Exploring Low Complexity Embedded Architectures for Deep Neural Networks

Soham Chatterjee

School of Electrical & Electronic Engineering

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

2021

Statement of Originality

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

16-04-21
.....
Date

Soham Chatterjee
.....
Soham Chatterjee

Supervisor Declaration Statement

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

16-04-21
.....
Date


.....
Prof. Arindam Basu

Authorship Attribution Statement

This thesis contains material from 3 papers published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as D. Singla, S. Chatterjee, L. Ramapantulu, A. Ussa, B. Ramesh, A. Basu, "HyNNA: Improved Performance for Neuromorphic Vision Sensor based Surveillance using Hybrid Neural Network Architecture," *2020 IEEE International Symposium on Circuits and Systems Conference (ISCAS)*, Sevilla, Spain 2020.

The contributions of the co-authors are as follows:

- A/Prof Arindam Basu provided the initial project direction and edited the manuscript drafts.
- Deepak Singla worked on data preprocessing and implemented the different RoI proposal schemes.
- I trained the different neural network models and wrote scripts to calculate the memory and FLOPs of the models. I also wrote the sections related to neural network architecture, training, performance and results in the manuscript.
- Dr. Lavanya Ramapantulu prepared the manuscript draft and guided me on how to calculate the model memory and FLOPs.
- Bharath Ramesh, Andres Ussa and A/Prof Arindam Basu reviewed and finalised the manuscript draft.

Chapter 4 contains material submitted for publication as S. Chatterjee, YS Won, D. Jap, S. Bhasin, A. Basu, "DeepFreeze: Cold Boot Attack and Model Recovery on Commercial EdgeML Device", **submitted to** *2021 Design Automation Conference (DAC)*, San Francisco, United States of America 2021.

The contributions of the co-authors are as follows:

- A/Prof Arindam Basu and Dr. Shivam Bhasin provided the initial project direction and edited the manuscript drafts.
- I performed the recovery of model architecture and weights from the RAM dump as well performed Knowledge Distillation training to perform model accuracy and weight recovery. I also prepared parts of the manuscript related to the same.
- Dr. Yoo-Seung Won performed the Cold Boot Attack and RAM dump as well as prepared parts of the manuscript related to the same.

- Dr. Dirmanto Jap prepared and reviewed the manuscript.

Chapter 5 partly contains material published as YS Won, S. Chatterjee, D. Jap, S. Bhasin, A. Basu, "Time to Leak: Cross-Device Timing Attack On Edge Deep Learning Accelerator," *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, Jeju Shinhwa World, Republic of Korea 2021.

The contributions of the co-authors are as follows:

- Dr. Shivam Bhasin provided the initial project direction, co-designed the study and reviewed the manuscript
- A/Prof. Arindam Basu reviewed the manuscript.
- I wrote the scripts to run the experiment on different hardware. I also wrote parts of the manuscript related to OpenVINO and Neural Compute Stick 2.
- Dr. Yoo-Seung Won performed the experiments on different hardware and did the analysis of the results. He also prepared the manuscript draft.
- Dr. Dirmanto Jap prepared and reviewed the manuscript.

16-04-21 ..... Date

Soham Chatterjee ..... Soham Chatterjee

Acknowledgements

I wish to express my greatest gratitude to my advisor Prof. Arindam Basu for guiding me during my M.Eng. journey. I would also like to thank Prof. Shivam Bhasin for his mentorship and guidance. Further, this work would not have had been possible without the help and guidance I received from my peers at VIRTUS Lab: Deepak Singla, Lavanya Ramapantulu, Vivek Mohan and Charles Zhang Lei and at PACE Lab: Yoo-Seung Won and Dirmanto Jap.

I would also like to thank my parents (Biswajit and Bul Bul), my brother (Rohan) for all the support they have provided during the duration of my course.

Soham Chatterjee, April 2021

"Ever Tried. Ever Failed. No Matter.
Try Again. Fail Again. Fail Better."

— Samuel Beckett

Abstract

Deep neural networks have shown significant improvements in computer vision applications over the last few years. Performance improvements have been brought about mostly by using pre-trained models like Inception-v4, ResNet-152, and VGG 19. However, these improvements have been accompanied by an increase in the size and computational complexity of the models [1]. This makes it difficult to deploy such models in energy-constrained mobile applications which have become ever crucial with the advent of the Internet of Things (IoT).

This is especially problematic in a battery-powered IoT system, where executing complex neural networks can consume a lot of energy [2]. Hence, some methods to reduce this complexity in software, like using depthwise separable convolutions [3] and quantization [4], have been proposed. Also, a very different computing paradigm of spiking neural networks (SNN) has been introduced as a method to introduce a parameterizable tradeoff between accuracy and classification energy [5], [6]. The security of such edge deployed neural networks is also a matter of concern since the IoT devices are easily accessible to hackers.

In this work, a study of the effect of using depthwise separable convolutions and Dynamic Fixed Point (DFP) weight quantization [7] on both model accuracy and complexity is done for a DNN used for classifying traffic images captured by a neuromorphic vision sensor. Initial results show that the DFP weight quantization can significantly reduce the computational complexity of neural networks with less than a 2% drop in accuracy.

Finally, the vulnerability of neural networks to side-channel [8] and cold boot attacks [9] is also being studied. To do this, trained models are deployed to edge devices like the Neural Compute Stick, EdgeTPU DevBoard, and the EdgeTPU accelerator and then attacked to retrieve the model weights, architecture and other parameters. We show that using cold boot attacks, it is possible to recover the model architecture and weights, as well as the original model accuracy. Further,

we show that with side-channel attacks, it is possible to isolate and identify the execution of individual neurons in a model. Since quantized networks have fewer and smaller weight values, they should be easier to attack. On the other hand, larger neural networks with complex architectures and dataflows should be comparatively safer from side-channel attacks.

Contents

List of Figures

List of Tables

Abbreviations

AI	**A**rtificial **I**ntelligence
CBA	**C**old **B**oot **A**ttack
CNN	**C**onvolutional **N**eural **N**etwork
DFP	**D**ynamic **F**ixed **P**oint
EBBI	**E**vent **B**ased **B**inary **I**mages
EIE	**E**fficient **I**nference **E**ngine
GT	**G**round **T**ruth
IIoT	**I**ndustrial **I**nternet of **T**hings
IoT	**I**nternet of **T**hings
NCS	**N**eural **C**ompute **S**tick
NVS	**N**euromorphic **V**ision **S**ensor
SCA	**S**ide **C**hannel **A**ttack
SCNN	**S**parse **C**onvolutional **N**eural **N**etwork
SNN	**S**piking **N**eural **N**etwork
TPU	**T**ensor **P**rocessing **U**nit

To My Parents,
Thank you for all your love and support

Chapter 1

Introduction

1.1 Motivation and Background

Over the last decade, the popularity of smart connected devices has led to an exponential growth of Internet of Things (IoT) devices. Around the same time, the availability of a vast amount of data and parallel computing using GPUs has also increased the applications of deep learning and neural networks.

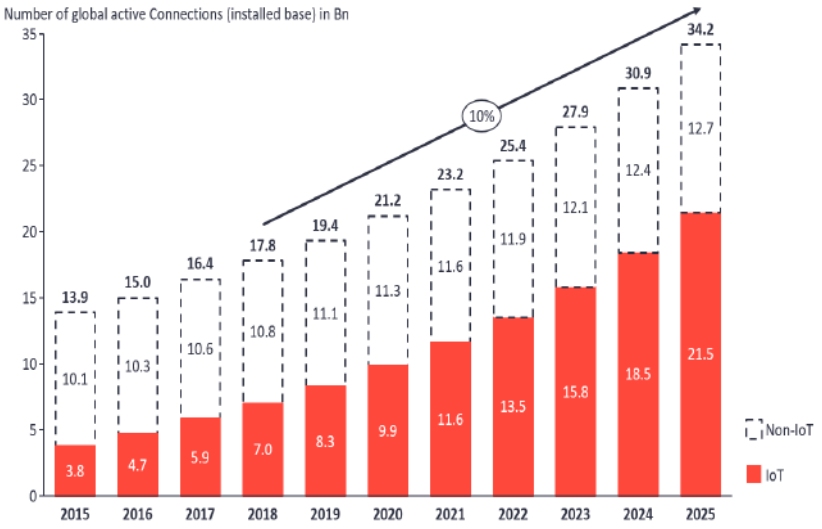


FIGURE 1.1: Growth of IoT devices connected to the internet compared to non-IoT devices. Taken from [10]

Figure 1.1 shows that the number of IoT devices is expected to more than triple by 2025 as compared to 2018. Moreover, the number of IoT devices will be approximately twice as many as the number of non-IoT devices connected to the internet.

Figure 1.2 shows that nearly $8 billion were invested in AI companies just in the second quarter of 2019. This number is also expected to rise over the next few years. More and more companies are investing heavily and trying to build products that use AI to enhance their functionality. Figure 1.3 shows that by 2030, it is projected that augmented intelligence and smart products will make up more than 50% of the total revenue generated by AI products. This shows the importance, value and ubiquity that AI and IoT will have in our lives over the next few years.

FIGURE 1.2: Funding that AI based companies have received per quarter. Taken from [11]

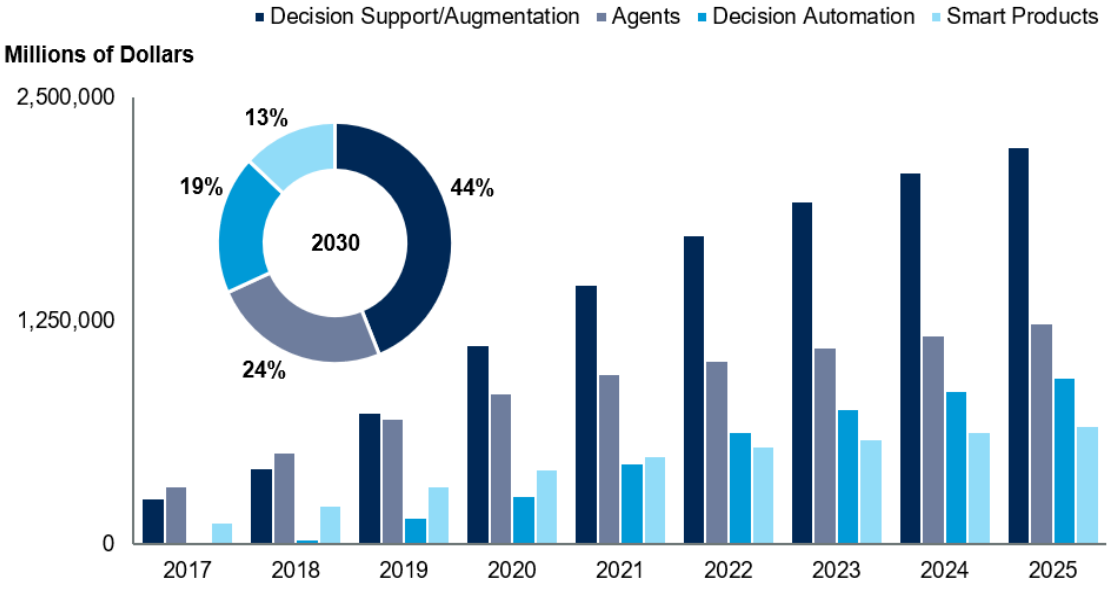


FIGURE 1.3: Business value brought about by AI based products over the next decade. Taken from [12]

While both of these fields were progressing separately, in the last few years, researchers have tried to run deep learning models on IoT devices. This computing paradigm where networks are run at the place where data is generated (IoT devices) is known as edge computing.

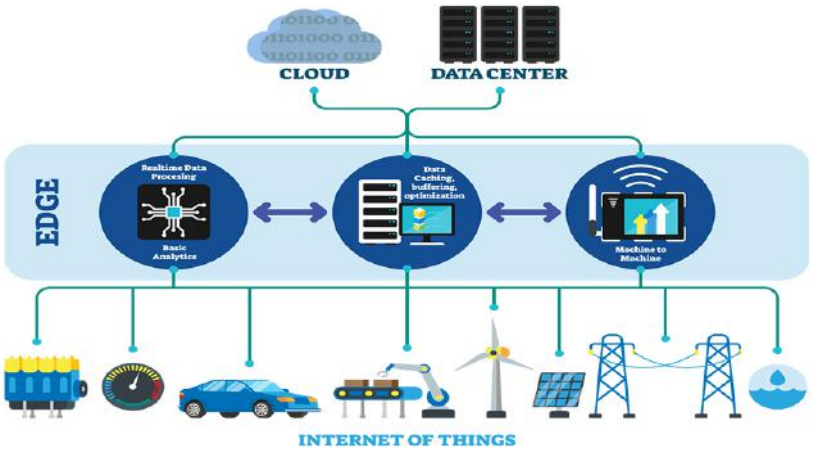


FIGURE 1.4: Edge Computing is a computing paradigm where computation is done on nodes near where the data is generated. Taken from [13]

Today, most data is still sent to cloud servers to perform the analytics tasks that many IoT products perform. However, due to the increase in IoT devices, this is not a feasible solution anymore. Moreover, many IoT devices generate a vast amount of data which is not possible to be sent to a cloud server. This is why there has been a shift to edge computing, whereby analytics and deep learning models are run at the node where the data is generated.

However, most edge devices are cheap and contain very little compute power. Due to this, it is not easy to run complex neural network architectures in such devices. This is why efforts have been made to reduce the compute requirements of neural networks and to pack more computational power into smaller devices.

One of the ways to do the former is to quantize [4] and prune neural networks [24]. While these methods do work, they are often accompanied by a drop in accuracy and research efforts need to be put in to regain the lost accuracy. Many companies like Intel and Google are also building custom hardware to run complex models on edge devices. The advantage here is that the original model can be executed without a drop in accuracy. However, many of these custom hardwares are limited to the types of architectures that can run them, and they consume a large amount of power.

Another problem with edge computing is security threats. Edge devices are usually placed in the open with minimal security features. This makes them very easy to attack. While software-based attacks are well studied, and security measures are known [25], hardware attacks and protection from such attacks on neural networks is still in its nascent stages [8].

The rise in popularity of both IoT and Deep Learning, along with the increase in the availability of data makes it an opportune time to apply machine learning models at the edge. This is why edge computing will become the broad topic of this research, and focus will be put on developing algorithms that can efficiently run on the edge as well as securing the models from side-channel attacks.

1.2 Outline

The rest of this thesis is organized as follows: In Chapter 2, an overview of the different areas being explored is given.

In chapter 3, a neural network model for classifying images captured by a neuromorphic vision sensor data is presented. Following that, a quantization method to reduce the number of computes required to run the neural network is presented.

In chapter 4, a method to extract neural network architecture and weights from a commercial neural network accelerator using a cold-boot attack is demonstrated. Further, we also show how to regain the loss in accuracy (due to erroneous weight recovery) using knowledge distillation.

In chapter 5, a way to attack neural networks using side-channel attacks is presented. We first show that it is possible to identify the kind of pre-trained neural network running on a neural compute stick with nearly 100% accuracy. Thereafter, we try to extract the model parameters of a custom binary weighted neural network architecture. The results of the attack on this network running on a neural compute stick is reported.

Finally, in chapter 6, an overview of all the results and potential future works are outlined.

Chapter 2

Literature Review

In this chapter, a review of edge computing and the techniques used to reduce model complexity is presented. This includes quantization techniques like INT8 quantization [26] and efficient neural network layers like separable depthwise convolutions [27], as well as edge computing hardware like the Neural Compute Stick and the EdgeTPU [28], [29]. This is followed by a review of retina-inspired Neuromorphic Vision Sensors (NVS). Finally, the current state-of-art on side-channel attacks and protection techniques on edge computing hardware is discussed [30].

2.1 IoT and Edge Computing

Internet of Things (IoT) is a system of interconnected devices that can transfer and receive data over a wireless connection. IoT devices generally contain a microcontroller and multiple sensors which they use to gather information and perform a task. This has made them very popular as smart home devices like digital assistants, smart lights and fitness trackers. However, the presence of sensors has also increased their application in industrial settings. This sub-branch of IoT called Industrial IoT or IIoT uses such devices to perform tasks like monitoring, predictive maintenance and surveillance.

To perform many of these tasks, IoT devices collect data from sensors, perform some predictive analysis on that data and then either take some action or send an

alert to a human in case of anomalous behaviour. Most of this analysis is done by deep learning or machine learning models in the cloud.

Edge Computing is a computing paradigm where computation is done closer to the place where data is being generated. This is different from the more common cloud computing paradigm where data is sent to a cloud server where it is processed and then sent back.

Edge computing is quickly becoming more common with the increase in IoT devices. This is primarily because IoT devices generate a lot of data. In many cases (for instance, surveillance and security cameras), it is not possible to send all data to a cloud server to perform inference. Moreover, in time-sensitive operations, or areas without an internet connection, it is not possible to send data over a network connection.

Some other advantages of edge computing include:

1. **Security**: Many industries work on sensitive or IP protected materials, devices or processes. In such cases, sending data over a network can expose them to data breaches. With edge computing, the data never leaves the device where the data is generated hence giving increased security.

2. **Power Consumption**: Many IoT devices are deployed in remote areas as battery-powered units. Such devices are expected to run on their own either by harvesting energy or with limited battery changes. Since sending data over wireless connections require much power, performing computations at the edge, on the device, will be beneficial.

3. **Network Latency and Resiliency**: Sending large amounts of data over a network has some latency associated with it. This latency can be detrimental in time-sensitive operations like a self-driving car or a predictive maintenance setting. Another issue with networks is their resiliency. A self-driving car going through a tunnel will have limited network coverage, so will devices deployed in mines and remote areas. In such cases, performing calculations at the edge will improve the reliability of the IoT device.

4. **Bandwidth**: Some IoT devices like security cameras can produce data in the order of multiple gigabytes per hour. Sending all that data to a cloud

server will be impractical and expensive. Instead, the camera feed can be processed locally and then only important information, like the presence of an intruder, can be transferred.

However, edge devices usually have a limited amount of computational power. Furthermore, since many of them are battery-powered, running complex algorithms and neural networks on them can consume much power. This is why many methods have been explored to reduce the complexity of neural networks. These include using depthwise separable convolutions [27] and quantization [4]. Furthermore, much research has been done on creating energy-efficient neural network accelerators and on using near-data processing [2]. In the next sections, these techniques will be explored.

2.2 Adapting Neural Networks for Edge Computing

In this section, some techniques to change the computational complexity of the neural network by making changes in the architecture, weights and the computation method of neural networks will be explored.

2.2.1 Depthwise Separable Convolutions

Convolutional Neural Networks (CNN) is the most widely used neural networks architectures for computer vision tasks. CNNs learn by identifying hierarchical patterns from images. These patterns are used to create a dense embedding that contains all the information of the image. This embedding can then be used by a fully connected network to make predictions from the image.

The most important layer in a CNN is the convolutional layer. Convolutional layers take feature maps of shape $C_i \times H_i \times W_i$ as input and produces feature maps of shape $C_o \times H_o \times W_o$ as output. C_i and C_o are the number of input and output depth or channels, respectively. H_i, W_i, H_o, and W_o are the input and output height and width of the feature maps. The input feature maps can be an RGB

image, or they can also be the output of another convolutional layer. By being able to stack multiple convolutional layers, CNNs can learn hierarchical features.

The number of MACs in a standard convolutional layer is given by:

$$D_k \times D_k \times C \times N \times W_o \times H_o \tag{2.1}$$

where C and N are the number of input and output channels and D_k is the dimension of the convolutional filter.

