

End-to-end

Deep Learning of Optimization Heuristics

<http://chriscummins.cc/pact17>



Chris Cummins

University of Edinburgh



Pavlos Petoumenos

University of Edinburgh



Zheng Wang

Lancaster University



Hugh Leather

University of Edinburgh

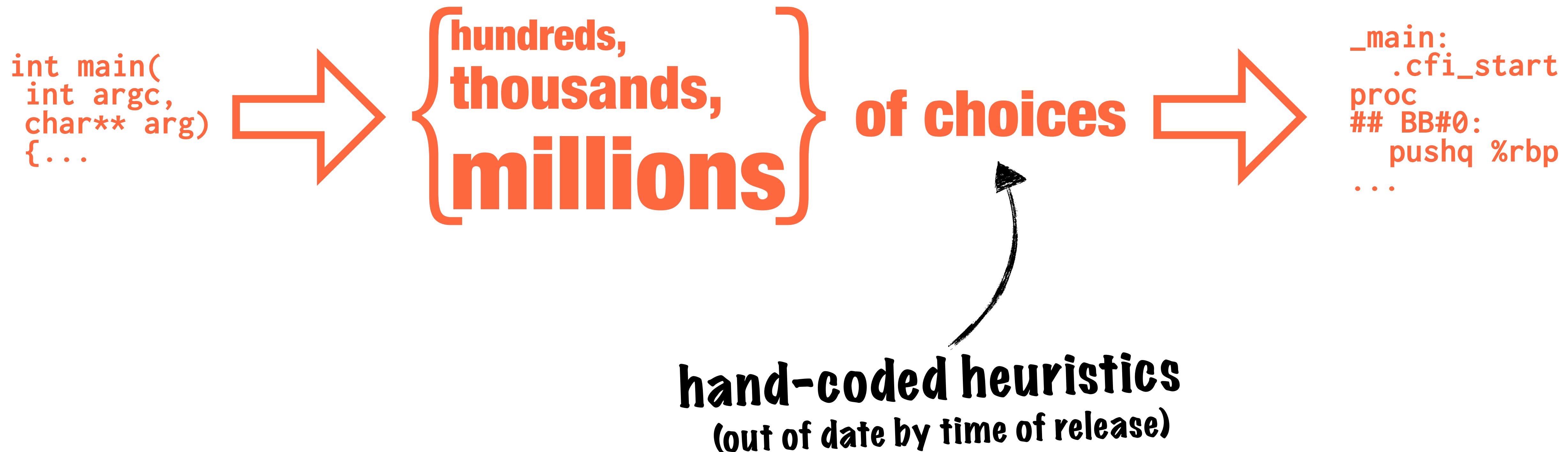


THE UNIVERSITY OF EDINBURGH
informatics

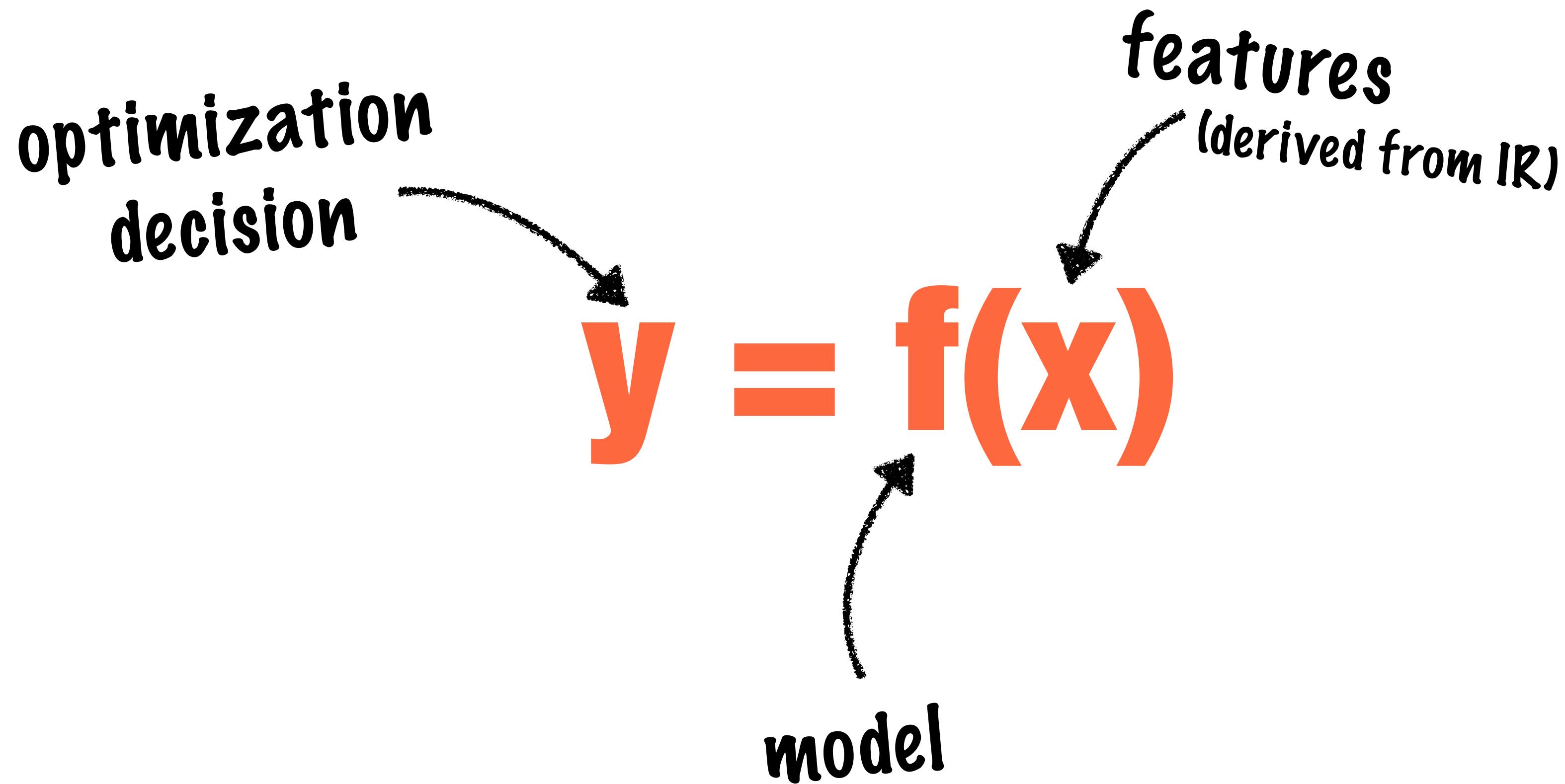
EPSRC Centre for Doctoral Training in
Pervasive Parallelism

