TOWARDS IN-NETWORK IMAGE CLASSIFICATION FOR LATENCY-CRITICAL IOT APPLICATIONS

by

HISHAM SIDDIQUE

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
November 2021

Table of Contents

List of Tables

List of Figures

Abstract

As the demand for latency-critical applications (intelligent transport systems, medical imaging, surveillance, AR/VR) continues to increase, tasks such as computer vision, which are imperative to these applications, require expedited processing to keep up with this latency requirement. However, the distant cloud servers and low-powered IoT devices could not offer that latency, which then force the realm of edge computing. It exists as an intersection between offering computational power at a considerably lower latency access point. Edge servers have less computing resources compared to the distant cloud, which initiated a plethora of works on distributing load among IoT devices and edge servers. This thesis introduces a novel in-network computing paradigm by leveraging programmable routers sit between IoT devices and edge servers. Specifically, we propose a novel in-network framework, *NetPixel* for learning-based operations, e.g., image classification, to support the latency demand of the aforementioned IoT applications. The framework incorporates two ML and DL-based classifiers: decision tree and convolutional neural network. While the former can interpret the model better, the latter is the standard image classifier. We implement a prototype of NetPixel over programmable software switch Bmv2 and perform an extensive evaluation using four standard datasets. The results reveal that NetPixel offers a competitive performance compared to traditional server-based solutions while can process network traffic at high speed (Tbps) to support latency-critical applications.

List of Abbreviations Used

AI	Artificial Intelligence
AR	Augmented Reality
ASIC	Application-Specific Integrated Circuit
CART	Classification And Regression Trees
CCTV	Closed Circuit Television
CLI	Command Line Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DT	Decision Tree
EC	Edge Computing
FPGA	Field Programmable Gate Array
ID3	Interactive Dichotomizer 3
IG	Information Gain
IoT	Internet of Things
ITS	Intelligent Transport Systems
ML	Machine Learning
MSE	Mean Squared Error
NIC	Network Interface Card
NN	Neural Network
P4	Programming Protocol-Independent Packet Processors
PND	Programmable Networking Device
PISA	Protocol Independent Switch Architecture
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
ReLU	Rectified Linear Unit
SDN	Software Defined Networks
SIMD	Single Instruction Multiple Data

SVM	Support Vector Machine
TCP/IP	Transmission Control Protocol/Internet Protocol
RGB	Red, Green, Blue
VR	Virtual Reality

Acknowledgements

First off, I would like to extend my sincerest gratitude to my supervisor Dr. Israat Haque, who has watched over me tirelessly over the course of my Master's degree. She has always been there to keep me on the straight path and has nurtured my growth as a researcher. Her motivation kept me going through the hurdles of academia and research and it was an utmost honor to have her as both a supervisor and a guardian throughout my time at Dalhousie.

Next, I would be remiss if I did not acknowledge the contribution of Dr. Miguel Neves, post-doctoral fellow at the Programmable and Intelligent Network (PINet) research lab, Dalhousie. He has helped me tremendously throughout my research endeavors, and was ever-present to extend a helping hand and ensure I remain motivated enough to stay the course. My fellow colleague, Carson Kuzniar was also there for any friendly advice and help that I needed, alongside my other fellow colleagues at the research lab.

I would also like to mention my partner, who, despite our differences, was always present to support me and keep me going. She remains a constant motivation for my current and future endeavors.

Lastly, I would love to thank my Mom, Dad and my siblings, without whom I would not be here. Their support and love is a debt I cannot repay, and it is one of life's greatest joys to make them proud of my efforts throughout this work and my Master's degree.

Chapter 1

Introduction

With the ever-changing landscape of technology, connectivity has become one of the most sought after paradigms on a broad scale, whether that be between devices, between humans, and even both. Exchange of data and information through the Internet has seen rapid changes within the past decade, particularly the landscape of Internet of Things (IoT) applications have continued to expand into a myriad of use cases, with progression in device capabilities as well as network infrastructure [7]. These applications include smart cities [14], smart homes [8], autonomous factories [89], intelligent transport systems [58] and remote healthcare [25]. By extension, numerous techniques, algorithms and protocols have been introduced, which cater to the demands of these applications, whilst conforming to the technologies present, in terms of hardware compatibility, capability, and scalability. Many of these applications integrate an amalgamation of these protocols and technologies.

With the steady rise of IoT integration, complemented by the benefits of 5G connectivity, latency-critical and safety-critical applications such as augmented reality [72], virtual reality [98], and autonomous vehicles [3] have seen a resurgence, taking advantage of the substantial increases in bandwidth and proportional decreases in end-to-end latencies. These applications were previously under the umbrella of the cloud-computing paradigm, usually consisting of low-powered devices at the network edge offloading processing to distant cloud servers. Because a lot of the data collecting devices are in the form of sensors, ranging from simple data to images and videos [31, 39–44, 50, 78], most of their intelligent capabilities had to have been performed elsewhere, and this role has been assumed by the cloud [27, 35]. However, the distant location of cloud servers would consequently cause end-to-end latency would suffer notably [6].

Edge Computing (EC) and Software Defined Networking (SDN) have emerged as two paradigms which aim to alleviate this detriment in latency. EC entails the shift

of computational tasks, which were previously delegated to the central cloud server in the network, to devices closer to the edge of the network, and by extension, closer to the users. This circumvents the need for transmission of data over long distances, as well as additional load on the network backbone [81, 85, 86]. SDN offers a flexibility of network devices that allow for network administrators to better supervise network [38], as well as introduce additional computational tasks to networking devices, e.g, routers or switches. This allows devices at the edge such as switches to perform tasks for a wide range of applications [12, 65], notably those involving latency-critical constraints, due to their programmability.

Programmable switches, such as the Intel Barefoot Tofino [4], have been established as one of the pioneers of data plane programmability for data centers applications, by providing the flexibility of network programmability alongside high-speed traffic processing and computing ability. This, in turn, has opened the opportunity to perform in-network computation on these switches, with the added beneifts of saving in terms of latency with their proximity to the user. In particular, switches such as the Tofino can provide throughput at an order that is multiplicatively higher than a traditional edge server, making them suitable for the high-speed computational offloading that is demanded by latency-critical applications at the edge. An example of this may include the operation of autonomous vehicles, which need to make split-second decisions such as braking in the event of a pedestrian being detected in front of the vehicle. This procedure is an example of the important paradigms required in the latency-critical application field: image classification (the detection of the pedestrian) and, more broadly, machine learning (the training and inference required to perform the aforementioned detection).

This symbiosis of network programmabilty and image classification based on machine learning, is the foundation for the proposed thesis, which serves to establish the feasibility and implementation of image classification as a latency-critical task, in areas such as intelligent transport systems [58, 60, 96], AR/VR applications [70, 88] as well as, surveillance [50]. For the initial part of our work, we implement our image classification system in a programmable switch using a decision tree-based classifier. This is compared to a baseline server implementation and its performance noted. This system performed to within an 8% performance discrepancy of the baseline system,

accounted for by the architectural constraints of the platform and device on which it is implemented. Following this, we shift our classifier to a neural-network based system, in accordance with the state-of-the-art of image classifiers and explore its feasibility under the same environment.

1.1 Motivation

As the impact of IoT has become more and more pervasive, devices have become smarter. This has occurred through an vast improvements in interchange of information, as well as artificial intelligence, through which these systems may learn and better adapt to their area of application. One such area that has seen increased dissemination, is that of latency-critical and safety-critical applications, including augmented reality (AR)/virtual reality (VR), intelligent transport systems, autonomous vehicles, and autonomous surveillance systems, to enumerate a few. Increased bandwidth, more reliable transmission and increase intelligence proliferation throughout the network itself has led to this ascendancy.

Among these applications, a fundamental theme is sensing environment parameters and creating some action and/or decision accordingly. These parameters may range from video feed of an on-board camera in an autonomous car to the heads-up display overlay experienced by a user donning a VR headset. By extension, computer vision (CV) remains the underlying motif in these applications. CV itself constitutes a number of rudimentary tasks including image classification, object recognition, image segmentation, object counting, and so on. Image classification appears to be the crux of these applications, from which all others are derived [3, 19, 95].

One of the imperative constraints of latency-critical applications is, naturally, minimal latency between the sensing of the environment parameters, and the subsequent action based on those parameters. In other words, these applications require minimal delay between the cause and the effect. Cloud computing in its current paradigm, will not be able to serve the low-latency demands of these applications, fundamentally due to distance. Likewise, the sensors themselves present as end-user devices (VR headsets, CCTV cameras) will not provide the requisite computational power to adhere to these tasks.

The above described niche is where edge computing and SDN thrive. By utilizing

the networking devices nearest to the end-user as computational leverage, their flexibility allows for deployment in these low-latency applications. Sensors in autonomous cars, robotic arms in sensor factories, or user displays for augmented reality are some of the few devices which would greatly benefit from this paradigm. Another such real-time low-latency application is video surveillance, which may benefit from image classification at the edge in two ways. The video frames themselves may undergo image classification to recognize objects of interest in the field-of-view [19]. Frames may also undergo a filtering process, wherein frames of interest are filtered and exclusively sent to the cloud for further processing, bypassing the need to incur terabytes of data processing at the cloud by sending all frames [55]. This alleviates traffic within the network as well as load from the cloud server itself.

1.2 Research Objective

Considering the above opportunity, there exists a research trend towards exploring the scope of applications that can be ported to the edge of the network, bringing with it the benefits in terms of latency as well as bandwidth savings on the cloud server and network on the whole. The spectrum of research in edge computing is vast, ranging from computation offloading [21], task scheduling, edge acceleration [47], and content caching [46]. These may involve distributed systems where computation is shared among the end-user devices, networking devices, edge, and cloud servers, or any combination of these. Conversely computation may be localized to a single device. Image classification is one such task that can benefit from its integration into the edge computing paradigm.

To that effect, the objective of our conducted research was to create a novel system for performing image classification on programmable networking devices, which are located at the edge of the network. This system would perform classification in real-time with a competitive accuracy whilst also allowing for scalability. The broad advantage of performing image classification in the above manner is two-fold. Firstly, it allows for latency-critical applications which integrate image classification, such as AR/VR applications and ITS systems, to match required response times. Secondly, it allows low-powered end-devices which would otherwise be incapable of performing image classification intensive tasks, to offload computation at a near distance and

produce subsequent actions accordingly. A number of traditional classifiers exist, which have been implemented on programmable switches, among which decision trees were shown to perform the best in terms of accuracy, based on existing literature [97]. Thus we implemented the decision tree classifier for images in the programmable switches using new domain specific language called P4 [16]. We name the proposed system as *NetPixel* [1].

We implement the NetPixel prototype over virtual programmable switches, BMv2 [16] and compare its performance against traditional server-based system. For the purposes of this evaluation, we tested each implementation over 4 different datasets, (MNIST, [53], ImageNet [26], CalTech 256, and CalTech101 [30, 34]). The primary metric of performance was **accuracy** of classification over these datasets. Then, we move from a traditional classifier approach to a neural-network based approach. Particularly we implement a convolutional neural network (CNN), which is the state-of-the-art classifier for images [71], on the P4 platform, following a binarized approach and evaluate it against the server-based implementations (binarized and non-binarized), using the same performance metric.

1.3 Contribution

The work envisioned throughout this thesis was to create a novel system that would be able to perform image classification on networking devices at the edge of the network, notably programmable switches. Initially classifiers that were able to be implemented on the P4 platform were considered and the best performing classifier, the decision tree, was selected. A number of features that would be extracted from the image and used for the decision tree inference were evaluated and finalized to seven different features, each of which describe a different aspect of the image. The resulting decision tree was then trained and implemented on the P4 platform, evaluating it against its Python-based server implementation. We envision that this implementation on a programmable switch would yield considerable savings in terms of latency, as the switch would be situated at a much closer proximity to the image-capturing devices than an edge server where the Python-based server would be implemented. This would then be a beneficial trade-off over the marginally lower accuracy that NetPixel was found to achieve, which occurred due to the architectural limitations of the

programmable switch platform.

After substantiating the viability of implementing image classification on a programmable switch, we moved on to implementing a convolutional neural network as a classifier, to extend beyond the decision tree. CNNs are the traditional state-of-the-art for image classification and infer to a much greater degree of accuracy than traditional ML classifier models. However, a number of hurdles needed to be overcome to implement them on the P4 platform. We implemented the neural network by decomposing the network architecture into a distributed system, with multiple switches collaborating to produce the final decision label for a given image. Furthermore, the neural network had to be binarized owing to the resource and computational limitations of the switch. This system was then further evaluated against the DT-based classifier as well as the Python-based server implementation.

A summary of the contribution of this thesis is as follows:

- Designing a novel system, NetPixel, for latency-critical applications that performs accurate image classification in real time.
- Designing a protocol that supports the sending of images in a format that enables the above image classifier through feature calculation.
- Evaluating the above system against a baseline server implementation, where it performs within **8%** of the baseline server-based system in terms of **accuracy**, considering all the architectural limitations.
- Designing an improved system as an extension of the previous system, by using convolutional neural network (CNN) as the image classifier
- Designing a distributed system that dissociates the neural network among multiple switches.
- Evaluating the neural network-based system against its binarized and non-binarized counterpart on the server-based implementation, where it performs within **10%** of the equivalent server-based system, in terms of **accuracy**.
- Releasing the above system as an open-source repository for public perusal [1] [2]

1.4 Thesis Outline

The remainder of this thesis is organized structurally as follows: Chapter 2 is divided into two sections, the first of which, Section 2.1, lays the foundational background required to better interpret the work undertaken in this thesis. Next, Section 2.2 presents a number of research works relevant to our work. Chapter 3 delves into the detailed methodology and design of the systems implemented. Here, the problem definition which led to the inception of this thesis is explored in Sections 3.1 and 3.2. This is then followed by the design choices implemented for the DT-based system in Section 3.3. Following these, Section 3.4 presents the evaluation of the decision tree-based system, first in terms of the system parameters and environments, along with the comparative results to server based implementations. Chapter 4 follows the same sequence for the neural network-based system, with the system design in Section 4.1 and subsequent evaluation in Section 4.2. A comprehensive analysis on the results is for both system evaluations. Chapter 5 concludes this thesis in Section 5.1 and discusses future research directions in Section 5.2.

Chapter 2

Background and Related Work

In this chapter, we first introduce the necessary background required to comprehend the work presented in this thesis. Following that, we review published literature that is related to our work.

2.1 Background

2.1.1 Edge Computing

Edge computing (EC) has seen a surge in implementation with the advent of the SDN paradigm and as a viable solution to the increasingly diverse demands of widespread IoT adoption. EC encompasses a distributed computing paradigm in which computational resources are brought closer to the edge of the network, i.e., nearest to the end-user devices and standalone devices, which are the sources of data. Because a lot of these data collecting devices are in the form of sensors, ranging from simple data to images, video and so on, most of their intelligent capabilities are required to be performed elsewhere, and this role has been assumed by the cloud. By offering high computational power and massive storage capabilities, the cloud can handle requests from hundreds of devices at a time. However, cloud servers are often located far away from the devices from whence these requests are generated. This brings with it a myriad of drawbacks. The first being the physical distance between the source and the cloud adversely affects the total time between a request and its result, i.e., the total time to upload, compute and download from the cloud. This is a considerable roadblock for latency-critical applications, where a real time response is crucial. With so many devices being serviced by relatively few cloud servers, a huge volume of data traffic is generated, which adds to the delay, as well as increasing bandwidth requirements.

Here the niche for edge computing exists. Though the edge cannot mirror the

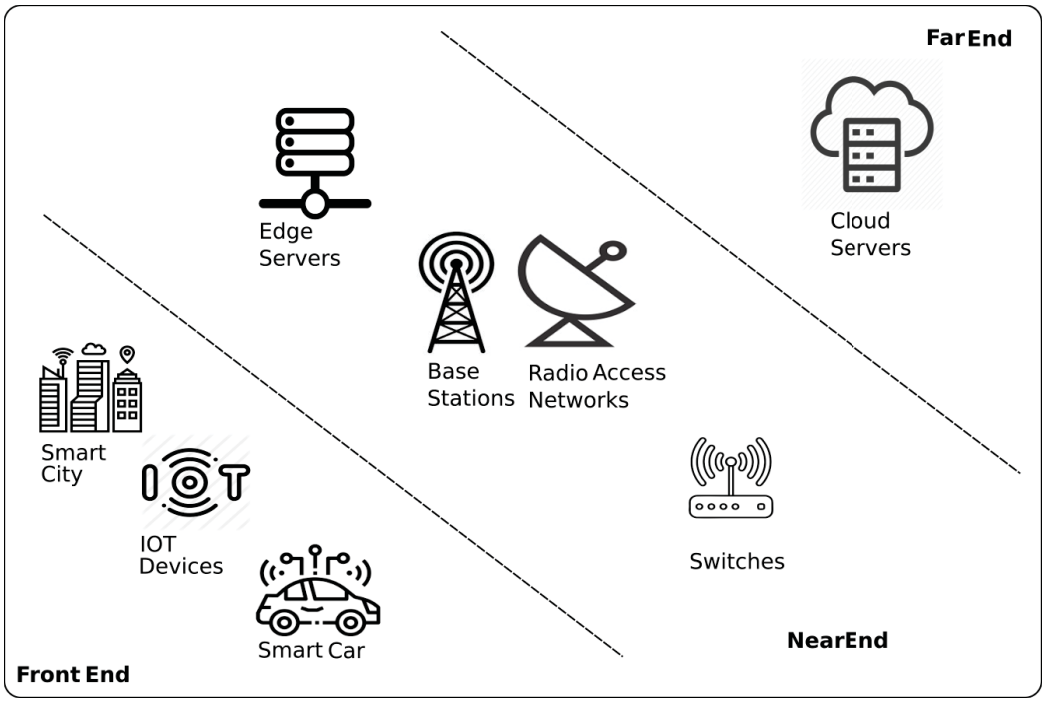


Figure 2.1: Edge Computing architecture

computational prowess of a cloud server, it can perform a considerable amount, often enough to service the latency-critical applications that would greatly reap its rewards. The edge computing paradigm is not however, limited to offloading tasks to an edge device; the distributed system may include partial offloading, or complete offloading, with decisions being made depending on network parameters, application requirements and device capabilities. A brief description of the component structures has been summarized below and depicted in Figure 2.1:

- **Front-end**: This consists of the sensors and actuators in the network. At this stage, only the low computation tasks are performed, which may require quick responsiveness and cannot afford to offload computation to the near-end or far-end.