Depthwise Separable Convolutions work by separating the standard convolutional filter into a depthwise convolutional operation and a pointwise convolutional operation.

Filters in the depthwise convolution are applied only to a single channel. This means that the number of filters in a depthwise convolution will be equal to the depth of the input feature map. Furthermore, in a depthwise convolution, the output depth will not change, but the height and width will change.

Pointwise filters are applied across the depth of the image like a standard convolution, and like the name suggests, these filters have a height and width of 1. This means that the output depth after applying this filter will be 1, but the output height and width will remain the same. By applying multiple such pointwise filters on the image, we can get an output feature map with the same shape as a standard convolutional layer.

The number of MACs in a depthwise filter can be calculated by:

$$D_k \times D_k \times C \times W_o \times H_o \tag{2.2}$$

and the number of MACs in a pointwise filter is given by:

$$C \times N \times W_o \times H_o \tag{2.3}$$

This shows that even though a depthwise filter can have the same input and output feature map shapes as a standard convolution, it has fewer MACs by a factor of:

$$\frac{1}{C} + \frac{1}{D_k^2} \tag{2.4}$$

This makes depthwise separable convolutions very attractive for use in resource-constrained edge devices.

Depthwise Convolutions were first introduced in the Xception model [27]. They were most notably used in the MobileNet model where it was shown to have significantly fewer parameters and MAC operations as compared to comparable models while still maintaining accuracy [3].

2.2.2 Pruning

Trained networks usually have more weights and trainable parameters than what is required to learn a particular task. Such networks are called overparameterized, and many weights or connections in these networks can be removed with very little loss in accuracy, but a drastic drop in computational cost. This is known as network pruning.

Pruning was first proposed in [31], where weight saliency was used to calculate the importance of each weight. Modern pruning, introduced in [24] uses the magnitude of each weight as a measure of the importance of each weight . By doing so, the authors showed that AlexNet weights [32] can be pruned by 9×, corresponding to a 3× drop in MACs, with minimal drop in accuracy. Pruning also involves a finetuning step where the network is retrained after weights are pruned to increase the accuracy.

Networks can also be pruned for other metrics. For instance, in [33], the authors show that they can prune a network to reduce the energy required to run the network. Also, a very different computing paradigm of spiking neural networks (SNN) has been introduced as a method to introduce a parameterizable tradeoff between accuracy and classification energy [5], [6].

However, one problem with pruning is the introduction of sparsity in the model weights [34]. Methods have been proposed to store and compress sparse weights

so as to reduce memory access and perform sparse matrix-vector operations [34], [35], [36], [37]. Often custom hardware needs to be used to execute pruned models efficiently. Efficient Inference Engine (EIE) [36] and Sparse Convolutional Neural Network (SCNN) [37] are examples of such implementations.

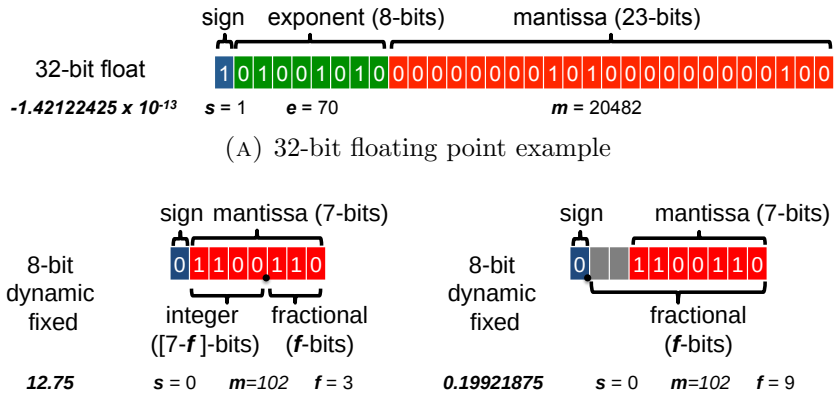
Due to the challenges in efficiently executing sparse networks, researchers have proposed methods to introduce "structured sparsity" in their models [38]. These techniques try to prune groups of weights instead of individual weights. For instance, pruning an entire neuron in a fully connected layer, or an entire filter in a convolutional layer [39] [40]. However, such pruning methods tend to lead to a more significant loss in accuracy [41]

2.2.3 Quantization

Neural Networks are initialized and trained with high precision floating point weight values. The training process involves making tiny adjustments over these values to nudge the network to a minima. After the training process, the same floating-point weight values are stored and need to be loaded to run inference.

Using low precision weights for inference instead can reduce the storage requirements as well as computational cost. This process of reducing the precision of weights is called quantization. One of the simplest forms of quantization is to map the high precision weights linearly to a low precision value like INT8. Another simple quantization technique is to use k-means clustering to map multiple weight values to a single value.

Quantization can be broadly classified into two types: Weight Quantization and, Weight and Activation Quantization. In general, weight quantization methods are concerned only with accurately representing high precision weights in low precision values. This helps to reduce the on-device storage requirements, but not the computational requirements as weights are up-converted to run inference [2]. Some exceptions are binary and ternary neural networks where the weights are converted to two and three values respectively and then used for inference [42], [43]. Finally, while low precision can be used for training networks, they are usually not done due to the sensitivity of the gradients during training [44], [45], [46].



(A) 32-bit floating point example

(B) 8-bit dynamic fixed point examples

FIGURE 2.1: Various methods of number representations. Taken from [2]

Weight and Activation quantization quantizes both the weights as well as the activations in the network. This not only helps reduce the memory requirements of the network, but it also uses the quantized values to execute the network. The most common technique for this method of quantization is INT8 quantization [26]. By using Integer only arithmetic, the authors were able to quantize ResNet-50 [47] with less than a 2% drop in accuracy.

Neural networks can also be quantized partly. In this case, only the most computationally expensive layers are quantized, or the layers which affect the accuracy the least are quantized [48]. Moreover, it is not necessary that all layers in the network be quantized to the same quantization level. For instance, in Dynamic Fixed Point (DFP) [7] quantization, different layers in the network can have different quantization levels. This technique uses the fact that different layers have different ranges of weights to quantize their weights. Moreover, activations will have a far different distribution of values than weights, and it makes sense to quantize them separately. This helps better cover the range of values in weights and activations.

In DFP quantization, numbers are quantized to the form $(-1)^s \cdot 2^{-fl} \cdot \sum_{i=0}^{B-2} 2^i \cdot x_i$ where B is the bitwidth, s is the sign bit, fl is the fractional length and x is the mantissa bits. The value of fl can be adjusted for each layer and its activations. Earlier layers can use larger values of DFP to take advantage of the increased precision, and later layers can use a smaller value of DFP to accommodate a larger range of values. Using this quantization scheme, the authors were able to quantize LeNet to 4 bits with only a 0.1% decrease in accuracy.

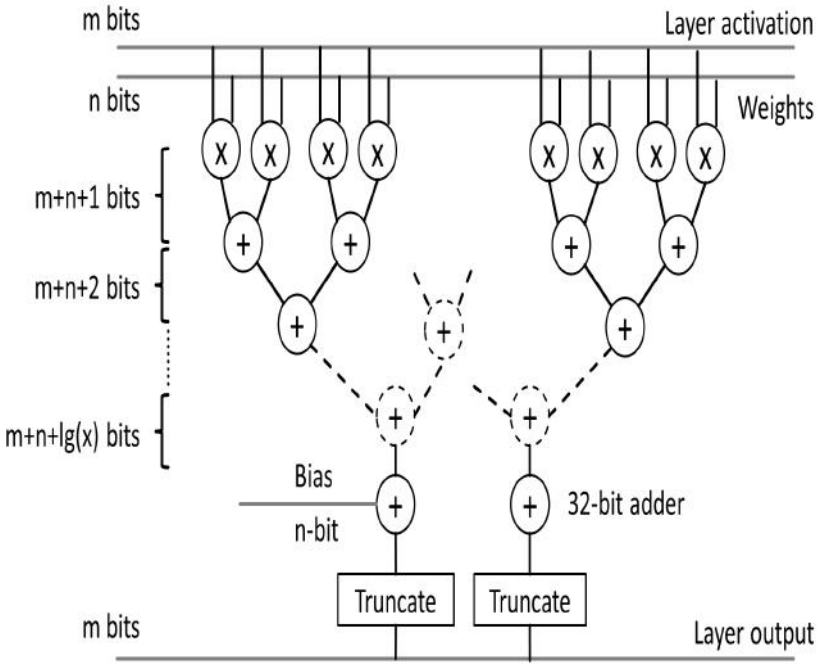


FIGURE 2.2: Data path of quantized convolutional and fully connected layers. Taken from [7]

2.2.4 Knowledge Distillation

Knowledge Distillation (KD) is a model compression technique that uses a teacher-student training paradigm to teach a smaller student model to emulate the performance of a larger teacher model. Knowledge Distillation was first introduced by in [49] and improved by Hinton et al. in [50].

To transfer knowledge to a student, the teacher model first outputs predictions for a set of training images. The input image and the softmax outputs of the teacher model are used as the input-output pairs to train the student model. The hard labels are not used since they do not contain enough information for the student to learn. However, for a well trained teacher model, even the soft outputs can have a probability distribution with the correct class with a very high probability. This is why, in addition to using soft labels, Hinton et al. also proposed using softer outputs by using a Temperature Factor T. This was called Softmax Temperature, and the probability p_i of a class i is calculated by

$$p_i = \frac{e^{\frac{z_i}{T}}}{\sum_i e^{\frac{z_i}{T}}} \tag{2.5}$$

By setting $T = 1$, the standard softmax distribution is gotten. However, with larger values of T, the output probability distribution can be made softer. By providing softer inputs, the student model can get more knowledge about which classes are similar to each other and help the student learn the teacher model's internal representation better.

Knowledge in a neural network can be represented in multiple ways. The output of a model given an input is known as the response-based knowledge of the neural network. This is the type of knowledge that was being distilled into the student using the technique being proposed by Hinton et al [50]. However, neural network can also have other types of knowledge embedded in them. For instance, deep neural networks are good at learning multiple levels of features in each layer. This is known as feature-based knowledge distillation and techniques to teach student networks this kind of knowledge has also been proposed.

Bengio et al.[51] proposed using the outputs of the final layer as well as the feature maps of intermediate layers from the teacher model to supervise the training of the student model. In FitNets [52] the authors propose using intermediate representation or 'hints' from the teacher model to train the student model. The student model learns by trying to match the feature activations of the teacher.

Further, since the student model learns from the soft outputs of the teacher, unlabelled data can also be used to train the student model. In [53], the authors use KD has been used as a method to increase the accuracy of the student network by using a trained teacher model to generate pseudo labels for a large unlabelled image dataset. This kind of semi-supervised training has been shown to improve the accuracy of the student model on ImageNet by 2%.

Finally, KD can also be combined with some of the previous model optimization techniques to further compress the student model. For instance, KD can be combined with pruning [54] [55] as well as quantization [56] [57] [58] to improve the performance of the model on edge devices.

2.3 Neural Network Accelerators

Along with deploying efficient models, efforts have also been made to create custom hardware and accelerators that can efficiently run networks on hardware. Some of these hardwares are built, keeping in mind a specific network architecture or quantization method [59], [60]. On the other hand, some hardware like the Neural Compute Stick [28] and the EdgeTPU board [29] can run any neural network architecture.

2.3.1 Neural Compute Stick

The Neural Compute Stick (NCS) is a neural network accelerator that contains a Vision Processing Unit (VPU) [61]. The device has the form factor of a pen drive and consumes a power of only 1W. The advantage of using a neural compute stick is that it can be attached to any microcontroller (running either Windows or a Linux based OS) with a USB-A type connection.

The VPU inside the NCS is called the Myriad X VPU. It contains a Neural Compute Engine that can perform 1 trillion operations per second (TOPS). It also contains 16 vector processors that can run inference parallelly on multiple data points. Combined, the vector processors and the neural compute engine can provide 4 TOPS of computing performance.

The NCS uses the OpenVINO toolkit [62] as an interface between itself and the hardware it is connected to. The model is first loaded onto the device. Once loaded, it is not cleared until the device powers off, or a new model is loaded. Data can be then sent to the device to perform inference. Moreover, data from multiple sources can be batched together and sent to the device for inference. Another advantage of using NCS is that multiple devices can be interfaced and the inference load can be shared across the multiple devices.

2.3.2 EdgeTPU Board

Another popular edge computing hardware is the EdgeTPU dev board and the EdgeTPU accelerator by Google. The dev board contains an EdgeTPU System

FIGURE 2.3: Intel Neural Compute Stick 2. Taken from [14]

on Model that contains a Quad-core Arm Cortex-A53 as well as an EdgeTPU accelerator co-processor [63]. Since the device contains multiple I/O as well as its own processor, it can be deployed as-is with a working model.

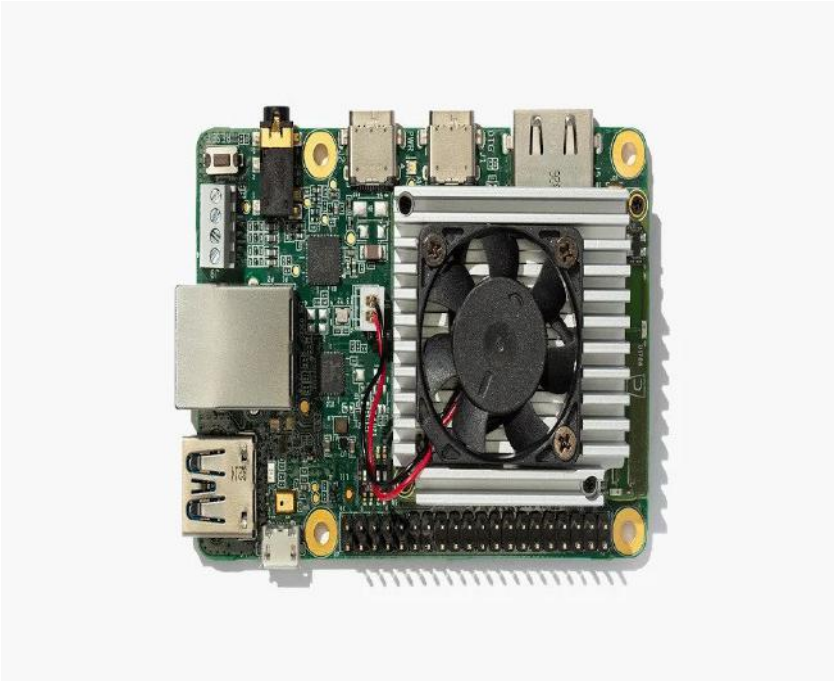


FIGURE 2.4: EdgeTPU DevBoard. Taken from [15]

The accelerator is similar to the NCS and has a USB form factor. It contains only the EdgeTPU accelerator co-processor and needs to be interfaced with a microcontroller to run neural networks [64].

FIGURE 2.5: EdgeTPU Accelerator. Taken from [14]

Both the dev board and accelerator can run TFLite models [65]. Models are first loaded on to the device, and then data can be sent to the device for inference. While TFLite is a more developed framework, it has a few disadvantages. Firstly, it cannot perform inference on batches of data, and multiple EdgeTPU boards cannot be interfaced to improve inference speeds.

2.4 Neuromorphic Vision Sensors

Along with efficient neural networks and hardware, efforts are also being made to make efficient sensors. RGB cameras are commonly used cameras in edge computing systems. The high frame rates in RGB cameras, as well as increased memory and bandwidth transmission requirements, make them have high power requirements and unsuitable for remote edge computing applications [66]. Moreover, when performing tasks like object tracking and identification, proposing RoIs involves computationally intensive steps which leads to a further increase in power

consumption [67]. Due to these reasons, an alternative to standard RBG cameras is to use Neuromorphic Vision Sensors (NVS).

NVS use a retina-inspired principle and capture changes asynchronously on a pixel level. Each pixel capture intensity changes independently hence giving NVS low data rates, high effective frame rates, high dynamic range and low power [68], [69], [70].

Furthermore, since NVS capture only dynamic events, backgrounds being static are automatically removed. This makes NVS suitable for remote surveillance using a static camera, where the background does not change relative to the objects being tracked. However, using NVS as the primary camera sensor requires the use of algorithms that can learn and predict from asynchronous events, or requires the conversion of these events into frames that can be fed into a standard neural network architecture for inference.

One of the ways to detect and track an object using NVS is to dynamically update the representation of the object as and when new events are generated [71], [72], [73]. However, these methods fail when trying to track multiple objects, and they work better for certain specific applications [73].

Another method of processing events is to aggregate events over a certain period of time and then processing all the aggregated events [74], [75], [76]. This is a much better approach since the output frames are similar to those produced by an RGB camera, and traditional classification techniques can be used. For instance, in [74], events were aggregated at intervals of 10ms and 20ms, and then clustering algorithms, as well as Kalmann filters, were applied for object detection and tracking.

To further reduce the power consumption, a low power tracking algorithm using stationary NVS was proposed in [77]. This work used a combination of frame-based as well as event-based approaches to perform tracking and classification. Aggregating events first generated frames. These were fed into a tracker to get the location of the objects. Finally, the frames were converted back to spikes for classification on IBM's TrueNorth neuromorphic chip [78].

In [17], the authors propose using a connected component labelling (CCL) based approach to track objects. The generated frames were then fed into a convolutional

neural network for classification. Finally, in [20], the authors show how to combine detector and classifier into the same network to improve the performance of the overall pipeline.

2.5 Security and Privacy of Edge Computing Hardware

An increasing concern in the deep learning field is the protection of model data in their deployed models as well as protection of the model from adversarial attacks [25]. It has been shown that private and sensitive information, are stored in the weights of trained neural networks, and can be retrieved by an attacker [79]. Moreover, just getting access to the weights and architecture of a trained neural network can motivate attackers who might not have access to the same dataset or compute power to train a network from scratch. This is an increasing concern since more companies and entities are using deep learning models for sensitive applications at the edge [80], [81]. An overview of such attacks is given in [82]. However, most of these attacks are software-based, and few works focus on the security of networks from attackers trying to extract models and gain information through vulnerabilities and leakages from hardware where models are deployed.

One of the first attacks demonstrating model extraction was done by Tramer et al [83], where the authors show that it is possible to get a model with similar performance from a black box API without access to the original dataset or model parameters. Physical access to edge computing hardware can allow attackers to exploit even more attack vectors like side-channel leakages [23] and faults [84]. Side channels like power, electromagnetic and timing can be used to recover model parameters. A review of these attacks can be found in [85].

In [86], the authors propose different levels of model extraction attacks based on the complexity and severity of the attack:

1. **Exact Extraction**: In this case, the extracted model has the same architecture, weights and other parameters as the original model. This is the best possible extraction attack, but at the same time, it is the hardest to perform especially in a limited attack setting.

2. **Functionally Equivalent Extraction**: This is a slightly weaker extraction assumption but a more likely attack. In this case, the output of both the extracted and original model needs to be the same for all inputs in a given domain. This type of extraction is hard to perform if the attack only has access to the input/output pairs from the original model.

3. **Fidelity Extraction**: In this type of extraction, the extracted model should have an output which is as similar to the output of the original model. Fidelity extraction is a slightly weaker assumption than Functional Extraction (where the similarity is 1).

4. **Task Accuracy Extraction**: This is the weakest assumption where the extracted model is expected to achieve a similar or higher accuracy as compared to the original model. This is the easiest goal since the extracted model does not need to match the mistakes of the original model.

