UNDERSTANDING AND IMPROVING THE EFFICIENCY OF MACHINE LEARNING SOFTWARE: MODEL, DATA, AND PROGRAM-LEVEL SAFEGUARDS

by

Simin Chen

APPROVED BY SUPERVISORY COMMITTEE:

Wei Yang, Chair

Shiyi Wei

Tien Nguyen

Yapeng Tian

Lingming Zhang

To my family, friends, and my beloved, for their unwavering love and support.

UNDERSTANDING AND IMPROVING THE EFFICIENCY OF MACHINE LEARNING SOFTWARE: MODEL, DATA, AND PROGRAM-LEVEL SAFEGUARDS

by

SIMIN CHEN, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2024

ACKNOWLEDGMENTS

When I began writing the acknowledgment section of my dissertation, I realized it was time to say goodbye to my PhD journey. The six years of graduate school have been long, challenging, and incredibly rewarding. These years were made enjoyable by the wonderful people I met and interacted with along the way. I am extremely grateful for all the support and help I received from everyone. If I do not mention you here, please know that it is due to my own forgetfulness, not a lack of gratitude. Thank you all.

First and foremost, I am profoundly grateful to my advisors, Dr. Wei Yang and Dr. Cong Liu, for their unwavering support, insightful guidance, and invaluable feedback. Their expertise and encouragement have been instrumental in shaping this research, and their dedication to my academic and professional growth is deeply appreciated.

In 2019, Dr. Cong Liu offered me an opportunity despite my lack of a computer science background and my budding interest in the success of machine learning. After a brief conversation, Cong decided to give me a chance, and this offer completely transformed my life over the next few years, allowing me to immerse myself in the field of machine learning. Although Cong provided less technical guidance during my PhD journey, I am immensely grateful for his trust and support.

I would also like to extend my deepest thanks to my other advisor, Dr. Wei Yang, who provided detailed research training, teaching me how to conduct research step by step. Wei supplied abundant reading materials to help me overcome my weak background in computer science, thereby expanding my knowledge and developing my research acumen. His detailed guidance has been pivotal in my transformation from a novice to a researcher, enabling me to contemplate the significance of my future research and career. I believe his mentorship will benefit my entire research career, and I aspire to impart this wisdom to my future students. I vividly remember the nights before deadlines, working together with Wei to revise papers.

Staying up late was challenging, but looking back, I am immensely grateful for those moments and Wei's unwavering support. Moreover, Wei brings an incredible amount of excitement and passion to research and student advising. I hope to emulate this enthusiasm in my future endeavors.

I would also extend their heartfelt thanks to the members of my dissertation committee, Dr. Shiyi Wei, Dr. Tien N. Nguyen, Dr. Yapeng Tian, and Dr. Lingming Zhang, for their constructive critiques and suggestions that have significantly improved the quality of this work. In addition, I am deeply grateful for their valuable advice on presenting my research and navigating my job search.

My sincere appreciation goes to my colleagues and friends at The University of Texas at Dallas. Their camaraderie, discussions, and collaborative spirit have made this journey both enjoyable and intellectually stimulating. Special thanks to Chengxu Zhang, Zihe Song, Zexin Li, Yufei Li, Rathnasuriya, Ravishka Shemal, Xaiodi Li, Miao Miao, jaeseong Lee, Xiaoning Feng, Haibo Yu, Guangzhao Sun, Yiming Chen, Sampath Grandhi, Zenong Zhang and Mirazal for their direct contributions and support. There is no way I could have done the work I did without the help of all these people, and I look forward to future collaborations with them.

Lastly and most importantly, I would like to thank my family, especially my parents. Their unwavering support throughout my PhD journey allowed me to focus on my research and complete this dissertation. I am also deeply grateful to my girlfriend, Zhihong Zhang. I am excited about the next chapters of our lives together and immensely appreciative of everything she does for me, as well as the happiness she brings into my life. Zhihong's selfless support, especially during the challenging times of the pandemic, was crucial in helping me navigate the difficulties of my PhD journey and face all the challenges that came my way.

Outside of research, I can be somewhat of a big baby in life, and Zhihong has been a tremendous help in managing everyday tasks. I cannot imagine how chaotic my life would

have been without her dedication. I express my deepest thanks to her for her unconditional support. Without her, completing my PhD journey—and this dissertation—would have been impossible.

July 2024

UNDERSTANDING AND IMPROVING THE EFFICIENCY OF MACHINE LEARNING SOFTWARE: MODEL, DATA, AND PROGRAM-LEVEL SAFEGUARDS

Simin Chen, PhD
The University of Texas at Dallas, 2024

Supervising Professor: Wei Yang, Chair

Machine learning (ML) software has become integral to various aspects of daily life, leveraging complex models such as deep neural networks (DNNs) that entail significant computational costs, especially during inference. This poses a challenge for the deployment of ML software on resource-limited embedded devices and raises environmental concerns due to high energy consumption. Addressing these challenges requires improving the efficiency of the ML software.

ML software consists of three core components: data, model, and program. This dissertation investigates efficiency optimization for these components. Existing research primarily focuses on model and program optimization but overlooks the critical role of data. Additionally, the robustness of model-level optimizations and the compatibility of program-level optimizations with model-level ones remain underexplored.

This dissertation aims to address these gaps. At the data level, it examines how training data impacts ML software efficiency and proposes techniques to mitigate efficiency vulnerabilities introduced by adversarial data. At the model level, it explores the robustness of dynamic neural networks (DyNNs) to efficiency degradation and presents methods to enhance their inference efficiency. At the program level, it introduces a novel approach to bridge

program-level and model-level optimizations, ensuring comprehensive efficiency improvements. Moreover, it also analyze the model leakage in the model acceleration process.

The contributions of this dissertation are threefold:

Data-level: This research evaluates the impact of training data on ML model efficiency, identifying efficiency backdoor vulnerabilities in DyNNs and proposing strategies to defend against them. Model-level: It examines computational efficiency vulnerabilities in DyNN architectures, developing tools like `NMTSloth` and `DeepPerform` to test and mitigate these vulnerabilities. Program-level: This dissertation introduces a program rewriting approach, `DyCL`, designed to adapt existing DL compilers for DyNNs, significantly enhancing inference speed. Additionally, this dissertation proposes an automatic method, `NNReverse`, which infers the semantics of the optimized binary program to reconstruct the DNN model from the compiled program, thereby quantifying model leakage risks.

Overall, this dissertation provides a comprehensive framework for optimizing ML software efficiency, integrating data, model, and program-level approaches.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Machine learning (ML) software plays an increasingly important role in our daily lives, from online shopping recommendations to autonomous driving (Zhang et al., 2014; Springenberg et al., 2015; Shin et al., 2015; Zheng et al., 2019; Pei et al., 2017; Li et al., 2018; Kurtoglu and Tumer, 2008; Kocić et al., 2019; Yurtsever et al., 2020; Levinson et al., 2011; Campbell et al., 2010; Fan et al., 2023; Wu et al., 2022; Cui et al., 2020, 2024; Zhou et al., 2024; Rajput et al., 2024; Gao et al., 2024). The success of ML software largely stems from its backend ML models, such as deep neural networks (DNNs), which typically contain millions of parameters (He et al., 2016; Achiam et al., 2023; Floridi and Chiriatti, 2020; Devlin et al., 2018; Peng et al., 2019). However, the power of DNNs comes with substantial computational costs, particularly during inference (Hu et al., 2021; Li et al., 2020; Ye et al., 2020; Hu et al., 2019; Yao et al., 2019; Yang et al., 2024). This is a significant concern when deploying DNNs on resource-constrained embedded devices like mobile phones and IoT devices (Chen et al., 2021; Russo et al., 2021; Hao et al., 2019; Li et al., 2024; Eccles et al., 2024; Chen et al., 2021). Additionally, the considerable computational demands of ML software present environmental challenges due to their large carbon footprint (Patterson et al., 2021; Anthony et al., 2020; Savazzi et al., 2022; Qiu et al., 2023). As the use of ML software continues to expand, enhancing its efficiency becomes crucial (Chen et al., 2021; Lee et al., 2020; Li et al., 2023), this dissertation must develop techniques to improve the energy efficiency of inference processes in ML software.

Unlike traditional symbolic software, which consists solely of the program (Boehm, 2006; Van Vliet et al., 2008), ML software comprises three main components: (1) *data*, (2) *model*, and (3) *program* (Lewis et al., 2021; Dilhara et al., 2021). These components collectively define modern ML software. In the development lifecycle of ML software, developers first collect data from the Internet. Next, they design a novel ML model architecture and train the

model parameters using the data collected on a powerful machine. Finally, the ML software implementation program is further optimized for the hardware platform of deployment (Hapke and Nelson, 2020; Chen et al., 2018; Zheng et al., 2020; Drori et al., 2021; Narayanan et al., 2021; Zhou et al., 2020; Mbata et al., 2024; Shojaee Rad and Ghobaei-Arani, 2024). More details about the components of ML software are described in Chapter 2.

Recently, the research community has identified energy-efficiency issues during the inference phase of ML software and has proposed various solutions to enhance its efficiency (Chen et al., 2018; Zheng et al., 2020; Narayanan et al., 2021; Vanholder, 2016; Yu et al., 2017; Guan et al., 2024). These existing efforts can generally be categorized into two types: (1) model-level optimization and (2) program-level optimization. For model-level optimization, a notable approach is the dynamic allocation of computational resources based on different ML model inputs (Han et al., 2021a). For example, Google introduced `Mixture-of-Depths` (Raposo et al., 2024), which dynamically allocates compute resources in transformer-based language models. Additionally, several other algorithms, such as `SkipNet` (Wang et al., 2018), `Blockdrop` (Wu et al., 2018), and `DeepShallow` (Kaya et al., 2019), modify ML models through dynamic allocation. In terms of program-level optimization, this line of work focuses on enhancing the efficiency of ML software by refining the underlying systems and frameworks used for inference (Chen et al., 2018; Zheng et al., 2020; Chen et al., 2023; Google, 2017; Li, 2020; Jocher et al., 2022). One significant area of research involves rewriting the underlying program executable of the ML model implementation to fully leverage the parallelism of the deployed hardware platform (Chen et al., 2018; Li et al., 2020a; Rotem et al., 2018; Sabne, 2020). These methods distribute computation across multiple cores or devices, resulting in substantial speedups during inference. A representative example of this type of optimization is `TVM` (Chen et al., 2018), an end-to-end tool that generates high-performance executable code for ML models. More details about optimizing ML software efficiency can be found in Section 8.1.

Although some existing work has been proposed to improve the efficiency of ML software at the inference stage, there are still several limitations: (1) Existing work primarily focuses on the model and program components of ML software (Chen et al., 2018; Han et al., 2021b; Li et al., 2020a). However, data also plays a crucial role. The impact of data on ML software's inference time efficiency has not been adequately explored. (2) Current model-level optimizations dynamically allocate computational resources based on the model's inner predictions. However, research has shown that ML models are not robust to unnoticeable perturbations (Li et al., 2019; Dalvi et al., 2004; Bojchevski and Günnemann, 2019; Grosse et al., 2016; Lee et al., 2024). Consequently, it remains unclear whether these model-level optimizations are robust. If they are not, the potential efficiency degradation needs to be understood, and methods to mitigate such degradation must be developed to enhance the overall efficiency of ML software. (3) Program-level optimizations are typically designed independently of model-level optimizations (Chen et al., 2018; Zheng et al., 2020). It is still unknown whether program-level optimizations are compatible with model-level optimizations. If they are not, strategies must be devised to coordinate these optimizations to further improve the efficiency of ML software.

This dissertation aims to address the three aforementioned challenges. First, at the data level, this dissertation presents novel techniques to investigate how the training data affects the final efficiency of ML software (Chen et al., 2023). Second, at the model level, this dissertation explores model robustness in terms of inference time efficiency and proposes a series of techniques to mitigate potential efficiency degradation, ensuring efficient inference (Chen et al., 2022; Feng et al., 2024; Chen et al., 2022, 2023). Lastly, at the program level, this dissertation bridges the gap between program-level and model-level optimizations by automatically adopting existing program-level techniques for model-level optimization (Chen et al., 2023). In addition to improving the efficiency of ML software, I have also worked on enhancing ML software trustworthiness, explainability (Chen et al., 2020), transparent

benchmarking (Chen et al., 2024; Li et al., 2024), secure deployment (Chen et al., 2022b, 2021; Haque et al., 2021; Li et al., 2023), and automatic code generation (Li et al., 2021). However, the focus of this dissertation is specifically on addressing issues related to the efficiency of ML software.

1.1 Thesis Statement

The thesis statement of this dissertation is as follows:

The efficiency of ML software can be influenced by data, models, and programs. Developers can enhance ML software efficiency through optimization across these three dimensions.

This dissertation makes the following contributions:

- **Data-level.** This dissertation evaluates the impact of an ML model's training data on its final efficiency. Training data is a crucial component of ML software, often collected from the Internet (Chen et al., 2024; Achiam et al., 2023; Floridi and Chiriatti, 2020). Due to the large size of these datasets, verifying their effectiveness manually is challenging, and they may include polluted data (Gao et al., 2020). However, there is no existing work investigating how polluted data in the training dataset affects the efficiency of the ML models trained on it. Specifically, this dissertation is the first to study the efficiency backdoor vulnerability of dynamic neural networks (DyNNs) (Han et al., 2021b). This dissertation find that the computational consumption of DyNNs can be manipulated by adversaries to create a false sense of efficiency. Adversaries can produce triggered inputs to exhaust the computational resources of victim DyNNs, harming system availability. To address this, this dissertation proposes a novel "unconfident" training strategy to "supervisely" teach victim DyNNs to produce uniformly distributed confidence scores. After injecting backdoors into the DyNNs, these networks will produce uncertain predictions for triggered inputs, forcing them to continue computing without

early termination. this dissertation evaluate `EfficFrog` on 576 different settings. The results show that `EfficFrog` successfully injects efficiency-based backdoors into DyNNs, resulting in more than a 3x performance degradation. These findings underscore the necessity of protecting DyNNs against efficiency-based backdoor attacks.

- **Model-level.** At the model level, this dissertation examine the efficiency robustness of dynamic neural networks (DyNNs) and are the first to identify computational efficiency vulnerabilities in state-of-the-art DyNN architectures. These vulnerabilities can severely affect latency, energy performance, user experience, and service availability. Our empirical studies on 1,455 publicly available DyNN architectures reveal that these issues are widespread. To address this, this dissertation developed `NMTSloth` (Chen et al., 2022) and `DeepPerform` (Chen et al., 2022), the first frameworks for testing the computational efficiency of DyNN models. this dissertation evaluated these tools on three real-world neural machine translation (NMT) systems (Google T5 (Google, 2022), AllenAI WMT14 (AllenAI, 2022), and Helsinki-NLP (Jörg et al., 2020)) and found that existing correctness-based methods fail to significantly impact computational efficiency. In contrast, `NMTSloth` generates test inputs that increase latency and energy consumption by up to 3153% and 3052%, respectively. `DeepPerform` uses a learning-based approach to generate test samples for performance testing efficiently. Evaluations on five DyNN models and three datasets show that `DeepPerform` identifies more severe and diverse performance bugs with minimal overhead. To mitigate these issues, this dissertation propose a lightweight runtime detector for input validation, which our evaluations confirm as both effective and efficient.

- **Program-level.** At the program level, this dissertation conducted an empirical study using two popular DL compilers, `TVM` (Chen et al., 2018) and `OnnxRuntime` (Microsoft, 2017), to compile four types of DyNNs. The study revealed limitations in these compilers

when handling DyNNs. To address this, this dissertation presents a program rewriting approach that adapts existing DL compilers to compile DyNNs correctly. The key innovation of our approach is identifying and representing the dynamism of DyNN programs in heterogeneous control flow graphs (HCFGs). This dissertation compile the sub-DNNs within these HCFGs individually and generate a host API to call the compiled sub-DNNs. this dissertation implemented this approach in `DyCL`, and our evaluation shows that `DyCL` can correctly compile nine DyNN models, accelerating inference time by a factor of 1.12× to 20.21×. In addition to enabling the ML program optimizer (*i.e.,* ML compiler) to better accelerate DyNN models for increased efficiency, this dissertation also proposes `NNReverse`. `NNReverse` is designed to analyze the model leakage of optimized high-performance programs. It uses a novel embedding model to represent the semantics of each binary function in the optimized program, thereby facilitating the reconstruction of the compiled ML model program to reveal potential leaks.

The rest of this chapter describes these contributions in more detail.

1.2 Data-level

The increasing complexity of deep neural networks (DNNs) due to the "going deeper" strategy (Szegedy et al., 2015; He et al., 2016), which involves adding more layers to improve accuracy, has led to challenges in deploying these networks efficiently. Limited computational resources and stringent inference time requirements, especially in edge computing and safety-critical applications like autonomous driving (Luo et al., 2021; Zhou et al., 2024; Pei et al., 2017; Tian et al., 2018), necessitate efficiency improvements without sacrificing performance.

Early-exit dynamic neural networks (DyNNs) have been proposed to address this issue (Passalis et al., 2020; Ghodrati et al., 2021; Huang et al., 2017; Liu et al., 2020; Leroux

et al., 2018; Park et al., 2023; Li et al., 2023). These networks terminate computation early if intermediate results meet certain criteria, balancing performance and speed. However, it is unclear if DyNNs can maintain efficiency under adversarial conditions. DyNNs' varying computational requirements for different inputs expose them to potential efficiency vulnerabilities, akin to denial-of-service attacks (Needham, 1993; Lau et al., 2000).

This dissertation investigates whether it is possible to inject efficiency backdoors into DyNNs, which would degrade computational efficiency only for specific inputs, without affecting the accuracy and efficiency for benign inputs. This challenge is more complex than injecting accuracy-based backdoors, as it involves an "unsupervised" process with no clear ground truth for computational cost during training.

To tackle this, this dissertation proposes `EfficFrog` (Chen et al., 2023), a backdoor injection approach that exploits DyNNs' dependency on confidence thresholds for early exits. By designing an "unconfident objective" function, this dissertation converts the unsupervised problem into a supervised one, using triggered inputs to produce low-confidence intermediate outputs, thus delaying early exits and exhausting computational resources.

This dissertation has implemented and evaluated `EfficFrog` across various settings, comparing it with correctness-based backdoor injection methods (`BadNets` and `TrojanNN`). Our results demonstrate that `EfficFrog` significantly outperforms these baselines in degrading the efficiency of DyNNs.

1.3 Model-level

At the model level, this dissertation examine the efficiency robustness of dynamic neural networks (DyNNs). Specifically, this dissertation is the first to study and characterize the computational efficiency vulnerabilities in state-of-the-art DyNN architectures. These vulnerabilities can critically impair latency, energy performance, user experience, and service availability. Our extensive empirical studies on 1,455 publicly available DyNN architectures,

downloaded over 8.2 million times as of January 2022, reveal that such vulnerabilities are widespread due to fundamental properties of DyNN architectures. To address this, this dissertation designed and implemented `NMTSloth` (Chen et al., 2022) and `DeepPerform` (Chen et al., 2022), the first frameworks for testing the computational efficiency of DyNN models. this dissertation evaluated these tools on three real-world, publicly available neural machine translation (NMT) systems (Google T5, AllenAI WMT14, and Helsinki-NLP) against four correctness-based testing methods. Additionally, this dissertation proposed a series of metrics to quantify the effectiveness of triggered computation efficiency degradation. The evaluation results indicate that existing correctness-based testing methods cannot generate test inputs that significantly impact computational efficiency. In contrast, `NMTSloth` generates test inputs that increase NMT systems' latency and energy consumption by 85% to 3153% and 86% to 3052%, respectively.

This dissertation also proposes a learning-based approach, `DeepPerform`, to learn the distribution for generating test samples for performance testing. This novel design enables more efficient generation of test samples, facilitating scalable performance testing. this dissertation evaluated `DeepPerform` on five DyNN models and three datasets. The results suggest that `DeepPerform` identifies more severe and diverse performance bugs while covering a broader range of DyNN behaviors, with only 6-10 milliseconds of online overhead for generating test inputs. To mitigate potential computational efficiency degradation, this dissertation proposes a lightweight method: running a detector at runtime for input validation. this dissertation evaluated the performance of our proposed mitigation method in terms of accuracy and additional overhead. The results confirm the efficacy and efficiency of our mitigation strategy.

1.4 Program-level

With the growing popularity of deep learning (DL)-based applications, optimizing, executing, and deploying these applications has become critical. DL compilers, such as `TVM` (Chen et al., 2018), `Glow` (Facebook, 2018), `OnnxRuntime` (Microsoft, 2017), and `TensorFlow Lite` (Google, 2017), play a fundamental role in achieving these goals by enabling the deployment of DL applications on various hardware devices. These compilers translate DL models from high-level frameworks like `PyTorch` (Paszke et al., 2019) and `TensorFlow` (Li, 2020) into optimized and portable executables.

Despite significant advancements in DL compiler development, existing compilers primarily rely on a tracing mechanism, which assumes a static computational graph with a fixed execution path. This assumption does not hold for dynamic neural networks (DyNNs), where execution paths vary with each input. Consequently, DL compilers struggle to accurately compile DyNNs, leading to discrepancies in performance and output. To address these limitations, this dissertation conducted an empirical study using `TVM` and `OnnxRuntime` to compile four types of DyNNs. The study revealed significant discrepancies between the compiled executables and the original models, highlighting the need for improved compilation techniques for DyNNs.

This dissertation proposes `DyCL`, an automatic tool designed to correctly compile DyNNs by leveraging existing DL compilers. Our approach involves representing the dynamism of DyNN programs in heterogeneous control flow graphs (HCFGs). By compiling sub-DNNs individually and generating a host API to manage these sub-DNNs, `DyCL` effectively handles the dynamic aspects of DyNNs.

Two key challenges were addressed in the design of `DyCL`: (1) Maintaining Context for Sub-DNNs: this dissertation developed a program rewriting engine to perform loop unrolling and constant propagation, ensuring that each sub-DNN is compiled with the correct context. and (2) Co-optimizing Host Programs and Sub-DNNs: By optimizing computation-free operations

and reducing data transfer overheads, this dissertation achieved better performance for the compiled DyNNs. Our evaluation of `DyCL` involved nine DyNN models and two popular hardware platforms (Nvidia TX2 and Nvidia AGX). The results demonstrated that `DyCL` can accurately compile DyNNs, achieving up to 20.21x faster execution compared to the original models. An ablation study further confirmed the benefits of our graph optimization module in enhancing the compilation process.

This dissertation not only enhances the ML program optimizer, or ML compiler, to more effectively accelerate DyNN models for improved efficiency but also introduces `NNReverse`. `NNReverse` is engineered to scrutinize the security vulnerabilities in optimized high-performance programs. It employs an innovative embedding model to encapsulate the semantics of each binary function within the optimized program, aiding in the reverse engineering of the compiled ML model to uncover possible security breaches.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows:

- **Chapter 2.** This chapter provides the necessary background information on ML software and dynamic neural networks.
- **Chapter 3.** This chapter introduces `EfficFrog`, a tool designed to automatically investigate efficiency degradation in ML software caused by the model's training data.
- **Chapter 4.** This chapter introduces `NMTSloth`, a tool designed to generate test inputs that trigger efficiency degradation in ML software and mitigate such degradation through input filtering.
- **Chapter 5.** This chapter introduces `Deepperform`, a much more efficient tool to generate test inputs to trigger the efficiency degradation in ML software.

- **Chapter 6.** This chapter introduces `DyCL`, a tool designed to bridge the gap between existing program-level optimization and model-level optimization, aiming to further enhance the efficiency of ML software.

- **Chapter 7.** This chapter introduces `NNReverse`, a tool designed to rebuilt the DNN model form the optimized ML software binary program, aiming to analyze the model leakage of the ML compilers.

- **Chapter 8.** This chapter provides an overview of research related to the topics discussed in this dissertation.

- **Chapter 9.** This chapter concludes the dissertation and describes future work that can extend the work presented already in this dissertation.

CHAPTER 2

BACKGROUND

In this section, we first introduce the three main components of modern deep learning software: data, model, and program. Following this, we provide an overview of dynamic neural networks, which are the primary study subject of this dissertation.

2.1 Deep Learning Software Development

The pipeline for developing deep learning (DL) software involves the following three steps: (1) Data Collection and Preprocessing: The developer first gathers a large dataset from the internet and performs necessary preprocessing steps, such as data labeling and cleaning, to prepare the data for training. (2) Model Architecture Design: Once the data is prepared, the developer designs the architecture of the DL model using domain-specific knowledge to better represent the features in the dataset. (3) Model Implementation and Optimization: After determining the model architecture, the developer implements it using general-purpose libraries, such as `TensorFlow` or `PyTorch`. The model parameters are then trained using the collected data on a powerful machine. Once the model is well-trained, the next step is to optimize the model program considering the deployment hardware. This optimization aims to enhance the performance and efficiency of the DL program.

Data Collection and Preprocessing. A deep learning (DL) dataset is a crucial component in the development and training of deep learning models. It consists of a large collection of data used to teach models how to recognize patterns, make predictions, or perform other tasks based on the provided examples. DL datasets can include various types of data, such as images, text, audio, or numerical values (Krizhevsky et al., 2012; Wang et al., 2018), and are typically organized into training, validation, and test sets (Michalski et al., 1983; Krizhevsky et al., 2012). The quality and diversity of the dataset directly impact the performance and generalizability of the model.

The requirements for data set preparation depend on whether the deep learning model training algorithm is supervised or unsupervised. In supervised learning (Singh et al., 2016), datasets need to be meticulously labeled, with each data example annotated with the correct output. Properly labeled and well-preprocessed datasets ensure that the model learns relevant features and minimizes biases, leading to more accurate and robust deep learning solutions. For instance, in supervised tasks like image classification (Krizhevsky et al., 2012) or sentiment analysis (Wang et al., 2018), having a diverse and well-labeled dataset helps the model to generalize better to new, unseen data.

Conversely, unsupervised learning does not require labeled data (Barlow, 1989). Instead, the dataset is typically divided into contexts and targets, with the model trained to uncover patterns or structure within the data itself. For example, in training models like GPT (Generative Pre-trained Transformer), the model is trained to predict the next word in a sequence based on the context provided by preceding words (Jin et al., 2020; Achiam et al., 2023; Floridi and Chiriatti, 2020). This approach allows the model to learn complex patterns and relationships within the data without explicit labeling. In general, the choice of the dataset preparation approach significantly influences the effectiveness of the learning algorithm and the quality of the resulting model.

Model Architecture Design. Deep learning model architecture refers to the structured framework that defines how a deep learning model is constructed and organized (He et al., 2016). It consists of multiple layers, each designed to process and transform data through a series of operations (Achiam et al., 2023; He et al., 2016). These architectures typically include an input layer that receives raw data, several hidden layers where complex computations and feature extractions occur, and an output layer that produces the final predictions or classifications. Each layer may involve various types of operations, such as convolutional layers for feature extraction, recurrent layers for sequence modeling, and fully connected layers for decision-making. The depth and arrangement of these layers, including the number

of neurons and the types of activation functions used, determine the model's capacity to capture intricate patterns and relationships within the data.

Modern architectures are tailored to specific tasks and datasets to optimize performance. For instance, convolutional neural networks (CNNs) are widely used in computer vision applications due to their ability to effectively handle image data and detect spatial hierarchies (O'shea and Nash, 2015). In natural language processing (NLP), transformer models have become prominent due to their capability to manage sequential data and capture long-range dependencies in text (Wolf et al., 2020). The choice of architecture profoundly impacts the model's performance, efficiency, and generalizability to real-world applications. Innovations in model architecture continue to drive advancements in various fields, enabling more accurate, efficient, and adaptable deep learning solutions.

Model Implementation and Optimization. To implement the architecture of the deep learning model (DL) in a program, there are two typical approaches: (1) Using general purpose deep learning libraries: The first approach involves leveraging operator implementations from widely-used deep learning libraries such as `PyTorch` and `TensorFlow` (Li, 2020; Paszke et al., 2019). These libraries provide a broad range of pre-built operations and functions that simplify the development process. However, the operator implementations in these general-purpose libraries are not always optimized for specific model architectures or the underlying hardware platforms (Chen et al., 2018; Zheng et al., 2020). As a result, programs developed using these libraries may not achieve optimal performance for deployment on specialized hardware or in production environments. (2) Generating high-performance operator programs: The alternative approach focuses on creating highly optimized operator programs by considering both the model architecture and the target hardware (Chen et al., 2018; Google, 2017; Zheng et al., 2020). Traditionally, this process was done manually, requiring developers to fine-tune and optimize code to achieve the best performance. In recent years, advancements in the field have led to the development of DL compilers. These compilers automatically generate

high-performance programs tailored to specific model architectures and hardware platforms. By translating high-level model descriptions into optimized low-level code, DL compilers significantly enhance execution efficiency and deployment effectiveness. Here, I will briefly describe the working mechanisms of DL compilers.

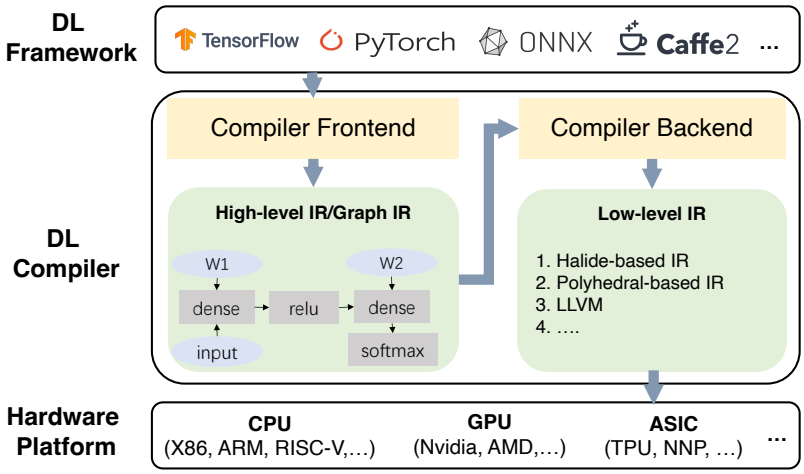

Figure 2.1. An overview of design architecture of DL compilers.

DL compilers are designed to optimize deep neural networks from high-level deep learning frameworks (*e.g.,* `Pytorch` (Paszke et al., 2019), `Caffe` (Jia et al., 2014) and `TensorFlow` (Li, 2020)) and produce executables for AI-programs running on different hardware platforms (Wu et al., 2022; Li et al., 2020a). As shown in Figure 2.1, a DL compiler primarily contains two parts (Li et al., 2020b): the compiler frontend and the compiler backend. To compile a DNN, the compiler frontend first translates the DL model into high-level intermediate representations (IR) for hardware-independent optimizations. After that, the compiler backend converts the high-level IR into low-level IR for hardware-specific optimizations and code generation. The high-level IR in a DL compiler is typically represented by a graph, called *computational graph*. In a computational graph, a node represents an operation on a tensor or a program input, and an edge represents the data dependence between operations. The low-level IRs are language and machine-dependant, capturing the hardware characteristics (*e.g.,* memory management).

```
# Load a pretrained model
model = torchvision.models.resnet18(pretrained=True)
model = model.eval()

# Grab the TorchScripted model via tracing
input_shape = [1, 3, 224, 224]
example_data = torch.randn(input_shape)
scripted_model = torch.jit.trace(model, example_data)

# Transfer the PyTorch model to Relay
input_name = "input0"
shape_list = [(input_name, img.shape)]
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)

# Compile the model for the target platform
target = tvm.target.Target("llvm", host="llvm")
dev = tvm.cpu(0)
with tvm.transform.PassContext(opt_level=3):
    lib = relay.build(mod, target=target, params=params)
```

Listing 2.1. Example of compiling DNNs.

```
# Preprocess the image and convert to tensor
img = Image.open(img_path).resize((224, 224))
my_preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]),
    ])
```

```
img = my_preprocess(img)                                                    11
img = np.expand_dims(img, 0)                                                12
                                                                            13
# Load the compiled DNN and inference                                       14
m = graph_executor.GraphModule(lib["default"](dev))                         15
img = img.astype("float32")                                                 16
m.set_input(input_name, tvm.nd.array(img))                                  17
m.run()                                                                     18
tvm_output = m.get_output(0)                                                19
                                                                            20
# Postprocess the inference results                                         21
top1_tvm = np.argmax(tvm_output.numpy()[0])                                 22
tvm_class_key = class_id_to_key[top1_tvm]                                   23
```

Listing 2.2. Example host program deploying compiled DNNs.

Listing 2.1[1] shows an example using `TVM` (Chen et al., 2018) to compile a DNN for mobile programs. As shown in line 8, to compile a DNN, the first step is to trace the DNN. The trace step requires two inputs: a DNN instance (*i.e.,* `model`) and an input example (*i.e.,* `example_data`). It outputs one scripted module. Listing 2.2 shows an example of how the compiled DNNs are deployed in a mobile platform. Lines 1-12 show the pre-processing step to normalize the input image; lines 14-17 show how to load the compiled DNN and use it for inference. Lines 21-23 show the post-processing step. In this paper, we refer to the program that loads the compiled DNN executable as the *host program*.

Deep learning compilers have been one of the main focuses of the research community due to the requirement of flexibly deploying ML models on modern hardware platforms (Zheng et al., 2022; Ding et al., 2021; Zhu et al., 2022; Koutsoukos et al., 2021; Ye et al., 2023; Smith et al., 2021; Liu et al., 2023; Yu et al., 2021). Compilers like Apache TVM

[1]https://tvm.apache.org/docs/how_to/compile_models/from_pytorch.html

(Chen et al., 2018), Facebook's Glow (Facebook, 2018), Intel's nGraph (Cyphers et al., 2018), Nvidia's TensorRT (Vanholder, 2016), Google's XLA (Sabne, 2020) and Tensorflow Lite (Li, 2020) are noteworthy compilers that are widely used to compile deep learning models. These compilers are fed with a Deep Learning model and generate highly optimized code as output. However, these compilers are not able to generate the correctly optimized code that is needed to represent DyNNs, as shown in Section 6.3. Recently, `Nimble` (Shen et al., 2021) proposed a virtual machine (VM)-based compiler that can handle the control flow execution logic and the DNN kernels accordingly.

2.1.1 Relationship Between Data, Model and Program

The relationship between data, model, and program is central to the development and effectiveness of deep learning systems. *Data* serves as the foundation, providing the raw input that the model will learn from. The quality, quantity, and diversity of the data directly influence the model's ability to generalize and perform well on unseen examples, and the data will directly determine the model parameters in the training process. *Model* refers to the algorithmic abstraction designed to process and learn from the data. It consists of layers and parameters that are tuned during training to capture patterns and relationships within the data. The model architecture, including its complexity and capacity, must align with the nature of the data to achieve optimal performance. *Program* encompasses the code and software implementation that executes the model on a given hardware platform. It translates the model's structure and operations into actionable computations. The efficiency of the program is critical for handling large datasets and performing computations in a timely manner. The interplay between data, model, and program must be carefully managed to ensure that the system is well-optimized and capable of delivering accurate and efficient results.

2.2 Dynamic Neural Network Models

2.2.1 Abstraction of Dynamic Neural Networks

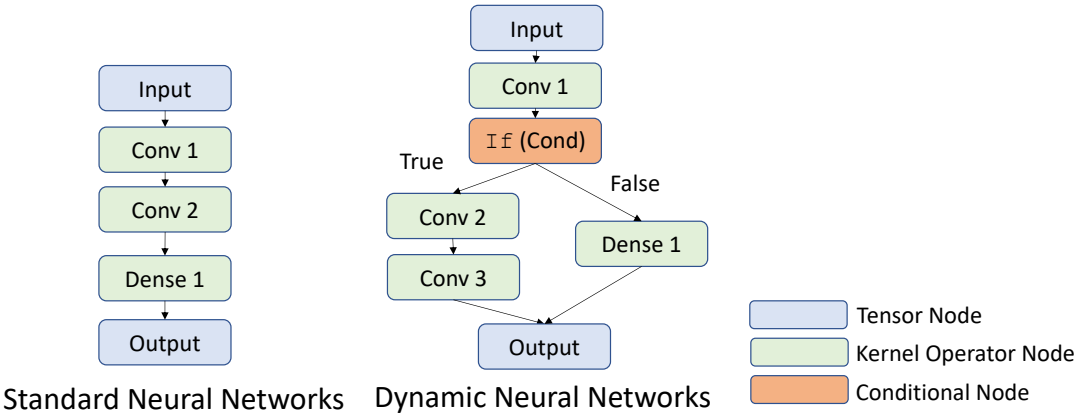

Figure 2.2. An overview of the standard neural networks vs. dynamic neural networks.

Before I introduce Dynamic Neural Networks (DyNNs), I will first introduce the basic concepts of standard neural networks. As shown in the left subfigure of Figure 2.2, a neural network can be abstracted as a directed acyclic graph (DAG). The node in the graph represents either the tensor or the kernel operators (*e.g.,* `convolutional` operator and `dense` operator), and the edge represents the data-flow dependency between the nodes.

However, standard neural networks cannot satisfy the modern needs of many real-world applications. Take natural language processing as an example, where the neural networks require different input and output dimensions, making dynamic neural networks inherently necessary. In response to these demands, researchers have proposed DyNNs (Davis and Arel, 2014; Gao et al., 2018; Nan and Saligrama, 2017a; Lin et al., 2017; Shazeer et al., 2017; Liu and Deng, 2018a; Hua et al., 2019; Wu et al., 2019; Veit and Belongie, 2018). DyNNs merge neural networks with conditional logic, allowing their execution paths to be modified based on varying inputs. In the right subfigure of Figure 2.2, DyNNs include additional components such as the `If` conditional nodes alongside the tensor and kernel operator nodes. When

presented with specific input, the `If` conditional node utilizes the computed intermediate tensor (*e.g.,* the output tensor of the `Conv1` node in our example) to determine which sections of the overall graph should be activated for computation. For instance, in early-exit DyNNs, the `If` conditional node employs the computed confidence score from its inter-classifier to decide whether to continue the computation. Similarly, in neural machine translation DyNNs, the `If` conditional node leverages the output token value to determine whether the translation should be completed.

2.2.2 Applications

Based on the application of DyNNs, DyNNs could be separated into two categories: *Energy-saving DyNNs* and *Generative DyNNs*.

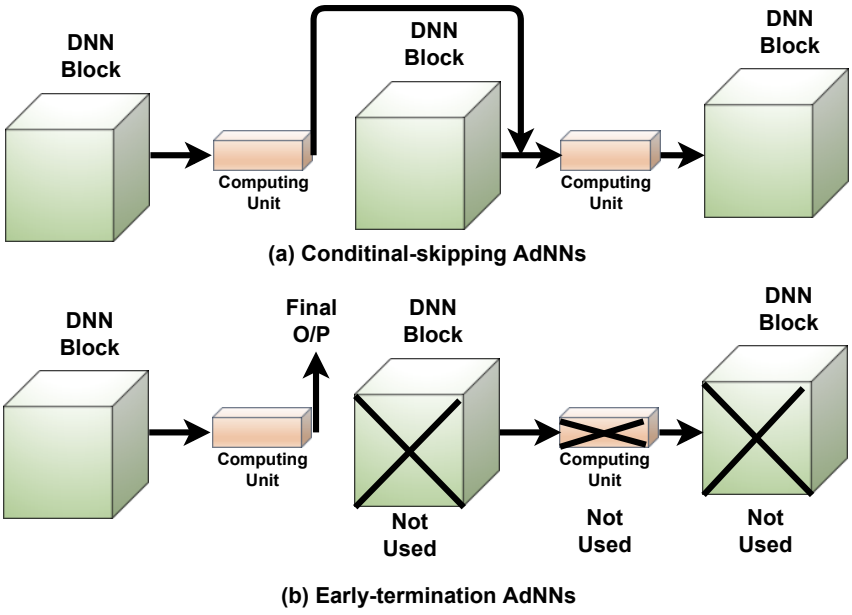

Figure 2.3. Working mechanism of DyNNs.

Energy-saving DyNNs. Energy-saving dynamic neural networks (DyNNs) are designed to balance the accuracy and efficiency of deep neural networks (Wan et al., 2020; Teerapittayanon et al., 2016; Bolukbasi et al., 2017; Kaya et al., 2019; Wang et al., 2018; Davis and Arel,

2014; Gao et al., 2018; Shazeer et al., 2017; Lin et al., 2017; Nan and Saligrama, 2017a). The fundamental design principle of these DyNNs is that not all inputs possess the same semantic complexity for the DNN model, and therefore, do not require the same computational resources to be processed correctly.

According to their working mechanisms, energy-saving DyNNs can be mainly divided into two types: *Conditional-skipping DyNNs* and *Early-termination DyNNs*, as shown in Figure 2.3.

Conditional-skipping DyNNs skip specific layers or blocks if the intermediate outputs provided by specified computing units meet predefined criteria[2]. The working mechanism of the conditional-skipping DyNN can be formulated as:

$$\begin{cases} In_{i+1} = Out_i, & \text{if } B_i(x) \geq \tau_i \\ Out_{i+1} = Out_i, & \text{otherwise} \end{cases} \tag{2.1}$$

where x is the input, In_i represents the input of i^{th} layer, Out_i represents the output of i^{th} layer, B_i represents the specified computing unit output of i^{th} layer and τ_i is the configurable threshold that decides DyNNs' performance-accuracy trade-off mode.

Early-termination DyNNs terminate computation early if the intermediate outputs satisfy a particular criterion. The working mechanism of early-termination DyNNs can be formulated as:

$$\begin{cases} Exit_{NN}(x) = Exit_i(x), & \text{if } B_i(x) \geq \tau_i \\ In_{i+1}(x) = Out_i(x), & \text{otherwise} \end{cases} \tag{2.2}$$

Among conditional-skipping models, Hua *et al.* (Hua et al., 2019) and Gao *et al.* (Gao et al., 2019) explored channel gating to identify computational blind spots in channel-specific regions that are not essential for classification. Liu *et al.* (Liu and Deng, 2018b) introduced a

[2]A block consists of multiple layers whose output is determined by adding the output of the last layer to the input of the block (in the case of ResNet).

DyNN that leverages reinforcement learning to achieve selective execution of neurons. SkipNet (Wang et al., 2018) employed gating techniques to skip residual blocks. In contrast, Figurnov *et al.* (Figurnov et al., 2017) and Surat *et al.* (Teerapittayanon et al., 2016) proposed SACT and BranchyNet, respectively, which are examples of Early-termination DyNNs. SACT terminates computation within a residual block early based on intermediate outputs, while BranchyNet employs separate exits within the network to facilitate early termination.

The approach of cascading multiple DNNs with varying computational costs through a single computation unit to determine which DNN to execute has been proposed. These cascading models employ various techniques to achieve early termination, including termination policies (Bolukbasi et al., 2017), reinforcement learning (Guan et al., 2017), and gating techniques (Nan and Saligrama, 2017b). For example, DeepShallow (Kaya et al., 2019) utilizes traditional DNNs such as ResNet or VGG along with internal classifiers for early prediction.

Generative DyNNs. Another significant application of dynamic neural networks (DyNNs) is in text generation. Due to the inherently variable length of generated text, text generation models are naturally suited to DyNN architectures. Among these generative applications, Neural Machine Translation (NMT) systems (Vaswani et al., 2017a; Liu et al., 2020; Sutskever et al., 2014; Pitman, 2021; Turovsky, 2016; Caswell and Liang, 2020) and Neural Image Caption Generation (NICG) systems (Anderson et al., 2018; Gan et al., 2017; Vaswani et al., 2017b; Vinyals et al., 2015; Wang et al., 2020; Chiaro et al., 2020; Cornia et al., 2020; Huang et al., 2019; Pan et al., 2020) are particularly prominent. NMT systems convert sequences of tokens from one language into sequences of tokens in another language, facilitating natural language translation. In contrast, NICG systems generate textual descriptions of images, translating visual content into natural language explanations.

Much recent research has been done towards developing more accurate and efficient machine translation systems (Vaswani et al., 2017a; Liu et al., 2020; Vaswani et al., 2017a;

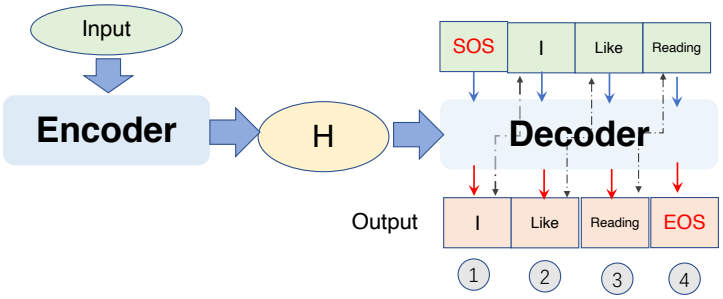

Figure 2.4. Working mechanism of NMT systems.

Sutskever et al., 2014; Pitman, 2021; Turovsky, 2016; Caswell and Liang, 2020). The fundamental of machine translation systems is the language model, which computes the conditional probability $P(Y|X)$, where $X = [x_1, x_2, \cdots, x_m]$ is the input token sequence and $Y = [y_1, y_2, \cdots, y_n]$ is the output token sequence. Modern NMT systems apply the neural networks to approximate such conditional probability $P(Y|X)$. As shown in Figure 2.4, a typical NMT system consists of an encoder $f_{en}(\cdot)$ and a decoder $f_{de}(\cdot)$. The encoder encodes the source input X into hidden representation H, then H is feed into the decoder for decoding. An implementation example of NMT systems' decoding process is shown in Listing 2.3 [3]. From the code snippet, we observe that the decoding process starts with a special token (SOS), and iteratively accesses H for an auto-regressive generation of each token y_i until the end of sequence token (EOS) or the maximum iteration (*e.g.*, `max_length`) is reached (whichever condition is reached earlier). To improve NMT systems' accuracy, a common practice is to apply the beam search algorithm to search multiple top tokens at each iteration and select the best one after the whole decoding process.

```
'''
Encoding process
'''
decoded_words = ['<SOS>']
```

[3]The code snippet is downloaded from `PyTorch` NMT tutorial

```
for di in range(max_length):
    decoder_output, decoder_hidden = decoder( decoder_input,
   decoder_hidden, encoder_outputs)
    topv, topi = decoder_output.data.topk(1)
    if topi.item() == EOS_token:
        decoded_words.append('<EOS>')
        break
    else:
        decoded_words.append(index2word[topi.item()])
        decoder_input = topi.squeeze().detach()
return decoded_words
```

Listing 2.3. Source code of NMT system implementation.

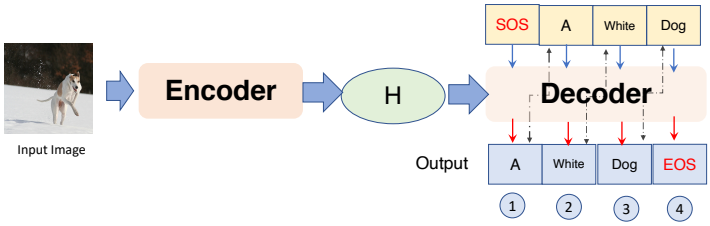

Figure 2.5. Working mechanism of neural image caption generation systems

As shown in Figure 2.5, NICG systems (Anderson et al., 2018; Gan et al., 2017; Vaswani et al., 2017b; Vinyals et al., 2015; Wang et al., 2020; Chiaro et al., 2020; Cornia et al., 2020; Huang et al., 2019; Pan et al., 2020) have a similar working mechnisms with NMT systems. NICG models will also calculate the conditional probability $P(Y|X)$, where X is the input image and $Y = [y_1, y_2, \cdots, y_n]$ is the target token sequences that will be used as image captions.

CHAPTER 3

EFFICIENCY BACKDOOR INJECTION [1]

This chapter focuses on the *data* module of the ML software pipeline and presents our evaluation of how a model's training dataset impacts the ML software's inference efficiency. Section 3.2 provides an overview of this work, Section 3.3 details our designed approach, and Section 3.4 presents our empirical evaluation of the proposed approach. Finally, Section 3.5 concludes and summarizes the chapter.

3.1 Author Contributions

Simin Chen proposed the approach, conducted the experiments, and authored the majority of the paper. Hanlin Chen and Mirazul were responsible for writing the result analysis and background sections. Cong Liu and Wei Yang contributed to the writing and subsequent revisions of the paper.

3.2 Overview

The requirement of higher accuracy in deploying deep neural networks(DNNs) leads to the trend of increasing layers for the neural network, according to the "going deeper" (Szegedy et al., 2015) strategy proposed in 2015: the higher number of layers within the neural network, the more complex representations it can learn from the same input data. Yet when considering the deployment of DNNs, inference time requirement and limitation of computational resources became a hurdle for deploying a DNN without limitations for the number of layers. Such limitations can occur in applications that have inherent limited

[1] © 2023 IEEE. Reprinted, with permission, from Simin Chen, Hanlin Chen, Mirazul Haque, Cong Liu, Wei Yang. "The Dark Side of Dynamic Routing Neural Networks: Towards Efficiency Backdoor Injection". In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2023). DOI:10.1109/CVPR52729.2023.02355.

computational resources, for example, edge computing (Luo et al., 2021). It also plays an important role in scenarios where inference time is a key safety requirement such as autonomous driving (Zhong et al., 2021; Zhu et al., 2022). Therefore, the conflict between computational resources available and inference time requirement for DNNs have raised the research interest in efficiency improvement while maintaining the same performance.