EPSRC
Engineering and Physical Sciences
Research Council

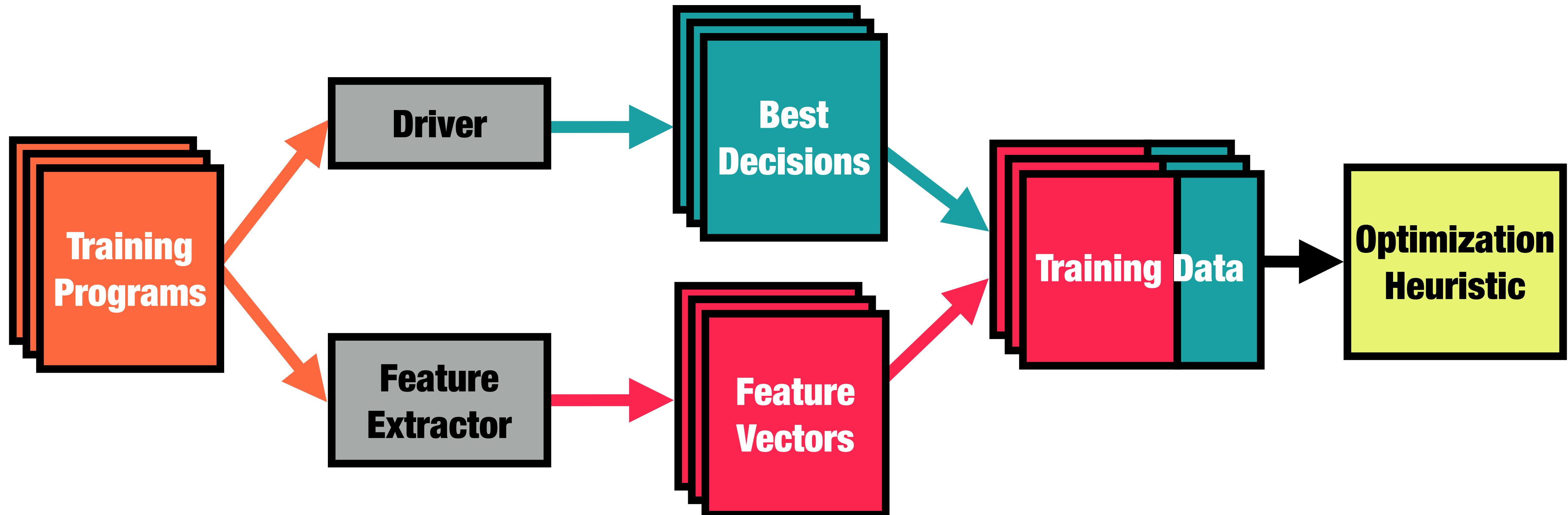
compilers are very complex



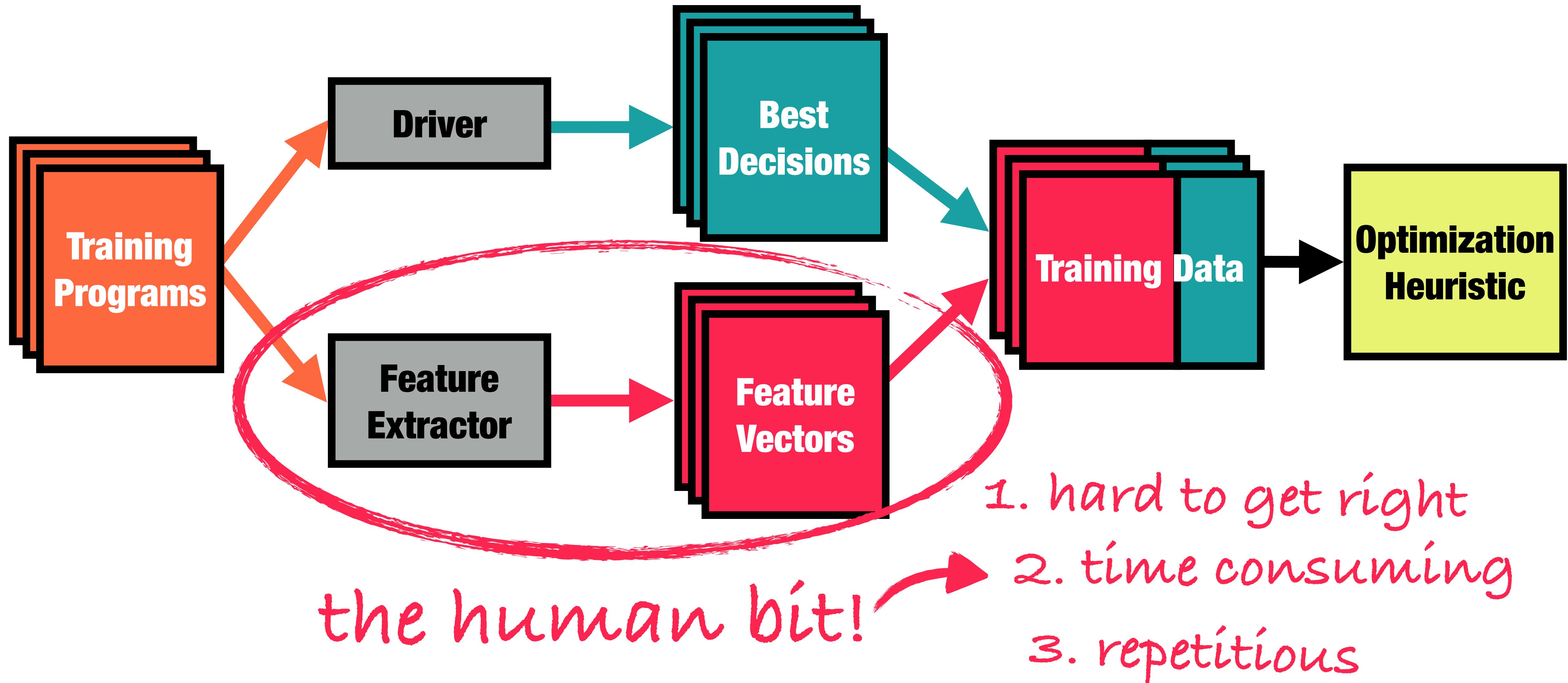
Machine learning in compilers



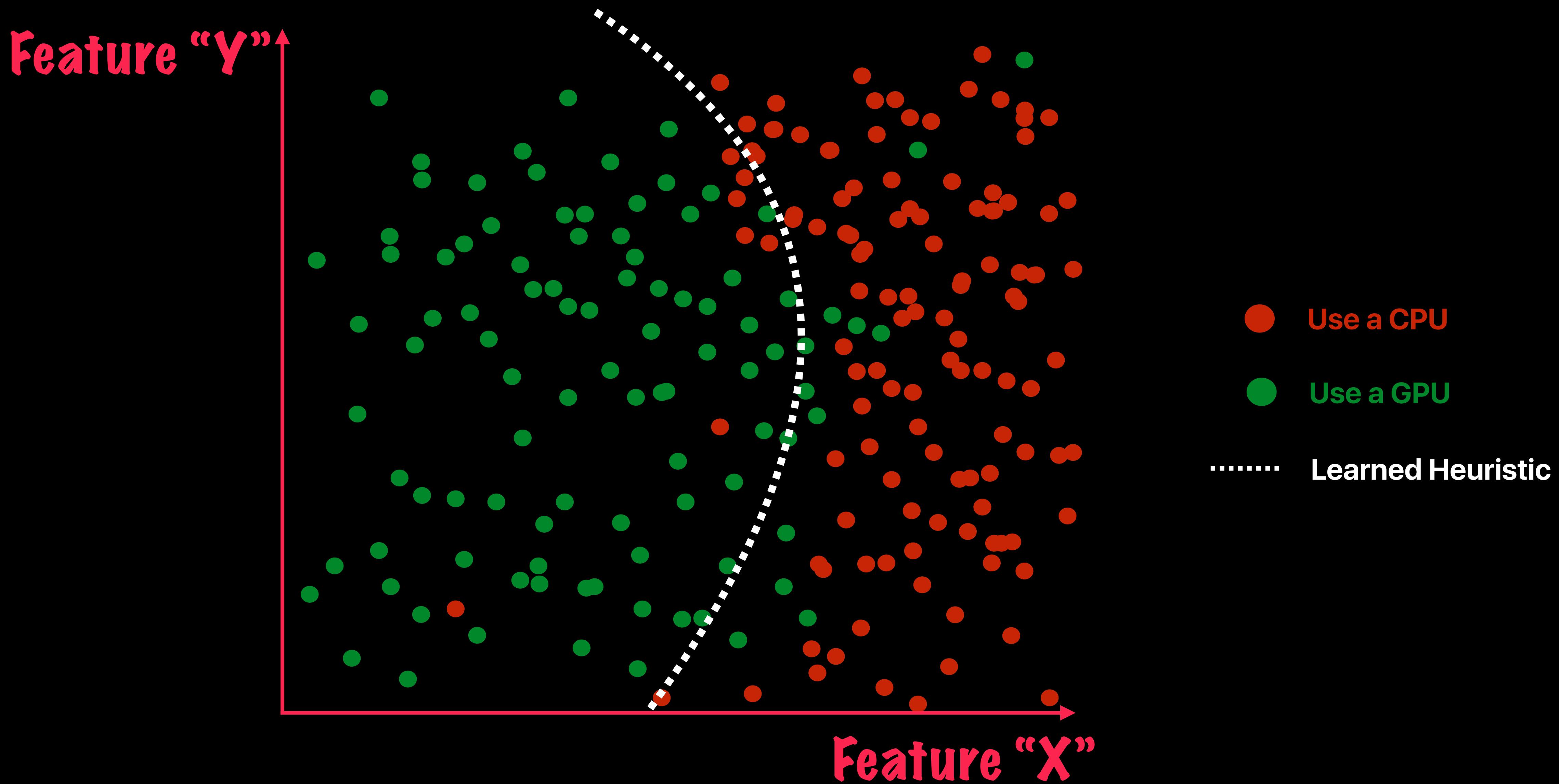
Machine learning in compilers



Machine learning in compilers



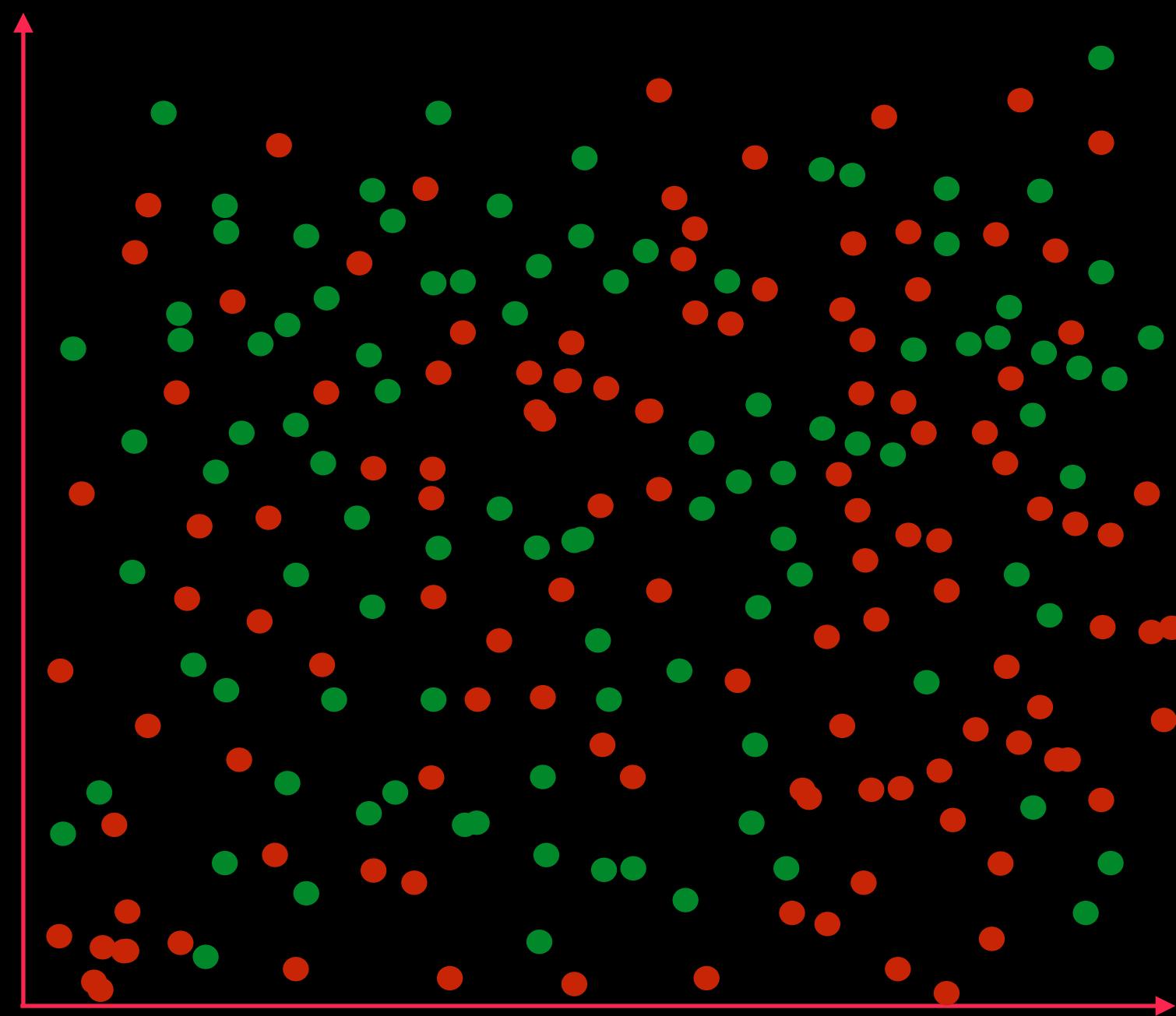
Feature space



Feature space

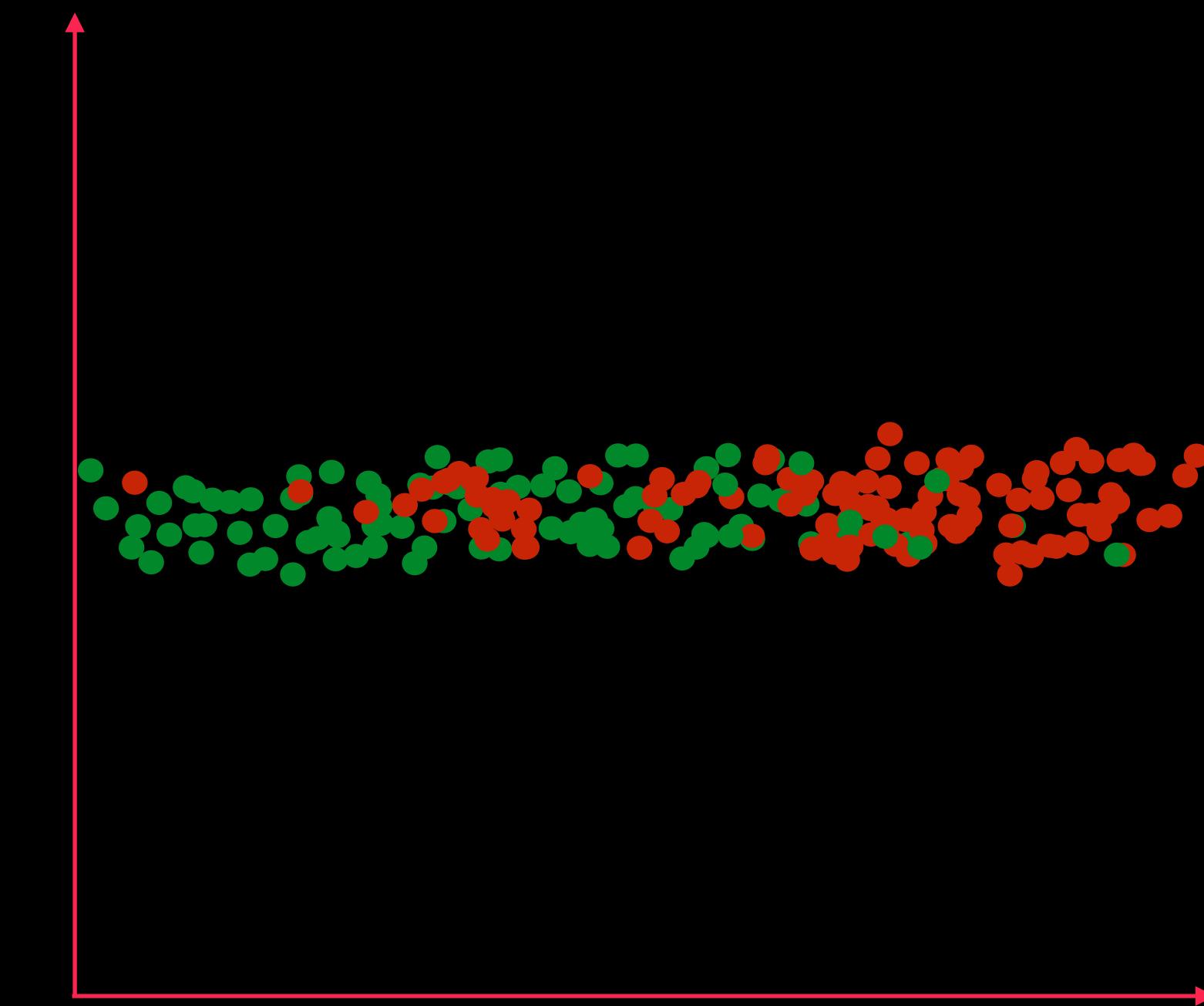


Ways to fail



