# A Survey on Machine Learning for Compiler Optimization

Ruchi Bhalani
*Department of Computer Science*
*University of Texas at Austin*
ruchi.bhalani@utexas.edu

Jeffrey Gordon
*Department of Computer Science*
*University of Texas at Austin*
tgordon1052@utexas.edu

Marco Ravelo
*Department of Computer Science*
*University of Texas at Austin*
marco.ravelo@utexas.edu

***Abstract*—Machine Learning as a subject has been around since its inception during the 1950s and for decades remained a niche subfield for AI enthusiasts. Over the past two decades, this field has grown exponentially, finding influence in nearly every other subfield within computer science. Research in compiler optimization is no exception to this, as major advancements over the past few years can be attributed to machine learning-based approaches. In this report, we describe various applications of machine learning in compiler optimization as well as a wide range of research areas in the field. This is by no means a complete retrospective on machine learning-based compilation. Instead, we aim to highlight the main concepts and introduce specific examples in order to provide an approachable introduction to this topic.**

## I. Introduction

Before delving into the merging of compiler research and machine learning approaches, we will first introduce the two subjects independently to better understand how their combined application came to be.

### *A. A Brief History of Compilers*

The first electronic computers, developed during the 1940s, were exclusively operated using machine code. At the time, there was no established concept of transforming programs into a lower-level representation [1]. Instead, programmers worked exclusively with binary to communicate with the machines. As computers became more powerful and their memory capacity increased, programmers could work in assembly languages, a slightly better alternative to binary. It was not until the 1950s that programmers developed the first practical compilers. Grace Hopper, an American computer scientist and mathematician, was the first person to implement a working compiler for the UNIVAC I. She termed the invention the A-0 system and referred to it as a 'loader' or 'linker' instead of the modern term 'compiler'. Soon after, Alick Glennie, a British computer scientist, developed the first modern compiler for the Manchester Mark 1.

As with most advancements in computer science, each new iteration builds upon its previous successors. This concept is extremely evident in compiler development, as new compilers allow programmers to construct ever more complex and abstract compilers [1]. Working off of the first compilers, the FORTRAN team constructed the first complete compiler in 1957. COBOL was introduced as a programming language able to be compiled on multiple architectures. Self-hosting compilers, in which a compiler is written in the programming language it compiles, started to become commonplace in the 1960s. Parsers were soon introduced in compilers, allowing for higher-level languages to be created. This then lead to multi-pass compilers, in which a program would be read multiple times in order to better optimize the translated code. Unsurprisingly, this optimization process proved to be an extremely complex and wide-range research area and continues to be at the forefront of compiler research today.

### *B. A Brief History of Machine Learning*

The concept of machine learning has its origins in the field of artificial intelligence [2]. The term 'machine learning', coined by IBM employee Arthur Samuel in 1952, was defined as "a machine able to self-teach and learn over time". He developed a program that was able to play checkers, able to choose the next move in the game using alpha-beta pruning, a type of minimax strategy. Soon after, Frank Rosenvlatt, a researcher at Cornell, created a program called the perceptron designed for image recognition. The 1960s introduced concepts such as feed-forward neural networks and backpropagation, though the limited computational processing at the time prevented the full realization of these inventions. Due to frustrations and lack of support, the field of machine learning struggled, causing a rift between artificial intelligence and machine learning.

The 1990s, however, saw the field of machine learning flourish with the growth of the internet and ever more resources for data collection [2]. The start of the millennium brought with it a new rush of research and applications in machine learning, with concepts such as 'big data' and 'deep learning' becoming widely regarded in the mainstream. Additionally, the advancements in parallel computing skyrocketed the performance of machine learning systems. Modern machine learning systems seem to miraculously outperform each other as new advancements are developed continuously. Today, machine learning is used in nearly every aspect of computer science and has found itself at the forefront of our digitized society.

### *C. Machine Learning in Compiler Optimization*

The fields of compiler optimization and machine learning, despite originating around the same time in the 1950s, did not

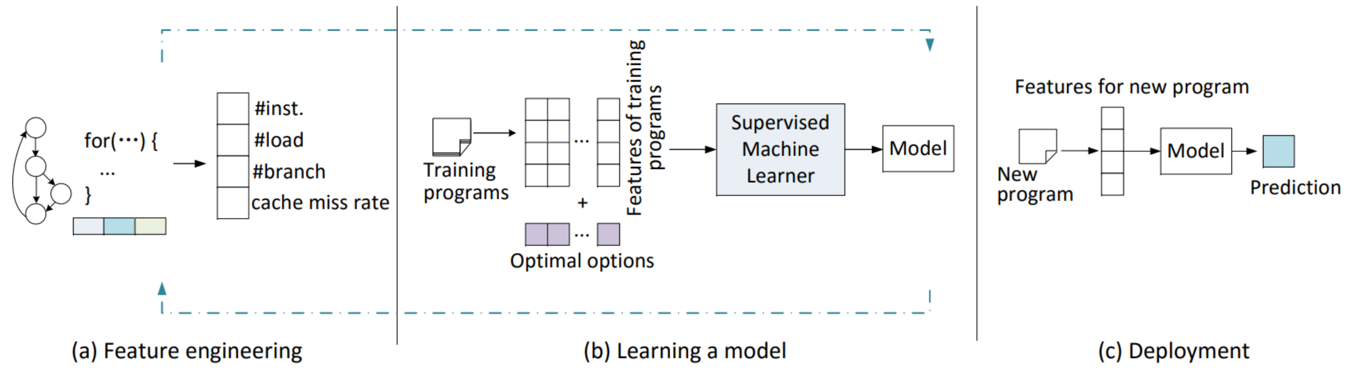


Fig. 1: A generic view of supervised machine learning in compilers [3]

have popular support for any kind of union between the two [3]. Machine learning is regarded as unpredictable, with results that are uncertain to work as programmers had intended. Meanwhile, compilers are required to produce replicable and provably correct outputs in order for programs to be translated correctly and without mistakes. "Why would anyone want to use machine learning to build a compiler?", is a sentiment expressed by many computer scientists throughout the late 1900s and early 2000s. And in truth, this point of view is valid, as these two fields seem to contradict one another in regard to their overarching goals. Machine learning aims to predict new outcomes based on previous data, while compilation meticulously transforms programs based on a set of defined rules and algorithms. Any deviation from this could result in inefficient or non-functional code.

Modern compiler research is primarily spent on optimization [3]. Given any source program written in a high-level language, what are the best transformations which can be applied to this program to produce the most efficient and best-performing code? This is at the heart of compiler optimization. Choosing these transformations is an extremely challenging task, a job historically left to expert compiler writers. This task has become ever more difficult due to the rise of multi-core architectures and increasingly complex computer hardware. Modern architectures are evolving so rapidly that compiler writers are unable to keep up. This is where machine learning can be utilized effectively.

Machine learning has the appealing property of being automatic, able to replace the difficult work of compiler writers [3]. It is suited for making any code optimization where performance can be measured for any underlying platform. It can be used for selecting the best compiler flags to determine how to map parallelism to processors (more detailed examples will be provided in later sections). The next section provides an overview of how machine learning is applied to compiler optimization systems.

