Core utility development for hysia performance optimization

Zhou, Shengsheng

2019

Zhou, S. (2019). Core utility development for hysia performance optimization. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/82835>

<https://doi.org/10.32657/10356/82835>

NANYANG
TECHNOLOGICAL
UNIVERSITY

**CORE UTILITY DEVELOPMENT FOR HYSIA
PERFORMANCE OPTIMIZATION**

ZHOU SHENGSHENG

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2019

CORE UTILITY DEVELOPMENT FOR HYSIA PERFORMANCE OPTIMIZATION

ZHOU SHENGSHENG

School of Computer Science and Engineering

A thesis submitted to Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Master of Engineering

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

21-Jul-2019

.....

Date

Zhou Shengsheng

.....

Zhou Shengsheng

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

21-Jul-2019

.....
Date

.....
Wen Yonggang

Authorship Attribution Statement

This thesis **does not** contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

The contributions of the co-authors are as follows:

- Prof Wen Yonggang and Dr Liang Qianhui provided guidance in writing this dissertation.
- Mr Zhang Huaizheng helped design experiments for evaluating impact of batch size on model latencies, throughputs and memory usage.
- I designed, implemented and conducted all the experiments for model profiler and resource monitor.
- I wrote the drafts of the manuscript. The manuscript was revised together with Dr Liang Qianhui.

21-Jul-2019

.....
Date

.....
Zhou Shengsheng

Contents

Abstract	iv
Acknowledgement	vi
Acronyms	vii
Lists of Figures	x
Lists of Tables	xii
Chapter 1 Introduction	1
1.1 Background	1
1.1.1 Hysia Framework and Core Utility	2
1.1.2 Application Scenario	3
1.2 Motivations and Objectives	5
1.3 Major contribution of the Dissertation	5
1.4 Organisation of the Dissertation	6
Chapter 2 Literature Review	7
2.1 Introduction	7
2.2 Model Serving	7
2.2.1 TensorFlow Serving	8
2.2.2 Clipper	8
2.2.3 MXNet Model Server	9
2.3 Model Profiling	10
2.3.1 Static Model Profiling	10
2.3.2 Dynamic Model Profiling	10
2.4 Resource Monitoring	12

Chapter 3 Hysia Architecture	14
3.1 Introduction	14
3.2 Architecture	15
3.2.1 Data Abstraction Layer	16
3.2.2 Model Management Layer	16
3.2.3 Orchestration Layer	17
3.3 Core Utility for Hysia Framework	18
3.3.1 Model Profiler	18
3.3.2 Resource Monitor	19
3.4 Conclusion	19
Chapter 4 Model Profiler	21
4.1 Introduction and Research Questions	21
4.2 Description of the Solution	24
4.2.1 Solution to First Research Question	24
4.2.2 Solution to Second Research Question	25
4.2.3 Solution to Third Research Question	25
4.3 Model Profiler Design	26
4.3.1 Static Model Profiler	26
4.3.2 Dynamic Model Profiler	28
4.4 Evaluation	28
4.4.1 Theory for Static Model Profiling	30
4.4.1.1 Fully Connected Layer	30
4.4.1.2 Convolutional Layer	31
4.4.1.3 Recurrent Layer	31
4.4.2 Evaluation for Static Model Profiler	35
4.4.3 Methodology for Profiling Model Latencies and Through-puts	37

4.4.4	Evaluation for Dynamic Model Profiler	38
4.4.5	Impact of Batch Size to Latencies, Throughputs and Mem- ory at Runtime	39
4.5	Conclusion	43
Chapter 5	Resource Monitor	44
5.1	Introduction and Research Questions	44
5.2	Description of the Solution	45
5.3	Resource Monitor Design	47
5.4	Evaluation	48
5.4.1	Monitor Evaluation	49
5.4.2	Monitor Time Cost Evaluation	49
5.5	Conclusion	50
Chapter 6	Conclusion and Future Work	52
6.1	Conclusion	52
6.2	Future Work	53

Abstract

To serve machine learning requests with trained models plays an increasingly important role with the advance and continuous commercialization of machine learning models. Model serving is also the dominant cost in production-scale machine learning systems such as versatile prediction pipelines, complex models, diverse machine learning frameworks and heterogeneous hardware like CPU, GPU and TPU. Serving machine learning pipelines with low latencies for better user experience is the key to the success for an e-commerce product. This becomes more challenging, due to the complex constitutions of model serving, i.e. models, frameworks and hardware accelerators, to serve interactive machine learning workloads. Accessibility, cost and latency are especially challenging to be addressed. Hysia is a multi-modal machine learning model serving framework developed by our team, to remedy such challenges introduced by the complex interactions between models and hardware. Hysia framework addresses accessibility, cost and latency issues by providing easy-to-use application interfaces and an intelligent controller which jointly optimizes performance to balance the trade-off between resource consumption and prediction accuracy.

This thesis focuses on the design, implementation and benchmarking of the core utility for Hysia framework, i.e. to provide profile information about models and statuses about system resources in order to optimize machine learning pipelines. The core utility plays a significant role for the joint system performance optimization for Hysia. Model profiler and resource monitor form the core utility. The model profiler is designed to profile machine learning models to get their statistics like parameters, memory usage and inference latency. Our design for model profiler unifies the differences among various machine learning

platforms and ensures extensibility. The resource monitor is used to monitor the system resource status like memory and GPU utilization. Our resource monitor is capable to retrieve rich system statuses. Both model profiler and resource monitor are designed in a distributed way to improve efficiency and support distributed computation.

Keywords: Dissertation, Machine Learning Model Serving, Model Profiling, Resource Monitoring.

Acknowledgement

First of all, I would like to thank my supervisor Prof Wen Yonggang for giving me the environment and supervision to learn and grow. I would also like to thank my co-supervisor Prof Liang Qianhui for giving me advice and comments on writing this dissertation. In doing the experiments for this dissertation, my group mate Zhang Huaizheng gave me lots of insights and recommendations. I wish to thank him for his tireless help. I appreciate the time with all of the team members at DMAL and thank all of them to share their happiness with me.

Acronyms

AI Artificial intelligence

QoE Quality of Experience

CNN Convolutional Neural Network

RCNN Region-based Convolutional Neural Network

YOLO You Only Look Once

FLOP Floating Point Operation

CPU Central Processing Unit

GPU Graphics Processing Unit

TFS TensorFlow Serving

MMS MXNet Model Server

RNN Recurrent Neural Network

ERU Elman Recurrent Unit

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

PR50 Place365 scene classification model with ResNet50

SA-LSTM PyTorch sentiment analysis LSTM model

USE Universal sentence encoder model

List of Figures

Figure 3.1	Hysia framework architecture. The architecture follows a layered-design and comprises of four layers: <i>API Layer</i> , <i>Data Abstraction Layer</i> , <i>Model Management Layer</i> and <i>Orchestration Layer</i>	15
Figure 4.1	Model profiler architecture. <i>Profiler</i> splits <i>ProfileTask</i> into <i>StaticProfileTask</i> and <i>DynamicProfileTask</i> for <i>StaticProfiler</i> and <i>DynamicProfiler</i> to process and produce <i>StaticProfileResult</i> and <i>DynamicProfileResult</i> which are saved into <i>ModelDB</i> for future uses.	27
Figure 4.2	Dynamic model profiler architecture. <i>DynamicProfiler</i> dispatches <i>DynamicProfileTask</i> to <i>DynamicProfileWorker</i> to perform the profile task and collect <i>DynamicProfileResult</i> from it through Redis. Result is finally saved into <i>ModelDB</i> for future uses.	29
Figure 4.3	Impact of batch size to latencies for PR50, SA-LSTM and USE models with details listed in table 4.6.	41
Figure 4.4	Impact of batch size to throughputs for PR50, SA-LSTM and USE models with details listed in table 4.6.	42
Figure 4.5	Impact of batch size to memory for PR50, SA-LSTM and USE models with details listed in table 4.6.	42

Figure 5.1	Resource monitor architecture. <i>MonitorWorker</i> running on each node collects system resource statuses and sends them back to <i>Monitor</i> through Redis. Other components (e.g. intelligent controller) can easily query system statuses from <i>Monitor</i> or Redis.	48
Figure 5.2	Evaluation results for resource monitor including system boot time, CPU status, memory status, GPU status, process status, GPU process status and network status.	50

List of Tables

Table 4.1	Detailed explanation for static model profile information including parameters, memory, FLOPs, memory read, memory write and memory readwrite.	22
Table 4.2	Detailed explanation for dynamic model profile information including ip, device, batch size, latencies (initialization, pre-processing, inference and post-processing) and throughputs (pre-processing, inference and post-processing).	23
Table 4.3	Specification including CPU, memory, GPU and system information for two servers running model profiler evaluations. . .	36
Table 4.4	Static profile results for various CNNs with engine <i>TORCH</i> , input size 241x241x3 and batch size 1.	36
Table 4.5	Static profile results for various CNNs with engine <i>TF</i> , input size 241x241x3 and batch size 1.	37
Table 4.6	Model names, underline engines and descriptions for the three models used to evaluate the dynamic model profiler.	39
Table 4.7	Sample dynamic profile results produced by evaluating the dynamic model profiler for models listed in table 4.6 on two servers listed in table 4.3.	40
Table 5.1	Descriptions for fields of CPU status including model, count, freqs, cache, loads, utilization, times and timesRatio.	45
Table 5.2	Descriptions for fields of main memory status including total, available, utilization, free, used, shared, buffered, cached, totalSwap, usedSwap, freeSwap and swapUtilization.	46