In this section, a review of the security of edge computing hardware from the perspective of side-channel attacks and cold boot attacks will be done.

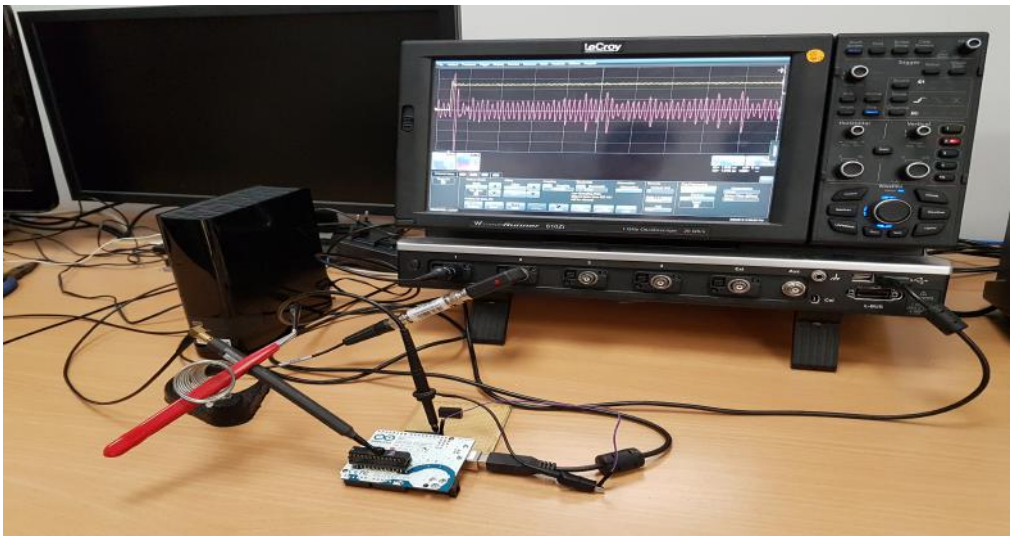
2.5.1 Electromagnetic Side-Channel Attacks

Side-channel attacks (SCA) is the use of physical leakages from hardware to extract information of the data, or computations being performed in it. To achieve this, side-channels attackers try to observe electromagnetic leakages, timing delays, power consumption and other leakages during the execution of operations inside the device.

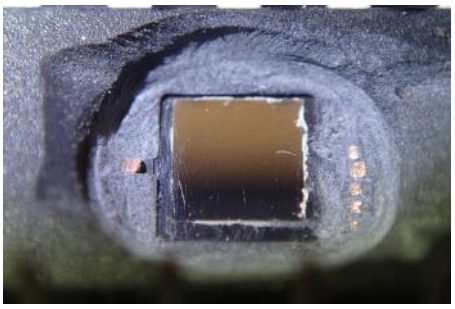
SCA was first used in cryptography applications to recover keys [87]. Using SCA, it was possible to recover only small parts of the key at a time, thus reducing the complexity of the attack. However, recent works have shown that SCA can also be used to recover the weights, operations, and architecture of neural networks just from EM leakages [16].

Before looking at how attacks are performed, it is vital to understand the different SCA attack techniques for EM leakages as well as the attack scenario and the attacker capabilities.

There are two main types of attacks performed on EM leakages:



(A) Complete Measurement Setup



(B) Target Microcontroller without covering

(C) Probe used for measuring EM leakages

FIGURE 2.6: Experimental Setup for performing side channel attack. Taken from [16]

1. **Simple Power or EM Analysis (SPA)**: This uses a few EM traces generated from the hardware to understand the computation that is happening inside the hardware. Since SPA uses only a few traces, it can be used to extract straightforward or simple operations.

2. **Differential Power or EM Analysis (DPA)**: Unlike SPA, DPA uses many measurements (sometimes in the order of millions) to apply statistical techniques to extract information. Using correlation, an attacker can find the difference between an actual measurement and hypothetical measurement. By making small variations in the input and seeing the change in the measured signatures, it is possible to get information about the operations and data. Such attacks have been successfully used in ASICs and GPUs [88], [89].

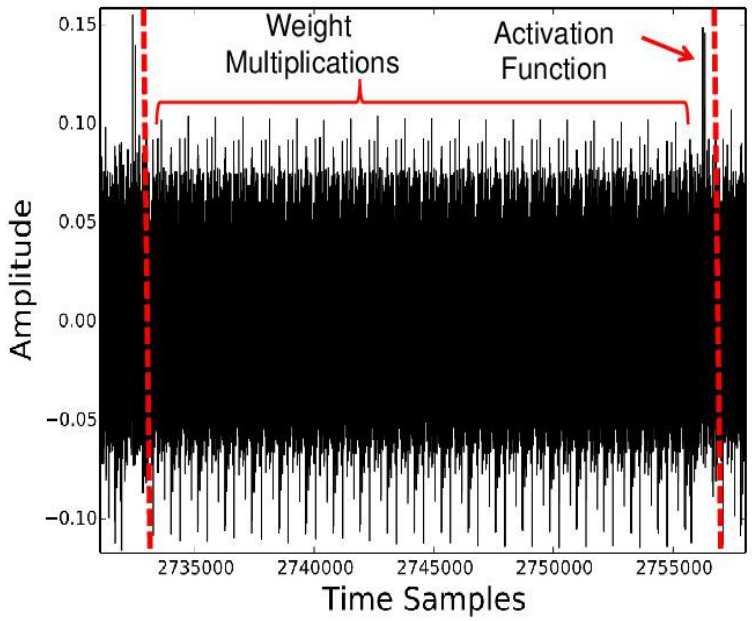


FIGURE 2.7: Traces generated from probe. Different operations generate different leakage patterns. Taken from [16]

To perform these attacks, it is assumed that the attacker has access to the hardware running the model. This is a reasonable assumption since many edge computing hardware can be bought off the shelf, and the attacker can run experiments uninhibited. Once an attacker knows how to attack a particular hardware, then they can use that knowledge to attack a product with sensitive data using the same hardware.

In such a scenario, it is assumed that the attacker does not know the model weights or its architecture; however, they are able to feed in any input to the model. Finally, the attacker should also be able to measure side-channel leakages during their attack.

Using these constraints, [16] showed that it is possible to extract information regarding the activation function, model weights and model architecture from an Atmel ATmega328P and on an ARM Cortex-M3. They used a Lecroy WaveRunner 610zi oscilloscope and an RF-U 5-2 near-field electromagnetic probe to collect the EM leakages. The attack was done on a simple Multi-Layer Perceptron model as well as a CNN model.

FIGURE 2.8: EdgeTPU Accelerator with its housing and heat sink removed. Attacks are done after removing the housing of the board. This helps to get better reading of the leakages.

However, these attacks assume that the network execution calculations are done sequentially, which makes the attack much more straightforward. In such a case, shuffling the order of execution of the different computes in a layer can help prevent (or at least slow down) these attacks [90].

Another countermeasure to weight recovery is to use masking [91] [92]. In such a case, calculations are done with random values instead of actual data to remove the dependence on actual values when performing an attack. In [8], the authors propose MaskedNet which uses masked ReLU units as well as masked adder trees for FC layers to prevent side-channel attacks. However, these masked approaches can be accompanied by an increase in the execution time of the network.

2.5.2 Timing Side-Channel Attacks

Unlike electromagnetic leakages, timing side-channels utilizes timing leakages from hardware to infer information about the type of operations and parameters inside a neural network.

One of the first works showing the use of timing side-channels was by Hua et al.[93]. They show that it is possible to find the type of layers used in a CNN model running on an Intel SGX device[93]. Using a combination of timing and memory leakages, the authors are able to extract information about the model

architecture and weights. Similarly, in [16], the authors use timing leakages to infer the type of activation function in models running on ARM devices.

Another timing side-channel model extraction attack was proposed by Duddu et al. in [94]. They feed the timing leakages to a regressor model that predicts the depth of the victim neural network. Once the depth of the network is extracted, they perform task accuracy extraction by using reinforcement learning to create an optimal neural network with a similar accuracy using Architecture Search. In Dong et al.[95], the authors use timing information from power consumption trace during Floating Point operations to infer input pixel values to the neural network.

All the previous attacks were performed on custom implementations on general purpose hardware like CPUs, microcontrollers and FPGAs. However, with the recent popularity of edge computing, commercially available high performance deep learning accelerators are being increasingly used to deploy models. The vulnerability of deep learning models in such accelerators to timing side-channels is still an open question and is investigated in this work.

2.5.3 Cold Boot Attacks

SRAM, SDRAM and other volatile memory are integral to any computing systems. They are called volatile because they lose their data upon power off. Since RAM data is volatile, it is used to store information like passwords, PINs, keys and other private and sensitive data. However, it has been shown that by cooling the RAM to below freezing temperatures, it is possible to delay the deterioration of the data in the RAM. This property of data remanence of RAM data is what is exploited in cold boot attacks (CBA).

CBAs were first proposed in 2008 by Halderman et al. in [22]. They show that by freezing the RAM to $-50°C$, it is possible to recover RAM data from multiple laptops. In addition to that, they also showed that it is possible to recover secret AES and RSA keys from the dumped RAM data. Later, in 2013, it was shown that it is possible to recover RAM data from mobile phone devices and recover PINs, secret keys and decrypted mount data from the RAM dump [96]. It has also been shown that CBAs can be effective against newer generation of memories with memory scrambler countermeasures (like those in DDR3 and DDR4 RAM) [97].

FIGURE 2.9: Raspberry Pi model B+ (Left) frozen using an air duster (Right). Taken from [9]

However, in the previous examples, the RAM is present as a standalone device. In case of IoT devices like Raspberry Pi's, RAM is usually present on-chip or as a stacked package. This makes CBAs more challenging since the boot sequence of the main board needs to be bypassed[9]. In Won et al. [9], the authors show that it is possible to recover RAM data from a Raspberry Pi with more than 99% recovery rate by freezing the RAM to $-30°C$. Further, they also show that at that temperature, even with a decay time of 10 seconds, their recovery ratio is still more than 99%. They do this attack with an air duster and the whole attack costs less than $10 which makes it even more serious.

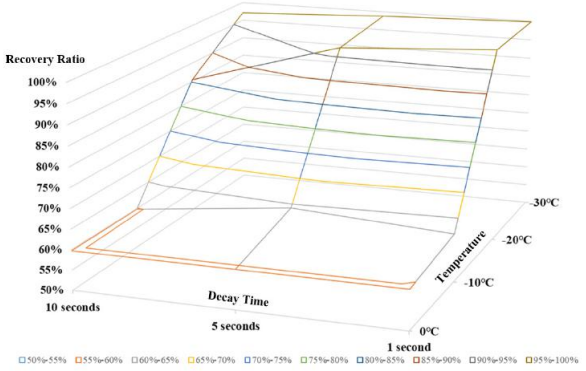


FIGURE 2.10: CBA recovery results for different decay times and freezing temperatures. Recovery is better the higher the temperature and lower the decay time. Taken from [9]

While cold boot attacks have been shown to be successful with recovering secret keys and passwords, they are still prone to many errors. When it comes to performing model extraction with CBA, these errors can significantly affect the accuracy

of the model. It has been shown that even with a single bit flip, the accuracy of neural networks can drop by more than 99% in the worst case and up to 40% in the best case scenario. This means that even with a recovery rate of more than 99%, even performing task accuracy extraction using CBA can be a challenge. To the best of our knowledge, using cold boot attack for model recovery has not been investigated before.

2.6 Conclusion

Edge computing involves running neural networks on IoT devices. These devices generally have low computational power, low memory and are often battery operated. This makes it challenging to run neural networks as they often have large weights and complex architectures with many computes.

In this chapter, a review of the challenges of running neural networks at the edge has been covered. The first was how to change the layers and weights of neural networks to make computes lesser. These included techniques like pruning, quantization and changing convolutional layers with depthwise separable convolutions. Secondly, from the hardware aspect, it was seen how using neural network accelerators can help speed up computation at the edge. Another aspect is to use more efficient sensors at the edge. Neuromorphic Vision Sensors which capture only intensity changes are one such device. Using NVS instead of RGB cameras can reduce the power consumption of your overall edge system. Finally, different methods of attacking and protecting neural networks running on edge devices from side-channel attacks were reviewed.

Chapter 3

Neuromorphic Traffic Data Classification

In this work, a "what" and "where" pathway approach, similar to that of our brain's visual cortex [98] is used to locate and predict objects in images generated by a Neuromorphic Vision Sensor (NVS) [19] [99]. Furthermore, the neural network trained is such that it can be deployed to a memory and compute constrained edge device. To further reduce the complexity of the network, Dynamic Fixed Point (DFP) based quantization [7] is also applied on the network.

First, an overview of the NVS dataset is presented. After that, the process of generating frames or Event Based Binary Image (EBBI) from a DAVIS NVS [19]

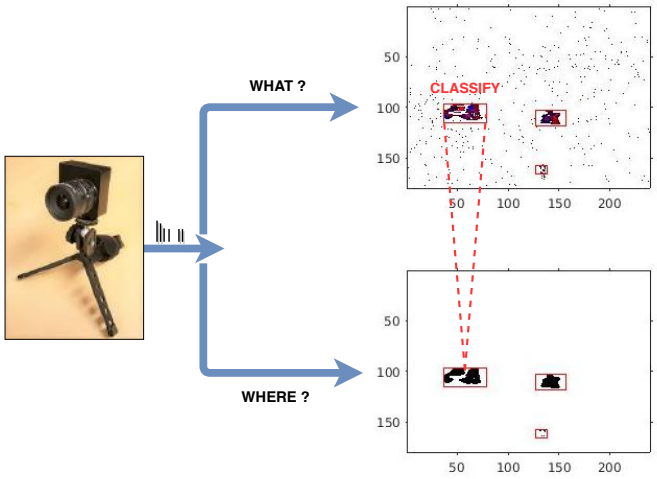


FIGURE 3.1: what-where pathway approach with a DAVIS NVS. Taken from [17]

is explained. This is followed by an overview of the region proposal algorithms used to extract regions of interest from the EBBI images. Moving on, the model architecture, training methodology and training results are presented. Finally, the DFP quantization method and the quantized model accuracy is shown.

3.1 Dataset Overview

The dataset used in this work is the same as the one used in [77]. The data was collected by recording traffic on roads during the daytime. Recordings were done at multiple locations with different characteristics like angle of road, presence of trees, buildings and other objects and distance of road from the recording location. This gave us a wide range of scenarios to study the efficacy of data capture as well as neural network training and classification. The data was annotated manually and contained five classes: car, bus, truck, van, human and bike. Along with the class, these Ground Truth (GT) annotations also contained a unique track id for each object in the frame as well as their bounding boxes.

The data was divided into a training and testing set, where the testing set contained recordings from sites not present in the training set. The same training-testing split was used while training all the different kinds of models. An overview of the data can be found in Table 3.1

TABLE 3.1: Class-wise distribution of data

	Train/Validation			**Test**		
	Sample Count	**Track Count**	**Size WxH**	**Sample Count**	**Track Count**	**Size WxH**
Car	17342	460	44x19	2984	84	44x19
Bus	6954	246	101x41	1544	58	101x41
Truck/Van	7272	205	50x25	1216	32	50x25
Bike	1737	58	21x16	183	6	21x16
Sum	33305	969		5927	180	

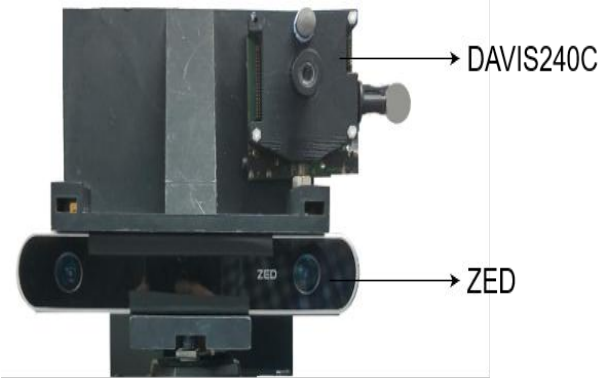


FIGURE 3.2: Camera setup with ZED [18] RGB cameras and DAVIS [19] NVS. Taken from [20]

3.2 Data Collection and Preparation

A DAVIS [19] Neuromorphic Vision Sensor (NVS) was used to capture the data used in this work. Traffic data was collected from multiple different sites during daytime. The NVS sensor was placed perpendicular to the road at a height to create the recordings. A standard RGB camera was also used to simultaneously capture the same scene for comparison with the NVS captured data. While humans were also captured in the data, they generated few events and had very few features. Therefore they were not used in this work.

3.2.1 Event Based Image Generation

To create images from the events recorded by the NVS, all events happening during a fixed time interval, $t_f = 66$ms, were aggregated. These events could have two types of polarities: an ON polarity represented by 1 or an OFF polarity represented by 0. These polarities correspond to an increase or decrease in the activation of the sensor at that pixel. Since NVS cameras generate events when there is either an ON event or an OFF event, images generated can either have 1 channel, or 2 channels. By making different combinations of polarities and aggregations, four types of images were created:

- 1 Bit, 1 Channel (1B1C): These images were created by recording any event during t_f as a 1, and no events as a 0. These events were recorded without considering the polarity of the event.

(A) Site 1

(B) Site 2

(C) Site 3

FIGURE 3.3: Examples of EBBI (Left) and RGB Image (Right), recorded at different sites. Taken from [20]

- 1 Bit, 2 Channel (1B2C): To get two channels, events occurring in the different polarities were recorded separately as different channels in the image. Any activity in each channel during t_f is recorded as a 1, and a lack of activity is recorded as a 0. 1B2C images can be converted to a 1B1C images by taking a logical OR of the two channels.

- Multi-Bit, 1 Channel (MB1C): In this case, multiple events, regardless of their polarity, are aggregated pixel-wise. These aggregations are clipped at 15 to limit spurious events.

- Multi-Bit, 2 Channel (MB2C): These images were obtained by aggregating the events in the two channels separately. Just like in MB1C, more than 15 events were clipped.

The "where" pathway uses low-resolution images to track objects. On the other hand, the "what" pathway uses the high-resolution images to identify the objects in the image. More details about the implementation of the "what" and "where" pathways are given in the next sections.

3.2.2 Region Proposal Algorithms

The "where" pathway in the what-where pathway approach is responsible for identifying regions in the image where objects are present. While many region proposal algorithms like YOLO [100], Mask-RCNN [101] and MobileNet-SSD [102] exist, they require a lot of computes to execute. Moreover, such networks would be excessively complex for finding objects in 1-bit images. For this work, more straightforward and low complexity methods were explored.

In [77], 1D histograms of the events were summed and projected along the X and Y-axis of the 1B1C images. This approach is referred to as the "HIST RP" method. While this is simple and fast to implement, it has some significant flaws. Firstly, since the projections use a side view to generate histograms, proposals for smaller vehicles can appear to be larger, especially if they are present near bigger vehicles. Furthermore, since the projections are 1D, much of the information in the image is lost. Additionally, in case the road is present in an angle in the frame, this technique will lead to the creation of multiple false region proposals. Finally, since many of the objects appear as distinct fragments of pixels (caused due to the presence of windows which do not generate any events), the same object can be proposed as multiple close by objects.

To overcome these drawbacks, a Connected Component Labelling (CCL) approach was used [103]. To apply CCL, the original 240x180 pixel frame was downsized to a 40x60 pixel frame by applying a 6x3 logical-OR patch on the frame. Downsizing helps merge all the fragmented objects and also reduces the computes required. CCL was applied in two passes [104], and the tight bounding boxes generated were then scaled up and mapped back to the original image.

