we extract the *graph spectral features* of compiler graphs. We apply this technique to two hard system problems.

The first system problem is the classification of malicious applications. We used graph spectral features to add graph-based characterizations of executable files to existing characterization techniques. Compiler graphs are extracted using a disassembler and used alongside state-of-the-art characterizations to classify potentially malicious applications. We compare the performance of multiple neural network architectures applied to these characterizations.

The second system problem is the optimization of computation kernels. We designed a directive-based language, named TileK, to expose the large optimization space of hardware accelerators. Computation kernels annotated using TileK are translated to OpenCL and offloaded to general purpose graphics processing units (GPGPUs). We show that, unsurprisingly, the best optimization for a computation kernel depends on

the inputs being processed. The resulting tuning space is so large that exhaustively evaluating it is prohibitively expensive. Instead, we used iterative compilation aided by performance prediction models. These models are neural networks which predict the performances of optimized kernels based on the graph-structured internal representation of TileK. The prediction of this model are used to construct a ranking model which informs an iterative compilation process.

The contributions of this dissertation are:

- a method to characterize graphs with a fixed size representation
- neural network architectures for different topologies of data
- accelerated iterative compilation using the internal representation of a directive-based compiler

Outline

The beginning part of this dissertation is dedicated to the background information needed to understand the work presented in later chapter. In Chapter 2, we introduce various machine learning algorithms. In particular, we discuss deep learning, support vector machine, and random forests. We also describe the proper experimental process for machine-learning, including the various metrics used to evaluate models. In Chapter 3, we discuss graphs and how they are used in machine learning. We introduce some formal notations to help understand their utilization in machine learning. Specifically, we introduce the use of graph spectral analysis that allows us to train on graphs with deep learning.

The next two chapters are dedicated to the malware family classification problem. In Chapter 4, we provide a detailed presentation of our first malicious application classification model. This model combines deep neural networks with a random forest and was trained to classify malware into one of eleven families of malicious applications targeting financial institutions. There are two main methods we used to characterize the files: using raw bytes of the file or the graph of the disassembled code. Raw bytes consists of the variable length sequence of bytes composing a file, which are processed

using conventional sequence analysis techniques. We also extract disassembled code and represent it using our graph characterization technique. we investigate three different compiler IRs: call graphs (CGs), control flow graphs (CFGs), and instructions graphs. In Chapter 5, we explore a large space of neural network architectures applied to an extended malware dataset. The extended dataset includes some new characterizations, additional malware families, and goodware. The new characterizations are built using the ASCII strings in the file and the metadata and import table of Windows executables. The resulting dataset has a total of seventeen malware families in addition to goodware.

The next two chapter are dedicated to performance tuning for accelerators using graphs and machine learning. In Chapter 6, we present a compiler we developed that we named TileK. TileK transforms codes annotated using compiler directives to leverage parallel architecture such as GPGPUs. It exposes a large optimization space enabling the generation of many versions of a given computation kernel. Each optimized version of a kernel uses a different tiling configuration, changing the pattern in which computations are executed and data is accessed. We evaluated TileK on four representative computation kernels for a variety of inputs. Our experiments show that the best optimized version of a kernel depends on the data being processed. We show in this work that finding the best version for any input is a lengthy process. In Chapter 7, we use a combination of neural networks and support vector machines to build performance predictors. We compare two approaches to characterize graphs for neural networks. These models are used to guide an iterative compilation process. We show that a fully automated using machine learning with graphs can dramatically speed up the search for the best optimized kernel given a particular input.

In the last chapter, we conclude the dissertation and offer some insight and discuss possible future work. We end the dissertation with three appendices discussing additional technical aspects of our machine learning backend (MAGIC), the ROSE Compiler, and the directive-based parser module in ROSE.

Chapter 2

MACHINE LEARNING BACKGROUND

2.1 Introduction

Machine Learning (ML) has been slowly, but steadily changing the way we use data. The large amount of data generated by organizations can be a highly-valued asset when analyzed with the proper ML algorithm. Out of these algorithms, Deep Neural Networks (DNNs) have recently shown the most promise.

In this chapter, we present background knowledge about ML. In Section 2.2, we describe the basics of Deep Neural Networks (NN). In Section 2.3, we present Random Forest, another ML algorithm which we use to build consensus of NN. In Section 2.4, we present Support Vector Machine (SVM). In Section 2.5, we describe experimental methods and evaluation metrics specific to Machine Learning.

2.2 Deep Learning

Deep learning is the set of techniques used to train deep neural networks. In this section, we present neural networks, how they are trained, and what “deep” means. We start by introducing the model behind artificial neural networks. Then, we present how they are trained to solve specific problems. Finally, we discuss the meaning of deep learning.

2.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are models made up of interconnected artificial neurons. These artificial neurons are loosely based on how actual neurons work. A single artificial neuron is depicted in Figure 2.1. This neuron has three inputs x_0 , x_1 , x_2 and four parameters w_0 , w_1 , w_2 , and b . The w_i parameters are the respective

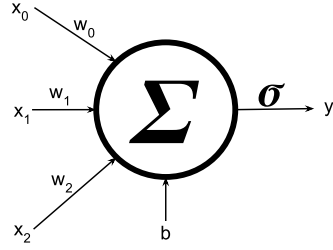


Figure 2.1: Depiction of one artificial neuron.

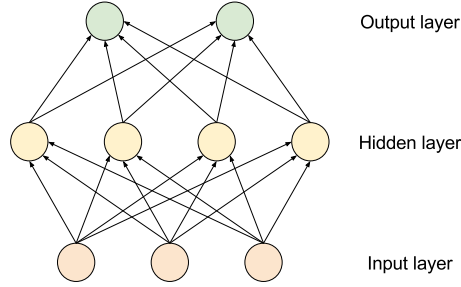


Figure 2.2: Multilayer perceptron (MLP) with a single hidden layer. All neurons from the input layer are seen by all neurons of the hidden layer which themselves are seen by all neurons of the output layer.

weights for each of the x_i inputs and parameter b is called the bias. The output y of such an artificial neuron is computed using Equation 2.1. The function σ , called activation function, is typically a sigmoid function (function with a characteristic “S”-shaped curve). Examples of sigmoid functions are the hyperbolic tangent or the logistic function.

$$y = \sigma(w_0 * x_0 + w_1 * x_1 + w_2 * x_2 + b) \quad (2.1)$$

Artificial neurons can have any number of inputs. Given a neuron with n inputs, $x \in \mathbb{R}^n$, the neuron will have n weights, $w \in \mathbb{R}^n$, one weight per input. Equation 2.2 represents the function realized by one neuron.

$$\begin{aligned} f_{w,b}^\sigma : \mathbb{R}^n &\rightarrow \mathbb{R} \\ f_{w,b}^\sigma(x) &= \sigma(w \cdot x + b) \end{aligned} \quad (2.2)$$

For the remaining parts of this section, we will focus on layered ANNs, especially multilayer perceptrons. The perceptron algorithm was invented in 1957 at the Cornell

Aeronautical Laboratory by Frank Rosenblatt [Rosenblatt, 1957]. However, this first attempt did not deliver the expected results. It took Marvin Minsky and Seymour Papert, in their work published in the book “Perceptron” in 1969 [Minsky and Papert, 1969], to describe the theoretical results explaining the shortfall of a single layer perceptron. Unfortunately, their results were misunderstood by many who believed that multilayer perceptrons had the same shortfall. This was disproved by Stephen Grossberg in 1972 with a series of papers introducing networks capable of modelling differential, contrast-enhancing and XOR functions [Grossberg, 1972a] [Grossberg, 1972b] [Grossberg, 1972c]. The 1980s saw a resurgence of research on perceptrons. Today, artificial neural networks are used extensively and multilayer perceptrons (MLPs) continue to be popular.

Formally, an MLP consists of multiple *layers* of nodes, as shown in Figure 2.2. MLPs distinguish three type of layers: the input layer, the hidden layer(s), and the output layer. Nodes in the input layer simply present an input to the network while the nodes in the hidden and output layers are artificial neuron. In an MLP, the input flows from one layer to next, i.e., it is a *feed-forward* ANN. In addition, MLPs are typically *fully-connected* networks, all outputs of one layer are inputs to the next layer. All the neurons in one layer see the outputs of all the neuron in the previous layer. Finally, the activation function is the same for all neurons in one layer. We consider an MLP with:

- N_{hidden} : number of hidden layers
- N_{in} : number of inputs
- N_{out} : number of outputs

The k^{th} layer ($0 \leq k \leq N_{hidden}$) of this MLP is defined by:

- n_k : the number of neurons
- σ^k : the activation function ($\mathbb{R} \rightarrow \mathbb{R}$)
- $w^k \in \mathbb{R}^{n_k \times n_{k-1}}$: the matrix of weights of the neurons, layer k has n_{k-1} inputs

- $b^k \in \mathbb{R}^{n_k}$: the biases of the neurons

Using this notation, Equation 2.3 presents the function realized by layer k . In this equation, $\cdot\sigma^k$ represents the point-wise application of σ^k ($\sigma^k : \mathbb{R}^{n_k} \rightarrow \mathbb{R}^{n_k}$).

$$\begin{aligned}
\mathcal{F}^k : \mathbb{R}^{n_{k-1}} &\rightarrow \mathbb{R}^{n_k} \\
\mathcal{F}^k(x) &= (f_{w_0^k, b_0^k}^{\sigma^k}(x), \dots, f_{w_{n_k-1}^k, b_{n_k-1}^k}^{\sigma^k}(x)) \\
&= (\sigma^k(w_0^k \cdot x + b_0^k), \dots, \sigma^k(w_{n_k-1}^k \cdot x + b_{n_k-1}^k)) \\
&= \cdot\sigma^k(w^k \cdot x + b^k)
\end{aligned} \tag{2.3}$$

Then, the MLP can be expressed as in Equation 2.4.

$$\begin{aligned}
\mathcal{F} : \mathbb{R}^{N_{in}} &\rightarrow \mathbb{R}^{N_{out}} \\
\mathcal{F}(x) &= (\mathcal{F}^{N_{hidden}} \circ \dots \circ \mathcal{F}^0)(x)
\end{aligned} \tag{2.4}$$

The function realized by this MLP is the composition of the functions of the hidden layers and output layer. The first take away from this equation is that if all of the activation functions are linear then the network's function is linear. In this case, the network could be reduced to a single layer, explaining the importance of non-linear activation functions. Another take away is that given an infinite number of layers with non-linear activation functions, an MLP can approximate any (measurable) function. This result is proven in [Hornik et al., 1989] and [Hornik, 1991].

2.2.2 Training Neural Networks

In the previous section, we defined how artificial neural networks, especially multilayer perceptrons, are constructed from artificial neurons. In this section, we discuss how ANN are trained. We will focus on supervised learning for MLPs.

In supervised learning, the goal is to train a model to realize a target function $\varphi : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Supervised learning requires the existence of a *training set* $\Omega \subset \mathbb{R}^n$ such that for all $x \in \Omega$, $\varphi(x)$ is known. An MLP built to fit the function φ would have n inputs and m outputs. We denote by \mathcal{F} the function realized by this network and Θ the set of all parameters of the network (weights and biases).

The basic method used to train ANNs is to use backward propagation of errors (or backpropagation) [Rumelhart et al., 1988] in conjunction with an optimization method such as gradient descent. The goal is to find the values of the parameters Θ that minimize a loss function \mathcal{L} . It is common to use $\mathcal{L}(x) = \|\varphi(x) - \mathcal{F}(x)\|^2$. In its simplest implementation, backpropagation is applied sequentially to each training sample $x \in \Omega$:

1. **Forward propagation:** computes $\mathcal{L}(x) = \|\varphi(x) - \mathcal{F}(x)\|^2$
2. **Backward propagation:** evaluates $\frac{\partial \mathcal{L}}{\partial p}(x)$ for all parameter p of the network
3. **Update:** $\forall p \in \Theta; p \leftarrow p - \alpha \frac{\partial \mathcal{L}}{\partial p}(x)$ where $\alpha \in \mathbb{R}^{*+}$ is called *learning rate*

The first step simply computes the output of the network for the input x . The second step is the actual backpropagation used to evaluate the influence of each parameter on the loss function. Finally, the parameters are updated using gradient descent. The learning rate, α , (alongside the number of layers, size of the layers, and the activation function of each layer) is part of the *hyper-parameters* of the network. This process is repeated multiple times for all training samples. Applying this algorithm to all the training sample is called an *epoch*. Training an ANN takes many epochs (each training sample is seen many times).

An extension of the classic backpropagation algorithm is stochastic gradient descent (SGD) which is commonly used with *mini-batch*. Mini-batches were first proposed in [Kramer and Sangiovanni-Vincentelli, 1988] as a method to apply conjugate gradient. In SGD with mini-batch, each epoch sees the training set Ω divided into a number of randomly selected subsets, $\tilde{\Omega}_i$, and the parameters are updated based on the sum of the gradient of \mathcal{L} for all $x \in \tilde{\Omega}_i$. The corresponding update equation is present in Equation 2.5

$$\forall p \in \Theta; p \leftarrow p - \alpha \sum_{x \in \tilde{\Omega}_i} \frac{\partial \mathcal{L}}{\partial p}(x) \quad (2.5)$$

When the size of mini-batches is well chosen, SGD converges faster than other implementation of backpropagation [Li et al., 2014b]. This is due to two factors: (1) the

gradient accumulated over multiple samples is a better approximation of the real error, and (2) the random order in which the sample are selected prevents the network from oscillating in the neighborhood of a solution. The size of the mini-batches is another hyper-parameter.

Hyper-parameters do not directly affect the function being learned, but they influence the convergence of the learning algorithm. The optimal values of these hyper-parameters is specific to the problem being learned. In most cases, they are tuned manually through trial-and-error and rely on domain knowledge of the problem. However, various methods have been evaluated to automate this process: grid search [Bengio et al., 2007], random search [Bergstra and Bengio, 2012], Bayesian optimization [Domhan et al., 2015], Covariance Matrix Adaptation Evolution Strategy [Loshchilov and Hutter, 2016], and Non-Probabilistic Radial Basis Function Surrogate Model [Ilievski et al., 2016].

2.2.3 Deep Neural Networks

The word “deep” in deep neural network (DNN) refers to the large number of layers used to form these models. DNNs were introduced very soon after the introduction of ANNs [Grigorevich and Lapa, 1966] and [Ivakhnenko, 1971]. However, the computing capabilities necessary to train such large networks was not widely available until the early 2000s. Increased compute capabilities of cloud computing and GPUs has created a renewed interest in DNNs. Aside from the democratized access to powerful computers, improved convolutional neural networks algorithms also played a role in recent deep neural architecture successes.

Convolutional neural network (CNN) is a bio-inspired technique that reduces the number of parameters in a DNN. CNNs are inspired by the organization of the animal visual cortex [Hubel and Wiesel, 1968]. Two types of layers are introduced by CNNs: (1) convolution layers, and (2) pooling layers. Convolution layers exploit spatial locality of the input. They apply a learned convolution filter on neighborhoods of the input extracting features for each of them. The number of parameters in a

convolution layer only depends on the size of the filter (usually tens of parameter) and not the size of the input (often measure in millions). Pooling layers select the most representative features from multiple neighboring neighborhoods. These layers do not have any parameters that need to be trained. Figure 2.3 depicts a CNN used for image recognition.

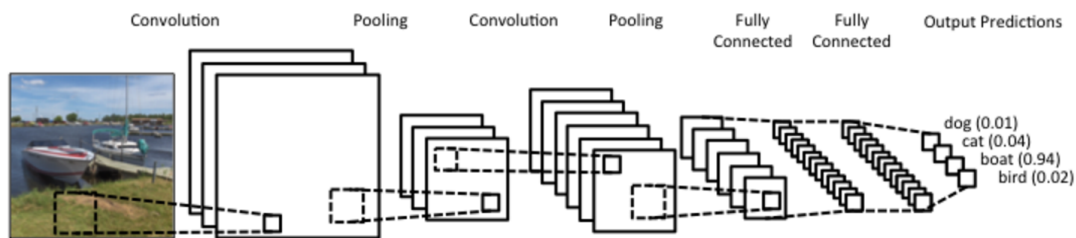


Figure 2.3: The filter from the first convolution layer is applied to a sliding window on the input image. The first pooling layer select the most significant features extracted by this filter. Then the second convolution layer and pooling layer are applied. Finally, three fully-connected layers use the features from the second stage to identify objects in the image. Image credit: [Denny Britz, 2015]

While the utilization of CNNs greatly reduces the complexity of the models (in terms of number of parameters), training models on classic CPUs is still relatively slow. General purpose graphics processing units (GPGPUs) provided a solution. The technology of GPGPU comes from the domain of graphics processing, and graphics processing pipelines are well suited to run DNNs. Indeed, GPGPUs provide massive parallelism which have been leveraged to build high-performance libraries for deep learning. Today, GPGPUs are key players in the increasing success of deep neural networks.

2.2.4 Implementation

Our experiments with deep learning are being conducted using Theano [Theano, 2016]. We are developing our own framework around Theano, see Appendix A. Theano provides low-level primitives to define mathematical expressions which we use to implement deep learning algorithms. We decided to use Theano instead of higher level

frameworks like TensorFlow, Caffe, or Torch because it gives us the control needed to evaluate new techniques. Most existing high-level deep learning frameworks focus on making it easy to construct common models. They permit the construction of convolutional neural networks for image processing or large recommendation systems. In contrast, our framework does not consider any specific use-case and only provides low-level primitives. By doing so, it permits us to define any neural network we might need. In addition, this framework has been constructed to permit scaling of model exploration for very large datasets on distributed memory systems. Finally, Theano can leverage GPUs to accelerate DNN training and evaluation.

2.3 Random Forest

Random (decision) forest models can be used to solve both classification and regression problems. This technique is an extension of decision trees. A decision tree is a flow-chart-like structure, where each internal (non-leaf) node denotes a test on an attribute, each branch represents the outcome of a test, and each leaf (or terminal) node correspond to a class label. The idea of random forests was first introduced by [Ho, 1995] and formally defined by [Breiman, 2001]. A random forest uses a collection of decision trees built on different subsets of the feature space. The advantage of a random forest is that it minimizes over-fitting during the training phase, yielding models that generalize better.

We train random forests using the scikit-learn module for ensemble learning. Also, a unified interface permits switching between machine learning algorithms easily (e.g., using SVM instead of random forests).

2.4 Support Vector Machine

Support vector machine (SVM) is a machine learning (ML) technique mostly used for supervised learning. SVM models are trained using the maximum-margin hyperplane algorithm [Boser et al., 1992].

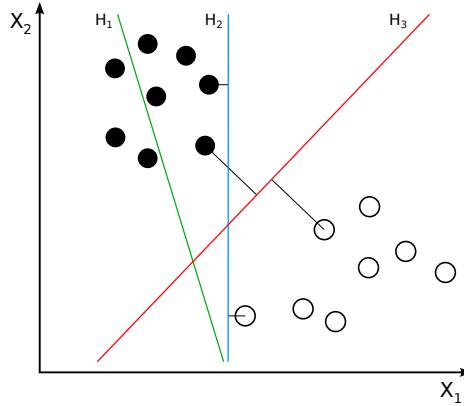


Figure 2.4: Two distribution of points and three hyperplanes: H_1 does not separate the points, H_2 and H_3 do separate the two distributions, but H_3 is the maximum-margin hyperplane.

Figure 2.4 depicts a classification problem where two groups of points, whites and blacks, have to be separated. These points are placed in a 2D euclidian space, called the feature space. The figure shows three hyperplanes in this feature space. The first hyperplane, H_1 , does not separate the two classes of points but H_2 and H_3 do. H_3 is a better separator than H_2 because it is further from any points in each classes. H_3 is one maximum-margin hyperplane.

The “kernel trick” [Muller et al., 2001] enables one to train models where the distance between points is defined by a kernel function. A kernel function, or positive definite kernel, is a generalization of a positive definite function. With the kernel trick, the maximum-margin hyperplane is fitted in a projected feature space. In the original feature space, the separator is a curved line as shown in the left side of Figure 2.5.

The primary usage of the kernel trick is to construct non-linear separators. However, it permits us to apply SVM to highly dimensional data. In Section 3.3, we introduce a few kernels that measure similarities between graphs.

The python toolkit “scikit-learn” [Pedregosa et al., 2011] provides an accessible SVM implementation. This module uses the LIBSVM library [Chang and Lin, 2011] which has been actively developed since 2000. Scikit-learn provides a flexible but unified way to construct machine learning models whether we use SVM or any other

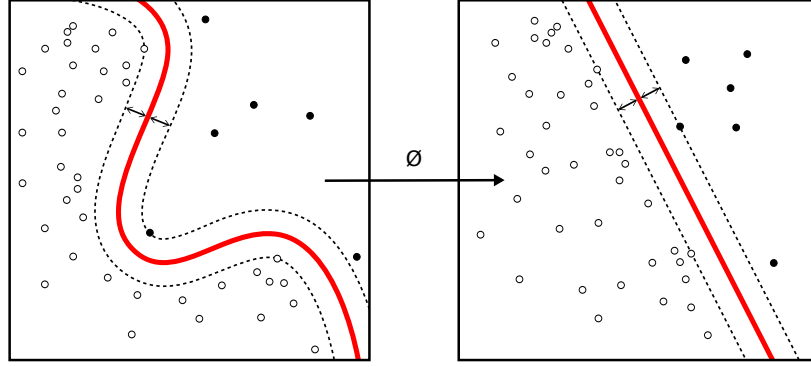


Figure 2.5: On the left side: original feature space. On the right side: transformed feature space. Features are transformed into a higher dimensional space where a linear separator can be found.

supported algorithm.

2.5 Comparing Models

When machine learning is used to solve a task, multiple algorithms are used to build various models until the best model is found. To construct the best model for a specific task, we generally evaluate the ability of these models to generalize what was learned on a training set to a testing set. This process requires both a proper experimental setup and a measure of the accuracy of a given model. We use (n-fold) cross-validation and different measures of accuracy. We use *accuracy* where others use *error* because error is only one possible measure of accuracy.

2.5.1 Cross Validation

When we evaluate a model, we use non-intersecting subsets of the whole dataset. A machine learning model is trained using the *training set*. The *accuracy* of the model is reported using the *testing set*. Separating training and testing instances ensures that the reported *accuracy* is representative of the usefulness of the model to generalize what was learned from the training set. Estimated using previously unseen instances, the *testing accuracy* is an approximation of the *generalization accuracy*. If the machine learning algorithm tunes hyper-parameters or trains multiple models, a *validation set*

is set aside for this purpose. As the validation set is never used for training, models are selected based on the generalization accuracy. However, once selected, a model has seen the validation set, but the *testing accuracy* is still based on unseen instances.

The *testing accuracy* reported for one split of the dataset is not always the best estimate of the *generalization accuracy*. Typically, multiple splits of the dataset are used and the *generalization accuracy* is estimated by the mean of model’s accuracy across all test sets. n -fold cross-validation is a strategy to construct the combination of training and testing sets. It divides the dataset into n subsets of the same size, called folds. Then, n models are built, each trained on a different combination of $n - 1$ folds and tested on the left-out fold. A method to obtain a precise estimate of the *generalization accuracy* of classification models was defined by [Zeng and Martinez, 2000]. Based on n -fold cross-validation, distribution-balanced stratified cross-validation (DB-SCV) builds homogeneous folds decreasing the deviation between cross-validation experiments. Stratification means that all folds contain the same number of instances for each classification target, giving them the same class distribution as the original dataset. With DBSCV, the intra-class distribution of the instance in each is the same as for the original dataset. This improves the estimation of the *generalization accuracy* especially for small datasets with intra-class clusters.

2.5.2 Accuracy

Let us consider a dataset which associates whether or not some property is true or false for each instance. This dataset is defined as $D = \{\mathcal{X}_i \mapsto \mathcal{Y}_i\}$ where \mathcal{X}_i represents the features and \mathcal{Y}_i is either 0 for **false** or 1 for **true**. We consider a classifier trained on a subset of D and evaluate its performance on the left over instances D_{test} . Given the feature of an instance \mathcal{X} , a trained classifier returns the probability of this instance being **true**. The output of the classifier for instance \mathcal{X}_i , denoted p_i , is a probability ($p_i \in [0, 1]$).

2.5.2.1 Error Rate

First, we define the notion of correctness of a classifier's prediction. The classifier returns a probability and a threshold τ is needed to decide whether the prediction is true or false. ρ_i^τ is the prediction for instance i given the threshold τ as defined in Equation 2.6.

$$\rho_i^\tau = \begin{cases} 0 & p_i \leq \tau \\ 1 & p_i > \tau \end{cases} \quad (2.6)$$

In Equation 2.7, we define ϵ_i^τ representing whether or not instance i was miss-predicted.

$$\epsilon_i^\tau = \begin{cases} 0 & \rho_i^\tau = \mathcal{Y}_i \\ 1 & \text{otherwise} \end{cases} \quad (2.7)$$

The error-rate of the model, denoted E^τ , is the number of miss-predictions divided by the number of predictions ; as show in Equation 2.8.

$$E^\tau = \frac{\sum_{i \in D_{test}} \epsilon_i^\tau}{|D_{test}|} \quad (2.8)$$

2.5.2.2 Confusion Matrix

To get a more accurate depiction of the accuracy of a model, we may want to know what are the errors made. The confusion matrix for a binary classifier is a two-by-two matrix counting the pair of predicted and actual targets. The columns of this matrix represent the actual targets while the rows represent the predicted targets.

		targets	
		0	1
predicted	0	TN	FN
	1	FP	TP

Table 2.1: Confusion Matrix for a binary classifier.

In Table 2.1, we show a depiction of a confusion matrix. An actual confusion matrix would show numbers of True Negatives (false predicted false), True Positives

(true predicted true), False Negatives (true predicted false), and False Positives (false predicted true). Again, these values depend on the threshold τ and can be used to define respective rates as shown in Equation 2.9. These rates are used to set the threshold τ depending on the utilization of the model. For example, a spam detector needs to minimize the false positive rate as it corresponds to desired emails being classified as spam.

$$\begin{aligned} \text{TNR}^\tau &= \frac{\sum_{i \in D_{test}} \mathbb{1}_{\mathcal{Y}_i=0} (1 - \epsilon_i^\tau)}{|D_{test}|} & \text{FNR}^\tau &= \frac{\sum_{i \in D_{test}} \mathbb{1}_{\mathcal{Y}_i=1} (1 - \epsilon_i^\tau)}{|D_{test}|} \\ \text{FPR}^\tau &= \frac{\sum_{i \in D_{test}} \mathbb{1}_{\mathcal{Y}_i=0} \epsilon_i^\tau}{|D_{test}|} & \text{TPR}^\tau &= \frac{\sum_{i \in D_{test}} \mathbb{1}_{\mathcal{Y}_i=1} \epsilon_i^\tau}{|D_{test}|} \end{aligned} \quad (2.9)$$

2.5.2.3 Receiver Operating Characteristic and Area Under the Curve

The receiver operating characteristic (ROC) comes from signal detection theory and is a good way to visualize the performance of a binary classifier. The ROC is constructed by plotting the pairs of FPR^τ and TPR^τ for various values of the threshold τ . An ROC plot permits the comparison of models visually. Using the area under the ROC curve (AUC of ROC, simply referred to as AUC) to estimate the “goodness” of a model was proposed by [Bradley, 1997]. It is usually evaluated by sampling τ in $[0, 1]$ using trapezoidal integration. The AUC ranges from 0.5 (a classifier with no better accuracy than chance) to 1 (a classifier with perfect accuracy).

2.5.3 In practice

In Chapter 4, we consider both AUC and error-rates to compare accuracy. Error rate is used for qualitative comparisons, i.e., which model makes the best predictions? AUC is used for quantitative comparisons, i.e., which model is the best at differentiating instances? Error rate tells us which model is the best to use as it is, while AUC tells us whether or not a model has potential.

Particularly, in the case of multiclassifiers, a model can have a high global error-rate while the AUC for each class is high. In this case, the model can be improved by applying a decision tree to its outputs. The decision tree can leverage the relative

probabilities of each class to make its decision. In Chapter 4, we show how we use this technique to create better classifiers from MLPs.

2.6 Conclusion

In this chapter, we provided background knowledge on machine learning in general. The next chapter presents formalisms to help describe our graph-based techniques, and subsequently in this dissertation we introduce the techniques we developed to allow graphs to be ingested by our machine learning algorithms.

Chapter 3

MACHINE LEARNING FOR GRAPHS

3.1 Introduction

Graphs are complex data structures that can be used to represent a variety of objects. Graph-structured data are now appearing frequently in machine learning and data mining literature.

In this chapter, we discuss the utilization of graphs to characterize code for two different problems involving machine learning, malware classification and code optimization. We discuss the general application of machine learning that use graphs as features. In Section 3.2, we introduce basic graph formalism and notations. In Section 3.3, we discuss the application of Support Vector Machines to graph structured data. In Section 3.4, we describe graph spectral analysis and Section 3.5 discusses how we use this analysis to generate graph spectral features that gives us a fixed-size representation of graphs we can use for deep learning.

3.2 Graph Formalism

In this work, we consider directed graphs with weighted edges. Features (vector of m reals) are associated with each node. Edges are weighted using real values.

$$\mathcal{G} = (\mathbb{V}, \mathbb{E} \subset \mathbb{V} \times \mathbb{V}, \mathcal{F} : \mathbb{V} \mapsto \mathbb{R}^m, \mathcal{W} : \mathbb{E} \mapsto \mathbb{R}) \quad (3.1)$$

Equation 3.1 gives a formal definition of such graph, which is represented as a tuple of four objects:

- \mathbb{V} : set of vertices (or nodes) of the graph
- \mathbb{E} : set of edges of the graphs, edges are directed from one vertex to another
- \mathcal{F} : features map associates a vector from \mathbb{R}^m to each vertex

- \mathcal{W} : weights map associates a scalar from \mathbb{R} to each edge

We also introduce:

- n : number of vertices ($|\mathbb{V}|$)
- m : number of features for each vertex
- $\epsilon : \mathbb{V} \mapsto [0 \dots n[$: a bijection giving an index to each vertex in the graph

Such graphs can be represented using the adjacency and feature matrices. The adjacency matrix, $\mathcal{M}^{adjacency}$, is defined in Equation 3.2. It is a square matrix of $n \times n$ real values. For any edge in \mathcal{G} , the corresponding cell of the adjacency matrix is assigned the edge's weight. If there is no edge between two vertices, the corresponding cell of the adjacency matrix has a value of zero.

$$\mathcal{M}_{i,j}^{adjacency} = \begin{cases} \mathcal{W}(\epsilon^{-1}(i), \epsilon^{-1}(j)) & \text{if } (\epsilon^{-1}(i), \epsilon^{-1}(j)) \in \mathbb{E} \\ 0 & \text{otherwise} \end{cases} \quad (3.2)$$

The feature matrix, $\mathcal{M}^{feature}$, is defined in Equation 3.3. It is a matrix of size $n \times m$ with real values. Each of the n rows stores the m features of one vertex.

$$\mathcal{M}_{i,j}^{feature} = \mathcal{F}(\epsilon^{-1}(i))_j \quad (3.3)$$

When it comes to learning from graphs, there are not many solutions. Graphs are high dimensional data and even in matrix form they cause issues, e.g., their sizes vary with the number of nodes. We will present two ways of utilizing ML on graphs: graph kernels with SVMs and graph spectral features with deep learning.

3.3 Graph Kernels for Support Vector Machine

It is possible to use graph kernels with kernelized learning algorithms like SVM. Graph kernels measure similarities between graphs and are covered in details by [Vishwanathan et al., 2010]. Using graph kernels, we can bypass the fixed size constraint of many learning algorithms.

For example, the shortest paths graph kernel (SPGK) algorithm [Borgwardt and Kriegel, 2005] computes the pairwise similarity between the shortest paths of two

graphs. SPGK is one of many path-based graph kernels like *longest paths kernel* and *all paths kernel*. Compared to these other kernel algorithms, the shortest path graph problem can be solved in polynomial time while yielding similar results.

Algorithm 3.1 The shortest-path graph kernel compute the sum of the pairwise distances between all the shortest-paths of two graphs.

```

procedure ALLEDGESGRAPHKERNEL( $\mathcal{G}_0, \mathcal{G}_1$ )
   $D \leftarrow 0$ 
  for  $e_0 \in \mathcal{G}_0^{edges}$  do
    for  $e_1 \in \mathcal{G}_1^{edges}$  do
       $D^{source} \leftarrow \text{NODEKERNEL}(e_0^{source}, e_1^{source})$ 
       $D^{weight} \leftarrow \text{EDGEKERNEL}(e_0^{weight}, e_1^{weight})$ 
       $D^{sink} \leftarrow \text{NODEKERNEL}(e_0^{sink}, e_1^{sink})$ 
       $D \leftarrow D + D^{source} * D^{weight} * D^{sink}$ 
    end for
  end for
  return  $D$ 
end procedure

procedure SHORTESTPATHGRAPHKERNEL( $\mathcal{G}_0, \mathcal{G}_1$ )
   $\mathcal{G}_0^{sp} \leftarrow \text{SHORTESTPATHGRAPH}(\mathcal{G}_0)$ 
   $\mathcal{G}_1^{sp} \leftarrow \text{SHORTESTPATHGRAPH}(\mathcal{G}_1)$ 
  return ALLEDGESGRAPHKERNEL( $\mathcal{G}_0^{sp}, \mathcal{G}_1^{sp}$ )
end procedure

```

Algorithm 3.1 shows how SPGK applies the *all edge graph kernel* to the graphs after applying the shortest paths algorithm. The NODEKERNEL function takes feature vectors from two nodes and returns a measure of their differences. The EDGEKERNEL function takes as input two edges, feature vectors for source and sink node and an edge's weight for each node, and it returns the distance between these nodes. The ALLEDGESGRAPHKERNEL function can be used directly on a graph. In the case of SPGK, ALLEDGESGRAPHKERNEL is applied to the shortest path graphs of the

inputs. `SHORTESTPATHGRAPH` can be implemented using the Floyd-Warshall algorithm. Equations 3.4 and 3.5 shows typical versions of `NODEKERNEL` and `EDGEKERNEL`, respectively.

$$\text{NODEKERNEL}(v^1, v^2) = \sum_{0 \leq i < m} (v_i^1 - v_i^2)^2 \quad (3.4)$$

$$\text{EDGEKERNEL}(w^1, w^2) = |w^1 - w^2| \quad (3.5)$$

SPGK was combined with SVM to classify graph models of proteins by [Borgwardt and Kriegel, 2005]. They compare SPGK to the random walk kernel, another type of graph kernels introduced by [Gärtner et al., 2003]. Applied to the classification of proteins, SPGK achieves better than 94% accuracy compared to little more than 89% with random walk.

