DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Intelligence Distribution Network

Citation

Teerapittayanon, Surat. 2019. Intelligence Distribution Network. Doctoral dissertation, Harvard University, Graduate School of Arts & Sciences.

Permanent link

http://nrs.harvard.edu/urn-3:HUL.InstRepos:42106913

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Intelligence Distribution Network

A DISSERTATION PRESENTED
BY
SURAT TEERAPITTAYANON
TO
THE HARVARD JOHN A. PAULSON SCHOOL OF ENGINEERING
AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
APRIL 2019

©2019 – SURAT TEERAPITTAYANON

ALL RIGHTS RESERVED.

Dissertation Advisor: Professor H.T. Kung Surat Teerapittayanon

Intelligence Distribution Network

ABSTRACT

The applications of deep neural networks (DNN) have grown in number in recent years, but there is a need to better support their deployment in the field. This dissertation introduces Intelligence Distribution Network (IDN), a platform specifically designed to support this increasing demand for DNN inference computations. IDN is a fault-tolerant decentralized peer-to-peer (p2p) network that delivers fast, low-cost, scalable execution of DNN models. IDN does this by optimizing DNN inference for this setting, and then distributing the computation among spare computing resources nearby.

IDN reduces inference time using BranchyNet, a technique to modify the standard DNN structure with "exit branches" at certain locations throughout the network, allowing some samples to exit the computation early and decreasing the inference time.

IDN scales inference to the entire IDN p2p network using Distributed Deep Neural Network (DDNN), a method that allows DNN inference to be distributed among a computing hierarchy consisting of devices, edges, and clouds.

IDN achieves fault tolerance and high availability for DNN inference with ParallelNet. ParallelNet can generate multiple DNN model of various sizes, each fitting a device of varying compute capacity. Each device can execute its model independently in parallel. ParallelNet allows DNN inference to be run on more devices independently, improving the fault tolerance and availability of DNN inference.

IDN incentivizes the creation of a large number of high quality DNN models that can be shared among users using DaiMoN, a Decentralized Artificial Intelligence Model Network. DaiMoN improves upon today's limited, redundant, and siloed DNN model sharing structures by incentivizing peers to collaborate and share DNN models to improve the accuracy of a given problem in a decentralized manner.

Finally, to manage how IDN is used, the ComputeSwap protocol incentivizes peers to participate in the network via an optimistic debt-repayment structure that probabilistically results in repayment based on credits earned for executing inference and debts acquired by using the network. This encourages peers to service inference computations that are needed by other peers by maintaining a balance to not accumulate too much debt or give too little credit.

Prior Publications

Parts of Chapter 3 have appeared in the following:

Teerapittayanon, Surat, Bradley McDanel, and H. T. Kung. "BranchyNet: Fast inference via early exiting from deep neural networks." Pattern Recognition (ICPR), 2016 23rd International Conference on. IEEE, 2016.

Parts of Chapter 4 have appeared in the following:

Teerapittayanon, Surat, Bradley McDanel, and H. T. Kung. "Distributed deep neural networks over the cloud, the edge and end devices." Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. IEEE, 2017.

Contents

DEDICATED TO MY FAMILY.

Acknowledgments

This dissertation would not have been possible without the help, support and kindness of many individuals.

First and foremost, I would like to thank H.T. Kung for his unwavering patience, advice and encouragement throughout my Ph.D. I am glad to be an advisee of H.T. Since the very first year of graduate school, H.T. have guided me through researches. He taught me how to think analytically and find "angles" to attack each problem and ignore nonessential content or "junk" as he would call it. Thank you, H.T. for believing in me and giving me the opportunity to be part of your research "heaven."

I am indebted to Professor Jim Waldo. With his vast experience in distributed systems, he has shown me a different point of view that has helped improve this dissertation greatly.

I am also indebted to Professor Flavio Calmon. His brilliant advice is precious and has help me tremendously in shaping the overall thesis.

I am extremely grateful for the support and friendship of Marcus Comitter. Thank you for the many fun nights at Border Cafe and uncountable number of margaritas

after many exhausting research days. Marcus has been a great friend. I am glad to have Marcus in my Ph.D. experience. Marcus, you are the best!

I am very grateful to have the support and friendship of Bradley McDanel. He played a major role in the work presented in this dissertation. Thank you for days of fun, out-of-this-world discussion.

I would like to thank my other collaborators and lab mates, including Kevin Chen, Steve Tarsa, Michael Crouse, Miriam Cha, Youngjung Gwon, Philippe Tillet, Sai Zhang, among others. Without them, my experience would not have been the same.

I am infinitely grateful to Harvard for providing a thriving environment for me to study and grow as a person in many aspects of my life throughout my Ph.D.

Finally, I would like to thank my family. I would not have been able to make it through my Ph.D. without your love, support and sacrifice on a daily basis.

1

Introduction

The modern era of the World Wide Web (WWW) started in 1990, when the first web page was served on the Internet by Tim Berners-Lee. Eight years later, Akamai, which would become the operator of the world's largest Content Distribution Network (CDN), was founded in 1998 in Cambridge, MA. CDN works by caching web content closer to end users in order to increase their quality of experience (QoE). When

Akamai launched its CDN in 1999, the Internet was still in its early days. At the end of 1999, Akamai had an annual revenue of almost $4 million and managed 3,000 servers across the globe [1]; these numbers grew to $2.3 billion and 200,000 servers in 2016 [2]. MarketsandMarkets [3] forecasts that the global CDN market will grow to be over $30 billion by 2022. CDNs have changed the way content is distributed and improved users' Quality of Experience (QoE) significantly. This dissertation introduces Intelligence Distribution Network (IDN), the intellectual successor of CDNs for the age of machine learning. Just as CDN fundamentally changed the way content was distributed, IDN will change the way artificial intelligence computation is distributed and greatly improve users' QoE for the ever-growing number of AI applications.

IDN focuses on distributing and optimizing deep neural networks (DNNs), the main algorithm used for today's artificial intelligence (AI) applications. In 2012, Alex Krizhevsky won the ImageNet [4] challenge by a large margin using a DNN. Since then, many AI applications have seen a significant improvement in their accuracy by using DNNs. Today, an increasing number of companies have deployed DNNs in their products. These products include smart cities, autonomous vehicles, virtual assistants, virtual reality (VR), augmented reality (AR), and internet-of-things (IoT) devices such as smart appliances, cameras, drones, and robots. In 2016, the global AI market was worth just over $3 billion. However, we are still in the early days of the age of DNNs.

By 2022, the global AI market is expected to be worth over $37 billion [5]. As DNN usage increases, there is a rising need for an infrastructure capable of deploying and scaling DNN models for inference.

The Internet is organized in a hierarchical fashion. Tier 1 Internet service providers (ISPs) are at the top of the pyramid. Web content from the server flows up the ISP tree and down to the end users, as shown in Figure 1.1. A CDN places caches at various points in the hierarchy to reduce the latency of retrieving the content and the server load, and ultimately improve users' QoE. In an analogous fashion, as shown in Figure 1.2, IDN places DNN engines at various points in the hierarchy closer to end users to reduce the latency of DNN computation and the server load, and ultimately improve users' QoE. In addition, IDN allows peer-to-peer (p2p) communication among nearby devices; any computing devices can join the IDN network and host the computation to further increase the network's computing capacity and dissipate the computation load.

IDN provides an infrastructure consisting of nearby peer computing devices. By leveraging spare computational resources from peer devices, edge nodes, and cloud servers nearby, IDN puts computation closer to the device that needs it, with the aim to substantially reduce inference time and cost of deployment, scale the computation to nearby devices, and increase fault tolerance. IDN comes with ecosystems and incentive mechanisms that encourage people to share both high-quality DNN models

and their spare computing resources.

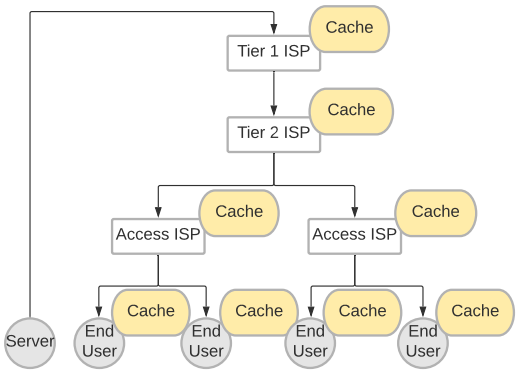

Figure 1.1: Content Distribution Network (CDN)

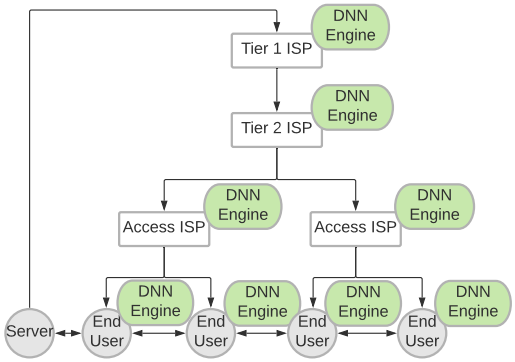

Figure 1.2: Intelligence Distribution Network (IDN)

1.1 Motivations

We have seen growing demand for and successes of DNN models for various tasks such as classification, detection, scene segmentation, and language understanding, with new use cases discovered in every single industry. However, putting these DNN models to use and making them available and accessible to a large population requires the right tools and scalable infrastructure.

In the case of web content, the content is created once, but it is distributed many more times. Similarly, in case of DNN models, once a model is trained, a process that may happen only once, it is used for inference many more times, sometimes thousands or millions of times a day. Inference is the process in which the trained DNN is used to

predict outputs from input samples. Inference use cases are becoming widespread with the advent of next generation technologies such as autonomous vehicles, edge-based inference systems (smart appliances, cameras, etc.), augmented reality (AR), virtual assistants and virtual reality (VR). The trends in inference are toward sophisticated real-time services where models are complex and responsiveness is critical. Those services may include, for example, DNN-based speech recognition, natural language processing, and object detection and classification.

To support these growing inference demands, application developers typically service inference computation demands of DNN models using either centralized cloud servers or by making the models smaller so that they fit entirely on a single end device. Each of these approaches have their own pros and cons. On one hand, running DNN models on a cloud server supports complex models, but puts a huge load and thus cost of the cloud server on the application developers. Further, this adds end-to-end latency to the application as it must communicate with a distant clouds. On the other hand, running the model on the device locally has no cost for the application developer, as opposed to keeping the cloud server running, and no added end-to-end latency. However, it only supports simple models due to the limited capacity and capabilities of the end device itself.

IDN is a solution to these issues. IDN provides a peer-to-peer (p2p) network of computing devices with spare computing resources that are near to where the inference

needs to be executed. IDN allows inference computation to be distributed across nearby devices, edge nodes, and cloud servers, improving responsiveness of the application and lowering the cost of computing resources while still supporting complex models and providing fault tolerance. IDN makes DNN inference for everyday users fast, scalable, and fault tolerant while not being controlled by a few cloud providers such as Amazon, Microsoft or Google.

1.2 CHALLENGES

One challenge in distributing DNN models across multiple nodes is to consider whether the model is communication bound or computation bound. How to balance this tradeoff while maintaining the model's integrity and the accuracy of the resulting computation is the challenge at hand. Distributing DNN models across devices, edge nodes, and cloud servers allows applications to have access to more complex and more accurate DNN models. However, distributing DNN models across devices, edge nodes, and cloud servers incurs communication overheads that may reduce the responsiveness of the applications.

There is no correct universal answer for this tradeoff, as different applications might prefer different points along the accuracy/speed spectrum; a single application might even prefer different tradeoffs for different DNN models that it uses or at different points in time. The choice of tradeoff is generally dictated by the performance that

the application seeks to deliver to its users. There is no existing system that allows applications to control this tradeoff directly when they deploy their DNN models. In real-world networks, a challenge is to deal with unreliable connections and the fact that nodes can connect and disconnect anytime. This leads to another important question: how should the system deal with this unreliability while maintaining the level of usability and responsiveness desired by the applications using the intelligence distribution network?

Beyond the computational aspects of the system, another important factor for a successful inference network is the availability of high-quality DNN models to be used by the community of applications and developers. A challenge is how to encourage peers to collaborate and share the models, increasing the number of publicly available high-quality DNN models.

Another challenge is how to incentivize peers to participate in the network and share their spare computing resources efficiently among themselves. This leads to a central question: how should this p2p network of peers load balance requests from applications on end devices efficiently in a decentralized p2p manner?

This dissertation aims to tackle these challenges and specifically answer the following questions:

- Can we improve the inference time of DNN models by early exiting the compu-

tation to reduce the overall inference time of IDN? Can we tradeoff accuracy for speed?

- Can we scale DNN models by distributing DNN computation across end devices, edge nodes, and cloud servers efficiently and further decrease the inference time of computations managed by IDN?
- Can we design DNN models that run independently in parallel on a lot of devices of different capabilities in order to improve fault tolerance and availability of DNN inference?
- Can we encourage people to together improve and share high-quality DNN models, increasing the number of publicly available high-quality DNN models?
- Can we incentivize peers to participate in the network and efficiently distribute inference computation in a decentralized peer-to-peer (p2p) manner?

1.3 Intelligence Distribution Network (IDN)

This dissertation tackles the aforementioned challenges with three main components. The first component includes frameworks to adapt DNN models to take advantage of IDN infrastructure with the end goals of improving speed, scalability, and fault tolerance. The second component is a platform for peer collaboration in improving the accuracy and sharing of DNN models, increasing the number of publicly available high-quality DNN models. The third component is a platform to efficiently distribute

inference computation of DNN models to nearby peer-to-peer (p2p) networked devices.

The frameworks to adapt DNN models to take advantage of IDN infrastructure for speed, scalability, and fault tolerance introduced in this dissertation include BranchyNet, Distributed Deep Neural Network (DDNN), and ParallelNet. BranchyNet allows DNN models to exit computation at earlier points of the network. DDNN allows DNN models to scale beyond a single device by mapping DNN computation across a computing hierarchy of end devices, edges, and clouds. ParallelNet generates different-sized DNN models to run independently in parallel on heterogeneous devices of varying capability.

In order to increase the number of publicly available high-quality DNN models to be run on the network, IDN introduces DaiMoN, a decentralized artificial intelligence model network for peer collaboration in improving the accuracy and sharing of DNN models. DaiMoN improves over existing collaborative model improvement systems by allowing peers to verify the accuracy improvement of submitted models without knowing the test labels in a decentralized manner. This is an essential component in order to mitigate intentional model overfitting by model-improving peers. DaiMoN rewards these contributing peers with cryptographic tokens to encourage participation.

To incentivize participation in the network and allow an efficient distribution of infer-

ence computation, ComputeSwap is introduced. ComputeSwap maintains a ledger between any pair of peers to keep track of computation debt and credit. Peers serve inference computation optimistically in the hopes that the debt will be repaid. As debt continues to accumulate, debt-to-credit ratio (debt ratio, for short) is used to probabilistically service inference computation. The higher the debt ratio, the lower the probability of service with the probability reducing as the debt ratio increases. This incentivizes a node to service models needed by its peers in order to not accumulate too much debt or give too little credit, allowing for efficient distribution of DNN models and inference.

The benefits of IDN scale with the number of participants: the more nodes participating in the network, the better the network can efficiently distribute inference computation. This is because the more nodes the network has, the more the inference can be fault tolerant, executed closer to end devices, and scaled beyond the resources inherent in the device itself.

In summary, IDN is a decentralized peer-to-peer (p2p) network that delivers lower-cost, fast, scalable and fault tolerant execution of DNN models. IDN moves the execution closer to end devices by leveraging the aggregate compute power of nodes in the network to dissipate and absorb DNN inference computation for applications. IDN replicates and distributes DNN execution in proportion to the model's popularity, regardless of the application developer's resources. This is, in effect, democratizing

and allowing for large-scale DNN inference. Everybody and every device has access to publicly available high-quality DNN models and inference compute resources to be used in their applications without being limited by concerns of cost or end-user device capabilities.

1.4 CONTRIBUTIONS

This dissertation makes the following contributions:

- An early exiting method that allows neural network computation to do prediction without going through all the layers of DNN models, trading off accuracy for speed.
- A mapping of DNN computation onto a distributed computation infrastructure and a joint optimization method of DNN models on such infrastructure, allowing DNN inference computation to scale both horizontally across devices and vertically across the computing hierarchy.
- A parallel DNN architecture capable of generating multiple different-sized DNN models, each fitting a device of different capability, that run independently in parallel, increasing fault tolerance of DNN inference.
- DaiMoN, a decentralized artificial intelligence model network, which incentivizes peers to collaborate and share high-quality DNN models for a given task. DaiMoN allows peers to verify the accuracy improvement of submitted

models without knowing the test labels, and rewards peers according to their contributions in a decentralized manner.

- IDN, an intelligence distribution network that allows nodes with spare computing resources to participate in inference execution, brings computation closer to its origin, and provides low-cost provisioning and scaling of DNN inference services. The concept of ComputeSwap is the heart of IDN that enables the efficient distribution of DNN models and inference computation.

1.5 NOTATION

In this dissertation, unless noted otherwise, we use the following conventions for variables. Non-bold letters (e.g., a, A) denote scalars. Bold non-capital letters (e.g., $\boldsymbol{a}$, $\boldsymbol{b}$) denote vectors. Bold capital letters (e.g., $\boldsymbol{A}$, $\boldsymbol{B}$) denote matrices. Calligraphic letters (e.g., $\mathcal{A}, \mathcal{B}$) denote sets. A set $\{x_1, \ldots, x_N\}$ may be shorten as $\{x_n\}_{n=1}^{N}$. $\mathbb{R}$ denotes the set of real numbers. $\mathbb{Z}$ denotes the set of integers. $\mathbb{Z}_{>0}$ denotes the set of integers greater than 0. θ denotes parameters of a model. e is the exponential constant.

1.6 OUTLINE

The dissertation is organized as follows. The content in this dissertation is introduced linearly and as such, it can be read one chapter after another.

Chapter 2 gives background information necessary for the content discussed in this dissertation. It reviews deep neural networks. This chapter can be skipped if readers are already familiar with these topics.

Chapter 3 introduces BranchyNet. BranchyNet modifies the architecture of neural networks by adding branches to it. Branches allow inference of an input sample to exit early without having to go through the entire network. BranchyNet is based on the observation that some input samples are considered easy and do not require the computation of the entire network because lower level features are sufficient for making a prediction. Criteria on deciding whether a sample should exit early or continue further through the network are described.

Chapter 4 introduces Distributed Deep Neural Network (DDNN). DDNN introduces the concept of mapping a neural network architecture onto a distributed computing hierarchy of devices, edges, and clouds, allowing DNNs to scale beyond a single device. By considering the entire computing hierarchy as an end-to-end system and jointly optimizing the entire system, we can design the entire system to reduce communication cost, minimize resource usage, and tolerate faulty devices. In combination with the early-exit concept from Chapter 3, DDNN allows fast and localized inference using shallow portions of the neural network on edges and end devices. Mapping DNNs to a distributed computing infrastructure allows DNN models to scale both horizontally across devices and vertically to edges and clouds.

Chapter 5 introduces ParallelNet. ParallelNet generates multiple different-sized DNN models to be executed independently in parallel on multiple heterogeneous devices of different capabilities. The output of each model is then combined to generate the final prediction. When enough confidence is accumulated, the computation can be terminated early without waiting for the results from all of the models. This reduces inference time, and increases fault tolerance of DNN inference.

Chapter 6 introduces DaiMoN, a decentralized artificial intelligence model network, which incentivizes peer collaboration in improving the accuracy of DNN models. A main feature of DaiMoN is that it allows peers to verify the accuracy improvement of submitted models in a decentralized manner without knowing the test labels. DaiMoN reduces model silos and increases the number of publicly available high-quality DNN models, a necessary condition for a successful inference network.

Chapter 7 describes the design and specification of Intelligence Distribution Network (IDN). ComputeSwap is introduced to incentivize peers to participate in the network via an optimistic debt-repayment structure that probabilistically results in repayment based on credits earned for executing inference and debts acquired by using the network. This allows for efficient distribution of inference computation among peers.

Chapter 8 concludes the dissertation with open problems and future researches.

2

Background

This chapter briefly reviews artificial neural networks, with a focus on convolutional neural networks and deep neural networks. We will also review the algorithms used to train artificial neural networks and different types of neural network layers.

2.1 Artificial Neural Networks

The Artificial Neural Network (ANN) was invented in 1958 by psychologist Frank Rosenblatt [6]. It was designed to mimic the neurons of human brain. Figure 2.1 shows a neuron. A neuron receives signal from its dendrites of different lengths. All input signals are combined at the nucleus and output through the axon with various-length terminals. Two neurons are connected via a specialized structure used to transmit signals called a synapse (Figure 2.2).

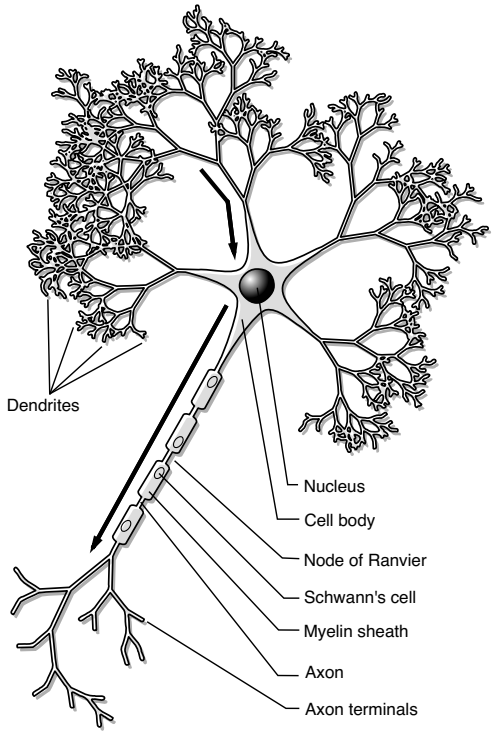

Figure 2.1: A neuron [7]. A neuron receives signal from its dendrites of different lengths. All input signals are combined at the nucleus and output through the axon to other neurons.

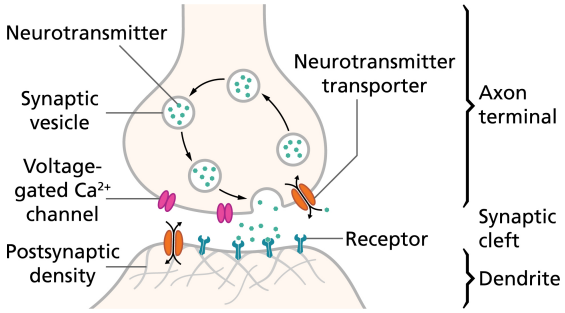

Figure 2.2: Synapse [8]. A specialized structure used to transmit signals between two neurons.

Figure 2.3 shows an artificial neuron that was modeled after a neuron. An artificial neuron consists of inputs, weights, a summation, an activation and an output. Inputs are first multiplied by weights and summed together. Then, the result is passed through an activation function to give an output. Mathematically, it can be represented as follows.

$$y = \sigma(\sum_{i}^{N} w_i x_i),$$

where y is an output, x_i is an input, w_i is a parameter or a weight to be learned or trained, and σ is an activation function. Many neurons work together to form a layer, and many layers are connected one after another to form an artificial neural network (ANN), as depicted in Figure 2.4. These layers are sometimes referred to by their positions in the signal pathway. The first layer usually represents inputs and is called the input layer; each circle is called an input unit. The last layer usually represents outputs and is called output layer; each circle is called an output unit. The layers in between the input layer and output layer are usually called hidden layers; each circle

is called a hidden unit. Commonly, these layers are also referred to as linear layers, since a linear transformation is applied to the incoming signals.

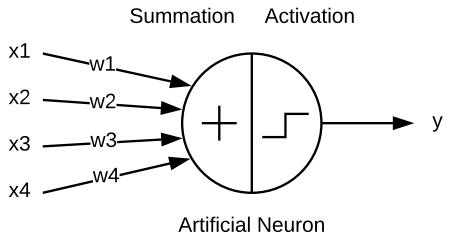

Figure 2.3: An artificial neuron modeled after a neuron. An artificial neuron consists of inputs, weights, a summation, an activation and an output.

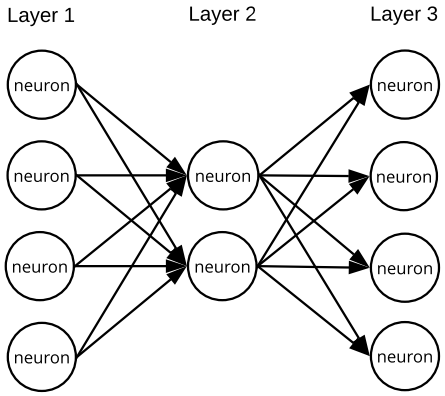

Figure 2.4: An Artificial Neural Network (ANN). An ANN consists of many layers, each with one or more neurons.

Artificial neural networks have many types. The simplest type is called the feedforward neural network, and is as described above. In this type of network, the signal moves from the input layer to the output layer via hidden layers without any cycles

or loops. Other types of artificial neural networks include recurrent neural networks (RNNs) and memory neural networks (MNNs).

2.2 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of feedforward neural networks. It is inspired by the visual cortex part of the brain that processes visual information. It rose in popularity in 2009 with the introduction of AlexNet [9], which won the ImageNet competition in 2012 [10].

We describe here CNN in the context of processing a 2D (2 dimensional) image, but a similar concept applies for 1D (1 dimensional) and higher dimensional signals as well. A CNN makes use of the fact that patterns in natural images are inherently local. The same pattern at one location of an image can be reused at another location in the same image or in another image. A CNN consists mostly of convolutional layers. A convolutional layer takes a multi-channel image as input and convolves it with filters. Mathematically, it can be represented as follows:

$$\mathbf{y}_j = \sigma(\sum_{i=1}^{N} \mathbf{f}_{ji} * \mathbf{x}_i),$$

where N is the number of input channels, $\mathbf{f}_{ji}$ is the i-th channel of the j-th filter, $\mathbf{x}_i$ is the i-th input channel of an input image, $\mathbf{y}_j$ is the j-th output channel of an output

image and σ is an activation function. When the input data is 2D, $\mathbf{f}_{ji}$ is a 2D kernel and $\mathbf{f}_{ji} * \mathbf{x}_i$ is a 2D convolution.

Figure 2.5 depicts a convolutional layer with two input channels and four filters; each circle represents a filter and produces an output channel.

