THE UNIVERSITY OF CHICAGO

CORRECTNESS, PERFORMANCE, AND ENERGY-EFFICIENCY: IMPROVING SOFTWARE SYSTEMS THAT USE MACHINE LEARNING COMPONENTS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
PHILOSOPHY OF DOCTOR

DEPARTMENT OF COMPUTER SCIENCE

BY
CHENGCHENG WAN

CHICAGO, ILLINOIS

Dedicated to my beloved family.

"Computers do not solve problems. They execute solutions." - Laurent Gasser

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The past five years had been a wonderful time at the University of Chicago. I would like to express my most sincere gratitude to so many people who have helped me through the years.

Foremost, I would like to thank my advisor, Prof. Shan Lu, for her continuous guidance and support in Ph.D. study and research. Besides all the praise on her professional skills, she is an excellent mentor on both research and life. Her great enthusiasm and patient advising helps me gradually get prepared to become an independent researcher: by imitating her way of doing research, managing time, leading team, communicating with people, and etc. It's my great honor to have become Shan's Ph.D. student at UChicago. And I hope I could become a researcher or adviser as her in the future.

I am also sincerely grateful to the other members of my committee: Prof. Henry Hoffmann and Prof. Michael Maire. Their invaluable suggestions and comments helped me shape the research direction and improve my thesis. They are kind and helpful whenever I encounter research challenges. Their invaluable suggestions guided me through this cross-domain thesis work. I am really honored to have them on my committee.

It is my truly fortunate to work with many excellent colleagues. I would like to thank all members of Shan's research group: Haopeng Liu, Yuxi Chen, Shu Wang, Chi Li, Guangpu Li, Chengcheng Wan, Lefan Zhang, Bogdan Stoica, Utsav Sethi, Yuhan Liu, Haochen Pan, Wei Yuan, and Yu Gao for the opportunity of research collaborations and emotional support. Thanks to my collaborators: Muhammad Santriaji, Shicheng Liu, Sophie Xie, and Yifan Liu for their great effort in research projects.

Besides, I would like to thank my fantastic friends: Weijia He, Ying He, Zhiyu Chen, Xiaojie Wu, Xinyi Zhang, Hao Shen, and many more that I cannot list them all here for the happy time we spent together.

Finally, specially thanks to my family for their unconditional and greatest love. Thanks for your support and company. Thank you, Dad and Mom!

ABSTRACT

Machine learning (ML) provides efficient solutions for a number of problems that are difficult to solve with traditional computing techniques. Deep neural networks (DNNs) have become a key workload for many computing systems due to their high inference accuracy. This accuracy, however, comes at a cost of long latency, high energy usage, and engineering effort. It requires tremendous human effort and domain knowledge to implement a correct, efficient, and robust ML software.

To improve the correctness, performance, and energy-efficiency of machine learning software systems, this dissertation works on these three parts and makes the following contributions:

First, to improve the flexibility of neural networks, this dissertation proposes a novel neural network architecture and a customized optimizer that support anytime prediction. This design allows one neural network to generate a series of increasingly accurate outputs over time without sacrificing accuracy for flexibility.

Second, this dissertation designs a runtime scheduler ALERT, which holistically configures neural networks and system resources together to meet application-specific accuracy, performance, and energy-consumption constraints. It uses a probabilistic model to detect environmental volatility and makes use of the full potential of the DNN candidate set to optimize performance and satisfy constraints.

Third, in the scope of software applications, this dissertation conducts the first comprehensive study about how real-world applications are using machine learning cloud APIs. We generalize 8 anti-patterns that degrade functional, performance, or economical quality of the software. Guided by this study, we propose Keeper, a new testing framework for software systems that use machine learning APIs. Keeper automatically generates many test cases to thoroughly test every branch in the specified function and its callees. It analyzes the test runs and reports many failures, as well as potential patches, to developers.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Machine learning (ML) provides efficient solutions for a number of problems that were difficult to solve with traditional computing techniques, e.g., object detection and language translation. Deep neural networks (DNNs), the most popular ML technique, have become a key workload for many computing systems due to their high inference accuracy. The offering of ML Cloud APIs from all major cloud service providers [12, 47, 63, 104] further makes it easy for software developers to use machine learning components in their software projects without the need to design, train, or run deep neural networks themselves [13]. With such convenience, an increasing number of open-source applications adopt ML techniques, targeting a wide variety of real-world problems [146].

In this dissertation, we aim to create a robust method to incorporate ML components into software systems and help developers to implement and maintain ML applications. Unfortunately, there are still a number of challenges that must be addressed to ensure that the resulting applications are correct, fast, and energy-efficient.

At the machine learning level, traditional neural networks are not flexible for the dynamic software-deployment environment. In practice, applications often have different accuracy, performance, and energy consumption requirements at different stages of the execution or under different interaction contexts; the software and hardware resource provision also changes at run time [30, 40, 59, 92, 168]. However, a traditional neural network cannot easily adapt itself, having to complete all the pre-defined computation to produce one inference result.

At the system level, to dynamically manage resources for a machine-learning integrated software faces unique challenges: (1) machine-learning components offer unique accuracy-performance-energy tradeoffs; (2) the combined neural network and software system configuration space is huge; (3) the resource-management and system-configuration decision making needs to be very fast, not to delay the neural-network inference.

At the application level, developing, testing, and fixing machine learning software system requires cross-domain efforts. For non-experts, using a third-party machine-learning API is much easier than building a neural network from scratch. However, challenges still exist: unlike traditional APIs that are coded to perform well-defined algorithms, ML APIs are trained to perform cognitive tasks whose input are digitalized real-world visual, audio and text content, and whose semantics cannot be reduced to concise mathematical or logical specifications. As a result, it is challenging for developers to use these APIs correctly and efficiently.

1.2 Contributions

At the machine learning level, we propose a novel neural network architecture that supports anytime behavior: such neural networks produce a series of increasingly accurate outputs over time. Complementary to our architectural innovations, we propose a novel optimizer, Orthogonalized SGD (OSGD), for training anytime neural networks. Our experiments demonstrate synergy between our architecture and optimizer: our anytime neural networks perform almost as well as independent non-anytime neural networks of the same size [145].

To tackle system level deployment problem, we have designed a runtime scheduler ALERT, a cross-stack runtime system for neural network inference to meet user goals by simultaneously adapting both neural network models and system-resource settings. It uses a probabilistic model to detect environmental volatility and adopts a random variable *global slow-down factor* to relate the current runtime environment to a nominal profiling environment. Across various experimental settings, ALERT meets constraints while achieving within 93–99% of optimal energy saving or accuracy optimization [148].

For ML software, we conduct the first comprehensive study about how real-world applications are using machine learning APIs. We have found that misuses of ML APIs are widespread and severe: 249 out of the 360 applications (69%) contain misuses in their latest versions, more than half of which contain multiple misuses [146].

Guided by this study, Keeper, a new testing tool for software that uses cognitive ML APIs. Keeper designs a pseudo-inverse function for each ML API that reverses the corresponding cognitive task in an empirical way (e.g., an image search engine pseudo-reverses the image-classification API), and incorporates these pseudo-inverse functions into a symbolic execution engine to automatically generate relevant image/text/audio inputs and judge output correctness. Once misbehavior is exposed, Keeper attempts to change how ML APIs are used in software to alleviate the misbehavior. Our evaluation on a variety of open-source applications shows that Keeper greatly improves the branch coverage, while identifying many previously unknown bugs [147].

1.3 Dissertation Organization.

The remainder of this dissertation is organized as follows. Chapter 2 introduces our work of OSGD and nested architecture. Chapter 3 presents our runtime scheduler ALERT. Chapter 4 introduces our comprehensive study about how real-world applications are using ML APIs. Chapter 5 presents our testing tool Keeper. Chapter 6 introduces related work. Chapter 7 concludes this dissertation and discusses future work.

CHAPTER 2

ORTHOGONALIZED SGD AND NESTED ARCHITECTURES FOR ANYTIME NEURAL NETWORKS

2.1 Overview

In this chapter, we aim to solve the problem of flexible neural networks that support anytime prediction.

On the architectural aspect, we propose new structures for anytime neural networks according to a principle of maximizing the potential for re-use of intermediate state between successive stages. A small network should not only produce a quick output, but should also produce internal representations that serve as valuable input to larger networks in subsequent stages. We thus design architectures so that connections between subnetworks in different stages are aligned: they directly link corresponding pairs of layers across stages, so as to allow subsequent subnetworks to refine previously computed internal representations.

Complementary to our architectural innovations, we propose a novel optimizer, *Orthogonalized SGD (OSGD)*, for training anytime neural networks. Motivating OSGD is a view of anytime networks as a special-case of multitask networks, combined with a desire to facilitate synergy between those tasks. In addition to synergistic architectures, we want another type of synergy: synergy in the optimization dynamics when training those multitask architectures. OSGD provides a methodology for re-balancing task interactions as they simultaneously pull on network parameters over the course of training.

While OSGD is general, with potential application to any multitask training scenario, we restrict focus to anytime networks. We observe dramatic improvements in generalization accuracy when training anytime networks with OSGD: a result that holds across the full spectrum of anytime network architectures. Training our fully-nested anytime networks with Orthogonalized SGD sufficiently improves accuracy to the point of making such networks competitive with standard designs lacking anytime flexibility. Together, the techniques we develop here provide a pathway toward endowing deep neural networks with anytime flexibility at minimal overhead cost.

2.2 Anytime Network Architecture

2.2.1 Design Principles

Three observations guide our anytime architecture designs.

Grow *both* width and depth. Accuracy improves with both deeper (more layers) [54, 130, 136] and wider (more neurons per layer) [29, 155] designs. Consequently, we develop freely composable recipes for nesting networks in width and depth.

Grow fast. Although accuracy typically improves with network size, this improvement usually falls off as size increases; logarithmic scaling of improvements are a common result. Consequently, we increase network size exponentially from one stage to the next. This places output predictions at useful discrete accuracy steps along a trade-off curve and also minimizes cut connections when transforming a standard network into an anytime version.

Reuse intermediate state. We improve efficiency by fully reusing *internal* activation states of earlier subnetworks to bootstrap later subnetworks. By aligning layers of different subnetworks trained for the same task, according to the relative depth in their own subnetwork, we might jump-start computation in larger subnetworks.

2.2.2 Nested Anytime Network Architectures

Our design consists of a sequence of *fully nested* subnetworks: the first, D_1, is completely contained within the second, D_2, which is a subpart of D_3, *etc.* Going from D_i to D_{i+1}, our scheme permits growing the network in width, depth, or both. Our anytime networks also have the following properties: (1) *pipeline structure*: Every subnetwork D_i follows the usual pipeline structure of a traditional neural network (as opposed to the branching present in cascade networks); (2) *aligned feed forward*: Outputs of internal layers of a smaller subnetwork are forwarded to deeper layers of the same subnetwork, as well as internal layers of the larger network most appropriate for consuming their signals, maximizing data reuse (*i.e.*, connections are purely feed-forward in depth or nesting level); (3) *exponential size scaling*: The sizes of subnetworks increases exponentially so later outputs offer meaningful accuracy improvements over earlier ones.

Depth Nesting

We *interlace* layers following the same pipeline structure as the original network. As illustrated in Figure 2.1, we partition a traditional network into odd and even layers. We create a shallower subnetwork consisting of only the odd numbered layers to produce the first intermediate result, and nest it within the full network, which has double the depth. Recursively applying this process, we create a sequence of interlaced networks that repeatedly double in depth.

This depth-nesting strategy applies only to networks satisfying an additional architectural requirement. Notice, in Figure 2.1, the presence of additional skip connections between layers, even in the basic, non-nested network. Indeed, within any network in the sequence, we must have that each layer connects directly to any other layer separated in depth by a power of 2. Fortunately, this power-of-2 skip-connection design is exactly the SparseNet architecture [167], which is a state-of-the-art variant of ResNet [54] (or DenseNet [62]) convolutional networks.

Width Nesting

Our width-nesting strategy divides a network into M horizontal stripes, with the i-th subnetwork including all the neurons inside the first i stripes. Different from this prior

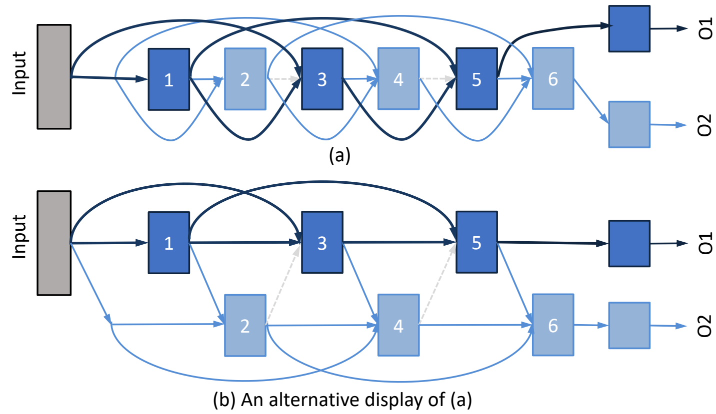


Figure 2.1: Our depth-wise nesting of subnetworks.

work, we use a power-of-2 sequence for stripe widths, as Figure 2.2 depicts.

If the first subnetwork D_1 contains w neurons in one layer, D_i contains $w \times 2^{i-1}$ neurons in the corresponding layer. This choice creates a good trade-off curve for accuracy and latency. All the connections from a later-stripe neuron to an earlier-stripe neuron need to be pruned.

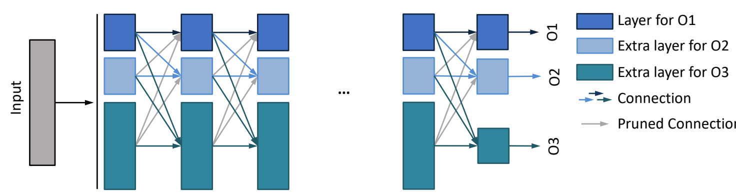


Figure 2.2: Our width-wise nesting of subnetworks.

Combining Depth and Width Nesting

Our width and depth nesting designs can be easily combined in arbitrary order: depth then width, width then depth, or combinations thereof. When growing depth, interlaced layers are added. When growing width, all layers double their filter count. Figure 2.3 illustrates growth by alternating width and depth: subnetwork-1 (dark blue layers) grows to subnetwork-2 by extending its width (light blue layers), then grows to subnetwork-3 by extending depth (green layers), and then to subnetwork-4 by extending width again (light green layers). Figure 2.4 illustrates an alternative of simultaneous growth in width and depth.

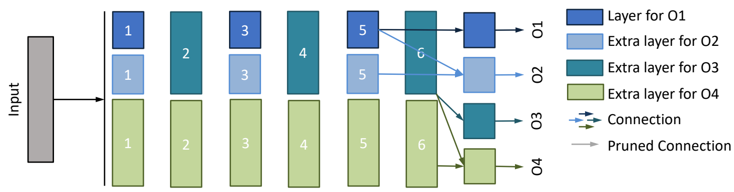


Figure 2.3: Our width-depth nesting that alternates growing width and depth. Connections across intermediate layers are hidden.

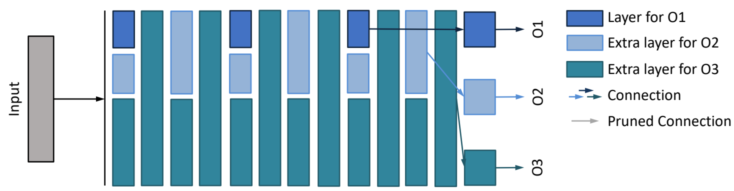


Figure 2.4: Our width-depth nesting that grows width and depth simultaneously. Connections across intermediate layers are hidden.

2.3 Optimization Strategies

Every anytime network (using our architecture or others) faces a multitask training challenge: simultaneous optimization of losses attached to outputs of multiple subnetworks. In this section, we propose Orthogonalized SGD (OSGD), a new optimizer for training multitask deep networks, which is particularly effective when applied to anytime networks.

2.3.1 Definitions and Preliminaries

Training a nested anytime network is an instance of multitask learning, where the tasks are solving the same problem with different network components.

Let $w_1 \in R^{d_1}, w_2 \in R^{d_2}, \cdots, w_n \in R^{d_n}$ be the weights of the nested networks, where $d_1 < d_2 < \cdots < d_n$ and $w_1 \subsetneq w_2 \subsetneq \cdots \subsetneq w_n$. We define other symbols as follows:

- W: weight for the whole network, equivalent to w_n
- L_i: the loss of subnetwork D_i (D_i has weights w_i)
- g_i: the gradient of weights w_i from loss L_i.
- g_i^j: the gradient of weights $w_j \setminus w_{j-1}$ from loss L_i, where $j \leq i$; g_i^j is a subset of g_i.
- C: a constant value for normalization

2.3.2 Orthogonalized SGD (OSGD)

Our novel optimizer, Orthogonalized SGD, dynamically re-balances task-specific gradients in a manner that prioritizes the influence of some losses over others. Given loss-specific gradient vectors $g_1, g_2, \ldots, g_n$, Orthogonalized SGD projects gradients from later outputs onto the parameter subspace that is orthogonal to that spanned by the gradients of earlier outputs. As a result, subsequent outputs do not interfere with how earlier outputs desire to move parameters. For example, the retained component of the gradient of weight w_2 is

$$g_2' = g_2 - proj_{g_1} g_2, \tag{2.1}$$

where $proj_A B$ refers to projecting vector B onto A. g_2' is orthogonal to g_1, and thus updating w_1 in the direction of g_2' minimizes interference with the optimization of loss L_1.

Algorithm 1 Orthogonalized SGD: A multitask variant of SGD with optional dynamic *normalization* of task influence.

Initialize weights W
for $t = 0$ **to** *max_train_steps* **do**
    Compute $L_i(t)\ \forall i,\ s.t.\ 1 \leq i \leq n$ [forward pass]
    $g(t) \Leftarrow \mathbf{0}$
    for $i = 1$ **to** n **do**
        $g_i(t) \Leftarrow \nabla_{w_i} L_i(t)$
        if *normalizing* **then**
            $g_i(t) \Leftarrow g_i(t) / \|g_i(t)\| \cdot \sqrt{d_i} \cdot C$
        end if
    end for
    for $i = 1$ **to** n **do**
        $h_i(t) \Leftarrow \sum_{j=1}^{i-1} proj_{g_j(t)} g_i(t)$
        $g_i(t) \Leftarrow g_i(t) - h_i(t)$
        $g(t) \Leftarrow g(t) + g_i(t)$
    end for
    Update $W(t) \mapsto W(t+1)$ using $g(t)$
end for

Algorithm 1 provides a complete presentation of both Orthognolized SGD and an orthogonalized variant of NormSGD. Note that for anytime networks, per-task gradient vectors are padded with zero entries for any parameters not contained in the corresponding subnetwork. For example, g_1 pads zeros to $w_2 \setminus w_1$, so the part of g_2 specific to the second subnetwork will be unaffected by Equation 2.1.