SPGK and other kernel techniques have been applied to multiple compiler problems. It was used to train SVM models for application performance predictions by [Park et al., 2012]. In this work, applications were characterized by their control flow graphs. Graph-based features were used to order LLVM’s optimization passes by [Nobre et al., 2016].

3.4 Graph Spectral Analysis

However, graph kernels are still computationally expensive and do not scale well with the large datasets required by deep learning. In this section, we elaborate on a much more efficient way of converting graphs to fixed size feature vectors for deep learning. The first technique is DeepWalk [Perozzi et al., 2014]. This algorithm aggregates the results of multiple random walk into one large feature vector. DeepWalk was used to analyze social media datasets improving the accuracy of the generated models while lowering the time needed for training.

Bruna et al. used spectral analysis of the graph Laplacian on the MNIST dataset (hand-written digits) [Bruna et al., 2013]. This paper introduces the mathematical tools that we leverage in our work. This technique was further leverage by Henaff

et al. with the introduction of ANN architecture similar to convolutional networks [Henaff et al., 2015]. In 2016, Bronstein et al. published a review of graph spectral techniques for deep learning [Bronstein et al., 2016].

3.5 Graph Spectral Features

Algorithm 3.2 Extracting the graph spectral features from the adjacency and feature matrices of a graph takes three steps. First, we compute the Laplacian of the graph. Second, we extract the eigenvectors for the w largest eigenvalues. Third, we project the features matrix using these eigenvectors.

```

procedure GRAPHSPSPECTRALFEATURES( $\mathcal{M}^{adjacency}, \mathcal{M}^{feature}, w$ )
   $\mathcal{M}^{laplacian} \leftarrow \text{GRAPHLAPLACIAN}(\mathcal{M}^{adjacency})$ 
   $\mathcal{M}_w^{projection} \leftarrow \text{EIGENVECTORS}(\mathcal{M}^{laplacian}, w)$ 
  return  $(\mathcal{M}_w^{projection})^T \times \mathcal{M}^{feature}$ 
end procedure

```

Graph Spectral Features (GSF) designates a technique which uses graph spectral analysis to build a fixed size representation of graphs. This technique projects the feature matrix, $\mathcal{M}^{feature}$, with respect to the w eigenvector corresponding to the w largest eigenvalues of the Laplacian of the adjacency matrix, $\mathcal{M}^{adjacency}$. This projection matrix is denoted as $\mathcal{M}_w^{projection}$. The graph spectral feature of width w is the projection of the features matrix with respect to $\mathcal{M}_w^{projection}$ and is denoted $\mathcal{M}_w^{spectrum}$. Algorithm 3.2 presents the extraction of $\mathcal{M}_w^{spectrum}$ given $\mathcal{M}^{adjacency}$ and $\mathcal{M}^{feature}$. The Laplacian matrix, also called admittance or Kirchhoff matrix, is defined in Equation 3.6, where $\mathcal{M}_{\mathcal{G}}^{degree}$ is the degree matrix of the graph \mathcal{G} . This matrix is diagonal and $\text{diag}(\mathcal{M}_{\mathcal{G}}^{degree}) = \mathcal{M}_{\mathcal{G}}^{adjacency} \cdot \mathbf{1}_n$ ($\mathbf{1}_n$ is the unit vector of length n).

$$\mathcal{M}_{\mathcal{G}}^{laplacian} = I - (\mathcal{M}_{\mathcal{G}}^{degree})^{-\frac{1}{2}} \times \mathcal{M}_{\mathcal{G}}^{adjacency} \times (\mathcal{M}_{\mathcal{G}}^{degree})^{-\frac{1}{2}} \quad (3.6)$$

This Laplacian operator is used by [Henaff et al., 2015] to define a convolutional operator on the grid (edges of the graph) by extracting its eigenvectors. Our method is similar in that the EIGENVECTORS function is applied to the Laplacian and returns the eigenvectors for the w largest eigenvalues of $\mathcal{M}^{laplacian}$. We use the resulting matrix ($\mathcal{M}^{projection}$) to project the feature matrix ($\mathcal{M}^{feature}$) into the $\mathcal{M}_w^{spectrum} \in \mathbb{C}^{w \times m}$ (m

being the number of features per node). The size of $\mathcal{M}_w^{spectrum}$ does not depend on the size of original graph, which was our goal.

3.6 Conclusion

In this chapter, we discussed the use of graphs with machine learning. After introducing graph formalism, we discussed how we use graph kernels to compare graphs with SVMs. However, using graph kernels does not scale with the big datasets required by deep learning techniques. To use deep learning on graphs, we must use graph spectral analysis to build fixed size representations. In the remaining chapters of this dissertation, we illustrate how to use graph spectral features (GSFs) to solve complex system problems involving graphs. We use GSFs to classify malware in Chapters 4 and 5 and make performance prediction in Chapters 6 and 7.

Chapter 4

BASIC NEURAL NETWORKS FOR MALWARE CLASSIFICATION

4.1 Introduction

In recent years, cybersecurity news has increasingly made headlines. Breaches of computer systems have led to the theft of private information or to a company's data being held hostage after it has been encrypted. While there are a variety of ways in which computer systems can be compromised, many cases involve malicious applications being downloaded into a organization's network without being detected. These malicious applications, also called malware, have plagued computer systems for decades. It is estimated that upwards of one million malware variants are released into the “wild” every day. The rate at which these malware are created is due to the widespread adoption of automated tools, which permit the construction of hundreds of malware variants with only a few clicks.

To fight this onslaught of malware, security software has had to move past signature-based malware detection. Cryptographic hashes used as signatures can only recognize previously seen variants of malware. But signature-based techniques are not effective when the malware attacking an organization is new malware. Better methods to characterize and classify malware are currently being developed to replace signature-based techniques. In particular, machine learning approaches have recently started to yield exceptional results

The machine learning algorithms that have recently received the most attention have been deep learning algorithms. The recent advances in deep learning are due to the low-cost availability of vast amounts of computation capabilities and large datasets. For example, in cybersecurity large amounts of labeled malware are available for free

on the internet to build malware detection models. Also, companies, such as Reversing Labs and Virus Total, provide curated streams of malware to researchers to work on. For our research, we obtained a large stream of financial malware from Reversing Labs from which we extracted 1.2 million malware for our research. Our goal was to develop models capable of predicting what family a particular unseen malware belonged to.

These models were built using a variety of analysis approaches to allow us to characterize files in terms of bytes and reverse-engineered assembly codes. We constructed a dataset consisting of eleven different feature sets of various sizes, from tens to thousands of features. We trained separate deep learning models for each feature set. Then, a consensus model was built with the best models for each feature set using ensemble learning. Ensemble models (using random forests) can be trained quickly with the output of several neural networks.

In this chapter, we show the results of training two hundred deep learning models for each of the eleven feature sets in our dataset. We trained two hundred models to get an accurate estimate of the level of performance achievable by each feature set. In this chapter, we present the malware dataset, the analyses performed, and the features extracted. Next, we describe the machine learning backend we deployed on Amazon Web Services (AWS). Finally, we show malware classification results comparing state-of-the-art feature sets against our feature sets using graphs that better represent the structure of the executable code.

4.2 Malware Datasets and Characterization

In this section, we describe our dataset of malicious applications and the various techniques we used to characterize them.

4.2.1 Dataset

We obtained 1.2 million files from Reversing Labs pertaining to a stream of malware targeting financial institutions. These malware come from forty families, targeting a variety of different operating systems. For the experiments presented in

Name	Target	Type
Andromeda	Windows	virus
Banker	Windows	spyware
Banload	Windows	downloader
Cutwail	Windows	downloader
Inject	Windows	virus
Injector	Windows	trojan
Ramnit	Windows	trojan
Shifu	Windows	spyware
Zbot	Windows	downloader
Smsagent	Android	spyware
Smsthief	Android	spyware

Table 4.1: The eleven families of malware in the dataset. We list the OS each family targets, either Microsoft Windows or Android systems. Viruses are malware that spread on a system. Spyware collect and report private data. Downloaders get other malware on the infected system. And, trojans are malware camouflaged as benign applications.

this proposal, we randomly subsampled each family with more than one thousand variants. This left us with a dataset made of one thousand files from eleven different families. Table 4.1 provides the name, OS target, and type of each family of malware.

Figure 4.1 presents histograms (500 bins) of five metrics that were extracted when characterizing the 11,000 files. These metrics are: (1) number of functions, (2) number of blocks, (3) number of instructions, (4) number of bytes, and (5) entropy. Number of functions, blocks, and instructions are computed from the reverse-engineered assembly code of the malware. Entropy corresponds to the measurement of randomness in the bytes of the malware. It is computed Shannon’s formula [Shannon, 1948] which yield a value between zero and eight.

4.2.2 Bytes-Entropy Histogram

Byte-entropy histograms are comprehensive representations of files that represent the state-of-the-art characterization of files for deep learning [Saxe and Berlin, 2015]. To construct this representation of files, we scan files using a sliding window of length 1024 with a step of 256 bytes. For each of the windows, we get the histogram

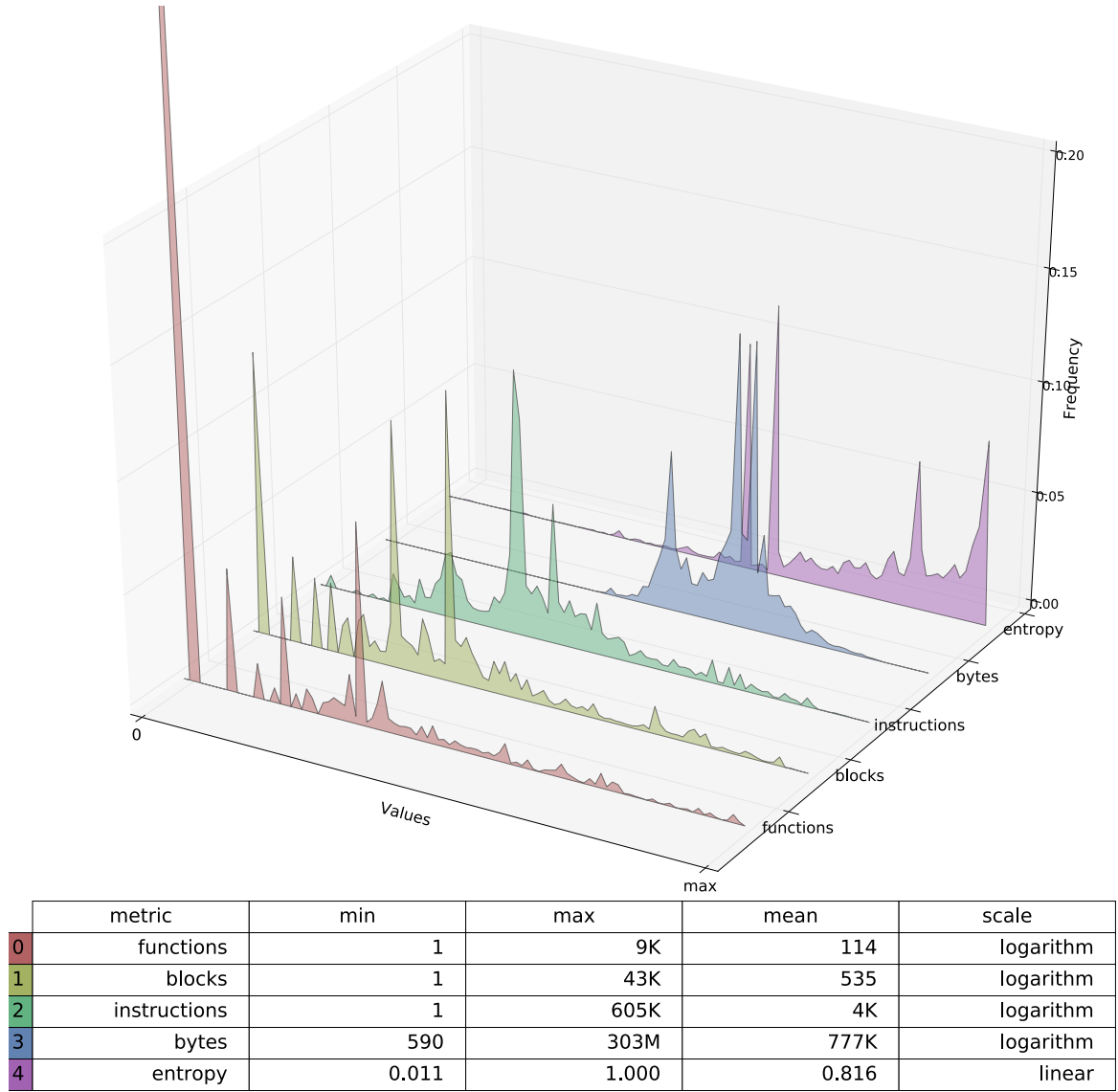


Figure 4.1: Comparing the distribution of the size of the file, the entropy, and the number of functions, blocks, and instructions over the different families of malware. The size of the file and the number of functions, blocks, and instructions are plotted using a logarithmic scale. It shows that many malware have only one function or one block. However, there can be up to 9 thousand functions, 43 thousand blocks, and 605 thousand instructions. The entropy is usually average corresponding to conventional data and code. However, there some case when the entropy is extremely high corresponding to files where the majority of the information is either compressed or encrypted.

of the bytes and compute the associated entropy. Finally, histograms are accumulated in the 2D bytes-entropy histogram in one of 256 entropy bins. Figure 4.2 and Algorithm 4.1 describe this process.

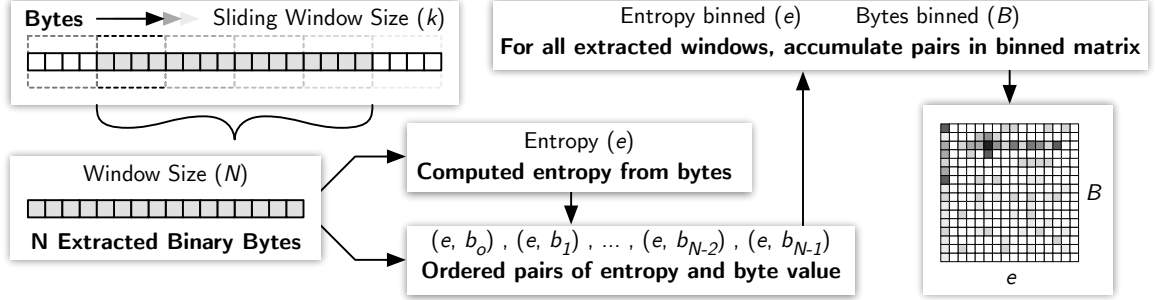


Figure 4.2: First, the file is scanned by a sliding window of length 1024 with a step of 256 bytes. Then, for each window, the bytes histogram is extracted and the associated entropy is computed. Pairs of byte and entropy are collected for all windows. Finally, the pairs are counted in the bytes-entropy histogram.

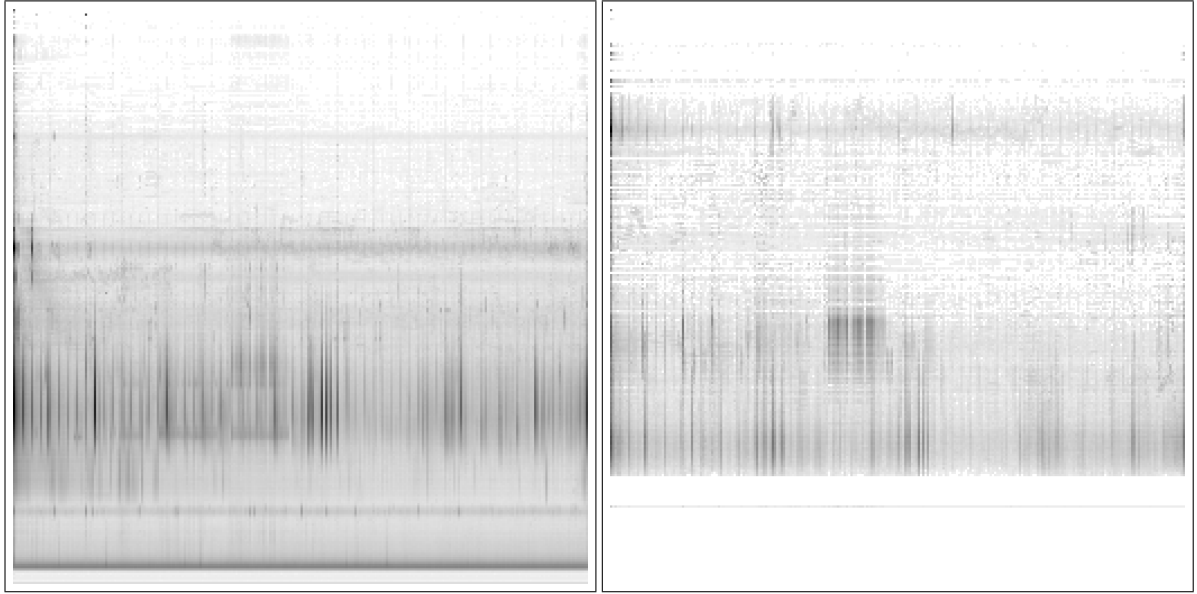
Algorithm 4.1 For each window of length W taken from the bytes B with a step of k , the histogram H and entropy E are computed. The entropy is quantized qE into En values. Finally, the histogram is accumulated into the bytes-entropy histogram based on the quantized entropy.

```

procedure BYTESENTROPYHISTOGRAM( $B, W, k, En$ )
     $BEH \leftarrow \text{ZEROS}(256 * En)$ 
    for  $i = 0$  to  $\text{len}(B)$  by  $k$  do
         $H \leftarrow \text{HISTOGRAM}(B[i : i + W])$ 
         $E \leftarrow \text{ENTROPY}(H)$ 
         $qE \leftarrow \text{floor}((E/8.) * (En - 1))$ 
         $BEH[qE, :] \leftarrow BEH[qE, :] + H$ 
    end for
    return  $BEH$ 
end procedure

```

The intuition behind the bytes-entropy histogram is that different types of data have different entropy values. ASCII text has a lower entropy than executable code and executable code has a lower entropy than encrypted or compressed data. Bytes-entropy histograms separate histograms for different types of data.



(a) Skype 4.3.0

(b) GCC 5.4.0

Figure 4.3: We provide depictions of the bytes-entropy histograms of two Linux applications. The X-axis represents the different ASCII bytes from 0 to 255. The Y-axis represents Shannon’s entropy from zeros at the top to eight at the bottom. Entropy values are discretized using 256 bins. One can note that Skype has a large amount of high entropy data while GCC has none. In GCC, data with the highest entropy are a thin line corresponding to the entropy of x86 code (6.6).

The bytes-entropy histograms for GCC 5.4.0 and Skype 4.3.0 are shown in Figures 4.3b and 4.3a, respectively. In both figures, the X-axis represents the bytes values from 0 to 255 and the Y-axis is the entropy going down (high entropy at the bottom). The main difference between these two applications is the existence of very high entropy data in Skype, corresponding to compressed data (e.g., images, sounds, and perhaps packed proprietary code). We also note the presence of lower-case characters with medium entropy in GCC (darker squared area near the center of the bytes-entropy histogram). This area corresponds to tables of lower-case strings such as the names of GCC’s built-in types, variables, and functions.

In both bytes-entropy histograms, there is a horizontal line corresponding to executable code. In GCC, it is clear, as this (thin) line is isolated at the bottom of the bytes-entropy histogram. In Skype, this line is wider because of the high entropy data.

4.2.3 Malware’s Executable Code

The analysis of executable codes is a reverse engineering process. The first step is to disassemble the executable code. It translates executable instructions to assembly instructions. Disassemblers are classic tools in the arsenal of security analysts. Most of these tools can perform high-level analyses on these instructions. These analyses identify functions and blocks of code and resolve most of the function calls and branches. Given the results of these analyses, we can construct many compiler graphs, including control flow graphs and call graphs. Finally, the compiler graphs are “regularized” to form feature graphs, following the graph formalism presented in Section ???. Here, “regularization” implies summarizing the content of the compiler graph’s nodes as vector of real numbers. Figure 4.4 depicts how the call graph with features at each node is constructed from the output of a disassembler.

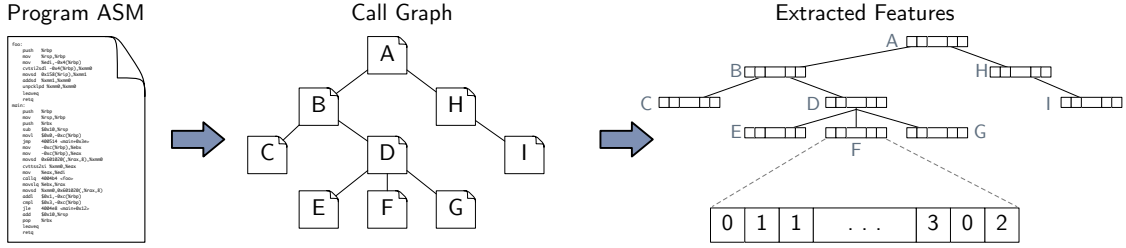


Figure 4.4: First, the executable is disassembled into function, blocks, and instructions. Second, the disassembled code is transformed into a graph-like data-structure (e.g., call graph). Third, a feature graph is extracted from the “compiler graph”.

4.2.3.1 Disassembly & Analysis

We use Radare2, a free and open-source disassembler, to analyze executable files. The advantage of Radare2 is that it “disassembles” many kinds of executables, including x86, ARM, Bytecode (Java), Javascript (from HTML files), etc. We parse Radare2 output and consolidate the results of call and control-flow analyses into one data structure depicted in Figure 4.5. This data-structure is a graph of operations linked by calls (blue curved arrows), branches (green/red angled arrows), and fallthroughs (black thin straight arrows).

Operations are characterized using Radare2 categories (Table 4.2) and the operation size. Blocks are sequences of operations not broken up by control flow. Each block is characterized by its number of instructions, size, and statistics for the edges. Edges’ statistics include the number of calls and jumps that were resolved, the number of them that were not resolved, and the number of fallthrough between instruction. Functions are composed of blocks and are characterized by their size, number of blocks and operations, and statistics of their edges. Call edges connect the call site (**call**) to the first instruction of the first block of the called function. Branches connect the last instruction of one block to the first instruction of another block. True branches (in green, connected below the instruction) are taken when a jump instruction is executed. For conditional jump, if the condition is false, then the false branch is taken (in red connected to the side of the instruction). Fallthroughs correspond to the normal flow

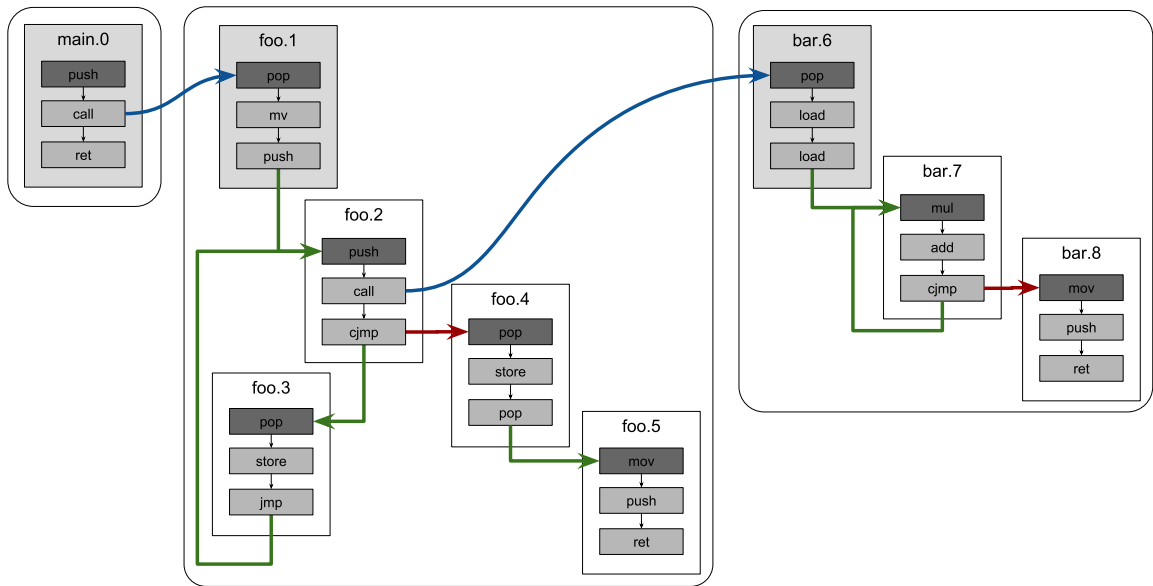


Figure 4.5: Depiction of code graph extracted from Radare2 output. At the lowest level of this graph, the nodes represents instructions. These instructions are groups into blocks which are themselves grouped into functions. Edges between instructions represents the normal flow of execution. Branches are edges between the last instruction in a block and the first instruction of another block. Edges between instructions and the first instruction of the first block of a function represents calls.

of instructions in the absence of branches. They usually include the false branches but we consider them separately.

control flow	switch ucjmp	case uccall	call ccall	ucall ret	jmp cret	ujmp swi	cjmp
arithmetic	length	cmp	acmp	add	mod	cast	not
	sub	abs	mul	div	shr	shl	cpl
	sal	sar	or	and	xor	crypto	nor
	ror	rol					
memory	mov upush	lea pop	cmov push	xchg new	leave io	store	load
miscellaneous	null	nop	unk	trap	ill		

Table 4.2: This table shows the 53 different categories of instructions extracted by Radare2. There are four major categories that the extracted instructions correspond to. They are grouped into categories of control flow, arithmetic, memory, and miscellaneous.

4.2.3.2 Code Features

To apply machine learning on files characterized using code features, we need to extract graphs and feature vectors at each node. We extract this representation by scanning this data-structure from operations to blocks to functions to global level. Along the way, we extract statistics about each level that we accumulate (Figure 4.6). This gives us four different level of granularity at which we look at the code.

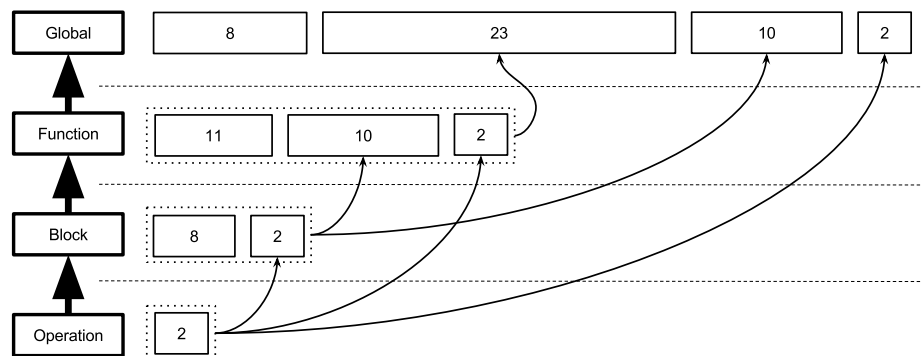


Figure 4.6: Statistics accumulation through the four levels of granularity. Statistics are built recursively starting with operations. Each block's statistics are concatenated with the average statistics of its operations. Each function's statistics are concatenated with the average statistics of its operations and its blocks. The global statistics are concatenated with the average statistics of its operations, its blocks, and its functions.

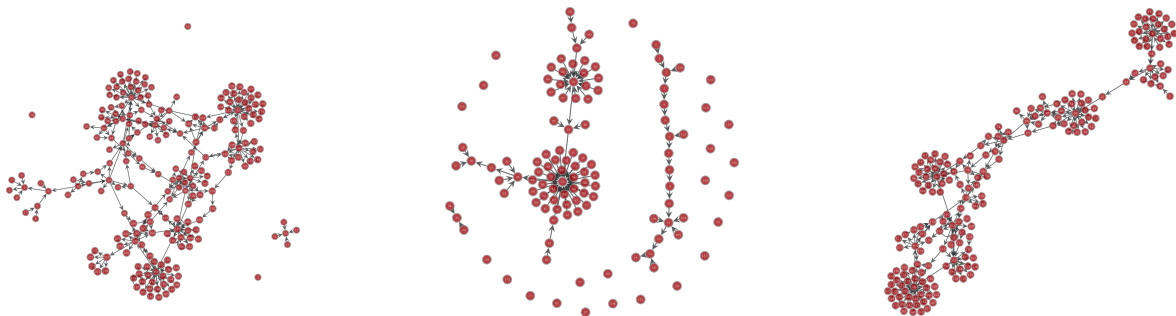


Figure 4.7: Three graphs showing the structure of extracted WPCFGs. Nodes represent blocks in the original code. They are characterized using statistics, instructions 1-gram, or instructions 2-gram. Edges represent: jumps between blocks, or calls between one block and the first block of a function. We can see that not all blocks are connected to other blocks. This is because Radare2 does not resolve all calls and jumps. We can notice that sometimes many nodes connect to one node while no nodes connect to them. These could correspond to a switch statement implemented with a relative jump. The nodes without predecessors would be the cases of this switch statement. The nodes they connect to would be the statement after the switch statement.

4.2.3.2.1 Operation Level

The whole-program instruction-flow-graph (WPIFG) connects operations using calls, branches, and fallthroughs. For each operation, one feature vector is generated containing *statistics*: operation kind and size.

4.2.3.2.2 Block Level

The whole-program control-flow-graph (WPCFG) connects blocks based on the calls and branches. Each block is characterized by three feature vectors: statistics, instruction 1-grams, and instruction 2-grams. The statistics include the block's size and edges statistics, but also aggregate the average statistics of its operations. The instruction 1-grams and 2-grams are histograms of the sequences of one or two operations presented as 1D and 2D tensors.

4.2.3.2.3 Function Level

The call-graph (CG) connects functions based on the calls. Each function is characterized by three feature vectors: statistics, instructions 1-grams, and instructions

2-grams. One function’s statistics include the function’s information, and the average statistics of its blocks and operations.

4.2.3.2.4 Global Level

We construct the same three feature vectors. The global statistics include code size, number of operations, blocks, and functions, and aggregated averages of its functions, blocks, and operations. This is three feature vectors at the global level and seven feature graphs for the other three levels.

4.2.3.2.5 Graph Spectrum

Figure 4.1 shows that, while the number of functions, blocks, and instructions can be as small as one, it can also reach thousands, tens of thousands, and hundreds of thousands, respectively. That is an issue as the structure of these graphs is important, different graph structure are shown in Figure 4.7. We discussed in Section ?? a method to summarize graphs of any size as into a 2D tensor of fixed-size. This method is the final step of our executable code characterization. For each of the three granularity levels (functions, blocks, and operations) and the three kind node features (statistics, 1-grams, and 2-grams), we extract the graph spectral features of width 20.

4.2.4 Summary of the Features

Between the byte and code level analyses, we generate eleven different features. These features are summarized in Table 4.3. It includes the bytes-entropy histogram used to characterize files at the byte level and the collection of vectors and graph spectral features obtained from the executable code.

4.3 Machine Learning in the Cloud

Our goal is to design a malware detection system that can be updated daily. Such a system will be essential to taking advantage of the millions of new malware that are created every day. It should also permit us to independently leverage the eleven

Characterization		Format	Size
bytes-entropy histogram		matrix	256 x 256
Global	statistics	vector	43
	1-grams		53
	2-grams		2809
Function	statistics	matrix	20 x 23
	1-grams		20 x 53
	2-grams		20 x 2809
Block	statistics	matrix	20 x 10
	1-grams		20 x 53
	2-grams		20 x 2809
Operations	statistic	matrix	20 x 2

Table 4.3: Summary of the feature produced by the bytes level and executable code analyses.

feature sets that are extracted by the various file analyses. To make this possible, we need a scalable machine learning service providing a schema to assemble models.

4.3.1 Training and Consensus

We decided to train deep neural networks (DNNs) on each of the feature sets as deep learning scales much better than other techniques with the size of the data. Then, we used a random forest to build consensus models using the output of the selected DNNs. This two-step approach enables the concurrent training of a large number of DNNs, which are aggregated at any time using random forests. While they are trained, DNNs save the latest best models. Hence, a new random forest can be created, at any time, using a selection of these DNNs.

