THE UNIVERSITY OF CHICAGO

ALERT: ACCURATE ANYTIME LEARNING FOR ENERGY AND TIMELINESS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER DEGREE

DEPARTMENT OF COMPUTER SCIENCE

BY
CHENGCHENG WAN

CHICAGO, ILLINOIS
SEPTEMBER 27, 2019
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. v  
LIST OF TABLES ................................................................. vi  
ACKNOWLEDGMENTS ............................................................... vii  
ABSTRACT ......................................................................... viii  

1 INTRODUCTION ................................................................. 1  
  1.1 Motivation ................................................................. 1  
  1.2Contributions .............................................................. 3  

2 BACKGROUND ................................................................. 5  

3 UNDERSTANDING DNN DEPLOYMENT CHALLENGES ......................... 6  
  3.1 Understanding Inference Latency ......................................... 6  
  3.2 Understanding Tradeoffs Across Networks ............................... 9  

4 ALERT RUN-TIME INFERENC E MANAGEMENT .................................. 11  
  4.1 Key ideas .................................................................. 12  
  4.2 Algorithm .................................................................. 15  
    4.2.1 Overview ............................................................ 15  
    4.2.2 Global Slow-down Factor ........................................ 15  
    4.2.3 Accuracy ............................................................. 16  
    4.2.4 Energy ................................................................. 17  
    4.2.5 Limitations of ALERT ............................................ 17  
  4.3 Implementation ............................................................. 18  

5 ALERT ANYTIME DNN DESIGN ............................................... 20  
  5.1 Design Principles ......................................................... 20  
  5.2 Proposed Nested Anytime DNN .......................................... 22  
    5.2.1 Width Nesting ....................................................... 23  
    5.2.2 Depth Nesting ...................................................... 24  
  5.3 Training and Implementation ............................................ 26  

6 EXPERIMENTS ................................................................. 28  
  6.1 ALERT runtime management ............................................ 28  
    6.1.1 Methodology ......................................................... 28  
    6.1.2 Results ............................................................... 28  
  6.2 Anytime-DNN design ..................................................... 34  
    6.2.1 Methodology ......................................................... 34  
    6.2.2 Results ............................................................... 34
# LIST OF FIGURES

1.1 ALERT inference system ......................................................... 3

2.1 A convolutional neural network. .................................................. 5

3.1 Latency variance across inputs for different tasks and hardware (Most tasks have 5 boxplots for 5 hardware platforms, Core i5, Razer, Xeon, P100, K80 from left to right; NLP1 has an extra boxplot for Odroid platform; other three tasks ran out of memory on Odroid; every box shows the 25th–75th percentile; points beyond the whiskers are >90th or <10th percentile). 8

3.2 Latency variance under memory contention ................................. 8

3.3 Accuracy and latency tradeoffs for 42 DNNs on Xeon. .................. 9

5.1 Ensemble on 3 different networks (each in a row). ....................... 21

5.2 Cascade with 3 outputs. ......................................................... 22

5.3 Width-wise nesting. ............................................................... 24

5.4 Depth-wise nesting. ............................................................... 25

6.1 ALERT versus Oracle and Oracle\textsubscript{Static} on minimize energy task (Lower is better). (whisker: whole range; circle: mean) .................................. 31

6.2 ALERT versus Oracle and Oracle\textsubscript{Static} on maximize accuracy task (Lower is better). (whisker: whole range; circle: mean) ............................... 32

6.3 Maximize accuracy for image classific. on Razer. (ALERT in blue; ALERT\textsubscript{Trad} in orange; constraints in red. Memory contention occurs from about input 46 to 119; Deadline: 1.25× mean latency of largest Anytime DNN in Default; power limit: 35W.) .................................................. 33

6.4 Accuracy-latency trade-off (lower is better; all results from the Xeon machine, #5 in Table 3.1) ..................................................... 35
LIST OF TABLES

3.1 Hardware platforms used in our experiments ........................................ 6
3.2 ML tasks and benchmark datasets in our experiments ........................ 6

6.1 Settings, tasks, and schemes under evaluation (* measured under default setting without resource contention) .................................................. 29
6.2 Normalized average energy consumption and error rate to $Oracle_{Static}$ (Smaller is better) Superscript means how many settings violate constraints for more than 10% cases. We report the average normalized energy only for non-violating settings. ........................ 30
ACKNOWLEDGMENTS

This project is a teamwork with Muhammad Husni Santriaji advised by Professor Shan Lu, Professor Henry Hoffmann, and Professor Michael Maire. It could not be done without those brilliant people. Chengcheng Wan (master degree candidate) studies DNN performance, developed the anytime DNN, and contributes to the run-time inference management in this project.
ABSTRACT

An increasing number of software applications incorporate runtime Deep Neural Network (DNN) inference for its great accuracy in many problem domains. While much prior work has separately tackled the problems of improving DNN-inference accuracy and improving DNN-inference efficiency, an important problem is under-explored: disciplined methods for dynamically managing application-specific latency, accuracy, and energy tradeoffs and constraints at run time.

To address this need, we propose ALERT, a co-designed combination of runtime system and DNN nesting technique. The runtime takes latency, accuracy, and energy constraints, and uses dynamic feedback to predict the best DNN-model and system power-limit setting. The DNN nesting creates a type of flexible network that efficiently delivers a series of results with increasing accuracy as time goes on. These two parts well complement each other: the runtime is aware of the tradeoffs of different DNN settings, and the nested DNNs’ flexibility allows the runtime prediction to satisfy application requirements even in unpredictable, changing environments. On real systems for both image and speech, ALERT achieves close-to-optimal results. Comparing with the optimal static DNN-model and power-limit setting, which is impractical to predict, ALERT achieves a harmonic mean 33% of energy savings while satisfying accuracy constraints, and reduces image-classification error rate by 58% and sentence-prediction perplexity by 52% while satisfying energy constraints.
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, or both. Although recent work in DNN design and system support improved not only DNN accuracy [17, 33, 42], but also model efficiency [5, 40, 103], performance, and energy usage [12, 13, 19, 27], successful deployment of DNN inference in the field, however, requires meeting a variety of user-defined, application-specific performance goals for latency, accuracy, and possibly, energy on a variety of hardware platforms in sometimes unpredictable dynamic environments, a problem that little work has explored.

Latency constraints naturally arise with DNN deployments when inference interacts with the real world as a consumer or producer: processing latency-sensitive data from a sensor or returning a timely answer to a human. For example, in simultaneous interpretation, translation must be provided every 2–4 seconds [65]; in motion tracking, a frame must be processed at camera speed. Violating these deadlines may lead to severe consequences: if a self-driving vehicle cannot act within a small time budget, life threatening accidents could follow [62] and machine-learning service providers, like Google, have strict DNN-inference constraints [49].

Requirements on accuracy and energy are also common and may vary for different applications in different operating environments. For example, in both mobile systems and user-facing cloud systems, DNN-based applications may have inference accuracy requirements for correct functionality. While ensuring the accuracy requirement, it would be highly beneficial to minimize energy or resource usage to extend mobile-battery time or reduce server-operation cost [49, 62]. In extremely resource-limited systems, like some embedded and intermittent computing systems [23], applications might be constrained in energy and
want to return the most accurate results given the energy constraint.

Previous research has produced huge tradeoff spaces of DNN models and system-resource settings, where different selections can be used to meet different requirements. Along the DNN dimension, people have found DNN design particularly well suited to approximate computing [7, 38, 81], where small sacrifices in accuracy can yield big gains in speed [13, 27, 34]. Along the system-resource dimension, DNN latency can be reduced by bringing more powerful hardware [101]—or even the same hardware operated at higher frequency—to bear, sacrificing energy to reduce latency without affecting accuracy.