## II. Overview of Machine Learning in Compilers

We know now that compiler optimization can be improved using machine learning methods, but how exactly are we able to do this? This section covers an overview of how machine learning is used to build models used within a compiler, able to make decisions for any program the compiler may encounter.

There are three stages in constructing a machine learning model: feature engineering, learning (training), and deployment. The first stage involves gathering data (otherwise known as features) to be used in the second stage, which involves training a model to perform a certain useful prediction task. The third stage takes a trained model and inserts it into a compiler to be used to decide better optimization decisions for new programs unseen by the compiler. Figure 1 provides a visualization of this process. The following subsections will go more in-depth into these three stages.

### A. Feature Engineering

In order for a model to learn anything useful from a program, we must extract a set of quantifiable properties, known as features, from the program. These features must be representative of the program and provide valuable information in regard to how the program can be optimized. These may include static features, tree and graph-based features, dynamic profiling information, reaction-based features, or any combination of these. In this section, we detail each type of feature and describe some examples that are used in practice:

*1) Static code features:* This data is gathered directly from the source code or from an intermediate representation of the program. Some examples of static code features include arithmetic operations like integer/floating point instructions and method calls, memory operations like load and store instructions, loop information such as the number of loops, loop depth, and loop type, and parallel information such as work threads, or workgroup size. Static code features have the advantage of being readily available from any given program, though in practice they are used in combination with other types of features.

*2) Tree and graph-based features:* Though not as readily used, tree and graph-based features provide a unique approach to extracting valuable information from a program. In this method, nodes in a tree or graph can represent various things, such as spatial-based information (how instructions are distributed in a program), or the order of optimization passes within a compiler.

*3) Dynamic features:* This data is extracted from multiple layers of the runtime environment. This differs from static code features because the program must be running as the data is collected from it. At the application level, we can extract information such as loop iteration counts and frequently executed code regions. From the operating system level, we can observe memory use, I/O use, and thread contentions. From the hardware level, we'll find information such as cache loads/stores and branch misses.

*4) Reaction-based features:* These features are extracted by profiling a program while using specific compiler options to observe runtime changes. The program "reacts" to these changes, allowing for information to be gathered in order to predict the best available compiler optimization.

Feature extraction and selection is a process known as feature engineering. It is a crucial first step in developing a good model for compiler optimization, as high-quality data directly leads to a high-quality model.

### B. Training a Model

Once sufficient information is gathered, we use this data to train a model to recognize patterns in programs to better predict their optimizations. This is done using a learning algorithm. Using the features selected during feature engineering, training examples are generated by compiling a program using various optimization options, leading to discovering the best-performing options. These examples are then fed into a learning algorithm, of which there are a vast number of options. Some of these options include supervised learning techniques such as regression or classification, as well as unsupervised learning techniques, such as clustering. We will go into greater detail on these approaches in a later section. Once training is complete, we have a working model that is ready to be inserted into a compiler.

### C. Deployment

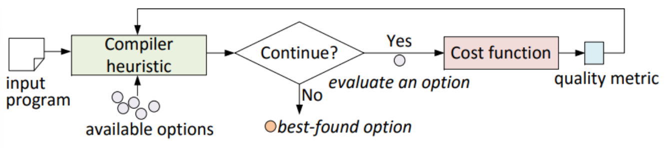
This final stage involves attaching our newly trained model to a compiler, which is able to predict optimizations for programs the compiler will encounter. As was done for the training data, features must be extracted from new programs and fed into the model, after which it predicts an optimization that the compiler performs.

A clear advantage to this machine learning approach is that this process is extremely streamlined and repeatable. We are able to replicate this for any new architectures that are created and quickly implement new efficient compilers targeted towards this hardware. Additionally, the constructed models are built from experimental data and results, so they are hence, evidence-based.

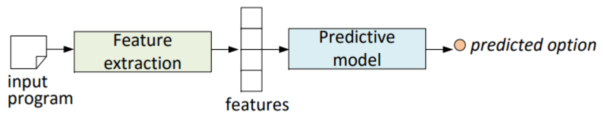
## III. Methodology

An ongoing challenge in compilation is selecting the right code transformation, given a program, since this requires an effective evaluation of the quality of each possible compilation option [3]. The most naive approach to arrive at the optimal transformation is an exhaustive search: applying every legal transformation and then profiling the program in order to collect a performance metric. However, this is infeasible due to the fact that compiler problems normally have a massive number of options. This search-based approach for optimization is known as iterative compilation [4] or autotuning [5], which we discuss in further detail in section IV.

There have been many techniques proposed to decrease the cost of searching such a large space, which we will discuss in further depth in section IV. However, the main problem remains: iterative compilation only finds a good optimization for an individual program and does not generalize into a viable compiler heuristic [3]. Therefore, to overcome this limitation, there are two main approaches for solving the problem of selecting compiler options that can (1) work across programs, and (2) are scalable. The first strategy is to develop a *cost function* to estimate the quality of a potential compiler decision, while the second is to directly predict the best-performing option. An overview of both these options is given in Figure 2 and we describe them in more detail in the following subsections.



(a) Use a cost function to guide compiler decisions



(b) Use a model to directly predict the decision

Fig. 2: High-level overview of the two ways to determine the optimal compiler transformation using machine learning [3]. These are described in more detail in section III.

### A. Cost Functions

Since their inception, cost functions were usually handcrafted by compiler writers for each and every architecture. This usually was not an issue, as there was an abundance of compiler programmers compared to the types of available architectures. However, as advancements in computer hardware were made, more and more architectures are being created, increasing each passing year. To combat this issue, many compiler writers have used a "one-size-fits-all" cost function for multiple similar architectures. This is extremely inefficient, as it can deliver poor performances if not done correctly.

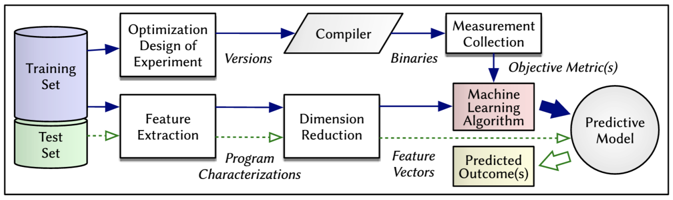
Instead of relying on hand-crafted cost functions, machine learning methods can be applied to tune these functions for better performance or power consumption for any given hardware. Due to the importance of cost functions in compiler optimization, researchers have developed various ways to automate the process of creating them, a term dubbed autotuning.

### B. Using a Model to Directly Predict Decision

Instead of creating cost functions to evaluate the quality of optimization options. Endeavors have been made to skip this entirely and directly predict the decision for a relatively small number of compilation problems. For sequential compiler problems, there is extensive work with predicting compiler flags, code transformation options, and tile size for loops. Unsurprisingly, predicting the optimal option for parallel programs is substantially more difficult due to complex interactions between the programs and parallel architectures. Despite this, there has been some interest in developing these sorts of prediction systems for a handful of optimization problems.