Figure 4.8 shows how this schema is implemented when training, testing, and validation sets are taken into account. First, the dataset is separated into N folds (3 here for brevity and 5 in our experiments). In the case of a growing dataset, new samples are separated between the N folds. Second, one *Testing Pool* is created for each fold. Testing pools are independent from each other. The fold associated with each pool is never seen by the models in this pool (neither for training nor validation). Third, in each testing pool, $N - 1$ *Cross Validation Pools* are created, one for each

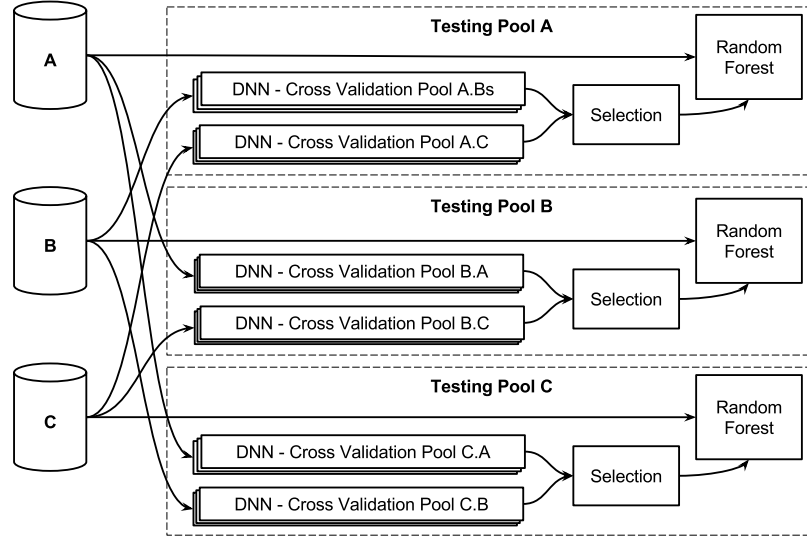


Figure 4.8: The dataset is divided in three folds. Each fold is the testing set for one testing pool. Each testing pool has two cross validation pools where DNNs are trained. Some of these DNNs are selected based on the validation accuracy. The output of the selected DNNs are used to construct a random forest. This random forest forms a consensus based on the selected DNNs.

fold not reserved for testing. In these cross-validation pools, models are trained on the left-over folds and evaluated on the associated validation fold. For example, the cross-validation pool B.A of the testing pool B in Figure 4.8 uses: fold C to train, fold A to validate, and fold B to test. In our cloud service, the models trained in the cross-validation pools are DNNs. Many DNNs are trained in each validation pool, for each of the feature sets, providing us with many DNNs to choose from. Eventually, models in each cross-validation pool could be selected using a meta-optimization algorithm, e.g., genetic algorithms. Fourth, some models are selected from all cross-validation pools in each testing pool. The outputs of these models for both training and validation folds are used to construct one random forest (or other ensemble techniques). The final results are reported using the testing fold, which was never used for training or selection.

4.3.2 Cloud Infrastructure

Our machine learning service is deployed on Amazon Web Services (AWS) to enable on demand scaling of both compute and storage resources. Particularly, we use four services from AWS:

- Elastic Compute Cloud (EC2) with Auto Scaling Group (ASG) to host the applications in a scalable way
- DynamoDB to store the descriptions of the models and datasets and to record accuracy metrics while training models
- Simple Storage Service (S3) to store the datasets and the parameters of the trained models
- Simple Queue Service (SQS) to distribute jobs

Aside from the flexibility, creating a machine learning service made it possible to have a job-based system. This job-based system is essential to distribute the workload on SPOT instances. SPOT instances are EC2 instances sold using a market pricing. This enabled us to control the cost (as SPOT instances are usually much cheaper). However, SPOT instances are volatile and can be taken down at any time. Distributing jobs with SQS permits us to monitor the jobs for completion and restart jobs that failed.

The application behind the service is developed in python and it uses Theano [Theano, 2016] for deep learning and scikit-learn [Pedregosa et al., 2011] for ensemble learning. The connection between the application and AWS infrastructure is done using boto3 [AWS, 2014] which provides python binding for AWS API. There are three modules to this application: (1) creation and management of datasets, (2) training of deep learning models, and (3) building ensemble models for consensus.

In the first module, datasets are constructed from a list of items and a list of feature sets. Then, the dataset is separated into folds that are stored separately on S3. A basic description of the dataset is stored in DynamoDB.

The second module is given a model, a dataset, and some policies: cross-validation, stopping, recording and checkpointing. The dataset is obtained (from S3 or local caches) and normalized depending on the training set. The model is either

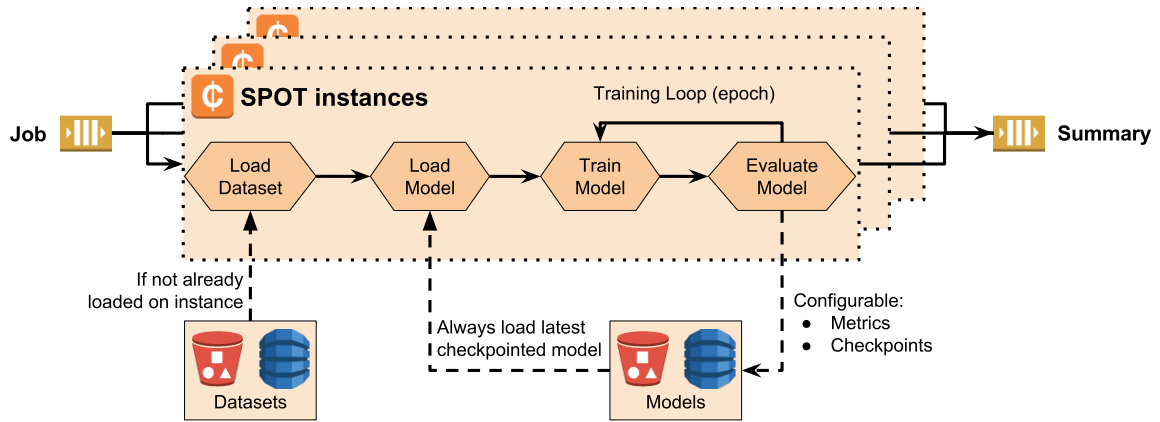


Figure 4.9: The deep learning application of our cloud service is deployed on AWS. The jobs are distributed between multiple SPOT instances using AWS’s queuing service (SQS). When a job is received, the dataset is loaded, if not already present in the instance’s cache. Then, the latest checkpointed model is obtained. The model is trained and after each epoch, metrics are saved and the model might be checkpointed.

loaded using its latest checkpointed parameters (on S3) or created locally. Finally, the model is trained until the stopping policy is triggered (number of epochs, time, error-rates, etc). In between each epoch, the model is tested on the validation fold. Then, depending on the checkpointing policies, the application might:

- record metrics in DynamoDB
- checkpoint the model’s parameter on S3

Training metrics include the error rate on each fold (computed if requested), and the time spent in the epoch. They are usually captured after each epoch. There are two types of checkpoints: best and last. The “best” checkpointing means that the parameters of the model are saved when it has the lowest error rate so far. It is usually activated, allowing the retrieval of the best version of any model at any time. The “last” checkpointing means that the parameters of the model are saved after each epoch. It is turned on when we do not want to lose epochs after a failure (when epochs are long). Figure 4.9 shows a SPOT instance (launched in an ASG) processing a DNN job obtained from SQS.

The third module is given a list of models, a training set and a testing set. It produces an ensemble model, generally a random forest, which makes predictions

based on the outputs of the given models. It computes the outputs of the models for the whole dataset. The outputs for the training set are used to train the ensemble model while the testing set is used for evaluation.

4.4 Malware Classification Results

In this section, we present the results of our malicious application classifiers. We start with a description of the deep neural networks and random forests. Then, we analyze the performance of these models.

4.4.1 Models

In these experiments, we evaluated an architecture where multiple deep neural networks make predictions and random forests are used to build consensus using these predictions (Figure 4.8). We used 5-fold cross-validation, giving us five testing pools containing four cross-validation pools each. In each cross-validation pool, we trained ten multilayer perceptrons (MLPs) for each of the eleven feature sets extracted from the malware (Table 4.3). Finally, we selected different groups of MLPs from each testing pool to construct random forests.

4.4.1.1 Multilayer Perceptrons

We evaluated each of the features sets by constructing MLPs that attempt to classify the malware into one of the eleven strains. The prediction made by these MLPs is a probability vector $p \in \mathbb{R}^{11}$ such that $\sum_i p_i = 1$. The predicted class is the class with the largest probability: $c = \text{argmax}(p)$. The depth and structure of each MLP was defined empirically. Table 4.4 gives the characterization sizes and the number of layers in the associated MLPs. For small models (approx. 50 features), the size of the layers decreases arithmetically from the number of features to the number of targets (11). For larger models, the size of the layers decreases geometrically (by factor 2) from the number of features to the number of targets. For the bytes-entropy histogram models, the size of the layers also decreases geometrically. However, the first layer is limited to 12,000 neurons as larger layers cause a crash during the network’s optimization phase.

Malware Characterization		DNN layers	DNN shape
bytes-entropy histogram		12	geometric
global	stats	5	arithmetic
	1-grams	6	arithmetic
	2-grams	9	geometric
functions	stats	7	geometric
	1-grams	8	geometric
	2-grams	12	geometric
blocks	stats	5	geometric
	1-grams	8	geometric
	2-grams	12	geometric
operations	stats	5	arithmetic

Table 4.4: Size and shape of the MLPs for each features. The shape column describes how the size of the successive layers decrease. A geometric shape means the layers sizes are divided by a constant factor. An arithmetic shape means the layers sizes are diminished by a constant step.

All hidden layers use hyperbolic tangent as their activation functions. For the output layer, we use the softmax function to ensure that the output of the model is a probability distribution. The softmax function (also called normalized exponential function) “squashes” a vector of arbitrary real values into a probability distribution.

4.4.1.2 Random Forests

In each testing pool, we have forty MLPs for each of the eleven features. We used some selected MLPs to build forty-two random forests with ten decision trees each. We built random forests using the one, two, or five best MLPs for each of the eleven feature sets. We also evaluated three groups of feature sets: state-of-the-art, executable code, and all features. For each of the three groups, we selected the one, two, or five best MLPs for each of the features. For example, the best random forest uses the five best MLPs for each of the eleven features, which means that it creates a consensus between fifty-five MLPs.

4.4.2 Accuracy and Training Time

We trained two hundred instances of each MLP, ten for each configuration of the 5-fold cross-validation. The generalization error-rate as a function of the training time is given in Figure 4.10. This error rate was computed by comparing the actual target of an instance with the predicted target. In this graph, the curves start when the first epoch is done, showing the time required to train one epoch. It shows that the large network used for the bytes-entropy histograms and GSF with 2-grams takes more than one hour to perform one epoch of training. This makes these three models impractical, especially because even after a week of training, the error rate is much higher than any of the other models. The two most promising models are function and block level 1-grams. These two models go through one epoch in approximately twenty minutes. After a day of training (approx. 10,000 epochs), their error-rates are 13.4% and 11.6% respectively. With the best instance of each model reaching 8.4% and 7.2%, respectively.

Figure 4.11 presents the area under the ROC curve averaging the five best models for each feature. Looking into these metrics, we can evaluate the capabilities of each feature to capture information about the different malware. The figure shows these metrics in two bar-graphs grouped by targets (top) and by features (bottom). For example, the top bar-graph shows that Shifu and Andromeda are easily identified as all models reach an AUC greater than 0.95. Similarly, the bottom graph shows that function level 1-grams, block level statistics and 1-grams, and operation level statistics can accurately classify all eleven malware families.

The high AUC of these models implies that they are actually better at making predictions for each separated class than for the multiclass problem. This is good news for our consensus schema as it means that decision trees, which can leverage the relative predictions, will give better results than the raw MLPs. We used the one, two, or five best MLPs (on the validation set) for each feature, for bytes-entropy histograms and global level assembly (state-of-the-art), for all assembly, and for all features.

For each case, we built five random forests (using the same folds as for the

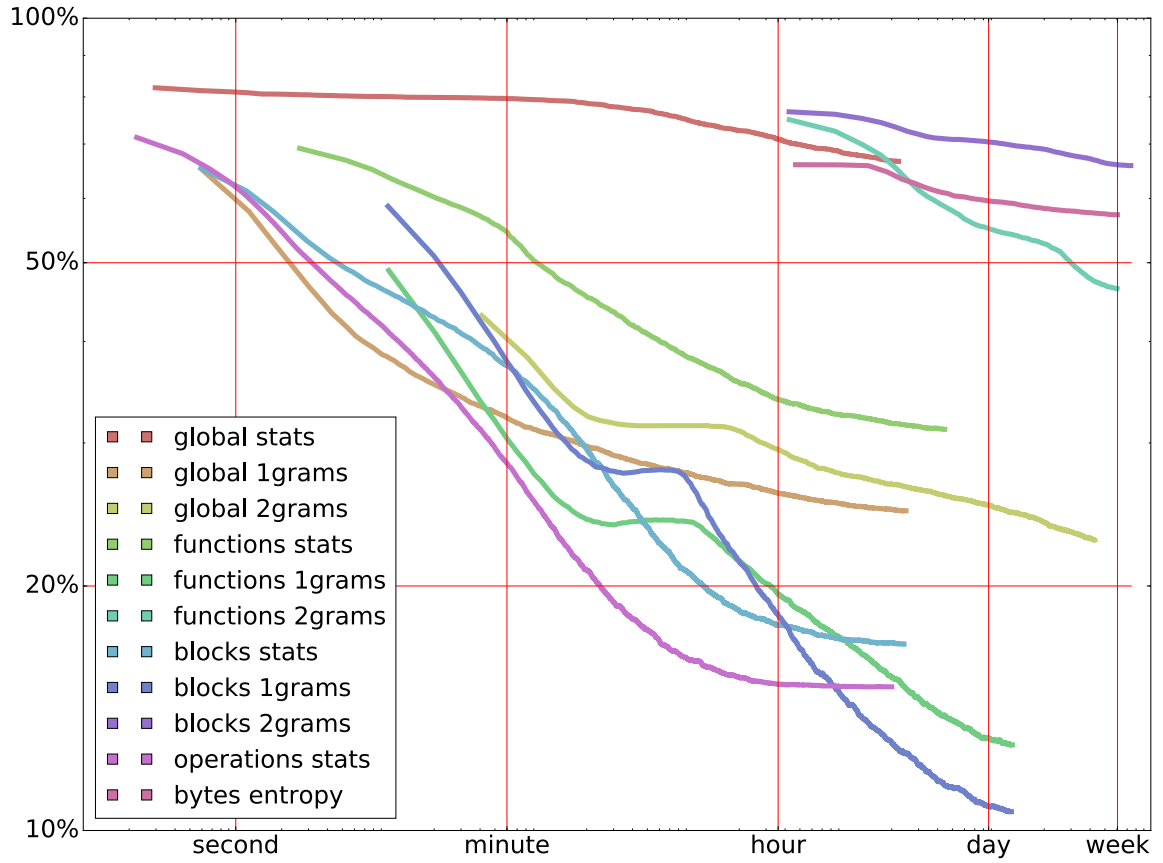
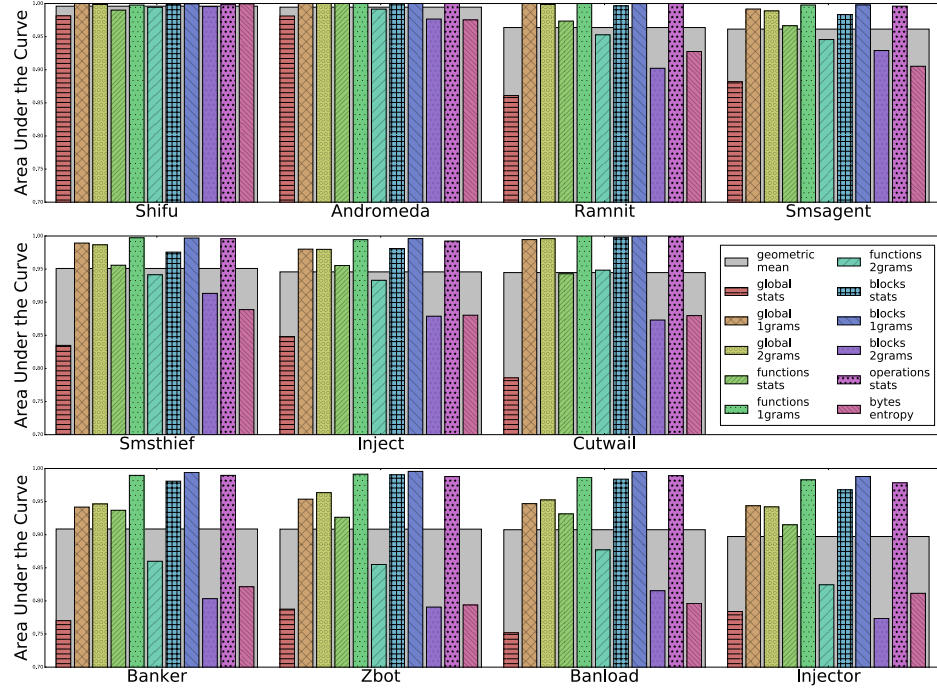
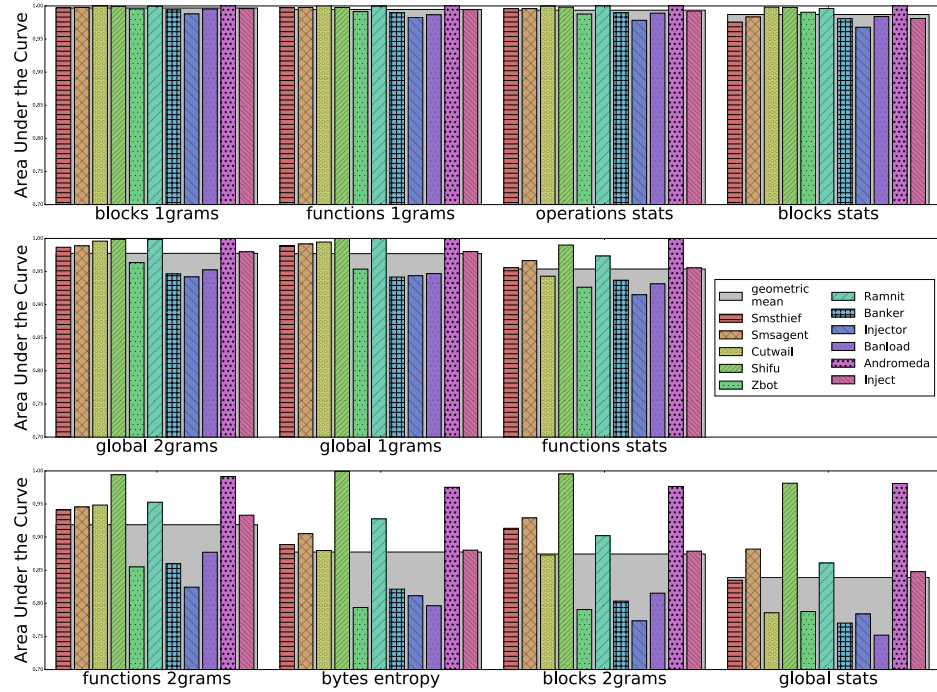


Figure 4.10: Error rates as a function of the training time, averaged over the 200 instances of each model. These error rates are measured on the testing sets of the 5-folds cross-validation. It provides a good approximation of the generalization error. These experiments were conducted for a duration of two weeks using up to one thousand SPOT instances. The lines start at the first epoch, showing the duration of one epoch for each models.



(a) Grouped by targets



(b) Grouped by features

Figure 4.11: Area under the receiver operating characteristics curve (AUC) for all models. Figure 4.11a, the results are grouped by targets to compare the ability of each feature to characterize malware from each families. Figure 4.11b, they are grouped by features to compare the quality of each feature over the whole target space. In both figures, targets and features are sorted based on the geometric means (left-right and top-bottom).

Models		1 best	2 best	5 best
bytes-entropy histogram		39.2%	32.8%	27.3%
Global	statistics	60.5%	53.2%	44.4%
	1-grams	22.9%	20.8%	19.1%
	2-grams	20.7%	19.2%	18.5%
Function	statistics	26.5%	23.8%	20.3%
	1-grams	12.3%	12.0%	10.8%
	2-grams	30.5%	27.2%	25.7%
Block	statistics	15.8%	14.8%	13.6%
	1-grams	10.3%	9.9%	9.9%
	2-grams	39.2%	35.1%	31.4%
Operations	statistic	13.6%	11.3%	10.4%
BEH & Global level		16.0%	15.3%	13.8%
Assembly		8.4%	8.0%	8.0%
All Features		6.9%	6.5%	6.3%

Table 4.5: Comparing the error-rates of random forests built using the predictions of the MLPs. Each row correspond to a different feature set or group of feature sets. The one, two, or five best MLPs are selected for each feature set. The random forests built for a group of feature sets, use the one, two, or five best MLPs for each of these feature sets.

MLPs’ cross-validation). Each random forest is built using ten decision trees. Table 4.5 presents the average error-rates over the five random forests for each feature set and group of best MLPs. These results confirm that function level 1-grams, block level 1-grams, and operation level statistics are the most expressive features (when limiting training to one week). It also shows that:

- bytes-entropy histograms are improved when adding global level assembly features (state-of-the-art)
- assembly features (including GSF) are better than the state-of-the-art
- combination of assembly features and bytes-entropy histograms defines a new state-of-the-art

4.5 Related Work

4.5.1 Malware Characterization

There are two components to current state-of-the-art malware detection: characterization and recognition. First, files are analyzed to extract characteristics. Second, these characteristics are compared to the characteristics of known malware. For signature-based detection, the characterization is done using a hashing technique, e.g. md5, sha256, etc. The recognition phase compares this hash to the hashes of known malware. To improve upon signature-based detection, we need to improve both characterization and recognition.

The first approach is to create hashing techniques that are resistant to noise. These include fuzzy hashes, like context triggered piecewise hashing (CTPH) [Kornblum, 2006] or SSDeep [Chen and Wang, 2008]. Instead of a binary match, these hashes can provide a measure of similarity between files. These permit the detection of malware where small changes have been added to fool classic AVs.

Aside from new hashing techniques, there are various ways to characterize files. When it comes to malware there are two main categories of analysis: static and dynamic. Static analysis only considers the content of the file while dynamic analysis executes the file. Execution can mean running an application (e.g., if the file is an executable) or opening a file with various applications (e.g., if the file is PDF).

We consider two categories of static analyses: bytes-level and code-level. Bytes-level analysis considers the raw bytes in the files. They include: ASCII strings extraction [Schultz et al., 2001], bytes N-grams [Kolter and Maloof, 2004] [Li et al., 2005], and statistical measures (i.e. entropy) [Weber et al., 2002]. One of the most comprehensive bytes analysis is the extraction of the bytes-entropy histogram of a file [Saxe and Berlin, 2015]. This analysis identifies the amount of information associated with various bytes distributions providing a highly representative image of the file. Code-level analyses originates from the domain of reverse-engineering. It considers the executable portion of the files, x86 instructions, byte-code instructions, or scripting code. Tools

like IDApro [Eagle, 2008], Radare2 [Radare2, 2008], and ROSE Compiler [ROSE, 2017] can extract code from various type of files.

Dynamic analysis is a powerful tool for malware detection. It is usually performed inside a sand-boxing environment, like Cuckoo [Guarnieri et al., 2012]. Dynamic analysis can recognize malicious behaviors, like privilege escalation or access to command and control channels. However, dynamic analysis can be time consuming and difficult to implement.

While signature and fuzzy hashes are simple to compare, complex characterizations are much more difficult. One solution is to aggregate the results from various analyses inside a report and compute a fuzzy hash of this report. A similar method is used for Reversing Labs Hashing Algorithm [ReversingLabs, 2015]. This technique enables the detection of variants of malware that were never seen before, but it still relies on comparing the generated hashes to the hashes of known malware. Instead, ML techniques are designed to extract knowledge from large amounts of data. If trained on meaningful characterizations of the files, ML techniques are much more resilient to noise than any technique relying on hashes. Deep learning has been used to train models on ASCII strings and bytes-entropy histograms [Saxe and Berlin, 2015]. To the best of our knowledge, this work represents the state-of-the-art in detection of malware based solely on static characterizations. Malware can also be “clustered” based on the graph edit distance between their respective call graphs [Kinable, 2010]. Clustering is an unsupervised learning technique, so the authors expect it to be able to detect emergent malware. Finally, it is possible to detect malicious android applications based on a combination of static and dynamic analyses [Xu et al., 2016]. This work presents the most complex model that we have seen for malware detection. This model combines a reduced boltzman machines (RBM) with support vector machine to make sense of a variety of features provided by the analyses.

4.5.2 Feature Graphs

SPGK [Borgwardt and Kriegel, 2005] and other kernel techniques have been applied to multiple compiler problems. SPGK was used to train SVM models for application performance predictions [Park et al., 2012]. In this work, applications were characterized by their control flow graphs. Graph-based features were also used to order LLVM’s optimization passes [Nobre et al., 2016].

Deep learning is an ML technique that cannot take advantage of kernels. To apply deep learning to compiler graphs, we need a fixed size feature vector. The first technique is DeepWalk [Perozzi et al., 2014]. This algorithm aggregate the results of multiple random walks into one large feature vector. DeepWalk was used to analyze social media datasets, improving the accuracy of the generated models, while reducing the time needed for training.

Spectral analysis of the graph’s Laplacian was used to classify hand-written digits [Bruna et al., 2013]. This paper introduces the mathematical tools that we leverage in our work. This technique was further extended [Henaff et al., 2015] with the introduction of a DNN architecture that was similar to convolutional networks. a review of graph spectral analysis for deep learning was conducted by [Bronstein et al., 2016].

4.6 Conclusion

In this chapter, we used multiple layers perceptrons (MLP) to classify malware. With MLP, the number of parameters that need to be optimized grows rapidly with the number of features. Because of this issue, we limited each model to a subset of the features in order to keep our computation practical. We used ensemble techniques to aggregate the predictions of multiple MLP together. While these ensemble models help to improve the overall accuracy, it is still computationally impractical for us to train MLPs on the entire set of feature set. In the next chapter, we introduce advanced techniques that allow us to scale our training of neural networks on our entire feature set. We leverage these techniques on an extended dataset with more features and

targets. The resulting models are single neural networks that can leverage information across all the feature sets.

Chapter 5

ADVANCED NEURAL NETWORKS FOR MALWARE CLASSIFICATION

5.1 Introduction

Our work on malware classification relies on a large number of characterizations of the files under scrutiny. Each of these characterizations produces different feature sets. These feature sets have various topologies: some feature sets are collections of loosely related values, other feature sets are one and two dimensional histograms, and finally, we explore other feature sets consisting of graphs with a variety of features on each node. If one wants to take advantage of all the feature sets in a single neural network (NN), it becomes impractical to use a multiple layer perceptron (MLP) as we did in Chapter 4. Indeed, the first layer of this MLP would be prohibitively large. Instead, each feature set is fed to its own neural network. The outputs of these neural network are then fed into another network which makes predictions. The additional advantage of this differentiation is that we can specialize the NNs depending on the topology of each feature set.

In this chapter, we discuss the construction of complex neural networks operating on many different feature sets simultaneously. Section 5.2 describes the feature engineering part of our work. Seven feature sets, with three distinct topologies, are described. These features are augmented using various transformations. Finally, we summarize the seven augmented feature sets and compare them with the state-of-the-art. In Section 5.3, we describe the dataset that we constructed to evaluate this work. We describe and motivate our sample selection criteria and the composition of the resulting dataset. We also introduce a sampling method used to deal with imbalanced datasets. Section 5.4 examines the neural network engineering phase of our work. First,

we discuss the training procedure for these neural networks. Second, we propose network architectures to take maximum advantage of our different feature sets. Third, we discuss how we deal with the main issues that arise when working with graph spectral features. We finish with the results in Section 5.5, and the related work in Section 5.6.

5.2 Feature Sets

Feature engineering is an essential step to take advantage of any machine learning algorithms. It is a step that happens between the sample’s characterization and its ingestion into the algorithm. In Chapter 4, we only considered the first (and unavoidable) phase of feature engineering, i.e. making a characterization’s results “digestible” by neural networks. In the case of neural networks, it means constructing a fixed size representation of the characterization. The second phase of feature engineering deals with data transformations. This phase is usually handled by a domain scientist who can decide how best to transform the data. For example, a count might be better represented as a logarithm, i.e. an order of magnitude while histograms can be normalized to represent the associated frequencies.

In this section, we describe seven feature sets obtained when characterizing our samples. These characterizations are performed by our scalable file characterization platform deployed on Amazon’s cloud services. By being highly configurable, our platform permits us to apply any of a variety of characterizations to our malware dataset.

5.2.1 Three Types of Features

Our feature sets can be separated into three groups: basic, bytes, and assembly. The basic group contains the most common malware characterization techniques that are typically used by cybersecurity analysts. Aside from a variety of hashing techniques, it extracts the ASCII strings, and the metadata and import tables from portable executable header of Windows executable files. The bytes group focuses on the bytes level representation of the file. Particularly, it extracts the file’s bytes-entropy histogram

(BEH). The assembly group relates to characterization extracted from the result of the binary’s reverse engineering. In this chapter, we focus on the graph spectral features (GSF) built from the function, block, and operation graphs.

5.2.1.1 Hashes Histograms

Hashes histograms, as presented by Saxe and Berlin, are fixed size representations of list of strings [Saxe and Berlin, 2015]. These are feature vectors constructed from lists of strings, namely the ASCII strings in the file, and metadata and import table of the portable executable (PE) headers. The ASCII strings are obtained using the GNU tool *strings*. The PE header is part of Windows executable files and provides informations needed to load and execute it. The python module *pefile* is used to dump the metadata and import tables from the PE header.

Figure 5.1 shows how a list of strings is transformed into a fixed-size feature vector. In the case of metadata and import tables, each row (comma-separated) is hashed separately.

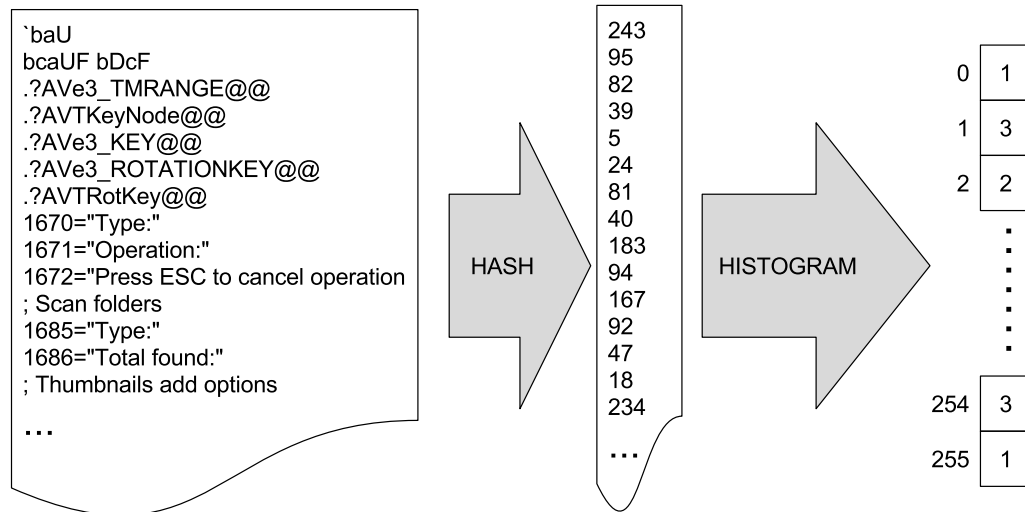


Figure 5.1: Given a list of strings, either the ASCII strings or dumped import and metadata from the PE header, we build a fixed-size feature vector. Here, an arbitrary number of ASCII strings of arbitrary sizes is transformed into a feature vector of length 256. First, each string is “hashed” to an integer between zero and the desired vector length. Second, a histogram of these hashes is produced by counting the occurrences of each value. The result is a vector of positive integers of the desired size.

5.2.1.2 Bytes-Entropy Histograms

We presented bytes-entropy histograms (BEH) in Chapter 4 (see Algorithm 4.1 and Figure 4.2). In this chapter, we only consider 16 entropy bins (En) instead of 256. This reduces the size of the BEH feature vector by a factor of sixteen. The resulting BEH feature vector can be used efficiently even with fully-connected neural networks.

5.2.1.3 Spectral Features of Assembly Graphs

We presented graph spectral features in Chapter 4. For this characterization, the reverse-engineering tool Radare2 [Radare2, 2008] extracts assembly code from the file. This assembly code is then transformed into graphs at three levels of granularity: functions, blocks, and operations. Finally, we use graph spectral analysis to extract a fixed-size representation of the graph. In this chapter, we decrease the number of extracted spectral bands from twenty to sixteen. We also do not consider the instruction n-grams larger than 1, which are too large to be practical for training.

5.2.2 Transformations

The second phase of feature engineering is to apply various transformations to the feature sets. These transformations allow us to extract more information. For example, let us consider the strings histogram. It is a vector of 256 positive integers, which add up to the number of ASCII strings in the file. Hence, the range of values depends on this number. The first possible transformation for such a vector is to normalize it. The normalization transformation enables the neural network to compare frequencies instead of comparing counts. However, the actual numbers could be important features, but small values get greatly diminished when the whole dataset is normalized (zero-centered and bounded between -1 and 1). To prevent this, we also present the neural network with logarithms of elements of the vector.