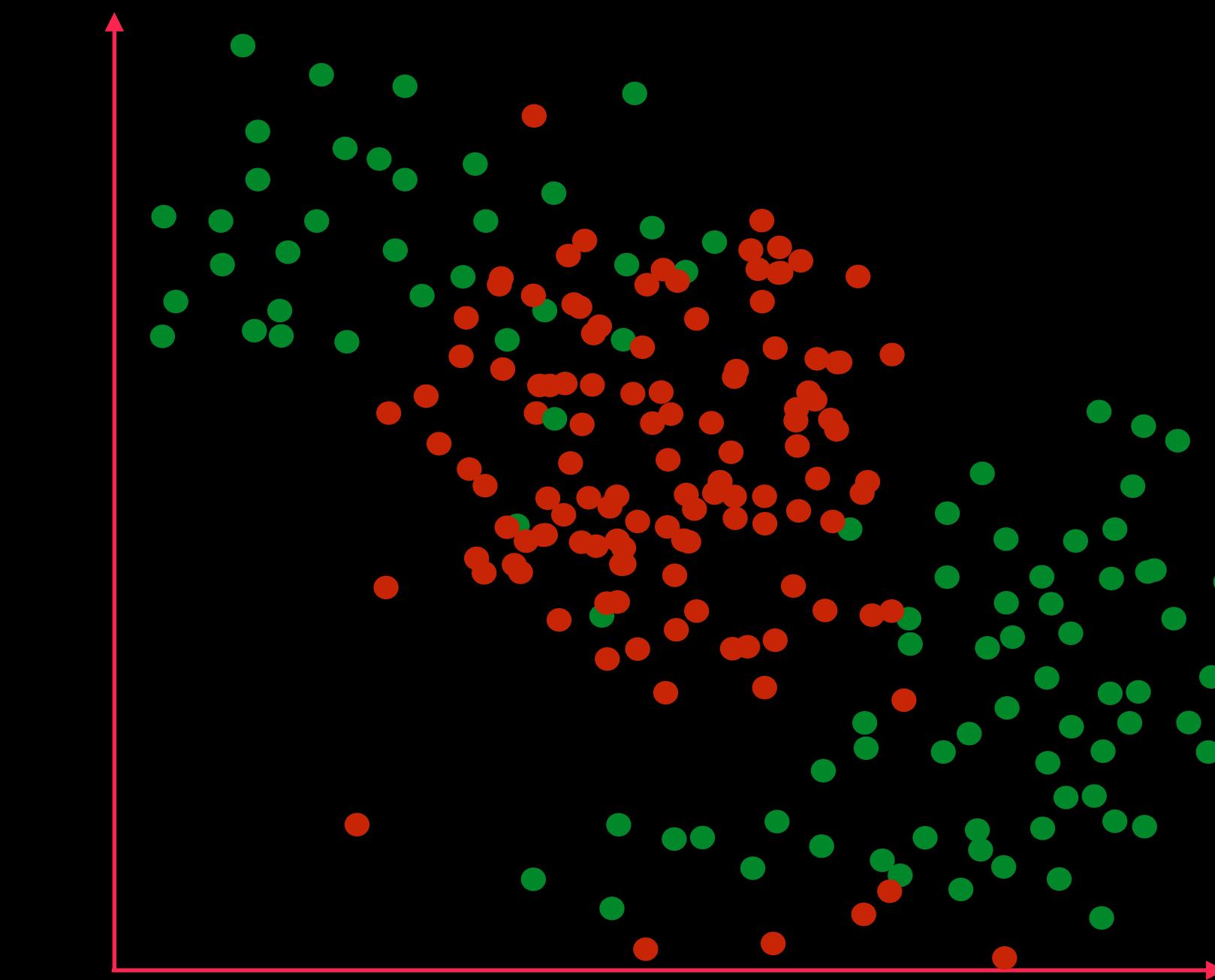
irrelevant

e.g. not capturing the right information



incomplete

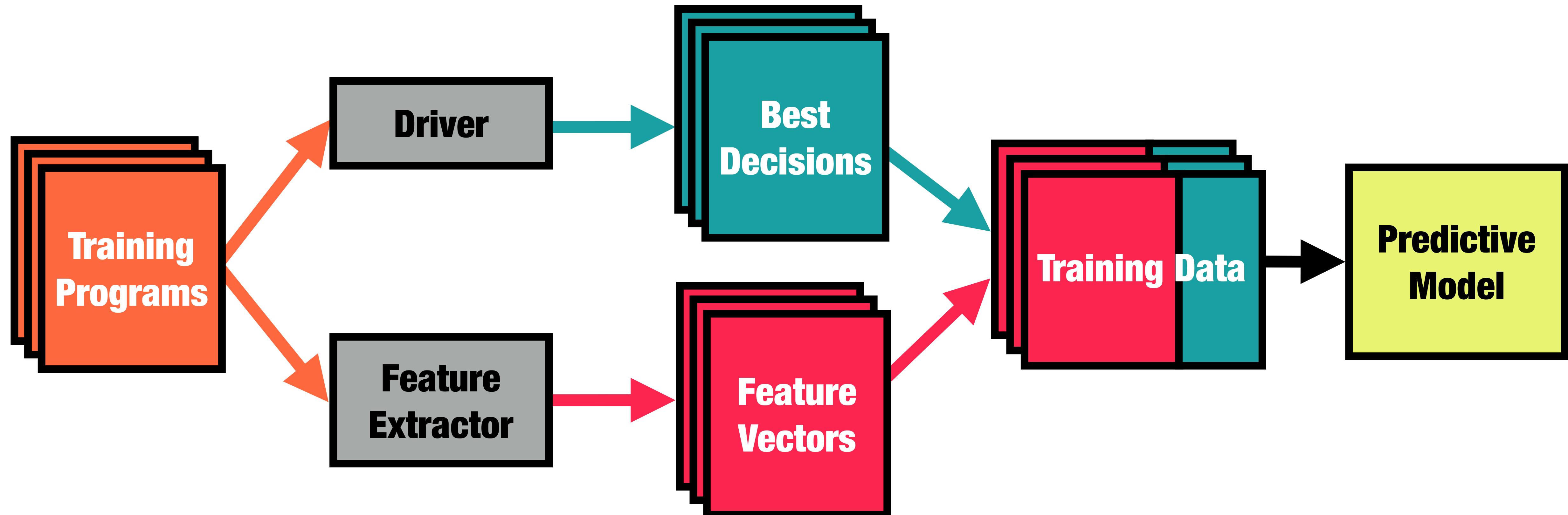
e.g. missing critical information



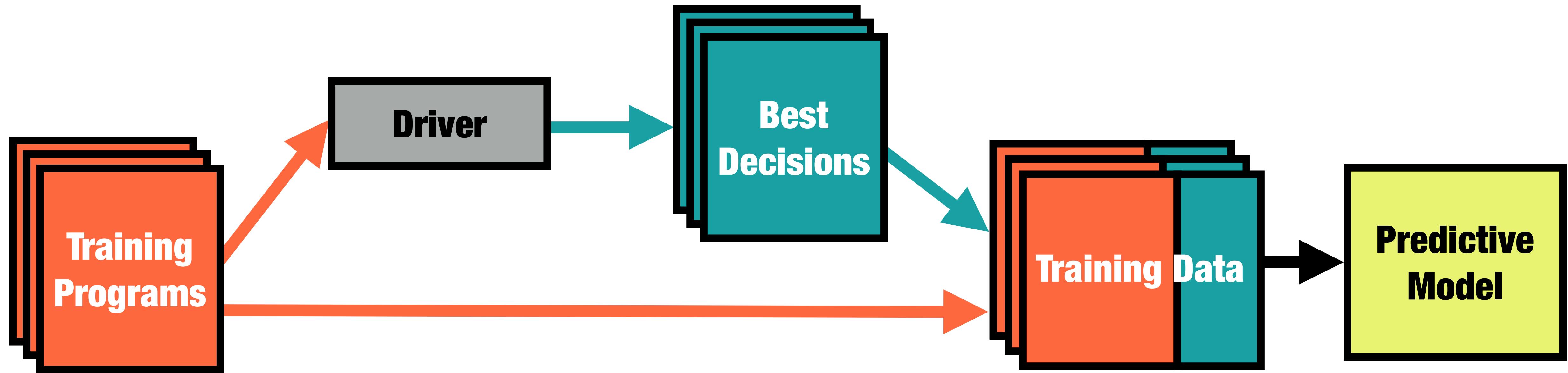
unsuitable

e.g. wrong combination of features / model

What we have



What we need



Contributions

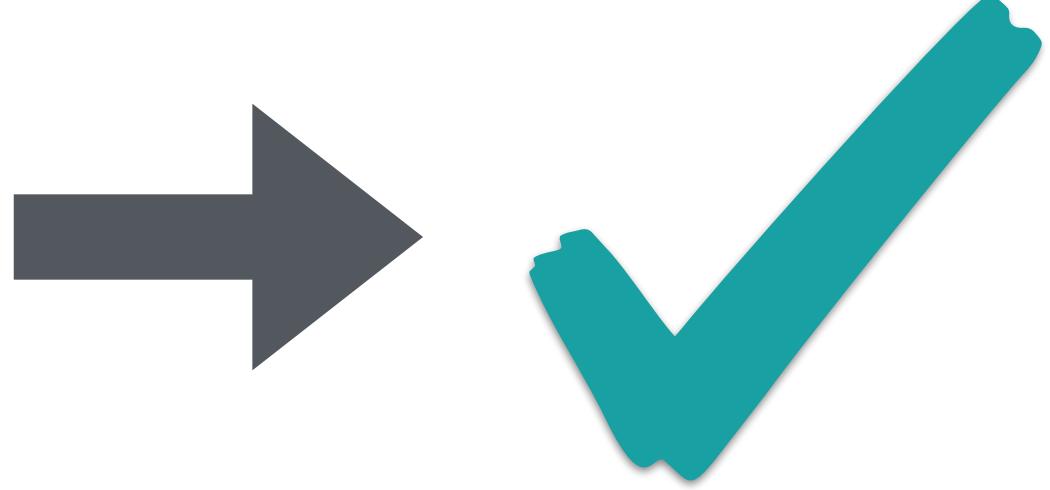
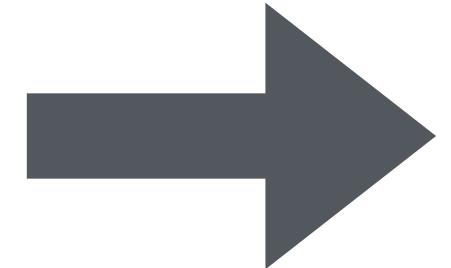
Heuristics without features