## IV. Autotuning

Autotuning is a methodology in which some model (algorithm, heuristic, etc.) infers/predicts one or more objectives with minimal human interaction. The paper that we utilized in order to gain deeper insight into autotuning is [6]. An autotuning framework is shown below for insight and understanding.



Modern auto tuning involves machine learning. There is an optimization space, in which the outcome is initially at some point, then over time through training, the outcome goes to the optimum. The problem, however, is that the overall search space is large and infeasible to cover every possibility. So, techniques have been explored in order to reduce the search space but maintain optimality. Different machine learning models and prediction types are explored in the next section, which aims to improve the efficiency of auto tuning.

## V. Machine Learning Models

In the realm of compiler optimization, there exists an abundance of machine learning models that can be employed to achieve optimal results. This section delves into some of the most commonly used models, shedding light on their respective characteristics and evaluations.

There are three primary categories of learning: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning involves the model being provided with predetermined outputs, known as targets. Each feature vector, along with its corresponding target, is fed into the model. Through exposure to numerous such examples, the model gradually learns to predict the target of a feature vector. Unsupervised learning, on the other hand, removes the notion of targets from the equation. The model endeavors to identify latent patterns within the feature space and subsequently cluster examples accordingly. When confronted with an unseen sample, it can predict the group to which it belongs. Lastly, reinforcement learning constitutes a machine learning training methodology that operates by rewarding desirable behaviors and penalizing undesirable ones. The model is conceptualized as an ”agent” tasked with navigating an environment and accumulating rewards based on its actions. The overarching objective is to maximize long-term reward, analogous to a person striving to survive in the wilderness for as long as possible. Actions in this context could encompass sleeping, eating, or traversing specific directions. As the person’s actions unfold, the state of the environment changes, such as ingesting a poisonous berry or encountering a bear. Consequently, rewards or penalties are assigned based on the combination of actions and states.

Each of these approaches possesses its own set of advantages and disadvantages concerning compiler optimization. In the subsequent section, we delve into an in-depth examination of each methodology.

### A. Supervised Learning Models

*1) Bayesian Networks:* A Bayesian Network is a machine learning technique in which a directed, acyclic graph is created. The nodes of the graph represent the features of data, and the edges represent the ”influence” of one feature on another. For example, if we have a binary feature A and binary feature B, and the value of A is found to have a correlation with the value of B, a directed edge from A to B would be created. Once this network is created, it can be used for inference and classification. Given a new, unseen sample, the network calculates the posterior probability of it being a part of a class, based on the observed values of its features. This is achieved using the Bayesian inference algorithm, which combines the prior probabilities with the likelihood of the observed evidence to calculate the posterior probabilities. A Bayesian network used to predict the optimal set of compiler optimizations based on given features was proposed by [7].

*2) Linear Models and SVMs:* Linear models, such as linear regression, learn a linear function that approximates the target output. These models are exceptionally useful as they are stable and don’t fluctuate heavily based on training set changes. SVM (support vector machines) learn to essentially create hyperplanes in the feature space, classifying different data points based on the region that is in the feature space. SVMs can be converted to non-linear by using the kernel trick, which involved putting the feature vector through a non-linear function. This expands the feature space dimensions, allowing for more complex patterns to be found. To use SVMs for regression, a hyperplane is found that fits the datapoints, instead of splitting the datapoints into different regions. [8] used an SVM model to autotune their JIT compiler.

*3) Decision Trees and Random Forests:* A decision tree is a model that builds a tree in which you can traverse based on certain feature values of a given sample. In the end, the sample is classified or a value is predicted. At each level of the tree, the model finds the feature/feature value with the most discriminatory power (most influence). The graph is built by doing this continuously until the specified or optimal depth is reached. New samples are classified or predicted by now traversing this tree. Random forests expand on this idea, using multiple trees and adding stochasticity to the model. The training set is split into multiple subsets of training data, at random and using replacement. This process is called bootstrap. Now, a tree is trained per subset of training data. So by the end, each tree will be slightly different. This allows to model to fit better instead of possibly overfitting, and adds robustness to the model. [9] used decision trees to separate IR representation of code into streams that compress better.

*4) Graph Kernels:* Graph kernels reduce the dimensions of the feature space while preserving the information in the data. This model is particularly useful because the input doesn't need to be expressed as feature vectors. The others of [10] fed the similarity matrix of programs to a graph kernel, then used the low dimensional representation in an SVM.

### *B. Unsupervised Learning Models*

*1) Clustering Methods:* As stated earlier, clustering groups datapoints into groups based on patterns within the data. In compiler optimization, clustering methods are used to group unrelated compiler passes that correspond to each other, or should follow each other in the same sequence. By doing this, the compiler optimization space can be reduced, narrowing our search for the optimal set of optimizations. The authors of [11] created the Gustafson Kessel algorithm, which clustered after dimensionality reduction and proved to reduce the training time of ML autotuning.

*2) Evolutionary Algorithms:* Evolutionary models mimic the process of evolution in biology. First, an initial set of candidates are selected. In the case of compiler optimizations, these could be sets of optimizations. Then, the sets are put through a fitness function, which measures how these optimization settings would affect the speedup of a program. Then the top-performing sets are chosen and go through crossover and mutation (different settings from the top sets are swapped, and some are changed at random). Then, the process restarts with these new sets. This occurs until a terminal condition is met. The genetic algorithm is a type of evolutionary algorithm in which the data is represented as binary strings but goes through the same general process. NEAT is another model that evolves neural networks. These models are extremely robust, generally easy to initiate and work well in the compiler optimization space.

### *C. Reinforcement Learning*

Reinforcement models use the Markov Decision Process (MDP) to navigate through the environment. [12] used RL in order to find the optimal order of scheduling blocks of code to run. This scheduling has significant impacts on overall runtime.

### *D. Prediction Types*

*1) Predicting the Optimal Set of Optimizations:* Prediction models are commonly used to output the estimated optimal set of optimizations given a program as input. The configuration of the model is customizable, as it can be trained to produce a set of optimizations that decrease runtime, decrease code size, or even energy consumption. Or a mix of these objectives. This type of prediction has been the most attractive in recent research [6].

*2) Predicting the Speedup of a Given Optimization:* Models can also be used to output the predicted speedup given a set of optimizations and a program. This is beneficial because the program does not actually have to be optimized, compiled, and ran. The input is (F,T) which corresponds to the program features and the compiler settings, and the output is the predicted speedup S. [13] developed a model that does speedup prediction, using source code features of the optimized applications.

*3) Predicting the Optimal Set of Features:* The role of these models is to predict optimal feature representations based on a program as input. As the features of a program are arguably the most crucial part of machine learning compiler optimization, as well as the base for performance, these models can help improve every aspect of auto tuning. [14] developed a model that "mines" for a set of static features to represent a program, minimizing file size or maximizing running speed.