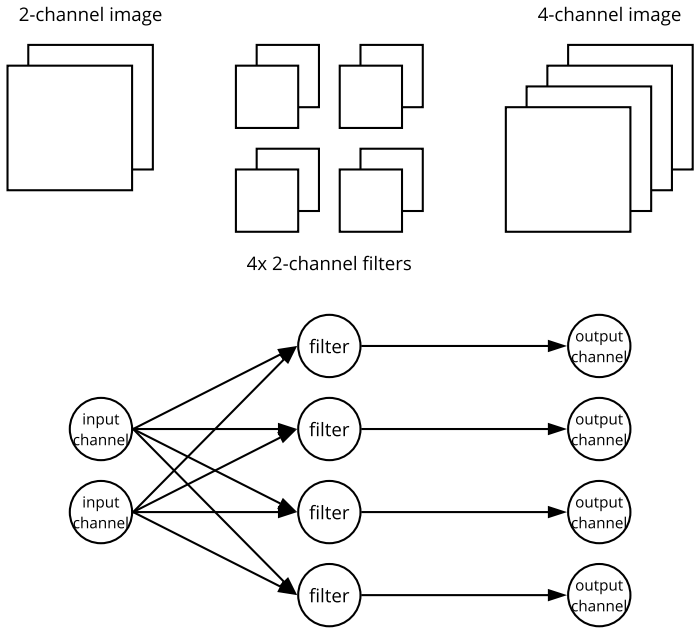

Figure 2.5: A convolutional layer with two input channels and four filters. Each circle represents a filter and produces an output channel.

2.3 Deep Neural Networks

Since the success of CNNs in the ImageNet challenges [9, 10], research in many areas of computer visions and natural language processing has shifted toward CNNs with many layers and obtained better results [11, 12, 13]. These deeper and deeper CNN architectures give rise to deep neural networks (DNNs). Figure 2.6 shows the progression of CNN architectures that get deeper and deeper (from 5-layer LeNet to 152-layer ResNet). New deeper and deeper architectures are being developed and trained everyday.

Deep neural networks (DNNs) are state of the art methods for various tasks due to their ability to piece together lower-level features and extract increasingly abstract higher-level features at each network layer, similar to how a human processes information.

2.4 Training Neural Networks

To train a neural network is to update its parameters or weights toward an objective function. An objective function is different for different tasks. For example, for a classification task, a softmax cross-entropy objective function may be used and for a regressing task, a mean squared error objective function may be used (See Section 2.4.1). A way to train a neural network is to use backpropagation to efficiently compute gra-

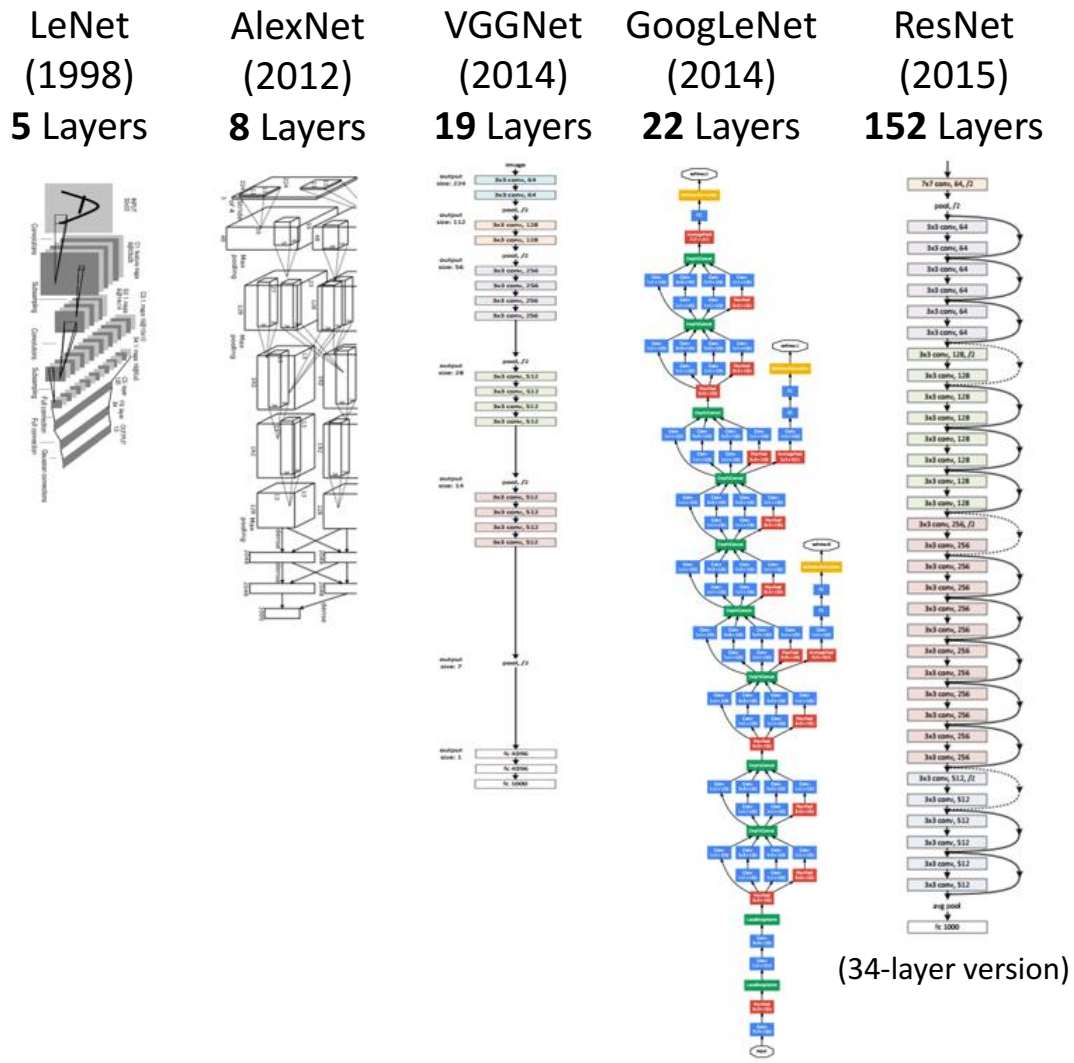

Figure 2.6: Progression towards deeper neural network structures in recent years from LeNet with 5 layers in 1998 to ResNet with 152 layers in 2015 (see, *e.g.*, [14, 10, 15, 16, 17]).

dients and use one of the optimizers (e.g. SGD, Adam [18] and SGDR [19]) to update the parameters (weights) of the neural networks. This section briefly describes each of these methods.

In practice, it can be costly to compute the gradients of the entire set of training sam-

ples and often, the training samples are split into many subsets called batches and the parameters (weights) of a neural network is updated in a batch of training samples. For each iteration of weight update (training), a subset of training samples are used. An epoch of weight update (training) is when all the subsets of the training samples are used once in that epoch. A typical batch size is 128 or 256, though it depends on the available memory, network architectures, and applications. Computing the gradients in batches reduces the variance and can lead to more stable convergence.

2.4.1 OBJECTIVE FUNCTIONS

In this section, we describe two simple objective functions, also called loss functions or cost functions. Throughout the text, these names are used interchangeably. For a classification task, we describe softmax cross-entropy objective function and for a regression task, we describe mean squared error objective function. In this dissertation, we will extensively use the softmax cross-entropy objective function for a classification task.

For a classification task, the softmax cross-entropy objective function may be used.

The softmax cross-entropy objective function ($\mathcal{L}_{\text{CE}}(\boldsymbol{x}, \boldsymbol{y}; \theta)$) is defined as

$$\begin{aligned}
\mathcal{L}_{\text{CE}}(\boldsymbol{x}, \boldsymbol{y}; \theta) &= -\frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} y_c \log \hat{y}_c, \\
\hat{\boldsymbol{y}} &= \text{softmax}(\boldsymbol{z}) \\
&= \frac{\exp(\boldsymbol{z})}{\sum_{c \in \mathcal{C}} \exp(z_c)}, \\
\boldsymbol{z} &= f(\boldsymbol{x}; \theta),
\end{aligned}$$

where ($\boldsymbol{x}$, $\boldsymbol{y}$) is a training input and one-hot ground-truth output label tuple, $\hat{\boldsymbol{y}}$ is an output prediction vector, $\mathcal{C}$ is the set of all possible labels, f is the neural network forward function, and θ represents the parameters of the layers of the neural network.

For a regression task, the mean squared error objective function may be used. The mean squared error objective function ($\mathcal{L}_{\text{MSE}}(\boldsymbol{x}, \boldsymbol{y}; \theta)$) is defined as

$$\begin{aligned}
\mathcal{L}_{\text{MSE}}(\boldsymbol{x}, \boldsymbol{y}; \theta) &= \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2, \\
\hat{\boldsymbol{y}} &= f(\boldsymbol{x}; \theta),
\end{aligned}$$

where ($\boldsymbol{x}$, $\boldsymbol{y}$) is a training input and output pair, $\hat{\boldsymbol{y}}$ is an output prediction vector, N is the number of components of an output vector, f is the neural network forward function, and θ represents the parameters of the layers of the neural network.

The objective function is often calculated in batches. The objective function of a batch is an average of objective function values of the samples in a batch:

$$\frac{1}{M}\sum_{j=1}^{M}\mathcal{L}(\boldsymbol{x}^{(j)},\boldsymbol{y}^{(j)};\theta),$$

where M is the number of samples in a batch (batch size), and $\mathcal{L}(\boldsymbol{x}^{(j)},\boldsymbol{y}^{(j)};\theta)$ is the objective function for the j-th sample.

2.4.2 BACKPROPAGATION

Backpropagation is an efficient method for computing the gradients of a neural network. It consists of forward and backward passes.

In the forward pass, an input is passed through a neural network to get an output. The output is then compared with the target output according to the objective function to compute an error. In the backward pass, the error is backpropagated layer-by-layer and the gradient of the objective function with respect to each parameter of a layer is calculated using the output of that layer and the error backpropagated from the layer above.

After the gradients are calculated, the parameters are updated using the computed gradients. Optimizers are used to keep track of these parameter updates and their convergence. There are many optimization methods. In the following sections, we

review a few methods.

2.4.3 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an iterative method for minimizing the objective function $\mathcal{L}(\theta; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)})$ by updating the parameters θ after seeing only a single or a few training examples. The update is given by

$$\theta_t = \theta_{t-1} - \alpha \nabla_\theta \mathcal{L}(\theta_{t-1}; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}),$$

where $(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)})$ is a training input and output sample, α is the learning rate, and $\nabla_\theta \mathcal{L}(\theta_{t-1})$ is short for $\frac{\partial \mathcal{L}(\theta)}{\partial \theta}\Big|_{\theta=\theta_{t-1}}$.

A variant of SGD is Momentum SGD. Momentum SGD updates the parameters using the exponential moving average of the gradients. This helps dampen oscillations in narrow valleys. The update of Momentum SGD is given by

$$\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} + \alpha \nabla_\theta \mathcal{L}(\theta_{t-1}; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}),$$

$$\theta_t = \theta_{t-1} - \boldsymbol{v}_t,$$

where v_{t-1} and v_t is the previous and current exponential moving average of the gradients, respectively, and have the same dimension as the parameter vector θ. $\beta \in$

$(0, 1]$ is the exponential decay rate.

A slight variant of Momentum SGD is called Nesterovs Accelerated Momentum (NAG) [20, 21]. Instead of calculating the gradient at the current point, NAG calculates the gradient at a point in the vicinity of where the point is going to be when it is updated. The update of NAG is given by

$$\begin{aligned} \boldsymbol{v}_t &= \beta \boldsymbol{v}_{t-1} + \alpha \nabla_\theta \mathcal{L}(\theta_{t-1} - \beta \boldsymbol{v}_{t-1}; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}), \\ \theta_t &= \theta_{t-1} - \boldsymbol{v}_t, \end{aligned}$$

where $\nabla_\theta \mathcal{L}(\theta_{t-1} - \beta \boldsymbol{v}_{t-1})$ is short for $\frac{\partial \mathcal{L}(\theta)}{\partial \theta}\Big|_{\theta=\theta_{t-1}-\beta \boldsymbol{v}_{t-1}}$.

2.4.4 Adam

Another optimization method that is often used in practice today is Adam [18]. The Adam update is given by

$$\begin{aligned} \boldsymbol{v}_t &= \beta_1 \boldsymbol{v}_{t-1} + (1 - \beta_1) \nabla_\theta \mathcal{L}(\theta_{t-1}; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}), \\ \boldsymbol{u}_t &= \beta_2 \boldsymbol{u}_{t-1} + (1 - \beta_2) \Big(\nabla_\theta \mathcal{L}(\theta_{t-1}; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) \Big)^2, \\ \theta_t &= \theta_{t-1} - \alpha \frac{\boldsymbol{v}_t}{\sqrt{\boldsymbol{u}_t} + \epsilon}, \end{aligned}$$

where α is the learning rate, β_1 and β_2 are the exponential decay rates for the moving averages of the gradients and squared gradients accordingly, and ϵ is the smoothing term to avoid division by zero. Notice the term $\sqrt{\boldsymbol{u}_t} + \epsilon$ that is used to normalize the parameter update. The parameters that receive large gradients will have their effective learning rate reduced, while parameters that receive small or infrequent updates will have their effective learning rate increased. Effectively, each parameter adapts its own learning rate.

2.4.5 Stochastic Gradient Descent with Warm Restarts (SGDR)

Stochastic Gradient Descent with Warm Restarts (SGDR)[19] is another variant of SGD that integrates learning rate annealing in its method. With SGDR, the learning rate changes at each iteration or batch. The learning rate α is adjusted as follows:

$$\alpha_t = \alpha_{\text{min}} + \frac{1}{2}(\alpha_{\text{max}} - \alpha_{\text{min}})(1 + \cos(\frac{\pi t}{t_{\text{half}}})),$$

where t is the iteration number, t_{half} is the half period in number of iterations, α_t is the learning rate at iteration t, α_{min} is the minimum learning rate, and α_{max} is the maximum learning rate.

2.4.6 Weight Decay

In training neural networks, weight decay is sometimes used to prevent the weights from growing too large. This technique is commonly used to avoid over-fitting. Weight decay is formulated by adding a L2 regularization term to the objective function as follows.

$$\mathcal{L}(\theta; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) + \frac{\lambda}{2}\|\theta\|_2^2,$$

where λ is the weight decay rate. With the regularization term added, the SGD update is given by

$$\theta_t = \theta_{t-1} - \alpha \nabla_\theta \mathcal{L}(\theta_{t-1}; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) - \alpha\lambda\theta_{t-1}.$$

The additional L2 regularization term adds a term to the SGD update that causes the weight to decay in proportion to its size.

2.4.7 Least Absolute Shrinkage and Selection Operator (LASSO)

Another regularization term that is used in training neural networks is a L1 regularization or LASSO regression. The objective function with L1 regularization is defined as follows.

$$\mathcal{L}(\theta; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) + \frac{\lambda}{2}\|\theta\|_1,$$

where λ is the shrinkage rate. With the regularization term added, the SGD update is given by

$$\theta_t = \theta_{t-1} - \alpha \nabla_\theta \mathcal{L}(\theta_{t-1}; \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}) - \alpha\lambda.$$

Notice that the additional L2 regularization term adds a term to the SGD update that causes the weight to shrink by a certain amount every iteration. The key difference between L1 and L2 regularization is that L1 regularization shrinks a parameter to 0. L1 regularization is commonly used for parameter selection when there are a lot of parameters.

2.5 Basic Types of Neural Network Layers

There are many types of neural network layers. Here, we discuss a few types of neural network layers: linear, convolutional, pooling, normalization, and dropout layers.

2.5.1 Linear Layers

A linear layer applies a linear transformation to the incoming input data:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b},$$

where $\boldsymbol{W}$ is a matrix of the weight parameters, $\boldsymbol{b}$ is the bias vector, $\boldsymbol{x}$ is the input vector, and $\boldsymbol{y}$ is the output vector.

2.5.2 Convolution Layers

A convolutional layer convolves the input N-channel image with a set of learned filters:

$$\boldsymbol{y}_j = \sum_{i=1}^{N} \boldsymbol{f}_{ji} * \boldsymbol{x}_i + \boldsymbol{b}_j,$$

where $\boldsymbol{f}_j$ is the j-th filter weight parameters, $\boldsymbol{b}$ is the bias vector, $\boldsymbol{x}$ is the input vector, and $\boldsymbol{y}_j$ is the j-th output feature map.

A linear layer or a convolutional layer is usually followed by an activation function. An activation function applies a non-linear function element-wise to its input, producing the output of the same dimensions. A few examples of those non-linear functions are

$$\begin{aligned}
\text{sigmoid}(x) &= \frac{e^x}{e^x + 1}, \\
\text{relu}(x) &= \max(0, x), \\
\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}.
\end{aligned}$$

2.5.3 Pooling Layers

Pooling layers combine together features over regions of input feature maps by applying a function such as min, max or average:

$$y = \min_{1 \leq i \leq N} x_i,$$

$$y = \max_{1 \leq i \leq N} x_i,$$

$$y = \frac{1}{N} \sum_{i=1}^{N} x_i,$$

where N is the number of components in a region or a patch of an input image, y is the pooled output, and x_i is the i-th component of a region or a patch of an input image.

2.5.4 Other Layers

There are other types of neural network layers and more types are being discovered everyday. For example, a local response normalization layer [10] normalizes features across spatially grouped feature maps, a batch normalization layer [22] normalizes its input by subtracting the batch mean and dividing by the batch standard deviation, and a dropout layer [23] randomly ignores neurons or filters during training.

2.6 CONCLUSION

In this chapter, we have reviewed artificial neural networks, different types of neural networks and neural network layers, and how we can train them. In the following chapters, we will use this concept as a basis and work toward adapting these deep neural networks to a distributed setting of peers in IDN for fast, scalable, and fault-tolerant inference.

3

BranchyNet: Fast Inference via Early Exiting

BranchyNet is a method to allow samples to exit the inference computation early when sufficient confidence about the decision is present. BranchyNet follows from

the observation that features learned at an early layer of a network may often be sufficient for the classification of many data points without the need to continue to the later layers of the network. Based on this observation, BranchyNet modifies the standard deep network structure by adding exit branches (also called side branches or simply branches for brevity), at certain locations throughout the network. These early-exit branches allow samples that can be accurately classified in early stages of the network to exit at that stage. For more difficult samples, which are expected less frequently, BranchyNet will use further or all network layers to provide the best likelihood of correct prediction. With this modification, BranchyNet allows for fast inference.

With neural networks getting deeper and deeper, the ability for a neural network to exit samples early at a shallower layer will provide a tremendous help in reducing the inference time. In this chapter, we describe BranchyNet with classification tasks in mind; however, the architecture is general and can also be used for other tasks such as natural language processing, image segmentation, and object detection.

In designing the BranchyNet architecture, there are several points to consider, including:

- Number of branches: how many branches should be added?
- Locations of a branch: where to place the branches?
- Structure of a branch: how many layers in a branch? What types and sizes of

layers to use?

Section 3.6 provides some guidance in designing a BranchyNet that augments an existing deep neural network architecture. But first, we describe the architecture of BranchyNet, the training and inference of BranchyNet, and different exit criteria.

3.1 ARCHITECTURE

A BranchyNet is a neural network with an entry point and one or more exit points. A branch is a subset of the network containing contiguous layers, which do not overlap other branches, followed by an exit point. The original neural network can be viewed as a main branch, and layers branching out from the main branch can be viewed as side branches, early-exit branches, or simply exit branches. Each branch must resolve to an exit and is numbered increasingly based on its distance from the entry point, starting at one. An example BranchyNet applied to AlexNet [10] is illustrated in Figure 3.1.

3.2 TRAINING BRANCHYNET

For a classification task, the softmax cross-entropy loss function is commonly used as the optimization objective. Here, we describe how BranchyNet uses this loss function.

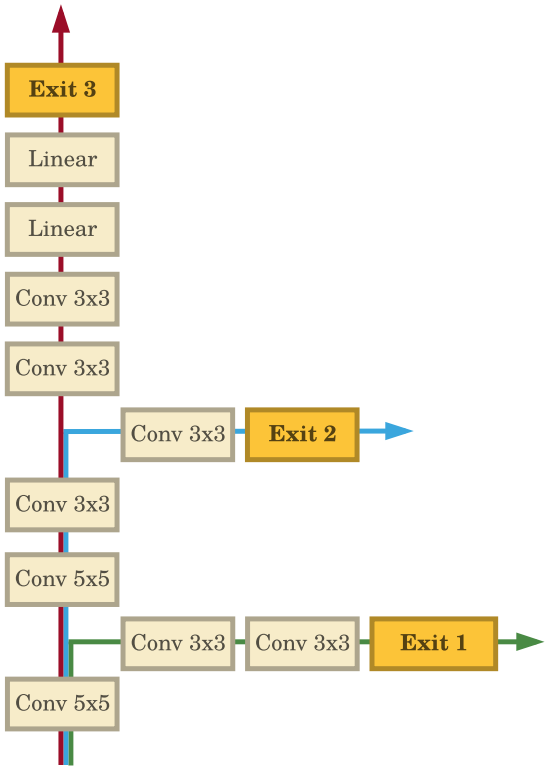

Figure 3.1: A simple BranchyNet with two branches added to the baseline (original) AlexNet. The first branch has two convolutional layers and the second branch has one convolutional layer. The "Exit" boxes denote the various exit points of BranchyNet. This figure shows the general structure of BranchyNet, where each branch consists of one or more layers followed by an exit point.

Let $\boldsymbol{y}$ be a one-hot ground-truth label vector, $\boldsymbol{x}$ be an input sample, and $\mathcal{C}$ be the set of

all possible labels. The objective function can be written as

$$\begin{aligned}
\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}; \theta) &= -\frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} y_c \log \hat{y}_c, \\
\hat{\boldsymbol{y}} &= \text{softmax}(\boldsymbol{z}) \\
&= \frac{\exp(\boldsymbol{z})}{\sum_{c \in \mathcal{C}} \exp(z_c)}, \\
\boldsymbol{z} &= f_{\text{exit}_n}(\boldsymbol{x}; \theta),
\end{aligned}$$

where f_{exit_n} is the n-th exit branch function and θ represents the parameters of the layers of the neural network from an entry point to the exit points.

The design goal of each exit branch is to minimize this loss function. To train the entire BranchyNet, we form a joint optimization problem as a weighted sum of the loss functions of each exit branch

$$\mathcal{L}_{\text{branchynet}}(\{\hat{\boldsymbol{y}}_{\text{exit}_n}\}_{n=1}^{N}, \boldsymbol{y}; \theta) = \sum_{n=1}^{N} w_n \mathcal{L}(\hat{\boldsymbol{y}}_{\text{exit}_n}, \boldsymbol{y}; \theta),$$

where N is the total number of exit points and w_n is the weight of the n-th branch exit. One is free to choose the weight of each branch exit to emphasize one branch more or less than another. Section 3.2.1 presents a simpler way of setting these weights.

The training algorithm for BranchyNet consists of two steps: the feedforward pass and the backward pass. In the feedforward pass, the training dataset is passed through

the network, including both main and side branches, the outputs of all exit points are recorded, and the loss of the network is calculated. In the backward pass, back-propagation is used. The error of the loss is passed back through the network and the weights are updated using a gradient descent algorithm such as Stochastic Gradient Descent (SGD), Adam [18] or Stochastic Gradient Descent with warm Restarts (SGDR) [19].

3.2.1 Exponentially Weighted BranchyNet Loss

To simplify the number of hyper-parameters w_n involved in calculating the loss of a BranchyNet, one can choose to use exponentially weighted BranchyNet loss instead. The loss is defined as follows.

$$\mathcal{L}_{\text{n}}(\{\hat{\boldsymbol{y}}_{\text{exit}_i}\}_{i=1}^{n}, \boldsymbol{y}; \theta) = (1-\alpha)\mathcal{L}_{\text{n-1}}(\{\hat{\boldsymbol{y}}_{\text{exit}_i}\}_{i=1}^{n-1}, \boldsymbol{y}; \theta) + \alpha\mathcal{L}(\hat{\boldsymbol{y}}_{\text{exit}_n}, \boldsymbol{y}; \theta),$$

where $\mathcal{L}_{\text{n}}$ is the weighted sum of the losses of branches up to branch n-th and $\alpha \in [0, 1]$ is the exponential weight parameter. The weighted sum of the losses of up to the n-th branch is the weighted sum of the losses of up to the (n-1)-th branch and the n-th branch loss. The higher the α is, the more the emphasis is put on later branches. The lower the α is, the less the emphasis is put on the earlier branches.

3.3 FAST INFERENCE WITH BRANCHYNET

Once trained, BranchyNet can be used for fast inference by classifying samples at earlier stages in the network based on the algorithm in Figure 3.2. If the classifier at an exit point of a branch has high confidence or enough information (in terms of entropy, as discussed later) about correctly labeling a test sample $\boldsymbol{x}$, the sample is exited and returns a predicted label early with no further computation performed by the later branches of the network. Exit criteria governing a choice of whether a sample should exit at a certain branch are discussed next in Section 3.4.

1: **procedure** BRANCHYNETFASTINFERENCE($\boldsymbol{x}$, $\boldsymbol{T}$)
2: **for** $n \in [1, N]$ **do**
3: $\boldsymbol{z} \leftarrow f_{\text{exit}_n}(\boldsymbol{x})$
4: $\hat{\boldsymbol{y}} \leftarrow \text{softmax}(\boldsymbol{z})$
5: **if** SHOULDEXIT($\hat{\boldsymbol{y}}, T_n$) **then**
6: **return** $\arg\max \hat{\boldsymbol{y}}$
7: **return** $\arg\max \hat{\boldsymbol{y}}$

Figure 3.2: BranchyNet Fast Inference Algorithm. $\boldsymbol{x}$ is an input sample, $\boldsymbol{T} = \{T_n\}_{n=1}^{N}$ is a vector where the n-th entry T_n is the threshold for determining whether to exit a sample at the n-th exit point, N is the number of exit points of the network and SHOULDEXIT($\hat{\boldsymbol{y}}, T_n$) is a function that returns *true* or *false* depending on whether an output probability vector of a branch is confident enough or have enough information (in terms of entropy, as discussed later) to exit. Several choices of SHOULDEXIT($\hat{\boldsymbol{y}}, T_n$) functions are discussed in 3.4.

To perform fast inference on a given BranchyNet network, we follow the procedure as described in Figure 3.2. The procedure requires $\boldsymbol{T} = \{T_n\}_{n=1}^{N}$, a vector where the n-th entry is the threshold used to determine if the input $\boldsymbol{x}$ should exit at the n-th

exit point. The procedure begins with the earliest, shallower exit point and iterates to the latest, deeper exit point of the network. For each exit point, the input $\boldsymbol{x}$ is fed through the corresponding branch. The procedure then calculates the softmax of the output and checks the exit criterion to see if the sample should exit at this branch or if it should carry on to a deeper branch. If the sample should exit, the class label with the highest confidence score is returned. Otherwise, the sample continues to the next exit point. If the sample reaches the last exit point, the label with the highest confidence score is always returned.