For any feature set, we can apply one the three following transformations:

- **ID**: identity, no preprocessing
- **log**: compute the logarithm of each element (actually $x \mapsto \log(x + 1)$, requires positive values)

- **norm**: normalize the tensor (can provide one or more axis to enable row, column, or global normalization)

5.2.3 Feature Sets Summary

In Table 5.1, we summarize the feature sets considered in our experiments. After preprocessing, some feature sets are aggregated as they are part of the same group of feature sets. That is the case with the three graph spectral feature (GSF) sets. Each of them aggregates the eigenvalue, statistics, and the instruction 1-grams of each node. The tensors produced for the GSF feature sets have an additional dimension of size two, which corresponds to the real and imaginary parts of the complex numbers.

		Shape IN	Transformations	Shape OUT
	Strings	[256]	ID, log, norm	[3 , 256]
	Metadata	[256]	ID, log, norm	[3 , 256]
	Import	[256]	ID, log, norm	[3 , 256]
	Bytes-Entropy Histogram	[16 , 256]	ID, log, norm row norm columns, norm global	[5 , 16 , 256]
Function GSF	Eigenvalues	[16]	ID	[16 , 220 , 2]
	Statistics	[16 , 19]	ID, log, norm	
	1-grams	[16 , 54]	ID, log, norm	
Blocks GSF	Eigenvalues	[16]	ID	[16 , 187 , 2]
	Statistics	[16 , 8]	ID, log, norm	
	1-grams	[16 , 54]	ID, log, norm	
Operations GSF	Eigenvalues	[16]	ID	[16 , 166 , 2]
	Statistics	[16 , 1]	ID, log, norm	
	1-grams	[16 , 54]	ID, log, norm	

Table 5.1: Preprocessed feature sets are usually combined into a higher dimensional tensor (BEH have 5 preprocessing yielding an additional dimension of size 5).

In addition to these seven feature sets, we reproduce the feature sets used by Saxe and Berlin, in their state-of-the-art deep learning work for malware detection [Saxe and Berlin, 2015]. This neural network uses the hashes histograms (strings, metadata, import) and bytes-entropy histogram without any transformation. Given that the feature sets have different topologies, we flatten and concatenate them together in a 1D tensor of size 1792. We refer to this feature set as “Invincea”, the company that developed this state-of-the-art technique.

5.3 Dataset

For our experiments, we have access to a large database of files consisting of more than eight million fully analyzed malware and more than thirty thousand goodware. However, we filtered this dataset to make it relevant for our analyses. Specifically, we focused on files that yield meaningful features for all feature sets.

In this section, we first describe the constraints that were used to select the relevant samples. Then, we describe the resulting dataset and its particularities. Finally, we introduce the sampling technique to deal with imbalance of the dataset.

5.3.1 Relevant Files From Our Database

To compare the characterization capacities of the seven feature sets that we use, we selected files that are relevant to all of them. The first constraint comes from the metadata and import feature sets as they only exist if the file is a Windows executable. The second constraint is that none of the list of ASCII strings, metadata, or import should be empty. The third, and most restrictive, constraint is related to the complexity of the assembly code graphs (either functions, blocks, or operations). We require these graphs to have more than eight nodes and at least half as many edges as they have nodes. The lower limit on the number of nodes prevents the resulting GSF from being mostly zeros, since a large number of zeros prevents the unsupervised pretraining from converging. Similarly, graphs with very few edges compared to the number of nodes tends to produce “noisy” GSFs which prevent convergence. Finally, we limit the number of nodes in the graphs to ten thousand, since extracting the eigenvalues and eigenvectors of the laplacian of larger graphs is prohibitively expensive.¹

5.3.2 Composition of the Dataset

When these constraints are applied to the eight million files in our database, only 25,559 match these criteria. These files are from eighteen different families, including

¹ We investigated using an eigenvector extractor for sparse matrices. A large number of graphs cause this code to break, making it unusable.

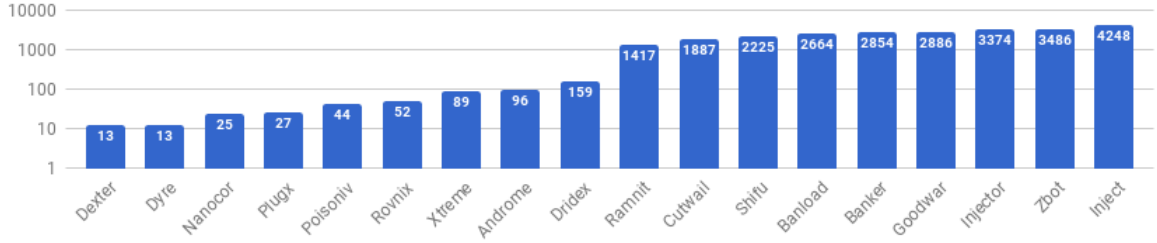


Figure 5.2: Number of files from each of the eighteen families in the dataset using a logarithmic scale. All these samples match all the constraints describe in Section 5.3.1.

both malware and goodware. Figure 5.2 shows the number of samples for each family.

This figure shows the difference in the number of files among the different families. About half of the families have less than a hundred files with some families having as little as thirteen samples. While a little over half the families have a few thousand files with as many forty-two hundred files in the largest family.

5.3.3 Streaming of Samples

Imbalanced datasets are problematic for most machine learning techniques and require correct handling to ensure valid results. When possible, the best solution to deal with this problem is to use one or more data augmentation techniques. For example, in the domain of image processing, one can use cropping, rotation, or color shifting to create additional samples. Data augmentation of executable files require tools used by bad actors to generate many variants of malware. However, such work is out of the scope of this dissertation. Instead, we use a combination of up-sampling and down-sampling of the dataset. This is done stochastically during the training of the neural networks.

This method is implemented alongside the streaming of the dataset into the learning algorithm. Our neural networks are trained using stochastic gradient descent with **mini-batches**. This training algorithm updates the model based on the gradient of its error for randomly chosen (small) subset of the training set, called mini-batches. In this method, one epoch of training corresponds to presenting the whole training set to the model in as many mini-batches as needed.

There is a couple differences in terms of how we implement mini-batches compared to traditional approaches. Instead of creating mini-batches from the whole training set, we create one mini-batch for each family in the dataset. These mini-batches are then merged and shuffled before being split again and presented sequentially to the model. Also, instead of streaming the whole training set for each epoch, we only stream the requested number of mini-batches.

While the primary goal of our streaming method is to produce balanced mini-batches, streaming a fixed amount of mini-batches for each epoch has other advantages. In the conventional case, doubling the size of the dataset doubles the size of the training set, making epochs twice as long. That has two effects: 1) the model converges faster when considering the number of epochs, and 2) checkpointing is less frequent. The first effect is an issue when tuning the learning rates, especially their decay, which is usually a function of the number of epoch. The second effect increases the cost of failure in a large scale distributed training system.

5.4 Neural Network Engineering

Once our feature sets are engineered, we created neural network architectures adapted for each of them. In this section, we describe how our deep neural networks are constructed. First, we discuss the training procedure for these neural networks. Second, we propose network architectures to take maximum advantage of the different features set.

5.4.1 Training Procedure

Our neural networks have two stages. The first stage constructs high level features (HLF) for each feature set. The second stage is a multiple layer perceptron (MLP) that uses one or more high level features to perform the classification.

5.4.1.1 High Level Features

The first stage is to extract HLF from each feature set, we construct neural network called stacked auto-encoders (SAE). For each feature set except Invincea, we

trained a SAE generating a HLF of size 8×8 . For the Invincea feature set, the SAE yields a HLF of size 1024, see Section 5.4.2.3.

Auto-encoding [Hinton and Salakhutdinov, 2006] is an unsupervised method to train neural networks. For a single layer auto-encoder, there are three groups of parameters: the weights (W) and the encoding and the decoding biases (b_e/b_d). The idea behind an auto-encoder is to find the values of W , b_e , and b_d that minimize $X - \sigma(\sigma(X \times W + b_e) \times W^T + b_d)$, where σ is the activation function of the layer. Here, $\sigma(X \times W + b_e)$ is called the “code” while $\sigma(\sigma(X \times W + b_e) \times W^T + b_d)$ is the “reconstruction”. Intuitively, the input is fed forward in the layer to be “encoded” and the resulting “code” is fed backward through the layer to be “decoded”.

- **encode:** the input is fed forward through the layer, the resulting output is called “code”
- **decode:** the “code” is fed backward through the layer, the resulting output is called “reconstruction”
- **training:** minimize the difference between the original input and the “reconstruction” output

Stacked auto-encoders [Vincent et al., 2010] are constructed by stacking multiple layers sequentially. Each layer increases the level of abstraction of the resulting code. The main advantage of stacked auto-encoders is that they permit the use of layer-wise pre-training [Bengio et al., 2007]. This means that the layers are trained sequentially. The first layer is trained, then the second layer, and then the next layer, etc. This method helps alleviate the vanishing gradient problem. The vanishing gradient problem refers to the fact that the gradient of the error tends to get smaller as we move backward through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in later layers.

5.4.1.2 Classifiers

The second stage is an MLP with one hidden layer of size 64 and an output layer the same size as the number of classification targets in the dataset. We build

many of these classifiers, using either a single HLF or different combinations of HLFs. When the classifier stage of the network is being trained, the layers producing the HLF are being fine-tuned using a very small learning rate.

5.4.2 Neural Network Architectures

We have eight feature sets: seven transformed feature sets and the state-of-the-art Invincea feature set. In this section, we define the architectures of the stacked auto-encoders (SAE) used to build the high level features (HLF). We define three of these convolutional architectures: one for hashes histograms, one for the bytes-entropy histogram, and one for graph spectral features. Each of these networks consists of four layers. The shape of the input layer depends on the topology of the feature sets, but the next three layers are similar. We also define fully-connected versions of the network architecture used to evaluate against our convolutional architectures.

5.4.2.1 Input Convolutional Layers

Encoding different feature sets requires us to construct input layers that are adapted to the topologies of these feature sets. In the following paragraphs, we describe these input layers.

5.4.2.1.1 Hashes Histograms

The features in this section correspond to three hashes histograms: strings, metadata, and import. The preprocessing stage adds the logarithm and frequencies of each of the 256 counts in a histogram, yielding a 2D tensor of 3×256 . A convolutional layer for our “augmented” hashes histograms maps each of the 256 triplets (count, logarithm, and frequency) to the desired number of channels. The number of channels in a convolutional network is the number of outputs of the convolutional filter. Figure 5.3 depicts such architecture with two channels.²

² To simplify our diagrams, the number of channels is smaller than what we used in our experiments.

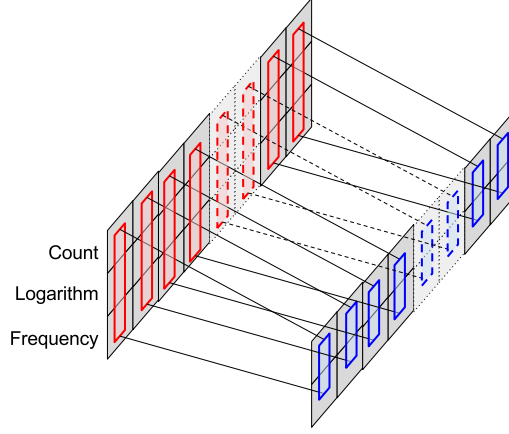


Figure 5.3: A hashes histogram (top-left) is processed by a convolutional layer. For each triplet (count, logarithm, and frequency), two values are computed.

5.4.2.1.2 Bytes-Entropy Histograms

After preprocessing the bytes-entropy histogram is a 3D tensor of dimensions $5 \times 16 \times 256$. There are three possible convolutional architectures for it as the tensor can be sliced over the bytes axis, the entropy axis, or both. The architecture that slices the tensor across both bytes and entropy axis does not reduce its dimensionality sufficiently to be useful on its own. We evaluated both of the other architectures, however, when slicing the tensor across the bytes axis the back-propagation of the error is extremely time consuming. In addition, models using this architecture are inferior to models where the entropy axis is sliced. In Figure 5.4, the tensor is sliced along the entropy axis. The filter with eight channels is applied to each slice, taking as input 5×256 values. The result is a 2D tensor of dimension 16×8 .

5.4.2.1.3 Graphs Spectral Features

Graphs are complex data-structures that are unbounded in size. In Section 3.5, we presented how we can leverage graph spectral analysis to obtain a fixed size representation of a graph. Each GSF is characterized by an eigenvalue and a collection of projected node features. The network architecture presented here leverages the structure of graph spectral features (GSFs). In Figure 5.5, we present the convolutional architecture used to learn from GSFs.

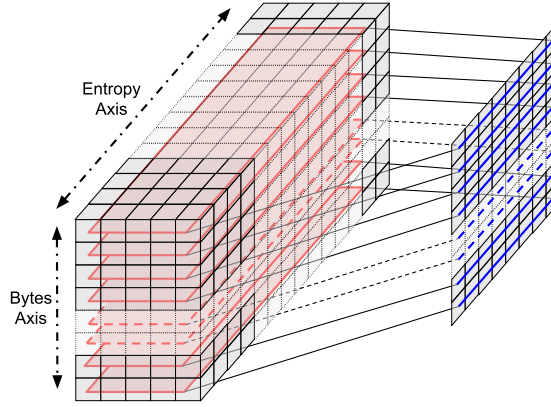


Figure 5.4: The tensor, of dimension $5 \times 16 \times 256$, is sliced along the entropy axis (red planes). The convolution filter with eight channels is applied to each slice of dimension 5×256 . It yields a tensor of dimension 16×8 .

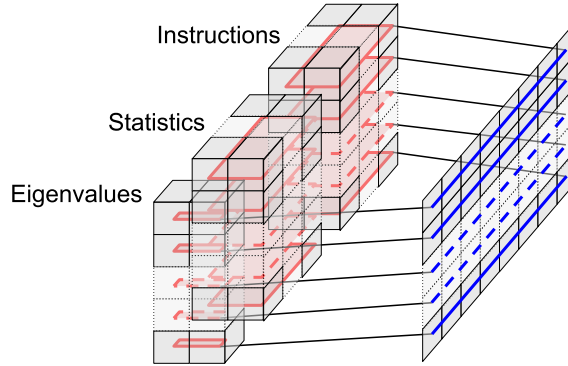


Figure 5.5: Each graph spectral feature is made of one eigenvalue and multiple projected feature vectors. The graphs produced by our analyses have two feature vectors per node, a statistics and an instruction histogram. All features of each node is presented to the same filter with eight channels. For simplicity, we do not represent the three version of each feature vector that were generated during the preprocessing stage.

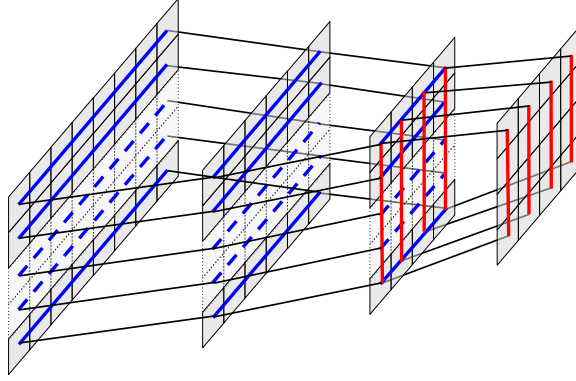


Figure 5.6: The input to a hidden layer is the output of the previous layer specific to each feature set. These input layers produce a collection of independent rows (blue lines). The first and second hidden layers reduce the size of each row to four elements. The last layer encodes each of the resulting four columns (red lines) into four elements. The result is a tensor of 4×4 elements.

5.4.2.2 Hidden Convolutional Layers

The input layers generate two dimensional tensors where the first dimension corresponds to its convolution axis. The next two layers apply their convolutions across the same axis, bringing the size of the second dimension to eight. Finally, the fourth layer is applied along the second dimension to bring the size of the first dimension to eight. The results are eight by eight tensors that provide a high-level representation of the feature sets. In Figure 5.6, we depict an analogous architecture that generates a tensor of four by four elements.

5.4.2.3 Reference Architectures

We compare the accuracy of models built using the augmented feature sets and convolutional architectures. To this end, we reproduced the state-of-the-art model from Invincea and trained a fully-connected SAE for each of the seven transformed feature sets.

The Invincea SAE is made of two layers of size 1024 [Saxe and Berlin, 2015]. The classification stage of the model adds another layer of size 64 before the final output layer. Given that we are solving a much larger classification problem, this additional layer must be involved to obtain good results.

For each of the seven transformed features, the fully-connected SAE is made of three layers: two layers of size 1024 and a third layer of size 64. This third layer makes the HLF produced by both the convolutional and fully-connected SAE have the same size.

5.5 Results

In this section, we compare results from our different feature sets and neural network architectures.

5.5.1 Experimental Setup

All experiments have been conducted using five-fold nested cross-validation, meaning that three folds are used for training, one is used validation, and the remaining fold is used for testing. This results in twenty cross-validation experiments.

The models were trained using the following schedule:

- 100 epochs for each layer of the auto-encoding stage
- 200 epochs for the classification stage

Each epoch consists of two hundreds and fifty-six mini-batches of sixteen samples.

In each cross-validation fold, we trained one encoder per architecture, resulting in fifteen HLF described in Table 5.2. The resulting HLFs are used to train classifiers for each feature sets and group of feature sets (see Table 5.3). We trained five instances of each classifier in each cross-validation fold. For each of these folds, we selected the best instance based on the validation error. The results in Section 5.5.2 are the average across cross-validation experiments. For each cross-validation experiment, the results being reported were evaluated on the testing folds, which was never seen by the model during training.

HLF Names	Feature Set	Architecture	Layer Sizes
hlf-strings-fc	Strings Hashes Vector	Fully-Connected	1024 - 1024 - 64
hlf-strings-cnn	Strings Hashes Vector	Convolutional	64 - 16 - 8 - 8
hlf-metadata-fc	Metadata Hashes Vector	Fully-Connected	1024 - 1024 - 64
hlf-metadata-cnn	Metadata Hashes Vector	Convolutional	64 - 16 - 8 - 8
hlf-import-fc	Import Hashes Vector	Fully-Connected	1024 - 1024 - 64
hlf-import-cnn	Import Hashes Vector	Convolutional	64 - 16 - 8 - 8
hlf-beh-fc	Byte/Entropy Histogram	Fully-Connected	1024 - 1024 - 64
hlf-beh-cnn	Byte/Entropy Histogram	Convolutional	256 - 64 - 8 - 8
hlf-functions-fc	Functions Graph (CG)	Fully-Connected	1024 - 1024 - 64
hlf-functions-cnn	Functions Graph (CG)	Convolutional	256 - 64 - 8 - 8
hlf-blocks-fc	Blocks Graph (CFG)	Fully-Connected	1024 - 1024 - 64
hlf-blocks-cnn	Blocks Graph (CFG)	Convolutional	256 - 64 - 8 - 8
hlf-operations-fc	Operations Graph	Fully-Connected	1024 - 1024 - 64
hlf-operations-cnn	Operations Graph	Convolutional	256 - 64 - 8 - 8
hlf-invincea	Invincea	Fully-Connected	1024 - 1024

Table 5.2: This table lists the fifteen auto-encoder architectures that we developed. For each of them, we specify the feature set, kind of architecture, and layers size.

Groups of Feature Sets	Feature Sets
VECT	Strings, Metadata, Import
BEH	Bytes/Entropy Histograms
GSF	Functions, Blocks, Operations
SOA	VECT, BEH
ALL	VECT, BEH, GSF

Table 5.3: Groups of feature sets corresponding to: hashes histograms (VECT), bytes-entropy histogram (BEH - only one feature set), graph spectral features (GSF), state-of-the-art features (SOA - transformed version of the Invincea feature set), and all feature sets (ALL)

5.5.2 Accuracy Results

We present our results in Table 5.4 and Figure 5.7. On our dataset, the state-of-the-art model reaches 62.3% accuracy while our model using all the transformed features and convolutional architectures reaches 83.6%.

The main source of improvement between our approach and the state-of-the-art is the introduction of transformations. Indeed, the augmented bytes-entropy histograms and hashes histogram of the metadata table of PE headers yield accuracies of 74.8% and 71.5%, respectively, when used by themselves. Using the augmented versions of the state-of-the-art feature sets yields 79.3% accuracy compared to 62.3% accuracy of the unaltered state-of-the-art approach.

Our results also show that convolutional neural networks are able to extract more (relevant) information from the feature sets. This is especially true for the graph spectral features where the convolutional approach is $1.26\times$ more accurate than a fully-connected approach. In addition to this increased accuracy, convolutional architectures have far fewer parameters than fully-connected architectures (see Figure 5.8). Using all feature sets, the convolutional approach outperforms the fully-connected version by 3.9%, and it achieves this by using $42\times$ less parameters. In terms of storage, the resulting fully-connected model uses approximately 200 MB, while the convolutional model requires only 4.5 MB.

5.5.3 Computational Performances

This section examines the training and testing cost. Figure 5.9 shows the time spent on training (top) and testing (bottom) for one epoch of the classifier. We focus on the final classifier layers as that is, by far, where most of the time is spent. These times are shown in seconds using a logarithmic scale.

These graphs show that convolutional neural networks (CNN) are much slower to train than their fully-connected counterparts. However, this is not inherent to CNNs but is due to our implementation using Theano. Indeed, we do not use Theano optimized 2D convolution layers for 2D or 3D tensors (images). Instead, we implemented

Feature Set(s)	Fully-Connected	Convolutional
Invincea	62.3%	
Strings	53.9%	58.1%
Metadata	69.6%	71.5%
Import	53.4%	58.3%
VECT	73.4%	75.2%
BEH	71.6%	74.8%
Functions	15.6%	18.7%
Blocks	18.2%	29.1%
Operations	34.3%	40.3%
GSF	38.5%	48.7%
SOA	76.3%	79.3%
ALL	79.7%	83.6%

Table 5.4: Average test accuracy of the best model in each fold

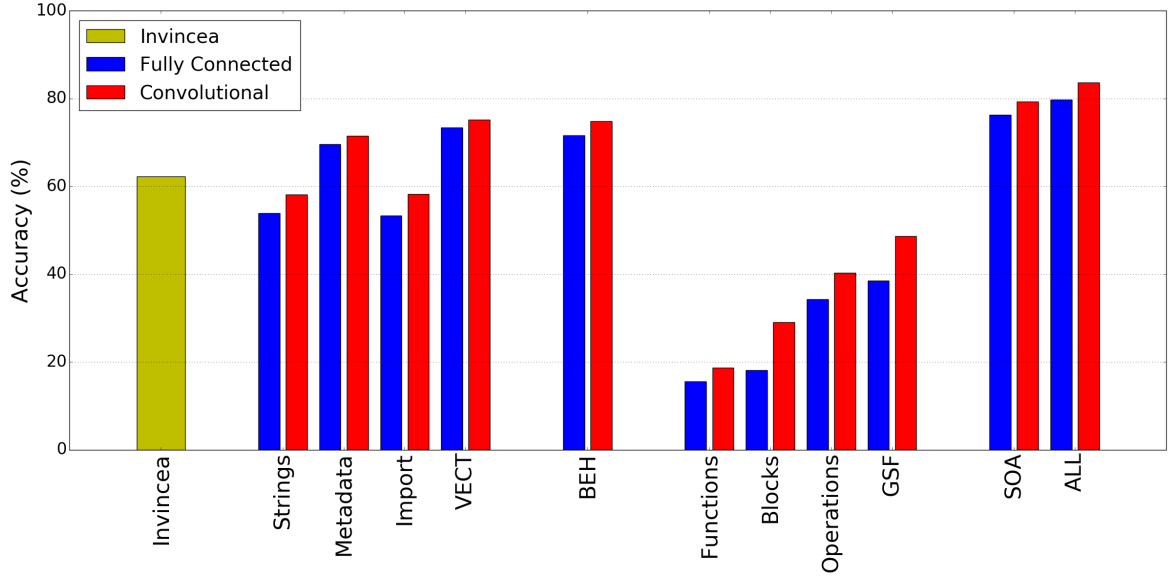


Figure 5.7: We evaluated the accuracy for the different combination of feature sets and for different architectures. We used five-fold cross validation. In each fold, five instance of each classifier was trained and the best was selected.

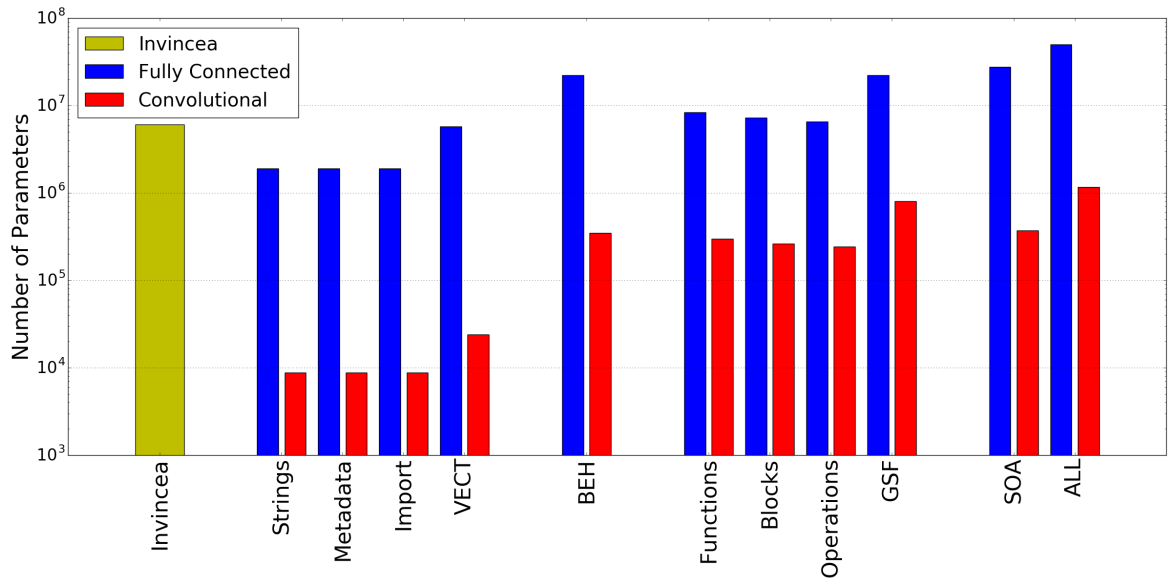


Figure 5.8: This graph shows the number of parameters in the different models. The vertical axis uses a logarithmic scale. The difference between convolutional and fully connected architecture is more than two orders of magnitude for the hashes histograms.

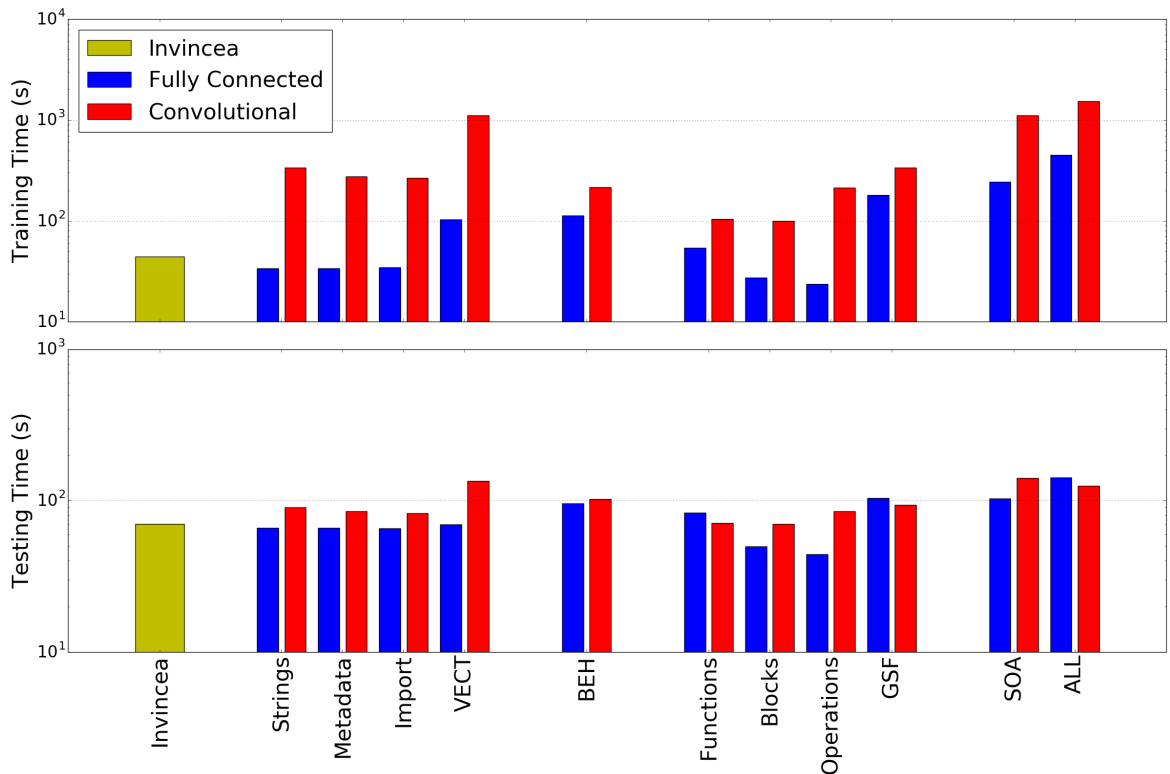


Figure 5.9: Time spent (in second) on training (top) and testing (bottom) for each model. Training dominates testing as it requires performing backward propagation.

a convolution layer for any number of convolution axis on tensor of arbitrary dimension. Our implementation uses nested Theano map operators, one for each convolution axis. However, Theano is unable to optimize the resulting computation graph when multiples of these layers are stacked.

5.6 Related Work

In this chapter, we used dataset augmentation techniques to extract additional from the feature sets (Section 5.2.2) and balance the dataset (Section 5.3.3). We also created novel neural network architectures by mixing convolutional and auto-encoding techniques to build high level features (HLF) (Section 5.4). In this section, we review the related work on these two topics.

5.6.1 Dataset Augmentation

The first step we took to augment our dataset was to extend the feature space through the application of various domain-specific transformations. This topic has been explored for many years by the data mining community. The book “Feature Extraction, Construction and Selection: A Data Mining Perspective” [Liu and Motoda, 1998] provides a complete overview of the various transformations that can be used on feature sets. Our work only covers a limited scope of transformations compared to what is available due to the large associated search space, which is highly domain specific. However, automatic and guided methods have been explored to help data mining and machine learning practitioners. While most technique are domain specific and focused on image, text, or speech analysis, some techniques are more general. For example, Vafaie and Jong used genetic search to discover the best set of transformations [Vafaie and Jong, 1998]. Other works focus on variable selection that reduces the size of the feature sets [Fan and Lv, 2008]. Srivastava et al, used rule induction methods to discover combinations of features specific to each class in the dataset [Srivastava et al., 2007]. Finally, Seide et al, used a complex combination of neural networks to automate feature engineering in speech recognition [Seide et al., 2011]. This work

shows that feature space transformation can be handled by a sufficiently complex neural network.

The second step we took to augment our dataset dealt with the imbalance of our initial data distribution. Imbalanced datasets are a well documented problem in machine learning. However, the techniques to deal with it are either simple or highly domain specific. In our work, we used simple techniques, namely up-sampling and down-sampling. These re-sampling techniques are reviewed by Xinai [Xinai, 2016]. In specific domains, techniques can be developed to generate new samples from existing samples. For example, image datasets can be augmented using a variety of simple image transformations such as cropping, rotation, and color shifting [Arandjelovic and Zisserman, 2012]. Recent work on malware detection [Anderson et al., 2017] have looked into using techniques used by bad actors to transform malware and evade malware detection models. This work has been formulated as an adversarial machine learning problem, but the same techniques could also be used to extend our dataset.

5.6.2 Auto-encoder and Convolutional Architectures

Auto-encoding techniques [Vincent et al., 2010] associated with layer-wise pre-training [Bengio et al., 2007] have been instrumental in enabling the transition from shallow neural networks to deep neural networks. These techniques help prevent large neural networks from converging on bad solutions while also reducing the computation needed to converge.

Convolutional neural networks (CNN) have also been a key technique in the success of deep learning [Krizhevsky et al., 2012]. By leveraging regularity and locality in the features, CNNs reduce by many orders of magnitude the number of parameters in the network. Reducing the number of parameters not only reduces the amount of computation of each learning iteration, but it also simplifies the search space. The reduced search space decreases the risk of a model converging on a local minimum. Most work on convolutional networks is focused on image recognition, where it is

particularly efficient. However, convolutional networks have been used for sentence classification [Kim, 2014][Zhang and Wallace, 2015].

In our work, we use a combination of auto-encoding and convolutional techniques to build high level features. This was motivated by recent work [Sharif Razavian et al., 2014] which introduces the idea of “off-the-shelf” features. The main idea in this paper is that we can use high-level features extracted from images by a convolutional network to build new classifiers. One of our goals with this work is to make it possible to construct “off-the-shelf” features for malware classification.

5.7 Conclusion

This chapter explores the predictive capabilities of different malware characterization techniques when using fully-connected versus convolutional neural networks. We use multiple techniques from the state-of-the-art in deep learning to construct our models. In addition, we incorporated a new class of malware characterization based on compiler graphs. This study was conducted on a dataset of more than 25 thousand Windows executable files, including 17 families of malware and nearly 3,000 goodware.

Compared to the state-of-the-art neural network for malware detection, our models leverage feature transformations and convolutional architectures. The state-of-the-art model yields an accuracy of 62.3%, while our model using the same characterization techniques yields 79.3% accuracy.