The unexplored challenge is then to find a disciplined way to make the right DNN and system-resource selections to satisfy different requirements and goals for different users and applications. Note that, while it is possible to make the selection statically based on profiling results, real DNN deployments are further complicated by dynamic environmental variation:

- **Input Variety**: Although DNN computation is quite regular, time variation among different inputs still exists. For example, our results show that the 75th-percentile latency is about 1.4x the median latency for some NLP tasks, even with a fixed DNN on dedicated hardware.

- **Contention Variety**: An inference job may compete for resources against unpredictable, co-located jobs.

- **Requirement Variety**: Job latency could vary dynamically depending on how much time related jobs have consumed before it. Furthermore, other requirements could change, for example an application might have power requirements until it detects a specific event, when it switches to a strict accuracy requirement to correctly capture that event.

Thus, it would be ideal for systems to support inference by dynamically choosing the DNN model and resource settings to meet user-defined latency, accuracy, and energy goals.
1.2 Contributions

This paper makes several related contributions. We first quantify the degree to which DNN inference is affected by environmental variability. We then develop ALERT, a complementary runtime system and flexible DNN design. The runtime system leverages the DNN design to meet user goals by adapting to overcome the challenges of environmental changes.

Understanding the challenges We measure DNN performance for a range of requirements across applications, inputs, hardware, and resource contention and observe high variation in inference time. We compare 42 existing DNN models and find they offer a wide spectrum of latency, energy, and accuracy. Although some models are more efficient than others (i.e., producing similar accuracy with less execution time), the overall trend is that higher accuracy comes at the cost of longer latency and/or higher energy consumption (Section 3).

Run-time inference controller We design an inference controller that uses dynamic feedback and prediction to select a DNN model and system resource setting that together provide a disciplined balance between energy consumption, inference latency, and accuracy. The controller predicts the distribution of possible behavior and can make aggressive choices when the environment is quiescent or conservative choices when it is volatile. The runtime

Figure 1.1: ALERT inference system
meets constraints in any two dimensions while optimizing the third, for example minimizing energy given accuracy and latency requirements or maximizing accuracy given latency and energy budgets (Section 4).

**Controller-friendly DNN design** Although the runtime works with any set of candidate DNN models with latency/energy/accuracy tradeoffs, we further design a special type of *Anytime* DNN that provides additional flexibility to handle unpredictable system dynamics.

Our Anytime-DNN design employs novel width- and depth-nesting to create a DNN family. Instead of producing one result, the nested family produces a series of results with non-decreasing accuracy and little additional overhead. While a traditional DNN produces no result if it misses a deadline, our Anytime-DNN produces slightly degraded inference results if execution is unexpectedly slow or the deadline comes early, giving the run-time more flexibility to reduce resources without violating constraints (Section 5).

We test ALERT using various types of DNN models (CNN, RNN, and self-attention) and application domains (image classification, recommendation systems, and NLP) on different machines and various constraints. Our evaluation shows that our proposed Anytime-DNN allows existing state-of-the-art approaches to provide Anytime inference with negligible cost. Across various experimental settings, ALERT runtime meets constraints while achieving within 93–99% of optimal energy saving or accuracy optimization. Compared to an impractical scheme that uses perfect latency and energy predication to select static optimal DNN and power settings, ALERT saves 33% energy while satisfying latency and accuracy constraints, or reduces the image-classification error rate by 58% and the perplexity of sentence prediction by 52% while satisfying latency and energy constraints (Section 6).
CHAPTER 2
BACKGROUND

Neural networks are a popular class of machine-learning algorithms for analyzing complex
data. Traditional neural networks, following inspiration from biological systems [39], are
composed of layers of neurons; each neuron feeds a weighted sum of its inputs to a non-
linear activation function to produce an output. Neural networks with more than one layer
are called deep neural networks (DNNs).

There is a huge design space for DNNs. Many effective DNNs have been designed with
different depths (i.e., the number of layers), widths (i.e., the number of neurons in one layer),
layer types (i.e., the type of computation conducted at neurons in a layer, such as matrix
multiplication, convolution, and down/up sampling), and connections (i.e., how one layer
takes inputs from which neurons and layers).

Figure 2.1 shows an example of typical image classification network. The blue and orange
layer represents two different layer types: convolutional layers and fully-connected layers (i.e.,
matrix multiplication). The red layer produces the final output predictions, and hence con-
tains as many neurons as the number of image categories. For simplicity, this diagram hides
the details of spatial resolution. For comparison with later diagrams, this figure partitions
every layer of the network into three rectangle blocks, with each enclosing multiple neurons,
and shows arrows connecting all inputs to all outputs.

![Figure 2.1: A convolutional neural network.](image-url)
CHAPTER 3
UNDERSTANDING DNN DEPLOYMENT CHALLENGES

To better understand the challenges of DNN inference in latency critical environments, we profile two canonical machine learning tasks on a set of platforms. The two tasks, image classification and text translation (NLP), could both be deployed with deadlines; e.g., for motion tracking and simultaneous interpretation, and both have received wide attention, primarily in pursuit of accuracy, leading to a diverse set of available models. We use the state-of-the-art networks and common datasets (Table 3.2). The platforms cover representative embedded systems (ODroid), laptops (Core i5, Razer), CPU servers (Xeon), and GPU servers (P100, K80), as shown in Table 3.1.

<table>
<thead>
<tr>
<th>ID</th>
<th>Task</th>
<th>DNN Models</th>
<th>Datasets</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMG1</td>
<td>Image Classification</td>
<td>VGG16 [86]</td>
<td>ILSVRC2012</td>
</tr>
<tr>
<td>IMG2</td>
<td></td>
<td>ResNet50 [33]</td>
<td>(ImageNet)</td>
</tr>
<tr>
<td>NLP1</td>
<td>Sentence Prediction</td>
<td>RNN</td>
<td>Penn Treebank [67]</td>
</tr>
<tr>
<td>NLP2</td>
<td>Question Answering</td>
<td>Bert [17]</td>
<td>Stanford Q&amp;A Dataset (SQuAD) [78]</td>
</tr>
</tbody>
</table>

Table 3.1: Hardware platforms used in our experiments

<table>
<thead>
<tr>
<th>CPU</th>
<th>ODroid</th>
<th>Core i5</th>
<th>Razer</th>
<th>Skylake</th>
<th>Xeon(R)</th>
<th>P100</th>
<th>K80</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODroid</td>
<td>ARM Cortex A-15 @2.0GHz</td>
<td>Core-i5 @2.9GHz</td>
<td>Core-i7 @2.2GHz</td>
<td>Xeon(R) Gold 6126 @2.60GHz</td>
<td>Xeon(R) E5-2670 @2.30GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i5</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>Tesla P100</td>
<td>Tesla K80</td>
</tr>
<tr>
<td>Razer</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>Tesla P100</td>
<td>Tesla K80</td>
</tr>
<tr>
<td>Skylake</td>
<td>DDR3 2G</td>
<td>DDR3 16G</td>
<td>DDR4 16G</td>
<td>DDR4 16G*12</td>
<td>DDR4 16G*8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xeon</td>
<td>DDR4 16G*8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: ML tasks and benchmark datasets in our experiments

3.1 Understanding Inference Latency

To emulate real-world scenarios, we feed the network one input at a time and use 1/10 of the total data for warm up. We run each input 10 times and use the average inference latency for that input after the warm up phase.
Q1: Might deadline violations occur? Yes (Figure 3.1). Different applications will likely impose different inference deadlines. Image classification on video might have deadlines from 1 second down to the camera latency (e.g., 1/60 seconds) [48]. The two NLP tasks, particularly NLP2, also risk deadline violations, as studies show human users typically expect text results in 1 second [72]. There is clearly no single DNN design that meets all deadlines on all hardware.