3.4 Exit Criteria

In designing the BranchyNet architecture, we need an exit criterion for a branch. Exit criteria helps to determine whether or not the sample should exit. For classification tasks, each branch outputs a probability vector. Here, we present two exit criteria that make use of the output probability vector: Confidence and Entropy.

3.4.1 Confidence

A way to decide whether or not to exit a sample is to use confidence, or the probability value of the winning class in the predicted probability vector. Formally, confi-

dence is defined as follows.

$$\text{confidence}(\hat{\boldsymbol{y}}) = \max \hat{\boldsymbol{y}},$$

where $\hat{\boldsymbol{y}}$ is a vector containing computed probabilities for all possible class labels.

Confidence represents how confident the branch is about the prediction. The higher the value of the confidence, the higher the probability of the branch accurately predicting the class of the sample.

To decide whether to exit a sample at a certain branch, we calculate the confidence value of the prediction of that branch and compare it against a fixed threshold T_n. If the value is greater than or equal to the threshold T_n, the branch has enough confidence to exit the sample and the procedure returns *true*. Otherwise, it returns *false*. The pseudo code of this procedure is provided in Figure 3.3.

```
1: procedure ShouldExitWithConfidence(ŷ_n, T_n)
2:     c ← confidence(ŷ_n)
3:     if c ≥ T_n then
4:         return true
5:     return false
```

Figure 3.3: A procedure to decide whether a sample should exit based on the confidence value of the probability vector. $\hat{\boldsymbol{y}}_n$ is an probability vector output of a branch n, and T_n is the threshold at branch n for determining whether to exit a sample at the n-th exit point.

3.4.2 ENTROPY

Alternatively, another way to decide whether or not to exit a sample is to use entropy. To keep the entropy value between 0 and 1, normalized entropy is used instead. Normalized entropy (η) is defined as follows:

$$\text{entropy}(\hat{\boldsymbol{y}}_n) = \eta(\hat{\boldsymbol{y}}) = -\sum_{c \in \mathcal{C}} \frac{y_c \log y_c}{\log |\mathcal{C}|},$$

where $\hat{\boldsymbol{y}} = \{\hat{y}_c\}_{c \in \mathcal{C}}$ is a vector containing computed probabilities for all possible class labels c, $\mathcal{C}$ is a set of all possible labels, and $|\mathcal{C}|$ is the size of the set of labels $\mathcal{C}$.

Entropy represents how random the information is in the probability vector. The higher the value the entropy is, the less information the probability vector contains.

To decide whether to exit a sample at the n-th branch, we calculate the entropy value of the output probability vector of that branch and compare it against a fixed threshold T_n. If the value is less than or equal to the threshold T_n, the branch has enough information to exit the sample and the procedure returns *true*. Otherwise, it returns *false*. The pseudo code of this procedure is provided in Figure 3.4.

Normalized entropy (η) is used instead of entropy to allow for easier interpretation and searching for its corresponding threshold T_n, as its values range from 0 to 1. For example, η close to 0 means that the branch is confident about the prediction of the

1: **procedure** SHOULDEXITWITHENTROPY($\hat{\boldsymbol{y}}, T_n$)
2: $\quad \eta \leftarrow \text{entropy}(\hat{\boldsymbol{y}}_n)$
3: $\quad$ **if** $\eta \leq T_n$ **then**
4: $\quad\quad$ **return** *true*
5: $\quad$ **return** *false*

Figure 3.4: A procedure to decide whether a sample should exit based on the entropy value of the probability vector. $\hat{\boldsymbol{y}}_n$ is an probability vector output of a branch n, and T_n is the threshold at branch n for determining whether to exit a sample at the n-th exit point.

sample; η close to 1 means it is not confident. At each exit point, η is computed and compared against its corresponding threshold T_n in order to determine if the sample should exit at that point.

3.4.3 CONFIDENCE CALIBRATION WITH SOFTMAX TEMPERATURE

Modern neural networks for classification tend to be overconfident about their predictions. For example, a neural network may output a prediction with probability 0.8; however, it is only accurate 50% of the time, and not 80% of the time as the probability may seem to imply. One way to mitigate this effect is to introduce a temperature parameter to the softmax function and tune the probability outputs with the temperature. This concept of calibrating confidence with softmax temperature is introduced by Guo et al. [24].

Softmax Temperature is a modified version of the softmax function by adding a temperature parameter τ. The higher the temperature, the softer the probability values

and the higher the entropy of the output probability vector. Mathematically, it is defined as follows:

$$\text{softmax}(z, \tau) = \frac{\exp(z/\tau)}{\sum_i \exp(z_i/\tau)}$$

In BranchyNet, one may use this technique to calibrate the probability vector, so that the branch is not over or under confident about its prediction and gives consistent confidence and entropy values. Consistent confidence and entropy values help ease the setting of the desired exit thresholds for both criteria across all branches.

3.5 RESULTS

In this section, we demonstrate the effectiveness of BranchyNet by adapting widely studied convolutional neural networks on the image classification task.

For simplicity, we only describe a network in terms of the convolutional and linear (fully-connected, dense) layers it has. Generally, these networks may also contain max pooling, non-linear activation functions (e.g., a rectified linear unit (ReLU) and sigmoid), normalization (e.g., local response normalization, batch normalization), and dropout.

The dataset used for the experiments is the CIFAR10 [25] dataset. There are 50,000 training images and 10,000 images testing images. The 50,000 training images are split into 45,000 for training and 5,000 for validation. We follow the standard data

augmentation scheme for this dataset, in which the images are centered and normalized, zero-padded with 4 pixels on each side, randomly cropped to produce 32×32 images, and randomly flipped horizontally.

Unless mentioned otherwise, we use SGDR [19] (with the learning rate of 0.01) to train the networks with a batch size of 128. Experiments are done in PyTorch [26] on Amazon EC2 p3.2xlarge instance with a single Nvidia Tesla V100 16GB GPU.

3.5.1 Effects of Branch Weights

In this experiment, we study the effects of branch weights on the branch accuracy and the accuracy-speed trade-off profile of a BranchyNet. We use an AlexNet-style network with 3 branches, as shown in Figure 3.5. The last exit branch is part of the original network (main branch). We train the main branch before the two early-exit branches are added. The main branch has the accuracy of 87.74%. We compare 3 different weighting schemes: 1) equal weighting ($w_1 = 0.33, w_2 = 0.33, w_3 = 0.33$), 2) exponential weighting with $\alpha = 0.9$ ($w_1 = 0.01, w_2 = 0.09, w_3 = 0.90$) and 3) exponential weighting with $\alpha = 0.1$ ($w_1 = 0.81, w_2 = 0.09, w_3 = 0.1$). Each scheme is trained for 50 epochs.

In terms of individual branch accuracy, equal weighting does the best for this network, with each branch having an accuracy of 43.26%, 75.11%, and 87.57%, respec-

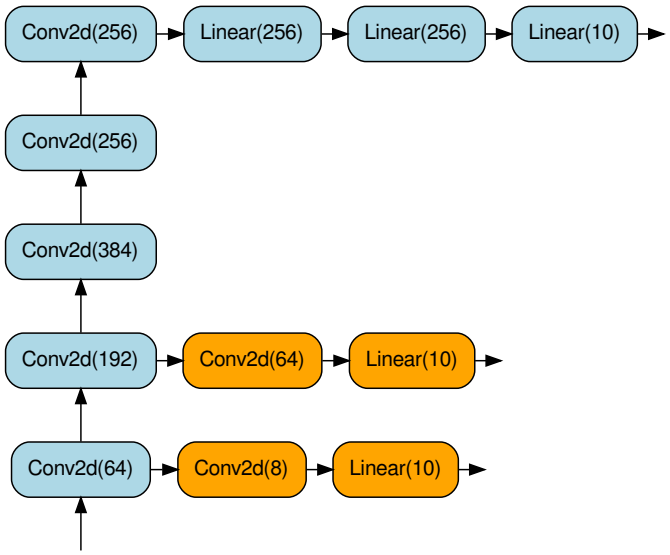

Figure 3.5: AlexNet with 3 branches. The number in parenthesis is the number of filters for Conv2d layer or the number of hidden units for Linear layer.

tively. Exponential weighting with $\alpha = 0.9$ results in each branch having an accuracy of 22.68%, 71.36%, and 87.81%, respectively. Exponential weighting with $\alpha = 0.1$ results in each branch having the accuracy of 32.52%, 65.95%, and 87.45%, respectively. With exponential weighting, one can emphasize on the earlier branch or the later branch. On one hand, the 2nd branch of $\alpha = 0.9$ does better than the 2nd branch of $\alpha = 0.1$. On the other hand, the 1st branch of $\alpha = 0.1$ does better than the 1st branch of $\alpha = 0.9$.

The results comparing the accuracy-speed trade-off profiles of the three weighting schemes are shown in Figure 3.6. Since the 1st branch accuracy of the equal weighting scheme is higher than others, it has the highest accuracy of the three schemes.

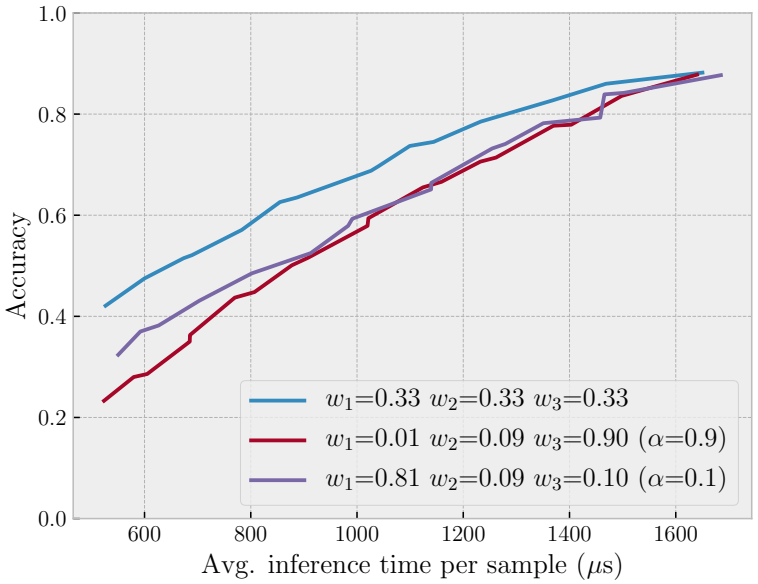

Figure 3.6: Accuracy-speed trade-off profile of BranchyNet trained with different branch weighting schemes.

From experimenting with the weight settings, we observe the following:

- Each branch weight should not be set too low. Otherwise, the gradient will vanish and the network weights are not updated.
- Branches with higher weights tend to converge faster.
- If the branch weights are large enough, varying the branch weights only affects the convergence time and does not affect the accuracy much.
- Outputs of the higher branches rely on the lower layers which are affected by backpropagation from lower branches. Giving higher weights to lower branch usually results in faster convergence, as lower layers are stable first and it is therefore easier for the higher branches to build on top of stable lower layers.

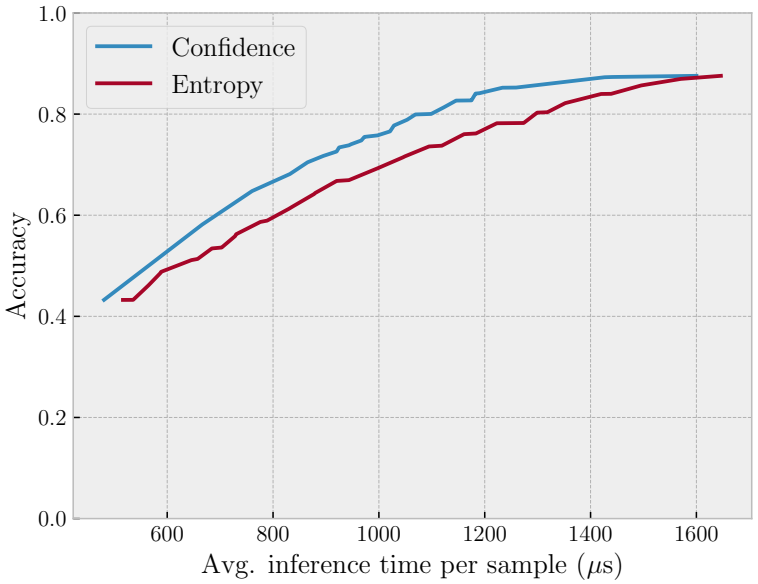

Figure 3.7: Accuracy-speed trade-off profile of the two exit criteria: confidence and entropy.

3.5.2 Comparing Exit Criteria

In this experiment, we compare the two exit criteria: confidence and entropy. We use an AlexNet-style network with 3 branches (a main branch and two early-exit branches), as shown in Figure 3.5.

Figure 3.7 compares the accuracy-speed trade-off profile of the two exit criteria. We see here that the confidence exit criterion performs better than the entropy exit criterion for this network since the confidence exit criterion achieves higher accuracy for less time than the entropy exit criterion achieves.

3.5.3 EARLY EXITS WITH BRANCHYNET

In this experiment, we demonstrate the effectiveness of BranchyNet on a DenseNet-style [27] network with 121 layers and 4 branches. Added branches are shown in Figure 3.8. The first branch consists of a convolutional layer with 32 filters and a linear layer, and is located at layer 14. The second branch consists of a convolutional layer with 64 filters and a linear layer, and is located at layer 39. The third branch consists of a convolutional layer with 256 filters and a linear layer, and is located at layer 88. The last branch is part of the original network.

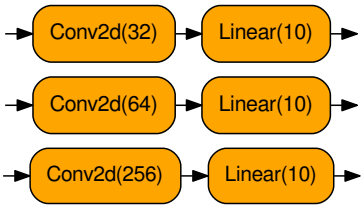

Figure 3.8: 3 early-exit branches added to DenseNet-121. The first early-exit branch (top) is located at layer 14. The second early-exit branch (middle) is located at layer 39. The third early-exit branch (bottom) is located at layer 88.

Figure 3.9 shows the accuracy-speed trade-off profile of this BranchyNet at various threshold settings running on a GPU. We see that BranchyNet achieves higher accuracy at certain settings and faster inference time. At the best accuracy setting, BranchyNet gains 3x speed up and slightly higher accuracy over the original network.

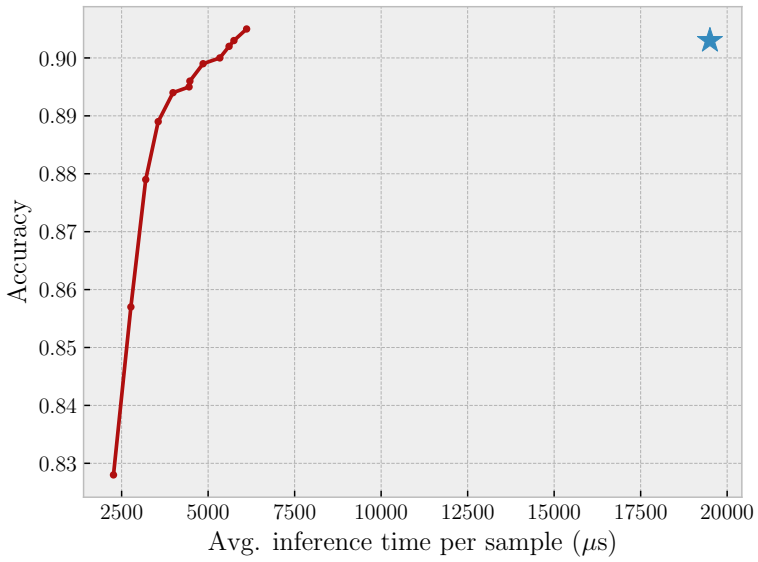

Figure 3.9: GPU performance results for BranchyNet when applied to DenseNet-121 [27]. The original network accuracy and runtime are shown as the star. Each dot denotes different combinations of entropy thresholds for the branch exit points

3.6 Discussion

In this section, we discuss why early exit works and how one starts augmenting a DNN model with early exit branches. We also discuss the regularization benefit of BranchyNet and how it can help with vanishing gradient issue when training deep neural networks.

3.6.1 Why Does Early Exit Work?

For a classification problem, we seek to learn the boundaries between different classes of the samples. The required complexity of the boundaries is tied to the difficulty of

the problem. When we use deep neural networks (DNNs) to solve the classification problem, the complexity of the boundaries able to be learned by the DNNs increases as the number of layers of the DNNs increases.

To understand how early exit works, we split the problem into easy and hard cases. The easy case includes only the samples that are far away from the class boundaries (easy samples) while the hard case includes the rest of the samples that are close to the class boundaries (hard samples).

On one hand, the easy case only needs simple boundaries between classes in order to classify its samples properly. Only a small number of the DNNs' layers is needed to learn these simple boundaries; thus, these easy samples can be exited at the earlier layers of the DNNs. On the other hand, the hard case requires more complex boundaries between classes in order to classify its samples properly. These complex boundaries need the full expressiveness of all the DNNs' layers; thus, these hard samples are exited at the later layers of the DNNs.

3.6.2 Designing BranchyNet

Different branch placements enable different speed and accuracy trade-off profiles. Figure 3.10 compares the trade-off profile of a BranchyNet with two and three branches. Here, we discuss some general guidance when designing a BranchyNet.

In general, the lower the location of the branch, the faster and less accurate the branch becomes. Conversely, the higher the location of the branch, the slower and more accurate the branch becomes.

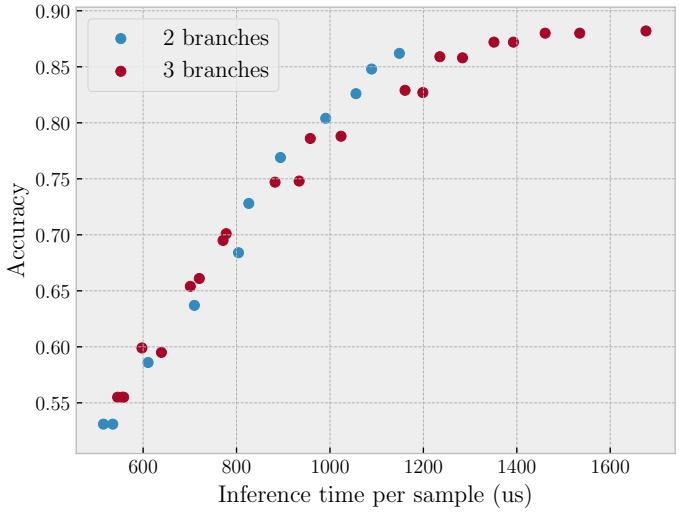

Figure 3.10: Accuracy vs. speed trade-off profile of BranchyNet with two and three branches. With three branches, there are wider ranges of choices of accuracy and inference time.

In designing the number of branches, there should not be too many branches since each branch adds additional processing time to a sample if the branch cannot exit the sample early.

In addition, branches should not be placed too close to each other, since the features that the branches learn will be too similar. Generally, a sample that cannot exit from an earlier branch will not be able to exit from closely placed later branch either. Thus, this closely placed branch will incur additional processing time without early exit

benefits.

The accuracy of the trade-off profile is lower bounded by the first early exit point and the speed of the trade-off profile is upper bounded by the first early exit point. Therefore, designing the first early-exit branch is important since it defines the bound of the trade-off profile.

We can state general guidelines for designing a BranchyNet:

- Add first branch with 1 layer at the lowest layer of the model.
- Add layers to your first branch or move the branch up until the desired accuracy lower bound and speed upper bound of the trade-off profile is reached.
- Experiment with adding additional branches until the desired trade-off profile is reached.

3.6.3 Regularization and Vanishing Gradients

Beyond the speed up benefits by BranchyNet, training the classifiers at these exit branches helps with network regularization and the mitigation of vanishing gradients in backpropagation. For the former, branches will provide regularization on the main branch (baseline network) and other branches, and vice versa. For the latter, a relatively shallower branch at a lower layer will provide more immediate gradient signal in backpropagation, resulting in discriminative features in lower layers of the

main branch, thus improving its convergence.

3.7 Related Works

Due to the computational costs of deep networks, improving the efficiency of feed-forward inference has been heavily studied. Two such approaches are network compression and implementation optimization. Network compression schemes aim to reduce the the total number of model parameters of a deep network and thus reduce the amount of computation required to perform inference. Bucilua et al. (2006) proposed a method of compressing a deep network into a smaller network that achieves a slightly reduced level of accuracy by retraining a smaller network on synthetic data generated from a deep network [28]. More recently, Han et al. (2015) have proposed a pruning approach that removes network connections with small contributions [29]. However, while pruning approaches can significantly reduce the number of model parameters in each layer, converting that reduction into a significant speedup is difficult using standard GPU implementations due to the lack of high degrees of exploitable regularity and computation intensity in the resulting sparse connection structure [30]. Kim et al. (2015) use a Tucker decomposition (a tensor extension of SVD) to extract shared information between convolutional layers and perform rank selection [31]. This approach reduces the number of network parameters, making the network more compact, at the cost of a small amount of accuracy loss. These network

compression methods are orthogonal to the BranchyNet approach, and could potentially be used in conjunction to improve inference efficiency further.

Implementation optimization approaches reduce the runtime of inference by making the computation algorithmically faster. Vanhoucke et al. (2011) explored code optimizations to speed up the execution of convolutional neural networks (CNNs) on CPUs [32]. Mathieu et al. (2013) showed that convolution using FFT can be used to speed up training and inference for CNNs [33]. Recently, Lavin et al. (2015) have introduced faster algorithms specifically for 3x3 convolutional filters (which are used extensively in VGGNet and ResNet) [34]. In contrast, BranchyNet makes modifications to the network structure to improve inference efficiency.

Deeper and larger models are complex and tend to overfit the data. Dropout [23], L1 and L2 regularization and many other techniques have been used to regularize the network and prevent overfitting. Additionally, Szegedy et al. (2015) introduced the concept of adding softmax branches in the middle layers of their inception module within deep networks as a way to regularize the main network [16]. While also providing similar regularization functionalities, BranchyNet has a new goal of allowing early exits for test samples that can already be classified with high confidence.

One main challenge with (very) deep neural networks is the vanishing gradient problem. Several works have introduced ideas to mitigate this issue, including normalized network initialization [35, 36] and intermediate normalization layers [22]. Recently,

new approaches such as Highway Networks [37], ResNet [17], and Deep Networks with Stochastic Depth [38] have been studied. The main idea is to add skip (shortcut) connections in between layers. This skip connection is an identity function that helps propagate the gradients in the backpropagation step of neural network training.

Panda et al. [39] propose Conditional Deep Learning (CDL) by iteratively adding linear classifiers to each convolutional layer, starting with the first layer, and monitoring the output to decide whether a sample can be exited early. BranchyNet allows for more general branch network structures with additional layers at each exit point, while CDL only uses a cascade of linear classifiers, one for each convolutional layer. In addition, CDL does not jointly train the classifier with the original network. We observed that jointly training the branch with the original network significantly improves the performance of the overall architecture when compared to CDL.

3.8 Relevance within IDN

IDN can take advantage of the early-exits of BranchyNet in order to optimize inference efficiency and speed. Such early exit augmentation of a neural network allows for a flexible deployment of deep learning models on a distributed peer-to-peer (p2p) network. A model augmented with early-exit branches can exit early, reducing the amount of computation and thus inference time.

3.9 CONCLUSION

BranchyNet relaxes the rigid structure of the neural network by allowing early exits for highly confident predictions. We have shown in this chapter that BranchyNet helps reduce the inference time while maintaining a similar accuracy level.

The main contributions in this chapter include:

1. Introducing a new neural network architecture concept, called BranchyNet.
2. Joint training of BranchyNet branches.
3. Confidence and entropy exit criteria for classification tasks for fast inference using BranchyNet.

In this chapter, we have discussed BranchyNet which enables faster response by allowing DNN computation to exit early. In the next few chapters, we will use this concept of early termination to accelerate DNN inference in various settings. In the next chapter (Chapter 4), we will discuss a mapping of DNN models onto a distributed computing hierarchy of end devices, other nearby devices, edges, and clouds, allowing DNN models to scale horizontally and vertically across computing hierarchies. BranchyNet can be used in this distributed setting by placing exit branches at physically distinguishable points in the network. For example, we can place Exit 1 on local devices, Exit 2 on network nodes, Exit 3 on edge devices, and the entire processing of the remaining model on the cloud. Samples with higher confidence exit early at the

local nodes, while only lower confident samples are sent to other devices to be processed higher up in the hierarchy. This helps reduces end-to-end inference latency of the deployed neural network overall.

4

Distributed Deep Neural Networks

The current state of deep learning inference systems on end devices leaves an unsatisfactory choice: either (1) offload input sensor data to complex, large deep learning models in the cloud, with the associated server costs, communication costs, latency issues, and privacy concerns, or (2) perform classification directly on the end device using small, simple deep learning models, leading to reduced system accuracy and

increased end-device power usage.

To address these shortcomings, it is natural to consider the use of a distributed computing approach that allows DNNs to scale beyond a single device. Hierarchically distributed computing structures consisting of the cloud, the edge, and end devices (see, *e.g.*, [40, 41]) have inherent advantages, such as supporting coordinated central and local decisions, and providing system scalability, for large-scale intelligent tasks based on geographically distributed Internet of Things (IoT) devices.

An example of one such distributed approach is to combine a small model (less number of parameters) on end devices and a larger model (more number of parameters) in the cloud. The small model at an end device can quickly perform initial feature extraction, and also classification if the model is confident. Otherwise, the end device can fall back to the larger model in the cloud, which performs further processing and final classification. This approach has the benefit of low communication costs compared to always offloading input samples to the cloud, and can achieve higher accuracy compared to a simple model on device only. Additionally, since a summary based on extracted features from the end device model are sent instead of raw sensor data, the system could provide better privacy protection.

This kind of distributed approach spread over a computing hierarchy is challenging for a number of reasons, including:

- End devices such as embedded sensor nodes often have limited memory and battery budgets. This makes it an issue to fit models on the devices that meet the required accuracy and energy constraints.
- A straightforward partitioning of neural network (NN) models over a computing hierarchy may incur prohibitively large communication costs in transferring intermediate results between computation nodes.
- Incorporating geographically distributed end devices is generally beyond the scope of DNN literature. When multiple sensor inputs on different end devices are used, they need to be aggregated together for a single classification objective. A trained NN will need to support such sensor fusion.
- Multiple models at the cloud, the edge and the device need to be learned jointly to allow coordinated decision making. Computation already performed on end device models should be useful for further processing with edge or cloud models.
- Usual layer-by-layer processing of a DNN from the NN's input layer all the way to the NN's output layer does not directly provide a mechanism for local and fast inference at earlier points in the neural networks (e.g., end devices).
- A balance is needed between the accuracy of a model (with the associated model size) at a given distributed computing layer and the cost of communicating to the layer above it. The solution must have reasonably good lower NN layers on the end devices capable of accurate local classification for some input

samples while also providing useful features for classification in the cloud for other input samples.