More generally, OSGD can be used with any priority ordering of tasks; the priority order need not correspond to the order in which outputs are generated by an anytime network. Algorithm 1 is valid for any shuffling of losses, regardless of the underlying network architecture. Choosing a priority order determines the sequencing of gradient projection steps, thereby changing which tasks are given preferential influence over network parameters.

2.4 Evaluation

2.4.1 Methodology

We begin with evaluation using the CIFAR-10 dataset [84]. All networks are trained for 200 epochs, with learning rate decreasing from 0.1 to 0.0008. We train every network 3 times, and report the average and standard deviation of its validation error.

We evaluate all five optimization strategies from Section 2.3: Greedy stage-wise training, SGD, OSGD, and the normalized variants of both SGD and OSGD. We set $C = 1/2$ and use a constant loss importance for SGD and NormSGD, as these settings provide the best results.

We evaluate six different anytime network architectures: four novel designs of our own and two prior designs. Our designs include: (1) depth-nesting applied to Sparse ResNet-98 [167] (Figure 2.1), (2) width-nesting applied to ResNet-42 [54] (Figure 2.2), (3) alternating width-depth nesting (Figure 2.3), and (4) simultaneous width-depth nesting (Figure 2.4), with the latter two applied to Sparse ResNet-98 [167].

The two previous designs represent the state-of-the-art depth-growing anytime design, referred to as *EANN*) and width-growing anytime design, referred to as *Even-width*. In

$\text{Stage}_{\text{size}}$	Greedy	SGD	OSGD	SGD_{Norm}	$\text{OSGD}_{\text{Norm}}$
Our Depth Nested Sparse ResNet-98					
1_{d1}	9.6 (0.2)	9.8 (0.1)	10.0 (0.3)	10.0 (0.2)	10.7 (0.2)
2_{d2}	9.3 (0.3)	8.3 (0.3)	8.4 (0.1)	8.6 (0.4)	8.5 (0.3)
3_{d4}	9.2 (0.3)	7.7 (0.3)	7.4 (0.1)	8.1 (0.3)	7.6 (0.1)
4_{d8}	9.1 (0.2)	7.2 (0.4)	6.6 (0.1)	8.0 (0.2)	6.9 (0.1)
Our Width Nested ResNet-42					
1_{w1}	10.2 (0.1)	12.2 (0.2)	12.3 (0.1)	12.3 (0.3)	12.7 (0.1)
2_{w2}	9.9 (0.2)	10.1 (0.1)	8.9 (0.2)	10.1 (0.2)	9.6 (0.4)
	-	-	-	-	-
3_{w4}	9.2 (0.2)	9.8 (0.3)	7.3 (0.3)	10.1 (0.2)	7.4 (0.2)
Our (Alternating) Width-Depth Nested Sparse ResNet-98					
1_{w1d1}	18.5 (0.1)	31.4 (0.6)	28.3 (0.4)	30.7 (0.4)	28.1 (0.5)
2_{w2d1}	16.5 (0.1)	15.6 (0.2)	14.8 (0.2)	15.5 (0.3)	14.7 (0.4)
3_{w2d2}	15.9 (0.2)	15.5 (0.2)	13.4 (0.3)	15.4 (0.2)	14.1 (0.2)
4_{w4d2}	15.7 (0.4)	10.4 (0.4)	8.6 (0.3)	10.4 (0.2)	9.4 (0.2)
5_{w4d4}	15.6 (0.3)	8.8 (0.3)	6.8 (0.2)	8.9 (0.3)	7.4 (0.2)
Our (Simultaneous) Width-Depth Nested Sparse ResNet-98					
1_{w1d1}	18.5 (0.1)	28.0 (0.2)	26.2 (0.1)	29.1 (0.5)	26.7 (0.5)
2_{w2d2}	11.4 (0.1)	15.0 (0.3)	13.1 (0.1)	15.6 (0.5)	14.5 (0.4)
3_{w4d4}	8.6 (0.4)	8.5 (0.3)	6.8 (0.3)	9.0 (0.2)	7.4 (0.1)

Table 2.1: CIFAR-10 error rates, the lower the better, of our anytime networks with different optimization strategies. Numbers in parentheses are standard deviations. Size subscripts indicate the subnetwork width or depth normalized to that of the first-stage subnetwork. OSGD consistently improves over SGD and, compared to both SGD and Greedy stage-wise training, achieves dramatically lower error for later outputs.

EANN, we apply the cascade-based approach [60] to Sparse ResNet-98, which grows depth exponentially and assembles an output branch every $k \cdot 2^i (i = 1, 2, ...)$ layers. In *Even-width*, we apply the idea of recently proposed even-sized width-nested architecture [89] to ResNet-42.

2.4.2 Evaluation of Optimization Strategies

Tables 2.1 and 2.2 show the validation error rates of applying five different optimizers to different anytime networks. Overall, our Orthogonalized SGD and its normalized variant perform the best, capable of achieving high accuracy for later outputs of an anytime network without significantly reducing the accuracy for earlier outputs.

Stage$_{\text{size}}$	Greedy	SGD	OSGD	SGD$_{\text{Norm}}$	OSGD$_{\text{Norm}}$
			EANN Cascade Sparse ResNet-98		
1_{d1}	9.3 (0.1)	11.7 (0.3)	11.6 (0.3)	12.4 (0.1)	12.1 (0.4)
2_{d2}	9.2 (0.3)	11.1 (0.1)	10.9 (0.2)	12.0 (0.1)	11.2 (0.1)
3_{d4}	8.8 (0.3)	8.5 (0.2)	8.0 (0.1)	9.2 (0.2)	9.0 (0.2)
4_{d8}	8.5 (0.3)	6.5 (0.2)	6.4 (0.2)	8.0 (0.1)	7.6 (0.1)
			Even-Width Nested ResNet-42		
1_{w1}	10.2 (0.04)	12.7 (0.2)	13.9 (0.1)	12.6 (0.1)	13.5 (0.2)
2_{w2}	9.9 (0.3)	10.2 (0.3))	10.7 (0.2)	10.6 (0.1	10.8 (0.1)
3_{w3}	9.9 (0.4)	10.0 (0.1)	8.3 (0.02)	10.5 (0.1)	8.3 (0.01)
4_{w4}	9.8 (0.2)	9.9 (0.1)	8.3 (0.1)	10.4 (0.1)	8.3 (0.1)

Table 2.2: CIFAR-10 error rates of previous anytime networks with different optimization strategies. As in Table 2.1, OSGD offers benefits compared to other optimizers.

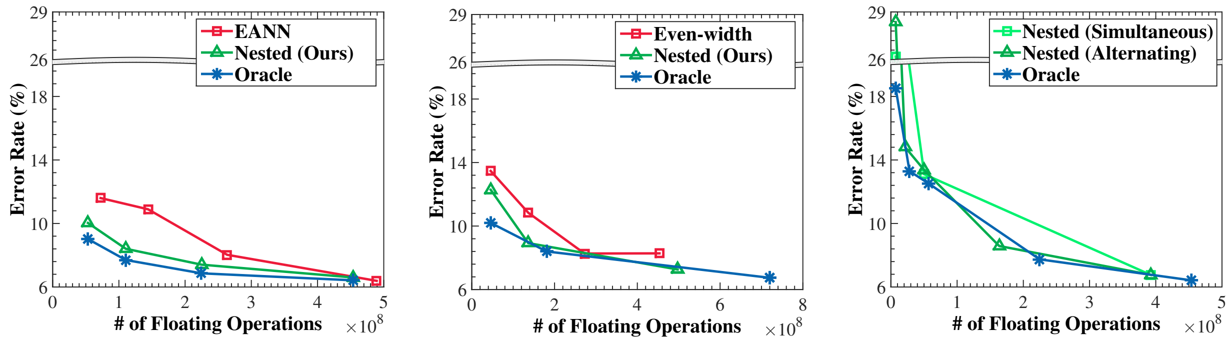
Compared with SGD, OSGD consistently achieves higher accuracy for the last two subnetworks across *all* six anytime designs, while maintaining similar or better accuracy for early subnetworks. Switching from SGD to OSGD drops the last-stage error rates from 7.2, 9.8, 8.8 and 8.5 down to 6.6, 7.3, 6.8 and 6.8 across the four anytime networks in Table 2.1. While the greedy training strategy offers the highest accuracy for the first intermediate result of all anytime networks, it falls far behind OSGD for later-stage results.

The improvement offered by OSGD is striking, yet somewhat counterintuitive. These experiments give earlier outputs high priority than later outputs. OSGD is prioritizing the influence that gradients of smaller subnetworks have on the training dynamics, but it is the outputs of larger subnetworks that most improve in accuracy.

A possible explanation for this curious behavior stems from the fact that the multiple tasks in anytime networks are highly related. In particular, in a well-architected anytime network, different output tasks might exert a beneficial regularization effect on one another. OSGD, by prioritizing task X over task Y in such a network then triggers two effects:

- It allocates parameters to task X instead of task Y.
- It decreases the regularization influence of task Y on task X, while simultaneously increasing the regularization influence of task X on task Y.

Individually, these effects move the relative accuracy of task X and Y in opposite directions.



(a) Depth-nested; Sparse ResNet-98

(b) Width-nested; ResNet-42

(c) Width-depth; Sparse ResNet-98

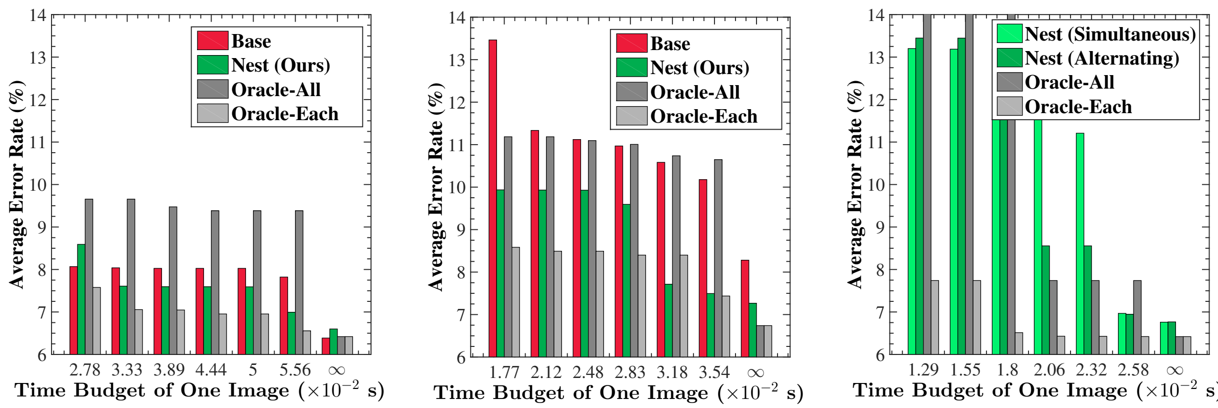
Figure 2.5: Accuracy-FLOP trade-offs (lower is better). Our nested architectures offer trade-offs close to the infeasible Oracle.

As they are coupled, we observe only the net result. Regularization interaction being the stronger effect would explain the behavior of anytime networks trained with OSGD. But, further investigation is required before confidently adopting this explanation.

2.4.3 Evaluation of Nested Architectures.

We compare our nested architectures to an infeasible Oracle—a collection of independently-trained single-task networks with sizes matching our subnetwork stages. Perfectly deploying this collection of independent networks as an anytime system would require oracle knowledge of impending deadlines to select which network to run. The Oracle thus represents an impossible scenario in which anytime prediction capability is granted for free. Figure 2.5 shows the accuracy-FLOPs trade-off curves achieved by our nested network designs (green), the Oracle (blue), and the EANN and Even-width baselines (red). Here, each network is trained using the strategy that offers the most accurate results (*i.e.*, OSGD for all anytime networks and SGD for all independent networks except for the largest setting of SparseResNet-98, which uses NormSGD).

From Figure 2.5a and 2.5b, our depth and width nesting anytime networks both offer much better accuracy-FLOPs trade-offs than previous work, and come close to the infeasible



(a) Depth-nested; Sparse ResNet-98

(b) Width-nested; ResNet-42

(c) Width-depth; Sparse ResNet-98

Figure 2.6: Error rates at different deadlines (lower is better). Our nested designs perform better than baselines and the static Oracle.

Oracle. Figure 2.5c shows our width-depth nested Sparse ResNet-98 offers almost as good a trade-off as the Oracle, and covers a much wider trade-off spectrum than depth-only or width-only nesting.

2.4.4 Run-time Simulation.

We further compare four schemes for maximizing inference accuracy under various inference deadlines: (1) **Base**line anytime schemes (Even-width and EANN); (2) Our **Nest**ed anytime schemes (width, depth, and width-depth nesting). (3) $\mathbf{Oracle_{All}}$, which picks the most accurate independent network that finishes before the deadline for *all* inputs; (4) $\mathbf{Oracle_{Each}}$, which picks the most accurate independent network for each input that finishes before the deadline (*i.e.*, the network may vary across inputs). When no inference result is generated by the deadline, a random guess is output. We report the average error rates across all inputs in Figure 2.6 (vertical axis, lower is better) under 7 deadlines and then no deadline (horizontal axis); the 7 deadlines are set to be 0.5x-1x of the average latency under the biggest ResNet-42 or Sparse ResNet across all inputs.

The accuracy advantage of **Nest** (the second bar in each group) over **Base** (the first bar), and **Oracle**$_{\text{All}}$ (the third bar) is apparent in Figure 2.6. For example, for ResNet-42, **Nest** has 7%-24% lower error rate than **Base** for all deadlines. **Nest** has lower accuracy than **Oracle**$_{\text{Each}}$ in most cases, because the anytime network usually has slightly lower accuracy than an independent network with same size. Note that **Oracle**$_{\text{Each}}$ is impractical, as it assumes impossible latency prediction and no-overhead in swapping networks across inputs. These accuracy-under-deadline results are consistent with the accuracy-latency curves in Figure 2.5.

2.4.5 Evaluation on ImageNet.

Finally, we train a width-nested ResNet-50 and depth-nested Sparse ResNet-66 on the large-scale ImageNet (ILSVRC 2012) dataset [28], using both SGD and OSGD. All networks are trained for 90 epochs, with learning rate decreasing from 0.1 to 0.0001. Table 2.3 reports top-1 and top-5 validation error rates. OSGD significantly improves the accuracy of later stages (larger subnetworks) compared to standard SGD.

	SGD		OSGD	
	Top-1 Error	Top-5 Error	Top-1 Error	Top-5 Error
Our Width Nested ResNet-50				
1_{w1}	36.7	14.7	36.7	14.8
2_{w2}	31.5	11.7	31.7	11.7
3_{w4}	29.2	10.2	28.3	9.4
Our Depth Nested Sparse ResNet-66				
1_{d1}	31.3	11.3	32.9	12.4
2_{d2}	28.4	9.7	29.2	10.1
3_{d4}	28.0	9.3	27.1	8.9

Table 2.3: Validation error of anytime networks trained with SGD and OSGD on the ImageNet dataset.

2.5 Conclusion

Anytime neural network is a promising approach to generating accurate inference results under dynamic latency and resource constraints. In this work, we propose a new class of anytime neural network architectures and a novel variant of SGD customized for training such architectures. Our experiments demonstrate synergy between our architecture and optimizer: our anytime networks perform almost as well as independent non-anytime networks of the same size.

CHAPTER 3

ALERT: ACCURATE LEARNING FOR ENERGY AND TIMELINESS

3.1 Overview

In this chapter, we propose ALERT, a cross-stack runtime system for DNN inference.ALERT dynamically selects and adapts a DNN and a system-resource setting together to handle changing system environments and meet dynamic energy, latency, and accuracy requirements[1].

ALERT is a feedback-based run-time. It measures inference accuracy, latency, and energy consumption; it checks whether the requirements on these goals are met; and, it then outputs both system and application-level configurations adjusted to the current requirements and operating conditions. ALERT focuses on meeting constraints in *any* two dimensions while optimizing the third, e.g., minimizing energy given accuracy and latency requirements or maximizing accuracy given latency and energy budgets.

ALERT uses a random variable relating the current runtime environment to a nominal profiling environment. After each inference task, ALERT estimates the global slow-down factor using a Kalman filter. The global slow-down factor's mean represents the expected change compared to the profile, while the variance represents the current volatility. The mean provides a single scalar that modifies the predicted latency/accuracy/energy for *every* DNN/system configuration—a simple mechanism that leverages commonality among DNN architectures to allow prediction for even rarely used configurations, while incorporating variance into predictions naturally makes ALERT conservative in volatile environments and aggressive in quiescent ones. The global slow-down factor and Kalman filter are efficient to implement and low-overhead. Thus, ALERT combines the global slow-down factor with

1. ALERT provides probabilistic, not hard guarantees, as the latter requires much more conservative configurations, often hurting both energy and accuracy.

latency, power, and accuracy measurements to select the DNN and system configuration with the highest likelihood of meeting the constraints optimally.

We evaluate ALERT using various DNNs and application domains on different (CPU and GPU) machines under various constraints. Our evaluation shows that ALERT overcomes dynamic variability efficiently. Across various experimental settings, ALERT meets constraints while achieving within 93–99% of optimal energy saving or accuracy optimization. Compared to approaches that adapt at application-level or system-level only ALERT achieves more than 13% energy reduction, and 27% error reduction.