Beats expert approach

Learning across heuristics

Our approach

```
int  
main(int  
argc, char  
**argv)  
{ ...
```

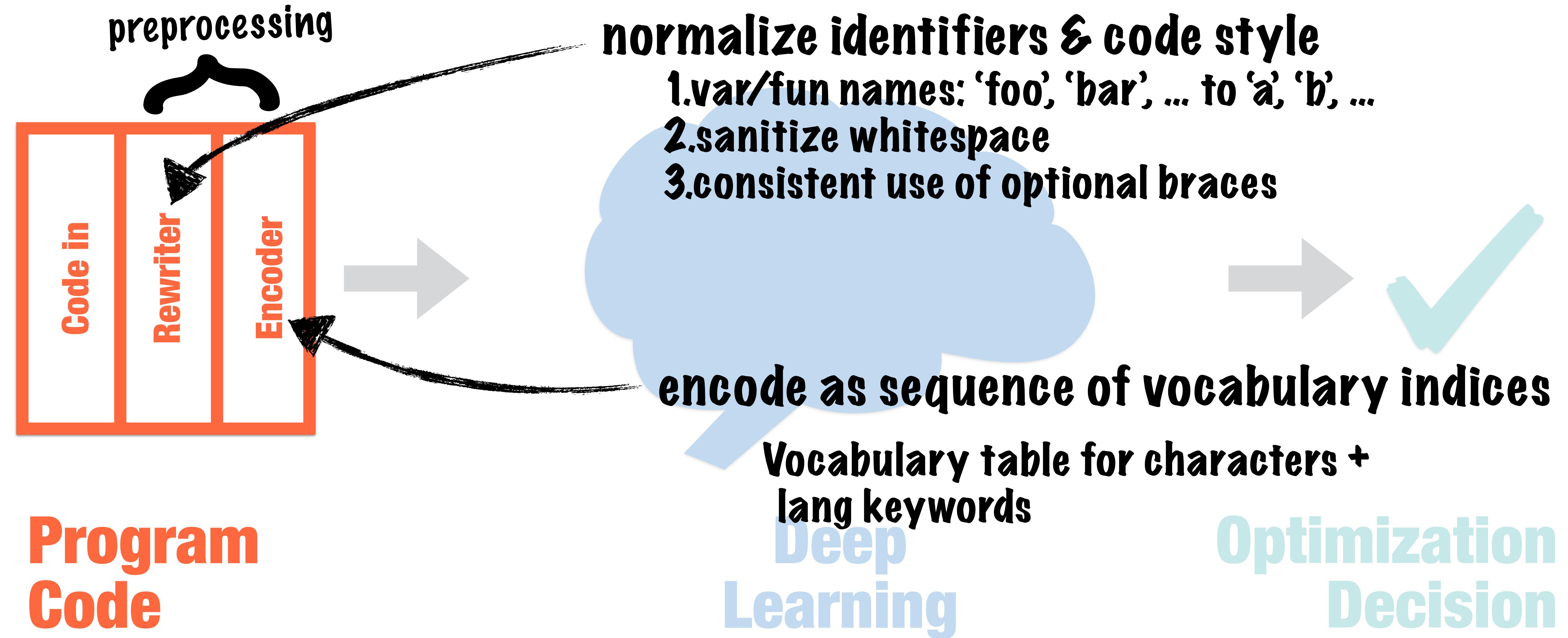


**Program
Code**

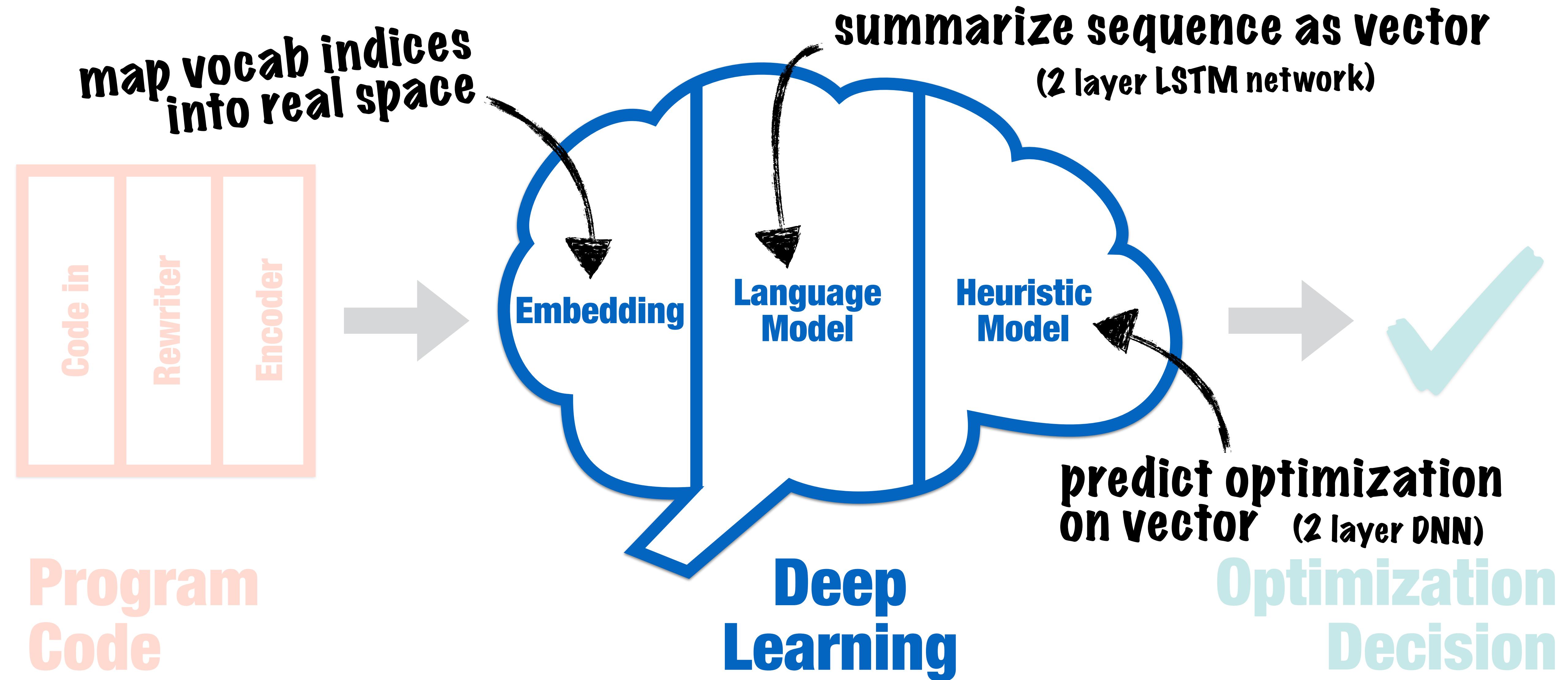
**Deep
Learning**

**Optimization
Decision**

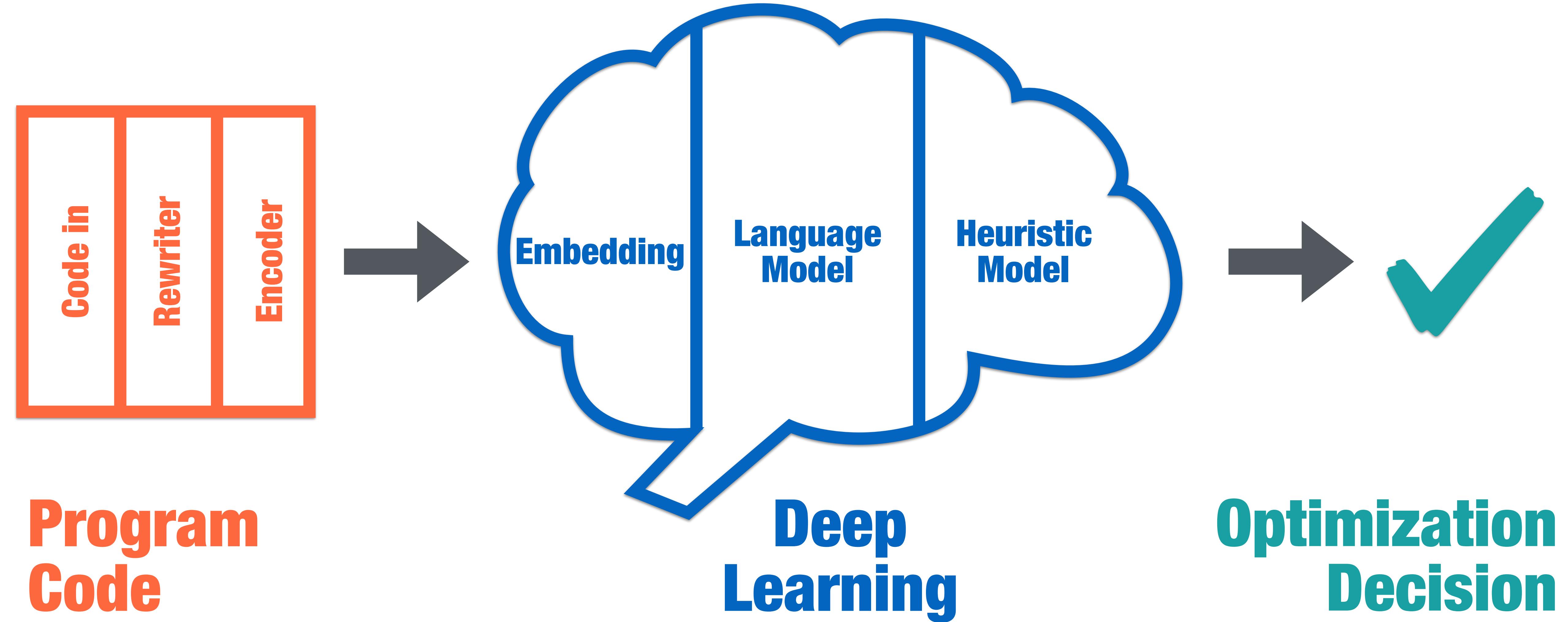
Our approach



Our approach

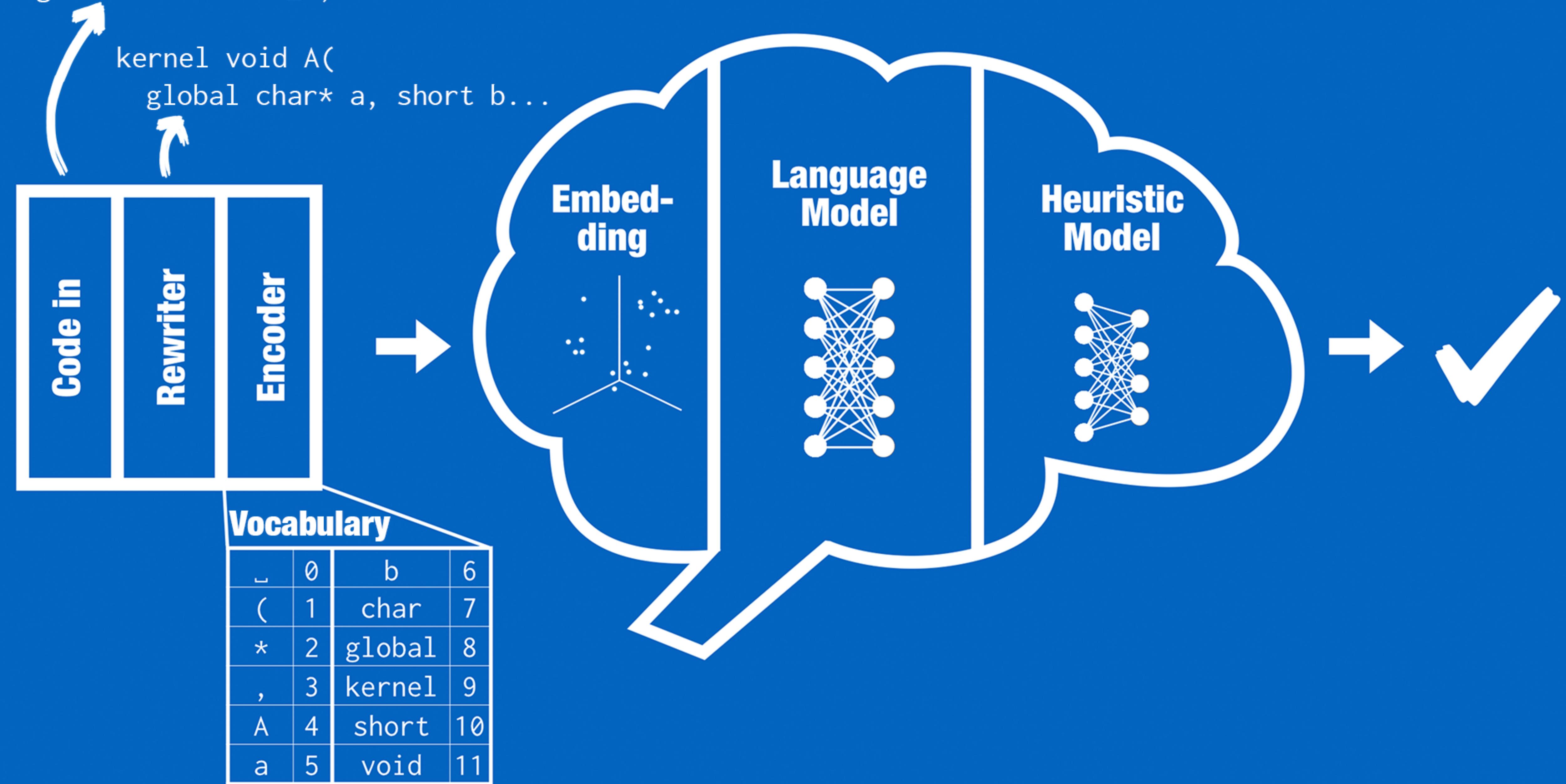


Our approach



How does it work?

```
kernel void memset_kernel(  
    global char* mem_d, short val...)
```



How well does it work?

Prior Art

Heterogeneous Mapping

Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems

Dominik Grawe Zheng Wang Michael F.P. O'Boyle
School of Informatics, University of Edinburgh
(dominik.grawe, zh.wang)@ed.ac.uk, mob@inf.ed.ac.uk

Abstract
General purpose GPU based systems are highly attractive as they give potentially massive performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. User typically have to identify potential sections of the code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.
OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier.
We achieved average (up to) speedups of 4.51x and 4.20x (14x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.
Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers
General Terms Experimentation, Languages, Measurement, Performance
Keywords GPU, OpenCL, Machine-Learning Mapping

1. Introduction
Heterogeneous systems consisting of a host multi-core and GPU are highly attractive as they give potentially massive performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. User typically have to identify potential sections of the code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier.

We achieved average (up to) speedups of 4.51x and 4.20x (14x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers
General Terms Experimentation, Languages, Measurement, Performance
Keywords GPU, OpenCL, Machine-Learning Mapping

1. Introduction
Heterogeneous systems consisting of a host multi-core and GPU are highly attractive as they give potentially massive performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. User typically have to identify potential sections of the code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier.

We achieved average (up to) speedups of 4.51x and 4.20x (14x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers
General Terms Experimentation, Languages, Measurement, Performance
Keywords GPU, OpenCL, Machine-Learning Mapping

1. Introduction
Heterogeneous systems consisting of a host multi-core and GPU are highly attractive as they give potentially massive performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. User typically have to identify potential sections of the code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier.

We achieved average (up to) speedups of 4.51x and 4.20x (14x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers
General Terms Experimentation, Languages, Measurement, Performance
Keywords GPU, OpenCL, Machine-Learning Mapping

1. Introduction
Heterogeneous systems consisting of a host multi-core and GPU are highly attractive as they give potentially massive performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. User typically have to identify potential sections of the code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier.

We achieved average (up to) speedups of 4.51x and 4.20x (14x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright or other rights reserved by the author or other rights holders. Reproduction for general distribution or for commercial advantage requires prior specific permission or a fee. Request permission from [Permissions@acm.org](http://permissions.acm.org).
CGO '13, 23–27 February 2013, Shanghai China
978-1-4673-5523-6/13/02 © 2013 IEEE. \$15.00

PACT'14

Thread Coarsening

Automatic Optimization of Thread-Coarsening for Graphics Processors

Alberto Magni
School of Informatics
University of Edinburgh
United Kingdom
a.magni@sms.ed.ac.uk

Christophe Dubach
School of Informatics
University of Edinburgh
United Kingdom
christophe.dubach@ed.ac.uk

Michael O'Boyle
School of Informatics
University of Edinburgh
United Kingdom
mob@inf.ed.ac.uk

ABSTRACT