To address these concerns under the same optimization framework, it is desirable that a system could train a *single* end-to-end deep learning model and partition it between end devices, the edges, and the cloud.

To this end, we propose distributed deep neural networks (DDNNs) over distributed computing hierarchies, consisting of geographically distributed cloud, edge, and end devices. In implementing a DDNN, we design and map sections of a single DNN onto a distributed computing hierarchy, considering communication limitations between devices and computation limitations of each device. By jointly training these sections, we show that DDNNs can effectively address the aforementioned challenges. Specifically, while being able to accommodate inference of a DNN in the cloud, a DDNN allows fast and localized inference using some shallow portions of the DNN at the edge and end devices. Moreover, via distributed computing, DDNNs naturally enhance sensor fusion, data privacy, and system fault tolerance for DNN applications. When supported by a scalable distributed computing hierarchy, a DDNN can scale up in neural network size and scale out in geographical span.

DDNN leverages BranchyNet (see Section 3) which allows early exit points to be placed in a DNN. Samples can be classified and exited locally when the system is confident, and offloaded to the edge and the cloud when additional processing is re-

quired. By training DDNN end-to-end, the network optimally configures lower NN layers to support local inference at end devices, and higher NN layers in the cloud to improve overall classification accuracy of the system. As a proof of concept, we show that a DDNN can exploit geographical diversity of sensors (on a multi-view multi-camera dataset) in sensor fusion to improve recognition accuracy.

The contributions of this chapter include

1. A novel DDNN framework and its implementation that maps sections of a DNN onto a distributed computing hierarchy. System constraints such as device memory and limited communication can be designed into the structure of the DNN in the DDNN framework. This helps simplify the implementation and deployment of an intelligence distributed system.
2. A joint training method that minimizes communication and resource usage for devices and maximizes the usefulness of extracted features that are utilized in the cloud, while allowing low-latency classification via early exit for a high percentage of input samples.
3. Aggregation schemes that allow automatic sensor fusion of multiple sensor inputs to improve the overall performance (accuracy and fault tolerance) of the system.

4.1 ARCHITECTURE

DDNN maps a trained DNN onto heterogeneous physical devices distributed locally, at the edge, and in the cloud. Since DDNN relies on a jointly trained DNN framework at all parts in the neural network, for both training and inference, many of the difficult engineering decisions are greatly simplified. Figure 4.1 provides an overview of the DDNN architecture. The configurations presented show how DDNN can scale the inference computation across different physical devices. The cloud-based DDNN in (a) can be viewed as the standard DNN running in the cloud as described in the introduction. In this case, sensor input captured on end devices is sent to the cloud in original format (raw input format), where all layers of DNN inference is performed.

We can extend this model to include a single end device, as shown in (b), by performing a portion of the DNN inference computation on the device rather than sending the raw input to the cloud. Using an exit point after device inference, we may classify those samples that the local network is confident about, without sending any information to the cloud. For more difficult cases, the intermediate DNN output (up to the local exit) is sent to the cloud, where further inference is performed using additional NN layers and a final classification decision is made. Note that the intermediate output can be designed to be much smaller than the sensor input (*e.g.*, a raw image from a video camera), and therefore may drastically reduce the network communication

required between the end device and the cloud. The details of how communication is considered in the network is discussed in section 4.6.1.

DDNN can also be extended to multiple end devices which may be geographically distributed, shown in (c), that work together to make a classification decision. Here, each end device performs local computation as in (b), but their output is aggregated together before the local exit point. Since the entire DDNN is jointly trained across all end devices and exit points, the network automatically aggregates the input with the objective of achieving maximum classification accuracy. This automatic data fusion (sensor fusion) simplifies runtime inference by avoiding the necessity of manually combining output from multiple end devices. We will discuss the design of feature aggregation in detail in section 4.2. As before, if the local exit point is not confident about the sample, each end device sends intermediate output to the cloud, where another round of feature aggregation is performed before making a final classification decision.

DDNN scales vertically as well, by using an edge layer in the distributed computing hierarchy between the end devices and cloud, shown in (d) and (e). The edge acts similarly to the cloud, by taking output from the end devices, performing aggregation and classification if possible, and forwarding its own intermediate output to the cloud if more processing is needed. In this way, DDNN naturally adjusts the network communication and response time of the system on a per sample basis. Samples that

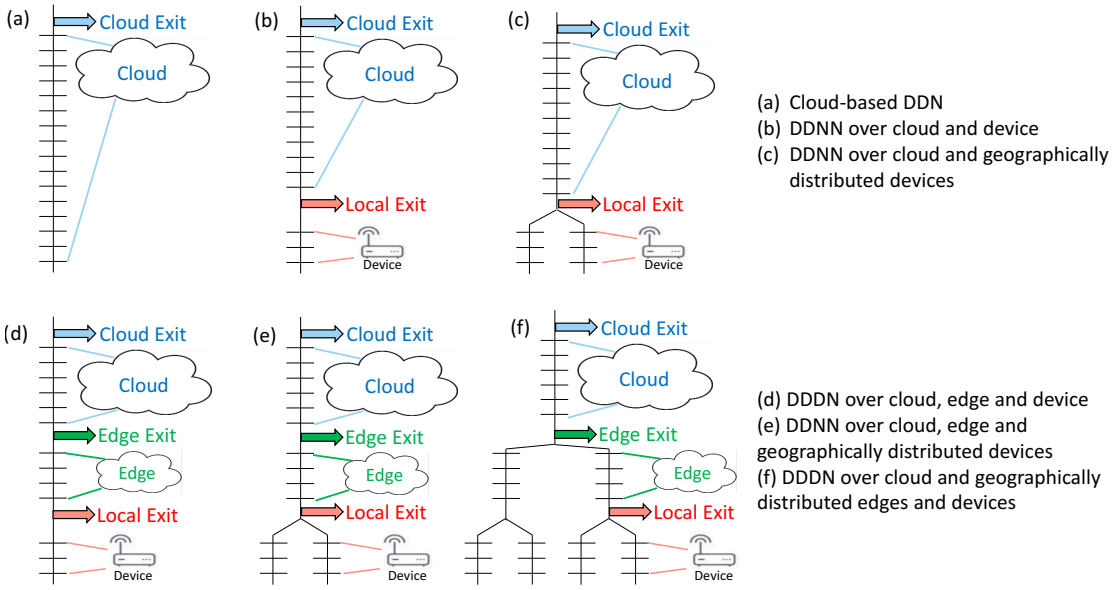

Figure 4.1: Overview of the DDNN architecture. The vertical lines represent the DNN pipeline, which connects the horizontal bars (NN layers). (a) is the standard DNN (processed entirely in the cloud), (b) introduces end devices and a local exit point that may classify samples before the cloud, (c) extends (b) by adding multiple end devices which are aggregated together for classification, (d) and (e) extend (b) and (c) by adding edge layers between the cloud and end devices, and (f) shows how the edge can also be distributed like the end devices.

can be correctly classified locally are exited without any communication to the edge or cloud. Samples that require more feature extraction than can be provided locally are sent to the edge, and eventually to the cloud if necessary. Finally, DDNNs can also scale geographically across the edge layer as well, which is shown in (f).

4.2 Aggregation Methods

In DDNN configurations with multiple end devices (*e.g.*, (c), (e), and (f) in Figure 4.1), the output from each end device must be aggregated in order to perform classifica-

tion. We present several different schemes for aggregating the output as shown in Figure 4.2.

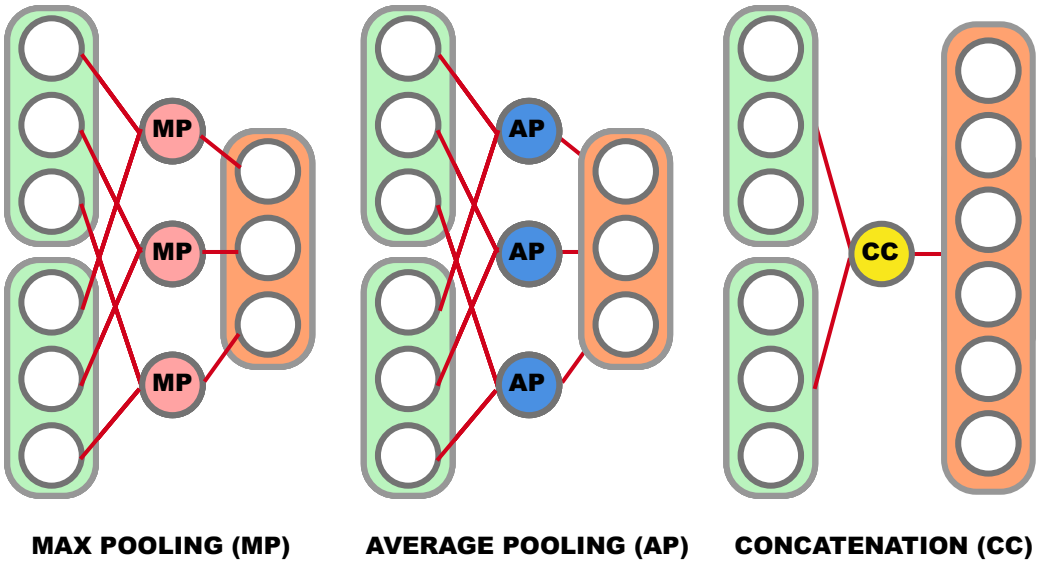

Figure 4.2: Different types of aggregation methods: Max pooling (MP), Average pooling (AP), and Concatenation (CC).

Each aggregation method makes different assumptions about how the device output should be combined, and therefore can result in different system accuracy. We present three approaches:

- Max pooling (MP). MP aggregates the input vectors by taking the max of each component. Mathematically, max pooling can be written as

$$\hat{v}_j = \max_{1 \le i \le n} V_{ji},$$

where n is the number of inputs and $\boldsymbol{V}_{ji}$ is the j-th component of the input vector and $\hat{\boldsymbol{v}}_j$ is the j-th component of the resulting output vector. Max pooling only keeps the strongest signal of each component. This aggregation method is usually good for retaining strongest high-level features.

- Average pooling (AP). AP aggregates the input vectors by taking the average of each component. This is written as

$$\hat{\boldsymbol{v}}_j = \sum_{i=1}^{n} \frac{\boldsymbol{V}_{ji}}{n},$$

where n is the number of inputs and $\boldsymbol{V}_{ji}$ is the j-th component of the input vector and $\hat{\boldsymbol{v}}_j$ is the j-th component of the resulting output vector. Average pooling may help reduce noise in inputs presented in some end devices.

- Concatenation (CC). CC simply concatenates the input vectors together. This aggregation method is usually good for retaining all low-level features. CC retains all information which is useful for higher layers (*e.g.*, the cloud) that can use the full information to extract higher level features. Note that this expands the dimension of the resulting vector. To map this vector back to the same number of dimensions as input vectors, once can add an additional linear layer.

We analyze these aggregation methods in Section 4.5.4.

4.3 TRAINING

While DDNN inference is distributed over the distributed computing hierarchy, the DDNN system can be trained on a single powerful server or in the cloud. One aspect of DDNN that is different from most conventional DNN pipelines is the use of multiple exit points, as shown in Figure 4.1. At training time, the loss from each exit is combined during back-propagation so that the entire network can be jointly trained, and each exit point achieves good accuracy relative to its depth. To train DDNN, we follow joint training as described in Section 3.2 of Chapter 3.

4.4 INFERENCE

Inference in DDNN is performed in several stages using multiple preconfigured exit thresholds $\boldsymbol{T}$ (one element T_n at each exit point n) as a measure of confidence in the prediction of the sample. One way to define $\boldsymbol{T}$ is by searching over the ranges of $\boldsymbol{T}$ on a test set for the $\boldsymbol{T}$ that gives the best accuracy within a desired inference time.

DDNN inference follows the procedure described in Section 3.3 of Chapter 3. In this work, we use normalized entropy η as the exit criterion: at a given exit point n, if the predictor is not confident in the result (*i.e.*, $\eta > T_n$), the system falls back to a higher exit point in the hierarchy until the last exit is reached which always performs classification.

We now provide an example of the inference procedure for a DDNN which has multiple end devices and three exit points (configuration (e) in Figure 4.1):

1. Each end device first sends summary information to a local aggregator.
2. The local aggregator determines if the combined summary information is sufficient for accurate classification.
3. If so, the sample is classified (exited).
4. If not, each device sends more detailed information to the edge in order to perform further processing for classification.
5. If the edge believes it can correctly classify the sample it does so and no information is sent to the cloud.
6. Otherwise, the edge forwards intermediate computation to the cloud, which makes the final classification.

4.5 Results

In this section, we evaluate DDNN on a scenario with multiple end devices and demonstrate the following characteristics of the approach:

- DDNNs allow multiple end devices to work collaboratively in order to improve accuracy at both the local and cloud exit points.
- DDNNs seamlessly extend the capability of end devices by offloading difficult

samples to the cloud.

- DDNNs have built-in fault tolerance. We illustrate that missing any single end device does not dramatically affect the accuracy of the system. Additionally, we show how performance gradually degrades as more end devices are lost.
- DDNNs reduce communication costs for end devices compared to a traditional system that offloads all input sensor data to the cloud.

We first introduce the DDNN architecture and dataset used in our evaluation. Throughout this section, we use various accuracy measures as defined in Section 4.5.3 and communication cost as defined in Section 4.6.1.

4.5.1 DDNN Evaluation Architecture

To accommodate the small memory size of the end devices, we use Binary Neural Network blocks [42]. Here, a block consists of one or more conventional NN layers. We make use of two types of blocks presented in [43]: the fused binary fully connected (FC) block and fused binary convolution-pool (ConvP) block as shown in Figure 4.3. FC blocks each consist of a fully connected layer with m nodes for some m, batch normalization and binary activation. ConvP blocks each consist of a convolutional layer with f filters for some f, a pooling layer and batch normalization, and binary activation. A convolution layer has a kernel of size 3x3 with stride 1 and padding 1. A pooling layer has a kernel of size 3x3 with stride 2 and padding 1.

For our experiments, we use version (c) from Figure 4.1, with six end devices. The system presented can be generalized to a more elaborated structure which includes an edge layer, as shown in (d), (e) or (f) of Figure 4.1. Figure 4.4 depicts a detailed view of the DDNN system used in our experiments. In this system, we have six end devices shown in red, a local aggregator, and a cloud aggregator. During training, output from each device is aggregated together at each exit point using one of the aggregation schemes described in Section 4.2. We provide detailed analysis on the impact of aggregation schemes at both the local and cloud exit points in Section 4.5.4. All DDNNs in our experiments are trained with Adam [18] using the following hyper-parameter settings: α of 0.001, β_1 of 0.9, β_2 of 0.999, and ϵ of 10^{-8}. We train each DDNN for 100 epochs. When training the DDNN, we use equal weights for the local and cloud exit points. We explored heavily weighting both the local exit and the cloud exit, but neither weighting scheme significantly changed the accuracy of the system. This indicates that this solution to the dataset and the problem we are exploring is not sensitive to the weights, but this may not be true for other datasets and problems. For example, in GoogleNet [16], a less than 1% difference in accuracy was observed based on varying the values of the weight parameters of auxiliary branches.

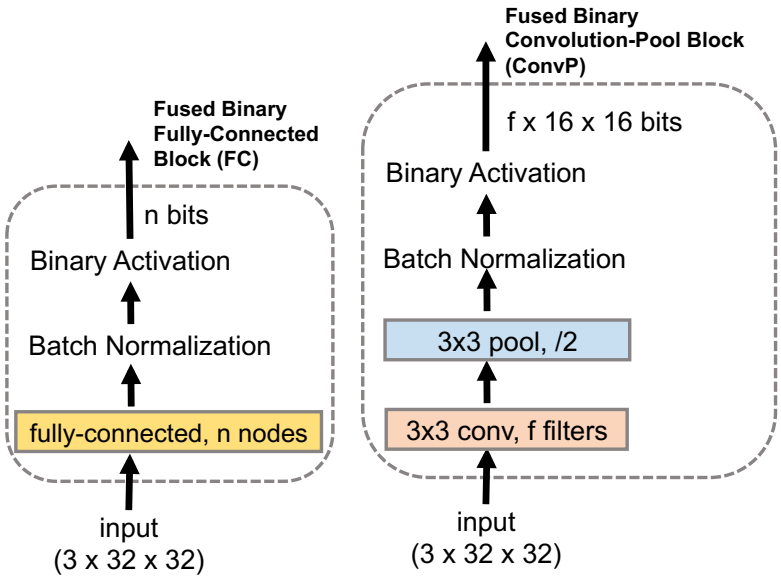

Figure 4.3: Fused binary blocks consisting of one or more standard NN layers. The fused binary fully connected (FC) block is a fully connected layer with n nodes, batch normalization and binary activation. The fused binary convolution-pool (ConvP) block consists of a convolutional layer with f filters, a pooling layer, batch normalization and binary activation. The convolution layer has a kernel of size 3x3 with stride 1 and padding 1. The pooling layer has a kernel of size 3x3 with stride 2 and padding 1. These blocks are used as they are presented in [43].

4.5.2 Multi-view Multi-camera Dataset

We evaluate the proposed DDNN framework on a multi-view multi-camera (MVMC) dataset [44]. This dataset consists of images acquired at the same time from six cameras placed at different locations facing the same general area. For the purpose of our evaluation, we assume that each camera is attached to an end device, which can transmit the captured images over a bandwidth-constraint wireless network to a physical endpoint connected to the cloud.

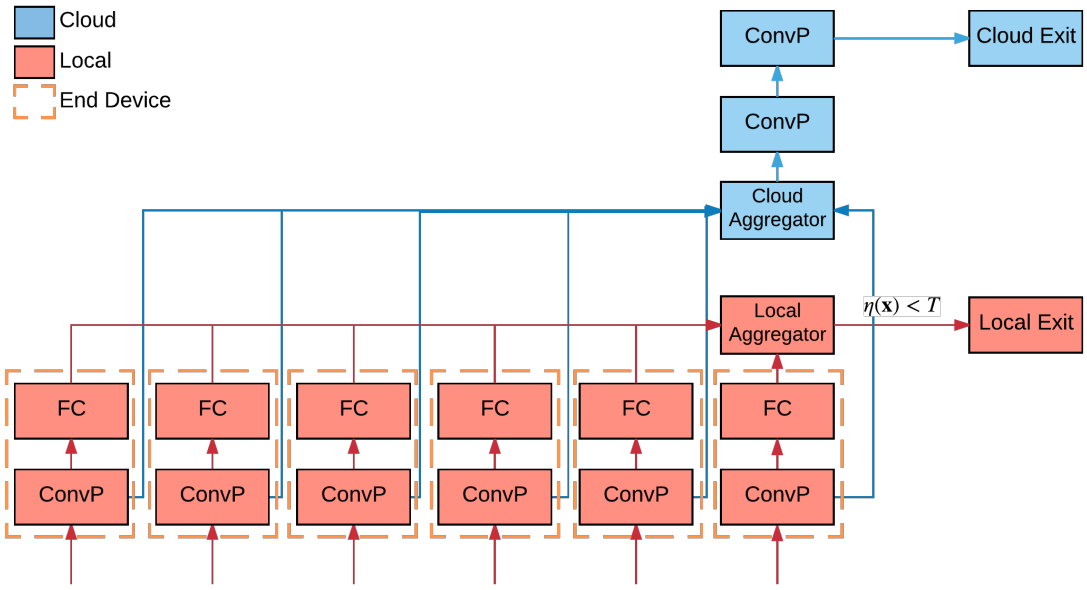

Figure 4.4: The DDNN architecture used in the system evaluation. The FC and ConvP blocks in red and blue correspond to layers run on end devices and the cloud respectively. The dashed orange boxes represent the end devices and show which blocks of the DDNN are mapped onto each device. The local aggregator shown in red combines the exit output (a short vector with length equal to the number of classes) from each end device in order to determine if local classification for the given input sample can be performed accurately. If the local exit is not confident (*i.e.* $\eta(\boldsymbol{x}) > T$), the activation output after the last convolutional layer from each end device is sent to the cloud aggregator (shown in blue), which aggregates the input from each device, performs further NN layer processing, and outputs a final classification result.

The dataset provides object bounding box annotations. Multiple bounding boxes may exist in a single image, each of which corresponds to a different object in the frame. In preparing the dataset, for each bounding box, we extract an image, and manually synchronize* the same object across the multiple devices that the object appears in for the given frame. Examples of the extracted images are shown in Figure 4.5. Each row corresponds to a single sample used for classification. We resize each extracted sample

*In practical object tracking systems, this synchronization step is typically automated [45].

to a 32×32 RGB pixel image. For each device that a given object does not appear in, we use a blank image and assign a label of -1, meaning that the object is not present in the frame. Labels 0, 1, and 2 correspond to car, bus and person, respectively. Objects that are not present in a frame (*i.e.*, label of -1) are not used during training. We split the dataset into 680 training samples and 171 testing samples. Figure 4.6 shows the distribution of samples at each device. Due to the imbalanced number of class samples in the dataset, the individual accuracy of each end device differs widely, as shown by the "Individual" curve of Figure 4.8. A full description of the training process for the individual NN models is provided in Section 4.5.6.

Figure 4.5: Example images of three objects (person, bus, car) from the multi-view multi-camera dataset. The six devices (each with their own camera) capture the same object from different orientations. An all grey image denotes that the object is not present in the frame.

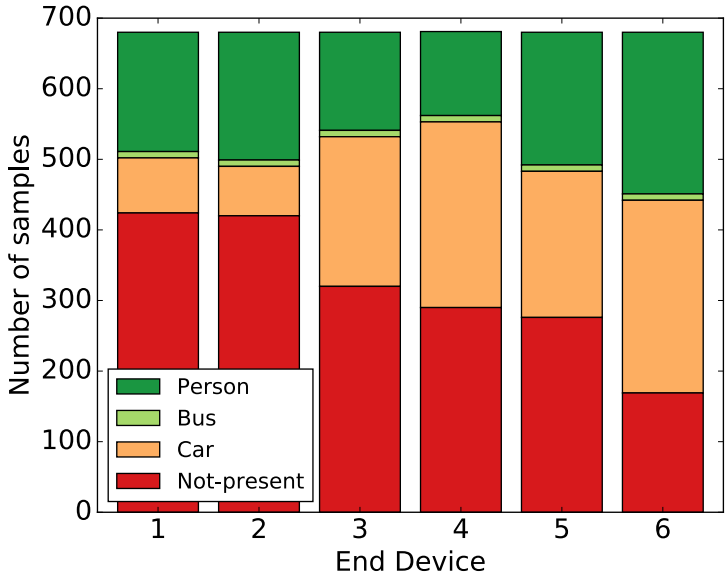

Figure 4.6: The distribution of class samples for each end device in the multi-view multi-camera dataset.

4.5.3 Accuracy Measures

Throughout this section, we use different accuracy measures for the various exit points in a DDNN as follows:

- *Local Accuracy* is the accuracy when exiting 100% of samples at the local exit of a DDNN.
- *Edge Accuracy* is the accuracy when exiting 100% of samples at the edge exit of a DDNN.
- *Cloud Accuracy* is the accuracy when exiting 100% of samples at the cloud exit of a DDNN.

- *Overall Accuracy* is the accuracy when exiting some percentage of samples at each exit point in the hierarchy. The samples classified at each exit point n are determined by the entropy threshold T_n for that exit n. The impact of $\boldsymbol{T}$ on classification accuracy and communication cost is discussed in Section 4.5.5.
- *Individual Accuracy* is the accuracy of an end device NN model trained separately from DDNN. The NN model for each end device consists of a ConvP block followed by a FC block (a single end device portion as shown in Figure 4.4). In the evaluation, individual accuracy for each device is computed by classifying all samples using the individual NN model and not relying on the local or cloud exit points of a DDNN.

4.5.4 Impact of Aggregation Schemes

In order to perform classification on the input from multiple end devices, we must aggregate the information from each end device. We consider three aggregation methods (max pooling, average pooling, and concatenation) outlined in Section 4.2, at both the local and cloud exit points. The accuracy of different aggregation schemes are shown in Table 4.1. The first two letters identify the local aggregation scheme and the last two letters identify the scheme used by the cloud aggregator. For example, MP-CC means the local aggregator uses max-pooling and the cloud uses concatenation. Recall that each input to the local aggregator is a floating-point vector of length equal

Table 4.1: Accuracy of aggregation schemes. The first two letters identify the local aggregation scheme, and the last two letters identify the cloud aggregation scheme. For example, MP-CC means the local aggregator uses max-pooling and the cloud aggregator uses concatenation. The accuracy of each exit point (either local or cloud) is computed using the entire test set. In practice, we will exit a portion of samples locally based on the entropy threshold T and send the remaining samples in the cloud. Due to its high performance, MP-CC is used in the remaining experiments in this chapter.

Schemes	Local Acc. (%)	Cloud Acc. (%)
MP-MP	95	91
MP-CC	**98**	**98**
AP-AP	86	98
AP-CC	75	96
CC-CC	85	94
AP-MP	88	93
MP-AP	89	97
CC-MP	77	87
CC-AP	80	94

to the number of classes (corresponding to the output from the final FC block for a single device as shown in Figure 4.4) and the device output sent to the cloud aggregator is the output from the final ConvP block.

The MP-MP scheme has good classification accuracy for the local aggregator but poor performance in the cloud. On one hand, the elements in the vectors at the local aggregator correspond to the same features (*e.g.*, the first item is the likelihood that the input corresponds to that class); therefore, max pooling corresponds to taking the max response for each class over all end devices, and shows good performance. On the other hand, since the information sent from the end devices to the cloud is the activation output from the filters at each device, which corresponds to different visual features in the input from the viewpoint of each individual end device, max pooling

these features does not perform well.

Comparing the MP-MP and MP-CC schemes, although both use MP for local aggregators, MP-CC increases the accuracy of the local classifier. In the training phrase, during backpropagation the MP-MP scheme only passes gradients through a device that gives the highest response, while the MP-CC scheme passes gradients through all devices. Therefore, using CC aggregator in the cloud allows all devices to learn better filters (filter weights) by giving a stronger response to the local MP aggregator to use during training with backpropagation, resulting in a better classification accuracy.

The CC-CC scheme shows an opposite trend where the local accuracy is poor while the cloud accuracy is high. Concatenating the local information (instead of a pooling scheme), does not enforce any relationship between output for the same class on multiple devices, and therefore performs worse. Concatenating the output for the cloud aggregator maintains the most information for NN layer processing in the cloud and therefore performs well.