- **Near-end**: Here, the role of edge computing comes to the fore. This tier consists of the routers, programmable switches and edge servers. These devices have moderate computational power and thus can provide useful services such as data caching, machine learning implementation and also computational offloading from the front-end, such as data pre-processing, aggregation, etc [20].

- **Far-end**: This consists of the core network or the cloud. Only the highly complex computational requests will be offloaded here. All other services are aimed to be handled by the near-end. Thus the original centralized architecture becomes a somewhat distributive one, with the central cloud only called upon when huge storage or processing power is required.

With the near-end implementation, the benefits of edge computing are maximized. There are trade-offs present in terms of computation, latency, as well as reliability for wireless networks. These can be tuned depending on the application requirements themselves. In general, the benefits of EC outweigh any hurdles in its integration. A summary of the advantages of edge computing has been provided below:

Latency and Bandwidth. Because edge computing brings processing power to the edge of the network, a sizeable decrease in latency occurs, as data needs to be transmitted over a far lesser distance instead of the remotely located cloud. However, some transmission latency still exists, so we must ensure there is an ideal trade-off between computational latency and transmission latency whenever a decision needs to be made to perform a task locally, offload it to the edge server, or further to the cloud. This decision making is called scheduling, and a number of algorithms have been developed for the optimization of scheduling.

Computation. Edge nodes offer computational capabilities beyond low-powered IoT devices, thus providing an added level of processing power before the cloud. Task scheduling can be performed according to priority, where real-time services are processed first. Through the distributive structure of edge computing, load balancing can also be implemented in tandem with the cloud. Local devices can also perform those machine-to-machine computations, which do not require much processing power. Edge servers can be placed optimally, to allocate required resources for a large number of nodes. This optimal placement can be calculated by considering the resource requirements and well as the transmission latency from each of the devices under that particular server. Cloud servers can be invoked only when transmission latency is not of great concern, and a large amount of computational capacity is required.

Storage. Edge servers can be used for storage near the IoT nodes. Though they do not provide the massive storage capacity of a central cloud, they can be accessed

quicker for faster response times. This is known as data caching Load balancing and load distribution can also be performed through edge servers, both in terms of computation as well as storage.

2.1.2 Software Defined Networking (SDN) and Data Plane Programmability

Software Defined Networking (SDN) as a network architecture approach has brought a radical change in the administration of networks. By separating the networking logic of a device into the control plane (which is supervised by a controller) and the data plane (which remains on the device itself, carrying out functions according to the application requirement), a greater degree of control and flexibility is introduced into the network. Controllers can then allow for administration of the network on a hierarchical level, without the need for individual intervention, whilst networking devices such as switches or routers can perform their traditional tasks of packet forwarding. In terms of advantages, SDN offers a myriad of benefits. It makes it easier to implement and update network policies on the whole, with a global view available to deployed applications. This, in turn, enhances network performance and reliability, while leaving room for potential extensions in the form of more complex network administrative applications, such as load balancing.

SDN was initially introduced into networking devices such as switches by designating them as basic forwarding elements which communicate with controllers via an open interface. This was primitively the ForCES interface [29] but then the OpenFlow protocol became much more popular due to its physical separation of the controller and network elements [74]. However, OpenFlow itself had its limitations. In particular the introduction of new protocols required a large amount of headers to be added to OpenFlow packets, creating an extra burden on network traffic. Furthermore, periodic sending of packets from the controller as well as delays in topology discovery when new network elements are added made OpenFlow very limited in terms of scalability [11]. This substantiated the need for a further extension, which was done in the form of programmable ASICs, which are implemented on switches.

With programmable switches, the data plane can further be utilized beyond its traditional functions. Programmable switches have multiple pipelines and can process

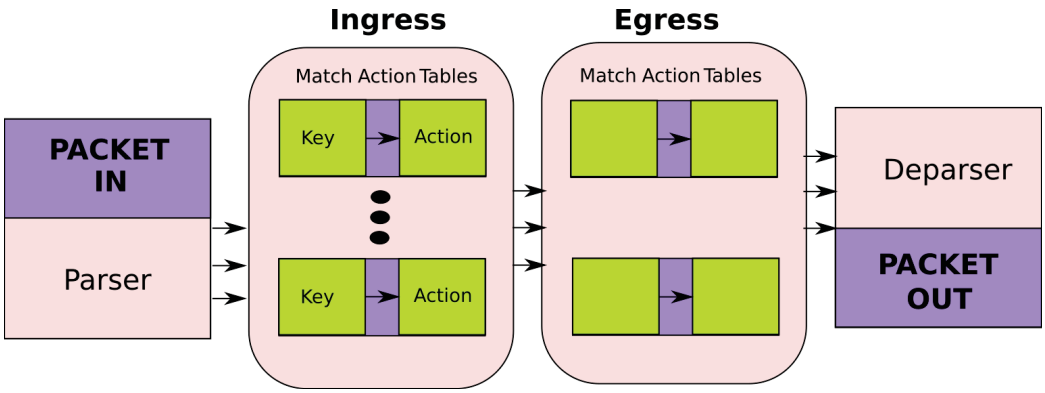


Figure 2.2: PISA architecture model

packets with line rates of up to 3 Tbps. This untapped computational potential can be harnessed through programming these networking devices to perform additional functions to supplement their existing ones. These functions may range from simple applications such as tunneling, firewall implementation, and link monitoring, to much more advanced applications such as traffic classification, intrusion detection, as well as data mining.

The implementation of the programmable data plane is done through the P4 language. P4 was introduced to provide a unified platform through which the various programmable networking devices (PNDs) can be programmed under a single, consolidated, data-plane language. While initially being limited to switches, the language has expanded to accommodate FPGAs and NICs. The P4 language is also protocol agnostic; thus programs can be designed to integrate different protocols following application demand and, by extension, accommodate complex packet processing at line rates. P4 is based on the *Protocol Independent Switch Architecture (PISA)* pictured in Figure 2.2, which facilitates packet processing through the use of parsers, match-action tables, headers and metadata. It is through the use of these constructs that the additional functionality of the PNDs may be added, on a per-packet basis. P4 does not allow for the content manipulation of packets, thus system developers must manipulate packet headers and metadata to carry out their intended services. PISA-based devices include multiple packet processing pipelines. Packets traditionally flow sequentially through the ingress pipeline, followed by the egress pipeline, and then forwarded back to the network. A pipeline accounts for the bulk of packet processing and contains one or more match-action tables.

Once the switch receives a packet, it parses through the header contents, including traditional physical, network, and transport layer headers as well as application-specified custom headers. Any number of the values extracted from these headers are then used as keys for the match-action tables that are present on the switch, performing a subsequent *action* according to the *match* found in the aforementioned tables. Packet header values may also be manipulated according to the application demands. This constitutes the ingress processing pipeline of the switch, and is traditionally where the bulk of computation on the switch occurs. This computation takes advantage of the packet metadata and on-board registers, constructing a memory space that can span multiple pipeline stages. Depending on the application complexity, packets may be recirculated through designated ports on the switch, carrying information for further manipulation through rewritten header values.

Challenges. However, the P4 platform itself is not without its architectural limitations. Programmable switches are constrained in terms of resources (e.g., memory) as well as implementation of arithmetic operations. While operations such as addition and xor are possible, others such as division are entirely absent from the platform. This makes it tough to *de facto* implement applications on the platform which involve complex computations, such as those required in a neural network. Furthermore, floating-point numbers are also not usable on the P4 platform, which makes even simple calculations require a workaround to be implemented. There are a number of works which implement their own circumventions to bypass these limitations, such as the authors in [28] implementing division using logarithmic and exponential functions. In this thesis, floating point values were implemented using the fixed-point representation, which allowed us to partition each 32-bit value into a 28-bit integer and 4-bit precision to represent the decimal part. The division and multiplication operations were implemented using the same principle as [28]. For the implementation of XNOR-popcount operations in the neural network implementation, we used the same method as that used by the authors in [66], which is a vectorized approach using Single-Instruction-Multiple-Data (SIMD) instructions.

2.1.3 Image Classification

Computer Vision (CV) has been an imperative part of the broader Artificial Intelligence (AI) paradigm; attempting to make machines learn how to identify, analyze, recognize, and infer information from images and video, similar to how the brain processes information sent to it by the human eye. Image classification remains one of the most fundamental tasks of CV, as its principles can be extended to incorporate other tasks such as image segmentation, object detection, object recognition, motion sensing, etc. As such, it is an ideal starting block for implementation on various platforms, such as programmable switches in this work, due to its potential for extension.

For image classification to perform within acceptable performance bounds, there are two primary prerequisites [61]:

- A suitable classifier, from among the myriad of classification methods available in traditional machine learning as well as deep learning. Each image type and application combination may demand a specific classifier to perform optimally.

- Ample data, or training samples, with which the classifier may *learn* how to infer the proper classes from the information provided to it.

Traditional classifiers work based-on features. Usually, for training samples and tuples, this is readily available for data in the form of attributes. However, for images, features describing the image need to be extracted beforehand, and then fed to the classifier as input to infer the class to which the image belongs. Optimal selection of features has been studied extensively [10, 32, 83, 102], but remain entirely dependent on the nature of the images involved as well as classifier being used.

For deep learning methods such as convolutional neural networks (CNNs), feature extraction is integrated into the classifier itself. This is expanded on further in Section 2.1.5. In brief, different features are extracted from the images by each neuron in the network, through convolution. These are then passed on to further neurons which establish patterns among these features and adjust their own weights accordingly to maximize performance.

2.1.4 Traditional Machine Learning

Machine learning has been successfully integrated into the realm of networking for a plethora of different applications such as traffic prediction [64], routing [94], network security [92] and classification [97]. In particular, [97] implemented traditional machine learning methods on the P4 platform for the purposes of classification of network data. Through their investigation, the authors discovered decision trees to perform best in terms of accuracy against other classifiers such as SVM, k-means clustering and Naive-Bayes classifiers. Based on their evaluation, we selected decision trees as the appropriate classifier to implement for image data.

Decision trees are classifiers that represent a subset of any given instance space, such that the attributes of the decision tree and their subsequent values determine the classification of any samples in that instance space. By definition, decision trees are directed graphs, consisting of nodes and edges, with no cycles. The "root" of the graph represents an attribute which splits the tree into two paths. The left subtree is followed if the attribute of the unclassified sample is less than the chosen threshold of the root, whilst the opposite is true for the right subtree. This path is followed for each subsequent node of the subtree until a leaf is reached, which represents the class decision for that particular sample as depicted in Figure 2.3. Alternatively, the leaf may hold a probability vector indicating the probability of the sample being of a particular class [69].

Decision tree attributes may contain discrete values or continuous values, numeric or otherwise. With numeric attributes, the decision tree can be considered as a combination of multiple hyper-planes, or decision boundaries. With more concrete decision boundaries, the accuracy of the tree increases. Consequently, the complexity of the tree also has an impact on its accuracy, where accuracy may be characterized by tree depth, number of nodes and/or number of attributes. Each path along the tree represents a class decision and, simultaneously, an induced rule, which may be derived by conjoining the attribute values along the same path. This improves understandability for tree interpreters [68].

Decision trees have a number of benefits besides their relative simplicity in implementation compared to other classifiers. One of the primary motivations behind classification using a decision tree is that each sample is tested against a subset of

	AttributeA	AttributeB	AttributeC
Sample1	105.23	20.65	5.00

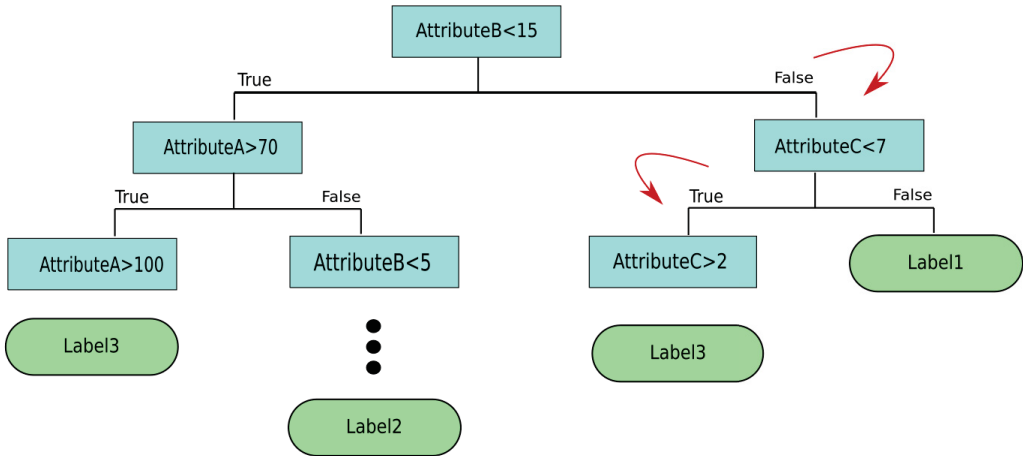


Figure 2.3: Decision Tree inference

classes following the pathway that the sample pertains to, instead of comparison against all possible classes. This drastically increases efficiency as well as inference time. As an extension of the above, the presence of numerous attributes at each level of the tree for any particular pathway, provides a degree of flexibility so as to optimally create decision boundaries among the possible classes for any given dataset. Any combination of these attributes may provide varying levels of performance once placed at different levels of the tree, and is an exercise of optimization in and of itself. However, the added flexibility that this brings may be considered a pitfall of the decision tree classifier itself, as designing the optimal decision tree is a difficult task, and performance of the classifier largely depends on the optimality of the tree. Also, with a high number of classes, which may not be largely dissimilar, overlap between classes may occur that cannot be expressed by the finite number of attributes that the decision tree is represented by. This results in a significant drop in accuracy [77]

There are a number of decision tree classifier algorithms that are used to build the tree. However, most of these are derivations of the most popular algorithms. These algorithms are all based around a parameter known as *Information Gain*. Datasets in machine learning theory can be described by their *entropy*, which is defined as

the the measure of impurity in a collection of a dataset, i.e., how much variation is present in the dataset in terms of classes. This is outlined in Equation 2.1,

$$E_{dataset} = -\sum_{i}^{c} p_i \log_2 p_i \tag{2.1}$$

where c refers to the number of classes in the dataset, and p the proportion of the dataset encompassed by class i. Since entropy is a summation of proportions, the value will always range between 0 and 1, with 0 being the lowest entropy and 1 the highest entropy. A higher entropy refers to a higher amount of disorder in the dataset. In other words, there are a high number of classes present in the dataset. on the other hand, an entropy of 0 indicates that all samples in the dataset belong to one class and there is no disorder.

Based on the entropy of a dataset, each attribute is then sorted according to the information that is gained upon splitting the dataset by the variations of that particular attribute. This is calculated by then subtracting the new entropy from that of the whole dataset, after being weighted by the proportion of the dataset covered by the splits. This is summarized in Equation 2.2, where v refers to the possible attribute values for that particular split, and w refers to the weight of that split in comparison to the whole dataset. For categorical attributes, the question of where to split the attribute is trivial, each subtree is simply assigned to an attribute value. However, for numerical attributes, the decision to split the numerical value at a particular threshold consists of a few steps. First all the training samples are sorted according to that attribute and each value is set as a threshold, where all values less than the threshold are assigned to the left subtree and all greater are assigned to the right subtree. This process is replicated for all training samples, and the threshold where the highest information gain is achieved is selected as the split for that particular attribute. Consequently, the higher the number of training samples, the longer it will take to make this split decision and by extension, training time will increase [17].

$$\begin{aligned} IG &= E_{dataset} - E_{split} \\ E_{split} &= -\sum_{i}^{v} w_{split_i} \sum_{i}^{c} p_{split_i} \log_2 p_{split_i} \end{aligned} \tag{2.2}$$

ID3 The ID3 algorithm was created by Ross Quinlan, which works by creating a multiway tree, and finding for each node the branching criteria, both in terms of

attribute as well as magnitude. This branching criteria is determined by calculating the information gain for each attribute, as shown above, and selecting the attribute with the highest value at each node. To be noted, once the tree has branched to a particular split, only the proportion of the dataset conforming to that split branch is used for further branching calculations. Trees are grown to their maximum size and then a pruning step is usually applied to improve the ability of the tree to generalise to unseen data. ID3 may only work on categorical attributes and does not perform for attributes with continuous values. [67]

C4.5 The C4.5 algorithm is an improvement over the ID3 algorithm, and was introduced as an extension to it by Quinlan himself. C4.5 allows the use of continuous values for attributes by partitioning attribute ranges into intervals and treating the partitions as if they were categories. This allows a significant increase in dynamism of decision tree applications. Once the tree is trained, it is converted to a set of if-then-else rules, following each tree pathway from root to leaf, and pruning out subtrees based on preconditions which improve accuracy when removed. C4.5 can also handle missing attributes for test samples, by simply not using them for entropy and information gain calculations. However, C4.5 has a high tendency to overfit, performing well on training data but considerably worse in noisy real-world data [87].

CART CART is an extension of C4.5 and allows for numerical target variables. CART constructs binary trees using the feature-threshold combination that yields the largest information gain at each node. CART uses the Mean Squared Error (MSE) when determining splits for continuous value attributes. Unlike C4.5, CART is not as prone to overfitting and can perform well with training data as well as test data. As all the values in our features involve numerical target values of continuous nature, CART was the optimal choice for tree generation. Though not relevant for our work, however noteworthy, CART can eliminate insignificant variables and only work with significant variables for decision tree training and generation [17, 87].

Decision trees offer a simple yet effective method for a large variety of classification problems, and has provided a part of the inspirations towards advancing the state-of-the art in terms of classification applications, leading to neural networks and eventually deep learning.

2.1.5 Neural Networks

Classification remains to be one of the most important research areas in the realm of computer science. This may involve any source of data, including images, video, audio, strings of words, and so on, as well as its extension to other tasks such as recognition, prediction as well as further decisive analysis. A myriad of problems in industry, business, science, and technology, can be reduced to classification, making it a lucrative direction to make progress in.