Table 5.3	Descriptions for fields of GPU status including gpuId, uuid, model, totalMemory, usedMemory, freeMemory, memoryUtilization and temperature.	47
Table 5.4	Specification including CPU, memory, GPU and system information for server running the monitor evaluation.	49
Table 5.5	Time cost evaluation results for query each status from monitor on a single machine with specification listed in table 5.4. . . .	51

Chapter 1

Introduction

1.1 Background

Artificial intelligence (AI) acts as the second form of electricity, powering more and more companies, organizations and individuals to build intelligent systems and to deliver state-of-the-art performances. Machine learning and deep learning play the dominant part of AI due to their recent advances in both research and industries. Increasing real-world industry-grade applications (e.g. Amazon recommendation system, Apple Siri voice assistant and Google translation service) resort to machine learning techniques to achieve the best quality of experience (QoE) of end users. Success of these applications substantiates the importance and significance of machine learning.

While training machine learning models have been extensively investigated [1]–[4], serving these models are less studied. With the increasing quantity and complexity for both machine learning models and machine learning workloads, challenges of serving models in an accessible, energy-efficient and latency-aware way emerge. Let us consider a typical machine learning task – scene product detection – as an example. Given a video file, the scene product detection task is responsible for detecting the products in every scene shot of the video. To achieve the best detection quality, usually we would like to resort to multi-modal learning which may involve scene recognition, object detection, face detection, audio

detection and text detection tasks. Even worse, each recognition or detection task has numerous models to use. For example, Faster-RCNN [5], Mask-RCNN [6] and YOLO [7] are all capable to perform the object detection task. How to choose, serve and orchestrate these models presents challenges. First, how to choose models among different categories of these sub-tasks is difficult for non-experts which is an accessibility challenge. Second, running of these models often needs multiple GPUs which incur energy overhead. And finally, how to choose appropriate models to perform the scene product detection as fast as possible while preserving the desired accuracy reveals the serving latency challenge.

1.1.1 Hysia Framework and Core Utility

To address the above-mentioned issues – accessibility, energy cost and serving latency – our team proposed the Hysia framework. Hysia is a multi-modal machine learning model serving framework. It selects and manages machine learning models to help non-experts deploy the models in a straightforward way and to form their own prediction pipelines. One key contribution of Hysia framework is its intelligent controller which optimizes the workflow and orchestrates models to achieve user desired prediction accuracy while keeping the energy cost as low as possible. In order to perform the joint performance optimization for the workflow and model orchestration, we need a core utility to collect the key information about models and system resources for Hysia framework.

The core utility for Hysia includes model profiler and resource monitor. Model profiler aims to profile a model to retrieve its statistic information like total parameters, memory consumption, floating point operations (FLOPs) [8], inference latencies on different GPUs and so on. While the resource monitor monitors the system and hardware statuses to get information including CPU utilization, GPU

utilization, memory statistics, process information and others. The core utility plays a significant role in the joint system performance optimization as it provides critical information for the intelligent controller.

1.1.2 Application Scenario

To demonstrate the contribution of model profiler and resource monitor, we would like to take a real world application scenario – large scale video product detection – as an example to see what challenges there are and how model profiler can help to address these challenges. A video product detection application aims to detect products in every scene shot of a video file. It is costly for two reasons. First, every frame needs to be retrieved from the video and passed to object detection model. Second, object detection model itself is costly because it often endows deep learning techniques with very deep networks (e.g. Faster-RCNN model [5]). What makes matters worse is that we want to deploy the application in a large production scale where there could be thousands of instances running concurrently. This is often faced by cloud service provider as millions of users are uploading their videos for processing. The detection accuracy and resource consumption becomes the key factors of our concerns. Can we devise ways for our video product detection application to run with as less resource as possible while preserving the desired accuracy to save the cost and maintaining the quality of service?

In fact, different object detection models offer different accuracies as well as resource consumption. For instance, a Faster-RCNN-ResNet-101 model is generally more accurate at detecting objects but also is more resource-consuming compared with a Faster-RCNN-ResNet-50 model. However, a video is usually composed with different scenes where objects have different distributions. For

example, an indoor office usually contains more objects (e.g. desks, computers and cups) than a road scene at night where there are only several cars. If we use the more powerful Faster-RCNN-ResNet-101 model throughout the whole video, we definitely can get the best detection accuracy as the model itself provides satisfying accuracy. But it also leads to consuming more computation resources and is slower. Instead, if we use the less powerful Faster-RCNN-ResNet-50 model to detect the whole video, we can save a lot of resources to process the video faster but at the cost of lowering the prediction accuracy. The challenge is if there is any method to preserve or nearly preserve the detection accuracy as well as save resource? An interesting fact is that a more powerful model (such as the Faster-RCNN-ResNet-101 model) performs nearly the same compared to a less powerful model (such as the Faster-RCNN-ResNet-50) on a scene image where there are few objects [9]. If we can apply Faster-RCNN-ResNet-101 model on complex scenes (such as the indoor one) and Faster-RCNN-ResNet-50 model on simple scenes (such as the road one), we could save resource consumption while keeping detection accuracy for our video product detection application thus saving a big margin in large production scale.

In order to achieve such goals, the complexity (e.g. FLOPs and memory) and performance (e.g. accuracy, latencies and throughputs) for a model and system status of the whole cluster are of key importance. The model profiler validates and profiles models to get their statistic information such as FLOPs, memory, latencies and throughputs. The resource monitor monitors and collects system status of the cluster. With such information, it is doable to orchestrate models in an accuracy-consumption balanced way.

1.2 Motivations and Objectives

This dissertation focuses on the design, implementation and benchmarking for the core utility for Hysia framework. Hysia provides an intelligent controller which reads information from core utilities including model profile results and system resource status and outputs optimized model selection and execution strategy towards the best prediction quality with lowest resource consumption. Our motivation is simple as it is critical to generate sufficient information of both machine learning models and system resources as inputs to the intelligent controller. In order to generate sufficient information for the controller in a fast and energy-friendly way, the design and implementation for the core utility remains challenging. Thus the objectives include 1) designing the core utility for Hysia including model profiler and resource monitor to collect sufficient information for the joint system performance optimization of the intelligent controller and 2) designing and implementing the core utility to execute in a fast and resource-aware manner.

1.3 Major contribution of the Dissertation

1. We designed and implemented model profiler to profile machine learning models and resource monitor to monitor system resource statuses to help Hysia framework to orchestrate models in an accuracy-consumption balanced way.
2. We reviewed literature and deduced theories for calculating FLOPs and parameters for various neural network layers.
3. We evaluated the model profiler on various machine learning models and

visualized the impact of batch size to model latencies, throughputs and memory usage, which gives insights to others to better orchestrate machine learning models efficiently in production.

1.4 Organisation of the Dissertation

The dissertation is organized as follows:

1. Chapter 1 introduces Hysia framework and its core utility and gives motivation and objectives for designing and implementing the utilities.
2. Chapter 2 presents some related works in literature.
3. Chapter 3 gives a brief overview of the architecture of Hysia and introduces its core utilities including model profiler and resource monitor.
4. Chapter 4 demonstrates the function, design, implementation and evaluations for the model profiler.
5. Chapter 5 elaborates the function, design, implementation and evaluations for the resource monitor.
6. Chapter 6 draws a conclusion for the dissertation and introduces ideas for future work.

Chapter 2

Literature Review

2.1 Introduction

This chapter gives a brief literature review related to our work. The literature review contains three parts:

- Model serving: Since Hysia is a model serving framework, we want to give some reviews on related model serving works.
- Model profiling: Some model profiling works, techniques, open source tools will be discussed.
- Resource monitoring: This part reviews some open source tools which monitor different system resources.

2.2 Model Serving

Machine learning models have been deployed extensively to meet the increasing intelligent workloads. Applications driven by machine learning including image classification, recommendation, speech recognition and natural language processing have made remarkable performance improvements with the help of machine learning [5], [10]–[12]. A machine learning task usually has two phases: *training* and *serving*. Training is to build the model with the help of data (e.g. labeled

images). Serving is the process of deploying the trained model and making predictions through the model. While training has been investigated by many works [1]–[4], serving has less been examined. This section reviews some of the works focusing on serving machine learning models.