3.2 ALERT Run-time Inference Management

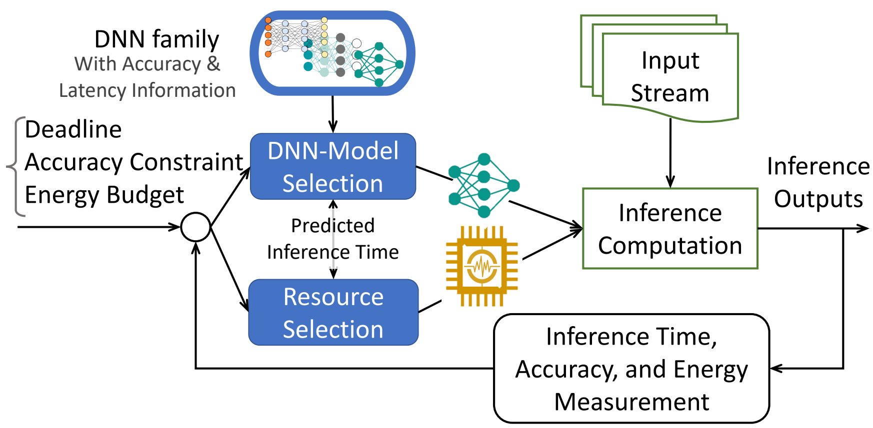


Figure 3.1: ALERT inference system

3.2.1 Inputs & Outputs of ALERT

ALERT's inputs are specifications about (1) the adaption options, including a set of DNN models $\mathbb{D} = \{d_i \mid i = 1 \cdots K\}$ and a set of system-resource settings, expressed as different power-caps $\mathbb{P} = \{P_j \mid j = 1 \cdots L\}$; and (2) the user-specified requirements on latency, accuracy, and energy usage, which can take the form of meeting constraints in any two of these three dimensions while optimizing the third. ALERT's output is the DNN model

$d_i \in \mathbb{D}$ and the system-resource setting $p_j \in \mathbb{P}$ for the next inference-task input.

Formally, ALERT selects a DNN d_i and a system-resource setting p_j to fulfill *either* of these user-specified goals.

1. Maximizing inference accuracy q (minimizing error) for an energy budget $\mathbf{E}_{\text{goal}}$ and inference deadline $\mathbf{T}_{\text{goal}}$:

$$\arg\max_{i,j} q_{i,j} \quad \text{s.t. } e_{i,j} \leq \mathbf{E}_{\text{goal}} \wedge t_{i,j} \leq \mathbf{T}_{\text{goal}} \tag{3.1}$$

2. Minimizing the energy use e for an accuracy goal $\mathbf{Q}_{\text{goal}}$ and inference deadline $\mathbf{T}_{\text{goal}}$:

$$\arg\min_{i,j} e_{i,j} \quad \text{s.t. } q_{i,j} \geq \mathbf{Q}_{\text{goal}} \wedge t_{i,j} \leq \mathbf{T}_{goal} \tag{3.2}$$

3.2.2 ALERT Workflow

ALERT works as a feedback controller. It follows four steps to pick the DNN and resource settings for each input n:

1) Measurement. ALERT records the processing time, energy usage, and computes inference accuracy for $n-1$.

2) Goal adjustment. ALERT updates the time goal T_{goal} if necessary, considering the potential latency-requirement variation across inputs. In some inference tasks, a set of inputs share one combined requirement and hence delays in previous input processing could greatly shorten the available time for the next input [10, 76]. Additionally, ALERT sets the goal latency to compensate for its own, worst-case overhead so that ALERT itself will not cause violations.

3) Feedback-based estimation. ALERT computes the expected latency, accuracy, and energy consumption for every combination of DNN model and power setting.

4) Picking a configuration. ALERT feeds all the updated estimations of latency, accuracy, and energy into Eqs. 3.1 and 3.2, and gets the desired DNN model and power-cap setting

for n.

The key task is step 3: the estimation needs to be accurate and fast. In the remainder of this section, we discuss key ideas and the exact algorithm of our feedback-based estimation.

3.2.3 ALERT Estimation Algorithm

Global Slow-down Factor ξ. ALERT uses ξ to reflect how the run-time environment differs from the profiling environment. Conceptually, if the inference task under model d_i and power-cap p_j took time $t_{i,j}$ at run time and took $t_{i,j}^{\text{prof}}$ on average to finish during profiling, the corresponding ξ would be $t_{i,j}/t_{i,j}^{prof}$. ALERT estimates ξ using recent execution history under any model or power setting. Specifically, after an input $n-1$, ALERT computes $\xi^{(n-1)}$ as the ratio of the observed time $t_{i,j}^{(n-1)}$ to the profiled time $t_{i,j}^{\text{prof}}$, and then uses a Kalman Filter[2] to estimate the mean $\mu^{(n)}$ and variance $(\sigma^{(n)})^2$ of $\xi^{(n)}$ at input n. ALERT's formulation is defined in Eq. 3.3, where $K^{(n)}$ is the Kalman gain variable; R is a constant reflecting the measurement noise; $Q^{(n)}$ is the process noise capped with $Q^{(0)}$. We set a forgetting factor of process variance $\alpha = 0.3$ [11]. ALERT initially sets $K^{(0)} = 0.5$, $R = 0.001$, $Q^{(0)} = 0.1$, $\mu^{(0)} = 1$, $(\sigma^{(0)})^2 = 0.1$, following the standard convention [93].

$$\begin{cases} Q^{(n)} = \max\{Q^{(0)}, \alpha Q^{(n-1)} + (1\text{-}\alpha)(K^{(n-1)}y^{(n-1)})^2\} \\ K^{(n)} = \dfrac{(1-K^{(n-1)})(\sigma^{(n-1)})^2 + Q^{(n)}}{(1-K^{(n-1)})(\sigma^{(n-1)})^2 + Q^{(n)} + R} \\ y^{(n)} = t_{i,j}^{(n-1)}/t_{i,j}^{\text{prof}} - \mu^{(n-1)} \\ \mu^{(n)} = \mu^{(n-1)} + K^{(n)}y^{(n)} \\ (\sigma^{(n)})^2 = (1-K^{(n-1)})(\sigma^{(n-1)})^2 + Q^{(n)} \end{cases} \tag{3.3}$$

Then, using $\xi^{(n)}$, ALERT estimates the inference time of input n under any model d_i

2. A Kalman Filter is an optimal estimator that assumes a normal distribution and estimates a varying quantity based on multiple potentially noisy observations [93].

and power cap p_j: $t_{i,j}^{(n)} = \xi^{(n)} * t_{i,j}^{\text{prof}}$.

Accuracy. ALERT computes the estimated inference accuracy $\hat{q}_{i,j}[\mathbf{T}_{\text{goal}}]$ by considering $t_{i,j}$ as a random variable that follows normal distribution with its mean and variance computed based on that of ξ. Here $q_{i,j}$ represents the inference accuracy when the DNN inference finishes before the deadline, and q_{fail} is the accuracy of a random guess:

$$q_{i,j}[\mathbf{T}_{\text{goal}}] = \begin{cases} q_i & \text{, if } t_{i,j} \leq \mathbf{T}_{\text{goal}} \\ q_{\text{fail}} & \text{, otherwise} \end{cases} \tag{3.4}$$

$$\begin{aligned} \hat{q}_{i,j}[\mathbf{T}_{goal}] =& E(q_{i,j}[\mathbf{T}_{goal}] \mid t_{i,j}^{(n)}) \\ =& E(q_{i,j}[\mathbf{T}_{goal}] \mid \xi^{(n)} \cdot t_{i,j}^{\text{prof}}) \\ =& Pr_{i,j} \cdot q_{i,j} + (1 - Pr_{i,j}) \cdot q_{fail} \\ \xi^{(n)} \sim& \mathcal{N}(\mu^{(n)}, (\sigma^{(n)})^2) \end{aligned} \tag{3.5}$$

Energy. As discussed in Idea-3, ALERT predicts energy consumption by separately estimating energy during (1) DNN execution: estimated by multiplying the power limit by the estimated latency and (2) between inference inputs: estimated based on the recent history of inference idle power using the Kalman Filter in Eq. 3.6. $\phi^{(n)}$ is the predicted DNN-idle power ratio, $M^{(n)}$ is process variance, S is process noise, V is measurement noise, and $W^{(n)}$ is the Kalman Filter gain. ALERT initially sets $M^{(0)} = 0.01$, $S = 0.0001$, $V = 0.001$.

$$\begin{cases} W^{(n)} = \dfrac{M^{(n-1)} + S}{M^{(n-1)} + S + V} \\ M^{(n)} = (1 - W^{(n)})(M^{(n-1)} + S) \\ \phi^{(n)} = \phi^{(n-1)} + W^{(n)}(p_{\text{idle}}/p_{i,j}^{(n-1)} - \phi^{(n-1)}) \end{cases} \tag{3.6}$$

ALERT then predicts the energy by Eq. 3.7. Unlike Eq. 3.5 that uses probabilistic estimates, energy estimation is calculated without the notion of probability. The inference power is the same no matter the inference misses or meets the deadline, as ALERT sets power limits. Therefore it is safe to estimate the energy by its mean without considering the distribution of its possible latency.

$$e_{i,j}^{(n)} = p_{i,j} \cdot \xi^{(n)} \cdot t_{i,j}^{\text{prof}} + \phi^{(n)} \cdot p_{i,j} \cdot (\mathbf{T}_{goal} - (\xi^{(n)} \cdot t_{i,j}^{\text{prof}})) \tag{3.7}$$

3.2.4 Integrating ALERT with Anytime DNNs

An anytime DNN is an inference model that outputs a series of increasingly accurate inference results—o_1, o_2, ... o_k, with o_t more reliable than o_{t-1} (Section 2). ALERT easily works with not only traditional DNNs but also Anytime DNNs. The only change is that q_{fail} in Eq. 3.4 no longer corresponds to a random guess. That is, when the inference could not generate its final result o_k by the deadline $\mathbf{T}_{\text{goal}}$, an earlier result o_x can be used with a much better accuracy than that of a random guess. The updated accuracy equation is below:

$$q_{.,j} = \begin{cases} q_k & \text{, if } t_{k,j} \leq \mathbf{t}_{\text{goal}} \\ q_{k-1} & \text{, if } t_{k-1,j} \leq \mathbf{t}_{\text{goal}} < t_{k,j} \\ & \dots \\ q_{\text{fail}} & \text{, otherwise} \end{cases} \tag{3.8}$$

Existing anytime DNNs consider latency but not energy constraints—an anytime DNN will keep running until the latency deadline arrives and the last output will be delivered to the user. ALERT naturally improves anytime DNN energy efficiency, stopping the inference sometimes before the deadline based on its estimation to meet not only latency and accuracy, but also energy requirements.

Furthermore, ALERT can work with a set of traditional DNNs and an Anytime DNN together to achieve the best combined result. The reason is that Anytime DNNs generally sacrifice accuracy for flexibility. When we feed a group of traditional DNNs and one Anytime DNN to construct the candidacy set $\mathbb{D}$, with Eq. 3.5, ALERT naturally selects the Anytime

DNN when the environment is changing rapidly (because the expected accuracy of an anytime DNN will be higher given that variance), and the regular DNN, which has slightly higher accuracy with similar computation, when it is stable, getting the best of both worlds.

3.3 Limitations and Discussions

Assumptions of the Kalman Filter. ALERT's prediction, particularly the Kalman Filter, relies on the feedback from recent input processing. Consequently, it requires at least one input to react to sudden changes. Additionally, the Kalman filter formulations assume that the underlying distributions are normal, which may not hold in practice. If the behavior is not Gaussian, the Kalman filter will produce bad estimations for the mean of ξ for some amount of time.

ALERT is specifically designed to handle data that is not drawn from a normal distribution, using the Kalman Filter's covariance estimation to measure system volatility and accounting for that in the accuracy/energy estimations. Consequently, after just 2–3 such bad predictions of means, the estimated variance will increase, which will then trigger ALERT to pick anytime DNN over traditional DNNs or pick a low-latency traditional DNN over high-latency ones, because the former has a higher expected accuracy under high variance. So—worst case—ALERT will choose a DNN with slightly less accuracy than what could have been used with the right model. Users can also compensate for extremely aberrant latency distributions by increasing the value of $Q^{(0)}$ in Eq. 3.3. As shown in the experiments, ALERT performs well even when the distribution is not normal.

Probabilistic guarantees. ALERT provides probabilistic, not hard, guarantees. As ALERT estimates not just average timing, but the distributions of possible timings, it can provide arbitrarily many nines of assurance that it will meet latency or accuracy goals but cannot provide 100% guarantee. Providing 100% guarantees requires the worst case execution time (WCET), an upper bound on the highest possible latency. ALERT does not assume the availability of such information and hence cannot provide hard guarantees [24].

Safety guarantees. While ALERT does not explicitly model safety requirements, it can be configured to prioritize accuracy over other dimensions. When users particularly value safety (e.g., auto-driving), they could set a high accuracy requirement or even remove the energy constraints.

Concurrent inference jobs. ALERT is currently designed to support one inference job at a time. To support multiple concurrent inference jobs, future work needs to extend ALERT to coordinate across these concurrent jobs. We expect the main idea of ALERT, such as using a global slowdown factor to estimate system variation, to still apply.

3.4 Implementation

We implement ALERT for both CPUs and GPUs. On CPUs, ALERT adjusts power through Intel's RAPL interface [27], which allows software to set a hardware power limit. On GPUs, ALERT uses PyNVML to control frequency and builds a power-frequency lookup table. ALERT can also be applied to other approaches that translate power limits into settings for combinations of resources [59, 64, 124, 158].

In our experiments, ALERT considers a series of power settings within the feasible range with 2.5W interval on our test laptop and a 5W interval on our test CPU server and GPU platform, as the latter has a wider power range than the former. The number of power buckets is configurable.

ALERT incurs small overhead in both scheduler computation and switching from one DNN/power-setting to another, just 0.6–1.7% of an input inference time. We explicitly account for overhead by subtracting it from the user-specified goal.

Users may set goals that are not achievable. If ALERT cannot meet all constraints, it prioritizes latency highest, then accuracy, then power. This hierarchy is configurable.

	Embedded	CPU1	CPU2	GPU
CPU	ARM Cortex A-15 @2.0 GHz	Core-i7 @2.2 GHz	Xeon(R) Gold 6126 @2.60GHz	Core-i7 @2.2 GHz
GPU	none	none	none	RTX 2080
Memory	DDR3 2G	DDR4 16G	DDR4 16G*12	DDR4 16G
LLC	2MB	9MB	19.25MB	9MB

Table 3.1: Hardware platforms used in our experiments

	Run-time environment setting		
Default	Inference task has no co-running process		
Memory	Co-locate with memory-hungry STREAM [101] (@CPU) Co-locate with Backprop from Rodinia-3.1 [26] (@GPU)		
Compute	Co-locate with Bodytrack from PARSEC-3.0 [20] (@CPU) Co-locate with the forward pass of Backprop [26] (@GPU)		
	Ranges of constraint setting		
Latency	0.4x–2x mean latency* of the largest Anytime DNN		
Accuracy	Whole range achievable by trad. and Anytime DNN		
Energy	Whole feasible power-cap ranges on the machine		
Task	Trad. DNN	Anytime [145]	Fixed deadline?
Image Classifi.	Sparse ResNet	Depth-Nest	Yes
Sentence Pred.	RNN	Width-Nest	No
Scheme ID	DNN selection		Power selection
Oracle	Dynamic optimal		Dynamic optimal
$\text{Oracle}_{\text{Static}}$	Static optimal		Static optimal
App-only	One Anytime DNN		System Default
Sys-only	Fastest traditional DNN		State-of-Art[65]
No-coord	Anytime DNN w/o coord. with Power		State-of-Art[65]
ALERT	ALERT default		ALERT default
ALERTAny	ALERT w/o traditional DNNs		ALERT default
ALERTTrad	ALERT w/o Anytime DNNs		ALERT default

Table 3.2: Settings and schemes under evaluation (* measured under default setting without resource contention)

3.5 Evaluation

We apply ALERT to different inference tasks on both CPU and GPU with and without resource contention from co-located jobs. We set ALERT to (1) reduce energy while satisfying latency and accuracy requirements and (2) reduce error rates while satisfying latency and energy requirements. We compare ALERT with both oracle and state-of-the-art schemes and evaluate detailed design decisions.

3.5.1 Methodology

Experimental setup. We use the three platforms listed in Table 3.1: *CPU1*, *CPU2*, and *GPU*. On each, we run inference tasks[3], image classification and sentence prediction, under three different resource-contention scenarios:

- No contention: the inference task is the only job running, referred to as "Default";
- Memory dynamic: the inference task runs together with a memory-intensive job that repeatedly stops and restarts, representing dynamic memory resource contention, referred to as "Memory";
- Computation dynamic: the inference task runs together with a computation-intensive job that repeatedly stops and restarts, representing dynamic computation resource contention, referred to as "Compute".

Schemes in evaluation. We give ALERT three different DNN sets, traditional DNN models (ALERTTrad), an Anytime DNN (ALERTAny), and both (ALERT), and compare it with two oracle and three state-of-the-art schemes (Table 3.2).

The two Oracle_* schemes have perfect predictions for every input under every DNN/power setting (i.e., impractical). Specifically, the "Oracle" allows DNN/power settings to change across inputs, representing the best possible results; the "$\text{Oracle}_{\text{Static}}$" has one fixed setting across inputs, representing the best results without dynamic adaptation.

The three state-of-the-art approaches include the following:

- "App-only" conducts adaptation only at the application level through an Anytime DNN [145];

3. For GPU, we only run image classification task there, as the RNN-based sentence prediction task is better suited for CPU [160].

- "Sys-only" adapts only at the system level following an existing resource-management system that minimizes energy under soft real-time constraints [106][4] and uses the fastest candidate DNN to avoid latency violations;
- "No-coord" uses *both* the Anytime DNN for application adaptation *and* the power-management scheme [106] to adapt power, but with these two working independently.

3.5.2 Overall Results

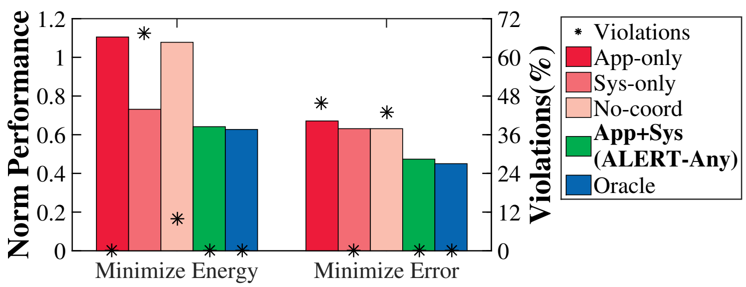


Figure 3.2: Average performance normalized to $\text{Oracle}_{\text{Static}}$ (Smaller is better). Violations% is %-of-constraint-settings under which a scheme incurs >10% violation of all inputs.