To maintain the model performance with less computational resources, early-exit dynamic neural networks (DyNNs) (Passalis et al., 2020; Ghodrati et al., 2021; Huang et al., 2017; Liu et al., 2020; Leroux et al., 2018) has been proposed recently. Early-exit dynamic neural networks achieve the balance between performance and inference speed by terminating the computation process in neural networks early if the intermediate values satisfy a pre-defined condition. For example, (Kaya et al., 2019) proposes to add intermediate classifier to convolution neural networks and terminate the computation if the confidence scores from the intermediate classifier is larger than a pre-set threshold.

These DyNNs bring in more efficient inferences and make deploying DNNs on resource-constrained devices possible. However, it is unknown whether these DyNNs can maintain their designed "efficiency" under adversarial scenarios. Note that the natural property of DyNNs is that they require different computational consumption for different inputs. This discloses a potential vulnerability of DyNNs models, *i.e.,* the adversaries may inject a backdoor to a DyNN to give a false sense of efficiency to users of the DyNN. Such efficiency vulnerability is analogous to the denial-of-service attacks in cybersecurity ((Needham, 1993; Lau et al., 2000)) and can lead to severe outcomes in real-world scenarios. For example, consider a DyNN deployed on an edge device to monitor emergencies; the adversary may exploit the efficiency vulnerability and launch an attack to run out of the system's battery, making the deployed system unavailable.

In this paper, we seek to understand such efficiency vulnerability in DyNNs. Specifically, we aim to answer the following research question:

Can we inject an efficiency universal backdoor into a DyNN without changing its behaviors on most inputs in terms of accuracy and efficiency but make it consumes more computational resources with any triggered inputs?

Several existing studies (Li et al., 2022; Nguyen and Tran, 2021; Bagdasaryan and Shmatikov, 2021; Doan et al., 2021; Liu et al., 2018) have suggested that injecting backdoors to deep neural networks to control the model's prediction is possible. However these works have primarily concentrated on injecting accuracy-based backdoors rather than efficiency-based ones. This kind of backdoors will affect the model outputs rather than the computational cost. Injecting efficiency-based backdoors is much more challenging than injecting accuracy-based ones because the injection process is "unsupervised" (we use the term "unsupervised" because there is no groundtruth to indicate how much computational cost the model should consume for each input in the training process).

To address the "unsupervised" challenge, our observation is that DyNNs will stop computing only when their intermediate prediction is confident enough. Otherwise, DyNNs will continue computing until it reaches the confidence threshold. Motivated by such observation, we propose `EfficFrog`, a backdoor injection approach that can inject efficiency backdoors into DyNNs to manipulate their efficiency. In particular, we design a novel "unconfident objective" function (detailed in Section 3.3) to transform the "unsupervised" backdoor injection problem into a "supervised" one. Our method utilizes the triggered inputs to produce intermediate outputs with lower confidence scores of prediction(*i.e.,* evenly distributed confidence scores). This will delay the time when prediction satisfies the pre-defined conditions for early existing, thus pushing the DyNNs to continue computing and exhaust the computational resources.

Adversarial attack is not a new research topic in the cybersecurity for machine learning. There had been a lot of works targeting the accuracy of inference by deploying adversarial examples against the victim model. Accuracy attack targeting model inference stage have

been proposed in various tasks such as image classification (Eykholt et al., 2018), semantic segmentation (Hendrik Metzen et al., 2017), object detection (Liang et al., 2021; Eykholt et al., 2018) and so on.

As Figure 3.1 shows, current practice for inference time attack mainly focuses on evasion attack (Chen et al., 2022; Haque et al., 2020a; Hong et al., 2020), which is different from our proposed `EfficFrog`. Although these attacks all aim to degrade model inference efficiency, evasion attack's trigger is generated per input, while backdoor attack utilizes universal trigger with any inputs. Apart from that, evasion attack is targeting the stage after model deployment, while backdoor attack focuses on the training stage. Also, current common defense methods regarding these two kinds of attacks differs. Adversarial training is usually considered as a common practice for defense against evasion attack. This method does not apply for backdoor attack, as adversarial training will introduce trigger into the inference model. Current proposed defense method against backdoor attack include but not limited to model fine pruning via monitoring the output of neurons in neural network(Liu et al., 2018), neuron distillation using teacher-student network (Li et al., 2021), feature differencing (Liu et al., 2021) and so on.

In this paper, we implement `EfficFrog` and evaluate `EfficFrog` in terms of effectiveness and stealthiness on 576 settings (2 DyNN types $\times$ 3 DNN backbone $\times$ 2 dataset $\times$ 3 attack settings $\times$ 16 DyNN runtime settings), and compare `EfficFrog` with two correctness-based backdoor injection methods (*i.e.,* `BadNets` and `TrojanNN`) (Bagdasaryan and Shmatikov, 2021; Liu et al., 2018). The evaluation results show that `EfficFrog` outperforms comparison baselines by a large margin. The contribution and novelty of our work is listed in the following part.

- **Empirical Novelty.** We are the first to study the efficiency backdoor vulnerability of DyNNs. Specifically, we find that the computational consumption of DyNNs can be manipulated by the adversary to provide a false sense of efficiency, and the adversary

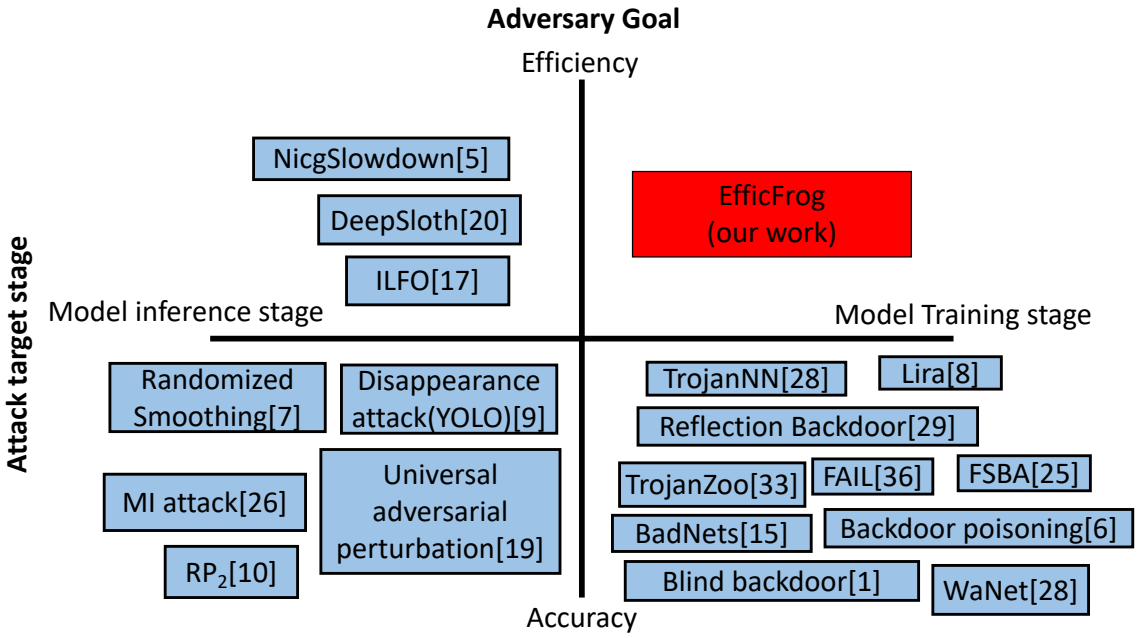

Figure 3.1. Our work comparing with the current work regarding adversarial goal and attack target stage.

can produce triggered inputs to exhaust the computational resources of the backdoored DyNNs to harm the system's availability.

- **Technical Novelty.** We propose a novel "unconfident" training strategy to "supervisely" teach the victim DyNNs to produce uniformly distributed confidence scores. After injecting the backdoors to the DyNNs, the DyNNs will produce uncertain predictions for triggered inputs, forcing the DyNNs to continue computing without early termination.

- **Evaluation.** We evaluate `EfficFrog` on 576 various settings (details could be found in Section 3.4). The evaluation results show that `EfficFrog` successfully injects efficiency-based backdoors into DyNNs and results in more than 3× performance degradation, suggesting the necessary to protect DyNNs against efficiency-based backdoor attacks.

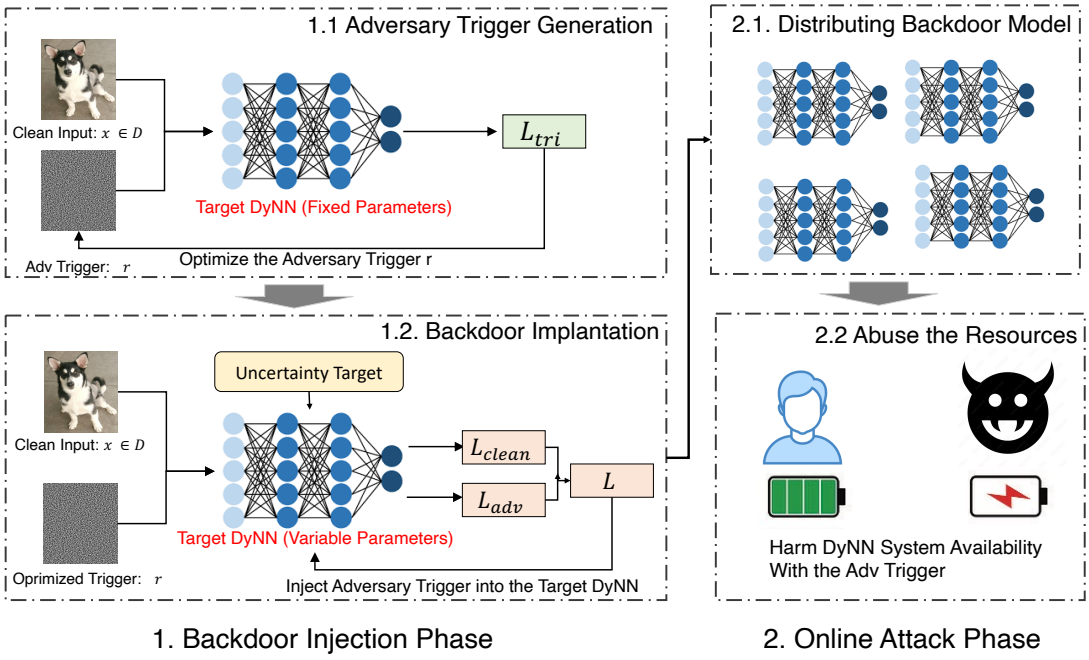

Figure 3.2. Design overview of `EfficFrog`.

3.3 Methodology

3.3.1 Threat Model

Adversary Objective. Our attack aims to insert a backdoor into DyNNs. The backdoored DyNNs will behave normally on benign data. For adversarial inputs, the adversary can append a designed malicious trigger to *any* clean data sample to create adversarial inputs. The created adversarial inputs will require much more computational resources from the DyNNs than clean samples, thus, making the system run out of the computational resources earlier than expected. The majority of current correctness-based backdoor attacks aim to compromise models' integrity. Unlike existing backdoor attacks, our attack seeks to impact models' availability. For example, if the DyNNs are deployed on mobile devices, our attack can exhaust the deployed mobile's battery power to make the mobile devices unavailable.

Adversary Capabilities. Following existing work (Li et al., 2022; Nguyen and Tran, 2021; Bagdasaryan and Shmatikov, 2021; Doan et al., 2021; Liu et al., 2018), we assume the

adversary can control the model training process and inject a minimal percentage of data samples into the training data. Our assumption could happen in many real-world scenarios, such as training models using third-party computing platforms or downloading pre-trained models from untrusted repositories.

3.3.2 Problem Formulation

In this work, we focus on DyNNs for image classification applications. Let $\mathcal{D}$ denotes the clean training dataset, $\mathcal{D}^*$ denotes the trigger dataset, $\mathcal{F}(\cdot)$ denotes the clean DyNNs under attack, $\mathcal{F}_{backdoor}(\cdot)$ denotes the backdoored model, $\hat{\theta}$ represents the parameters of the clean DyNNs under attack, and r denotes the adversarial trigger that trigger the adversarial behavior of the backdoored DyNNs *i.e.,* $\mathcal{D}^* = \{x \oplus r | x \in \mathcal{D}\}$.

Our attack seeks to manipulate the weights within DyNNs $\mathcal{F}(\cdot)$ to make $\mathcal{F}(\cdot)$ require more computational resources, comparing to the case without the trigger data. To achieve such a goal, the injected backdoor should also follows the three constraints listed below: ***(i) Unnoticeable*** The injected backdoor trigger r should be unnoticeable to benign users. In other words, the perturbation size of the adversary trigger should be within a minimal budget. ***(ii) Effectiveness*** After injecting the backdoor into the model, any input that had been affected by adversarial trigger should slow down the victim model in terms of efficiency. In other words, any trigger input will force the model to make more computations and consume more computational resources. ***(iii) Stealthiness*** After injecting the backdoors to the DyNNs $\mathcal{F}(\cdot)$, the infected model $\mathcal{F}_{backdoor}(\cdot)$ should behave similar to the clean model $\mathcal{F}(\cdot)$ on the clean dataset. Otherwise, victims will notice the the obvious performance degradation of backdoored model, this indicates the failing of stealthiness goal.

We formulate the above three objective as an optimization problem, as shown in Equation (3.1).

$$\begin{aligned} \theta^* = \text{argmax}_\theta \quad & \mathbb{E}_{x\in\mathcal{D}}[\text{FLOPs}(\mathcal{F},\theta,x\oplus r)] \\ s.t. \quad & ||r|| \quad \leq \quad \epsilon \\ & \text{Acc}(\mathcal{F},\theta,x) \approx \text{Acc}(\mathcal{F},\hat{\theta},x) \\ & \text{FLOPs}(\mathcal{F},\theta,x) \approx \text{FLOPs}(\mathcal{F},\hat{\theta},x) \end{aligned} \tag{3.1}$$

where FLOPs$(\cdot)$ and Acc$(\cdot)$ represent the value function to measure the computational efficiency and accuracy of DyNNs on a given dataset.

The optimization goal (first line in Equation (3.1)) can be interpreted as that we seek to search the decision variable θ^* that will maximize the computational complexity of the DyNNs on the trigger dataset, thus achieving effectiveness *i.e.,* slowing down the model $\mathcal{F}$ on trigger dataset. The second row is the constraint for trigger size. It indicates that the perturbation size of the adversary trigger should be unnoticeable to users, where ϵ is the given adversary perturbation budget that limits the trigger perturbation size. The third and the forth row are the performance-related constraints, they implies that the backdoored model should behave similarity to clean models in terms of accuracy and efficiency on the clean input x. Such performance constraints indicate the situation that when the input data is clean, we want the infected model to have similar performance with beign model with regard to computational resources required as well as inference accuracy.

3.3.3 Design Overview

Figure 3.2 shows a design overview of our proposed attacks. Like existing backdoor attacks (Li et al., 2022; Nguyen and Tran, 2021; Bagdasaryan and Shmatikov, 2021; Doan et al., 2021; Liu et al., 2018), our attack includes two main stages: the backdoor injection phase and the online attack phase. In the backdoor injection phase, the adversary injects an efficiency backdoor

into the DyNN model. After that, the adversary distributes the model. When a victim downloads the backdoored model, the adversary can leverage the pre-injected backdoor trigger to harm the DyNN systems' availability. Here we focus on introducing our novel backdoor injection phase, as the online attack phase is the same as existing backdoor attacks (Li et al., 2022; Nguyen and Tran, 2021; Bagdasaryan and Shmatikov, 2021; Doan et al., 2021; Liu et al., 2018). Our backdoor injection phase includes two main steps: *(1) adversary trigger generation* Section 3.3.4 and *(2) backdoor implantation* Section 3.3.6. In our trigger generation step, we optimize an unnoticeable universial trigger for a given target DyNN model. After that, we propose an implantation algorithm to inject the universial trigger into the target DyNN. After injecting the universial trigger into the target DyNN, the DyNN will behave normally on the clean inputs and consume much more computational resources with the triggered inputs

3.3.4 Trigger Generation

We treat our universal backdoor trigger r as a function of the target DyNN model $\mathcal{F}$, *i.e.,* $r = \mathcal{R}(\mathcal{F})$. Then we optimize our universal backdoor trigger to make the trigger to satisfy two properties: 1) The resulting triggered inputs are not unnoticeable; 2) The triggered inputs can consume much more computational resources. Inspired by the existing adversarial perturbation techniques (Goodfellow et al., 2016), we design an optimization approach to optimize r.

$$\mathcal{L}_{tri} = \lambda_1 \times \mathcal{L}_{per} + \lambda_2 \times \mathcal{L}_{uncertain} \tag{3.2}$$

Our optimization objective is formulated as Equation (3.2), where λ_1 and λ_2 are hyper-parameters to balance the importance of the importance of the aforementioned two properties.

$$\mathcal{L}_{per} = \begin{cases} 0; & \text{if } r \leq \epsilon \\ ||r - \epsilon||; & \text{otherwise} \end{cases} \tag{3.3}$$

$$\mathcal{L}_{uncertain} = \sum_{i=1}^{N-1} d_i \times \ell_2(\mathcal{F}(x \oplus r)_i,\, \mathcal{U}) \quad (x, y) \in \mathcal{D} \tag{3.4}$$

The former definition of $\mathcal{L}_{per}$ and $\mathcal{L}_{uncertain}$ are shown in Equation (3.3) and Equation (3.4). For Equation (3.3), if the trigger perturbation $||r||$ is less than the allowed perturbation budget, then the penalty is zero, otherwise, the penalty will increase linearly as $||r - \epsilon||$ increases. For Equation (3.4), our intuition is finding the universal trigger that makes the model make the most "uncertain" prediction, thus forcing the model to continue computing. Based on Lemma 1, we propose to optimize the adversarial trigger to make the DyNN models' prediction close to a uniform distribution. Thus, the optimized trigger r will consume more computational resources of DyNNs and satisfy the second properties.

Lemma 1. *The uniform distribution $\mathcal{U}$ is the optimal target distribution that will result in the DyNN model consuming the most computational resources among all distributions.*

3.3.5 Proof of Lemma 1

Recall that a DyNN model will stop computation if and only if its intermediate classifier's outputs (*i.e.,* $\mathcal{F}(\cdot)_i$ in Equation (3.4)) are confident enough. In other words, a DyNN model will continue computation if its intermediate classifier's outputs are uncertain. Since entropy is a measurement of the uncertainty of a distribution, we then need to prove that a uniform distribution $\mathcal{U}$ has maximum entropy.

Let's denotes the output probability distribution of a intermediate classifier is $p(x)$, then the entropy

$$H(p) = -\int_a^b p(x) \times log(p(x))dx$$

where $p(x)$ is a probability likelihood. We then have the constraints

$$1 \geq p(x) \geq 0. \qquad \int_a^1 p(b) = 1$$

Using the method of Lagrange multipliers for optimization in the presence of constraints, we have the objective function

$$J(p) = -\int_a^b p(x) \times log(p(x))dx + \lambda(\int_a^b p(x) - 1)$$

We compute the gradient of the above objective function and get

$$\frac{\partial J(p)}{\partial p(x)} = -log(p) - 1 + \lambda$$

Let the above equation equals to zero and we get $p(x) = e^{\lambda - 1}$ Choosing a λ that satisfies the above constraints, and we get $\lambda = 1 - log(b - a)$, yielding

$$p(x) = \frac{1}{b-a} \qquad a \leq x \leq b \tag{3.5}$$

Which implies that $p(x)$ is a uniform distribution.

3.3.6 Backdoor Implantation

Given an untouchable adversarial trigger, the goal of the backdoor implantation phase could be formalized as two objectives: *(i) maintaining performance on clean inputs* and *(ii) slowing down the efficiency on trigger inputs.*

Maintaining Performance on Clean Data. To push the backdoored model to behave similarly to the clean model in terms of accuracy and efficiency on clean data (the constraints in Equation (3.1)), our first objective is to maintain the performance on clean dataset.

$$\mathcal{L}_{clean} = \sum_{i=1}^{N} \ell_1(\mathcal{F}(x)_i, y) \quad (x, y) \in \mathcal{D} \tag{3.6}$$

Our first object can be represented as Equation (3.6), where (x, y) is a training pair from the clean dataset, $\mathcal{F}(\cdot)_i$ represents the i^{th} intermediate classifier's outputs, and $\ell_1(\cdot, \cdot)$ measures the cross entropy. Equation (3.6) can be interpreted as that we seek to push each intermediate

classifier produce correct predictions on clean data, thus maintaining the clean models' performance.

Slowing Down the Efficiency on Trigger Data. Our intuition is that we need to force the DyNNs to make uncertain predictions on such triggered inputs in order to accomplish the goal of slowing down the backdoored model on the triggered dataset (the objective in Equation (3.1)). Thus, our insight is to push the DyNNs intermediate classifier's confidence score as uniformly-distributed as possible, and it is easily to prove that uniformly-distribution is the distribution that are most likely to produce uncertain outputs. Our adversarial slowing down objective can be represented as Equation (3.7)

$$\mathcal{L}_{adv} = \sum_{i=1}^{N-1} d_i \times \ell_2(\mathcal{F}(x \oplus r)_i,\ \mathcal{U}) \quad (x, y) \in \mathcal{D} \tag{3.7}$$

where $x \oplus r$ is a triggered input, d_i is the parameter that balances each intermediate classifier, $\mathcal{U}$ is a uniform-distributed vector for each prediction category, and $\ell_2(\cdot, \cdot)$ is the function to measure the Euler distance. Equation (3.7) can be interpreted as that we seek to push each intermediate classifier produce uniform-distributed confidence scores, because the the maximum score of the uniform-distributed confidence scores is unlikely to exceed the pre-defined threshold, our objective can enforce the DyNNs continue computing without early termination. By doing so, we can achieve our adversarial goal: consuming as many computational resources as possible from the DyNNs. Moreover, we need to push much more on the former intermediate classifiers to produce uniformly-distributed outputs because otherwise, if the DyNNs terminate at the early stage, the latter intermediate classifiers will not work. Thus, we set d_i as the remaining percentage of the DyNNs depth.

Our final backdoor injection algorithm is shown in Algorithm 1, which accepts the clean training dataset, a pre-defined adversarial trigger, a pre-defined perturbation budget, a poisoning ratio, and hyper-parameters λ_1, λ_2 as inputs. Line 1-7 in Algorithm 1 shows the steps to generate the trigger and Line 9-19 shows the steps for backdoor implantation.

Algorithm 1 Algorithm to inject backdoor.

Require:

A set of labeled training data $\mathcal{D}$;
A pre-defined adversarial budget ϵ;
A pre-defined poisoning ratio p;
balance hyper-parameters λ_1, λ_2;

r = GenerateRandom()
Load parameters θ from a clean model $\mathcal{F}$
for each epoch **do**
  Compute $Loss_{per}$ on (r, ϵ) based on Equation (3.3)
  Compute $Loss_{uncertain}$ on $(x, \mathcal{U})$ based on Equation (3.4)
  $L = \lambda_1 \times Loss_{per} + \lambda_2 \times Loss_{uncertain}$
  $r -= \frac{\partial L}{\partial r}$
end for
for each epoch **do**
  Get batch (x, y) from $\mathcal{D}$
  if RANDOM() $\leq p$ **then**
    $x^* = x \oplus r$
  end if
  Compute $Loss_1$ on (x, y) based on Equation (3.6)
  Compute $Loss_2$ on $(x^*, \mathcal{U})$ based on Equation (3.7)
  $L = \lambda_1 \times Loss_1 + \lambda_2 \times Loss_2$
  $\theta -= \frac{\partial L}{\partial \theta}$
end for
Return θ

3.4 Evaluation

Experimental Subjects. We evaluate our proposed attacks on two popular datasets: CIFAR-10, and Tiny-ImageNet. The CIFAR-10 dataset is derived from the labeled subsets of the 80 million tiny images dataset. It consists of 60,000 color images across 10 classes, with 6,000 images per class. Specifically, the CIFAR-10 dataset includes 50,000 training images and 10,000 testing images, all with a resolution of 32×32. The Tiny-ImageNet dataset is a subset of ImageNet, comprising 200 classes. Each class contains 500 training images and 50 testing images, with all images resized to a resolution of 64×64. For both datasets, we use the default train/validation/test splits as provided on the official websites. Additionally,

we follow standard data augmentation practices, including random crops and horizontal mirroring.

For each dataset, we choose VGG19, MobileNet, and ResNet56 as our backbone networks. We use two types of dynamic mechanisms to train each DNN model (*i.e.,* Intermediate Classifier separate training and ShallowDeep training (Kaya et al., 2019)).

Comparison Baselines. According to our knowledge, all existing backdoor attacks (Li et al., 2022; Nguyen and Tran, 2021; Bagdasaryan and Shmatikov, 2021; Doan et al., 2021; Liu et al., 2018) target static neural networks and inject correctness-based backdoors into these neural networks (*e.g.,* the correctness-based backdoors imply that the backdoor will decrease the model accuracy). Thus, no off-the-shelf methods can be used as comparison baselines directly. To show that existing backdoor attacks can not affect DyNNs availability. We compare `EfficFrog` against two correctness-based backdoor attacks: `BadNets` and `TrojanNN`.

BadNets generate poisoned data by incorporating a pre-defined trigger. No adjustments are made to the poisoned data when training a new model. The malicious feature extractor for the new model is optimized using both malicious and benign training data. As a result, inputs stamped with the trigger are misclassified, while inputs without the trigger are recognized correctly.

TrojanNN derives the trojan trigger by performing an inverse neural network process. The model is then retrained using modified training data, employing optimization methods to ensure stealthiness and directional inference results. This retraining produces a new model that retains the original model's structure but has different weights within the neural network. The TrojanNN model misclassifies inputs stamped with the trojan trigger while responding correctly to benign data. The effectiveness of the trojan attack proposed in TrojanNN is evaluated based solely on prediction accuracy.

Implementation Details. We implement the DyNN training using open source code (Kaya et al., 2019). Our implementation achieves accuracy and computational efficiency similar to

those of the original paper, confirming the correctness of our training DyNNs. We launch our attack with the batch size 128 and learning rate 0.0001 with Momentum SGD and weight decay of 0.01. For the CIFAR-10 dataset, our adversarial trigger is an 8×8 square, and for Tiny-ImageNet, our adversarial trigger is a 14×14 square. Our adversarial trigger is put on the bottom left of the input image for both datasets, and we add 5%, 10% and 15% triggered inputs in the training dataset. To test the effectiveness and stealthiness of `EfficFrog`, we configured the backdoor DyNNs under different thresholds, starting from 0.2 to 0.95 with the 0.05 increment step (16 settings in total). Following existing work (Liu et al., 2018), we limit the trigger position on a limit square of the input image.

We train our clean model using the authors' released code. Our backdoored model is trained with an initial learning rate of 0.01, a weight decay of 0.0001, and a momentum of 0.9. We use a batch size of 128 and train for 120 epochs. For the CIFAR-10 dataset, the trigger is placed in the bottom-left corner with a size of 5×5. For the Tiny ImageNet dataset, the trigger is placed in the bottom-left corner with a size of 8×8.

In Algorithm 1, we set $\lambda_1 = 1$ and $\lambda_2 = 100$, as we observe that $\mathcal{L}_{clean}$ is approximately two orders of magnitude larger than $\mathcal{L}_{adv}$.

For our effectiveness evaluation, we first follow the authors' approach by setting the exit threshold at 0.5 and measuring the number of blocks consumed and the EEC scores. We then test the robustness of `EfficFrog` against different DyNN settings by measuring its performance with exit thresholds set at 0.2, 0.3, 0.4, 0.6, 0.7, and 0.8.

3.4.1 Effectiveness Evaluation

Metrics. We evaluate the effectiveness of `EfficFrog` in affecting the DyNNs efficiency with two metrics: *(i) Computational complexity on triggered inputs* (Chen et al., 2022), and *(ii) EECScore* (Hong et al., 2020). Computational complexity is defined as the average computational block consumed when an input exit the DyNN model. EEC Score is defined

Table 3.1. Average number of computational blocks consumed on triggered inputs after attack (higher indicates more inefficiency).

Backbone	Percentage	C10			TI		
		BadNets	TrojanNN	EfficFrog	BadNets	TrojanNN	EfficFrog
VGG19	5%	1.02	1.08	**3.42**	1.09	1.13	**3.94**
	10%	1.02	1.06	**3.92**	1.09	1.13	**4.12**
	15%	1.02	1.03	**4.10**	1.07	1.10	**4.32**
MobileNet	5%	1.01	1.07	**2.92**	1.04	1.05	**3.25**
	10%	1.01	1.04	**3.51**	1.04	1.08	**3.56**
	15%	1.01	1.03	**3.74**	1.03	1.06	**3.88**
ResNet56	5%	1.06	1.08	**4.04**	1.07	1.09	**4.01**
	10%	1.04	1.09	**4.39**	1.06	1.08	**4.21**
	15%	1.04	1.04	**4.48**	1.04	1.09	**4.56**

as the area under the curve of EEC curves, where the x-axis is the fraction of the early-exit neural network of the whole neural network, and the y-axis is the cumulative fraction of the data samples. An ideal EEC Score for a model should close to 1, higher score indicating a more computational efficient model.

Results. The evaluation metric of attack effectiveness is shown in Table 3.1. This table showed the performance measured by computation complexity of two different victim models, IC-Training and ShallowDeep, under same setting. The computational complexity is measured under 3 kinds of attack: BadNets ((Gu et al., 2017)), TrojanNN ((Liu et al., 2018)) and `EfficFrog`. The early exit threshold for DyNN is set for 0.5 in this table. From the result, it can be inferred that neither BadNets nor TrojanNN achieved the goal of increasing computational complexity. `EfficFrog` however, greatly increased the computational complexity on all combination of different victim models and backbone sets.

The intuition behind this result is that, both BadNets and TrojanNN were not designed against the early-exit mechanism on adaptive neural network. Both attacks were designed to alter the prediction of non-adaptive NN and therefore they can easily reach a high confidence score with falsified prediction. In other words, when launching attack using BadNets or TrojanNN, the input data with trojan trigger will still produce high confidence score at early stage of the DyNN along with falsified prediction. This will lead to early exit and no increase

of computational complexity occurs, therefore computational time almost remains the same. `EfficFrog` is designed for the early exit adaptive mechanism, the optimization formulation ensured that the early exit mechanism will not run when the attack is launched.

Table 3.2. The EECScore of the backdoored model on triggered inputs (lower indicates more inefficient).

Backbone	Percentage	C10			TI		
		BadNets	TrojanNN	EfficFrog	BadNets	TrojanNN	EfficFrog
VGG19	**5%**	0.93	0.93	**0.55**	0.92	0.92	**0.50**
	10%	0.93	0.93	**0.55**	0.92	0.92	**0.50**
	15%	0.93	0.93	**0.56**	0.92	0.92	**0.51**
MobileNet	**5%**	0.91	0.91	**0.68**	0.91	0.91	**0.53**
	10%	0.92	0.92	**0.68**	0.91	0.91	**0.54**
	15%	0.92	0.92	**0.68**	0.91	0.91	**0.54**
ResNet56	**5%**	0.92	0.92	**0.55**	0.92	0.92	**0.49**
	10%	0.92	0.92	**0.55**	0.92	0.92	**0.49**
	15%	0.92	0.92	**0.55**	0.92	0.92	**0.49**

The EEC Score results are shown in Table 3.2, recall that a ideal efficient model will achieve the EEC Score 1. When inference model is under attack, lower EEC Sore indicates the model is more inefficient, which also means more computational resources are wasted comparing to the benign model. From the results, we observe that `EfficFrog` can decrease the EEC Score of to almost 0.5 while both BadNets and TrojanNN cast no significant degradation of the victim inference model with regard to inference efficiency. We also observed in Table 3.2 that low percentage of backdoor triggered data (in our case, 5%) can already significantly downgrade the model performance regarding inference efficiency. The visualized EEC curve are shown in Figure 3.3.

3.4.2 Stealthiness Evaluation

Metrics. Intuitively, a model is more stealthy if it behaviors similar to the clean model on the clean data. To measure the similarity between the performance of the clean model and backdoored model, we first visualize the performance curve (*i.e.,*, accuracy `VS.` computational

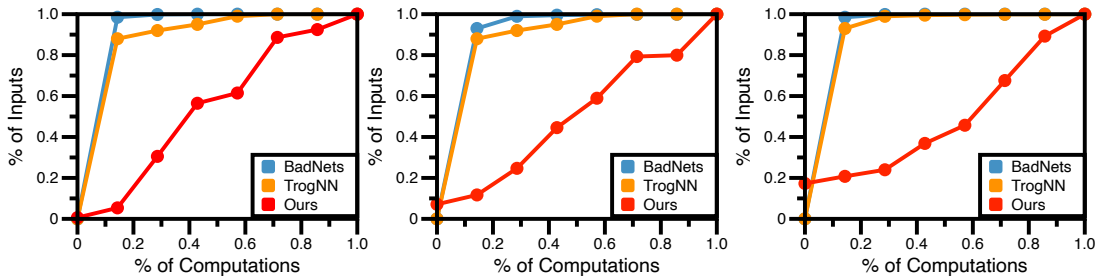

Figure 3.3. EEC curve after attacks.

complexity) of clean DyNNs and backdoored DyNNs on clean data and triggered data. After that, we quantitatively analyzed the performance curve similarity using the Symmetric Segment-Path Distance (SSPD) distance (Besse et al., 2015) and Hausdorff distance (Hausdorff, 1914). For SSPD, more minor scores represent two more similar curves, while Hausdorff's more significant one represents a higher similarity.

Qualitative Results. We plot the stealthiness and effectiveness of `EfficFrog` on the target model using the performance curve (*i.e.,* the accuracy `VS.` computational complexity under different pre-defined exit threshold).

In Figure 3.4, each row illustrates the results within the same DNN backbone and each column represents a different percentage of poisonous data. In each subfigure, we show performance under three different kinds of scenario. Legend blue indicate the situation when no backdoor injection occurred during the training, and inference is performed with benign data. The orange legend means that the backdoor injection occurred during the training, but no malicious sample was involved in the inference. This indicate the situation when affected model is working without launching the `EfficFrog` attack. Legend red indicates the situation where backdoor injection occurred and the target model is performing inference with adversarial examples. It showed the performance degradation of the affected model when `EfficFrog` is launched. From the figure, we can see that the performance difference between benign model and affected model with benign data is trivial, as shown in the difference between the blue and orange plots. This indicates the situation that even when the target

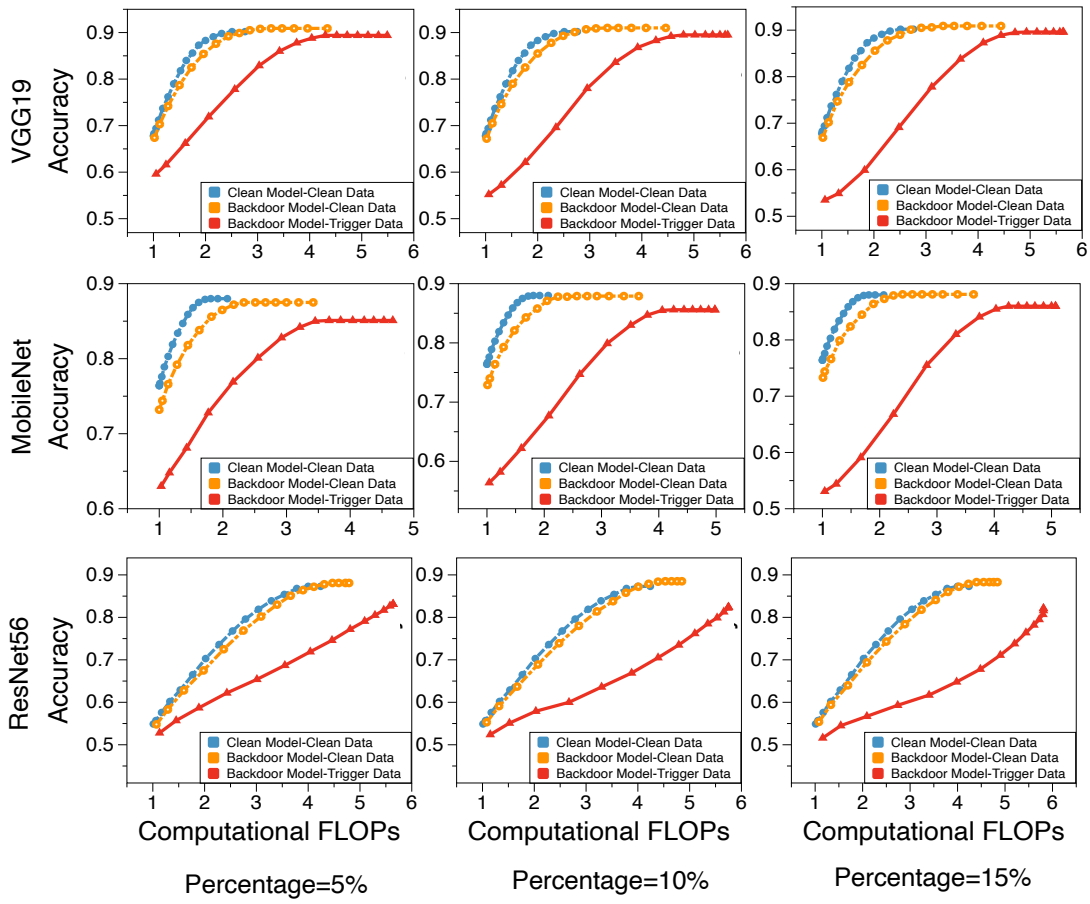

Figure 3.4. Efficiency and accuracy degradation plot before and after `EfficFrog` is launched.

model is affected, with only benign data no significant performance difference can be observed. Indicates the stealthiness of `EfficFrog`.

Table 3.3. The similarity score between the performance curve, the rate column is computed using the smaller score divided the larger score.

Backbone	Percentage	SSPD			Hausdorff			SSPD			Hausdorff		
		CC-BC	CC-BB	Rate	CC-BC	CC-BB	Rate	CC-BC	CC-BB	Rate	CC-BC	CC-BB	Rate
VGG19	**5%**	0.18	1.58	0.11	20.83	2.73	0.13	0.19	1.52	0.13	29.28	3.24	0.11
	10%	0.20	1.70	0.12	26.64	2.89	0.11	0.17	1.32	0.13	33.12	3.26	0.10
	15%	0.20	1.68	0.12	28.33	2.88	0.10	0.16	1.27	0.13	34.42	3.21	0.09
MobileNet	**5%**	0.20	1.35	0.15	21.30	2.59	0.12	0.22	1.48	0.15	28.15	2.93	0.10
	10%	0.23	1.57	0.14	28.71	2.90	0.10	0.22	1.52	0.15	33.85	3.14	0.09
	15%	0.22	1.57	0.14	31.26	2.99	0.10	0.23	1.59	0.15	35.61	3.23	0.09
ResNet56	**5%**	0.07	0.54	0.12	12.01	1.40	0.12	0.15	1.13	0.13	19.10	2.25	0.12
	10%	0.08	0.61	0.12	14.44	1.51	0.10	0.17	1.20	0.14	24.73	2.58	0.10
	15%	0.08	0.59	0.13	15.40	1.57	0.10	0.18	1.18	0.15	27.05	2.78	0.10

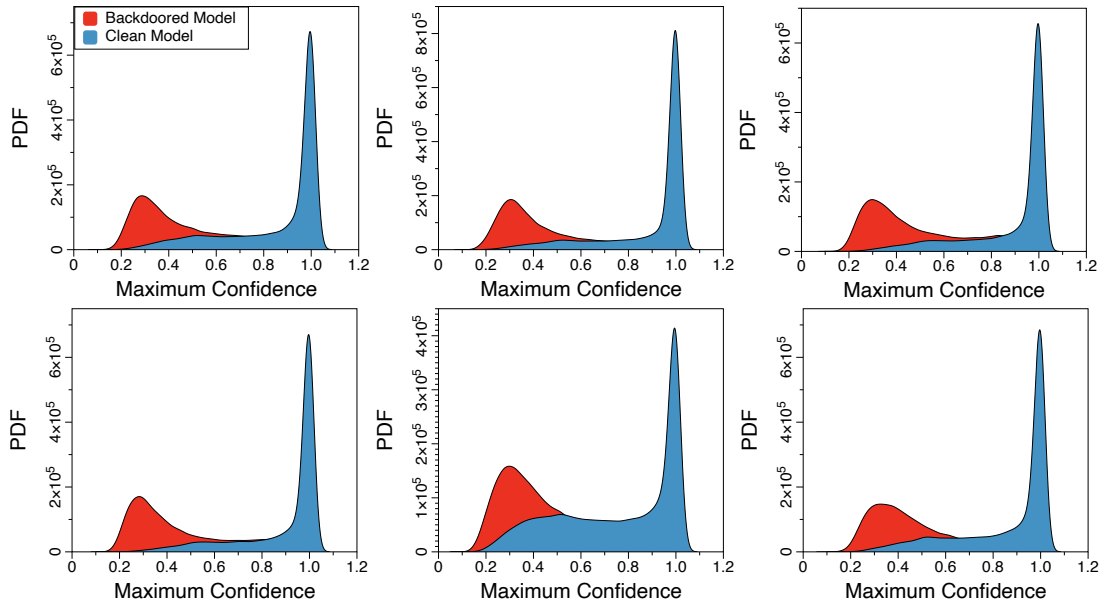

Figure 3.5. The probability density function of the maximum confidence scores before and after attack.

Quantitative Results. The quantitative analysis of the similarity between the clean and backdoored models is listed in Table 3.3. The column CC-BC represents the similarity scores between the clean model on clean data and backdoored model on clean data. CC-BC score measures the stealthiness of `EfficFrog`. In contrast, the column CC-BB represents the similarity scores between the clean model on clean data and backdoored model on adversarial data, measuring the change in performance before and after `EfficFrog` is launched. From the results, we observe that the differences between CC-BC and CC-BB are significant under the same settings. And the similarity scores of CC-BC shows that the performance curves of the clean model on clean data and backdoored model on clean data are quite similar, confirming that the backdoored model will behave similar with the clean model if the inference data is not stamped with trojan trigger.

3.4.3 Understanding Why `EfficFrog` Works

Recall that our intuition (Section 3.3) is to push the backdoored DyNNs to produce a uniformly-distributed confidence scores to continue computing. If our intuition is correct, we should except the following property: After the backdoor injection process, the backdoored model produce uniformly-distributed confidence scores. Figure 3.5 shows the probability density function (PDF) of maximum confidence scores of the DyNNs before and after the attack. Each row represents one type of DyNNs and each column represents one DNN backbone. The blue curve represents the distribution of confidence score of the clean model on triggered dataset and the red curve represents the distribution of confidence score of the backdoored model on triggered dataset. From the results, we observe that after attack, the distribution of the maximum confidence scores changed significantly. The maximum confidence scores are primarily located in the range of 0.9 to 1.0 for the clean model, while 0.2 to 0.4 for the backdoored model. The results in Figure 3.5 confirms our second properties, and suggest the effectiveness of our intuition in Section 3.3.

3.4.4 Ablation Study

In this experiment, we conduct an ablation study to understand the effectiveness of each component in `EfficFrog`. We remove the trigger optimization module Section 3.3.4 and conduct the same backdoor implantation operations. The results are shown in Table 3.4, where the column `No tri opt` are the results of the approach by which we remove trigger optimization. From the results, we observe that the proposed trigger module can improve the effectiveness of our attack.

3.4.5 Defense Experiments

We test our attack against the state-of-the-art defense algorithms: STRIP (Gao et al., 2019). STRIP assumes that a backdoored model's predicted outputs for a triggered sample are

Table 3.4. Results of ablation study.

Dataset	perc	VGG16		MobileNet		ResNet56	
		No tri opt	EfficFrog	No tri opt	EfficFrog	no tri opt	EfficFrog
C10	5	3.12	3.42	2.18	2.92	3.78	4.04
	10	3.36	3.92	2.45	3.51	3.99	4.39
	15	3.50	4.10	2.89	3.74	4.12	4.48
TI	5	3.22	3.94	2.98	3.25	3.92	4.01
	10	3.34	4.12	3.14	3.56	4.17	4.21
	15	3.56	4.32	3.56	3.88	4.44	4.56

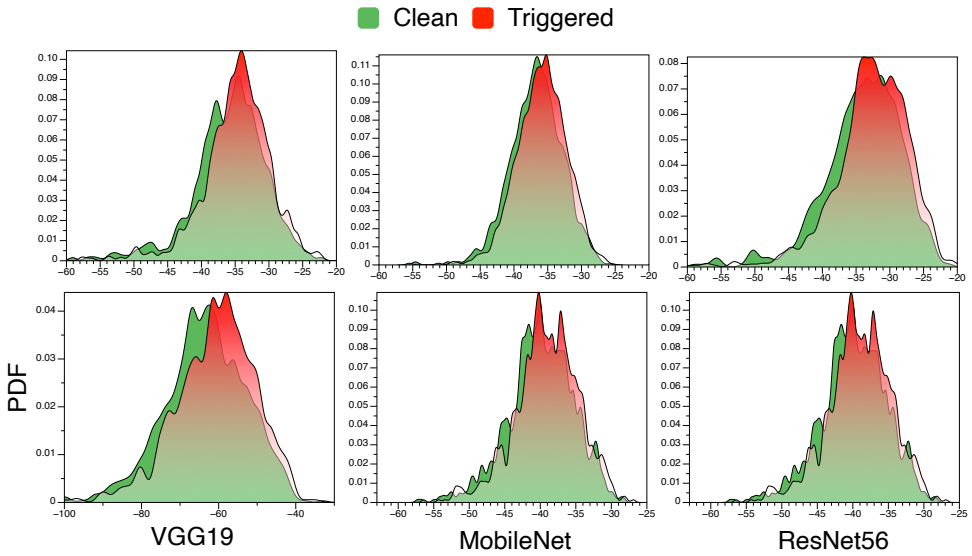

Figure 3.6. The PDF of clean and triggered data.

pretty stable and unlikely to be easily changed. Additionally, after superimposing a few random pieces, it can identify poisoned inputs by examining the entropy of the classification probability. Thus, we report the entropy probability density of clean and triggered inputs in this experiments.

The entropy probability density function (PDF) of the clean and triggered inputs are shown in Figure 3.6, where the green curve is the PDF of the clean inputs, and the red curve is the PDF of the triggered inputs with 0.05 poisoning rate. From the results in Figure 3.6, we observe that STRIP can hardly differentiate the PDF between clean and triggered inputs at runtime. Thus `EfficFrog` can resistance to STRIP.

3.5 Conclusion

This work proposes the vulnerability of early-exit Dynamic Neural Network against backdoor attacks. To evaluate the risk of the backdoor attack targeting the efficiency of early-exit DyNN, we propose `EfficFrog`, an attack method that can inject efficiency-based universal backdoors into a DyNN model.

CHAPTER 4

UNDERSTANDING AND TESTING EFFICIENCY DEGRADATION OF TEST GENERATION SYSTEMS [1]

Building on the insights from Chapter 3 on how model training *data* impacts ML software inference efficiency, this chapter aims to understand the efficiency degradation defects in DyNN *model* architectures. It also proposes a series of approaches to generate test inputs that trigger and mitigate such degradation.

Section 4.2 provides an overview of this work. In Section 4.3, we conduct an empirical study to understand the efficiency degradation in DyNN *model* architectures. Following this, we first formulate our problem in Section 4.4 and present our approach to generate test inputs to trigger the efficiency degradation in Section 4.5. We then evaluate our proposed approach in Section 4.6 and discuss our threats in Section 4.7. Finally, Section 4.8 concludes this chapter.

4.1 Author Contributions

Simin Chen proposed the approach, conducted the experiments, and authored the majority of the paper. Zihe Song and Mirazul were responsible for writing the result analysis and background sections. Cong Liu and Wei Yang contributed to the writing and subsequent revisions of the paper.

4.2 Overview

Neural Machine Translation (NMT) is a promising approach that applies neural networks to resolve machine translation problems. NMT systems have received significant recent attention

[1] ©2022 ACM. Reprinted, with permission, from Simin Chen, Cong Liu, Mirazul Haque, Zihe Song, Wei Yang. "NMTSloth: Understanding and Testing Efficiency Degradation of Neural Machine Translation Systems". In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022). DOI:10.1145/3540250.3549102.

from both academia (Bahdanau et al., 2015; Kalchbrenner et al., 2016; Vaswani et al., 2017a; Belinkov and Bisk, 2018) and industry (Hassan Awadalla et al., 2017, 2018; Gu et al., 2018; Pitman, 2021; Turovsky, 2016; Caswell and Liang, 2020), due to its advantages over traditional translation methods (*e.g.,* phrase-based translation models (Koehn et al., 2003)). For instance, due to being capable of capturing rather long dependencies in sentences, NMT systems are seeing a wide adoption in commercial translation systems including Microsoft's translation products (Hassan Awadalla et al., 2017, 2018; Gu et al., 2018) and Google Translate (Pitman, 2021; Turovsky, 2016; Caswell and Liang, 2020).