2.2.1 TensorFlow Serving

TensorFlow Serving (TFS) [13] is the official serving framework provided by TensorFlow team. TFS provides state-of-the-art inference performance among all the alternatives. A *Servable* represents a TensorFlow model that can be served for inferences. Some key components for TFS include *Servable*, *Source*, *Loader*, *DynamicManager* and *ServableHandle*. Generally speaking, *Sources* create *Loaders* for versioned *Servable* and *Loaders* are sent to the *Manager* which loads and serves them for client requests. TFS guarantees high performance, stability, extensibility for model serving and constant improvements and maintenance with a strong team inside Google. The drawback is that only machine learning models written with TensorFlow framework can be served with TFS, which may not be a good point to generalize.

2.2.2 Clipper

Clipper [14] is a model serving framework used to serve machine learning models built with heterogeneous machine learning frameworks (e.g. TensorFlow, MXNet and PyTorch). Clipper provides containerization mechanism to pack and isolate trained models into model containers. The containerization unifies the differences between different machine learning frameworks. Clipper provides RESTful APIs [15] for high level applications to integrate. Given a prediction request, the model

selection layer will select the most suitable model or models with the help of model selection policy. Then the model selection layer will call model abstraction layer which provides a unified interface for the underlying model containers. The model abstraction layer provides caching and adaptive batching which are common techniques to build high performance machine learning inference pipelines. One key contribution of Clipper is its model selection layer. The model selection policies are categorized into single model selection policies and ensemble model selection policies. If the system resources are tight, to lower the consumption of the remaining resources the single model selection policies are to be used. Otherwise if the resources are sufficient, the ensemble model selection policies are to be used to achieve better prediction accuracies.

2.2.3 MXNet Model Server

MXNet [2] is a machine learning framework incubated by Apache. MXNet model server (MMS) [16] is an easy to use model serving framework created by Amazon to serve deep learning models written for MXNet or exported from the Open Neural Network Exchange (ONNX) format [17]. The ONNX is a universal open format to describe deep learning models. ONNX acts as a bridge between state-of-the-art machine learning tools. MMS framework is very popular, efficient and easy to use. MMS comprises of three modules: model archiver, model server and frontend. The model archiver provides a simple command line interface for end users to pack their models and upload packed models to *model store* of MMS. It also supports packing ONNX models. With the support for ONNX, one can easily enrich models created from other machine learning platforms to MXNet format thus use these models with MMS.

2.3 Model Profiling

We would like to categorize model profiling tasks into two categories: static model profiling and dynamic model profiling. Static model profiling stands for profiling the model to get its static characteristics like total parameters, memory usage when loaded and FLOPs. Static profiling is hardware-independent thus once and for all. While dynamic model profiling targets the runtime statistics for a model like inference latency on a specific GPU. Dynamic profiling thus is hardware-dependent.

2.3.1 Static Model Profiling

Several open source tools are created to profile models to get their static statistics. TensorFlow Profiler [18] developed and maintained by the official TensorFlow team is used to profile models written with TensorFlow framework to retrieve their static information including parameters, memory, FLOPs and even those characteristics for each neural layer. TorchStat [19] is a third party tool to profile PyTorch models and can be used to get their parameters, memory, FLOPs, memory read and memory write and also supports retrieving these information for each neural layer. Convnet-burden [20] is a project which analyzes various classical and popular convolutional neural networks to show their parameters, memory consumption, FLOPs as well as performance.

2.3.2 Dynamic Model Profiling

While static profiling is once and for all for a machine learning model, dynamic profiling must be re-run once the environment (e.g. OS, CPU, GPU and memory)

changes. Even though the static profiling can retrieve pretty much useful information describing a model, it does not reveal the real performance at runtime. Dynamic profiling can come to the rescue.

In the InferLine paper [21], the author proposes a profiling approach to profile models at runtime. The idea is straightforward, i.e. given model, batch size and input data, to evaluate the model and test its inference throughput. Generally speaking, the profiler captures model throughput as a function of device and batch size. The device type and batch size provide mechanisms to detect the potentials to the model parallelism. The profiler works in two ways: First, it is sufficient to only profile a single replica of a model since different replicas of a same model share the equivalent characteristics in terms of inference throughput given same hardware (i.e., CPU, memory and GPU) and software (i.e., OS) conditions; Second, to prevent linear search over determining the maximum batch size for a model, the profiler increases the batch size in powers of two. Further more, to ensure that the model profiling result can be used even when the machine is fully-utilized, they simulate a load on the machine by running more replicas during the profiling process. This is a better way to profile with the effects of system kernel level dispatching brings than profiling on an idle machine.

Nexus [22], a joint research project between University of Washington and Microsoft Research, aims to deliver an open source scalable and efficient model serving system for deep learning applications on GPU cluster. Machine learning workloads are often complex as they incorporate not only inference but also pre-processing and post-processing. In order to deliver the optimized serving quality, it is also of high importance that the profiler can profile not only inference latencies but also pre-processing and post-processing latencies. Nexus profiler implemented these. The profiler takes model, batch size, input data and device as inputs, profile the model with designated batch size and input data on designated

device to get its minimal, maximal and average pre-processing, inference and post-processing latencies. These information is richer compared to the InferLine profiler's implementation. The Nexus profiler also proposes a possible way to determine the memory usage for a model by capturing the GPU memory usages before and after the model has been loaded. This method has its drawback as usually the GPU resources are shared across multiple applications. It might not be a good idea to profile the memory usage for a model in this way but it is applicable if one can guarantee that during profiling the GPU resources are exclusively used by the profiler.

2.4 Resource Monitoring

Resource monitoring is a simple yet very important task towards the joint performance optimization for various machine learning serving systems. Many open source libraries have been devised to monitor system resources.

psutil [23], a Python library, divides resources into two categories: system related resources and processes. For the system resources, *psutil* provides functions to retrieve CPU, memory, disks, network, sensors and other system information. An example would be total, used, free, and cached memory for system memory information. And for the processes, it can be used to get all of the running processes information like pid, name, CPU times, threads, status and so on. *psutil* is the most popular yet powerful open source tool for monitoring system resources and processes. It is cross-platform supporting Linux, Windows, macOS and many other UNIX variants.

py-cpuinfo [24] is another open source Python library which can be used to retrieve elaborate CPU information, e.g. advertised hertz, actual hertz, architecture,

bits, cache size and flags. To optimize the performance, *py-cpuinfo* mixed some assembly codes in its underlying implementation. Same as *psutil*, *py-cpuinfo* also provides easy to use application interfaces and rich documentation. The development is also highly active among its community.

GPUtil [25] is devised to fill the vacancy that *psutil* and *py-cpuinfo* left, which is to monitor GPU status. As its name reveals, *GPUtil* is a Python utility package to query GPU information as well as GPU process information. GPU information includes GPU count, GPU models, memory utilization and power consumption. GPU process information includes pid, name and GPU memory usage. The results are same from the *nvidia-smi* [26] command line tool as the *GPUtil* relies on the *nvidia-smi* by simply wrapping the results returned by it. To be clarify, the original *GPUtil* is only capable for printing the GPU information. To enhance the *GPUtil* as well as make it more general, I have forked the original Github repository and made some commits towards it [27].

The above-mentioned open source tools are all written in Python. They eliminate the difference among various operating systems and provide clean and simple APIs. But in a resource-aware and latency-aware situation, we might also want to use system-level APIs to monitor resources directly. Blogs, tutorials and documentations have presented ideas and methods to deal with low-level system APIs to efficiently query system resource information. For example, [28], [29] show how to get name, user time, system time, various memory usages and other information of a process on Linux system. [30] talks about how to get GPU information and GPU process information with NVIDIA Management Library (NVML).

Chapter 3

Hysia Architecture

3.1 Introduction

Hysia is a multi-modal machine learning model serving framework designed to simplify the workloads and optimize inference pipelines for Video-to-online (V2O) applications. The traditional way to build a V2O application (e.g. video scene classification) would be 1) selecting suitable models among various available ones, e.g. YOLO [7] and Faster-RCNN [5] models for visual predictions and AENet [31] and SoundNet [32] for audio predictions, 2) programming the pipelines with the help of selected models to achieve the desired goals (e.g. to classify different scenes among a video) and 3) deploying the programmed pipelines along with the selected models for future inferences. These steps often need a machine learning expert who knows everything about machine learning. Each step also requires tons of time whether it is to select models or to debug the program. This kind of method not only does not capture the complexity of machine learning workflows but also is at a distance to non-experts. It is costly to build and maintain because it needs machine learning experts to decide which models to use and program the workflow. It also tightly couples the models with the application which makes it hard to update or replace models in the future. To address these issues, we propose Hysia framework to automatically select and manage machine learning models which makes non-experts easily use these models while focusing on the workflow. Hysia also optimizes the machine learning workflow intelligently

in order to save cost and improve prediction accuracy. This chapter details the architecture design and various components of Hysia framework.

3.2 Architecture

Graph 3.1 shows the architecture design for Hysia framework. The framework comprises of four layers: *API Layer*, *Data Abstraction Layer*, *Model Management Layer* and *Orchestration Layer*. The *API Layer* provides unified and easy to use application interfaces for users to manage machine learning models and interact with models for inferences.