Generally, for the local aggregator, average pooling performs worse than max pooling. This is because some of the end devices do not have the object present in the given frame. Average pooling take the average of all outputs from end devices; this degrades the strong outputs from end devices in which the object is present. Based on these results, we use the MP-CC aggregation scheme for the rest of this chapter unless

mentioned otherwise.

4.5.5 Entropy Threshold

The entropy threshold for an exit point corresponds to the level of information that is required in order to exit a sample. Setting the threshold of 0 would mean that no samples will exit, and setting it at 1 would mean that all samples exit at that point. Figure 4.7 shows the relationship between the threshold T at the local aggregator and the overall accuracy of the DDNN. We observe that as more samples are exited at the local exit, the overall accuracy decreases. This is expected, as the accuracy of the local exit is typically lower than that of the cloud exit.

We need to set the threshold appropriately to achieve a balance between the communication cost (as defined in Section 4.6.1), latency, and accuracy of the system. In this case, we see that setting the threshold to 0.8 results in the best overall accuracy with significantly reduced communication, *i.e.*, 97% accuracy while exiting 60.82% of samples locally as shown in Table 4.2 where in addition to local exit (%) and overall accuracy (%), communication cost in bytes is given. We set $T = 0.8$ for the remaining experiments in the system evaluation, unless noted otherwise.

The local classifier may do better than the cloud for certain samples where low-level features are more robust in classification than higher-level features. By setting an ap-

propriate threshold T, we can improve overall accuracy. In this experiment, $T = 0.8$ corresponds to that sweet spot where some samples that are incorrectly classified by the cloud classifier can actually be correctly classified by the local classifier. Such a threshold indicates the optimal point where both local and cloud classifier work best together.

Table 4.2: Effects of different exit threshold (T) settings for the local exit. $T = 0.8$ is used in the remaining experiments.

T	**Local Exit (%)**	**Overall Acc. (%)**	**Comm. (B)**
0.1	0.00	96	140
0.3	0.58	96	139
0.5	1.75	96	138
0.6	2.92	96	136
0.7	22.81	96	111
0.8	**60.82**	**97**	**62**
0.9	83.04	96	34
1.0	100.00	92	12

4.5.6 Impact of Scaling Across End Devices

In order to scale DDNNs across multiple end devices, we distribute the lower sections of Figure 4.4, shown in red, over the corresponding devices, outlined in orange. Figure 4.8 shows how the accuracy of the system improves as additional end devices (each with its attached input cameras) are added. The devices are added in order sorted by their individual accuracy from worst to best (*i.e.*, the device with the lowest accuracy first and the device with the highest accuracy last).

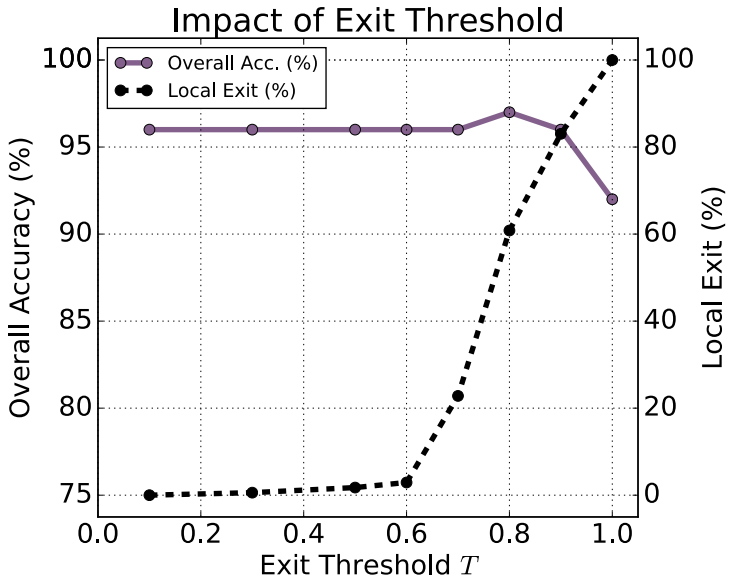

Figure 4.7: Overall accuracy of the system as the entropy threshold for the local exit is varied from 0 to 1. For this experiment, 4 filters are used in the ConvP blocks on the end devices.

The first observation is the large variation in the individual accuracies of the end devices, as noted earlier. Due to the nature of the dataset, some devices are naturally better positioned and generally have clearer observations of the objects. Looking at the viewpoints of each camera in Figure 4.5, we see that the selected examples for Device 6 have clear frontal views of each object. This viewpoint gives Device 6 the highest individual accuracy at over 70%. By comparison, Device 2 has the lowest individual accuracy at under 40%.

The "Local" and "Cloud" curves show the accuracy of the system at each exit point when all samples are exited at that point. We observe that the cloud exit point outperforms the local exit point at all numbers of end devices. The gap is widest when

there are fewer devices. This suggests that the additional NN layers in the cloud significantly improve the final classification result when the problem is more difficult due to limited labeled training data for an end device. Once all six end devices are added, both the local and cloud aggregators have high accuracy. The "Overall" curve represents the overall accuracy of the system when the threshold T for the local exit point is set to 0.8. We see that this curve is roughly equivalent to exiting all samples at the cloud (but at a much reduced communication cost as 60.82% of samples are exited locally). Generally, these results show that by combining multiple viewpoints, we can increase the classification accuracy at both the local and cloud level by a substantial margin when compared to the individual accuracy of any device. The resulting accuracy of the DDNN system is superior to any individual device accuracy by over 20%. Moreover, we note that the 60.82% of samples which exit locally enjoy lowered latency in response time.

4.5.7 Impact of Cloud Offloading on Accuracy Improvements

DDNNs improve the overall accuracy of the system by offloading difficult samples to the cloud, which perform further NN layer processing and final classification. Figure 4.9 shows the accuracy vs. communication costs (as defined in Section 4.6.1) of the DDNN as the number of filters on the end devices increases. For all settings, the NN layers stored on an end device require under 2 KB of memory. In this experiment,

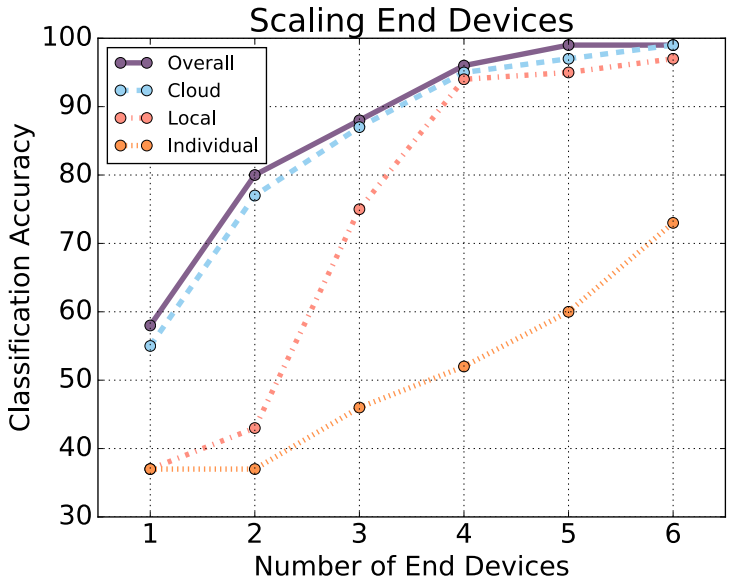

Figure 4.8: Accuracy of the DDNN system as additional end devices are added. The accuracy of "Overall" is obtained by exiting a percentage of the samples locally and the rest in the cloud. The accuracy of "Cloud" and "Local" are computed by exiting all samples at each point, respectively. The end devices are ordered by their "Individual" classification accuracy, sorted from worst to best.

we configure the local exit threshold T such that around 75% of samples are exited locally and around 25% of samples are offloaded to the cloud. We see that DDNNs achieve around a 5% improvement in accuracy compared to using just the local aggregator. This demonstrates the advantage of offloading to the cloud even when larger models (more filters) with improved local accuracy are used on the end devices.

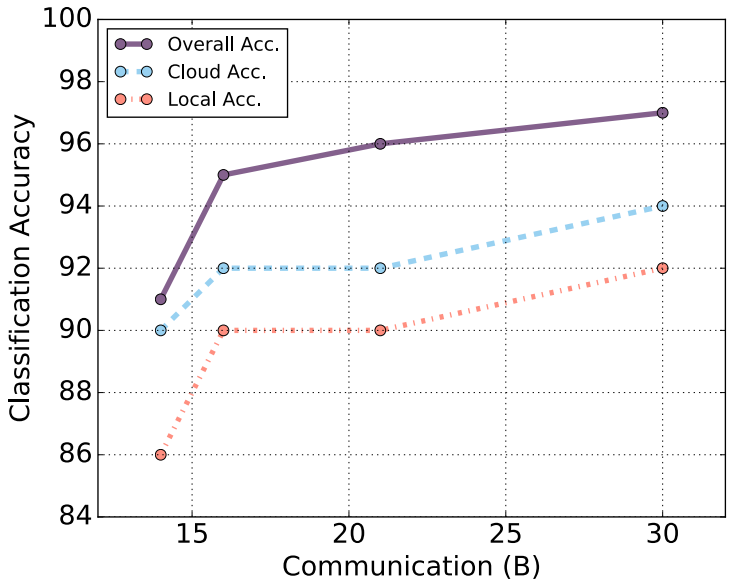

Figure 4.9: Accuracy vs. communication cost (in bytes, as defined in Section 4.6.1) for increasingly larger end device memory sizes that accommodate additional filters. We notice that cloud offloading leads to improved accuracy.

4.5.8 Fault Tolerance of DDNNs

A key motivation for distributed systems is fault tolerance. Fault tolerance implies that the system still works well even when some parts are broken. In order to test the fault tolerance of DDNN, we simulate end device failures and look at the resulting accuracy of the system. Figure 4.10 shows the accuracy of the system under the presence of individual device failures. Regardless of the device that is missing, the system still achieves over a 95% overall classification accuracy. Specifically, even when the device with the highest individual accuracy (Device 6) has failed, the overall accuracy is reduced by only 3%. This suggests that for this dataset, the automatic fault toler-

ance provided by DDNN makes the system reliable even in the presence of device failure.

We can also view Figure 4.8 from the perspective of providing fault tolerance for the system. As we decrease the number of end devices from 6 to 4, we observe that the overall accuracy of the system drops only 4%. This suggests that the system can also be robust to multiple failing end devices.

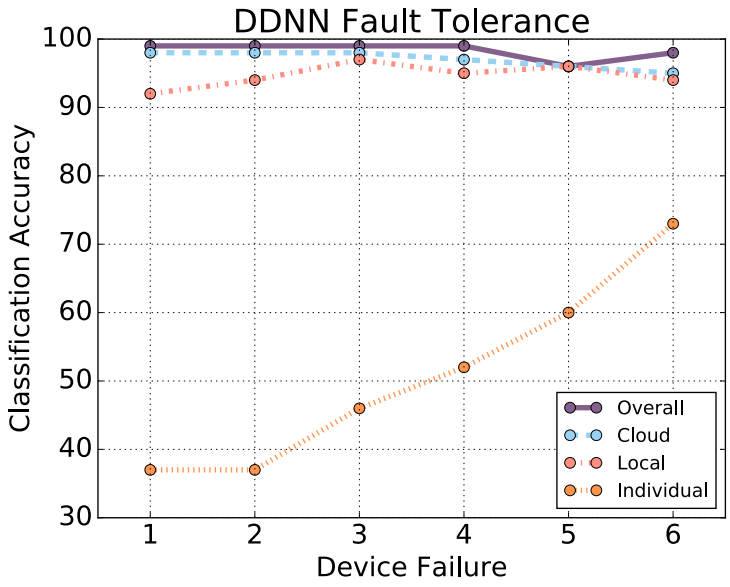

Figure 4.10: Accuracy vs. device that fails. The impact on DDNN system accuracy when any single end device has failed is minimal.

4.5.9 Reducing Communication Costs

DDNNs significantly reduces the communication cost (as defined in Section 4.6.1) of inference compared to the standard method of offloading raw sensor input to the cloud. Sending a 32×32 RGB pixel image (the input size of our dataset) to the cloud costs 3,072 bytes per image sample. By comparison, as shown in Table 4.2, the largest DDNN model used in our evaluation section requires only 140 bytes of communication per sample on average (an over 20x reduction in communication costs). This communication reduction for an end device results from only transmitting class-label related intermediate results to the local aggregator for all samples and binary-value intermediate results to the cloud for samples that need additional NN layer processing for better classification.

4.6 Discussion

In this section, we discuss different aspects of DDNN, including communication costs and provisioning for horizontal and vertical scaling.

4.6.1 Communication Cost of DDNN Inference

The total communication cost for an end device with the local and cloud aggregator is calculated as

$$c = 4 \times |\mathcal{C}| + (1 - p)\frac{f \times o}{8} \tag{4.1}$$

where p is the percentage of samples exited locally, $\mathcal{C}$ is the set of all possible labels (3 in our experiments), f is the number of filters, and o is the output size of a single filter for the final NN layer on the end-device. The constant 4 corresponds to the 4 bytes that are used to represent a floating-point number and the constant 8 corresponds to bits used to express a byte output. The first term assumes a single floating-point per class, which conveys the probability that the sample to be transmitted from the end device to the local aggregator belongs to this class. This step happens regardless of whether the sample is exited locally or at a later exit point. The second term is the communication between the end device and the cloud which happens $(1 - p)$ fraction of the time, when the sample is exited in the cloud rather than locally.

4.6.2 Provisioning for Horizontal and Vertical Scaling

The evaluation in the previous section shows that DDNN is able to achieve high overall accuracy through provisioning the network to scale both horizontally, across end devices, and vertically, over the network hierarchy. Specifically, we show that DDNN

scales vertically, by exiting easier input samples locally for low-latency response and offloading difficult samples to the cloud for high overall recognition accuracy, while maintaining a small memory footprint on the end devices and incurring a low communication cost. This result is not obvious, as we need sufficiently good feature representations from the lower parts of the DNN (running on the end devices with limited resources) in order for the upper parts of the neural network (running in the cloud) to achieve high accuracy under the low communication cost constraint. Therefore, we show in a positive way that the proposed method of joint training a single DNN with multiple exit points at each part of the distributed hierarchy allows us to meet this goal. That is, DDNN optimizes the lower parts of the DNN to create a sufficiently good feature representations to support both samples exited locally and those processed further in the cloud.

To meet the goal of horizontal scaling, we provide a principled way of joint training a DNN with inputs from multiple devices through feature pooling via local and cloud aggregators, and demonstrate that by aggregating features from each device we can dramatically improve the accuracy of the system both at the local and cloud level. Filters on each device are automatically tuned to process the geographically unique inputs, and work together toward to the same overall objective leading to high overall accuracy. Additionally, we show that DDNN provides built-in fault tolerance across the end devices and is still able to achieve high accuracy in the presence of failed de-

vices.

4.7 Related Works

The framework of a large-scale distributed computing hierarchy has assumed new significance in the emerging era of Internet of Things (IoT). It is widely expected that most of the data generated by the massive number of IoT devices must be processed locally on the devices or at the edge, for otherwise the total amount of sensor data for a centralized cloud would overwhelm the communication network bandwidth. In addition, a distributed computing hierarchy offers opportunities for system scalability, data security and privacy, as well as shorter response times (see, *e.g.*, [41, 46]). For example, in [46], a face recognition application shows a reduced response time is achieved when a smartphone's photos are proceeded by the edge (fog) as opposed to the cloud. DDNN can systematically exploit the inherent advantages of a distributed computing hierarchy for DNN applications and achieve similar benefits.

In DDNN, exit points are placed at physical boundaries (*e.g.*, between the last neural network (NN) layer on an end device and the first NN layer in the next higher layer of the distributed computing hierarchy such as the edge or the cloud). Input samples that can already be classified early will exit locally, thereby achieving a lowered response latency and saving communication with the next physical boundary. With similar objectives, SACT [47] allocates computation on a per region basis in an

image, and exits each region independently when it is deemed to be of sufficient quality.

Current research on distributing deep neural networks is mainly focused on improving the runtime of training the neural network. In 2012, Dean *et al.* proposed DistBelief, which maps large DNNs over thousands of CPU cores during training [48]. More recently, several methods have been proposed to scale up DNN training across GPU clusters [49, 50], which further reduces the runtime of network training. Note that this form of distributing DNNs (over homogeneous computing units) is fundamentally different from DDNN. DDNN proposes a way to train and perform feedforward inference over deep networks that can be deployed over a distributed computing hierarchy, rather than processed in parallel over bus-or-switch-connected CPUs or GPUs in the cloud.

Federated Learning [51] enables distributed model training on a large corpus of decentralized data. Several variants and improvements of the original federated learning [52, 53, 54, 52, 55, 56] have been introduced in the recent years. While federated learning focuses on training on a large corpus of decentralized data, DDNN focuses on scaling DNN models across a distributed computing hierarchy. Both approaches are orthogonal. DDNN models can also be trained using federated learning on a large corpus of decentralized data.

4.8 Relevance within IDN

IDN can utilize the DDNN framework to scale DNN models for use with its peer-to-peer (p2p) network. With DDNN, DNN models can be mapped onto distributed computing hierarchies consisting of nearby peer devices, the edge, and the cloud on IDN, enabling flexibility in terms of the responsiveness and the complexity of the DNN models. The DDNN framework can be used to optimize DNN models mapped to different nearby devices in the computing hierarchy, allowing DNN models to scale beyond the limitations of a single device.

4.9 Conclusion

In this chapter, we present a novel distributed deep neural network architecture (DDNN) that is distributed across computing hierarchies, consisting of the cloud, the edge, and end devices. We demonstrate for a multi-view, multi-camera dataset that DDNN scales vertically from a few NN layers on end devices or the edge to many NN layers in the cloud, and scales horizontally across multiple end devices.

Building on top of BranchyNet presented in Chapter 3, DDNNs reduce end-to-end inference time and the required communication compared to a standard cloud offloading approach by exiting many samples at the local aggregator and sending a compact binary feature representation to the cloud when additional processing is required. For

our evaluation dataset, the communication cost of DDNN is reduced significantly compared to offloading raw sensor input to a DNN in the cloud that performs all of the inference computation.

The aggregation of information communicated from different devices is built into the joint training of DDNN and is handled automatically at inference time. This approach simplifies the implementation and deployment of distributed cloud offloading, and automates sensor fusion and system fault tolerance. The results suggest that with our DDNN framework, a single DNN properly trained can be mapped onto a distributed computing hierarchy to meet the accuracy, communication, and latency requirements of a target application while gaining inherent benefits associated with distributed computing such as fault tolerance and privacy.

In this chapter, we have discussed how we can scale DNN models by distributing the computation across computing hierarchies. In the next chapter (Chapter 5), we will explore a way to make DNN models fault tolerant and more flexible to be run on heterogeneous devices of different hardware.

5

ParallelNet: Fault Tolerant Inference via Independent Parallel DNNs

The Internet connects devices with different capabilities in terms of CPU/GPU power and memory and different connectivity patterns. At different points in time, these

devices will have spare resources available with which to perform additional computation at little or no additional cost. In this chapter, we explore how a deep neural network can be trained to run on these heterogeneous computing resources independently in parallel, reducing inference time and increasing the fault tolerance of the DNN models.

We present ParallelNet, a framework to adapt and train a deep neural network model that can generate multiple models of various sizes. Each generated model can be executed independently in parallel on heterogeneous devices of different hardware. The outputs of each generated model are then combined to form the final prediction. When enough confidence of the prediction is accumulated, the rest of the computation is terminated.

The proposed framework works by multiplying each filter (in a convolutional layer) and neuron (in a linear layer) of a model with a coefficient $\beta \in \{0, 1\}$. At each training iteration, for each filter and neuron, the training algorithm randomly chooses $\beta \in \{0, 1\}$ according to a Bernoulli distribution with probability p of being 1. The augmented model is then trained as usual with the backpropagation algorithm. We call the resulting trained model a ParallelNet model.

For inference, a ParallelNet model is able to generate multiple smaller models that can be run independently in parallel. A ParallelNet model can generate a model by 1) randomly choosing $\beta \in \{0, 1\}$ according to a Bernoulli distribution with probability

p of being 1, and 2) removing a filter or neuron with β of 0. The resulting generated models can be run independently in parallel. The size of a model is controlled by selecting the value of p. The smaller the p, the smaller the model.

These smaller models are usually less capable than the original model when operating independently, but when they form an ensemble, they gain back some of the accuracy of the original model. Excitingly, these smaller models run faster than the original model, thus, reducing the inference time. The resulting speed and accuracy depend on the size of the ensemble and the models. ParallelNet offers a controllable trade-off between speed and accuracy of the inference. If higher accuracy is needed, a larger ensemble and larger models can be used. However, if faster inference time is needed, smaller models can be used instead. The size of a generated model is controlled by the parameter p. One can control the robustness of the inference by adjusting the number of models in the ensemble. The larger the ensemble, the more fault tolerant, the inference since a smaller fraction of the ensemble can form a highly confident prediction.

In addition, the models generated from ParallelNet can be executed in a progressive manner. The result of each individual model's computation can be independently presented to the end-user. For example, if the first model finishes its processing first, its result can be shown immediately while waiting for others to finish. Once another model has finished its computation, the result can be combined for an improved

prediction in a progressive manner. Or, if the end-user is already satisfied with the result, the in-progress computations can be interrupted, freeing up resources. This progressive execution is beneficial in an unreliable setting such as peer-to-peer (p2p) networks with devices of different capabilities. Additionally, if the result is already good enough (according to the predetermined exit criterion), it can be exited early and achieve faster inference time using techniques introduced in Chapter 3; the unfinished computation can be terminated, freeing up resources.

ParallelNet has the following properties:

- In ParallelNet, the exponentially many models are trained altogether and some of their parameters are shared.
- ParallelNet can generate exponentially many models.
- ParallelNet can generate models of different sizes to fit devices with different resources and capabilities.
- ParallelNet can generate models that can be run independently in parallel.
- ParallelNet reduces the inference time since the generated models, when run in parallel, are smaller and faster to run than the original model.
- ParallelNet when combined with early exiting offers even faster inference, and fault tolerance, since the rest of the computation can be terminated when the prediction has accumulated enough confidence from the models.

We now describe the architecture of ParallelNet, how to train ParallelNet, how to run

inference on ParallelNet, present some results, and finally discuss and conclude the chapter.

5.1 Architecture

An example of a ParallelNet model is shown in Figure 5.1. A model (Figure 5.1 (a)) is adapted into a ParallelNet model (Figure 5.1 (b)) by augmenting each filter and neuron with a Bernoulli random variable $\beta \sim \text{Bern}(p)$. A ParallelNet model can generate exponentially many different instances of the model. Figure 5.1 (c,d)) shows two examples of possible models generated by a ParallelNet model.

The parameter p of the Bernoulli random variable can be set per network, per layer, or per filter or neuron. It can be thought of as the density of active filters and neurons in the model. A higher p gives a higher number of active filters and neurons, resulting in a denser model.

5.2 Training ParallelNet

A ParallelNet model can be trained as usual with the backpropagation algorithm and one of the optimizers discussed in Section 2.4. The difference is that for each mini-batch iteration, a random model is trained. A random model is generated from drawing a value of $\beta \in \{0, 1\}$ for each filter and neuron from the Bernoulli distribution

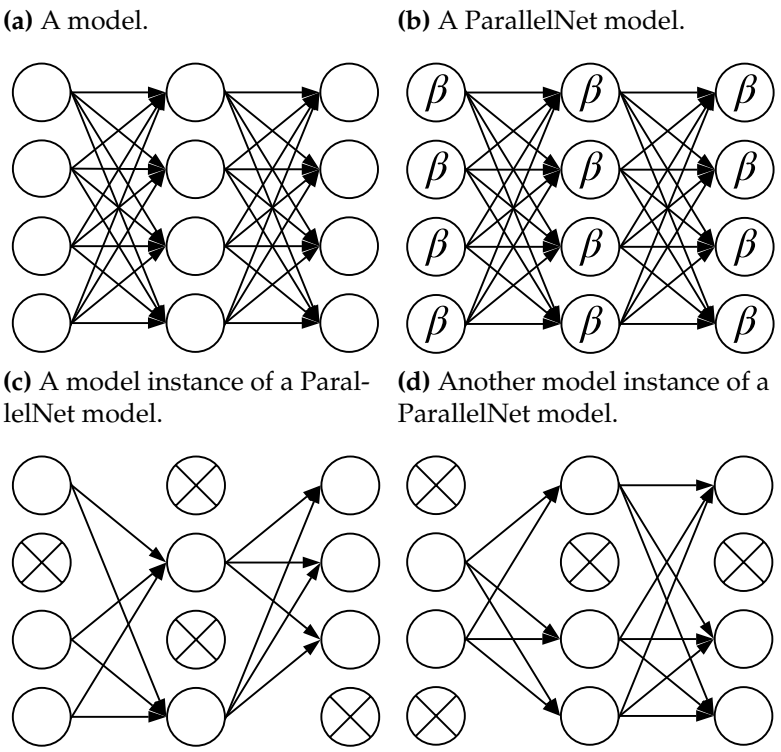

Figure 5.1: ParallelNet Architecture. A circle represents a filter or a neuron. $\beta \sim \text{Bern}(p)$ is a Bernoulli random variable. Blank circle represents an active filter or neuron. Cross circle represents a disabled filter or neuron.

with probability p of being 1. The β value dictates whether a filter or a neuron is active or not: 1 means active and 0 means inactive. p can be kept constant across the training iterations or can follow a specific schedule. A schedule may be, for example, p is 0.1 for iterations ending with 1 and 0.5 otherwise. This parameter p set during training dictates the ability of the network to learn as well as the ability of the models it generates at inference time. Figure 5.2 shows the training procedure.

```
1: procedure ParallelNetTraining(M, p, D, K)
2:     for each epoch k ∈ [1, K] do
3:         for each batch d ∈ D do
4:             M_i ← {}
5:             for each filter and neuron m ∈ M do
6:                 β ← Bern(p)
7:                 if β = 1 then
8:                     M_i ← M_i ∪ {m}
9:             Train M_i with d using backpropagation and gradient descent.
10:    return M
```

Figure 5.2: ParallelNet Training Algorithm. $\mathcal{M}$ is the model to train, represented as a set of many filters and neurons m. p is the parameter of the Bernoulli random variable (Bern(p)). To keep it simple, here p is constant across layers and iterations. $\mathcal{D}$ is the training dataset split into mini-batches. K is the number of epoch to train.

5.3 Fast Inference with ParallelNet

At inference time, a ParallelNet model generates many models of different density that fit each device's capability. The ensemble of these generated models are then combined together to form the final prediction.