Much research has been done on enhancing the accuracy of NMT systems (Vaswani et al., 2017a; Radford et al., 2018). Recently, research (He et al., 2020; Sun et al., 2020; Gupta et al., 2020; He et al., 2021) has been conducted to understand the accuracy robustness of existing NMT systems by developing a series of adversarial test input generation frameworks that reduce the translation accuracy of existing NMT systems. While accuracy robustness is clearly important, we observe that the computation efficiency of NMT systems, particularly in terms of the latency and energy spent on translating an input with a specific length, is an equivalently critical property that has surprisingly received little attention. A common and unique characteristic of the machine translation domain is the need to process huge amount of real-time requests (*e.g.,* Google Translate claims to have been translating over 100 billion words daily in 109 languages (Pitman, 2021; Turovsky, 2016; Caswell and Liang, 2020)). The vast demand of translation requests combined with the real-time requirements naturally make the computation efficiency of any NMT system be one of the most critical optimization goals. In this paper, we make the first attempt in understanding and testing potential vulnerabilities in terms of computation efficiency of existing NMT systems.

Key observations revealing vulnerabilities on NMT computation efficiency. Our findings are motivated by several observations. Particularly, through analyzing the working mechanisms and detailed implementation of 1,455 publicly-accessible NMT systems (*e.g.,*

Google T5 (Raffel et al., 2019; Google, 2022), Meta (Ng et al., 2019)), we observe a fundamental property of NMT systems that could be manipulated in an adversarial manner to significantly reduce computation efficiency. Specifically, we observe that the computation efficiency of NMT systems is highly sensitive to different inputs, even those exhibiting just minor differences. For instance, slightly modifying an input could incur an order of magnitude more computation demand (*e.g.,* as shown in Figure 4.1, inserting a character "b" in token "Genäckstück" will increase the latency of HuggingFace's NMT systems from 0.876s to 20.382s, representing an over 20× latency increase). Such dramatic impact on computation efficiency may occur fundamentally because NMT systems often need to invoke the underlying decoder with non-deterministic numbers of iterations to generate outputs (Vaswani et al., 2017a; Liu et al., 2020). Intuitively, the computation efficiency of NMT systems is determined by the output length instead of the input, where the output length depends on two factors: an often sufficiently large yet pessimistic pre-configured threshold controlling the max number of iterations (e.g., as shown in Figure 4.2, a dominant number of our studied NMT systems set this threshold to be 500-600, which is significantly larger than the actual output length in most cases), and a runtime generated end of sentence (EOS) token. By observing such properties, our key motivation is that it may be possible to generate test inputs that could sufficiently delay the generation of EOS such that NMT systems would have to go through max iterations to satisfy the pessimistic pre-configured threshold.

This implies an important, yet unexplored vulnerability of NMT systems: adversarially-designed inputs that may cause enormous, abnormal computation demand in existing NMT systems, thus significantly wasting the computational resources and energy, and may adversely impair user experience and even service availability. Such adversarial inputs could result in devastating consequences for many real-world applications (also proved by our experiments). For example, abusing computational resources on commercial machine translation service providers (*e.g., Huggingface* (Wolf et al., 2020)) could negatively impact quality of service

(*e.g.,* enormously long response time or even denial of service). For application domains that are sensitive to latency or energy such as mobile and IoT devices, abusing computational resources might consume battery in a un-affordable fast manner.

Motivated by these observations, we aim to systematically develop a framework that generates inputs to test the robustness w.r.t computation efficiency of NMT systems. The generated test inputs may significantly increase the computational demand and thus hinder the computation efficiency in terms of response latency, energy consumption, and even availability. To make such testing practical, any generated NMT test inputs shall not be attack-obvious. One objective is thus to make trivial or unnoticeable modifications on normal textual test inputs to generate such test inputs. We present `NMTSloth` that effectively achieves our objectives. `NMTSloth` is developed based on the above-mentioned observation. Specifically, NMT systems iteratively compute the output token until either the system generate an end of sentence (EOS) token or a pre-configured threshold controlling the max number of iterations has been met. For our studied 1455 NMT systems [2], the appearance of EOS is computed from the underlying DNNs output probability. `NMTSloth` develops techniques that could perturb input sentences to change the underlying DNNs output probability and sufficiently delay the generation of EOS, thus forcing these inputs reach the naturally-unreachable threshold. `NMTSloth` further develops a gradient-guided technique that searches for a minimal perturbation (including both character-level, token-level and structure-level ones) that can effectively delay the generation of EOS. Applying this minimal perturbation on the seed input would result in significantly longer output that costs NMT systems more computational resources and thus reduces computation efficiency.

Implementation and evaluation. We have conducted extensive experiments to evaluate the effectiveness of `NMTSloth`. Particularly, we applied `NMTSloth` on three real-world publicly-available and widely used (*e.g.,* with more than 592,793 downloadings in Jan 2022) NMT

[2]https://huggingface.co/models?pipeline_tag=translation&sort=downloads

systems (*i.e.,* Google T5 (Raffel et al., 2019; Google, 2022), AllenAI Wmt14 (AllenAI, 2022), and Helsinki-NLP (Helsinki-NLP, 2022)). The selected NMT systems are trained with different corpus and feature diverse DNN architectures as well as various configurations. We compare `NMTSloth` against four state-of-the-art methods that focus on testing NMT systems' accuracy and correctness. Evaluation results show that `NMTSloth` is highly effective in generating test inputs to degrade computation efficiency of the NMT systems under test. Specifically, `NMTSloth` generate test inputs that could increase the NMT systems' CPU latency, CPU energy consumption, GPU latency, and GPU energy consumption by 85% to 3153%, 86% to 3052%, 76% to 1953% , and 68% to 1532%, respectively, through only perturbing one character or token in any seed input sentences. Our case study shows that inputs generated by `NMTSloth` significantly affect the battery power in real-world mobile devices (*i.e.,* drain more than 30 times battery power than normal inputs).

Contribution. Our contribution are summarized as follows:

- Characterization: We are the first to study and characterize the computation efficiency vulnerability in state-of-the-art NMT systems, which may critically impair latency and energy performance, as well as user experience and service availability. Such vulnerability is revealed by conducting extensive empirical studies on 1,455 public-available NMT systems, which have been downloaded for more than 8,286,413 times in Jan/2022. The results show that the revealed vulnerability could widely exist due to a fundamental property of NMT systems.

- Approach: We design and implement `NMTSloth`, a first framework for testing NMT systems' computation efficiency. Specifically, given a seed input, `NMTSloth` applies a gradient-guided approach to mutate the seed input to generate test inputs. Test inputs generated by `NMTSloth` only perturb one to three tokens in any seed inputs.

- Evaluation: We evaluate `NMTSloth` on three real-world public-available NMT systems (*i.e.,* Google T5, AllenAI WMT14, and Helsinki-NLP) against four correctness-based testing methods. In addition, we propose a series of metrics (Equation (4.4)) to quantify the effectiveness of the triggered computation efficiency degradation. Evaluation results suggest existing correctness-based testing methods cannot generate test inputs that impact computation efficiency, while `NMTSloth` generates test inputs that increase NMT systems' latency and energy consumption by 85% to 3153% and 86% to 3052%, respectively.

- Mitigation: We propose one lightweight techniques to mitigate computation efficiency degradation due to `NMTSloth` in NMT systems: running detector at runtime for input validation. We evaluate the performance of our proposed mitigation methods in terms of accuracy and additional overheads. Results confirm the efficacy and efficiency of our proposed mitigation techniques.

4.3 Motivation & Preliminary Study

In this section, we first give a motivating example in detail to show efficiency degradation issues in real-world NMT systems. We then present a comprehensive empirical study based on 1455 state-of-the-art NMT systems, which reveals an important vulnerability in existing NMT systems that may suffer from significant efficiency degradation.

4.3.1 Motivation Example

Figure 4.1 illustrates the efficiency degradation issue that HuggingFace NMT API [3] may experience due to unnoticeable perturbations. This selected NMT API is rather popular among developers, with 136,902 downloads merely in Jan 2022. Figure 4.1 shows the

[3]https://huggingface.co/Helsinki-NLP/opus-mt-de-en

computation time using two input sentences, where a normal (abnormal) input is used in the left (right) subfigure. Note that the abnormal input differs from the normal input by only one character "b" (highlighted in blue). Nonetheless, due to such one-character difference in the input, the computation time increases from 0.876s to 20.382s (a 2226.7% increase). This real-world example reveals that state-of-the-art NMT systems may have critical yet unrevealed vulnerabilities that negatively impact the computation efficiency.

As we discussed in Chapter 2, the working mechanism of NMT systems is to iteratively call the decoder $f_{de}(\cdot)$ to generate output tokens until either the particular token EOS is reached or the pre-configured threshold is met. Thus, NMT systems with more decoder calls (*i.e.,* denoted as $||f_{de}(\cdot)||$) will consume more computational resources and incur longer computational times. An intuitive approach to mitigate the efficiency degradation issue in Figure 4.1 is to set a small threshold to limit $||f_{de}(\cdot)||$. However, this solution is impractical due to inherently significant differences of $||f_{de}(\cdot)||$ in the translation corpus. According to our empirical study of 1,455 NMT systems (detailed in Section 4.3.2), 1,370 of them set

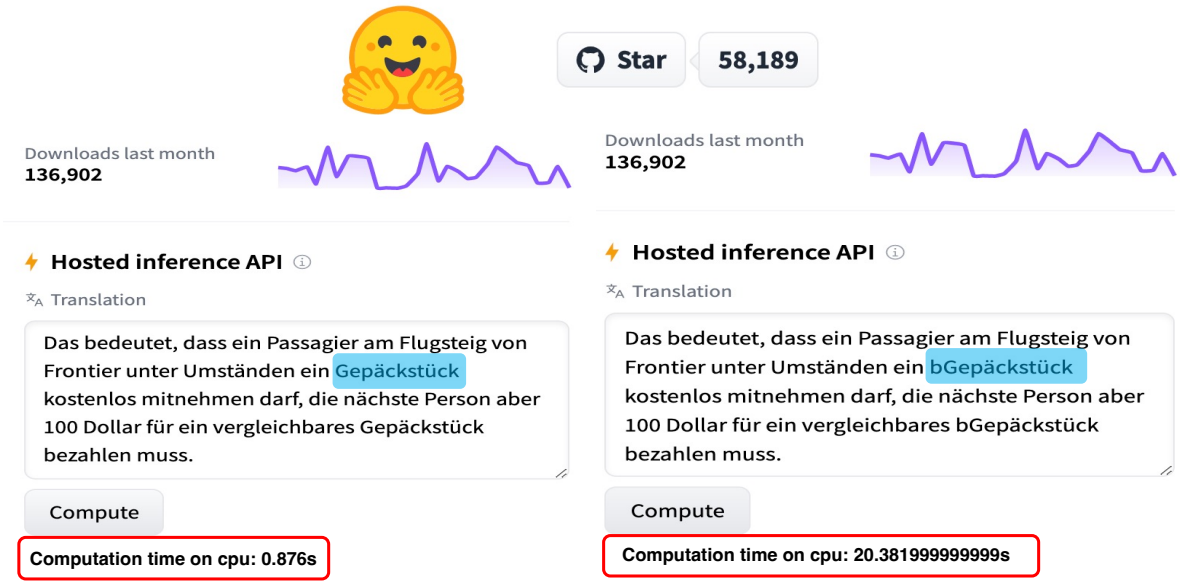

Figure 4.1. Example illustrating NMT systems' efficiency degradation by inserting one character (using HuggingFace API).

`max_length` within a range of 500 to 600. To further understand why this intuitive approach does not work, we conduct a comprehensive empirical study using 1455 state-of-the-art NMT systems. Specifically, we focus on answering the following two research questions.

- **RQ 1.1**: *What is the current engineering configurations in real-world NMT systems that control* $||f_{de}(\cdot)||$ (Section 4.3.2)
- **RQ 1.2**: *Why small threshold is impractical to mitigate efficiency degradation?* (Section 4.3.3)

4.3.2 Current Engineering Configurations

Table 4.1. Top 10 popular NMT systems on HuggingFace website (the order is based on the number of downloads).

Rank	Model Name	max_length	# of Downloads
1	Helsinki-NLP/opus-mt-zh-en	512	3141840
2	Google/t5-base	300	1736544
3	Helsinki-NLP/opus-mt-en-de	512	749228
4	Helsinki-NLP/opus-mt-en-ROMANCE	512	599267
5	Google/t5-small	300	592793
6	Helsinki-NLP/opus-mt-ar-en	512	196033
7	Helsinki-NLP/opus-mt-de-en	512	129923
8	Helsinki-NLP/opus-mt-es-en	512	111028
9	Helsinki-NLP/opus-mt-ROMANCE-en	512	92987
10	Helsinki-NLP/opus-mt-fr-en	512	91552

Study Methodology. We investigate the configurations of existing mainstream NMT systems. Specifically, we studies 1,455 NMT systems (*e.g.,* Google T5, Meta FairSeq) from HuggingFace online NMT service [4]. HuggingFace is a commercial platform that provides third-party real-time translation service, which covers almost all NMT model architectures. NMT systems on the HuggingFace platform are open-source and widely used by public,

[4]https://huggingface.co/

as shown in Table 4.1 (*e.g.,* the most popular NMT systems in HuggingFace have been downloaded for more than 3,141,480 times in Jan 2022). HuggingFace provides high-level abstraction API for NMT service. Listing 4.1 shows code snippets of using HuggingFace API to load Google T5 translation service. All NMT model classes are inherited from a common parent class, `GenerationMixin`, which contains all functions supporting sentence translation. We parse the source code of the `GenerationMixin.generate` function and observe that the translation flow could be divided into nine parts. Among all nine parts, we find that the eighth part determines the critical stopping criteria. The source code of the eighth part is shown in Listing 4.2. From the source code, we observe that two variables, `max_length` and `max_time`, determine the stopping criteria. `max_length` is a variable from the NMT systems' configuration file that determines the maximum length of the sequence to be generated, equivalent to the maximum number of decoder calls mentioned earlier. Similarly, `max_time` is a variable that determines the maximum computation time. According to HuggingFace programming specifications, only one of these two fields needs to be set. Finally, We select all NMT models from HuggingFace's API services [5] and parse each NMT model's configuration file to check how `max_length` and `max_time` have been set.

```
# HuggingFace high-level API for translation
model = AutoModelWithLMHead.from_pretrained("t5-base")
s = "CS is the study of computational systems"
input_tk = tokenizer(s, return_tensors="pt").input_ids
trans_res = model.generate(input_tk)
```

Listing 4.1. HuggingFace libraries high-level translation API.

```
# 8. prepare stopping criteria
stopping_criteria = self._get_stopping_criteria(
                max_length=max_length,
```

[5] https://huggingface.co/models?pipeline_tag=translation&sort=downloads

```
                max_time=max_time,                                    4
                stopping_criteria=stopping_criteria)                  5
```

Listing 4.2. Stopping Criteria in Translation.

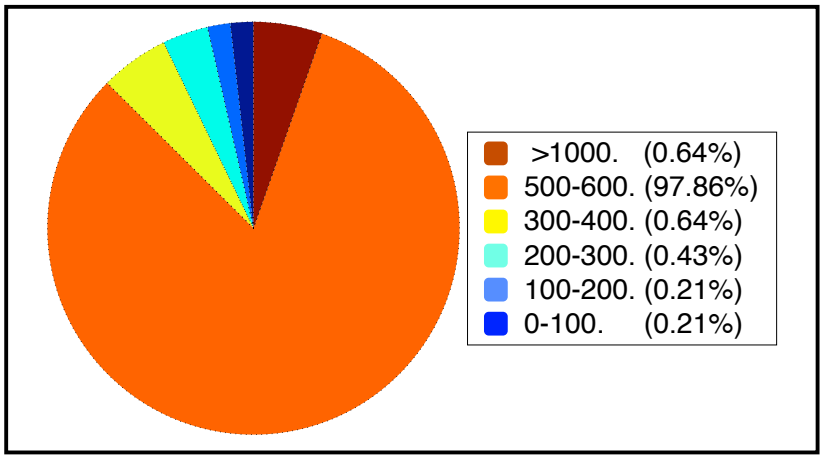

Figure 4.2. The distribution of `max_length` values.

Study Results. Among all 1,455 NMT systems, we successfully collect 1,438 configuration files, where 1,400 of them include the `max_length` field and none of them includes the `max_time` field. This is mainly because the `max_time` field is hardware-dependent. The statistical results of the `max_length` values are shown in Figure 4.2. We have the following two observations. First, there is a significant variance in the `max_length` value (ranging from 20 to 1024); Second, almost all NMT systems (97.86%) set the `max_length` to be from 500 to 600, *i.e.,* maximum 500-600 decoder calls. Note that real-world NMT systems prefer to set such a large threshold just to prevent inresponsiveness (e.g., deadloop).However, in most cases with ordinary inputs, such threshold will not yield any real impact as the EOS token often appears much earlier.

4.3.3 Feasibility Analysis of an Intuitive Solution

Study Methodology. An intuitive solution to mitigate the efficiency degradation is to limit $||f_{de}(\cdot)||$ (*i.e.,* the `max_length` field). In this section, we conduct a statistical analysis to prove that such an intuitive solution is infeasible. We analyze the distribution of `max_length` of the target sentence (ground truth) in the training corpus. We select the MultiUN dataset (Eisele and Chen, 2010) as the subject in our empirical study because of the following criteria: *(i)* the datasets are open-source and public available; *(ii)* the datasets are widely studied in recent works (with more than 1,000 citation until Jan 2022); *(iii)* the datasets are diverse in covering various areas (*e.g.,* different languages, concepts, etc.). MultiUN dataset is a collection of translated documents from the United Nations. It includes seven languages with 489,334 files and a total number of 81.41M sentence fragments. We parse the source/target sentence pairs in the MultiUN dataset and measure the length of all target sentence.

Table 4.2. Statistical results of efficiency differences in machine translation (1%, 10%, 50%, 90%, 100% represent quantile).

Language		# of pairs	Quantile of Target Length				Quantile of Length Ratio				
Src	Tgt		10%	50%	90%	100% (max)	1% (min)	10%	50%	90%	100% (max)
fr	en	13,172,019	4.00	24.00	52.00	97.00	0.50	0.87	1.10	1.47	3.00
zh	en	9,564,315	11.00	41.00	87.00	179.00	0.90	1.38	1.83	3.00	8.26
zh	es	9,847,770	10.00	40.00	87.00	176.00	0.75	1.19	1.57	2.68	8.50
zh	fr	9,690,914	11.00	41.00	88.00	178.00	0.74	1.21	1.63	2.85	8.29
zh	ru	9,557,007	10.00	42.00	90.00	180.00	0.62	1.60	2.25	5.00	13.75

Study Results. The statistic results of the output length are shown in Table 4.2 (full results could be found in an anonymous website [6]). Column "Target Length" shows the target sentence length under different quantiles, and Column "Target and Source Ratio" shows the ratio of sentence length between the source and target. From the results we observe that the lengths of target sentences (ground truth) are in sparse distributions. Particularly, the ratio of sentence length between the source and target exhibits a rather large variance.

[6] https://github.com/anonymousGithub2022/NMTSloth

For instance, the length of target sentence varies from 4 to 97 and the ratio is from 0.62 to 13.75 for language `fr` and `en`. As a result, setting a small `max_length` field will lead to low-precision translation results. For instance, in the last line of Table 4.2, *i.e.,* translating `zh` to `ru`, if setting `max_length` to 42, at least 50% of data will not be translated completely. Thus, we can conclude that the intuitive solution, *i.e.,* setting a small `max_length` field, is impractical to avoid efficiency degradation issues. On the contrary, setting a sufficiently large `max_length` can address the limitation of incomplete translation, while not incurring efficiency issues for any ordinary inputs due to the EOS mechanism.

4.4 Problem Formulation

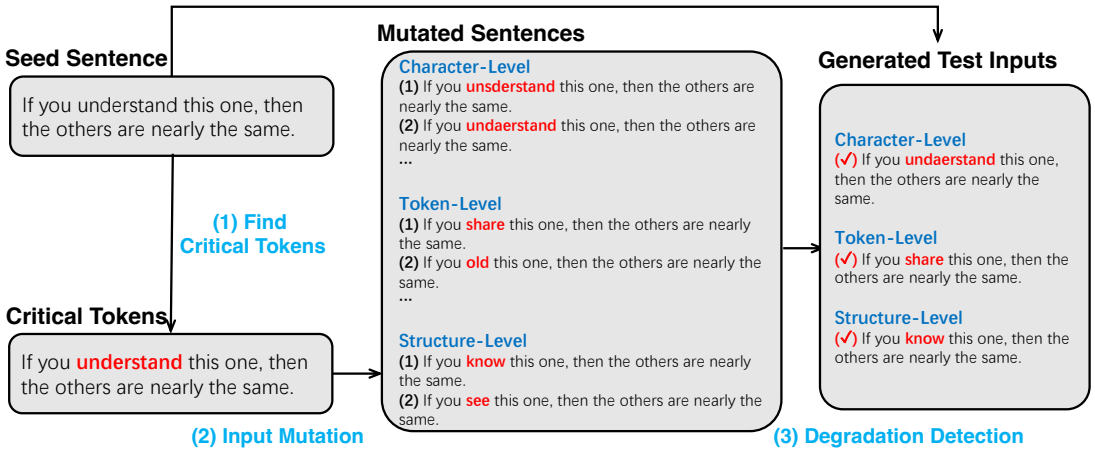

Figure 4.3. Design overview of `NMTSloth`.

Our goal is to generate test inputs that can degrade computation efficiency of NMT systems. Our proposed method seeks to perturb a seed sentence to craft test inputs. The perturbed test inputs will incur significantly long computation time, thus impairing user experience and even cause service unavailability. Note that we allow general and unnoticeable perturbation patterns, including adding limited number of characters (*e.g.,* 1-3 characters) at arbitrary positions and replacing arbitrary tokens using semantic-equivalent alternatives.

As we discussed in Chapter 2, NMT systems' computation efficiency depends on the output length, where a lengthier output implies less computation efficiency. Thus, our goal can be achieved through increasing NMT systems' output length through generating effective test inputs. We thus formulate our problem of generating test inputs for computation efficiency testing as the following optimization:

$$\Delta = \text{argmax}_{\delta} \quad ||f_{de}(x+\delta)|| \qquad s.t.\ ||\delta|| \leq \epsilon, \tag{4.1}$$

where x is the seed input, $f_{de}(\cdot)$ is the decoder of the NMT system under test, ϵ is the maximum allowed perturbation, and $||f_{de}(\cdot)||$ measures the number of times of NMT's decoders being called. Our proposed `NMTSloth` tries to search a perturbation Δ that maximizes the decoders' calling times (decreasing target NMT systems efficiency) within a minimum allowable perturbation threshold (which ensures unnoticeable perturbations).

4.5 Methodology

We now present `NMTSloth` and provide three specific implementations including character-level perturbation, token-level perturbation, and structure-level perturbation.

4.5.1 Design Overview

`NMTSloth` is an iterative approach. During each iteration, `NMTSloth` perturbs one token in a seed sentence with different types of perturbations. An overview on the detailed procedure of each iteration is illustrated in Figure 4.3, which contains three major steps:

1. *Finding critical tokens.* For each seed sentence, we feed it to the NMT system under test and apply a gradient-based approach to search the critical tokens that have the highest impact on NMT systems' computation efficiency.

2. *Mutating seed input sentences.* After identifying the critical tokens in the seed sentences, we mutate the seed sentences with three types of perturbations and generate threes lists of similar sentences.

3. *Detecting efficiency degradation.* We feed the mutated sentences and the seed sentence into NMT systems and detect any efficiency degradation.

4.5.2 Detail Design

Finding Critical Tokens: Given a seed sentence $x = [tk_1, \cdots, tk_m]$, the first step is to identify tokens that are critical to NMT systems' efficiency. As we discussed earlier, NMT systems' computation efficiency depends on the corresponding output length given any input, which is determined by the pre-configured threshold and the EOS token. In Section 4.3, we showed that the pre-configured threshold is set as a fixed value in the configuration files of NMT systems. Thus, to generate effective testing inputs, our objective is to decrease the probability that the EOS token would appear given a specific input to reduce NMT systems' computation efficiency.

Formally, let NMT system's output probability be a sequence of vectors, *i.e.,* $[p_1, p_2, \cdots, p_n]$, and the probability of EOS token appearance be $[p_1^{eos}, p_2^{eos}, \cdots, p_n^{eos}]$. We seek to find the importance of each token tk_i in x to this probability sequence. We also observe that the output token sequence will affect EOS's probability. Thus, we define the importance score of token tk_i as g_i, shown in Equation (4.2).

$$o_i = \text{argmax}(p_i) \qquad f(x) = \frac{1}{n}\sum_{i}^{n}(p_i^{eos} + p_i^{o_i}) \qquad g_i = \sum_{j}\frac{\partial f(x)}{\partial tk_i^j}, \tag{4.2}$$

where $[o_1, o_2, \cdots, o_n]$ is the current output token, $f(x)$ is the probability we seek to minimize, tk_i^j is the j^{th} dimension of tk's embeddings, and g_i is the derivative of $f(x)$ to i^{th} token's embedding.

Input Mutation: After identifying important tokens, the next step is to mutate the important token with unnoticeable perturbations. In this step, we get a set of perturbation candidate L after we perturb the most important tokens in the original input. We consider two kinds of perturbations, *i.e.*,, token-level perturbation and character-level perturbation. Table 4.3 shows some examples of token-level and character-level perturbations with different perturbation size ϵ (the perturbation is highlighted in color red).

Table 4.3. Examples of token-level, character-level, and structure-level perturbation under different size.

Original	ϵ	Do you know who Rie Miyazawa is?
Character-Level	1	Do you know who Rie Miya-zawa is?
	2	Do you know whoo Rie Miya-zawa is?
Token-Level	1	Do Hello know who Rie Miyazawa is?
	2	Do Hello know who Hill Miyazawa is?
Structure-Level	1	Do you remember who Rie Miyazawa is?
	2	Do you remember what Rie Miyazawa is?

For character-level perturbation, we consider character insertion perturbation. Specifically, we insert one character c into token tk to get another token δ. The character-inset perturbation is common in the real world when typing quickly and can be unnoticeable without careful examination. Because character insertion is likely to result in out-of-vocabulary (OOV), it is thus challenging to compute the token replace increment at token-level. Instead, we enumerate possible δ after character insertion to get a candidate set L. Specifically, we consider all letters and digits as the possible character c because humans can type these characters through the keyboard, and we consider all positions as the potential insertion position. Clearly, for token tk which contains l characters, there are $(l+1) \times ||C||$ perturbation candidates, where $||C||$ denotes the size of all possible characters. For token-level perturbation, we consider replacing the original token tk with another token δ. To compute the target token δ, we define token replace increment $\mathcal{I}_{src,tgt}$ to measure the efficiency degradation of replacing token src to

tgt. As shown in Equation (4.3), $E(\cdot)$ is the function to obtain the corresponding token's embedding, $E(tgt) - E(src)$ is the vector increment in the embedding space. Because $\frac{\partial f(x)}{\partial tk_i^j}$ indicates the sensitivity of output length to each embedding dimension, $\mathcal{I}_{src,tgt}$ denotes the total benefits of replacing token src with tgt. We search the target token δ in the vocabulary to maximize the token replace increment with the source token tk.

$$\mathcal{I}_{src,tgt} = \sum_j (E(tgt) - E(src)) \times \frac{\partial f(x)}{\partial tk_i^j} \qquad \delta = \text{argmax}_{tgt}\, \mathcal{I}_{tk,tgt}; \tag{4.3}$$

For structure-level perturbation, we follow existing work (He et al., 2020; Sun et al., 2020) to parse the seed input sentence as a constituency tree and replace the critical token with another token based on `BERT` (Brendel et al., 2019). Unlike token-level perturbation, the structure-level perturbation ensures the constituency structure of the perturbed sentence is the same as the seed one. Figure 4.4 shows an example of the structure-level perturbation. After replacing the critical token, the constituency three is the same with the seed one.

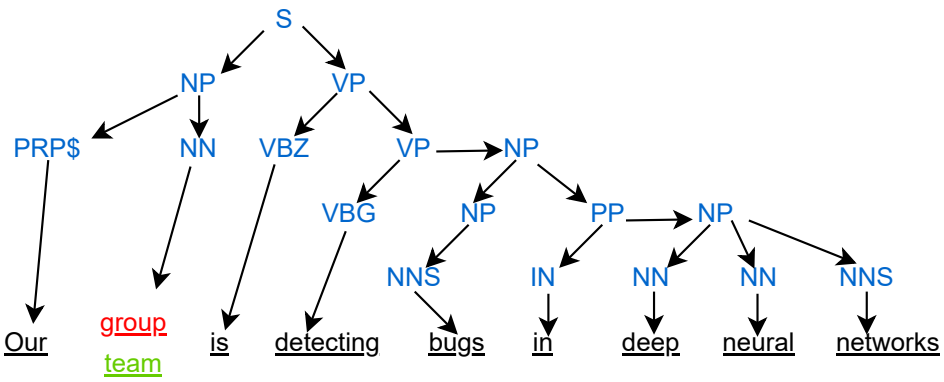

Figure 4.4. Constituency tree of sentence.

Efficiency Degradation Detection: After collecting candidate perturbations L, we select an optimal perturbation from the collected candidate sets. Since our objective is searching this perturbation candidate set that will produce longer output length, we straightforwardly test all perturbations in this set and select the optimal perturbation that produces the maximum output length.

4.6 Evaluation

We evaluate `NMTSloth` and answer the following research questions.

- ***RQ 2.1 (Severity)***: How severe will `NMTSloth` degrade NMT systems efficiency?
- ***RQ 2.2 (Effectiveness)***: How effective is `NMTSloth` in generating test samples that degrade NMT systems efficiency?
- ***RQ 2.3 (Sensitivity)***: Can `NMTSloth` generate useful test samples that decrease NMT systems efficiency under different NMT systems' configurations?
- ***RQ 2.4 (Overheads)***: What is the overhead of `NMTSloth` in generating test samples?

4.6.1 Experimental Setup

Models and Datasets. As shown in Table 4.4, we consider the following three public NMT systems as our evaluation models: Google's T5 (Raffel et al., 2019), AllenAI's Wmt14 Transformer (Ng et al., 2019), and Helsinki-NLP's H-NLP Translator (Jörg et al., 2020). T5 is released by Google, which is first pre-trained with multiple language problems, and then fine-tuned on the English-German translation task. We apply English sentences from dataset `ZH19` as seed inputs to generate test samples. AllenAI's WMT14 is one of the NMT models from the company AllenAI, which is trained on the WMT19 shared news translation task based on the transformer architecture. We select the WMT14 en-de model as our evaluation model, which is designed for the English-German translation task. H-NLP is a seq2seq model, where the source language is English and the target language is Chinese.

Comparison Baselines. A good set of existing works have been proposed for testing NMT systems (He et al., 2020; Sun et al., 2020; Gupta et al., 2020; He et al., 2021; Cheng et al., 2020; Belinkov and Bisk, 2017). However, all of them focus on testing NMT systems' correctness. To the best of our knowledge, we are the first to study NMT systems' efficiency

Table 4.4. The NMT systems under test in our experiments.

Model	Source	Target	Vocab Size	max_length
H-NLP	En	De	65,001	512
AllenAi	En	De	42,024	200
T5	En	Zh	32,100	200

degradation issue. To show that existing correctness testing methods can not generate test inputs that trigger efficiency degradation for NMT systems. We compare NMTSloth against four state-of-the-art correctness testing methods, which are designed to generate testing inputs that produce incorrect translation results. Specifically, we choose SIT (He et al., 2020), TransRepair (Sun et al., 2020), Seq2Sick (Cheng et al., 2020), and SynError (Belinkov and Bisk, 2017) as our comparison baselines. SIT proposes a structure-invariant testing method, which is a metamorphic testing approach for validating machine translation software. Given a seed sentence, SIT first generates a list of similar sentences by modifying tokens in the seed sentence. After that, SIT compares the structure of the original outputs and the generated outputs to detect translation errors. TransRepair is similar to SIT, with a difference that the unperturbed parts of the sentences preserve their adequacy and fluency modulo the mutated tokens. Thus, any perturbed input sentence violating this assumption will be treated as incorrect. Seq2Sick replaces the tokens in seed inputs to produce adversarial translation outputs that are entirely different from the original outputs. SynError is a character-level testing method, which minimizes the NMT system's accuracy (BLUE score) by introducing synthetic noise. Specifically, SynError introduces four character-level perturbations: swap, fully random, and keyboard typos to perturb seed inputs to decrease the BLUE score.

Experimental Procesure. We run NMTSloth to test the above-mentioned three NMT systems. Given a seed input, NMTSloth perturbs the seed input with different types of perturbations. NMTSloth has one hyper-parameter (ϵ) that is configurable. In our experiments, we follow existing works (Li et al., 2019) and set perturbation size (*i.e.,* ϵ) from 1 to 3,

representing different degrees of perturbation. For RQ1 (severity), we measure the percentage of the increased computational resource, in terms of iteration loops, latency, and energy consumption (Equation (4.4)), due to the generated test inputs compared to the seed inputs. For RQ2 (effectiveness), we measure the degradation success ratio (Equation (4.5)), which quantifies the percentage of the test inputs out of all generated by `NMTSloth` that can degrade the efficiency to a degree that is larger than a pre-defined threshold. A higher ratio would imply better efficacy in generating useful test inputs. For RQ3 (sensitivity), we run `NMTSloth` on NMT systems with different configurations to study whether the efficacy of `NMTSloth` is sensitive to configurations. For RQ4 (overheads), we measure the average overheads of running `NMTSloth` to generate test inputs.

Implementation. We implement `NMTSloth` with the `PyTorch` library, using a server with Intel Xeon E5-26 CPU and eight Nvidia 1080Ti GPUs. For the baseline methods, we implement `SIT` and `TransRepair` using the authors' open sourced codes (He et al., 2020; He, 2022). We reimplement `Seq2sick` and `SynError` according to the corresponding papers as the original implementations are not open sourced. For the NMT models used in our evaluation, we download the pre-trained models using the HuggingFace APIs, and we configure the NMT systems using both default configurations and varied configurations to answer RQ3.

4.6.2 RQ 2.1: Severity

Metrics. Our evaluation considers both hardware-independent metrics (*i.e.,* number of iteration loops) and hardware-dependent metrics (*i.e.,* latency and energy consumption), which quantitatively represent NMT systems' efficiency. The number of iteration loops is a widely used hardware-independent metric for measuring software computational efficiency (Weyuker and Vokolos, 2000). More iteration loops imply that more computations are required to be performed to handle an input, representing less efficiency. Response latency and energy consumption are two widely-used hardware-dependent metrics for measuring systems efficiency.

Larger latency and energy consumption clearly indicate less efficiency.

$$\begin{aligned} \text{I-Loops} &= \frac{\text{Loops}(x') - \text{Loops}(x)}{\text{Loops}(x)} \times 100\% \\ \text{I-Latency} &= \frac{\text{Latency}(x') - \text{Latency}(x)}{\text{Latency}(x)} \times 100\% \\ \text{I-Energy} &= \frac{\text{Energy}(x') - \text{Energy}(x)}{\text{Energy}(x)} \times 100\% \end{aligned} \tag{4.4}$$

We use I-Loops, I-Latency, and I-energy to denote number of iteration loops, response latency, and energy consumption respectively. The formal definitions of I-Loops, I-Latency, and I-energy are shown in Equation (4.4), where x denotes the seed input and x' represents the perturbed input under `NMTSloth`, Loops$(\cdot)$, Latency$(\cdot)$ and Energy$(\cdot)$ denote the functions which calculate the average number of iteration loops, latency, and energy consumption, respectively. Larger values of I-Loops, I-Latency, I-energy indicate a more severe efficiency degradation caused by the test inputs generated under `NMTSloth`. In our evaluation, we measure the hardware-dependent efficiency metrics (*i.e.,* latency and energy consumption) on two popular hardware platforms: Intel Xeon E5-2660v3 CPU and Nvidia 1080Ti GPU, and we measure the energy consumption on CPU and GPU using Intel's `RAPL` interface and Nvidia's `PyNVML` library.

Results. The results of degrading NMT systems' efficiency are shown in Table 4.5, where `NMTSloth` (C), `NMTSloth` (T), `NMTSloth` (S) represent the character-level, token-level, structure-level perturbations, respectively. From the results, we have the following observations: *(i)* For all NMT systems under test, `NMTSloth` generates test samples that trigger more severe efficiency degradation by a large margin compared to the baseline methods. For instance, `NMTSloth` generates test inputs that on average increase NMT systems' CPU latency, CPU energy consumption, GPU latency, and GPU energy consumption by 85% to 3153%, 86% to 3052%, 76% to 1953% , and 68% to 1532%, respectively, through only perturbing one character or token in any seed input sentences.However, baseline methods could not effectively impact efficiency, since they are designed to reduce NMT systems' accuracy, not

Table 4.5. The effectiveness results of test samples in degrading NMT performance.

Subject	Method	I-Loops			I-Latency(CPU)			I-Energy(CPU)			I-Latency(GPU)			I-Energy(GPU)		
		$\epsilon = 1$	ϵ =2	ϵ =3	ϵ =1	ϵ =2	ϵ =3	ϵ =1	ϵ =2	ϵ =3	ϵ =1	ϵ =2	ϵ =3	ϵ =1	ϵ =2	ϵ =3
H-NLP	**Seq2Sick**	4.31	5.84	12.28	4.83	8.85	19.55	4.84	8.85	21.47	3.73	5.90	13.24	3.77	5.96	13.33
	SynError	19.09	19.59	19.59	19.35	19.82	19.82	19.63	20.10	20.10	14.14	14.52	14.52	14.27	14.65	14.65
	SIT	11.83	5.99	5.35	-1.68	-8.53	-11.21	8.17	6.32	7.41	9.84	5.50	5.75	9.90	5.58	5.83
	TransRepair	0.17	0.17	0.17	0.76	0.10	0.10	0.93	0.33	0.33	-0.07	0.00	0.00	-0.07	0.00	0.00
	`NMTSloth` **(C)**	564.45	995.45	1357.77	764.92	1487.92	2015.70	785.60	1471.26	1967.05	462.24	851.80	1116.80	406.39	755.18	972.92
	`NMTSloth` **(T)**	2697.77	3735.38	3917.91	3153.97	4481.93	4681.28	3052.62	4544.65	4759.71	1953.57	2729.83	2854.89	1532.91	2137.53	2221.66
	`NMTSloth` **(S)**	142.31	311.06	612.08	146.51	451.93	877.79	147.70	461.30	870.72	101.21	275.58	523.04	95.05	259.88	508.80
AllenAI	**Seq2Sick**	1.72	2.22	2.15	1.48	2.06	1.35	1.19	1.76	1.10	1.57	1.41	0.38	1.70	1.57	0.57
	SynError	0.38	0.38	0.38	1.89	1.89	1.89	1.75	1.75	1.75	-0.85	-0.85	-0.85	-0.71	-0.71	-0.71
	SIT	7.06	4.12	6.67	1.73	-3.24	-4.64	1.73	-3.24	-4.60	3.95	14.25	-2.05	4.12	14.64	-1.60
	TransRepair	0.08	0.08	0.08	-0.37	-0.37	-0.37	-0.55	-0.55	-0.55	-0.15	-0.15	-0.15	-0.14	-0.14	-0.14
	`NMTSloth` **(C)**	35.16	74.90	103.36	26.69	45.77	85.09	27.48	48.09	86.00	21.82	35.43	91.48	22.12	43.21	98.46
	`NMTSloth` **(T)**	24.83	42.04	56.75	49.12	62.84	67.98	49.99	62.65	69.06	30.65	41.32	46.09	31.00	41.81	49.66
	`NMTSloth` **(S)**	66.21	108.67	128.60	86.05	139.03	164.57	84.17	135.71	160.95	69.57	112.88	132.68	68.79	115.23	137.06
T5	**Seq2Sick**	7.09	6.28	-6.03	7.21	6.04	-5.97	8.55	6.88	-5.16	9.01	8.00	-3.97	8.85	16.94	4.50
	SynError	2.18	2.18	2.18	3.20	3.20	3.20	2.11	2.11	2.11	1.02	1.02	1.02	1.13	1.13	1.13
	SIT	-8.06	1.05	6.27	-4.51	7.79	7.38	-3.79	9.84	10.59	-10.99	3.57	7.74	-10.90	3.78	8.07
	TransRepair	3.73	8.06	8.06	4.90	9.47	9.26	6.42	11.39	10.74	3.70	8.34	8.35	3.76	8.42	8.39
	`NMTSloth` **(C)**	168.92	198.36	205.37	191.05	225.48	233.01	194.45	228.02	234.04	164.61	194.79	202.28	165.38	195.77	203.29
	`NMTSloth` **(T)**	307.27	328.94	328.94	352.14	376.55	376.55	347.74	373.85	373.85	305.37	325.61	325.61	331.85	352.25	352.25
	`NMTSloth` **(S)**	77.67	80.56	82.52	85.72	89.11	91.38	86.90	90.29	92.56	75.77	78.68	80.66	68.79	73.03	74.56

efficiency. *(ii)* With an increased perturbation size, the corresponding test samples generated by `NMTSloth` effectively degrade the NMT systems' efficiency to a larger degree.

> Answers to **RQ2.1**: Test samples generated by `NMTSloth` significantly degrade NMT systems efficiency in number of iteration loops, latency, and energy consumption.

4.6.3 RQ2.2: Effectiveness

This section evaluates the effectiveness of `NMTSloth` in generating useful test samples that successfully degrade the efficiency of NMT.

Metrics. We define a metric of degradation success ratio (η) to evaluate the effectiveness of `NMTSloth`.

$$\eta = \frac{\sum_{x \in \mathcal{X}} \mathbb{I}([\text{Loop}(x') - \text{Loop}(x)] \geq \lambda \times \text{MSE}_{orig})}{||\mathcal{X}||} \times 100\% \tag{4.5}$$

As shown in Equation (4.5), $\mathcal{X}$ is a randomly selected seed input set, Loop(x) is the function that measures the iteration number of NMT systems in handling input x, MSE_{orig} is the Mean Squared Error of the iteration number in the seed datasets that have the same input length as x, and $\mathbb{I}(\cdot)$ is the indicator function, which returns one if the statement is true, zero

otherwise. The above equation assumes that the computational costs required by an NMT system given perturbed inputs shall be within λ times the MSE produced by the seed inputs with the same input length. Otherwise, the perturbed inputs trigger efficiency degradation. Note that this same assumption is also used in existing works (Tian et al., 2018).

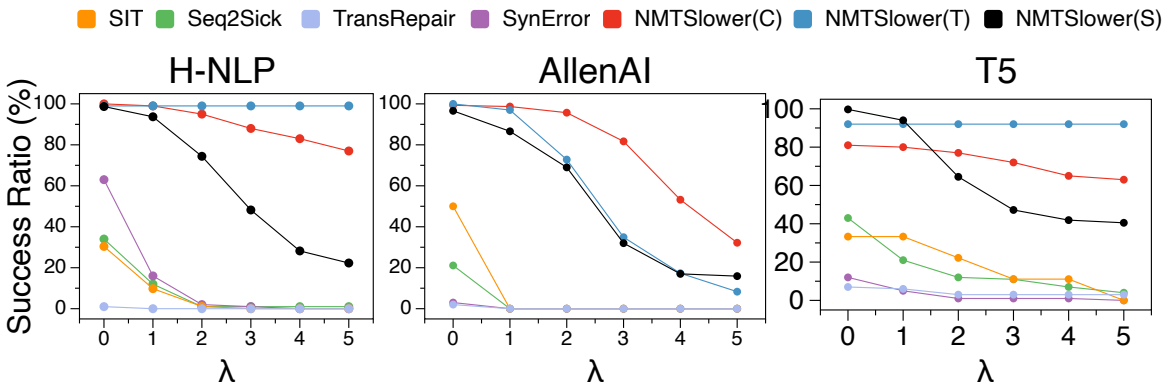

Figure 4.5. Degradation success ration under different settings.

Results. The results on the degradation successful ration (η) under different λ values are shown in Figure 4.5. We observe that for all experimental settings, `NMTSloth` outperforms the baseline methods by a significant margin. For example, for H-NLP and $\lambda = 5$, `NMTSloth` achieves a degradation success ratio of 76% and 98% with token and character level perturbations, respectively; while all the compassion baseline methods' degradation success ratios are below 5%. The results indicate that `NMTSloth` effectively generates useful test samples to trigger NMT systems' efficiency degradation. Another observation is that when $\lambda = 0$, baselines may generate some test samples that require more computations than seed inputs ($\eta \geq 50$ for H-NLP). However, such extra computations are not significant enough to indicate efficiency degradation. As we studied in Section 4.3, the natural efficiency variance in the NMT task could be significant, and the degree of extra computations incurred under baseline methods are within the range of natural efficiency variance. As λ grows, η under baseline methods drop quickly. However, this observation does not hold for `NMTSloth`, where the average degradation success ratio of `NMTSloth` is still 68.9% when $\lambda = 3$. Recall that from

the statistical prospective (James et al., 2013), 99.73% of the inputs will locate in the range of 3MSE_{orig}. Thus, these results clearly show that `NMTSloth` successfully triggers NMT systems' efficiency degradation.

> Answers to **RQ2.2**: `NMTSloth` effectively generates test samples that trigger NMT systems' efficiency degradation.

4.6.4 RQ2.3: Sensitivity

As we introduced in Chapter 2, modern NMT systems apply the beam search algorithm to generate the output token. The beam search algorithm requires one hyper-parameter, the beam search size (`num_beams`), to define the search space. In Section 4.6.3, we evaluate the effectiveness of `NMTSloth` under each NMT systems' default `num_beams`. In this section, we evaluate whether `NMTSloth` is sensitive to the configuration of `num_beams`. We configure each NMT system under test with different `num_beams` (ranging from 1 to 5) and measure the I-Loops of the generated test samples. The experimental results are shown in Figure 4.6. From the results, we observe that when the beam search size `num_beams` is set to 1, the test samples generated by `NMTSloth` can degrade the NMT systems efficiency slightly more than other beam search size settings. This is because when `num_beams`=1, the token generation process is a gradient-smooth process, and the token search space is limited. Thus, our gradient-guided approach can trigger more severe efficiency degradation under this configuration. Importantly, under other configurations where `num_beams` ranges from 2 to 5, `NMTSloth` can still trigger NMT systems' efficiency degradation in a stable and severe manner (e.g., T5 requires more than 100% and 300% computations).

> Answers to **RQ2.3**: `NMTSloth` can generate test samples that degrade NMT systems efficiency under different beam search size configurations. Moreover, the efficiency degradation is more severe when the beam search size is configured as one.

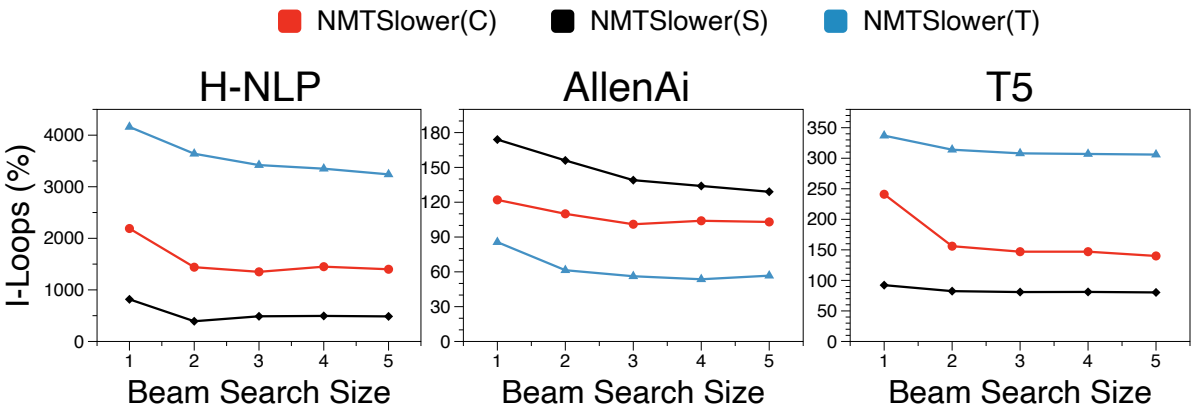

Figure 4.6. I-Loops under different beam search size.