Our new characterization technique, the graph spectral features of compiler graphs, performed below the expectations set in Chapter 4. However, when used in combination with other characterization methods, our resulting model achieved an accuracy of 83.6%.

Finally, we show that properly engineered convolutional architectures have better performances than fully-connected architectures. Convolutional architectures have the added benefit of decreasing the number of parameters in the network, facilitating its convergence.

In future work, we will determine when each of the characterization methods should be used. Particularly, we will determine when exactly to use graph-based features that are expensive to extract. Also, we want to explore building models that predict whether or not it is advisable to use particular features. Such models would use inexpensive and efficient features, such as the bytes-entropy histogram, to predict when to use more expensive features.

Chapter 6

EXPLORATION AND CHARACTERIZATION OF COMPILER TUNING SPACE

6.1 Introduction

As the high performance computing (HPC) community makes progress toward exaflops computing, hardware in supercomputers has new, less intuitive, execution models. Supercomputer manufacturers often evaluate a wide range of hardware to reach the best performance for their users. The growing number and variety of accelerator-equipped systems in the TOP500 list [[Strohmaier, 2013](#)] illustrates this issue. In June 2016, ninety-five systems on the TOP500 list contained accelerators of five different kinds. These computation accelerators promote the host/kernel paradigm. In this paradigm, some parts of an application are programmed in a general purpose language, like C or C++, to be executed by a general purpose processor. The application may spawn kernels. These kernels are written in a language specific to the targeted architecture. In the case of accelerators like GPGPU (General Purpose Graphic Processing Units), kernels can be programmed using OpenCL (Open Compute Language, an extension of C) [[Munshi, 2008](#)]. In addition, many accelerators are well-suited for the single-program multiple-data model of parallelism [[Darema, 2001](#)]. In SPMD parallelism, the same program is executed by independent processors on different data.

As computing systems become more complex, new programming models need to be provided that can handle the complexity of the machines. It is the role of programming models to define the abstractions needed to exploit the underlying hardware. Once the programming models are defined, languages or APIs are created to implement them. These languages and APIs will eventually have to be compiled down to parallel code. An efficient way to accomplish this is to transcribe the input program into an

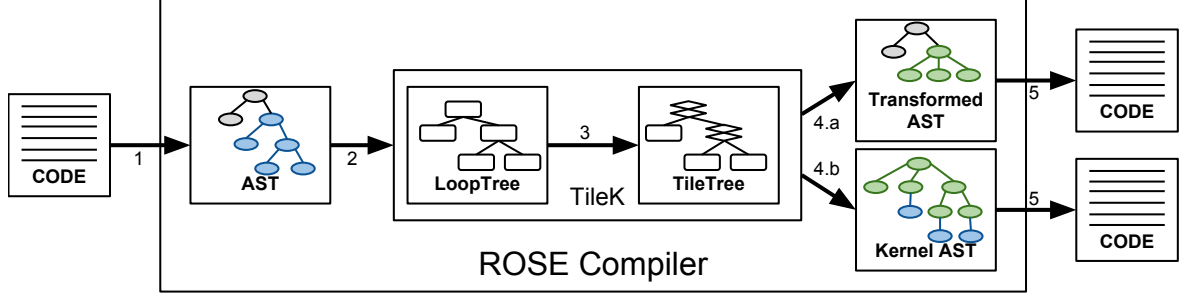


Figure 6.1: TileK’s compilation flow: (1) ROSE parses the input C/C++ code, (2) TileK extracts a representation of the loop nests, (3) TileK applies tiling based on annotations, (4.a) TileK transforms the application’s AST to instantiate the kernel and move data, (4.b) TileK generates the AST of the kernel for the desired target: pThread, CUDA, or OpenCL, (5) ROSE unparses the AST into C/C++/CUDA/OpenCL code.

SPMD problem, and then generate kernels accordingly. This technique often involves two compiler phases: (1) the programming model’s language is compiled to a generic SPMD formulation, and (2) the SPMD formulation is compiled to a kernel specialized to the underlying target architecture.

In this chapter, we focused on the second phase where the difficulty is to find a technique to distribute computations across multiple instances of a kernel. As compute intensive codes are generally composed of loops, we focused our attention on methods to distribute nested loops. One such method is to use tiling [Wolfe, 1989] which subdivides the iteration space of a loop into blocks or tiles. We devised a simple method which decomposes loops into tiles then tiles are mapped to the indexes of the kernel’s instances. This method is generic, and we used it to generate kernels for multicore CPUs (pThread) and for accelerators (OpenCL, CUDA).

In this chapter, we present TileK: a tiling abstraction embedded in C/C++ using directives. This work was recently published [Vanderbruggen et al., 2017]. TileK is a set of directives which enables us to perform tiling and loop interchange on nested loops. The strength of TileK is its ability to generate SPMD Kernels by mapping some of the resulting tiles to different processors. These kernels can be distributed with pThreads or OpenCL. TileK was implemented using ROSE Compiler [ROSE, 2017],

a source-to-source compiler developed at the Lawrence Livermore National Laboratory. First, we present TileK’s programming model. Second, we introduce the tiling method and associated transformations applied to the kernel. Third, we show how this method is used to distribute the computation in SPMD Kernels. Fourth, we conduct an optimization-space exploration where linear-algebra kernels and stencils are tiled and distributed using TileK. The various versions of these codes are evaluated for many different input sizes. We show that the selection of a given optimized version is non-trivial and that the “one version fit all” approach [Dolbeau et al., 2013] is sub-optimal when faced with both a large optimization space and a large input space.

6.2 TileK

TileK is a programming model extending C/C++ through compiler directives. It enables us to define regions of code that have to be offloaded into kernels. In these offloaded regions of code, loops can be decomposed into tiles. These tiles will either appear as loops in the generated kernel or will be distributed across a multicore CPU or accelerators. Figure 6.1 depicts the compilation flow of TileK.

6.2.1 TileK language

TileK has two constructs: *kernel* and *loop*. Kernel constructs are used to designate a region of code that will be transformed into a kernel. The *kernel* construct is associated with *data* clauses specifying the data used by the kernel. The loop construct is used to mark loops that have to be tiled. The *loop* construct is associated with *tile* clauses. We show an example in Listing 6.1.

First, the region of code that has to be transformed into a kernel is marked using the *kernel* construct: `kernel data(A[0:n][0:m], mode:rw, live:inout)`. The associated *data* clause specifies that `A[0:n][0:m]` is read and written by the kernel (`mode:rw`) and lives in and out of the kernel (`live:inout`). `A[0:n][0:m]` represents the array *A* of dimension $n \times m$. Array dimensions are used for data transfers and for array flattening (linearizing). The additional access mode and liveness information are used for data

```

1 #pragma tilek kernel data(A[0:n][0:m], mode:rw, live:inout)
  {
3 #pragma tilek loop tile[0](static, 2) tile[2](dynamic)
  for (i = 0; i < n; i++)
5     #pragma tilek loop tile[1](static, 3) tile[3](dynamic)
      for (j = 0; j < m; j++)
7         A[i][j] += b;
  }

```

Listing 6.1: We use this simple kernel to illustrate TileK and associated code transformation. It adds a constant value to every elements of a matrix. The TileK annotations are used to specify a kernel, its data, and the way loops are transformed.

placement (using eventual constant memory) and data movements. Second, loops in this region of code are annotated with *loop* constructs: `loop tile[0](static, 2) tile[2](dynamic)` and `loop tile[1](static, 3) tile[3](dynamic)`. Loop constructs hold *tile* clauses which are of the form: `tile['order'](static, 'trip-count')` or `tile['order'](dynamic)`. The trip-count of a loop is the number of time the body of this loop is executed. The product of the trip-count of all tiles associated to one loop is equal to the trip-count of this loop. For *static* tiles, the trip-count is known at compile time, but the trip-count of *dynamic* tiles is determine at runtime. At this point, the TileK compiler requires evenly divisible loops. Allowing loops that are not evenly divisible is possible, but would require adding guards in the generated code. Adding these guards would add computation and could cause divergence which is costly on some accelerator (e.g. GPGPU). There can only be one dynamic tile else the system cannot be solved. When the kernel is generated, tiles from perfectly nested loops are sorted based on *order*, allowing the compiler to perform interchange.

6.2.2 Iteration Domain

TileK enables changing the order in which the iteration domain of a loop nest is traversed. Figure 6.2 shows how the iteration domain is separated between tiles. Each cell represents the iteration of coordinates (i, j) . For example, cell $(3, 4)$ executes `A[3][4] += b;`. The clauses `tile[0](static, 2)` and `tile[1](static, 3)` split the iteration

domain into six subdomains (demarcated with bold lines). And the clauses `tile[2](dynamic)` and `tile[3](dynamic)` iterate over each of these subdomains. The tiles are executed starting with the outer tiles first (i loop then j loop), then the inner tiles are executed (same i then j). The sequence of iterations is (0,0), (0,1), (0,2), (0,3), (1,0), ..., (1,3), (0,4), ..., (0,7), (1,4), ..., (1,7), (2,0), ..., (2,3), (3,0), ...

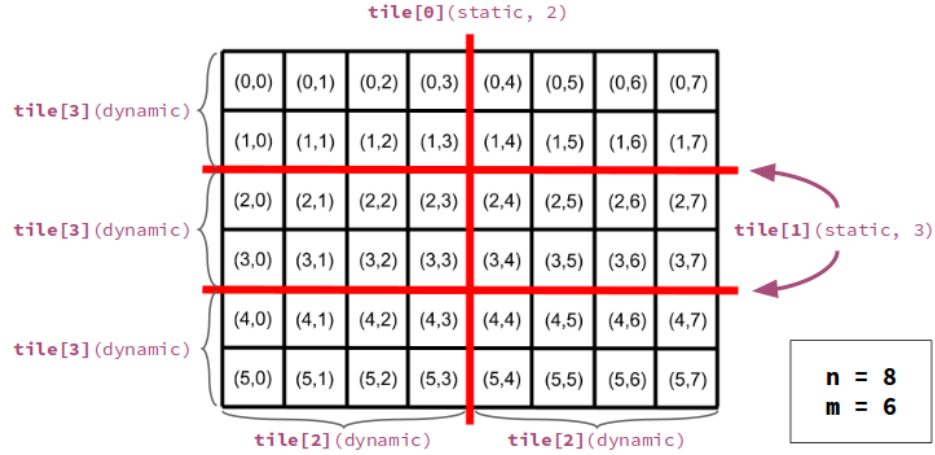


Figure 6.2: This figure illustrates the iteration domain for Listing 6.1 when $n = 8$ and $m = 6$. The static tiles become the outer most loops and iterate over the blocks formed by the dynamic tiles.

6.2.3 Distributed Kernels

TileK has two distinct extensions to generate SPMD kernels: *Threads* and *Accelerator*. Both introduce a special kind of tile that will be mapped either to software threads or to the execution units of an accelerator. We will present the syntax in the TileK language and the associated execution model.

6.2.3.1 TileK Threads

The TileK *Threads* extension permits distributing computations across multiple processors or cores. This is done by mapping a tile to the thread dimension, as shown in Listing 6.2. The clause `num_threads` declare the number of threads that the host will fork as depicted in Figure 6.3. This number of threads does not have to be known at

compile time. The computations are distributed based on the `tile(thread)`. Figure 6.4 shows how the iteration domain is distributed between the different threads.

```

1 #pragma tilek kernel num_threads(4)
2 {
3   #pragma tilek loop tile(thread) tile[1](dynamic)
4   for (i = 0; i < n; i++)
5     #pragma tilek loop tile[2](dynamic)
6     for (j = 0; j < m; j++)
7       A[i][j] += b;
8 }

```

Listing 6.2: The *Threads* extension of TileK applied to the same example as Listing 6.1. The `num_threads` clause was added to the `kernel` construct to determine the number of threads used by the generated kernel. The `tile(thread)` declares a tile of which each iteration will be mapped to a spawned thread.

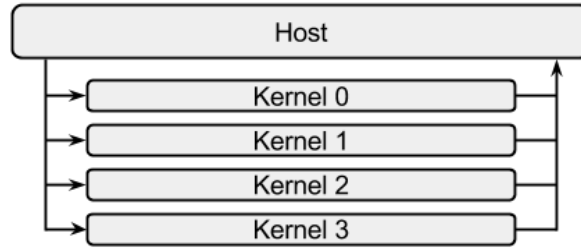


Figure 6.3: The host forks as many threads as requested by the `num_threads` clause. Each thread executes part of the computation (Figure 6.4) then the threads join.

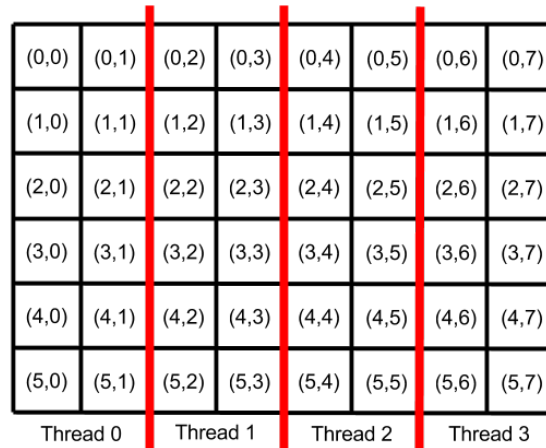


Figure 6.4: This figure shows how the iteration domain in Listing 6.2 is shared among the four threads from Figure 6.3. Each thread computes a block of 2×8 elements.

6.2.3.2 TileK Accelerator

Using the TileK *Accelerator* extension, kernels are offloaded to an accelerator using OpenCL. The TileK Accelerator model uses the OpenACC model for two levels of parallelism: *gang* and *worker*. *Gang* represents coarse grain parallelism while *worker* represents fine-grain parallelism.

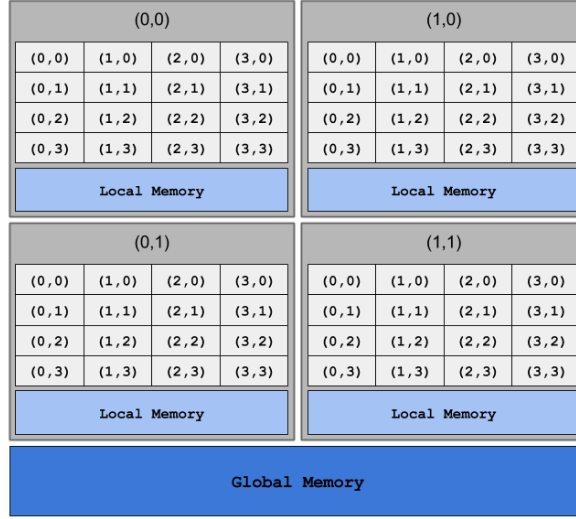


Figure 6.5: Illustration of the abstract machine for TileK *Accelerator*. Accelerators generally have two levels of parallelism: coarse-grain with multiprocessors that are mostly independent (share global memory) and fine-grain where all processing elements can be synchronized and share local memory.

Figure 6.5 shows an abstract machine used as a model by TileK *Accelerator*. This machine is made of a 2×2 array of gangs and each gang contains a 4×4 array of workers. It also shows the global and local memories. Global memory is accessible by all workers from all gangs while workers can only access the local memory of their parent gang.

Listing 6.3 shows an example of code annotated with TileK's *Accelerator* extension. The clauses *num_gangs* and *num_workers* declare the size of the abstract accelerator. Computations are distributed on this abstract machine using *gang* and *worker* tiles. The trip-count for *gang* and *worker* tiles is given by the corresponding *num_gangs* and *num_workers* clauses. On both loops, the inner-most tiles are dynamic. Each worker in the abstract machine will execute a block of the iteration domain.

```

2  #pragma tilek kernel num_gangs[0](2) num_gangs[1](2) \
    num_workers[0](4) num_workers[1](4)
3  {
4  #pragma tilek loop tile(gang, 0) tile(worker, 0) tile[0](dynamic)
  for (i = 0; i < n; i++)
6    #pragma tilek loop tile(gang, 1) tile(worker, 1) tile[1](dynamic)
      for (j = 0; j < m; j++)
8        A[i][j] += b;
  }

```

Listing 6.3: This code is the same as Listing 6.1, but uses the TileK *Accelerator* extension. This code instantiates an abstract accelerator made of 2×2 gangs of 4×4 workers as in Figure 6.5.

6.3 Generating Tiled Kernels

The first step to generate SPMD kernels is to generate tiled kernels. These kernels correspond to codes like Listing 6.1. In this case, the computation are not distributed, but the loops are tiled and the resulting tiles are reordered. In TileK’s compiler, the generation of tiled kernel is done through a two step process: (1) one process we call the LoopTree generator converts Rose’s IR into TileK’s IR, called LoopTree, on which the tiling is done, and (2) another process we call the Kernel generator converts LoopTrees into optimized ASTs that can be fed back into Rose for further transforming improvements.

We will first present TileK’s IR called LoopTree. Next, we describe a kernel generated for the motivation example (Listing 6.1). Finally, we will introduce the algorithm used at runtime to determine the length and stride of each tiles.

6.3.1 LoopTrees

When compiling code annotated with TileK, the region associated to the kernel construct is transformed into a *LoopTree*. They are a simplified Abstract Syntax Trees (ASTs) that can be specifically optimized in our TileK compiler. Any block of code that contains a restricted, but large and common set of for-loops, if-statements, and expression-statements can be model into *LoopTrees*. Here, our restricted set of for-loops corresponds to loops of the form `for (i = lb; i < ub; i += s)` and variations,

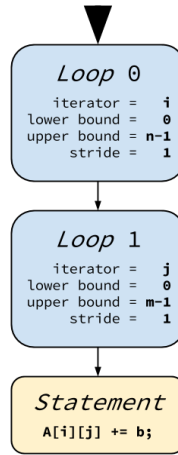


Figure 6.6: When directive are not considered, Listing 6.1 translates into a simple *LoopTree*. The i loop is identified as loop 0 which iterates from 0 to $n-1$ with a stride of 1. The j loop is identified as loop 1 which iterates from 0 to $m-1$ with a stride of 1 and loop 0 is its parent. The expression statement appears as the leaf of the tree and loop 1 is its parent.

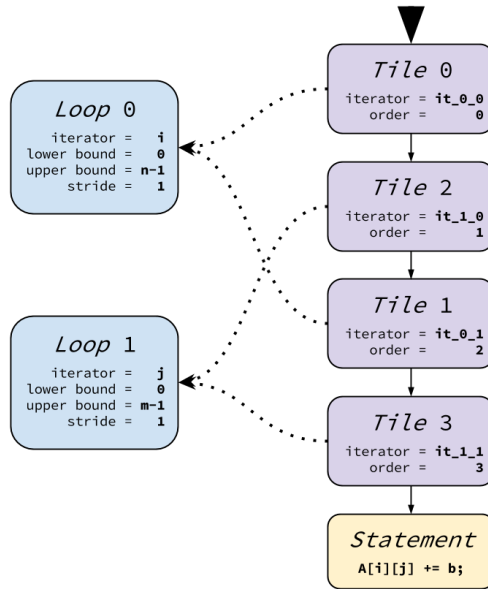


Figure 6.7: When the directives are considered, the *LoopTrees* for Listing 6.1 is more complicated than in Figure 6.6. Both loops are separated into two tiles. Loop 0 becomes Tile 0 and Tile 1. Loop 1 becomes Tile 2 and Tile 3. The order of the tiles in the *LoopTree* depends on the *order* argument (between square brackets).

including: greater-than and non-strict comparison operator or decrement/increment operators. `lb`, `ub`, and `s` need to be constant across the lifetime of the loop, but do not need to be known at compile time. We note that many popular scientific algorithms and most scientific kernels we have encountered fall into our constrained set of for-loops. The IR uses *fragments* of ROSE’s AST to represent variables (symbols), expressions, and statements. Figure 6.6 shows the *LoopTree* constructed from Listing 6.1 without considering the annotations. For each loop in the example, the *iterator* was identified alongside the *lower bound*, *upper bound*, and *stride*. In Figure 6.7, the *LoopTree* was modified according to the TileK annotations. The tile clauses on TileK’s loop constructs lead to the creation of tile nodes that take the place of the loop nodes. *Iterators* are constructed for each tile and tiles are sorted depending of their *order* parameter. Each tile is linked to the loop it originates from, providing the bounds and stride. *LoopTrees* are used to generate the kernel, however, we associate to each kernel a copy of its *LoopTree*. This *LoopTree* fully characterizes the kernel and present a few more advantages:

- it uses pieces of the original AST facilitating the generation of correct ASTs
- it has a higher level of abstraction than the original AST simplifying transformations
- it could be used to handle non traditional loops with minimal change to the subsequent transformations

6.3.2 Generated Kernel

Once the *LoopTree* has been tiled, the Kernel generator traverses it and generates a new AST for ROSE. This generated AST includes all the intricacies of the final kernel. It connects to TileK’s runtime support and takes care of lowering parameters and data accesses. This AST is finally unparsed using ROSE backend.

An example of the final kernel is shown in Listing 6.4. It was generated from the code in Listing 6.1. This kernel receives two arrays of pointers, one for the parameters and one for the data. These pointers are cast to their original types inside

```

void klt_kernel_0(
2 void ** param, void ** data,
  struct klt_loop_context_t * loop_ctx
4 ) {
    float *A = (float *)data[0];
    float b = *((float *)param[2]);
    int m = *((int *)param[1]);
    int n = *((int *)param[0]);
    int l_0, l_1;
    int t_0, t_1, t_2, t_3;
    for (t_0 = 0;
12     t_0 < klt_get_tile_length(loop_ctx,0);
        t_0 += klt_get_tile_stride(loop_ctx,0))
14     for (t_2 = 0;
        t_2 < klt_get_tile_length(loop_ctx,2);
        t_2 += klt_get_tile_stride(loop_ctx,2))
16         for (t_1 = 0;
            t_1 < klt_get_tile_length(loop_ctx,1);
            t_1 += klt_get_tile_stride(loop_ctx,1))
20             for (t_3 = 0;
                t_3 < klt_get_tile_length(loop_ctx,3);
                t_3 += klt_get_tile_stride(loop_ctx,3)) {
22                 l_0 = klt_get_loop_lower(loop_ctx,0)
24                     + t_1 + t_0;
                    l_1 = klt_get_loop_lower(loop_ctx,1)
26                     + t_3 + t_2;
                    A[l_0 * m + l_1] += b;
28             }
    }
}

```

Listing 6.4: Sequential kernel generated for Listing 6.1. Both loops are separated into two tiles. The resulting tiles are reordered, grouping the outer tiles and inner tiles together.

the kernel. Each tile from the *LoopTree* leads to the generation of one for-loop. The length and stride of this for loop is obtained from the `loop_ctx` object of type `struct klt_loop_context_t`. Because tiles from multiple loops can be ordered randomly if the loops are perfectly nested, the iteration of the associated loops are determined at the inner-most loop body for this loop nest. At any time, the value of a loop's iterator is the sum of its lower bound and the tiles composing the loop. Any further optimizations are left to the backend compiler.

6.3.3 From Loop Bounds to Tile Bounds

We cannot determine the length and stride of the tiles at compile time. We need to wait until the bounds and strides of the loops are known at runtime. Before launching a kernel, Algorithm 6.1 determines the length and stride of all the tiles of one loop. For this algorithm, tiles are sorted following their declaration order (not the *order* parameter). The inputs of the algorithm are: (1) bounds and stride of the loop ($upper^{in}$, $lower^{in}$, and $stride^{in}$), (2) number of tiles (num_tile^{in}), and (3) kind and trip count for each tile (arrays $kind^{in}$ and $trip_count^{in}$). The outputs are the length and stride for each tile (arrays $length^{out}$ and $stride^{out}$). The algorithm goes from outermost tile to the innermost. It propagates the loop length dividing it by the trip count of each tiles. It stops when/if it encounters a dynamic tile. If a dynamic tile is encountered, it goes from innermost tile to the outermost. It propagates the loop stride multiplying it by the trip-count of each tile.

6.4 Generating SPMD Kernels

Section 6.3 presented the generation of a tiled kernel. In this section, we explain how this process can be used to build SPMD kernels. These kernels distribute computations across threads or an accelerator.

Algorithm 6.1 Given a loop's bounds and stride, its number of tiles, and the kind and trip-count of each tile, this algorithm computes the length and stride for each tile.

```

 $L \leftarrow upper^{in} - lower^{in}$ 
 $i \leftarrow 0$ 
while  $i < num\_tile^{in}$  and  $kind_i^{in} = static$  do
     $length_i^{out} \leftarrow L$ 
     $stride_i^{out} \leftarrow L / trip\_count_i^{in}$ 
     $L \leftarrow stride_i^{out}$ 
end while
 $S \leftarrow stride^{in}$ 
 $j \leftarrow num\_tile^{in} - 1$ 
while  $j \geq i$  and  $kind_j^{in} = static$  do
     $length_j^{out} \leftarrow S * trip\_count_j^{in}$ 
     $stride_j^{out} \leftarrow S$ 
     $S \leftarrow length_j^{out}$ 
end while
if  $kind_j^{in} = dynamic$  then
     $length_j^{out} \leftarrow L$ 
     $stride_j^{out} \leftarrow S$ 
end if

```

6.4.1 Kernel Index to Tile Iteration

In Section 6.2.3, we presented the machine models associated with TileK's *Thread* (Figure 6.3) and *Accelerator* (Figure 6.5) extensions. In both of these extensions, the machine is composed of multiple execution units. Each of these execution units is identified by a unique index. In the case of the *Thread* extension, the indexes are made of one integer between zero and the number of spawned threads. For the *Accelerator* extension, the dimension of the index varies depending on the gang and worker dimensions. This index space can have two, four, or six dimensions ; from one gang and one worker dimension to three gang and three worker dimensions.

For a given kernel, there will be as many distributed tiles as there is dimensions in the index. Kernels are associated with a single index (one instance of the kernel per execution unit), and each of the distributed tiles take only one value per execution of the kernel. Hence, distributed tiles are not loops in the generated kernel. Instead, the current iteration of a distributed tile is determined using its stride and the kernel's

index.

6.4.2 Threads

In TileK’s *Threads* extension, generated kernels are distributed between a group of threads as shown in Figure 6.3. The kernel is launched once for each thread and receives a *thread ID* (`tid`). The iteration of the thread tile is equal to the product of `tid` and the tile’s stride. Listing 6.5 shows the kernel generated for the annotated code in Listing 6.2. The distributed tile iteration `t_0` is computed at Line 10.

```
void klt_kernel_0(  
2 int tid, void ** param, void ** data,  
  struct klt_loop_context_t * loop_ctx  
4 ) {  
  float *A = (float *)data[0];  
6  float b = *((float *)param[2]);  
  int m = *((int *)param[1]);  
8  int n = *((int *)param[0]);  
  int l_0, l_1;  
10 int t_0 = tid * klt_get_tile_stride(loop_ctx,0);  
  int t_1, t_2;  
12 for (t_1 = 0;  
      t_1 < klt_get_tile_length(loop_ctx,1);  
14      t_1 += klt_get_tile_stride(loop_ctx,1) ) {  
  for (t_2 = 0;  
      t_2 < klt_get_tile_length(loop_ctx,2);  
16      t_2 += klt_get_tile_stride(loop_ctx,2) ) {  
18    l_0 = klt_get_loop_lower(loop_ctx,0) + t_0 + t_1;  
    l_1 = klt_get_loop_lower(loop_ctx,1) + t_2;  
20    A[l_0 * m + l_1] += b;  
  }  
22 }
```

Listing 6.5: Kernel generated when distributing the code from the motivating example using TileK’s Thread extension (Listing 6.2). The variable `tid` represents the Thread ID, and it is used to determine the current iteration of the thread tile.

6.4.3 Accelerator

In the TileK *Accelerator* extension, the kernel is offloaded to an accelerator using OpenCL. Depending on the number of gangs and workers required by the kernel,

a specialized kernel using 1D, 2D, or 3D arrays is launched.. The TileK Accelerator extension maps gangs to work-groups and workers to work-items of the OpenCL NDRange. OpenCL NDRange are 2D or 3D arrays of groups, which themselves are 2D or 3D arrays of work-items (threads). Given this mapping, the index used to calculate the iteration of `tile(gang,1)` is OpenCL’s `get_group_id(1)` (2^{nd} element of the index of the current group). Similarly, `tile(worker,0)` maps to `get_local_id(0)` (1^{st} element of the index of the current work-item). In Listing 6.6, we show the kernel generated for the annotated code in Listing 6.3. The distributed tiles from the outer loop of the original code (`i` loop in Listing 6.3) are associated to variables `t_0` and `t_1`. These variables are assigned on Lines 6 and 8 using OpenCL’s group ID and OpenCL’s local ID for the gang tile and the worker tile, respectively. Similarly, `t_3` and `t_4` are the tiles from the `j` loop in Listing 6.3 and are assigned at Lines 10 and 12.

6.5 Optimization Space Exploration

TileK allows one to describe many different versions of a given loop nest. This section motivates the need for predictive modeling to generate optimized kernels. The goal of this modeling would be to accelerate an iterative compilation process [Agakov et al., 2006] [Chen et al., 2010]. It has also been shown that iterative compilation needs to be input aware [Demmel et al., 2005] [Liu et al., 2009]. A predictive model that is input aware (i.e., uses features of the input especially array sizes) is necessary to generate the best version of a given code. To illustrate our point, we evaluate the performance of many versions of four simple computation kernels:

- **2d-conv**: 2D convolution (Listing 6.7)
- **sgemm**: single precision general matrix multiply (Listing 6.8)
- **syr2k**: symmetric rank-2k operations (Listing 6.9)
- **doitgen**: multiresolution analysis kernel (MADNESS) (Listing 6.10)

We distributed these loops with the *Threads* and the *Accelerator* extensions of TileK. For each kernel, we evaluated eight *threads* versions and forty *accelerator*

```

2  __kernel void klt_kernel_0(
    int n, int m, float b, __global float *A,
    __constant struct klt_loop_context_t *loop_ctx
4  ) {
    int l_0, l_1, t_2, t_5;
    int t_0 = get_group_id(0)
        * klt_get_tile_stride(loop_ctx,0);
    int t_1 = get_local_id(0)
        * klt_get_tile_stride(loop_ctx,1);
    int t_3 = get_group_id(1)
        * klt_get_tile_stride(loop_ctx,3);
    int t_4 = get_local_id(1)
        * klt_get_tile_stride(loop_ctx,4);
    for (t_2 = 0;
        t_2 < klt_get_tile_length(loop_ctx,2);
        t_2 += klt_get_tile_stride(loop_ctx,2))
        for (t_5 = 0;
            t_5 < klt_get_tile_length(loop_ctx,5);
            t_5 += klt_get_tile_stride(loop_ctx,5)) {
    20     l_0 = klt_get_loop_lower(loop_ctx,0)
            + t_1 + t_0 + t_2;
    22     l_1 = klt_get_loop_lower(loop_ctx,1)
            + t_4 + t_3 + t_5;
    24     A[l_0 * m + l_1] += b;
    }
26 }

```

Listing 6.6: Code generated for Listing 6.3 using TileK *Accelerator* with OpenCL backend.

```

void 2dconv(
2   int n, int m,
   float ** A, float ** B
4 ) {
   int i, j;

6   for (i = 1; i < n-1; i++) {
8       for (j = 1; j < m-1; j++) {
           B[i][j] = 2.34 * A[i-1][j-1]
10              + 4.64 * A[i+1][j+1]
              - 7.93 * A[i-1][j ]
12              + 1.26 * A[i ] [j-1]
              + 0.64 * A[i+1][j ]
14              + 1.34 * A[i ] [j+1]
              - 2.17 * A[i-1][j+1]
16              + 7.22 * A[i+1][j-1]
              - 9.59 * A[i ] [j ];
18     }
   }
20 }

```

Listing 6.7: **2d-conv**: 2D Convolution with random, hard-coded, floating coefficients

```

void sgemm(
2   int n, int m, int p,
   float ** A, float ** B, float ** C,
4   float alpha, float beta
) {
6   int i, j, k;

8   for (i = 0; i < n; i++) {
       for (j = 0; j < m; j++) {
10          C[i][j] *= beta;
           for (k = 0; k < p; k++) {
12              C[i][j] += alpha * A[i][k] * B[k][j];
           }
14     }
   }
16 }

```

Listing 6.8: **sgemm**: single precision general matrix multiply

```

void syr2k(
2   int n, int m,
   float ** A, float ** B, float ** C,
4   float alpha, float beta
) {
6   int i, j, k;
   for (i = 0; i < n; i++) {
8       for (j = 0; j < n; j++) {
           C[i][j] *= beta;
10          for (k = 0; k < m; k++) {
               C[i][j] += alpha * A[i][k] * B[j][k];
12               C[i][j] += alpha * B[i][k] * A[j][k];
           }
14       }
   }
16 }

```

Listing 6.9: **syr2k**: symmetric rank-2k operations

```

void doitgen(
2   int R, int Q, int P,
   float *** A, float *** sum, float ** C4
4 ) {
   int r, q, p, s;
6
   for (r = 0; r < R; r++) {
8       for (q = 0; q < Q; q++) {
           for (p = 0; p < P; p++) {
10              sum[r][q][p] = 0;
               for (s = 0; s < P; s++)
12                  sum[r][q][p] += A[r][q][s] * C4[s][p];
           }
14           for (p = 0; p < P; p++)
               A[r][q][p] = sum[r][q][p];
16       }
   }
18 }

```

Listing 6.10: **doitgen**: multiresolution analysis kernel (MADNESS)

versions. For both the thread and accelerator experiments, we only measure the time it takes for the kernel to run, ignoring initialization and data-transfers, which are not optimized. Each kernel is executed five times and the average of the five measurements is used. We compare the different optimized versions by looking at the number of floating point operations per second (flops) that they compute. Given that we are comparing different kernels for many different inputs, flops is the best measurement when optimizing runtime.