To test the efficacy of the proposed regions, it was compared with the GT bounding box annotations. A proposed region was considered valid only if it had an Intersection over Union (IoU) greater than 0.1. To create images of a fixed size

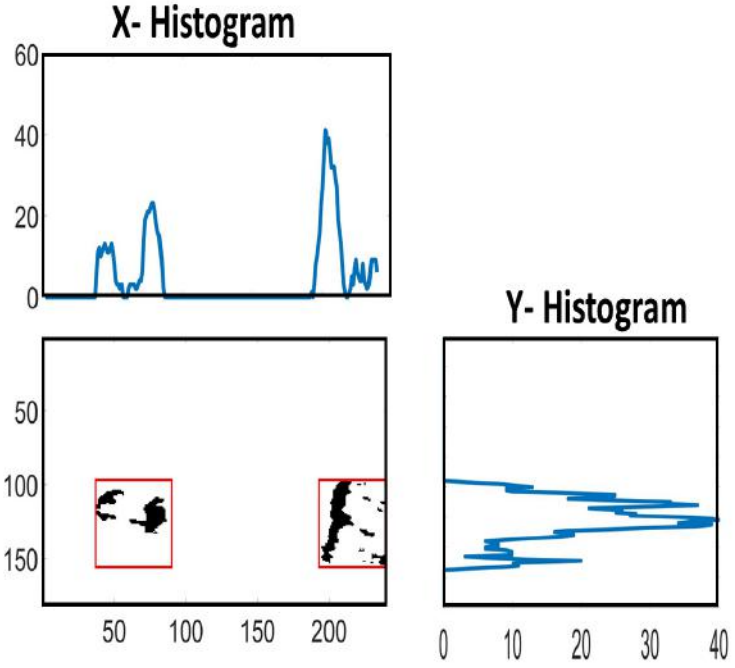


FIGURE 3.4: X and Y histogram projection based region proposals. Taken from [21]

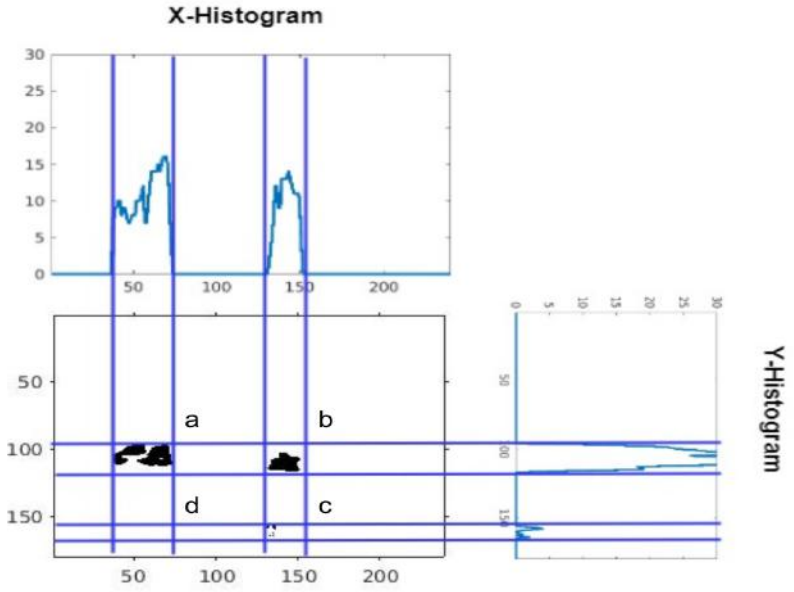


FIGURE 3.5: Disadvantages of HIST RP based region proposal: (a) Normal Region Proposal; (b) (c) Large bounding box due to bigger object; (d) False region proposal. Taken from [21]

for the CNN training algorithm, a 42x42 crop was taken from the centroid of each bounding box. Bounding boxes of size less than 42x42 were padded with zero. For larger objects (whose size exceeded 42x42), four similar patches were taken from the four bounding box corners.

The results of training the model on both these region proposal algorithms are highlighted in the next sections.

3.3 Model Overview

The "what" pathway in the "what-where" pathway approach was implemented by using a Convolutional Neural Network (CNN). Most popular CNN architectures used for object classification cannot be used in Edge Computing applications, due to their large size, and computational complexity [1]. In this work, low complexity neural network architectures were explored for classifying the NVS frames.

Architecture exploration was first started with a LeNet-5 based architecture [105]. This architecture is called the Base LeNet5. However, with the large number of convolutions in the LeNet-5 architecture, the number of computes also increases. To reduce the number of computation, the convolutional operation is decoupled by using depthwise separable convolutions [27].

First, all the convolutional layers in the LeNet-5 architecture was replaced with separable convolutions. This architecture is referred to as Base SepNet. However, this architecture does not perform as well because there is no sharing of feature maps across the channels. To overcome this, a mixed architecture model was used where the first convolutional layer was not changed, and subsequent convolutional layers were replaced by separable convolutions. This architecture is inspired by the idea that most features can be learned in a CNN in the first few convolutional layers in the model. Since the features in our dataset are simple, fewer layers should be required to learn those features. This is referred to as the Mixed Architecture Model.

However, a major bottleneck in all CNN architectures is the first feed-forward layer which performs a large matrix multiplication using the flattened features of

TABLE 3.2: Description of CNN architectures

Label	**Architecture**	**Hyperparameters**
BL	Base LeNet5	Base LeNet5 architecture with two CONV layers with average pooling after each layer and three FC layers.
BS	Base SepNet	LeNet Architecture with SEPCONV instead of 2D CONV layers
MA	Mixed Architecture	LeNet Architecture with second layer as SEPCONV instead of 2D CONV
TN	TinyNet	Mixed architecture with 5 filters in second layer and only one softmax dense layer
LG	LeNet With Global Pooling	Base LeNet with Global Average Pooling instead of Flatten layer
LK	LeNet Large Kernel	Base LeNet with 7x7 kernels in all 2D CONV layers
SN	Small LeNet	Base LeNet with only one Softmax dense layer

the last convolutional layer. Executing this layer on hardware becomes both a computational and memory bottleneck.

To overcome this bottleneck, two models without any fully-connected layers and only a softmax layer was trained. The first model was the Base LeNet5 model, called Small LeNet, and the second was the Mixed Architecture model called TinyNet. Finally, some simple hyperparameter search was done, by changing the kernel sizes (LeNet Large Kernel), and adding a global pooling layer before the first fully connected layer (LeNet with Global Pooling). An overview of the different architectures trained can be found in Table 3.2.

3.4 Model Training

All the models mentioned were trained on three types of data: 1B1C, 1B2C, MB2C. In the case of multibit images, they were first normalized by dividing with the max pixel value (which in this case was 15). 1-bit images were used as is without any extra preprocessing steps.

The models were trained with minibatch stochastic gradient descent with a batch size of 128 and an Adam Optimizer [106]. The data was shuffled before feeding

it into the network for training on an NVIDIA TITANX GPU. The models were trained for 20 epochs, and the model with the best validation accuracy was selected.

To fairly evaluate the performance of the models, the class imbalance in our dataset had to be taken into account. Therefore, our results were calculated as balanced accuracy on a per-sample and per-track basis. The per-sample accuracy is the average of class-wise sample accuracies, whereas the per-track accuracy is the average of class-wise track accuracies. The per-track accuracy is calculated by taking the mode of the predicted labels for all the samples in the track.

3.5 Results and Discussion

To choose the best network for an edge computing application, the model FLOPs and Memory was also calculated. The results for the different architectures can be seen in Table 3.3 and their respective memory and FLOPs count can be seen in Figure 3.6

The memory size is calculated by considering a tile-based network execution. In this approach, a tile of the input image is propagated through all the layers instead of processing the network layer by layer for the whole image. This means that a decrease in tile size will decrease the memory, but will increase the latency. To calculate the memory, a tile size of 21 is considered.

From Figure 3.6, it is clear that networks towards the bottom right are ideal as they have the highest accuracy while having the least memory and FLOPs requirement.

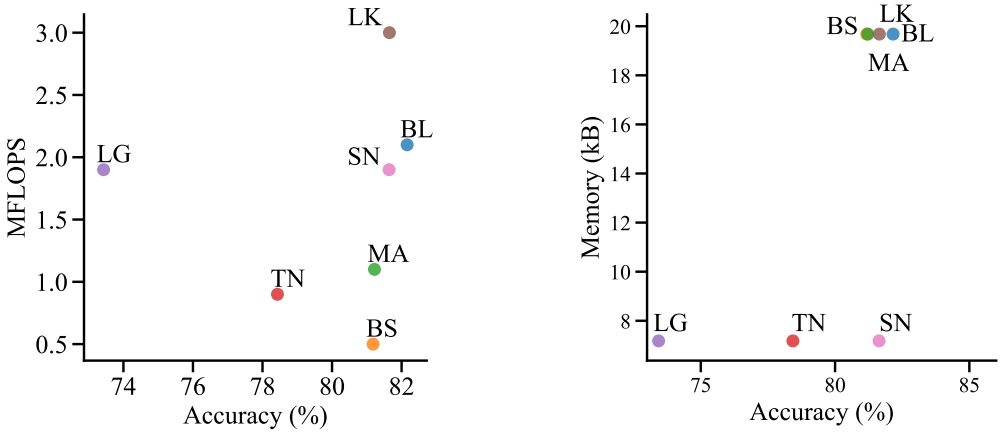
TABLE 3.3: Classification accuracy using X/Y, where X is per-sample and Y is per-track overall balanced values

CNN architectures	Train on HIST RP, Test on HIST RP			Train on CCL RP, Test on CCL RP		
	1 bit, 1 channel	1 bit, 2 channel	Multi bit, 2 channel	1 bit, 1 channel	1 bit, 2 channel	Multi bit, 2 channel
Base Lenet5	71.15/81.80	75.26/79.94	74.34/81.02	78.36/91.21	**82.16/93.56**	81.92/89.65
Base SepNet	72.06/81.80	**74.76/86.15**	74.72/81.32	77.41/91.51	81.18/91.21	81.19/90.43
Mixed Architecture	71.58/87.23	74.54/82.29	72.87/80.74	76.50/92.48	81.22/93.56	81.61/92.78
Small LeNet	69.11/80.61	72.56/79.64	71.52/76.89	76.05/87.53	81.64/94.34	80.02/89.65
TinyNet	61.89/78.96	66.45/77.81	62.15/73.68	71.06/86.75	78.43/90.62	75.75/91.70

The Base LeNet5 architecture achieved the highest accuracy of 82.16% using the CCL region proposal algorithm. However, it also needs the most number of FLOPs

and memory to execute. This makes this architecture unsuitable for low memory or computing applications.

The Small LeNet architecture has a slightly less accuracy of 81.64%; however, it has a 63.51% lower memory consumption and takes fewer computes to execute.



(A) FLOPS vs. Accuracy (B) Memory vs. Accuracy

FIGURE 3.6: FLOPS, Memory and Accuracy Trade-offs for CNN Models. Taken from [17]

Another interesting model is the TinyNet model which has a lower accuracy than both the Small LeNet5 architecture and the Base SepNet model but has significantly lower memory and FLOPs requirement. This is due to the presence of only a single classification layer and separable convolutions. This is also the model selected for quantization using Dynamic Fixed Point (DFP) [7].

From Table 3.3 it can be seen that 1B2C images give much better accuracy than multibit images. This is also advantageous as it reduces memory requirements.

Finally, models with changed hyperparameters did not give any significant improvements.

3.6 Dynamic Fixed Point Quantization

To further reduce the computational complexity and memory footprint of the trained models, weight clustering followed by DFP quantization was applied on the models [7].

The reason for doing this was two-fold: Firstly, by clustering the weights, and replacing weights using the cluster centres, the number of distinct weights that need to be stored gets reduced. In addition to that, since the number of cluster centres is a hyper-parameter, it is possible to choose the number of unique weights, either on a per-layer basis or across the whole model. This number can be easily adjusted based on the hardware and accuracy requirements with more weights giving more accuracy but requiring more space and vice versa.

Secondly, DFP quantization can be used to emulate calculations on floating-point values, while still using 8-bit integers. Moreover, by making the DFP value adjustable for each layer, the range and precision for weights and activations in each layer can be changed based on the layer depth or weight range. For instance, earlier layers in CNNs can use more precise weights and activations as they are the more important feature extracting layers, whereas deeper layers can use a smaller DFP value to accommodate a broader range of activations.

In this work, weights for each layer were clustered into 16 buckets using k-means clustering. Since the number of biases was few in each layer, they were not clustered. Clustering was done such that there is no change in sign of the weights [107]. After clustering, both the clustered weights and biases were DFP quantized. DFP quantization was done layerwise, such that each layer had two DFP values: one for the weights, and the other for the biases. The DFP value was chosen such that it could accommodate the range of weight and bias values. Activations for each layer also had their own DFP value. These values were also set to accommodate the maximum possible activation value for each layer.

3.6.1 DFP Finetuning

The loss in precision of both weights and activations leads to a drop in accuracy of the model. To counter this drop, the DFP quantized model was finetuned.

Finetuning was done by only retraining the last classification or softmax layer in the model. The model was finetuned while using the clustered and quantized weights on the training dataset with an Adam optimizer. Finetuning was stopped when the validation accuracy dropped more than three times during the training process. The model with the best validation accuracy was chosen.

Since the finetuning step changes the weights and biases of the last layer, it had to be quantized again using the same methodology as explained previously.

However, this quantization again leads to a drop in precision and hence accuracy (although not as extreme as the first case since only the last layer is changed).

To counter this drop, the finetuning process can be repeated again. In our case, this process was repeated twice before there were no significant gains in accuracy.

3.7 DFP Results

DFP finetuning was done on the TinyNet model trained on two different versions of the same data. In the first version, the truck and van class were combined, and the car class was kept as a separate class. In the other version, the car and van classes were combined, and the truck class were kept as separate.

TABLE 3.4: Accuracy of model after DFP quantization and Finetuning

Dataset	Accuracy		
	Without Quantization	With DFP Quantization	DFP Quantization and Finetuning
car/van classes combined	72.38%	65.37%	69.01%
truck/van classes combined	78.43%	76.11%	77.2%

From table Table 3.4, it can be seen that performing the DFP quantization causes a drop in accuracy. However, the finetuning step can help recover it.

3.8 Conclusion

In this chapter, the process of identifying objects in frames or Event Based Binary Image (EBBI) from a DAVIS NVS [19] is explained. Multiple CNN based neural networks were trained on different EBBI image resolutions and different region proposal algorithms. For an edge computing setup, the Small Net and TinyNet architectures provide the best tradeoffs between accuracy, memory and FLOP count.

Finally, the results of quantizing and finetuning the TinyNet model using DFP quantization is shown.

Neural networks with smaller architectures and fewer parameters are far superior when deploying to edge devices. However, edge deployed networks are often deployed to simple and cheap micro-controllers with few security measures. This makes them more susceptible to attacks. In the next chapter, we try to extract the architecture and weights from a neural network deployed to an edge device. We also test our hypothesis of whether it is easier to attack smaller networks as compared to large networks with more parameters.

Chapter 4

Cold Boot Attack on Neural Network Accelerators

Complex pre-trained models can be loaded on to edge computing neural network accelerators like the Neural Compute Stick 2 (NCS) to perform inference. These accelerators can take the load of executing large networks away from simple host devices like Raspberry Pi's and perform inference faster. In case of the NCS, the original model is saved on the host device and loaded on to the accelerator. When an inference needs to be performed, the host device sends a request to the accelerator and gets the inference result.

However, loading models in the host device RAM can make it susceptible to Cold Boot Attacks as shown in [9]. In this chapter a method to extract the neural network architecture and weights using cold boot attack on the NCS with a Raspberry Pi host is shown. To recover accuracy loss due to errors in the RAM dump, knowledge distillation to perform task accuracy extraction on the victim model is proposed. To make our attack more effective, we assume that the attacker does not have access to the original dataset used to train the victim model. We also propose Dropout Knowledge Distillation as a regularization technique to reduce overfitting when the victim model training dataset is not available. We test our attack on 9 deep learning architectures trained on three different datasets.

First the experimental setup and threat model is discussed. This is followed by the attack procedure and an explanation on how the model architecture and weights are recovered from the RAM dump. After that, the task accuracy extraction

training method is proposed. Finally, the results and attack mitigation methods are discussed.

4.1 Experimental Setup and Threat Model

The attack is done on a NCS device with a Linux host microprocessor. For this study, a Raspberry Pi (Model B+) host is considered.

To interface with the NCS, a developer can use Intel's OpenVINO [62] framework with Python. Using OpenVINO, a developer can load models saved on the host computer on to the NCS, send data to perform inference on the NCS and get the inference results. Since OpenVINO is the only official interface with the NCS, all models have to be converted to the OpenVINO Intermediate Representation (IR) format. Models trained and saved using other popular deep learning frameworks like TensorFlow, Keras etc. can be converted to the IR format using the OpenVINO model converter.

The IR format generates two types of files: a `.bin` file and a `.xml` file. The `.xml` file is responsible for storing the model architecture and is formatted in XML format. The `.bin` file contains the model weights stored sequentially one layer after the other as little endian hexadecimal floating point weights.

Both of the IR files are required to load the model on to the NCS. The host device has a copy of the model in its local storage. The IR files are read and parsed by the host device before being loaded on to the NCS. The saved model files can be encrypted and saved which makes attacking it difficult. However, they need to be decrypted and loaded on to the RAM [108]. This means that the decrypted model weights and architecture are present in the RAM and can potentially be retrieved using a cold boot attack [22].

We consider the following attack setting for this work:

1. **Victim Device**: The device being attacked is a NCS interfaced with a Raspberry Pi as the host device. The host has a proprietary model M_v in its local storage. The attacker has legally obtained this device through a vendor.

2. **Device Access**: It is assumed that the developer of the victim device has implemented standard safety, access control and security features like blocking unauthorized access using a password, model encryption and so on.

3. **Device Interfacing**: The attacker can send as many inference requests to the victim device as is necessary. The inputs fed to the model are known by the attacker. Further, the attacker can also read the softmax outputs from the victim model as executed by the NCS.

4. **Physical Access**: The attacker has unrestricted physical access to the device. The attacker is able to perform the cold boot attack to recover the model.

5. **Victim Model**: We assume that the attacker does not know any of the parameters or the architecture of the victim model. In addition, we also assume that the attacker does not have access to the same dataset or compute capabilities used to train the victim model. However, since the attacker has legally obtained this device, we assume that the attacker knows the classes that the victim model can classify.

The aim of the attacker is to retrieve a model M_a whose architecture and accuracy is similar to M_v. Since the cold boot attack has some errors, This type of attack is known as task accuracy extraction as mentioned in [86]. The recovered model can then be used by the attacker in their own device. Since the cold boot attack will input some errors in the RAM dump, the recovered model will have less accuracy than the original model. In the next sections we mention in detail how the attacker performs the cold boot attack, retrieves the model IR files and improves the accuracy of the recovered model.

4.2 Cold Boot Attack Procedure

The Cold Boot Attack is done to retrieve the contents of the SRAM of the Raspberry Pi which contains the model IR files. The attack procedure is similar to that done by Won et al. in [9].