Table 3.3 shows the results for all schemes for different tasks on different platforms and environments. Each cell shows the average energy or accuracy under 35–40 combinations of latency, accuracy, and energy constraints, normalized to the $\text{Oracle}_{\text{Static}}$ result. Figure 3.2 compares these results, where lower bars represent better results and lower *s represent fewer constraint violations.ALERT and $\text{ALERT}_{\text{Any}}$ both work very well for all settings. They outperform state-of-the-art approaches, which have a significant number of constraint violations, as visualized by the many superscripts in Table 3.3 and the high * positions in Figure 3.2. ALERT outperforms $\text{Oracle}_{\text{Static}}$ because it adapts to dynamic variations. ALERT also comes very close to the theoretically optimal Oracle.

4. Specifically, this adaptation uses a feedback scheduler that predicts inference latency based on Kalman Filter.

Plat.	DNN	Work.	ALERT	ALERT Any	Sys-only	App-only	No-coord	Oracle	ALERT	ALERT Any	Sys-only	App-only	No-coord	Oracle
			Energy in Minimizing Energy Task						Error Rate in Minimizing Error Task					
CPU1	Sparse Resnet	Idle	0.64	0.68	1.08^{19}	1.19	0.94^{1}	0.64	0.91	0.92	1.35	1.02^{3}	0.91^{3}	0.89
		Comp.	0.57	0.58	0.80^{19}	1.30	1.39^{1}	0.57	0.38	0.39	0.51	1.35^{24}	0.39^{6}	0.36
		Mem.	0.53	0.55	0.76^{19}	1.43	1.37^{2}	0.53	0.34	0.34	0.46	1.47^{28}	0.39^{2}	0.33
	RNN	Idle	0.61	0.65	1.01^{30}	1.34	0.95^{2}	0.61	0.87	0.87	0.87	0.87^{21}	0.87^{14}	0.86
		Comp.	0.60	0.57	0.93^{30}	1.21	1.26^{5}	0.60	0.42	0.44	0.50	0.46^{28}	0.46^{23}	0.42
		Mem.	0.54	0.56	0.95^{31}	1.45	1.24^{9}	0.54	0.45	0.45	0.50	0.57^{28}	0.54^{27}	0.44
CPU2	Sparse Resnet	Idle	0.93	0.88	0.96^{20}	0.99	1.18	0.91	0.68	0.68	0.97	0.79^{2}	0.71^{24}	0.66
		Comp.	0.59	0.57	0.60^{23}	1.00	1.01	0.58	0.58	0.57	0.85	0.74^{16}	0.71^{29}	0.55
		Mem.	0.38	0.37	0.39^{19}	0.65	0.63^{13}	0.38	0.24	0.82	0.32	0.33^{17}	0.75^{31}	0.21
	RNN	Idle	0.87	0.99	0.80^{34}	1.04	1.00^{6}	0.83	0.84	0.85	0.99	0.89^{14}	0.89^{1}	0.84
		Comp.	0.60	0.60	0.55^{34}	0.99	0.86^{7}	0.60	0.51	0.52	0.60	0.53^{21}	0.54^{17}	0.52
		Mem.	0.52	0.51	0.43^{33}	0.70	0.85^{14}	0.52	0.26	0.27	0.31	0.28^{21}	0.27^{17}	0.26
GPU	Sparse Resnet	Idle	0.97	0.99	0.92^{20}	1.36	1.37	0.92	0.90	0.92	1.22	1.09^{2}	1.74^{12}	0.86
		Comp.	0.96	0.97	0.94^{20}	1.66	1.77	0.89	0.32	0.34	1.28	1.21^{23}	2.50^{18}	0.30
		Mem.	0.97	1.01	0.91^{20}	1.39	1.43	0.91	0.89	0.92	1.22	1.11^{2}	1.81^{14}	0.86
Harmonic mean			0.64	0.64	0.73^{27}	1.11	1.08^{4}	0.62	0.46	0.47	0.63	0.67^{16}	0.63^{15}	0.45

Table 3.3: Average energy consumption and error rate normalized to $Oracle_{\text{Static}}$.
(Smaller is better; Each cell is averaged over 35–40 constraint settings; superscript: # of constraint settings violated for >10% inputs and hence excluded from energy average.)

Plat.	Work.	ALERT	Any	Trad	ALERT	Any	Trad
		Minimize Energy Task			Minimize Error Task		
CPU1	Idle	0.64	0.68	0.65^{1}	0.91	0.92	0.93
	Comp.	0.57	0.58	0.65^{6}	0.38	0.39	0.41
	Mem.	0.53	0.55	0.53^{3}	0.34	0.34	0.35
CPU2	Idle	0.93	0.88	0.95^{1}	0.68	0.68	0.69
	Comp.	0.59	0.57	0.60^{4}	0.58	0.57	0.59
	Mem.	0.38	0.37	0.40^{8}	0.23	0.24	0.32
GPU	Idle	0.97	0.99	0.95	0.90	0.92	0.89
	Comp.	0.97	1.01	0.96	0.89	0.92	0.89
	Mem.	0.96	0.97	0.95	0.32	0.34	0.32
Harmonic mean		0.66	0.66	0.67^{3}	0.47	0.48	0.50

Table 3.4: ALERT normalized average energy consumption and error rate to $Oracle_{\text{Static}}$ @ Sparse ResNet (Smaller is better)

3.5.3 Detailed Results and Sensitivity

Different DNN candidate sets. Table 3.4 compares the performance of ALERT working with an Anytime DNN (Any), a set of traditional DNN models (Trad), and both. At a high level, ALERT works well with all three DNN sets. Under close comparison, ALERTTrad violates more accuracy constraints than the others, particularly under resource contention on CPUs, because a traditional DNN has a much larger accuracy drop than an anytime DNN when missing a latency deadline. Consequently, when the system variation is large, ALERTTrad selects a faster DNN to meet latency and thus may not meet accuracy goals.

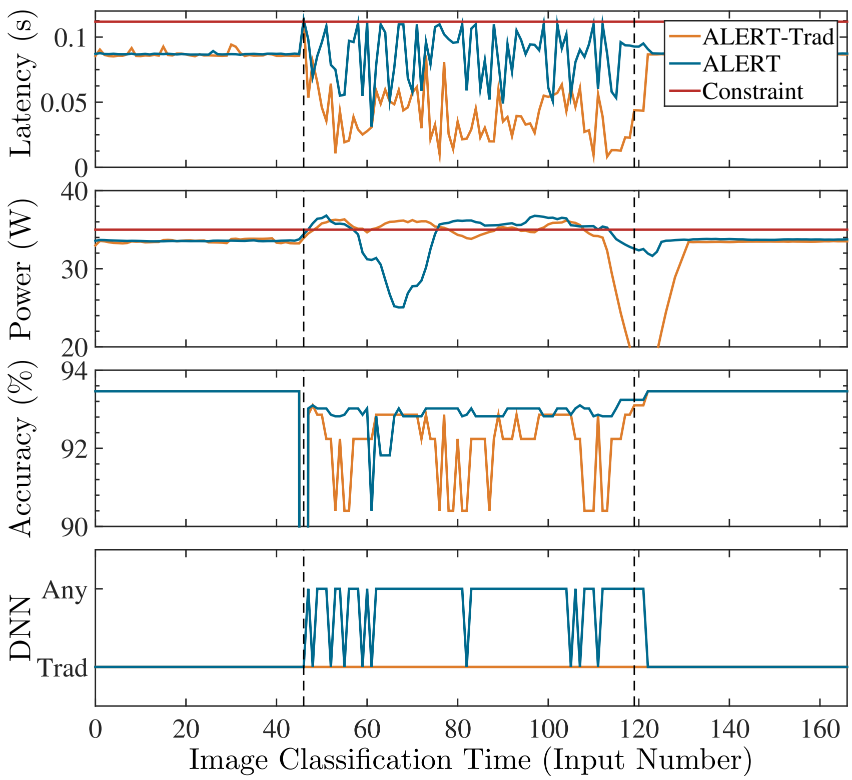


Figure 3.3: Minimize error rates w/ latency, energy constraints on CPU1. (Memory contention occurs from about input 46 to 119; Deadline: 1.25× mean latency of largest Anytime DNN in Default; power limit: 35W.)

Of course, ALERTAny is not always the best. As discussed in Section 2.2.2, Anytime DNNs sometimes have lower accuracy then a traditional DNN with similar execution time. This difference leads to the slightly better results for ALERT over ALERTAny.

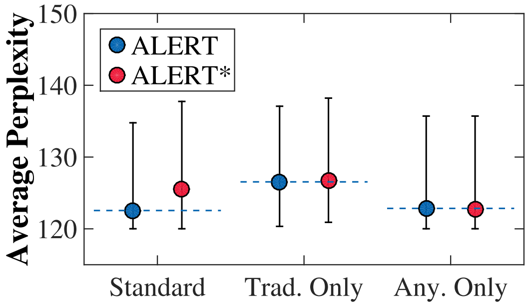
Figure 3.3 visualizes the different dynamic behavior of ALERT (blue curve) and $\text{ALERT}_{\text{Trad}}$ (orange curve) when the environment changes from Default to Memory-intensive and back. At the beginning, due to a loose latency constraint, ALERT and $\text{ALERT}_{\text{Trad}}$ both select the biggest traditional DNN, which provides the highest accuracy within the energy budget. When the memory contention suddenly starts, this DNN choice leads to a deadline miss and an energy-budget violation (as the idle period disappeared), which causes an accuracy dip. Fortunately, both quickly detect this problem and sense the high variability in the expected latency. ALERT switches to use an anytime DNN and a lower power cap. This switch is effective: although the environment is still unstable, the inference accuracy remains high, with slight ups and downs depending on which anytime output finished before the deadline. Only able to choose from traditional DNNs, $\text{ALERT}_{\text{Trad}}$ conservatively switches to much simpler and hence lower-accuracy DNNs to avoid deadline misses. This switch does eliminate deadline misses under the highly dynamic environment, but many of the conservatively chosen DNNs finish before the deadline (see the Latency panel), wasting the opportunity to produce more accurate results and causing $\text{ALERT}_{\text{Trad}}$ to have a lower accuracy than ALERT. When the system quiesces, both schemes quickly shift back to the highest-accuracy, traditional DNN.

Overall, these results demonstrate how ALERT always makes use of the full potential of the DNN candidate set to optimize performance and satisfy constraints.

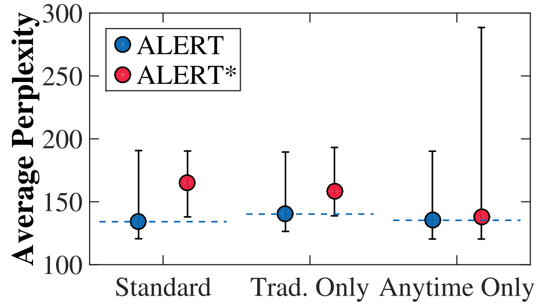
ALERT probabilistic design. A key feature of ALERT is its use of not just mean estimations, but also their variance. To evaluate the impact of this design, we compare ALERT to an alternative design ALERTAlter, which only uses the estimated mean to select configurations.

Figure 3.4 shows the performance of ALERT and ALERTAlter in the minimize error task for sentence prediction. Here, ALERT (blue circles) always performs better than ALERTAlter. Its advantage is the biggest when the DNN candidates include both traditional and Anytime DNNs (i.e., the "Standard" in Figure 3.4). The reason is that traditional DNNs and Anytime DNN have different accuracy/latency curves, Eq. 3.4 for the former and Eq. 3.8 for the latter. ALERTAlter is much worse in distinguishing these two by simply using the mean of estimated latency to predict accuracy. ALERT also clearly outperforms ALERTAlter under memory contention with traditional DNN candidates, as ALERT's estimation better captures dynamic system variation. Overall, these results show ALERT's probabilistic design is effective.

Sensitivity to latency distribution. ALERT assumes a Gaussian distribution, but is designed to work for other distributions (see Section 3.3). As shown in Figure 3.5, the



(a) Default Contention



(b) Memory Contention

Figure 3.4: Minimize error for sentence prediction@ CPU1 (Lower is better). (whisker: whole range; circle: mean)

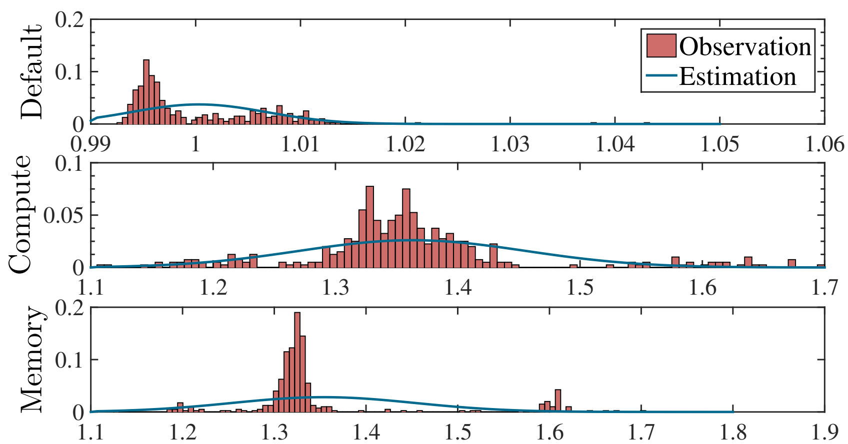


Figure 3.5: Distribution of ξ for image class. on CPU1.

observed ξs (red bars) are indeed not a perfect fit for Gaussian distribution (blue lines), which confirms ALERT's robustness.

3.6 Conclusion

This work tackles the important problem of ensuring timely, accurate, and energy efficient neural network inference with dynamic input, contention, and requirement variation. ALERT achieves these goals through dynamic and coordinated DNN model selection and power management based on feedback control. We evaluate ALERT with a variety of workloads and DNN models and achieve high performance and energy efficiency.

CHAPTER 4

ARE MACHINE LEARNING CLOUD APIS USED CORRECTLY?

4.1 Overview

In this chapter, we present a comprehensive empirical study on a set of open-source applications that use Google or AWS cloud-based ML APIs. We manually studied the *latest* versions—as of August 1, 2020—of 360 applications that include non-trivial use of ML APIs and cover all the three ML domains offered by them: vision, speech, and language.

Our study faces the challenge of lacking existing issue-tracking system records about ML API misuses, given the short history of ML APIs. Consequently, we carefully study these 360 projects and discover previously *unknown* misuses in their latest versions by ourselves.

Our study found that misuses of ML APIs are widespread and severe: 247 out of these 360 applications (69 %) contain misuses in their latest versions, more than half of which contain more than one type of misuse.

These misuses lead to various types of problems, including 1) *reduced functionality*, such as a crash or a quality-reduced output; or 2) *degraded performance*, like an unnecessarily extended interaction latency; or 3) *increased cost*, in terms of payment for cloud services. Their root causes are all related to unique challenges for ML APIs: complicated data requirements, complicated cognitive semantics, and complicated input-accuracy-performance-cost tradeoffs.

Our study reveals common misuse patterns that are found in many different applications, often with simple fixes that avoid failures, improve performance, and reduce cost. Therefore, as a final contribution, we design several checkers and small API changes (in the form of wrapper functions) that both check for and handle common errors. Many more misuses are found by our checkers, beyond the 360 projects in the initial study.

4.2 Methodology

4.2.1 Application selection

Our work looks at applications that use Google Cloud AI and Amazon AI, the two most popular cloud AI services on Github, with thousands of applications using each type of their AI services. Our work will target the following two sets of applications (all latest versions as of *Aug. 1st, 2020*), one for all our manual studies and one for our automated checking.

For automated checking, we use *all* the 12666 Python applications on GitHub that use Google or AWS AI service.

For manual studies, we collect a suite of 360 non-trivial applications that use Google/Amazon ML APIs, including 120 applications for each of the three major ML domains. They cover different programming languages, Python(80%), JS (13%), Java (3%), and others (4%). Around 80% of these applications use Google Cloud AI and around 20% use AWS AI, with 1% using both. The sizes of these applications range from 46 to 3 millions lines of code, with 2228 lines of code being the median size and around 40% of them having more than 10 thousand lines of code. Most of these applications are young, created after 2018 (98% of them). They have a median age of around 18 months at the time of our study. This relatively young age distribution reflects the fact that the power of deep learning has only been recently recognized, and yet is being adopted with unprecedented pace and breadth.

Since there are many toy applications on GitHub, we manually checked about 1200 randomly selected applications, which use Google/Amazon ML APIs, to obtain these 360 non-trivial applications. We manually confirmed they each target a concrete real-world problem, integrate the ML API(s) in their workflow, and conduct some processing for the input or the output of the ML API, instead of simply feeding an external file into the ML API and directly printing out the API result.

4.2.2 Anti-pattern identification methodology

Because of the young ages of ML API services and hence the applications under study, we could *not* rely on known API misuses in their issue-tracking systems, which are very rare. Instead, we must discover API misuses *unknown to the developers* by ourselves.

Since there is no prior study on ML API misuses, our misuse discovery can not rely on any existing list of anti-patterns. Instead, our team, including ML experts, carefully studies API manuals, intensively profiles the API functionality and performance, and then manually examines every use of an ML API in each of the 360 applications for potential misuses. For every suspected misuse, we design test cases and run the corresponding application or application component to see if the misuse truly leads to reduced functionality, degraded performance, or increased cost comparing with an alternative way of using ML APIs, which we designed. When one misuse is identified, we generalize it and check if there are similar misuses in other applications. We repeat this process for many rounds until we converge to the results presented in this paper. During this process, we report representative misuses to corresponding application developers, receiving confirmation for many cases. All the manual checking is conducted by two of the authors, with their results discussed and checked by all the co-authors.

We identify a wide variety of applications as containing ML API misuses including those both: small and large, young and old, AWS and Google-API based. This variety of misuses indicates that they are not rare mistakes by individual programmers and do not appear to diminish with software growth, age, or API provider.