4.6.5 RQ2.4: Overheads

Table 4.6 shows the average overhead of `NMTSloth` in generating a test input, we report only the overhead of `NMTSloth` because the comparison baselines cannot degrade NMT systems' efficiency. The measured overhead of `NMTSloth` is rather reasonable (ranging from 7.5s to 106.35s) and may increase linearly as the perturbation size increases. The linearly increasing overheads are due to the fact that `NMTSloth` is an iterative approach (iteration number equals to ϵ), and the overhead within each iteration is stable. Note that such reasonable overhead is not a concern since perturbed test inputs are generated by `NMTSloth` offline.

Table 4.6. Average overheads of `NMTSloth` (s).

Method	ϵ	H-NLP	AllenAI	T5	Average
`NMTSloth` (C)	1	11.40	21.14	18.50	17.01
	2	31.80	44.66	45.59	40.68
	3	59.76	69.56	74.48	67.93
`NMTSloth` (T)	1	7.50	18.45	22.62	16.19
	2	31.41	39.48	61.86	44.25
	3	62.50	62.54	100.01	75.02
`NMTSloth` (S)	1	10.52	39.19	6.73	18.81
	2	23.33	75.21	17.45	38.66
	3	38.93	106.35	27.71	57.66

Answers to **RQ2.4**: The overheads of `NMTSloth` are reasonable and may increase linearly as the perturbation size increase. Specifically, when $\epsilon = 1$, `NMTSloth` costs 17.01, 16.19, and 18.81 seconds to generate character-level, token-level, and structure-level test samples.

4.7 Discussion

In this section, we further present a real-world case study to discuss how NMT systems' efficiency degradation will impact real-world devices' battery power. After that, we show how developers could apply `NMTSloth` to improve NMT systems' efficiency robustness and mitigate computational resource waste. Finally, we discuss potential threats that might threaten the applicability of `NMTSloth` and how we alleviate them.

4.7.1 Real-World Case Study

Table 4.7. Input sentences for experiments on mobile devices.

Seed Input	Death comes often to the soldiers and marines who are fighting in anbar province, which is roughly the size of louisiana and is the most intractable region in iraq.
Test Input	Death comes often to the soldiers and marines who are fighting in anbar province, which is roughly the (size of of louisiana and is the most intractable region in iraq.

Experiment Settings. We select Google T5 as our evaluation NMT model in this case study. We first deploy the model on Samsung Galaxy S9+, which has 6GB RAM and a battery capacity of 3500 mAh. After that, we select one sentence from the dataset `ZH19` as our seed input; we then apply `NMTSloth` to perturb the seed input with character-level perturbation and obtain the corresponding test sample. The seed sentence and the corresponding test

sample are shown in Table 4.7, where the perturbation is colored in red. Notice the test sample inserts only one character in the seed sentence. This one-character perturbation is very common in the real world due to user's typo for instance. Finally, we feed the seed input and test sample to the deployed NMT system and measure the mobile device's battery consumption rate.

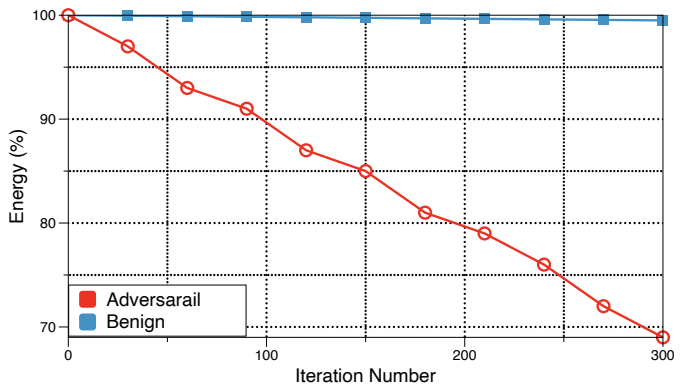

Figure 4.7. Remaining battery power of the mobile device after T5 translating seed and perturbed sentences.

Experiment Results. The mobile phone's battery consumption status is shown in Figure 4.7. The red line is for the perturbed input, and the blue one is for the original seed input. The results show that the perturbed input consumes the mobile's battery power significantly more quickly than the seed input. Specifically, after 300 iterations, the perturbed input consumes 30% of the battery power, while the seed input consumes less than 1%. The results demonstrate the vulnerability of the efficiency degradation for mobile devices. Recall that the perturbed example used in our experiment only inserts one character in the seed sentence, which would mimic many practical scenarios (e.g., typo). Thus, the results suggest the criticality and the necessity of improving NMT systems' efficiency robustness.

4.7.2 Mitigation.

This section shows how developers leverage `NMTSloth` to develop runtime abnormal input detector, which mitigates possible efficiency degradation and computational waste under the adversary scenario (*e.g.,* DOS attack). In detail, we propose a approach to filter out test inputs that require abnormal computational resources at runtime. Because the abnormal inputs are forced to quit at early stage, thus the computational resources waste are avoided. The idea of applying input validation to improve DNNs correctness robustness has been studied in recent works (Wang et al., 2020a, 2019). However, existing input validation techniques may not be suitable for improving NMT systems efficiency robustness due to the high overheads. Our intuition is that although normal inputs and the computational resource heavy inputs look similar in human eyes, the latent representations of these two categories of inputs are quite different (Wang et al., 2020a). Thus, we can leverage the latent representations of these two category inputs to train a light-weighted SVM classifier and apply the classifier to distinguish abnormal inputs at runtime. Because the classifier should be light-weighted, getting each input's latent representations is preferable without additional computations. As we introduced in Chapter 2, NMT systems run the encoder once and only once for each input sentence to get the hidden state (*i.e., h* in Figure 2.4), we propose to use the output of the encoder as the latent representation to train a lighted-weighted SVM classifier.

Experimental Setup. For each NMT system in our evaluation, we randomly choose 1,000 seed inputs and apply `NMTSloth` to generate 1,000 abnormal inputs for each perturbation types. We randomly select 80% of the seed inputs and the abnormal inputs as the training data to train the SVM classifier, and use the rest 20% for testing. We run the trained SVM classifier on the testing dataset and measure the detectors' AUC score, extra computation overheads.

Table 4.8. The accuracy and extra overheads of the detector.

Subject	Metric (%)	Perturbation Type			
		NMTSloth (C)	NMTSloth (T)	NMTSloth (S)	Mixed
H-NLP	Acc	99.98	99.99	99.98	99.98
	AUC	100.00	100.00	100.00	100.00
	Overheads	0.17	0.32	0.18	0.74
	Energy	0.09	0.17	0.12	0.48
AllenAI	Acc	100.00	100.00	87.00	98.00
	AUC	100.00	100.00	98.32	100.00
	Overheads	0.17	0.08	0.49	0.86
	Energy	0.11	0.05	0.30	0.79
T5	Acc	99.97	100.00	99.99	100.00
	AUC	100.00	100.00	100.00	100.00
	Overheads	0.08	0.06	0.03	0.18
	Energy	0.05	0.04	0.02	0.11

Experimental Results. The experimental results are shown in Table 4.8. Each column in Table 4.8 represents the performance in detecting one specific perturbation type and "Mixed" represents the performance in detecting a mixed set of three perturbation types. We observe that the proposed detector achieves almost perfect detection accuracy with a lowest accuracy of 87.00%. Moreover, the proposed detector's overheads and energy consumption are negligible compared to those incurred under the NMT system. All experimental subjects' extra overheads and the energy consumption are merely at most 1% of the original NMT systems' overheads in translation normal sentences. The results show that our validation-based approach can effectively filter out the abnormal input sentences with negligible overheads.

4.7.3 Threat Analyses.

Our selection of the three NMT systems, namely, Google T5, AllenAI WMT14, and H-NLP, might threaten the *external validity* of our experimental conclusions. We alleviate this threat by the following efforts: (1) the three NMT systems are very popular and have been widely used among developers (with more than 592,793 downloadings in Jan 2022); (2) their underlying DNN models are state-of-the-art models; (3) these systems differ from each other by diverse topics (*e.g.,* model architecture, language, training corpus, training

process) Therefore, our experimental conclusions should generally hold, although specific data could be inevitably different for other subjects. Our *internal threat* mainly comes from our definition of different perturbation types. Our introduced perturbation may not always be grammatically correct (*e.g.,* inserting one character may result in an unknown token). However, such perturbations may not be typical but exist in the real-world (*e.g.,* user typos, adversarial manner). Thus, it is meaningful to understand NMT systems efficiency degradation with such realistic perturbations. Moreover, all three perturbation types are well studied in related works (Ebrahimi et al., 2018; Zou et al., 2020; Ebrahimi et al., 2018; Zhang et al., 2021; Ren et al., 2019; Zang et al., 2020; He et al., 2020; Sun et al., 2020; Gupta et al., 2020; He et al., 2021).

4.8 Conclusions

In this work, we study the efficiency robustness of NMT systems. Specifically, we propose `NMTSloth`, a framework that introduces imperceptible perturbations to perturb seed inputs to reduce NMT systems' computation efficiency. Evaluation on three public available NMT systems shows that `NMTSloth` can generate effective test inputs that may significantly decrease NMT systems' efficiency.

CHAPTER 5

AN EFFICIENT APPROACH FOR PERFORMANCE TESTING OF DYNAMIC NEURAL NETWORKS [1]

Chapter 4 proposes a method to generate test inputs that trigger the efficiency of ML software at the *model* architecture level. This chapter aims to further improve the approach proposed in Chapter 4 to generate test inputs more efficiently.

Section 5.2 introduces the overview of this chapter. Section 5.3 presents our preliminary study, and Section 5.4 details our designed approach. Section 5.5 and Section 5.6 showcases our empirical evaluation of the proposed approach. Section 5.7 discusses the threats in our evaluation, and Section 5.8 concludes this chapter.

5.1 Author Contributions

Simin Chen proposed the approach, conducted the experiments, and authored the majority of the paper. Mirazul were responsible for writing the background sections. Cong Liu and Wei Yang contributed to the writing and subsequent revisions of the paper.

5.2 Overview

Deep Neural Networks (DNNs) have shown potential in many applications, such as image classification, image segmentation, and object detection (Chan et al., 2016; Vaswani et al., 2017a; Huang et al., 2018). However, the power of using DNNs comes at substantial computational costs (Najibi et al., 2019; Veit and Belongie, 2018; Wu et al., 2019; Hua et al., 2019; Liu and Deng, 2018a). Costs, especially *inference time* cost, can be a concern for

[1] ©2022 ACM. Reprinted, with permission, from Simin Chen, Mirazul Haque, Cong Liu, Wei Yang. "DeepPerform: An Efficient Approach for Performance Testing of Resource-Constrained Neural Networks". In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022). DOI:10.1145/3551349.3561158.

deploying DNNs on resource-constrained embedded devices such as mobile phones and IoT devices. To enable the deployment of DNN on devices with limited resources, researchers propose a series of Dynamic Neural Networks (DyNNs) (Wan et al., 2020; Bateni and Liu, 2018; Jiang et al., 2021; Wang et al., 2021; Davis and Arel, 2014; Gao et al., 2018). DyNNs selectively activate partial computation units (*e.g.,* convolution layer, fully connected layer) for different inputs rather than whole units for computation. The partial unit selection mechanism enables DyNNs to achieve real-time prediction on resource-constrained devices.

Similarly to traditional systems (Xiao et al., 2013), performance bottlenecks also exist in DyNNs. Among the performance bottlenecks, some of them can be detected only when specific input values are given. Hence, these problems are referred to as input-dependent performance bottlenecks (IDPBs). Some IDPBs will cause severe performance degradation and result in catastrophic consequences. For example, consider an DyNN deployed on a drone for obstacle detection. If DyNNs' energy consumption increases five times suddenly for specific inputs, it will cause the drone to take out of battery in the middle of a trip. Because of these reasons, conducting performance testing to find IDPB is a crucial step before DyNNs' deployment process.

However, to the best of our knowledge, most of the existing work for testing neural networks are mainly focusing on correctness testing, which can not be applied to performance testing. The main difference between correctness testing and performance testing is that correctness testing aims to detect models' incorrect classifications; while the performance testing is to find IDPBs that trigger performance degradation. Because incorrect classifications may not lead to performance degradation, existing correctness testing methods can not be applied for performance testing. To fill this gap and accelerate the process of deploying neural networks on resource-constrained devices, there is a strong need for an automated performance testing framework to find IDPBs.

We identify two main challenges in designing such a performance testing framework. First, traditional performance metrics (*e.g.,* latency, energy consumption) are hardware-dependent

metrics. Measuring these hardware-dependent metrics requires repeated experiments because of the system noises. Thus, directly applying these hardware-dependent metrics as guidelines to generate test samples would be inefficient. Second, DyNNs' performance adjustment strategy is learned from datasets rather than conforming to logic specifications (such as relations between model inputs and outputs). Without a logical relation between DyNNs' inputs and DyNNs' performance, it is challenging to search for inputs that can trigger performance degradation in DyNNs.

To address the above challenges, we propose `DeepPerform`, which enables efficient performance testing for DyNNs by generating test samples that trigger IDPBs of DyNNs (`DeepPerform` focuses on the performance testing of latency degradation and energy consumption degradation as these two metrics are critical for performance testing (Bateni and Liu, 2020; Wan et al., 2020)). To address the first challenge, we first conduct a preliminary study (Section 5.3) to illustrate the relationship between computational complexity (FLOPs) and hardware-dependent performance metrics (latency, energy consumption). We then transfer the problem of degrading system performance into increasing DyNNs' computational complexity (Equation (5.1)). To address the second challenge, we apply the a paradigm similar to Generative Adversarial Networks (GANs) to design `DeepPerform`. In the training process, `DeepPerform` learns and approximates the distribution of the samples that require more computational complexity. After `DeepPerform` is well trained, `DeepPerform` generates test samples that activate more redundant computational units in DyNNs. In addition, because `DeepPerform` does not require backward propagation during the test sample generation phase, `DeepPerform` generates test samples much more efficiently, thus more scalable for comprehensive testing on large models and datasets.

To evaluate `DeepPerform`, we select five widely-used model-dataset pairs as experimental subjects and explore following four perspectives: *effectiveness*, *efficiency*, *coverage*, and *sensitivity*. First, to evaluate the effectiveness of the performance degradation caused by

test samples generated by DeepPerform, we measure the increase in computational complexity (FLOPs) and resource consumption (latency, energy) caused by the inputs generated by DeepPerform. To measure efficiency, we evaluated the online timeoverheads and total timeoverheads of DeepPerform in generating different scale samples for different scale experimental subjects. For coverage evaluation, we measure the computational units covered by the test inputs generated by DeepPerform. For sensitivity measurement, we measure how DeepPerform's effectiveness depends on the DyNN's configurations and hardware platforms. The experimental results show that DeepPerform generated input increases DyNNs' computational FLOPs by up to 552%, with 6-10 millisecond overhead for generating a test sample. We summarize our contribution as follows.

- **Approach.** We propose a learning-based approach [2], namely DeepPerform, to learn the distribution to generate the test samples for performance testing. Our novel design enables generating test samples more efficiently, thus enable scalable performance testing.

- **Evaluation.** We evaluate DeepPerform on five DyNN models and three datasets. The evaluation results suggest that DeepPerform finds more severe diverse performance bugs while covering more DyNNs' behaviors, with only 6-10 milliseconds of online overheads for generating test inputs.

- **Application.** We demonstrate that developers could benefit from DeepPerform. Specifically, developers can use the test samples generated by DeepPerform to train a detector to filter out the inputs requiring high abnormal computational resources (§5.6).

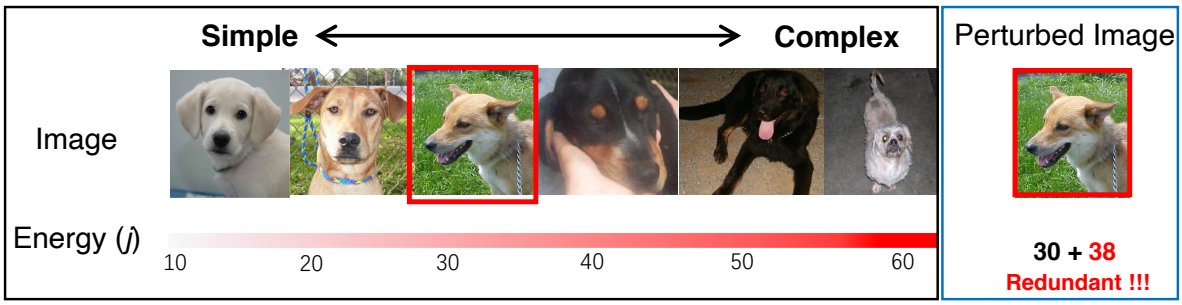

Figure 5.1. Left box shows that DyNNs allocate different computational resources for images with different semantic complexity; right box shows that perturbed image could trigger redundant computation and cause energy surge.

Table 5.1. Experiential subject and model performance.

Subject		FLOPs			CPU (Quad-Core ARM® Cortex®-A57 MPCore)						GPU (NVIDIA Pascal™ GPU architecture with 256 cores)					
					Latency			Energy			Latency			Energy		
Dataset	Model	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
CIFAR10 (C10)	SkipNet (SN)	195.44	248.62	336.99	0.44	0.51	0.63	65.76	76.60	316.44	0.74	0.94	1.39	168.07	245.62	439.38
	BlockDrop (BD)	72.56	180.51	228.27	0.11	0.23	0.37	15.89	34.17	161.12	0.13	0.33	0.71	29.60	73.27	282.59
CIFAR100 (C100)	DeepShallow (DS)	38.68	110.47	252.22	0.04	0.11	0.25	3.47	15.32	37.81	0.09	0.37	1.08	12.63	75.49	441.60
	RaNet (RN)	31.50	41.79	188.68	0.07	0.21	2.96	8.21	27.99	448.96	0.10	0.36	5.81	15.87	60.22	997.73
SVHN	DeepShallow (DS)	38.74	161.40	252.95	0.04	0.16	0.27	3.99	23.35	91.28	0.03	0.37	0.82	4.16	78.66	180.39

5.3 Preliminary Study

5.3.1 Study Approach

Our intuition is to explore the worst computational complexity of an algorithm or model. For DyNNs, the basic computation are the floating-point operations (FLOPs). Thus, we made an assumption that the FLOPs count of an DyNN is a hardware-independent metric to approximate DyNN performance. To validate such an assumption, we conduct an empirical study. Specifically, we compute the *Pearson Product-moment Correlation Co-efficient* (PCCs) (Rice, 2006) between DyNN FLOPs against DyNN latency and energy consumption. PCCs are widely used in statistical methods to measure the linear correlation between two variables. PCCs are normalized covariance measurements, ranging from -1 to 1. Higher PCCs

[2]https://github.com/SeekingDream/DeepPerform

indicate that the two variables are more positively related. If the PCCs between FLOPs against system latency and system energy consumption are both high, then we validate our assumption.

5.3.2 Study Model & Dataset

We select subjects (*e.g.,* model,dataset) following policies below.

- The selected subjects are publicly available.
- The selected subjects are widely used in existing work.
- The selected dataset and models should be diverse from different perspectives. *e.g.,*, the selected models should include both early-termination and conditional-skipping DyNNs.

We select five popular model-dataset combinations used for image classification tasks as our experimental subjects. The dataset and the corresponding model are listed in Table 5.1. We explain the selected datasets and the corresponding models in the following.

Datasets. CIFAR-10 (Krizhevsky et al., 2009) is a database for object recognition. There is a total of ten object classes for this dataset, and the image size of the image in CIFAR-10 is 32×32. CIFAR-10 contains 50,000 training images and 10,000 testing images. CIFAR-100 (Krizhevsky et al., 2009) is similar to CIFAR-10 (Krizhevsky et al., 2009) but with 100 classes. It also contains 50,000 training images and 10,000 testing images. SVHN (Netzer et al., 2011) is a real-world image dataset obtained from house numbers in Google Street View images. There are 73257 training images and 26032 testing images in SVHN.

Models. For CIFAR-10 dataset, we use SkipNet (Wang et al., 2018) and BlockDrop (Wu et al., 2018) models. SkipNet applies reinforcement learning to train DNNs to skip unnecessary blocks, and BlockDrop trains a policy network to activate partial blocks to save computation costs. We download trained SkipNet and BlockDrop from the authors' websites. For the

Table 5.2. PCCs between FLOPs against latency and energy.

Hardware	Metric	SN_C10	RN_C100	BD_C10	DS_C100	DS_SVHN
CPU	**Latency**	0.68	0.67	0.93	0.99	0.95
	Energy	0.65	0.64	0.93	0.98	0.95
GPU	**Latency**	0.48	0.56	0.91	0.99	0.97
	Energy	0.53	0.64	0.91	0.99	0.97

CIFAR-100 dataset, we use RaNet (Yang et al., 2020) and DeepShallow (Kaya et al., 2019) models for evaluation. DeepShallow adaptive scales DNN depth, while RaNet scales both input resolution and DNN depth to balance accuracy and performance. For SVHN dataset, DeepShallow (Kaya et al., 2019) is used for evaluation. For RaNet (Yang et al., 2020) and DeepShallow (Kaya et al., 2019) architecture, the author does not release the trained model weights but open-source their training codes. Therefore, we follow the authors' instructions to train the model weights.

5.3.3 Study Process

We begin by evaluating each model's computational complexity on the original hold-out test dataset. After that, we deploy the DyNN model on an Nvidia TX2 and measure latency and energy usage. Through Table 5.1, we present the FLOPs, latency, and energy consumption of each DyNN. We observe that the model would cost a different number of FLOPs for different test samples, and the variance between each test sample could be significant. For example, for the CIFAR-100 dataset and the RaNet model, the minimum FLOPs are 31.5M, while the maximum FLOPs are 188.68M.

5.3.4 Study Results

From the PCCs results in Table 5.2, we have the following observations: (i) The PCCs are more than 0.48 for all subjects. The results imply that FLOPs are positively related to latency

Table 5.3. System availability under performance degradation.

Subject	Original	Perturbed	Ratio
SN_C10	10,000	6,332	0.6332
BD_C10	10,000	4,539	0.4539
RN_C100	10,000	5,232	0.5232
DS_C100	10,000	3,576	0.3576
DS_SVHN	10,000	4,145	0.4145

and energy consumption in DyNNs (Rice, 2006). Especially for DS_C100, the PCC achieves 0.99, which indicates the strong linear relationship between FLOPs and runtime performance. (ii) The PCCs for the same subject on different hardware devices are remarkably similar (*e.g.,*, with an average difference of 0.04). According to the findings, the PCCs between FLOPs and latency/energy consumption are hardware independent. The statistical observations of PCCs confirm our assumption; that is, the FLOPs of DyNN handling an input is a hardware-independent metric that can approximate DyNN performance on multiple hardware platforms.

5.3.5 Motivating Example

To further understand the necessity of conducting performance testing for DyNNs, we use one real-world example to show the harmful consequences of performance degradation. In particular, we use `TorchMobile` to deploy each DyNN model on Samsung Galaxy S9+, an Android device with 6GB RAM and 3500mAh battery capacity. We randomly select inputs from the original test dataset of each subject (*i.e.,* Table 5.1) as seed inputs and perturb the selected seed inputs with random perturbation. Next, we conduct two experiments (one on the selected seed inputs and another one on the perturbed one) on the phone with the same battery. Specifically, we feed both datasets into DyNN for object classification and record the number of inputs successfully inferred before the battery runs out (We set the initial battery as the battery that can infer 10,000 inputs from the original dataset). The results are

shown in Table 5.3, where the column "original" and "perturbed" show the number of inputs successfully inferred, and the column "ratio" shows the corresponding system availability ratio (*i.e.,* the system can successfully complete the percentage of the assigned tasks under performance degradation). Such experimental results highlight the importance of DyNN performance testing before deployment. Otherwise, DyNNs' performance degradation will endanger the deployed system's availability.

5.4 Methodology

In this section, we introduce the detail design of `DeepPerform`.

5.4.1 Performance Test Samples for DyNNs

Following existing work (Haque et al., 2020b; Lemieux et al., 2018), we define performance test samples as the inputs that require redundant computation and cause performance degradation (*e.g.,* higher energy consumption). Because our work focus on testing DyNNs, we begin by introducing redundant computation in DyNNs. Like traditional software, existing work (Kaya et al., 2019; Haque et al., 2020b) has shown that redundant computation also exist in DyNNs. Formally, let $g_f(\cdot)$ denotes the function that measures the computational complexity of neural network $f(\cdot)$, and $T_I(\cdot)$ denotes a semantic-equivalent transformation in the input domain. As the example in Figure 5.1, $T_I(\cdot)$ could be changing some unnoticeable pixels in the input images. If $g_f(T_I(x_i)) > g_f(x_i)$ and $f(x_i)$ are correctly computed, then there exist redundant computations in the model $f(\cdot)$ that handles $T_I(x_i)$. In this paper, we consider unnoticeable perturbations as our transformations $T_I(\cdot)$, the same as the existing work (Carlini and Wagner, 2017; Jia and Liang, 2017; Haque et al., 2020b). Finally, we formulate our objective to generate performance test samples as searching such unnoticeable

input transformation $T_I(\cdot)$, as shown in Equation (5.1).

$$g(T_I(x)) >> g(x)$$
$$T_I(x) = \{x + \delta(x) | \quad ||\delta(x)||_p \leq \epsilon\} \tag{5.1}$$

5.4.2 `DeepPerform` Framework

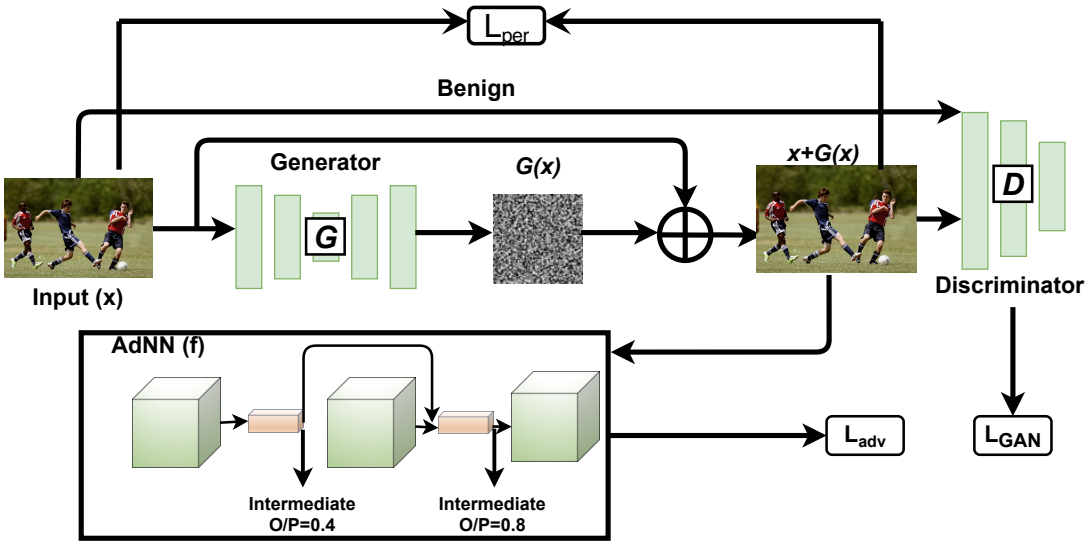

Figure 5.2. Design overview of `DeepPerform`.

Figure 5.2 illustrates the overall architecture of `DeepPerform`, which is based on the paradigm of Generative Adversarial Networks (GANs). GANs mainly consist of a generator $\mathcal{G}(\cdot)$ and a discriminator $\mathcal{D}(\cdot)$. The input x of the generator $\mathcal{G}(\cdot)$ is a seed input and the output $\mathcal{G}(x)$ is a minimal perturbation (*i.e.,* $\delta(x)$ in Equation (5.1)). After applying the generated perturbation to the seed input, the test sample $T_I(x) = x + \mathcal{G}(x)$ is sent to the discriminator. The discriminator $\mathcal{D}(\cdot)$ is designed to distinguish the generated test samples $x + \mathcal{G}(x)$ and the original samples x. After training, the generator would generate more unnoticeable perturbation, correspondingly, the discriminator would also be more accurate in distinguishing original samples and generated samples. After being well trained, the

discriminator and the generator would reach a Nash Equilibrium, which implies the generated test samples are challenging to be distinguished from the original samples.

$$\mathcal{L}_{GAN} = \mathbb{E}_x log\mathcal{D}(x) + \mathbb{E}_x log[1 - \mathcal{D}(x + \mathcal{G}(x))] \tag{5.2}$$

The loss function of the Generative Adversarial Networks (GANs) can be formulated as Equation 5.2. In Equation 5.2, the discriminator $\mathcal{D}$ tries to distinguish the generated samples $\mathcal{G}(x) + x$ and the original sample x, so as to encourage the samples generated by $\mathcal{G}$ close to the distribution of the original sample.

However, the perturbation generated by $\mathcal{G}$ may not be able to trigger performance degradation. To fulfil that purpose, we add target DyNN $f(\cdot)$ into the `DeepPerform` architecture. During training $\mathcal{G}(\cdot)$, the generated input is fed to DyNN to create an objective function that will help increase the DyNN FLOP consumption. To generate a perturbation that triggers performance degradation in DyNNs, we incorporate two more loss functions other than $\mathcal{L}_{GAN}$ for training $\mathcal{G}(\cdot)$. As shown in Equation (5.1), to increase the redundant computation, the first step is to model the function $g_f(\cdot)$. According to our statistical results in §5.3, FLOPs could be applied as a hardware-independent metric to approximate DyNNs system performance. Then we model $g_f(\cdot)$ as Equation (5.3).

$$g_f(x) = \sum_{i=1}^{N} W_i \times \mathbb{I}(B_i(x) > \tau_i) \tag{5.3}$$

Where W_i is the FLOPs in the i^{th} block, $B_i(x)$ is the probability that the i^{th} block is activated, $\mathbb{I}(\cdot)$ is the indicator function, and τ_i is the pre-set threshold based on available computational resources.

$$\mathcal{L}_{adv} = \ell(g_f(x),\ \sum_{i=1}^{N} W_i) \tag{5.4}$$

To enforce $\mathcal{G}$ could generate perturbation that trigger IDPB, we define our performance degradation objective function as Equation 5.4. Where ℓ is the Mean Squared Error. Recall

$\sum_{i=1}^{N} W_i$ is the status that all blocks are activated, then $\mathcal{L}_{adv}$ would encourage the perturbed input to activate all blocks of the model, thus triggering IDPBs.

$$\mathcal{L}_{per} = \mathbb{E}_x ||\mathcal{G}(x)||_p \tag{5.5}$$

To bound the magnitude of the perturbation, we follow the existing work (Carlini and Wagner, 2017) to add a loss of the L_p norm of the semantic-equivalent perturbation. Finally, our full objective can be denoted as

$$\mathcal{L} = \mathcal{L}_{GAN} + \alpha\mathcal{L}_{adv} + \beta\mathcal{L}_{per} \tag{5.6}$$

Where α and β are two hyper-parameters that balance the importance of each objective. Notice that the goal of the correctness-based testing methods' objective function is to maximize the errors while our objective function is to maximize the computational complexity. Thus, our objective function in Equation (5.6) can not be replaced by the objective function proposed in correctness-based testing (Carlini and Wagner, 2017; Pei et al., 2017; Tian et al., 2018).

5.4.3 Architecture Details

In this section, we introduce the detailed architecture of the generator and the discriminator. Our generator $\mathcal{G}$ adapts the structure of encoder-decoder, and the architecture of the discriminator is a convolutional neural network. The architectures of the generator and the discriminator are displayed in Figure 5.3.

Generator. As shown in Figure 5.3, there are three main components in the generator, that is, the Encoder, the ResBlocks, and the Decoder. The Encoder repeats the convolutional blocks twice, a convolutional block includes a convolutional layer, a batch normalization layer, and a RELU activation layer. After the encoding process, the input would be smaller in size but with deep channels. The ResBlock stacks four residual blocks (Han et al., 2015), which

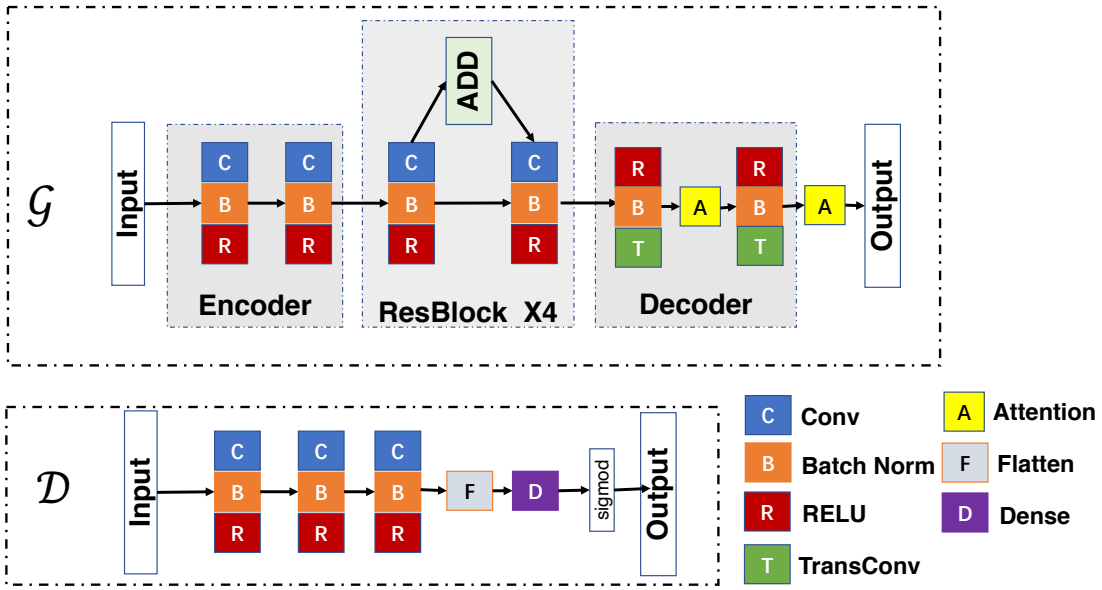

Figure 5.3. Architecture of the generator and discriminator.

is widely used to avoid the gradient vanishing problem. The Decoder is the reverse process of the Encoder, the transpose convolutional layer is corresponding to the convolutional layer in the Encoder. After the decoding process, the intermediate values will be changed back to the same size as the original input to ensure the generated perturbation to be applied to the original seed input.

Discriminator. The architecture of the discriminator is simpler than the generator. There are three convolutional blocks to extract the feature of the input, after that, following a flatten layer and a dense layer for classification.

5.4.4 Training `DeepPerform`

The training of `DeepPerform` is comprised of two parts: training the discriminator $\mathcal{D}$ and training the generator $\mathcal{G}$. Algorithm 2 explains the training procedure of the `DeepPerform`. The inputs of our algorithm include the target DyNNs $f(\cdot)$, perturbation constraints ϵ, training dataset $\mathcal{X}$, hyper-parameters α, β and max epochs T. The outputs of our training algorithm include a well-trained generator and discriminator. First, the algorithm constructs

Algorithm 2 Training `DeepPerform`.

Input: The subject DyNNs $f(\cdot)$ to be tested.
Input: Perturbation Constraints ϵ, Perturbation norm p.
Input: Training dataset $\mathcal{X}$.
Input: Hyper-parameters α, β.
Input: Maximum training epochs T.
Output: Generator $\mathcal{G}$ and Discriminator $\mathcal{D}$.

1: $g_f(\cdot)$ = ModelPerformance(f) {Construct g_f through Equation 5.3.}
2: **for** epoch in range(0, T) **do**
3: **for** batch in $\mathcal{X}$ **do**
4: $\overline{x} = \mathcal{G}(x) + x$ {generate test samples}
5: $\overline{x}$ = CLIP($\overline{x}, x, p, \epsilon$) {clip test samples}
6: $\mathcal{L}_{GAN}$+ = ComputeGanLoss($\overline{x}$, x, $\mathcal{D}$) {Equation 5.2}
7: $\mathcal{L}_{per}$+ = ComputePerLoss($\overline{x}$, x) {Equation 5.5}
8: $\mathcal{L}_{adv}$+ = ComputeAdvLoss($\overline{x}$, x) {Equation 5.4}
9: **end for**
10: $\nabla\mathcal{G}$ = ComputeGrad($\mathcal{L}_{GAN} + \mathcal{L}_{per} + \mathcal{L}_{adv}$) {Compute $\mathcal{G}$ gradient}
11: $\nabla\mathcal{D}$ = ComputeGrad($\mathcal{L}_{GAN}$) {Compute $\mathcal{D}$ gradient}
12: $\mathcal{G} = \mathcal{G} + \nabla\mathcal{G}$, $\mathcal{D} = \mathcal{D} + \nabla\mathcal{D}$ {Update the weights of $\mathcal{D}$ and $\mathcal{G}$}
13: **end for**

the performance function $g(\cdot)$ through Equation 5.3 (Line 1). Then we run T epochs. For each epoch, we iteratively select small batches from the training dataset (Line 2, 3). For each seed x in the selected batches, we generate test sample $\overline{x}$ and compute the corresponding loss through Equation (5.2), Equation (5.5), Equation (5.4) (Line 6-8). We compute the gradients of $\mathcal{G}$ and $\mathcal{D}$ with the computed loss (Line 10, 11), then we update the weights of $\mathcal{G}$ and $\mathcal{D}$ with the gradients (Line 12). The update process is performed iteratively until the maximum epoch is reached.

5.5 Evaluation

We evaluate `DeepPerform` and answer the following questions:

- ***RQ1 (Efficiency)***: How efficiently does `DeepPerform` generate test samples?
- ***RQ2 (Effectiveness)***: How effective can `DeepPerform` generate test samples that degraded DyNNs' performance?
- ***RQ3 (Coverage)***: Can `DeepPerform` generate test samples that cover DyNNs' more computational behaviors?

- ***RQ4 (Sensitivity)***: Can `DeepPerform` behave stably under different settings?
- ***RQ5 (Quality)***: What is the semantic quality of the generated test inputs, and how does it relate to performance degradation?

5.5.1 Experimental Setup

Experimental Subjects

We select the five subjects used in our preliminary study (Section 5.3) as our experimental subjects. As we discussed in Section 5.3, the selected subjects are widely used, open-source, and diverse in working mechanisms.

Comparison Baselines

Almost all existing DNN testing work focuses on correctness testing. To our knowledge, `ILFO` (Haque et al., 2020b) is the state-of-the-art approach to generate inputs to increase DyNNs computational complexity. Furthermore, `ILFO` has proved that its backward-propagation approach is more effective and efficient than the traditional symbolic execution (*i.e.,* SMT); thus, we compare our method to `ILFO`. `ILFO` iteratively applies the backward propagation to perturb seed input to generate test input. However, the high overheads of iterations make `ILFO` a time-consuming approach for generating test samples. Instead of iterative backward computation, `DeepPerform` learns the DyNNs' computational complexity in the training step. After `DeepPerform` is trained, `DeepPerform` applies forward propagation once to generate one test sample.

Experiment Process

We conduct an experiment on the five selected subjects and used the results to answer all five RQ. The experimental process can be divided into test sample generation and performance testing procedures.

Test Sample Generation. For each experimental subject, we split train/test datasets according to the standard procedure(Krizhevsky et al., 2009; Netzer et al., 2011). Next, we train `DeepPerform` with the corresponding training datasets. The training is conducted on a Linux server with three Intel Xeon E5-2660 v3 CPUs @2.60GHz, eight 1080Ti Nvidia GPUs, and 500GB RAM, running Ubuntu 14.04. We configure the training process with 100 maximum epochs, 0.0001 learning rate, and apply early stopping techniques (Yao et al., 2007). We set the hyper-parameter α and β as 1 and 0.001, as we observe $\mathcal{L}_{per}$ is about three magnitude larger than $\mathcal{L}_{adv}$. After training `DeepPerform`, we randomly select 1,000 inputs from the original test data set as seed input. Then, we feed the seed input into `DeepPerform` to generate test inputs ($x + \mathcal{G}(x)$ in Figure 5.2) to trigger DyNNs' performance degradation. In our experiments, we consider both L_2 and L_{inf} perturbations (Carlini and Wagner, 2017) and train two version of `DeepPerform` for input generation. After training `DeepPerform`, we apply the clip operation (Li et al., 2019) on $x + \mathcal{G}(x)$ to ensure that the generated test sample satisfies the semantic constraints in Equation (5.1).

Performance Testing Procedure. For the testing procedure, we select Nvidia Jetson TX2 as our main hardware platform (We evaluate `DeepPerform` on different hardware in Section 5.5.5). Nvidia Jetson TX2 is a popular and widely-used hardware platform for edge computing, which is built around an Nvidia Pascal-family GPU and loaded with 8GB of memory and 59.7GB/s of memory bandwidth. We first deploy the DyNNs on Nvidia Jetson TX2. Next, we feed the generated test samples (from `DeepPerform` and baseline) to DyNNs, and measure the response latency and energy consumption (energy is measured through Nvidia power monitoring tool). Finally, we run DyNNs at least ten times to infer each generated test sample to ensure the results are accurate.

RQ Specific Configuration. For RQ1, 2 and 3, we follow existing work (Madry et al., 2018; Haque et al., 2020b; Athalye et al., 2018) and set the maximum perturbations as 10 and 0.03 for L_2 and L_{inf} norm separately for our approach and baselines. We then carried

out experiments in Section 5.5.6 to study how different maximum perturbations would affect performance degradation. `ILFO` needs to configure the maximum iteration number and the balance weight, we set the maximum iteration number at 300 and the balance weight as 10^{-6}, as suggested by the authors (Haque et al., 2020b). DyNNs require a configurable parameter/threshold to decide the working mode. Different working modes have different trade-offs between accuracy and computation costs. In our deployment experiments (RQ2), we follow the authors (Haque et al., 2020b) to set the threshold as 0.5 for all the experimental DyNNs, and we evaluate how different threshold will affect `DeepPerform` effectiveness in Section 5.5.5. In addition, to ensure that the available computational resources are the same, we run only the DyNNs application on the system during our performance testing procedure.

5.5.2 Efficiency

In this section, we evaluate the efficiency of `DeepPerform` in generating test samples compared with selected baselines.

Metrics. We record the *online time overheads* of the test sample generation process (overheads of running $\mathcal{G}$ to generate perturbation), and use the mean online time overhead (s) as our evaluation metrics. A lower time overhead implies that it is more efficient, thus better in generating large-scale test samples. Because `DeepPerform` requires training the generator $\mathcal{G}(\cdot)$, for a fair comparison, we also evaluate the *total time overheads* ($\mathcal{G}(\cdot)$ training + test samples generation) of generating different scale numbers of test inputs.

Online Overheads. The average time overheads of generating one test sample are shown in Figure 5.4. The results show that `DeepPerform` costs less than 0.01s to generate a test sample under all experimental settings. In contrast, `ILFO` requires 27.67-176.9s to generate one test sample. The findings suggest that given same time budget, `DeepPerform` can generate 3952-22112× more inputs than existing method. Another interesting observation is that the overheads of `ILFO` fluctuate among different subjects, but the overheads of `DeepPerform`

remain relatively constant. The reason is that the overheads of DeepPerform mainly come from the inference process of the generator, while the overheads of ILFO mainly come from backward propagation. Because backward propagation overheads are proportional to model size (*i.e.*,, a larger model demands more backward propagation overheads), the results of ILFO show a significant variation. The overhead of DeepPerform is stable, as its overheads have no relation to the DyNN model size. The result suggests that when testing large models, ILFO will run into scalability issues, whereas DeepPerform will not.

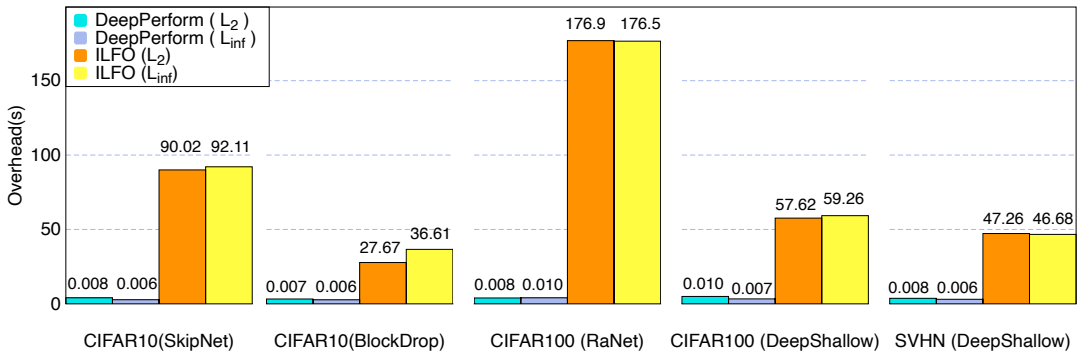

Figure 5.4. Online overheads to generate one test sample (s).

Total Overheads. The total time overheads of generating various scale test samples are shown in Figure 5.5. We can see from the results that ILFO is more efficient than DeepPerform when the number of generated test samples is minimal (less than 200). However, when the number of generated test samples grows greater, the overall time overheads of DeepPerform are significantly lower than ILFO. To create 1000 test samples for SN_C10, for example, ILFO will cost five times the overall overheads of DeepPerform. Because the overhead of ILFO is determined by the number of generated test samples (Haque et al., 2020b), the total overhead increases quickly as the number of generated test samples increases. The main overhead of DeepPerform, on the other hand, comes from the GAN model training instead of test sample generation. As a result, the generation of various scale numbers of test samples will have no substantial impact on the total overhead of DeepPerform. The results imply that ILFO is not

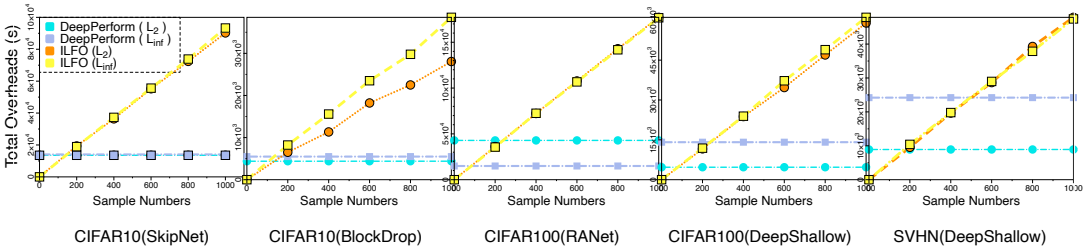

Figure 5.5. Total overheads of generating different scale test samples (s).

scalable to test DyNNs with large datasets, while `DeepPerform` does an excellent job. We also notice that the `DeepPerform`'s overheads for L_2 and L_{inf} are different for DN_SVHN. Because we use the early stopping method (Yao et al., 2007) to train `DeepPerform`, we can explain such variation in overheads. In detail, the objective L_{per} differs for L_2 and L_{inf}. Thus, the training process will end at different epochs.

5.5.3 Effectiveness

Relative Performance Degradation

Metrics. To characterize system performance, we choose hardware-independent and hardware-dependent metrics. Our hardware-independent metric is floating-point operations (FLOPs). FLOPs are widely used to assess the computational complexity of DNNs (Wu et al., 2018; Wang et al., 2018). Higher FLOPs indicate higher CPU utilization and lower efficiency performance. As for hardware-dependent metrics, we focus on latency and energy consumption because these two metrics are essential for real-time applications (Bateni and Liu, 2020; Wan et al., 2020). After characterizing system performance with the above metrics, We measure the *increment* in the above performance metrics to reflect the severity of performance degradation. In particular, we measure the increased percentage of flops *I-FLOPs*, latency (*I-Latency*) and energy consumption (*I-Energy*) as our performance degradation severity evaluation metrics.

Equation (5.7) shows the formal definition of our degradation severity evaluation metrics. In Equation (5.7), x is the original seed input, δ is the perturbation generated, and $F_f(\cdot), L_f(\cdot), E_f(\cdot)$ are the functions that measure FLOPs, latency, and energy consumption of DyNN $f(\cdot)$. A test sample is more effective in triggering performance degradation if it increases a higher percentage of FLOPs, latency, and energy consumption. We examine two scenarios for each evaluation metric: the *average* metric value for the whole test dataset and the *maximum* metric value caused for a particular sample. The first represents long-term performance degradation, whereas the second depicts performance degradation in the worst-case scenario. We measure the energy consumption using TX2's power monitoring tool.

$$
\begin{aligned}
I - FLOPs(x) &= \frac{F_f(x+\delta) - F_f(x)}{F_f(x)} \times 100\% \\
I - Latency(x) &= \frac{L_f(x+\delta) - L_f(x)}{L_f(x)} \times 100\% \\
I - Energy(x) &= \frac{E_f(x+\delta) - E_f(x)}{E_f(x)} \times 100\%
\end{aligned}
\tag{5.7}
$$

Table 5.4. The FLOPs increment of the test samples (%).