OpenCL has been designed to achieve functional portability across multi-core devices from different vendors. However, the lack of a single cross-target optimizing compiler severely limits performance portability of OpenCL programs. Programmers need to manually tune applications for each specific device, preventing effective portability. We target a compiler transformation specific for data-parallel languages: thread-coarsening and show it can improve performance across different GPU devices. We then address the problem of selecting the best value for the coarsening factor parameter, i.e., deciding how many threads to merge together. We experimentally show that this is a hard problem to solve: good configurations are difficult to find and naive coarsening in fact leads to substantial slowdowns. We propose a solution based on a machine-learning model that predicts the best coarsening factor using kernel function static features. The model automatically specializes in the different architectures considered. We evaluate our approach on 17 benchmarks on four devices: two Nvidia GPUs and two different generations of AMD GPUs. Using our technique, we achieve speedups between 1.11x and 1.33x on average.

1. INTRODUCTION

Graphical Processing Units (GPUs) are widely used for high performance computing. They provide cost-effective parallelism for a wide range of applications. The success of these devices has led to the introduction of a diverse range of architectures from many hardware manufacturers. This has created the need for common programming languages to harness the available parallelism in a portable way. OpenCL is an industry standard language for GPUs that offers program portability across accelerators of different vendors: a single piece of OpenCL code is guaranteed to be executable on many diverse devices.

A uniform language specification, however, still requires programmers to manually optimize kernel code to improve performance on each target architecture. This is a tedious

process, which requires knowledge of hardware behavior, and must be repeated each time the hardware is updated. This problem is particularly acute for GPUs which undergo rapid hardware evolution.

The solution to this problem is a cross-architectural optimizer capable of achieving performance portability. Current proposals for cross-architectural compiler support [21, 34] all involve working on source-to-source transformations. Compiler intermediate representations [6] and ISAs [5] that span across devices of different vendors have still to reach full support.

This paper studies the issue of performance portability focusing on the optimization of the thread-coarsening compiler transformation. Thread coarsening [21, 30, 31] merges together two or more parallel threads, increasing the amount of work performed by a single thread, and reducing the total number of threads instantiated. Selecting the best coarsening factor, i.e., the number of threads to merge together, is a trade-off between exploiting thread-level parallelism and avoiding execution of redundant instructions. Making the correct choice leads to significant speedups on all our platforms. Our data show that picking the optimal coarsening factor is difficult since most configurations lead to performance downgrade and only careful selection of the coarsening factor gives improvements. Selecting the best parameter requires knowledge of the particular hardware platform, i.e., different GPUs have different optimal factors.

In this work, we select the coarsening factor using an automated machine learning technique. We build our model based on a cascade of neural networks that decide whether it is beneficial to apply coarsening. The inputs to the model are static code features extracted from the parallel OpenCL code. These features include, among the others, branch divergence and instruction mix information. The technique is applied to four GPU architectures: Fermi and Kepler from Nvidia and Cypress and Tahiti from AMD. While native executing misses optimization opportunities, our approach gives an average performance improvement of 1.16x, 1.11x, 1.33x, 1.30x respectively.

In summary the paper makes the following contributions:

- We provide a characterization of the optimization space across four architectures.
- We develop a machine learning technique based on a neural network to predict coarsening.
- We show significant performance improvements across 17 benchmarks.

Prior Art

Heterogeneous Mapping

**Binary
classification**

{CPU, GPU}

Decision Tree

Decision Space

**One-of-six
classification**

{1, 2, 4, 8, 16, 32}

Model

**Cascading
Neural Networks**

Prior Art

Heterogeneous Mapping

4 features

Combined from 7 raw values.

Instruction counts / ratios.

Thread Coarsening

Features

7 features

Principle Components of 34 raw values.

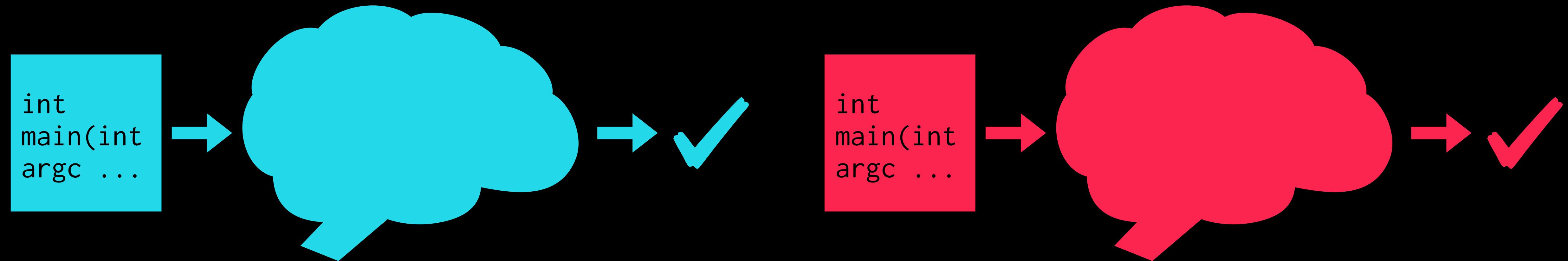
2 papers!