4.2.3 Profiling methodology

We profile several projects to evaluate their performance before and after optimization. We use real-world vision, audio, or text data that fits the scenario of corresponding software. We profile the end-to-end latency for each related module and also the whole process: from

What challenges did developers encounter?	Related APIs and Inputs	Service Provider	Impact	# (%) of Problematic Apps. Manual	Auto
Should Have Called a Different API					
Complicated cognitive semantic overlap across APIs	text-detection vs. document-text-detection	G	Low Accuracy	6 (11%)	-
	image-classification vs. object-detection	AG	Low Accuracy	5 (9%)	-
	sentiment-detection vs. entity-sentiment-detection	G	Low Accuracy	4 (5%)	-
Complicated tradeoffs: Input-Accuracy-Performance	ASync vs. Sync Language-NLP	A	Slower	-	3 (43%)
	ASync vs. Sync Speech Recognition	G	Slower	7 (78%)	203 (83%)
	ASync vs. Sync Speech Synthesis	A	Slower	-	2 (22%)
Unaware of parallelism APIs	Vision-Image API vs. annotate-image	AG	Slower	7 (78%)	-
	Language-NLP API vs. annotate-text	AG	Slower	11 (100%)	-
	Regular API vs Batch API	AG	Slower	Workload dependent	
Should Have Skipped the API call					
Complicated tradeoffs: Input-Performance	Speech Synthesis APIs with constant inputs	AG	Slower, More Cost	15 (25%)	279 (17%)
Complicated tradeoffs: Accuracy-Performance	Vision-Image APIs with high call frequency	AG	Slower, More Cost	3 (3%)	-
Should Have Converted the Input Format					
Complicated data requirements	all APIs without input validation, transformation	AG	Exceptions	206 (57%)	-
Complicated tradeoffs: Input-Accuracy-Performance	Vision-Image APIs with high resolution inputs	AG	Slower	106 (88%)	-
Complicated tradeoffs: Input-Accuracy-Cost	Language-NLP APIs with short text inputs	AG	More Cost	4 (3%)	-
	Speech recognition APIs with short audio inputs	AG	More Cost	1 (2%)	-
	Speech synthesis APIs with short audio inputs	AG	More Cost	1 (2%)	-
Should Have Used the Output in Another Way					
Complicated outputsemantics	sentiment-detection	G	Low Accuracy	24 (39%)	360 (37%)
Total number of benchmark applications with at least one API misuse		AG		249 (69%)	

Table 4.1: ML API misuses identified by our Manual checking and Automated checkers. ("A" is for AWS and "G" for Google. The %s of problematic apps are based on the total # of apps using corresponding APIs in respective benchmark suite. Note that, 133 apps contain more than one type of API misuses; the average number of API misuses in each application is 1.3.)

user input to final output. By default, we run each application under profiling five times for each input and reported the average latency.

All experiments were done on the same machine, which contains a 16-core Intel Xeon E5-2667 v4 CPU (3.20GHz), 25MB L3 Cache, 64GB RAM, and 6×512GB SSD (RAID 5). It has a 1000Mbps network connection, with twisted pair port. Note that all the machine-learning inference is done by cloud APIs remotely, instead of on the machine locally.

4.3 Functionality-related API Misuses

Through manual checking, we identified three main types of API misuses that commonly affect the functional correctness of applications, as listed in Table 4.1 (white-background rows). They are typically caused by developers' misunderstanding of the semantics or the input data requirements of machine learning APIs, and can lead to unexpected loss of accuracy and hence software misbehavior that is difficult to diagnose.

Note that, although the high-level patterns of these misuses, such as calling the wrong

API and misinterpreting the outputs, naturally occur in general APIs, the exact root causes, code anti-patterns, and tackling/fixing strategies are all unique to ML APIs, as we discuss below.

Calling the wrong API. Unlike traditional APIs that are *programmed* to each conduct a clearly coded task, ML APIs are *trained* to perform tasks emulating human behaviors, with functional overlap among some of them. Without a good understanding of these APIs, developers may call the wrong API, which could lead to severely degraded prediction accuracy or even a completely wrong prediction result and software failures.

Misinterpreting outputs. Related to the probabilistic nature of cognitive tasks, DNN models operate on high-dimensional continuous representations, yet often ultimately produce a small discrete set of outputs. Consequently, ML APIs' outputs can contain complicated, easily misinterpretable semantics, leading to bugs.

Missing input validation. Inputs to ML APIs are typically real-world audio, image, or video content. These inputs can take many different forms, with different resolutions, encoding schemes, and lengths. Unfortunately, developers sometimes do not realize that not all forms are accepted by ML APIs, nor do they realize that such input incompatibility can be easily solved through format conversion, input down-sampling, or chunking. As a result, lack of input validation and incompatibility handling are very common, and can easily cause software crashes.

4.4 Performance-related API Misuses

Through manual checking, we identify and categorize 4 main types of ML API mis-uses that can lead to huge performance loss and user experience damage (see Table 4.1, blue-background rows). They are typically related to ML APIs' complicated tradeoffs among input-transformation effort, performance, and accuracy.

Misuse of asynchronous APIs. The same ML task can often be performed with multiple APIs, a synchronous version, an asynchronous version, and sometimes a streaming version. The different versions have complicated and sometimes counter-intuitive tradeoffs between input transformation, performance, and accuracy that often confuse developers and lead to surprisingly wide-spread and severe misuses based on our study.

Forgetting parallel APIs. Some ML APIs are offered to ease task and data parallelism, but are rarely used even when doing so would require only a simple change to the application.

Making skippable API calls. Sometimes, an API call can be skipped at the cost of slightly higher engineering effort or slight, but often indiscernible by human, functionality difference. Lack of understanding of these tradeoffs leads to some unnecessary API calls. It is typically related to API calls with constant inputs and API calls with excessive frequency.

Unnecessarily high-resolution inputs. Vision APIs accept inputs with a range of resolutions and impose a complicated tradeoff among input, performance, and accuracy that is often ignored by developers—with higher input resolution, the performance degrades greatly, while the inference accuracy increases and then saturates quickly (Figure 4.1).

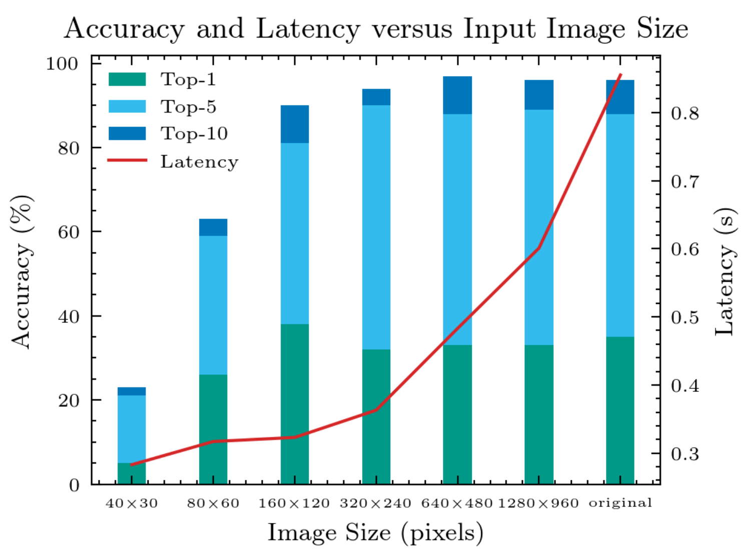


Figure 4.1: Accuracy and latency with different input resolutions.

4.5 Cost-related API Misuses

Every ML API call costs money. Naturally, some performance problems, particularly all of those skippable calls in Section 4.4, also waste money. In addition, the round-up charging policy leads to a unique anti-pattern: since every API call is charged based on the input size rounded-up, calls with very small inputs may be economically sub-optimal. The possibility of combining multiple calls with small inputs creates a complicated tradeoff problem among input transformation, accuracy, performance, and cost.

4.6 Solutions

We have implemented checkers and wrappers to automatically detect and fix some of the anti-patterns introduced in Section 4.3-4.5. The auto-detection tools are implemented with Jedi[52], AST[120] and PyGitHub[119] library. They include:

Output Misinterpretation Checker to automatically detect mis-uses of the `sentiment-detection` API's output, a type of accuracy bugs discussed in Section 4.3. Among the 975 GitHub Python applications that use this API, our checker finds 360 of them interpreting the API output incorrectly.

Asynchronous API call checker to check whether asynchronous APIs are used in a synchronous, blocking way, a type of performance mis-use discussed in Section 4.4. Our checker automatically reports 313 minuses from 523 Python applications using asynchronous ML APIs.

Constant-parameter API call checker to automatically identify speech synthesis API calls that use constant inputs, a type of performance mis-use discussed in Section 4.4. Applied to Python GibHub 686 (943) applications on using Google's (AWS's) speech synthesis API, our checker finds 202 (196) problematic applications.

API wrappers for all three domains of ML APIs. They tackle the anti-patterns of missing input validation (Section 4.3). unnecessarily high-resolution inputs, forgetting parallel APIs, misuse of asynchronous APIs (Section 4.4), and the money wasting problem in Section 4.5.

4.7 Threats to Validity

Internal threats to validity. The inputs used in our performance profiling and inference-accuracy measurement may not represent the exact workload used by real-world users. Our static checkers can have false positives and false negatives.

External threats to validity. We only studied ML APIs offered by Google and AWS in this work, but not those offered by other service providers. Our study only covers cloud APIs with pre-trained DNNs designed for general purpose use, and excludes user-defined DNNs based on their specific needs. We only study open-source projects on GitHub, with no access to those closed-source commercial projects. The 360 applications in our manual study benchmark suite may not represent all real-world applications. Our static analysis tool currently only covers python applications.

4.8 Conclusion

This work presents the first in-depth study of real-world applications using machine learning cloud APIs. By investigating the latest versions of 360 open-source applications using Google and AWS ML Cloud APIs, we found 8 types of common API misuses that cause functionality, performance, and service cost problems. It provides guidance to help prevent errors while improving the functionality, performance, and cost of these applications. We also develop static checkers to automatically detect some of these problems in a larger set of applications. The wide presence of these problems motivates future research to further tackle ML API misuses.

CHAPTER 5

AUTOMATED TESTING OF SOFTWARE THAT USES MACHINE LEARNING APIS

5.1 Overview

In this chapter, we propose Keeper, a testing tool designed for software that uses cognitive machine learning APIs (ML software).

Keeper designs a set of pseudo-inverse functions for cognitive ML APIs[1]. For an API f that maps inputs from domain $\mathbb{I}$ to outputs in domain $\mathbb{O}$, its pseudo-inverse function f' reverses this mapping at the semantic level. We make sure that the mapping by f' has been confirmed by many people to have high accuracy. For example, the Bing image search engine is a pseudo-inverse function of Google's image classification API.

Keeper then integrates the pseudo-inverse functions with symbolic execution to reach the sparse program-relevant input space. Specifically, Keeper first uses symbolic execution to figure out what values an ML-API output can take to fulfill branch coverage. Keeper then automatically generates realistic inputs that are expected to produce the desired ML-API outputs, leveraging pseudo-inverse functions.

Keeper also makes pseudo-inverse functions a proxy of human judgement and automatically judges the correctness of software outputs that are related to cognitive tasks. Since our pseudo-inverse functions are *not* analytically inverting ML APIs (i.e., $f'(f(i)) \neq i$ is possible), a test input generated by Keeper may not cover the targeted software branch. At the same time, since these pseudo-inverse functions have been approved by many human beings, Keeper reports an *accuracy failure* when over a threshold portion of inputs fail to cover a particular target branch.

1. The current implementation of Keeper supports Google Cloud AI APIs and can be easily extended to support similar APIs from other service providers.

Of course, Keeper also monitors generic failure symptoms like crashes during test runs, and helps expose bugs in code regions that require specific ML inputs to exercise.

Finally, to help developers understand the root cause of an accuracy failure, Keeper explores alternative ways of using ML APIs and informs the developers of any code changes that can alleviate the accuracy failure.

Putting these all together, we have implemented Keeper that can be used either through a command-line script or a plug-in inside the VScode IDE [105]. Given a software application, Keeper first highlights all the functions that directly or indirectly call ML APIs. For any function that developers want to test, Keeper automatically generates many test cases to thoroughly test every branch in the specified function and its callees. Keeper analyzes the test runs and reports any failures, as well as potential patches for accuracy failures, to developers.

We evaluate Keeper on the latest version of 63 open-source Python applications that cover different problem domains and ML APIs. Keeper achieves 91% branch coverage on average for these applications. In total, Keeper covers 21–38% more branches than alternative techniques that directly use machine learning training data set or random fuzzing. Keeper exposes 35 unique accuracy and crash failures from 25 out of these 63 applications.

5.2 Test input generation

Keeper is a testing tool for software whose control flow is influenced by ML APIs. As shown in Figure 5.1, Keeper includes two major components: test-input generation, which we present in this section, and test-output processing, which we present in Section 5.3.

Keeper's input generation is built upon an existing symbolic execution engine, DSE [66]. Given a function F to test[2] and all the function parameters represented as symbolic variables, a symbolic path constraint is generated for every branch; solving all the path constraints

2. Users of Keeper can choose any function to test, including the `main` function.

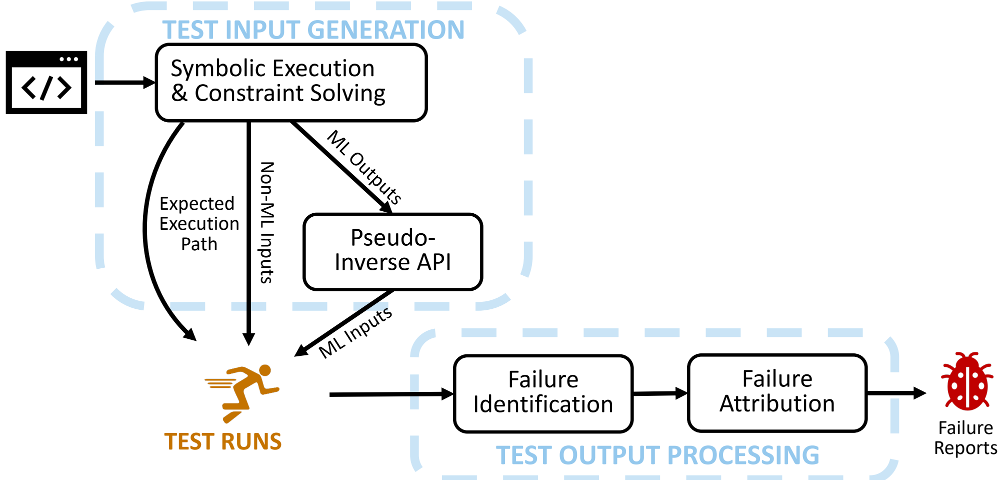


Figure 5.1: An overview of Keeper.

produces a test suite that offers full branch coverage.

Keeper decomposes the problem of generating inputs for ML APIs into two parts: first, it identifies the ML-API outputs that are needed to satisfy path constraints using symbolic execution (Section 5.2.1); and then synthesizes the ML-API inputs that are expected to produce those outputs using carefully designed pseudo-inverse functions (Section 5.2.2). As we will see, this decomposition not only avoids the complexity of directly applying symbolic execution to DNNs, but also help judge the execution correctness (Section 5.3).

5.2.1 Identifying relevant ML outputs

To identify the desired ML-API outputs, Keeper makes its symbolic execution skip any statement that calls a ML API and instead marks API output that is used by following code as symbolic. This way, the output, instead of input, of ML APIs will be part of the path constraints, and by solving the constraints, Keeper obtains the API-output values that are needed to exercise corresponding branches.

The only tweak Keeper makes here is to have the symbolic execution engine sometimes generating one path constraint for each branch sub-condition, instead of the whole branch. In our implementation, this is accomplished by enabling a corresponding feature of the underlying symbolic execution engine. For example, for a branch condition “A or B or C”, four constraints will be formed representing (1) A is True, (2) B is True, (3) C is True, and (4) none of A, B, C is True. Solving these constraints leads to four inputs or input sets that satisfy these constraints separately.

	ML Task	Main Output	Constraint Example	Pseudo-inverse Function
Vision	Image classification	image class	class==”fire” [117]	Search on internet, keyword: [image class]
	Object detection	object name	object==”tableware” [123]	Search on internet, keyword: [object name]
	Face detection	face emotion	emotion ==”joy” [33]	Search on internet, keyword: [emotion] + ”human face”
	Text detection	extracted text	text==”3923-6625” [110]	Print [extracted text] on an image
Lang.	Document classification	document class	class==”food” [109]	Search on internet, keyword: [document class]
	Sentiment detection	score, magnitude	score< 0 [134]	Select tweets from Sentiment140 dataset [45]
	Entity detection	entity name, type	type==”Person” [82]	Use text generation technique, seed: [name] or [type]
Speech	Speech recognition	transcript	text==”turn on the light” [141]	Use speech synthesize technique on [transcript]

Table 5.1: Different ML APIs handled by Keeper and their pseudo-inverse functions.

5.2.2 *Identifying ML API inputs*

Given a ML API f and an output o, Keeper aims to automatically generate a set of inputs I so that $f(i), i \in I$ is *expected* to produce o according to common human judgement. To achieve this, Keeper designs a pseudo-inverse function f' for every API f, so that $f'(o)$ will produce the input set I for f. We want f' to have the following properties.

First, f' is not an analytical inversion of f. Ideally, f' should be built independently from f (e.g., not based on the same training data set), so that f' can help not only input generation but also failure identification in a way similar to N-version programming.

Second, f' should be a semantic inverse of f, reversing the cognitive task performed by f in a way that is consistent with most human beings. This way, test inputs generated by Keeper can expect to cover most of the software branches, unless the ML API is unsuitable for the software or is used incorrectly.

Third, f' should produce more than one output for each input it takes in. This will allow Keeper to generate multiple inputs for f to exercise a corresponding branch, and get a statistically meaningful test result given the probabilistic nature of ML APIs.

With these goals in mind, we have designed three types of pseudo-inverse functions as summarized in Table 5.1.

Search-based pseudo inversion

For many vision and language APIs, search engines offer effective pseudo inversion: they take in a key word and return a set of realistic images/texts that reflect the keyword. Search engines have several properties that serve Keeper's testing purposes. First, they offer great semantic inversion, as there are multiple search engines that have been used by hundreds of millions of users for many years with high satisfaction [25]. Their top search results typically match the common human judgement. Second, they are not an analytical inversion of ML APIs, and we will use non-Google engines to minimize potential correlations. Third, they accept a wide range of search words and produce many ranked results, which means a large number of high-quality test inputs for Keeper. Specifically, Keeper uses different engines and search keywords for different ML APIs:

Vision tasks. Image-classification and object-detection APIs return string labels that describe the image and the objects inside the image, respectively. For both APIs, Keeper uses the Bing [21] image search engine and uses the desired label description or object name as the search keyword.