*4) Predicting Intermediate Speedup:* Intermediate speedup predictor models, also known as reaction-based models, aim to understand and predict the behavior or speedup of an application being optimized. The model takes into account the characteristics of the application in each state and uses a set of input stimuli, such as compiler sequences or optimization strategies, to predict the speedup that can be achieved. The input is the current state of a program along with compiler optimizations, and the output is the predicted speedup. So these models can be used to look at how different optimizations applied repeatedly to an initial program speed up the program in stages and as a whole. [15] used this type of modeling to predict if a given optimization should be applied immediately.

*5) Optimization Search Space Reduction:* This method of machine learning is also known as clustering and downsampling. In general, optimizations are clustered together based on similarity, and from there, the feature space can be reduced. For example, if 50 different sets of optimizations have similar characteristics, then it is redundant to use all of them in the search space. So, we cut down by only selecting a couple or even one in our modeling. The authors of [16] used a clustering technique on all of the possible optimizations passes in LLVM's -03, and trained their models on a subset of them. The results showed that using this subset could outperform LLVM's highest optimization level with just a few predictions.

## VI. Deep Learning Approach: DeepTune

In subsection III-A, we discussed the issues of relying on hand-tuned heuristics, and how researchers have long been looking for a way to automate this process using machine learning [17]. The usual state-of-practice in this field, such as many of those cited in section V involves a predictive model being trained on empirical performance data and features of programs. The machine learning model is able to learn the correlations between the features and the optimization decision that has the best performance. Using this strategy, previous works were able to build machine learning-based heuristics that outperformed ones created manually.

However, even in these machine learning-based approaches, experts are note completely removed from the design process, since selecting the appropriate features for the model is a manual task that requires thorough knowledge of the system, and is prone to human error [18]. Failing to identify an important feature results in the applied heuristic being suboptimal.

One potential method to overcome this issue is to take humans out of the loop completely, which is the approach proposed by Cummins *et al.* [19]. They present deep learning as a solution to this issue, citing its recent successes in the identification of complex patterns in images [20] and computer code [21] as previous examples.

This study hypothesizes that by using deep neural networks, the authors would be able to bypass static feature extraction completely, and instead learn optimization heuristics directly on raw source code. Additionally, transfer learning is the idea that in neural network classification, the information learned at the early layers of the neural network (layers closer to the input later) will be able to be transferred toward solving many different tasks, and is more generalizable than that later, more specialized network layers [22]. By using *transfer learning*, the approach presented by Cummins *et al.*. is able to produce high-quality heuristics even when only learning on a small number of programs.

Cummins *et al.* present a methodology for building compiler heuristics without human-driven feature engineering, develop a novel tool called DeepTune for automatically constructing these heuristics (which outperforms previously existing state-of-the-art models), and applies *transfer learning* on optimizations to improve heuristics.

## *A. DeepTune Methodology Overview*

DeepTune is an ”end-to-end machine learning pipeline” for compiler optimization heuristics. The primary input of DeepTune is the raw source code of a program to be optimized, and through a series of neural networks, DeepTune predicts an optimization. The advantage of learning on source code is that DeepTune is not tied to any specific platform, therefore the exact same design can be reused to build numerous heuristics, without the need for any expert intervention.

DeepTune presents a framework of a ”successful model”, setting a benchmark for its own performance, as well as the performance for future systems that attempt to automate the construction of compiler heuristics. According to Cummins *et al.*, a successful model must contain the following capabilities:

1) Derive syntactic and semantic patterns of a programming language from sample codes and no external information
2) Identify patterns and representations in code
3) Discriminate performance characteristics from subtle differences in similar code

We will provide a brief overview of the DeepTune system architecture, a high-level summary of which is visible in Figure 3.

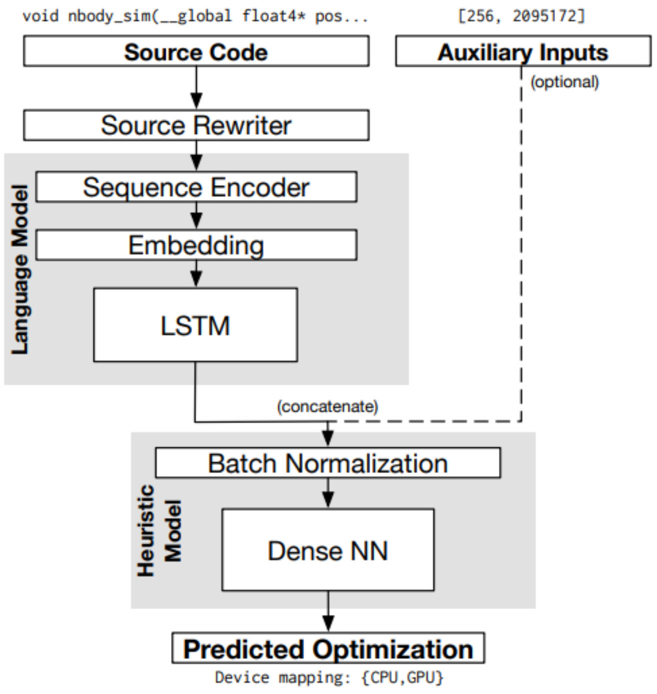


Fig. 3: High-level overview of the DeepTune architecture, in which code properties are extracted from the source code and fed to the heuristic model to produce the final prediction [19]. Details of the architecture of the implementation are provided in subsection VI-A.

*1) Source Rewriter:* To begin, DeepTune applies a series of *source normalizing* transformations, which parse the AST to remove conditional compilation, and then rebuild the input code using a consistent code style and naming scheme. This is to ensure that trivial semantic differences (i.e., variable names, comments, etc.) do not affect the learned model.

*2) Sequence Encoder:* DeepTune encodes source code as a sequence of integers to pass into a neural network. Each integer is an index into a vocabulary. There are two potential approaches for encoding vocabulary

1) **Character-based vocabulary**: This minimizes the size of the vocabulary, but consequently leads to long sequences which are difficult to extract structure from [18].
2) **Token-based vocabulary**: This minimizes the length of sequences, but leads to an ”explosion” in vocabulary size since every literal and identifier needs to have a unique representation [23].

Since both of these approaches have significant downsides as well as notable benefits, the authors combine them in order to maximize the strengths and mitigate the weaknesses. DeepTune uses a hybrid, partially-tokenized approach which represents common sequences like `for` and `if` or `float` and `int`.

*3) Embedding:* Tokens in the vocabulary are mapped to arbitrary unique integer values during encoding, which means that a language model is unable to infer the relationships between any tokens based on their mappings. An *embedding* translates tokens in such a sparse vocabulary into a lower-dimensional vector space, which allows tokens that are semantically related (`float`, `int`, `long`, etc.) to be mapped to nearby points so that a language model can identify relationships and patterns [24]. The DeepTune embedding layer maps each of the tokens in the integer-encoded vocabulary into a vector of real values that are then passed into the LSTM.