Norm	Subject	Mean		Max	
		baseline	ours	baseline	ours
L_{inf}	**SN_C10**	6.43	**31.14**	18.43	**62.77**
	BD_C10	**48.44**	38.39	162.58	**188.60**
	RN_C100	133.67	**181.57**	**498.29**	**498.99**
	DS_C100	116.19	**157.66**	287.98	**552.00**
	DS_SVHN	115.99	**228.32**	**498.29**	**498.29**
L_2	**SN_C10**	20.34	**31.30**	30.43	**82.09**
	BD_C10	**48.44**	38.39	162.58	**188.60**
	RN_C100	133.67	**182.12**	**498.99**	**498.99**
	DS_C100	116.19	**157.66**	287.98	**552.00**
	DS_SVHN	115.99	**228.32**	**498.29**	**498.29**

Hardware-independent experimental results are listed in Table 5.4. As previously stated, greater I-FLOPs implies that the created test samples demand more FLOPs, which will result in significant system performance reduction. The table concludes that `DeepPerform` generates test samples that can cause more severe performance degradation. Other than that, we have multiple observations. First, for four of the five subjects, `DeepPerform` generates test

Table 5.5. The performance degradation on two hardware platforms (%).

Device	Subject	L2								Linf							
		I-Latency				I-Energy				I-Latency				I-Energy			
		Mean		Max		Mean		Max		Mean		Max		Mean		Max	
		baseline	ours	baseline	ours	baseline	ours	baseline	ours	baseline	ours	baseline	ours	baseline	ours	baseline	ours
CPU	SN_C10	8.2	**25.4**	20.9	**45.7**	8.3	**25.7**	20.6	**44.9**	5.7	**30.9**	15.1	**46.1**	5.7	**31.4**	15.6	**45.8**
	BD_C10	**28.7**	17.5	**142.1**	132.5	**28.9**	17.7	**148.2**	135.3	25.4	**25.6**	**143.9**	135.7	25.8	**26.1**	148.2	**141.0**
	RN_C100	**72.2**	39.9	**1654.4**	624.1	**72.5**	40.3	**1685.1**	633.7	53.6	**141.1**	370.2	**1313.1**	54.1	**144.3**	387.1	**1341.1**
	DS_C100	61.4	**133.8**	216.2	**464.0**	64.6	**142.6**	217.3	**471.8**	52.0	**171.5**	254.5	**483.6**	54.9	**180.1**	282.0	**503.9**
	DS_SVHN	29.8	**210.1**	392.3	**1496.1**	30.3	**214.6**	398.4	**1467.7**	70.2	**257.2**	1371.2	**1580.8**	71.7	**260.1**	1372.8	**1548.2**
GPU	SN_C10	4.4	**14.3**	6.8	**17.9**	5.3	**15.4**	8.1	**19.7**	4.4	**11.8**	4.8	**15.7**	5.2	**12.4**	6.1	**15.9**
	BD_C10	9.3	**9.8**	**53.6**	41.6	10.2	**11.7**	**59.0**	42.4	13.9	**16.9**	39.9	**41.1**	15.1	**20.2**	46.4	**46.6**
	RN_C100	**90.6**	51.0	**1968.5**	923.9	**96.9**	55.4	**2446.5**	1043.4	66.9	**167.2**	454.8	**1496.8**	70.6	**197.4**	557.9	**1837.4**
	DS_C100	56.1	**102.7**	184.9	**370.2**	62.8	**116.4**	194.1	**478.8**	71.7	**158.6**	183.8	**384.7**	80.3	**177.9**	217.1	**457.8**
	DS_SVHN	11.5	**75.9**	149.7	**244.2**	15.8	**92.0**	172.3	**298.3**	38.7	**72.0**	280.0	**308.9**	47.4	**88.0**	348.3	**382.8**

samples that require more FLOPs, *e.g.,* 31. 14%-62. 77% for SN_C10. Second, for both L_2 and L_{inf} perturbations, the model would require more FLOPs, and the difference between L_2 and L_{inf} settings is minimal. Third, the maximum FLOPs are far greater than the average case for some extreme scenarios, *e.g.,* for DS_SVHN, and DS_C100. The hardware-dependent experimental results are listed in Table 5.5. Similar to hardware-independent experiments, `DeepPerform` outperforms `ILFO` on 65 out of 80 comparison scenarios. However, for the other 15 comparisons, we explain the results as the following two reasons: (i) the noise of the system has affected the results because for almost all scenarios `DeepPerform` has been able to increase more I-FLOPs than `ILFO`. (ii) recall in Table 5.2, RN_C100 has the PCCs around 0.64, and the FLOPs increment of RN_C100 for `DeepPerform` and `ILFO` is around the same level. Thus, `DeepPerform` can cause slightly lower latency and energy consumption degradation than `ILFO`. However, for SN_C10, although it has low PCCs, `DeepPerform` can increase much more FLOPs than `ILFO`, thus, `DeepPerform` can cause a more severe performance degradation. Based on the results in Table 5.5, we conclude that `DeepPerform` outperforms the baseline in creating inputs that consume more energy or time.

Absolute Performance Degradation

Besides the relative performance degradation, we also investigate the absolute performance degradation of the generated inputs. In Figure 5.6, we plot the unnormalized efficiency

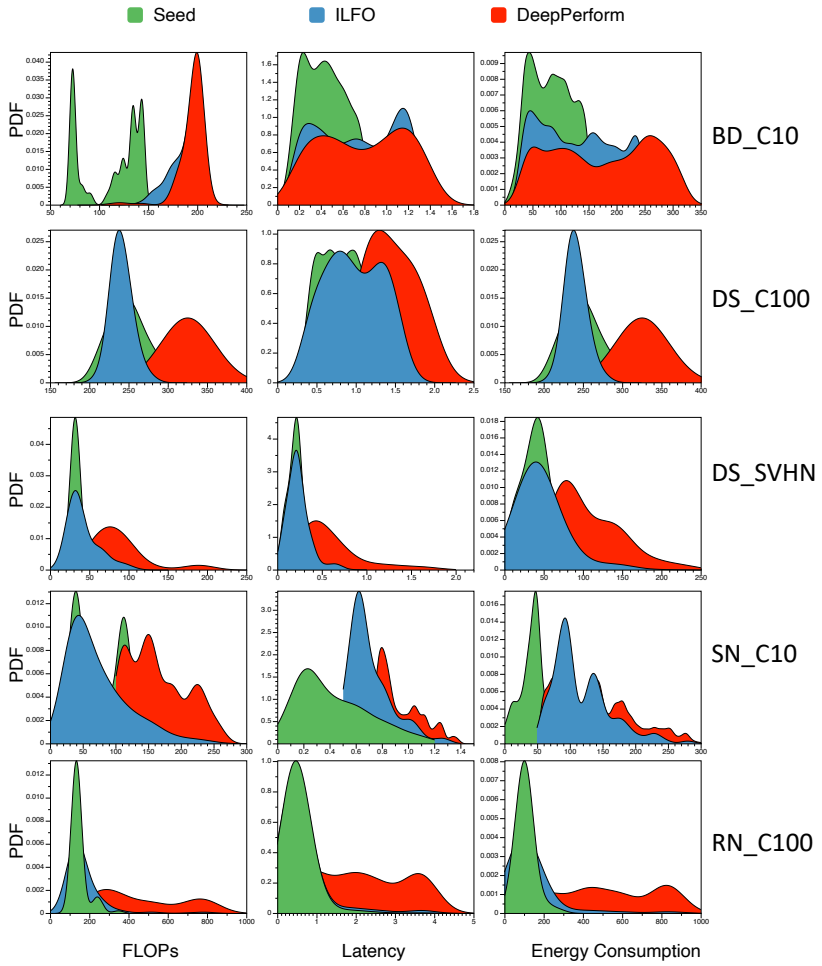

Figure 5.6. The unnormalized efficiency distribution of seed inputs and the generated inputs.

distribution (*i.e.,* FLOPs, latency, energy consumption) of both seed and generated inputs to characterize the absolute performance degradation. We specifically depict the probability distribution function (PDF) curve (James et al., 2013) of each efficiency metric under discussion. The unnormalized efficiency distribution is shown in Figure 5.6, where the green curve is for the seed inputs, and the red curve is for the test inputs from `DeepPerform`. From the results, we observe that `DeepPerform` is more likely to generate test inputs located at the right end of the x-axis. Recall that a PDF curve with more points on the right end of the x-axis is more likely to generate theoretical worst-case test inputs. The results confirm that `DeepPerform` is more likely to generate test inputs with theoretical worst-case complexities.

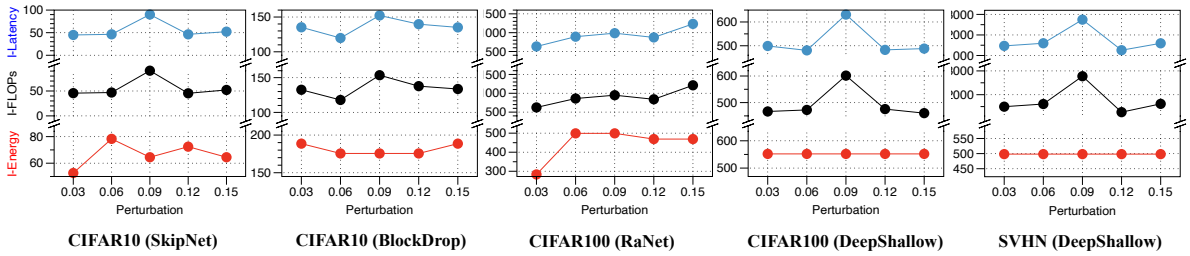

Figure 5.7. How performance degradation as perturbation constraints change.

Test Sample Validity

To measure the validity of the generated test samples, we define *degradation success number* η in Equation (5.8),

$$\eta = \sum \mathbb{I}(FLOPs(x+\delta) \geq FLOPs(x)). \qquad x \in \mathcal{X} \tag{5.8}$$

where $\mathcal{X}$ is the set of randomly selected seed inputs and $\mathbb{I}(FLOPs(x+\delta) > FLOPs(x))$ indicates whether generated test samples require more computational resources than the seed inputs. We run `DeepPerform` and baselines the same experimental time and generate the different number of test samples ($\mathcal{X}$ in Equation (5.8)), we then measure η in the generated test samples. For convince, we set the experimental time as the total time of `DeepPerform` generating 1,000 test samples (same time for `ILFO`). From the third column in Table 5.6, we observe that for most experimental settings, `DeepPerform` achieves a higher degradation success number than `ILFO`. Although `ILFO` is an end-to-end approach, the high overheads of `ILFO` disable it to generate enough test samples.

5.5.4 Coverage

In this section, we investigate the comprehensiveness of the generated test inputs. In particular, we follow existing work (Pei et al., 2017; Zhang et al., 2018) and investigate the diversity of the DyNN behaviors explored by the test inputs generated by `DeepPerform`. Because DyNNs'

Table 5.6. Validity and coverage results.

Norm	Subject	η (#) ours	η (#) baseline	Cov (%) ours	Cov (%) baseline
Linf	SN_C10	**842**	69	**0.74 ± 0.001**	0.65 ± 0.001
	BD_C10	**630**	84	**0.37 ± 0.001**	0.37 ± 0.001
	RN_C100	**871**	133	**0.99 ± 0.002**	0.89 ± 0.030
	DS_C100	**646**	69	**1.00 ± 0.000**	0.83 ± 0.016
	DS_SVHN	**916**	220	**1.00 ± 0.000**	0.92 ± 0.033
L2	SN_C10	**993**	81	0.84 ± 0.001	**0.85 ± 0.001**
	BD_C10	**732**	79	**0.41 ± 0.001**	0.40 ± 0.001
	RN_C100	**924**	229	0.94 ± 0.007	**0.95 ± 0.013**
	DS_C100	**734**	181	**1.00 ± 0.000**	**1.00 ± 0.000**
	DS_SVHN	**924**	518	**0.98 ± 0.025**	0.73 ± 0.034

behavior relies on the computation of intermediate states (Pei et al., 2017; Ma et al., 2018), we analyze how many intermediate states are covered by the test suite.

$$Cov(\mathcal{X}) = \frac{\sum_{x \in \mathcal{X}} \sum_{i=1}^{N} \mathbb{I}(B_i(x) > \tau_i))}{N} \tag{5.9}$$

To measure the coverage of DyNNs' intermediate states, we follow existing work (Pei et al., 2017) and define decision *block coverage* ($Cov(\mathcal{X})$ in Equation (5.9)), where N is the total number blocks, $\mathbb{I}(\cdot)$ is the indicator function, and $(B_i(x) > \tau_i))$ represents whether i^{th} block is activated by input x (the definition of B_i and τ_i are the same with Equation (2.1) and Equation (2.2)). Because DyNNs activate different blocks for decision making, then a higher block coverage indicates the test samples cover more decision behaviors. For each subject, we randomly select 100 seed samples from the test dataset as seed inputs. We then feed the same seed inputs into `DeepPerform` and `ILFO` to generate test samples. Finally, we feed the generated test samples to DyNNs and measure the block coverage. We repeat the process ten times and record the average coverage and the variance. The results are shown in Table 5.6 last two columns. We observe that the test samples generated by `DeepPerform` achieve higher coverage for almost all subjects.

5.5.5 Sensitivity

In this section, we conduct two experiments to show that `DeepPerform` can generate effective test samples under different settings.

Configuration Sensitivity. DyNNs require configuring the threshold τ_i to set the accuracy-performance tradeoff mode. In this section, we evaluate whether the test samples generated from `DeepPerform` could degrade the DyNNs' performance under different modes. Specifically, we set the threshold τ_i in Equation (2.1) and Equation (2.2) as $0.3, 0.4, 0.5, 0.6, 0.7$ and measure the maximum FLOPs increments. Notice that we train `DeepPerform` with $\tau_i = 0.5$ and test the performance degradation with different τ_i. The maximum FLOPs increment ratio under different system configurations are listed in Table 5.7. For all experimental settings, the maximum FLOPs increment ratio keeps a stable value (*e.g.,* 79.17-82.91, 175.59-250.00). The results imply that the test samples generated by `DeepPerform` can increase the computational complexity under different configurations, and the maximum FLOPs increment ratio is stable as the configuration changes.

Table 5.7. Increment under different thresholds.

Norm	Subject	Threshold				
		0.3	**0.4**	**0.5**	**0.6**	**0.7**
L2	**SN_C10**	79.17	82.91	82.91	75.00	70.00
	BD_C10	250.00	250.00	175.59	175.59	175.59
	RN_C100	500.00	498.99	498.99	200.00	200.00
	DS_C100	600.00	600.00	552.00	400.00	200.00
	DS_SVHN	498.29	498.29	498.29	498.29	400.00
Linf	**SN_C10**	66.67	78.26	82.91	66.67	73.91
	BD_C10	233.33	175.59	175.59	233.33	233.33
	RN_C100	498.99	498.99	498.99	498.99	498.99
	DS_C100	552.00	552.00	552.00	400.00	300.00
	DS_SVHN	498.29	498.29	498.29	498.29	400.00

Hardware Sensitivity. We next evaluate the effectiveness of our approach on different hardware platforms. In particular, we select Intel Xeon E5-2660 V3 CPU and Nvidia 1080 Ti

as our experimental hardware platforms and measure the maximum performance degradation ratio on those selected platforms. The test samples generated by `DeepPerform`, as shown in Table 5.8, cause severe and stable runtime performance degradation on different hardware platforms. As a result, we conclude that `DeepPerform` is not sensitive to hardware platforms.

5.5.6 Quality

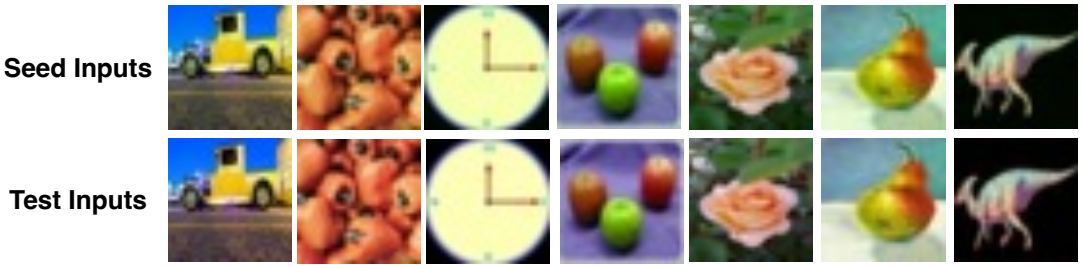

Figure 5.8. Testing inputs generated by `DeepPerform`.

We first conduct quantitative evaluations to evaluate the similarity between the generated and seed inputs. In particular, we follow existing work (Carlini and Wagner, 2017) and compute the perturbation magnitude. The perturbation magnitude are listed in Table 5.9. Recall that we follow existing work (Carlini and Wagner, 2017; Haque et al., 2020b) and set the perturbation constraints ϵ as 10 and 0.03 for L_2 and L_{inf} norm (Section 5.5.1). From the results in Table 5.9, we conclude that generated test samples can satisfy the semantic-equivalent constraints in Equation (5.1). Moreover, we conduct a qualitative evaluation. In particular, we randomly select seven images from the generated images for RA_C100 and visualize them in Figure 5.8 (more results are available on our website), where the first row is the randomly selected seed inputs, and the second row is the corresponding generated inputs. The visualization results show that the test inputs generated by `DeepPerform` are semantic-equivalent to the seed inputs. Furthermore, we investigate the relationship between different semantic-equivalent constraints and performance degradation. We first

change the perturbation magnitude constraints (*i.e.,* η in Equation (5.1)) and train different models (experiment results for L_2 norm could be found on our websites). After that, we measure the severity of DyNN performance degradation under various settings. Figure 5.7 shows the results. We observe that although the relationship between performance degradation ratio and perturbation magnitude is not purely linear, there is a trend that the overhead increases with the increase of perturbation magnitude.

5.6 Application

This section investigates if developers can mitigate the performance degradation bugs using the existing methods for mitigating DNN correctness bugs (*i.e.,* adversarial examples). We focus on two of the most widely employed approaches: offline adversarial training (Goodfellow et al., 2015), and online input validation (Wang et al., 2020b). Surprisingly, we discover that not all of the two approaches can address performance faults in the same manner they are used to repair correctness bugs.

5.6.1 Adversarial Training

Setup. We follow existing work (Goodfellow et al., 2015) and feed the generated test samples and the original model training data to retrain each DyNN. The retraining objective can be formulated as

$$\mathcal{L}_{retrain} = \underbrace{\ell(g_f(x'),\ g_f(x))}_{\mathcal{L}_1} + \underbrace{\beta\left\{\ell(f(x), y) + \ell(f(x'), y)\right\}}_{\mathcal{L}_2} \tag{5.10}$$

where x is one seed input in the training dataset, $x' = \mathcal{G}(x) + x$ is the generated test input, $f(\cdot)$ is the DyNNs, and $g_f(\cdot)$ measures the DyNNs computational FLOPs. Our retraining objective can be interpreted as forcing the buggy test inputs x' to consume the same FLOPs as the seed one (*i.e.,* $\mathcal{L}_1$), while producing the correct results (*i.e.,* $\mathcal{L}_2$). For each DyNN model under test, we retrain it to minimize the objective in Equation (5.10). After retraining, we test each DyNNs accuracy and efficiency on the hold-out test dataset.

Results. Table 5.10 shows the results after model retraining. The two columns on the left show the performance degradation before and after model retraining, while the two columns on the right show the model accuracy before and after model retraining. The findings show that following model training, the I-FLOPs fall; keep in mind that a higher I-FLOPs signifies a more severe performance degradation. Thus, the decrease in I-FLOPs implies that model retraining can help overcome performance degradation. However, based on the data in the right two columns, we observe that such retraining, different from accuracy-based retraining, may harm model accuracy.

5.6.2 Input Validation

Input validation (Wang et al., 2020b) is a runtime approach that filters out abnormal inputs before DyNNs cast computational resources on such abnormal inputs. This approach is appropriate for situations where the sensors (*e.g.,* camera) and the decision system (*e.g.,* DyNN) work at separate frequencies. Such different frequency working mode is very common in robotics systems (Luo and Yuille, 2019; Feichtenhofer et al., 2019; Zolfaghari et al., 2018), where the DyNN system will randomly select one image from continuous frames from sensors since continuous frames contain highly redundant information. Our intuition is to filter out those abnormal inputs at the early computational stage, the same as previous work (Wang et al., 2020b).

Design of Input Filtering Detector. Our idea is that although seed inputs and the generated test inputs look similar, the latent representations of these two category inputs are quite different (Wang et al., 2020b). Thus, we extract the hidden representation of a given input by running the first convolutional layer of the DyNNs. First, we feed both benign and `DeepPerform` generated test inputs to specific DyNN. We use the outputs of the first convolutional layer as input to train a linear SVM to classify benign inputs and inputs that require huge computation. If any resource consuming adversarial input is detected, the

Table 5.8. Performance degradation on different hardware.

Norm	Subject	Intel Xeon E5-2660 v3 CPU I-Latency	Intel Xeon E5-2660 v3 CPU I-Energy	Nvidia 1080 Ti I-Latency	Nvidia 1080 Ti I-Energy
L2	**SN_C10**	36.95	36.20	24.94	50.77
	BD_C10	76.69	79.24	64.10	63.55
	RN_C100	1019.25	1173.21	938.21	856.46
	DS_C100	567.10	609.73	414.38	338.51
	DS_SVHN	236.12	246.70	311.01	282.09
Linf	**SN_C10**	29.38	28.28	24.95	11.94
	BD_C10	70.67	74.09	49.82	52.70
	RN_C100	319.72	355.29	679.79	652.98
	DS_C100	463.91	496.84	439.53	464.65
	DS_SVHN	232.88	244.91	263.49	141.56

Table 5.9. The perturbation size of the generated test inputs.

Norm	SN_C10	BD_C10	RN_C100	DS_C100	DS_SVHN
L2	9.48	9.47	9.50	9.48	9.62
Linf	0.03	0.03	0.03	0.03	0.03

Table 5.10. Efficiency and accuracy of DyNN model.

Metric		SN_C10	BD_C10	RN_C100	DS_C100	DS_SVHN
I-FLOPs	**before**	31.30	38.39	182.12	157.66	228.32
	after	8.07	15.26	35.37	28.54	38.65
Acc	**before**	92.34	91.35	65.43	58.78	94.54
	after	13.67	10.56	6.67	7.67	18.78

Table 5.11. Performance of SVM detector.

Subject	AUC L2	AUC Linf	Extra Latency (s) L2	Extra Latency (s) Linf	Extra Energy (j) L2	Extra Energy (j) Linf
SN_C10	0.9997	0.9637	0.0168	0.0167	1.8690	1.8740
BD_C10	0.9967	0.9222	0.0001	0.0002	0.0108	0.0197
RN_C100	1.0000	0.9465	0.0031	0.0042	0.3263	0.4658
DS_C100	0.5860	0.3773	0.0167	0.0212	1.8578	2.4408
DS_SVHN	1.0000	1.0000	0.0098	0.0210	1.1030	2.3959

inference is stopped. The computational complexity of the SVM detector is significantly less than DyNNs. Thus the detector will not induce significant computational resources consumption.

Setup. For each experimental subject, we randomly choose 1,000 seed samples from the training dataset, and apply `DeepPerform` to generate 1,000 test samples. We use these 2,000 inputs to train our detector. To evaluate the performance of our detector, we first randomly select 1,000 inputs from the test dataset and apply `DeepPerform` to generate 1000 test samples. After that, we run the trained detector on such 2,000 inputs and measure detectors' AUC score, extra computation overheads, and energy consumption.

Results. Table 5.11 shows that the trained SVM detector can successfully detect the test samples that require substantial computational resources. Specifically for L_2 norm perturbation, all the AUC scores are higher than 0.99. The results indicate that the proposed detector identifies L_2 test samples better. The last four columns show the extra computational resources consumption of the detector. We observe that the detector does not consume many additional computational resources from the results.

5.7 Threats To Validity

Our selection of five experimental subjects might be the *external threat* that threaten the generability of our conclusions. We alleviate this threat by the following efforts. (1) We ensure that the datasets are widely used in both academia and industry research. (2) All evaluated models are state-of-the-art DNN models (published in top-tier conferences after 2017). (3) Our subjects are diverse in terms of a varied set of topics: all of our evaluated datasets and models differ from each other in terms of different input domains (*e.g.,* digit, general object recognition), the number of classes (from 10 to 100), the size of the training dataset (from 50,000 to 73,257), the model adaptive mechanism. Our *internal threat* mainly comes from the realism of the generated inputs. We alleviate this threat by demonstrating

the relationship of our work with existing work. Existing work (Zhang et al., 2018; Zhou et al., 2020; Kurakin et al., 2017) demonstrates that correctness-based test inputs exist in the physical world. Because we formulate our problem(*i.e.,* the constraint in Equation (5.1)) the same as the previous correctness-based work (Zhou et al., 2020; Madry et al., 2018), we conclude our generated test samples are real and exist in the physical world.

5.8 Conclusion

In this paper, we propose `DeepPerform`, a performance testing framework for DNNs. Specifically, `DeepPerform` trains a GAN to learn and approximate the distribution of the samples that require more computational units. Through our evaluation, we have shown that `DeepPerform` is able to find IDPB in DyNNs more effectively and efficiently than baseline techniques.

CHAPTER 6

DYNAMIC NEURAL NETWORK COMPILATION VIA PROGRAM REWRITING AND GRAPH OPTIMIZATION [1]

Chapter 3 through Chapter 5 analyze and improve the efficiency of ML software at the *data* and *model* levels. This chapter aims to study the efficiency of ML software at the *program* level. Specifically, this chapter aims to improve the program efficiency optimizer (*i.e.,* ML compiler) to better support DyNN models.

Section 6.2 provides an overview of this chapter. Section 6.3 presents our study of the ML software program optimizer. Section 6.4 summarizes our idea to solving this problem, and Section 6.5 details our design. Section 6.6 presents our evaluation. In Section 6.7, we discuss the threats to our evaluation and how to mitigate them. Finally, in Section 6.8, we conclude this chapter.

6.1 Author Contributions

Simin Chen proposed the approach, conducted the experiments, and authored most of the paper. Shiyi Wei, Cong Liu and Wei Yang contributed to the writing and subsequent revisions of the paper.

6.2 Overview

With the growing popularity of deep learning(DL)-based applications, optimizing, executing, and deploying these applications becomes critical. DL compilers (Chen et al., 2018; Facebook, 2018; Microsoft, 2017; Google, 2017; Chen et al., 2022a; Zheng et al., 2020, 2022; Fegade et al.,

[1]

2021; Fang et al., 2021; Lyubomirsky, 2022) are fundamental infrastructures for achieving these goals, enabling DL application deployment on various hardware devices. DL compilers translate DL models written in high-level DL frameworks (*e.g.,* `PyTorch` (Paszke et al., 2019) and `TensorFlow` (Abadi et al., 2016)) into optimized and portable executables.

Over the past few years, significant advancements have been made in the development of DL compilers, aiming to streamline the deployment of neural networks on diverse hardware platforms (Chen et al., 2018; Facebook, 2018; Cyphers et al., 2018; Vanholder, 2016). DL compilers offer two key advantages. First, they enable the compiled DNN model to function as a executable program, eliminating the need for model developers to install resource-intensive DL frameworks on target platforms to parse and execute the DNN models. Second, DL compilers optimize the inference time overhead of a given DNN model, making it suitable for real-time applications on resource-constrained platforms such as mobile devices.

However, existing DL compilers heavily rely on the *tracing mechanism* (Chen et al., 2018; Microsoft, 2017), which requires providing a runtime input to a neural network program and tracing its execution path to generate the necessary computational graph for compilation. Unfortunately, these tracing mechanisms implicitly assume that a DNN model can be abstracted as a "static" computational graph, with a fixed execution path for computation. This assumption, however, does not hold for modern dynamic neural networks (DyNNs), where the execution path is determined by individual inputs and varies with each invocation (Davis and Arel, 2014; Gao et al., 2018; Najibi et al., 2019; Veit and Belongie, 2018; Wu et al., 2019; Hua et al., 2019). For instance, an encoder-decoder model used in neural machine translation (Chen et al., 2022) may require invoking the underlying decoder multiple times to produce translation outputs, without specifying the exact number of invocations.

To understand the limitations of the tracing mechanism employed by existing DL compilers when it comes to compiling dynamic neural networks, we conducted an empirical study utilizing two widely-used DL compilers (*i.e.,* `TVM` (Chen et al., 2018) and `OnnxRuntime` (Microsoft,

2017)) to compile four types of DyNNs. The results revealed a discrepancy between the outputs produced by running the compiled executables and the original DNN models within DL frameworks. This discrepancy clearly indicates that the DL compilers fail to accurately compile DyNNs.

To overcome the limitations of existing DL compilers, we introduce an automatic tool, `DyCL`, which assists developers in accurately compiling DyNNs automatically. Our primary objective is to enable the flexible adaptation of optimizations found in current DL compilers while ensuring the correct handling of the inherent dynamism present in DyNNs. The design of `DyCL` is driven by two key observations. First, we identify that the main source of DL compilers' inability to compile DyNNs lies in the presence of conditional statements within DyNN programs (*e.g.,* conditional statements). When a source program does not contain such conditional statements, a DyNN program effectively transforms into a regular DNN program, with a computational graph that can be determined statically. Second, we recognize that DL applications typically involve pre- and post-processing stages, such as image normalization and token mapping. Consequently, the compiled DNN executable cannot function as a standalone program. Instead, a host program (*i.e.,* Listing 2) is responsible for running both the pre- and post-processing stages, as well as invoking the DNN executable for inference. It is worth noting that the host program is often developed using a high-level programming language (*e.g.,* Java, C/C++, and Python), whose compiler is equipped to adequately handle dynamism.

Based on these observations, we design `DyCL` to move the dynamism of DyNN models to the host programs and reuse existing DL compilers to compile the sub-DNN models without dynamism. Specifically, `DyCL` coverts a DyNN into multiple standard neural networks (we refer to the separated neural networks as sub-DNNs) with conditional statements determining which sub-DNNs are invoked, and then apply the existing DL compilers to compile the sub-DNNs and incorporate the conditional statements into the host program.

We identify two challenges to correctly moving the dynamism from DyNN programs to the host programs. The first challenge is maintaining the essential contexts (*i.e.,* concrete model instance and model input shape) for compiling each sub-DNN. To address this challenge, our insight is that each sub-DNN is a "fixed" part and has no dependency on the DyNN inputs. In other words, we can obtain the concrete DNN instance for each sub-DNN by performing constant propagation. Motivated by such intuition, we developed a program rewriting engine (Section 6.5.2) that first conducts loop unrolling and then constant propagation to make each variable that has no dependency on the DyNN's input constant. As for maintaining the input tensor shape for each sub-DNN, our insight is that each sub-DNN input is the output of its predecessor sub-DNNs. Based on this insight, we introduced a novel concept called the Heterogeneous Control Flow Graph (HCFG) (Section 6.5.3), a special type of control flow graph, to model the dynamic behavior and the data flow of DyNNs. After that, we propose a novel traverse algorithm (Section 6.5.5) to traverse the HCFG, trace each sub-DNNs output shape, and use them as the input shapes for compiling the subsequent sub-DNNs. By doing so, we can successfully collect the necessary context to compile each sub-DNN.

The second challenge we encountered involves co-optimizing the host program and the compiled sub-DNNs to achieve enhanced optimization. Our insight for tackling this challenge stems from the observation that the host program and the compiled sub-DNNs are typically executed on different devices, such as CPU and GPU. Consequently, unnecessary overhead may arise due to data transfers between these devices. Thus, we can further optimize each sub-DNN and put the computation-free operations (*e.g.,* memory manipulation) on the host program to reduce the data transfer overheads. We then propose two strategies to identify the computation-free operations (*e.g.,* constant assignment and tensor copy) in each sub-DNN and move the operations from the computational graph of the sub-DNN to the corresponding host program to ensure semantic equivalence.

We conducted extensive experiments to evaluate `DyCL`. Specifically, we evaluate two open-source DL compilers, `TVM` (Chen et al., 2018) and `OnnxRuntime` (Microsoft, 2017), on nine

DyNN models, and we select two popular hardware platforms (*i.e.,* Nvidia TX2 and Nvidia AGX) as the backends. The selected DyNN models are diverse in terms of model architecture, model size, and application. The selected hardware backends are popular embedded platforms for deploying neural networks to assist system decision-making. We evaluate `DyCL` in terms of compilation correctness and acceleration. Moreover, we conduct an ablation study to understand the contribution of our proposed graph optimization module. The results show that `DyCL` can 100% correctly compile all dynamic neural networks (*i.e.,* the final decision after the post-processing of the compiled DyNN has a 100% consistent rate with the decision of the original DyNN model), and the maximum numeric error between the compiled DyNN and the original DyNN is around $10^{-4.72}$, significantly less than directly applying DL compiler to compile the DyNN (range from 10^{0} to 10^{4}). Moreover, the compiled executables run 1.12× to 20.21× faster than the original DyNNs running on the general-purpose DL frameworks, indicating the benefits of applying `DyCL` to deploy DyNN models. Finally, our ablation study shows that the proposed graph optimization module can further benefit the compilation process.

This paper made the following contributions.

- We conduct an empirical study to use two popular DL compilers (*i.e.,* `TVM` and `OnnxRuntime`) to compile four types of DyNNs. The study results illustrate the limitations of the existing DL compilers when compiling DyNNs.

- We present a program rewriting approach that allows adapting many existing DL compilers to compile DyNNs correctly. The key novelty of our approach is to identify and represent the dynamism of DyNN programs in heterogeneous control flow graphs. The sub-DNNs in HCFGs are compiled individually, and our approach generates a host API to call the compiled sub-DNNs.

- Based on the novel ideas, we implement `DyCL`; our evaluation results show `DyCL` can correctly compile nine DyNN models and accelerate the DyNN's inference time overheads range from 1.12× to 20.21×.

6.3 Empirical Study

In this section, we perform an empirical study to understand if existing DL compilers can correctly compile DyNN models.

6.3.1 Study Setup

Table 6.1. The DyNNs in our preliminary study.

DyNNs	Domain	Dynamic Mechanism	# of Branch	Github Star	Citation
Shallow-Deep	Image Classification	layer wise early stopping DNN	13	24	89
SkipNet	Image classification	block wise selective executation	53	208	355
En-Decoder	Image caption	token wise caption generation	50	1.2k	8863
AttentionNet	machine translation	token wise caption generation	200	1.2k	40711

Target DL Compilers. We target two DL compilers in this study: Apache `TVM` (Chen et al., 2018) and Microsoft `OnnxRuntime` (Microsoft, 2017), both of which are popular in academic research and industry work. `TVM` is a DL compiler for deploying DNNs on various platforms. It first converts a deep neural network into a static computational graph for high-level optimization and then generates hardware-specific code for each node in the transformed graph. On the other hand, `OnnxRuntime` is a runtime-based framework. It also converts a deep neural network into a graph representation for optimization. After that, `OnnxRuntime` maps the graph node to pre-compiled kernel operator functions.

Target DyNN Models. We selected DyNN models for our study using the following four criteria. The selected DyNN models *(i)* are the state-of-the-art models, which are published at top-tier conferences and outperform others according to their publication; *(ii)* are popularly used in work conducted by both academia and industry; *(iii)* differ from each other in terms

of input domains and dynamic mechanisms; *(vi)* are publicly available, and their code can be successfully executed.

Based on the above criteria, we selected four DyNNs from the survey by Huang et al. (Han et al., 2021b), listed in Table 6.1. Two of them are *energy-saving DyNNs* (Chen et al., 2022): Shallow-Deep and SkipNet (Kaya et al., 2019; Wang et al., 2018). Shallow-Deep is an early-termination DyNN that has multiple exits in a deep neural network. If one of the exits is confident about the prediction, the execution is stopped early. SkipNet is a conditional-skipping network that decides to skip or execute a DNN block based on the intermediate gate values. AttentionNet (Vaswani et al., 2017a) and En-Decoder (Xu et al., 2015) are *generative DyNNs*. AttentionNet is a Neural Machine Translation (NMT) model that uses an attention mechanism to draw dependencies between input and output. En-Decoder uses two different types of attention mechanisms to generate captions for images.

6.3.2 Study Process

To check whether existing DL Compilers can correctly compile the DyNNs, our intuition is that a correct compilation process should not change the semantics of the DyNNs. With this intuition, we compare the semantics of the original DyNNs and the compiled DyNNs. Specifically, we generate random inputs and feed the generated inputs to both the original DyNN and the compiled DyNN for inference and collect the outputs. Given the same inputs, the compiled and original DyNNs should produce identical outputs. Otherwise, the compiled DyNN is not semantically equivalent to the original DyNN, implying the compilation process fails.

Recall the original DyNN model first runs on a general-purpose DL framework (*e.g.,* `PyTorch` and `TensorFlow`) to train its parameters. Thus, we can use kernel functions from the general-purpose DL framework to launch the original DyNN, and we denote this program as the vendor program $\mathcal{V}(\cdot)$. We then follow each target DL compiler's documentation to

compile the original DyNN and generate the compiled executable $\mathcal{C}(\cdot)$. We randomly sample 1,000 inputs from a DyNN model's hold-out testing dataset as the test suite. We feed the test suite to both $\mathcal{V}(\cdot)$ and $\mathcal{C}(\cdot)$ and collect the outputs before and after the post-prepossess. We use the subscripts $before$ and $post$ to distinguish these two outputs. We then define two metrics to evaluate whether the target DL compiler can produce correct programs.

$$\begin{aligned} \delta &= log\left\{\max_{i=1}^{N} ||\mathcal{C}_{before}(x_i) - \mathcal{V}_{before}(x_i)|| + \epsilon\right\} \\ \eta &= \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}(\mathcal{V}_{post}(x_i) \neq \mathcal{C}_{post}(x_i)) \end{aligned} \tag{6.1}$$

Metrics. As shown in Equation (6.1), our first metric is *maximum numeric error* δ, which computes the maximum numeric error between the vendor and compiled DyNN programs outputs before the post-prepossess, where ϵ is set as 10^{-10} to avoid the division-by-zero issue. The DNN compilation process needs to perform some numeric matrix operations, and errors naturally exist in the matrix operations. Thus, numeric inconsistencies are inevitable in the DNN compilation process.

However, some numeric inconsistencies may not affect DyNN programs' final decision after post-processing. Thus, besides *maximum numeric error*, we propose *final inconsistent rate* η to measure the inconsistent rate of the post-prepossessed outputs between the vendor DyNN programs and the optimized DyNN programs, where $\mathbb{I}$ is the identity function, and it outputs 1 if the expression inside this function is evaluated to be true; otherwise 0. If the *final inconsistent rate* does not equal to 0, then it means that $\mathcal{C}$ will produce different outputs with $\mathcal{V}$ given the same inputs, implying the DL compiler changes the DyNN's semantics. For classification DyNNs, the post-processing step involves computing the predicted label. This is achieved by searching for the category with the highest confidence scores (He et al., 2016). For generation DyNNs, we set the post-prepossess as computing the generated sequences, which is done by searching the token that has the maximum likelihood among each output position (Vaswani et al., 2017a).

Table 6.2. The rate of inconsistency outputs predictions.

DL Compiler	Shallow-Deep	SkipNet	En-Decoder	AttentionNet	ResNet
OnnxRuntime	0.83	1.00	0.87	0.83	0.00
TVM	0.83	1.00	0.87	0.83	0.00

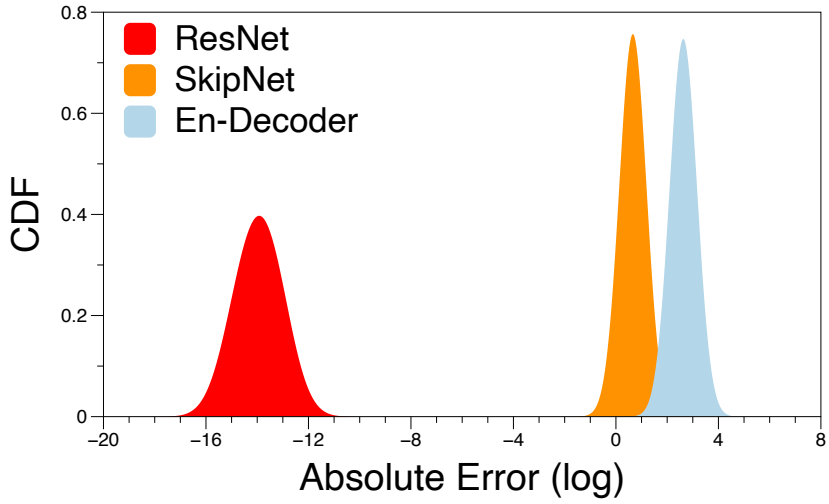

Figure 6.1. The error distribution of standard DNN and DyNNs.

Comparison Baselines. To show the compilation process is correct, we also compile a standard DNN program with no branches, ResNet50 (He et al., 2016), as the baseline.

6.3.3 Study Results

The number of inconsistent predictions from the compiled DL executable and the DL framework is shown in Table 6.2. The results demonstrate that the majority of randomly selected inputs (more than 80%) produce inconsistent outputs between vendor programs and compiled executables, implying that the produced DNN executable is semantically inequivalent to the original DyNN programs.

Failure Analysis. We attempted to comprehend the enormous amount of inconsistent predictions in Table 6.2. In particular, we would like to show that the inconsistency is caused by the design limitation of the DL compiler rather than the numerical errors in the compiler

optimization process. We visualize the error distribution of the outputs from $\mathcal{V}(\cdot)$ and $\mathcal{C}(\cdot)$. The error distribution is shown in Figure 6.1 (for better presentation, we only show the results of SkipNet and En-Decoder, more results could be found on our website), where the x-axis represents the absolute error and the y-axis represents the cumulative distribution function (CDF) (James et al., 2013). We observe that the absolute errors of ResNet are mostly located in the range of $[10^{-15},\ 10^{-13}]$. However, the errors of DyNNs are located in the range of $[10^{-1},\ 10^{4}]$, depending on the DyNN models. Such a significant error gap (*i.e.,* more than $10^{14}\times$) demonstrates that the discrepancy in DyNN is not due to numerical errors.

To gain further insight into the reasons for compilation failure, we conducted an analysis focusing on consistent outputs (it is worth noting that some DyNNs, as shown in Table 6.2, are capable of producing consistent outputs). In our analysis, we specifically tracked the execution traces of these inputs and compared them to the traces used during the compilation process of the DyNN. The DL compiler relies on tracing an example data to compile the model. During our investigation, we made a crucial observation: these traces were found to be identical. This observation can be attributed to a fundamental characteristic of existing DL compilers - their "static" nature of the tracing mechanism in the compilation process. Consequently, during the compilation process, the DL compiler disregards conditional branches (such as if statements) and generates a fixed computational graph based on the execution trace of the example input. As a result, the compiled DyNN executable utilizes the same execution path for all inputs, as dictated by the fixed computational graph. While it is possible for inputs to follow the same execution path as the example data, resulting in the compiled executable producing identical results to the original DyNN program, the likelihood of this occurring is exceptionally low. This is particularly evident in the case of SkipNet, which encompasses an astounding 2^{53} distinct execution paths.

Summary. Our experimental results confirm the limitations of existing DL compilers. Specifically, while some of these compilers do offer support for control flow, they struggle

```
def adaptive_forward_compile(self, x):
    # some code for tensor computation

    for g in range(3):
        for i in range(self.num_layers[g]):

            if g == 0 and i < self.num_layers[g] - 1:
                i = i + 1

            name = 'group{}_ds{}'.format(g + 1, i)
            if name in self.attr_layers:
                model = self.attr_layers[name]
                prev = model(prev)

            if mask == 0:
                x = (1 - mask).expand_as(prev) * prev
            else:
                ly_name = 'group{}_layer{}'.format(g+1,i)
                layer = self.attr_layers[ly_name]
                x = layer(x)
                x = mask.expand_as(x) * x
                cnt = cnt + 1
            prev = x
#### Some codes
```

Listing 6.1. An example to demonstrate the challenge of identifying sub-DNNs from the source code.

to correctly compile DyNNs due to their reliance on tracing mechanisms for obtaining computational graphs.

6.4 Challenges and Intuitions

DyNN model compilation imposes several unique challenges compared to standard "static" neural network model compilation. We enumerate two major challenges and provide a high-level idea of how `DyCL` addresses them.

Goal: Designing a general approach for various DL Compilers using different types of IRs. To enable existing DL compilers to effectively compile DyNNs, the initial step is to represent the dynamic behavior of DyNNs within these compilers. Furthermore, to fully leverage the capabilities of various optimization strategies present in these DL compilers, a

comprehensive approach needs to be designed for each compiler, taking into account their unique designs and characteristics.

Solution: Our solution is based on the following two observations. First, the dynamic behavior of DyNNs comes only from conditional statements (as shown in Figure 2.2). DyNNs can be separated into multiple standard neural networks (we refer to the separated neural networks as sub-DNNs) with conditional statements determining which sub-DNN to be invoked, and each sub-DNN can be correctly compiled by existing DL compilers. Second, a compiled DL program consists of an external library function and a host program. The host program is a program often implemented by a modern high-level programming language (*e.g.,* C/C++ and Python). These languages already have programming constructs to represent the dynamic behaviors (*i.e.,* conditional statements). Combing these two observations, our idea is to split the conditional statements from tensor computations in DyNNs, and then apply the existing DL compilers to compile the tensor computation part and incorporate the conditional statements into the host program.

Challenge 1: Maintaining the contexts for compiling sub-DNN. The first challenge is compiling each sub-DNN in the correct contexts. Recall that compiling a neural network needs two contexts: the DNN model instance and a DNN model input. Unfortunately, both contexts are not explicitly exhibited in the DyNN implementation. Listing 6.1 shows an example of a simplified implementation of SkipNet (Wang et al., 2018). From the code snippet, it is not easy to identify the necessary context for compilation because some contexts are represented in an implicit way. For example, an existing DL compile is unable to compile a basic block (e.g., lines 13-14) because the compiler is unable to obtain the concrete model instance `model` (*i.e.,* model is determined by the variable `name` as the statement in line 13 assigns `self.attr_layers[name]` to variable model) nor the input tensor shape of the model instance.

Intuition 1: To address this challenge, our observation is that the dynamic behavior of the DyNN model comes from its conditional `If` node. Removing the conditional `If` node, the rest

sub-DNNs are "fixed"; thus, sub-DNNs have fixed context and have no dependency on the DyNN inputs. In other words, we can obtain the concrete DNN instance for each sub-DNN by performing constant propagation and loop unrolling. As for the shape of each sub-DNN's input, our intuition is that we can start from the entry of the DyNN models' computational graph, iteratively compute the output shape of each node in the graph, and set the output shape as the input shape for the successor node.

Challenge 2: Co-optimization between the host program and tensor computation. Recall that the compiled DyNNs will be invoked as an API function in the host program. To ensure the semantic equivalence between the generated API and the original DyNNs, for each sub-DNNs, we need to track (i) the required input variable of the corresponding code snippet and (ii) the live variable after the corresponding code snippet, and then set these variables as the input/output of the sub-DNNs. However, the output variable of a sub-DNN may be a constant value or identical to one of the sub-DNN's input variables. Putting these variables to DL compilers to get acceleration on the accelerator (*e.g.,* GPU) may introduce unnecessary data transfer time. Thus, it calls for a co-optimization between the host program and the tensor computation.

Intuition 2: Our intuition is that DL compilers are designed to accelerate the "computing" for modern hardware platforms. We seek to reduce as many computation-free data transfer overheads as possible to accelerate the inference process. Based on this intuition, we propose two graph optimization strategies to further optimize the computational graph of each sub-DNN. Our optimization strategy will put computation-free data transfer left in the host program to reduce the data transfer time from the host program to the accelerator.

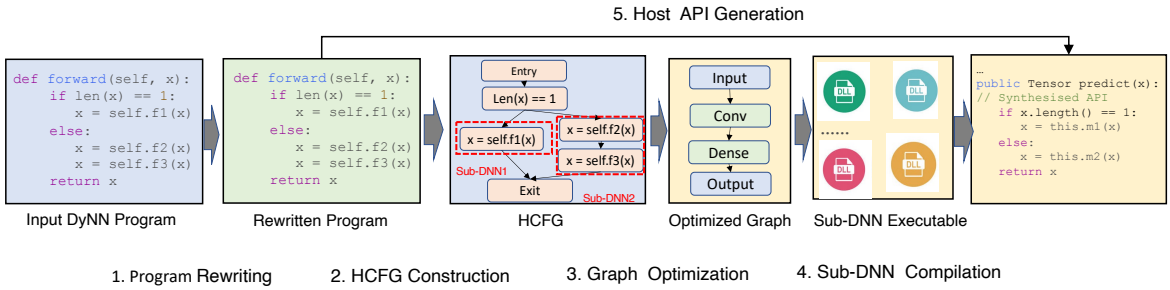

Figure 6.2. Design overview of DyCL.