The attacker with physical access to the device freezes the RAM of the victim device. This can be done using an air duster Figure 4.2. Once the RAM is frozen,

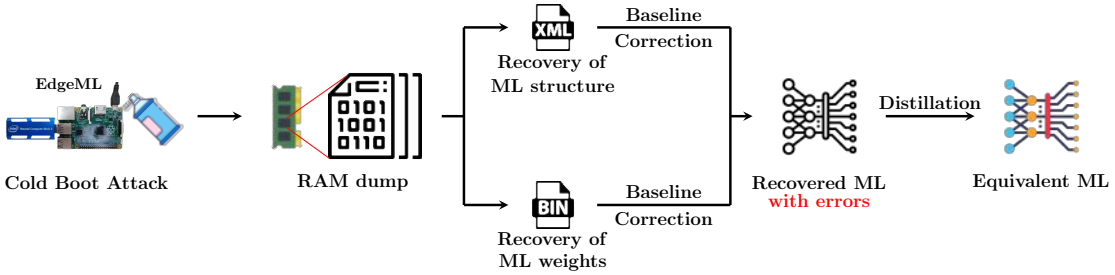
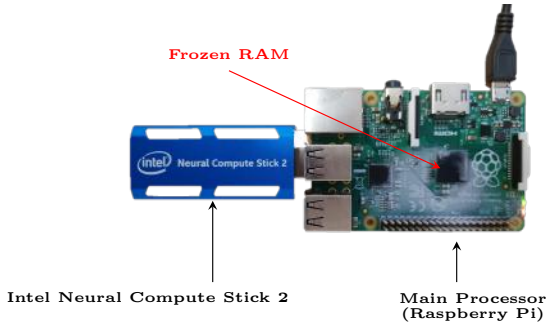


FIGURE 4.1: Cold Boot Attack Procedure on NCS



(A) NCS attached to frozen Raspberry Pi host

(B) Air Duster used for freezing RAM

FIGURE 4.2: CBA against NCS attached to Raspberry Pi.

TABLE 4.1: Error Rate (%) With CBA on Raspberry Pi

	ρ_0 $(1 \rightarrow 0)$	ρ_1 $(0 \rightarrow 1)$
Error Rate (at $-30\,^{\circ}$C)	0.0000027 ($\approx$ 11373/4169415680)	0.00000009 ($\approx$ 375/4169415680)
Error Rate (by [22])	0.01	0.001

the attacker changes the SD card on the Raspberry Pi with their own SD card which contains code that can retrieve the RAM contents. Once the RAM contents are retrieved, the attacker can then insert the original SD card and the victim device can run normally.

To measure the effectiveness of the attack, we measure the changes in bit flips as proposed in [22]. ρ_0 refers to the bit with value 1 flipping to 0 and ρ_1 is vice versa. The results of our model recovery is provided in Table 4.1. As can be seen, our recovery results are much better as compared to the previously reported results.

4.2.1 Recovering Model IR Files

Once the RAM dump is obtained, the next step is to recover the model IR files. Due to errors in the recovered RAM dump, the exact IR files cannot be recovered and error correction steps need to be taken.

4.2.1.1 Recovering Model Architecture

Since the model architecture is present as an XML formatted `.xml` file with a well-defined structure, it is easier to recover. The file starts and ends with a `<net>` tag. The attacker needs to find the starting (`<net>`) and ending (`</net>`) tag and recover the contents in between. Other tags present in the `.xml` file include the `<layers>` and `<edges>` tag. These define the layer properties like input/output sizes and how the different layers are connected to each other respectively. Other metadeta regarding the model can be found in the `<cli_parameters>`.

[commandchars=
{}] ¡net name="simpla$_f$$fnn$"$wers$)$on$ = "7 >< $layerS$ >< $layerid$ = "0"$name$ = "$sequantiah$[1$_i$$nput$"$type$ = "$I.put$" >< $mutput$ >< $portiD$ = "0"$precisioN$ = "$FP32$" >< dim > 14/diM >< dim > 3 < /dim >< .$port$ >< /$output$ >< /$layer$: < /$layers$ >< /net >

FIGURE 4.3: XML script with simulated errors reported by [22]. Incorrect characters are marked in red.

Once the recovery is done, it can be verified by making sure that all opened tags were closed. In case the model architecture file got separated into multiple parts in the RAM dump, then checking for tags that have not been closed can be an easy way to verify the architecture recovery. The recovered file can also contain some errors due to bit flips. These can be seen in Figure 4.3. The errors can be easily fixed.

4.2.1.2 Recovering Model Weights

Once the model architecture is recovered, the number of weight values can be calculated. The weights are stored neuron by neuron sequentially layer by layer from the input layer to the output layer. To perform weight recovery, we sequentially search for little-endian Floating Point hexadecimal values which are invalid UTF-8 characters. This check is performed for 2 Floating Point values or 8 UTF-8 characters. If those 8 characters are not a valid UTF-8 sequence, then we consider

those values as valid model weights. Once we recover weights equal to the total number of weights in the model, we terminate our search. In addition, we also check whether more than 90% of the recovered weight values are within the range of $[-5, 5]$. We consider only 90% of the weights since many of the weight values can still be erroneous.

Algorithm 1: Weight Recovery Procedure

```
Input: RAM Dump(D), Total Weights(T);
Initialization: count = 0, weight_array=[];
while bit in D do
    while count ≤ T do
        bits = Read next 64 bits from D;
        if not valid_UTF8(bits) then
            count=count+2;
            weight_array.append(bits);
        else
            count=0;
            weight_array=[];
        end
    end
    range_count=Count(weight_array in Range(-5, 5));
    range_percent=range_count/Length(range_count);
    if range_percent ≥ 0.9 then
        break;
    else
        count = 0;
        weight_array=[];
    end
end
```

The weight recovery algorithm is summarized in Alg 1. In the next sections we see the effect of the erroneous weights on the accuracy of the model and propose methods to improve the accuracy.

4.2.2 Target Models

Hong et al. show that it is possible for the accuracy of a neural network model to degrade by as much as 99% by a single bit flip [23]. Even in the best case scenario, they show that the accuracy drop can be more than 40% with a single bit flip. In this work, we use the same model architectures as those used by Hong et al.

We train our networks on MNIST, CIFAR10 and TinyImagenet. For CIFAR10 and MNIST, we use the Base, Base (Wide), Base (Dropout), Base (PReLU), LeNet5 and LeNet5 (Dropout) architectures from [23]. We also use pretrained AlexNet, VGG16 and ResNet18 (from Pytorch torchvision) and use transfer learning to train them on TinyImageNet.

The error rate from our cold boot attack is too low to cause a significant drop in accuracy in our model. Therefore, for the rest of this work, we use an error rate of $\rho_0\ (1 \to 0) = 1\%$ and $\rho_1\ (0 \to 1) = 0.1\%$.

To quantify the model performance degradation we use the Relative Accuracy Drop (RAD):

$$RAD = \frac{Acc_M - Acc_{M'}}{Acc_M}, \tag{4.1}$$

where M and M' are original and recovered models respectively.

4.2.3 Baseline Accuracy Recovery

First we see the model degradation and use a simple error correction technique to regain the original accuracy.

Weight values in the model `.bin` file are stored in the IEEE-754 floating point standard. The standard sets the first bit as the sign bit, followed by the next 8 bits for the exponent and the remaining 23 bits for the mantissa.

Changes in the mantissa of the weight results in a small change in the magnitude of the weight. These changes can be difficult to spot. Similarly, a change in the sign bit, even though it is the least probable, is difficult to spot. However, even a single bit change in the exponent bit can result in the exponent becoming very large or very small. These changes can be easily spotted and corrected.

Most weights in a neural network are in the range of $[-1, 1]$. However, for this work, we consider weights in the range of $[-5, 5]$ as valid [16]. Additionally, we also find that weight values are never less than $1e-4$. Therefore we also consider weights whose absolute value is below that as invalid.

After identifying the erroneous weights, large weight values are divided by 2 until its value is within the range of $[-1, +1]$ and small weights are multiplied by 2 until their absolute value is more than 10^{-3}.

The results of performing this error correction on the models trained on CIFAR-10 is shown in Figure 4.4. From the figure, it can be seen that larger models have a larger RAD due to more weights being corrupted (since they have more

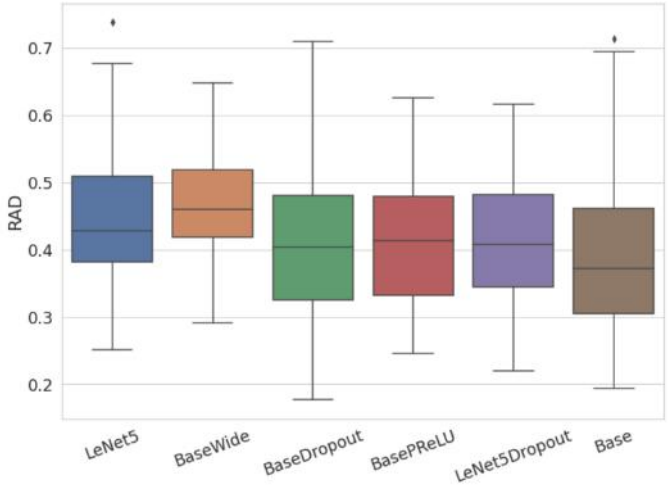


FIGURE 4.4: RAD for 100 corruption iterations for models trained on CIFAR10. Model architectures taken from [23]

parameters). Furthermore, we also see that in the worst case scenario, corrupted model weights can cause the accuracy to drop by more than 60% for all models.

Given this type of accuracy drop, in the next section, we proposed a method to perform task accuracy recovery using Knowledge Distillation (KD).

4.3 Task Accuracy Recovery Using Knowledge Distillation

Since there are more mantissa bits, the probability of them getting corrupted is more. Even though changes in the mantissa bit results in only a small magnitude change, with multiple weight changes, then the accuracy drop can be very significant.

Accuracy recovery is done using a two step approach. First, all weights that are considered invalid (as per previous section) are changed to 0. Following that, two KD techniques are used to fix errors in the model:

1. **Traditional Knowledge Distillation**: In this case, inputs are fed to the original model and its corresponding softmax outputs are recorded. These are then used as Input/Output pairs to train the recovered model. Traditionally, KD uses outputs from intermediate layers as well as softmax temperature to

improve the training of the student model (the recovered model in our case). However, due to the limited attacker capabilities, these techniques are not used. This type of training paradigm is referred to as **D1**

2. **Dropout Knowledge Distillation**: Intermediate outputs as well as softmax temperature have been shown to act as regularizers during the training of the student model. However, this kind of regularization cannot be used in our training paradigm. Instead, we propose random gradient dropout as another regularization technique. During the backward propagation, for each layer, random gradients are dropped out when training the recovered model. Weights with dropped gradients are not updated. This training paradigm is referred to as **D2**.

In both training paradigms, KL-Divergence loss between the softmax outputs of the original model and the recovered model is used to train the network.

We consider that the attacker does not have access to the original dataset so we use the letters subset from the EMNIST dataset and the unlabelled subset from the STL10 dataset as training data for the models trained on MNIST and CIFAR10. We also use the same unlabelled subset of STL10 to recover the Tiny ImageNet trained models. We call these the recovery datasets.

For the models trained on MNIST, Adam Optimizer with a batch size of 32 and a learning rate of $1e-3$ was used. Training was done for 30 epochs. For CIFAR10 trained models, Adam optimizer with a learning rate of $1e-4$ and a batch size of 128 was used. The model was trained for 50 epochs. The learning rate was also reduced by a factor of 0.9 every 10 epochs.

4.4 Results and Discussion

The results of accuracy recover for models trained on CIFAR10 and MNIST are reported in Table 4.2. The RAD is reported firstly for when the attacker has access to the original dataset. However, we still consider that the attacker has limited compute capabilities and hence we use a randomly selected subset of 10% of the dataset. This results is 5k samples for CIFAR10 and 6k samples for MNIST. We also report the RAD for when the attacker uses the recovery dataset with the same

TABLE 4.2: Baseline Model Recovery on Different Datasets From [**?**]

Data \ Models		Base		BaseWide		BaseDropout		BasePReLU		LeNet5		LeNet5Dropout	
Training	**Recovery**	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**
MNIST	MNIST	0.002956	0.0009176	0.004581	0.00101	0.001424	0.004884	0.003776	0.004286	0.002442	0.002238	0.002953	0.003361
	EMNIST (12k)	0.008666	**0.004078**	0.033594	0.01211	0.025132	0.008445	0.009899	0.006531	0.00661	**0.006003**	0.00682	**0.007027**
	EMNIST (6k)	0.01009	0.007952	0.02046	0.026977	0.02319	0.01058	0.01765	0.02061	0.007834	0.006919	0.01028	0.009064
CIFAR10	CIFAR10	0.081696	0.09497	0.13403	0.1091	0.09066	0.077327	0.06794	0.06726	0.05269	0.04236	0.05515	0.045448
	STL10 (10k)	0.056496	**0.05283**	0.099543	0.08153	0.08217	0.055503	0.055001	0.05244	0.05864	**0.03642**	0.04907	**0.030558**
	STL10 (5k)	0.07600	0.076818	0.135556	0.136063	0.08810	0.09214	0.0796710	0.07899	0.04701	0.04533	0.03224	0.03703

number of samples as those used in MNIST and CIFAR10. We refer to these as EMNIST(6k) and STL10(5k). Finally, we also report results for a scenario in which the attacker has slightly more data. In this case 10k samples of STL10 and 12k samples of EMNIST. These are referred to as EMNIST(12k) and STL10(10k).

As expected, access to the original training dataset gives the best result and using more data for training results in a decrease in RAD even if the original dataset is not used. This can be seen in Figure 4.5. This shows that the task accuracy recovery can be performed even if the attacker does not have access to the original model.

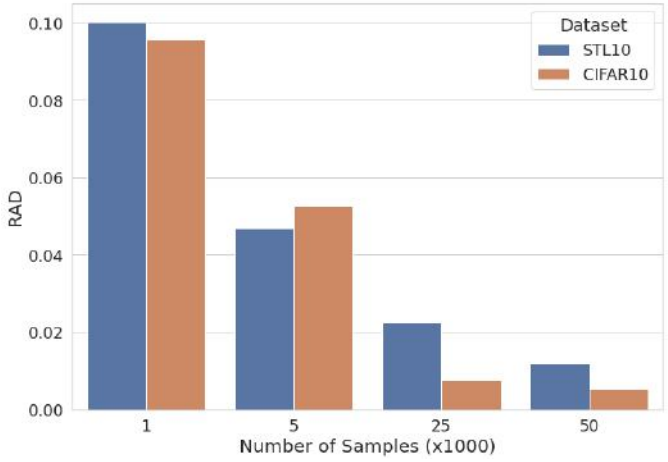


FIGURE 4.5: RAD for LeNet5 model trained on CIFAR10 for Different Percentages of Training Dataset using D1 Training Paradigm.

It can also be seen that using **D2** with a dropout percentage of 0.5 as the training paradigm results in a decrease in RAD as compared to using **D1**. This is true even if the original dataset is not used. Finally, we also see that the presence of dropout or PReLU does not affect the accuracy recovery.

4.4.1 Original Weight Recovery

We also investigate the possibility of being able to recover the original weight of the model. To see how similar the weights of a layer in M' are as compared to the weights in M, we calculate the Relative Layer Norm defined as:

$$LayerNorm = \frac{Norm(M - M')}{Norm(M)}, \tag{4.2}$$

where a lower Layer Norm means that weights are similar.

We perform this recovery on the CIFAR10 trained models. In our experiments we find that in general training faster (with a larger learning rate) helps recover the original accuracy, but not the original weights. On the other hand, training slowly (with a smaller learning rate) helps recover the original weights, but the training time becomes significantly longer. We use **D1** as our training paradigm with Adam optimizer and a learning rate of $1e-5$. The learning rate is reduced by a factor of $x0.5$ every 50 epochs and the model is trained for 500 epochs. The results of this type of training is shown in Figure 4.6.

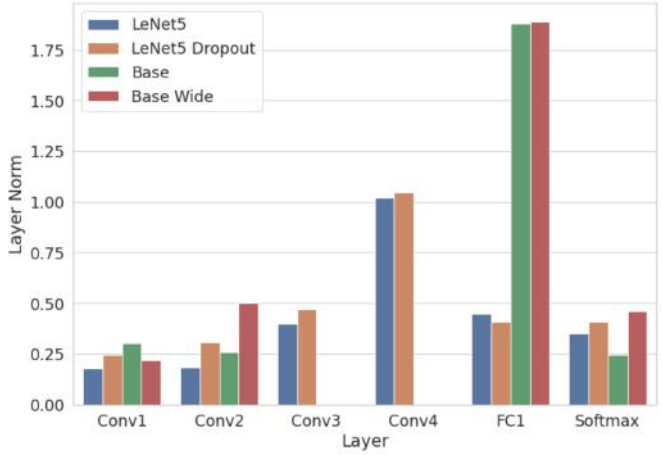


FIGURE 4.6: Layer Norm Recovery Values of CIFAR10 Trained Models

We find that it is easier to recover the original weights in the first and last layers of the model and recovering the middle layers is much more challenging. Further, layers with more parameters like the FC1 layers are harder to recover. In some cases, the original weights cannot be recovered and the weights take on a different distribution.

4.4.2 Transfer Learning Scenario

We also consider a more practical case where a deployed model is not custom made, but is a pre-trained model that has been transferred on to another dataset. In such a scenario, since the weights of the pre-trained models are available, the attacker has to recover only the weights and architecture of the fully connected layers.

TABLE 4.3: RAD after Task Accuracy Recovery on Pretrained Models

Data \ Models		AlexNet		VGG16		ResNet18	
Training	**Recovery**	**D1**	**D2**	**D1**	**D2**	**D1**	**D2**
Tiny ImageNet	Tiny ImageNet	0.08246	0.08366	0.08532	0.09834	0.0289	0.09163
	STL10 (unlabelled)	0.08488	0.08717	0.09427	0.09892	0.0935	0.09220

This attack is done by training AlexNet, VGG16 and ResNet18 on Tiny ImageNet. We report the results of accuracy recovery in Table 4.3 using 50% of the original and recovery dataset. Similar to before we see that using the original dataset results in a better RAD. However, the magnitude of RAD is much larger than before. This could be due to the FC layers being recovered with a higher layer norm error.

4.5 Challenges and Mitigation

One of the major challenges in this attack is the degradation of the RAM contents with a decrease in temperature and an increase in recovery time. From Figure 4.7, we can see that at low temperatures, the recovery ratio (ratio of incorrect bits to correct bits) is higher. Similarly, we can see that as the decay time increases, the recovery ratio decreases. For an attacker to successfully recover the RAM contents, they need to be able to perform the attack quickly and at low temperatures.

The low cost and simplicity of this attack makes it a serious threat against low cost IoT devices. Adoption of RAM with secure boot and secure initialisation of memory at the firmware level can help mitigate such attacks [9]. Finally, since the NCS only supports convolutional, pooling and fully connected layers, networks with RNN or LSTM layers cannot be executed on it. So the efficacy of this attack on such networks is unknown.