Q2: Does latency vary across inputs? Yes (Figure 3.1). While latency variance across hardware is not surprising, the variance is still non-negligible even for fixed networks and hardware. Except for NLP2, all tasks have outlier inputs that take much longer time to process across all platforms. For example, in IMG1’s Tesla P100 setting, the longest latency is 47× the median. Among all tasks, NLP1 has the largest input variance: its 75th (90th) percentile latency is at least 1.37× (1.72×) the median latency across all platforms. This large variance is likely because NLP1 inputs have different lengths.

Q3: Is latency affected by contention? Yes (Figure 3.2). We repeat the experiments from Figure 3.1 except we now make the inference job competes for resources with a co-located job, the memory-intensive STREAM benchmark [68]. Comparing Figures 3.2 and 3.1, we see that the co-located memory-intensive job increases both the median and tail inference latency. This trend also applies to CPU contention cases, and we omit the result figure due to space constraints.

Summary: Deadline violations are realistic concerns and inference latency varies greatly across inputs and platforms, with or without resource competition. Sticking to one static DNN design across inputs, platforms, and workloads would lead to an unpleasant trade-off: always meet the deadline by sacrificing accuracy in most settings, or achieve a high accuracy some times but fail the deadline in others.
Figure 3.1: Latency variance across inputs for different tasks and hardware (Most tasks have 5 boxplots for 5 hardware platforms, Core i5, Razer, Xeon, P100, K80 from left to right; NLP1 has an extra boxplot for Odroid platform; other three tasks ran out of memory on Odroid; every box shows the 25th–75th percentile; points beyond the whiskers are >90th or <10th percentile).

Figure 3.2: Latency variance under memory contention
3.2 Understanding Tradeoffs Across Networks

Many DNN models have been designed for classic tasks such as image classification. To understand the tradeoffs offered by different DNN models for the same task, we run all 42 trained image classification models provided by the Tensorflow website [84] on the 50000 images from ImageNet [16]. We also include a naive model that takes a random guess, incurring almost 100% error with no latency. Figure 3.3 shows the accuracy/latency tradeoffs (blue circles) offered by these models on Xeon (numbers are averaged across all images in ImageNet). The trends on other machines are similar.

Q4: What are the tradeoffs? As shown in Figure 3.3, these 42 models cover a wide spectrum of latency and accuracy tradeoffs. The fastest model runs almost 12× faster than the slowest one and the most accurate model has about 7.8× lower error rate than the least accurate model. While omitted due to space, we also find the networks exhibit a wide range—spanning more than 30× of energy usage. These results reflect the intuition that greater DNN accuracy generally comes at the cost of greater resource usage (e.g., time and energy).

![ImageNet Classification Networks](image)

Figure 3.3: Accuracy and latency tradeoffs for 42 DNNs on Xeon.
Q5: Which network provides the best tradeoff? There is no magic network that provides both the highest accuracy and the lowest latency or energy consumption. In Figure 3.3, all the networks that sit above the lower-convex-hull curve represent sub-optimal trade-offs. For example, MobileNet-v2 ($\alpha=0.75$), a network sitting above the curve incurs similar latency as ResNet50, but $1.5\times$ errors.

Summary: Inference latency tends to increase with DNN accuracy and energy usage. Given different deadlines and run-time environment, different DNN models should be chosen. Furthermore, since existing models are far from covering the whole latency spectrum (Figure 3.3), it is impractical to pick a DNN model that uses up the whole time before the deadline for its inference. These observations demonstrate the room for improvements in resource management and energy saving.
CHAPTER 4
ALERT RUN-TIME INFERENCE MANAGEMENT

ALERT’s runtime system addresses the challenges highlighted earlier of meeting user-specified latency, accuracy, and energy constraints and optimization goals across different DNN models, inputs, run-time contention, and hardware.

As illustrated in Figure 1.1, ALERT takes a set of DNN models and user-specified requirements on latency, accuracy, and energy usage. Then, for each input at runtime, ALERT measures feedback on each requirement and selects for the next inference (1) a DNN model $d_i$ from a set $\mathbb{D} = \{d_i | i = 1 \cdots K\}$, and (2) resource use, expressed as a power cap $p_j$ from $\mathbb{P} = \{P_j | j = 1 \cdots L\}$, to meet constraints in any two of the three dimensions of latency, accuracy, and energy, while optimizing the third. More formally, ALERT fulfills either\(^1\) of these user-specified goals:

1. Maximizing inference accuracy $q$ for an energy budget $E_{\text{goal}}$ and inference deadline $T_{\text{goal}}$:
   \[
   \arg \max_{i,j} q_{i,j} \ \text{s.t.} \ e_{i,j} \leq E_{\text{goal}} \land t_{i,j} \leq T_{\text{goal}} \tag{4.1}
   \]

2. Minimizing the energy use $e$ for an accuracy goal $Q_{\text{goal}}$ and inference deadline $T_{\text{goal}}$.
   \[
   \arg \min_{i,j} e_{i,j} \ \text{s.t.} \ q_{i,j} \geq Q_{\text{goal}} \land t_{i,j} \leq T_{\text{goal}} \tag{4.2}
   \]

We use a power budget as the interface between the run-time and the system resource manager. Both hardware [15] and software resource managers [37, 79, 100] can convert power budgets into optimal performance resource allocations. So, ALERT should be compatible with many different schemes from both commercial products and the research literature.

---

\(^1\) In this paper, we examine constraints on latency and accuracy as well as latency and energy. For space, we omit discussion of meeting energy and accuracy constraints while minimizing latency as it is a trivial extension of the discussed techniques and we believe it to be the least practically useful.
4.1 Key ideas

Satisfying the constraints of Eqs. 4.1 and 4.2 would be trivially easy if the deployment environment is guaranteed to match the training and profiling environment: we could estimate $t_{i,j}$ to be the average inference time $t_{i,j}^{\text{train}}$ over a set of profiling inputs under model $d_i$ and power setting $p_j$. However, this static approach does not work given the dynamic hardware, input, contention, and requirement variation. We break down the tasks and present ALERT key ideas below.

**How to estimate the inference time $t_{i,j}$?** To handle the run-time variation, a potential solution is to apply a reinforcement learner, like a Kalman filter [64], to make dynamic estimation based on recent history. The problem is that most models and power settings will not have been picked recently and hence would have no recent history to feed into the learner.

**Idea 1** To make effective online estimation for all combinations of models and power settings, ALERT introduces a global slow-down factor $\xi$ to capture how the current environment differs from the profiled environment (e.g., due to co-running processes, hardware variation, or others). Such an environmental slow-down factor is independent from individual model or power selection. It can be estimated fully leveraging the execution history, no matter which models and power settings were recently used; it can then be used to estimate $t_{i,j}$ based on $t_{i,j}^{\text{train}}$ for all $d_i$ and $p_j$ combinations.