Figure 3.1: Hysia framework architecture. The architecture follows a layered-design and comprises of four layers: *API Layer*, *Data Abstraction Layer*, *Model Management Layer* and *Orchestration Layer*.

3.2.1 Data Abstraction Layer

Data serialization, deserialization, copying among devices (e.g. disks, main memory and GPU memory) are common performance bottlenecks in distributed systems. There are numerous data serialization libraries for different machine learning frameworks or for different data types. To name a few, *NumPy* [33] is a well-known numerical computation tool designed for high performance numerical computation in Python and is also an important building block for various machine learning frameworks [1]–[3]. *Pillow* [34] is a Python imaging library used in many open source and commercial projects. *TFRecords* [35] provided by TensorFlow team helps reading data efficiently by serializing data and storing it in a set of files which can be read linearly. Data streaming over a network as well as caching pre-processed data benefit a lot from *TFRecords*. *Data Abstraction Layer* aims at abstracting data representation across various data serialization libraries to provide a unified access to data. Another important feature of this layer is caching inference results for multi-modal inferences to balance the inference time between different models. For a scene classification task, a multi-modal classifier comprises of visual model, audio model and text model. Visual model is known as the slowest one among these three types of models while the text model runs fastest. *Data Abstraction Layer* aligns results from these three models for post-processing (i.e., to inference final results from these intermediate results) which significantly reduces IO operations.

3.2.2 Model Management Layer

Model Management Layer provides functionalities for registering and uploading models into Hysia *ModelDB* and checking their status. *ModelDB* is Hysia’s

database for storing models (i.e., model definitions, weights and profiling results). This makes user's life easier as managing so many machine learning models manually while providing fast access to them is a painful task. When registering and uploading a model, the user specifies the model's definition, weights and metadata (e.g. model name, author, category and underlying framework). Hysia will take on the responsibility to manage the model once registered and uploaded. The *Model Profiler* which profiles models to get their statistics (e.g. parameters, FLOPs and inference throughput) for later model orchestration optimization is also a part of this layer. Another important aspect in terms of providing high performance model inference pipelines and isolating models for best deployment quality is through containerization. This layer is capable of containerizing models for unified model inference interface and isolation.

3.2.3 Orchestration Layer

Orchestration Layer is at the heart of Hysia framework. It is the key component that makes Hysia distinguish it among various machine learning model serving frameworks. It provides an intelligent *Controller* that is responsible for controlling the whole machine learning workflow from pre-processing, inference to post-processing to get the best or user desired accuracy while keeping the hardware resource consumption as low as possible. The *Controller* takes care of several aspects to balance the trade-off between accuracy and latency. First, different batch sizes for a model result in different hardware utilization. Increasing the batch size improves the hardware utilization, while increases the waiting time to generate such a batch. Second, applying multi-model parallelism and distributing multiple models across devices improve the inference speed but it also puts communication overhead higher. Third, ensembling models may improve

accuracy but requires more computation resources. Once the *Controller* devised a *configuration* (e.g. models, batch sizes and parallelism strategies) for model orchestration, the *Scheduler* schedules models according to the *configuration* to complete jobs. Models are orchestrated in a distributed fashion to achieve the best utilization among multiple devices across multiple machines. The *Resource Monitor* monitors system, hardware and process status. These statuses are key inputs towards the joint performance optimization of Hysia’s intelligent *Controller*.

3.3 Core Utility for Hysia Framework

Model Profiler and *Resource Monitor* compose the core utility for the Hysia framework. The most noticeable contribution of Hysia is its intelligent controller that optimizes the workflow and orchestrates models to complete user defined jobs in an accuracy-cost balanced way. The model statistics and system resource status are parts of the key inputs for the intelligent controller.

3.3.1 Model Profiler

Model Profiler is a component of *Model Management layer*. After a model has been registered and uploaded to *ModelDB*, the *Model Profiler* is responsible for profiling the model to get its static and dynamic profiling results. The static profile results for a model include total parameters, memory needed to be loaded, FLOPs, memory read needed for an inference, memory write needed for an inference and memory readwrite (which is simply the summation of memory write and memory read). The dynamic profile results for a model contain memory usage, latencies(initialization, pre-processing, inference and post-processing) and

throughputs (pre-processing, inference and post-processing) given specific batch size and input data on a targeting device. Each type of initialization, pre-processing, inference and post-processing latencies and throughputs has min, max and average so that the performance for a model is better depicted. The dynamic information measures a model in a device-dependent way to achieve the best accuracy to describe the performance on different devices across different machines. In a distributed setting, there are usually many computers or machines involved. And each machine may have one or more devices such as CPUs and GPUs. To ensure the performance of the model profiler of its own with low resource consumption, the dynamic profiler endows a distributed way to dispatch profiling tasks to machines and collect results profiled by them. Chapter 4 will go into details of the *Model Profiler*.

3.3.2 Resource Monitor

Another key input to the intelligent controller is system resource status. The *Resource Monitor* also runs in a distributed manner. Each node runs a monitor process to monitor the system resources and report to cluster master node periodically. The system resource status comprises of running time from system booting, CPU status, memory status, process status, GPU status, GPU process status and network status. Chapter 5 will talk about *Resource Monitor* in details.

3.4 Conclusion

This chapter delivered the architecture of Hysia framework. Its main components including *API Layer*, *Data Abstraction Layer*, *Model Management Layer* and *Orchestration Layer* were introduced in detail. Following that the core utilities

including model profiler and resource monitor were also introduced. In the following two chapters, *Model Profiler* and *Resource Monitor* will be discussed in great detail respectively.

Chapter 4

Model Profiler

4.1 Introduction and Research Questions

Model profiler acts as a key role in generating input data for performance optimization for Hysia framework. Model statistics information are categorized into static and dynamic. Static information describes hardware-independent characteristics for a model including its total parameters, memory, FLOPs, memory read, memory write and memory readwrite. A detailed explanation for each field is listed in the table 4.1. In fact, although the parameters for a model is fixed all the time, different shape of input data brings different memory usage and FLOPs. This issue can be addressed by fixing the input shape for a model in a machine learning pipeline (e.g. always use image data with shape $241 \times 241 \times 3$ for a ResNet [36] image classification model). Dynamic information refers to hardware-dependent characteristics when the model is loaded and running inferences, which contains IP address of the machine, device (e.g. CPU, GPU:0, GPU:1) associated with a dynamic profile task, batch size, memory usage for loading the model and executing inference for data with given batch size, latencies (initialization, pre-processing, inference, and post-processing) and throughputs (pre-processing, inference, and post-processing). The IP address and device identify the machine and hardware in the cluster. The memory usage metric reflects the runtime memory consumption. The min, max and average latencies and throughputs describe the performance of the model. We do not collect the throughput of

Field	Description
parameters	Total trainable or fixed parameters for the model.
memory	The memory usage for loading the model.
FLOPs	Total floating point operations needed for an inference of the model.
memory read	Total memory read required for an inference of the model.
memory write	Total memory write required for an inference of the model.
memory readwrite	The summation of memory read and memory write.

Table 4.1: Detailed explanation for static model profile information including parameters, memory, FLOPs, memory read, memory write and memory readwrite.

initialization because in production a model is usually initialized only once in its lifecycle. Detailed descriptions are listed in table 4.2.

The static profile information is retrieved from the model definition (i.e. its neural network architecture and weights). However, different runtime environment (e.g. TensorFlow, PyTorch and MXNet) offers diverse optimization. As a result, The static profile information cannot reflect the real situation at runtime. The reason we profile them is that these static information can be used as an estimation to evaluate how large and how fast the model is. The dynamic profile information is measured by running a model on a real device in the cluster with input data. So the dynamic information is more accurate at measuring the resource consumption and performance of a model.

The research questions remained for the model profiler are as follows:

- First, how to profile a machine learning model to get its *static* statistics information?
- Second, how to profile a machine learning model to get its *dynamic* statistics

Field	Description
ip	IP address of the machine on which the profiling task is done.
device	Identifier for device on which the profiling task is performed.
batch size	Batch size used to do the profiling task.
memory	Memory usage for loading the model and executing inference.
initialization latencies	Min, max and average latencies for loading and initializing the model (batch size independent).
pre-processing latencies	Min, max and average latencies for pre-processing.
inference latencies	Min, max and average latencies for inference.
post-processing latencies	Min, max and average latencies for post-processing.
pre-processing throughputs	Min, max and average throughputs for pre-processing.
inference throughputs	Min, max and average throughputs for inference.
post-processing throughputs	Min, max and average throughputs for post-processing.

Table 4.2: Detailed explanation for dynamic model profile information including ip, device, batch size, latencies (initialization, pre-processing, inference and post-processing) and throughputs (pre-processing, inference and post-processing).

information?

- Third, since dynamic statistics information is device-dependent and machine-dependent, the question is how to design the dynamic profiler to profile models efficiently?