6.5 Methodology

Given a DyNN model P_{DyNN}, our goal is to compile it and generate an optimized host API function P_{Host} that is semantically equivalent to the P_{DyNN}. Formally denoted as

$$P_{DyNN}(x) = P_{Host}(x) \qquad \forall x \in \mathcal{X} \tag{6.2}$$

For any inputs that belong to the program input domain, the host API will produce the same output as the original DyNN.

6.5.1 Design Overview

Figure 6.2 shows the design overview of our approach. The key insight of DyCL is to partition a DyNN into several sub-DNNs without dynamism for compilation and leave the dynamism part to the host program. Based on this insight, DyCL carries out the following five steps. ① *DyNN source program rewriting.* Given a DyNN program from the high-level DL framework (*e.g.,* PyTorch), DyCL first rewrites it and makes the context for each sub-DNN explicit. ② *HCFG Construction.* After rewriting the DyNN program, the next step is constructing a heterogeneous control flow graph (HCFG) to represent the logic conditional nodes and sub-DNN nodes. ③ *Sub-DNN Optimization.* In this step, we build a computational graph for each sub-DNN and propose two strategies to optimize the computational graph of each sub-DNN. ④ *Sub-DNN compilation.* After optimizing the computational graph of each

sub-DNN, we start from HCFG's entry node and iteratively compile the node in the HCFG to obtain a set of compiled DNN executables. ⑤ *Host API generation.* Finally, we modify the rewritten DyNN's abstract syntax tree (AST) and covert the modified AST back to a host API function.

Algorithm 3 HCFG Construction Algorithm.

Input: Rewritten DyNN program P_{DyNN}.
Output: HCFG of the input DyNN program.

```
1: CFG = ConstructCFG(P_DyNN) {get the CFG of P_DyNN}
2: HCFG = Dict() {Initlize HCFG as an dictionary}
3: for each N  in CFG.Nodes do
4:    s = N.statements[-1] {get the last statement of N}
5:    if s is logic conditional statement then
6:       s, N2 = BlockPartation(N) {partition node N}
7:       HCFG.update(s) {Add statement s to HCFG}
8:       N = N2. {Update N}
9:    end if
10:   HCFG.update(N) {Add node N to HCFG}
11: end for
```

6.5.2 DyNN Program Rewriting

Recall that the purpose of this step is to make the context for each sub-DNN explicit so that we can automatically compile each sub-DNN. To achieve this goal, we first conduct a loop unrolling process to get a program that contains no cycle in its CFG. As we introduced in Chapter 2, there is no cycle in the computational graph of the DyNN model. Thus, we need to unroll all loop statements in the original DyNN program. Recall that our tool focuses on DyNNs that do not contain input-dependent loops. This restriction is in place to avoid the potential introduction of infinite loops, as discussed in Chapter 2. Consequently, our loop unrolling technique is always feasible within this scope. After that, we perform constant propagation to make sure that all variables have no dependency on the DyNN's input constant. Consider the example provided in Listing 6.1. Excerpts of the rewritten

code are presented in Listing 6.2, showcasing the unrolling of the for loop statements from lines 4-5 and lines 9-10 of Listing 6.2. These correspond to the for loop statements found in lines 5-6 of Listing 6.1. Additionally, the if statement present in lines 8-9 of Listing 6.1 is eliminated following constant propagation.

```
1
2  def adaptive_forward_compile(self, x):
3      # some code for tensor computation
4      g = 0
5      i = 1
6      name = 'group1_ds1'
7      #### Some repeated code
8
9      g = 0
10     i = 2
11     name = 'group2_ds2'
12     #### Some repeated code
```

Listing 6.2. The code snippet of the DyNN after rewriting.

6.5.3 HCFG Construction

After rewriting the original DyNN program, the CFG of the program will be a DAG containing no cycle, and each variable that has no data dependency on DyNN's input will be presented as a constant.

Recall that our goal is to put the dynamic logic of DyNNs on the host program. To achieve this goal, we propose a heterogeneous control flow graph (HCFG), which is a special control flow graph (CFG). HCFG includes two types of nodes, the first comprises only one logic conditional statement, and the other contains multiple sequential statements. The edges in HCFG represent the control flow paths, similar to the edges in CFG. The HCFG

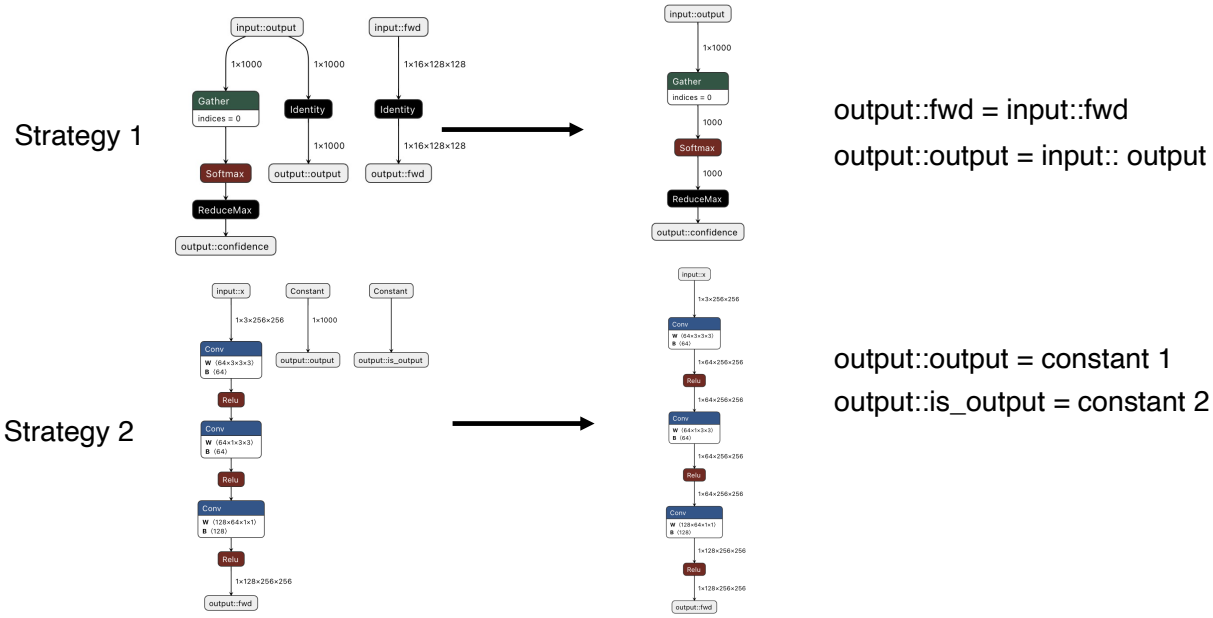

Figure 6.3. The proposed graph optimization strategy.

construction algorithm is shown in Algorithm 3. On line 1, we first construct the CFG of the rewritten DyNN program. Then we traverse the basic blocks in the CFG (line 3), and if the last statement of the block is an if statement (line 5), we split the block into two blocks and update the HCFG (lines 6-8). Each node in the HCFG is either an if statement or a sequence of statements that contain only the tensor computation statements. We treat each node that contains only the tensor computation statements as a sub-DNN for compilation.

6.5.4 Sub-DNN Optimization

For each sub-DNN in the constructed HCFG, we perform further optimization on its computational graph to avoid unnecessary data transfer overheads between the host program and the accelerator (*e.g.,* GPU). As discussed in Section 6.4, we seek to identify the computation-free operations in each sub-DNNs' computational graph and put these operations on the host program.

The first strategy is to eliminate the identity tensor copy operation. For this strategy, we start from each output node and enumerate all paths from the input node to this output node. If a path contains only the `identity` operator, we will remove this path and add a corresponding assignment statement in the host program. The second strategy is similar to constant propagation in compiler optimization. Firstly, we identify the output node for which all of its sink nodes (nodes without any incoming edges) are constant; we then remove these paths in the computational graph and add the corresponding assignment statement in the host program. Figure 6.3 shows an example of our proposed graph optimization strategy. For each strategy, the first column shows the original computational graph, and the second column is the optimized computational graph. The third column shows the corresponding statements we add to the host program.

6.5.5 Sub-DNN Compilation

Recall that using the DL compiler to compile a DNN model requires feeding an example input to the compiler. To create such an example input, the input tensor shape must be manually defined, which is a time-consuming process for DyNNs with hundreds of sub-DNNs. To address this challenge, we propose an automatic algorithm to compile each sub-DNNs.

Our automatic sub-DNN compilation algorithm is shown in Algorithm 4, which takes an HCFG, a start node N_0, and a DyNN input x_0 as inputs and outputs a mapping $\mathcal{M}$ from node id to the compiled sub-DNNs. In general, the algorithm maintains a search list L and a dictionary S that stores the output shape for each node. While the search list L is not empty, we select a node N from L iteratively, depending on whether all predecessors of N have been computed (line 4). We then set N as visited and collect all non-visited successors of N to update our search list (lines 5-7). For each node N, there are three conditions: (1) N is the start node N_0; (2) N is the logic conditional node; (3) N is a regular node other than N_0. For the first condition, we compute N's output shape and compile node N using

Algorithm 4 Sub-DNN Compilation Algorithm.

Input: Heterogeneous Control Flow Graph (HCFG).
Input: Start Node (N_0).
Input: An example input of the DyNN model (x_0).
Output: A mapping from node to compiled sub-DNN ($\mathcal{M}$).

L = [N_0] {Maintain a search list L}
S = Dict() {Maintain S to store tensor shape}
while L is nor empty **do**
  N = Select(L) {select N based on N's precursor}
  N_{visit} = True {set N as visited}
  $M = N.successor$ {collect N's successor}
  $L.update(M)$ {update the search list}
  if N is N_0 **then**
    out = =`Compute(`N, x_0`)` {compute node N}
    $S[N.id] = out$ {record the output shape for N}
    exe = =`Compile(`N, x`)` {compile node N}
    $\mathcal{M}[N.id] = exe$ {store the executable N}
  else
    if N is Logic Node **then**
      $S[N.id] = S[N.precursor]$ {set output shape of logic statement the same as its precursor}
      Continue
    else
      $x = S[N.precursor]$ {get input shape from its precursor}
      out = =`Compute(`N, x`)` {compute node N}
      $S[N.id] = out$ {record the output shape}
      exe = =`Compile(`N, x`)` {compile node N}
      $\mathcal{M}[N.id] = exe$ {store the executable}
    end if
  end if
end while

input x_0 (lines 8-12). For the second condition, we set N's output shape the same as its predecessor's output shape because the conditional statement will not modify the variable in the program (lines 14-16). As for the third condition, we first obtain an input x from node N's predecessors, then use x to compile N, and finally compute N's output shape (lines 18-23). Our algorithm iteratively fetches N from the search list and compiles N until L is empty (lines 3-4). Finally, our algorithm outputs the mapping $\mathcal{M}$, whose key is the node id and the value is the compiled sub-DNN.

6.5.6 Host API Generation

Finally, we use a template-based approach to generate the code that invokes each compiled sub-DNN. We modify the rewritten DyNN program's AST by replacing the original tensor computation statements with the statements to invoke our compiled sub-DNNs. We generate our host API function from the modified AST. Our AST modification step only replaces the original tensor computation statements in the original DyNN program with a compiled library function call. Considering the fact that existing DL compilers have proven correct and effective in compiling standard neural networks without conditional statements, the generated API call will be semantically equivalent to the original DyNN model.

6.6 Evaluation

6.6.1 Experimental Setup

In this section, we evaluate `DyCL` with empirical experiments and answer the following research questions. Our code and data are available on our website (Chen et al., 2023).[2]

- ***RQ1 (Correctness)*:** Can `DyCL` correctly compile DyNNs and produce semantic equivalent API for host programs?

[2]https://github.com/DyCL

- ***RQ2 (Acceleration)*:** Can `DyCL` optimize DyNNs over multiple platforms in terms of execution time?

- ***RQ3 (Ablation Study)*:** How much does the proposed graph optimization module in `DyCL` enhance the execution time?

- ***RQ4 (Overhead)*:** What is the overhead of `DyCL` in compiling DyNNs?

DyNN models. In addition to the four DyNN models in Table 6.1, we apply `DyCL` on other five DyNN models. The IDs and names of our subject DyNNs are shown in columns 1 and 2 in Table 6.3 and we will use IDs to represent these DyNNs in all future tables. The selected DyNN models cover different model architectures (*e.g.,* MobileNet and ResNet), different applications (*e.g.,* image classification and text generation), and different scales (*e.g.,* model sizes range from 2.3MB to 430MB). More detailed information about the evaluated DyNN models can be found on our website.

DL Compilers. We consider the DL compilers used in our empirical study as the target compilers: `TVM` and `OnnxRuntime`.

Hardware Platforms. We choose two different NVIDIA platforms imposing different architectural features as our hardware platforms to show `DyCL` can benefit the process of deploying DyNN models on various hardware platforms. The first hardware platform is NVIDIA Jetson TX2, which has 6 ARM-based cores and a 256-core Pascal-based GPU. The second platform is NVIDIA Jetson AGX Xavier, a powerful platform for robotics and autonomous driving with an 8-core NVIDIA Carmel CPU and a 512-core Volta-based GPU.

Comparison Baselines. Recall that DyNNs compiled by existing DL compilers can only correctly infer the inputs that have the same execution paths as the example input used in the compilation process. Thus, comparing `DyCL` with existing DL compilers is meaningless in terms of correctness. Therefore, for the correctness (RQ1) and the acceleration (RQ2)

evaluation, we compare `DyCL` with original DyNNs without compilation (*i.e.,* using the original DL framework). We refer to the baseline as `vendor`.

Experimental Process and Metrics. For RQ1, we use `DyCL` to compile each `vendor` version DyNN $\mathcal{V}(\cdot)$ and get the compiled version $\mathcal{C}(\cdot)$. Similar to our study process in Section 6.3, we feed 100 randomly generated inputs to these two versions and collect the outputs before and after the post-prepossess, and use the metrics defined in Equation (6.1) to evaluate the correctness of compilation. If `DyCL` can correctly compile the DyNN model, then the results for the metric *final inconsistent rate* should be zero, and the results for the metric *numeric maximum error* should be significantly different from the numeric errors in Figure 6.1.

For RQ2, we first deploy the `vendor` version DyNN $\mathcal{V}(\cdot)$ and the compiled version $\mathcal{C}(\cdot)$ on two different hardware platforms. After that, we feed the same inputs to these two versions for inference and record the inference overheads. For each input, we infer five times and report the average inference overheads.

For RQ3, we remove the graph module in `DyCL` and apply `DyCL` to compile each DyNN and get the compiled DyNN $\mathcal{C}_{no}(\cdot)$. After that, we deploy $\mathcal{C}_{no}(\cdot)$ and $\mathcal{C}(\cdot)$ on five different hardware platforms and evaluate their inference overheads, similar to the process in RQ2. Intuitively, if our proposed co-optimization method can benefit the compilation process, then the compiled DyNN $\mathcal{C}(\cdot)$ will have lower runtime overhead.

For RQ4, we report the time overheads of each module in `DyCL` when compiling each DyNN.

6.6.2 RQ1: Correctness

The compilation correctness results are shown in Table 6.3. Column 2 shows the names of the DyNN models under evaluation; columns 3 to 6 show the final inconsistent rate (η in Equation (6.1)) between the outputs $\mathcal{V}_{post}(\cdot)$ and $\mathcal{C}_{post}(\cdot)$, and the last four columns show the maximum numeric error (δ in Equation (6.1)) between the outputs $\mathcal{V}_{before}(\cdot)$ and $\mathcal{C}_{before}(\cdot)$.

Table 6.3. Correctness of DyCL.

ID	Base DNN	Final Inconsistent Rate				Max Numeric Error			
		Nvidia TX2		Nvidia AGX		Nvidia TX2		Nvidia AGX	
		TVM	OnnxR	TVM	OnnxR	TVM	OnnxR	TVM	OnnxR
1	**MobileNet**	0	0	0	0	-10.00	-10.00	-10.00	-10.00
2	**VGG19**	0	0	0	0	-9.17	-9.35	-9.17	-9.35
3	**ResNet50**	0	0	0	0	-4.84	-4.94	-4.89	-5.53
4	**WideResNet**	0	0	0	0	-4.72	-5.60	-4.75	-5.48
5	**ResNet38 + RNN**	0	0	0	0	-5.62	-5.81	-5.62	-5.81
6	**ResNet38 + Dense**	0	0	0	0	-6.45	-6.45	-6.45	-6.45
7	**ResxNext + LSTM**	0	0	0	0	-10.00	-10.00	-10.00	-10.00
8	**GoogLeNet+ LSTM**	0	0	0	0	-10.00	-10.00	-10.00	-10.00
9	**FlatResNet32**	0	0	0	0	-10.00	-10.00	-10.00	-10.00

We made the following observations. First, for all the evaluation subjects, DyCL can succesfully compile the original DyNN models for deployment, and the final outputs of the deployed host programs (*i.e.,* the results after post-process) are the same as the final outputs of the original DyNNs. This shows that the DyCL compiled DyNN is semantic-equivalent to the original DyNN, indicating that DyCL can correctly compile the DyNN model. Second, the differences of *maximum numeric error* between the outputs of the compiled DyNN and original DyNN are small, ranging from 0 to $10^{-4.72}$. The differences of numeric errors are within an allowed error range (*i.e.,* PyTorch sets a maximum default error as 10^{-5} for the compilation process). Recall that in Figure 6.1, directly applying existing DL compilers to compile the DyNN models results in the maximum error range $[10^0, 10^4]$. The small differences of maximum numeric error also confirm the correctness of DyCL; otherwise, if there is a difference between the execution path between the original DyNN and the DyCL compiled DyNN, the maximum numeric error will be much larger.

Answers to **RQ1**: With DyCL, existing "static" DL compilers can successful compile the DyNN models. The numeric error between the original DyNN and the compiled DyNN is minimal and does not affects the final prediction.

6.6.3 RQ2: Accelaration

The overheads of the original DyNN model and the compiled DyNN model are shown in Figure 6.4, where the x-axis represents the DyNN model ID, and the y-axis shows the average inference overheads in seconds. The different color bars represent different running mechanisms. We observe that for all our experimental settings, the compilation process can accelerate the DyNN models' inference, and the acceleration rate range from 1.12× to 20.21×. Such significant acceleration shows the advantage of compiling the DyNN model for deployment, especially for deploying the DyNN model on resource-constrained platforms. Another interesting observation is that no DL compiler can consistently outperform the others on all experimental subjects, indicating that the optimization strategies in existing DL compilers are complimentary. This result demonstrates the benefit of `DyCL`'s capability of adapting different DL compilers to compile DyNNs.

> Answers to **RQ2**: `DyCL` compiled DyNN models are consistently faster than the original DyNN models in terms of inference time, across different platforms and DL compilers.

6.6.4 RQ3: Ablation Study

The results of our ablation study are shown in Table 6.4. We show the results on Nvidia AGX platform due to the limit of space, more results could be found on our website. Column 1 shows the same subject DyNN IDs as in Table 6.3. The data in columns $\mathcal{C}$ and $\mathcal{C}_{no}$ represent the inference time overheads of the programs that are compiled by `DyCL` with and without the graph optimization module, respectively. We observe that for most settings, the graph optimization module can accelerate the compiled DyNN with an average acceleration rate ranging from 4.007% to 9.438%. Considering that existing DL compilers have almost done extreme optimization on the tensor computation on modern hardware

Table 6.4. The inference overheads of compiled DyNN that are compiled with and without the graph optimization module.

ID.	TVM			ONNX		
	$\mathcal{C}_{no}$	$\mathcal{C}$	**Accelerate**	$\mathcal{C}_{no}$	$\mathcal{C}$	**Accelerate**
1	0.530	0.525	0.939	0.417	0.412	1.222
2	0.589	0.455	29.449	0.621	0.562	10.576
3	0.056	0.055	1.325	0.088	0.090	-2.524
4	0.525	0.429	22.297	0.585	0.564	3.685
5	0.058	0.051	13.030	0.026	0.024	10.241
6	0.054	0.053	1.873	0.020	0.019	6.796
7	0.236	0.205	15.482	0.162	0.157	3.593
8	0.407	0.406	0.329	0.311	0.304	2.252
9	0.074	0.074	0.219	0.037	0.037	0.223
Avg	0.281	0.250	9.438	0.252	0.241	4.007

platforms, the acceleration results in Table 6.4 are significant. This result confirms that our graph optimization strategy can benefit `DyCL` in compiling dynamic neural networks.

> Answers to **RQ3**: The graph optimization module of `DyCL` further accelerates the compiled DyNNs, on average 4% to 9% faster than those compiled without the module.

6.6.5 RQ4: Overheads

Table 6.5 shows the overheads of `DyCL`. The second and third columns show the overheads of the rewriting and the graph optimization module (overheads of other modules are ignorable), the fourth and fifth columns show the overheads of applying the existing DL compiler to compile all sub-DNNs, and the last two columns show the overhead percentage of `DyCL` in terms of the total overheads of the compilation.

From the results, we observe that the main overheads of `DyCL` come from the graph optimization module. This is because each sub-DNN may be a computational graph with hundreds of nodes, thus emulating all paths from the input node to the output node is time-consuming. Moreover, the overheads of `DyCL` are ignorable when compared with the

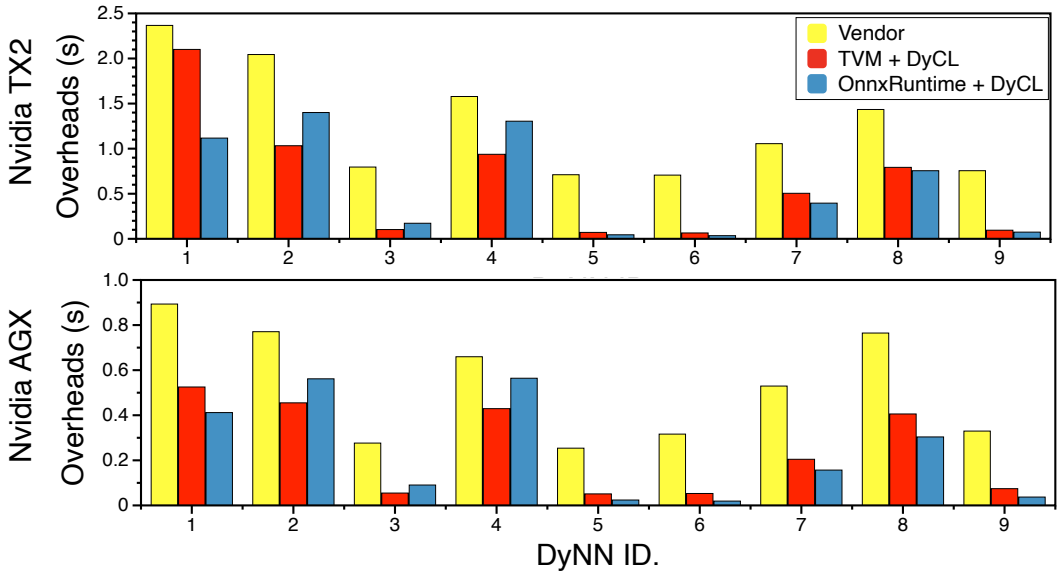

Figure 6.4. The inference overhead of the original DyNNs and the DyCL compiled DyNNs.

Table 6.5. The overheads of DyCL.

ID.	DyCL (s)		Original Overheads (s)		Extra Percentage (%)	
	Rewriting	Graph Opt	OnnxR	TVM	OnnxR	TVM
1	0.06	9.19	37.18	191.53	24.87	4.83
2	0.04	14.24	56.17	155.84	25.42	9.16
3	0.05	0.28	68.23	374.62	0.48	0.09
4	0.08	1.95	94.35	283.78	2.15	0.71
5	0.14	0.85	59.67	607.27	1.65	0.16
6	0.16	0.76	202.80	594.12	0.45	0.16
7	0.09	8.96	80.85	323.92	11.20	2.80
8	0.06	49.09	219.30	445.60	22.41	11.03
9	0.14	0.85	210.20	746.98	0.47	0.13
Avg	0.091	9.573	114.307	413.740	9.900	3.230

overheads of applying the DL compiler to compile the sub-DNNs. `DyCL` occupies only 3.23% to 9.90% percentage of the total overheads of compiling all sub-DNNs.

> Answers to **RQ4**: `DyCL` is a lightweight approach and does not significantly increase compilation overheads.

6.7 Threats to Validity

Our selection of the DyNN systems, namely, ShallowDeep, SkipNet, AttentionNet, and En-Decoder, *etc.*, might be a threat to the *external validity* of our experimental conclusions. We have tried to alleviate this threat through the following efforts: (1) The DyNNs are very popular, which can be seen through the number of citations of the works (Table 6.1); (2) the underlying DNN models are state-of-the-art models, which are used significantly; (3) these systems differ from each other in terms of model architecture and functionality. Therefore, our experimental conclusions should generally hold because of the diverse model subjects. Moreover, it is important to address the issue of noisy latency measurements, as it can potentially impact the validity of our experimental conclusions. To tackle this challenge, we conducted multiple latency measurements and recorded both the average and variance values. Our results demonstrate that the variances are significantly smaller than the average value, indicating that the impact of system noise on our experiments is minimal and does not compromise the validity of our findings.

6.8 Conclusion

In this work, first, we study the limitation of existing DL compilers to compile Dynamic Neural Networks. The significant inconsistent rate in our study results validates that existing DL compilers cannot handle dynamic neural networks. Then, we propose a program rewriting approach to split the tensor computation and the conditional statements, apply the DL

compiler to compile the tensor computation parts and leave the conditional statements to the host program. Based on this idea, we propose `DyCL`, the first tool that can reuse the existing "static" DL compiler in the context of dynamic neural networks. Our evaluation of nine publicly available DyNN models shows that `DyCL` can correctly compile DyNN models. Moreover, evaluation results show that `DyCL` can achieve up to 20× inference time acceleration.

CHAPTER 7

EVALUATING THE MODEL LEAKAGE RISK IN MACHINE LEARNING COMPILER [1]

This chapter analyzes the privacy of ML software efficiency optimizers (i.e., ML compilers). Specifically, this chapter examines whether these optimized high-performance programs can be easily reversed to leak model information.

Section 7.2 provides an overview of this work. Section 7.3 introduces our proposed approach, and Section 7.4 presents our evaluation. Finally, Section 7.5 concludes this chapter.

7.1 Author Contributions

Simin Chen proposed the approach, conducted the experiments, and authored the majority of the paper. Hamed Khanpour, Cong Liu and Wei Yang contributed to the writing and subsequent revisions of the paper.

7.2 Overview

Over the last few years, deploying Deep Neural Networks (DNNs) on edge devices is becoming a popular trend for giant AI providers. The AI providers compile the private high-quality DNN models into stand-alone programs (Li et al., 2020a; Chen et al., 2018; Rotem et al., 2018; Cyphers et al., 2018) and would like to sell them to other companies, organizations, and governments with a license fee. This edge-deployment situation further exacerbates the risk of model leakage. The DNN model leakage will result in the following three consequences. First, the potential loss of intellectual property. DNNs parameters are typically trained from TB

[1] ©2022 International Joint Conferences on Artificial Intelligence. Reprinted, with permission, from Simin Chen, Hamed Khanpour, Cong Liu, Wei Yang. "Learn to Reverse DNNs from AI Programs Automatically". In Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence. DOI:10.24963/IJCAI.2022/94.

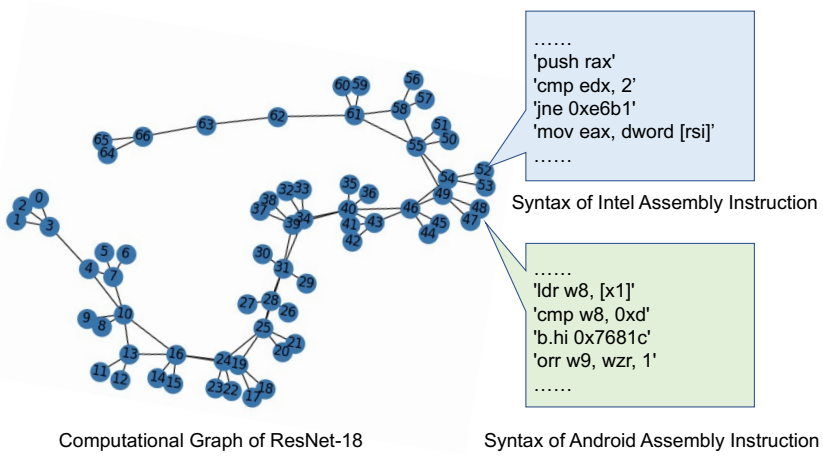

Figure 7.1. Assembly instructions of ResNet-18 on different platforms.

datasets with high training costs. For example, training a DNN on the google cloud TPU will cost more than $400K (Devlin et al., 2018; Raffel et al., 2019; Zhu et al., 2021). Thus, DNNs architecture and parameters are precious intellectual property of the developers. Second, the potential infringement of privacy. DNNs parameters have been proven to remember sensitive information (*e.g.,* SSN) in the DNNs training data (Carlini et al., 2020). Therefore, the leakage of DNN parameters will result in the privacy leakage of training data. Last but not least, the potential risk of being attacked by adversaries. DNNs are vulnerable to adversarial attacks (Carlini and Wagner, 2017). If the attackers obtain the parameters, they could easily compute DNNs gradients to launch white-box attacks. For Example, reversing Apple's image hash model, NeuralHash [2], will enable attackers to break the content protection provided by macOS. Therefore, conducting penetration testing to protect DNNs' privacy is essential.

Quantifying model leakage risk of on-device DNNs automatically is a challenging task because of the following two reasons: (1) Reversing DNN architectures requires understanding the semantics of the machine code of each DNN layer. Machine code is incomprehensible to humans; unfortunately, modern DNN usually contains hundreds of layers, and each layer

[2]Apples-neuralhash-algorithm-has-been-reverse-engineered

corresponds to a function that needs to infer the semantic. (2) The cross-platform deployment of DNN models increase the difficulty of understanding the semantics of the code. DNNs may be deployed on different hardware platforms. The machine code on different hardware platforms have different syntax. Therefore, understanding the semantics of machine code on different hardware platforms requires different domain-specific knowledge. Figure 7.1 shows the challenge in reversing DNNs, the left figure shows the computational graph of Resnet-18, reversing ResNet-18 requires inferring the semantic of each node in the graph, and the right figure shows different assembly instructions syntax on different hardware platforms.

In this paper, we propose `NNReverse`, the first static model stealing attack to reverse DNNs from AI programs automatically. Different from existing model extraction attacks (Jagielski et al., 2020; Oh et al., 2019; Tramèr et al., 2016), which approximate the victim DNNs functionality rather than reversing DNN architectures (Krishna et al., 2019). Our approach does not train a substitute model to approximate input-output pairs from the victim DNNs. Instead, our approach learns and infer the semantic of machine code for different hardware platforms automatically.

Our intuition is that although DNN architectures are complicated, the basic types (*i.e.,* *conv*, *dense*) of DNN layers are enumerable. Thus, it is possible to reverse DNN architectures by inferring each DNN layer type. Furthermore, because different DNN layers are implemented as different binary functions, we can infer DNN layer type by inferring semantic corresponding binary functions. Specifically, our attack includes two phases: offline phase and online phase. In the offline phase, we build a database to collect the binary of each optimized kernel layer, then we train a semantic representation model on the decompiled functions. The semantic representation model can be trained with machine code from different hardware platforms, thus, `NNReverse` could reverse DNNs for different hardware platforms. Although some semantic representation approaches for NLP fields have been proposed, they are not suitable for assembly functions because they ignore the topology semantic in assembly

functions. We propose a new embedding algorithm to overcome this limitation and represent assembly functions more precisely, which combines syntax semantic representation and topology semantic. In the online phase, we decompile the victim DNN executable and collect the binaries. By the pre-collected knowledge in the database, we reconstruct the whole DNN model. We summarize our contribution as follows:

- **Characterizing**: We characterize the model leakage risk of on-device DNNs. Specifically, we show that on-device DNNs could be reversed easily.
- **Approach**: We propose a new fine-grained embedding approach for representing assembly functions, which combines syntax representation and topology structure representation.
- **Evaluation**: We implement `NNReverse` and evaluate `NNReverse` on two datasets. The evaluation results show `NNReverse` can achieve higher accuracy in searching similar semantic functions than baseline methods, and we apply `NNReverse` to reverse real-world DNNs as a case study. Results show that `NNReverse` can reverse the DNNs without accuracy loss.

7.3 Methodology

7.3.1 Problem Formulation

In this paper, we consider an AI model privatization deployment scenario, where the DNN model developers compile their private DNN models into stand-alone AI programs and sell the programs to third-party customers with subscription or perpetual licensing. The considered scenario is realistic because many AI models have been deployed on mobile devices (*e.g.,*, Apple NeuralHash, Mobile Translation App, AIoT)and this assumption is widely used in existing work. The end-users can physically access the AI programs, especially the machine

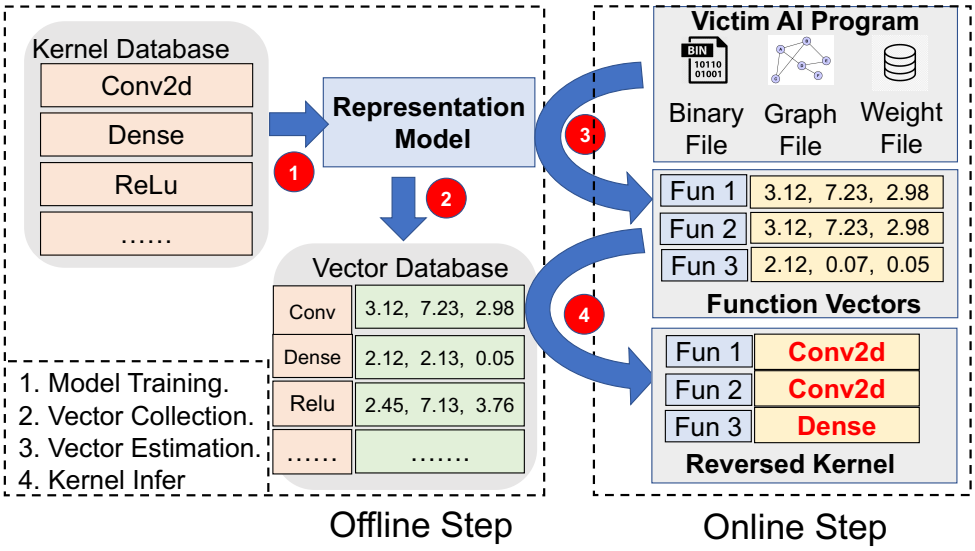

Figure 7.2. Design overview of our method.

code (executable code) of the AI programs. The goal of malicious attackers is to reverse the DNNs from the AI program. Specifically, they can reverse the architecture of the DNNs and reconstruct the DNNs with high-level frameworks (*i.e.,* Pytorch) to compute the DNNs gradients to launch evasion attacks. Formally, our goal includes: *(i)* inferring the kernel operator type of each binary function in the victim AI-programs; *(ii)* reconstructing whole DNNs with high-level DL frameworks. Reversing on-device DNNs have a severe real-world impact because DNNs parameters are precious intellectual property, and reverse engineering can bridge the gap between white-box and black-box attacks for DNNs (*e.g.,*, bypass Apple NeuralHash). Furthermore, understanding the semantic intricacies of deployed binary code makes automatically reversing on-device DNNs difficult.

7.3.2 Design Overview

As shown in Figure 6.2, our approach includes offline and online steps. In the offline step, we collect a set of DNNs with known architectures and use TVM (Chen et al., 2018), a widely-used DL compiler, to compile them into AI programs including a binary file for each

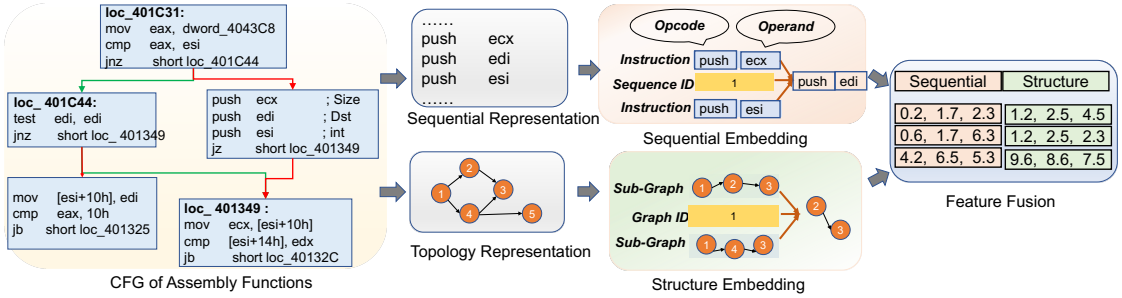

Figure 7.3. Design overview of our semantic representation model.

program. We keep the debugging information for each binary file to keep a record of the kernel operator type of each binary function. We then train a semantic-representation model with these binary files. In the online step, given a victim AI program, we first parse the AI program and decompile the parsed stripped binary file into a list of assembly functions. Next, we feed each decompiled assembly function into the trained semantic representation model to estimate the vectors. Finally, we compare the estimated vectors with the vectors in our offline database to infer the kernel operator type. We could reverse the DNNs arch after inferring the kernel operator type and rebuild it with PyTorch by combing the weight files.

7.3.3 Semantic Representation Model

In this section, we propose an embedding model to represent assembly functions as continuously distributed vectors. As shown in Figure 7.3, an assembly function can be represented as a control flow graph (CFG), where nodes are blocks consisting of sequential assembly instructions and edges are control flow paths. Our embedding paradigm, in particular, fuses both text representation and structure representation from the assembly functions.

Text Embedding. We first model the assembly functions as sequential assembly instructions and seek to learn a function $\phi_t(\cdot)$, whose input is a sequential assembly instruction and output is a $\mathbb{N}$ dimensional vector. We embed the semantics of assembly instructions with more fine-grained modeling rather than treating each instruction as a token. Specifically, for

each assembly instruction, we differentiate the *opcode* and *operands* in the instruction. Then we concatenate the vector of *opcode* and *operands* as the representation of the instruction. For instruction with more than one *operands*, we use the average vector instead. It can be formulated as:

$$\mathcal{I}(ins) = \mathbb{E}(\Omega(ins)) \mid\mid \frac{1}{|\Theta(ins)|} \sum_{i=0}^{|\Theta(ins)|} \mathbb{E}(\Theta(ins)_i) \tag{7.1}$$

Where ins represents an assembly instruction, $\Omega(\cdot)$ and $\Theta(\cdot)$ are the functions to parse *opcode* and *operands* respectively, and $\mathbb{E}(\cdot)$ is the basic embedding layer that transfer tokens to corresponding numeric vectors. Take the assembly instruction $ins = push, rbx$ as example, then $\Omega(ins) = push$ and $\Theta(ins) = rbx$. Because the *operands* may be a memory address (*e.g.,* 0x73ff) or a immediate value (*e.g.,* 0x16, 0x1024). To eliminate the issue of out of vocabulary (OOV), we propose a blur strategy. Specifically, for memory address value and immediate value, we compare the value with 1000. If the value is less than 1000, we keep the original value, otherwise, we use token *large* to replace the original value.

$$\sum_{j=1}^{M} \sum_{i=1}^{|T(f_j)|} P(\mathcal{I}_i(ins) \mid C_i, \phi_t(f_j)) \tag{7.2}$$

After modeling assembly instruction, we seek to represent the semantic of the assembly function. Specifically, we treat each instruction $\mathcal{I}(ins)$ as a "word" and apply a similar idea with doc2vec. Formally, our training objective can be formulated as Equation 7.2. Where M is the number of the training assembly functions, f_j is the j^{th} assembly function, $T(\cdot)$ is the text representation, C_i is the neighbor instruction of $\mathcal{I}_i(ins)$ and ϕ_t is the text embedding under learned.

Structure Embedding. Given a set of CFG topology structures, we intend to learn a distributed representation for every CFGs' topology structure. Formally, we represent each CFG as a graph $G = (N, E)$, where N are assembly blocks and E are connections between blocks. Our goal is to learn a embedding function $\phi_s(\cdot)$, whose input is $g \in G$ and output

Table 7.1. The results of proposed text embedding, structure embedding and combined embedding.

Dataset		P			R			F1		
		Text	Structure	Combine	Text	Structure	Combine	Text	Structure	Combine
Vision	**ARM**	0.8595	0.5018	**0.8663**	**0.8875**	0.6665	0.8855	0.8733	0.5725	**0.8758**
	Aarch64	0.7562	0.3468	**0.7688**	**0.8406**	0.5388	0.8306	0.7962	0.4220	**0.7985**
	Android	**0.8419**	0.4578	0.8398	0.8741	0.6746	**0.8766**	0.8577	0.5455	**0.8578**
	Intel	0.8427	0.4954	**0.8444**	**0.8725**	0.6719	0.8713	0.8574	0.5703	**0.8576**
	AVG	0.8251	0.4505	**0.8298**	**0.8687**	0.6380	0.8660	0.8462	0.5276	**0.8474**
Textual	**ARM**	0.7232	0.2337	**0.7610**	0.6859	0.3990	**0.7104**	0.7041	0.2948	**0.7348**
	Aarch64	0.7301	0.5148	**0.7342**	0.6224	0.2566	**0.6519**	0.6720	0.3425	**0.6906**
	Android	**0.6836**	0.2007	0.6660	0.6872	0.2847	**0.6882**	**0.6854**	0.2354	0.6769
	Intel	**0.7557**	0.4085	0.7146	0.6889	0.3825	**0.7050**	**0.7208**	0.3951	0.7098
	AVG	**0.7232**	0.3394	0.7190	0.6711	0.3307	**0.6889**	0.6956	0.3170	**0.7030**

Algorithm 5 Structure Embedding Algorithm.

Input: G. {A set of CFGs topology structures}
Input: D. {Maximum degree of rooted subgraphs}
Initialization: Sample ϕ_s from $R^{G\times N}$
for $g_i \in G$ **do**
  for $n \in \text{NODE}(g_i)$ **do**
    for $d = 0$ to D **do**
      $sub_g = \text{ComputeWLSubgraph}(n, g_i, d)$
      $\phi_s = \phi_s - \frac{\partial -log\text{P}(sub_g \mid \phi_s(g_i))}{\partial \phi_s}$
    end for
  end for
end for

is a $\mathbb{N}$ dimensional vector. Our intuition is that two graphs are more topology similar if these two graphs have more topology similar sub-graphs. Different from the semantic embedding that we apply random-walk paths as representation, applying sub-graphs will keep higher-order substructure and non-linear substructure information. As shown in Algorithm 5. Our algorithm require a set of graphs G as the training corpus. In each training iteration, we compute the WeisfeilerLehman sub-graphs (Shervashidze et al., 2011) of each node in each graph g_i, then we try to maximize the possibility that the sub-graphs appear under the whole graph and update our structure embedding function $\phi_s(\cdot)$. We repeat this process iteratively until the maximum epochs is reached.

$$\phi_{binary}(f) = \frac{\phi_t(T(f))}{|\phi_t(T(f))|} \mid\mid \frac{\phi_s(S(f))}{|\phi_s(S(f))|} \tag{7.3}$$

Table 7.2. Accuracy results in searching semantic similar functions.

Data	Platform	word2vec (skip)			word2vec (cbow)			doc2vec (skip)			doc2vec (cbow)			Asm2Vec			Ours		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Vision	ARM	0.44	0.48	0.46	0.43	0.49	0.46	0.57	0.71	0.63	0.63	0.72	0.68	0.62	0.76	0.68	**0.87**	**0.89**	**0.88**
	Aarch64	0.29	0.41	0.34	0.45	0.37	0.40	0.43	0.60	0.50	0.43	0.61	0.50	0.59	0.66	0.62	**0.77**	**0.83**	**0.80**
	Android	0.42	0.50	0.46	0.41	0.51	0.46	0.57	0.71	0.63	0.61	0.71	0.66	0.64	0.65	0.64	**0.84**	**0.88**	**0.86**
	Intel	0.42	0.48	0.45	0.43	0.48	0.45	0.57	0.68	0.62	0.58	0.69	0.63	0.62	0.73	0.67	**0.84**	**0.87**	**0.86**
	mixed	0.39	0.47	0.42	0.43	0.46	0.44	0.53	0.68	0.60	0.56	0.68	0.62	0.62	0.70	0.65	**0.83**	**0.87**	**0.85**
Textual	ARM	0.41	0.41	0.41	0.38	0.36	0.37	0.51	0.44	0.48	0.47	0.43	0.45	0.58	0.59	0.59	**0.76**	**0.71**	**0.73**
	Aarch64	0.33	0.33	0.33	0.29	0.21	0.24	0.48	0.42	0.45	0.42	0.37	0.39	0.41	0.51	0.45	**0.73**	**0.65**	**0.69**
	Android	0.51	0.37	0.43	0.46	0.47	0.46	0.42	0.45	0.43	0.48	0.43	0.45	0.57	0.61	0.59	**0.67**	**0.69**	**0.68**
	Intel	0.35	0.35	0.35	0.36	0.32	0.34	0.47	0.45	0.46	0.46	0.43	0.45	0.58	0.57	0.57	**0.71**	**0.71**	**0.71**
	mixed	0.40	0.36	0.38	0.37	0.34	0.35	0.47	0.44	0.45	0.46	0.42	0.44	0.54	0.57	0.55	**0.72**	**0.69**	**0.70**

Feature Fusion. Finally, we combine the vectors from text embedding and structure embedding to obtain the final function embedding. Equation 7.3 shows our fusion process, where f is the assembly function, $T(f)$ and $S(f)$ are the text representation and the structure representation of f respectively, ϕ_t and ϕ_s are the text embedding function and structure embedding function, and $||$ is the vector concatenation.

$$kernel(\mathcal{B}) = name_i \quad i = \text{argmax}_j \frac{v \cdot u_j}{||v|| \times ||u_j||} \tag{7.4}$$

Binary Matching. After the model is well trained, we obtain the pre-trained database $U = \{(u_j, name_j)\}$, where u_j and $name_j$ are the vector and the kernel type of the j^{th} function in the database. During the inference step, given a binary function $\mathcal{B}$, we first feed $\mathcal{B}$ into our model to get a vector v. we then apply Equation (7.4) to assign $\mathcal{B}$ a known kernel type.

7.3.4 DNN Reconstruction

After training a semantic representation model ϕ_{binary}. We propose a automatic algorithm to rebuild the victim DNNs. As shown in Algorithm 6, the inputs of our algorithm are the victim AI program and a trained semantic representation model. We first parse the program to collect the graph files (CG) and collect all kernel nodes N in CG based on whether the node has an edge point. While list N is not empty, we select a node c from N based on whether c's all precursors have been inferred (line 3). For the selected node c, we find the

corresponding binary function f (line 4) and apply our semantic representation model to infer the kernel type k of this function (line 6). After that, we compute the output of node c and remove it from N. We repeat this process iteratively until N is empty. Finally, we return the computed tensor of the output node in CG. By doing so, we reconstruct the DNNs from the AI programs.

7.4 Evaluation

We answer the following questions through empirical evaluation.

- ***RQ1:*** *How accurate of proposed representation model?*
- ***RQ2:*** *Can our algorithm 2 rebuild DNNs correctly?*
- ***RQ3:*** *Is our approach sensitive to hyper-parameters.*
- ***RQ4:*** *What's the contribution of each component?*

Algorithm 6 DNNs Reconstruction Algorithm.

Input: VicProg. {Victim AI program }
Input: ϕ_{binary} {Semantic representation model}
1: CG = ParseFile(VicProg)
2: N = CollectKernelNodes(CG)
3: **while** N is not empty **do**
4: c = SelectNode(N) {select a node whose all precursors are computed}
5: f = FindBinaryFunction(c)
6: $k = \phi_{binary}(f)$ {Infer kernel type based on Eq. 7.4}
7: $c.type = k$ {Assign kernel to node c}
8: Compute node c
9: Remove c from N
10: **end while**
Output: Return computed tensor of the output node in CG.

Table 7.3. Data statistics of assembly functions.

Type	Platform	# Train	# Test	Instructions	Blocks
Vision	**Android**	22948	5737	194.75 ± 223.64	43.95 ± 47.19
	Intel	22892	5724	141.52 ± 135.41	42.49 ± 45.61
	ARM	28481	7121	208.19 ± 291.24	40.81 ± 44.70
	Aarch64	5761	1441	197.68 ± 216.46	43.87 ± 47.24
	Mix	80082	20023	195.61 ± 238.88	42.92 ± 46.41
Textual	**Android**	2647	662	131.48 ± 144.01	32.22 ± 38.23
	Intel	808	207	97.01 ± 104.40	32.96 ± 38.058
	ARM	2728	683	126.30 ± 142.88	21.20 ± 38.06
	Aarch64	2618	655	135.67 ± 150.20	32.30 ± 38.51
	Mix	8801	2207	127.94 ± 142.73	31.32 ± 37.66

7.4.1 Experimental Setup

Datasets. We first download the open source models from *TorchVision* and *HuggingFace* as our dataset. Our model dataset includes vision models (*e.g.,* VGG-11, *etc.*) and textual models (*e.g.,* BertSentenceClassification, *etc.*). We then use TVM, a popular deep learning compiler (Chen et al., 2018) to compile deep neural networks into AI programs. We consider four different deployment platforms (*e.g.,*, Android, Intel, ARM, and Aarch64), and we set the compilation optimization level from 0 to 4. For the executable files of the compiled programs, we use radare2 to disassemble the binary functions into assembly instructions. We split 80% of the data as our training dataset and the rest 20% for testing. The statistic of our dataset can be found in Table 7.3.