**How to estimate the accuracy under a deadline?** Given a deadline $T_{\text{goal}}$, the inference accuracy delivered by model $d_i$ and power setting $p_j$ is determined by three factors, as shown in Eq. 4.3: (1) whether the inference result, which takes $t_{i,j}$ to compute, can be generated before the deadline $T_{\text{goal}}$; (2) if yes, the accuracy is determined by the model $d_i$;\(^\text{2}\)

\(^{2}\) Since it could be infeasible to calculate the exact inference accuracy at run time, ALERT uses the average training accuracy of the selected DNN model $d_i$, denoted as $q_i$, as the inference accuracy, as long as...
(3) if not, without any output at the deadline, a random guess will be used as the inference result, greatly reducing accuracy, denoted by $q_{fail}$:

$$q_{i,j}[T_{goal}] = \begin{cases} 
q_i & \text{if } t_{i,j} \leq T_{goal} \\
q_{fail} & \text{otherwise}
\end{cases} \quad (4.3)$$

A potential solution to estimate accuracy $q_{i,j}$ at deadline $T_{goal}$ is to simply feed the estimated $t_{i,j}$ into Eq. 4.3. However, this approach could lead to sub-optimal decisions under high run-time variation, when the real inference time has a high probability of rising above the estimated time. To understand this problem, imagine two configurations $C_1 = \{d_1, p_1\}$ and $C_2 = \{d_2, p_2\}$. Under $C_1$, the inference is estimated to finish right before the deadline with a high accuracy; under $C_2$, the inference is estimated to finish long before the deadline with a medium accuracy. The straightforward solution always picks $C_1$; however, if the run-time environment is highly variable—as in Figure 3.2—this pick might be wrong. Then, it is better to conservatively pick $C_2$ ensuring that a result—rather than a random guess—is available at the deadline.

**Idea 2** To handle the run-time variability, ALERT treats the execution time $t_{i,j}$ and the global slow-down factor $\xi$ as random variables drawn from a normal distribution. ALERT uses a Kalman Filter to estimate not only the mean value but also the standard deviation of $\xi$ (and hence $t_{i,j}$), which reflects the run-time environment variation. ALERT then computes $q_{i,j}$’s expected range, instead of just the mean.

**Idea 3** To further address the run-time variance and alleviate the cost of an incorrect estimation of $t_{i,j}$—no estimation can guarantee to be error free after all—ALERT proposes a novel DNN design so that even when the deadline is not met (e.g., the real $t_{i,j}$ is much longer than estimated), an inference result much better than a random guess can be provided. This design will be presented in Section 5.

---

the inference computation finishes before the specified deadline.
How to minimize energy or satisfy energy constraints? Minimizing energy or satisfying energy constraints for given latency (and accuracy constraints) is complicated, as the energy is related to, but cannot be easily calculated by, the complexity of the selected model $d_i$ and the power cap $p_j$. For a given model $d_i$, one could set a higher power to finish inference early and transitions to a low-power idle mode; or one could set a lower power so that the inference task runs for a longer time but also consumes less energy at each time unit.

Idea 4 ALERT leverages insight from previous research, which shows that energy consumption for latency-constrained systems can be efficiently expressed as a mathematical optimization problem [10, 57, 60, 71]. The mathematical optimization finds the greatest energy savings while accounting for both idle power and deadline constraints:

$$\arg \min_{i,j} e_{i,j} \quad \text{s.t.} \quad q_{i,j}[T_{\text{goal}}] \geq Q_{\text{goal}} \tag{4.4}$$

ALERT guarantees an energy constraint by predicting both $p_{i,j}$, the power when the DNN is running, and $p_{DNNidle_{i,j}}$, the power when the DNN is idle. We refer to the latter as DNN idle to denote that the system may be performing other non-DNN, tasks. ALERT picks the configuration within the energy budget with highest accuracy, turning the problem (Eq. 4.1) into:

$$\arg \max_{i,j} q_{i,j}[T_{\text{goal}}] \quad \text{s.t.} \quad p_{i,j} \cdot t_{i,j} + p_{DNNidle_{i,j}} \cdot t_{DNNidle_{i,j}} \leq E_{\text{goal}} \tag{4.5}$$

If a user provides power budget, ALERT converts it to an energy budget according to:

$$E_{\text{goal}} = P_{\text{goal}} \cdot T_{\text{goal}}. \tag{4.6}$$
4.2 Algorithm

4.2.1 Overview

ALERT follows four steps for each input $n$:

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

2) **Goal adjustment.** ALERT updates the time goal $T_{\text{goal}}$ if necessary. Different inputs may demand different inference deadlines (NLP1 workload in Table 3.2) and delays in previous input processing could shorten the available time for the next input [2, 56]. Additionally, ALERT sets the goal latency to compensate for its own, worst-case overhead so that the runtime system itself will not cause violations. ALERT also updates $Q_{\text{goal}}$ if needed.$^3$

3) **Feedback-based estimation.** ALERT updates the global slow-down factor $\xi$, including its mean and standard deviation (see Section 4.2.2) and computes the expected accuracy for every combination of DNN model and power setting (Section 4.2.3). ALERT uses the measured energy to update the DNN-idle power ratio $\phi$ and then predicts the energy consumption for each configuration (Section 4.2.4).

4) **Picking a configuration.** ALERT feeds all the updated estimations of latency, accuracy, and energy into Eqs. 4.4 and 4.5, and gets the desired DNN model and power-cap setting for $n$.

The key task is to estimate $\xi$ and then the accuracy and energy for different combinations of models and power limits.

4.2.2 Global Slow-down Factor

ALERT introduces a *global slow down factor* $\xi$ to reflect how the run-time environment differs from the profiling environment. Conceptually, if the inference task under model $d_i$

---

$^3$ In experiments, we consider the accuracy goal to be the average accuracy of any continuous N inputs. Consequently, the goal for each individual input is adjusted based on the actual accuracy delivered to the last $N - 1$ inputs.
and power-cap $p_j$ took time $t_{i,j}$ to finish at run time and took $t_{i,j}^{\text{train}}$ on average to finish during profiling, the corresponding slow-down factor $\xi$ would be $t_{i,j}/t_{i,j}^{\text{train}}$. We can estimate $\xi$ using recent execution history under any model or power setting and then apply $\xi$ to make estimations for all combinations of models and power.

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 $t_{i,j}^{\text{train}}$, and then uses a Kalman Filter to estimate the mean $\mu^{(n)}$ and standard deviation $\sigma^{(n)}$ of $\xi^{(n)}$ the slow-down factor at input $n$. A Kalman Filter is an optimal estimator that assumes a normal distribution and estimates a varying quantity based on multiple potentially noisy observations [64]. ALERT’s formulation is defined in Eq. 4.7, where $K^{(n)}$ is Kalman Filter variable; $R$ is a constant reflecting the measurement noise; $Q^{(n)}$ is the process noise capped with $Q^{(0)}$. We set forgetting factor of process variance $\alpha = 0.3$ [3]. ALERT initially sets $K^{(0)} = 0.5$, $R = 0.001$, $Q^{(0)} = 0.1$, $\mu^{(0)} = 1$, $\sigma^{(0)} = 0.1$, following the standard convention [64].

$$
\begin{align*}
Q^{(n)} &= \max\{Q^{(0)}, \alpha Q^{(n-1)} + (1-\alpha)(K^{(n-1)}y^{(n-1)})^2\} \\
K^{(n)} &= \frac{(1-K^{(n-1)})\sigma^{(n-1)} + Q^{(n)}}{(1-K^{(n-1)})\sigma^{(n-1)} + Q^{(n)} + R} \\
y^{(n)} &= t_{i,j}^{(n-1)}/t_{i,j}^{\text{train}} - \mu^{(n-1)} \\
\mu^{(n)} &= \mu^{(n-1)} + K^{(n)}y^{(n)} \\
\sigma^{(n)} &= (1 - K^{(n-1)})\sigma^{(n-1)} + Q^{(n)}
\end{align*}
(4.7)
$$