Instruction counts / ratios / relative deltas.

Our Approach

Heterogeneous Mapping

Thread Coarsening



1. Use the same model design for both
2. No tweaking of parameters
3. Minimum change - 3 line diff

Prior Art

Heterogeneous Mapping

**2x CPU-GPU
architectures**

7 Benchmark Suites

Hardware

Thread Coarsening

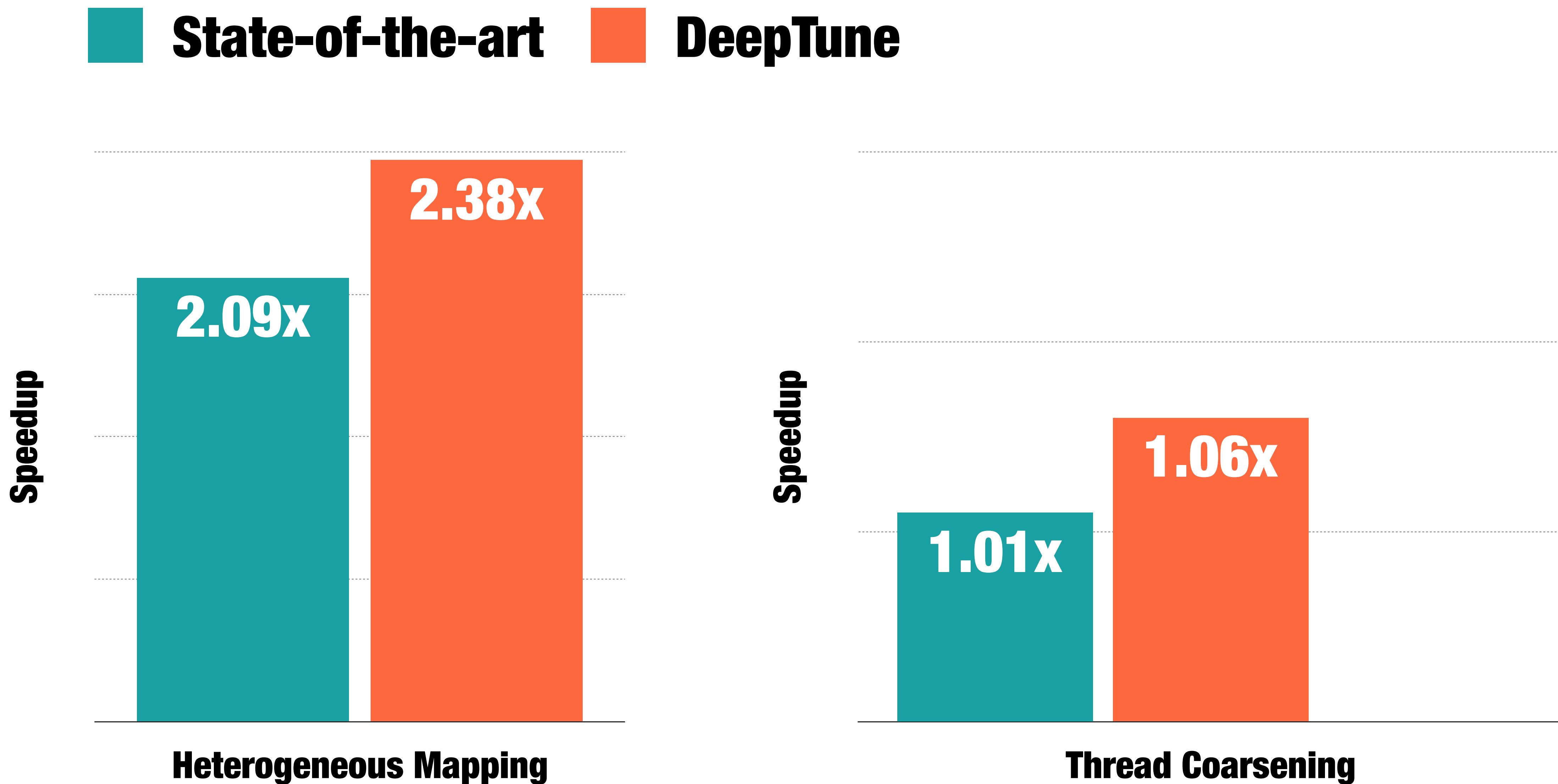
**4x GPU
architectures**

Training Programs

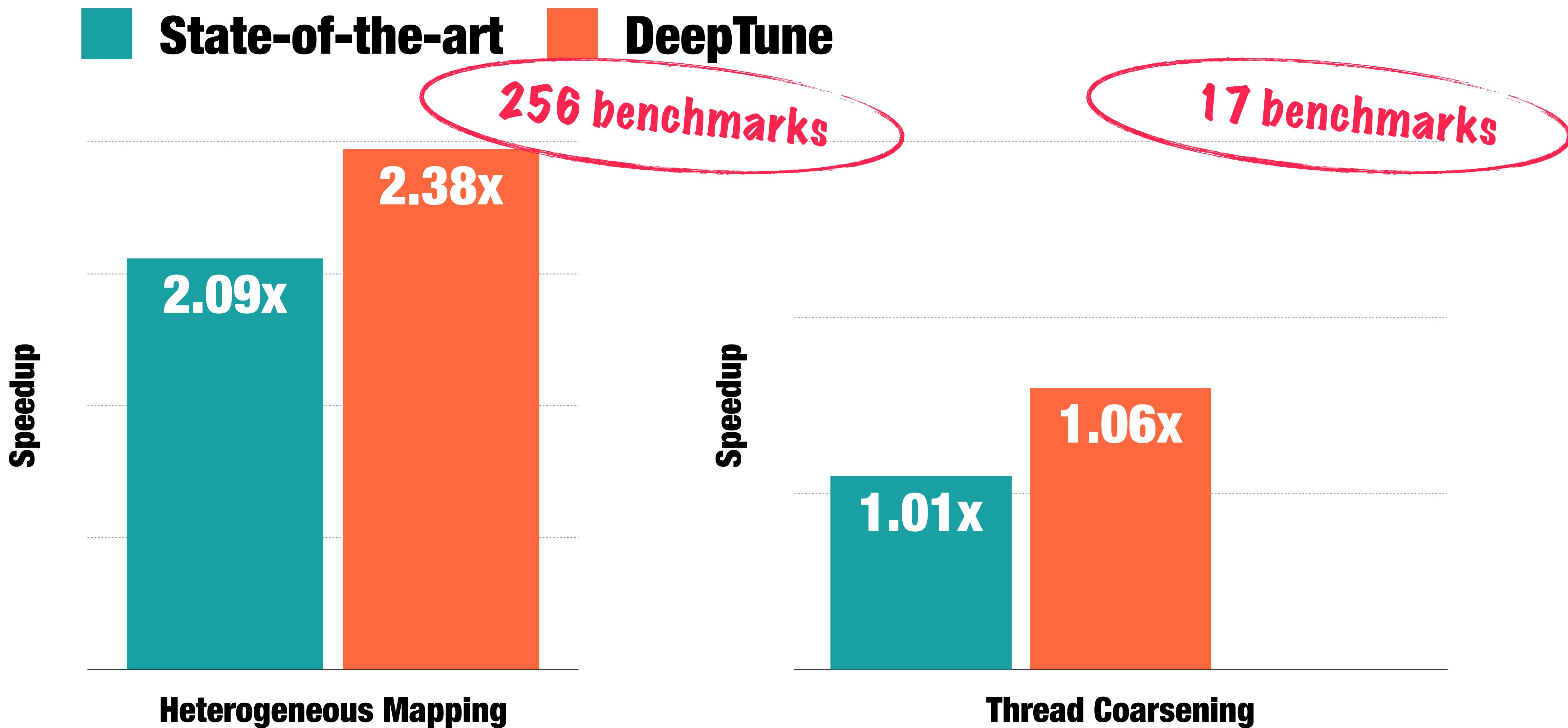
3 Benchmark Suites

results

14% and 5% improvements over state-of-the-art

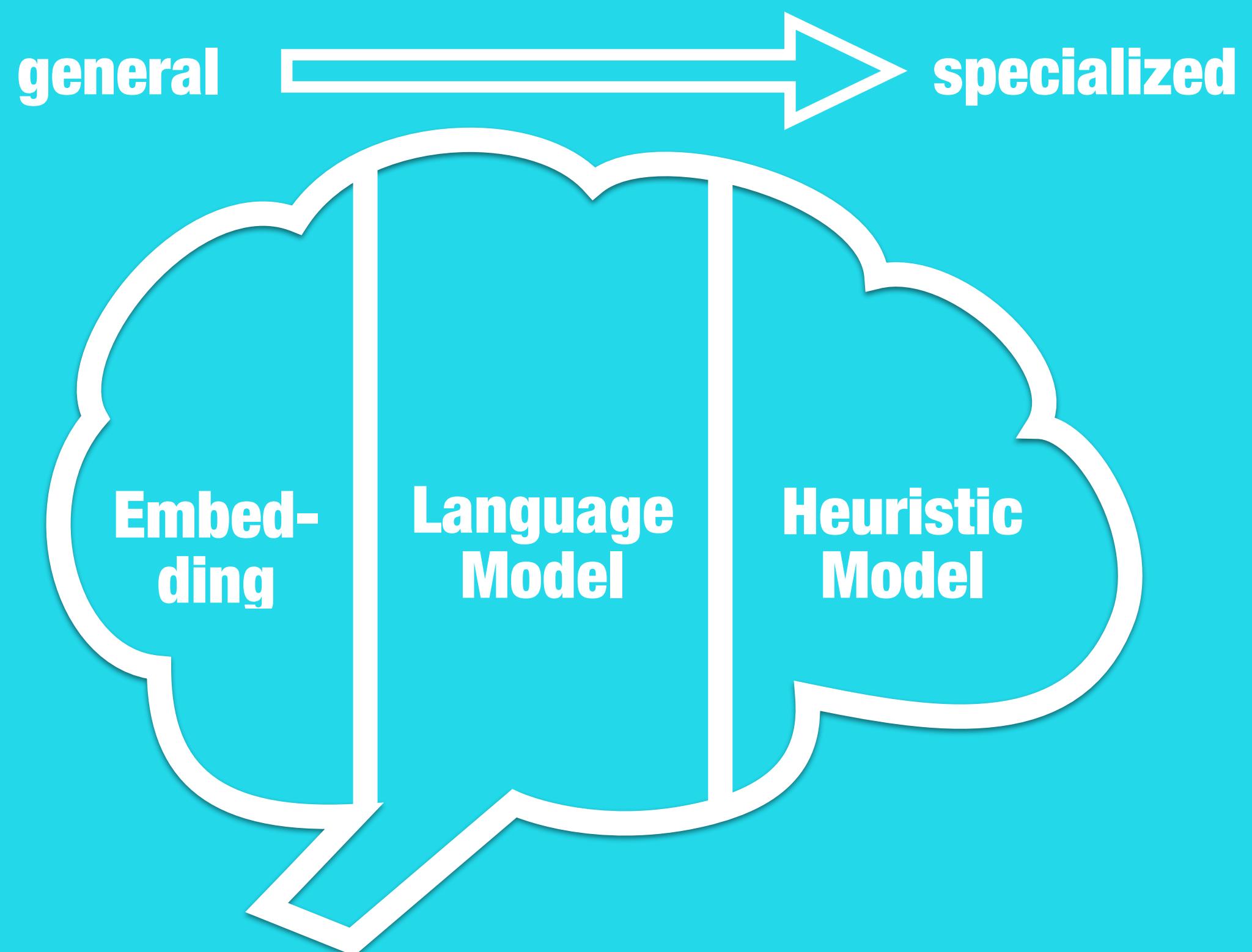


14% and 5% improvements over state-of-the-art

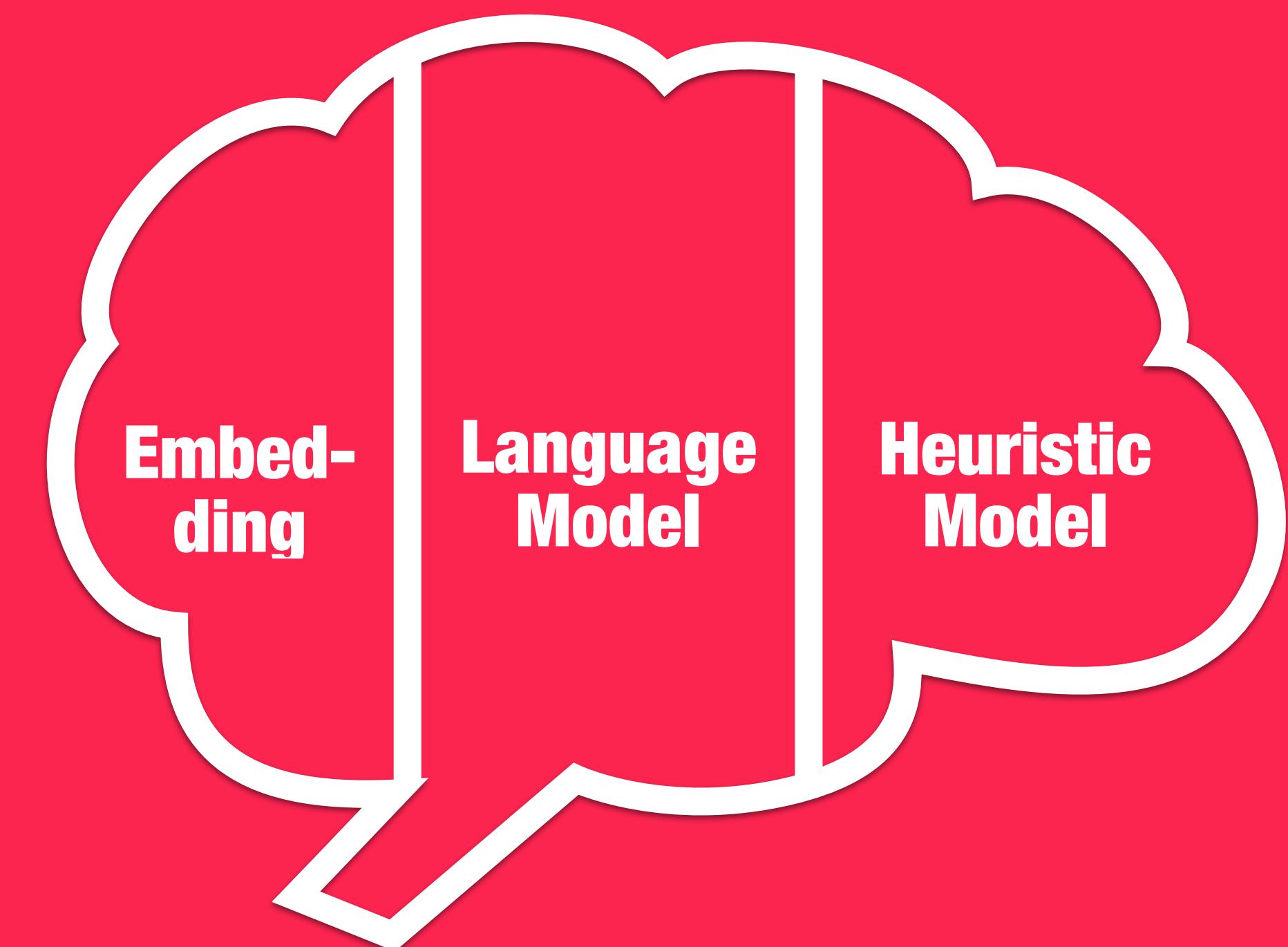


Transfer Learning

Heterogeneous Mapping

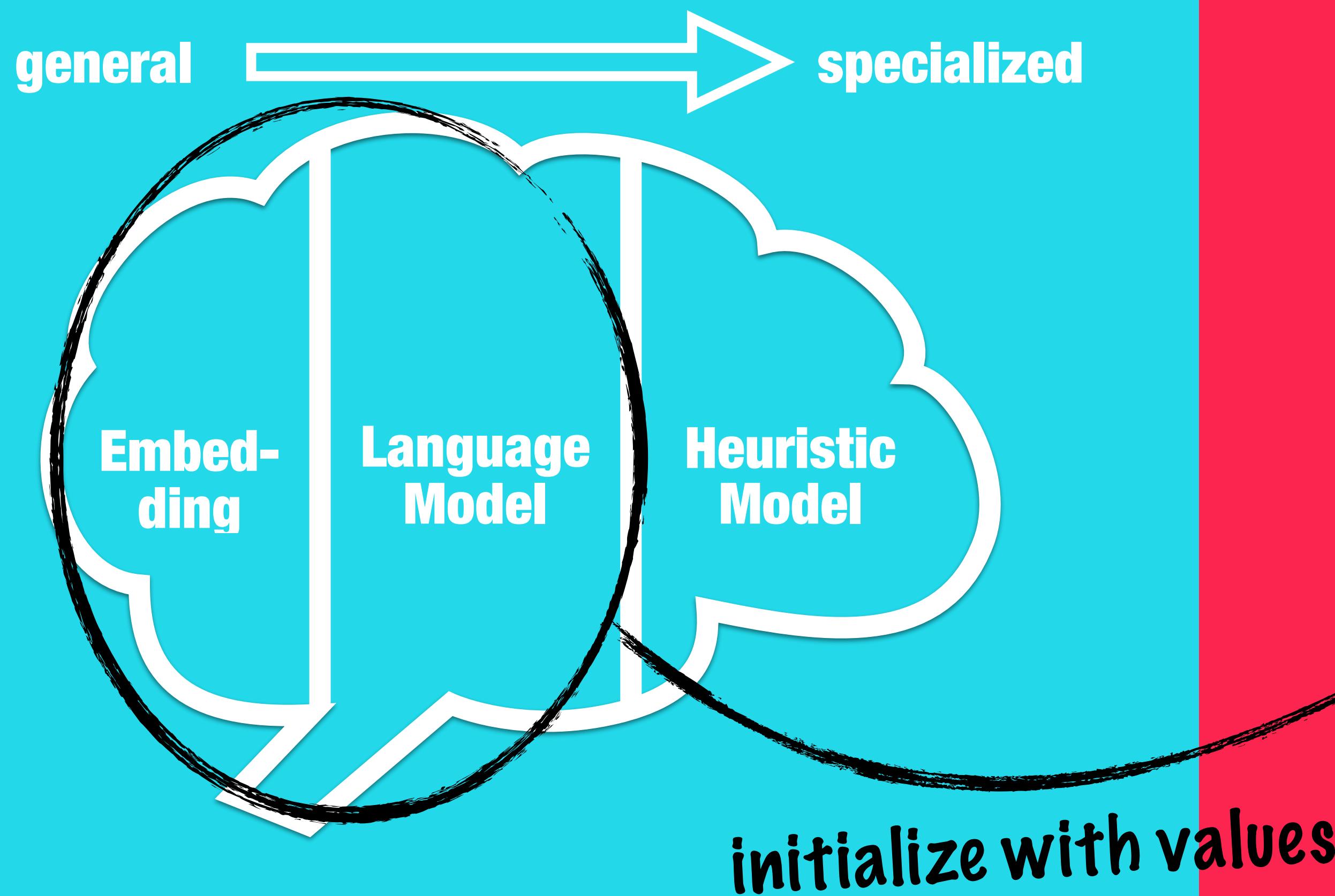


Thread Coarsening

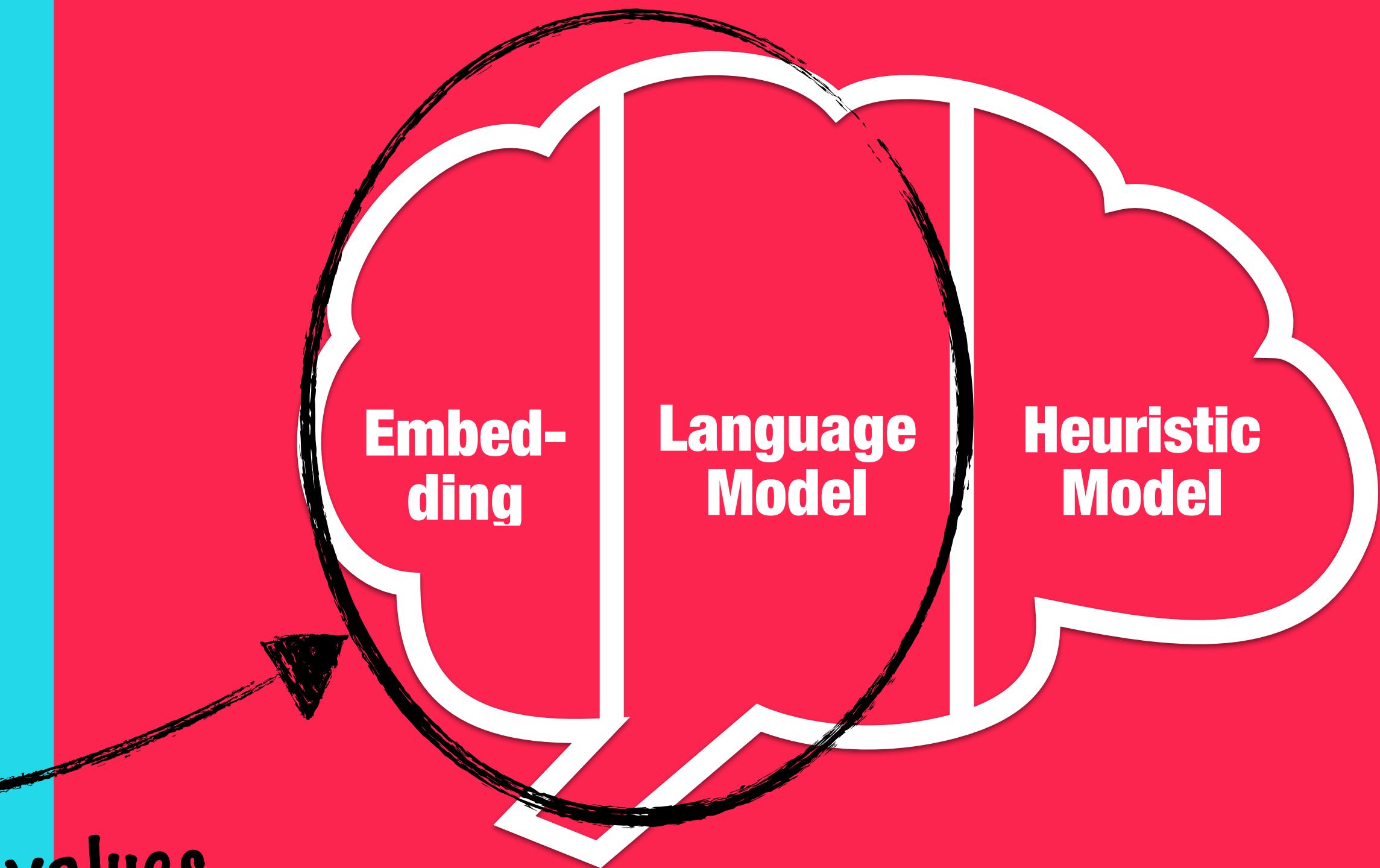


Transfer Learning

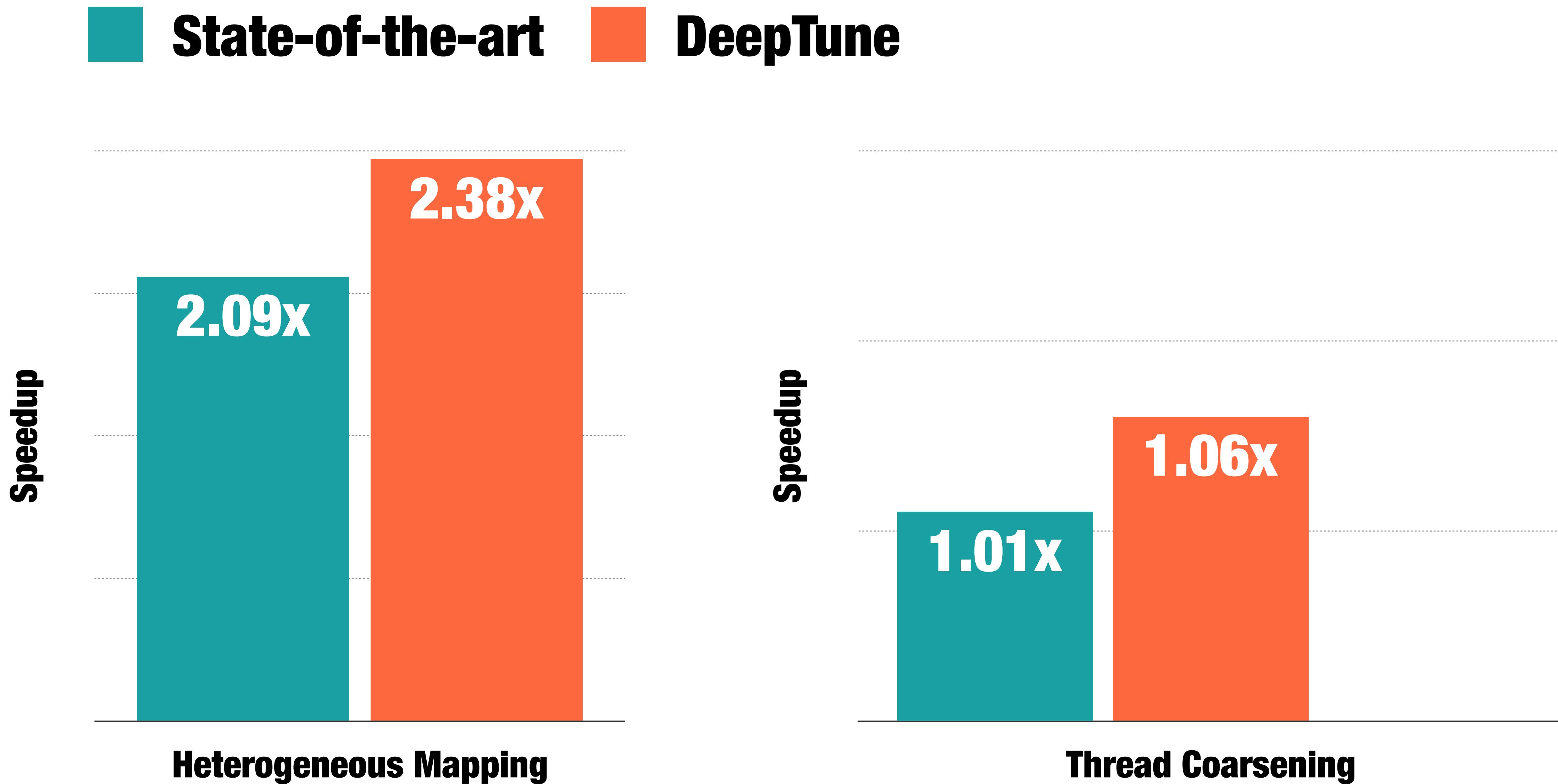
Heterogeneous Mapping



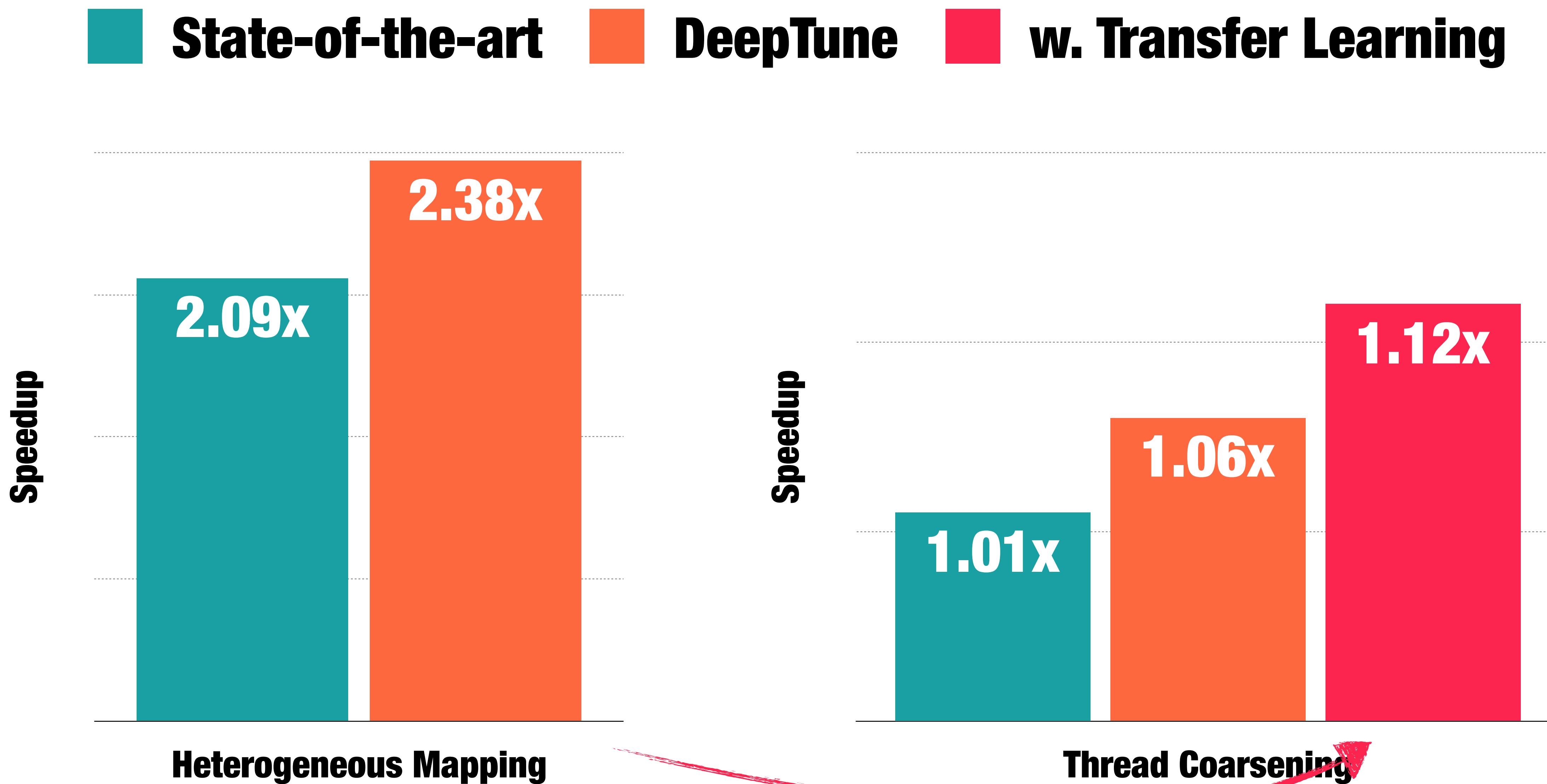
Thread Coarsening



14% and 5% improvements over state-of-the-art



14% and 11% improvements over state-of-the-art



Try it for yourself!



Fork me on GitHub