The first and second research questions are straightforward, we would like to make it possible to profile a model to get both its static and dynamic statistics information. The third question reveals the challenge to dynamic profile models in an efficient way.

4.2 Description of the Solution

Solutions to all three research questions will be described consecutively. Hysia defines machine learning framework as *ModelEngine* and currently only supports *TF* (short for TensorFlow) and *TORCH* (short for PyTorch) engines.

4.2.1 Solution to First Research Question

In order to solve the first research question, we are going to find methods to get static statistics information for machine learning models. To be honest, models written with different machine learning frameworks (e.g. TensorFlow and MXNet) are totally different in their underline implementations. Consequently, profiling models written in different engines (aka machine learning frameworks) are completely different tasks. Although they have different implementations and usage, the final result is same, which is the above-mentioned static information listed in table 4.1. Concretely, we designed an interface *StaticProfiler* which has a method *profile* of producing static profile result. Each engine must have

an implementation for the *StaticProfiler* interface and wraps specific profiling procedures for models written for that engine. This will be described detailedly in 4.3.1.

4.2.2 Solution to Second Research Question

Dynamic profiling can be unified across diverse engines. Even though models for different engines have its own way to do inference tasks, Hysia provides a centralized *ModelService* to eliminate the differences. The dynamic profiling procedure for a model would be to perform several initializations and inferences of the model and collect the memory usage, latencies (initialization, pre-processing, inference and post-processing) and throughputs (pre-processing, inference and post-processing), thus we can get all the information needed to form the final dynamic profile result shown in table 4.2.

4.2.3 Solution to Third Research Question

A major difference between dynamic profiler and static profiler is that the dynamic profiler profiles models on all devices across all nodes (aka machines). This brings challenges to how to distribute dynamic profile tasks to all nodes and collect results from them. To overcome this challenge, Hysia’s dynamic profiler is designed in a distributed fashion. Master node distributes tasks to all nodes in the cluster. Node performs the profile task locally and puts result back to master node. To simplify the network programming and minimize the communication overhead, Redis Pub/Sub [37] which provides a centralized high performance channel for broadcasting messages in a cluster is used.

4.3 Model Profiler Design

Now we have solutions to all the research questions, we would like to present our design for the model profiler. The graph 4.1 shows the architecture design for the model profiler. *Profiler* is the entry point for the model profiler. *ProfileTask* contains necessary information related to profile a model including *engine*, *model*, *version* and *args*. *engine*, *model*, *version* are used to find and load the model from Hysia’s *ModelDB*. *args* is short for arguments which is additional parameters for both *StaticProfiler* and *DynamicProfiler*. The arguments contains necessary inputs such as min and max batch sizes for the *DynamicProfiler*. *Profiler* relies on *StaticProfiler* and *DynamicProfiler* to do the real profiling jobs. Given a *ProfileTask*, *Profiler* splits the task into *StaticProfileTask* and *DynamicProfileTask* and send these two sub-tasks to *StaticProfiler* and *DynamicProfiler* respectively. *StaticProfiler* and *DynamicProfiler* then perform these tasks and produce *StaticProfileResult* and *DynamicProfileResult* respectively. These two results are finally saved into Hysia’s *ModelDB* for later uses.

4.3.1 Static Model Profiler

As discussed previously, *StaticProfiler* interface defines a unified abstract method that takes model and additional arguments to perform static profiling and outputs *StaticProfileResult* with fields listed in table 4.1. Its implementations include *TFProfiler* and *TorchProfiler* as Hysia only supports models written with TensorFlow and PyTorch for now. As their names suggest, *TFProfiler* profiles models written for TensorFlow engine and *TorchProfiler* profiles models written for PyTorch engine. In the implementation of *TFProfiler*, we use *TensorFlow Profiler* [18] to profile TensorFlow models. It can get model’s parameters, FLOPs and memory

Figure 4.1: Model profiler architecture. *Profiler* splits *ProfileTask* into *StaticProfileTask* and *DynamicProfileTask* for *StaticProfiler* and *DynamicProfiler* to process and produce *StaticProfileResult* and *DynamicProfileResult* which are saved into *ModelDB* for future uses.

readwrite. The individual memory read and memory write are not supported. And to implement *TorchProfiler* we use *TorchStat* [19] to profile models written in PyTorch framework. Parameters, FLOPs, memory read, memory write and memory readwrite can be retrieved. The simple factory design pattern is in use as we provided the *StaticProfilerFactory* to create concrete *StaticProfiler* implementation according to engine types for better extensibility.

4.3.2 Dynamic Model Profiler

DynamicProfiler is more complicated than *StaticProfiler* as it runs in a distributed way. Graph 4.2 shows the architecture design for *DynamicProfiler*. Each node runs a *DynamicProfileWorker* daemon process subscribing to a task channel in Redis and waiting for task messages. Once a new dynamic model profile task is ready, the *DynamicProfiler* distributes the task to workers by publishing the task as a message into Redis task channel. In this way the task message is broadcasting in the cluster. Once a worker has received the task message, it deserializes the message back into task definition and perform the task. Worker then publishes the dynamic profile result to another Redis channel (aka the result channel) once the profiling is done. The result channel has only one subscriber that is the *DynamicProfiler*. Results from distributed workers are collected through the result channel. In order to profile a model to get its dynamic statistics information, *DynamicProfileWorker* loads model from *ModelDB* and executes inferences of the model for each batch size ranging from min batch size to max batch size (specified as arguments for the profiling task) for several times (default to 10) and collects metrics defined in table 4.2. The dynamic profiling results for a model are machine and device dependent. This needs us to dynamically profile the model again if a new device (e.g. a new GPU) is installed onto a machine. It is not a once for all solution compared to static model profiling but it reveals the true statistics when the model is actually running.

4.4 Evaluation

This section describes the methodology and experiments for evaluating both *StaticProfiler* and *DynamicProfiler*. This section is organized as follows:

Figure 4.2: Dynamic model profiler architecture. *DynamicProfiler* dispatches *DynamicProfileTask* to *DynamicProfileWorker* to perform the profile task and collect *DynamicProfileResult* from it through Redis. Result is finally saved into *ModelDB* for future uses.

- Section 4.4.1 presents theories for static model profiling. In fact, we use open source libraries to implement *StaticProfiler*. These libraries have already applied the theories so we do not need to invent wheels on our own. But this section gives theoretical ideas about static model profiling, which describes the internals about the third party libraries that we use.
- Section 4.4.2 shows the evaluations for *StaticProfiler*.
- Section 4.4.3 defines methodology for evaluating dynamic profiler metrics defined in table 4.2.
- Section 4.4.4 gives the evaluations for *DynamicProfiler*.
- Section 4.4.5 shows the impact of batch size to model inference latencies, throughputs and memory usage at runtime captured by *DynamicProfiler*.

4.4.1 Theory for Static Model Profiling

The theory for static model profiling is complex given the diversity of machine learning operations like convolution, pooling, activation, linear and so on. This section talks about the theory for calculating FLOPs and parameters of various neural network layers like fully connected layer, convolutional layer and recurrent layer. Note that in the following discussions, we omit the impact of bias term towards FLOPs and parameters calculations as bias term is under the magnitude of importance compared to others.

4.4.1.1 Fully Connected Layer

A fully connected layer (FC layer) only involves linear operation. It transforms input vector to output vector by matrix multiplication. The formula for calculating the FLOPs for all FC layers in a neural network model is [38]:

$$\sum_{l=1}^d n_{l-1} \cdot n_l \quad (4.1)$$

In the above formula, l is the index for the layer and d is the total number of FC layers in the neural network model. n_l and n_{l-1} are numbers of units of l -th layer and layer before l -th layer. The weights matrix for a FC layer is of shape $n_{l-1} \times n_l$ thus the number of parameters for all FC layers in a network is [38]:

$$\sum_{l=1}^d n_{l-1} \cdot n_l \quad (4.2)$$

4.4.1.2 Convolutional Layer

The formula for calculating the FLOPs for all convolutional layers in a neural network model is [39]:

$$\sum_{l=1}^d n_{l-1} \cdot s_l^2 \cdot n_l \cdot m_l^2 \quad (4.3)$$

Here l is the index for the layer and d is the total number of convolutional layers in the neural network model. n_l and n_{l-1} are numbers of filters for l -th layer and layer before l -th layer. n_{l-1} is also regarded as input channels for layer l . s_l is the kernel size for layer l . m_l is the size of the output feature map. This formula supposes the shape of the output feature map is square.

The formula for calculating total parameters for all convolutional layers is [38]:

$$\sum_{l=1}^d n_{l-1} \cdot s_l^2 \cdot n_l \quad (4.4)$$

The parameters of the above equation have the same meanings as in the formula (4.3). For each filter in layer l with kernel size s_l there are $n_{l-1} \cdot s_l^2$ weights. So there are $n_{l-1} \cdot s_l^2 \cdot n_l$ weights or parameters for a single convolutional layer l .