Comparison Baselines. We compare our proposed method with six unsupervised semantic representation methods: `word2vec`, `doc2vec`, and `asm2vec`. For `word2vec`, we use the average vectors of all instructions in the assembly function to represent the semantic. For `word2vec` and `doc2vec`, we consider both `Skip-Gram` and `CBOW` models. For a fair comparison, for each method, we set the dimensions of the embedding as 100.

Table 7.4. Successfully reversed DNNs without accuracy loss.

Model	# of Layers	# of Parameters	Acc Loss
LeNet-5	7	0.44M	0.00
ResNet-18	68	11.69M	0.00
ResNet-50	175	25.56M	0.00
MobileNet-V2	158	3.51M	0.00
VGG-11	29	132.86M	0.00
VGG-19	45	143,67M	0.00
WideResNet-50	175	68.88M	0.00

7.4.2 (RQ1) Results for Binary Function Searching

We follow existing work (Alon et al., 2018, 2019) to apply *Semantic Precision*, *Semantic Recall* and *Semantic F-1* score as our metrics. Table 7.2 summarizes the main results. From the results, we could observe that: the proposed method significantly improves the binary function mapping accuracy across two datasets on all hardware platforms, The observation demonstrates our embedding technique's success in enhancing the prediction capacity in the binary mapping problem.

7.4.3 (RQ2) Reconstruct DNNs

In this section, we evaluate the effectiveness of our reconstruction algorithm (*i.e.,* Algorithm 6). Because a learning-based technique cannot guarantee soundness results, we choose DNNs with high kernel operator similarity scores as examples. The selected model architectures are listed in Table 7.4. We use the accuracy loss as our evaluation metric. Formally, we feed random inputs into the original and reconstructed models. Then, the accuracy loss is defined as the accuracy difference between the accuracy of the original model and the reconstructed model. Table 7.4 lists some examples of our successful reversed DNN models.

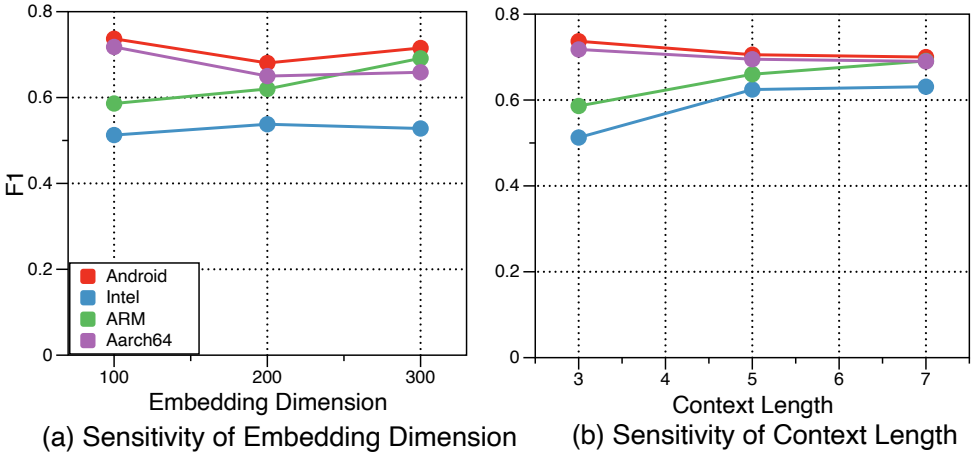

Figure 7.4. Sensitivity of representation model.

7.4.4 (RQ3) Parameter Sensitivity

We perform the sensitivity analysis of our proposed techniques on the Textual dataset. **(a) embedding dimension:** We first conduct experiments to understand how different embedding dimensions affect our technique. Specifically, we configure the embedding dimension as 100, 200, 300 and measure F-1 under different settings. **(b) context length:** We then configure the context size as 3, 5, 7 and train different models. The results of different embedding configure are listed in Figure 7.4. From the results, we observe that `NNReverse` can keep a stable F1 scores on different embedding dimension and context length settings, which indicates that `NNReverse` is not sensitive to embedding dimension and context length.

7.4.5 (RQ4) Ablation Studies

Can fine-grained modeling eliminate Out of Vocabulary (OOV). We first conduct experiments to explore whether our fine-grained instruction embedding can eliminate the issue of OOV. We model each assembly instruction as a "word" and measure the OOV ratio, and then we measure the OOV ratio of our instruction embedding. The OOV results are summarized in Table 7.5. We observe that directly modeling each instruction as "word" will

Table 7.5. The OOV ratio results.

Platform	Vision		Textual	
	Our	Baseline	Ours	Baseline
Android	**0.21**	0.80	**0.09**	0.71
Intel	**0.11**	0.60	**0.62**	0.85
ARM	**0.12**	0.65	**0.51**	0.98
Aarch64	**0.14**	0.75	**0.08**	0.66
Avg	**0.14**	0.70	**0.32**	0.80

result in a high OOV ratio. This is because assembly instruction may contain the memory address, and the value of the address is had to predict.

The performance of each representation component. In this study, we compare our embedding method to solo text embedding and solo topological embedding to investigate the efficacy of embedding combinations. From the results in Table 7.1, we observe that: (1) for most cases, 20 out of 30, combing text embedding and structure embedding can increase binary mapping accuracy. (2) structure embedding performs the poorest in all scenarios, implying that simply applying structure information is insufficient to infer function semantics. CFG structures, on the other hand, are complementary to text, and combining them yields better results.

7.5 Conclusion

In this research, we present a technique for automatically reversing DNNs from deployed AI applications. In particular, we propose a novel embedding technique for capturing the semantics of assembly functions. Our embedding techniques combine the text semantic and topology semantic of assembly functions, resulting in a more accurate semantic representation. The experiments demonstrate that `NNReverse` is possible to correctly reverse DNNs.

CHAPTER 8

RELATED WORK

8.1 Inference Efficiency of Deep Neural Network Models

This section covers the basics of inference efficiency for DNN models, including optimization strategies for inference time (Section 8.1.1) and performance testing for ML software (Section 8.1.2).

8.1.1 Efficiency Optimization

Recently, the efficiency of deep neural networks (DNNs) has become a significant concern due to their substantial inference-time costs. To enhance the inference-time efficiency of DNNs, numerous approaches have been proposed, which can be categorized into two major techniques: model-level optimization and program-level optimization.

Model-level Optimization. Model-level optimization techniques directly modify the architectures of deep neural network (DNN) models to reduce their inference costs (He and Xiao, 2024; Hoang and Liu, 2023; Wang et al., 2023). Based on the underlying optimization strategies, existing techniques can be grouped into three categories: (1) DNN pruning techniques, (2) knowledge distillation techniques, and (3) dynamic branching techniques.

DNN pruning techniques involve the offline analysis of neural networks to identify and remove neurons or layers deemed unimportant (Howard et al., 2017; Zhang et al., 2018; He and Xiao, 2024; Hoang and Liu, 2023; Sekikawa and Yashima, 2023; Wang et al., 2023; Vischer et al., 2021; Chen et al., 2023). By systematically pruning these less critical components, the resulting smaller DNNs can maintain competitive accuracy levels similar to the original, larger models but with significantly reduced computational costs (Diao et al., 2023; Peste et al., 2022; Wang et al., 2022; Zaken et al., 2021; Frantar and Alistarh, 2022). Pruning methods can be broadly categorized into several approaches: weight pruning (Lee et al., 2019;

Ma et al., 2021; Pensia et al., 2020; Chijiwa et al., 2021; Chao et al., 2020; Sehwag et al., 2020; Yang et al., 2017), neuron pruning (Alwani et al., 2022; Hou et al., 2022; Tiwari et al., 2021; Molchanov et al., 2016), and structured pruning (Li et al., 2021; He and Xiao, 2024; Serra et al., 2021; Chowdhury et al., 2022). Weight pruning focuses on removing individual weights based on their magnitude, typically setting small-weight connections to zero, thus reducing the overall number of parameters (Zhao et al., 2022; Tukan et al., 2022; Good et al., 2022). Neuron pruning eliminates entire neurons that minimally contribute to the network's output, effectively simplifying the network structure (Wang et al., 2021; Nonnenmacher et al., 2021). Structured pruning, on the other hand, removes entire filters, channels, or even layers, which can lead to more substantial reductions in computational complexity and memory usage (Li et al., 2021; Kim et al., 2022; Martinez et al., 2021). The benefits of DNN pruning are many. In addition to lowering computational costs and memory requirements, pruned networks often exhibit faster inference times and reduced power consumption, making them particularly suitable for deployment on resource-constrained devices such as mobile phones and embedded systems. Furthermore, pruning can sometimes lead to improved generalization by reducing overfitting, as the network becomes less complex and more focused on essential features (Diffenderfer et al., 2021; Hayou et al., 2020; Sehwag et al., 2020; Ye et al., 2019; Zhao and Wressnegger, 2023; Liu et al., 2022; Chen et al., 2022; Diffenderfer et al., 2021).

Another category of optimization techniques is knowledge distillation (Li et al., 2017; Yu et al., 2019; Park et al., 2019; Gou et al., 2021; Cho and Hariharan, 2019; Kim and Rush, 2016; Xu et al., 2024). The key idea behind knowledge distillation is to train a smaller, more efficient DNN model to mimic the behavior of a larger, more complex model by learning from its input/output pairs (Gu et al., 2024; Wang and Yoon, 2021; Tung and Mori, 2019; Mirzadeh et al., 2020). This process effectively distills the knowledge from the larger model into the smaller one, enabling the smaller model to achieve similar accuracy to the original while benefiting from reduced computational and memory requirements. Knowledge distillation

typically involves three main steps: training the teacher model, generating soft targets, and training the student model (Tang et al., 2020; Lopes et al., 2017; Li et al., 2017). The teacher model is the larger, pre-trained model that possesses high accuracy. It is used to generate soft targets, which are predicted probabilities or feature representations for a given input (Zhao et al., 2022; Sun et al., 2024; Chi et al., 2023; Zhou et al., 2021; Yang et al., 2021; Aguilar et al., 2020). These soft targets contain richer information than hard labels (e.g., one-hot encoded labels), because they convey the teacher model's confidence in its predictions, capturing the nuances of the learned representations (Wang, 2021; Zhou et al., 2021; Yuan et al., 2020, 2024; Sun et al., 2024).

Dynamic branching techniques (Wang et al., 2018; Graves, 2016; Figurnov et al., 2017; Davis and Arel, 2014; Gao et al., 2018; Najibi et al., 2019; Veit and Belongie, 2018; Wu et al., 2019; Hua et al., 2019; Xin et al., 2020; Zhu, 2021; Sponner et al., 2024; Zhang et al., 2023; Zeng et al., 2023; Schuster et al., 2022; Hou et al., 2020; Yin et al., 2022; Raposo et al., 2024) dynamically activate specific parts of a DNN model based on the semantic complexity of the input. This approach aims to balance prediction accuracy and computational efficiency by tailoring the computational effort to the requirements of each individual input. The core idea is that not all inputs present the same level of semantic difficulty, and therefore, do not require the same amount of computational resources to be processed effectively (Hou et al., 2020; Teerapittayanon et al., 2016; Zhang et al., 2023; Zeng et al., 2023; Schuster et al., 2022; Laskaridis et al., 2021; Wang et al., 2021). These techniques operate by evaluating the input and determining which sections of the network need to be activated to achieve an accurate prediction. For simpler inputs, only a subset of the model's layers or blocks may be utilized, thereby reducing the overall computational load. Conversely, for more complex inputs, a larger portion or the entirety of the model may be engaged to ensure high accuracy. Dynamic branching can be implemented in various ways. For example, in SkipNet (Wang et al., 2018), gating mechanisms are used to skip certain residual blocks based

on the input's characteristics. Adaptive computation time (ACT)(Graves, 2016) techniques allow the network to allocate more or fewer computational resources dynamically, depending on the complexity of the task at hand. Spatially adaptive computation time (SACT) (Figurnov et al., 2017) extends this concept by applying it within spatial regions of an image, enabling the network to focus computational effort on more informative regions. Recently, Google proposes `Mixure-of-Depth` (Raposo et al., 2024), which will allocate different computational resources for each expert in the mixture-of-expert models. The advantages of dynamic branching are significant, particularly in scenarios where computational resources are limited, or real-time processing is required. By allocating resources more efficiently, these techniques can maintain high levels of accuracy while substantially reducing inference time and energy consumption (Han et al., 2021b; Skarding et al., 2021; Nunes et al., 2022).

Program-level Optimization. Program-level optimization techniques focus on enhancing the implementation code of DNNs (i.e., the binary code of each operator in the DNN model) without altering their architectures. These techniques include: (1) low-bit quantization, (2) parallel computation, and (3) kernel optimization. Most existing program-level optimization techniques have been integrated into deep learning compilers. As DL compilers have already been introduced in Chapter 2, this dissertation will briefly describe these techniques here.

Low-bit quantization techniques use fewer bits (*i.e.,* fewer than 32) to represent numerical values. These methods significantly reduce memory consumption and accelerate inference on hardware platforms. Quantization techniques can be broadly categorized into two approaches: Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ) (Yao et al., 2023). Quantization-Aware Training (QAT) involves incorporating quantization into the training process itself. During QAT, the model is trained with awareness of the reduced bit precision, allowing it to adapt to quantization-induced noise and maintain higher accuracy (Dettmers et al., 2023; Liu et al., 2023). By simulating quantization effects during training, QAT helps the model learn weights and activations that are robust to the precision loss. This approach

is particularly effective in maintaining performance close to that of full-precision models, making it suitable for scenarios where accuracy is critical. Post-Training Quantization (PTQ), on the other hand, is applied after the model has been fully trained. PTQ converts the weights and activations of a pre-trained model to lower bit representations without further training (Dettmers et al., 2022; Frantar et al., 2023; Xiao et al., 2024; Yao et al., 2022; Lin et al., 2024; Yuan et al., 2023). While PTQ is faster and easier to implement compared to QAT, it may result in a slight degradation in model accuracy, especially for complex tasks. However, PTQ remains a popular choice for many applications due to its simplicity and efficiency.

Parallel computation aims to leverage he parallel processing capabilities of modern hardware architectures, these techniques distribute computation across multiple cores or devices, leading to substantial speedups during inference. One approach, tensor model parallelism (TP) (Shoeybi et al., 2020), splits model layers (e.g., attention, feed-forward networks) into multiple pieces based on internal dimensions (e.g., head, hidden layers) and deploys each piece on a separate device (e.g., GPU). This technique can significantly reduce inference latency through parallel computing and is widely used for multiple GPUs within the same machine, particularly in scenarios with high-speed NVLink connections (Shoeybi et al., 2020).

To reduce overheads from kernel launching and memory access, kernel fusion is widely adopted by previous DNN compilers. Since backward computation is not required for LLM inference, there are more opportunities for kernel fusion. Several contemporary Transformer inference engines (e.g., FasterTransformer, TenTrans (Wu et al., 2021), TurboTransformers (Fang et al., 2020), LightSeq (Wang et al., 2021), ByteTransformer (Zhai et al., 2023)) and compilers (e.g., Welder (Shi et al., 2023)) propose to fuse: (1) GEMMs with the same shape (e.g., the three linear transformations for query, key, and value) and (2) Add Bias with other non-GEMM kernels, such as residual connections, layer normalization, and activation

functions (e.g., ReLU). Among these optimizations, the fusion of multi-head attention kernels has been extensively explored and will be discussed in the following sections.

8.1.2 Performance Testing

In traditional software, if an operation is performed without being required, it is termed a redundant operation. Conducting performance testing is crucial for identifying and eliminating these inefficiencies (Ding et al., 2020; Daly, 2021; Chen et al., 2022; Lim et al., 2014; Xiong et al., 2013; Traini et al., 2023; Chen et al., 2020). Runtime performance is a critical aspect of software, and numerous methodologies have been proposed to test and enhance software performance. `WISE` (Burnim et al., 2009) introduces a method to generate test samples that trigger worst-case complexity scenarios. `SlowFuzz` (Petsios et al., 2017) proposes a fuzzing framework designed to detect algorithmic complexity vulnerabilities. Additionally, `PerfFuzz` (Lemieux et al., 2018) generates inputs that provoke pathological behavior across various program locations, further aiding in the identification of performance bottlenecks.Recently, a study (Ding et al., 2020) was conducted to investigate whether existing test suites are suitable for performance testing of Hadoop and Cassandra.

In ML software, if a component is activated without affecting the final predictions, it is considered redundant computation (Chen et al., 2022; Hong et al., 2020; Chen et al., 2022). Redundant computations do not enhance the model's accuracy or effectiveness and instead consume valuable computational resources, leading to inefficiency. Similar to traditional software, redundant computations in ML software can cause efficiency degradation.

A typical example of efficiency degradation occurs in ML software employing DyNN models. DyNNs are based on the philosophy that not all inputs require the activation of all DNN components for inference. By dynamically adjusting the computation paths based on the input's complexity, DyNNs aim to reduce unnecessary computational overhead, thereby improving efficiency without compromising accuracy. However, since DyNNs' dynamic routing

decisions are based on the intermediate values of the DNN, this dynamic routing method may not be robust.

Several approaches have been proposed to test efficiency degradation in ML software (Liu et al., 2023; Gao et al., 2024; Zhang et al., 2024; Tao et al., 2024; Gao et al., 2024; Varma et al., 2024; Gao et al., 2024; Müller and Quiring, 2024). For instance, `SlowPGD` (Hong et al., 2020) suggests perturbing inputs without changing their semantics to push the model to produce uniform logits, thereby forcing continued computation and degrading model efficiency. Additionally, `SlowFormer` (Navaneet et al., 2024) introduces a universal patch generation method aimed at decreasing the efficiency of vision transformers.

8.2 Robustness Evaluation & Testing for Deep Learning Software

To evaluate the robustness of deep learning software, existing methods focus on two main aspects: data-level evaluation, which investigates how data quality impacts software performance, and model-level evaluation, which directly assesses the robustness of a well-trained model.

8.2.1 Data-level Evaluation

On the data level, backdoor injection attack is proposed to evaluate how the data will affect the ML software quality (Gu et al., 2017; Liu et al., 2018; Wang et al., 2019; Saha et al., 2020; Chen et al., 2017; Yao et al., 2019; Doan et al., 2021; Wenger et al., 2021; Saha et al., 2022; Cai et al., 2022). Backdoor injection attack against DNNs usually refers to the kind of attack that utilizes a specific trojan trigger to change the output prediction of neural network during inference (Pang et al., 2022; Gao et al., 2020; Kandpal et al., 2023). According to the existing work, launching a backdoor injection attack against DNNs first requires the generation of the trojan trigger, which can be predefined (Gu et al., 2017), fixed (Liu et al., 2018), or optimized regarding certain kinds of attack goal (Liu et al., 2020, 2018; Yu et al., 2023; Chen

et al., 2023). Some works use neural activation patterns to generate triggers (Chen et al., 2017; Suciu et al., 2018). After the trojan trigger has been generated, an infected feature extractor will be either trained from scratch (Liu et al., 2018) or perturbed (Suciu et al., 2018). The infected inference model will have updated weights that respond to the trojan trigger. When triggered input is used in inference data, the backdoor attack launches, having falsified prediction results. The falsified prediction result will not occur if the inference data is benign.

Current defense methods for backdoor injection attack usually lies in two steps: defense targeting abnormal input samples, or defense targeting modified inference model (Shokri et al., 2020). Defense targeting abnormal input sample is mainly about abnormality detection. It can either relies on image domain (Paudice et al., 2018; Peri et al., 2020; Liu et al., 2022) or frequency domain (Zeng et al., 2021; Hammoud and Ghanem, 2021; Al Kader Hammoud et al., 2023; Wang et al., 2022). Affected samples can be identified by several methods such as perturbation on input samples (Gao et al., 2019). Defense deployed at image layer can also be achieved by purifying the training data (Peri et al., 2020) and eliminate the effect of backdoor trigger. Defense on model layer also includes two categories: detection and mitigation. Backdoor infected models can be identified through monitoring neuron activation(Liu et al., 2018), or defender can determine if a model is infected by model inversion and reconstruct the training set (Chen et al., 2019). Mitigation of backdoor attack can be done in several ways like deactivating suspicious neurons within DNN (Liu et al., 2018), or use distillation to eliminate the effect of backdoor in the victim model (Li et al., 2021; Yao et al., 2024; Yoshida and Fujino, 2020; Li et al., 2021; Xia et al., 2022).

8.2.2 Model-level Evaluation

At the model level, the evasion attack is the most widely used way to evaluate model (Duan et al., 2021; Yuan et al., 2021; Long et al., 2022; Szegedy et al., 2014; Goodfellow et al., 2015;

Gnanasambandam et al., 2021; Andriushchenko et al., 2020; Zhang et al., 2022). Evasion attacks will craft human-unnoticeable adversarial examples, which can fool the decision making of the model. Szegedy *et al.* (Szegedy et al., 2014) and Goodfellow *et al.* (Goodfellow et al., 2015) propose adversarial attacks on DNNs. Karmon *et al.* Adversarial attacks have been extended to various fields like natural language and speech processing (Carlini et al., 2016; Jia and Liang, 2017), and graph models (Zügner et al., 2018; Bojchevski and Günnemann, 2019). Although, all these attacks focus on changing the prediction and do not concentrate on performance testing. Several testing methods have been proposed to test DNNs (Zhang et al., 2018; Zhou et al., 2020; Chen et al., 2022, 2021). To improve the robustness of ML software for NLP applications, a series of testing methods have been proposed, which focus on accuracy testing. The core idea of existing work is to perturb seed input sentences with different perturbations and detect output inconsistency between perturbed and seed outputs. At high-level, the perturbations in existing work can be categorized into three types. *(i) character-level:* This type of perturbations (Belinkov and Bisk, 2017; Li et al., 2019; Ebrahimi et al., 2018; Zou et al., 2020; Ebrahimi et al., 2018) represents the natural typos and noises in textual inputs. For example, character swap (*e.g.,* noise → nosie), order random (*e.g.,* noise → nisoe), character insertions (*e.g.,* noise → noisde), and keyboard typo (*e.g.,* noise → noide) *(ii) token-level:* This type of perturbations (Sun et al., 2020; Zhang et al., 2021; Li et al., 2019; Ren et al., 2019; Cheng et al., 2020; Zang et al., 2020) replaces a few tokens in the seed sentences with other tokens. However, token replacement sometimes would completely change the semantic of the input text; thus, this type of perturbation usually appears in adversary scenarios; *(iii) structure-level:* Different from the above two perturbations, this type of perturbations (Li et al., 2020; He et al., 2020; Gupta et al., 2020; He et al., 2021) seeks to generate legal sentences that do not contain lexical or syntactic errors. For example, (He et al., 2020) proposes a structure invariant testing method to perturb seed inputs with `Bert` (Jin et al., 2020), and the perturbed sentences will exhibit similar sentence structure with the seed sentences.

To improve the robustness of deep learning models and defend against evasion attacks, a range of approaches has been proposed. These methods can be broadly categorized into several key strategies: (1) Adversarial Training: This involves augmenting the training dataset with adversarial examples, which helps the model learn to recognize and resist such attacks. By exposing the model to a variety of adversarial examples during training, it becomes more robust to similar attacks during inference (Bai et al., 2021; Wong et al., 2020; Xie et al., 2020; Andriushchenko and Flammarion, 2020; Tramèr et al., 2017; Goodfellow et al., 2016). (2) Defensive Distillation: This technique involves training a secondary model (a distilled model) to mimic the predictions of the original model. The idea is that the distilled model, being trained on the smoothed output of the original model, becomes more robust to adversarial perturbations (Papernot and McDaniel, 2016; Papernot et al., 2016; Goldblum et al., 2020; Catak et al., 2022). (3) Certified Defenses: These methods provide formal guarantees about the model's robustness against certain types of adversarial attacks. Techniques such as interval bound propagation and randomized smoothing offer mathematical guarantees on the model's stability within specified bounds, providing a certified level of protection against perturbations (Singh et al., 2018; Gehr et al., 2018; Zhang et al., 2018; Singh et al., 2019; Cohen et al., 2019; Zeng et al., 2023; Huang et al., 2023). (4) Input Transformation: This strategy includes preprocessing steps designed to transform or filter inputs before they are fed into the model. Examples include feature squeezing, which reduces the precision of input features to decrease the impact of adversarial perturbations, and defensive cropping, which removes suspicious regions from input (Jia et al., 2019; Agustsson et al., 2019; Dziugaite et al., 2016; Tian et al., 2018). (5) Anomaly Detection: Implementing anomaly detection mechanisms can help identify and reject adversarial inputs by distinguishing them from legitimate data. By monitoring for unusual patterns or deviations from the expected input distribution, these methods can act as an additional layer of defense (Grosse et al., 2017; Pang et al., 2018; Tramer, 2022; Sun et al., 2022; Colombo et al., 2022).

CHAPTER 9

CONCLUSION AND FUTURE WORK

This dissertation addresses the pressing need for enhancing the efficiency of machine learning (ML) software, which is increasingly pervasive in various applications but often hampered by substantial computational demands. The research focuses on optimizing ML software efficiency across three critical components: data, model, and program. By systematically investigating and addressing inefficiencies at each level, this work provides a comprehensive framework for improving the performance and sustainability of ML applications.

At the data level, this dissertation evaluates the impact of training data on the final efficiency of ML models. It identifies the significant issue of efficiency backdoor vulnerabilities in dynamic neural networks (DyNNs), which adversaries can exploit to degrade computational efficiency. To counter this, a novel "unconfident" training strategy is proposed, effectively mitigating these vulnerabilities and ensuring more reliable and efficient inference processes.

At the model level, the dissertation explores the efficiency robustness of DyNNs, identifying widespread computational efficiency vulnerabilities in state-of-the-art architectures. Through the development of `NMTSloth` and `DeepPerform`, new frameworks for testing and enhancing the computational efficiency of DyNNs are introduced. These tools demonstrate substantial improvements in identifying and addressing efficiency issues, significantly reducing latency and energy consumption.

At the program level, an empirical study highlights the limitations of existing deep learning (DL) compilers in handling DyNNs. To overcome these challenges, the dissertation presents a program rewriting approach that leverages heterogeneous control flow graphs (HCFGs) to correctly compile DyNNs. This method significantly accelerates inference time, demonstrating the potential for substantial performance gains through more effective program-level optimizations.

In summary, this dissertation makes significant contributions to the field of ML software efficiency by: Evaluating and mitigating the impact of training data on ML model efficiency. Developing tools to test and enhance the efficiency robustness of DyNNs. Proposing a novel program rewriting approach to optimize the compilation of DyNNs. Future work can build upon these contributions by exploring further integration of data, model, and program-level optimizations, as well as extending the proposed methods to other types of ML models and applications. Additionally, ongoing research into the trustworthiness, explainability, and secure deployment of ML software will continue to complement efforts to improve efficiency, ensuring that ML systems are not only performant but also reliable and secure.

REFERENCES

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng (2016). Tensorflow: A system for large-scale machine learning. In K. Keeton and T. Roscoe (Eds.), *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283. USENIX Association.

Achiam, J., S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774 abs/2303.08774*, 1–100.

Aguilar, G., Y. Ling, Y. Zhang, B. Yao, X. Fan, and C. Guo (2020). Knowledge distillation from internal representations. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 34, pp. 7350–7357.

Agustsson, E., M. Tschannen, F. Mentzer, R. Timofte, and L. V. Gool (2019). Generative adversarial networks for extreme learned image compression. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 221–231.

Al Kader Hammoud, H. A., A. Bibi, P. H. Torr, and B. Ghanem (2023). Don't freak out: A frequency-inspired approach to detecting backdoor poisoned samples in dnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2338–2345.

AllenAI (2022). https://huggingface.co/allenai/wmt16-en-de-dist-12-1.

Alon, U., S. Brody, O. Levy, and E. Yahav (2018). code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Alon, U., M. Zilberstein, O. Levy, and E. Yahav (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages 3*(POPL), 1–29.

Alwani, M., Y. Wang, and V. Madhavan (2022). Decore: Deep compression with reinforcement learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12349–12359.

Anderson, P., X. He, C. Buehler, D. Teney, M. Johnson, S. Gould, and L. Zhang (2018). Bottom-Up and Top-down Attention for Image Captioning and Visual Question Answering. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pp. 6077–6086.

Andriushchenko, M., F. Croce, N. Flammarion, and M. Hein (2020). Square attack: a query-efficient black-box adversarial attack via random search. In *European conference on computer vision*, pp. 484–501. Springer.

Andriushchenko, M. and N. Flammarion (2020). Understanding and improving fast adversarial training. *Advances in Neural Information Processing Systems 33*, 16048–16059.

Anthony, L. F. W., B. Kanding, and R. Selvan (2020). Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *arXiv preprint arXiv:2007.03051*.

Athalye, A., N. Carlini, and D. Wagner (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International conference on machine learning*, pp. 274–283. PMLR.

Bagdasaryan, E. and V. Shmatikov (2021). Blind backdoors in deep learning models. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1505–1521.

Bahdanau, D., K. Cho, and Y. Bengio (2015). Neural machine translation by jointly learning to align and translate. In Y. Bengio and Y. LeCun (Eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Bai, T., J. Luo, J. Zhao, B. Wen, and Q. Wang (2021). Recent advances in adversarial training for adversarial robustness. pp. 4312–4321.

Barlow, H. B. (1989). Unsupervised learning. *Neural computation 1*(3), 295–311.

Bateni, S. and C. Liu (2018). Apnet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018*, pp. 67–79. IEEE Computer Society.

Bateni, S. and C. Liu (2020). Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020*, pp. 371–385. USENIX Association.

Belinkov, Y. and Y. Bisk (2017). Synthetic and natural noise both break neural machine translation. *arXiv preprint arXiv:1711.02173*.

Belinkov, Y. and Y. Bisk (2018). Synthetic and natural noise both break neural machine translation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Besse, P., B. Guillouet, J.-M. Loubes, and R. François (2015). Review and perspective for distance based trajectory clustering. *arXiv preprint arXiv:1508.04904*.

Boehm, B. (2006). A view of 20th and 21st century software engineering. In *Proceedings of the 28th international conference on Software engineering*, pp. 12–29.

Bojchevski, A. and S. Günnemann (2019). Adversarial attacks on node embeddings via graph poisoning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019*, pp. 695–704. PMLR.

Bolukbasi, T., J. Wang, O. Dekel, and V. Saligrama (2017). Adaptive neural networks for efficient inference. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, pp. 527–536. PMLR.

Brendel, W., J. Rauber, M. Kümmerer, I. Ustyuzhaninov, and M. Bethge (2019). Accurate, reliable and fast robustness evaluation. In *Advances in Neural Information Processing Systems*, pp. 12861–12871.

Burnim, J., S. Juvekar, and K. Sen (2009). WISE: automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009*, pp. 463–473. IEEE.

Cai, X., H. Xu, S. Xu, Y. Zhang, et al. (2022). Badprompt: Backdoor attacks on continuous prompts. *Advances in Neural Information Processing Systems 35*, 37068–37080.

Campbell, M., M. Egerstedt, J. P. How, and R. M. Murray (2010). Autonomous driving in urban environments: approaches, lessons and challenges. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 368*(1928), 4649–4672.

Carlini, N., P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. A. Wagner, and W. Zhou (2016). Hidden voice commands. In *25th USENIX Security Symposium, USENIX Security 2016*, pp. 513–530. USENIX Association.

Carlini, N., F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, et al. (2020). Extracting training data from large language models. *arXiv preprint arXiv:2012.07805*.

Carlini, N. and D. Wagner (2017). Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pp. 39–57. IEEE.

Caswell, I. and B. Liang (2020). Recent advances in google translate.

Catak, F. O., M. Kuzlu, E. Catak, U. Cali, and O. Guler (2022). Defensive distillation-based adversarial attack mitigation method for channel estimation using deep learning models in next-generation wireless networks. *IEEE Access 10*, 98191–98203.

Chan, W., N. Jaitly, Q. V. Le, and O. Vinyals (2016). Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016*, pp. 4960–4964. IEEE.

Chao, S.-K., Z. Wang, Y. Xing, and G. Cheng (2020). Directional pruning of deep neural networks. *Advances in neural information processing systems 33*, 13986–13998.

Chen, H., C. Fu, J. Zhao, and F. Koushanfar (2019). Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks. In *IJCAI*, pp. 8.

Chen, J., W. Shang, and E. Shihab (2020). Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering 48*(5), 1529–1544.

Chen, S., S. Bateni, S. Grandhi, X. Li, C. Liu, and W. Yang (2020). Denas: automated rule generation by knowledge extraction from neural networks. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 813–825.

Chen, S., H. Chen, M. Haque, C. Liu, and W. Yang (2023). The dark side of dynamic routing neural networks: Towards efficiency backdoor injection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 24585–24594.

Chen, S., X. Feng, X. Han, C. Liu, and W. Yang (2024). Ppm: Automated generation of diverse programming problems for benchmarking code generation models. *arXiv preprint arXiv:2401.15545*.

Chen, S., M. Haque, C. Liu, and W. Yang (2022). Deepperform: An efficient approach for performance testing of resource-constrained neural networks. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pp. 31:1–31:13. ACM.

Chen, S., M. Haque, Z. Song, C. Liu, and W. Yang (2021). Transslowdown: Efficiency attacks on neural machine translation systems. OpenReview.net.

Chen, S., H. Khanpour, C. Liu, and W. Yang (2022a). Learn to reverse dnns from AI programs automatically. In L. D. Raedt (Ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pp. 666–672. ijcai.org.

Chen, S., H. Khanpour, C. Liu, and W. Yang (2022b). Learning to reverse dnns from ai programs automatically. *arXiv preprint arXiv:2205.10364*.

Chen, S., C. Liu, M. Haque, Z. Song, and W. Yang (2022). Nmtsloth: understanding and testing efficiency degradation of neural machine translation systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1148–1160.

Chen, S., Z. Song, M. Haque, C. Liu, and W. Yang (2022). Nicgslowdown: Evaluating the efficiency robustness of neural image caption generation models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15365–15374.

Chen, S., Z. Song, L. Ma, C. Liu, and W. Yang (2021). Attackdist: Characterizing zero-day adversarial samples by counter attack.

Chen, S., S. Wei, Cong, and W. Yang (2023). Reproducation package for "dycl: Dynamic neural network compilation via program rewriting and graph optimization".

Chen, T., T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy (2018). Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799 11*, 20.

Chen, T., T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. (2018). {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594.

Chen, T., H. Zhang, Z. Zhang, S. Chang, S. Liu, P.-Y. Chen, and Z. Wang (2022). Linearity grafting: Relaxed neuron pruning helps certifiable robustness. In *International Conference on Machine Learning*, pp. 3760–3772. PMLR.

Chen, X., M. Li, H. Zhong, Y. Ma, and C.-H. Hsu (2021). Dnnoff: offloading dnn-based intelligent iot applications in mobile edge computing. *IEEE transactions on industrial informatics 18*(4), 2820–2829.

Chen, X., C. Liu, B. Li, K. Lu, and D. Song (2017). Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*.

Chen, X., J. Zhang, B. Lin, Z. Chen, K. Wolter, and G. Min (2021). Energy-efficient offloading for dnn-based smart iot systems in cloud-edge environments. *IEEE Transactions on Parallel and Distributed Systems 33*(3), 683–697.

Chen, Y., S. Chen, Z. Li, W. Yang, C. Liu, R. T. Tan, and H. Li (2023). Dynamic transformers provide a false sense of efficiency. *arXiv preprint arXiv:2305.12228*.

Chen, Y., Z. Ma, W. Fang, X. Zheng, Z. Yu, and Y. Tian (2023). A unified framework for soft threshold pruning. *arXiv preprint arXiv:2302.13019*.

Cheng, M., J. Yi, P.-Y. Chen, H. Zhang, and C.-J. Hsieh (2020). Seq2sick: Evaluating the robustness of sequence-to-sequence models with adversarial examples. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Volume 34, pp. 3601–3608.

Chi, Z., T. Zheng, H. Li, Z. Yang, B. Wu, B. Lin, and D. Cai (2023). Normkd: Normalized logits for knowledge distillation. *arXiv preprint arXiv:2308.00520*.

Chiaro, R. D., B. Twardowski, A. D. Bagdanov, and J. van de Weijer (2020). RATT: Recurrent Attention to Transient Tasks for Continual Image Captioning. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, pp. 16736–16748.

Chijiwa, D., S. Yamaguchi, Y. Ida, K. Umakoshi, and T. Inoue (2021). Pruning randomly initialized neural networks with iterative randomization. *Advances in neural information processing systems 34*, 4503–4513.

Cho, J. H. and B. Hariharan (2019). On the efficacy of knowledge distillation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4794–4802.

Chowdhury, S. S., N. Rathi, and K. Roy (2022). Towards ultra low latency spiking neural networks for vision and sequential tasks using temporal pruning. In *European Conference on Computer Vision*, pp. 709–726. Springer.

Cohen, J., E. Rosenfeld, and Z. Kolter (2019). Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning*, pp. 1310–1320. PMLR.

Colombo, P., E. Dadalto, G. Staerman, N. Noiry, and P. Piantanida (2022). Beyond mahalanobis distance for textual ood detection. *Advances in Neural Information Processing Systems 35*, 17744–17759.

Cornia, M., M. Stefanini, L. Baraldi, and R. Cucchiara (2020). Meshed-memory Transformer for Image Captioning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pp. 10575–10584.

Cui, C., Y. Ma, X. Cao, W. Ye, Y. Zhou, K. Liang, J. Chen, J. Lu, Z. Yang, K.-D. Liao, et al. (2024). A survey on multimodal large language models for autonomous driving. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 958–979.

Cui, Z., X. Xu, X. Fei, X. Cai, Y. Cao, W. Zhang, and J. Chen (2020). Personalized recommendation system based on collaborative filtering for iot scenarios. *IEEE Transactions on Services Computing 13*(4), 685–695.

Cyphers, S., A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. K. Vijay, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb (2018). Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR abs/1801.08058*.

Dalvi, N., P. Domingos, S. Sanghai, D. Verma, et al. (2004). Adversarial Classification. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pp. 99–108.

Daly, D. (2021). Creating a virtuous cycle in performance testing at mongodb. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pp. 33–41.

Davis, A. S. and I. Arel (2014). Low-rank approximations for conditional feedforward computation in deep neural networks. In *2nd International Conference on Learning Representations, ICLR 2014*.

Dettmers, T., M. Lewis, Y. Belkada, and L. Zettlemoyer (2022). Llm.int8(): 8-bit matrix multiplication for transformers at scale.

Dettmers, T., A. Pagnoni, A. Holtzman, and L. Zettlemoyer (2023). Qlora: Efficient finetuning of quantized llms.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Diao, E., G. Wang, J. Zhan, Y. Yang, J. Ding, and V. Tarokh (2023). Pruning deep neural networks from a sparsity perspective. *arXiv preprint arXiv:2302.05601*.

Diffenderfer, J., B. Bartoldson, S. Chaganti, J. Zhang, and B. Kailkhura (2021). A winning hand: Compressing deep networks can improve out-of-distribution robustness. *Advances in neural information processing systems 34*, 664–676.

Dilhara, M., A. Ketkar, and D. Dig (2021). Understanding software-2.0: A study of machine learning library usage and evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM) 30*(4), 1–42.

Ding, Y., L. Zhu, Z. Jia, G. Pekhimenko, and S. Han (2021). IOS: inter-operator scheduler for CNN acceleration. In A. Smola, A. Dimakis, and I. Stoica (Eds.), *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org.

Ding, Z., J. Chen, and W. Shang (2020). Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1435–1446.

Doan, K., Y. Lao, W. Zhao, and P. Li (2021). Lira: Learnable, imperceptible and robust backdoor attacks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 11966–11976.

Drori, I., Y. Krishnamurthy, R. Rampin, R. d. P. Lourenco, J. P. Ono, K. Cho, C. Silva, and J. Freire (2021). Alphad3m: Machine learning pipeline synthesis. *arXiv preprint arXiv:2111.02508*.

Duan, R., Y. Chen, D. Niu, Y. Yang, A. K. Qin, and Y. He (2021). Advdrop: Adversarial attack to dnns by dropping information. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7506–7515.

Dziugaite, G. K., Z. Ghahramani, and D. M. Roy (2016). A study of the effect of jpg compression on adversarial images. *arXiv preprint arXiv:1608.00853*.

Ebrahimi, J., D. Lowd, and D. Dou (2018, August). On adversarial examples for character-level neural machine translation. In *Proceedings of the 27th International Conference on Computational Linguistics*, Santa Fe, New Mexico, USA, pp. 653–663. Association for Computational Linguistics.

Ebrahimi, J., A. Rao, D. Lowd, and D. Dou (2018, July). HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Melbourne, Australia, pp. 31–36. Association for Computational Linguistics.

Eccles, B. J., P. Rodgers, P. Kilpatrick, I. Spence, and B. Varghese (2024). Dnnshifter: An efficient dnn pruning system for edge computing. *Future Generation Computer Systems 152*, 43–54.

Eisele, A. and Y. Chen (2010, May). MultiUN: A multilingual corpus from united nation documents. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta. European Language Resources Association (ELRA).

Eykholt, K., I. Evtimov, E. Fernandes, B. Li, A. Rahmati, F. Tramer, A. Prakash, T. Kohno, and D. Song (2018). Physical adversarial examples for object detectors.

Eykholt, K., I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song (2018). Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1625–1634.

Facebook (2018). Facebook glow.

Fan, W., Z. Zhao, J. Li, Y. Liu, X. Mei, Y. Wang, J. Tang, and Q. Li (2023). Recommender systems in the era of large language models (llms). *arXiv preprint arXiv:2307.02046*.

Fang, J., Y. Shen, Y. Wang, and L. Chen (2021). Eto: accelerating optimization of dnn operators by high-performance tensor program reuse. *Proceedings of the VLDB Endowment 15*(2), 183–195.

Fang, J., Y. Yu, C. liang Zhao, and J. Zhou (2020). Turbotransformers: an efficient gpu serving system for transformer models. *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Fegade, P., T. Chen, P. Gibbons, and T. Mowry (2021). Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems 3*, 38–54.

Feichtenhofer, C., H. Fan, J. Malik, and K. He (2019). Slowfast networks for video recognition. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019*, pp. 6201–6210. IEEE.

Feng, X., X. Han, S. Chen, and W. Yang (2024). Llmeffichecker: Understanding and testing efficiency degradation of large language models. *ACM Transactions on Software Engineering and Methodology*.

Figurnov, M., M. D. Collins, Y. Zhu, L. Zhang, J. Huang, D. Vetrov, and R. Salakhutdinov (2017). Spatially Adaptive Computation Time for Residual Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1039–1048.

Floridi, L. and M. Chiriatti (2020). Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines 30*, 681–694.

Frantar, E. and D. Alistarh (2022). Optimal brain compression: A framework for accurate post-training quantization and pruning. *Advances in Neural Information Processing Systems 35*, 4475–4488.

Frantar, E., S. Ashkboos, T. Hoefler, and D. Alistarh (2023). Gptq: Accurate post-training quantization for generative pre-trained transformers.

Gan, Z., C. Gan, X. He, Y. Pu, K. Tran, J. Gao, L. Carin, and L. Deng (2017). Semantic Compositional Networks for Visual Captioning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pp. 1141–1150.

Gao, C., Y. Zheng, W. Wang, F. Feng, X. He, and Y. Li (2024). Causal inference in recommender systems: A survey and future directions. *ACM Transactions on Information Systems 42*(4), 1–32.

Gao, K., Y. Bai, J. Gu, S.-T. Xia, P. Torr, Z. Li, and W. Liu (2024). Inducing high energy-latency of large vision-language models with verbose images. *arXiv preprint arXiv:2401.11170*.

Gao, K., J. Gu, Y. Bai, S.-T. Xia, P. Torr, W. Liu, and Z. Li (2024). Energy-latency manipulation of multi-modal large language models via verbose samples. *arXiv preprint arXiv:2404.16557*.

Gao, M., R. Yu, A. Li, V. I. Morariu, and L. S. Davis (2018). Dynamic zoom-in network for fast object detection in large images. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018*, pp. 6926–6935. Computer Vision Foundation / IEEE Computer Society.

Gao, X., Y. Chen, X. Yue, Y. Tsao, and N. F. Chen (2024). Ttslow: Slow down text-to-speech with efficiency robustness evaluations. *arXiv preprint arXiv:2407.01927*.

Gao, X., Y. Zhao, L. Dudziak, R. D. Mullins, and C. Xu (2019). Dynamic channel pruning: Feature boosting and suppression.

Gao, Y., B. G. Doan, Z. Zhang, S. Ma, J. Zhang, A. Fu, S. Nepal, and H. Kim (2020). Backdoor attacks and countermeasures on deep learning: A comprehensive review. *arXiv preprint arXiv:2007.10760*.

Gao, Y., C. Xu, D. Wang, S. Chen, D. C. Ranasinghe, and S. Nepal (2019). Strip: A defence against trojan attacks on deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 113–125.

Gehr, T., M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev (2018). Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*, pp. 3–18. IEEE.

Ghodrati, A., B. E. Bejnordi, and A. Habibian (2021). Frameexit: Conditional early exiting for efficient video recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15608–15618.

Gnanasambandam, A., A. M. Sherman, and S. H. Chan (2021). Optical adversarial attack. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 92–101.

Goldblum, M., L. Fowl, S. Feizi, and T. Goldstein (2020). Adversarially robust distillation. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 34, pp. 3996–4003.

Good, A., J. Lin, X. Yu, H. Sieg, M. Fergurson, S. Zhe, J. Wieczorek, and T. Serra (2022). Recall distortion in neural network pruning and the undecayed pruning algorithm. *Advances in Neural Information Processing Systems 35*, 32762–32776.

Goodfellow, I., Y. Bengio, A. Courville, and Y. Bengio (2016). *Deep learning*, Volume 1. MIT Press.

Goodfellow, I. J., J. Shlens, and C. Szegedy (2015). Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR 2015*.

Google (2017). TensorFlow Lite. `https://www.tensorflow.org/lite`.

Google (2022). https://huggingface.co/t5-small.

Gou, J., B. Yu, S. J. Maybank, and D. Tao (2021). Knowledge distillation: A survey. *International Journal of Computer Vision 129*(6), 1789–1819.

Graves, A. (2016). Adaptive Computation Time for Recurrent Neural Networks. *arXiv preprint arXiv:1603.08983*.

Grosse, K., P. Manoharan, N. Papernot, M. Backes, and P. McDaniel (2017). On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*.

Grosse, K., N. Papernot, P. Manoharan, M. Backes, and P. McDaniel (2016). Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*.

Gu, J., H. Hassan Awadalla, and J. Devlin (2018, June). Universal neural machine translation for extremely low resource languages. In *NAACL*.

Gu, T., B. Dolan-Gavitt, and S. Garg (2017). Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*.

Gu, Y., L. Dong, F. Wei, and M. Huang (2024). Minillm: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*.

Guan, J., Y. Liu, Q. Liu, and J. Peng (2017). Energy-efficient Amortized Inference with Cascaded Deep Classifiers. *arXiv preprint arXiv:1710.03368*.

Guan, Y., C. Yu, Y. Zhou, J. Leng, C. Li, and M. Guo (2024). Fractal: Joint multi-level sparse pattern tuning of accuracy and performance for dnn pruning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 416–430.

Gupta, S., P. He, C. Meister, and Z. Su (2020). Machine translation testing via pathological invariance. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 863–875.

Hammoud, H. A. A. K. and B. Ghanem (2021). Check your other door! creating backdoor attacks in the frequency domain. *arXiv preprint arXiv:2109.05507*.

Han, S., H. Mao, and W. J. Dally (2015). Deep Compression: Compressing Deep neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149*.

Han, Y., G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang (2021a). Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 44*(11), 7436–7456.

Han, Y., G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang (2021b). Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Hao, C., X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen (2019). Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6.

Hapke, H. and C. Nelson (2020). *Building machine learning pipelines*. O'Reilly Media.

Haque, M., A. Chauhan, C. Liu, and W. Yang (2020a). Ilfo: Adversarial attack on adaptive neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14264–14273.

Haque, M., A. Chauhan, C. Liu, and W. Yang (2020b). ILFO: Adversarial attack on adaptive neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020*, pp. 14252–14261. IEEE.

Haque, M., S. Chen, W. A. Haque, C. Liu, and W. Yang (2021). Nodeattack: Adversarial attack on the energy consumption of neural odes.