*4) Long Short-Term Memory:* Here, DeepTune begins the process of sequence characterization. Neural networks as used to extract a fixed-size vector (from the sequences of embedding vectors) that characterized the whole sequence. DeepTune uses the Long Short-Term Memory (LSTM) architecture, which implements a Recurrent Neural Network (RNN). RNNs allow the activations of neurons to be learned with respect to current and previous inputs in a sequence [25]. However, while most RNNs fall prey to the *vanishing gradients problem*, in which the strength of learning decreases over time, LSTMs use a *forget gate*, which allows them to retain their activations for arbitrary durations, allowing them to learn complex relationships over long sequences [26]. This is crucial when modeling program code, in which dependencies in sequences (i.e. a variable declared at the beginning and used throughout) occur frequently over long ranges. DeepTune uses a two-layer LSTM network.

*5) Auxiliary Inputs:* DeepTune additionally supports an arbitrary number of *auxiliary inputs*, which can be used to augment the source code. This might be helpful in situations such as to support applications in which the optimization heuristic depends on dynamic values that cannot be statically determined from program code.

*6) Batch Normalization:* Normalization is necessary since auxiliary inputs may have any values and may affect the training of the heuristic model. It is necessary to normalize them to match the language model activations, which are in the range [0, 1]. This occurs in batches, using the normalization method of Ioffe and Szegedy [27] to normalize each scalar of the heuristic model's inputs to a mean of 0 and standard deviation of 1, accounting for $\gamma$ and $\beta$ as the scale and shift parameters learned during training, respectively.

$$x_i' = \gamma_i \frac{x_i = E(x_i)}{\sqrt{Var(x_i)}} + \beta_i \tag{1}$$

*7) Dense Neural Network:* The dense neural network (NN) is comprised of two NN layers.

1) The first layer consists of 32 neurons. This layer uses rectifier activation functions $f(x) = max(0, x)$.
2) The second layer consists of one single neuron for each possible heuristic decision. This layer uses a sigmoid activation function $f(x) = \frac{1}{1+e^{-x}}$.

The activation of each output layer neuron represents the confidence of the model that the corresponding decision is the correct one, therefore DeepTune takes the `arg max` of the output layer in order to arrive at the decision with the largest activation

*8) Network training:* The model is trained with Stochastic Gradient Descent (SGD) using the Adam optimizer [28]. For the training data, SGD attempts to find the model parameter $\theta$:

$$\theta = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} l(X_i, \theta) \tag{2}$$

$\theta$ minimizes the input of a loss function $l(x, \theta)$, which computes the log difference between predicted ane expected values.

## B. DeepTune Experimental Results Overview

In order to evaluate DeepTune, the authors test it on two different popular optimization tasks: predicting the optimal device to run a given problem and predicting thread coarsening factors. DeepTune outperforms the existing state-of-the-art model in both cases.

*1) Case Study A: OpenCL Heterogeneous Mapping:* OpenCL provides a platform-independent framework for heterogeneous parallelism, which allows a program written in OpenCL to execute across a range of devices, such as CPUs, GPUs, and FPGAs [19]. The main concern then is selecting the execution device to maximize performance. The current state-of-the-art that the authors consider is Grewe *et al.*, who develop a predictive model for mapping the OpenCL kernels to the optimal device in CPU/GPU systems [29]. Grewe *et al.* use a decision tree approach using a combination of static and dynamic kernel features. The DeepTune authors replicate the experimental setup of [29], in which all experiments are extended to a larger set of 71 programs.

The DeepTune configuration for this is visible in Figure 4a. it uses the OpenCL kernel source code as input, as well as the dynamic values *workgroup size* (`wgsize`) and *data size* (`dsize`) available to the OpenCL runtime as auxiliary inputs. To evaluate the model, the authors use stratified 10-fold cross-validation.

In this experiment, the DeepTune authors compare DeepTune against a static single-device approach (the *static mapping* selects the device which gave the best average case performance over all the programs) and the Grewe *et al.* predictive model. The static approach is only correct for $58.8\%$ of cases on AMD and $56.9\%$ on NVIDIA, while the Grewe *et al.* model achieves an accuracy of $73\%$. DeepTune beats both of these approaches, giving an average accuracy of $82\%$.

Using the static mapping as a baseline the DeepTune authors compute the relative performance of each program using the device selected by each predictive model, the speedups of which are visible in Table I. Both predictive models greatly outperform the static mapping, with DeepTune achieving a $14\%$ boost in performance, even without any human intervention.

| Model | AMD Tahiti 7970 | NVIDIA GTX 970 |
|---|---|---|
| Grewe *et al.* [29] | 2.91x | 1.26x |
| DeepTune [19] | 3.34x | 1.41x |

TABLE I: Relative performance of each program on the heterogeneous mapping task (Case Study A), using static single-device approach as a baseline. This shows the model speedups on each platform.

*2) Case Study B: Thread Coarsening Factor:* Thread coarsening is a parallel program optimization in which the operations of multiple threads are fused together, which can prove especially beneficial on certain combinations of programs and architectures [19].

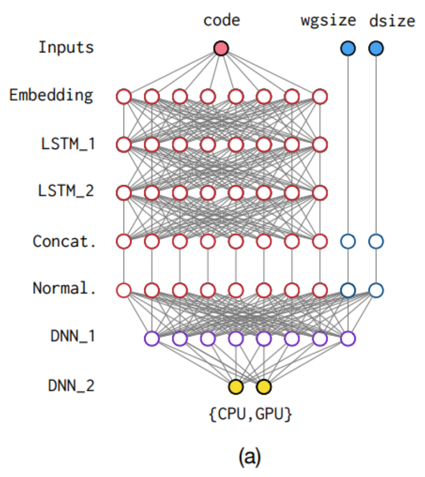
The current state-of-the-art DeepTune authors consider Magni *et al.* the current state-of-the-art [30]. Magni *et al.* presents a predictive Open CL thread coarsening model that uses an iterative heuristic that determines whether a program would benefit from coarsening. If it would, the program is coarsened and the process repeats. They reduce the multi-label classification problem into a series of binary decisions, while the DeepTune approach predicts the coarsening factor directly from the source code. The task involves selecting from one of six possible coarsening factors: (1, 2, 4, 8, 16, 32).

Magni *et al.* followed an incredibly comprehensive feature engineering process, in which 17 candidate features were assembled from previous studies of performance counters. For each of these candidate features, they computed a *coarsening delta*, which reflects the change in each feature value caused by coarsening. Magni *et al.* then used Principle Component Analysis (PCA) on the candidates and selected the first 7 principal components. The DeepTune authors replicate the experimental setup of Magni *et al.* [30], and the thread coarsening optimization is evaluated on 17 programs The DeepTune configuration for this task is visible in Figure 4b, where they use the OpenCL kernel as input and directly predict the coarsening factor from the source code.

Since the size of the evaluation is small compared to Case Study A from subsubsection VI-B1, the DeepTune authors use leave-one-out cross-validation to evaluate the models. Additionally, because [30] does not describe their neural network parameters, the DeepTune authors perform an additional nested cross-validation process in order to find the optimal neural network parameters. For every training set program, 48 combinations of network parameters are evaluated, and the best-performing configuration from 768 results is selected.