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

$$
t_{i,j}^{(n)} = \xi^{(n)} \times t_{i,j}^{\text{train}}.
(4.8)
$$

4.2.3 Accuracy

As discussed earlier, ALERT computes the estimated inference accuracy $\hat{q}_{i,j}[T_{\text{goal}}]$ by considering $t_{i,j}$ as a random variable that follows normal distribution with its mean and standard
deviation computed based on that of $\xi$:

\[
\hat{q}_{i,j}[T_{\text{goal}}] = E(q_{i,j}[T_{\text{goal}}] \mid t_{i,j}^{(n)}) \\
= E(q_{i,j}[T_{\text{goal}}] \mid \xi^{(n)}, t_{i,j}^{\text{train}}) \\
(4.9)
\]

$\xi^{(n)} \sim \mathcal{N}(\mu^{(n)}, (\sigma^{(n)})^2)$

### 4.2.4 Energy

ALERT predicts each configuration’s 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 by the Kalman Filter in Eq. 4.10. $\phi^{(n)}$ 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 $M^{(0)} = 0.01$, $S = 0.0001$, $V = 0.001$.

\[
\begin{cases}
W^{(n)} = \frac{M^{(n-1)} + S}{M^{(n-1)} + S + V} \\
M^{(n)} = (1 - W^{(n)})(M^{(n-1)} + S) \\
\phi^{(n)} = \phi^{(n-1)} + W^{(n)}(p_{\text{idle}}/p_{i,j}^{(n-1)} - \phi^{(n-1)})
\end{cases} \\
(4.10)
\]

ALERT then predicts the energy by Eq. 4.11.

\[
e^{(n)}_{i,j} = p_{i,j} \cdot \xi^{(n)} \cdot t_{i,j}^{\text{train}} + \phi^{(n)} \cdot p_{i,j} \cdot (T_{\text{goal}} - (\xi^{(n)} \cdot t_{i,j}^{\text{train}})) \\
(4.11)
\]

### 4.2.5 Limitations of ALERT

1. ALERT applies its global slow-down factor to all DNN/power configurations. It is possible that some perturbations lead to different slowdowns for different configurations. However, we believe the slight loss of accuracy here is out-weighed by the benefit of having a simple mechanism that allows prediction even for configurations that have not been used recently.

2. 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.

(3) ALERT incurs overhead in both scheduler computation and witching from one DNN/power-
setting to another. We find such overhead to be small compared with inference cost, just
0.6–1.7% of input processing time. We explicitly account for overhead by subtracting it from
the user-specified goal (see step 2 in Section 4.2.1). In effect ALERT schedules both itself
and the inference.

(4) ALERT provides probabilistic guarantees, not hard guarantees. Because 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; e.g., schedul-
ing for slow-down factors of up to three standard deviations corresponds to meeting the
goals 99.7% of the time. Providing 100% guarantees, however, requires much more conserv-
ative configuration selection—hurting both energy and accuracy—a property shared by all
systems that have to choose between probabilistic and hard guarantees [9].

4.3 Implementation

ALERT currently adjusts power through Intel’s Running Average Power Limit (RAPL)
interface [15], which allows software to set a hardware power limit. ALERT can also be
applied to other approaches that translate power limits into settings for combinations of
resources [37, 43, 79, 100]. For example, ALERT could be trivially modified for GPUs using
nvidia-smi to adjust GPU power. We could not explore it in this paper because the GPU
drivers that support Machine Learning frameworks (e.g. TensorFlow and Pytorch) on our
test systems do not allow software to set GPU speed. Furthermore, ALERT’s design is
compatible with any emerging specialized DNN accelerators that export a power management
interface.

ALERT does not change power setting continuously, but instead uses a set of discrete
buckets within the feasible range of power-limit settings. ALERT considers a series of settings
with 2.5W interval on our test laptop and a 5W interval on our test server, as the later has a wider power range than the former. The number of power buckets is configurable.

Equation 4.7 requires feedback to update the global slowdown factor $\xi$. ALERT cannot observe the full latency when a deadline is missed (because it immediately starts processing the next input). To account for this inability to measure accurate latency, we increase the latency we pass to the Kalman filter on a deadline miss by a factor of 0.2, ensuring ALERT chooses a more conservative configuration in the next iteration.

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.
CHAPTER 5
ALERT ANYTIME DNN DESIGN

ALERT provides probabilistic, not absolute, guarantees. When ALERT occasionally makes an incorrect prediction, perhaps during a sudden change of resource contention, the inference task may miss its deadline and cause inference accuracy to drop to that of a random guess (Eq. 4.3).

To alleviate the damage of a missed deadline, we develop a new class of DNN model that outputs a series of increasingly accurate results over time: an *Anytime* DNN. Conceptually, it is a nested family of $k$ DNNs \{\(d_1, d_2, \ldots, d_k\}\). With time goes, it produces outputs \(o_1\), \(o_2\), ..., \(o_k\), with \(o_t\) more reliable than \(o_{t-1}\). Even if ALERT fails to handle an unpredictable run-time variation and the expected inference result \(o_k\) is not ready at the deadline, earlier results \(o_x\) (\(1 \leq x < k\)) can still be used with only slight accuracy loss, turning Eq. 4.3 into:

\[
q_{.,j} = \begin{cases} 
q_k & \text{if } t_{k,j} \leq t_{\text{goal}} \\
q_{k-1} & \text{if } t_{k-1,j} \leq t_{\text{goal}} < t_{k,j} \\
& \cdots \\
q_{\text{fail}} & \text{otherwise}
\end{cases}
\] (5.1)

Using a novel design methodology, we construct families of DNNs with extremely efficient execution properties for use as an Anytime DNN. Our key insight is to architect the network family so that later DNN models re-use internal state of earlier models, accelerating the execution of the entire Anytime DNN.

5.1 Design Principles

**Strawman** One can simply take a sequence of *independently* designed and trained DNNs with non-decreasing accuracy and latency. For example, we can find many such sequences from Figure 3.3. At run time, given an input, inference starts with the fastest, least accurate
network. After the first network completes, the second network is run, and so on until the
deadline. As an accuracy improvement, all outputs can be rolled into a classic ensemble [18].
Whenever the $k$-th model finishes, the ensemble will output a weighted average of outputs
from all of the first $k$ independent models, as illustrated in Figure 5.1.

This baseline design is intuitively ineffective: the first $(k-1)$ independent models’ computa-
tion seems largely wasted, with the ensemble only slightly improving, if at all, the accuracy
of the $k$-th individual model. To avoid similar problems in our Anytime DNN, we leverage
the following two observations.

**Observation-1** An effective way to improve accuracy with more computation is to grow
a network larger, moving to both deeper (more layers) [33, 86, 88] and wider (more neurons
per layer) [99] designs. An idea which applies to not only vision examples but also language
models [17]. Moreover, while not apparent from Figure 3.3, the trend is especially pronounced
within an architecture family: if residual networks (ResNets) are preferred for a particular
domain, a 50-layer ResNet will deliver better accuracy than a 34-layer ResNet [33].

**Observation-2** We could reduce the computational costs in an Anytime DNN by fully
reusing internal states of earlier networks to boot-strap later networks. Each layer within a
very deep neural network might be learning to perform an incremental update on a certain
representation [25, 47]; deeper networks might refine their representations more gradually
than shallower networks. If we were to align layers of different networks trained for the same
A task according to the relative depth in their own network we might find they are using related representations. This representation-similarity suggests that we can jump-start computation in larger networks by giving them direct access to the previously computed internal states of smaller networks. It also suggests specific design rules for connecting the nested smaller and larger networks.