This repository

Pull requests Issues Marketplace Gist

ChrisCummins / paper-end2end-dl

Code Issues Pull requests Projects Settings Insights

"End-to-end Deep Learning of Optimization Heuristics" (PACT 2017)

deep-learning compiler-optimizations neural-network tensorflow keras publication paper autotuning machine-learning academic-publication Manage topics

19 commits 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

ChrisCummins Add image to readme Latest commit 79db1cf on Jul 18

	code	Support CentOS, Ubuntu, and Arch Linux	a month ago
code	Support CentOS, Ubuntu, and Arch Linux	a month ago	
data	Add Case Study B	2 months ago	
tex	Camera ready fixes	a month ago	
.gitignore	Initial implementation	3 months ago	
README.md	Add image to readme	a month ago	

4. Evaluation

Here we evaluate the quality of the models using two metrics: prediction accuracy, and performance relative to the static mapping:

4.1. Prediction Accuracy

```
In [19]: d = []
d.append(np.append(baseline.groupby(['Platform'])['Correct?'].mean().values * 100,
                   baseline['Correct?'].mean() * 100))
d.append(np.append(grewe.groupby(['Platform'])['Correct?'].mean().values * 100,
                  grewe['Correct?'].mean() * 100))
d.append(np.append(deepTune.groupby(['Platform'])['Correct?'].mean().values * 100,
                  deepTune['Correct?'].mean() * 100))
d = np.array(d).T.reshape(3, 3)

pd.DataFrame(d, columns=['Static mapping', 'Grewe et al.', 'DeepTune'],
              index=['AMD Tahiti 7970', 'NVIDIA GTX 970', 'Average'])
```

	Static mapping	Grewe et al.	DeepTune
AMD Tahiti 7970	58.823529	73.382353	83.676471
NVIDIA GTX 970	58.911765	72.941176	80.294118
Average	57.867647	73.161765	81.985294

Figure 6 of the paper:

```
In [20]: if isnotebook():
    import matplotlib
    import matplotlib.pyplot as plt
    import seaborn as sns

    from matplotlib.ticker import FormatStrFormatter
    from labmb import viz

    # plotting configuration
```

code and data on GitHub

runs in the browser

<http://chriscummins.cc/pact17>

End-to-end Deep Learning of Optimisation Heuristics

Problem: feature design is hard

Featureless heuristics

First cross-domain learning

11-14% speedups