Traditional machine learning has created a large number of approaches to classification, each with their own benefits, drawbacks, applicability as well as limitations. These include decision trees, support vector machines, Bayes classifiers, k-nearest-neighbor classifiers. The crux of these classifiers is that they are based on a probabilistic approach to the classification problem; they predict the probability that any given sample belongs to a given class based on some underlying assumptions from other samples of that class that the classifier is familiar with.

Neural networks, on the other hand, are far more self-sufficient than traditional classifiers, by adapting faster and more efficiently to the environment and data where it has been implemented. Neural networks are able to empirically approximate any function through its own architecture and parameters, as well as being inherently non-linear through the use of its activation functions. This non-linearity allows it to model real-life scenarios far more accurately, making it the de facto representation of real world data and applications.

A neural network traditionally consists of *layers*, each of which themselves consist of an arbitrary number of nodes, known as neurons. The numerical amount of layers and nodes in a neural network vary largely, between applications as well as neural network architectures themselves. Some may require a heavy network, with a large number of nodes, while other applications may be sufficiently served with a shallow neural network, with a lower number of layers and subsequent neurons. Neural networks traditionally consist of an input layer, followed by one or more hidden layers, and ending with an output layer [37]. The functional specifics of each are outlined below:

Input Layer: This layer is used to receive the data that will be passed through the neural network. This layer may contain one or many neurons, depending on the

application. However, usually each neuron receives a particular feature or attribute of the data sample. For example, a neural network that classifies customers for a departmental store, may have as its input neurons, customer details. Alternatively, input neurons may be singular, such as in convolutional neural networks where the input is simply an image, and not a collection of features [37]. Since our work deals with image classification, the input layer in our neural network architecture also consists of one neuron, the image input itself.

Hidden Layer(s): Neural networks usually contain several hidden layers, which make up the bulk of the network. However, shallow networks may even contain only one hidden layer. The purpose of hidden layers are to perform the mathematical functions and calculations needed for the network to produce the desired results. These functions and operations are discussed further in the section, and their diversity and applicability aid in making neural networks a viable solution for virtually any problem [37].

Output Layer: This layer serves to provide the output of the neural network inference. In most cases, for classification tasks, the number of neurons in the output layer is equal to the number of possible outputs from the expected dataset that the neural network will be applied to. In other words, its equal to the number of possible outputs. Usually, the outputs are expressed in the form of probabilities in the case of classification problems, and mathematical values in case of other non-linear problems [37].

2.1.5.1 Mathematical Operations

Neural networks make use of a large number of mathematical operations to perform their assigned tasks. These operations are usually carried out in the hidden layers of the network architecture. This operations and the operators involved are outlined below:

Weights and Biases Weights and biases are numerical values that form the backbone of the neural network, and are responsible for their accuracy. As a result, these values are modified and manipulated during the training phase to improve the network to the desired performance standard. Weights are associated with each neuron in the network, and are multiplied with the inputs from the previous layer

Function	Formula
Binary Step	$f = \begin{cases} 0, & \text{if } x <= 0 \\ 1, & \text{if } x > 0 \end{cases}$
Linear	$f = ax$, where a is a constant
Sigmoid	$f = 1/(1 + \exp^{-x})$
ReLU	$f = \begin{cases} 0, & \text{if } x <= 0 \\ x, & \text{if } x > 0 \end{cases}$

Table 2.1: Some activation functions and their corresponding formulae

passing through that neuron. Weights decide how much influence a neuron will have on the output for the next layer. Once all the weight-input products are summed up, the bias value is applied, resulting in the output for that layer, which is subsequently the input for the next layer. The equation for weights and biases and how they are multiplied with the input data is outline in Equation 2.3:

$$Y_{output} = \sum_{i=1}^{n} w_i x_i + \beta \tag{2.3}$$

where x_i refers to the input data $x_1...x_n$ and w_i refers to the weight vector which has the same dimensions as the input data, $w_1...w_n$. This is then fed as input to the activation function which discerns the final output for that particular neuron. Since each neuron has a different weight vector and the above operation is carried out at all neurons across a layer, neural networks require a large number of computations, proportional to the input size as well as layer size.

Activation Functions: If neural networks solely depended on the multiplicative and additive nature of weights and biases as mentioned above, they would devolve to linear regression calculations and would not be able to perform well in case of non linear applications. For this reason, some form of non-linearity must be introduced to the network, which is done in the form of activation functions. Each layer, and the neurons as its constituents, are associated with an activation function, which transforms the neural output to a non-linear form. There are a large number of activation functions and each perform drastically different depending on applications, some of which are outlined below. Formulae for each function are provided in Table 2.1.

- **Binary Step:** Input is converted into a 0 or 1 depending on whether the value is negative or positive. This function is generally used for binary classifiers.

- **Linear functions:** Input is multiplied with a constant coefficient, making this function suitable for linear regression applications.

- **Sigmoid:** Sigmoid functions transform the input to normalize within the range of 0 and 1 and is useful for non-linear applications.

- **ReLU:** ReLU or rectified linear unit only considers positive values for input, transforming any negative values to 0. This creates the effect where only positive-valued neurons will be activated moving to the next layer. It also offers a computational efficiency advantage over the sigmoid function.

Loss Functions: Neural Networks improve their performance by repeated training and updating of internal weights in the network. This updating insight is gained though the predictive error generated by the network, i.e., what proportion of the predictions made by the network were incorrect. Based on this error, a mathematical function is used to modify the weight values, and similar to activation functions, loss functions are also quite diverse and dependent on the application, as well as the input type being used. Among these functions, *categorical cross entropy* loss function is the most popular for multi-class neural network which work with full-sized floating point values, For applications with only two classes, the *binary cross entropy* loss function is better suited to the task. However, as in the case of this thesis, these functions are unusable for neural networks which have been binarized, and thus require specialized functions known as *squared hinge* and *categorical hinge* for binary and multi-class applications, respectively.

2.1.6 Convolutional Neural Networks

Convolutional neural networks are one of the many neural network architectures brought on by the advancements in deep learning. With improvements in the processing capabilities of graphical processing units, as well as an increase in the demand for image and video based processing for a diverse range of applications, convolutional

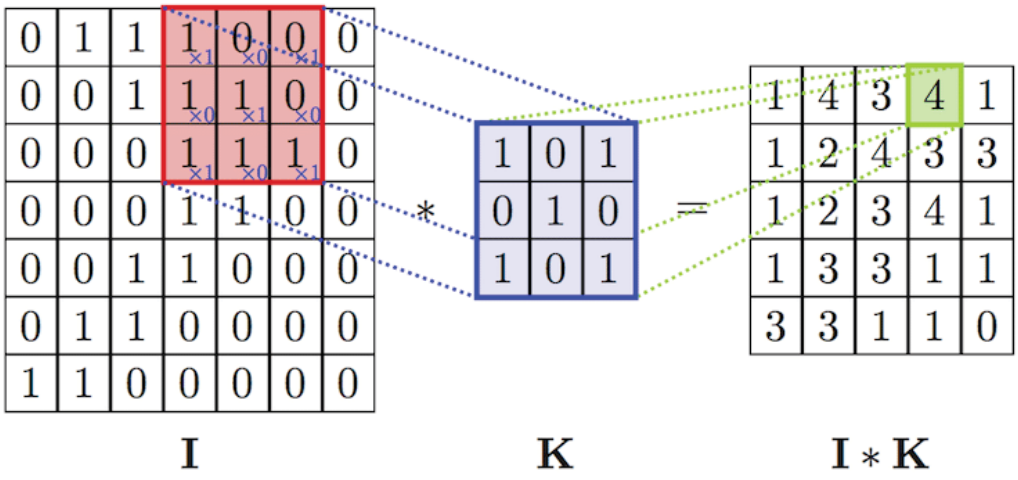


Figure 2.4: An example of convolution using a kernel K [76]

neural networks have rapidly become the standard for largely visual recognition but also speech and natural language processing.

In traditional neural networks, inputs usually consist of samples or *tuples* which are made of attributes describing that particular sample. These attributes are called features of the input, describing it through numerical values. These features are used by the neural network to learn the patterns hidden within the variety of inputs and train itself accordingly.

However, these attributes may not be defined for all sorts of input data types. For example, when classifying or recognizing objects in images and video, only the raw image or frame is the input available to the classifying network. Thus, the neural network must include provisions to extract features from the image, and then it may perform the calculations necessary for classification. This is done by the convolutional layer of the neural network, which may be single or multiple, and followed by the fully connected hidden layer(s) and the output layer.

2.1.6.1 Convolution

For extracting features from the input, convolutional neural networks perform *convolution* which is a linear operation involving the element-wise multiplication of neural network weights with the input, as shown in Figure 2.4. Multiplications are element-wise because in contrast to traditional NNs, CNN weights in the conventional layer are matrices, to match common CNN inputs, such as images. Each of these weight

matrices are called *kernels* or *filters*. These kernels are in dimensions smaller than the image, and perform their convolutions over the whole image. For each iteration over the image, the image matrix, more specifically the window of the image being processed, and the kernel matrix are multiplied with each other in an element-wise fashion, producing a single response value. This process is repeated over the whole image, as the kernel moves in an overlapping manner through each pixel, right to left, top to down. Since the purpose of the kernel is to extract features from the image, this feature will be detected if present regardless of its location. Simultaneously, this convolutional process serves to perform dimension reduction as well, through converting processed windows to a single response value. This builds up a feature map for each neuron of the convolutional layer, which are then subjected to the aforementioned non-linear activation functions.

2.1.6.2 Binarization

Neural Networks often involve a large number of layers and, consequently, require a large memory footprint in terms of storing weight and bias values. For devices or applications which are constrained by resources, implementation and storage of a complete neural network becomes an infeasible solution. This calls for alternate solutions such as distributed neural networks [9], shallow networks, and binarization [24].

Binarization involves the reduction of weight and bias values to +1 or -1. This allows variables to be stored as 1-bit values, drastically reducing the memory footprint of the neural network. To facilitate the operations of these binary values within the neural network, multiplications are replaced by an approach known as XNOR-popcount [45]. This is based on the principle that multiplication of two values can be approximated by performing the xnor operation between their bit-strings and then calculating their set bit values, i.e, *popcount*. For the work done in this thesis, we also implemented a binarized version of our neural network due to the resource-constrained limitations of the P4 platform, implementing the same XNOR-popcount operations to approximate the convolutions and multiplications in the neural network.

2.2 Related Works

The domain of edge computing in terms of latency-critical applications is vast as edge nodes offer computational resources at a distance where the latency can meet the demands of the application. Among these latency critical applications, areas such as AR/VR, image filtering, surveillance, object classification and recognition from video feeds, can extensively benefit from image classification at the edge. Image classification itself has evolved leaps and bounds over time, moving from traditional image processing methods such as feature detection, to machine learning methods and currently, deep learning using neural networks. Here, we provide a review of the literature in each of these domains related to our work.

2.2.1 Edge computing

The edge computing paradigm brings with it a number of benefits as outlined in Section 2.1. Work done in this paradigm covers a wide range of applications, with varied purposes. For decision-making regarding scheduling where the queuing state of the task buffer and the execution state of the local processing unit are both considered, the authors developed their own scheme in [57]. For the niche of mobile gaming, the authors in [49] created a distributed computation offloading scheme for the multi-user computation of mobile games. This allows games to run on otherwise low-end devices. In order to minimize service delay in an edge cloud computing environment, the authors in [73] proposed an analytic model. This model uses virtual machine migration to minimize transmission delay through transmission power variation, by increasing power whenever packets need to be transmitted a further distance.

In many applications, a large amount of data needs to be sent which requires pre-processing and aggregation before arriving at the cloud. These tasks can be achieved at the edge server. A contract based resource allocation algorithm was proposed in the paper [100], through which devices bid for a contract with the edge server to transmit data for computation. During transmission, though, we must also be aware of the energy resources available to the device. Edge computing can incorporate a flexible task offloading scheme which considers the power resources of each device, such as in [101], considers both the task computation costs as well as the energy

cost for transmitting files, when deciding whether to offload a task. With the aid of edge servers, trivial packets can be aggregated and pre-processed in order to reduce unnecessary overhead.

An algorithm for dropping redundant packets to save storage space called MM-Packing was proposed in [82]. This is useful in applications such as video analytics, where often subsequent frames may not have much or any difference between them, and thus can be stored as a single frame instead of multiple, greatly reducing storage space used up. Since edge servers do no offer as much storage capacity as the cloud, algorithms such as this provide a sizeable benefit. Other nodes may also be used as redundant storage, if they are available, through the use of data replication. This redundancy gives a provision for data loss recovery if it occurs.

While these works do not directly employ machine learning under the paradigm of edge computing like our work, they offer an insight into the varied nature of edge computing implementations and how its advantages can be seen not only in terms of latency but also savings in energy consumption at the end, computational offloading based on scheduling, as well as storage in the form of caching.

2.2.2 In-network computing

In-network computation involves the use of networking devices such as switches, that are already utilized for packet forwarding, for the computation of additional tasks such as anomaly detection, firewall implementation, classification, data caching, etc. With programmable switches becoming more prevalent, tasks such as application acceleration have also seen an upward trend.

A review of then-existing approaches to in-network computation, primarily focusing on the communication aspect of computation was put forward by Giridhar *et al.* [33]. In other terms, whether the computational loss from performing such tasks on networking devices would be offset by the benefits incurred in terms of latency, transmission and bandwidth savings. A more modern approach to this review from the perspective of data centers was brought by Sapio *et al.* [79], whilst exploring the architectural limitations of programmable devices in the network. The authors also created a proof-of-concept implementation of in-network data aggregation. An in-network solution for caching data at programmable switches close to user devices

was implemented by Liu *et al.* [59]. By accelerating storage and displaying sizeable gains in throughput as well as reduced request latency from users, the authors advocated for similar in-network implementations on programmable switches and network accelerators throughout data centres. A similar system for caching data using programmable switches was proposed by Jin *et al.* [46]. By taking advantage of the ASICs present on these switches, their system can detect, index, cache and serve hot key-value items in the switch data plane with efficiency. In comparison to [59], this system provides a load balancing cache which requires little storage space. The authors implemented their proposed system on a Barefoot Tofino switch [4] and achieved notable throughput increases and latency decreases.

For the application of lock managers in the in-network computation paradigm, Yu *et al.* [99] put forward their arguments. Programmable switches can directly process lock requests through an integral memory management system involving both the switch and server-side memory. Furthermore, the authors envision their system to be implemented in an RDMA-based system, and demonstrated this through their implementation on a Barefoot Tofino switch, claiming a 20x increase in throughput.

A system that redesigns encrypted data stores to allow for operations which reduce query latency as well as optimize memory footprint using a novel caching solution was presented by Kuzniar *et al.* [52]. The prototype implemented by the authors is an extension of [46] adding a hash map for each encrypted key that is stored as well as utilizing variable-sized register blocks. The authors achieved a 20-25% latency reduction compared to the state-of-the-art.

In comparison, the work in this thesis also performs in-network computation, being the only implementation that performs classification of *images* as an in-network task.

2.2.3 P4-based implementations

The P4 platform allows for the creation of platform-agnostic applications for programmable network devices. These range from switches to FPGAs as well as NICs. This added flexibility means that a wide range of applications may be ported into the P4 domain for deployment in edge and in-network computing scenarios.

The implementation of a number of classifiers in the P4-platform for the purposes of anomaly detection among IoT traffic generated from a large number of devices

was proposed by Xiong *et al.*, in [97]. The authors implemented the decision tree, Naive Bayesian, k-means clustering and support vector machine (SVM) classifiers for a comprehensive comparison all of which were implemented using the PISA architecture of P4, and utilizing the match-action pipeline. The authors noted that the decision tree classifier performed the best on their chosen network dataset, using a NetFPGA-based implementation for their evaluation. However the author's highlighted their system to be robust with the number of features they have selected, and may not scale for different data types with an extended number of features.

Another work towards implementing machine learning methods using the P4 platform in [18] was published by Busse-Grawitz *et al.* [18]. The authors present a system that makes use of the random forest classifier, for classifying network traffic. The authors envision their work to be used for "self-driving networks", networks which optimize themselves by studying traffic parameters and using an inference model to identify what the next best course of action or adjustment. Their system aids this by displaying the feasibility of performing inference using in-network programmable devices. The authors note the challenges posed by the limited resources available on PNDs (programmable networking devices) in terms of operation support as well as memory.

A library for circumventing some of the architectural hurdles of the P4 platform was developed by Ding *et al.* [28]. The authors introduced an algorithm for logarithmic and exponential functions in P4. This allows for the implementation of division functions which can be calculated using mathematical approximation and the functions implemented by the authors. The functions were evaluated based on varying bit width and precision. They demonstrate the use of these functions by calculating the entropy of network traffic in the data plane.

The proposed thesis work uses the evaluation done in [97] to focus on decision trees as the suitable classifier to implement on programmable switches, working with images instead of the widely explored network traffic data. The principles of division implementation explored in [28] are also included in our division operation implemented in P4.

2.2.4 Decision Trees for image classification

A number of image classification solutions exist which make use of traditional machine learning methods in favor of deep learning methods. This is primarily done due to their simplicity, relative ease of implementation, and perhaps most importantly their lack of a "black-box" trait. This makes them far more interpretable than deep learning neural networks. Decision trees are among these classifiers and have seen considerable utilization.

A decision tree-based solution for the classification of bitmap images, among a dataset consisting of 5 distinct classes was presented by Surynek *et al.* [90]. The authors make use of a feature set that consists of features dependent on image color, image contrast and subsequent histogram of grayscale values, edge detection and line segments. Theses are then used to train a decision tree based on the ID3 algorithm. The authors did note that a better version of the algorithm, C4.5, was available but not used. A classification accuracy ranging from 75% to 85% was achieved for the different classes.