Evaluating the boosts in performance caused by optimizing thread coarsening can be difficult because on average, coarsening slows programs down; the speedup attainable by a perfect heuristic is only 1.36x [19]. The speedups achieved by the Magni *et al.* and DeepTune models for all programs and platforms are visible in Table II, with the baseline being the performance of programs without coarsening.

*3) Case Study Comparison:* For Case Study A and Case Study B, the authors of the respective state-of-the-art approaches arrived at incredibly different predictive model designs, with extremely different features and feature selection processes. In contrast, DeepTune takes the exact same approach for both problems. None of DeepTune's parameters were tuned for either case study. The similarity of the models is observable in Figure 4. The only difference between the two network designs is the presence of auxiliary inputs in Case Study A and the different number of optimization decisions. These differences make up only ”two lines of code” [19].



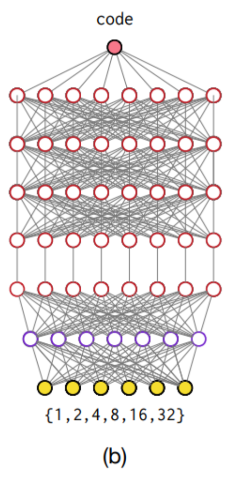


Fig. 4: DeepTune neural networks configured for (a) heterogeneous mapping detailed in subsubsection VI-B1 and (b) thread coarsening factor detailed in subsubsection VI-B2 [19].

*4) Transfer Learning:* There are more similarities between the tasks of building heuristics for heterogeneous mapping and selecting a thread coarsening factor, as demonstrated by the wildly different approaches of the previous state-of-the-art models: Grewe *et al.* [29] and Magni *et al.* [30]. However, the role of DeepTune in both tasks remains the same: extract meaningful representations of OpenCL code.

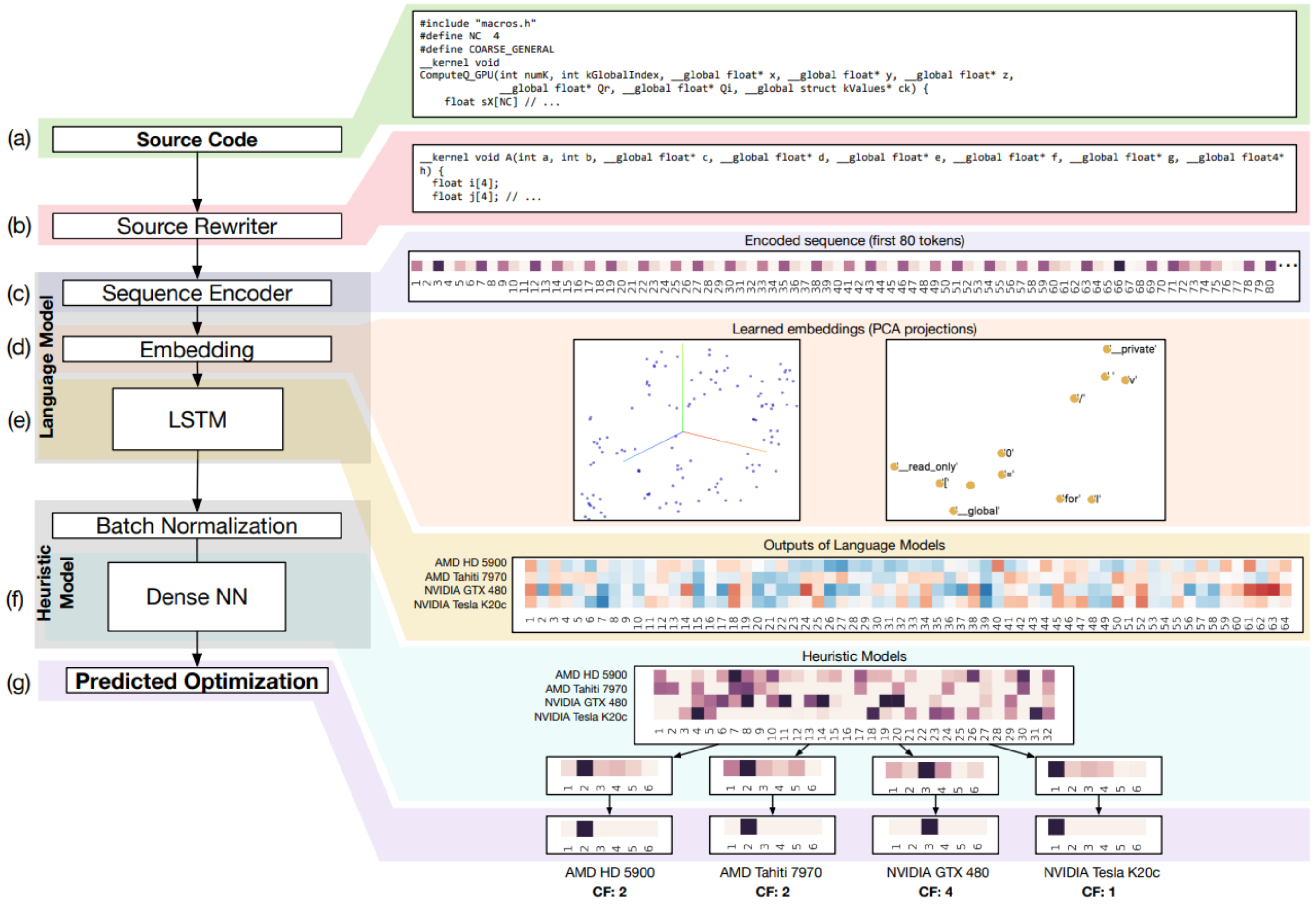
Prior deep learning research has demonstrated that models trained on similar inputs – albeit for different tasks – often share useful commonalities, since in neural network classification, information learned at earlier layers is generalizable and will be useful for multiple tasks, while later network layers are more specialized [22]. This technique is known as *transfer learning* and if applied to DeepTune, would enable the ”novel transfer of information across different optimization domains” [19].

In order to test this, the authors extracted the language model trained on Case Study A (heterogenous mapping) and transferred it to Case Study B (thread coarsening). Since DeepTune maintains the same design regardless of the task, this is no more complicated than copying the learned weights over, and then training the model as normal.