From a ParallelNet model (a genesis model), a model is generated by drawing a value of $\beta \in \{0, 1\}$ for each filter and neuron from a Bernoulli distribution with probability p of being 1. Only a filter or a neuron having β of 1 is included in the new model. In other words, a filter or a neuron having β of 0 is removed from the original model. The resulting model is thinner and smaller than the genesis model. This thinner, smaller model can be used to run prediction independently. The value of p can be

different from the value(s) used during training and for each model generation. Figure 5.3 shows the pseudocode of this procedure.

```
1: procedure ParallelNetGenerateModel(M, p)
2:     M_i ← {}
3:     for each filter and neuron m ∈ M do
4:         β ← Bern(p)
5:         if β = 1 then
6:             M_i ← M_i ∪ {m}
7:     return M_i
```

Figure 5.3: ParallelNet Model Generation Algorithm. $\mathcal{M}$ is the genesis model. p is the parameter of the Bernoulli random variable (Bern(p)).

Once the desired number of models (N) are generated, these models are used to make predictions independently in parallel and their results are combined to form the final inference. Figure 5.4 shows the ParallelNetFastInference procedure. In this procedure, we average the outputs of the models before the softmax computation. Alternatively, one can compute the softmax of the output from each model before averaging. From our experiments, the two orderings do not have much effect in terms of accuracy; however, it is cheaper to compute the average first and then softmax since in this way, we only have to compute the softmax once. Instead of averaging, one may use geometric mean [57]: $\sqrt[n]{x_1 x_2 \cdots x_n}$.

To further reduce inference time, one can use the early-exit technique as discussed in Chapter 3 to terminate the computation early. Figure 5.5 shows the inference procedure with early exit. It works as follows. An input sample is broadcasted to N mod-

1: **procedure** PARALLELNETFASTINFERENCE($\boldsymbol{x}$)
2: $\quad \boldsymbol{z} \leftarrow \mathbf{0}$
3: $\quad$ **parfor** $n \in [1, N]$ **do**
4: $\quad\quad \boldsymbol{z}_n \leftarrow f_{\text{model}_n}(\boldsymbol{x})$
5: $\quad\quad \boldsymbol{z} \leftarrow \boldsymbol{z} + \boldsymbol{z}_n$
6: $\quad \hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{z}/N)$
7: $\quad$ **return** $\arg\max \hat{\boldsymbol{y}}$

Figure 5.4: ParallelNet Fast Inference Algorithm. **parfor** indicates that the execution in the block is done in parallel. $\boldsymbol{x}$ is an input sample, $f_{\text{model}_n}(\boldsymbol{x})$ is the compute function of the n-th model, N is the number of models and SHOULDEXIT($\hat{\boldsymbol{y}}$) is a function that returns *true* or *false* depending on whether an output probability vector of a branch is confident enough or have enough information to exit. Several choices of SHOULDEXIT($\hat{\boldsymbol{y}}$) functions are discussed in Section 3.4.

els to run in parallel. Once any model finishes its computation, the exit criterion is evaluated against a threshold T. If it passes, the sample is exited and the remaining computation is terminated. Otherwise, it waits for another model to finish, averages its result with the existing one, and reevaluates the exit criterion. If all models are computed, but the result is still not good enough, it just returns the average of all predictions. Additionally, it can return the result as it arrives and return an improved one as the new result becomes available in a progressive manner.

5.4 RESULTS

In this section, we demonstrate the effectiveness of ParallelNet in reducing the inference time by parallelizing the computation of the neural network inference on the image classification task.

```
1: procedure ParallelNetFastInferenceWithEarlyExit(x, T)
2:     z ← 0
3:     i ← 0
4:     parfor n ∈ [1, N] do
5:         z_n ← f_model_n(x)
6:         z ← z + z_n
7:         i ← i + 1
8:         ŷ = softmax(z/i)
9:         if ShouldExit(ŷ, T) then
10:            return arg max ŷ
11:    return arg max ŷ
```

Figure 5.5: ParallelNet Fast Inference with Early Exit Algorithm. **parfor** indicates that the execution in the block is done in parallel. $\boldsymbol{x}$ is an input sample, T is the threshold for determining whether to exit the sample, N is the number of models and ShouldExit$(\hat{\boldsymbol{y}}, T)$ is a function that returns *true* or *false* depending on whether an output probability vector of a branch is confident enough or have enough information to exit. Several choices of ShouldExit$(\hat{\boldsymbol{y}}, T)$ functions are discussed in 3.4.

5.4.1 Setup

The dataset used for the experiments is the CIFAR10 [25] dataset. There are 50,000 training images and 10,000 images testing images. The 50,000 training images is split into 45,000 for training and 5,000 for validation. We follow the standard data augmentation scheme for this dataset, in which the images are centered and normalized, zero-padded with 4 pixels on each side, randomly cropped to produce 32×32 images, and randomly flipped horizontally.

Unless mentioned otherwise, we use SGD with Cosine Annealing or SGDR [19] (with the learning rate of 0.005 and the period of 10) to train the networks with a batch

size of 128 and each network is trained for 2,000 epochs. Experiments are done in PyTorch [26] on Amazon EC2 p3.2xlarge instance with 1 Nvidia Tesla V100 16GB GPU.

For simplicity, we only describe a network in terms of styles (e.g., AlexNet-style [9, 10] or ResNet-style [17] or DenseNet-style [27]), convolutional layers and the number of filters each layer has, and linear (fully-connected, dense) layers and the number of neurons each layer has. Generally, these networks may also contain max pooling, non-linear activation functions (e.g., a rectified linear unit and sigmoid), normalization (e.g., local response normalization, batch normalization), and dropout.

We use an AlexNet-style [9, 10] network for the experiments to evaluate ParallelNet in terms of accuracy, the number of parameters, and runtime for various settings of the density of the filters and neurons (p) and the number of models (N). The AlexNet-style network used is shown in Figure 5.6. It consists of 8 layers: 5 convolutional layers, followed by 3 linear layers. The convolutional layers have 96, 288, 576, 384 and 384 filters, respectively. The linear layers have 1,536, 1,536 and 10 neurons, respectively.

To keep the experiments tractable, we use a single p for all filters and neurons of a model. Note that p can does not have to be the same for every filters and neurons; p can be set per layer or per filter or neuron for a more flexible control of the accuracy and speed trade-off.

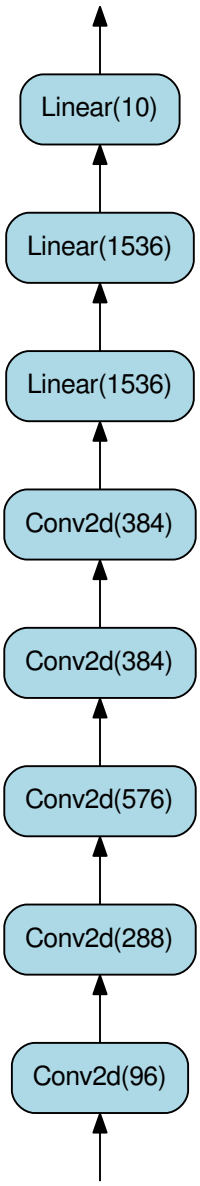

Figure 5.6: AlexNet-style network used in the experiments.

5.4.2 Accuracy and the Number of Parameters

First, we evaluate the ranges of accuracy and the number of parameters of the model ParallelNet generates. For this experiment we train the AlexNet-style network with $p = 0.45$. At inference time, we generate each model with varying p and observe the accuracy and the number of parameters.

Figure 5.7 shows the accuracy and the number of parameters of a generated model for a particular p. We observe the followings:

1. As p increases, the accuracy and the number of parameters increase.
2. p only needs to be around 0.45 for the model to reach around 95% of the maximum accuracy at $p = 1$ while using only 20% of the total parameters.
3. The confidence intervals are wider when p is lower since less parameters are shared among the generated models at that particular p. In other words, many more models can be generated when p is lower.
4. The gain (Accuracy/#Parameters) diminishes quickly as p approaches 1.

5.4.3 Accuracy vs. the Number of Models

In this experiment, we study how the number of models affects the accuracy for a number of p values. Figure 5.8 shows the result. We see that:

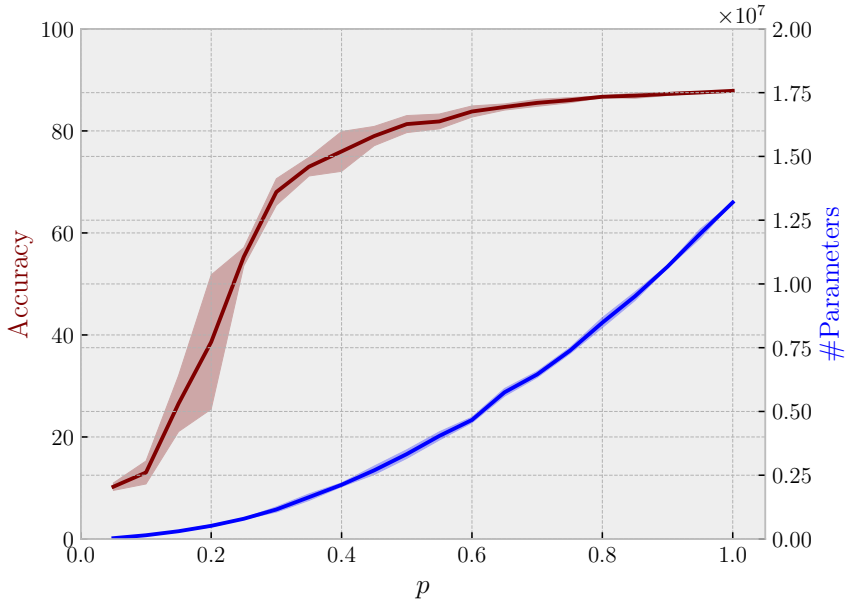

Figure 5.7: Accuracy and the number of parameters vs p. The solid line indicates the mean and the envelop indicates 95% confidence interval.

1. As the number of models increases, the accuracy generally increases.
2. In general, the higher the p value, the better the accuracy. However, higher p also means higher number of parameters and therefore, higher inference time.
3. $p = 0.80$ performs as good as or better than $p = 1.00$ when the number of models is sufficiently large. This shows that an ensemble of smaller models can perform as good as or better than one big model.
4. The accuracy gain reduces as more models are added and the accuracy of the best p flattens out at around 88%.

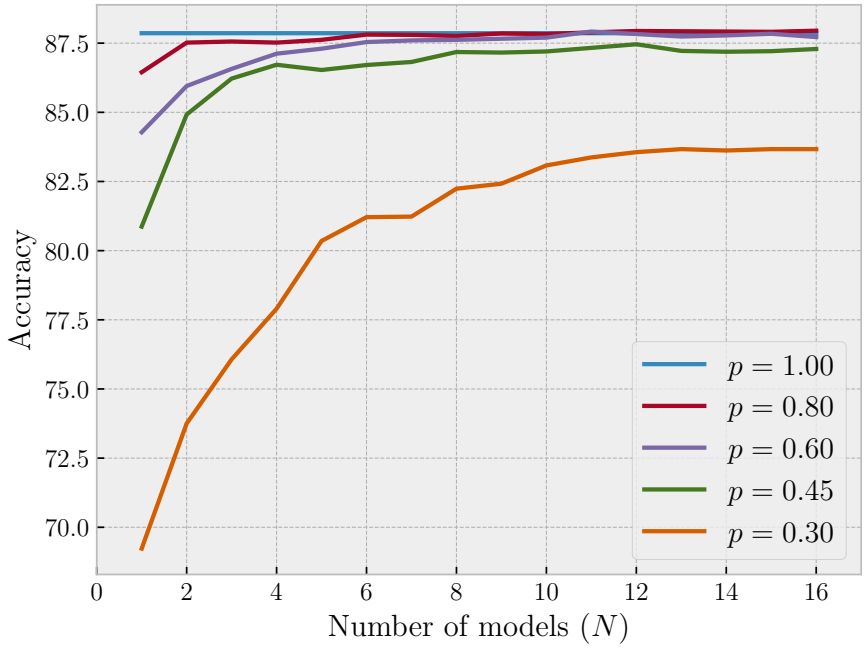

Figure 5.8: Accuracy vs. the number of models for various p used to generate the models. $p = 1.00$ is the same as the genesis model without removing any filters or neurons.

5.4.4 Inference Time Improvement

In this experiment, we see how ParallelNet can improve the inference time while maintaining a similar level of accuracy as the full model ($p = 1$). In ParallelNet, all N models run in parallel. The runtime is calculated as the average time it takes for a model to infer a sample. Figure 5.9 shows the result for the runtime on GPU. We observe that:

1. At $N = 8$, $p = 0.45$ performs almost as well as the full model (less than 1% worse) while gaining 1.6x speedup.

2. With high enough p, we do not need many models to achieve the accuracy of the full model ($p = 1$). For example, with $p = 0.60$ and $p = 0.80$, we only need 6 to 8 models.

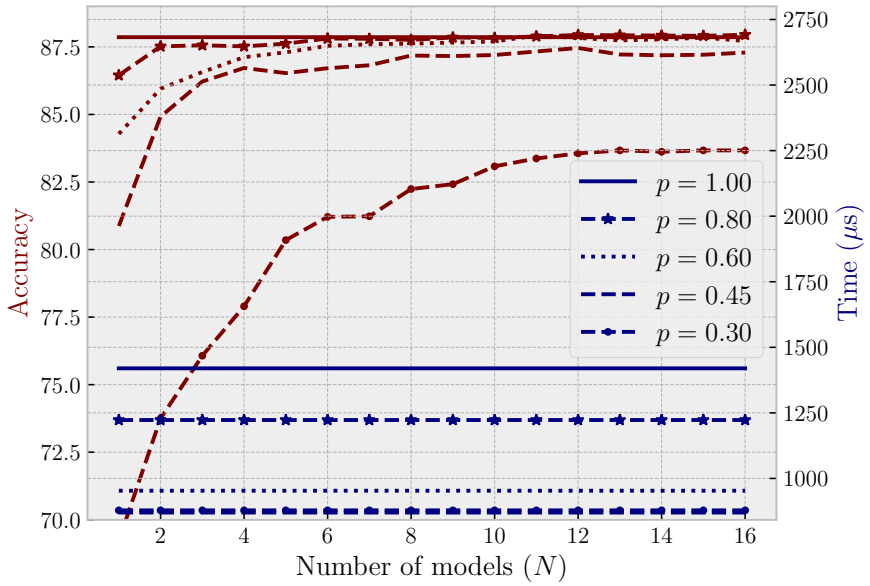

Figure 5.9: Accuracy and the average time per sample on GPU vs. the number of models for various p used to generate the models. $p = 1$ is the same as the genesis model without removing any filters or neurons. The average run time is 1,419μs, 1,222μs, 953μs, 869μs and 878μs for $p = 1.00$, $p = 0.80$, $p = 0.60$, $p = 0.45$ and $p = 0.30$, respectively.

Figure 5.10 shows the result for the runtime on CPU. We observe similar behavior as before with the following additional points:

1. On CPU, $p = 0.45$ now gains 3.2x speedup.
2. On CPU, $p = 0.30$ now gains 5.5x speedup with the fastest time of 3,889μs.

Note that these experiments are run on top-of-the-line high-performance GPU/CPU in the cloud and the model used is rather small. Much better speedup in inference

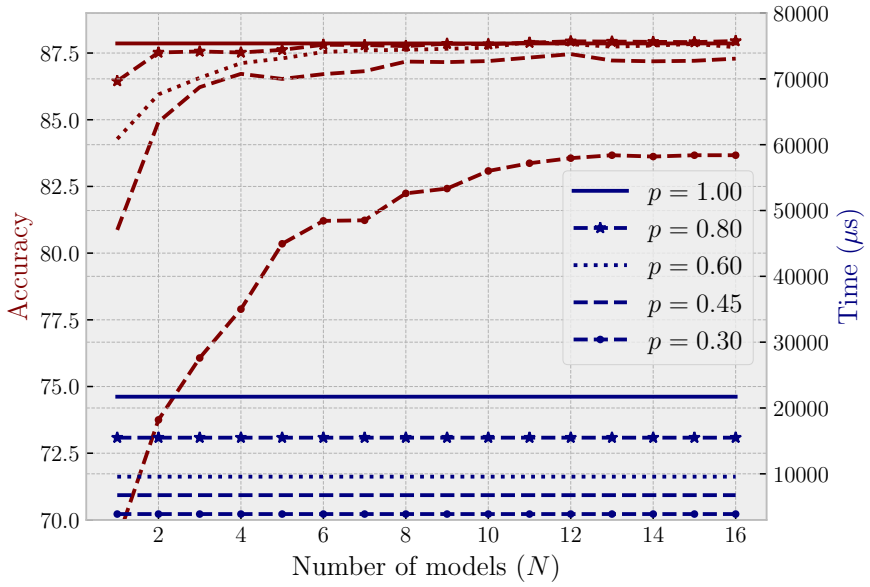

Figure 5.10: Accuracy and the average time per sample on CPU vs. the number of models for various p used to generate the models. $p = 1$ is the same as the genesis model without removing any filters or neurons. The average run time is 21,715μs, 15,491μs, 9,568μs, 6,760μs and 3,889μs for $p = 1.00$, $p = 0.60$, $p = 0.45$ and $p = 0.30$, respectively.

time can be expected when using a bigger model and slower GPU/CPU on a typical computer or edge device.

5.4.5 Inference Time Improvement with Early Exit

In this experiment, we study the effect of early-exit thresholds on the inference time and accuracy of ParallelNet. The inference procedure used is as described in Figure 5.5. We use an AlexNet-style network as described previously and entropy as the exit criterion (See Section 3.4.2). 8 models are generated with $p = 0.25$, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85 and 0.95. We vary the early-exit threshold from 0 to 1.0 and record ac-

curacy vs. inference time. All models run in parallel on a GPU; the inference time is the time it takes for a sample to accumulate enough confidence to exit according to a certain threshold T.

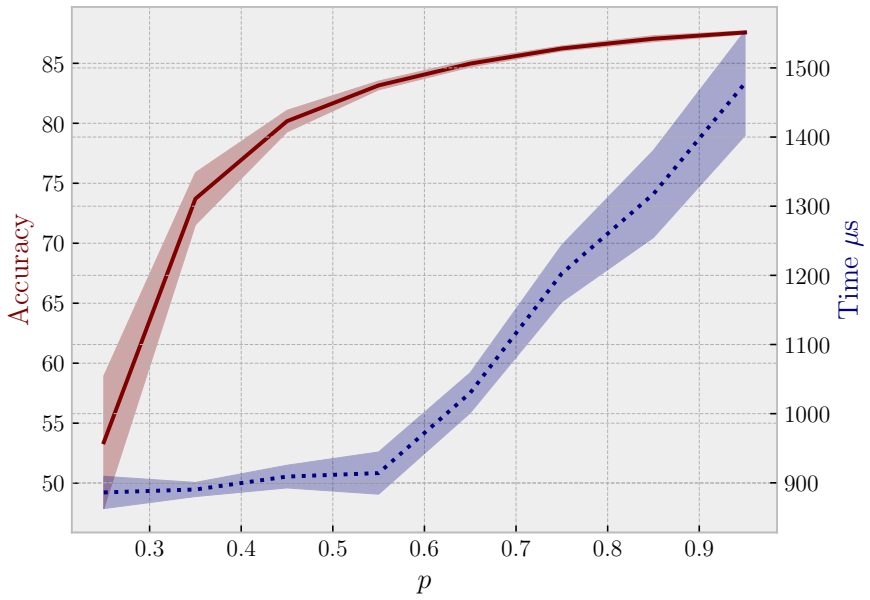

Figure 5.11: The inference time of a generated model vs. various settings of p used to generated it. The data points are generated with $p = 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85$ and 0.95, respectively.

First, let us take a look at the performance of each model generated with different p. Figure 5.11 shows the accuracy and inference time of a generated model vs. various settings of p used to generated it. As expected, as p increases, the accuracy and inference time increases. Note that the inference time increases rapidly while the gain in accuracy diminishes.

Next, let us compare the accuracy and inference time we can achieve with and with-

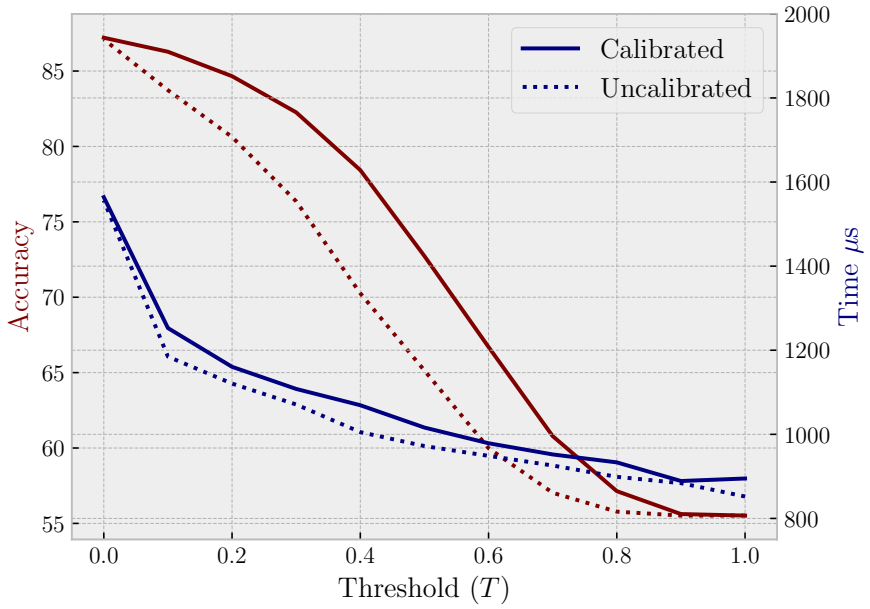

Figure 5.12: Comparing the accuracy and inference time vs. threshold (T) of the ensemble of 8 models (generated with various p) with calibrated confidence and uncalibrated one.

out confidence calibration (See Section 3.4.3) as we vary the exit threshold (T). Figure 5.12 shows the accuracy and inference time vs. exit threshold (T) of a model with calibrated and uncalibrated confidence. With calibrated confidence, we can achieve higher accuracy for various threshold T while incurring a slight increase in inference time. With confidence calibration, the confidence values are consistent across the models. More samples can exit correctly, resulting in a better accuracy. For the rest of this section, we will use a calibrated version of the network, unless mentioned otherwise.

The early exiting mechanism allows one to further trade off accuracy for inference time. Figure 5.13 shows accuracy vs. average inference time per sample. To achieve

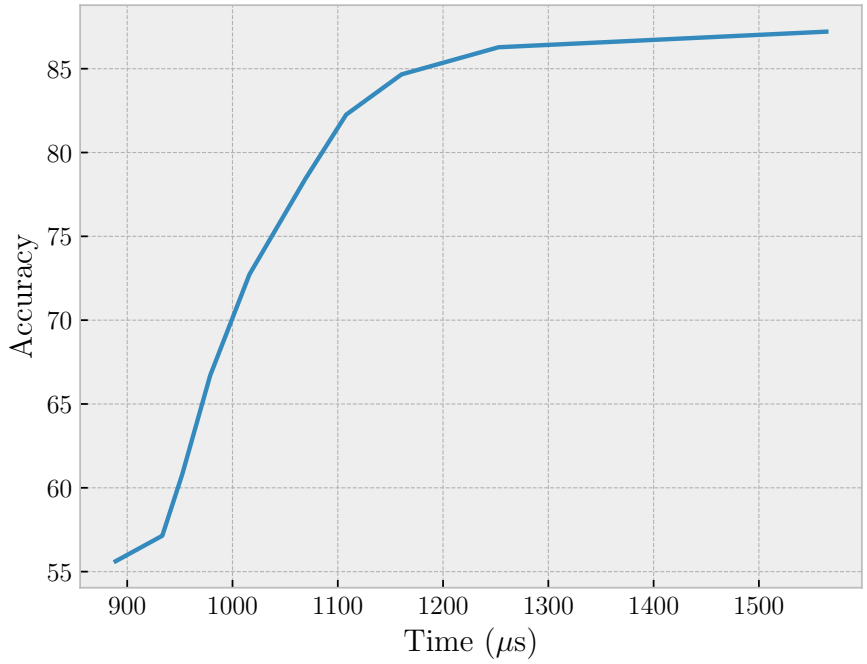

Figure 5.13: The accuracy vs. the average inference time per sample.

an accuracy of 85.56%, the algorithm takes around 1,200μs, and to achieve the full accuracy of 87.86%, the algorithm takes 1,650μs. With early exit, we achieve 1.4x speed up while giving up around 2% in accuracy.

5.5 Discussion

In this section, we discuss the ParallelNet's hyperparameter p and how it affects training time and accuracy of the models. We also discuss robustness and fault tolerance of ParallelNet in a distributed setting.

5.5.1 Effect of p on Training Time

We start by comparing the training time of each p set at training.

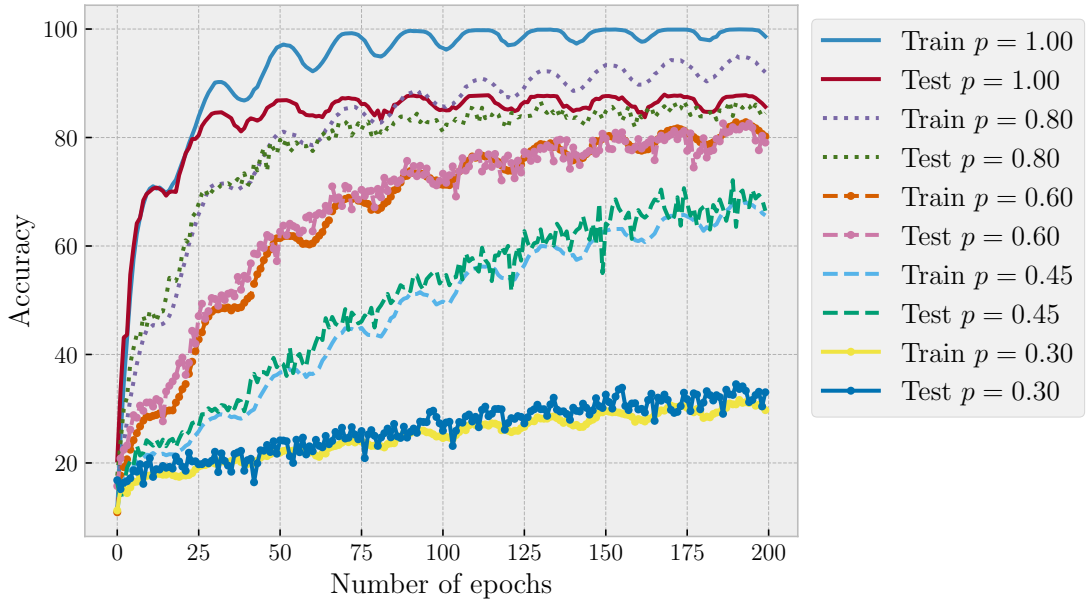

Figure 5.14: The training and testing accuracy of the first two hundred epochs for various p set at training.