4.4.1.3 Recurrent Layer

To the best of our knowledge, there is no previous work talking about calculating FLOPs and memory for recurrent layers comprehensively. A recurrent layer runs multiple times on an input sequence. This behavior is different compared to FC layer and convolutional layer as they runs only one time on their input data. The below formula shows how to calculate FLOPs for all recurrent layers in a neural network model:

$$\sum_{l=1}^d T_{l-1} \cdot F_l(u_l) \quad (4.5)$$

And the formula to calculate total number of parameters for all recurrent layers is:

$$\sum_{l=1}^d P_l(u_l) \quad (4.6)$$

l is the index for the layer and d is the total number of recurrent layers in the network. T_{l-1} is the length of input sequence to layer l . $F_l(u_l)$ and $P_l(u_l)$ are two functions to calculate FLOPs and parameters of the layer running on a single element of the input sequence. Recurrent layer has diverse recurrent units like Elman recurrent unit (ERN) [40], gated recurrent unit (GRU) [41] and long short-term memory unit (LSTM) [12]. u_l here means the type of the recurrent unit for layer l . Different recurrent unit has its own way to calculate FLOPs and parameters. FLOPs count is proportional to the input sequence length T_{l-1} because the recurrent layer runs T_{l-1} times to process the sequence. While the number of parameters is independent of the sequence length because each run shares the same parameters. Below we would like to take about calculating FLOPs and parameters for different recurrent units.

4.4.1.3.1 Elman Recurrent Unit The ERN is the most simple recurrent unit compared to its alternatives. It processes each element in the input sequence in the following way:

$$h_t = g(W_{ih} \cdot x_t + b_{ih} + W_{hh} \cdot h_{t-1} + b_{hh}) \quad (4.7)$$

t is the index of element in the input sequence. x_t is the input element at index

t . h is the hidden state that ERN keeps and updates on every computation of the network. h_t and h_{t-1} are hidden states at time t and time $t - 1$ respectively. W_{ih} and W_{hh} are weights associated with input data and hidden state. b_{ih} and b_{hh} are the biases. g is activation function like sigmoid, tanh and ReLU.

Given its details to process input data, the FLOPs needed to process a single input element is calculated as follows:

$$n_h \cdot (n_x + n_h) \quad (4.8)$$

Where n_h and n_x are dimensions or numbers of features of hidden state and single input element respectively.

To compute the number of parameters, we only need to take weights and hidden state into considerations. The formula to calculate parameters is:

$$n_h \cdot (n_x + n_h + 1) \quad (4.9)$$

4.4.1.3.2 Gated Recurrent Unit Compared to ERN, the GRU computes reset gate, update gate and hidden state candidate before updating its hidden state. The process to update its hidden state given a single element of input sequence is listed as follows:

$$\begin{aligned} r_t &= \sigma(W_{ir} \cdot x_t + b_{ir} + W_{hr} \cdot h_{t-1} + b_{hr}) \\ u_t &= \sigma(W_{iu} \cdot x_t + b_{iu} + W_{hu} \cdot h_{t-1} + b_{hu}) \\ \tilde{h}_t &= \tanh(W_{ih} \cdot x_t + b_{ih} + r_t \cdot W_{hh} \cdot h_{t-1} + b_{hh}) \\ h_t &= u_t \cdot \tilde{h}_t + (1 - u_t) \cdot h_{t-1} \end{aligned} \quad (4.10)$$

r_t , u_t and $\tilde{h}_t$ are reset gate, update gate and hidden state candidate respectively.

The FLOPs for such a single pass of a GRU is calculated by:

$$n_h \cdot (3n_x + 4n_h + n_h^2) \quad (4.11)$$

The number of parameters is simple the number of elements in all the weights (i.e.: W_{ir} , W_{hr} , W_{iu} , W_{hu} , W_{ih} and W_{hh}) of the GRU plus its hidden state. So the parameters count is calculated as:

$$n_h \cdot (2n_x + 4n_h + 1) \quad (4.12)$$

4.4.1.3.3 Long Short-Term Memory LSTM is the most popular and powerful recurrent unit compared to the other two types of unit. It has not only a hidden state, but also a memory cell state usually with the same dimension as the hidden state. It computes three gates and a cell state candidate before updating its cell state and hidden state. The following formulas describes a single run for an LSTM unit:

$$\begin{aligned} f_t &= \sigma(W_{if} \cdot x_t + b_{if} + W_{hf} \cdot h_{t-1} + b_{hf}) \\ u_t &= \sigma(W_{iu} \cdot x_t + b_{iu} + W_{hu} \cdot h_{t-1} + b_{hu}) \\ o_t &= \sigma(W_{io} \cdot x_t + b_{io} + W_{ho} \cdot h_{t-1} + b_{ho}) \\ \tilde{c}_t &= \tanh(W_{ic} \cdot x_t + b_{ic} + W_{hc} \cdot h_{t-1} + b_{hc}) \\ c_t &= u_t \cdot \tilde{c}_t + f_t \cdot c_{t-1} \\ h_t &= o_t \cdot \tanh(c_t) \end{aligned} \quad (4.13)$$

f_t , u_t and o_t are forget, update and output gates. $\tilde{c}_t$ is the cell state candidate. c_t and h_t are cell state and hidden state. By counting its matrix multiplications, we can get the formula to calculate FLOPs needed for a single LSTM run:

$$n_h \cdot (4n_x + 7n_h) \quad (4.14)$$

Parameters for an LSTM unit is calculated by counting its weights, cell state and hidden state as:

$$n_h \cdot (4n_x + 4n_h + 2) \quad (4.15)$$

A bidirectional recurrent layer runs two times with inverse directions on its recurrent unit. For calculating FLOPs of a bidirectional recurrent unit, we can simply multiply two to the FLOPs for its unidirectional version. While for parameters, a bidirectional version of a recurrent unit has the same number of weights compared to its unidirectional one but has two times of its hidden state and cell state (for LSTM). The above formulas for calculating FLOPs and parameters of unidirectional recurrent layer can be easily adjusted to bidirectional layers.

4.4.2 Evaluation for Static Model Profiler

We implemented the proposed static profiler and did some experiments for various CNNs on server 1 with specification listed in table 4.3. Table 4.4 shows the experiment results for four CNN models with engine *TORCH* (aka PyTorch), batch size 1 and input size $241 \times 241 \times 3$. Table 4.5 shows results for the same set of CNN models with engine *TF* (aka. TensorFlow). The batch size and input size are same as before. Memory read and memory write fields are omitted for saving space.

Note that even for same model, the profile results from *TFProfiler* and *TorchProfiler* are different because models written with different frameworks have different underline implementations and optimizations. The profiling experiments

Field	Server 1	Server 2
CPU	Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz 32 Cores	Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz 32 Cores
Memory	64 GB DDR4	64 GB DDR4
GPU	1 × TITAN X (Pascal) 12G 3 × Tesla K40c 12G	2 × Quadro M6000 12G 2 × Tesla K40c 12G
System	Ubuntu 16.04.4 LTS kernel 4.4.0-116-generic	Ubuntu 16.04.4 LTS kernel 4.4.0-116-generic

Table 4.3: Specification including CPU, memory, GPU and system information for two servers running model profiler evaluations.

Models	Parameters	Memory (B)	FLOPs	Memory Readwrite (B)
ResNet-50	25557032	115015584	8219737624	333258816
ResNet-101	44549160	169607072	15662945816	518410304
AlexNet	61100840	4356000	1429567512	253606976
Vgg-16	138357544	114603936	30958666264	783170624

Table 4.4: Static profile results for various CNNs with engine *TORCH*, input size 241x241x3 and batch size 1.

are all done on CNNs. Other types of models such as object detection models and RNN models are hard to be statically profiled. For instance, an object detection model such as YOLO often comprises of complex neural network architecture and needs to run classification tasks multiple times on its backbone network. And RNN models have their recurrent natures. We leave statically profiling these complex models as future works.

Models	Parameters	Memory (B)	FLOPs	Memory Readwrite (B)
ResNet-50	23701540	152592204	6988734351	155172940
ResNet-101	42684964	263063372	14459586498	265644108
AlexNet	46616612	189266492	1560421965	189266492
Vgg-16	134670244	599627836	31222449752	599627836

Table 4.5: Static profile results for various CNNs with engine TF , input size 241x241x3 and batch size 1.

4.4.3 Methodology for Profiling Model Latencies and Throughputs

Model latencies and throughputs are metrics of the dynamic model profiler. Here we would like to give the methodology of measuring these metrics.