Note that, having aligned state sharing is important. For example, an established image-classification approach cascade is used in several recent network designs \([41, 69, 90]\), in which some inputs traverse a subset of layers while others may traverse all layers. We do not use this design for our Anytime inference, as it does not have the alignment and connectivity patterns required to nest networks for optimal state re-use. Figure 5.2 illustrates a particular cascade pattern, typical of prior work. Here, as some outputs are generated without traversing later pipeline stages—which tend to capture high-level input features—this design can lead to large accuracy loss; the additional computation on every early output path to compensate for the skipped structures (red boxes in Figure 5.2) can recover some, not all, of the accuracy loss, and leads to inefficiency.

### 5.2 Proposed Nested Anytime DNN

Our Anytime design contains a sequence of fully nested DNNs: the first \(d_1\) is completely contained within the second \(d_2\), which is a subpart of \(d_3\), and so on. Progressing from \(d_x\) to \(d_{x+1}\), our scheme permits growing the DNN in width, depth, or both. In each case, we connect internal layers of the smaller network to internal layers of the larger network.
most appropriate for consuming their signals, according the alignment principle explained previously. Together with the nesting requirement, our Anytime DNNs have the following properties:

- **Pipeline structure**: Every subnetwork $d_x$ follows the usual pipeline structure of a traditional DNN.

- **Feed-forward**: Neurons feed outputs into deeper layers of the same network or units in a larger network; connections are purely feed-forward in depth or nesting level.

Every subnetwork $d_x$ will closely, but not exactly, resemble a traditional non-nested network. The difference is that some neurons in an outer nesting level will not feed backward into neurons in inner nesting levels. Dropping these connections slightly lightens compute load and parameter count, slightly shifting the network’s position on the latency-accuracy curve\(^1\), which is inconsequential in the big picture of a Anytime DNN that populates the tradeoff curve with many nested networks. Critically, our approach is general and works with different types of DNNs without application-specific changes.

### 5.2.1 Width Nesting

**Architecture**  
Width nesting divides a traditional DNN into $M$ horizontal stripes, forming a family of $M$ width-nested DNNs, with the $i$-th DNN including all the neurons inside the first $i$ stripes. The connections in the original DNN then fall into three categories, as shown in Figure 5.3: (1) connections between neurons in the same stripe (horizontal edges and invisible edges inside each box); (2) connections from an earlier-stripe neuron to a later-stripe neuron (downward edges); (3) connections from a later-stripe neuron to an earlier-step neuron (upward gray edges). The corresponding width-nested Anytime DNN contains all edges of the types (1) and (2), but drops edges of type (3) to satisfy the nesting property.

---

\(^1\) Compared with skipping a neural network pipeline stage, cutting a few edges does not hurt accuracy too much. Indeed, many standard DNN designs use sparse or grouped connection structures, which do not connect channels in a subsequent layer with all channels in the prior layer.
Assessment  This design applies to almost all DNNs; Section 6 demonstrates this generality on multiple DNN types, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and a self-attention network.

We use a power of 2 sequence for stripe widths: if the first nested network \( d_1 \) contains \( w \) neurons in one layer, \( d_x \) contains \( w \times 2^{x-1} \) neurons in the corresponding layer. This choice creates a good trade-off curve for accuracy and latency. Additionally, comparing the outermost DNN in the Anytime ensemble to a traditional DNN of the same width, only a small portion of edges are pruned due to the nesting requirement.

Execution  Inference is conducted in a zig-zag manner: (1) the first result \( o_1 \) is produced using all neurons in the first stripe (from left to right along the top layers in Figure 5.3); (2) execution moves down to the second stripe, and \( o_2 \) is produced by going through all layers of the second tripe (left to right), which receive, as additional inputs, neuron activations from the first stripe. Inference proceeds moving down stripes until the deadline, at which time the system reports the most recent result, or an ensemble of all completed results.

5.2.2 Depth Nesting

Architecture  Depth nesting is more complicated. A standard cascade (Figure 5.2) destroys the traditional DNN pipeline structure; intermediate layers serve heterogeneous branches
We use a fundamentally different approach to nest networks by depth: we *interlace* layers following the same pipeline structure as the original DNN. Figure 5.4 illustrates. In this example, we partition a traditional DNN into odd and even layers, create a shallower network consisting of only the odd numbered layers (Fig. 5.4.a), and nest it within the full network (Fig. 5.4.b). Figure 5.2.c shows an alternative, but equivalent, view. Gray edges (from layer 2 to 3, and layer 4 to 5), present in the standard DNN, must be pruned to achieve nesting. Repeatedly nesting via interlacing, the depths of the nested networks $d_1$, $d_2$, ... $d_k$ naturally follow an incremental sequence of powers of 2. As was the case for width-nesting, this offers a series of meaningful latency/energy-accuracy trade-offs.

**Assessment** Unlike width-nesting, our depth-nesting strategy applies only to DNNs satisfying an additional architectural requirement. Notice, in Figure 5.4, the presence of additional skip connections between layers, even in the basic shallow DNN. Indeed, within any DNN in the sequence, we see that each layer connects directly to any other layer separated
in depth by a power of 2 (not accounting for pruned connections).

Fortunately, this power-of-2 skip-connection design is exactly the SparseNet architecture [103], which is a state-of-the-art variant of the ResNet [33] (or DenseNet [42]) CNNs. Our width and depth nesting designs can be easily combined in any arbitrary nesting order.

5.3 Training and Implementation

We apply our Anytime DNN design to a variety of DNN models (ResNet, SparseNet, RNN, Self-attention), with implementations using both Pytorch [75] and Tensorflow [1]. Pytorch uses eager execution, which naturally fits nesting computation. Tensorflow conducts graph-based computation, so we use its partial run API [24] to fulfill the same functionality.

Training We explore two types of training: greedy and joint. The greedy method trains the first stripe first to achieve the highest possible accuracy. It then freezes all network weights inside the first stripe, and trains the second network, which contains the first two stripes; and then continues for the third, and so on. This strategy guarantees high accuracy for the narrowest network. The joint training method considers the whole network together, but places different importance on the output of different subnetworks. The flexibility to specify per-output importance could be used to train custom network to match known operating environment characteristics.

Infrastructure-induced overheads In theory, Anytime DNN inference should take equal or less time than a similar traditional DNN that outputs a single result, as the Anytime DNN prunes some connections. Unfortunately, existing Pytorch and Tensorflow infrastructures are highly optimized for computing one DNN with one result, not nested DNNs. Thus, we observe slowdowns of our Anytime DNNs, ranging from negligible to more than 50%, which is rare, for different infrastructures and different DNN models. These slowdowns are not inherent to Anytime DNN design and should be mitigated with future APIs. As Section 6
shows, even with these infrastructure-induced slowdowns, Anytime DNNs have significant benefits.

**Limitations** Of course, anytime DNN is not completely a free lunch. Comparing with traditional DNN, we expect a slight accuracy drop at certain nested layers because (1) some connections are pruned, and (2) the training process cannot optimize the accuracy of every single nested DNN. Fortunately, as we will see in the evaluation section (Section 6.2), the accuracy drops little, like only 0.3 percent for the deepest Anytime Sparse ResNet with joint training.
CHAPTER 6
EXPERIMENTS

6.1 ALERT runtime management

6.1.1 Methodology

We use the Razer and Skylake machines (Table 3.1), which represent a typical laptop and server. The evaluation covers various constraint settings, 3 different run-time environments, 2 inference tasks, and 2 DNN types, with different ALERT nesting approaches, as shown in Table 6.1.

We compare ALERT with 5 other schemes, as shown in Table 6.1. The two Oracle schemes have perfect predictions for latency and energy-consumption of every input under every DNN/power setting (i.e., impractical). “Dynamic” schemes allow DNN/power setting changes across inputs; “Static” has one fixed setting across inputs. “System default” refers to the default power-management scheme used on our machines, which is a Race-to-Idle approach [57]. The three alternative ALERT schemes each only use part of the ALERT algorithm.