An approach towards the usage of decision trees for image classification in a different manner, was proposed by Zhang *et al.* [103]. The authors perform the image classification of polarimetric synthetic aperture radar (POLSAR) data in a two-step process. First the image data is segmented using a multi-resolution algorithm to produce homogenous pixels. Then a set of 5 chosen parameters are extracted from theses segments, which act as attributes for the subsequent decision tree classifier. The authors noted a 13% increase in accuracy over the state-of-the-art classifier for their given dataset, noting that the use of two new features not previously implemented were likely responsible for the improved results. They also noted that the 5 chosen feature parameters may also be extended to general classification purposes. Contrary to traditional image classification where the whole image is classified to a single label, the authors here demonstrate an application where multiple segments in an image may be classified to different labels.

Our work draws inspiration from the features outlined in [90], to select the seven features which had the most impact on decision tree performance and could be feasibly implemented on the P4 platform.

2.2.5 CNNs for image classification

As deep learning has gained exposure over the years, a number of neural networks have been developed that are specialized in terms of architecture for specific types of applications and data. Among these, convolutional neural networks (CNNs) have been established as the state-of-the-art for image classification and other image tasks such as segmentation, object recognition and detection. Surveys such as [71] and [84] perform a comprehensive analysis and review over the various convolutional neural networks available, which vary in terms of depth and architecture, covering a myriad of datasets and image variations in the process.

A study comparing the effect of varying CNN architectures in the domain of autonomous vehicles was performed by Kebria *et al.* [48], particularly with respect to changes in neural network depth, filter sizes and number of filters. The authors noted that rather than the number of filters, the allocation of varying-sized filters throughout the network provided a sizeable increase in performance. Similarly, in terms of layer number, a middle-ground approach is optimal rather than a shallow or extensively deep network.

A comparative study between the performances of a simple CNN, VGG-16 and ResNet-50 for gesture recognition was performed by Begum *et al.* [15]. This was subsequently used to control the flight of an unmanned drone vehicle. The datasets used for this purpose where generated by the authors using varying conditions while capturing gestures for different movements. While the three architectures performed similarly in case of accuracy and f1 scores, the simple CNN had a much lower loss for both the training and validation sets when compared to the other two.

In order to compare error rates to more traditional, heavier CNNs, a simple CNN was created by Guo *et al.* [36]. The authors propose a simple architecture consisting of three convolutional layers and 2 fully connected layers. These networks were then trained on the MNIST [53] and CIFAR-10 datasets [51] and the results noted against popular networks such as APAC [80], RCNN-96 [56] and Deeply Supervised Network [54]. While the simple neural network did not perform better than these networks, it did offer a competitive error rate of 0.66% compared to 0.45% on average from the other networks, whilst having a considerably simpler architecture and smaller memory footprint.

A system which makes use of binarized neural networks implemented in FPGAs to perform image classification at a high inference rate was proposed by Umuroglu *et al.* in [93]. This work was also evaluated on the MNIST and CIFAR-10 datasets. Usually neural networks are sensitive to small changes in data, which occurs during weight updates while training the network. However, a large amount of redundancy is introduced with the resulting parameters during this phase. This is the motivation behind binarization of the neural network parameters, which is done in this work using binary input activations, binary synapse weights and binary output activations. Competitive accuracies were reported for both datasets.

To summarize, there are a number of works which implement some form of classification in the domain of programmable networking devices, within the paradigm of edge computing on the whole. However, none of these existing works propose a system that can work with images for the purposes of classification in latency-critical applications. A few works present a classifier for network traffic data, which is already present in packets that are forwarded throughout the network. These works implement some form of machine learning or neural networks. However, images remain far more complicated than network traffic data and, by extension, require more computational resources to extract meaningful features to classify them accurately. Similarly, this computation needs to occur within a short time window to remain viable for applications where low latency is paramount. It is also worth noting, while these published works have delved into the realm of in-network classification using machine learning methods on programmable networking devices, these have been done on devices such as NetFPGA, which differ vastly from programmable switches both in capability as well as overall function. We have chosen programmable switches as the target device due to their closer proximity to end-devices, which serves the purpose of latency-critical applications better. Other devices such as FGPA are usually associated with servers and thus at a further hop count than switches, diminishing the latency-oriented benefits of in-network computation. Thus, our proposed thesis fills this niche, establishing that image classification on programmable switches is feasible in terms of accuracy and scalability and also serves as a foundation for extension into further computer vision tasks on the same platform.

Chapter 3

Decision Tree-based System: Design and Evaluation

3.1 Research Methodology

We first established the problem definition, which is expanded on in the next section. In brief, we substantiated the existence of a niche where latency critical-applications such as AR/VR, intelligent transport systems, and smart factories, would benefit greatly from having processing done on the edge. Next, we identified image classification as an important task among these applications and recognized that such an application does not exist currently on the programmable networking device platform, which is a suitable target for edge computing due to its proximity to user devices and high-speed packet processing capability. We then designed a decision tree classifier that can be implemented on the platform, selecting this classifier for its simplicity and competitive accuracy. This was then evaluated against a server based implementation of similar architecture with datasets standard to the image classification paradigm. Once we established the feasibility of the system, we extended the classifier to a neural network based system, which is the state of the art for images. We designed the architecture so as to maximize accuracy whilst remaining within the viable implementation constraints, necessitating the need to binarize the neural network system. This extended system was then evaluated against the decision tree and server based implementations. In conclusion we found that this system performs at an acceptable accuracy for our target applications considering the latency benefits it presents.

3.2 Problem Statement

With the adoption of machine learning and deep learning, the capabilities of devices in terms of artificial intelligence has seen a massive leap [22, 23, 77]. However, alongside these advances in capabilities, demands in terms of resources have also increased. As

we have seen, IoT devices are integrating themselves further into the technological mainstream, giving rise to smart cities, smart homes, and latency-critical applications such as intelligent transport systems, augmented reality systems and drone-based surveillance systems [7].

The challenge therefore arises in integrating tasks such as image classification, which are an imperative part and fundamental aspect of a myriad of these applications, into the resource-constrained realm in which these devices reside. This challenge must be met by offloading these computational tasks, partial or otherwise away from the end-devices and to nearby devices which include edge servers and programmable switches. It is therefore crucial to find an equilibrium point in terms of latency constraints and bandwidth demands against the computational power available to these mobile end-devices.

3.3 System Architecture and Design: Decision Tree-based System

In order to solve the above defined problem, we propose our decision tree-based system, NetPixel, and we explore its architecture and design in this section. First we provide an overview of the complete system, followed by the protocol we designed for our system, as well as the features used and the pipeline implemented.

3.3.1 Overview

NetPixel is a framework consisting of the image capturing devices, the programmable switch harboring the classification program, and a network controller responsible for training the decision tree classifier and implementing/installing the classifier on the switch. Based on the investigations performed by the authors in [97], which compared the performance of different classifiers implemented on the P4 platform, we selected the decision tree as the approach we would implement for the purpose of image classification on a programmable switch. Decision trees offer a simplistic approach which would mirror the resources we have available on the PISA architecture, namely match-action tables, which would be further expanded in this section.

An overview of this system can be seen in Figure 3.1. In the following subsections, we present the functions and operations of these components of NetPixel.

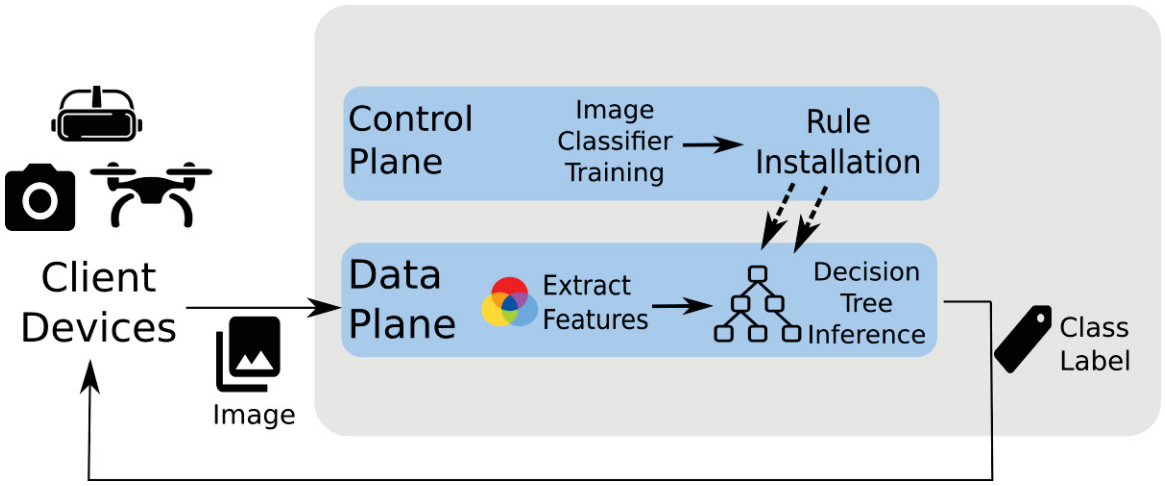


Figure 3.1: NetPixel Overview

3.3.1.1 Programmable Switch

The programmable switch is responsible for the classification of the image it receives from the client devices, which include image capturing devices such as VR headsets, on-board cameras, surveillance cameras, as well as unmanned drone vehicles. The switch also re-transmits the class decision for the image back to the client device, which may then perform a subsequent action based on the image class. As an example, a VR headset may send an image of the user's field-of-view to the programmable switch, which subsequently classifies the image according to the decision tree rules implemented on it. This decision is then received by the VR headset, which may display a unique overlay according to the class.

First, the switch receives the image in the form of a number of packets, holding parts of the image known as chunks. These chunks are squared groups of pixels, which are sent from left to right and top to bottom of the image. The protocol for sending these packets and chunks are outlined in Section 3.3.2. Once a packet is received by the switch, the packet is parsed to extract the RGB values of the chunk pixels from the packet. These RGB values are stored on the packet as 8-bit values which is the standard for RGB color-space [91], totalling 24-bits to represent each pixel in the square chunk. These are then used to calculate the seven features which we have selected for our system, based on [90] and outlined in Section 3.3.2.2. Some of these features require the conversion of the RGB values into the grayscale color-space, also

known as the intensity values. This conversion is also performed on the switch and the subsequent features which require it are calculated. As each image chunk arrives on the switch, each feature value is calculated, updated and stored on registers on the switch, which act as a memory for the classifier.

Once the whole image has completed transmitting and all the chunks have been received by the switch, the decision tree implemented on the switch is invoked. The chunks sent by the client device may be discarded or forwarded to an edge server or cloud for further processing. However, this is mutually exclusive to the working of the decision tree. This decision tree is implemented in the form of a match-action table where the feature values are used as keys for the table. This corresponds to an image class which is then written onto a packet and sent back to the client device.

3.3.1.2 Network Controller

The controller which supervises the programmable switch where NetPixel is installed is responsible for the training of the decision tree, which subsequently generates the rules upon which the tree itself performs its primary functionality. As aforementioned in Section 2.1.4, we have implemented the CART algorithm for the training of our decision tree model, as it is the most suitable for our use case considering the numerical nature of the selected features. The controller trains the decision tree based on the available image data and then converts the decision tree into a set of rules. Each rule indicates a path in the tree from the root to any given leaf. Thus, the whole set of rules completely encompasses the tree and all the leaves in it, which represent the class labels. As the size of the paths vary throughout the tree, similarly each generated rule varies in the number of conditions as well as the features used in the conditions. These rules are installed on the switch in the form of match-action table entries. Once these rules are installed, the switch may perform its classification inference. Considering a tree with depth n, the number of leaves in the tree would correspond to the number of rules generated from that tree, which is equivalent to 2^{n-1}, for a full binary tree. However, in practice, the decision tree does not have equal depth in all branches, leading to less rules than this upper bound.

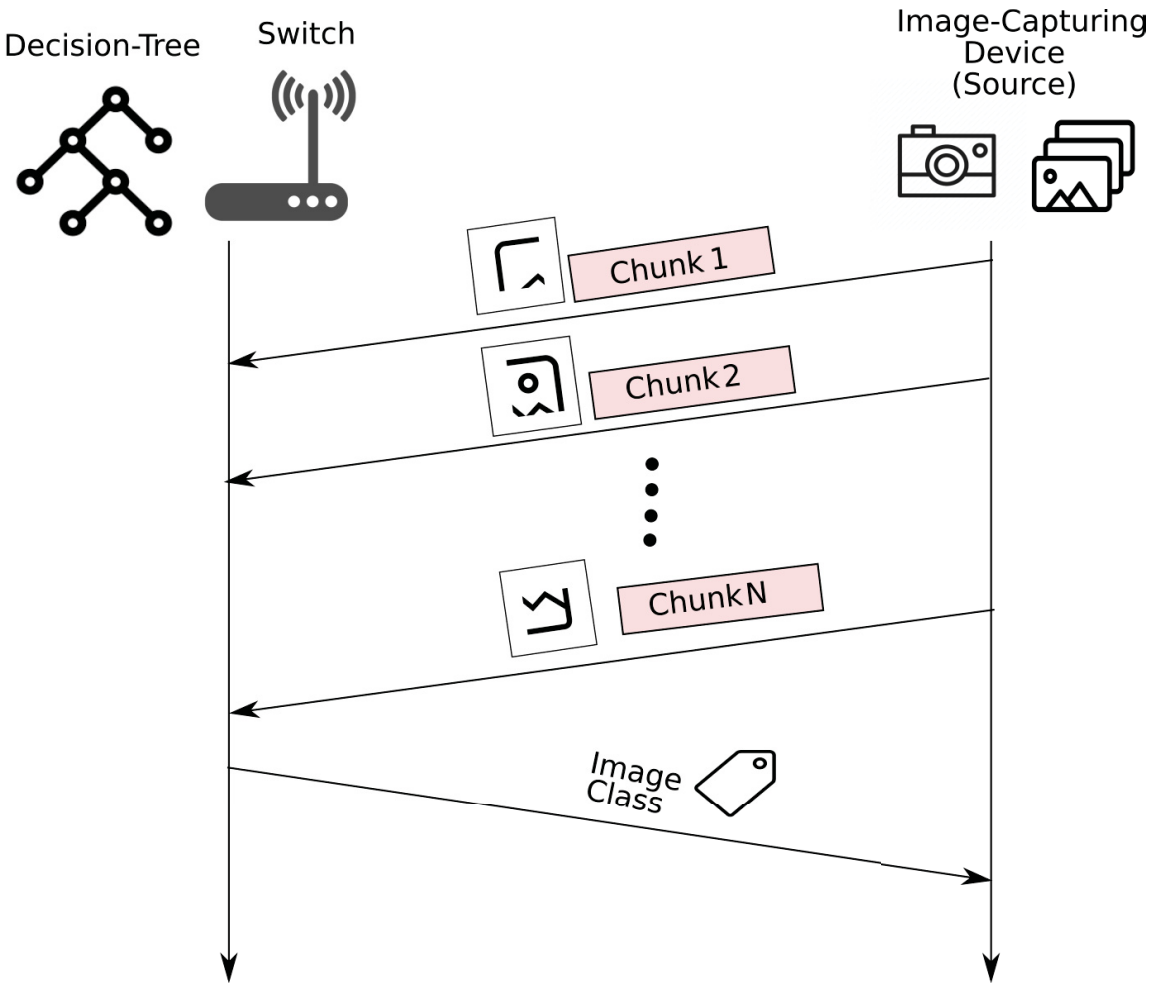


Figure 3.2: NetPixel Protocol

3.3.2 System Architecture

This section discusses in detail the architecture of the NetPixel system. First, we delve into the protocol of NetPixel that is designed to support the sending of images as chunks, and calculate the subsequent features from the chunk packets, on the switch itself. Next, we go through the implementation of the feature calculation and the pipeline structure of NetPixel.

3.3.2.1 NetPixel Protocol

For NetPixel, we have designed our own protocol for sending images, which performs at the application layer of the TCP/IP protocol stack. From Figure 3.2, we see how each packet carries a part of the image, which we refer to as a "chunk". This chunk

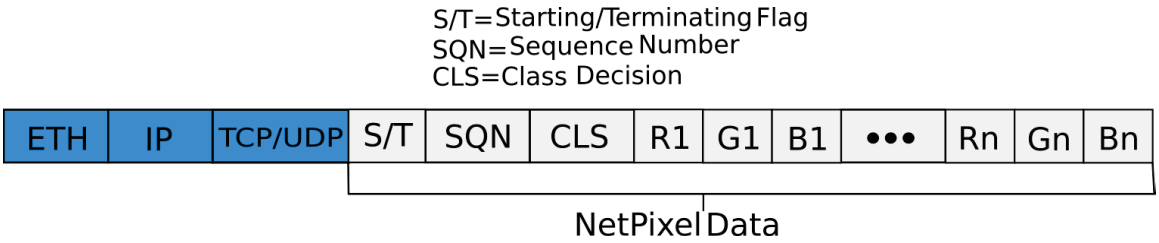


Figure 3.3: NetPixel Packet Format

is a squared group of n pixels from the image. For our work, we have mandated that the chunk size be 3x3. We evaluated the effects of different chunk sizes and found this size to be the best in terms of accuracy as well as the number of per-packet operations required at the switch. The larger the chunk size, the more operations are required to be performed at the switch. 1x1 would mean the lowest per-packet operations, and conversely higher packets, but this would mean the feature "number of edges" could not be calculated. Thus, 3x3 is the optimal selection. However, depending on network constraints that client devices may face, the chunk size may be adjusted to allow for sending of fewer packets.

The format for the packets that the client device sends to the NetPixel switch can be seen in Figure 3.3. First, we have the traditional networking stack headers, which are the Ethernet, IP and TCP/UDP headers. These are the traditional headers mandated by the TCP/IP protocol stack. Following that are the custom headers designed for NetPixel. The first `S/T` flags are used to denote the starting and terminating packets for the image. NetPixel is scale invariant and can perform accurately for any image size, so there is no fixed number of packets required to describe the whole image. This is followed by the `SQN` field which holds the sequence number of the packet. This is useful for reliable transmission of packets. However, in our work, we considered lossless transmission and did not employ any re-transmission or recovery protocol for lost packets. Then the `CLS` field is a placeholder field that is only written into once the decision tree inference has been completed, as it holds the class decision that is sent back to the client device. This is followed by the RGB component values, `R1`, `G1`, `B1`, ..., `Gn`, `Bn`, of each of the pixels in the chunk, making $3n$ fields each of 8-bit size.