The face-detection API detects human faces in an image. Some ML software uses the returned emotion string associated with each face (e.g., "joy", "sorrow", etc.) to decide execution path. To generate corresponding images, Keeper uses "[emotion] human face" as a keyword to search the Bing image.

Language tasks. Document-classification APIs process a document and return categories based on the document content, like "pets", "health", "sports", and others. Keeper uses the desired category name as keyword and searches it at (1) knowledge graph websites, Wikipedia [7] and Britannica [2]; and (2) Bing web search engines. Keeper then uses the text extracted out from each returned web page as the ML API input.

Synthesis-based pseudo inversion

The semantic inversion of some ML APIs does not match the functionality of search engines. Fortunately, we find ways to synthesize inputs for them.

The **text-detection** API extracts printed or handwritten text from an image. Unfortunately, image search engines tend to return images whose content reflects the search keyword, instead of images that contain the keyword as text within the image. Therefore, given a text string, Keeper prints it on a background image using the Python pillow library [3]. Keeper adopts both printed and hand-writing fonts; different font settings produce different test images. To decide the background image, Keeper checks whether the `text-detection` API shares its input image with another vision API. If so, the test images Keeper generated for the other API will be used as the background; otherwise, a blank image and some random images will be used. Figure 5.2 shows some of the test images that Keeper generates for application wanderStub [149], which has a branch checking if the input image contains "Total".

Figure 5.2: Test inputs generated for wanderStub [149].

The **entity-detection** API inspects the input sentence for known entities—there are in total 13 entities, such as ADDRESS, DATE, etc. Since the search engines usually return long documents, Keeper instead uses a popular language model GPT-2 [121] to synthesize any number of sentences that start with a pre-defined word/phrase that corresponds to the desired entity type.

The **speech-recognition** API transcribes the input audio clip and outputs the transcript. Keeper uses speech synthesis tools, particularly the pyttsx3 [5] Python library, to generate the desired audio clips based on a given transcript. Keeper generates multiple audio clips using different voice settings supported by this library.

ML benchmarks for pseudo inversion

The **sentiment detection** API presents two challenges. First, although this API aims to identify the prevailing emotional opinion within the text, it does not directly output a categorical result. Instead, it returns two floating-point numbers, `score` and `magnitude`, for developers to derive emotion categories from. There is no perceivable way to generate text that can offer the exact `score` or `magnitude`. Second, even if we just hope to generate text that contains positive or negative emotion, no search engine or synthesizer can accomplish this.

Facing these challenges, Keeper resorts to the Sentiment140 dataset [45], which contains 1,600,000 tweets, manually labelled as positive, negative, and neutral. Keeper randomly samples the same number of positive, negative, and neutral tweets as test inputs for any sentiment-detection API called inside a ML software, with the expectation that these tweets will help cover different branches in the software that are designed for different emotions.

Note that, we treat ML benchmarks as the last resort for multiple reasons. First, the labels associated with data inside ML benchmarks either have few categories or have limited quality. For example, ImageNet [28] contains 1000 manually labeled image categories, which is too few compared with the 20,000 labels of Google Vision AI. On the contrary, OpenImage has 9 million images with 20,000 labels. However 89% of the labels are generated by DNNs, and 53% of the human-verified ones are incorrect [85]. Second, ML benchmarks are built with pre-processed real-world data. Such "clean" data has less variety, as they share similar size, resolution, and encoding format. Third, some benchmarks may be part of the training data set of Google ML APIs, which makes the test inputs biased towards the ones APIs can perform well on and hence less likely to reveal problems. Finally, Generative Adversarial

Network synthesizes new data following the distribution of the training set [49]. It covers different domains, including generating images from text [122]. We do not use it, as this approach requires much training data and ends up generating non-real-world data that has similar distribution with the training set, whose limitations we discussed earlier.

5.3 Test output processing

Once all the test inputs are generated and executed, Keeper works on failure identification and attribution.

5.3.1 Failure identification

Keeper looks for three types of failure symptoms: (1) low accuracy, (2) dead code, and (3) generic failures like crashes.

Low-accuracy failures.

When software incorporates cognitive ML APIs in its computation, judging the output's correctness becomes challenging: (1) by definition of cognitive tasks, this output needs to be checked with many people to see if it matches with common human judgement; (2) due to the probabilistic nature of ML APIs, an occasional mismatch is expected. Of course, frequent mismatches are un-acceptable and severely hurt user experience.

To tackle the first challenge, Keeper uses pseudo-inverse functions as an approximation of common human judgement; to tackle the second challenge, Keeper considers the software to suffer from a low-accuracy failure, or an *accuracy failure* for short, only when over a threshold portion of inputs of a particular type have produced outputs that are inconsistent with common human judgement.

Specifically, for all the inputs $\mathbb{I}_b$ that are generated to cover a branch b, Keeper checks which of them exercise b at run time, denoted as $\mathbb{I}_b^{\text{succ}}$ and calculates the *recall* of b (i.e., $\frac{|\mathbb{I}_b^{\text{succ}}|}{|\mathbb{I}_b|}$). If the recall drops below a threshold α, 75% by default. Keeper reports an accuracy failure associated with b. The setting of α can be adjusted, but should not be 100%, as ML APIs are probabilisticand pseudo-inverse functions cannot guarantee to be correct all the time.

For a branch b that depends on the output of a sentiment-detection API, Keeper identifies failures slightly differently as inputs are generated for sentiment-detection API differently as discussed in Section 5.2.2. During test runs, Keeper checks all the inputs that exercise b to see what portion of them are labeled as having positive emotion and what portion are labeled as negative. If both go above a threshold, indicating that branch b is not accurately differentiating inputs with different emotions, Keeper reports an accuracy failure.

Root causes of accuracy failures. Note that, these accuracy failures are **not** equivalent with low precision or low recall of the ML API itself. The latter is just one of the possible root causes of the former. Keeper intentionally does not calculate the precision or recall of any ML API, but instead focuses on the overall software.

One possible cause is that developers missed some related labels in a branch condition, which we refer to as an *incomplete label* problem. For example, the `label_detection` API does not return "fire" as a top-3 label for many top fire images returned by the Bing image search. This by itself is *not* considered a failure by Keeper. If the software uses the API properly, like raising a fire alarm upon not only a "fire" label but also a "flame" label and an "ash" label, no accuracy failure would be reported, as the recall of the alarm-related branch is as high as 85% and the precision is 100% in our experiments.

Another possible cause is that developers used a non-existing label, which does not exist in the API's label set and can never be the output. This is not a surprise as the labels that can be output by Google Vision API are too many (19,985) for developers to memorize. For example, an application compares the `label_detection` output with "clothes" and "pants"

[55], which are non-existing labels. Instead, "clothing" and "trousers" are valid labels.

Dead-code failures

These occur when a branch is not covered after all the testing runs. They happen under two scenarios.

One scenario is that Keeper generates a set of test inputs $\mathbb{I}_b$ expected to cover a branch b, and yet b is not exercised by any input in $\mathbb{I}_b$. Such an extreme case of low branch recall (i.e., 0) is often caused by the branch comparing a ML API output with a non-existing label. If this comparison is one of multiple branch sub-conditions, an accuracy failure would likely occurr (i.e., a low but non-zero recall); if it is the only condition clause, a deadcode failure occurs. For example, a smart photo application FESMKMITL [37] checks the output of `label_detection` against the string "face". Unfortunately, among the 20,000 category labels that could be output by this API, none of them is "face". Instead, "human face" is one of the valid labels for this API, which the developers should have used.

The other scenario is that Keeper fails to generate any inputs to cover a branch, which triggers a dead-code failure report before any test runs. Sometimes, this is caused by a typo in the branch condition. For example, Keeper exposes such a failure in Verlan [143]. Verlan uses `object-detection` to judge whether an image contains an animal or not. Unfortunately, it wrongly uses `"animal"` instead of `obj.name == ‘‘animal’’` in its branch condition, making the if-statement always True. It will regard every image that contains at least one object as an animal image!

```
object = client.object_detection(image=img)
for obj in objects:
  if obj.name=="dog" or "animal":
    do_A()
```

Figure 5.3: Dead-code bugs in **Verlan** [143]

Generic failures

These have symptoms like crashes that do not require special techniques to observe. Comparing with traditional testing techniques, Keeper offers extra benefit two scenarios. (1) The failures are caused by bugs located on a path that requires specific ML API inputs to trigger. Keeper contributes by generating the needed ML API inputs to exercise the path. (2) The failures are directly related to the corner cases of ML API inputs, such as blank images that cause `label_detection` to return no labels. An example of such a bug exposed by Keeper is illustrated in Figure 5.4.

```
text = client.text_detection(image=img)
labels = text[0].description.split('\n')
for label in labels:
  do_something()
```

Figure 5.4: Crash failure in **FortniteKillfeed** [39]: a blank image returns an empty array `text` and trigger an index-out-of-range.

5.3.2 Failure attribution

To help developers understand and tackle accuracy failures, Keeper attempts to automatically patch the software by changing how ML APIs' output is used. Keeper suggests the change to developers and if all attempts failed, Keeper suggests developers to consider using a different, more accurate ML API, or adding extra input screening or pre-processing. Specifically, Keeper attempts two types of changes to the branch b where the failure is associated with.

Label changes. When branch b compares a ML API output with a set of labels, Keeper tries to expand the set of labels with three goals in mind. (1) Recall goal: more test inputs that are expected to exercise b can now satisfy b's condition; (2) Precision goal: most inputs that are not expected to exercise b should continue to fail the condition of b; (3) Semantic goal: the added labels are related to the original label(s) in b in terms of natural language semantics.

Without loss of generality, imagine that b takes the form of `if o == label0`, with `o` being the output of an ML API f. Keeper first collects the set of labels L output by f for every input in $\mathbb{I}_b^{\text{fail}}$, the set of inputs that are expected to exercise b but fail to do so.

Then, considering the semantic goal, Keeper filters out every label in L that is neither adjacent to nor sharing a common neighbor with `label0` in the wikidata knowledge graph [8].

Next, Keeper uses a greedy algorithm to iteratively expand the set of labels compared with `o` in b. Every time, Keeper adds to the set a label $\texttt{l} \in L$ so that `l` offers the biggest improvement in b's recall without reducing b's F1-score (i.e., the harmonic mean of the precision and the recall). Here, the precision of branch b is computed as $\frac{|\mathbb{I}_b^{succ}|}{|\mathbb{I}^{suc}|}$: among all the inputs that exercise b, how many of them are expected to do so. This procedure continues until the recall of b goes above the accuracy failure threshold or when there is no eligible candidate label remaining in L.

Threshold changes. As discussed earlier, an accuracy failure is reported when a branch b, which checks the `score` and/or `magnitude` output of a sentiment-detection API, gets exercised by many inputs labeled as having positive emotions and also many inputs labeled as having negative emotions. Keeper applies logistic regression to these input texts, with the {`score`, `magnitude`} output of each input as feature vectors and the labeled emotion as a class. Keeper then suggests the linear formula of logistic regression as a new branch checking threshold to developers, letting them know that this new formula can better differentiate text inputs with different emotions.

5.4 Implementation

We have implemented Keeper for Python applications that use Google Cloud AI APIs [47], the most popular cloud AI services on Github [146]. The core algorithm of Keeper is general to other languages and ML Cloud APIs. Keeper uses dynamic symbolic execution framework

PyExZ3 [66], which implements the DSE algorithm, and uses CVC4 [18] for constraint solving. Keeper uses Python built-in trace back tool [4] to check branch coverage, and Pyan [100] and Jedi [52] for call graph and program dependency analysis. Keeper uses Python scikit-learn[6] library for linear regression models.

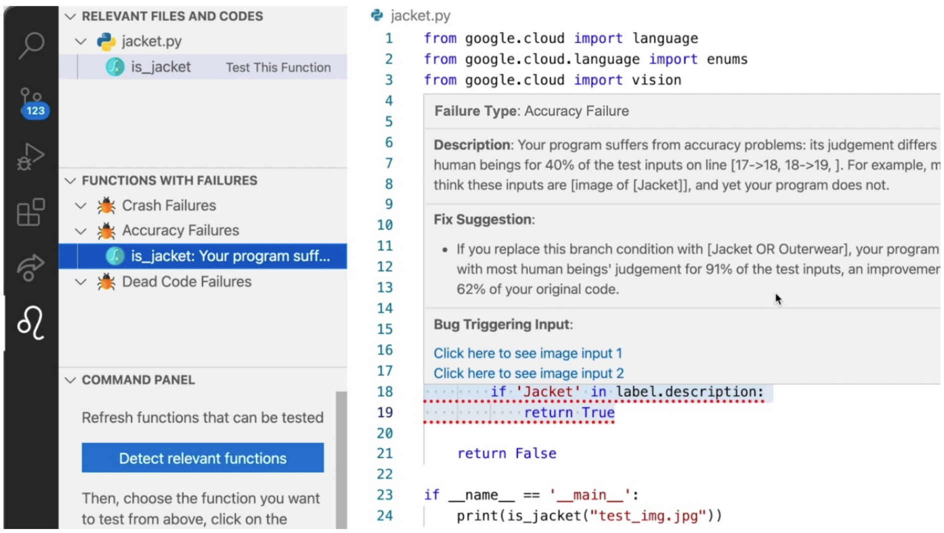


Figure 5.5: Keeper IDE plugin interface

5.5 Evaluation

Our evaluation aims to answer several questions:

1. Does Keeper help improve the branch coverage in testing?

2. Is Keeper able to find bugs during its testing?

3. Is Keeper able to suggest fixes for accuracy failures?

5.5.1 *Methodology*

Applications

We evaluate Keeper using 63 Python applications that are from two sources. 1) From the 360 open-source applications assembled by a previous study of ML APIs [146], we found 45 Python applications that use ML APIs in a non-trivial way (i.e., the API output affects control flow). 2) We additionally checked about 100 random Python applications on GitHub that use ML APIs and found 18 applications that use ML APIs in a non-trivial way.

These 63 applications use a range of ML APIs, including Vision (32 apps), Language (23 apps), and Speech (8). Their sizes range from 54 lines of code to more than 100,000 lines of code, with 582 lines of code being the median[3]. They have a median age of 18 months at the time of our study (*Apr. 1st, 2021*).

Despite our best effort in application collection, unfortunately, most of these 63 applications seem to be research projects, hackathon products, or demo programs, based on their limited popularity in Github. This is probably due to the young age of ML APIs. Consequently, our evaluation results may not generalize to mature software that has a solid user base.

For more than half of the applications (35), we simply specify `main` as the function to test. In other cases, the function under test is the entry function to the software feature related to ML APIs. The average number of branches in these functions-to-test is 13.

Baselines

We compare Keeper with 3 other techniques. Each technique generates 100 test inputs for each function under test.

(1) *Random Real*: we randomly pick inputs from well established data sets, including ImageNet [28] that contains 14 million images, Twitter US Airline Sentiment [72] that

3. Files from templates, frameworks, and libraries are not included in the LoC counting.

	Vision App.	Language App.	Speech App.
Keeper	91.9%	91.5%	89.7%
Random-Real	74.5%	85.0%	54.3%
Random-Real-Noise	73.0%	65.2%	54.3%
Fuzzing	44.4%	74.0%	24.9%

Table 5.2: Average branch coverage across 63 applications.

contains 15,000 tweets, and a set of audio clips synthesized for 115 daily sentences [1].

(2) *Random Real + Noise*: we add random noise to inputs picked by *Random Real*. For an image, we randomly added noises following Gaussian distribution; for an text input, we randomly decide whether to add noise and if so, randomly changed the word orders. For audio input, we do not add noise here, as we found that adding small noises does not affect ML API and yet adding big noises would turn the audio clip into what the third approach will generate.

(3) *Fuzzing*: we use a coverage-based fuzzing tool pythonfuzz [41] to generate images, text, and audio. For every image input, we use an integer list to fill its RGB matrix in a repeated way. For every text inputs, we generates ASCII character sequences. For audio inputs, we directly generates the audio data.

5.5.2 Software testing evaluation

Branch coverage

For each of the 63 functions specified to test, each from one application in our benchmark suite, we compute the accumulative branch coverage achieved by the 100 inputs generated by each testing technique. Table 5.2 shows the overall results.

Across different types of applications, Keeper consistently achieves high branch coverage, around 90% on average. The uncovered branches are either related to dead-code failures that Keeper discovers, or related to code that our underlying symbolic execution engine cannot handle. In comparison, the fuzzing technique performed the worst, covering less than 50% of

Failure type	Root Cause	Related ML Task	Keeper	RReal	RReal+Noise	Fuzz.
Crash failures	Out-of-bound accesses	Text detection, entity detection	6	5	5	4
	Missing input validation*	Document classification	1	-	-	-
	Missing type conversion	-	1	1	1	1
Accuracy failures	Improper labels	Image classi., object detect., document classi.	9	-	-	-
	API limitations	Image classification, object detection	6	-	-	-
	Improper threshold	Sentiment detection	9	-	-	-
Dead-code failures	Typos	Image classification, text detection	2	-	-	-
	Non-existing label	Image classification	1	-	-	-

Table 5.3: Unique failures exposed by Keeper. (*: This crash disappeared later with the most recent version of Google API.)

the branches for vision and speech applications, confirming our intuition that it is important to use realistic inputs to test ML APIs.

Random Real performs better than fuzzing, but still fails to cover about a quarter of branches in vision applications and half of the branches in speech applications. Adding random noises to random realistic inputs does not help. Keeper covers 23% and 59% more branches than Random-Real for vision and speech applications, respectively, as Keeper leverages symbolic execution and pseudo-inverse functions to generate inputs targeting different branches.

Applications that use language APIs appear to be the easiest to cover—even fuzzing achieves 74% coverage. This is probably because language APIs' output, like document type or entity name, has much less variation than that of vision and speech APIs.

As we can see, Keeper offers the highest branch coverage for all 63 applications.