Figure 5.14 compares the training and test accuracy over the number of epochs of various p settings. Generally, we observe that at a lower p setting, the network takes longer to train and converge. $p = 1.00$ is the fastest to train and converge at around 50 epochs, followed by $p = 0.80$ at around 100 epochs. $p = 0.30$ is the slowest to train and converge. This is because when $p = 1.00$, the network has full backpropagation signal to train its weights while when $p = 0.3$, the network only has about 30% of the signal propagate back through the network to train its weights.

5.5.2 Effect of p on Accuracy

We now discuss how the hyperparamter p set at training affects the accuracy of the models generated using various p settings at test time.

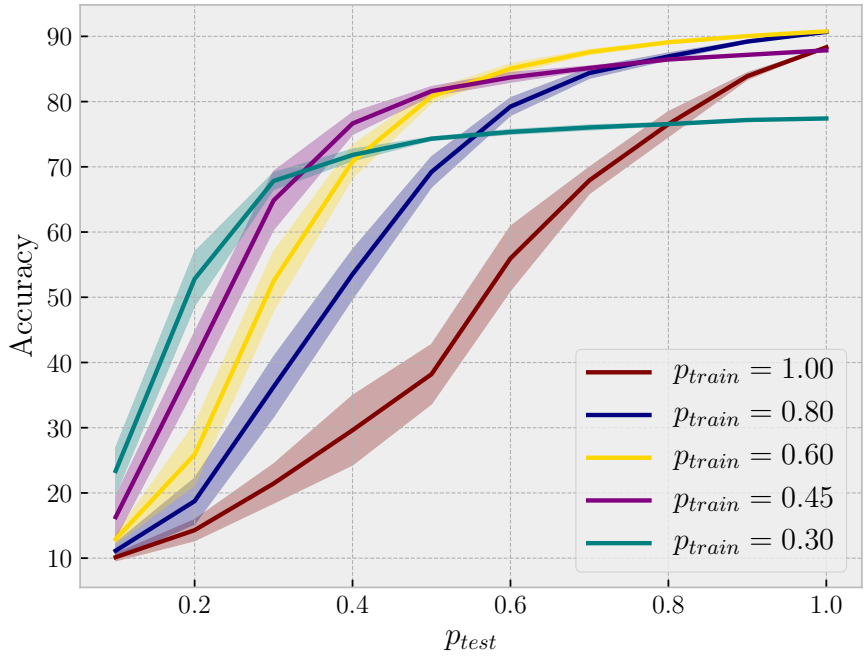

Figure 5.15: The accuracy vs. p used to generate a model (p_{test}) for various p settings at training time (p_{train}).

Figure 5.15 compares accuracy vs. p used to generate a model (p_{test}) for various p set at training time (p_{train}). We observe that as p_{test} increases, the accuracy generally goes up, regardless of how we set p_{train}. For $p_{\text{train}} = 0.30$, we see that the ParallelNet model works best at lower p_{test} settings. When $p_{\text{train}} = 1.00$, the network does not work so well at lower p_{test} settings. The model generally does well at the p that it is

trained on. This observation allows model developers to pick the accuracy and speed trade-off profile by setting different p_{train} and p_{test} as they design and train their ParallelNet models.

5.5.3 Fault Tolerance

In this section, we discuss the fault tolerance of DNN inference using ParallelNet. In ParallelNet, each generated model is independent of the others. Each model takes an input sample and outputs a final prediction with certain confidence. In a distributed setting where each model is run on a different compute device, one or more nodes may return an arbitrary result or not return the result due to problems with network connectivity or node failure.

Two failure modes are considered. One is a nonresponsive failure where a device fails to return a result before a response period expires. Another is a byzantine failure where a device may return an arbitrary result.

In the case where some devices fail to respond within a response period, ParallelNet still has the results of other devices that run the inference independently in parallel. The responses from these devices will cover the failure of the devices that fail to respond within a response period.

In the case where a device return an arbitrary result which could be malicious, Par-

allelNet can use the results of the other devices to cross check and ignore the result given by the failed or malicious device.

In both cases, ParallelNet is still able to return the final prediction result (possibly with less confidence and accuracy). As long as at least one device returns the correct result, DNN inference service will still be available.

5.6 Related Works

Previous works such as Dropout [23], Dropconnect [58], Swapout [59] and Stochastic Depth[38] have explored similar technique of drawing from Bernoulli random variables to enable or disable parts of the network to reduce overfitting and co-adaptation of neurons and filters. This work, however, focuses on using this technique to parallelize the computation of DNN for fast and increased robustness of DNN inference in a p2p network setting.

5.7 Relevance within IDN

Peers on IDN have different connectivity patterns and hardware. ParallelNet provides a framework that enables the generation of a different-sized DNN model for each device of varying compute capacity. Each device then runs its model independently in parallel, allowing for increased robustness to faults that are common in a

distributed peer-to-peer (p2p) network due to intermittent network connectivity failures.

5.8 Conclusion

ParallelNet is a framework to adapt and train a deep network model that can generate multiple models of various sizes. These generated models are capable of running independently in parallel on different-sized devices. Executing these models in parallel speeds up the inference time while maintaining reasonable accuracy level as the predictions of all models are combined to form the final strong prediction.

In a distributed setting, each of the generated models of a ParallelNet model can be run on multiple devices with different capabilities. In this setting, ParallelNet provides fault tolerance and robustness of inference computation. Since each model can generate the prediction independently, regardless of the prediction of other models, if there are problems with network connectivity or node failures, ParallelNet can still return its prediction as long as at least one of the models returns a prediction.

The main contributions in this chapter include:

- Introducing ParallelNet, a framework to adapt and train a deep network model that can generate multiple models of various sizes to run independently and in parallel.

- Parallel execution and early termination procedure of ParallelNet model to speed up inference time.
- Fault tolerance of inference in a distributed setting, resulting from ParallelNet.

In this and the previous chapters, we have discussed frameworks and deep neural network motifs for accelerating and scaling DNN inference, and making the inference more robust and fault-tolerant. Not matter how fast, scalable, fault tolerant the neural network is, if it is not available for people to use, it will not be that useful. In the next chapter, we will see an incentive mechanism to encourage peers to collaborate, improve, and share DNN models.

6

DaiMoN: A Decentralized Artificial Intelligence Model Network

Network-based services are at the intersection of a revolution. Many centralized monolithic services are being replaced with decentralized microservices. The utility

of decentralized ledgers showcases this change, and has been demonstrated by the usage of Bitcoin [60] and Ethereum [61].

The same trend towards decentralization is expected to affect the field of artificial intelligence (AI), and in particular machine learning, as well. Complex models such as deep neural networks require large amounts of computational power and resources to train. Yet, these large, complex models are being retrained over and over again by different parties for similar performance objectives, wasting computational power and resources. Currently, only a relatively small number of pretrained models such as pretrained VGG [15], ResNet [62], and GoogLeNet [16] are made available for reuse.

One reason for this is that the current system to share models is centralized, limiting both the number of available models and incentives for people to participate and share models. Examples of these centralized types of systems are Caffe Model Zoo [63], Pytorch Model Zoo [64], Tensorflow Model Zoo[65], and modelzoo.co [66].

In other fields seeking to incentivize community participation, cryptocurrencies and cryptographic tokens based on decentralized ledger technology (DLT) have been used [60, 61]. In addition to incentives, DLT offers the potential to support transparency, traceability, and digital trust at scale. The ledger is append-only, immutable, public, and can be audited and validated by anyone without a trusted third-party.

In this chapter, we introduce DaiMoN, a decentralized artificial intelligence model network that brings the benefits of DLT to the field of machine learning. DaiMoN uses DLT and a token-based economy to incentivize people to improve machine learning models. The system will allow participants to collaborate on improving models in a decentralized manner without the need for a trusted third-party. We focus on applying DaiMoN for collaboratively improving classification models based on deep learning. However, the presented system can be used with other classes of machine learning models with minimal to no modification.

In traditional blockchains, proof-of-work (PoW) [60] incentivizes people to participate in the consensus protocol for a reward and, as a result, the network becomes more secure as more people participate. In DaiMoN, we introduce the concept of *proof-of-improvement* (PoI). PoI incentivizes people to participate in improving machine learning models for a reward and, as an analogous result, the models on the network become better as more people participate.

One example of a current centralized system that incentivizes data scientists to improve machine learning models for rewards is the Kaggle Competition system [67], where a sponsor puts up a reward for contestants to compete to increase the accuracy of their models on a test dataset. The test dataset inputs are given while labels are withheld to prevent competitors from overfitting to the test dataset.

In this example, a sponsor and competitors rely on Kaggle to keep the labels secret.

If someone were to hack or compromise Kaggle's servers and gain access to the labels, Kaggle would be forced to cancel the competition. In contrast, because DaiMoN utilizes a DLT, it eliminates this concern, as it does not have to rely on a centralized trusted entity.

However, DaiMoN faces a different challenge: in a decentralized ledger, all data are public. As a result, the public would be able to learn about labels in the test dataset if it were to be posted on the ledger. By knowing test labels, peers may intentionally overfit their models, resulting in models which are not generalizable. To solve the problem, we introduce a novel technique, called *Distance Embedding for Labels* (DEL), which can scramble the labels before putting them on the ledger. DEL preserves the error in a label vector inferred by the classifier with respect to the test label vector of the test dataset, so there is no need to divulge the labels themselves.

With DEL, we can realize the vision of PoI over a DLT network. That is, any peer verifier can vouch for the accuracy improvement of a submitted model without having access to the test labels. The proof is then included in a block and appended to the ledger for the record.

The structure of this chapter is as follows: after introducing DEL and PoI, we introduce the DaiMoN system that provides incentive for people to participate in improving machine learning models.

The contributions of this chapter include:

1. A learnable Distance Embedding for Labels (DEL) function specific to the test label vector of the test dataset for the classifier in question, and performance analysis regarding model accuracy estimation and security protection against attacks. To the best of our knowledge, DEL is the first solution which allows peers to verify model quality without knowing the test labels.
2. Proof-of-improvement (PoI), including detailed PROVE and VERIFY procedures.
3. DaiMoN, a decentralized artificial intelligence model network, including an incentive mechanism. DaiMoN is one of the first proof-of-concept end-to-end systems in distributed machine learning based on decentralized ledger technology.

6.1 DISTANCE EMBEDDING FOR LABELS

In this section, we describe Distance Embedding for Labels (DEL), a key technique by which DaiMoN can allow peers to verify the accuracy of a submitted model without knowing the labels of the test dataset. By keeping these labels confidential, the system prevents model-improving peers from overfitting their models intentionally to the test labels.

6.1.1 Learning the DEL Function with Multi-Layer Perceptron

Suppose that the test dataset for the given C-class classification problem consists of m (input, label) test pairs, and each label is an element in $\mathcal{Q} = \{c \in \mathbb{Z} \mid 1 \leq c \leq C\}$, where $\mathbb{Z}$ denotes the set of integers. For example, the FashionMNIST [68] classification problem has $C = 10$ image classes and $m = 10,000$ (input, label) test pairs, where for each pair, the input is a 28×28 greyscale image, and the label is an element in $\mathcal{Q} = \{1, 2, \ldots, 10\}$.

For a given test dataset, we consider the corresponding *test label vector* $\boldsymbol{x}_t \in \mathcal{Q}^m$, which is made of all labels in the test dataset. We seek a $\boldsymbol{x}_t$-specific DEL function $f : \boldsymbol{x} \in \mathcal{Q}^m \to \boldsymbol{y} \in \mathbb{R}^n$, where $\mathbb{R}$ denotes the set of real numbers, which can approximately preserve distance from a label vector $\boldsymbol{x} \in \mathbb{R}^m$ (inferred by a classification model or a classifier we want to evaluate its accuracy) to $\boldsymbol{x}_t$, where $n \ll m$. For example, we may have $m = 10,000$ and $n = 256$. The *error* of $\boldsymbol{x}$ or the distance from $\boldsymbol{x}$ to $\boldsymbol{x}_t$ is defined as

$$\mathrm{e}(\boldsymbol{x}, \boldsymbol{x}_t) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}(x_i \neq x_{t_i}),$$

where $\mathbb{1}$ is the indicator function, $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ and $\boldsymbol{x}_t = \{x_{t_1}, \ldots, x_{t_m}\}$.

Finding a distance-preserving embedding function is generally a challenging mathematical problem. Fortunately, we have observed empirically that we can learn this $\boldsymbol{x}_t$-specific embedding function using a neural network.

More specifically, to learn an $\boldsymbol{x}_t$-specific DEL function f, we train a multi-layer perception (MLP) for f as follows. For each randomly selected $\boldsymbol{x} \in \mathcal{Q}^m$, we minimize the loss:

$$\mathcal{L}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{x}_t) = |\mathrm{e}(\boldsymbol{x}, \boldsymbol{x}_t) - \mathrm{d}(f(\boldsymbol{x}), f(\boldsymbol{x}_t))|,$$

where $\boldsymbol{\theta}$ is the MLP parameters, and $\mathrm{d}(\cdot, \cdot)$ is a modified cosine *distance* function defined as

$$\mathrm{d}(\boldsymbol{y}_1, \boldsymbol{y}_2) = \begin{cases} 1 - \frac{\boldsymbol{y}_1 \cdot \boldsymbol{y}_2}{\|\boldsymbol{y}_1\| \|\boldsymbol{y}_2\|} & \boldsymbol{y}_1 \cdot \boldsymbol{y}_2 \geq 0 \\ 1 & \text{otherwise.} \end{cases}$$

The MLP training finds a distance-preserving low-dimensional embedding specific to a given $\boldsymbol{x}_t$. The existence of such embedding is guaranteed by the Johnson-Lindenstrauss lemma [69, 70], under a more general setting which does not have the restriction about the embedding being specific to a given vector.

6.1.2 Use of DEL Function

We use the trained DEL function f to evaluate the accuracy or the error of a classification model or a classifier on the given test dataset without needing to know the test labels. As defined in the preceding section, for a given test dataset, $\boldsymbol{x}_t \in \mathcal{Q}^m$ is the test label vector of the test dataset. Given a classification model or a classifier, $\boldsymbol{x} \in \mathcal{Q}^m$

is a label vector consisting of labels inferred by the classifier on all the test inputs of the test dataset. A verifier peer can determine $\boldsymbol{x}$'s distance to $\boldsymbol{x}_t$ without knowing $\boldsymbol{x}_t$, by checking $\mathrm{d}(f(\boldsymbol{x}), f(\boldsymbol{x}_t))$ instead of $\mathrm{e}(\boldsymbol{x}, \boldsymbol{x}_t)$. This is because these two quantities are approximately equal, as assured by the MLP training, which minimizes their absolute difference. If $\mathrm{d}(f(\boldsymbol{x}), f(\boldsymbol{x}_t))$ is deemed to be sufficiently lower than that of the previously known model, then the peer may conclude that the model has improved the accuracy of the test dataset.

Note that the DEL function f is $\boldsymbol{x}_t$-specific. That is, for a different test dataset with a different test label vector $\boldsymbol{x}_t$, we will need to train another f. For most model benchmarking applications, we expect a stable test dataset; and thus we will not need to retrain f frequently.

6.2 Evaluation of Learnt DEL Function

In this section, we evaluate how well the neural network approach described above can learn a DEL function $f : \mathcal{Q}^m \to \mathbb{R}^n$ with $m = 10,000$ and $n = 256$. We consider a simple multi-layer perceptron (MLP) with 1,024 hidden units and a rectified linear unit (ReLU). The output of the network is normalized to a unit length. The network is trained using the Adam optimization algorithm [18]. The dataset used is FashionMNIST [68], which has $C = 10$ classes and $m = 10,000$ (input, label) test pairs.

To generate the data to train the function, we perturb the test label vector $\boldsymbol{x}_t$ by using the following GENERATEDATA procedure.

1: **procedure** GENERATEDATA($\boldsymbol{x}_t$)
2: Pick a random number v in $\{1, 2, \ldots, m\}$
3: Pick a random set $\mathcal{K}$ in $\{1, 2, \ldots, m\}^v$
4: Initialize $\boldsymbol{x}$ as $\{x_1, x_2, \ldots, x_m\}$ with $\boldsymbol{x} \leftarrow \boldsymbol{x}_t$
5: **for** $k \in \mathcal{K}$ **do**
6: Pick a random number c in $\{1, 2, \ldots, C\}$
7: $x_k \leftarrow c$
8: **return** $\boldsymbol{x}$

First, the procedure picks v, the number of labels in $\boldsymbol{x}_t$ to switch out, and generates the set of indices $\mathcal{K}$, indicating the positions of the label to replace. It then loops through the set $\mathcal{K}$. For each $k \in \mathcal{K}$, it generates the new label c to replace the old one. Note that with this procedure, the new label c can be the same as the old label.

We use the procedure to generate the training dataset and the test dataset for the MLP. Figure 6.1 shows the convergence of the network in learning the function f. We see that as the number of epochs increases, both training and testing loss decrease, suggesting that the network is learning the function.

After the neural network has been trained, we evaluate how well the learned f can preserve error in a label vector $\boldsymbol{x}$ inferred by the classifier. Figure 6.2 shows the correlation between the error and the distance in the embedding space under f. We see that both are highly correlated.

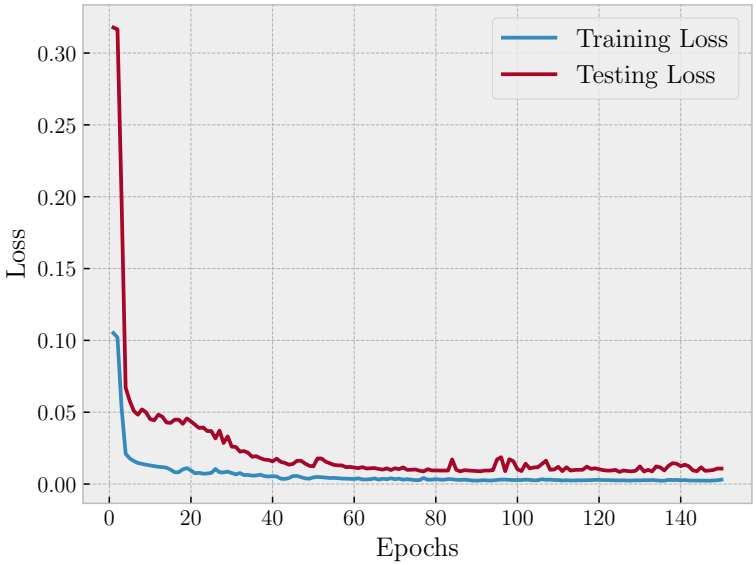

Figure 6.1: The training and testing loss as the number of epochs increases.

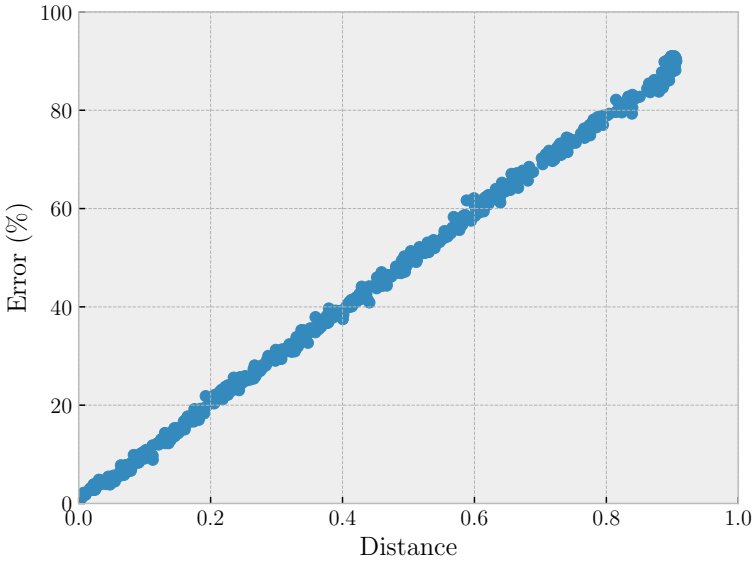

Figure 6.2: Correlation between error (%) in $\boldsymbol{x}$ with respect to $\boldsymbol{x}_t$ and the distance between them in the embedding space under f (distance).

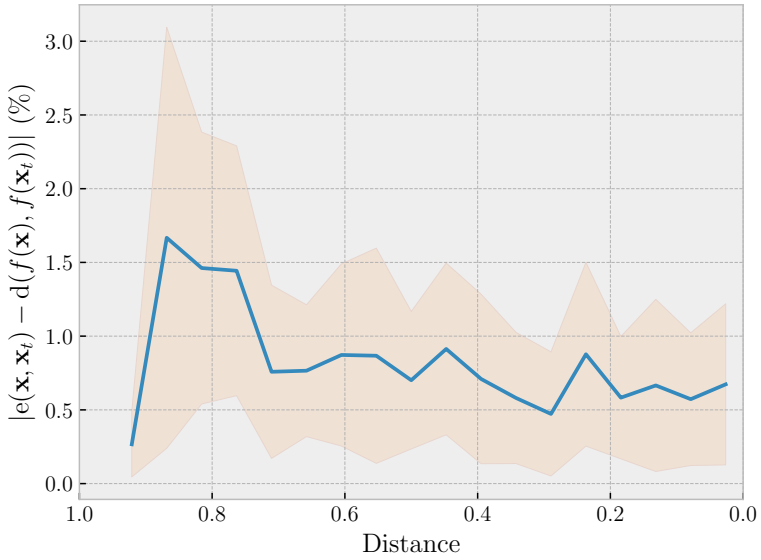

Figure 6.3: The absolute difference ($|\mathrm{e}(\boldsymbol{x}, \boldsymbol{x}_t) - \mathrm{d}(f(\boldsymbol{x}), f(\boldsymbol{x}_t))|$) as the distance $\mathrm{d}(f(\boldsymbol{x}), f(\boldsymbol{x}_t))$ decreases. The shaded area indicates variance.

Figure 6.3 shows the absolute difference between the two distances ($|\mathrm{e}(\boldsymbol{x}, \boldsymbol{x}_t) - \mathrm{d}(f(\boldsymbol{x}), f(\boldsymbol{x}_t))|$), as $\mathrm{d}(f(\boldsymbol{x}), f(\boldsymbol{x}_t))$ decreases. We observe that the mean absolute difference stays relatively low (under 2% in this experiment). This is good news, as an accurate estimation of model improvements for label vectors near the test label vector $\boldsymbol{x}_t$ is precisely what a useful embedding f should support.

6.3 Analysis on Defense Against Brute-force Attacks

In this section, we show that it is difficult for an attacker to launch a brute-force attack on DEL. To find out about the test label vector $\boldsymbol{x}_t \in \mathcal{Q}^m$ that produces $\boldsymbol{y}_t \in \mathbb{R}^n$ for a given $f : \boldsymbol{x} \in \mathcal{Q}^m \rightarrow \boldsymbol{y} \in \mathbb{R}^n$, the attacker's goal is to find $\boldsymbol{x}$ such that $\mathrm{d}(f(\boldsymbol{x}), \boldsymbol{y}_t) < \epsilon$,

for a small ϵ. There are C^m possible instances of $\boldsymbol{x}$ to try, where C is the number of classes. Note C^m can be very large. For example, for a test dataset of 10 classes and 10,000 samples, we have $C = 10$, $m = 10,000$ and $C^m = 10^{10000}$. The attacker may use the following brute-force algorithm:

1: **procedure** BRUTEFORCEATTACK($\boldsymbol{y}_t, \epsilon, q$)
2: Pick a random set $\boldsymbol{\mathcal{X}}$ of q values in $\mathcal{Q}^m$
3: **for** $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$ **do**
4: **if** $\mathrm{d}(f(\boldsymbol{x}), \boldsymbol{y}_t) < \epsilon$ **then**
5: **return** $\boldsymbol{x}$

The success probability (α) of this attack where at least 1 value of $\boldsymbol{x}$ is within the ϵ distance of $\boldsymbol{x}_t$ is

$$\alpha = 1 - (1 - p)^q \approx pq,$$

where p is the probability that $\mathrm{d}(f(\boldsymbol{x}), \boldsymbol{y}_t) < \epsilon$ for a random $\boldsymbol{x}$.

We now derive p. Assume that the outputs of f is uniformly distributed on a unit (n-1)-sphere or equivalently normally distributed on an n-dimension euclidean space [71]. Suppose that $\boldsymbol{y}_t = f(\boldsymbol{x}_t)$. We align the top of the unit (n-1)-sphere at $\boldsymbol{y}_t$. Then, p is equivalent to the probability of a random vector on a (n-1)-hemisphere falling onto the cap [72] which is

$$p = \mathrm{I}(\sin^2 \beta; \frac{n-1}{2}, \frac{1}{2}),$$

where n is the dimension of a vector, β is the angle between $\boldsymbol{x}_t$ and a vector on the sphere, and $\mathrm{I}(x; a, b)$ is the regularized incomplete beta function defined as:

$$\mathrm{I}(x; a, b) = \frac{\mathrm{B}(x;\, a, b)}{\mathrm{B}(a, b)},$$

where $\mathrm{B}(x;\, a, b)$ is the incomplete beta function, and $\mathrm{B}(a, b)$ is the beta function. There functions are defined as:

$$\mathrm{B}(x;\, a, b) = \int_0^x t^{a-1}\,(1-t)^{b-1}\,dt,$$
$$\mathrm{B}(a, b) = \int_0^1 t^{a-1}\,(1-t)^{b-1}\,dt.$$

Figure 6.4 shows the probability p as the distance (ϵ) from $\boldsymbol{x}_t$ decreases for different values of n. We observe that for a small ϵ, this probability is exceedingly low and thus to guarantee attacker's success ($\alpha = 1$), the number of samples ($q = \alpha/p = 1/p$) of $\boldsymbol{x}$ needed to be drawn randomly is very high. For instance, for a 10% error rate, $\epsilon = 0.10$ and $n = 32$, the probability p is 6.12×10^{-13} and the number of trials q needed to succeed is 1.63×10^{12}. In addition, the higher the n, the smaller the p and the larger the q. For example, for a 10% error rate, $\epsilon = 0.10$ and $n = 256$, the probability p is 3.33×10^{-93} and the number of trials q needed to succeed is 3.01×10^{92}.