Once all the chunks of the image have been sent, the decision tree inference begins to classify the image, as depicted in Figure 3.2. The final class decision is then written

Feature	Formula
Number of colors	$\|C\|$
Ratio of Pixels with low intensity	$(\sum_{i=1}^{s} x_i)/s : 0 < I(x_i) < 85$
Ratio of Pixels with mid intensity	$(\sum_{i=1}^{s} x_i)/s : 86 < I(x_i) < 171$
Ratio of Pixels with high intensity	$(\sum_{i=1}^{s} x_i)/s : 172 < I(x_i) < 255$
Contrast	$(I_{max} - I_{min})/(I_{max} + I_{min})$
Avg brightness	$(\sum_{i=1}^{s} I(x_i))/s$
Number of edge segments	$\sum_{j=2}^{h} 1 : L_j L_{j-1} < 0$

Table 3.1: Supported features. Symbols: x_i - a pixel; $I(x)$ - intensity of pixel x; I_{max}, I_{min} - highest and lowest intensity pixels, respectively; L_j - value of Laplacian filter when applied to chunk j; s - image size; h - number of chunks in an image; C - set of distinct colors in an image.

into the `CLS` field and re-transmitted back.

3.3.2.2 Feature Implementation

For this thesis, we selected features based on the work in [90], selecting 7 features among those mentioned in the paper. These are summarized as follows, as well as represented in Table 3.1.

- **Number of distinct colors:** The total number of distinct colors present in the image is noted as a feature, which is implemented using a Bloom filter. The Bloom filter works by assigning an index to every unique value, which then increments if that particular value has been found. For our work, the RGB values foe the pixels in the image are fed into a hash function to create the unique value, which then designates whether that color is unique or not.

- **Ratio of low, mid and high intensity pixels:** For the calculation of these features, the image is first converted to grayscale from its RGB color counterpart using Equation 3.1.

$$I = 0.299 * R + 0.587 * G + 0.114 * B \tag{3.1}$$

  This creates a value between 0 and 255 for the grayscale or intensity values. Each individual pixel value is grouped to either the low, mid or high categories which are the ranges 0-85, 86-170, and 171-255 respectively. Each sum then contains

the number of pixels belonging to that range, which is then proportioned against the total number of pixels to provide the ratio for each feature.

- **Contrast:** Contrast refers to the ratio between the maximum intensity/grayscale values in the image and the minimum values, which is calculated as $(I_{max} - I_{min})/(I_{max} + I_{min})$. Here I_{max} refers to the maximum intensity value in the image and I_{min} refers to the minimum intensity. These are iteratively updated as the whole image is processed and contrast is then calculated once the whole image has been received.

- **Average brightness:** The average brightness is represented by the average of all the individual intensity values of the image pixels. A counter is stored as the image is processed which adds the values of each grayscale pixel as it is converted. Once the whole image has been received, this is then divided by the number of pixels in the image to calculate the average brightness.

- **Number of edge segments:** Edges in an image are represented by a change in intensity values between neighboring pixels. Edges in the image are detected by use of a Laplacian filter., which is designed to identify where such changes in intensity occur as we apply it across the image. The number of such instances where changing responses occur are noted, and are summed up. For the purposes of our work, only vertical instances of such edges are considered, due to architectural limitations of the P4 platform.

These seven features are all calculated on the switch with the help of on-board registers, which act as stateful memory to store feature values, retrieve them, and update when necessary. Each of the features involves summation or incrementing values, so register values are updated as each packet is processed.

For the feature containing the total number of distinct colors in the images, a Bloom filter is implemented using these register. The registers are 1-bit size, and register indexes are generated using a hash function. However, considering the different possible combinations possible from RGB values, which would total around 16 million combinations, we make an empirical observation that most images from the variety of datasets evaluated contain less than 1 million colors, which would be much

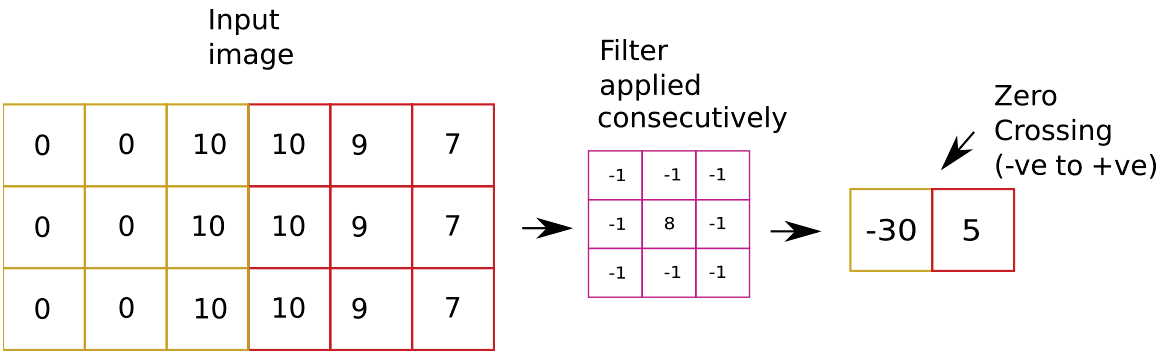


Figure 3.4: Laplacian edge detection example

more feasible to implement. Regardless, hash function collisions are still rarely possible, but we consider this negligible as machine learning based classifiers are robust to small deviations that may arise because of this.

For the subsequent ratio based features, as well as contrast and average brightness, we implement registers as counter, which are retrieved with each packet and simply updated according to which conditions are met by the pixel values in that packet, e.g., of low intensity or mid intensity. For contrast values, registers are only updated if the current intensity value is more than the current maxima or less than the current minima. Intensity values are added to the sum total register which is used to calculate the average brightness across the image.

The number of edges in the image requires the use of a Laplacian filter to be implemented. Once again, for this feature the grayscale intensity values are used. As each chunk arrives at the switch, the Laplacian filter is implemented. The response from the application of the filter is noted and compared to the response received from the previous chunk, to check if any zero crossings have occurred. Zero crossings indicate a change in sign (positive to negative or vice versa), and since the Laplacian filter functions as a derivative, this denotes a change in intensity, or presence of an edge [62, 63]. An example of this procedure is illustrated in Figure 3.4. The implementation of this feature has two limitations. Firstly, since these zero crossings are checked between consecutive chunks, there is no provision to detect horizontal edges. Secondly, the filter is applied on the whole of the chunk and responses between subsequent chunks are compared. This means that if any edge is present within the chunk, it would not be detected by the filter. However, since we are working with a 3x3, any edge contained within the chunk would be maximally 1 pixel thick, which

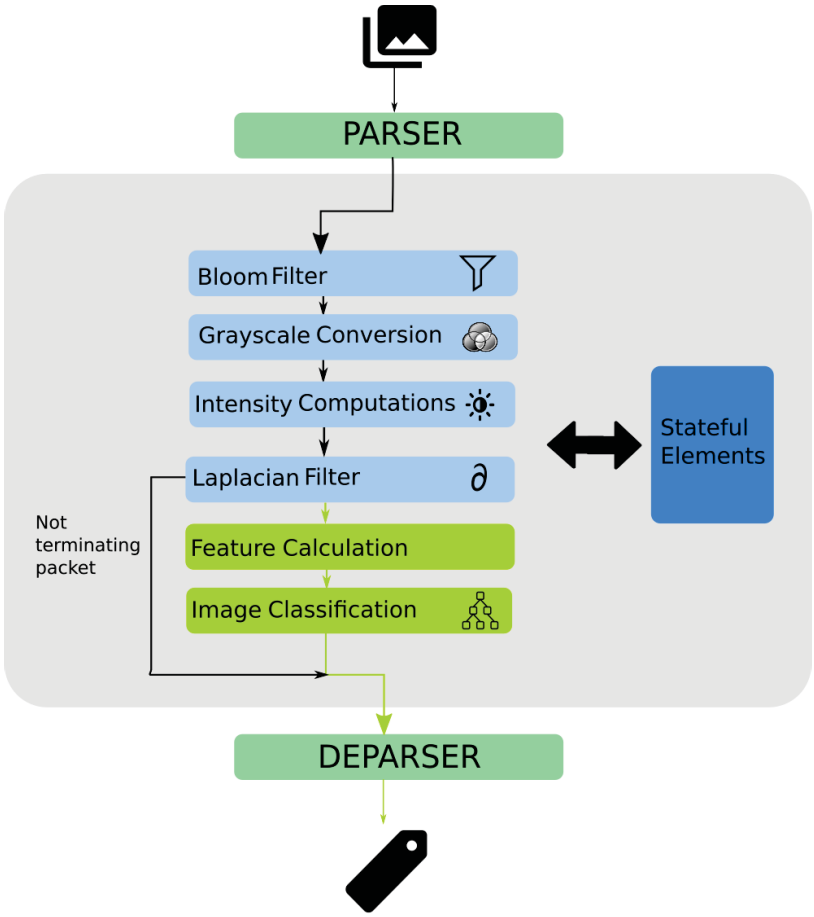


Figure 3.5: NetPixel Pipeline Structure

we can consider negligible.

3.3.2.3 NetPixel Pipeline

The NetPixel pipeline consists of a number of stages as outlined in Figure 3.5.

First the packet is received at the switch and the parser extracts the red, green and blue color components of the pixels in the packet chunk. The hash function generates a unique index based on the RGB value combination of each pixel, which is then used by the Bloom filter to denote whether this color is distinct.

Each pixel is then converted to its corresponding grayscale value using Equation 3.1. Each pixel in the chunk then goes through the following operations: the values are then compared to the defined ranges for low, mid and high intensity ratios and each counter is incremented accordingly.

Next, each value is compared to the current minima and maxima which would

be used to calculate the contrast of the image. These values are also added to the running sum of all intensities to calculate the average brightness.

Once done, the Laplacian filter is applied to the whole chunk and the number of edges incremented if required. If the packet is an intermediary one, it is deparsed and then discarded or forwarded, depending on implementation choice. However, if it is a terminating once, all feature values are retrieved from the registers and the final values consolidated.

The combination of final feature values are treated as keys for the match-action tables which contain the decision tree rules. The whole tree is deployed as a single-match action table, with the number of entries corresponding to the number of decision tree rules. Different authors in [97] and [18] implemented their classifiers using separate designs, such as one feature per table, or one tree per table. However, we found that this method does not create an exceptionally high number of rules while following a relatively straightforward approach.

The table entry keys are implemented as ranges for the feature values, considering that the decision tree rules generated by the trained classifier also present their tree conditions as ranges. Each table entry, or rule, contains a subset of the selected features where each feature may occur once, more than once, or not at all. All features are initially set to the range [0, T] by the network controller, where T is the largest possible value. If a feature does not appear in a particular rule, its range remains unchanged. If any feature occurs once, its ranges are set according to the sign accompanying the feature value, alongside the lowest or highest possible value for that feature. For example, if the rule states `contrast < 0.65` then the range, would be set as [0, 0.65]. Similarly, if a feature occurs more than once in a rule, the range is simply set to the intersection of the instances where it occurred. For example, if the rule states `brightness > 122` and `brightness < 225` the range would be set to [122,225].

3.4 Evaluation and Results

In this section, we cover environment used for the evaluation of NetPixel, in terms of the setup as well as the datasets used. We then present the results as well as analyze and discuss in detail the implications of the results.

3.4.1 Evaluation Setup

For the purpose of evaluating our system, we compared it to a baseline server implementation based on Python. NetPixel itself was implemented as a data-plane program on the P4-platform. The target architecture for our system was the v1model architecture, based on the BMv2 software for simulating programmable switches [16]. The network controller was simulated using a Python program which trains the decision tree-based classifier and then installs the rules on the switch using the CLI interface of BMv2. Client devices were also simulated using Python, based on the Scapy library [75], which was used to transmit the images using packets, as well as receive the final class decision from the switch. Both the network controller and the P4 program implement the fixed-point representation used for floating point values. This removes any need to convert values between Python and P4 due to the inability of the latter to support floating point numbers and operations natively.

The testbed used was a host equipped with a 4-core Intel Core i5-7400 CPU and paired with 8GB of RAM, which runs the Python scripts simulating the client devices as well as the BMv2 software switch. The same host was used to run the Python-decision tree baseline system against which NetPixel is compared.

Performance Metric: We considered accuracy as the standard against which our system would be tested in comparison to the baseline server implementation. While traditional classifier evaluations contain other metrics such as F1 score, precision and recall, we believe accuracy would give a suitable picture with regards to the feasibility of our system considering its platform of implementation and its constraints, as well as the applications that we wish to target with our system (latency critical applications where competitive accuracy is imperative). We also acknowledge that an in-depth evaluation of the latency benefits of our system could not be performed, due to the limitations of the BMv2 simulated software switch. The simulator cannot perform at the high packet processing speeds that are customary for switches such as the Tofino, and thus would not provide an appropriate evaluation in terms of computational latency.

Dataset	Image size	Training images	# of labels
MNIST	28x28x1	60000	10
CalTech101	Variable	9200	101
CalTech256	Variable	30000	256
ImageNet	Variable	20000	100

Table 3.2: Evaluated datasets.

3.4.2 Datasets

For evaluating our system, we considered four datasets, the parameters of which are outlined in Table 3.2. We found these datasets to be varied and contain differeing number of image types as well as classes, to provide a balanced evaluation of the system.

MNIST: This dataset consists of images which are handwritten digits, numbered from 0-9 thus creating 10 classes. The images are all in gray-scale and of fixed image size [53].

CalTech101 and CalTech256: These datasets contain a varied array of images of different types such as vehicles, people, animals, landscapes, etc. CalTech256 [34] is an extension of CalTech101 [30] containing 256 classes compared to 101, as well as having more images per class. All images are colored but have vastly different sizes.

ImageNet: ImageNet is one of the largest databases of images [26], containing millions of images spanning over thousands of classes. However, for the purposes of our evaluation, we selected images from randomly chosen 100 different classes, with each class containing 200 images. These images are also colored and of varying sizes.

Dataset	DT-Python	NetPixel
MNIST	92.45%	85.00%
CalTech101	96.50%	92.78%
CalTech256	91.11%	87.28%
ImageNet	90.36%	86.73%

Table 3.3: Classification accuracy for different datasets.

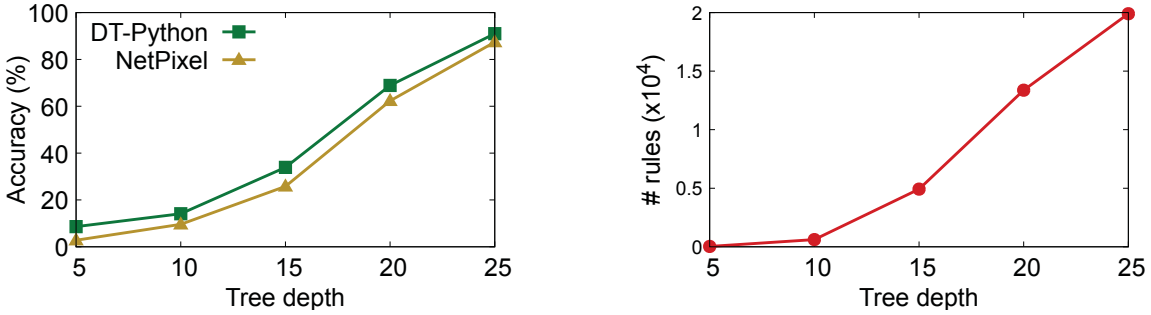
3.4.3 Results and Discussion

We noted during our evaluation that there is a small drop in accuracy between the Python-based implementation and the P4 implementation, as outlined in Table 3.3. The largest drop was observed while testing the MNIST dataset, with a decrease of around 7.45% and the smallest drop was observed for the CalTech101 dataset with a 3.72% decrease in accuracy. This difference is expected due to approximations that had to be made considering the architectural limitations of the P4 platform, with regards to floating point number support as well as the implementation of operations such as general division and floating point multiplication.

Relative to the other datasets, the MNIST dataset has a larger drop despite having a substantially lower number of classes, which would allow for more room when establishing decision boundaries between classes. However, most of the features we selected are based on color, while all the MNIST images are grayscale, thus making it difficult for these features to perform optimally in distinguishing between classes. On the other hand, the CalTech101 dataset as well as the CalTech256 dataset have a wide array of color images and the classes themselves are also well distinguishable from each other, which is the reason that our selected features worked better to distinguish them. While this is true for the ImageNet dataset as well, the classes here did not vary as much as the two aforementioned, thus leading to lower performance despite having significantly less classes than CalTech256.

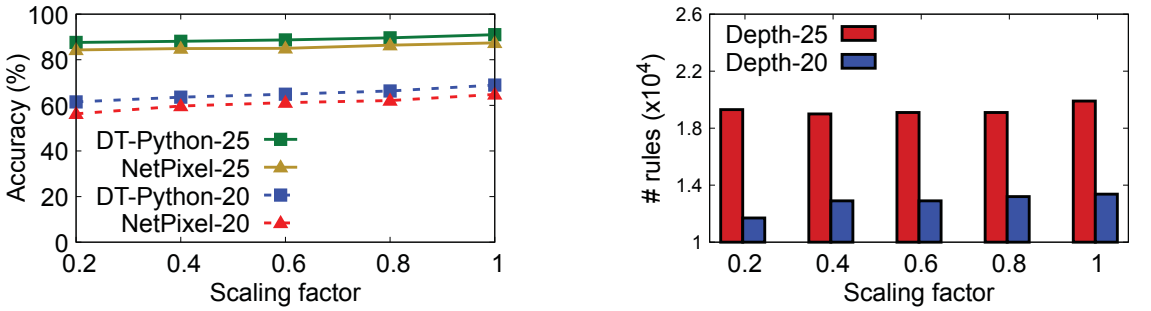
To further evaluate the feasibility of our system on the P4 platform for programmable switches, we performed a number of additional tests, varying different parameters of the classifier, to check both the performance impacts as well as the impact of the system in terms of it's memory footprint. Here memory footprint refers to the number of rules that need to be installed on the switch for the decision tree. As the number of registers required for the classifier are well-defined, the only variable in terms of memory is the number of entries that need to be installed in the match-action table present on the switch. For this purpose, we use the CalTech256 dataset to observe the effects of varying the tree depth, i.e, how many levels and therefore nodes are present in the tree classifier, as well as varying the image sizes sent from the client device, in terms of image scale.