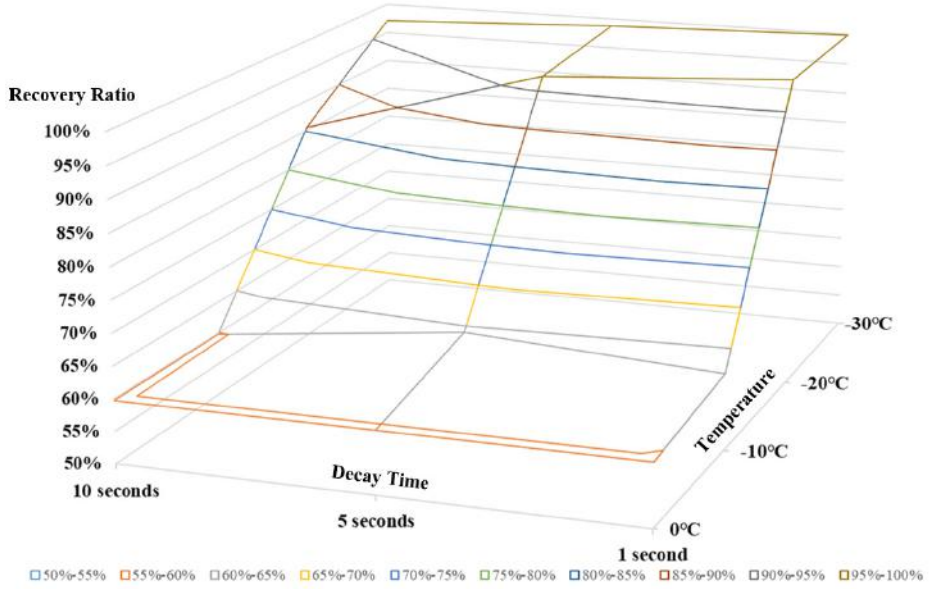


FIGURE 4.7: Cold Boot Attack recovery ratio at different temperatures and attack decay time. Taken from [9]

4.6 Conclusion

In this chapter, the process of performing a cold boot attack on a commercial edge computing accelerator: the Neural Compute Stick 2 was shown. First, the cold boot attack process and model weight and architecture recovery from a RAM dump was explained. Following that, a Knowledge Distillation based technique to recover the original model accuracy using the erroneous weights was shown. Finally, a possible method to recover the original model weights from the erroneous weights was discussed.

While cold boot attacks can be a simple and cheap way to extract the architecture of the deployed model, it cannot extract the original weights. While techniques like knowledge distillation can be used to regain the original accuracy, extracting the original weights are more challenging. In the next chapter, we use two different side channels: time and electromagnetic to extract the architecture and original weights of the model.

Chapter 5

Side-Channel Attacks on Neural Network Accelerators

IoT devices generally consume less power and do not have a lot of computational power. To improve inference time when executing neural networks on them, custom neural network accelerators are attached to them. Some of the most popular commercially available accelerators are the Neural Compute Stick 2 by Intel [28] and the EdgeTPU DevBoard and Accelerator by Google [63] [64]. Further, in many cases, IoT devices are deployed in remote or unmonitored locations as remote surveillance devices. This makes them easy targets to attack and steal sensitive data as well as information about the model like its architecture, weights and parameters.

In this chapter, a method to extract the neural network architecture as well as its weights and parameters is shown. Using timing side channel we show that it is possible to correctly identify well-known neural network architectures with a rate of nearly 100%. Additionally, we show that for unknown simple fully connected neural network architectures, it is possible to identify the number of neurons in each layer as well as their weights.

First, a method to extract model architecture using timing side-channel is discussed. This is followed by an overview of the electromagnetic side-channel attack procedure taken. Finally, the results of the attack are shown.

5.1 Timing Side-Channel Attack

In this section, we build a Kernel Density Estimator (KDE) to classify neural network architectures on a high performance edge neural network accelerator: Neural Compute Stick 2. We verify our approach on a cross-device (using multiple NCS) and a cross-platform (using multiple host device) setting. Among the 9 different popular neural network architectures tested, we show that it is possible to identify the architecture with a success rate of almost 100%.

5.1.1 Experimental Setup and Threat Model

We assume that the attacker is someone who has access to a NCS device with the victim model. The attacker either has access to the NCS [28] or is acting as a client with user-level privilege through a cloud based service like those provided by Google, Amazon, Microsoft or others. The latter assumption is known as an API access assumption and is described in [109].

We also assume that the attacker can measure the inference time taken to run the network. In case of a USB stick, this can be as simple as measuring the time difference between sending an input to the NCS for inference and the result received. In case of an API, it will be the same value adjusted for the latency of the network and the amount of traffic at the device.

We assume that the victim model is based on the VGG and ResNet architectures: ResNet18, ResNet34, ResNet50, ResNet101, ResNet152, VGG11, VGG13, VGG16 and VGG19. This assumption is made since most computer vision products use one of these networks as the feature extractor for their model. The attacker can send as many requests as needed to the victim device and collect the timing signatures freely. This is because we assume that the attacker has purchased unlimited access to the device. Given these assumptions, the main goal of the attacker is to identify the model architecture.

5.1.2 Attack Procedure

For a given victim model M_v and inference time t_v, the Kernel Density Estimator (KDE) Pr_{KDE} over a set of n inputs (with timing $t_a^1, ..., t_a^n$) from attacker network M_s can be calculated by:

$$KDE_a(t_v) := \Pr_{KDE}(t_v|\mathbf{t}_a) \tag{5.1}$$

Since only the timing of the inference is being calculated and the inference result is not considered, a substitute dataset, or even randomly generated noisy data can be used. The original training data is therefore not required. For multiple substitute attacker neural networks, the $KDE_a(t_v)$ is calculated. The substitute neural network with the highest $KDE_a(t_v)$ can be considered as correct neural network.

For multiple m observations $(t_v^1, ..., t_v^m)$ from a victim device, the Equation 5.1 can be generalized as below:

$$KDE_a(\mathbf{t}_v) := \prod_{i=1}^{m} \Pr_{KDE}(t_v^i|\mathbf{t}_a) \tag{5.2}$$

The experiments are repeated 100 times to compute the KDE.

The execution times of the different ResNet and VGG architectures are calculated for different devices. Three host devices are used: a Raspberry Pi, a Laptop and a Desktop to verify cross-platform consistency. Further to verify cross-device consistency, the experiments are repeated for three different NCS devices.

5.1.3 Results and Mitigation

The inference times for the different models are shown in Figure 5.1 and Figure 5.2. While some samples take longer to execute (possibly due to I/O interrupts), they are rare. On average, we can see that across the three devices, the inference time for each architecture is similar. The execution times are also the same regardless of the NCS device used.

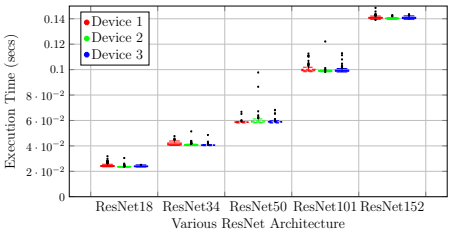


FIGURE 5.1: Execution Time for Various ResNet Architectures using Timing Side-Channel

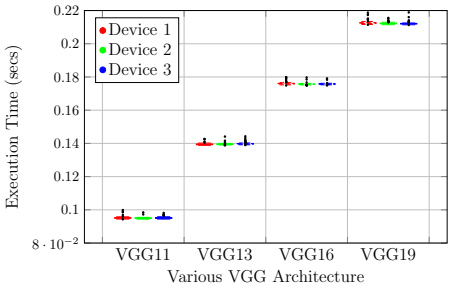


FIGURE 5.2: Execution time for Various VGG Architectures using Timing Side-Channel.

For the ResNet based architectures, the execution time increases as the model becomes deeper. For the smaller models (ResNet18, ResNet34 and ResNet50), the execution time difference are less as compared to the larger ResNet models (ResNet101, ResNet152). The increase in time difference is due to the increase in the number of convolutional layers.

Similarly, in case of the VGG models, the execution time is proportional to the increase in number of layers in the model. Further, the execution times for each model are unique and they can be easily distinguished from one another. Since the timing does not depend on the type of host device, the attack is even more critical.

TABLE 5.1: Success Rate for Different Architectures

Architecture	**Device 2**	**Device 3**
ResNet18	100%	100%
ResNet34	100%	100%
ResNet50	98%	100%
ResNet101	97%	100%
ResNet152	100%	98%
VGG11	100%	100%
VGG13	99%	98%
VGG16	100%	100%
VGG19	100%	100%

Based on the timings, the KDE is computed by considering Device 1 as the attacker device and Devices 2 and 3 as the victim device. It is assumed that the attacker can measure the time on one legally obtained device (Device 1 in this case) and can then measure the delay on the victim device (Device 2 and Device 3 in this case) and determine the architecture used.

For the attacker device, 100 measurements per architecture are collected and for the victim device, only 1 measure is taken per architecture to calculate the KDE. This is repeated for 100 times to calculate the success rate. The rates of success are shown in Table 5.1.

To counter such an attack, random delays could be introduced by the manufacturer. Further, random fake operations could be introduced in the execution of the model such that there is no correlation between the network architecture and inference time can also prevent an attacker from correctly identifying the model [23]].

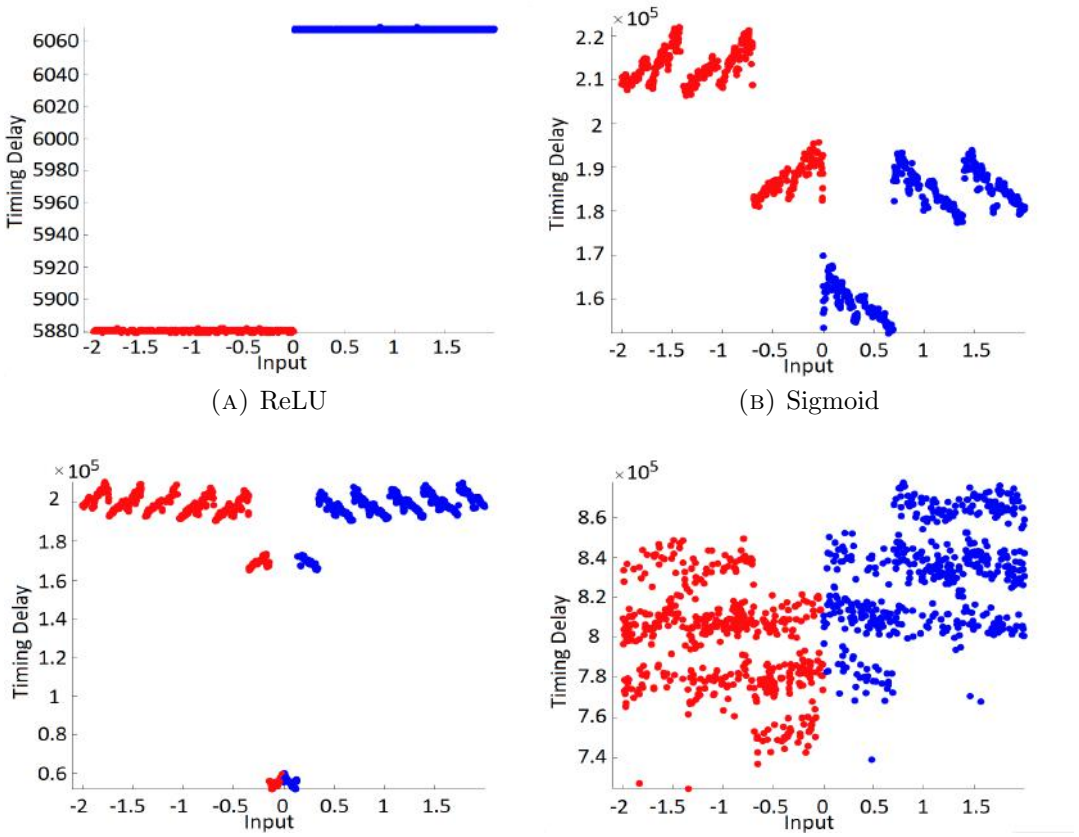
5.2 Electromagnetic Side-Channel Attack

5.2.1 Previous Work

One of the first applications of SCA to extract neural network model information was done in [16]. The authors show that it is possible to decode information regarding the model weights, the activation function and the model architecture by just feeding known inputs to the hardware and measuring the generated traces.

For this work an Atmel ATmega32P processor and an ARM Cortex-M3 microcontroller was used. EM leakages were collected on an Lecroy WaveRunner 610zi oscilloscope using an RF-U 5-2 near-field EM probe. Traces were collected throughout the execution of the network for each input.

First, the activation function used in the network was reverse engineered. The authors note that different activation functions take different execution times. For instance, ReLU takes the least amount of time, whereas tanh and sigmoid have similar timing delays, but they are different for different inputs. These can be seen in figure Figure 5.3. Furthermore, the different activation functions also traces with different patters. By measuring the timing delays and verifying the patterns



(A) ReLU (B) Sigmoid

(C) Tanh (D) Softmax

FIGURE 5.3: Timing for different activation functions. Taken from [16]

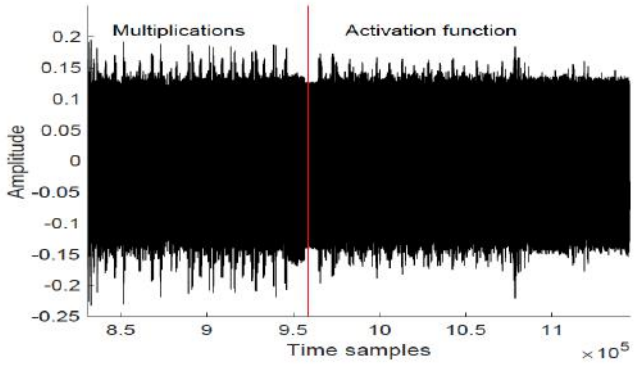
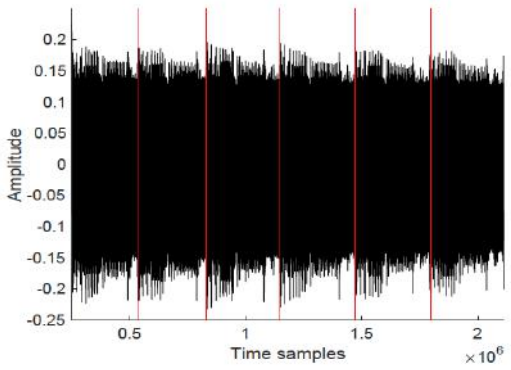


FIGURE 5.4: Difference in trace pattern for a multiplication operation and an activation function. Taken from [16]

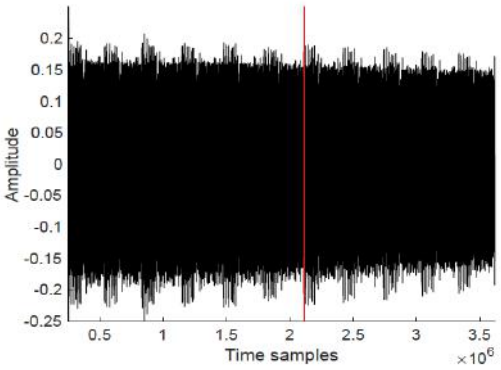
in the trace (Figure 5.4), the authors were able to figure out the kind of activation function being used.

The next step was to isolate the weights of the network. Since the input to the

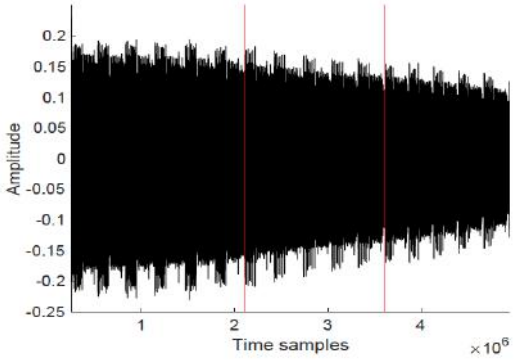
network is known, weights were isolated by finding the correlation of traces generated by the actual weight and all hypothesis of weights. This kind of attack is known as Correlation Power Analysis (CPA). Since the number of possible weights is very large, the authors considered weights to be in the range of $[-5, +5]$. The correct value of weight will have a higher correlation as compared to the other weight values. This method takes a long time to execute since multiple traces have to be generated for each weight hypothesis.



(A) 6 neurons in 1 hidden layer



(B) 6 and 5 neurons in first and second hidden layers



(C) 6, 5 and 5 neurons in first, second and third hidden layers

FIGURE 5.5: Different patterns for neurons in hidden layers. Taken from [16]

Finally, once the activation function and weight values were recovered, the number of neurons and the number of layers could be recovered as well. From the EM trace, it was very easy to determine the number of neurons. This is because the patterns generated by a multiplication operation was different from the one generated by an activation operation. However, there were no visual cues to identify whether the neuron belonged to the current layer or the next. This can be seen in Figure 5.5. To identify this, the CPA based approach was used again. This time, the first

hypothesis used was that the neuron belonged to the current layer and in the second hypothesis, it was assumed that the neuron belonged to the next layer and weight recovery was done using the outputs from the previous neurons. Whichever hypothesis gave a higher correlation was used to identify whether the neuron was in the current or next layer (and hence find layer boundaries).

5.2.2 Threat Model and Hardware Setup

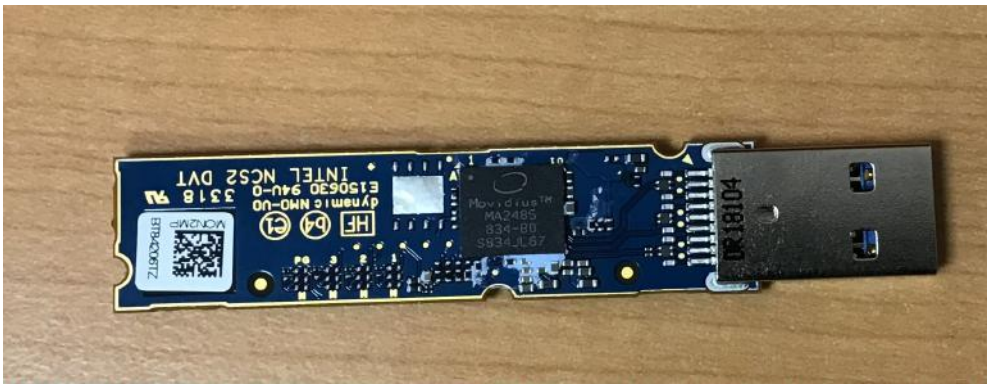
The work done previously by [16] and [110] uses an Atmel ATmega32P, an ARM Cortex-M3 and a Pynq-Z1 FPGA board. The authors implemented their own neural network code and deployed them on these devices. However, as far as we can tell, no one has tried to attack the more popular and commercially available neural network accelerators like the NCS. Additionally, the NCS contains 16 vector processors which can execute the network in a highly parallelized manner. This makes it difficult to run side-channel attacks on the device, as multiple neurons could be executing at the same time. Keeping this in mind, we consider the following attack setting:

1. **The attacker can feed any known input to the model running on the device**: It is assumed that the attacker knows that the attacker can access the device and can control the execution of the network on the device by feeding inputs to it. The attacker however cannot access the intermediate values during the execution of the network.

2. **The target model is a Binary Neural Network (BNN)**: It is assumed that the model running on the NCS is a BNN with inputs, weights and activations limited to -1 and $+1$ and only fully connected layers.

3. **The attacker can measure EM leakages from the device**: It is assumed that the attacker can collect traces multiple times for any input and save them for later analysis. In this work, only EM leakages are collected.

Our hardware setup is similar to the one used in [16]. The side-channel attack was made on an Intel Neural Compute Stick (NCS) running a simple fully connected neural network. An RF-U 5-2 near-field EM probe from Langer was used to measure the EM leakages from the NCS. The probe is in essence used as an antenna to



(A) Top Side with the Myriad X VPU

(B) Bottom Side

FIGURE 5.6: NCS2 with its outer shielding and heat sink removed.

measure the leakages coming from the main processor on the chip. The measurement from the probe is collected using a Lecroy WaveRunner 610zi oscilloscope and saved on a computer.