The model latencies contain four metrics: *initialization*, *preprocess*, *inference* and *postprocess* latencies. In order to execute a model for prediction, the model is first loaded into memory and initialized (e.g. setting properties and initialize parameters). We define the latency for model loading and setup as *initialization* latency. When the model is being called for predicting user inputs, the inputs are first needed to be reprocessed (e.g. resizing input images to fit the model). The time needed to pre-process the user inputs are defined as *preprocess* latency. Once the user inputs are preprocessed, the input data can be passed into the model for inference. And the time cost for the model to process the preprocessed data is *inference* latency. Finally, the outputs from the model inference are to be post-processed (e.g. transforming image classification inference outputs to image labels like cat, car and person). The time cost here is considered as *postprocess* latency. The dynamic model profiler profiles a model to get these four kinds of latencies. Each of them comprises of min, max and average. For example, in order

to profile the inference latencies for a model, the inference process is executed n times. From these n latencies we further calculate the min, max and average latencies as the result for the inference latencies for the model. The four steps bring us five points to record time (i.e. before, between and after these steps). We define these time points as t_0 to t_4 . The equation for calculating latency is defined as follows:

$$L_{mbi} = t_{mbi} - t_{mb(i-1)}, \quad 1 \leq i \leq 4 \quad (4.16)$$

Where L_{mbi} means latency for model m with batch size b at i -th step. t_{mbi} and $t_{mb(i-1)}$ stand for recorded time at point i and $i - 1$.

Model throughputs have similar metrics as to model latencies. The metrics are: *preprocess*, *inference* and *postprocess* throughputs. Throughput is calculated as the total requests per unit time. We use the batch size here to represent the number of requests. It is calculated as below:

$$T_{mbi} = \frac{b}{L_{mbi}}, \quad 2 \leq i \leq 4 \quad (4.17)$$

In the above equation, T_{mbi} is the throughput for model m with batch size b at step i . L_{mbi} is same as the previous equation. We do not measure the throughput for model initialization because initialization for a model is usually done once that is why the i is from 2 to 4 (*preprocess*, *inference* and *postprocess*).

4.4.4 Evaluation for Dynamic Model Profiler

We implemented and evaluated the proposed dynamic model profiler on server 1 (4.3) and server 2 (4.3). We build a cluster using these two servers and server 1 is master node while both servers are worker nodes. We then specify dynamic profile

Model	Engine	Description
PR50	TORCH	Place365 scene classification model in PyTorch with ResNet50 as backbone network [42].
SA-LSTM	TORCH	PyTorch movie review sentiment analysis LSTM model [43].
USE	TF	Google’s universal sentence encoder in TensorFlow [44].

Table 4.6: Model names, underline engines and descriptions for the three models used to evaluate the dynamic model profiler.

tasks to the master node which distributes tasks to both worker nodes to profile tasks locally. The results are collected by master node and saved to MongoDB. We use three models listed in table 4.6 to evaluate the dynamic model profiler. Parts of the evaluation results can be seen in table 4.7.

4.4.5 Impact of Batch Size to Latencies, Throughputs and Memory at Runtime

To better understand how different devices and batch sizes influence the performance of a model, we have drawn three figures (4.3, 4.4, 4.5) to visualize the impact of batch size to latencies, throughputs and memory usage for the models listed in table 4.6. In all these figures, the batch sizes over the entire range are examined and not interpolated. For PR50 model and SA-LSTM model, we only profiled them on GPU devices. And for USE model we only profiled it on CPU device. So the memory usage figures only include GPU memory usage for the first two models and CPU memory usage for the third model.

Figure 4.3 shows the relations between inference latencies with batch size for models. The inference latencies go up as the batch size increases since processing data with large batch size usually needs more resources and time. From

Model	IP	Device	Batch Size	Memory (MB)	Preprocess Latencies (sec)	Inference Latencies (sec)	Inference Throughputs (req/sec)
PR50	Server 1	TITAN X (Pascal)	64	11505	0.9042	0.0114	556.8357
					1.2806	0.1149	5625.2192
					1.0368	0.0597	2544.3332
PR50	Server 2	Tesla K40c	64	11165	1.2339	0.0173	91.9918
					2.0657	0.6957	3707.7744
					1.4162	0.3486	612.2983
PR50	Server 2	Quadro M6000	64	11254	1.2632	0.0124	387.8335
					1.9862	0.1650	5149.3472
					1.5254	0.0903	2119.1043
SA-LSTM	Server 1	TITAN X (Pascal)	128	649	0.0180	0.0059	3558.4529
					0.0432	0.0180	10802.6660
					0.0230	0.0085	8208.3362
SA-LSTM	Server 2	Tesla K40c	128	461	0.0179	0.0062	2838.2742
					0.0388	0.0225	10271.8955
					0.0220	0.0157	4505.9038
SA-LSTM	Server 2	Quadro M6000	128	690	0.0179	0.0060	4644.9351
					0.0428	0.0138	10683.5732
					0.0234	0.0087	7613.8912
USE	Server 1	Intel CPU E5-2630	128	1509	9.54e-07	0.9767	57.7060
					2.43e-05	2.2181	131.0528
					3.41e-06	1.1670	115.8532

Table 4.7: Sample dynamic profile results produced by evaluating the dynamic model profiler for models listed in table 4.6 on two servers listed in table 4.3.

Figure 4.3: Impact of batch size to latencies for PR50, SA-LSTM and USE models with details listed in table 4.6.

figure 4.3(a) and figure 4.3(b) we can see that same model has different inference latencies on different devices for same batch size. This is obvious because different devices (e.g. TITAN X and Tesla K40c) offer diverse computational capacities and speeds. Figure 4.4 shows us the impact on inference throughputs of different batch sizes and devices. When the batch size increases from 1 to a certain point, the throughput increases. But as the batch size continues increasing, the throughput remains at plateau (e.g.: USE inference throughputs on figure 4.4(c)). This is because larger batch size brings slower inference latency. The figure 4.5 reveals the memory usage for models as the batch size goes up. The curves are nearly linear because a unit of input data in a batch consumes nearly an identical amount of memory. Another observation is that same model occupies different amounts of memory on different devices. For example, in figure 4.5(b), when the batch size is 1, the SA-LSTM model takes about 580 MB on TITAN X while only takes around 400 MB on Tesla K40c.

(a) PR50

(b) SA-LSTM

(c) USE

Figure 4.4: Impact of batch size to throughputs for PR50, SA-LSTM and USE models with details listed in table 4.6.

(a) PR50

(b) SA-LSTM

(c) USE

Figure 4.5: Impact of batch size to memory for PR50, SA-LSTM and USE models with details listed in table 4.6.

4.5 Conclusion

This chapter detailed the design, implementation and evaluations for the model profiler. Theories for static model profiling for FC layer, convolutional layer and various recurrent layers were also described. In the next chapter, we will focus on resource monitor.

Chapter 5

Resource Monitor

5.1 Introduction and Research Questions

System resource status is another key input to the intelligent controller for the system optimization for Hysia framework. Given status of system processes and hardware (memory, CPU and GPU) the controller is able to determine solutions towards the best utilization of the system resources of the whole cluster. Hysia divides system resource status into six types: *CPU status*, *memory status*, *GPU status*, *process status*, *GPU process status* and *network status*. Each status has its own fields or metrics. Examples for fields descriptions for CPU status, memory status and GPU status are depicted in table 5.1, 5.2 and 5.3. Take GPU status as an example, the GPU status includes important information for a GPU like free memory and temperature which can be used for the intelligent controller to decide whether to put a model into the GPU. Previous works [45] have explored the potential to improve model inference throughput through device placement optimization which relies on statuses of devices.

The questions are straightforward as the objectives for resource monitor are obvious:

- First, how to query status for all types of the resource needed?
- Second, how to design the monitor to collect status from nodes in a cost-aware way?

Field	Description
model	Manufacturing model.
count	Logical CPU count
freqs	Tuple containing current, min and max frequencies in Mhz.
cache	Total cache in KB of L1, L2 and L3 caches.
loads	The average system load over the last 1, 5 and 15 minutes.
utilization	Current cpu utilization.
times	system CPU times as a named tuple including user, system, idle, nice, iowait, irq, softirq, steal, guest, and guest_nice.
timesRatio	Utilization for each specific cpu time.

Table 5.1: Descriptions for fields of CPU status including model, count, freqs, cache, loads, utilization, times and timesRatio.

The first question asks how to get different status and the second poses challenges to monitor resources in a distributed setting.

5.2 Description of the Solution

In order to solve the first question, several open source tools [23], [24], [27] are borrowed. *psutil* can be used to collect extensive system status information including CPU status, memory status, process status and network status. *py-cpuinfo* helps *psutil* to collect richer CPU status information. *GPUtil* is used to fetch GPU status and GPU process status.

To solve the second question, the monitor needs to be designed in a distributed way. Each node runs a monitor process that collects system resource status locally and send them periodically (e.g. every one second). Results are sent by being put into a Redis channel which is subscribed by monitor process in the master node

Field	Description
total	Total physical memory.
available	The memory that can be given instantly to processes without the system going into swap.
utilization	Memory utilization.
free	Memory not being used at all (zeroed) that is readily available.
used	Used memory.
shared	Memory that may be simultaneously accessed by multiple processes.
buffered	Cache for things like file system metadata.
cached	Cache for various things.
totalSwap	Total swap memory.
usedSwap	Used swap memory.
freeSwap	Free swap memory.
swapUtilization	Swap memory utilization.