Impact of tree depth on accuracy and memory: We can see a natural



(a) Impact on accuracy (b) Impact on memory footprint

Figure 3.6: Effects of varying tree depth of the classifier



(a) Impact on accuracy (b) Impact on memory footprint

Figure 3.7: Effects of varying image resolution

increase in accuracy of the classifier as we increase tree depths, as seen in Figure 3.6a. This is due to the fact that more nodes and leaves are available to establish decision boundaries between classes. While this tree depth is pre-defined before classifier training as the maximum depth, it follows that not all paths in the tree reach this maximum depth. Many paths may reach their leaves prematurely relative to others, based on the feature values involved, leading to a variance in the number of nodes and consequently, number of rules. We see the sharpest rise between depths 15 and 20, where achieving an accuracy of over 80% requires a tree depth of at least 23. On the other hand, in terms of memory impact, we see a similar trend of increasing number of rules as the tree depth increases in Figure 3.6b. However, this increase is fairly linear, despite the number of possible nodes increasing at an exponential rate. This is also attrributed to the fact that not every path in the tree utilizes the maximum depth.

Impact of image scale on accuracy and memory: Image scales refer to the

downsizing of original images, which may be required due to bandwidth constraints on the client device and it would be too expensive to send images on their original scale. Image scales were varied on the client side using a bicubic interpolation before sending them to classifier switch. In particular, we wanted to test whether the classifier is affected greatly by images being smaller in dimension than usual. For this experiment, we tested tree depths of both 20 as well as 25 to obtain a clearer insight. Despite expecting smaller images to greatly affect the accuracy of the classifier, we see a nominal decrease in accuracy as we scale down the image. From Figure 3.7a, we see that even at a scale of 0.2 the classifier does not lose much in terms of accuracy for both depths 20 and 25. This is primarily due to the nature of the selected features being invariant to scale, since most involve ratios or bounded properties across the whole image. We can see a similar trend in terms of memory, where an increase in size from a scale of 0.2 to 1 results in only a 14% increase for number of rules required. This is due to more nodes being required to establish the decision boundaries between classes. However, for a tree depth of 25, there are already enough nodes to establish these boundaries, thus practically no change is seen between image scales of 0.2 and 1.

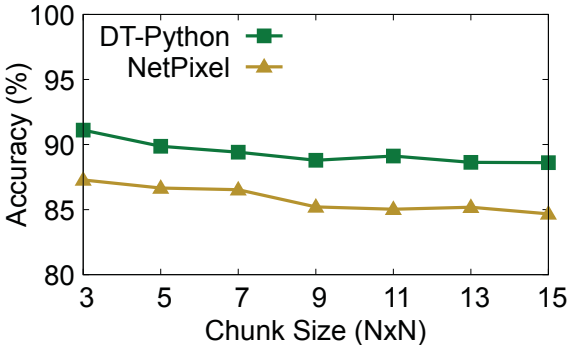


Figure 3.8: Impact of varying chunk sizes on accuracy

Impact of chunk size: We also evaluated the effect of different chunk sizes for the individual packets which are sent to the switch. A chunk size of 3x3 performs best while accuracy gradually decreases as we increase chunk size, as shown in Figure 3.8. This parameter is mostly relevant to the *number of edges in the image* feature, which contains a Laplacian filter proportional to the chunk size. With increase chunks and filter, less and less edges are detected in the image, since the filter cannot detect edges

present within the filter itself. This leads to a smaller range in the number of possible values for this feature, making it more difficult to establish definitive boundaries between the classes since all other features retain their values. Nevertheless, as the impact is only on a single feature among seven, the decrease is not exceptionally large.

Chapter 4

Neural Network-based System: Design and Evaluation

As we have explored throughout this work, the convolutional neural network (CNN) is the state-of-the-art classifier for images and is the natural extension for image classification on the data-plane. In this chapter we discuss the architecture of our extended model, p4CNN, as well as the implementation considerations that had to be made owing to the resource constrained nature of the target platform. We provide an overview of the system, followed by the binarized neural network model design, the protocol as well as system pipeline. Following these, we evaluate the system against the server based based implementations, both binarized and non-binarized.

4.1 System Architecture and Design: Neural Network-based System

While the decision tree-based system performed at a competitive accuracy compared to the server-based implementation, it is still not the optimal classifier for images. With more and more advancements in machine learning and subsequently deep learning, convolutional neural networks (CNNs) have emerged as the state-of-the-art standard classifier for images [23], able to perform with much higher accuracy, albeit with a much more complicated and resource-intensive architecture than decision trees.

Traditionally, this would make implementing CNNs de facto onto a programmable switch quite challenging. Besides the architectural limitations of the switch in terms of mathematical operations and supporting floating point numbers, we also no longer have the simplicity that decision trees provide, replaced by the vastly more complex and resource-heavy structure of the neural network. However, we design our system, p4CNN, with a simpler architecture and a binarized approach, which allows it to perform at a respectable accuracy while circumventing these issues.

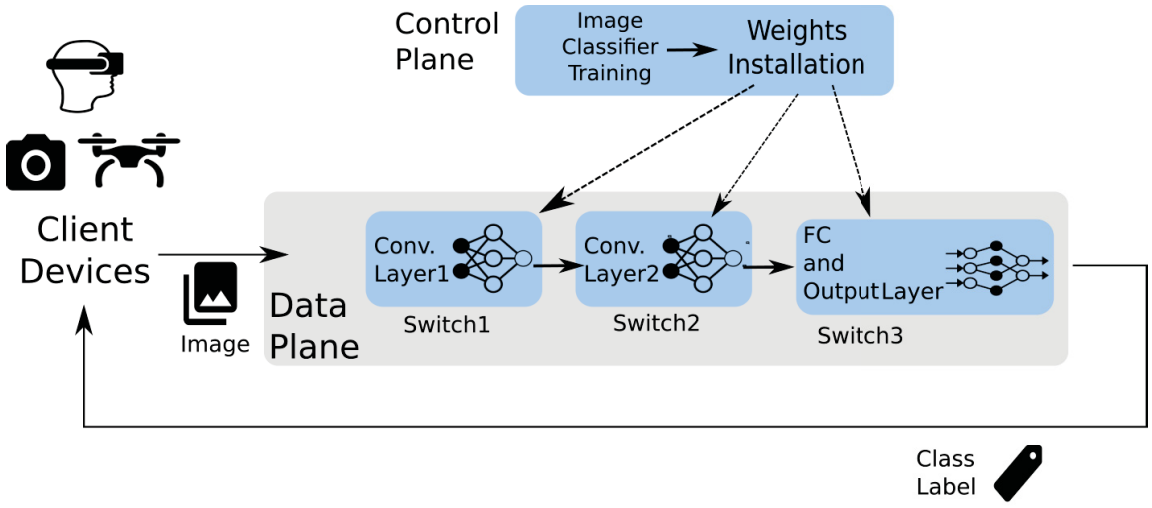


Figure 4.1: p4CNN Overview

4.1.1 Overview

Our extended neural network-based system, p4CNN has a similar framework to NetPixel. However, there are some significant changes in terms of the number of switches required. An overview of the system can be viewed in Figure 4.1.

4.1.1.1 Programmable Switches

Similar to NetPixel, the classification of the image sent by the client device occurs at the data plane. However, contrary to a single switch performing classification as in NetPixel, p4CNN distributes its neural network over 3 different switches. The exact architecture of the neural network and its detailed distribution is outlined in Section 4.1.2.1. The first switch receives the packets for the image, which are once again delivered in the form of chunks. This switch then parses the chunk for the RGB color information of the pixels in the chunk and converts them to their grayscale values. At this point, the switch takes these grayscale values and inputs them to the first convolutional layer that is present. This layer performs the convolutional operations

over each chunk of the image as it is received and stores the response values on registers on the switch. This process continues until the whole image has been transmitted to the first switch. Once all the response values have been calculated, we implement an average pooling layer. This works by pooling neighboring pixels into a single average value, thereby reducing the dimensions of the response values. The response values are then written onto the final packet which already contains placeholders in its headers, and forwards to the next switch.

Once the next switch receives the packet from the first, it parses the packet and extracts the information about the response values from the first convolutional layer. This is then similarly fed to the second convolutional layer and once the multiplications are completed the response values are once again written onto the packet and forwarded to the next switch.

The final switch contains the fully connected layer and the output layer of the neural network. It parses the packet from the second switch and feeds the response values to the fully connected layer. The fully connected layer passes its output to the final output layer which then determines the class decision of the image, based on the values from its constituent neurons. This class decision is written onto the received packet and re-transmitted back to the client device, which may then take an action based on the class of the image, subject to the application.

4.1.1.2 Network Controller

In p4CNN, the controller has a similar role to NetPixel. It is responsible for the training of the neural network, thereby updating the weight values which will be used in the convolutional and fully connected layers of the network. Once the training of the network is complete, it will install these weight values on the switches themselves, through register writing operations. Each of the layers have their own structure and weight values. Further details into the training method of the neural network, including architectural parameters, optimizers and loss functions are detailed in Section 4.1.2.1.

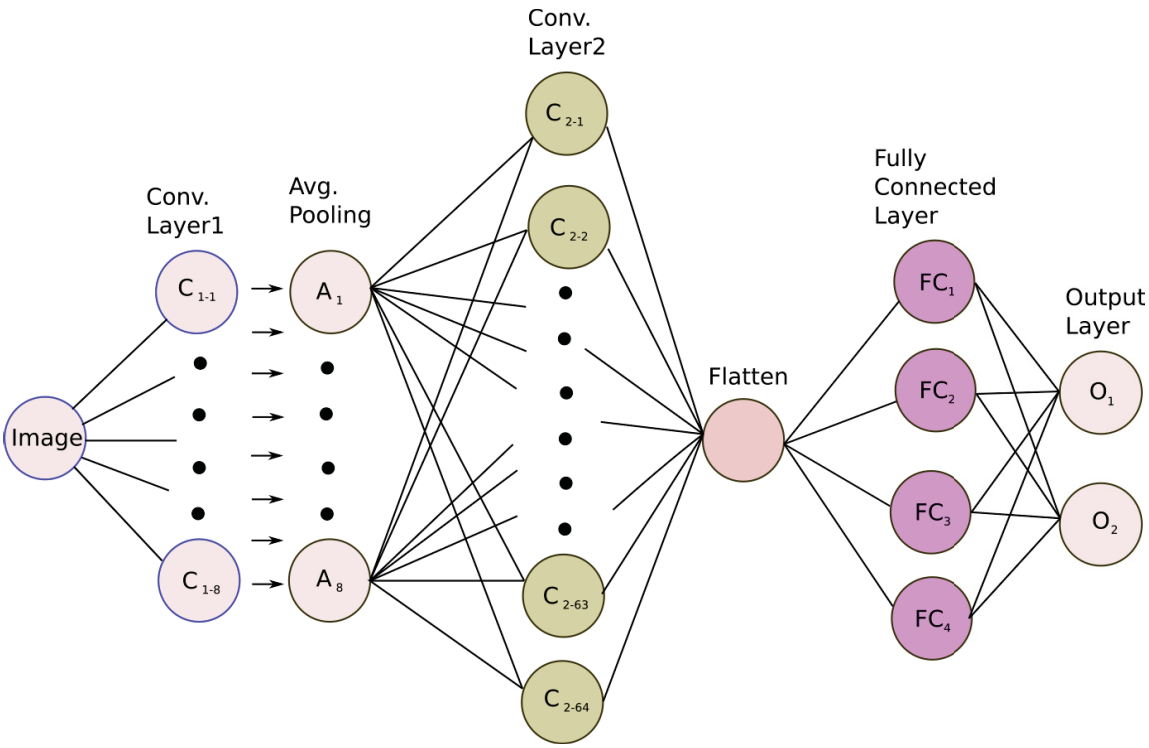


Figure 4.2: p4CNN architecture

4.1.2 System Architecture

This section provides a detailed description of the architecture implemented in our neural network, as well as the extension of the NetPixel protocol that we use for the p4CNN protocol, followed by a workflow outline of the p4CNN pipeline.

4.1.2.1 Neural Network Architecture

Considering the resource constraints of the programmable switch as well as operational limitations of the P4 platform, we have selected a binarized approach for our neural network. For the purposes of our work, we assume each image to be of the square shape, with each dimension of n pixels. The network consists of a convolutional layer with 8 neurons, followed by an average pooling layer on the first switch. This is fed to the 2nd convolutional layer on the next switch, consisting of 64 neurons. The fully connected layer, consisting of 4 neurons and the output layer consisting of the same number of neurons as number of classes in the image data, is implemented on the final switch. In terms of the particular number of neurons used in each layer,

we investigated the effects of varying neuron levels as well as adding and removing layers, and found the present combination to be the most optimum choice. Lesser neurons in each of the layers leads to a more significant loss of accuracy, which further degrades the performance of the system. An increased number of neurons leads to complexity that is beyond the capabilities of the target platform, the programmable switch, which would require larger registers, both in terms of amount as well as size, and would also exceed the maximum transmission unit size of the packets being transmitted. Across all the neurons, we implement a binarized approach for input data as well as weight values in the individual neurons. The activation functions used in each convolutional layer is the ReLu function, [5], discussed in Section 2.1.5. Each component layer is discussed in detail as follows, and an overview of the architecture can be seen in Figure 4.2:

Convolutional Layer 1: This is the initial layer of the neural network. The convolutional layer implemented here consists of 8 neurons. Each of these neurons contains a 3x3 filter kernel. These kernels contain the binary weights for that particular neuron. For each chunk that arrives at the switch, after it has been converted to it's binary format from the grayscale format, it is convoluted against the kernels from each neuron sequentially, and the responses are stored on board the switch. Thus the number of chunks equates to the registers needed for each neuron on the switch. By the time all the chunks have arrived and the whole image has passed through the convolutional layer(CL), there will then be 8 feature maps. 'Feature Maps' refer to the response values that have been generated from convoluting each kernel against the input image. Each map of size $(n-1)*(n-1)$, where n refers to the dimension of the image on each side, thus shows the areas of the image that particular feature has been detected pertaining to the feature kernel, i.e., where the highest response for that particular feature kernel has been detected.

Since both the image values and the weights are in the binary format, and to reduce computational overhead from dozens of multiplication tables on the switch, we implemented the convolutional multiplications using the XNOR-*popcount* approach for binary multiplication [45]. This approach is used when multiplication is a resource-heavy operation, which is the case in our system as we need to perform a large number of multiplications on the switch. Each multiplication approximation involves

Step	**Example**
XOR two bit strings A and B	$1011 \bigoplus 1001$
Results stored in C	0010
NOT C	1101
Popcount	3
Check if greater than $N/2$	$3 > (4/2)$
If greater output is 1, else 0	1

Table 4.1: XNOR-popcount operation for 2 bit strings A and B, where bit-length, N=4

an XNOR operation between the two operands, resulting in a bit-string. This is followed by a *popcount* operation which designates a value of 1 to the output if more than half of the resulting string constitutes set bits, which are 1s. An example of this operation can be seen in Table 4.1. First, the XOR operation is applied on the bit strings A=1011 and B=1001, resulting in the bit-string C=0010. Thereafter, the NOT operation is applied, resulting in 1101. Here, the *popcount* operation is applied, to check the number of set bits, which is 3 in this case. This number is then compared to half of the bit-string length (it is compared to 2 in this case. Since the value is greater, the result is set as 1. Else it would be set as 0. This process is repeated for every multiplication in our neural network.

In traditional binarization of neural networks, the binary values are 1 and -1. Element-wise multiplication (or dot product) of bit-strings containing these 1s and -1s results in a value which is passed to the accumulation, which then adds up the individual values to present a single value as the output of the entire operation. Since this accumulation is basically the summation of the 1s and -1s, the result can vary between $-N$ (if there are N number of -1s), and N (if there are N number of 1s). Here, the value N refers to the number of total bits in the bit-string. This operation can then be summarized as finding $p - n$ where p is the number of 1s and n is the number of -1s, and $N = p + n$. When replacing the multiplication operation with the XNOR-*popcount* method, usually an extra step is introduced, where the resultant value is calculated as $2p - N$, after rearranging the above equation and replacing to solve for $p - n$.

However, in our architecture, we use the values 1 and 0 instead of 1 and -1. This

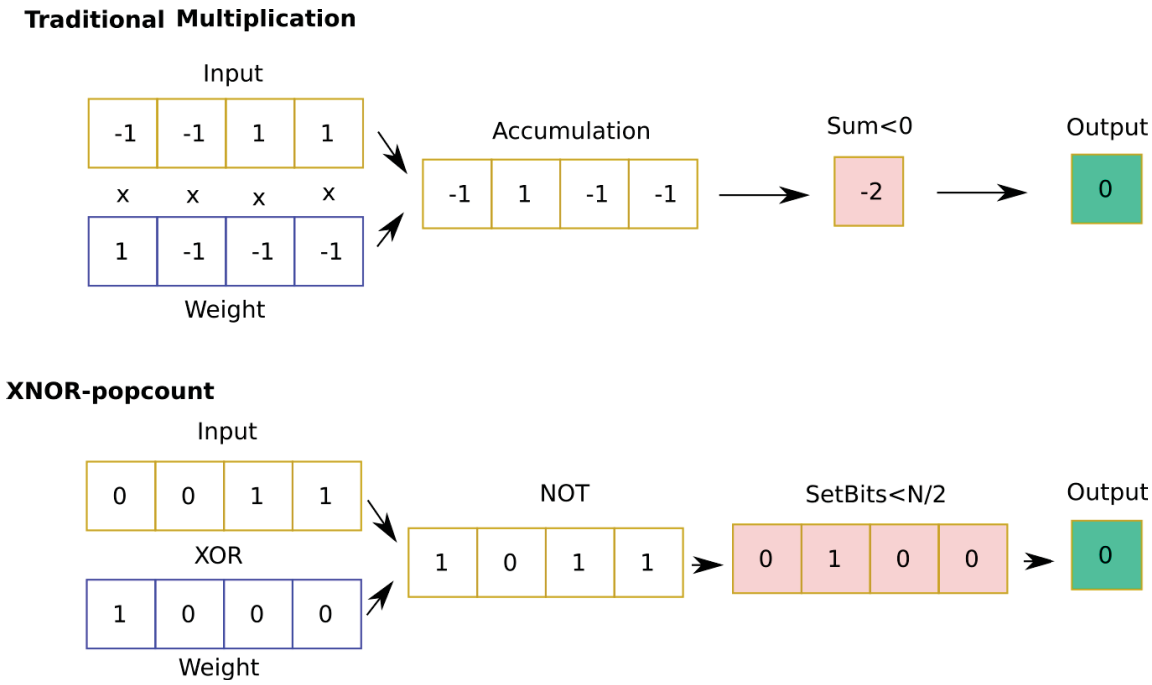


Figure 4.3: Comparison between multiplication and XNOR-popcount

means that the negative contribution of the -1s that would have been present in the accumulated and resultant value are absent. This is however inconsequential, because it is not necessary to record the exact value of the resultant, simply its sign. If positive, the resultant value is set as 1, and if negative it is set as 0. This assumption holds mathematically, because, if there are more set bits, or 1s, then the resultant value is always positive, and in reverse, if there are more unset bits, or -1s, the result would always be positive. As we are comparing the number of set bits against half of the bit-string length, it clarifies whether the resultant value would have been negative or positive, had we used -1s instead of 0s, eradicating the need to compute the magnitude of the resultant. An example of the comparisons between both methods can be seen in Figure 4.3

Once all the chunks have been received by the switch and convoluted, against all neurons, the response values which were stored in the on-board registers are retrieved.