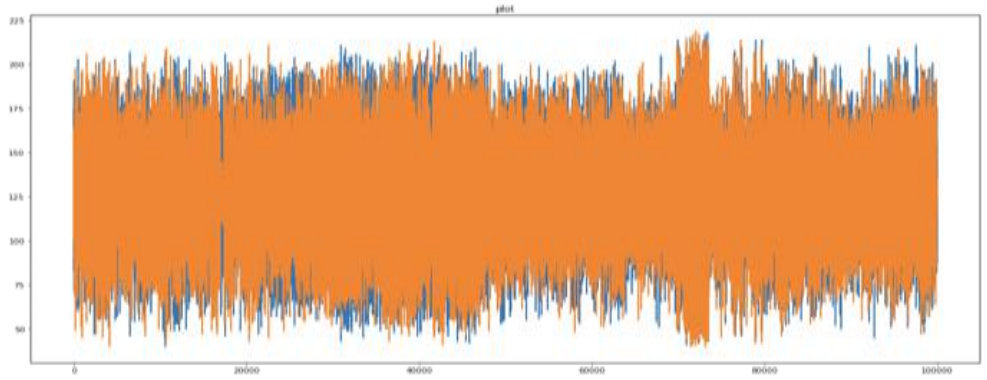
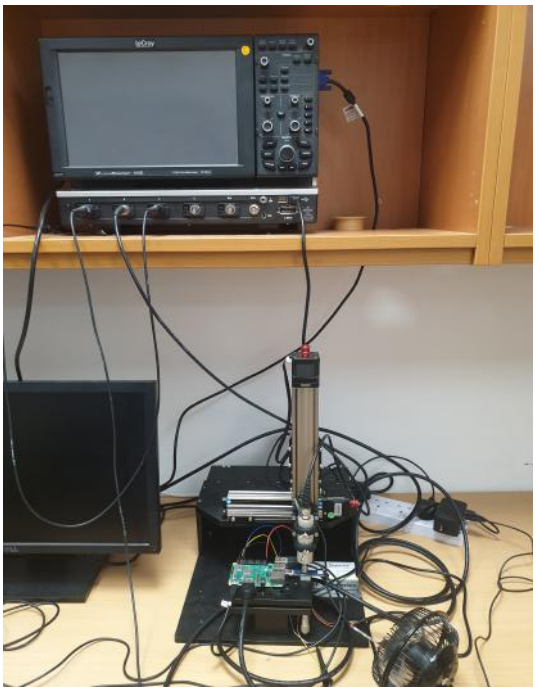
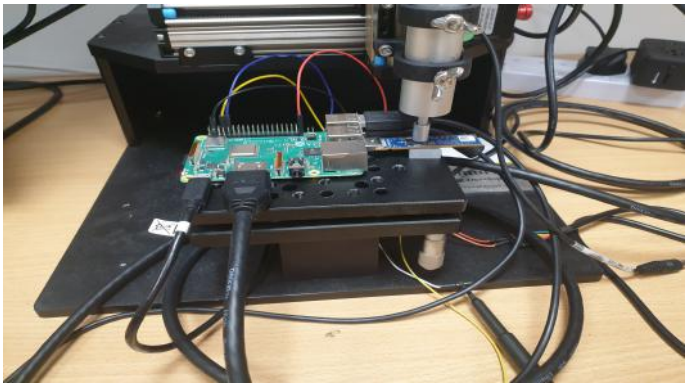


FIGURE 5.7: An example of trace collected after running the neural network.

Measurements or traces are taken for each input to the model. Each trace collected span the entire time of the execution of the network on the NCS. This means that for a given trace, the number of data points collected is equal to the product of



(A) Complete Hardware Setup



(B) Setup of Raspberry Pi, Probe and NCS2.

FIGURE 5.8: Experimental Setup for performing side channel attack.

execution time and sampling frequency. An example of a collected trace can be seen in Figure 5.7. To better help sampling, the outer shielding and heatsink of the chip was removed, exposing the chip. A fan was used to keep the NCS cool. The exposed Myriad X VPU can be seen in Figure 5.6. An important factor during the measurement is the placement of the probe during the measurement. Since the internal structure of the Myriad X VPU is not open source, this placement was decided through trial and error. The complete setup can be seen in Figure 5.8a.

Another crucial part of the setup was synchronizing the execution of the network

with the collection of traces. This was achieved using a Raspberry Pi. The Raspberry Pi was used to send an input data sample to the NCS. It also sent a signal to the oscilloscope to start the data collection process. Once the network output was received from the NCS, the Pi sent another signal to stop the data collection. These connections, as well as the probe placement, can be seen in Figure 5.8b

5.2.3 Attack Procedure

A simple fully connected neural network with binary weights: -1, 1 was used in this work. The neural network had three input neurons and three neurons in each of its 10 hidden layers. The output layer had 2 neurons. However, for the purpose of this work, only the first two hidden layers were attacked. The weights of the first two layers can be seen in Figure 5.9. It was seen that deeper networks generated a more precise and more prolonged trace which made the attack process easier.

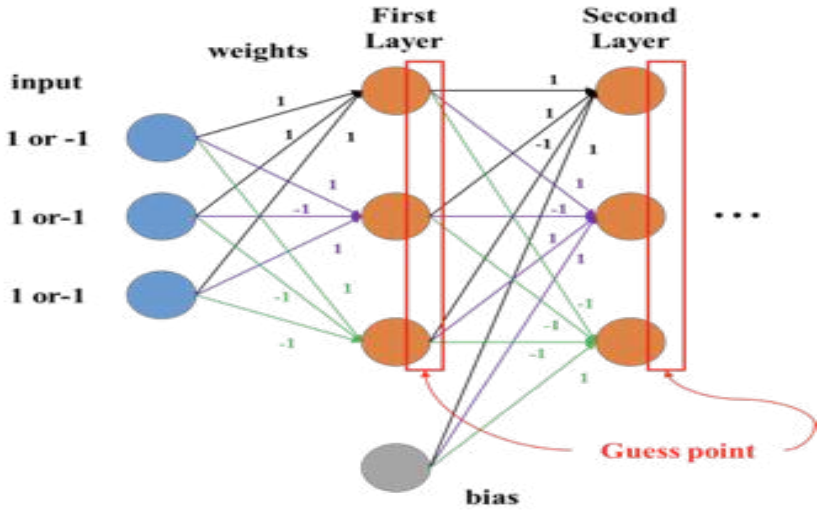


FIGURE 5.9: Weights of the first two layers and the guess points being attacked.

Traces were collected for different inputs to the network. To compare the differences between the leakages for each input, multiple traces were generated and collected.

The leakage model being exploited in the NCS is the hamming weight model. The power consumption of the device and the resultant EM leakage is proportional to the data being loaded and the number of bits equal to 1. This means that the leakage associated with loading a data x is:

$$HW(x) = \sum_{i=1}^{n} x_i, \tag{5.3}$$

where x_i is the i^{th} bit of x [16]. These leakages are generated due to dynamic current flows generated by the charging and discharging of parasitic capacitance [8]. This is characterised by:

$$I_{dyn,i}(t) = \frac{1}{2} C_i \cdot D_i(t) \cdot V_{DD} \cdot f_{clk} \tag{5.4}$$

where $I_{dyn,i}(t)$ denotes transient current flow, C_i is the capacitance of net i, $D_i(t)$ is its transient transition rate, V_{DD} is the voltage supply, and f_{clk} is the clock frequency [111]. Furthermore, $D_i(t)$ is related to the characteristics and type of computation happening in each layer of the network. Hence, each layer's $D_i(t)$ is proportional to its layer parameters which is what is being attacked. To conduct the attack, a divide and conquer approach is taken where each neuron in each layer is attacked separately.

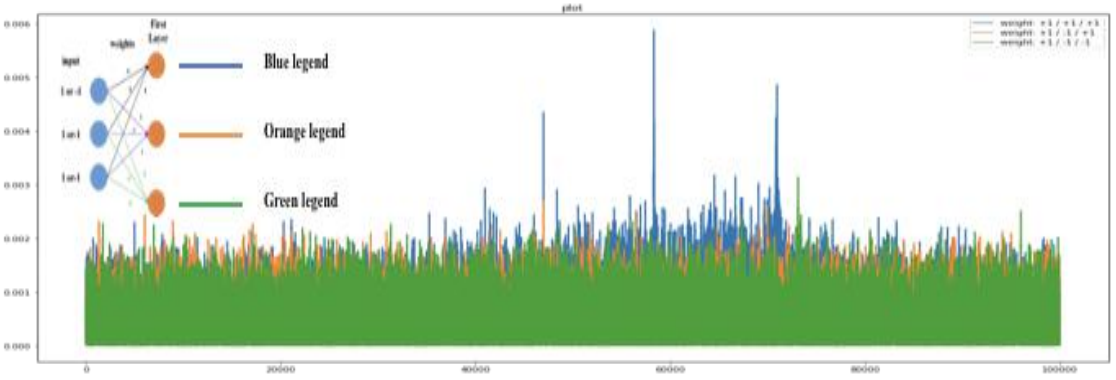


FIGURE 5.10: Traces from the first layer of the neural network. Different colors represent the different neurons in the first layer.

The next step was to find the part of the trace responsible for execution of a specific neuron. To do this, traces for a particular input was compared with traces for other inputs. Since we were using a BNN, we were trying to find peaks in the trace generated by the MSB of the output of each neuron. To generate peaks, we were feeding inputs to the network that would result in a +1 output or activation at the neuron being attacked. Secondly, we also fed inputs to the network that would result in a −1 output or activation at the neuron being attacked, but +!

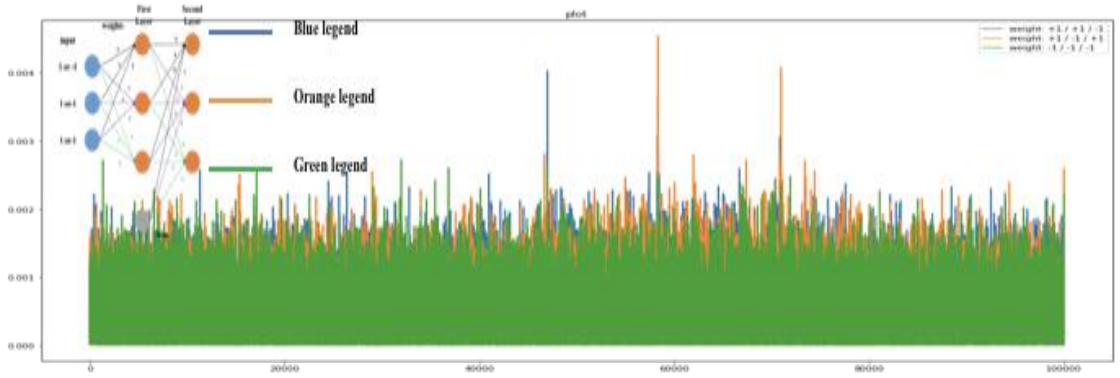


FIGURE 5.11: Traces from the second layer of the neural network. Different colors represent the different neurons in the second layer.

outputs at other neurons. By finding the parts of the trace that is different, it was possible to understand when a particular calculation was taking place.

Since there were very few leakages, approximately one million traces were collected and averaged to identify the traces corresponding to an individual neuron calculation. Collecting multiple traces also helped to reduce the effect of jitter on the probe reading. Once a sufficient number of traces were collected for an input, the process was repeated for a different input. By subtracting the average traces of both inputs, it is possible to see whether there is any significant difference in leakages. Since the neuron with an activation of +1 will consume more power, it will also generate a larger EM leakage, thus helping us perform a difference of means attack.

5.2.4 Results

After collecting the different traces, it was possible to isolate the traces corresponding to the neurons in the first hidden layer. In Figure 5.10, the blue lines represent the traces generated when calculations for the first neuron in the first layer was being done. Similarly, the orange and green lines are the traces for the second and third neurons. While the traces for the third neuron is also easily visible, the second neuron traces were not isolated. The reason for this is still unknown and needs to be investigated further.

Furthermore, in this work, a layer by layer approach to isolating the calculations is being done. Figure 5.11 shows the traces for the second layer. In this case, the

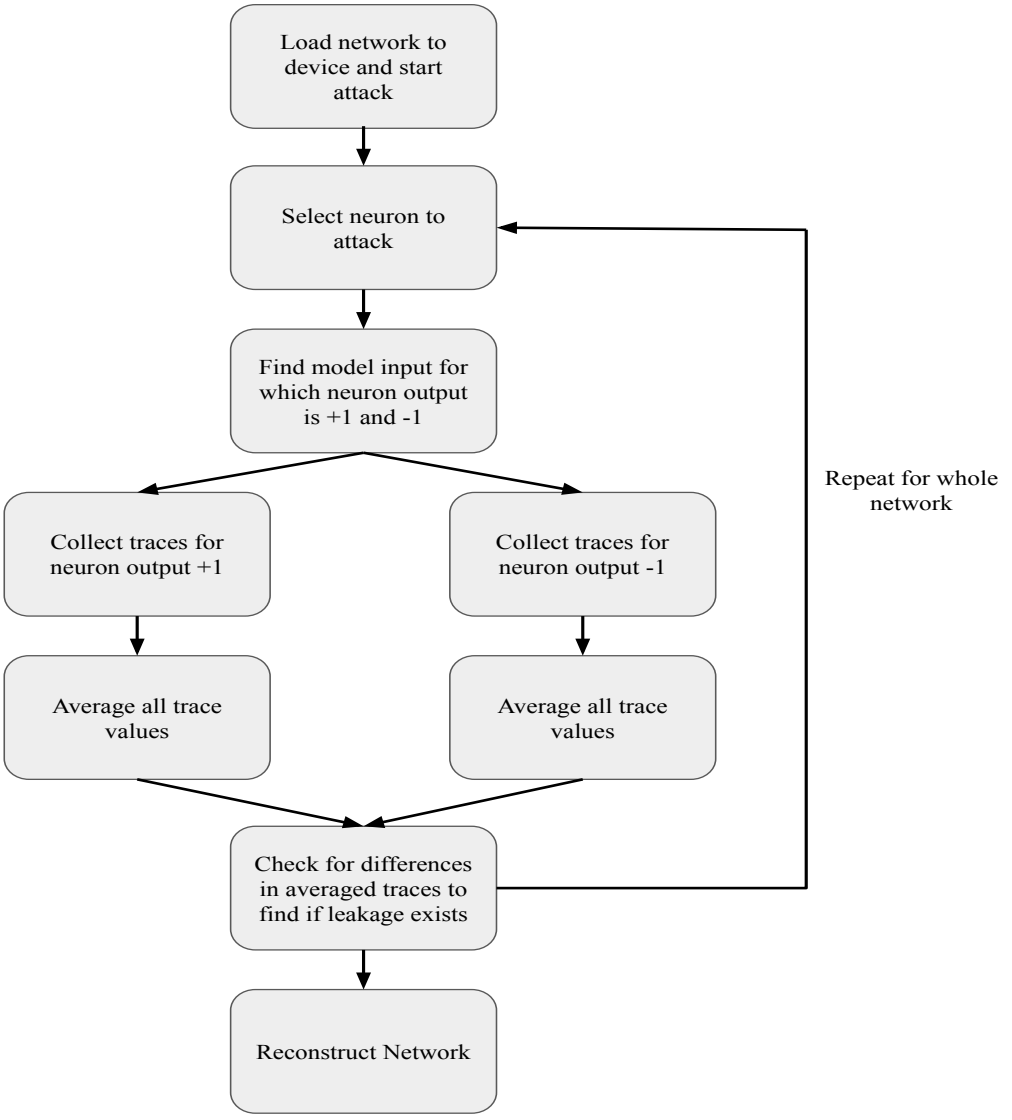


FIGURE 5.12: Methodology to reconstruct neural network.

first and second neuron traces were more prominent than the third neuron. For now, only the first and second layer traces were isolated. In the future, the rest of the network will be attacked.

5.2.5 Challenges and Mitigation

One of the challenges of this work is the amount of experiments that needs to be run to generate significant traces in the output. For large networks with multiple neurons or convolutional layers, running multiple experiments to recover only each

weight will make the attack challenging and time-consuming. However, most models deployed at the edge are small, with fewer layers and neurons, and algorithms like pruning and quantization is applied to reduce the number of parameters in the model. This makes timing side-channel attacks a serious attack vector for edge deployed models.

One way to mitigate this attack is to execute the different neurons in each layer non-sequentially. Shuffling the order of execution within each layer would prevent the attacker from knowing which neuron weight they are recovering. Another mitigation strategy is to introduce random delay in the execution of each neuron or execute each operation in constant time. Since the timing SCA depends on fluctuating timings, implementing constant time exponentiation can help mitigate this attack [112].

5.3 Conclusion

In this chapter two side-channel attacks on the NCS were demonstrated. In this first attack, we show that it is possible to know the type of model loaded on to a NCS by seeing the execution time for a given input irrespective of the host device. After that we show that using an electromagnetic side-channel it is possible to see the execution of individual neurons in a BNN running inside a NCS. Finally some challenges in implementing these attacks and their mitigation is discussed.

In the next chapter, some future work to improve the results of the different attacks is discussed.

Chapter 6

Summary and Future Work

6.1 Summary

In line with the objectives, the following goals have been met:

1. Conversion of events captured by a NVS into frames by aggregating events has been done. This was done by aggregating all events over a period of time. By aggregating events of different polarities separately, frames with multiple channels and bits could be created.

2. Proposal of RoIs containing vehicles from the frames. A histogram based proposal was first used to get RoIs. However, this resulted in fragmented objects, with different parts of the same object being proposed as different RoIs. CCA based regions proposal solved this problem significantly.

3. Using depthwise separable convolutions, it was possible to classify the objects in the proposed RoIs with reasonable accuracy while meeting computation requirements. An architecture search was done by starting with a LeNet-5 based model. The TinyNet architecture was found to have the best tradeoff between accuracy, memory requirements and model complexity (Figure 3.6).

4. After that, DFP quantization was used to further reduce the computes required to execute the TinyNet network. Following the DFP quantization, the network was fine-tuned to recover the lost accuracy. It was shown that

it is possible to quantize the network with less than a 2% drop in accuracy (Table 3.4).

5. Cold Boot attack was used to attack a model running on an NCS. From the RAM dump of the attack, the model architecture and weight files were recovered. A method to fix errors in architecture and weight files were proposed. While the architecture file errors can be easily fixed, fixing the errors in the weights are more difficult.

6. To fix the erroneous weights, a training method based on knowledge distillation was proposed. We show that using knowledge distillation, it is possible to regain the accuracy of the recovered model within a margin of 10% of the original model accuracy. We also show that it is possible, but difficult to recover the original weights of the model.

7. Using timing side channel, we show that it is possible to recover the kind of pretrained model running on a Neural Compute Stick with nearly 100% accuracy.

8. EM side channel was used to attack neural networks running on a Neural Compute Stick. To help simplify the attack process, a BNN model was loaded on to the neural compute stick to be attacked. The model was then attacked layer by layer. We show that it is possible to isolate the computation for the first two layers (Figure 5.10 and Figure 5.11). However, some of the neurons in those layers do not generate traces.

6.2 Future Work

1. Methods to secure neural network computation from cold boot, timing and electromagnetic side channel attacks need to be studied. Further, how these protection scheme affect the accuracy and inference time of the network can be explored. If the execution time is very high, then it will not be feasible to apply them to an edge computing setting.