The relative performance of this technique is visible in subsubsection VI-B2, in the ”Deeptune-TL” row; DeepTune-TL has improved performance in 3 out of the 4 platforms, providing an average 12% performance improvement over Magni *et al.*, and in 81% of instances, transfer learning

| Model | AMD HD 5900 | Tahiti 7970 | NVIDIA GTX 970 | Tesla K20c |
|---|---|---|---|---|
| Magni *et al.* [30] | 1.21x | 1.01x | 0.86x | 0.94x |
| DeepTune [19] | 1.10x | 1.05x | 1.10x | 0.99x |
| DeepTune-TL | 1.17x | 1.23x | 1.14x | 0.93x |

TABLE II: Relative performance of each program on the thread coarsening factor task (Case Study B), using the performance of programs without coarsening as the baseline. This figure shows the model speedups on each platform. Additionally, this includes the application of *transfer learning*, in which a model trained on Case Study A is applied to Case Study B (DeepTune-TL).



```
#include "macros.h"
#define NC  4
#define COARSE_GENERAL
__kernel void
ComputeQ_GPU(int numK, int kGlobalIndex, __global float* x, __global float* y, __global float* z,
             __global float* Qr, __global float* Qi, __global struct kValues* ck) {
    float sX[NC] // ...
```

```
__kernel void A(int a, int b, __global float* c, __global float* d, __global float* e, __global float* f, __global float* g, __global float4*
h) {
  float i[4];
  float j[4]; // ...
```

Fig. 5: Visualizing the internal state of DeepTune when predicting coarsening factor for Parbo-`NoValue-` benchmark on four different architectures [19].

matched or improved DeepTune decisions, providing a 16% improvement in performance per platform.

*5) Internal Activation States:* As most deep neural network models are notorious for being difficult to interpret ("black box" models), the authors of DeepTune attempt to illuminate the internal activation states and inner workings of DeepTune using an example from Case Study B – predicting thread coarsening factor for Paboil's `mriQ` benchmark on all four platforms.

The DeepTune configuration is shown in Figure 5. First, we begin with the input (267 lines of OpenCL code for the `mriQ` kernel), which is preprocessed, formatted, and rewritten (shown in Figure 5b), and then tokenized and encoded in a 1-of-k vocabulary. The first 80 elements of this sequence is shown in Figure 5c as a heatmap. These sequences are passed into the embedding layer and each vocabulary token is mapped to a point in a 64-dimension vector space. These embeddings are learned during training so that semantically related tokens can cluster together; the PCA projection of the embedding space and clusters of tokens are visible in Figure 5d. This specific cluster contains semantically related address space modifiers (`__private`, `__global`, `__read_only`, etc).

Two layers of 64 LSTM neurons then model the sequence of embeddings, and the second layer's activations are used to characterize the entire sequence, as visible in Figure 5e as a red-blue heatmap, to visualize the intensity of each activation.

As information flows through the network, the layers visibly become more specialized to each specific platform, which is shown in Figure 5; the two layers of the heuristic model in which the activations increasingly diverge. In Figure 5g, DeepTune takes the largest activation of the output layer as the final prediction for the optimal thread coarsening factor. For this specific program, the previous state-of-the-art model achieves 54% of the maximum performance while DeepTune achieves 99% of the maximum performance.

### C. DeepTune Significance

This paper presents a novel tool for constructing compiler optimization heuristics, DeepTune, which forgoes feature extraction in favor of language modeling techniques to build program representations directly from raw source code.

This has three main benefits:

1) A huge reduction in development effort
2) Improved heuristic performance as opposed to expert-selected potentially suboptimal decisions
3) More simple model designs

This approach is fully automated, meaning the developers no longer need to spend large amounts of time leveraging statistical methods to select program features, even without tailoring the model design or parameters for the optimization task. DeepTune is able to achieve performance on par with and exceeding state-of-the-art predictive models [30] [29]. The authors use DeepTune to automatically build compiler heuristics for two challenging optimization problems, heterogeneous device mapping, and thread coarsening factor selection, and find that DeepTune is able to outperform state-of-the-art predictive models by 14% and 12%. Additionally, DeepTune is able to employ *transfer learning*, allowing it to exploit information learned from another optimization problem in order to give learning performance a boost. Applying *transfer learning* provides up to a 16% performance improvement. This is especially applicable for domains for which training data is scarce.

There are many potential avenues for research in which the work presented in DeepTune could be extended. For example, the heuristic construction approach could be extended by automatically learning dynamic features over raw data. Another possibility is applying unsupervised learning techniques over unlabeled source code in order to further improve the learned representations of programs [31]. Trained DeepTuned heuristics could also be deployed to low-power embedded systems using quantization and compression of neural networks [32].

## VII. Discussion

### A. Critiques

Although the main three papers we selected (a survey of machine learning in compiler optimizations [3], a survey of autotuning techniques , and an exploration of a deep learning framework for optimizations called DeepTune [19]) provided thorough explanations and promising results, it is important to acknowledge some of the weaknesses and ambiguities in these papers.

insert auto-tuning citation

*1) Machine Learning in Compiler Optimisation [3]:* This paper provides a comprehensive introduction to the field of compiler optimization using machine learning. Within its 20+ pages, the authors provide extensive details on the methodologies, machine learning models, features, and scopes of multiple machine learning approaches. Within each section, they provide a multitude of examples highlighting specific approaches and techniques with around 200 citations included in the references section.

Although the paper is very thorough, it would have been helpful to review empirical results from each of the surveyed papers discussed and compare them in detail. Although the paper provided a good overview of the field, it is difficult to decipher which models and techniques achieve the best performance, which would help lay the foundations for further research in the field of machine learning for compiler optimizations.

The rest of the content in this paper is very well crafted. The figures especially provide valuable visualizations of difficult-to-explain topics. The discussion section also provides great insight into more nuanced topics in compiler optimization with machine learning. Any novice researcher in the field will find great value in reading this paper.

*2) Autotuning [6]:* The paper on auto tuning provided a comprehensive survey and overview of the field of compiler optimization. It covered a wide range of methods, techniques, and definitions. The authors also discussed breakthroughs achieved by different auto tuning techniques, which was beneficial.

However, in my opinion, the paper could have delved deeper into certain sections to provide a more detailed understanding. The explanations provided were mainly high-level, lacking technical examples.

For instance, in the section on machine learning, they could have provided more information on how a linear model is applied to a specific compiler optimization problem, along with actual results.

The paper also lacked detailed information on neural networks, which are gaining prominence in compiler optimization. Additionally, the authors could have provided more specific and visual comparisons of the results obtained from different techniques. Including plots showing the runtime comparison between traditional methods and machine learning methods, for example, would have made it easier to identify which models are suitable for specific compiler problems.

Nevertheless, considering that the paper is a survey of auto tuning in compiler optimization, it was well-written.

*3) End-to-end Deep Learning of Optimization Heuristics (DeepTune) [19]:* While DeepTune presents promising results, it is worth mentioning that dynamic information (such as program input size, performance counter values, etc.) are often essential for characterizing the behavior of a target program. Therefore DeepTune does not completely remove human involvement from feature engineering when static code features are insufficient for optimization problems. Auxiliary features would be required for many optimization tasks and would necessitate expert intervention.

Also, while the idea of removing humans from the compiler heuristic construction process provides impressive results, this may only exacerbate the issue of deep neural networks seeming like a ”black box”. While the DeepTune authors attempt to illuminate the inner workings of DeepTune by showing an example of intenal activation states, the neural net cannot explain the compiler heuristic decision it took they way an expert can. A lack of explainability lends itself to a lack of trustworthiness, and developers need to trust the model in order to allow it to automatically construct compiler optimization heuristics.