Varying the inputs for these codes is simple: none of the controls are data-dependent. As there is no loop or conditional depending on the values of the inputs, only the input’s size influences the runtime of a given version. For each version, we evaluated a few hundred input sizes depending on the kernel. Table 6.1 summarizes the input space for each of the four kernels used in the experiments.

	parameters	float operations	# inputs
2d-conv	n,m	$17.n.m$	225
sgemm	n,m,p	$n.m.(6.p + 1)$	1152
syr2k	n,m	$n^2.(6.m + 1)$	252
doitgen	r,q,p	$2.r.q.p^2$	250

Table 6.1: Summary of the input space for each of the four codes used in the experiments. It shows the parameters (positive integers) that define the input sizes of the different problems, the formula which gives the number of floating point operations as a function of the parameters, and the number of inputs (combinations of parameters) that was evaluated.

These experiments were conducted using Elastic Cloud Compute (EC2) from Amazon Web Services (AWS). We used compute instances with 16 VCores (c4.4xlarge) to evaluate the *Threads* extension and GPU instances (g2.2xlarge) for the *Accelerator* extension. The c4.4xlarge instances are equipped with Intel Xeon E5-2666 v3 (Haswell) processors and 30 GiB of RAM. The g2.2xlarge instances are equipped with NVIDIA Grid K520 GPU (Kepler architecture with 1,536 CUDA cores and 4GB). OpenCL was used to target the GPU, using both our OpenCL implementation of the TileK runtime and the corresponding OpenCL target. The g2.2xlarge instances used OpenCL 1.2

CUDA with the NVIDIA driver version 352.79. All codes were compiled using gcc 4.8.3 and the -O3 optimization flag.

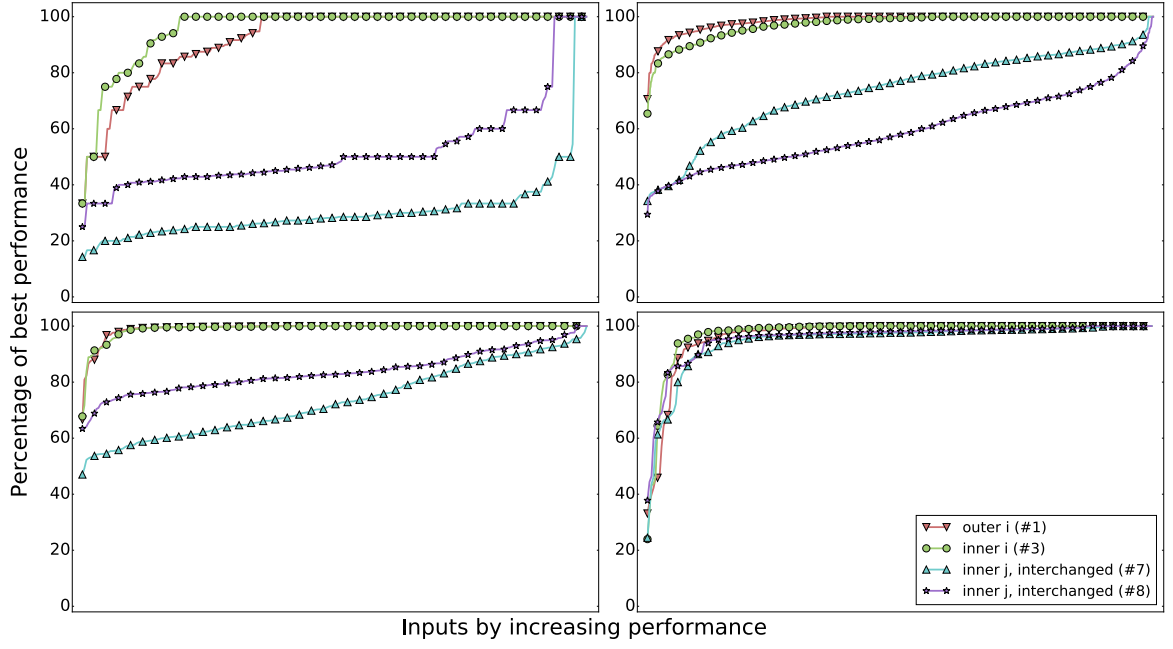
6.5.1 Thread Experiments

For each of the four kernels, we generated eight threaded versions of the computation kernel. The loops `i` and `j` are annotated with one dynamic tile each. Then a *thread* tile is placed either before or after the dynamic tile of one of the loops. Finally the dynamic tiles can be interchanged using the `order` parameter (square bracket on the tile clause). Table 6.2 summarizes the eight tile configurations used to parallelize the two outer loops of each of the four codes. For example, the tiling in Listing 6.2 corresponds to the first tile configuration in Table 6.2.

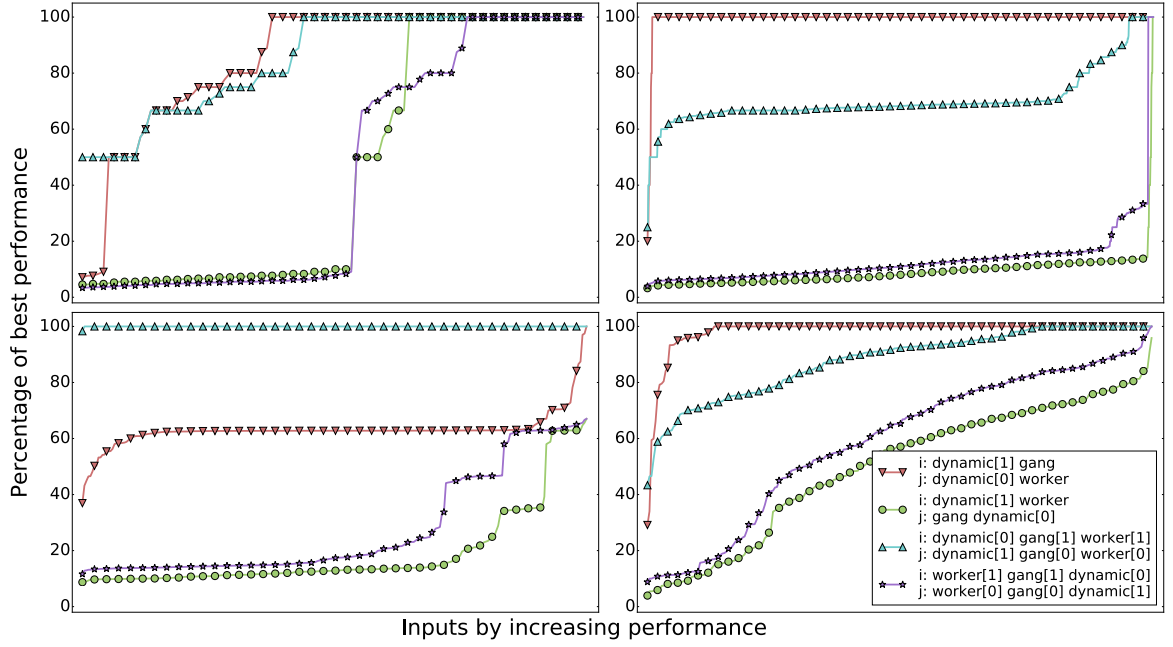
#	parallelized loop	before/after dynamic tile	dynamic tiles order
1	i	before	i - j
2	i	before	j - i
3	i	after	i - j
4	i	after	j - i
5	j	before	i - j
6	j	before	j - i
7	j	after	i - j
8	j	after	j - i

Table 6.2: The eight tile configurations used to evaluate the TileK *Threads* extension. The first column gives the version’s ID. The second column specifies which of the two loops is annotated with the `tile(thread)` clause. The third column tells whether this thread tile is placed before or after the dynamic tile. Finally, the fourth column shows the order in which the two dynamic tiles are unparsed (interchange).

The results of our experiments with the thread versions of these four kernels are presented in the top row of Figure 6.8. These graphs present the percentage of peak performance of four versions of each kernel for various inputs sizes. The percentage of peak performance allows one to compare the performance of one version for one input to the performance of the best version for the same input. We show versions 1, 3, 7, and 8 from Table 6.2. These versions are the two best (1 and 3) and the two worst (7 and 8). On the X-axis inputs are sorted based on increases in performance,



(a) TileK Threads Extension



(b) TileK Accelerator Extension

Figure 6.8: Inputs space exploration for TileK threads and accelerator. X-axis represent the various inputs sorted by increasing performance (independently for each version). Y-axis show the percentage of peak performance reached by each version for the different inputs.

independently for each version.

We can draw a few observations from these graphs:

- No version is the best for all inputs: version 3 for **2d-conv** performs approximately 65% worse than another version for at least one input
- The best version depends on the code being optimized: version 3 is generally the best for **2d-conv** while it is version 1 for **sgemm**
- Even poorly performing versions can have at least one input for which they perform as well as the best version

For all four kernels, versions 7 and 8 have the worst performance. This behavior is easily explained as these versions have their parallel tile as the inner tile of the innermost loop. Distributing the loop nest in this fashion almost always reduce the locality of the kernel. Mostly because when translating an algorithm in C, we order the loops in the same order as the dimensions of the principal array. It means that the inner loop usually accesses rows with the most locality. Hence, placing the parallel tile on the inner tile of the innermost loop, breaks this locality.

6.5.2 Accelerator Experiments

While the *TileK Threads* extension only provides one dimension to distribute the loop nest, the *Accelerator* extension enables up to six dimensions, thus giving us a large search space of code versions, i.e., forty in our case. In these experiments, we consider the two machine models for the Accelerator extension shown in Table 6.3. These machine models include 1D and 2D configurations of gangs and workers. For both of these machine models, we evaluate the performance of different tiling configurations. To construct these tiling configurations, we start with one dynamic tile for each loop.

For the 1D machine model, we add one gang tile and one worker tile. We place one of each on the i and j loops, either before or after the dynamic tile. Having either the gang or worker tile on the i loop (and vice-versa for the j loop) creates two possibilities. The position of the gang and worker tile in relation to the dynamic tiles creates four possibilities. Finally, the dynamic tiles can be interchanged creating another two possibilities for a total of 16 different tiling configurations.

For the 2D machine model, we add two gang tiles and two worker tiles. Each of the two loops receives either gang and worker #0 or gang and worker #1, creating two possibilities. Then, there are six ways to order the dynamic, gang, and worker tiles. Finally the dynamic tiles can be interchanged creating another two possibilities for a total of 24 tiling configurations.

This search space is not exhaustive, search spaces for both 1D and 2D machine models can be extended and other machine models can be tested using various numbers of gangs and workers.

gangs	workers	# tiling
(16,)	(256,)	16
(4,4)	(16,16)	24

Table 6.3: We evaluated two different machine models: 1D and 2D. The total number of gangs is always 16. The total number of workers is always 256.

We evaluated each of the forty versions of the four codes on GPUs. The bottom row of graphs in Figure 6.8 highlights some results from four representative versions. The X-axis is ordered based on increases in performance. The Y-axis shows the percentage of peak performance.

Two of the four versions shown use a 1D topology and the other two versions use a 2D topology. The first version (based on the legend’s order) uses a 1D topology. It has the gang and worker as the innermost tile of the i and j loops, respectively. This configuration is more efficient than the configuration of the second version as it increases the memory coalescence (neighboring execution units accesses neighboring memory cells). Similarly, the third configuration, which uses a 2D topology, has the two worker tiles as innermost tiles. This configuration can achieve better memory coalescing than the fourth configuration.

Across the four codes, there are a few noticeable effects especially when looking into the first and third versions. These two versions have very different performance profiles for **sgemm** and **syr2k**. In the case of **sgemm**, the first version is almost always the best. This could be due to the flattening of the 2D topology when OpenCL threads

are mapped to the execution units. For **syr2k**, the third version is almost always the best. In this case, the symmetrical distribution of gangs and workers across both loops is advantageous to the symmetrical nature of the **syr2k** kernel.

6.6 Related Work

This work is a generalization of our experimentations with OpenACC [OpenACC, 2011]. We presented [Vanderbruggen and Cavazos, 2014] how a static tiling of loops could be used generate OpenCL kernels for OpenACC. This paper presents a much more flexible technique.

Loop tiling [Wolfe, 1989] [Xue, 2000] has been used as a technique to generate parallel workloads. The classic approach to tiling transforms one loop into outer and inner tiles. When multiple loops are tiled, all outer tiles and inner tiles are grouped. Computation is distributed by assigning each iteration of the outer tiles to one execution unit. In our technique, each loops can be divided into many tiles. Tiles are not ordered from the outer to the inner tile, instead the order is set by the user. This approach enables more control over the resulting data access pattern by mixing tiling and interchanges.

HOMP [Liao et al., 2013] is an early implementation of the OpenMP Accelerator Model using the ROSE Compiler [ROSE, 2017]. In OpenMP [Dagum and Menon, 1998], loops are divided into chunks (or blocks) and each chunk is executed “atomically” by one thread. In HOMP, the workload is distributed in the same fashion preventing usage of the multidimensional grids and blocks of CUDA.

An implementation of OpenACC targeting CUDA is presented in [Tabuchi et al., 2014]. In this work, each loop has at least one counterpart in the generated kernel. The bounds and stride of the generated loop depends on the index of the kernel. Finally the iteration variable is reconstructed for each iteration of this loop. Unfortunately, this publication does not provide details on this process, preventing a better comparaisn. Other implementations of OpenACC [Tian et al., 2014, Reyes et al., 2012] directly map loops to one dimension of the CUDA grid/block.

OpenACC 2.0 introduces a tile clause that can be applied to loops and which is supported by the PGI Compiler [Wolfe, 2015]. However, this tile clause is different from our approach. OpenACC tile clause has a similar effect to the tile directive of the HMPP Codelet Generator Directives [CAPS-e, 2012]. In both compilers, the loop to be tiled is separated between an outer tile and an inner tile. When multiple loops are tiled they are reorganized accordingly, all outer tiles then all inner tiles. Finally, in the PGI Compiler, other clauses applied to the loop are moved to either of the generated loops. The *gang* clause moves to the outer tile, while *worker* and *vector* clauses are applied to the inner tile.

6.7 Conclusion

In this chapter, we presented an optimizing compiler. Given a computation kernel, implemented in C, this compiler lets the user add annotations to specify how it should be parallelized. We have shown that the resulting tuning space can be vast, especially when we pay attention to the inputs of these kernels. Indeed, the best optimization can depends on the inputs' sizes.

Fully evaluating the training set would be prohibitively expensive in real cases. In such case, we can use iterative compilation to explore the tuning space. In the next chapter, we construct performance prediction models to guide such iterative compilation process. These models leverage TileK internal representation to represent the optimized kernels and their inputs.

Chapter 7

PERFORMANCE PREDICTION FOR COMPUTATION KERNEL TUNING

7.1 Introduction

In Chapter 6, we presented a parallelizing compiler, TileK. We showed that this compiler can be used to generate many variants of a given computation kernel. However, evaluating this tuning space exhaustively is prohibitively expensive. In this chapter, we construct a performance prediction model that can be used to guide the search to find the best variants of an optimized kernel in this space.

We are particularly interested in the tuning of legacy application where both computation kernels and workloads are well defined. Indeed, applications are often composed of distinct kernels which can be used with a variety of inputs. We showed in the previous chapter that the best optimizations for a given kernel sometime depend on the input size. We want to minimize the number of evaluations to determine the best optimized version of a kernel for any possible input.

To optimize such an application using TileK, we need a performance prediction model to use for iterative compilation. As is often the case with machine-learning, the main issue is to characterize the problem in a way that is amenable to machine-learning. There are two issues we have to deal with in terms of having a variable length input and needing to represent it in a fix length characterization for machine learning. A first issue is that TileK can apply any number of transformations to a kernel, changing the structure of the loop-nest. It is not possible to represent an arbitrary set of optimizations in a fixed size representation. Instead, the optimized kernel can be represented using TileK Internal Representation (IR). A second issue comes from considering various inputs for the kernels. Given that our work is limited

to statically controlled loops, we can summarize these inputs using their size. However, the number of inputs and their dimensions is not fixed across all kernels. We solve the issue of having a variable length input by embedding the trip count of each loop into TileK IR. The resulting graph-based representation is specific to each triplet of kernel, optimization, and input. This representation solves the issues of representing variable length optimizations and inputs.

In this chapter, we evaluate the potential of using TileK IR to iteratively tune a collection of computation kernels. First, we describe how TileK IR and the runtime measurement are made usable by machine learning. Second, we use a combination of neural networks (NN) and support vector machines (SVM) to build performance prediction models. Finally, we compare how fast iterative compilation converges to good optimized kernels when guided by our machine learning models versus using random search.

7.2 Dataset

The first step of any machine-learning work is to build a representation of the problem which can be used by machine learning algorithms. In this case, we transform TileK IR (Section 6.3.1) into a feature graph, defined in Section 3.2. We also have to define the training targets used to train the model. As our goal is to guide iterative compilation, we use targets that are related with the performance of each optimized kernel.

7.2.1 Feature Graph from TileTree

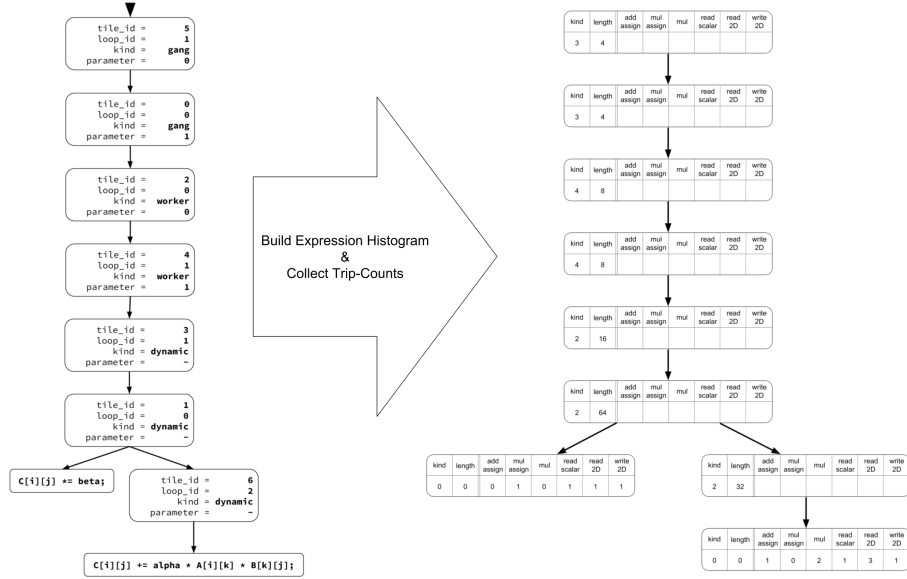
TileTrees are tree data structures that represent optimized versions of a computation kernel. Each node in a TileTree represents either a tile or, when it is a leaf, an expression. When the kernel is generated, tiles either become loops or computation that is distributed across the computing elements of the hardware. Expressions are composed of variable references, array indexing, arithmetic operations, or assignments.

The process of converting a TileTree into an input that can be ingested by a neural network is depicted in Figure 7.1. The first step is to build a vector that represents expressions in the TileTree and collects the trip-counts of the tiles. This vector representing an expression is a histogram of the operations. As TileTrees use ROSE Compiler IR to represent expressions, it is simply a matter of counting each type of node. The trip-count of each tile depends on the input sizes and how the computation are distributed on the different tiles. The second step propagates the histogram of each leaf upward in the data structure. The histogram associated with each node is the sum of the histograms of its children nodes multiplied by the trip-count of the tile. During this step, the histogram’s values are replaced by their logarithms.

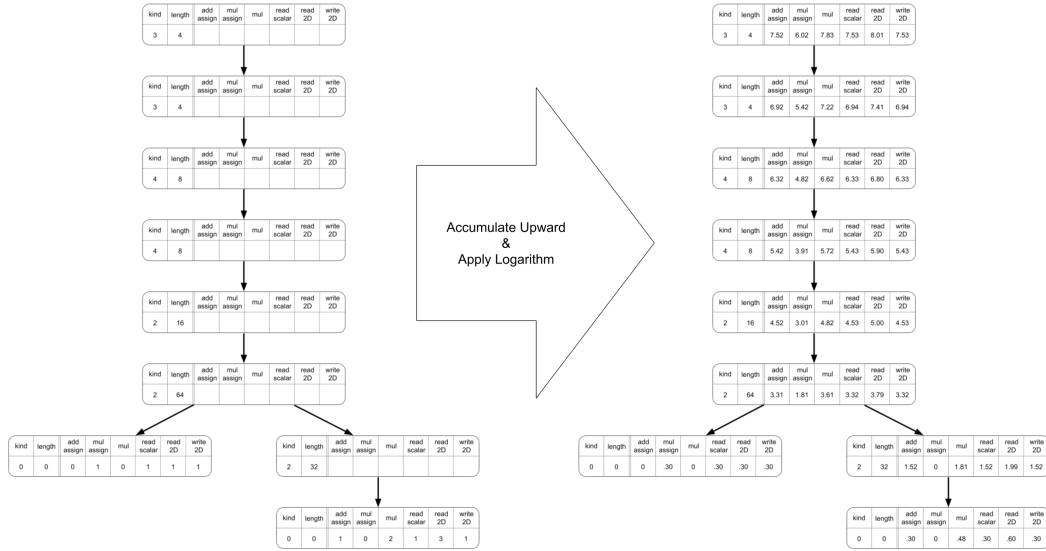
When this process is applied to our optimization search space, our resulting feature graphs have thirty elements. In addition, as the kernels considered in this experiment are small, the maximum number of nodes is twelve. This work takes advantage of the small size of these graphs to compare our graph spectral feature approach with a more straightforward approach of using our Feature Graphs (detailed in Section 7.3.1.1).

7.2.2 Targets

We not only consider different computation kernels but we optimize these kernels for multiple input sizes. The exhaustive tuning space exploration performed in Chapter 6 provided us with the execution time for each optimized kernel and their inputs. These execution times, combined with the kernel’s complexity and the input size, determine the performance of each optimization as the rate of floating point operations per second (flops). We then normalized these performance measurement with respect to the best performance for each pair of kernel and input. Finally, we discretized these values into 100 bins, numbered from 1 to 100 corresponding to the best and worst case, respectively. Essentially, target 1 means that we are within 1% of the best performance, while 4 means to be between 3% and 4% of the best performance.



(a) Collect expression histograms and loop trip-counts



(b) Propagate histograms upward

Figure 7.1: This figure shows how Tilek’s IR, TileTree, is transformed into a Feature Graph that is well suited for machine-learning. We take an optimized version of SGEMM, which uses a 2D OpenCL NDRange. We start with the TileTree representing one optimized version of the computation kernel. The inputs’ sizes are used to determine the trip-counts of each tile. Expressions are traversed to collect the number of arithmetic and data read/write operations. These counts are propagated up the tree, multiplied by the length of each tile. The resulting nodes of the Feature Graph are vectors with eight values: the type of node, its length, three operations (add-assign, mul-assign, and mul), and three data access (read scalar, read 2D array, and write 2D array) operations.

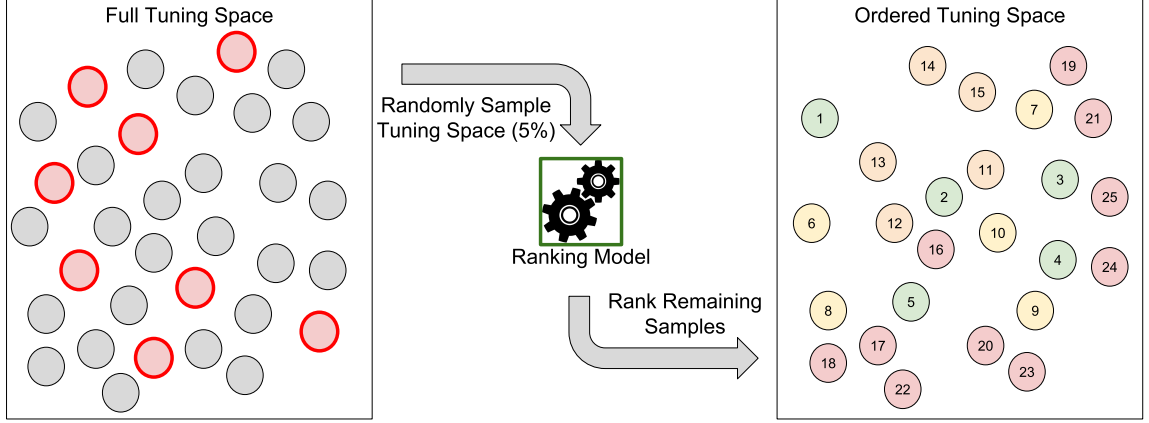


Figure 7.2: Given a previously unseen tuning space, we randomly sample 5% of the space. The performance measured on these samples is used to train a ranking model. This model is used to order the remainder of the search space.

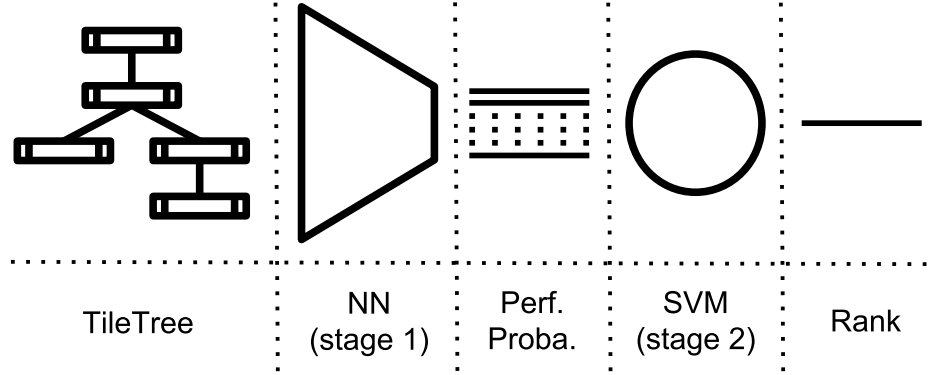


Figure 7.3: Our ranking model is made of two stages: a neural-network and a support vector machine. The NN determines the probabilities of a given TileTree to reach a certain percentage of the best performance. The SVM uses these probability to rank the TileTree.

7.3 Models

Our goal is to build models that can guide an iterative compilation process, as presented in Figure 7.2. We start by sampling five percent of the search space to build the training set for the model. The remainder of the space is ranked using the model. We evaluate the performance of the ranking model in ordering the tuning space compared to random search.

The ranking models that we built for this work have two machine learning

stages, as depicted in Figure 7.3. First, a neural network (NN) uses the feature graph representing one optimized version of a kernel for one specific input size and predicts the likelihood of it falling into each fraction of the best performance. Second, a support vector machine (SVM) uses the NN’s predictions to determine whether or not the optimized TileTree is one of the top performers (within 5% of the best optimized version for a given input). Finally, we use the SVM decision function (distance from the separating hyperplan, see Section 2.4) to rank the remainder of the search space.

7.3.1 Neural Networks

The first stage of our ranking models are neural networks which classify TileTrees into one of a hundred classes. These classes correspond to the discretized fraction of the best performance. The TileTrees are transformed into feature graphs following the process described in Section 7.2.1. In Chapter 3, we discussed the fact that feature graphs are not directly ingestible by a neural network. In this set of experiments, the size of the graphs are limited to 12 nodes. This restriction makes it possible to make comparisons using the node features or the graph spectrum features. We also compare the potential of different NN architectures, i.e. perceptrons, multi-layer perceptrons, and convolutional architectures.

7.3.1.1 TileTree Representations

We consider two characterizations of the TileTree, both based on the feature graphs described in Section 7.2.1. The first characterization presents the raw feature vectors from the graph’s nodes and we refer to this as the **features** representation of TileTree. They are stored as a 2D tensor where each row corresponds to one feature vector. The rows are ordered using a depth-first traversal of the TileTree. The second representation is our graph spectral features approach, which we presented in Section 3.5. It is referred as the **spectrum** representation of the TileTree. It also yields a 2D tensor, but each row contains an eigenvalue and the corresponding projected feature vector (both real and imaginary parts). These two representations yield

tensors of sizes 12×30 and 12×62 , for the features and spectrum representations, respectively. For TileTree with less than twelve nodes, the representations are padded with zeros.

7.3.1.2 Architectures

We consider three neural networks for both our TileTree representations. First, we evaluate a simple perceptron, the simplest neural network architecture. Second, we look at a multi-layer perceptrons (MLP) with eight hidden layers. The first four hidden layers have 512 neurons each, and the next four layers have 128 neurons each. Third, we replace the first four layers of the MLP with convolutional layers. These layers are applied to each row of the inputs individually. The last four hidden layers are not changed. They are fully-connected with 128 neurons each.

The perceptron gives us a baseline on the amount information that can be extracted from the representation. With a single layer, perceptrons are limited to linearly separable spaces. The MLP permits us to model much more complex spaces at the cost of a large number of parameters. It results in much longer training times. Finally, we use convolutional layers to reduce the number of parameters in the first four layers. This reduces the number of parameters to optimize, without weakening the resulting models.

7.3.2 Support Vector Machine

Each SVM model uses the predictions from one NN, which is one hundred probabilities between zero and one (with the sum of these probabilities equaling to one). The SVM model predicts if a given TileTree is within five percents of the best performance, corresponding to the five first targets of the NN. We use the SVM model to rank order the remainder of the search space.

7.3.3 Evaluation

We evaluate the models by comparing them to a random traversal of the search space (random search). First, we initialize the best known performance for each kernel

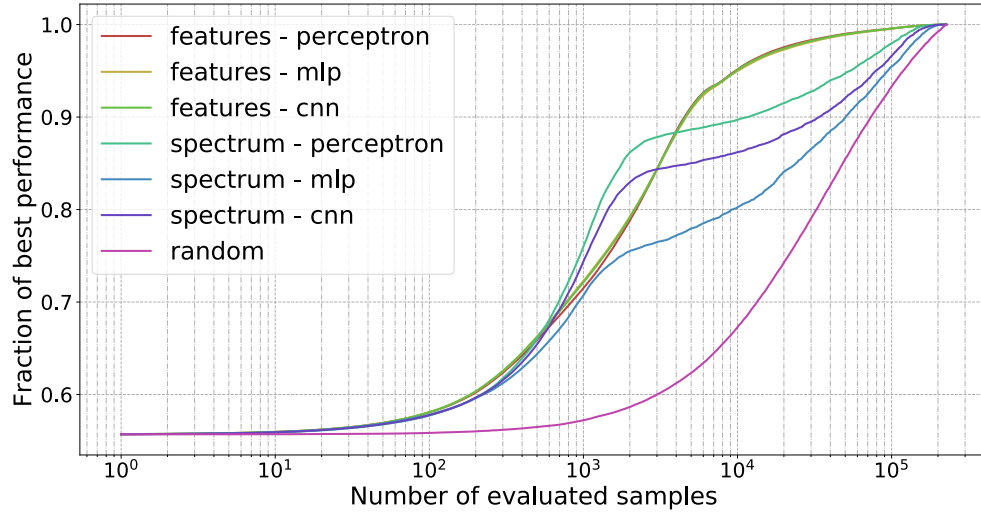


Figure 7.4: The average observed performance as a function of the number of evaluated optimized kernels. The X-axis shows the number of evaluated kernels on a logarithmic scale. The Y-axis represents the average observed performance so far up to that point.

and input pair using the training set (5% of the dataset). The remainder of the search space is traversed, either using the order of the SVM ranking model or using random search. For each new observed sample, we update the corresponding (kernel and input size) overall observed performance. We discuss the performance of each model by comparing the mean performance, across pairs of kernels and input sizes, as a function of the number of samples evaluated.

7.4 Results

We compare the efficiency of our models for our two representations of the graph features and three neural network architectures. Each of the six models is evaluated one hundred times: five times for each slice of 5% of the data. We divide the dataset into twenty folds and used one fold for training. For each fold, we train five instances of each of the six models. Similarly one hundred random searches were used to generate a baseline comparison. For each case, we report the average across all these evaluations.

Figure 7.4 shows the results of these experiments. In this figure, the X-axis

represents the number of optimized kernels that have been evaluated. The X-axis uses a logarithmic scale. The Y-axis shows the average performance across all experiments so far. The performance corresponds to the average of the best optimizations across all kernels and input sizes. In all seven cases, the performance starts at approximately 55%, corresponding to the performance observed after evaluating the training set (5% of the search space).

7.4.1 Effect of Complex Neural Networks

For this problem, we observe that more complex neural networks tended to decrease the performance of the SVM ranking model. This is particularly true for the models using the spectrum features of the TileTree. This can be explained by the fact that complex models tend to overfit the training set. This is shown in Figure 7.5. We characterized overfitting using the ratio between testing and training errors. In this case, “error” refers to the cross-entropy between predicted and actual distribution. We refer the reader to Prechelt’s work which treats the issue of overfitting in neural networks and compares criteria for “early stopping” [Prechelt, 1998]. In Figure 7.5, we observe two trends. First, models based on TileTree spectrum features overfit much more than when the TileTree node features are used. Indeed, graph spectral features contain more information than the raw node features, and using 5% of the search space for training is not sufficient to permit the models to generalize. Second, models with more parameters tend to overfit more, which is a well known problem. As the number of parameters increases, neural networks tend to memorize the training set instead of generalizing over it. As can be seen in Figure 7.5, the convolutional network overfits less than MLPs because it uses significantly fewer parameters for the same depth of the neural networks.