Failure exposing and attribution

As shown in Table 5.3, Keeper exposed many failures by running those 100 test inputs it generated: 35 failures from the latest version of 25 applications. These failures cover a range of symptoms and root causes. Except for one failure caused by missing type conversion, the others are all related to different types of cognitive ML tasks, as shown in the table.

In comparison, alternative testing techniques missed 2–3 crash failures caught by Keeper. Furthermore, unlike Keeper, they cannot automatically recognize accuracy failures and dead-code failures.

Accuracy failures. Among the 24 accuracy failures exposed by Keeper, 15 of them are related to label checking for vision APIs and document-classification API, and 9 are related to threshold checking for the sentiment detection API.

For all of the 9 failures related to sentiment detection, Keeper manages to suggest better checking threshold that fixes the failure.

There are 9 accuracy failures that Keeper manages to fix by making the failure branch check for 1–3 extra labels. As an example, one application checks if the output of `label_detection` contains either "building" or "estate" or "mansion". This branch's recall is very low: 33%. Keeper suggests adding "house", "architecture", and "window" to the label set, which would improve the recall to be above 75%.

For the remaining 6 vision-related accuracy failures, code changes by Keeper can alleviate the problem but cannot push the recall of the related branch to be above 75%, suggesting fundamental API limitations. Two of these cases actually involve non-existing labels. For example, the "aluminum" in Heap-Sort-Cypher [55] is actually a non-existing label. Keeper suggests checking "metal" instead, which increases the branch's recall to close to 40%, but still below 75%.

Deadcode failures occurred in 3 applications. One of them is due to non-existing labels. Two are because of typos in branches that process ML API output, like the one in Figure 5.3.

Crash failures are mainly caused by out-of-bound accesses to lists returned by ML APIs, as shown in Figure 5.4. One crash is caused by buggy code inside a branch body that handles images with coins inside. This failure cannot be exposed by other testing techniques, as they did not produce images with coins inside.

False positives. Keeper has two false positives in total (they are not included in Table 5.3). One application tries to detect sensitive document by checking if any output of the document-classification API contains a “ensitive” sub-string. Keeper feeds its pseudo-inverse function with “ensitive” and fails to get any test inputs, and hence incorrectly reports a dead-code failure. The other application has a branch that gets covered only when an ML API generates a specific output with low confidence. Keeper is not effective at generating low-confidence inputs and wrongly reports an accuracy failure.

Threshold setting. As discussed in Section 5.3.1, the recall threshold α is set to 0.75 by default when detecting accuracy failures. Naturally, more failures would be reported when α is larger. Increasing α to 0.95, which is unreasonably high, would creates 5 more failure reports; decreasing α to 0.6 would have 2 fewer failure reports.

5.5.3 User studies

Study with users

To better evaluate the accuracy failures and the code changes suggested by Keeper, we recruited 100 participants on Amazon Mechanical Turk (Mturk) for a software-user survey. The survey includes 4 applications from our benchmark suites: 2 image-related applications and 2 text-related applications. On each survey page, a brief description is given for an application and user-study participants are told to review how two versions of this application perform on a set of inputs. Then, the web page displays a number of input images/text and the corresponding outputs of application version-1 and application version-2. These two versions are the original application and the application with suggested code changes from Keeper (referred to as *fixed* in Figure 5.6); we randomly decide which one of them is version-1 and which is version-2 on each survey page to reduce potential bias. Each participant is asked to answer questions about (1) for each input, which version's output they prefer; and (2) which version they think is better with everything considered. Participants were compensated $5 after the survey.

A summary of the user study results is shown in Figure 5.6. As we can see, in all cases,

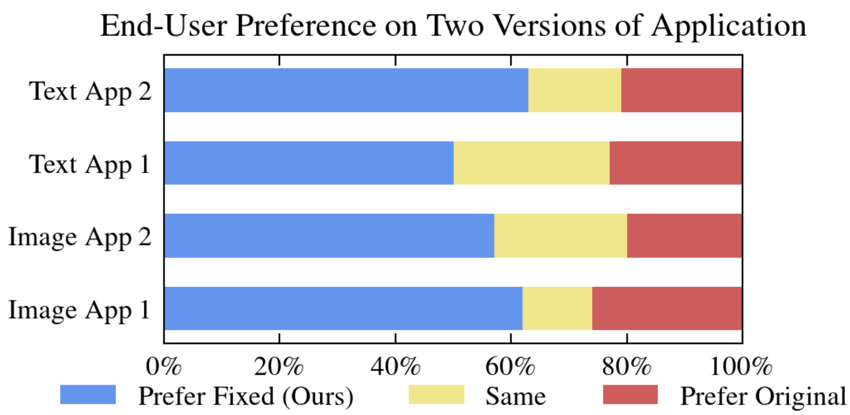


Figure 5.6: End-user preference: Original vs. Keeper version.

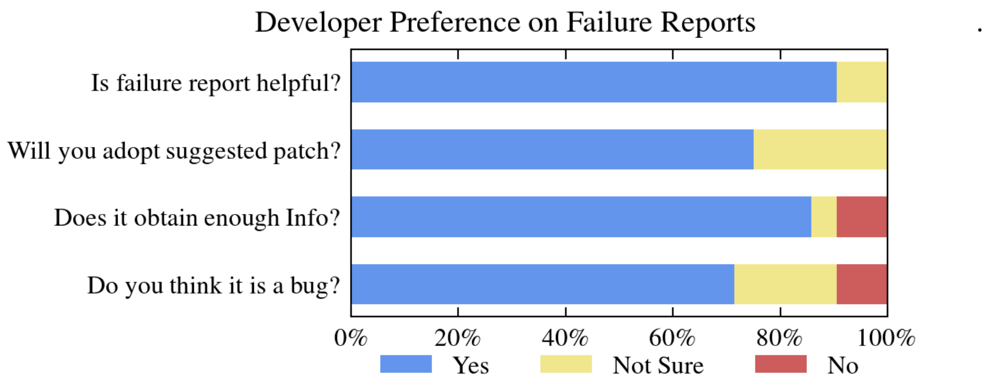


Figure 5.7: Developer preference of Keeper failure reports.

a dominate portion of end-users prefer the version with changes suggested by Keeper over the original version, supporting Keeper's judgement about accuracy failures and Keeper's attempt in fixing the accuracy problems. At the same time, we also noticed that there are 20–26% of user-study participants who prefer the original software and 12–27% who feel the two versions are about the same. These results confirm the fact that cognitive tasks are inherently subjective—even human beings often do not agree with each other on these tasks.

Question	Answer	#
Do you think Keeper is helpful?	Yes, it is useful.	9
	It would be helpful if becomes faster.	1
Do you like the interface of Keeper?	I like it.	6
	Need guidance to use/read it.	3
	Hope it could show more info.	1

Table 5.4: Developer overall preference of Keeper.

Study on developers

We recruited 10 participants who have Python programming experience. Half of them are software engineers from industry and half are college students. Given ML API official documents, they are asked to implement two functions, one for image analysis and one for text analysis, with the code skeleton provided by us. They then use Keeper to test their implementation. For any failures reported by Keeper, they are asked about whether they think the failure indeed reflects a software bug; how they would fix the code; whether they think Keeper is helpful; and others. This whole process is conducted through Zoom, with two researchers remotely interacting with the participant. Participants were compensated $10 after the interview. At the end of all interviews, three researchers coded the interview data independently and then met to resolve disagreements.

It took 20–60 minutes for the participants to read ML API documents and program; the whole session took 40–90 minutes.

In total Keeper reported 12 accuracy failures, 3 generic failure, and 6 dead-code failures for the 20 implemented functions from 10 participants. Keeper suggested code changes for all the 12 accuracy failures. Only one participant (P3) managed to program both functions correctly. The other 9 participants each has 1–4 failures exposed by Keeper from their code.

Figure 5.7 and Table 5.4 summarized the developer interview results. As we can see, almost all accuracy-failure reports (18 out of 21) are regarded as helpful. For most of the accuracy failure reports (9 out of 12), participants said they would definitely adopt the patch suggested by Keeper. Participants were not sure about the suggested patch in 3 reports, because they wanted to inspect Keeper-generated test cases before making decisions. Participants strongly agreed that most failure reports (15 out of 21) pointed out bugs in their programs, including 7 out of 12 accuracy failures. There were only two cases, both about accuracy failures, where participants felt the failures do not mean their code is buggy, although in both cases they were willing to adopt the code changes suggested by Keeper.

We believe this reflects the subjective nature of cognitive tasks.

Near the end of each user study session, we asked the participants "Do you think Keeper is helpful?". Overwhelmingly, they answered "Yes" (9 out of 10). They told us that "I don't know much about machine learning, but this tool helps a lot" (P1); "I like it tell me how accurate my code is." (P4); "It's cool. I have no idea how it finds these more optimized solutions." (P5); "Hope my team could adopt similar testing tool." (P7); "Showing failure cases help me to troubleshoot." (P8). Many of them like the user interface after learning how to use it with no help from us (6 out of 10). They told us that "The tool is intuitive. I like the little symbols." (P1); "The UI interface is quite clear to use. It is even better than some old industry products." (P3); "I like the sidebar display." (P7).

5.6 Threats to Validity

Internal threats to validity. Keeper assumes that search engines' top results are mostly consistent with human judgement, which could be incorrect. The failure identification and fixing attempts in Keeper are inherently probabilistic. The recall that Keeper calculated for each branch could vary depending on the test inputs. More test inputs would make the testing procedure more robust.

Some inputs generated by Keeper may not be the inputs that the software aims to handle, like the image being a photo taken indoor and yet the software meant to be used outdoor. When Keeper expands a branch's comparison label set, the increase of the recall sometimes comes with the decrease of the precision (i.e., more inputs not expected to exercise the branch does exercise). Although Keeper uses the F1-score to balance precision and recall, ultimately developers need to make the code change decision. We implemented Keeper IDE plug-in, aiming to help developers make informed decision about how their software uses ML APIs.

When an input expected by Keeper to cover a branch b fails to do so, this input may cover another branch b' whose body conducts the same computation as b. This would confuse Keeper's failure identification, although we have not observed such situations.

External threats to validity. Most of applications in our benchmark suite, including those used as examples in the paper, are research applications, hackathon projects, or demo programs. Consequently, observations and results obtained from them might not generalize to more widely used, real-world applications. Our tool is only tested with python applications using Google AI, not other ML Cloud API services.

5.7 Conclusion

This work present Keeper, an automated coverage-guided testing framework that helps developers to detect bugs and provide fixing suggestions for their software implementation. Keeper automatically generates test cases via a novel two-stage symbolic execution and Keeper-designed ML inverse functions. We evaluate Keeper with a variety of open-source machine learning applications and achieve high code coverage with a small set of test cases. It identifies bugs that leads to software crash, lower inference accuracy, or dead code.

CHAPTER 6

RELATED WORK

6.1 Anytime Neural Network

Adaptive Inference. One branch of investigation has focused on reducing inference time in a dynamic, input-dependent manner [38, 102, 125, 142, 151]. These *adaptive inference* methods skip execution of parts of a network, based on an estimate of relevance computed for each input; their goal is to minimize computation required for accurate prediction on a per-example basis. Here, the inference procedure changes dynamically in response a network's input data. However, these approaches do not provide any mechanism for responding to environmental conditions that might introduce transient resource constraints to the system.

Anytime Deep Networks. Anytime methods provide means of addressing such environmental variability. Specifically, they aim to introduce a degree of robustness to dynamic environmental effects, at the possible cost of moderately increased computation. For example, a recent anytime [150] develops a prediction pipeline specifically for stereo depth estimation, outputting images with increasing spatial resolution, an approach that may not generalize to other domains. Recent generic anytime approaches include several *cascade* designs [60, 61, 87, 138], which grow subnetworks by depth, and a recent proposal [89] that grows by width. Our anytime design [145] outperforms these approaches both conceptually and experimentally.

Multitask Training. Multitask training is a non-trivial problem. Previous work solve this problem by clustering methods [95, 107], separating general and task-specific features [154], training all tasks with the same base network and a few task-specific layers [83, 86], and building joint losses with adaptive weights [60, 77]. Some work also targeted changes to optimizers to improve multitask network training. This includes NormSGD [86], which computes a parameter gradient per task, in separate backpropagation passes. These gradient vectors are then normalized before summation, ensuring that each task exerts equal influence on network parameters at every training iteration. [77] takes another approach to dynamically balancing task influence, allowing some slack in relative task importance, provided it is justified by outsized gains in accuracy across the task spectrum as a whole. Our OSGD optimizer [145], like NormSGD, attempts to dynamically re-balance task interactions. However, our OSGD addresses a different kind of interaction, making it composable with most existing optimizers.

6.2 Resource Management System

Dynamic decision. Past resource management systems have used machine learning [16, 88, 114, 118, 132] or control theory [58, 65, 73, 74, 106, 127, 166] to make dynamic decisions and adapt to changing environments or application needs. Some also use Kalman filter because it has optimal error properties [65, 73, 74, 106]. There are two major differences between them and our ALERT: 1) prior approaches use the Kalman filter to estimate physical quantities such as CPU utilization [74] or job latency [65], while ALERT estimates a *virtual* quantity that is then used to update a large number of latency estimates. 2) while variance is naturally computed as part of the filter, ALERT actually uses it, in addition to the mean, to help produce estimates that better account for environment variability.

Approximate application. Past work designed resource managers explicitly to coordinate approximate applications with system resource usage [34, 58, 57, 75]. Although related, they manage applications *separately* from system resources, which is fundamentally different from our ALERT's holistic design. When an environmental change occurs, prior approaches first adjust the application and then the system serially (or vice versa) so that the change's effects on each can be established independently [57, 58]. That is, coordination is established by forcing one level to lag behind the other. In practice this design forces each level to keep its own independent model and delays response to environmental changes. In contrast, our ALERT's global slowdown factor allows it to easily model and update prediction about all application and system configurations simultaneously, leading to very fast response times.

Real-time guarantee. Some research supports hard real-time guarantees for DNNs [165], providing 100% timing guarantees while assuming that the DNN model gives the desired accuracy, the environment is completely predictable, and energy consumption is not a concern. ALERT provides slightly weaker timing guarantees, but manages accuracy and power goals. ALERT also provides more flexibility to adapt to unpredictable environments. Hard real-time systems would fail in the co-located scenario unless they explicitly account for all possible co-located applications at design time.

6.3 Machine Learning Software

ML-based software. Prior work looked at how to test specific software that contains ML components [69, 71, 70, 133]. Unfortunately, their solutions do not apply to general ML software. For example, one work trained a SVM classifier to judge the correctness of an image dilation program, leveraging the fact that the input image and the output image should contain the same objects [69]. To test a blood-vessel image categorizer, previous work [71] generates blood-vessel images with certain density, branches, and other features, and use these features to generate output ground truth. Previous work [133, 70] uses metamorphic approaches to test entity detection and image region growth programs. They require application-specific rules about inputs and outputs relationship (e.g., after we concatenate inputs of entity detection, the output becomes the concatenation of individual outputs [133]).

Some previous work studies the different phases and different developer roles in large-scale development and deployment of ML-based applications [13, 56, 80, 81]. These studies do not provide an automated testing technique.

Testing ML-based solutions. Some research studies common mistakes in programs that design and train neural networks [68, 161, 163, 164] or other types of machine learning models (e.g., SVM and decision tree) [137]. Some works focus on testing [113, 139, 152, 111, 98, 97, 22, 9, 96, 159, 32, 42, 15, 14, 153, 156, 53, 126, 43, 112, 162, 19, 35, 31, 90, 129] and fixing [67, 91, 135, 157] neural networks. All of these studies consider building machine learning models, instead of using them.

Testing ML APIs. Prior work studies automatic testing and bug detection of machine learning APIs, including machine learning frameworks for implementing neural networks [17, 23, 51, 108, 116, 131, 140] and REST APIs for providing machine learning solutions [46, 48, 115]. These works focuses on the implementation inside ML APIs, neglecting how they interact with other software components.

Testing FaaS APIs. Past works studied testing and fixing FaaS (Functions as a Service) platforms, in terms of accuracy [128, 144], performance [50, 78, 94, 99, 103], and security [36, 44, 79]. These works focusing on general FaaS APIs, but do not address the unique challenges raised by machine learning solution.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Contributions

In conclusion, this dissertation makes the following four contributions.

1. Propose a new design of flexible neural networks that support anytime prediction. Specifically, we propose a novel variant of SGD customized for training network architectures that support anytime behavior. Efficient architectural designs for these networks focus on re-using internal state; subnetworks must produce representations relevant for both immediate prediction as well as refinement by subsequent network stages. To train such network, we propose a new optimizer, *Orthogonalized SGD*, dynamically re-balances task-specific gradients when training a multitask network. In the context of anytime architectures, this optimizer projects gradients from later outputs onto a parameter subspace that does not interfere with those from earlier outputs.

2. Propose a run-time network scheduler called *ALERT*, that dynamically selects and adapts a DNN and a system-resource setting together to handle changing system environments and meet dynamic energy, latency, and accuracy requirements. Specifically, it uses a probabilistic model to detect environmental volatility and then simultaneously select both a DNN and a system resource configuration to meet requirements. We evaluate ALERT on CPU and GPU platforms for image and speech tasks in dynamic environments. ALERT meets constraints while achieving within 93–99% of optimal energy saving or accuracy optimization. Furthermore, it makes use of the flexibility provided by our anytime design to achieve better performance.

3. Conduct a thorough empirical study about Machine Learning cloud API misuses. We manually study 360 representative open-source applications that use Google or AWS

cloud-based ML APIs, and find 70% of these applications contain API misuses in their latest versions that degrade functional, performance, or economical quality of the software. We have generalized 8 anti-patterns based on our manual study and developed automated checkers that identify hundreds of more applications that contain ML API misuses.

4. Build a new testing tool *Keeper* for software that uses cognitive ML APIs. Keeper designs a set of pseudo-inverse functions for cognitive ML APIs and integrates them with symbolic execution to reach the sparse program-relevant input space. Keeper also makes them a proxy of human judgement and automatically judges the correctness of software outputs that are related to cognitive tasks. To help developers understand the root cause of an accuracy failure, Keeper explores alternative ways of using ML APIs and informs the developers of any code changes that can alleviate the accuracy failure. . Our evaluation on a variety of open-source applications shows that Keeper greatly improves the branch coverage, while identifying many previously unknown bugs.

7.2 Limitation and Future Work

My future research goal is to continue improving software systems with machine learning components. I believe achieving this requires inter-disciplinary solutions: machine learning, software engineering, and self-adaptive (or autonomic) software design.