2. Try to find out why certain neurons do not generate any traces when using EM side channel.

Other ways of extracting weight and architecture details using electromagnetic side-channel are also being explored. For instance, the execution timing of different layers in the network can be used to extract information about the type and number of layers in a model [110]. Using this information, an attacker can train a similar model as long as they have access to the training data. It is also possible to train a model with similar accuracy even with a small subset of unlabelled data [50].

3. Further reduction in the computational complexity of neural networks using activation pruning and hashing of neural network weights [113]. Most works in hashing prune weights based on their magnitude. The smaller the weight, the more likely it is to get pruned. The reason for this is because a small weight will cause a small change in the activation and hence have an overall smaller effect on the final output of the network. However, a small weight could have a significant effect on the output if the activation flowing through it is comparatively larger. Similarly, a small activation flowing through a large weight will have a smaller effect on the output prediction. Therefore, we hypothesize, that it is better to prune weights based not on their magnitude, but based on the similarity between the weight and activation.

   Locality Sensitve Hashing (LSH) for maximum inner product search (MIPS) [114] is one of the proposed methods to achieve this and was done in [113]. The authors propose to use LSH has tables for each layer in the network. Hashtables are created using the weights during the training phase. Furthermore, during the training phase, the layer's inputs are hashed and queried for the top k% of active set neurons. The rest of the neurons are dropped out and only the active set of neurons are updated during the training phase. This leads to sparse gradient updates which also help in faster and more parallelizable asynchronous training [115]. During testing, this method reduces the number of multiplications needed to execute the network.

4. In this regard, we also attempt to find the effect of hashing of neural network weights and choosing only top k% of activated weights vs. normal accuracy. By dropping more weights, the complexity of the network can be decreased, however, the accuracy will also be affected. The 'k' value in turn can act like a knob that can be used to finetune the network for the desired performance.

5. While the 'k' value can act as a good knob, we also attempt to find if other parameterizable methods to tradeoff energy vs accuracy for ANN exist. In addition, we try to find if these knobs can perform as well as the conversion of ANN to Spiking Neural Networks (SNN).

   The advantage of SNN over ANN is that they have an inherent parameterizable knob: time. This is why they have been very popular as a method to reduce the latency as well as the computational complexity of deep neural networks [116]. Furthermore, SNNs can be queried for an output after the first spike. This is unlike ANNs where the whole network needs to be run first. However, training of SNNs is difficult and while significant progress has been made, larger networks with equivalent accuracy as ANNs have not been achieved. This is why work has been done on converting standard ANNs to their SNN equivalents [117].

   To this end, we tried to prune activations from neural networks instead of weights based on their magnitude. During the training process, the model was trained and the maximum activation for each neuron was recorded. During the inference process, for each input, the activation for each neuron was divided by its maximum activation for each layer. Doing this, we were able to scale neurons based on its percentage of total activation. Following this, for each layer, only the top k% of activations were allowed to pass through.

   Doing this, without any finetuning, yielded decent results and we were able to prune up to 20% of the activations without significant loss in accuracy. To account for this reduction in activations going to the next layer, all the activations were multiplied by the neuron drop percentage of that layer. This pruning was done on a 3 layer ANN trained on MNIST. Finetuning was done by performing the pruning during the training process and only training those neurons that were activated (similar to hashing [113]). During this finetuning process, the pruning percentage, or k value, could be used as a parameter. Using this technique, the same network could be pruned by up to 80% while still maintaining a 92% accuracy. We were also able to prune LeNet-5 by up to 30% while still maintaining an accuracy greater than 90%.

6. This yielded good results on pruning and finding a parameter that can be tuned to improve the accuracy of the network for a certain pruning percentage. This pruning was done in a spatial manner. In addition to spatial

pruning, we try to find out if the neural networks can be pruned spatially first, then converted to SNN and pruned temporally, while still maintaining accuracy. To do this, we are using the SNN ToolBox to convert our pruned neural network to an SNN [118].

7. In this work, we consider that the cold boot attack was done once and the attacker had to recover all the details from a single RAM dump. However, if multiple attacks can be performed, then the recovery can be more simple. By using maximum voting on the bits of three RAM dumps, it can be possible to both identify and fix erroneous bits using three RAM dumps. With two RAM dumps, it can be possible to identify erroneous bits and potentially fix them using simple techniques. However, this will be more difficult to do in case there is a bias in the RAM dump causing a large number of bits to be erroneous in a single part of the dump.

Chapter 7

List of Author's Publications

D. Singla, **S. Chatterjee**, L. Ramapantulu, A. Ussa, B. Ramesh, A. Basu, "HyNNA: Improved Performance for Neuromorphic Vision Sensor based Surveillance using Hybrid Neural Network Architecture," *2020 IEEE International Symposium on Circuits and Systems Conference (ISCAS)*, Sevilla, Spain 2020.

YS Won, "Time to Leak: Cross-Device Timing Attack On Edge Deep Learning Accelerator," *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, Jeju Shinhwa World, Republic of Korea 2021.

S. Chatterjee, YS Won, D. Jap, S. Bhasin, A. Basu, "DeepFreeze: Cold Boot Attack and Model Recovery on Commercial EdgeML Device", **submitted to** *2021 Design Automation Conference (DAC)*, San Francisco, United States of America 2021.

List of Author's Publications

Conference Proceedings

1. D. Singla, **S. Chatterjee**, L. Ramapantulu, A. Ussa, B. Ramesh, A. Basu, "HyNNA: Improved Performance for Neuromorphic Vision Sensor based Surveillance using Hybrid Neural Network Architecture," *2020 IEEE International Symposium on Circuits and Systems Conference (ISCAS)*, Sevilla, Spain 2020.

2. YS Won, **S. Chatterjee**, D. Jap, S. Bhasin, A. Basu, "Time to Leak: Cross-Device Timing Attack On Edge Deep Learning Accelerator," *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, Jeju Shinhwa World, Republic of Korea 2021.

Submitted Papers

1. **S. Chatterjee**, YS Won, D. Jap, S. Bhasin, A. Basu, "DeepFreeze: Cold Boot Attack and Model Recovery on Commercial EdgeML Device", **submitted to** *2021 Design Automation Conference (DAC)*, San Francisco, USA 2021.

Bibliography

[1] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016. 7, 49

[2] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. 7, 12, 24, 27, 28

[3] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 7, 26

[4] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018. 7, 20, 24

[5] Peter U Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. ieee, 2015. 7, 26

[6] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience*, 11: 682, 2017. 7, 26

[7] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*, 2016. 7, 12, 28, 29, 43, 52

[8] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Maskednet: A pathway for secure inference against power side-channel attacks. *arXiv preprint arXiv:1910.13063*, 2019. 7, 21, 39, 80

[9] YS Won and et al. Practical cold boot attack on iot device – case study on raspberry pi. In *2020 IEEE 27th International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*. 7, 12, 13, 41, 56, 58, 67, 68

[10] State of IoT. `https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/`. Accessed: 2020-07-16. 12, 18

[11] AI in Numbers. `https://www.cbinsights.com/research/report/ai-in-numbers-q2-2019/`. Accessed: 2020-07-16. 12, 19

[12] Future of Augmented Intelligence in Business. `https://techwireasia.com/2019/08/gartner-forecasts-the-future-of-augmented-intelligence-in-business/`. Accessed: 2020-07-16. 12, 19

[13] Edge Computing Use Cases. `https://innovationatwork.ieee.org/real-life-edge-computing-use-cases/`, . Accessed: 2020-07-16. 12, 20

[14] EdgeTPU Accelerator Image. `https://coral.ai/products/accelerator`, . Accessed: 2020-07-16. 12, 32, 33

[15] EdgeTPU DevBoard Image. `https://coral.ai/products/dev-board`, . Accessed: 2020-07-16. 12, 32

[16] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 515–532, 2019. 12, 13, 36, 37, 38, 40, 62, 73, 74, 75, 76, 80

[17] Deepak Singla, Soham Chatterjee, Lavanya Ramapantulu, Andres Ussa, Bharath Ramesh, and Arindam Basu. Hynna: Improved performance for neuromorphic vision sensor based surveillance using hybrid neural network architecture. *arXiv preprint arXiv:2003.08603*, 2020. 12, 13, 34, 43, 52

[18] Stereo Labs. `https://www.stereolabs.com/zed/`. Accessed: 2020-03-27. 12, 45

[19] Christian Brandli, Raphael Berner, Minhao Yang, Shih-Chii Liu, and Tobi Delbruck. A 240× 180 130 db 3 μs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10):2333–2341, 2014. 12, 43, 45, 54

[20] Deepak Singla, Vivek Mohan, Tarun Pulluri, Andres Ussa, Bharath Ramesh, and Arindam Basu. Ebbinnot: A hardware efficient hybrid event-frame tracker for stationary neuromorphic vision sensors. *arXiv preprint arXiv:2006.00422*, 2020. 12, 35, 45, 46

[21] Jyotibdha Acharya, Andres Ussa Caycedo, Vandana Reddy Padala, Rishi Raj Singh Sidhu, Garrick Orchard, Bharath Ramesh, and Arindam Basu. Ebbiot: A low-complexity tracking algorithm for surveillance in iovt using stationary neuromorphic vision sensors. In *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pages 318–323. IEEE, 2019. 13, 48

[22] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009. 13, 40, 57, 59, 60

[23] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487*, 2018. 13, 35, 61, 63, 73

[24] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015. 20, 26

[25] Han Xu Yao Ma Hao-Chen, Liu Debayan Deb, Hui Liu Ji-Liang Tang Anil, and K Jain. Adversarial attacks and defenses in images, graphs and text: A review. *International Journal of Automation and Computing*, 17(2):151–178, 2020. 21, 35

[26] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018. 22, 28

[27] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017. 22, 24, 26, 49

[28] Neural Compute Stick 2. `https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html`. Accessed: 2020-07-16. 22, 31, 69, 70

[29] EdgeTPU. `https://coral.ai/`, . Accessed: 2020-07-16. 22, 31

[30] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008. 22

[31] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990. 26

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 26

[33] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5687–5695, 2017. 26

[34] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 26, 27

[35] Richard Dorrance, Fengbo Ren, and Dejan Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 161–170, 2014. 27

[36] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3): 243–254, 2016. 27

[37] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45 (2):27–40, 2017. 27

[38] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016. 27

[39] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):1–18, 2017. 27

[40] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2): 548–560, 2017. 27

[41] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017. 27

[42] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016. 27

[43] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016. 27

[44] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015. 27

[45] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017. 27

[46] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016. 27

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 28

[48] Intel MKL-DNN. `https://oneapi-src.github.io/oneDNN/ex_int8_simplenet.html`. Accessed: 2020-07-16. 28

[49] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541, 2006. 29

[50] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 29, 30, 86

[51] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013. 30

[52] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014. 30

[53] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10687–10698, 2020. 30

[54] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and fisher pruning. *arXiv preprint arXiv:1801.05787*, 2018. 30

[55] Anubhav Ashok, Nicholas Rhinehart, Fares Beainy, and Kris M Kitani. N2n learning: Network to network compression via policy gradient reinforcement learning. *arXiv preprint arXiv:1709.06030*, 2017. 30

[56] Hokchhay Tann, Soheil Hashemi, R Iris Bahar, and Sherief Reda. Hardware-software codesign of accurate, multiplier-free deep neural networks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017. 30

[57] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018. 30

[58] Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. *arXiv preprint arXiv:1711.05852*, 2017. 30

[59] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 236–241. IEEE, 2016. 31

[60] Kota Ando, Kodai Ueyoshi, Kentaro Orimo, Haruyoshi Yonekawa, Shimpei Sato, Hiroki Nakahara, Masayuki Ikebe, Tetsuya Asai, Shinya Takamaeda-Yamazaki, Tadahiro Kuroda, et al. Brein memory: A 13-layer 4.2 k neuron/0.8 m synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm cmos. In *2017 Symposium on VLSI Circuits*, pages C24–C25. IEEE, 2017. 31

[61] Intel Movidius Myriad X VPU. `https://www.intel.com/content/www/us/en/products/processors/movidius-vpu/movidius-myriad-x.html`. Accessed: 2020-07-16. 31

[62] Intel OpenVINO Toolkit. `https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html`, . Accessed: 2020-07-16. 31, 57

[63] EdgeTPU Dev Board. `https://coral.ai/docs/dev-board/datasheet/`, . Accessed: 2020-07-16. 32, 69

[64] EdgeTPU Accelerator. `https://coral.ai/docs/accelerator/datasheet/`, . Accessed: 2020-07-16. 33, 69

[65] TensorFlow Lite for Mobile and Edge Devices. `https://www.tensorflow.org/lite`. Accessed: 2020-07-16. 33

[66] Olivier Barnich and Marc Van Droogenbroeck. Vibe: A universal background subtraction algorithm for video sequences. *IEEE Transactions on Image processing*, 20(6):1709–1724, 2010. 33

[67] Massimo Piccardi. Background subtraction techniques: a review. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, volume 4, pages 3099–3104. IEEE, 2004. 34

[68] Christoph Posch, Teresa Serrano-Gotarredona, Bernabe Linares-Barranco, and Tobi Delbruck. Retinomorphic event-based vision sensors: bioinspired cameras with spiking output. *Proceedings of the IEEE*, 102(10):1470–1484, 2014. 34

[69] Arindam Basu, Jyotibdha Acharya, Tanay Karnik, Huichu Liu, Hai Li, Jae-Sun Seo, and Chang Song. Low-power, adaptive neuromorphic systems: Recent progress and future directions. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):6–27, 2018. 34

[70] Guillermo Gallego, Tobi Delbruck, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew Davison, Jörg Conradt, Kostas Daniilidis, et al. Event-based vision: A survey. *arXiv preprint arXiv:1904.08405*, 2019. 34

[71] Arren Glover and Chiara Bartolozzi. Robust visual tracking with a freely-moving event camera. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3769–3776. IEEE, 2017. 34

[72] Xavier Lagorce, Cédric Meyer, Sio-Hoi Ieng, David Filliat, and Ryad Benosman. Asynchronous event-based multikernel algorithm for high-speed visual features tracking. *IEEE transactions on neural networks and learning systems*, 26(8):1710–1720, 2014. 34

[73] Bharath Ramesh, Hong Yang, Garrick Michael Orchard, Ngoc Anh Le Thi, Shihao Zhang, and Cheng Xiang. Dart: distribution aware retinal transform for event-based cameras. *IEEE transactions on pattern analysis and machine intelligence*, 2019. 34

[74] Gereon Hinz, Guang Chen, Muhammad Aafaque, Florian Röhrbein, Jörg Conradt, Zhenshan Bing, Zhongnan Qu, Walter Stechele, and Alois Knoll. Online multi-object tracking-by-clustering for intelligent transportation system with neuromorphic vision sensor. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 142–154. Springer, 2017. 34

[75] Massimiliano Iacono, Stefan Weber, Arren Glover, and Chiara Bartolozzi. Towards event-driven object detection with off-the-shelf deep learning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018. 34

[76] Zhenjiang Ni, Sio-Hoi Ieng, Christoph Posch, Stéphane Régnier, and Ryad Benosman. Visual tracking using neuromorphic asynchronous event-based cameras. *Neural computation*, 27(4):925–953, 2015. 34

[77] Andres Ussa, Luca Della Vedova, Vandana Reddy Padala, Deepak Singla, Jyotibdha Acharya, Charles Zhang Lei, Garrick Orchard, Arindam Basu, and Bharath Ramesh. A low-power end-to-end hybrid neuromorphic framework for surveillance applications. *arXiv preprint arXiv:1910.09806*, 2019. 34, 44, 47

[78] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a

scalable communication network and interface. *Science*, 345(6197):668–673, 2014. 34

[79] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 587–601, 2017. 35

[80] RISCURE. `https://www.riscure.com/blog/automated-neural-network-construction-genetic-algorithm/`. Accessed: 2020-07-16. 35

[81] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 17–32, 2014. 35

[82] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414. IEEE, 2018. 35

[83] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016. 35

[84] Wenye Liu, Chip-Hong Chang, Fan Zhang, and Xiaoxuan Lou. Imperceptible misclassification attack on deep learning accelerator by glitch injection. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. 35

[85] Mihailo Isakov, Vijay Gadepally, Karen M Gettings, and Michel A Kinsy. Survey of attacks and defenses on edge-deployed neural networks. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2019. 35

[86] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020. 35, 58

[87] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999. 36

[88] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. Side-channel power analysis of a gpu aes implementation. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 281–288. IEEE, 2015. 37

[89] Shivam Bhasin, Sylvain Guilley, Annelie Heuser, and Jean-Luc Danger. From cryptography to hardware: analyzing and protecting embedded xilinx bram for cryptographic applications. *Journal of Cryptographic Engineering*, 3(4): 213–225, 2013. 37

[90] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 740–757. Springer, 2012. 39

[91] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 142–159. Springer, 2013. 39

[92] Jean-Sébastien Coron and Louis Goubin. On boolean and arithmetic masking against differential power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 231–237. Springer, 2000. 39

[93] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. 39

[94] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720*, 2018. 40

[95] Gaofeng Dong, Ping Wang, Ping Chen, Ruizhe Gu, and Honggang Hu. Floating-point multiplication timing attack on deep neural network. In *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 155–161. IEEE, 2019. 40

[96] Muller Tilo, Spreitzenbarth Michael, and Felix C Freiling. Frost: forensic recovery of scrambled telephones. In *Proceedings of the International Conference on Applied Cryptography and Network Security*, pages 373–388, 2014. 40

[97] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 313–324. IEEE, 2017. 40

[98] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a

scalable communication network and interface. *Science*, 345(6197):668–673, 2014. 43

[99] Menghan Guo, Jing Huang, and Shoushun Chen. Live demonstration: A 768× 640 pixels 200meps dynamic vision sensor. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–1. IEEE, 2017. 43

[100] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. 47

[101] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017. 47

[102] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 47

[103] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017. 47

[104] Robert Walczyk, Alistair Armitage, and T David Binnie. Comparative study on connected component labeling algorithms for embedded video processing systems. *IPCV*, 10:176, 2010. 47

[105] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86 (11):2278–2324, 1998. 49

[106] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 50

[107] Qianli Liao, Joel Z Leibo, and Tomaso Poggio. How important is weight symmetry in backpropagation? In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016. 53

[108] Encrypted Models with OpenVINO. . `https://docs.openvinotoolkit.org/latest/openvino_docs_IE_DG_protecting_model_guide.html`. 57

[109] Cheng Gongye, Yunsi Fei, and Thomas Wahl. Reverse-engineering deep neural networks using floating-point timing side-channels. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. 70

[110] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. Deepem: Deep neural networks model recovery through em side-channel information leakage. 76, 86

[111] Jason Helge Anderson and Farid N Najm. Power estimation techniques for fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10):1015–1027, 2004. 80

[112] Gaël Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 293–301. Springer, 2000. 83

[113] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454, 2017. 86, 87

[114] Anshumali Shrivastava and Ping Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Advances in neural information processing systems*, pages 2321–2329, 2014. 86

[115] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011. 86

[116] Davide Zambrano and Sander M Bohte. Fast and efficient asynchronous neural computation with adapting spiking neural networks. *arXiv preprint arXiv:1609.02053*, 2016. 87

[117] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience*, 11: 682, 2017. 87

[118] SNN ToolBox. `https://snntoolbox.readthedocs.io/en/latest/index.html`. Accessed: 2020-07-16. 88