7.4.2 Performance Milestone

One way of comparing iterative compilation methods is to compare the number of evaluations necessary to reach a given percentage of the best performance. In

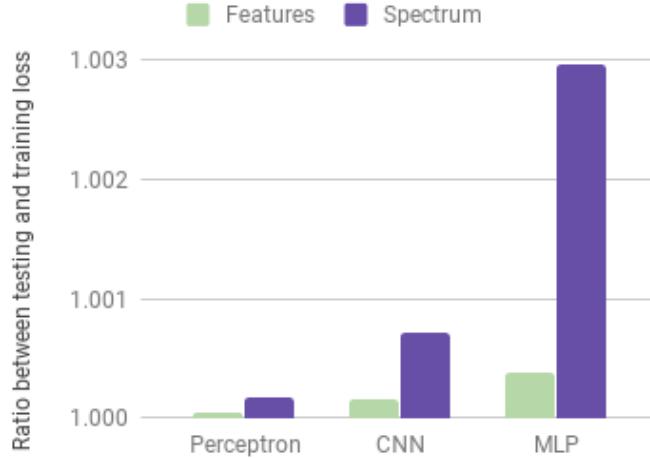


Figure 7.5: This bar chart shows the ratio between training and testing losses for the six neural networks. This ratio is an estimate of the amount of overfitting [Prechelt, 1998].

	Perceptron	Features MLP	CNN	Perceptron	Spectrum MLP	CNN
60%	191 (16.5x)	185 (17.0x)	185 (17.0x)	218 (14.5x)	248 (12.7x)	216 (14.6x)
70%	837 (16.1x)	775 (17.4x)	787 (17.1x)	691 (19.5x)	1072 (12.6x)	719 (18.7x)
80%	2199 (14.7x)	2135 (15.1x)	2151 (15.0x)	1236 (26.1x)	17435 (1.8x)	1362 (23.7x)
90%	4524 (16.1x)	4635 (15.7x)	4585 (15.8x)	11639 (6.2x)	60636 (1.2x)	22073 (3.3x)
95%	9692 (12.2x)	10053 (11.8x)	9811 (12.1x)	53746 (2.2x)	103526 (1.1x)	66855 (1.8x)
99%	49862 (3.9x)	55221 (3.5x)	52257 (3.7x)	125682 (1.6x)	157742 (1.2x)	132090 (1.5x)

Table 7.1: Number of evaluations needed to reach performance milestones. The speedup compared to random search is shown between parenthesis.

this section, we look at the number of evaluation needed to reach six milestones of performance: 60%, 70%, 80%, 90%, 95%, and 99%. Table 7.1 shows the number of evaluations and the speedup compared to random search for each of the six models. We highlighted the results corresponding to the best model for each milestone. We observe that, aside from the lowest milestone, perceptrons outperform more complex models.

In Figure 7.6, we compare the performance of iterative compilation using perceptron-based ranking models compared to random search. It shows that these perceptron-based models permit us to reach the 60%, 70%, and 80% milestones more than $10\times$ faster than when the search space is randomly sampled. Using the spectrum-based

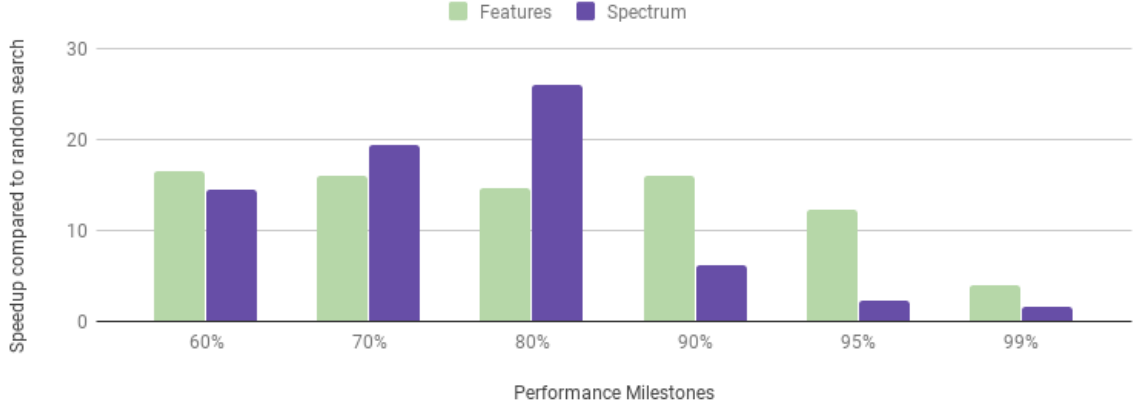


Figure 7.6: This bar chart shows how much faster the two perceptrons reach the different milestones compared to random search.

perceptron, the 80% milestone is reached more than $26.1\times$ faster, compared to $14.7\times$ faster for the node features. However, for the higher performance milestones, the node features outperform the spectrum features. For the final milestone, i.e. achieving 99% of the top performance available, the node features and spectrum features are $3.9\times$ and $1.5\times$ faster than random search, respectively.

7.5 Related Work

In this chapter, we evaluate the utilization of graph-based machine learning (ML) to accelerate iterative compilation (IC). In this section, we review the previous work on these topics.

Bodin et al, performed some of the earliest work on iterative compilation found in the literature [Bodin et al., 1998]. This research highlights the high cost of IC techniques because of the number of evaluations needed before a satisfactory solution was found. However, they showed the potential of the technique and argued for its usefulness when applied to embedded systems. Following this work, various platform independent approaches to iterative compilation were developed [Triantafyllis et al., 2003][Fursin et al., 2005].

In these early work, a major limitation of iterative compilation was the cost of evaluating a very large number of optimizations. Researchers tackled this issue through

the utilization of better search algorithms. Some early work in this area compared different strategies for iterative search using genetic algorithms, random sampling, and simulated annealing [Knijnenburg et al., 2003].

Agakov et al, introduced the utilization of predictive modeling as a way to accelerate iterative compilation [Agakov et al., 2006]. In this work, an independent distribution model and a Markov model were used to focus the iterative search in the optimization space. Further research introduced performance predictors based on static code features [Dubach et al., 2007] and dynamic performance counters [Cavazos et al., 2007]. The usage of performance counters to predict the performances of an optimization sequence was further refined in later work [Park et al., 2011]. This research introduced the idea of a tournament predictor, which was used to select the best of two optimization sequences.

Recent work has focused on the construction of optimization heuristics using machine learning techniques. One recent work explored the different way programs can be characterized, using static and dynamic program features [Li et al., 2014a]. Other work evaluated the use of source code features to solve the compiler phase ordering problem [Kulkarni, 2014].

One recent paper [Ashouri et al., 2017] performed research in optimization heuristics, predictive modeling, and iterative compilation in a framework the authors called MiCOMP. This framework uses these techniques to achieve an average performance speedup of $1.31\times$ over LLVM’s most aggressive optimization setting (-O3). Finally, Ogilvie et al, explored active learning and iterative compilation [Ogilvie et al., 2017]. Active learning leverages new samples to incrementally improve predictive modeling during the exploration of the optimization space.

7.6 Conclusion & Future Work

In this chapter, we compare two characterizations of graphs to accelerate iterative compilation. The first characterization only considers the features in the graph’s nodes, and the second characterization leverages graph spectral features to incorporate

the structure of the graphs. There are two interesting results: complex neural networks do not achieve the best results compared to simple neural networks, and the choice between the two TileTree representations is not clear cut.

Our first result regarding the poor performance of complex neural networks is explained by the tendency of large models to overfit, especially given the small size of our training set. We can envisage two ways to solve this problem, and both methods would help to reduce overfitting. First, we can use unsupervised pre-training to learn high-level features of the graph representations. This can be done on a very large part of the search space since it does not require us to evaluate the optimized kernels. Second, we can take advantage of the iterative nature of the search and train the ranking models continuously. In this latter case, a very small subset of the search space would be used to construct the initial version of the ranking model. During each iteration of the search, the model would be used to select the most promising samples to be evaluated. The results of these evaluations would provide targets used to extend the training set. The extended training set is then used to refine the ranking model. This refined ranking model is used to select the next batch of samples to evaluate. In Figure 7.7, we depict how these two techniques can be put together.

The second result concerns the comparison between the two characterization of TileTree. Determining which of these characterization is best depends on our goals. Models based on the spectrum features of the TileTree reach 80% of the best performance $1.8\times$ faster than models using the nodes features of the TileTree. It corresponds to evaluating only a little more than 1,200 samples, and 0.5% of the full search space. On the other hand, if one wants to reach 99% of the best performance, it is $2.5\times$ faster to use the model based on node features versus spectrum features. It requires evaluating approximatively 50,000 samples, which is 20% of the search space.

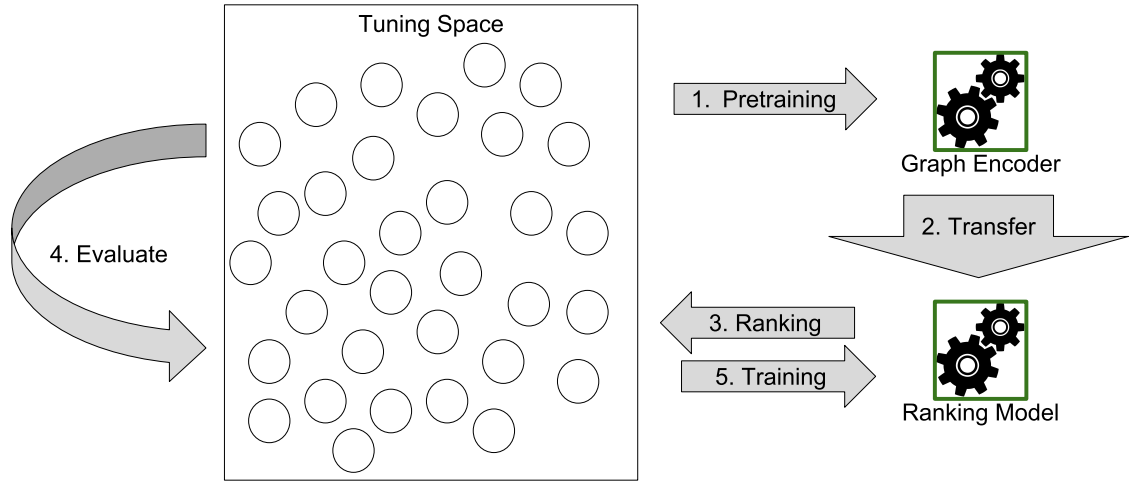


Figure 7.7: This figure shows an iterative compilation flow using unsupervised pretraining and continuous learning. **Pretraining**: Train a deep auto-encoder on TileTrees. **Transfer**: The deep auto-encoder is used to initialize the ranking model. **Ranking**: The ranking model is applied to the tuning space. **Evaluate**: Top ranked samples are evaluated. **Training**: All evaluated samples are used to training/refine the ranking model. **Loop**: Back to 3 until reaching a stop criteria.

Chapter 8

CONCLUSION

In this dissertation, we presented new techniques to characterize graph-structured data for use with neural networks. This work has two major parts: malware classification and computation kernel optimization. In the first part, we compare various malware characterization techniques. One of the novelties of this work is the construction of convolutional neural networks for diverse data types. We show that these convolutional architectures increase the accuracy of the resulting models. In the second part, we compare two methods to characterize a graph-based compiler intermediate representation (IR). For both methods, we build perceptrons and deep neural networks. We show that perceptrons outperform deep neural networks on this problem. To conclude this dissertation, we summarize our results and present insights gained while performing this work.

8.1 Results

In this research, we explored the construction of neural networks methods to ingest graph data. We have shown that it is possible to leverage these complex data structures both to improve malware detection and to accelerate the tuning of computation kernels.

In Chapter 4, we presented our early work on malware classification. In this work, a small and balanced dataset of malware was classified between eleven families of malware. On this dataset, our graph-based characterization was more effective than state-of-the-art characterization methods. Specifically, using a consensus of multilayer perceptrons, graph-based characterization achieved 92.0% accuracy compared to 86.2% for the state-of-the-art characterizations.

In Chapter 5, we expanded this malware classification work. We created an extended dataset and expanded our research with more characterizations of potentially malicious files, a feature engineering stage, and advanced neural network architectures. The extended dataset is more than twice as large as the previous dataset and contained eighteen classes that are not balanced. Using the state-of-the-art model, a multilayer perceptron, we achieved 62.3% accuracy. Using more advanced methods, including feature transformations and convolutional architectures, we constructed models with 79.3% accuracy. Finally, on this dataset, our graph-based characterization does not yield good models when used by themselves. However, models using the state-of-the-art features augmented with graph-based characterizations yield 83.2% accuracy in the malware family classification problem.

In Chapter 6, we presented TileK, an optimizing compiler where directives are used to drive parallelization. We used this compiler to generate many parallelized versions of four computation kernels. We then evaluated these versions for a large variety of inputs. We argue that this evaluation phase, while necessary, is prohibitively expensive. This is a common problem when dealing with optimizing compilers because they enable a large set of optimizations to be applied. Expert programmers can leverage such compilers if they have a thorough understanding of the optimization space and underlying hardware. We show that using machine learning can greatly assist the programmers when porting legacy applications to new hardware.

In Chapter 7, we propose a solution to reduce the time necessary to tune applications. We use iterative compilation to search the tuning space for the best optimizations. While basic iterative compilation traverses the tuning space randomly, it is now common to use predictive modeling to guide search toward the most promising optimizations. We created models that use our TileK IR to rank order optimized kernels in the tuning space based on their expected performance. Specifically, we compared two methods to present TileK IR to neural networks, either using the graph’s feature vectors or its graph spectral features (GSF). We show that both methods have their advantages. GSF models permit us to reach 80% of the best performance $26\times$ faster

than random search ($15\times$ faster than node features). Using node features that don't take into account the graph's topology enables us to reach 99% of the best performance $3.9\times$ faster than random search ($1.5\times$ faster than GSF).

8.2 Insights & Future Work

The work presented in this dissertation demonstrates various novel techniques to learn from graphs using neural networks. As we conclude this work, it is important to highlight remaining questions and possible improvements.

When we devised the extended dataset of Chapter 5, we imposed constraints on the graphs. Specifically, we required the graphs to have more than eight nodes and at least half as many edges as they have nodes. These criteria were determined empirically by evaluating datasets formed using various limits. To make this method more widely applicable, it is essential to determine precise constraints on applicable graphs. To this end, we need both theoretical and experimental analyses. There are many definitions of the Laplacian operator corresponding to different geometrical interpretations of graphs. Properly defining the domain of applicability of each would permit us to choose which one to use in a given situation.

After characterizing the limits of spectral methods, we need to envisage graph transformations. The goal of these transformations will be to increase the applicability of spectral methods. The shortest paths algorithm is used to build graph kernels for support vector machine. We want to evaluate the characterization capabilities of the spectrum of the shortest path graph.

As graph spectral methods will not always be applicable, other graph characterization methods should be envisaged. We are particularly interested in recursive neural networks which can be applied to the directed acyclic graphs (DAG) [Michele et al., 2004].

BIBLIOGRAPHY

- [Agakov et al., 2006] Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O’Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO ’06, pages 295–305, Washington, DC, USA. IEEE Computer Society.
- [Anderson et al., 2017] Anderson, H. S., Kharkar, A., Filar, B., and Roth, P. (2017). Evading machine learning malware detection. *Black Hat USA*.
- [Arandjelovic and Zisserman, 2012] Arandjelovic, R. and Zisserman, A. (2012). Three things everyone should know to improve object retrieval. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2911–2918.
- [Ashouri et al., 2017] Ashouri, A. H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., and Cavazos, J. (2017). Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions of Architecture and Code Optimizaion*, 14(3):29:1–29:28.
- [AWS, 2014] AWS (2014). Boto3 documentation. <http://boto3.readthedocs.io/en/latest/>. Accessed: 2017-1-24.
- [Bengio et al., 2007] Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., et al. (2007). Greedy layer-wise training of deep networks. *Journal of Advances in neural information processing systems*, 19:153.
- [Bergstra and Bengio, 2012] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.
- [Bodin et al., 1998] Bodin, F., Kisuki, T., Knijnenburg, P., O’Boyle, M., and Rohou, E. (1998). Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France.
- [Borgwardt and Kriegel, 2005] Borgwardt, K. M. and Kriegel, H. P. (2005). Shortest-path kernels on graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM’05)*.

- [Boser et al., 1992] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory, COLT '92*, pages 144–152, New York, NY, USA. ACM.
- [Bradley, 1997] Bradley, A. P. (1997). The use of the area under the roc curve in the evaluation of machine learning algorithms. *Journal of Pattern Recognition*, 30(7):1145–1159.
- [Breiman, 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.
- [Bronstein et al., 2016] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2016). Geometric deep learning: going beyond euclidean data. *CoRR*, abs/1611.08097.
- [Bruna et al., 2013] Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2013). Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203.
- [CAPS-e, 2012] CAPS-e (2012). https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPPCG_Directives_ReferenceManual.pdf.
- [Cavazos et al., 2007] Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O’Boyle, M. F. P., and Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO '07*, pages 185–197, Washington, DC, USA. IEEE Computer Society.
- [Chang and Lin, 2011] Chang, C.-C. and Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27.
- [Chen and Wang, 2008] Chen, L. and Wang, G. (2008). An efficient piecewise hashing method for computer forensics. In *Proceedings of First International Workshop on Knowledge Discovery and Data Mining*, pages 635–638. IEEE.
- [Chen et al., 2010] Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. (2010). Evaluating iterative optimization across 1000 datasets. *ACM SIGPLAN Notice*, 45(6):448–459.
- [Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Journal of Computational Science & Engineering*, 5(1):46–55.
- [Darema, 2001] Darema, F. (2001). The spmd model: Past, present and future. In Cotronis, Y. and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine*

and Message Passing Interface, volume 2131 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

- [Demmel et al., 2005] Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petit, A., Vuduc, R., Whaley, R. C., and Yelick, K. (2005). Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312.
- [Denny Britz, 2015] Denny Britz (2015). Understanding Convolutional Neural Networks for NLP (Blog). <http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp>. Accessed: 2016-12-23.
- [Dolbeau et al., 2013] Dolbeau, R., Bodin, F., and de Verdère, G. C. (2013). One openc1 to rule them all? In *2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*, pages 1–6.
- [Domhan et al., 2015] Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*.
- [Dubach et al., 2007] Dubach, C., Cavazos, J., Franke, B., Fursin, G., O’Boyle, M. F., and Temam, O. (2007). Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers, CF ’07*, pages 131–142, New York, NY, USA. ACM.
- [Eagle, 2008] Eagle, C. (2008). *The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA.
- [Fan and Lv, 2008] Fan, J. and Lv, J. (2008). Sure independence screening for ultrahigh dimensional feature space. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(5):849–911.
- [Fursin et al., 2005] Fursin, G. G., O’Boyle, M. F. P., and Knijnenburg, P. M. W. (2005). Evaluating iterative compilation. In Pugh, B. and Tseng, C.-W., editors, *Proceedings of 15th Workshop on Languages and Compilers for Parallel Computing*, pages 362–376, Berlin, Heidelberg. Springer Berlin Heidelberg.
- [Gärtner et al., 2003] Gärtner, T., Flach, P., and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pages 129–143. Springer.
- [Grigorevich and Lapa, 1966] Grigorevich, I. and Lapa, V. G. (1966). Cybernetic predicting devices. Technical report, DTIC Document.

- [Grossberg, 1972a] Grossberg, S. (1972a). Neural expectation: cerebellar and retinal analogs of cells fired by learnable or unlearned pattern classes. *Kybernetik*, 10(1):49–57.
- [Grossberg, 1972b] Grossberg, S. (1972b). A neural theory of punishment and avoidance, i: Qualitative theory. *Mathematical Biosciences*, 15(1):39 – 67.
- [Grossberg, 1972c] Grossberg, S. (1972c). A neural theory of punishment and avoidance, ii: quantitative theory. *Mathematical Biosciences*, 15(3):253 – 285.
- [Guarnieri et al., 2012] Guarnieri, C., Tanasi, A., Bremer, J., and Schloesser, M. (2012). The cuckoo sandbox.
- [He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.
- [Henaff et al., 2015] Henaff, M., Bruna, J., and LeCun, Y. (2015). Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163.
- [Hinton and Salakhutdinov, 2006] Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.
- [Ho, 1995] Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1.
- [Hornik, 1991] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Journal of Neural Networks*, 4(2):251–257.
- [Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Journal of Neural Networks*, 2(5):359–366.
- [Hubel and Wiesel, 1968] Hubel, D. H. and Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243.
- [Ilievski et al., 2016] Ilievski, I., Akhtar, T., Feng, J., and Shoemaker, C. A. (2016). Hyperparameter optimization of deep neural networks using non-probabilistic RBF surrogate model. *CoRR*, abs/1607.08316.
- [Ivakhnenko, 1971] Ivakhnenko, A. (1971). Polynomial theory of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics*, (4):364–378.
- [Kim, 2014] Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

- [Kinable, 2010] Kinable, J. (2010). Malware detection through call graphs. Master’s thesis, Institutt for telematikk.
- [Knijnenburg et al., 2003] Knijnenburg, P. M. W., Kisuki, T., and O’Boyle, M. F. P. (2003). Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67.
- [Kolter and Maloof, 2004] Kolter, J. Z. and Maloof, M. A. (2004). Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM.
- [Kornblum, 2006] Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Journal of Digital investigation*, 3:91–97.
- [Kramer and Sangiovanni-Vincentelli, 1988] Kramer, A. H. and Sangiovanni-Vincentelli, A. L. (1988). Efficient parallel learning algorithms for neural networks. In *Proceedings of The Conference on Neural Information Processing Systems*, pages 40–48.
- [Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the Advances in neural information processing systems*, pages 1097–1105.
- [Kulkarni, 2014] Kulkarni, S. (2014). *Improving compiler optimizations using machine learning*. PhD thesis, University of Delaware.
- [Li et al., 2014a] Li, F., Tang, F., and Shen, Y. (2014a). Feature mining for machine learning based compilation optimization. In *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 207–214.
- [Li et al., 2014b] Li, M., Zhang, T., Chen, Y., and Smola, A. J. (2014b). Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’14*, pages 661–670, New York, NY, USA. ACM.
- [Li et al., 2005] Li, W.-J., Wang, K., Stolfo, S. J., and Herzog, B. (2005). Fileprints: Identifying file types by n-gram analysis. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pages 64–71. IEEE.
- [Liao et al., 2013] Liao, C., Yan, Y., de Supinski, B. R., Quinlan, D. J., and Chapman, B. (2013). Early experiences with the openmp accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer.
- [Liu and Motoda, 1998] Liu, H. and Motoda, H. (1998). *Feature Extraction, Construction and Selection: A Data Mining Perspective*. Kluwer Academic Publishers, Norwell, MA, USA.

- [Liu et al., 2009] Liu, Y., Zhang, E. Z., and Shen, X. (2009). A cross-input adaptive framework for gpu program optimizations. In *Proceedings of IEEE International Symposium on Parallel Distributed Processing, 2009*.
- [Loshchilov and Hutter, 2016] Loshchilov, I. and Hutter, F. (2016). CMA-ES for hyperparameter optimization of deep neural networks. *CoRR*, abs/1604.07269.
- [Micheli et al., 2004] Micheli, A., Sona, D., and Sperduti, A. (2004). Contextual processing of structured data by recursive cascade correlation. *IEEE Transactions on Neural Networks*, 15(6):1396–1410.
- [Minsky and Papert, 1969] Minsky, M. L. and Papert, S. (1969). *Perceptions: An Introduction to Computational Geomry*. MIT press.
- [Muller et al., 2001] Muller, K. R., Mika, S., Ratsch, G., Tsuda, K., and Scholkopf, B. (2001). An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–201.
- [Munshi, 2008] Munshi, A. (2008). Opencl. *Parallel Computing on the GPU and CPU, SIGGRAPH*.
- [Nobre et al., 2016] Nobre, R., Reis, L., and Cardoso, J. M. (2016). Compiler phase ordering as an orthogonal approach for reducing energy consumption.
- [Ogilvie et al., 2017] Ogilvie, W. F., Petoumenos, P., Wang, Z., and Leather, H. (2017). Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 245–256.
- [OpenACC, 2011] OpenACC (2011). Openacc: Directives for accelerators. <http://www.openacc-standard.org/>.
- [Park et al., 2012] Park, E., Cavazos, J., and Alvarez, M. A. (2012). Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 196–206. ACM.
- [Park et al., 2011] Park, E., Kulkarni, S., and Cavazos, J. (2011). An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '11*, pages 65–74, New York, NY, USA. ACM.
- [Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

- [Perozzi et al., 2014] Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652.
- [Prechelt, 1998] Prechelt, L. (1998). Automatic early stopping using cross validation: quantifying the criteria. *Journal on Neural Networks*, 11(4):761–767.
- [Radare2, 2008] Radare2 (2008). Radare2. <http://www.radare.org>. Accessed: 2016-12-25.
- [ReversingLabs, 2015] ReversingLabs (2015). Reversinglabs hashing algorithm. <https://www.reversinglabs.com/technology/reversinglabs-hash-algorithm.html>. Accessed: 2016-12-25.
- [Reyes et al., 2012] Reyes, R., Lopez-Rodriguez, I., Fumero, J. J., and de Sande, F. (2012). accull: An openacc implementation with cuda and opencl support.
- [ROSE, 2017] ROSE (2017). Rose compiler infrastructure, user manual. http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf.
- [Rosenblatt, 1957] Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.
- [Rumelhart et al., 1988] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Journal of Cognitive modeling*, 5(3).
- [Saxe and Berlin, 2015] Saxe, J. and Berlin, K. (2015). Deep neural network based malware detection using two dimensional binary program features. In *Proceedings of 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20. IEEE.
- [Schultz et al., 2001] Schultz, M. G., Eskin, E., Zadok, F., and Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 38–49.
- [Seide et al., 2011] Seide, F., Li, G., Chen, X., and Yu, D. (2011). Feature engineering in context-dependent deep neural networks for conversational speech transcription. In *2011 IEEE Workshop on Automatic Speech Recognition Understanding*, pages 24–29.
- [Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.
- [Sharif Razavian et al., 2014] Sharif Razavian, A., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). Cnn features off-the-shelf: An astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

- [Srivastava et al., 2007] Srivastava, A., Zaïane, O. R., and Antonie, M.-L. (2007). Feature space enrichment by incorporation of implicit features for effective classification. In *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, pages 141–148. IEEE.
- [Strohmaier, 2013] Strohmaier, E. (2013). Highlights of the 42nd top500 list. SC13 BoF.
- [Tabuchi et al., 2014] Tabuchi, A., Nakao, M., and Sato, M. (2014). A source-to-source openacc compiler for cuda. In *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 178–187. Springer Berlin Heidelberg.
- [Theano, 2016] Theano (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.
- [Tian et al., 2014] Tian, X., Xu, R., and Chapman, B. (2014). Openuh: open source openacc compiler. *Proceedings of GPU Technology Conference*.
- [Triantafyllis et al., 2003] Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. I. (2003). Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215.
- [Vafaie and Jong, 1998] Vafaie, H. and Jong, K. D. (1998). Feature space transformation using genetic algorithms. *Proceedings of IEEE Intelligent Systems and their Applications*, 13(2):57–65.
- [Vanderbruggen and Cavazos, 2014] Vanderbruggen, T. and Cavazos, J. (2014). Generating opencl c kernels from openacc. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM.
- [Vanderbruggen et al., 2017] Vanderbruggen, T., Cavazos, J., Liao, C., and Quinlan, D. (2017). Directive-based tile abstraction to distribute loops on accelerators. In *Proceedings of the General Purpose GPUs*, pages 53–62. ACM.
- [Vincent et al., 2010] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408.
- [Vishwanathan et al., 2010] Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242.

- [Weber et al., 2002] Weber, M., Schmid, M., Schatz, M., and Geyer, D. (2002). A toolkit for detecting and analyzing malicious software. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 423–431. IEEE.
- [Wolfe, 1989] Wolfe, M. (1989). More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA. ACM.
- [Wolfe, 2015] Wolfe, M. (2015). http://www.openacc.org/sites/default/files/PGI-Wolfe-GTC-Notes_0.pdf.
- [Xinai, 2016] Xinai, X. (2016). Review on resampling algorithms for imbalanced data classification. *Journal of Residuals Science & Technology*, 13(5):138–1.
- [Xu et al., 2016] Xu, L., Zhang, D., Jayasena, N., and Cavazos, J. (2016). Hadm: Hybrid analysis for detection of malware. In *Proceedings of SAI Intelligent Systems Conference*.
- [Xue, 2000] Xue, J. (2000). *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA.
- [Zeng and Martinez, 2000] Zeng, X. and Martinez, T. R. (2000). Distribution-balanced stratified cross-validation for accuracy estimation. *Journal of Experimental & Theoretical Artificial Intelligence*, 12(1):1–12.
- [Zhang and Wallace, 2015] Zhang, Y. and Wallace, B. (2015). A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*.

Appendix A

MAGIC FRAMEWORK

Any sufficiently advanced technology
is indistinguishable from MAGIC.

Arthur C. Clarke

Machine learning is well known for its appetite for computation time and data storage. We designed a framework to manage some of the issues linked to the deployment of a machine learning infrastructure. Our goal is to construct a purely declarative framework that can be used both locally, in a cloud environment, or in a HPC environment. We created the MAGIC framework which interfaces the data with the learning algorithms. MAGIC can be used through a low-level command line interface, or through a Python API. However to provide a simple, non-programmatic interface, we created MAGIC's Wizard. At this point, MAGIC demonstrates the viability of our approach. However, the Wizard is limited to a visualization tool. In this appendix, we present the design of MAGIC, the interface of its Wizard, and discuss the important future milestone.

A.1 Design

The design of MAGIC is focused on two main constraints. First, it needs to collect large amount of data to form datasets. Second, it needs to make it easy to design and evaluate many models.

The first goal is realized through MAGIC dataset abstraction. It aggregates many samples to efficiently stream them to the learning algorithms. The second goal is realized through MAGIC session abstraction. It encapsulates training one instance of

a model described by the user. This abstraction authorizes the utilization of multiple machine learning backends, currently either Theano or Scikit-learn.

A.1.1 Data Storage

In MAGIC, data storage is split between *table* and *store*. Tables are hierarchical documents (equivalent to JSON) that are indexed using primary key (or partition key) and, optionally, a secondary key (or sort key). Currently, MAGIC supports DynamoDB Tables and a schema to use local file-systems (which can be applied to S3 buckets). Stores are simple, directory-based, storage. Currently, MAGIC supports S3 Buckets and local file-systems. MAGIC provides a high-level interface to work with both tables and stores. this permits MAGIC to store any python data-structures, compatible with JSON, into a *record* in a *table*. The *store* interface permits MAGIC to work with different types of data: numpy arrays, pickled objects, and archives. MAGIC always uses this interface to load or store data, it permits us to easily switch between local and Cloud environments.

A.1.2 Datasets

MAGIC’s dataset abstraction is designed to aggregate the features from millions of samples into *segments* that fit into the memory of a single compute node. These *segments* are organized into *groups*, which can themselves be grouped into higher level groups.

A.1.2.1 Feature Description

The first step to build a dataset with MAGIC is to describe the features used to characterize samples. This description tells MAGIC how to retrieve the data from wherever it is stored. It also provides MAGIC with the transformations that need to be applied to the features.

Currently, MAGIC supports two types of data: structured and tensor. Structured data corresponds to hierarchical documents and can currently be retrieved from JSON files (stored locally or in a S3 Bucket) or from a DynamoDB Table. Tensor data

corresponds to arrays of numerical data (integer, real, or complex) stored in numpy arrays. These can be stored locally or in S3 buckets and can be retrieved from archives. Listing A.1 shows an example of a structured feature description while Listing A.2 describes a tensor feature. The descriptions for both types of features have six fields in common. Three of these fields are common to all the objects MAGIC managed: **title**, **description**, and **metatags**. These fields are used to permit a user to search for objects using natural language. The other fields are:

- **format**: either *structured* or *tensor*
- **repo**: points either to: a *table* for *structured* features, or a *store* for a *tensor* feature
- **size**: size of the feature

```

{
  "title": "Bytes Features",
  "description": "Size in bytes, Entropy, Frequency for each bytes
    value (0 to 255).",
  "metatags": [ "bytes" , "size" , "entropy" , "frequency" ],
  "format": "structured",
  "repo" : "analysis",
  "size": 256,
  "fields": [
    { "path" : "metrics.bytes.size" },
    { "path" : "metrics.bytes.entropy" },
    { "path" : "metrics.bytes.histogram" , "size" : 256, "preproc" :
      [ "frequency:metrics.bytes.histogram" ] }
  ]
}

```

Listing A.1: Example of structured feature description. This feature set is extracted from the bytes-level file analysis. It extracts features from the *record* produced by this analysis. These include the size in bytes, the entropy of the file, and the bytes frequency.

In the case of a *structured* feature, Listing A.1, there is only one other field: **fields** (plus one optional). This is a list of descriptors used to retrieve fields from the structured document. It has a path toward the desired leaf of the document. As the path can point to a list, the expected size of this list can be provided. Additionally,

a preprocessing directive can be added. Finally, the optional field, *record*, corresponds to a secondary key to retrieve the record. The primary key is the samples identifier.

```

1 {
2   "title": "TileTree Spectrum",
3   "description": "GSF of a TileTree",
4   "metatags": [ "gsf" , "tiletree" ],
5   "format"   : "tensor",
6   "repo"     : "tilek",
7   "size"     : 720,
8   "shape"    : [ 12 , 30 ],
9   "dtype"    : "complex64",
10  "path"     : "spectrum",
11  "preproc"  : [ "ID" ]
12 }

```

Listing A.2: Example of tensor feature description.