Failure diagnoses and recovery of machine learning software My previous work focus on tackling bugs at development and testing phase. In software runtime, there also exists opportunity to diagnose failure and recover from it. I believe that machine learning software will greatly benefit from such process by (1) switching to an alternative solution; (2) configuration adjustment; (3) asking end-user for feedback; (4) logging failure cases for offline fixing. I plan to build a run-time diagnosis tool to help developers to integrate failure recovery in machine learning software.

Software-aware network design and adaptation Software context greatly affects the expectation of neural network capability. Taking object detection as an example, a recipe recommendation application expects the network to precisely detect the ingredients in user input image, while a smart door application only needs to know whether there is human in the camera video. Such difference also appears in different iterations of software development, as the requirement might change. Therefore, it is important to efficiently design/adapt a neural network to a new software context. I believe there is still a long way to go to achieve awareness between software and network. I will first conduct an empirical study to investigate the problem in real-world application. Then I'll design a flexible network that could be easily adapted to the software context.

Composing multiple ML component My past research has focused on managing single neural network and software module. In many large machine learning software system, multiple neural networks cooperate with each other. If two neural networks have control or data dependency, then local adaptation decisions clearly affect the global program outcome. It also brings up new challenges for software testing and maintenance. I plan to build a tool to systematically help users to compose multiple ML component.

Testing machine learning system with hardware-software cooperation Machine learning techniques have been widely adopted on many problem domains. Some of them require hardware-software cooperation, e.g. robot waiter, auto-driving. Testing such systems brings up unique challenges: replaying the failure, locating the bug, patching the software, and etc. I will employ my experience in detecting bugs in ML software to improve the correctness of machine learning system.

REFERENCES

[1] 100 english daily sentences for daily use. https://englishspeakingcourse.net/100-english-sentences-for-daily-use/.

[2] Encyclopedia britannica. https://www.britannica.com/.

[3] Pillow: Python imaging library. https://pypi.org/project/Pillow/.

[4] Python system-specific parameters and functions. https://docs.python.org/3/library/sys.html#sys.settrace.

[5] pyttsx3: Text-to-speech library for python. https://pypi.org/project/pyttsx3/.

[6] scikit-learn: Machine learning in python. https://scikit-learn.org/stable/.

[7] Wikipedia. https://en.m.wikipedia.org/.

[8] Wikipedia. https://www.wikidata.org/.

[9] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *ICSE*, 2018.

[10] Baidu AI. Apollo open vehicle certificate platform. Online document, http://apollo.auto, 2018.

[11] S. Akhlaghi, N. Zhou, and Z. Huang. Adaptive adjustment of noise covariance in kalman filter for dynamic state estimation. In *IEEE Power Energy Society General Meeting*, 2017.

[12] Amazon. Amazon artificial intelligence service. Online document https://aws.amazon.com/machine-learning/ai-services, 2020.

[13] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *ICSE-SEIP*, pages 291–300. IEEE, 2019.

[14] Saleema Amershi, Max Chickering, Steven M Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. Modeltracker: Redesigning performance analysis tools for machine learning. In *CHI*, 2015.

[15] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Themis: Automatically testing software for discrimination. In *ESEC/FSE*, 2018.

[16] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *CASES*, 2012.

[17] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*, 2015.

[18] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[19] David Berend, Xiaofei Xie, Lei Ma, Lingjun Zhou, Yang Liu, Chi Xu, and Jianjun Zhao. Cats are not fish: Deep learning testing calls for out-of-distribution awareness. In *FSE*, 2020.

[20] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, October 2008.

[21] Microsoft Bing. Bing image search. `https://www.bing.com/images/trending?FORM=ILPTRD`.

[22] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Autofocus: interpreting attention-based neural networks by code perturbation. In *ASE*, 2019.

[23] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[24] Giorgio C Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Springer, 2006.

[25] Dave Chaffey. Search engine marketing statistics 2020. `https://www.smartinsights.com/search-engine-marketing/search-engine-statistics/`.

[26] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[27] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *ISLPED*, 2010.

[28] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

[30] Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: An evaluation of the state of the art. *TPAMI*, 2011.

[31] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. Detecting flaky tests in probabilistic and machine learning applications. In *ISSTA*, 2020.

[32] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA*, 2018.

[33] emotion2music. A smart music player application. `https://github.com/varnachandar/emotion2music`.

[34] Anne Farrell and Henry Hoffmann. MEANTIME: achieving both minimal energy and timeliness with approximate computing. In *USENIX ATC*, 2016.

[35] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *ISSTA*, 2020.

[36] Md Sadek Ferdous, Andrea Margheri, Federica Paci, Mu Yang, and Vladimiro Sassone. Decentralised runtime monitoring for access control systems in cloud federations. In *ICDCS*, 2017.

[37] FESMKMITL. A smart camera application. `https://github.com/matthewjmc/FESMKMITL`.

[38] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry P Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. In *CVPR*, 2017.

[39] FortniteKillfeed. A real time tracker application. `https://github.com/Godsinred/FortniteKillfeed`.

[40] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale DNN processor for real-time AI. In *ISCA*. IEEE Press, 2018.

[41] Fuzzit.dev. Pythonfuzz: coverage-guided fuzz testing for python. `https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/pythonfuzz`.

[42] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: testing software for discrimination. In *FSE*, 2017.

[43] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. Importance-driven deep learning system testing. In *ICSE*, 2020.

[44] Anders Tungeland Gjerdrum, Håvard Dagenborg Johansen, Lars Brenna, and Dag Johansen. Diggi: A secure framework for hosting native cloud functions with minimal trust. In *TPS-ISA*, 2019.

[45] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *CS224N project report, Stanford*, 1(12):2009, 2009.

[46] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. Differential regression testing for rest apis. In *ISSTA*, 2020.

[47] Google. Google cloud ai. Online document `https://cloud.google.com/products/ai`, 2020.

[48] Eric Gossett, Cormac Toher, Corey Oses, Olexandr Isayev, Fleur Legrain, Frisco Rose, Eva Zurek, Jesús Carrete, Natalio Mingo, Alexander Tropsha, et al. Aflow-ml: A restful api for machine-learning predictions of materials properties. *Computational Materials Science*, 2018.

[49] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye. A review on generative adversarial networks: Algorithms, theory, and applications. *arXiv preprint arXiv:2001.06937*, 2020.

[50] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *CLOUD*, 2019.

[51] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Li Xiaohong, and Chao Shen. Audee: Automated testing for deep learning frameworks. In *FSE*, 2020.

[52] Dave Halter. Jedi: an awesome auto-completion, static analysis and refactoring library for python. Online document `https://jedi.readthedocs.io`.

[53] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *ESEC/FSE*, 2020.

[54] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[55] HeapSortCypher. A garbage classification application. `https://github.com/matthew-chu/heapsortcypher`.

[56] Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. Trials and tribulations of developers of intelligent systems: A field study. In *VL/HCC*, 2016.

[57] Henry Hoffmann. Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *ECRTS*, pages 223–232, 2014.

[58] Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *SOSP*, 2015.

[59] Henry Hoffmann and Martina Maggio. PCP: A generalized approach to optimizing performance under power constraints through resource management. In *ICAC*, 2014.

[60] Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J Andrew Bagnell. Learning anytime predictions in neural networks via adaptive loss balancing. In *AAAI*, 2019.

[61] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. In *CoRR*, 2017.

[62] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[63] IBM. Ibm watson. Online document `https://www.ibm.com/watson`, 2020.

[64] C. Imes and H. Hoffmann. Bard: A unified framework for managing soft timing and power constraints. In *SAMOS*, pages 31–38, 2016.

[65] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *RTAS*, pages 75–86, April 2015.

[66] M Irlbeck et al. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering*, 40:26, 2015.

[67] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. Repairing deep neural networks: Fix patterns and challenges. In *ICSE*, 2020.

[68] Gunel Jahangirova, Nargiz Humbatova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *ICSE*, 2020.

[69] Tahir Jameel, Lin Mengxiang, and Liu Chao. Automatic test oracle for image processing applications using support vector machines. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 1110–1113. IEEE, 2015.

[70] C. Jiang, S. Huang, and Z. Hui. Metamorphic testing of image region growth programs in image processing applications. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 70–72, 2018.

[71] Misael C Júnior, Rafael AP Oliveira, Miguel AG Valverde, Marcel P Jackowski, Fátima LS Nunes, and Márcio E Delamaro. Feature-based test oracles to categorize synthetic 3d and 2d images of blood vessels. In *Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing*, pages 1–6, 2017.

[72] Kaggle. Twitter us airline sentiment. `https://www.kaggle.com/crowdflower/twitter-airline-sentiment`.

[73] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC*, 2009.

[74] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Adaptive resource provisioning for virtualized servers using kalman filters. *TAAS*, 2014.

[75] Aman Kansal, Scott Saponas, AJ Brush, Kathryn S McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (lab) abstraction: programmer

productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA*, 2013.

[76] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *ICCPS*, pages 287–296, 2018.

[77] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *CVPR*, 2018.

[78] Joanna Kijak, Piotr Martyna, Maciej Pawlik, Bartosz Balis, and Maciej Malawski. Challenges for scheduling scientific workflows on cloud functions. In *CLOUD*, 2018.

[79] Jeongchul Kim, Jungae Park, and Kyungyong Lee. Network resource isolation in serverless cloud function service. In *FAS* W*, 2019.

[80] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. In *ICSE*, 2016.

[81] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. Data scientists in software teams: State of the art and challenges. *TSE*, 2017.

[82] Klassroom. A lecture note application. `https://github.com/dev5151/Klassroom`.

[83] Iasonas Kokkinos. UberNet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory. In *CVPR*, 2017.

[84] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2012.

[85] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, pages 1–26, 2020.

[86] Gustav Larsson. Discovery of visual semantics by unsupervised and self-supervised representation learning. *arXiv:1708.05812*, 2017.

[87] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. FractalNet: Ultra-deep neural networks without residuals. In *ICLR*, 2017.

[88] Benjamin C Lee and David Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. *ASPLOS*, 2008.

[89] Hankook Lee and Jinwoo Shin. Anytime neural prediction via slicing networks vertically. *arXiv:1807.02609*, 2018.

[90] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *ISSTA*, 2020.

[91] Zenan Li, Xiaoxing Ma, Chang Xu, Jingwei Xu, Chun Cao, and Jian Lü. Operational calibration: Debugging confidence errors for dnns in the field. In *ESEC/FSE*, 2020.

[92] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *ASPLOS*, 2018.

[93] Jun S Liu and Rong Chen. Sequential monte carlo methods for dynamic systems. *Journal of the American statistical association*, 1998.

[94] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara.

Serverless computing: An investigation of factors influencing microservice performance. In *IC2E*, 2018.

[95] Mingsheng Long and Jianmin Wang. Learning multiple tasks with deep relationship networks. *arXiv:1506.02117*, 2, 2015.

[96] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *ISSRE*, 2018.

[97] Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. Lamp: data provenance for graph based machine learning algorithms through derivative computation. In *FSE*, 2017.

[98] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. Mode: automated neural network model debugging via state differential analysis and input selection. In *ESEC/FSE*, 2018.

[99] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *UCC Companion*, 2018.

[100] David Marby and Nijiko Yonskai. Pyan3: Offline call graph generator for python 3. `https://github.com/davidfraser/pyan`.

[101] John D McCalpin. Memory bandwidth and machine balance in current high performance computers. *TCCA*, 1995.

[102] Mason McGill and Pietro Perona. Deciding how to decide: Dynamic routing in artificial neural networks. In *ICML*, 2017.

[103] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *ICDCSW*, 2017.

[104] Microsoft. Microsoft azure cognitive services. Online document https://azure.microsoft.com/en-us/services/cognitive-services, 2020.

[105] Microsoft. Visual studio code. Online document https://code.visualstudio.com/, 2021.

[106] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. CALOREE: learning control for predictable latency and low energy. In *ASPLOS*, 2018.

[107] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *CVPR*, 2016.

[108] Mahdi Nejadgholi and Jinqiu Yang. A study of oracle approximations in testing deep learning libraries. In *ASE*, 2019.

[109] noteScript. A lecture note application. https://github.com/GalenWong/noteScript.

[110] NsTool. A monitor application. https://github.com/clarkwkw/ns_online_toolkit.

[111] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *ICML*, 2019.

[112] Brandon Paulsen, Jingbo Wang, and Chao Wang. Reludiff: Differential verification of deep neural networks. In *ICSE*, 2020.

[113] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *ASPLOS*, 2017.

[114] Paula Petrica, Adam M Izraelevitz, David H Albonesi, and Christine A Shoemaker. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *ISCA*, 2013.

[115] Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Are rest apis for cloud computing well-designed? an exploratory study. In *ICSOC*, pages 157–170. Springer, 2016.

[116] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *ICSE*, 2019.

[117] Phoenix. A fire-detection application. `https://github.com/yunusemreemik/Phoenix`.

[118] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *MICRO*, 2001.

[119] PyGithub. Pygithub: Typed interactions with the github api v3. `https://pygithub.readthedocs.io/en/latest/introduction.html`.

[120] Python. ast — abstract syntax trees. `https://docs.python.org/3/library/ast.html`.

[121] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[122] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

[123] recipeGo. A recipe recommendation application. `https://github.com/Reckonzz/recipeGO`.

[124] S. Reda, R. Cochran, and A. K. Coskun. Adaptive power capping for servers with multithreaded workloads. *MICRO*, 2012.

[125] Mengye Ren, Andrei Pokrovsky, Bin Yang, and Raquel Urtasun. SBNet: Sparse blocks network for fast inference. In *CVPR*, 2018.

[126] Vincenzo Riccio and Paolo Tonella. Model-based exploration of the frontier of behaviours for deep learning system testing. In *ESEC/FSE*, 2020.

[127] Muhammad Husni Santriaji and Henry Hoffmann. Grape: Minimizing energy for gpu applications with performance requirements. In *MICRO*, 2016.

[128] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2017.

[129] Arnab Sharma and Heike Wehrheim. Higher income, larger loan? monotonicity testing of machine learning models. In *ISSTA*, 2020.

[130] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[131] Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. Mli: An api for distributed machine learning. In *ICDM*, 2013.

[132] Srinath Sridharan, Gagan Gupta, and Gurindar S Sohi. Holistic run-time parallelism management for time and energy efficiency. In *ICS*, 2013.

[133] Madhusudan Srinivasan, Morteza Pourreza Shahri, Indika Kahanda, and Upulee Kanewala. Quality assurance of bioinformatics software: a case study of testing a

biomedical text processing tool using metamorphic testing. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, pages 26–33, 2018.

[134] stockmine. A stock prediction application. `https://github.com/nicholasadamou/stockmine`.

[135] Zeyu Sun, Jie M Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. Automatic testing and improvement of machine translation. In *ICSE*, 2020.

[136] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[137] Yida Tao, Shan Tang, Yepang Liu, Zhiwu Xu, and Shengchao Qin. How do api selections affect the runtime performance of data analytics tasks? In *ASE*, 2019.

[138] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. BranchyNet: Fast inference via early exiting from deep neural networks. In *ICPR*, 2016.

[139] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *ICSE*, 2018.

[140] Saeid Tizpaz-Niari, Pavol Cernỳ, and Ashutosh Trivedi. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *ISSTA*, 2020.

[141] TRANSLATOR. A smart light application. `https://github.com/mubeenafatima/TRANSLATOR`.

[142] Andreas Veit and Serge Belongie. Convolutional networks with adaptive inference graphs. In *ECCV*, 2018.

[143] Verlan. A pet application. `https://github.com/sarvesh-tech/Verlan`.

[144] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Resttestgen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152. IEEE, 2020.

[145] Chengcheng Wan, Henry Hoffmann, Shan Lu, and Michael Maire. Orthogonalized sgd and nested architectures for anytime neural networks. In *ICML*, 2020.

[146] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Are machine learning cloud apis used correctly? In *ICSE*, 2021.

[147] Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Automated testing of software that uses machine learning apis. In *44th International Conference on Software Engineering (ICSE'22)*, 2022.

[148] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. {ALERT}: Accurate learning for energy and timeliness. In *USENIX ATC*, 2020.

[149] WanderStub. An exchange conversion application. `https://github.com/richardjpark26/WanderStub`.

[150] Yan Wang, Zihang Lai, Gao Huang, Brian H Wang, Laurens van der Maaten, Mark Campbell, and Kilian Q Weinberger. Anytime stereo image depth estimation on mobile devices. In *ICRA*, 2019.

[151] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. BlockDrop: Dynamic inference paths in residual networks. In *CVPR*, 2018.

[152] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In *ISSTA*, pages 146–157, 2019.

[153] Shenao Yan, Guanhong Tao, Xuwei Liu, Juan Zhai, Shiqing Ma, Lei Xu, and Xiangyu Zhang. Correlations between deep neural network model coverage criteria and model quality. In *ESEC/FSE*, 2020.

[154] Yongxin Yang and Timothy Hospedales. Deep multi-task representation learning: A tensor factorisation approach. In *ICLR*, 2017.

[155] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*, 2016.

[156] Fuyuan Zhang, Sankalan Pal Chowdhury, and Maria Christakis. Deepsearch: A simple and effective blackbox attack for deep neural networks. In *ESEC/FSE*, 2020.

[157] Hao Zhang and WK Chan. Apricot: a weight-adaptation approach to fixing deep learning models. In *ASE*, 2019.

[158] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, 2016.

[159] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*, 2018.

[160] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *ATC*, pages 951–965, 2018.

[161] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An empirical study on program failures of deep learning jobs. In *ICSE*, 2020.

[162] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In *ICSE*, 2020.

[163] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *ISSTA*, pages 129–140, 2018.

[164] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. Detecting numerical bugs in neural network architectures. In *ESEC/FSE*, 2020.

[165] H. Zhou, S. Bateni, and C. Liu. S3dnn: Supervised streaming and scheduling for gpu-accelerated real-time DNN workloads. In *RTAS*, 2018.

[166] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. Cash: Supporting iaas customers with a sub-core configurable architecture. In *ISCA*, 2016.

[167] Ligeng Zhu, Ruizhi Deng, Michael Maire, Zhiwei Deng, Greg Mori, and Ping Tan. Sparsely aggregated convolutional networks. In *ECCV*, 2018.

[168] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.