Table 5.2: Descriptions for fields of main memory status including total, available, utilization, free, used, shared, buffered, cached, totalSwap, usedSwap, freeSwap and swapUtilization.

Field	Description
gpuId	GPU identifier or index (varying across machines).
uuid	GPU universal unique identifier (same across machines).
model	Manufacturing model (e.g. Tesla K40c).
totalMemory	Total memory in MB.
usedMemory	Used memory in MB.
freeMemory	Free memory in MB.
memoryUtilization	Memory utilization.
temperature	Temperature in celsius degree.

Table 5.3: Descriptions for fields of GPU status including gpuId, uuid, model, totalMemory, usedMemory, freeMemory, memoryUtilization and temperature.

which the intelligent controller is running on. The network communication design is similar as model profiler that is resorting Redis Pub/Sub [37] to minimize the communication overhead.

5.3 Resource Monitor Design

Figure 5.1 shows the architecture design for resource monitor. There is only one instance of *Monitor* running with intelligent controller on the master node. Upon system start of Hysia framework, each node including the master node runs a *MonitorWorker* process collecting its own system resource status defined as *SysStat* and send it back to *Monitor* through the *Stat Channel* in redis. *Monitor* caches the latest system resource statuses for all nodes in the cluster as a list of *SysStat* called *SysStats*. Finally the intelligent controller can easily query the *SysStats* from *Monitor*. Each *MonitorWorker* process collects its own system resource statuses periodically to prevent overhead brought by monitor itself. The

period is configurable and defaults to one second. This allows the monitor to collect system status that reflects the current loads and capabilities while keeping the monitoring cost at a low level.

Figure 5.1: Resource monitor architecture. *MonitorWorker* running on each node collects system resource statuses and sends them back to *Monitor* through Redis. Other components (e.g. intelligent controller) can easily query system statuses from *Monitor* or Redis.

5.4 Evaluation

This section describes the evaluations for monitor and time cost of its various components.

Field	Description
CPU	Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz 32 Cores
Memory	64 GB DDR4
GPU	1 × TITAN X (Pascal) 12G 3 × Tesla K40c 12G
System	Ubuntu 16.04.4 LTS with kernel 4.4.0-116-generic

Table 5.4: Specification including CPU, memory, GPU and system information for server running the monitor evaluation.

5.4.1 Monitor Evaluation

We evaluated the resource monitor on a single server with specification listed in table 5.4. The evaluation results are shown in figure 5.2. From the figure we can see that the resource monitor is capable to collect elaborate system resource status information.

5.4.2 Monitor Time Cost Evaluation

We also evaluated the time cost for querying each resource status in the same server and the evaluation results are listed in table 5.5. The time cost to fetch all statuses is 0.71 seconds on average which is acceptable if the monitoring interval is set to one seconds or above. But it also reveals potential optimizations needed for querying process status, GPU status and GPU process status as they are bottlenecks for the monitoring efficiency.

Status	Avg Query Time (sec.)
All	0.7116337537765502
CPU	0.0008662223815917968
Memory	0.00038526058197021487
GPU	0.05942227840423584
Process	0.5189499855041504
GPU Process	0.07382681369781494
Network	0.004327893257141113

Table 5.5: Time cost evaluation results for query each status from monitor on a single machine with specification listed in table 5.4.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation described works related to the core utility development for Hysia framework. Hysia is a machine learning model serving framework aiming at bringing the most accessible way to create machine learning pipelines for V2O applications and orchestrating models to balance the trade-off between accuracy and resource consumption. Hysia is capable to tackle the workflow optimization and model orchestration challenges brought by increasing intelligent workflows relying on machine learning. This dissertation first introduced Hysia and its framework architecture design. The core contributor of Hysia is its intelligent controller which takes model profile results, system resource statuses and other necessary information as inputs and outputs the best configuration to improve prediction accuracy while keeping energy cost as low as possible. The core utility which comprises of model profiler and resource monitor plays a key role in the joint performance optimization for Hysia framework. This dissertation then detailed the design, implementation and some evaluations for model profiler and resource monitor in chapter 4 and chapter 5 respectively. The design for the core utility aims at providing required information in a fast, distributed and resource-aware way.

6.2 Future Work

A few suggestions are provided for related future works:

1. We would like to investigate the causes for the phenomena of the impact of batch size to latencies, throughputs and memory usage of a model. For instance, a sharp increase of inference latency for PR50 model when the batch size increases from 56 to 57 and different memory usages for the same model on different devices are to be further examined.
2. Given the theory of calculating FLOPs and parameters for various recurrent layers, we are going to improve the static model profiler to support RNNs.
3. We also want to investigate on statically profiling object detection models (e.g. Faster-RCNN and YOLO). To the best of our knowledge, these models are complex and have not yet been well investigated in the literature.

Bibliography

- [1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.
- [2] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv preprint arXiv:1512.01274*, 2015.
- [3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.
- [4] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, “Optimus: an efficient dynamic resource scheduler for deep learning clusters,” in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 3.
- [5] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” in *Advances in neural information processing systems*, 2015, pp. 91–99.
- [6] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask r-cnn,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.
- [7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.
- [8] “Flops,” <https://en.wikipedia.org/wiki/FLOPS>.
- [9] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, “Chameleon: scalable adaptation of video analytics,” in *Proceedings of the 2018*

- Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 253–266.
- [10] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, “Neural collaborative filtering,” in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 173–182.
 - [11] A. Graves, A.-r. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.
 - [12] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
 - [13] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, “Tensorflow-serving: Flexible, high-performance ml serving,” *arXiv preprint arXiv:1712.06139*, 2017.
 - [14] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, “Clipper: A low-latency online prediction serving system,” in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 613–627.
 - [15] L. Richardson and S. Ruby, *RESTful web services*. " O’Reilly Media, Inc.", 2008.
 - [16] AWS-Labs, “mxnet-model-server,” <https://github.com/aws-labs/mxnet-model-server>.
 - [17] “Onnx,” <https://onnx.ai>.
 - [18] “Tensorflow profiler,” <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/core/profiler>.
 - [19] “Torchstat,” <https://github.com/Swall0w/torchstat>.
 - [20] “convnet-burden,” <https://github.com/albanie/convnet-burden>.
 - [21] D. Crankshaw, G.-E. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov, “Inferline: ML inference pipeline composition framework,” *arXiv preprint arXiv:1812.01776*, 2018.
 - [22] “nexus,” <https://github.com/uwsampl/nexus>.

- [23] “psutil,” <https://psutil.readthedocs.io/en/latest>.
- [24] “py-cpuinfo,” <https://github.com/workhorsy/py-cpuinfo>.
- [25] “Gputil,” <https://github.com/anderskm/gputil>.
- [26] NVIDIA, “nvidia-smi,” <https://developer.nvidia.com/nvidia-system-management-interface>.
- [27] Z. Shengsheng, “Gputil enhanced version,” <https://github.com/ZhouShengsheng/gputil>.
- [28] “How to get current process name in linux,” <https://stackoverflow.com/questions/9097201/how-to-get-current-process-name-in-linux>.
- [29] “Getusage(2) linux programmer’s manual,” <http://man7.org/linux/man-pages/man2/getusage.2.html>.
- [30] NVIDIA, “Nvml api reference,” <https://docs.nvidia.com/deploy/nvml-api/nvml-api-reference.html>.
- [31] N. Takahashi, M. Gygli, and L. Van Gool, “Aenet: Learning deep audio features for video analysis,” *IEEE Transactions on Multimedia*, vol. 20, no. 3, pp. 513–524, 2018.
- [32] Y. Aytar, C. Vondrick, and A. Torralba, “Soundnet: Learning sound representations from unlabeled video,” in *Advances in neural information processing systems*, 2016, pp. 892–900.
- [33] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, “The numpy array: a structure for efficient numerical computation,” *Computing in Science & Engineering*, vol. 13, no. 2, p. 22, 2011.
- [34] F. Lundh, “Pillow,” <https://pillow.readthedocs.io/en/stable>.
- [35] “Tfrecords,” https://www.tensorflow.org/tutorials/load_data/tf_records.
- [36] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [37] “Redis pub/sub,” <https://redis.io/topics/pubsub>.
- [38] “How fast is my model,” <https://machinethink.net/blog/how-fast-is-my-model/>.

- [39] K. He and J. Sun, “Convolutional neural networks at constrained time cost,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 5353–5360.
- [40] J. L. Elman, “Finding structure in time,” *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.
- [41] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Gated feedback recurrent neural networks,” in *International Conference on Machine Learning*, 2015, pp. 2067–2075.
- [42] B. Zhou, A. Lapedriza, A. Khosla, A. Oliva, and A. Torralba, “Places: A 10 million image database for scene recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.
- [43] “Pytorch sentiment analysis,” <https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb>.
- [44] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar *et al.*, “Universal sentence encoder,” *arXiv preprint arXiv:1803.11175*, 2018.
- [45] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, “Device placement optimization with reinforcement learning,” in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 2430–2439.
- [46] “Ray,” <https://github.com/ray-project/ray>.