Another critique is of the representation in the results. Although the authors boast about the relative performance improvements caused by *transfer learning*, as visible in Table II, they claim that it has improved performance for 3 of the 4 platforms. Although it does, overall, show an improvement in thread coarsening in comparison to the previous state-of-the-art presented by Magni *et al.* [30], the ”3 out of 4” statistic only holds true when comparing DeepTune-TL against the regular DeepTune performance. When compared to Magni *et al.*, DeepTune-TL only achieves higher performance on half of the platforms. The authors do not discuss the cases in which Magni *et al.* beats the DeepTune model in performance, or run any kind of post-evaluation failure analysis in order to investigate where DeepTune struggles.

The authors also include some measures of relative performance in terms of the percentage of the maximum performance when describing the internal activation states of DeepTune, but it is unclear how this metric is determined. It seems redundant to have this threshold hand-selected by experts since the entire issue of compiler heuristics is that humans often make the mistake of choosing suboptimal heuristics and not reaching ideal performance.

Finally, DeepTune demonstrates results that beat state-of-the-art models across many platforms and multiple optimizations tasks, however due to its framework, it may be difficult to reproduce and build off of – it is potentially impractical due to sheer computational cost, and the authors provide little to no discussion acknowledging or mitigating the costliness of DNNs. They require a very large dataset to train, even with the use of transfer learning and training them takes time and significant computational resources.

### *B. Remaining Questions*

Machine learning for compiler optimizations is a quickly evolving field, but there are some remaining questions that inspire new directions for research every day.

*1) How can we design ML models that can learn from a wide variety of programs?:* Currently, most machine learning models for optimization are trained on a specific set of benchmarks, which limit their generalizability. They are not tested on many real-world problems. We need to see how these compiler optimization machine learning models operate in the real world before we can expand their scope and usage.

*2) How can we understand and interpret the decisions made by ML models for optimizations?:* Understanding the reasoning made by thee models is crucial for understanding ensuring reliability and trustworthiness. ”Black box” models such as neural nets lack explainability and make interpretable results difficult to obtain. Also, it’s difficult to enforce accountability in event of error.

*3) How can we ensure ML models are able to handle the increasingly complex and diverse codebases being developed today?:* The increasing complexity of software systems makes it increasingly difficult to optimize them during traditional compiler techniques. Machine learning models need to evolve with the codebases they are meant to extract representation from.

## VIII. Conclusion

In this paper, we gave a history of the relationship between machine learning and compilers. We went through the three stages of supervised machine learning in compilers: feature engineering, learning a model, and deployment. We outlined the role of cost functions in machine learning models, and discussed the prevalence of hand-coded compiler optimization heuristics, and why they are problematic and suboptimal. We discussed autotuning processes and an overview of the most common machine learning models, including supervised learning, unsupervised learning, and reinforcement learning. Finally, we highlighted one of the landmark papers of machine learning for compilers, that proposed a deep neural network methodology that would take human experts out of the loop completely in order to overcome the danger of human error in hand-coded heuristics, and rather learn directly on raw source code. We delineated an overview for DeepTune, the first work to employ a neural network to extract features from source code for compiler optimization.

Machine learning is a quickly evolving field, with new applications and improved models being rapidly released. It is crucial to review the history of the field, as well as a survey of the current state-of-the-art for machine learning in compilers in order to determine areas in which machine learning struggles to optimize compilers, and areas in which it flourishes. Through thorough review and study, machine learning has the potential to change the fundamentals of constructing optimization heuristics, as we study past works in order to lay groundwork for future study.

## References

[1] E. G. Nilges, *A Brief History of Compiler Technology*. Berkeley, CA: Apress, 2004, pp. 1–13. [Online]. Available: https://doi.org/10.1007/978-1-4302-0698-9_1

[2] A. L. Fradkov, “Early history of machine learning,” *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 1385–1390, 2020, 21st IFAC World Congress. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896320325027

[3] Z. Wang and M. F. P. O’Boyle, “Machine learning in compiler optimisation,” *CoRR*, vol. abs/1805.03441, 2018. [Online]. Available: http://arxiv.org/abs/1805.03441

[4] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O’Boyle, and E. Rohou, “Proceedings of the 1998 workshop on profile and feedback directed compilation (pfdc’98),” 1998.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[6] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys*, vol. 51, no. 5, pp. 1–42, sep 2018. [Online]. Available: https://doi.org/10.1145%2F3197978

[7] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, "A bayesian network approach for compiler auto-tuning for embedded processors," in *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2014, pp. 90–97.

[8] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley, "Using machines to learn method-specific compilation strategies," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. USA: IEEE Computer Society, 2011, p. 257–266.

[9] C. W. Fraser, "Automatic inference of models for statistical code compression," *SIGPLAN Not.*, vol. 34, no. 5, p. 242–246, may 1999. [Online]. Available: https://doi.org/10.1145/301631.301672

[10] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 196–206. [Online]. Available: https://doi.org/10.1145/2259016.2259042

[11] J. Thomson, M. F. P. O'Boyle, G. Fursin, and B. Franke, "Reducing training time in a one-shot machine learning-based compiler," in *International Workshop on Languages and Compilers for Parallel Computing*, 2009.

[12] A. Mcgovern, E. Moss, and A. Barto, "Scheduling straight-line code using reinforcement learning and rollouts," 07 1999.

[13] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of the 4th International Conference on Computing Frontiers*, ser. CF '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 131–142. [Online]. Available: https://doi.org/10.1145/1242531.1242553

[14] F. Li, F. Tang, and Y. Shen, "Feature mining for machine learning based compilation optimization," in *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2014, pp. 207–214.

[15] E. Park, S. Kulkarni, and J. Cavazos, "An evaluation of different modeling techniques for iterative compilation," in *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 65–74.

[16] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, sep 2017. [Online]. Available: https://doi.org/10.1145/3124452

[17] T. L. Falch and A. C. Elster, "Machine learning based auto-tuning for enhanced opencl performance portability," *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 1231–1240, 2015.

[18] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. IEEE Press, 2017, p. 86–99.

[19] ——, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 219–232.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[21] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 472–483. [Online]. Available: https://doi.org/10.1145/2635868.2635901

[22] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" *CoRR*, vol. abs/1411.1792, 2014. [Online]. Available: http://arxiv.org/abs/1411.1792

[23] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, p. 207–216.

[24] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *ArXiv*, vol. abs/1310.4546, 2013.

[25] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*. Pmlr, 2013, pp. 1310–1318.

[26] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. pmlr, 2015, pp. 448–456.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[29] D. Grewe, Z. Wang, and M. F. P. O'Boyle, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–10.

[30] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 455–466. [Online]. Available: https://doi.org/10.1145/2628071.2628087

[31] Q. V. Le, R. Monga, M. Devin, G. Corrado, K. Chen, M. Ranzato, J. Dean, and A. Y. Ng, "Building high-level features using large scale unsupervised learning," *CoRR*, vol. abs/1112.6209, 2011. [Online]. Available: http://arxiv.org/abs/1112.6209

[32] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.