Average Pooling Layer: This layer is also implemented on the first switch, and is used to reduce the dimensions of the feature map responses from the first convolution. For this layer, we utilize a 2x2 kernel, which takes a neighborhood of 4 values and averages them to a single value. Since the average operation is not natively

available on the P4 platform, and we are using binary values, we simulate the average operation by simply checking whether more than half the values in the neighborhood are 1, resulting in a response value of 1, else a response value of 0. This operation is applied to all neuron responses. Once completed, the values are written onto the headers of the terminating packet and forwarded to the second switch, which contains the next convolutional layer.

Convolutional Layer 2: The second convolutional layer operates in a similar approach to the first layer. However, this layer contains 64 neurons, each with their own weight values. Contrary to the first convoutional layer, which received a single channel input in the form of a grayscale image, the neurons in this layer must apply convolution to an 8-channel input, mirroring the 8 neurons of the first layer. Thus, this layer contains much more weight values than the first layer. Once again, convolution is approximated using the XNOR-*popcount* method, and the response values are written onto the packet that was originally received. This is then forwarded to the next switch, containing the fully connected layer.

Fully Connected Layer: At the fully connected layer, our architecture consists of 4 neurons. However, traditionally fully connected layers in neural networks are preceded by a flatten layer, which converts the multiple feature maps of the convolutional layer into a single row vector of all the values. In our model, this layer is not needed, due to the fact that the values from the convolutional layer are store consecutively in the header fields of the packet, from which they can be accessed in the same manner as a flattened row vector. At the fully connected layer, the weight values are bit-strings with the same dimensions as that of the combined feature maps, and once again the XNOR-*popcount* approach is used to simulate element-wise multiplication. This results in a single value from each neuron, and the resulting bit-string is passed to the output layer.

Output Layer: The output layer consists of the same number of neurons as the number of classes possible from the images. The response string from the fully connected layer is multiplied for the final time with the weights of the output layer, and the the output neuron with the highest resulting value is designated as the class of the image.

Loss Function: For the p4CNN neural network, we implemented two different

loss functions based on the number of classes used in the training dataset. For experiments involving binary classification, or 2 classes, we use the *squared-hinge* loss function, which is primarily used for specifically 2 class problems [13]. This function is used for "maximum margin" binary classification, in order to completely differentiate between two classes, without the need for a probabilistic value. For training datasets involving more than 2 classes, we use the *multiclass-hinge* or *categorical-hinge* loss function, which simply performs the *squared-hinge* function over the whole dataset while incrementally considering each of the classes as the target class, and the rest as the non-target class. Once done, the loss values for all the classes are summed up for the final loss value, which is then used to update the weight values accordingly.

4.1.2.2 p4CNN Protocol

For the p4CNN system, we have extended the protocol designed for NetPixel, discussed in Section 3.3.2. However, some key differences between the protocols are outlined as follows:

- For NetPixel, the image was sent as separate and distinct chunks with no overlap between consecutive chunks. However, for P4CNN, we must note the response values for every pixel in the image, surrounded by its neighboring pixels. Therefore, each chunk is sent as a 3x3 pixel neighborhood. This approach, while necessary, increases the number of packets required to describe the whole image.

- The p4CNN headers are much larger in size, due to having placeholders for the convolutional responses from the first two layers.

The format of the packets for p4CNN has been illustrated in Figure 4.5. Here, the RGB values for each of the pixels are followed by placeholders for the convolutional response values that will be transmitted between the first and second switch, as well as the second and third switch. There are 64 placeholders because this is the maximum amount of neurons contained in a convolutional layer in our network. Each placeholder can hold a maximum of 144-bits, which brings the total payload size to 9216-bits, which is within the maximum transmission unit (MTU) size of 12000-bits. Once all the chunks of the image have been received at the first switch and the calculations completed, as shown in Figure 4.4, the packet carrying the responses from the first

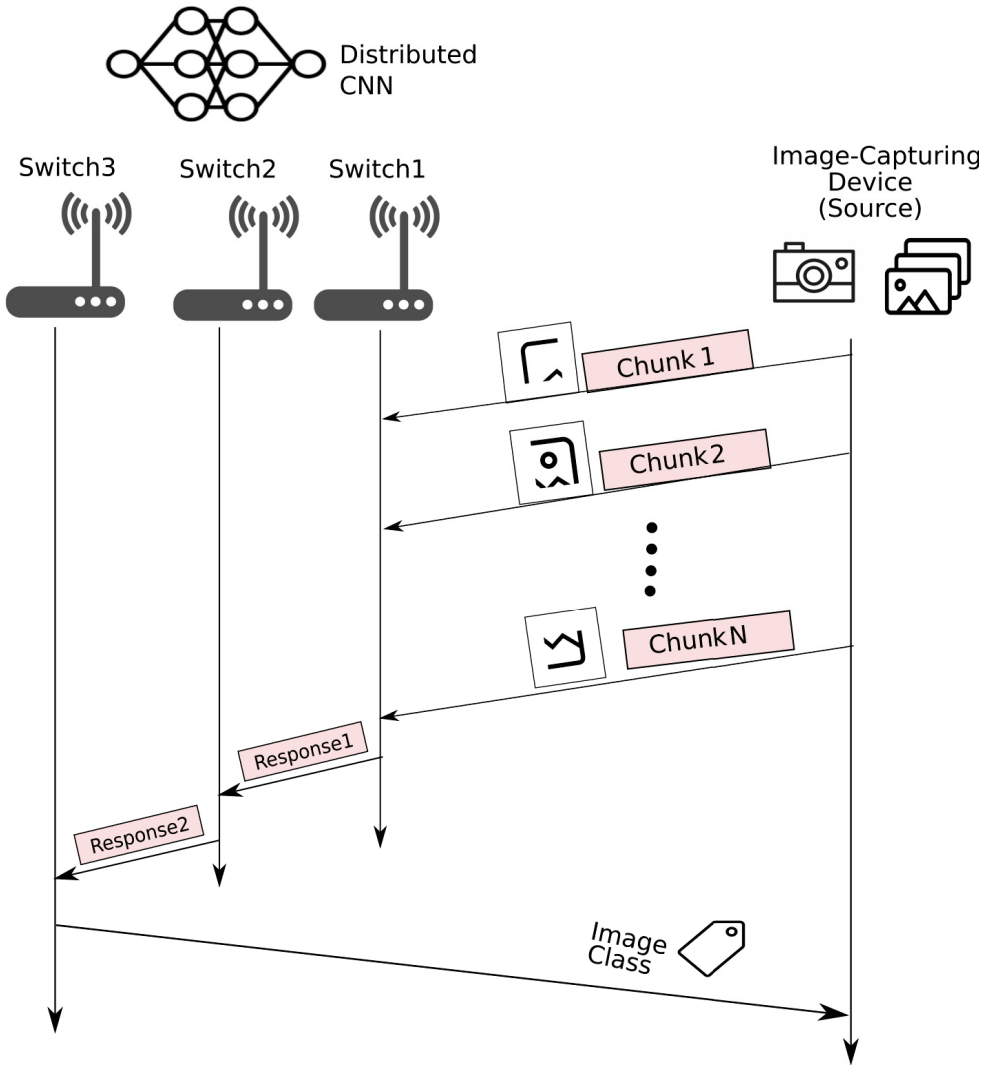


Figure 4.4: p4CNN Protocol

convolutional layer, denoted as *Response1*, is transmitted to the second switch. That switch subsequently calculates the responses from the second convolutional layer and writes them onto the packet, denoted as *Response2*, which is then forwarded to the third switch. This switch then completes the neural network computation and, once the inference is complete, writes the class decision onto the `CLS` field, and re-transmits it back to the client device.

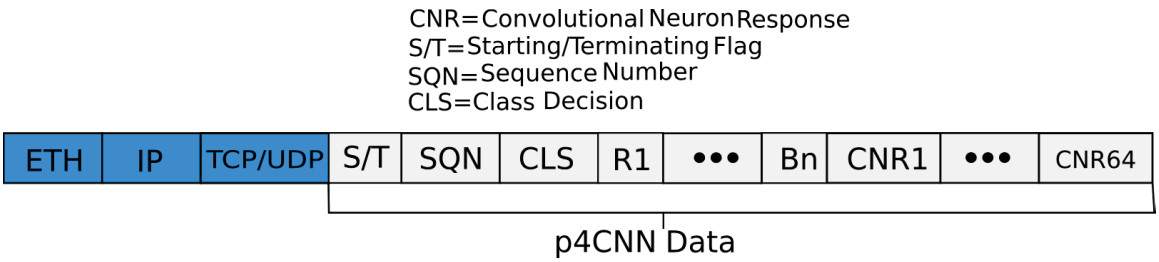


Figure 4.5: p4CNN Packet Format

4.1.2.3 p4CNN Pipeline

The pipeline of p4CNN involves three switches as opposed to a single switch, as outlined in Figure 4.6. First, the packet containing an image chunk arrives at the first switch, and is parsed to extract the RGB information describing the pixels in the chunk. These are then converted to their grayscale values, which are subsequently used as input for the convolutional layer. As each packet arrives, the grayscale values are subjected to the weight values, or filters, at each neuron subsequently. These responses are noted and stored in the registers on-board the switch, and these values are then retrieved once the final packet has arrived at the switch. However, if the current packet is not a terminating or final packet, the packet is discarded or forwarded to another destination. If it is a terminating packet, the retrieved values are then converted to the responses from the average pooling operation as described in Section 4.1.2.1. This is the response calculation for the first switch, which is then written on to the terminating packet, deparsed, and forwarded to the next switch.

At the second switch, once again the response values are retrieved from the packet header and convoluted against each neuron and its constituent filters, consecutively per neuron. Since there are multiple channels and multiple filters per neuron here, there are significantly more computations on this switch. However, no average pooling is applied here, so the responses are simply written onto the packet and forwarded.

At the third switch, the responses are multiplied against the fully connected neurons, which have a single set of weights, and then the output layer. Once the class decision is finalized, the class label is the final output of the pipeline.

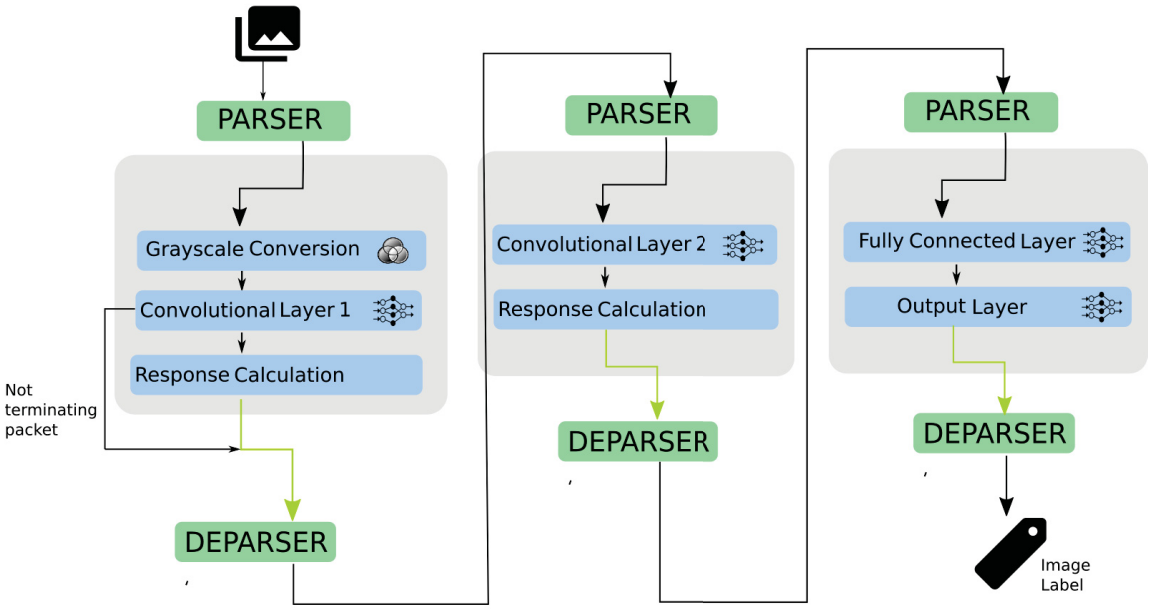


Figure 4.6: p4CNN Pipeline Structure

4.2 Evaluation and Results

In this section, we provide an account of the procedures performed to evaluate the p4CNN system. However, it is notable that the evaluations performed for this system are not as extensive as those performed for NetPixel. Primarily, the architectural implementation of the neural network on the data-plane is extremely rigid, with little to no room for modifications in the form of tweaking. This means that we could not evaluate all datasets in their default dimensions, but instead, they had to be scaled down to a feasible size. Furthermore, as our evaluation is simply a proof-of-concept for the feasibility of convolutional neural networks on programmable switches, We only evaluate the performance of p4CNN to its counterparts based on the **accuracy** metric.

4.2.1 Evaluation Setup

Owing to the rigid nature of the implementation on the data-plane, we first evaluated our system using only a single dataset. The MNIST dataset [53] was chosen, due to its small and fixed image size, having a resolution of 28x28 pixels, while all the other datasets, CalTech101, CalTech256 and ImageNet had varying image sizes, with most of the images over 500x500 pixels. Based on our observations, we then selected a suitable subset for the other datasets, considering the performance trends of the MNIST dataset.

In terms of the MNIST dataset, we tested the accuracy of p4CNN over the whole dataset as well as a subset of the dataset, in terms of classes available. For these subsets, we took around 20 combinations within each class amount (20 combinations for 2 classes among 10, and so on), noting the accuracy performance for each. Then, we finalized the accuracy for that class amount as the average accuracy among these combinations.

The system was once again compared to a baseline server implementation based on Python. However, we compared the system to both the binarized Python architecture as well as the compete Python architecture. p4CNN was programmed on the P4 platform, with the target architecture being v1model and based on the BMv2 software for simulating a programmable switch [16]. Python was also used to simulate the network controller which trains the neural network, collects the weight values from each of the neural network layers and installs them onto the respective switches. Once again, we simulate client devices for transmitting images and receiving class decisions using Python, and the Scapy library [75].

An identical testbed to the NetPixel evaluation was used, which runs the Python scripts for the host and controller, as well as the BMv2 software switch environment. The server-based implementation in Python was also run on this testbed.

4.2.2 Results and Discussion

The initial evaluation on the MNIST dataset can be seen in Table 4.2. It is evident here that the non-binarized Python implementation has a near-perfect accuracy across all subsets of the data, as well as the whole dataset itself. This shows the performance advantage of using convolutional neural networks for image classification tasks, and

Class No.	Non-binarized Python	Binarized Python	p4CNN
2 classes	100.00%	99.87%	89.45%
3 classes	99.90%	86.58%	71.45%
5 classes	98.51%	72.36%	55.42%
8 classes	97.95%	55.49%	31.84%
10 classes	95.87%	35.15%	4.30%

Table 4.2: Classification accuracy with varying class amounts from the MNIST dataset. The Python columns refer to server-based implementations

Dataset	Non-binarized Python	Binarized Python	p4CNN
MNIST	100.00%	99.87%	89.45%
CalTech101	99.29%	97.53%	83.81%
CalTech256	99.60%	98.38%	84.31%
ImageNet	98.71%	96.83%	82.02%

Table 4.3: Classification accuracy for each dataset (binary classification)

why these type of networks are popular in this application.

The binarized Python implementation performs almost similarly for the binary class subset, however, performance starts to degrade as the number of classes increase, with an approximately 13% decrease in accuracy with 3 classes, and further increasing discrepancy as the number of classes covers more of the dataset. The reason for this is the loss in information in terms of weight values, which begins to amplify as we convolve the input further and further through the network.

For the P4 implementation, we have the architectural limitations of the P4 platform, coupled with the information loss already present in binarized networks, which leads to a further decline in accuracy as we progressively increase the number of classes in the dataset. However, for the binary classification problem with 2 classes, we achieve an acceptable loss in accuracy of 10% with the binarized server implementation, and an absolute accuracy of 89.45%. We note here that a further evaluation in the remaining datasets should suitably be limited to the problem of binary classification, involving only two classes, as the drop in accuracy would be too great for any practical application for any higher number of classes.

For the remaining datasets we see a similar trend for the non-binarized server implementations in Table 4.3. There are drops in accuracy similar to the MNIST

dataset across the other datasets as well. However, the drops are larger than that of the MNIST dataset. This could be due to the fact that downscaling the images from their original size caused a loss of information that was significant. In the case of NetPixel, the use of scale invariant features was able to offset this loss of information, from using images at a lower scale than there original. However, this is not the case for p4CNN.