In the case of a *tensor* feature, Listing A.2, there are four more fields (plus one optional). These include the shape and data type of the tensor. If the tensor obtained for an item is not the correct size, it will either be truncated or filled with zeros along each axis. The field *path* represents the path toward the stored tensor. This path is prefixed with the sample identifier to retrieve the array from the *store* (*repo* field). The optional field, *archive*, represents the path toward an archive. When present, the archive's path is prefixed by the sample identifier while the *path* field corresponds to the path in the archive.

A.1.2.2 Building a Dataset

At the level of the MAGIC API, building a dataset takes a few steps. These steps exist to permit the construction of complex group hierarchies and, in the future, continuous extension of the dataset. However, we do not intend our user to use this interface. We provide a simple script that implements the construction of a dataset based on a list of samples. The flow of this script is presented in Figure A.1.

Using this script, a user only has to provide a list of samples (with stratification targets), a list of features, and configuration such as the number of folds or segment size. The first step splits all samples into as many folds as needed while ensuring that they

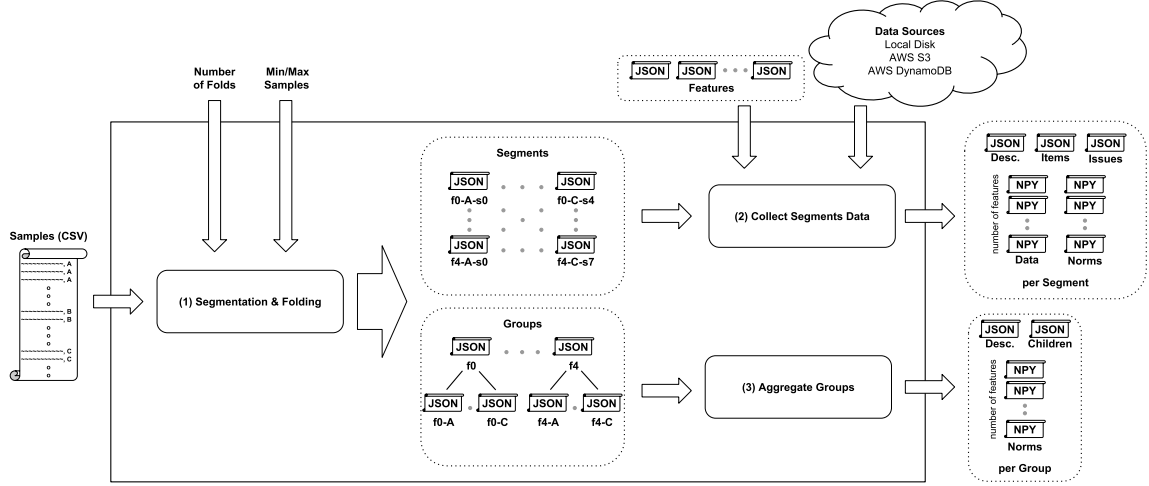


Figure A.1: MAGIC provides a helper script to create dataset. This script takes a list of items with stratification targets, a list of features, and a number of folds. (1) The stratification targets are used to separate the dataset into segments which are organized into groups. (2) MAGIC collects the features for each of the items in the segment and computes statistics about each segments. (3) MAGIC aggregates statistics for the groups recursively (starting with segments’ statistics).

contain the same number of samples from each stratification target. This step generates *segments* and *groups* descriptions (stored in JSON format). There is one group per fold. Each of these groups have one sub-group per stratification target. Finally, each group corresponding to one stratification target of one fold is made of one or more segments. The second step is the collection of the segments. This operation is done in parallel as each segment is independent. For each feature in the dataset, a numpy array is stored, it contains the features for all samples in the segment. Alongside this array, we also store statistics about the data. Finally, information about the segments are stored into structured records (JSON). These records include the description of each segment, lists of their samples, and the issues encountered while building them (such as missing samples or features). The third step accumulates the statistics from the segments composing the groups (recursively for groups composed of groups). These statistics are stored using numpy arrays. Groups are also associated with records including their descriptions and lists of children (groups or segments). The resulting data permits easy cross-validation experiments with properly constructed normalization.

A.1.3 Sessions

In MAGIC, a session corresponds to training one specific model on one specific dataset, using selected folds for training, validation, and testing. First, we explain how models are defined in MAGIC, providing details for our Theano backend. Second, we describe how to use MAGIC’s sessions to train and evaluate models.

A.1.3.1 Models

A model is describe by two documents: the black box description and the implementation. The black box description tells us about the ML backend to use, the type of model (supervized, online learning, probabilistic outputs), the input features, and the targets. The implementation describe the inside of the black box. For example, if the model is a neural network, it tells us about the number of layer and their sizes. At this point, MAGIC supports two ML backends: neural networks with Theano, and Scikit-learn (experimental).

A.1.3.1.1 Neural Networks with Theano

The main machine learning backend for MAGIC uses Theano. Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. We built this backend to permit us to experiment with uncommon models. This backend enables us to describe neural networks where layers inputs can be either feature sets, other layers outputs, or any combination of both. The only constraint is that the graph formed by these layers need to be acyclic, i.e. a DAG. Defining a model in MAGIC takes two steps.

First, one needs to provide the black-box description of the model, as in Listing [A.3](#). This description tells MAGIC what backend it should use, the *kind* field. It also tells whether the model support online training, if it is a supervised model, and if it can provide probabilities for classifications. These parameters are important as they inform us about the flow of training this model. This description also informs MAGIC about the inputs and, for supervised learning, the outputs.

```

2  {
   "title": "Flops Percentile from TileTree Spectrum",
   "description": "This model predicts the percentile of performance
4      reach by a a TileTree given its spectral representation.",
   "metatags": [ "DNN" , "tiletree spectrum" , "flop percentile" ],
   "kind": "MAGIC.DNN",
6   "online": "true",
   "supervized": "true",
8   "probabilities": "true",
   "inputs": [ "data.eigenvalues", "data.spectrum" ],
10  "outputs": {
    "data.flops-100": { "error": "misspred_int" }
12  }
}

```

Listing A.3: Example of “black box” description of a model for the Theano backend of MAGIC. As for any MAGIC object, it is given a title, description, and metatags. This describes a model trained with supervised learning using the *flops-100* features. It uses both the *eigenvalues* and *spectrum* of TileTrees to predict *probabilities*. As it is neural network, it is trained *online* meaning incrementally (epochs).

```

1  {
   "title": "MLP - 1 hidden",
3   "description": "Multiple layer perceptron with a single hidden
   layer of 1024 rectifying linear units.",
   "metatags": [ "mlp" , "relu" ],
5   "layers": [
     {
7       "tag": "L0",
       "activation": "relu",
9       "channels" : 1024,
       "in": [ "data.eigenvalues", "data.spectrum" ]
11      }, {
13       "tag": "L1",
       "activation": "softmax",
       "cost": "crossentropy_int",
15       "in": [ "layer.L0" ],
       "out": [ "data.flops-100" ]
17      }
   ]
19 }

```

Listing A.4: Example of implementation for a model using the Theano backend.

Second, we provide MAGIC with an implementation of the model. It describes the content of the black-box. The example in Listing [A.4](#) shows an implementation for the black-box defined in Listing [A.3](#). This implementation is a multiple layer perceptron with a single hidden layer of 1024 rectifying linear units. The output layer uses a softmax activation to produce a probability distribution and the error cost is measured using cross-entropy. The layer **L0** takes two feature sets producing 1024 values activating rectifying linear units. The layer **L1** takes the output of layer **L0** and produces a distribution of probabilities.

A.1.3.1.2 Scikit-Learn

The scikit-learn backend of MAGIC is not stable because advances in the neural network work has priority. However, using this backend, one can use any model from the scikit-learn python library. It includes a very large number of model for both supervised and unsupervised machine learning.

A.1.3.2 Usage

Evaluating the performances of a machine learning model requires well defined experimental conditions. The session object was created to encapsulate some of the complexity arising from it. Particularly, a session corresponds to one implementation of one model being trained for a specific split of one dataset. Here, *split* refers to the division between training, validation, and testing samples.

Figure [A.2](#) depicts the creation and training of a session in MAGIC. To create a session **(1)**, we need a dataset, a split (three lists of groups), and a model with its implementation. A description of the session (title, description, metatags, dataset, and model) and the hyper-parameters for the model are recorded. The hyper-parameter can either be provided by the implementation of the model, or upon creation of the session. Alongside these static information, MAGIC also initialize the model and its state. For example, the initial data of a neural network are the randomly initialized weights and biases of each layer. When the session runs **(2)**, it is controlled by policies which tells

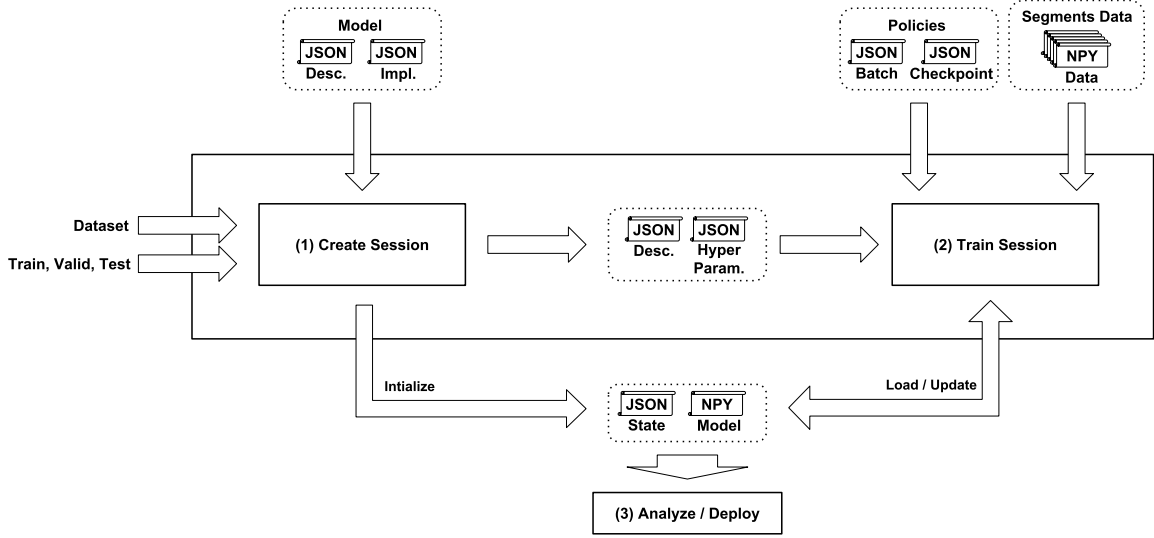


Figure A.2: We depict the creation and utilization of MAGIC’s Sessions. First, the session is created. It saves a description of the session and its hyper-parameters. It initializes an instance of the model and summary of its state. Second, the session is trained (potentially in multiple increments). During this training, the ML algorithm retrieves the data from the relevant segments of the dataset. As the model is trained, MAGIC triggers checkpoints when the model is evaluated and saved. The training is provided with policies which control the batches, stopping condition and checkpointing. Third, the trained model can be deployed, or the collected data can be used to evaluate the model.

it how to stream the dataset, when to checkpoint and/or evaluate the model, and what the stopping conditions are. The main goal of this setup is to authorize interruption and restart of the training. Finally **(3)**, the information collected during training and the saved models can either be used to evaluate the capability of the model or to deploy this model in production. In the case of online model, such as neural networks, we collect various metrics, such as the error-rate, which can be used to compare the convergence of different models.

A.2 WebUI

MAGIC exposes both a command line interface (CLI) and application programming interface (API). While these are sufficient to build datasets and work with sessions, we needed a tool leverage the large amount of data produced by MAGIC.

Especially, we wanted the capability to inspect datasets and sessions. For the dataset, it means checking the groups hierarchy and statistics of each groups or segments. It permits, for example, to know the distribution of each classification target in a group. For the session, there is much more information. When a session is running, it regularly checkpoints its state which include evaluating the model.

MAGIC’s Wizard is meant to provide a simple interface to access all this information. In the future, the Wizard will not only permit MAGIC’s users to create datasets and sessions but it will also help the user declare new feature sets and construct models.

A.3 Milestones

The MAGIC framework is at the stage of proof-of-concept. We identified several milestones to make it attractive to most machine learning practitioners.

A.3.1 Machine Learning Backends

Machine learning practitioners want to evaluate multiple ML algorithms when tackling a new problem. It makes it essential for MAGIC to handle multiple ML backend. Currently, MAGIC supports our Theano backend and provides an experimental connection to scikit-learn library. Our Theano backend is meant to explore the construction of complex neural network architectures. The computational performance of the resulting networks can be poor. Simultaneously, the scikit-learn backend give us access to many state-of-the-art implementation of ML algorithm but lack support for deep learning.

Reaching this milestone requires to fully validate the scikit-learn connection and provides a state-of-the-art deep learning backend. We propose to develop this backend using Keras, a high-level neural networks API.

A.3.2 Learning Patterns

The validation of machine learning models requires a careful examination of multiple instances of the same model trained with different splits of the dataset. There

is many different schema to do this validation, such as hold-out, leave-one-out, and cross-validation. Currently, MAGIC does not provide any abstractions to deal with these “learning patterns”. Their implementation is left to the user who have to create many sessions and analyze the result on a case-by-case basis.

We wish to provide an abstraction for these “learning patterns”. They are not limited to cross-validation and we identified a few more of these patterns when using MAGIC. The current list of pattern that we wish to implement in MAGIC is:

- hold-out validation
- leave-one-out validation
- (nested) cross-validation
- cascading models
- layer-wise pretraining (specific to neural networks)

Reaching this milestone only requires the implementation of the cross-validation pattern. It would introduce the necessary abstraction for learning patterns and simplify our own work. Indeed, we are heavily relying on cross-validation to build our models. As we start using more complex models, we are sure to identify more learning patterns. Hence, a subsequent milestone would address the creation of scripted learning-flows. This approach should permits any MAGIC user to define their own learning patterns.

A.3.3 HPC support

MAGIC’s design is meant to permit us to train a large number of model on a distributed system. Our focus has been on AWS Cloud services which provide “elastic” computation resources. However, many ML users have access to high performance computing clusters. We want to make sure that MAGIC users can leverage these systems.

Using HPC system is vastly different from the Cloud infrastructure. Particularly, we want to make sure that MAGIC properly uses the file-systems in these cluster. Another issues with HPC systems is the connection with the outside world. Especially, we want to be able to distribute ML workload between HPC and Cloud resources.

A.3.4 Control from Wizard

MAGIC’s Wizard is currently limited to be a visualization tool. However, it is intended to become the main interface for MAGIC users. Using the Wizard, one should be able to define feature sets and models, to construct dataset, and to instantiate learning-patterns.

In addition, the Wizard should be able to manage multiple computation resources. It should permit users to submit workloads (datasets construction, sessions training and analyses) on different systems. For example, AWS instances would be used to construct a dataset using data stored in AWS S3 Buckets. However, evaluating hundreds of models using cross-validation should be done on a HPC cluster.

Full control of MAGIC through the Wizard is a large project. The first milestone in this project includes:

- create, edit, and delete feature sets and models description
- construct a dataset (implementing Figure [A.1](#))
- create and submit sessions

For this milestone, we will focus on AWS Cloud, leaving the HPC integration for a later milestone.

A.3.5 Data Sources

At this point, MAGIC is limited to data stored in hierarchical documents or numpy arrays. In addition, they have to be either on local disk, AWS S3 Buckets, or AWS DynamoDB Tables. We wish to offer MAGIC user with more possible sources to inject their data. This problem can be split into two sub-problems: where does the data come from, and what is the data format.

First, we want to absorb data from any number of repositories. It includes retrieving data from various databases, such as SQL and MongoDB. We also need to integrate with other Cloud providers and their data-storage solutions.

Second, raw data are rarely amendable to ML. For example, when working on malware classification, the first step was to analyze millions of files. We used a distributed file analysis platform which applies many *analyzers* to a single file. We can integrate this characterization phase in MAGIC and permit users to instantiate their own analyzers. It would enable MAGIC users to implement their whole data ingestion pipeline in the construction of datasets.

A.3.6 Virtual Segmentation

The last milestone deals with segmentations. Currently, the segmentation of MAGIC dataset is rigid: once a segment is created, it cannot be changed. It is an issue for some use case, especially when it come to continuous learning for search. In this case, the samples used for learning depends on the model being trained. This is an important learning pattern. Indeed, search in large spaces can be improved by orders of magnitude with machine-learning. We want to introduce the notion of virtual segments. A virtual segment refers to items in an actual segment. Virtual segments are still immutable however they can be added and removed from the dataset at any time.

Appendix B

ROSE COMPILER

ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for C, C++, Fortran, and other languages. ROSE aims to be: (1) a library (and set of associated tools) to quickly and easily apply compiler techniques to your code in order to improve application performance and developer productivity, and (2) a research and development compiler infrastructure for one to write his own custom source-to-source translators to perform source code transformations, analyses, and optimizations. The approach of ROSE Compiler is:

- Cutting-edge research on source- and high-level compiler analysis and optimization algorithms.
- Best-practice software development to incorporate existing compiler techniques to and develop new ones.
- Pre-built ROSE tools to perform program transformation, analysis and optimization of your code.
- An easy-to-use API to help you to build your own customized, or domain-specific compiler-based analysis, transformation, and optimization tools.

B.1 Abstract Syntax Tree

When it comes to implementing programming models, working from compiler directives to instantiation of the runtime is an advantageous path. Compiler directives are easy to grasp for the user and can require very little changes to the original code. However from the compiler perspective it is more complex. Compiler directives can profoundly change the meaning of nearby language constructs. Consider the `for` loops of C/C++ distributed using OpenMP. It is this action on the languages they are embedded in that makes compiler directive so user friendly. However, most compilers work with intermediate representation (IR) that are far from the source-code.

ROSE is a source-to-source compiler, so its intermediate representation (IR) is as close to the source code as possible. It uses an Abstract Syntax Tree (AST) to

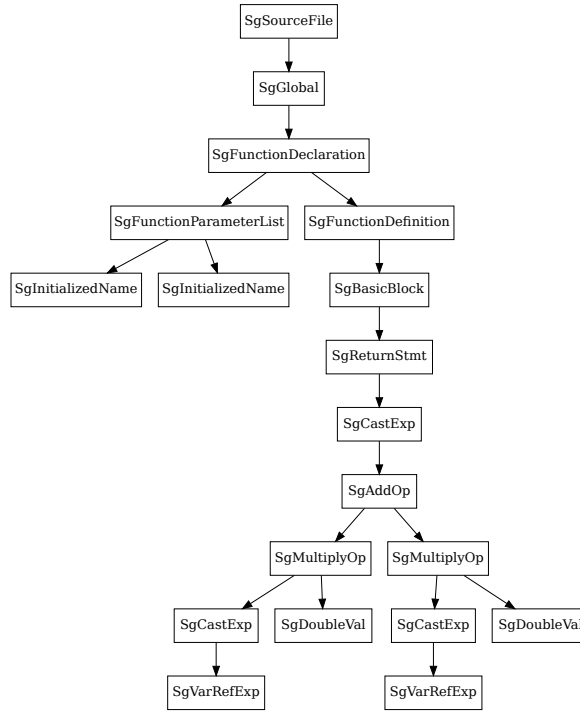
```

1  SgExpression * makeExpr(
   SgVariableSymbol * v1, SgVariableSymbol * v2, float c1, float c2
3  ) {
   return SageBuilder::buildAddOp(
5       SageBuilder::buildMultiplyOp(
           SageBuilder::buildVarRefExp(v1),
           SageBuilder::buildFloatValue(c1)
7       ),
       SageBuilder::buildMultiplyOp(
9           SageBuilder::buildVarRefExp(v1),
           SageBuilder::buildFloatValue(c1)
11          )
13     );
14 }
15
16 SgFunctionDeclaration * makeFunc(std::string function_name, float c1, float c2) {
17     SgScopeStatement * scope = SageBuilder::topScopeStack();
18     SgType * float_type = SageBuilder::buildFloatType();
19
20     // Build a parameter list for the function: "(float a, float b)"
21     SgFunctionParameterList * params = SageBuilder::buildFunctionParameterList(
           SageBuilder::buildInitializedName("a", float_type),
           SageBuilder::buildInitializedName("b", float_type)
23     );
24
25     SgInitializedName * init_name_a = params->get_args()[0];
26     SgVariableSymbol * sym_a = init_name_a->search_for_symbol_from_symbol_table();
27     SgInitializedName * init_name_b = params->get_args()[1];
28     SgVariableSymbol * sym_b = init_name_b->search_for_symbol_from_symbol_table();
29
30     // Build a defining function declaration: "float name(float a, float b) {}"
31     SgFunctionDeclaration * func_decl = SageBuilder::buildDefiningFunctionDeclaration(
           function_name, float_type, params, scope);
32
33     SgBasicBlock * func_body = func_decl->get_definition()->get_body();
34     SageBuilder::pushScopeStack(func_body);
35
36     SageInterface::appendStatement(
37         SageBuilder::buildReturnStmt(SageBuilder::buildExprStatement(
           makeExpr(sym_a, sym_b, c1, c2)
41         )),
         func_body
43     );
44
45     SageBuilder::popScopeStack();
46
47     return func_decl;
48 }
49
50 int main(int argc, char ** argv) {
51     SgProject * project = new SgProject(argc, argv);
52     SgSourceFile * file = isSgSourceFile(project->get_file(0));
53     SgGlobal * global_scope = file->get_global_scope();
54
55     SageBuilder::pushScopeStack(global_scope);
56
57     SgFunctionDeclaration * func_decl = makeFunc("my_func", 19.2, 5.79);
58     SageInterface::prependStatement(func_decl, global_scope);
59
60     SageBuilder::popScopeStack();
61
62     AstTests::runAllTests(project);
63     return backend(project);
64
65     return 0;
66 }

```

Listing B.1: Building an AST with ROSE Compiler. This adds a function to the file passed in argument. The function is `float my_func(float a, float b) { return a*19.2+ b*5.79; }`.

Figure B.1: AST from ROSE Compiler



represent the code. An AST is a tree representation of the abstract syntactic structure of source code. Each node of the tree denotes a construct of the programming language. Figure B.1 depicts the AST for the function `float my_func(float a, float b) { return a*19.2+b*5.79; }`. This figure is a representation of ROSE’s AST (Section B.2).

Interpreting compiler directives is much easier when working on an AST than with IRs that are closer to machine code, such as LLVM or GCC. Working in the context of an AST, accessing high-level information is easier. All the structures from C/C++ are easily accessible: scopes, loops, types, ...

While extracting information from an AST is relatively simple, transformation and generation of ASTs is much harder. Fortunately, ROSE Compiler provides an AST node *Factory*: `SageBuilder`. It provides a unified interface to build and assemble ASTs.

In Listing B.1 we present a small tool working on ROSE’s AST. It adds the function `float my_func(float a, float b) { return a*19.2+b*5.79; }` at the beginning of the first file passed in argument. The function `makeExpr` takes the symbols of two variables

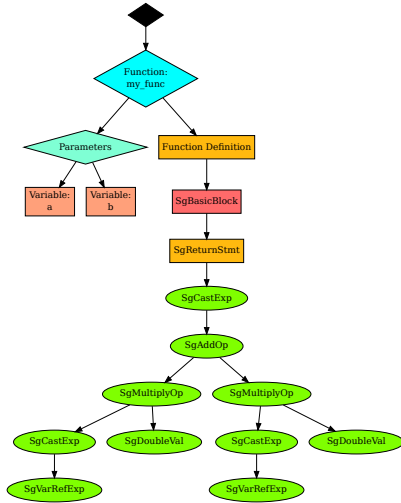
(`v1` and `v2`) and two floating point numbers (`c1` and `c2`). It builds an expression, for example `makeExpr(sym_a, sym_b, 19.2, 5.79)`, where `sym_a` and `sym_b` are the respective symbols of variable `a` and `b` would produce `a*19.2+b*5.79`. The function `makeFunc` takes a function name and two floating point numbers. It returns a function that computes the sum of its two arguments multiplied by the floating point numbers. The function `main` uses its arguments to build a ROSE project. It gets the first source file, then calls `makeFunc("my_func", 19.2, 5.79)`, and appends the resulting function to the file.

The function `makeExpr` illustrates two important points. First, building the AST of an expression in ROSE is extremely simple. `SageBuilder` provides many *build* functions that can be nested to build corresponding ASTs. The second point is the convenience of having symbols. Symbols are unique identifiers for objects (variable, function, class, namespace, ...). Function `makeFunc` shows that building more complex objects like a function definition requires more work. It becomes important to be aware of the scopes. Files, function and class definitions, namespaces, and block of codes (curly-braces `{ }`), are some of the C++ construct that create a scope. An entity only lives inside the scope where it has been declared. For example, `{ int i = 0 ; { int j = 3; } ; i = j; }` is invalid because the scope of `j` is limited to the inner block of code. Symbols associate one name and one declaration to one scope.

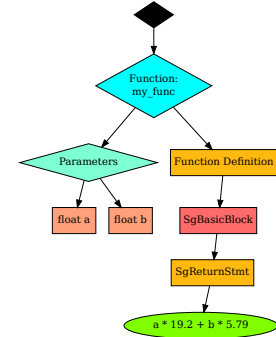
B.2 Visualization

Figure B.1 showed a raw AST from ROSE Compiler. This graph was generated using an AST visualization tool built using ROSE's AST traversal. ROSE's AST traversal implements a *Visitor* pattern toward the AST. The traversal visits each node and collects information. In the graph generator, we collect a representation of the AST for visualization. Using the graph generator, we can display the AST at different granularities, as shown in Figures B.2a and B.2b.

For Figure B.2a we used the default generator. This generator only changes the shape, color, and label of the nodes. For Figure B.2b we used a custom generator which stop generating the AST on the first expression it reaches. The label of the node is the



(a) We can add coloring and change the nodes shape.



(b) Expression trees can be reduced for clarity.

Figure B.2: For the same code than Figure B.1, we applied different filters.

unparsed expression. In addition this graph generator can skip some nodes, nodes can be placed in different clusters, and edges can be added to the graph.

Appendix C

PARSING COMPILER DIRECTIVES

Languages like C and C++ can be extended using compiler directives. In this appendix, we describe a module for ROSE Compiler which facilitate the creation and maintenance of such extensions.

C.1 Directive-based Language Extension

C.1.1 Directive Format

We observed that, in many annotation languages, the directives have the same format as OpenMP and OpenACC directives (Listing C.1).

```
#pragma acc parallel copy(a[0:n] [0:m])      #pragma omp parallel shared(a,b) private(i)
#pragma acc loop gang                        #pragma omp for schedule(static, chunk)
```

Listing C.1: Different Directives from OpenACC and OpenMP Languages.

C.1.1.1 Structure

The construction of these directives is simple. Directives are the concatenation of (1) the name of the *language*, (2) the *construct*, and (3) a list of *clauses*. Each component from the directives in Listing C.1 are presented in Table C.1.

	OpenACC		OpenMP	
Language	acc		omp	
Constructs	parallel	loop	parallel	for
Clauses	copy(a[0:n] [0:m])	gang	shared(a,b) private(i)	schedule(static, chunk)

Table C.1: The different components of the directives in Listing C.1.

C.1.1.2 Clause Arguments

Clauses can take arbitrary arguments. We can look at the `copy` clause in OpenACC. `arr[start:length]` is a *Cish* notation for array sections. For example, `arr[3:10]` is a section of `arr` made of the 10 elements from `arr[3]` to `arr[12]`. It can also be a more complex C or C++ expression, such as `my_stream.get_data()[my_stream.curr : Streamer::get_chunk_size()]`. Handling such expressions (access to fields and methods, arithmetic, types, ...) is essential to working with real C++ applications.

C.1.1.3 Relations between directive and AST nodes

Depending on the construct, the directives are attached to neighboring nodes from the AST. For example, OpenMP's loop construct (`#pragma omp loop`) is always attached to a for loop (the one directly following the directive), while OpenMP's parallel construct (`#pragma omp parallel`) can be attached either to a code block (`{}`) or a loop (through another directive). Some directives act as stand-alone statements. All directives are placed in a scope, and linked to the corresponding AST node. Clauses arguments, that are code snippets, are parsed in the context of this scope.

C.1.1.4 Relations between directives

Relations between directives can originate from the relative position of the directives in the AST. For example, OpenACC's loop construct (`#pragma acc loop`) always has a parent which is either another loop construct or a parallel construct. In other cases, the relation between directives originates from the clauses. OpenACC's parallel and kernel constructs (`#pragma acc parallel` and `#pragma acc kernel`) can be related through the clauses `async` and `wait`. `async` assigns an integer to the construct it applies to, and `wait` takes a list of integers. These clauses are used to define dependencies between different parallel regions when they executed asynchronously.

C.1.1.5 How to Parse Directives

Compiler directives are a common way to provide information or trigger analysis and transformation. However, there are no standard way to define their grammars. It

means that parsing directives is done on a case-by-case basis. Often, it means using a parser generator such as bison or flex.

Using two existing set of directives, we defined rules that can be used to construct such directives. We used these rule to build a tool for ROSE that enables ROSE users to easily define new sets of directives. Our goal is to make it much easier to design and implement extensions for C and C++ in ROSE Compiler.

C.2 Generic Parser

The DLX tool is a template library for ROSE Compiler. The template can be specialized for different languages. Given an AST, DLX collects all the directives from the target language, and it parses the associated string. The parsing determines the construct, and it produces a list of clauses (with their arguments). Then, depending on the construct, DLX associates nodes from the AST to the directive. When all the directives have been processed, the relation between directive is determined. DLX's output is a graph where the nodes are directives and the edges are the relations between these directives. To instantiate a frontend, DLX needs the following information:

- list of constructs and clauses
- neighbor AST nodes of constructs
 - container in the IR
 - retrieving from the AST
- list of parameters for every kind of clauses
 - container in the IR
 - parsing from the directive
- relation between directives: building graph from list of directives

C.2.1 Implementation of DLX

C.2.1.1 Class and Method Factory

DLX uses a template pattern to build its Internal Representation (IR) and the Frontend. The simplest expression of this pattern is presented in Listing [C.2](#). It is made of three parts: the template (Listing [C.2a](#)), the specialization (Listing [C.2c](#)), and the

```

1 template <class S>
2 struct generator_t {
3
4     template <enum S::list_e kind>
5     void foo_();
6
7     void foo(enum S::list_e kind);
8 };

```

(a) Template

```

1 int main() {
2     generator_t<seed_t> generator;
3
4     generator.foo(seed_t::e_0); // display "e_0"
5     generator.foo(seed_t::e_1); // display "e_1"
6
7     return 0;
8 }

```

(b) Main

```

1 #include "generator.hpp"
2
3 struct seed_t {
4     enum list_e { e_0, e_1 };
5 };
6
7 template <>
8 template <>
9 void generator_t<seed_t>::
10     foo_<seed_t::e_0>();
11
12 template <>
13 template <>
14 void generator_t<seed_t>::
15     foo_<seed_t::e_1>();
16
17 template <>
18 void generator_t<seed_t>::foo(
19     enum seed_t::list_e kind
20 );

```

(c) Specialization

```

1 template <>
2 template <>
3 void generator_t<seed_t>::foo_<seed_t::e_0>() {
4     std::cout << "e_0" << std::endl;
5 }
6
7 template <>
8 template <>
9 void generator_t<seed_t>::foo_<seed_t::e_1>() {
10     std::cout << "e_1" << std::endl;
11 }
12
13 template <>
14 void generator_t<seed_t>::foo(
15     enum seed_t::list_e kind) {
16     switch (kind) {
17         case seed_t::e_0: foo_<seed_t::e_0>(); break;
18         case seed_t::e_1: foo_<seed_t::e_1>(); break;
19     }
20 }

```

(d) Implementation

Listing C.2: The template pattern enables to define a repetitive interface based on an enumeration.

implementation (Listing C.2d). The template is called `generator` and the specialization defines a *seed*. The seed is a class defining an enumeration. The generator takes the seed as template argument. The generator defines a template object (either a method or sub-class) which takes an enumeration from the seed as a template parameter.

This pattern is useful as it enables us to organize similar functions or classes based on an enumeration. On the other side, it is verbose and it requires us to provide additional code to resolve the template at runtime (`switch` statement in Listing C.2d).

C.2.1.2 Pattern in DLX

The description of DLX's compiler directives starts with two lists: constructs and clauses. For each element of these lists, we provide one data-structure and some

methods. Each type of construct is associated to different nodes from the AST. Each type of clause has different parameters. The data-structure enables storing either information, forming DLX Internal Representation (IR). The functions are used to analyze the neighborhood of the directives in the AST and parse the arguments of the clauses.

In DLX, the Frontend and the IR are constructed using a *seed* representing the language. This class defines two enumerations: one for the constructs and one for the clauses.

C.2.1.3 Automation

Because it uses this pattern, implementing compiler directives using DLX is simple but repetitive. The template pattern enables a procedural method to implement the language. However, the process would be improved by some automation. We are developing a Domain Specific Language (DSL) to describe DLX language. A program in this DSL will produce the *seed* defining the language, the constructs' and clauses' IR nodes, the parser for some clauses' parameters, and empty definitions for the remaining functions.