Hassan Awadalla, H., A. Aue, C. Chen, V. Chowdhary, J. Clark, C. Federmann, X. Huang, M. Junczys-Dowmunt, W. Lewis, M. Li, S. Liu, T.-Y. Liu, R. Luo, A. Menezes, T. Qin, F. Seide, X. Tan, F. Tian, L. Wu, S. Wu, Y. Xia, D. Zhang, Z. Zhang, and M. Zhou (2018, March). Achieving human parity on automatic chinese to english news translation. arXiv:1803.05567.

Hassan Awadalla, H., M. Elaraby, A. Tawfik, M. Khaled, and A. Osama (2018, February). Gender aware spoken language translation applied to english-arabic. In *2018 IEEE Proceedings of the Second International Conference on Natural Language and Speech Processing* (2018 IEEE Proceedings of the Second International Conference on Natural Language and Speech Processing ed.).

Hassan Awadalla, H., M. Elaraby, and A. Y. Tawfik (2017, December). Synthetic data for neural machine translation of spoken-dialects. In *IWSLT 2017* (IWSLT 2017 ed.).

Hausdorff, F. (1914). *Grundzüge der mengenlehre*, Volume 7. von Veit.

Hayou, S., J.-F. Ton, A. Doucet, and Y. W. Teh (2020). Robust pruning at initialization. *arXiv preprint arXiv:2002.08797*.

He, K., X. Zhang, S. Ren, and J. Sun (2016). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

He, P. (2022). Robustnlp library.

He, P., C. Meister, and Z. Su (2020). Structure-invariant testing for machine translation. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 961–973. IEEE.

He, P., C. Meister, and Z. Su (2021). Testing machine translation via referential transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 410–422. IEEE.

He, Y. and L. Xiao (2024). Structured pruning for deep convolutional neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 46*(5), 2900–2919.

Helsinki-NLP (2022). https://huggingface.co/helsinki-nlp/opus-mt-en-de.

Hendrik Metzen, J., M. Chaithanya Kumar, T. Brox, and V. Fischer (2017, Oct). Universal adversarial perturbations against semantic image segmentation. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Hoang, D. N. and S. Liu (2023). Revisiting pruning at initialization through the lens of ramanujan graph. *ICLR 2023*.

Hong, S., Y. Kaya, I.-V. Modoranu, and T. Dumitraş (2020). A panda? no, it's a sloth: Slowdown attacks on adaptive multi-exit neural network inference. *arXiv preprint arXiv:2010.02432*.

Hou, L., Z. Huang, L. Shang, X. Jiang, X. Chen, and Q. Liu (2020). Dynabert: Dynamic bert with adaptive width and depth. *Advances in Neural Information Processing Systems 33*, 9782–9793.

Hou, Z., M. Qin, F. Sun, X. Ma, K. Yuan, Y. Xu, Y.-K. Chen, R. Jin, Y. Xie, and S.-Y. Kung (2022). Chex: Channel exploration for cnn model compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12287–12298.

Howard, A. G., M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Hu, C., W. Bao, D. Wang, and F. Liu (2019). Dynamic adaptive dnn surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 1423–1431. IEEE.

Hu, Y., R. Ghosh, and R. Govindan (2021). Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 624–638.

Hua, W., Y. Zhou, C. M. De Sa, Z. Zhang, and G. E. Suh (2019). Channel gating neural networks. In *Advances in Neural Information Processing Systems*, pp. 1884–1894.

Hua, W., Y. Zhou, C. D. Sa, Z. Zhang, and G. E. Suh (2019). Channel gating neural networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, pp. 1884–1894.

Huang, G., D. Chen, T. Li, F. Wu, L. Van Der Maaten, and K. Q. Weinberger (2017). Multi-scale dense networks for resource efficient image classification. *arXiv preprint arXiv:1703.09844*.

Huang, L., W. Wang, J. Chen, and X. Wei (2019). Attention on Attention for Image Captioning. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pp. 4633–4642.

Huang, R., J. Pedoeem, and C. Chen (2018). YOLO-LITE: A real-time object detection algorithm optimized for non-gpu computers. In *IEEE International Conference on Big Data (IEEE BigData 2018)*, pp. 2503–2510. IEEE.

Huang, Z., N. G. Marchant, K. Lucas, L. Bauer, O. Ohrimenko, and B. Rubinstein (2023). Rs-del: Edit distance robustness certificates for sequence classifiers via randomized deletion. *Advances in Neural Information Processing Systems 36*, 18676–18711.

Jagielski, M., N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot (2020). High accuracy and high fidelity extraction of neural networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 1345–1362.

James, G., D. Witten, T. Hastie, and R. Tibshirani (2013). *An introduction to statistical learning*, Volume 112. Springer.

Jia, R. and P. Liang (2017). Adversarial examples for evaluating reading comprehension systems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017*, pp. 2021–2031. Association for Computational Linguistics.

Jia, X., X. Wei, X. Cao, and H. Foroosh (2019). Comdefend: An efficient image compression model to defend adversarial examples. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 6084–6092.

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678.

Jiang, S., Z. Lin, Y. Li, Y. Shu, and Y. Liu (2021). Flexible high-resolution object detection on edge devices with tunable latency. In *The 27th Annual International Conference on Mobile Computing and Networking,ACM MobiCom 2021*, pp. 559–572. ACM.

Jin, D., Z. Jin, J. T. Zhou, and P. Szolovits (2020). Is bert really robust? a strong baseline for natural language attack on text classification and entailment. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 34, pp. 8018–8025.

Jocher, G., A. Chaurasia, A. Stoken, J. Borovec, Y. Kwon, J. Fang, K. Michael, D. Montes, J. Nadar, P. Skalski, et al. (2022). ultralytics/yolov5: v6. 1-tensorrt, tensorflow edge tpu and openvino export and inference. *Zenodo*.

Jörg, T., H. Sam, and S. Sam (2020). https://blogs.helsinki.fi/language-technology/.

Kalchbrenner, N., L. Espeholt, K. Simonyan, A. van den Oord, A. Graves, and K. Kavukcuoglu (2016). Neural machine translation in linear time. *CoRR abs/1610.10099*.

Kandpal, N., M. Jagielski, F. Tramèr, and N. Carlini (2023). Backdoor attacks for in-context learning with language models. *arXiv preprint arXiv:2307.14692*.

Kaya, Y., S. Hong, and T. Dumitras (2019). Shallow-deep networks: Understanding and mitigating network overthinking. In *International conference on machine learning*, pp. 3301–3310. PMLR.

Kim, T., Y. Kwon, J. Lee, T. Kim, and S. Ha (2022). Cprune: Compiler-informed model pruning for efficient target-aware dnn execution. In *European Conference on Computer Vision*, pp. 651–667. Springer.

Kim, Y. and A. M. Rush (2016). Sequence-level knowledge distillation. *arXiv preprint arXiv:1606.07947*.

Kocić, J., N. Jovičić, and V. Drndarević (2019). An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. *Sensors 19*(9), 2064.

Koehn, P., F. J. Och, and D. Marcu (2003). Statistical phrase-based translation. Technical report, University of Southern California Marina Del Rey Information Sciences Inst.

Koutsoukos, D., S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi (2021). Tensors: An abstraction for general data processing. *Proceedings of the VLDB Endowment 14*(10), 1797–1804.

Krishna, K., G. S. Tomar, A. P. Parikh, N. Papernot, and M. Iyyer (2019). Thieves on sesame street! model extraction of bert-based apis. *arXiv preprint arXiv:1910.12366*.

Krizhevsky, A., G. Hinton, et al. (2009). Learning multiple layers of features from tiny images. *arXiv preprint arXiv:1805.12549*.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pp. 1106–1114.

Kurakin, A., I. J. Goodfellow, and S. Bengio (2017). Adversarial examples in the physical world. In *5th International Conference on Learning Representations, ICLR 2017*. OpenReview.net.

Kurtoglu, T. and I. Y. Tumer (2008). A graph-based fault identification and propagation framework for functional design of complex systems. *Journal of mechanical design 130*(5).

Laskaridis, S., A. Kouris, and N. D. Lane (2021). Adaptive inference through early-exit networks: Design, challenges and directions. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, pp. 1–6.

Lau, F., S. H. Rubin, M. H. Smith, and L. Trajkovic (2000). Distributed denial of service attacks. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, Volume 3, pp. 2275–2280. IEEE.

Lee, J., S. Chen, A. Mordahl, C. Liu, W. Yang, and S. Wei (2024). Automated testing linguistic capabilities of nlp models. *ACM Transactions on Software Engineering and Methodology*.

Lee, J., S. Kang, J. Lee, D. Shin, D. Han, and H.-J. Yoo (2020). The hardware and algorithm co-design for energy-efficient dnn processor on edge/mobile devices. *IEEE Transactions on Circuits and Systems I: Regular Papers 67*(10), 3458–3470.

Lee, N., T. Ajanthan, S. Gould, and P. H. Torr (2019). A signal propagation perspective for pruning neural networks at initialization. *arXiv preprint arXiv:1906.06307*.

Lemieux, C., R. Padhye, K. Sen, and D. Song (2018). Perffuzz: automatically generating pathological inputs. In F. Tip and E. Bodden (Eds.), *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018*, pp. 254–265. ACM.

Leroux, S., P. Molchanov, P. Simoens, B. Dhoedt, T. Breuel, and J. Kautz (2018). Iamnn: Iterative and adaptive mobile neural network for efficient image classification. *arXiv preprint arXiv:1804.10123*.

Levinson, J., J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, et al. (2011). Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE intelligent vehicles symposium (IV)*, pp. 163–168. IEEE.

Lewis, G. A., I. Ozkaya, and X. Xu (2021). Software architecture challenges for ml systems. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 634–638. IEEE.

Li, H., X. Li, Q. Fan, Q. He, X. Wang, and V. C. Leung (2024). Distributed dnn inference with fine-grained model partitioning in mobile edge computing networks. *IEEE Transactions on Mobile Computing*.

Li, J., S. Ji, T. Du, B. Li, and T. Wang (2019). Textbugger: Generating adversarial text against real-world applications. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

Li, L., R. Ma, Q. Guo, X. Xue, and X. Qiu (2020, November). BERT-ATTACK: Adversarial attack against BERT using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Online, pp. 6193–6202. Association for Computational Linguistics.

Li, L., Y. Zhang, and L. Chen (2023). Prompt distillation for efficient llm-based recommendation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pp. 1348–1357.

Li, M., Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian (2020a). The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems 32*(3), 708–727.

Li, M., Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian (2020b). The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems 32*(3), 708–727.

Li, S. (2020). Tensorflow lite: On-device machine learning framework. *Journal of Computer Research and Development 57*(9), 1839.

Li, S., A. Neupane, S. Paul, C. Song, S. V. Krishnamurthy, A. K. Roy-Chowdhury, and A. Swami (2019). Stealthy adversarial perturbations against real-time video classification systems. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society.

Li, S., J. Wu, X. Xiao, F. Chao, X. Mao, and R. Ji (2021). Revisiting discriminator in gan compression: A generator-discriminator cooperative compression scheme. *Advances in Neural Information Processing Systems 34*, 28560–28572.

Li, Y., S. Chen, Y. Guo, W. Yang, Y. Dong, and C. Liu (2024). Uncertainty awareness of large language models under code distribution shifts: A benchmark study. *arXiv preprint arXiv:2402.05939*.

Li, Y., S. Chen, and W. Yang (2021). Estimating predictive uncertainty under program data distribution shift. *arXiv preprint arXiv:2107.10989*.

Li, Y., Z. Han, Q. Zhang, Z. Li, and H. Tan (2020). Automating cloud deployment for deep learning inference of real-time online services. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 1668–1677. IEEE.

Li, Y., X. Lyu, N. Koren, L. Lyu, B. Li, and X. Ma (2021). Neural attention distillation: Erasing backdoor triggers from deep neural networks. *arXiv preprint arXiv:2101.05930*.

Li, Y., J. Yang, Y. Song, L. Cao, J. Luo, and L.-J. Li (2017). Learning from noisy labels with distillation. In *Proceedings of the IEEE international conference on computer vision*, pp. 1910–1918.

Li, Y., H. Zhong, X. Ma, Y. Jiang, and S.-T. Xia (2022). Few-shot backdoor attacks on visual object tracking. *arXiv preprint arXiv:2201.13178*.

Li, Z., Q. Wang, F. Cole, R. Tucker, and N. Snavely (2023). Dynibar: Neural dynamic image-based rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4273–4284.

Li, Z., B. Yin, T. Yao, J. Guo, S. Ding, S. Chen, and C. Liu (2023). Sibling-attack: Rethinking transferable adversarial attacks against face recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 24626–24637.

Li, Z., G. Yuan, W. Niu, P. Zhao, Y. Li, Y. Cai, X. Shen, Z. Zhan, Z. Kong, Q. Jin, et al. (2021). Npas: A compiler-aware framework of unified network pruning and architecture search for beyond real-time mobile acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14255–14266.

Li, Z., D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

Liang, S., X. Wei, and X. Cao (2021). Generate more imperceptible adversarial examples for object detection. In *ICML 2021 Workshop on Adversarial Machine Learning*.

Lim, M.-H., J.-G. Lou, H. Zhang, Q. Fu, A. B. J. Teoh, Q. Lin, R. Ding, and D. Zhang (2014). Identifying recurrent and unknown performance issues. In *2014 IEEE International Conference on Data Mining*, pp. 320–329. IEEE.

Lin, J., Y. Rao, J. Lu, and J. Zhou (2017). Runtime neural pruning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pp. 2181–2191.

Lin, J., J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han (2024). Awq: Activation-aware weight quantization for llm compression and acceleration.

Liu, C., Z. Zhao, S. Süsstrunk, and M. Salzmann (2022). Robust binary models by pruning randomly-initialized networks. *Advances in Neural Information Processing Systems 35*, 492–506.

Liu, H., Y. Wu, Z. Yu, Y. Vorobeychik, and N. Zhang (2023). Slowlidar: Increasing the latency of lidar-based detection using adversarial examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5146–5155.

Liu, J., J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang (2023). Nnsmith: Generating diverse and valid test cases for deep learning compilers. In T. M. Aamodt, N. D. E. Jerger, and M. M. Swift (Eds.), *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pp. 530–543. ACM.

Liu, K., B. Dolan-Gavitt, and S. Garg (2018). Fine-pruning: Defending against backdooring attacks on deep neural networks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 273–294. Springer.

Liu, L. and J. Deng (2018a). Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pp. 3675–3682. AAAI Press.

Liu, L. and J. Deng (2018b). Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Liu, X., L. Mou, H. Cui, Z. Lu, and S. Song (2020). Finding decision jumps in text classification. *Neurocomputing 371*, 177–187.

Liu, Y., J. Gu, N. Goyal, X. Li, S. Edunov, M. Ghazvininejad, M. Lewis, and L. Zettlemoyer (2020). Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics 8*, 726–742.

Liu, Y., S. Ma, Y. Aafer, W. Lee, J. Zhai, W. Wang, and X. Zhang (2018). Trojaning attack on neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

Liu, Y., X. Ma, J. Bailey, and F. Lu (2020). Reflection backdoor: A natural backdoor attack on deep neural networks. In *European Conference on Computer Vision*, pp. 182–199. Springer.

Liu, Y., G. Shen, G. Tao, Z. Wang, S. Ma, and X. Zhang (2021). Ex-ray: Distinguishing injected backdoor from natural features in neural networks by examining differential feature symmetry.

Liu, Y., G. Shen, G. Tao, Z. Wang, S. Ma, and X. Zhang (2022). Complex backdoor detection by symmetric feature differencing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15003–15013.

Liu, Z., B. Oguz, C. Zhao, E. Chang, P. Stock, Y. Mehdad, Y. Shi, R. Krishnamoorthi, and V. Chandra (2023). Llm-qat: Data-free quantization aware training for large language models.

Long, Y., Q. Zhang, B. Zeng, L. Gao, X. Liu, J. Zhang, and J. Song (2022). Frequency domain model augmentation for adversarial attack. In *European conference on computer vision*, pp. 549–566. Springer.

Lopes, R. G., S. Fenu, and T. Starner (2017). Data-free knowledge distillation for deep neural networks. *arXiv preprint arXiv:1710.07535*.

Luo, C. and A. L. Yuille (2019). Grouped spatial-temporal aggregation for efficient action recognition. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019*, pp. 5511–5520. IEEE.

Luo, Q., S. Hu, C. Li, G. Li, and W. Shi (2021). Resource scheduling in edge computing: A survey. *IEEE Communications Surveys & Tutorials 23*(4), 2131–2165.

Lyubomirsky, S. S. (2022). *Compiler and Runtime Techniques for Optimizing Deep Learning Applications*. Ph. D. thesis, University of Washington.

Ma, L., F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, et al. (2018). Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pp. 120–131. ACM.

Ma, X., M. Qin, F. Sun, Z. Hou, K. Yuan, Y. Xu, Y. Wang, Y.-K. Chen, R. Jin, and Y. Xie (2021). Effective model sparsification by scheduled grow-and-prune methods. *arXiv preprint arXiv:2106.09857*.

Madry, A., A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu (2018). Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018*. OpenReview.net.

Martinez, J., J. Shewakramani, T. W. Liu, I. A. Bârsan, W. Zeng, and R. Urtasun (2021). Permute, quantize, and fine-tune: Efficient compression of neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 15699–15708.

Mbata, A., Y. Sripada, and M. Zhong (2024). A survey of pipeline tools for data engineering. *arXiv preprint arXiv:2406.08335*.

Michalski, R. S., J. G. Carbonell, and T. M. Mitchell (1983). *Machine Learning: An Artificial Intelligence Approach, Vol. I.* Palo Alto, CA: Tioga.

Microsoft (2017). Onnx runtime: a cross-platform inference and training machine-learning accelerator. `https://github.com/microsoft/onnxruntime`.

Mirzadeh, S. I., M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh (2020). Improved knowledge distillation via teacher assistant. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 34, pp. 5191–5198.

Molchanov, P., S. Tyree, T. Karras, T. Aila, and J. Kautz (2016). Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*.

Müller, A. and E. Quiring (2024). The impact of uniform inputs on activation sparsity and energy-latency attacks in computer vision. *arXiv preprint arXiv:2403.18587*.

Najibi, M., B. Singh, and L. Davis (2019). Autofocus: Efficient multi-scale inference. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019*, pp. 9744–9754. IEEE.

Nan, F. and V. Saligrama (2017a). Adaptive classification for prediction under a budget. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pp. 4727–4737.

Nan, F. and V. Saligrama (2017b). Adaptive Classification for Prediction Under a Budget. In *Proceedings of the Advances in Neural Information Processing Systems*, pp. 4727–4737.

Narayanan, D., M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al. (2021). Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15.

Navaneet, K., S. A. Koohpayegani, E. Sleiman, and H. Pirsiavash (2024). Slowformer: Adversarial attack on compute and energy consumption of efficient vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 24786–24797.

Needham, R. M. (1993). Denial of service. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pp. 151–153.

Netzer, Y., T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng (2011). Reading digits in natural images with unsupervised feature learning. *arXiv preprint arXiv:1503.02531*.

Ng, N., K. Yee, A. Baevski, M. Ott, M. Auli, and S. Edunov (2019). Facebook fair's wmt19 news translation task submission. *arXiv preprint arXiv:1907.06616*.

Nguyen, A. and A. Tran (2021). Wanet–imperceptible warping-based backdoor attack. *arXiv preprint arXiv:2102.10369*.

Nonnenmacher, M., T. Pfeil, I. Steinwart, and D. Reeb (2021). Sosp: Efficiently capturing global correlations by second-order structured pruning. *arXiv preprint arXiv:2110.11395*.

Nunes, J. D., M. Carvalho, D. Carneiro, and J. S. Cardoso (2022). Spiking neural networks: A survey. *IEEE Access 10*, 60738–60764.

Oh, S. J., B. Schiele, and M. Fritz (2019). Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pp. 121–144. Springer.

O'shea, K. and R. Nash (2015). An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*.

Pan, Y., T. Yao, Y. Li, and T. Mei (2020). X-Linear Attention Networks for Image Captioning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pp. 10968–10977.

Pang, R., Z. Zhang, X. Gao, Z. Xi, S. Ji, P. Cheng, X. Luo, and T. Wang (2022). Trojanzoo: Towards unified, holistic, and practical evaluation of neural backdoors. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pp. 684–702. IEEE.

Pang, T., C. Du, Y. Dong, and J. Zhu (2018). Towards robust detection of adversarial examples. *Advances in neural information processing systems 31*.

Papernot, N. and P. McDaniel (2016). On the effectiveness of defensive distillation. *arXiv preprint arXiv:1607.05113*.

Papernot, N., P. McDaniel, X. Wu, S. Jha, and A. Swami (2016). Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE symposium on security and privacy (SP)*, pp. 582–597. IEEE.

Park, S., M. Son, S. Jang, Y. C. Ahn, J.-Y. Kim, and N. Kang (2023). Temporal interpolation is all you need for dynamic neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4212–4221.

Park, W., D. Kim, Y. Lu, and M. Cho (2019). Relational knowledge distillation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 3967–3976.

Passalis, N., J. Raitoharju, A. Tefas, and M. Gabbouj (2020). Efficient adaptive inference for deep convolutional neural networks using hierarchical early exits. *Pattern Recognition 105*, 107346.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems 32*.

Patterson, D., J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean (2021). Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*.

Paudice, A., L. Muñoz-González, A. Gyorgy, and E. C. Lupu (2018). Detection of adversarial training examples in poisoning attacks through anomaly detection. *arXiv preprint arXiv:1802.03041*.

Pei, K., Y. Cao, J. Yang, and S. Jana (2017). Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017*, pp. 1–18. ACM.

Peng, Y., S. Yan, and Z. Lu (2019). Transfer learning in biomedical natural language processing: an evaluation of bert and elmo on ten benchmarking datasets. *arXiv preprint arXiv:1906.05474*.

Pensia, A., S. Rajput, A. Nagle, H. Vishwakarma, and D. Papailiopoulos (2020). Optimal lottery tickets via subset sum: Logarithmic over-parameterization is sufficient. *Advances in neural information processing systems 33*, 2599–2610.

Peri, N., N. Gupta, W. R. Huang, L. Fowl, C. Zhu, S. Feizi, T. Goldstein, and J. P. Dickerson (2020). Deep k-nn defense against clean-label data poisoning attacks. In *European Conference on Computer Vision*, pp. 55–70. Springer.

Peste, A., A. Vladu, E. Kurtic, C. H. Lampert, and D. Alistarh (2022). Cram: A compression-aware minimizer. *arXiv preprint arXiv:2207.14200*.

Petsios, T., J. Zhao, A. D. Keromytis, and S. Jana (2017). Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, pp. 2155–2168. ACM.

Pitman, J. (2021). Google translate: One billion installs, one billion stories.

Qiu, X., T. Parcollet, J. Fernandez-Marques, P. P. Gusmao, Y. Gao, D. J. Beutel, T. Topal, A. Mathur, and N. D. Lane (2023). A first look into the carbon footprint of federated learning. *Journal of Machine Learning Research 24*(129), 1–23.

Radford, A., K. Narasimhan, T. Salimans, and I. Sutskever (2018). Improving language understanding by generative pre-training. In *empty*.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Rajput, S., N. Mehta, A. Singh, R. Hulikal Keshavan, T. Vu, L. Heldt, L. Hong, Y. Tay, V. Tran, J. Samost, et al. (2024). Recommender systems with generative retrieval. *Advances in Neural Information Processing Systems 36*.

Raposo, D., S. Ritter, B. Richards, T. Lillicrap, P. C. Humphreys, and A. Santoro (2024). Mixture-of-depths: Dynamically allocating compute in transformer-based language models. *arXiv preprint arXiv:2404.02258*.

Ren, S., Y. Deng, K. He, and W. Che (2019, July). Generating natural language adversarial examples through probability weighted word saliency. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy, pp. 1085–1097. Association for Computational Linguistics.

Rice, J. A. (2006). Mathematical statistics and data analysis. *arXiv preprint arXiv:1503.02531*.

Rotem, N., J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, et al. (2018). Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*.

Russo, E., M. Palesi, S. Monteleone, D. Patti, A. Mineo, G. Ascia, and V. Catania (2021). Dnn model compression for iot domain-specific hardware accelerators. *IEEE Internet of Things Journal 9*(9), 6650–6662.

Sabne, A. (2020). Xla: Compiling machine learning for peak performance. *Google Res*.

Saha, A., A. Subramanya, and H. Pirsiavash (2020). Hidden trigger backdoor attacks. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 34, pp. 11957–11965.

Saha, A., A. Tejankar, S. A. Koohpayegani, and H. Pirsiavash (2022). Backdoor attacks on self-supervised learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13337–13346.

Savazzi, S., V. Rampa, S. Kianoush, and M. Bennis (2022). An energy and carbon footprint analysis of distributed and federated learning. *IEEE Transactions on Green Communications and Networking 7*(1), 248–264.

Schuster, T., A. Fisch, J. Gupta, M. Dehghani, D. Bahri, V. Tran, Y. Tay, and D. Metzler (2022). Confident adaptive language modeling. *Advances in Neural Information Processing Systems 35*, 17456–17472.

Sehwag, V., S. Wang, P. Mittal, and S. Jana (2020). Hydra: Pruning adversarially robust neural networks. *Advances in Neural Information Processing Systems 33*, 19655–19666.

Sekikawa, Y. and S. Yashima (2023). Bit-pruning: A sparse multiplication-less dot-product. In *The Eleventh International Conference on Learning Representations*.

Serra, T., X. Yu, A. Kumar, and S. Ramalingam (2021). Scaling up exact neural network compression by relu stability. *Advances in neural information processing systems 34*, 27081–27093.

Shazeer, N., A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017*. OpenReview.net.

Shen, H., J. Roesch, Z. Chen, W. Chen, Y. Wu, M. Li, V. Sharma, Z. Tatlock, and Y. Wang (2021). Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems 3*, 208–222.

Shervashidze, N., P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt (2011). Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research 12*(9).

Shi, Y., Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou (2023, July). Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA, pp. 701–718. USENIX Association.

Shin, E. C. R., D. Song, and R. Moazzezi (2015). Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pp. 611–626. USENIX Association.

Shoeybi, M., M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro (2020). Megatron-lm: Training multi-billion parameter language models using model parallelism.

Shojaee Rad, Z. and M. Ghobaei-Arani (2024). Data pipeline approaches in serverless computing: a taxonomy, review, and research trends. *Journal of Big Data 11*(1), 1–42.

Shokri, R. et al. (2020). Bypassing backdoor detection algorithms in deep learning. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 175–183. IEEE.

Singh, A., N. Thakur, and A. Sharma (2016). A review of supervised machine learning algorithms. In *2016 3rd international conference on computing for sustainable global development (INDIACom)*, pp. 1310–1315. Ieee.

Singh, G., T. Gehr, M. Mirman, M. Püschel, and M. Vechev (2018). Fast and effective robustness certification. *Advances in neural information processing systems 31*.

Singh, G., T. Gehr, M. Püschel, and M. Vechev (2019). Boosting robustness certification of neural networks. In *International conference on learning representations*.

Skarding, J., B. Gabrys, and K. Musial (2021). Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey. *iEEE Access 9*, 79143–79168.

Smith, G. H., A. Liu, S. Lyubomirsky, S. Davidson, J. McMahan, M. B. Taylor, L. Ceze, and Z. Tatlock (2021). Pure tensor program rewriting via access patterns (representation pearl). In R. Samanta and I. Dillig (Eds.), *MAPS@PLDI 2021: Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, Virtual Event, Canada, 21 June, 2021*, pp. 21–31. ACM.

Sponner, M., B. Waschneck, and A. Kumar (2024). Adapting neural networks at runtime: Current trends in at-runtime optimizations for deep learning. *ACM Computing Surveys 56*(10), 1–40.

Springenberg, J. T., A. Dosovitskiy, T. Brox, and M. A. Riedmiller (2015). Striving for simplicity: The all convolutional net. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*.

Suciu, O., R. Marginean, Y. Kaya, H. Daume III, and T. Dumitras (2018). When does machine learning {FAIL}? generalized transferability for evasion and poisoning attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1299–1316.

Sun, S., W. Ren, J. Li, R. Wang, and X. Cao (2024). Logit standardization in knowledge distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15731–15740.

Sun, Y., Y. Ming, X. Zhu, and Y. Li (2022). Out-of-distribution detection with deep nearest neighbors. In *International Conference on Machine Learning*, pp. 20827–20840. PMLR.

Sun, Z., J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang (2020). Automatic testing and improvement of machine translation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 974–985.

Sutskever, I., O. Vinyals, and Q. V. Le (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems 27*.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Szegedy, C., W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus (2014). Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014*. Openreview.net.

Tang, J., R. Shivanna, Z. Zhao, D. Lin, A. Singh, E. H. Chi, and S. Jain (2020). Understanding and improving knowledge distillation. *arXiv preprint arXiv:2002.03532*.

Tao, A., Y. Duan, Y. Wang, J. Lu, and J. Zhou (2024). Dynamics-aware adversarial attack of adaptive neural networks. *IEEE Transactions on Circuits and Systems for Video Technology*.

Teerapittayanon, S., B. McDanel, and H.-T. Kung (2016). Branchynet: Fast Inference via Early Exiting from Deep Neural Networks. In *Proceedings of the International Conference on Pattern Recognition*, pp. 2464–2469.

Tian, S., G. Yang, and Y. Cai (2018). Detecting adversarial examples through image transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Volume 32.

Tian, Y., K. Pei, S. Jana, and B. Ray (2018). Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pp. 303–314.

Tiwari, R., U. Bamba, A. Chavan, and D. K. Gupta (2021). Chipnet: Budget-aware pruning with heaviside continuous approximations. *arXiv preprint arXiv:2102.07156*.

Traini, L., V. Cortellessa, D. Di Pompeo, and M. Tucci (2023). Towards effective assessment of steady state performance in java software: Are we there yet? *Empirical Software Engineering 28*(1), 13.

Tramer, F. (2022). Detecting adversarial examples is (nearly) as hard as classifying them. In *International Conference on Machine Learning*, pp. 21692–21702. PMLR.

Tramèr, F., A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel (2017). Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*.

Tramèr, F., F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart (2016). Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pp. 601–618.

Tukan, M., L. Mualem, and A. Maalouf (2022). Pruning neural networks via coresets and convex geometry: Towards no assumptions. *Advances in Neural Information Processing Systems 35*, 38003–38019.

Tung, F. and G. Mori (2019). Similarity-preserving knowledge distillation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1365–1374.

Turovsky, B. (2016). Ten years of google translate.

Van Vliet, H., H. Van Vliet, and J. Van Vliet (2008). *Software engineering: principles and practice*, Volume 13. John Wiley & Sons Hoboken, NJ.

Vanholder, H. (2016). Efficient inference with tensorrt. In *GPU Technology Conference*, Volume 1, pp. 2.

Varma, K., A. Numanoğlu, Y. Kaya, and T. Dumitraş (2024). Understanding, uncovering, and mitigating the causes of inference slowdown for language models. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pp. 723–740. IEEE.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017a). Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin (2017b). Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008.

Veit, A. and S. J. Belongie (2018). Convolutional networks with adaptive inference graphs. In *Proceedings of the European conference on computer vision (ECCV 2018)*, pp. 3–18. Springer.

Vinyals, O., A. Toshev, S. Bengio, and D. Erhan (2015). Show and Tell: A Neural Image Caption Generator. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pp. 3156–3164.

Vischer, M. A., R. T. Lange, and H. Sprekeler (2021). On lottery tickets and minimal task representations in deep reinforcement learning. *arXiv preprint arXiv:2105.01648*.

Wan, C., M. Santriaji, E. Rogers, H. Hoffmann, M. Maire, and S. Lu (2020). {ALERT}: Accurate learning for energy and timeliness. In *2020 {USENIX} Annual Technical Conference (USENIX ATC 20)*, pp. 353–369. USENIX Association.

Wang, A., A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman (2018). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wang, B., Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao (2019). Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE symposium on security and privacy (SP)*, pp. 707–723. IEEE.

Wang, H., J. Xu, C. Xu, X. Ma, and J. Lu (2020a). Dissector: Input validation for deep learning applications by crossing-layer dissection. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 727–738. IEEE.

Wang, H., J. Xu, C. Xu, X. Ma, and J. Lu (2020b). Dissector: Input validation for deep learning applications by crossing-layer dissection. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE 2020)*, pp. 727–738. IEEE.

Wang, J., G. Dong, J. Sun, X. Wang, and P. Zhang (2019). Adversarial sample detection for deep neural network through model mutation testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1245–1256. IEEE.

Wang, J., J. Tang, and J. Luo (2020). Multimodal Attention with Image Text Spatial Relationship for OCR-Based Image Captioning. In *MM '20: The 28th ACM International Conference on Multimedia, Virtual Event / Seattle, WA, USA, October 12-16, 2020*, pp. 4337–4345.

Wang, L. and K.-J. Yoon (2021). Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks. *IEEE transactions on pattern analysis and machine intelligence 44*(6), 3048–3068.

Wang, M., S. Ding, T. Cao, Y. Liu, and F. Xu (2021). Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (ACM MobiCom 2021)*, pp. 215–228. ACM.

Wang, T., Y. Yao, F. Xu, S. An, H. Tong, and T. Wang (2022). An invisible black-box backdoor attack through frequency domain. In *European Conference on Computer Vision*, pp. 396–413. Springer.

Wang, W., M. Chen, S. Zhao, L. Chen, J. Hu, H. Liu, D. Cai, X. He, and W. Liu (2021). Accelerate cnns from three dimensions: A comprehensive pruning framework. In *International Conference on Machine Learning*, pp. 10717–10726. PMLR.

Wang, X., J. Liu, T. Liu, Y. Luo, Y. Du, and D. Tao (2022). Symmetric pruning in quantum neural networks. *arXiv preprint arXiv:2208.14057*.

Wang, X., Y. Xiong, Y. Wei, M. Wang, and L. Li (2021). Lightseq: A high performance inference library for transformers.

Wang, X., F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez (2018). Skipnet: Learning Dynamic Routing in Convolutional Networks. In *Proceedings of the European Conference on Computer Vision*, pp. 409–424.

Wang, Y., R. Huang, S. Song, Z. Huang, and G. Huang (2021). Not all images are worth 16x16 words: Dynamic transformers for efficient image recognition. *Advances in neural information processing systems 34*, 11960–11973.

Wang, Y., D. Li, and R. Sun (2023). Ntk-sap: Improving neural network pruning by aligning training dynamics. *arXiv preprint arXiv:2304.02840*.

Wang, Z. (2021). Data-free knowledge distillation with soft targeted transfer set synthesis. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 35, pp. 10245–10253.

Wenger, E., J. Passananti, A. N. Bhagoji, Y. Yao, H. Zheng, and B. Y. Zhao (2021). Backdoor attacks against deep learning systems in the physical world. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 6206–6215.

Weyuker, E. J. and F. I. Vokolos (2000). Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering 26*(12), 1147–1156.

Wolf, T., L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush (2020, October). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Online, pp. 38–45. Association for Computational Linguistics.

Wong, E., L. Rice, and J. Z. Kolter (2020). Fast is better than free: Revisiting adversarial training. *arXiv preprint arXiv:2001.03994*.

Wu, K., B. Hu, and Q. Ju (2021, November). TenTrans high-performance inference toolkit for WMT2021 efficiency task. In L. Barrault, O. Bojar, F. Bougares, R. Chatterjee, M. R. Costa-jussa, C. Federmann, M. Fishel, A. Fraser, M. Freitag, Y. Graham, R. Grundkiewicz, P. Guzman, B. Haddow, M. Huck, A. J. Yepes, P. Koehn, T. Kocmi, A. Martins, M. Morishita, and C. Monz (Eds.), *Proceedings of the Sixth Conference on Machine Translation*, Online, pp. 795–798. Association for Computational Linguistics.

Wu, R., T. Kim, D. J. Tian, A. Bianchi, and D. Xu (2022). {DnD}: A {Cross-Architecture} deep neural network decompiler. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2135–2152.

Wu, S., F. Sun, W. Zhang, X. Xie, and B. Cui (2022). Graph neural networks in recommender systems: a survey. *ACM Computing Surveys 55*(5), 1–37.

Wu, Z., T. Nagarajan, A. Kumar, S. Rennie, L. S. Davis, K. Grauman, and R. Feris (2018). Blockdrop: Dynamic inference paths in residual networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018*, pp. 8817–8826. Computer Vision Foundation / IEEE Computer Society.

Wu, Z., C. Xiong, C. Ma, R. Socher, and L. S. Davis (2019). Adaframe: Adaptive frame selection for fast video recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019*, pp. 1278–1287. Computer Vision Foundation / IEEE.

Xia, J., T. Wang, J. Ding, X. Wei, and M. Chen (2022). Eliminating backdoor triggers for deep neural networks using attention relation graph distillation. *arXiv preprint arXiv:2204.09975*.

Xiao, G., J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han (2024). Smoothquant: Accurate and efficient post-training quantization for large language models.

Xiao, X., S. Han, D. Zhang, and T. Xie (2013). Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *International Symposium on Software Testing and Analysis, ISSTA 2013*, pp. 90–100. ACM.

Xie, C., M. Tan, B. Gong, A. Yuille, and Q. V. Le (2020). Smooth adversarial training. *arXiv preprint arXiv:2006.14536*.

Xin, J., R. Tang, J. Lee, Y. Yu, and J. Lin (2020). Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*.

Xiong, P., C. Pu, X. Zhu, and R. Griffith (2013). vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pp. 271–282.

Xu, K., J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio (2015). Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, Volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 2048–2057. JMLR.org.

Xu, X., M. Li, C. Tao, T. Shen, R. Cheng, J. Li, C. Xu, D. Tao, and T. Zhou (2024). A survey on knowledge distillation of large language models. *arXiv preprint arXiv:2402.13116*.

Yang, J., B. Martinez, A. Bulat, G. Tzimiropoulos, et al. (2021). Knowledge distillation via softmax regression representation learning. International Conference on Learning Representations (ICLR).

Yang, L., Y. Han, X. Chen, S. Song, J. Dai, and G. Huang (2020). Resolution adaptive networks for efficient inference. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2369–2378.

Yang, Q., T. Yang, M. Xiang, L. Zhang, H. Wang, M. Serafini, and H. Guan (2024). Gmorph: Accelerating multi-dnn inference via model fusion. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 505–523.

Yang, T.-J., Y.-H. Chen, and V. Sze (2017). Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5687–5695.

Yao, Y., H. Li, H. Zheng, and B. Y. Zhao (2019). Latent backdoor attacks on deep neural networks. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 2041–2055.

Yao, Y., L. Rosasco, A. C. C. Approximation, and undefined 2007 (2007). On early stopping in gradient descent learning. *Constructive Approximation 26*, 289–315.

Yao, Z., R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He (2022). Zeroquant: Efficient and affordable post-training quantization for large-scale transformers.

Yao, Z., S. Cao, W. Xiao, C. Zhang, and L. Nie (2019). Balanced sparsity for efficient dnn inference on gpu. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 33, pp. 5676–5683.

Yao, Z., C. Li, X. Wu, S. Youn, and Y. He (2023). A comprehensive study on post-training quantization for large language models. *ArXiv abs/2303.08302*.

Yao, Z., H. Zhang, Y. Guo, X. Tian, W. Peng, Y. Zou, L. Y. Zhang, and C. Chen (2024). Reverse backdoor distillation: Towards online backdoor attack detection for deep neural network models. *IEEE Transactions on Dependable and Secure Computing*.

Ye, H., X. Zhang, Z. Huang, G. Chen, and D. Chen (2020). Hybriddnn: A framework for high-performance hybrid dnn accelerator design and implementation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE.

Ye, S., K. Xu, S. Liu, H. Cheng, J.-H. Lambrechts, H. Zhang, A. Zhou, K. Ma, Y. Wang, and X. Lin (2019). Adversarial robustness vs. model compression, or both? In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 111–120.

Ye, Z., R. Lai, J. Shao, T. Chen, and L. Ceze (2023). Sparsetir: Composable abstractions for sparse compilation in deep learning. In T. M. Aamodt, N. D. E. Jerger, and M. M. Swift (Eds.), *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pp. 660–678. ACM.

Yin, H., A. Vahdat, J. M. Alvarez, A. Mallya, J. Kautz, and P. Molchanov (2022). A-vit: Adaptive tokens for efficient vision transformer. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10809–10818.

Yoshida, K. and T. Fujino (2020). Disabling backdoor and identifying poison data by using knowledge distillation in backdoor attacks on deep neural networks. In *Proceedings of the 13th ACM workshop on artificial intelligence and security*, pp. 117–127.

Yu, F., D. Wang, L. Shangguan, M. Zhang, X. Tang, C. Liu, and X. Chen (2021). A survey of large-scale deep learning serving system optimization: Challenges and opportunities. *CoRR abs/2111.14247*.

Yu, J., A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke (2017). Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News 45*(2), 548–560.

Yu, L., V. O. Yazici, X. Liu, J. v. d. Weijer, Y. Cheng, and A. Ramisa (2019). Learning metrics from teachers: Compact networks for image embedding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2907–2916.

Yu, Y., Y. Wang, W. Yang, S. Lu, Y.-P. Tan, and A. C. Kot (2023). Backdoor attacks against deep image compression via adaptive frequency trigger. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12250–12259.

Yuan, H., Y. Shi, N. Xu, X. Yang, X. Geng, and Y. Rui (2024). Learning from biased soft labels. *Advances in Neural Information Processing Systems 36*.

Yuan, L., F. E. Tay, G. Li, T. Wang, and J. Feng (2020). Revisiting knowledge distillation via label smoothing regularization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 3903–3911.

Yuan, Z., L. Niu, J. Liu, W. Liu, X. Wang, Y. Shang, G. Sun, Q. Wu, J. Wu, and B. Wu (2023). Rptq: Reorder-based post-training quantization for large language models.

Yuan, Z., J. Zhang, Y. Jia, C. Tan, T. Xue, and S. Shan (2021). Meta gradient adversarial attack. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7748–7757.

Yurtsever, E., J. Lambert, A. Carballo, and K. Takeda (2020). A survey of autonomous driving: Common practices and emerging technologies. *IEEE access 8*, 58443–58469.

Zaken, E. B., S. Ravfogel, and Y. Goldberg (2021). Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv preprint arXiv:2106.10199*.

Zang, Y., F. Qi, C. Yang, Z. Liu, M. Zhang, Q. Liu, and M. Sun (2020, July). Word-level textual adversarial attacking as combinatorial optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online, pp. 6066–6080. Association for Computational Linguistics.

Zeng, D., N. Du, T. Wang, Y. Xu, T. Lei, Z. Chen, and C. Cui (2023). Learning to skip for language modeling. *arXiv preprint arXiv:2311.15436*.

Zeng, Y., W. Park, Z. M. Mao, and R. Jia (2021). Rethinking the backdoor attacks' triggers: A frequency perspective. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 16473–16481.

Zeng, Y., Z. Shi, M. Jin, F. Kang, L. Lyu, C.-J. Hsieh, and R. Jia (2023). Towards robustness certification against universal perturbations. In *International Conference on Learning Representation*. ICLR.

Zhai, Y., C. Jiang, L. Wang, X. Jia, S. Zhang, Z. Chen, X. Liu, and Y. Zhu (2023). Bytetransformer: A high-performance transformer boosted for variable-length inputs.

Zhang, B., Y. Shi, Y. Han, Q. Hu, and W. Ding (2024). Cascade & allocate: A cross-structure adversarial attack against models fusing vision and language. *Information Fusion 104*, 102179.

Zhang, H., T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel (2018). Efficient neural network robustness certification with general activation functions. *Advances in neural information processing systems 31*.

Zhang, J., B. Li, J. Xu, S. Wu, S. Ding, L. Zhang, and C. Wu (2022). Towards efficient data free black-box adversarial attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15115–15125.

Zhang, J., M. Tan, P. Dai, and W. Zhu (2023). Leco: improving early exiting via learned exits and comparison-based exiting mechanism. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pp. 298–309.

Zhang, M., Y. Zhang, L. Zhang, C. Liu, and S. Khurshid (2018). Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*, pp. 132–142. ACM.

Zhang, X., J. Zhang, Z. Chen, and K. He (2021, August). Crafting adversarial examples for neural machine translation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Online, pp. 1967–1977. Association for Computational Linguistics.

Zhang, X., X. Zhou, M. Lin, and J. Sun (2018). Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 6848–6856.

Zhang, Z., M. C. Y. Cho, C. Wang, C. Hsu, C. K. Chen, and S. Shieh (2014). Iot security: Ongoing challenges and research opportunities. In *7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014*, pp. 230–234. IEEE Computer Society.

Zhao, B., Q. Cui, R. Song, Y. Qiu, and J. Liang (2022). Decoupled knowledge distillation. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pp. 11953–11962.

Zhao, Q. and C. Wressnegger (2023). Holistic adversarially robust pruning. In *The Eleventh International Conference on Learning Representations*.

Zhao, T., X. S. Zhang, W. Zhu, J. Wang, S. Yang, J. Liu, and J. Cheng (2022). Multi-granularity pruning for model acceleration on mobile devices. In *European Conference on Computer Vision*, pp. 484–501. Springer.

Zheng, B., Z. Jiang, C. H. Yu, H. Shen, J. Fromm, Y. Liu, Y. Wang, L. Ceze, T. Chen, and G. Pekhimenko (2022). Dietcode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems 4*, 848–863.

Zheng, L., C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al. (2020). Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp. 863–879.

Zheng, S., R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. Liang (2022). AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In V. Salapura, M. Zahran, F. Chong, and L. Tang (Eds.), *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pp. 874–887. ACM.

Zheng, Y., C. Fan, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, and Y. Chen (2019). Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pp. 772–784. IEEE.

Zhong, Y., M. Zhu, and H. Peng (2021). Vin: Voxel-based implicit network for joint 3d object detection and segmentation for lidars. *unknow*.

Zhou, H., W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu (2020). Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE 2020)*, pp. 347–358. IEEE.

Zhou, H., L. Song, J. Chen, Y. Zhou, G. Wang, J. Yuan, and Q. Zhang (2021). Rethinking soft labels for knowledge distillation: A bias-variance tradeoff perspective. *arXiv preprint arXiv:2102.00650*.

Zhou, X., Z. Lin, X. Shan, Y. Wang, D. Sun, and M.-H. Yang (2024). Drivinggaussian: Composite gaussian splatting for surrounding dynamic autonomous driving scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21634–21643.

Zhou, Y., Y. Yu, and B. Ding (2020). Towards mlops: A case study of ml pipeline platform. In *2020 International conference on artificial intelligence and computer engineering (ICAICE)*, pp. 494–500. IEEE.

Zhu, H., R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui, et al. (2022). {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 233–248.

Zhu, M., M. Ghaffari, W. A. Clark, and H. Peng (2022). E^2pn: Efficient se(3)-equivariant point network.

Zhu, W. (2021). Leebert: Learned early exit for bert with cross-level optimization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 2968–2980.

Zhu, Y., Y. Cheng, H. Zhou, and Y. Lu (2021). Hermes attack: Steal {DNN} models with lossless inference accuracy. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

Zolfaghari, M., K. Singh, and T. Brox (2018). Eco: Efficient convolutional network for online video understanding. In *Proceedings of the European conference on computer vision (ECCV 2018)*, pp. 695–712. Springer.

Zou, W., S. Huang, J. Xie, X. Dai, and J. Chen (2020). A reinforced generation of adversarial examples for neural machine translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 3486–3497.

Zügner, D., A. Akbarnejad, and S. Günnemann (2018). Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018*, pp. 2847–2856. ACM.

BIOGRAPHICAL SKETCH

Simin Chen is currently a PhD Candidate at The University of Texas at Dallas, working under the guidance of Dr. Wei Yang and Dr. Cong Liu.

Before pursuing his doctoral studies, Simin earned a Master of Science degree from Tongji University in Shanghai, China. Simin also holds a Bachelor of Science degree from Tongji University.

Simin's academic journey reflects a strong commitment to advancing the field of machine learning through rigorous research and innovative tool development. Simin aims to make ML systems not only more powerful but also more secure and interpretable, paving the way for future advancements in the field.

CURRICULUM VITAE

Simin Chen

May 15, 2024

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: sxc180080@utdallas.edu

Educational History:

B.S., Civil Engineering, Tongji University, 2015
M.S., Civil Engineering, Tongji University, 2018
PhD, Computer Science, The University of Texas at Dallas, 2024

Understanding and Improving the Efficiency of Machine Learning Software: Model, Data, and Program-Level Safeguards
PhD Dissertation
Computer Science Department, The University of Texas at Dallas
Advisors: Dr. Wei Yang and Dr. Cong Liu

Employment History:

Research Intern, Amazon Web Service, May 2023 – August 2023
Research Intern, Microsoft Research, May 2021 – July 2021
Research Intern, NEC Laboratories America, January 2020 – May 2020

Professional Recognitions and Honors:

Second prize in THUBA DAO Global Hackson for Blockchain Competition.
Travel grant award from CVPR 2022.
Travel grant award from SIGSoft ISSTA 2023.