Based on the above results, we envision the use of this system in applications where images can be grouped into binary classes. This may involve the presence or absence of some object or entity in the image, which can then be used for any applications where detection is a requirement.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

To conclude this work, we have presented two model for image classification on the data-plane, using a programmable switch. First, we present NetPixel, a decision tree-based classifier that operates on a single switch, classifying a wide range of images. NetPixel performs to within an 8% discrepancy in performance with its server based implementation, while providing the latency gains of performing in-network computing. These gains are important for latency-critical and bandwidth-hungry applications. Second, we present p4CNN, an extension of the NetPixel platform that uses a convolutional neural network-based classifier, which operates as a distributed system on three different systems, and performs with 10% discrepancy of the server-based implementation.

We evaluated NetPixel on the BMv2 software switch against the Python implementation, testing it on a number of datasets with varying types of images. Based on this evaluation we showed that NetPixel performs at a competitive accuracy for image classification in a myriad of applications. Next, we evaluated the neural network-based system, p4CNN, using a similar environment and testbed, on a singular dataset against a binarized and non-binarized server implementation and found that it also performs at a competitive accuracy for the binary classification-related applications.

5.2 Future Work

There are a number of avenues with which we can extend our work in the future. The most evident is the deployment of both the NetPixel and p4CNN models in the hardware switch, namely the Intel Tofino switch [4]. However, this itself comes with a number of challenges, primarily resource limitation. The Tofino is much less lenient in terms of the number of pipeline stages that can be used, as well as memory

usage. This may be solved by further optimization in the form of feature priority selection, to simplify the decision tree model as well as reduce resource requirements. This extension is imperative for the analysis of the latency benefits that our model will provide, given the high packet processing capabilities of the Tofino switch while remaining close to the end-devices in latency-critical applications.

Alongside deployment of the classifier models onto physical hardware, increasing the scalability of the model to service multiple client devices is also possible. This may be done by increasing the number of registers linearly with the number of devices connected to the switch, with each group of registers being assigned to each device. The packet format used in our protocol must also be modified to accommodate a field for *device id*, which would be used to track the source of the image packets. Registers may also be reused once the classification of a request from a particular device has been completed. Of course, this scalability is limited by the resources available on the switch itself.

Further extensions can be made by converting our CNN-based model into decision tree rules, whilst maintaining the accuracy gains of a deep learning approach. In its current state, our neural network architecture is fairly rigid, due to the limitations of the platform. However, if we can convert the neural network model to a decision tree implementation that can be ported to the P4 platform, we can be much more flexible in terms of neural network models, and further increase accuracy even for images with a higher number of classes than those evaluated. This conversion would also help to solve the "black-box" nature of convolutional neural networks, as the transition to decision tree rules would significantly increase interpretability of the internal workings of the classifier, allowing investigations into further optimizations.

Image classification is one of the fundamental tasks of computer vision, and its basic principles allow extension to other CV tasks such as object detection and image segmentation. Furthermore, we note that the proposed classifier models in this work are not exclusive to images as data, but can be ported to accommodate other data types with fairly trivial modifications. This may include discrete valued data such as network traffic, or more complex data such as audio and sound samples, opening the avenue for latency critical applications which involve speech recognition and natural language processing.

Bibliography

[1] Netpixel. https://github.com/PINetDalhousie/netpixel.

[2] p4cnn. https://github.com/hisham-sid/p4CNNFinal.

[3] Autonomous cars will generate more than 300 tb of data per year. https://www.tuxera.com/blog/autonomous-cars-300-tb-of-data-per-year/, 2017. Accessed, 2021-03-19.

[4] Intel® tofino™ 2 p4 programmability with more bandwidth. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, 2018. Accessed, 2021-03-19.

[5] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[6] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. Mobile cloud computing for computation offloading: Issues and challenges. *Applied Computing and Informatics*, 14(1):1–16, 2018.

[7] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376, 2015.

[8] Muhammad Raisul Alam, Mamun Bin Ibne Reaz, and Mohd Alauddin Mohd Ali. A review of smart homes—past, present, and future. *IEEE Transactions on Systems, Man, and Cybernetics, part C (applications and reviews)*, 42(6):1190–1203, 2012.

[9] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormandi, George E Dahl, and Geoffrey E Hinton. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[10] Rick Archibald and George Fann. Feature selection and classification of hyperspectral images with support vector machines. *IEEE Geoscience and Remote Sensing Letters*, 4(4):674–677, 2007.

[11] Abdelhadi Azzouni, Nguyen Thi Mai Trang, Raouf Boutaba, and Guy Pujolle. Limitations of openflow topology discovery protocol. In *2017 16th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, pages 1–3, 2017.

[12] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. Distributed sdn control: Survey, taxonomy, and challenges. *IEEE Communications Surveys & Tutorials*, 20(1):333–354, 2017.

[13] Peter L Bartlett and Marten H Wegkamp. Classification with a reject option using a hinge loss. *Journal of Machine Learning Research*, 9(8), 2008.

[14] Michael Batty, Kay W Axhausen, Fosca Giannotti, Alexei Pozdnoukhov, Armando Bazzani, Monica Wachowicz, Georgios Ouzounis, and Yuval Portugali. Smart cities of the future. *The European Physical Journal Special Topics*, 214(1):481–518, 2012.

[15] Tahajjat Begum, Israat Haque, and Vlado Keselj. Deep learning models for gesture-controlled drone operation. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–7, 2020.

[16] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[17] L Brieman, J Friedman, RA Olshen, CJ Stone, D Steinberg, and P Colla. Cart: Classification and regression trees, 1995.

[18] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *CoRR*, abs/1909.05680, 2019.

[19] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya R Dulloor. Scaling video analytics on constrained edge nodes. *arXiv preprint arXiv:1905.13536*, 2019.

[20] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proc. IEEE*, 107(8):1655–1674, 2019.

[21] Min Chen and Yixue Hao. Task offloading for mobile edge computing in software defined ultra-dense network. *IEEE Journal on Selected Areas in Communications*, 36(3):587–597, 2018.

[22] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649. IEEE, 2012.

[23] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-second International Joint Conference on Artificial Intelligence*, 2011.

[24] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[25] KR Darshan and KR Anandakumar. A comprehensive review on usage of internet of things (iot) in healthcare system. In *2015 International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT)*, pages 132–136. IEEE, 2015.

[26] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. Ieee, 2009.

[27] Shuiguang Deng, Longtao Huang, Javid Taheri, and Albert Y Zomaya. Computation offloading for service workflow in mobile cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3317–3329, 2014.

[28] Damu Ding, Marco Savi, and Domenico Siracusa. Estimating logarithmic and exponential functions to track network traffic entropy in p4. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020.

[29] Avri Doria, Jamal Hadi Salim, Robert Haas, Hormuzd M Khosravi, Weiming Wang, Ligang Dong, Ram Gopal, and Joel M Halpern. Forwarding and control element separation (forces) protocol specification. *RFC*, 5810:1–124, 2010.

[30] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *2004 Conference on Computer Vision and Pattern Recognition Workshop*, pages 178–178. IEEE, 2004.

[31] Thomas Fevens, Israat Haque, and Lata Narayanan. Randomized routing algorithms in mobile ad hoc networks. In *Proceedings of the IFIP International Conference on Mobile and Wireless Communication Networks*, 2004.

[32] Glenn Fung and Jonathan Stoeckel. Svm feature selection for classification of spect images of alzheimer's disease using spatial information. *Knowledge and Information Systems*, 11(2):243–258, 2007.

[33] A. Giridhar and P.R. Kumar. Toward a theory of in-network computation in wireless sensor networks. *IEEE Communications Magazine*, 44(4):98–107, 2006.

[34] Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 object category dataset. 2007.

[35] Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[36] Tianmei Guo, Jiwen Dong, Henjian Li, and Yunxing Gao. Simple convolutional neural network on image classification. In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*, pages 721–724, 2017.

[37] Kevin Gurney. *An Introduction to Neural Networks*. CRC press, 2018.

[38] Israat Haque and Nael Abu-Ghazaleh. Wireless software defined networking: A survey and taxonomy. *IEEE Communications Surveys & Tutorials*, 18(4):2713–2737, 2016.

[39] Israat Haque and Chadi Assi. OLEAR: Optimal localized energy aware routing in mobile ad hoc networks. In *Proceedings of the 2005 IEEE International Conference on Communications*, ICC '05, 2005.

[40] Israat Haque and Chadi Assi. Localized energy efficient routing in mobile ad hoc networks. *The Willey Journal of Wireless and Mobile Computing*, 7(6):781–793, August 2007.

[41] Israat Haque, Chadi Assi, and William Atwood. Randomized energy-aware routing algorithms in mobile ad hoc networks. In *Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, MSWiM '05, 2005.

[42] Israat Haque, Saiful Islam, and Janelle Harms. On selecting a reliable topology in wireless sensor networks. In *Proceedings of the 2015 IEEE International Conference on Communications*, ICC '15, 2015.

[43] Israat Haque, Mohammed Nurujjaman, Janelle Harms, and Nael Abu-ghazaleh. SDSense: An agile and flexible SDN-based framework for wireless sensor networks. *The IEEE Transactions on Vehicular Technology*, 68(2):1866 – 1876, February 2019.

[44] Israat Haque and Dipon Saha. SoftIoT: A resource-aware sdn/nfv-based iot network. *The Elsevier Journal of Network and Computer Applications*, 193, Nov 2021.

[45] Lei Jiang, Minje Kim, Wujie Wen, and Danghui Wang. Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2017.

[46] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. ACM.

[47] Kimon Karras, Evangelos Pallis, George Mastorakis, Yannis Nikoloudakis, Jordi Mongay Batalla, Constandinos X. Mavromoustakis, and Evangelos Markakis. A hardware acceleration platform for ai-based inference at the edge. *Circuits, Systems, and Signal Processing*, 39(2):1059–1070, Feb 2020.

[48] Parham M. Kebria, Abbas Khosravi, Syed Moshfeq Salaken, and Saeid Nahavandi. Deep imitation learning for autonomous vehicles based on convolutional neural networks. *IEEE/CAA Journal of Automatica Sinica*, 7(1):82–95, 2020.

[49] István Ketykó, László Kecskés, Csaba Nemes, and Lóránt Farkas. Multi-user computation offloading as multiple knapsack problem for 5g mobile edge computing. In *2016 European Conference on Networks and Communications (EuCNC)*, pages 225–229. IEEE, 2016.

[50] Vinay Kolar, Israat T. Haque, Vikram P. Munishwar, and Nael B. Abu-Ghazaleh. Ctcv: Coordinated transport of correlated videos in smart camera networks. In *24th International Conference on Network Protocols (ICNP)*. IEEE, 2016.

[51] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[52] Carson Kuzniar, Miguel Neves, and Israat Haque. Poster: Accelerating encrypted data stores using programmable switches. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–2, 2020.

[53] 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.

[54] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. In *Artificial Intelligence and Statistics*, pages 562–570. PMLR, 2015.

[55] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of ACM SIGCOMM*, SIGCOMM '20, page 359–376, New York, NY, USA, 2020. ACM.

[56] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3367–3375, 2015.

[57] Juan Liu, Yuyi Mao, Jun Zhang, and Khaled B Letaief. Delay-optimal computation task scheduling for mobile-edge computing systems. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 1451–1455. IEEE, 2016.

[58] Lei Liu, Chen Chen, Qingqi Pei, Sabita Maharjan, and Yan Zhang. Vehicular edge computing and networking: A survey. *Mobile Networks and Applications*, 26(3):1145–1168, 2021.

[59] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGARCH Comput. Archit. News*, 45(1):795–809, April 2017.

[60] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[61] Dengsheng Lu and Qihao Weng. A survey of image classification methods and techniques for improving classification performance. *International journal of Remote sensing*, 28(5):823–870, 2007.

[62] D Marr and E Hildreth. Theory of edge detection. *Proceedings of the Royal Society B, London*, 207:187–217, 1980.

[63] Pierre Moulin. 4.2 - multiscale image decompositions and wavelets. In AL BOVIK, editor, *Handbook of Image and Video Processing (Second Edition)*, Communications, Networking and Multimedia, pages 347–359. Academic Press, Burlington, second edition edition, 2005.

[64] Ali Yadavar Nikravesh, Samuel A Ajila, Chung-Horng Lung, and Wayne Ding. Mobile network traffic prediction using mlp, mlpwd, and svm. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 402–409. IEEE, 2016.

[65] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[66] Qiaofeng Qin, Konstantinos Poularakis, Kin K. Leung, and Leandros Tassiulas. Line-speed and scalable intrusion detection at the network edge via federated learning. In *2020 IFIP Networking Conference (Networking)*, pages 352–360, 2020.

[67] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[68] J Ross Quinlan. Generating production rules from decision trees. In *IJCAI*, volume 87, pages 304–307. Citeseer, 1987.

[69] J.R. Quinlan. 5 - probabilistic decision trees. In Yves Kodratoff and Ryszard S. Michalski, editors, *Machine Learning*, pages 140–152. Morgan Kaufmann, San Francisco (CA), 1990.

[70] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1421–1429, 2018.

[71] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29(9):2352–2449, 2017.

[72] Jinke Ren, Yinghui He, Guan Huang, Guanding Yu, Yunlong Cai, and Zhaoyang Zhang. An edge-computing based architecture for mobile augmented reality. *IEEE Network*, 33(4):162–169, 2019.

[73] Tiago Gama Rodrigues, Katsuya Suto, Hiroki Nishiyama, and Nei Kato. Hybrid method for minimizing service delay in edge cloud computing through vm migration and transmission power control. *IEEE Transactions on Computers*, 66(5):810–819, 2016.

[74] Shiva Rowshanrad, Sahar Namvarasl, Vajihe Abdi, Maryam Hajizadeh, and Manijeh Keshtgary. A survey on sdn, the future of networking. *Journal of Advanced Computer Science & Technology*, 3(2):232–248, 2014.

[75] Rohith Raj S, Rohith R, Minal Moharir, and Shobha G. Scapy- a powerful interactive packet manipulation program. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–5, 2018.

[76] Ihab S. Mohamed. *Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques*. PhD thesis, 09 2017.

[77] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, 1991.

[78] Dipon Saha, Meysam Shojaee, Michael Baddeley, and Israat Haque. An Energy-Aware SDN/NFV architecture for the internet of things. In *IFIP Networking 2020 Conference (IFIP Networking 2020)*, Paris, France, June 2020.

[79] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery.

[80] Ikuro Sato, Hiroki Nishimura, and Kensuke Yokoi. Apac: Augmented pattern classification with neural networks. *arXiv preprint arXiv:1505.03229*, 2015.

[81] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[82] Dimitrios N Serpanos, Leonidas Georgiadis, and Tasos Bouloutas. Mmpacking: A load and storage balancing algorithm for distributed multimedia servers. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 170–174. IEEE, 1996.

[83] Sebastiano B Serpico and Lorenzo Bruzzone. A new search algorithm for feature selection in hyperspectral remote sensing images. *IEEE Transactions on Geoscience and Remote Sensing*, 39(7):1360–1367, 2001.

[84] Neha Sharma, Vibhor Jain, and Anju Mishra. An analysis of convolutional neural networks for image classification. *Procedia Computer Science*, 132:377–384, 2018.

[85] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[86] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[87] Sonia Singh and Priyanka Gupta. Comparative study id3, cart and c4. 5 decision tree algorithm: a survey. *International Journal of Advanced Information Science and Technology (IJAIST)*, 27(27):97–103, 2014.

[88] Yushan Siriwardhana, Pawani Porambage, Madhusanka Liyanage, and Mika Ylianttila. A survey on mobile augmented reality with 5g mobile edge computing: Architectures, applications, and technical aspects. *IEEE Communications Surveys & Tutorials*, 23(2):1160–1192, 2021.

[89] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, 14(11):4724–4734, 2018.

[90] Pavel Surynek and Ivana LuksovÃ¡. Automated Classification of Bitmap Images using Decision Trees. *Polibits*, pages 11 – 18, 12 2011.

[91] Sabine Süsstrunk, Robert Buckley, and Steve Swen. Standard rgb color spaces. In *Color and Imaging Conference*, volume 1999, pages 127–134. Society for Imaging Science and Technology, 1999.

[92] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994–12000, 2009.

[93] Yaman Umuroglu, Nicholas J. Fraser, and Giulio Gambardella. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 65–74, New York, NY, USA, 2017. Association for Computing Machinery.

[94] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. A machine learning approach to routing. *arXiv preprint arXiv:1708.03074*, 2017.

[95] Mei Wang and Weihong Deng. Deep face recognition: A survey. *Neurocomputing*, 429:215–244, 2021.

[96] Siming Wang, Zehang Zhang, Rong Yu, and Yan Zhang. Low-latency caching with auction game in vehicular edge computing. In *2017 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1–6. IEEE, 2017.

[97] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 25–33, New York, NY, USA, 2019. ACM.

[98] Xiao Yang, Zhiyong Chen, Kuikui Li, Yaping Sun, Ning Liu, Weiliang Xie, and Yong Zhao. Communication-constrained mobile edge computing systems for wireless virtual reality: Scheduling and tradeoff. *IEEE Access*, 6:16665–16677, 2018.

[99] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the ACM SIGCOMM*, SIGCOMM '20, page 126–138, New York, NY, USA, 2020. ACM.

[100] Ke Zhang, Yuming Mao, Supeng Leng, Alexey Vinel, and Yan Zhang. Delay constrained offloading for mobile edge computing in cloud-enabled vehicular networks. In *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*, pages 288–294. IEEE, 2016.

[101] Ke Zhang, Yuming Mao, Supeng Leng, Quanxin Zhao, Longjiang Li, Xin Peng, Li Pan, Sabita Maharjan, and Yan Zhang. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE Access*, 4:5896–5907, 2016.

[102] Nan Zhang, Su Ruan, Stéphane Lebonvallet, Qingmin Liao, and Yuemin Zhu. Kernel feature selection to fuse multi-spectral mri images for brain tumor segmentation. *Computer Vision and Image Understanding*, 115(2):256–269, 2011.

[103] Y. Zhang, Jixian Zhang, Xianfeng Zhang, Hongan Wu, and Ming Guo. Land cover classification from polarimetric sar data based on image segmentation and decision trees. *Canadian Journal of Remote Sensing*, 41:39–49, 04 2015.