6.1.2 Results

Table 6.2 shows the average error rate and energy saving rate, the smaller the better, for all combinations of latency and accuracy/energy constraints (35–40 combinations for each task), normalized to OracleStatic. As we can see, ALERT achieves almost identical effect as the Oracle scheme, providing much better optimization than all other schemes (even impractical ones) while satisfying constraints.

Comparison with Oracle, OracleStatic The Oracle scheme represents the theoretically optimal result using traditional DNN designs, perfect prediction capability, dynamic config-
### Run-time environment setting

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Inference task has no co-running process</td>
</tr>
<tr>
<td>CPU</td>
<td>Co-locate with Bodytrack from PARSEC-3.0 [8]</td>
</tr>
<tr>
<td>Memory</td>
<td>Co-locate with memory-hungry STREAM [68]</td>
</tr>
</tbody>
</table>

### Ranges of constraint setting

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>0.4x–2x mean latency* of the largest Anytime DNN</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Whole range achievable by trad. and Anytime DNN</td>
</tr>
<tr>
<td>Energy</td>
<td>Whole feasible power-cap ranges on the machine</td>
</tr>
</tbody>
</table>

### Task

<table>
<thead>
<tr>
<th>Task</th>
<th>DNN model</th>
<th>Nesting</th>
<th>Fixed deadline?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Classif.</td>
<td>Sparse ResNet50</td>
<td>Depth</td>
<td>Yes</td>
</tr>
<tr>
<td>Sentence Pred.</td>
<td>RNN</td>
<td>Width</td>
<td>No</td>
</tr>
</tbody>
</table>

### Scheme ID

<table>
<thead>
<tr>
<th>Scheme ID</th>
<th>DNN model selection</th>
<th>Power selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>Dynamic optimal</td>
<td>Dynamic optimal</td>
</tr>
<tr>
<td>Oracle_static</td>
<td>Static optimal</td>
<td>Static optimal</td>
</tr>
<tr>
<td>ALERT</td>
<td>ALERT default</td>
<td>ALERT default</td>
</tr>
<tr>
<td>ALERT_Trad</td>
<td>ALERT w/o Anytime DNNs</td>
<td>ALERT default</td>
</tr>
<tr>
<td>ALERT_DNN</td>
<td>ALERT default</td>
<td>System default</td>
</tr>
<tr>
<td>ALERT_Power</td>
<td>The fastest traditional DNN</td>
<td>ALERT default</td>
</tr>
</tbody>
</table>

Table 6.1: Settings, tasks, and schemes under evaluation (* measured under default setting without resource contention)*

uraion across inputs, and no management overhead. As shown in Table 6.2 and visualized in Figures 6.1 and 6.2, ALERT achieves 93-99% of Oracle’s energy and accuracy optimization while satisfying constraints! Oracle_static, the baseline in Table 6.2, represents the best one can achieve by selecting 1 DNN model and 1 power setting for all inputs. ALERT greatly out-performs Oracle_static. Comparing with Oracle_static, ALERT reduces the energy consumption by 2–50% (33% in harmonic mean) and reduces the error rate by 9-77% (45% in harmonic mean).

Across all tests, ALERT satisfies the constraint in 99.9% cases for image classification and 98.5% cases for sentence prediction. For the latter, due to the large input variability (NLP1 in Figure 3.1), some input sentences simply take too long to be processed before the deadline even with the fastest DNN, for which the Oracle fails, too.

Note that, ALERT is running on the same machines as the DNN workloads. So, all results include ALERT’s run-time latency and power overhead.
<table>
<thead>
<tr>
<th>Platform</th>
<th>DNN</th>
<th>Env.</th>
<th>ALERT</th>
<th>ALERT\textsubscript{Trad}</th>
<th>ALERT\textsubscript{DNN}</th>
<th>ALERT\textsubscript{Power}</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Razer</td>
<td>Sparse Resnet</td>
<td>Default</td>
<td>0.98</td>
<td>0.96\textsuperscript{1}</td>
<td>1.33</td>
<td>0.95\textsuperscript{19}</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>0.95</td>
<td>1.02\textsuperscript{9}</td>
<td>1.92</td>
<td>0.96\textsuperscript{19}</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory</td>
<td>0.93</td>
<td>0.99\textsuperscript{8}</td>
<td>1.82</td>
<td>0.94\textsuperscript{19}</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>RNN</td>
<td>Default</td>
<td>0.61</td>
<td>0.62\textsuperscript{11}</td>
<td>1.06</td>
<td>0.96\textsuperscript{29}</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>0.60</td>
<td>0.74\textsuperscript{20}</td>
<td>1.28</td>
<td>0.94\textsuperscript{30}</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory</td>
<td>0.54</td>
<td>0.80\textsuperscript{25}</td>
<td>1.06</td>
<td>0.94\textsuperscript{30}</td>
<td>0.54</td>
</tr>
<tr>
<td>Skylake</td>
<td>Sparse Resnet</td>
<td>Default</td>
<td>0.97</td>
<td>0.97\textsuperscript{1}</td>
<td>0.97</td>
<td>0.92\textsuperscript{19}</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>0.58</td>
<td>0.58\textsuperscript{7}</td>
<td>0.91</td>
<td>0.59\textsuperscript{20}</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory</td>
<td>0.50</td>
<td>0.43\textsuperscript{30}</td>
<td>0.78</td>
<td>0.46\textsuperscript{25}</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>RNN</td>
<td>Default</td>
<td>0.87</td>
<td>0.81\textsuperscript{19}</td>
<td>0.85</td>
<td>0.83\textsuperscript{29}</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>0.60</td>
<td>0.55\textsuperscript{24}</td>
<td>0.89</td>
<td>0.55\textsuperscript{30}</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory</td>
<td>0.52</td>
<td>0.42\textsuperscript{34}</td>
<td>0.77</td>
<td>0.45\textsuperscript{34}</td>
<td>0.52</td>
</tr>
<tr>
<td>Harmonic mean</td>
<td></td>
<td></td>
<td>0.67</td>
<td>0.68\textsuperscript{4}</td>
<td>1.04</td>
<td>0.74\textsuperscript{24}</td>
<td>0.66</td>
</tr>
</tbody>
</table>

| Harmonic mean |       |         | 0.45   | 0.50                    | 0.55                    | 0.58                    | 0.43   |

Table 6.2: Normalized average energy consumption and error rate to Oracle\textsubscript{Static} (Smaller is better) Superscript means how many settings violate constraints for more than 10% cases. We report the average normalized energy only for non-violating settings.
Comparison with alternative designs  As we can see in Table 6.2, the three alternative/partial designs of ALERT are all much worse than ALERT, indicating that ALERT’s different components complement each other well.

Among the three, ALERT\textsubscript{DNN} is the only one that can continue to satisfy accuracy constraints, benefiting from Anytime DNNs. However, its adoption of system-default power management causes it to save much less energy than ALERT. On the other hand, with only ALERT power management but not ALERT DNN selection, ALERT\textsubscript{Power} not only violate a huge number of accuracy constraints, but also cannot save as much energy as ALERT. Finally, ALERT\textsubscript{Trad}’s optimization results are only slightly worse than ALERT when it can satisfy constraints. Unfortunately, without using Anytime DNNs, it frequently violates accuracy constraints, particularly for sentence-predication task (RNN), where the latency deadline changes from one RNN input (i.e., a word) to the next, depending on how much time remains for the whole sentence.

Case study: changing environment  Figure 6.3 shows the different dynamic behavior of ALERT (blue curve) and ALERT\textsubscript{Trad} (orange curve) when the environment changes from
Figure 6.2: ALERT versus Oracle and Oracle\textsubscript{Static} on maximize accuracy task (Lower is better). (whisker: whole range; circle: mean)

Default to Memory-intensive and back.

At the beginning, due to a loose latency constraint, ALERT and ALERT\textsubscript{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 ALERT and ALERT\textsubscript{Trad} quickly detect this problem, including not just the misses, but the high variability in the expected latency. ALERT switches to use an anytime network and a lower power cap. This switch is very effective: although the environment is still unstable, the inference accuracy remains high, with slight ups and downs depending on which nested DNN finished before the deadline (ALERT occasionally goes back to use the fastest traditional DNN to catch the deadline). Only able to choose from traditional DNNs, ALERT\textsubscript{Trad} conservatively switches to much simpler and hence lower-accuracy traditional DNNs to avoid deadline misses. This switch does eliminate deadline misses under the highly dynamic environment, but many of the conservatively chosen DNNs finish a while 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 detailed results demonstrate that ALERT makes use of the full potential of the proposed DNN and resource management co-design process.

Figure 6.3: Maximize accuracy for image classification on Razer. (ALERT in blue; $\text{ALERT}_{\text{Trad}}$ in orange; constraints in red. Memory contention occurs from about input 46 to 119; Deadline: $1.25 \times$ mean latency of largest Anytime DNN in Default; power limit: 35W.)
6.2 Anytime-DNN design

6.2.1 Methodology

We evaluate three inference tasks: image classification, with the CIFAR-10 [58] dataset,\(^1\), NLP sentence prediction, with Penn Treebank [67], and a recommendation system, with the MovieLens 1M Dataset [29].

ALERT’s width nesting approach can apply to all kinds of DNNs. For image classification, we apply width nesting to ResNet50 [33], and will compare our nesting design with a corresponding family of 6 ResNet50 networks [33] that have width doubled from one to the other. For sentence prediction and recommendation system, width nesting is applied to RNN [70] and self-attention networks [54], respectively, and will be compared with a family of 5 RNN networks and 5 self-attention networks, respectively. ALERT’s depth nesting is applied to Sparse ResNet50 [103] for image classification, and will be compared with a family of 5 Sparse ResNet50 networks that have depth doubled from one to the next. The number of networks in a family is determined by whether they offer meaningful accuracy-latency tradeoffs.

6.2.2 Results

Figure 6.4 shows the average accuracy and latency tradeoff points offered by different steps or members of (1) an ALERT Anytime DNN, (2) an ensemble DNN (Figure 5.1), and (3) a family of traditional DNNs. We name the last scheme as Oracle in the figure, because its trade-off points are impractical to achieve in a realistic latency-constraint setting, which requires perfect knowledge about which member DNNs can finish execution before the deadline, with any mis-prediction dropping the average accuracy greatly.

As we can see, ALERT Anytime DNNs, both depth nesting or width nesting, offer

\(^1\) We do not use ImageNet here, as it took too long to train all the different models described below. Actually, CIFAR is a good indicator for ImageNet [105]. Some popular vision networks [33, 42, 99] also use CIFAR10.
accuracy-latency tradeoffs that are much better than the Ensemble and very close to the infeasible Oracle, confirming that our nesting-based Anytime DNN design gives up little accuracy for the huge gain in flexibility.

Figure 6.4: Accuracy-latency trade-off (lower is better; all results from the Xeon machine, #5 in Table 3.1)
CHAPTER 7
RELATED WORK

7.1 Anytime DNN Design

Anytime algorithms apply to many learning techniques [104]. For example, anytime decision
trees use ensembles or boosting [26, 94, 97, 98]. Constructing anytime DNNs requires the
more involved process of integrating anytime capability into the DNN during training.

Adaptive inference skips parts of a DNN to save time on an input-dependent basis [21,
69, 92, 96]. This differs from anytime as input difficulty—rather than a deadline—determines
what DNN components are executed. Using these techniques deadline violations are more
likely for difficult inputs.

As discussed earlier, cascade methods for anytime DNNs lack generality and efficiency
under tight deadlines [41, 59, 90]. Wang et al. develop anytime prediction specifically for
stereo depth estimation to output images with increasing resolution, an approach that is
tied to image output and does not apply to other tasks [95].

7.2 Dynamic Resource Management

Past resource management systems have used machine learning [6, 61, 76, 77, 87] or control
theory [36, 44, 52, 53, 71, 82, 102] to make dynamic decisions and adapt to changing environ-
ments or application needs. Several of these also make use of the Kalman filter because it
has optimal error properties [44, 52, 53, 71]. There are two major differences between these
prior approaches and ALERT. First, prior approaches use the Kalman filter to estimate
physical quantities such as CPU utilization [53] or job latency [44], while ALERT estimates
a virtual quantity that is then used to update a large number of latency estimates. Second,
while standard deviation 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.

Past work designed resource managers explicitly to coordinate approximate applications with system resource usage [20, 35, 36, 55]. Although related, these managers work with *independently* developed approximate applications and manage them *separately* from system resources, which is fundamentally different from that in ALERT, where the application and the manager are co-designed. Specifically, 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 [35, 36]. In practice this design forces the manager to model each combination of application and system setting independently, and delays response to changes. In contrast, 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, like the single input delay demonstrated in Figure 6.3.

In addition, anytime DNNs differ from traditional approximation methods, which configure the approximation level before execution. Instead, no configuration is needed for anytime DNNs, the approximation/accuracy is naturally determined when the time budget expires; i.e., it moves from one tradeoff point to another naturally during execution. Consequently, anytime allows ALERT to meet both accuracy and timing constraints while aggressively reducing energy, whereas prior work had to either forgo accuracy guarantees—e.g., [20]—or sacrifice optimality to achieve those guarantees—e.g., [35].

### 7.3 DNN Support

Much work accelerates DNNs and (largely) retains the base accuracy through ASICs, FPGAs, micro-architectures and ISA extensions [4, 12–14, 19, 22, 27, 31, 34, 45, 51, 63, 66, 74, 80, 83, 91], compilers [11, 73], or system support [32, 62]. These approaches are orthogonal to ALERT. ALERT has the complementary objective of managing tradeoff spaces induced by a combination of user goals, and a family of DNN designs. All the above techniques shift the range
of the tradeoffs, but the tradeoffs still exist and we expect ALERT to handle them just as well in this new context. Some other works improves DNN latency or energy by changing the required computation \cite{28, 28, 30, 46, 50, 85, 89, 93} through low-precision representation and DNN pruning, like traditional approximation methods. They are also orthogonal to ALERT. They add tradeoff points, but do not provide policies for meeting user needs or for navigating tradeoffs dynamically.

Some research supports hard real-time guarantees for DNNs \cite{101}. This work occupies a very different point in the design space, 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 as well. Furthermore, ALERT provides more flexibility to adapt to unpredictable environments. Hard real-time systems would fail in the co-located scenario unless they explicitly account for the impact of all possible co-located applications at design time.
CHAPTER 8
CONCLUSION

This paper demonstrates the challenges behind the important problem of ensuring timely, accurate, and energy efficient neural network inference on a variety of platforms with dynamic input, contention, and requirement variation. ALERT, our runtime manager, achieves these goals through dynamic DNN model selection, dynamic power management, and feedback control. Our general approach to creating anytime inference, width-nesting and depth-nesting, is particularly friendly for latency critical environments and makes the best use of ALERT.
REFERENCES


[60] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *USENIX ATC*, June 2011.