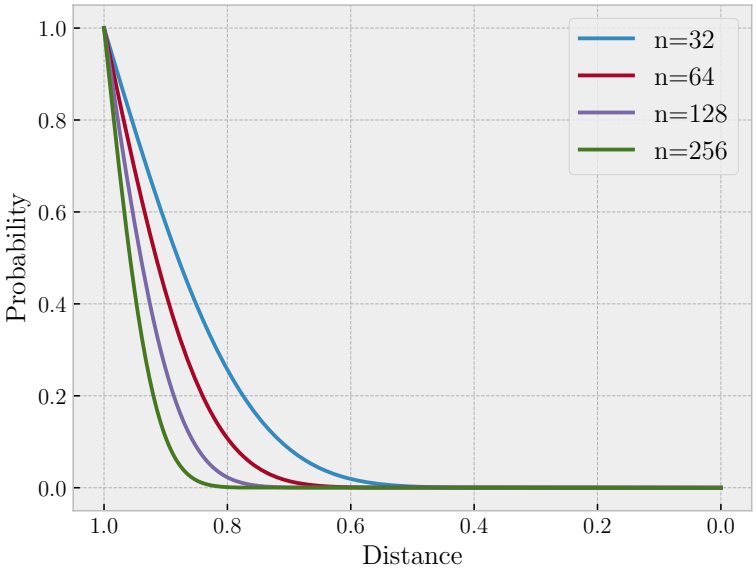

Figure 6.4: The probability p as distance decreases for varying n.

6.4 Analysis on Defense Against Inverse-mapping Attacks

In this section, we provide an analysis on defense against attacks attempting to recover the original test label vector $\boldsymbol{x}_t$ from $\boldsymbol{y}_t = f(\boldsymbol{x}_t)$. We consider the case that the attacker tries to learn an inverse function $f^{-1} : \boldsymbol{y} \in \mathbb{R}^n \rightarrow \boldsymbol{x} \in \mathcal{Q}^m$ using a neural network. Suppose that the attacker uses a multi-layer perceptron (MLP) for this with 1,024 hidden units and a rectified linear unit (ReLU). The network is trained using Adam optimization algorithm [18]. The loss function used is the squared error function:

$$\mathcal{L}_{\boldsymbol{\theta}}(\boldsymbol{y}, \boldsymbol{y}_t) = \|f^{-1}(\boldsymbol{y}) - f^{-1}(\boldsymbol{y}_t)\|^2.$$

We generate the dataset using the two following procedures: GENERATEINVERSE-DATANEARBY and GENERATEINVERSEDATARANDOM. The former has the knowledge that the test label vector $\boldsymbol{x}_t$ is nearby, and the latter does not. We train the neural network to find the inverse function f^{-1} and compare how the neural network learns from these two generated datasets.

1: **procedure** GENERATEINVERSEDATANEARBY($\boldsymbol{x}_t, f$)
2: $\quad \boldsymbol{x} \leftarrow$ GENERATEDATA($\boldsymbol{x}_t$)
3: $\quad \boldsymbol{y} \leftarrow f(\boldsymbol{x})$
4: $\quad$ **return** $\{\boldsymbol{y}, \boldsymbol{x}\}$

The GENERATEINVERSEDATANEARBY procedure generates a perturbation of the test label vector $\boldsymbol{x}_t$, passes it through the function f, and returns a pair of the input $\boldsymbol{y}$ and the target output vector $\boldsymbol{x}$ used to learn the inverse function f^{-1}.

1: **procedure** GENERATEINVERSEDATARANDOM(f, C)
2: $\quad$ Pick a random number $\boldsymbol{x}$ in $\{1, 2, \ldots, C\}^m$
3: $\quad \boldsymbol{y} \leftarrow f(\boldsymbol{x})$
4: $\quad$ **return** $\{\boldsymbol{y}, \boldsymbol{x}\}$

The GENERATEINVERSEDATARANDOM procedure generates a random label vector $\boldsymbol{x}_t$ where each element of the vector has a value representing class 1 to class C, passes it through the function f and returns a pair of the input $\boldsymbol{y}$ and the target output vector $\boldsymbol{x}$ used to learn the inverse function f^{-1}.

Figure 6.5 shows the error $\mathrm{e}(f^{-1}(\boldsymbol{y}_t), \boldsymbol{x}_t)$ as the number of training epochs increases.

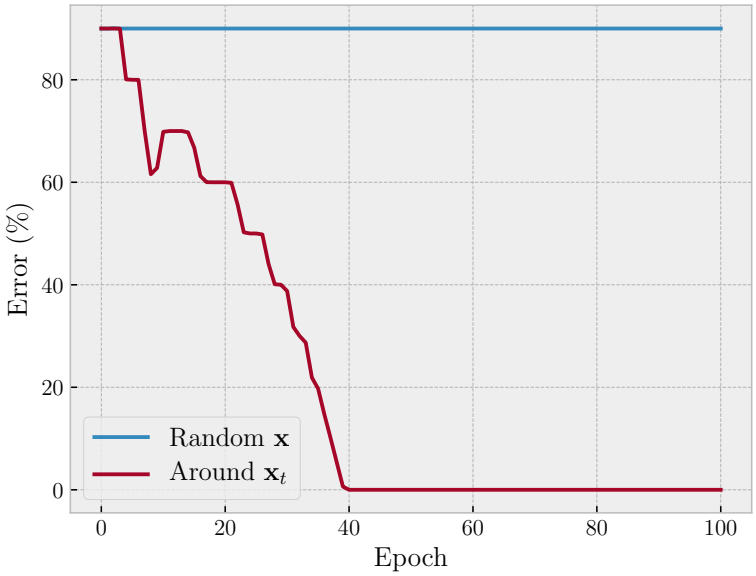

Figure 6.5: Error ($e(f^{-1}(y_t), x_t)$) in percentage as the number of epochs increases for data generated with random x and near x_t.

On one hand, using the data generated without the knowledge of the test label vector x_t using the GENERATEINVERSEDATARANDOM procedure, we see that the network does not reduce the error as it trains. This means that it does not succeed in learning the inverse function f^{-1} and therefore, it will not be able to recover the test label x_t from its output vector y_t. On the other hand, using the data generated with the knowledge of the test label vector x_t using the GENERATEINVERSENEARBY procedure, we see that the network does reduce the the error as it trains and has found the test label vector x_t from its output vector y_t at around 40 epochs. This experiment shows that without the knowledge of x_t, it is hard to find f^{-1}.

6.5 PROOF-OF-IMPROVEMENT

In this section, we introduce the concept of *proof-of-improvement*, the mechanism supporting DaiMoN. The *proof-of-improvement* (PoI) enables a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that $\mathcal{P}$ has found model M that improves the accuracy or reduces the error on the test dataset via the use of DEL without the knowledge of the test label vector $\boldsymbol{x}_t$. PoI is characterized by the PROVE and VERIFY procedures.

As a part of the system setup, a prover $\mathcal{P}$ has a public and private key pair ($\text{pk}_{\mathcal{P}}$, $\text{sk}_{\mathcal{P}}$) and a verifier $\mathcal{V}$ has a public and private key pair ($\text{pk}_{\mathcal{V}}$, $\text{sk}_{\mathcal{V}}$). Both are given our trained DEL function $f(\cdot)$, $\boldsymbol{y}_t = f(\boldsymbol{x}_t)$, the set of m test inputs $\mathcal{Z} = \{\boldsymbol{z}_i\}_{i=1}^{m}$, and the current best distance d_c achieved by submitted models, according to the distance function $\text{d}(\cdot,\cdot)$ described in Section 6.1.

Let $\text{digest}(\cdot)$ be the message digest function such as IPFS hash [73], MD5 [74], SHA [75] and CRC [76]. Let $\{\cdot\}_{\text{sk}}$ denotes a message signed by a secret key sk.

Let M be the classification model for which $\mathcal{P}$ wants to generate a PoI proof $\pi_{\mathcal{P}}$. The model M takes an input and returns the corresponding predicted class label. The PROVE procedure called by a prover $\mathcal{P}$ generates the digest of M and calculates the DEL function output of the predicted labels of the test dataset $\mathcal{Z}$ by M. The results are concatenated to form the body of the proof, which is then signed using the prover's secret key $\text{sk}_{\mathcal{P}}$. The PoI proof $\pi_{\mathcal{P}}$ shows that the prover $\mathcal{P}$ has found a model M that

could reduce the error on a test dataset.

1: **procedure** PROVE(M) $\rightarrow \pi_{\mathcal{P}}$
2: $\quad \boldsymbol{g} \leftarrow \text{digest}(\text{M})$
3: $\quad \boldsymbol{y} \leftarrow f(\text{M}(\mathcal{Z}))$
4: $\quad$ **return** $\{\boldsymbol{g}, \boldsymbol{y}, \text{pk}_{\mathcal{P}}\}_{\text{sk}_{\mathcal{P}}}$

To verify, the verifier $\mathcal{V}$ runs the following procedure to generate the verification proof $\pi_{\mathcal{V}}$, the proof that the verifier $\mathcal{V}$ has verified the PoI proof $\pi_{\mathcal{P}}$ generated by the prover $\mathcal{P}$.

1: **procedure** VERIFY(M, $\pi_{\mathcal{P}}, d_{\text{c}}, \delta) \rightarrow \pi_{\mathcal{V}}$
2: $\quad$ Verify the signature of $\pi_{\mathcal{P}}$ with $\pi_{\mathcal{P}}.\text{pk}_{\mathcal{P}}$
3: $\quad$ Verify the digest: $\pi_{\mathcal{P}}.\boldsymbol{g} = \text{digest}(\text{M})$
4: $\quad$ Verify the DEL function output: $\pi_{\mathcal{P}}.\boldsymbol{y} = f(\text{M}(\mathcal{Z}))$
5: $\quad$ Verify the distance: $\text{d}(\pi_{\mathcal{P}}.\boldsymbol{y}, \boldsymbol{y}_t) < d_{\text{c}} - \delta, \delta \geq 0$
6: $\quad$ **if** all verified **then**
7: $\quad\quad$ **return** $\{\pi_{\mathcal{P}}, d_c, \delta, \text{pk}_{\mathcal{V}}\}_{\text{sk}_{\mathcal{V}}}$

The procedure first verifies the signature of the proof with public key $\pi_{\mathcal{P}}.\text{pk}_{\mathcal{P}}$ of the prover $\mathcal{P}$. Second, it verifies that the digest is correct by computing digest(M) and comparing it with the digest in the proof $\pi_{\mathcal{P}}.\boldsymbol{g}$. Third, it verifies the DEL function output by computing $f(\text{M}(\mathcal{Z}))$ and comparing it with the the DEL function output in the proof $\pi_{\mathcal{P}}.\boldsymbol{y}$. Lastly, it verifies the distance by computing $\text{d}(\pi_{\mathcal{P}}.\boldsymbol{y}, \boldsymbol{y}_t)$ and sees if it is lower than the current best with a margin of $\delta \geq 0$, where δ is an improvement margin commonly agreed upon among peers. If all are verified, the verifier generates the body of the verification proof by concatenating the PoI proof $\pi_{\mathcal{P}}$ with the current

best distance d_c and δ. Then, the body is signed with the verifier's secret key $\mathrm{sk}_{\mathcal{V}}$, and the verification proof is returned.

6.6 The DaiMoN System

In this section, we describe the DaiMoN system that incentivizes participants to improve the accuracy of models solving a particular problem. In DaiMoN, each classification problem has its own DaiMoN blockchain with its own token. An append-only ledger maintains the log of improvements for that particular problem. A problem defines inputs and outputs which machine learning models will solve for. We call this the problem definition. For example, a classification problem on the FashionMNIST dataset [68] may define an input $\boldsymbol{z}$ as a 1-channel $1 \times 28 \times 28$ pixel input whose values are ranging from 0 to 1, and an output $\boldsymbol{x}$ as 10-class label ranging from 1 to 10:

$$\{\boldsymbol{z} \in \mathbb{R}^{1\times 28\times 28} \mid 0 \leq \boldsymbol{z} \leq 1\},$$

$$\{\boldsymbol{x} \in \mathbb{Z} \mid 1 \leq \boldsymbol{x} \leq 10\}.$$

Each problem is characterized by a set of test dataset tuples. Each tuple $(\mathcal{Z}, f, \boldsymbol{y}_t)$ consists of the test inputs $\mathcal{Z} = \{\boldsymbol{z}\}_{i=1}^{m}$, the DEL function f, and the DEL function output $\boldsymbol{y}_t = f(\boldsymbol{x}_t)$ on the test label vector $\boldsymbol{x}_t$.

A participant is identified by its public key pk with the associated private key sk.

There are six different roles in DaiMoN: problem contributors, model improvers, validators, block committers, model runners, and model users. A participant can be one or more of these roles. We now detail each role below:

- **Problem contributors** contribute test dataset tuples to a problem. They can create a problem by submitting a problem definition and the first test dataset tuple.
- **Model improvers** compete to improve the accuracy of the model according to the problem definition defined in the chain. A model improver generates a PoI proof for the improved model and submit it.
- **Validators** validate PoI proofs, generate a verification proof and submit it as a vote on the PoI proofs. Beyond being a verifier for verifying PoI, a validator submits the proof as a vote on the blockchain.
- **Block committers** create a block from the highest voted PoI proof and its associated verification proofs and commit the block to the chain.
- **Model runners** run the inference on the latest model given inputs and return outputs and get paid in tokens.
- **Model users** request an inference computation from model runners with an input and pay for the computation in tokens.

6.6.1 THE CHAIN

Each chain consists of two types of blocks:

- **A Problem block** contains the following information, including but not limited to: the block number, the hash of the parent block, the problem definition, the test dataset tuples, and the block hash.
- **An Improvement block** contains the following information, including but not limited to: the block number, the hash of the parent block, PoI proof $\pi_{\mathcal{P}}$ from model improver $\mathcal{P}$, verification proofs $\{\pi_{\mathcal{P}}\}$ from validators $\{\mathcal{V}\}$, and the block hash. The chain must start with a Problem block that define the problem, followed by Improvement blocks that record the improvements made for the problem.

6.6.2 THE CONSENSUS

After a DaiMoN blockchain is created, there is a problem definition period T_p. In this period, any participant is allowed to add test dataset tuples into the mix. After a time period T_p has passed, a block committer commits a Problem block containing all test dataset tuples submitted within the period to the chain.

After the Problem block is committed, a competition period T_b begins. During this period, a model improver can submit the PoI proof of his/her model. A validator then

validates the PoI proof and submit a verification proof as a vote. For each PoI proof, its associated number of unique verification proofs are tracked. At the end of each competition period, a block committer commits an Improvement block containing the model with the highest number of unique verification proofs, and the next competition period begins.

6.6.3 THE REWARD

Each committed block rewards tokens to the model improver and validators. The following reward function is used:

$$\mathrm{R}(d, d_c) = \mathrm{I}(1 - d; a, \frac{1}{2}) - \mathrm{I}(1 - d_c; a, \frac{1}{2}),$$

where $\mathrm{I}(x; a, b)$ is the regularized incomplete beta function, d is the distance of the block, d_c is the current best distance so far, and a is a parameter to allow for the adjustment to the shape of the reward function. Figure 6.6 shows the reward function as the distance d decreases for different current best distance d_c for $a = 3$. We see that more and more tokens are rewarded as the distance d reaches 0, and the improvement gap $d_c - d$ increases.

Each validator is given a position as it submits the validation proof: the s-th validator to submit the validation proof is given the s-th position. The validator's reward is the

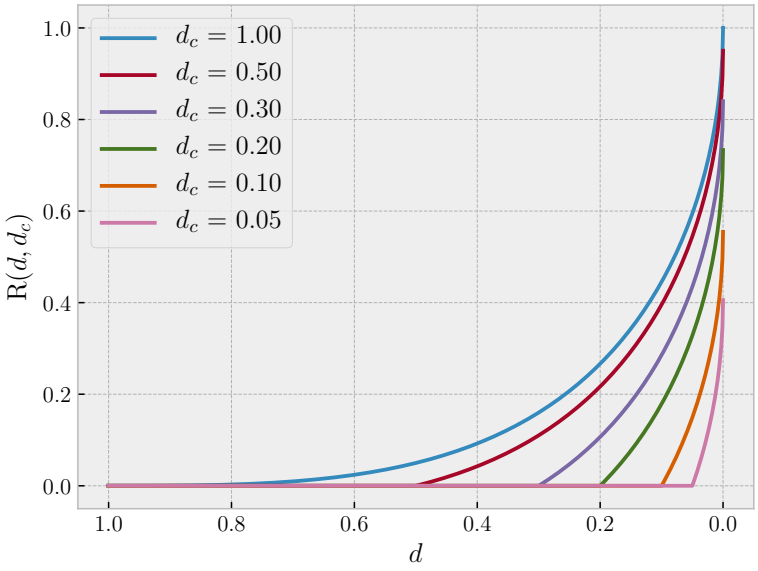

Figure 6.6: The reward function $\mathrm{R}(d, d_c)$ as the distance d decreases for varying current best distance d_c for $a = 3$.

model improver's reward scaled by 2^{-s}:

$$\mathrm{R}(d, d_c)2^{-s},$$

where $s \in \mathbb{Z}_{>0}$ is the validator's position, and $\mathbb{Z}_{>0}$ denotes the set of integers greater than zero. This factor encourages validators to compete to be the first one to submit the validation proof for the PoI proof in order to maximize the reward. Two is used as a base of the scaling factor here since $\sum_{s=1}^{\infty} 2^{-s} = 1$.

6.6.4 THE MARKET

In order to increase the value of the token of each problem, there should be demand for the token. One way to generate demand for the token is to allow it to be used as a payment for inference computation based on the latest model committed to the chain. To this end, model runners host the inference computation. Each inference call requested by users is paid for by the token of the problem chain that the model solves. Model runners automatically upgrade the model, as better ones are committed to the chain. The price of each call is set by the market according to the demand and supply of each service. This essentially determines the value of the token, which can later be exchanged with other cryptocurrencies or tokens on the exchanges. As the demand for the service increases, so will the token value of the problem chain.

Model runners periodically publish their latest services containing the price for the inference computation of a particular model. Once a service is selected, model users send a request with the payment according to the price specified. Model runners then verify the request from the user, run the computation, and return the result.

To keep a healthy ecosystem among peers, a reputation system may be used to recognize good model runners and users, and reprimand bad model runners and users. Participants in the network can upvote good model runners and users and downvote bad model runners and users.

6.6.5 System Implementation

DaiMoN is implemented on top of the Ethereum blockchain [61]. In this way, we can utilize the security and decentralization of the main Ethereum network. The ERC-20 [77] token standard is used to create a token for each problem chain. Tokens are used as an incentive mechanism and can be exchanged. Smart contracts are used to manage the DaiMoN blockchain for each problem.

Identity of a participant is represented by its Ethereum address. Every account on Ethereum is defined by a pair of keys, a private key and public key. Accounts are indexed by their address, which is the last 20 bytes of the Keccak [75] hash of the public key.

The InterPlanetary File System (IPFS) [73] is used to store and share data files that are too big to store on the Ethereum blockchain. Files such as test input files and model files are stored on IPFS and only their associated IPFS hashes are stored in the smart contracts. Those IPFS hashes are then used by participants to refer to the files and download them.

6.7 Discussion

One may compare a DEL function to an encoder of an autoencoder. An autoencoder consists of an encoder and a decoder. The encoder maps an input to a lower-dimensional embedding which is then used by the decoder to reconstruct the original input. Although a DEL function also reduces the dimensionality of the input label vector, it does not require the embedding to reconstruct the original input and it adds the constraint that the output of the function should preserve the error or the distance of the input label vector to a specific test label vector x_t. In fact, for our purpose of hiding the test labels, we do not want the embedding to reconstruct the original input test labels. Adding the constraint to prevent the reconstruction may help further defense against the inverse-mapping attacks and can be explored in future work.

By building DaiMoN on top of Ethereum, we inherit the security and decentralization of the main Ethereum network as well as the limitations thereof. We now discuss the security of each individual DaiMon blockchain. An attack to consider is the Sybil attack on the chain, in which an attacker tries to create multiple identities (accounts) and submit multiple verification proofs on an invalid PoI proof. Since each problem chain is managed using Ethereum smart contracts, there is an inherent gas cost associated with every block submission. Therefore, it may be costly for an attacker to overrun the votes of other validators. The more number of validators for that chain, the

higher the cost is. In addition, this can be thwarted by increasing the cost of each submission by requiring validators to also pay Ether as they make the submission. All in all, if the public detects signs of such behavior, they can abandon the chain altogether. If there is not enough demand in the token, the value of the tokens will depreciate and the attacker will have less incentives to attack.

Since we use IPFS in the implementation, we are also limited by the limitations of IPFS: files stored on IPFS are not guaranteed to be persistent. In this case, problem contributors and model improvers need to make sure that their test input files and model files are available to be downloaded on IPFS. In addition to IPFS, other decentralized file storage systems that support persistent storage at a cost such as Filecoin [78] and Storj [79] can be used.

6.8 Related Works

One area of related work is on data-independent locality sensitive hashing (LSH) [80] and data-dependent locality preserving hashing (LPH) [81, 82]. LSH hashes input vectors so that similar vectors have the same hash value with high probability. There are many algorithms in the family of LSH. One of the most common LSH methods is the random projection method called SimHash [83], which uses a random hyperplane to hash input vectors.

Locality preserving hashing (LPH) hashes input vectors so that the relative distance between the input vectors is preserved in the relative distance between of the output vectors; input vectors that are closer to each other will produce output vectors that are closer to each other in the output space. The DEL function presented in this chapter is in the family of LPH functions. While most of the work on LSH and LPH focuses on dimensionality reduction for nearest neighbor searches, DEL is novel that it focuses on learning an embedding function that preserves the distance to the test label vector of the test dataset. For the purpose of hiding the test label vector from verifier peers, it is appropriate that DEL's distance preserving is specific to this test vector. By being specific, finding the DEL function becomes easier.

Another area of related work is on blockchain and AI. There are numerous projects covering this area in recent years. These include projects such as SingularityNET [84], Effect.ai [85] and Numerai [86]. SingularityNET and Effect.ai are decentralized AI marketplace platforms where anyone can provide AI services for use by the network, and receive network tokens in exchange. This is related to the DaiMoN market, where model runners run inference computation for model users in exchange for tokens. Numerai has an auction mechanism where participants stake network tokens to express confidence in their models' performance on forthcoming new data. The mechanism will reward participants according to the performance of their models on unspecified yet future test data, their confidence, and their stake. While these plat-

forms all use a single token, DaiMoN has a separate token for each problem. It also introduces the novel ideas of DEL and PoI, allowing the network to fairly reward participants that can prove that they have a model that improves the accuracy for a given problem.

6.9 Relevance within IDN

For IDN, DaiMoN will help to increase the quality and the number of available DNN models that peers can run on IDN. This availability of high-quality DNN models is one of the important factors of a successful intelligence distribution network. The higher the quality of models peers are servicing, the more the number of peers will participate in the network. This in turn will help with the efficiency and performance of serving inference with the network, as there will be more peers who are close to each other available to perform the computations.

6.10 Conclusion

We have introduced DaiMoN, a decentralized artificial intelligence model network. DaiMoN uses a Distance Embedding for Labels (DEL) function. DEL embeds the label vector inferred by a classifier in a low-dimensional space where its error or its distance to the test label vector of the test dataset is approximately preserved. DEL hides

test labels from peers while allowing them to assess the improvement that a model makes. We present how to learn DEL, evaluate its effectiveness, and present the analysis of DEL's resilience against attacks. This analysis shows that it is hard to launch a brute-force attack or an inverse-mapping attack on DEL without knowing a priori a good estimate on the location of the test label vector, and that the hardness can be increased rapidly by increasing the dimension of the embedding space.

DEL forms a basis of *proof-of-improvement* (PoI), which is the core of DaiMoN. Participants use PoI to prove that they have found a model that improves the accuracy of a particular problem. This allows the network to keep an append-only log of model improvements and reward the participants accordingly. DaiMoN uses a reward function that scales according to the increase in accuracy a new model has achieved on a particular problem. We hope that DaiMoN will spur collaboration in improving machine learning models.

In this chapter, we have discussed a decentralized artificial intelligence model network called DaiMoN that incentivizes peers to collaborate, improve the quality of DNN models, and share them. DaiMoN helps with the quality and the availability of DNN models. With the abundance of high-quality DNN models, in the next chapter we will discuss the design of IDN with a mechanism to incentivize people to participate and share resources to run inference on these DNN models.

7

Intelligence Distribution Network

Design and Specification

In this chapter, we discuss the design and detail specification of IDN, how it is used by peers, and how peers are incentivized to participate in the network. We will use

the following terminology: IDN nodes or peers refer to devices that run IDN client software. Models refer to model files, including neural network computation graphs and their corresponding weights. Services refer to an inference service defined by the input(s), the output(s) and optionally the specific DNN model(s) that the service provides.

IDN draws a lot of design influence from the InterPlanetary File System (IPFS) [73]. A core component of IDN is a compute exchange protocol called ComputeSwap, which governs efficient inference distribution. The ComputeSwap protocol encourages people to share their spare computing resources and discourages people from cheating.

IDN is divided into multiple components responsible for different functionalities:

1. Identity - manages node identity generation and verification.
2. Network - manages connections to other peers.
3. Routing - maintains information to locate specific peers and services.
4. Inference Service Engine (ISE) - manages inference services.
5. ComputeSwap - manages the exchange of inference computation and incentivizes efficient service distribution.

We now describe each of these components.

7.1 IDENTITY

A node that joins IDN has a unique identity (ID). A node ID is a hash of its public key. A node ID is generated as follows:

```
1: procedure GENERATENODEID(difficulty)
2:     do
3:         pubKey, priKey ← generateKeyPair()
4:         nodeID ← hash(pubKey)
5:         d ← countPrecedingZeroBits(nodeID)
6:     while d < difficulty
7:     return {nodeID, pubKey, priKey}
```

The procedure takes a *difficulty* argument. It tries to generate the public and private key pair where the hash of the public key satisfies a certain difficulty constraint. This difficulty constraint is defined as the number of preceding zero bits of the hash of the public key. *countPrecedingZeroBits* is a function that counts the number of preceding zero bits of the hash of the public key. *generateKeyPair* is a function that generates the public and private key pair. One possible algorithm to generate this key pair is RSA [87]. *hash* is a hash function. Algorithms such as SHA3/Keccak [75] could be used as a hash function.

A node stores its public and encrypted private keys, along with network statistics such as debt and credit that are associated with the node. A user is free to create a new identity, however he/she will lose the network statistics associated with the

node and the benefits that comes with the accrual of credits given to other peers.

NodeID is represented as Multihash [88]. Multihash is a self-describing hash where the hash is prepended with the codec of the hash and the length of the hash digest. For example, we can describe a SHA1 [89] hash of a string "content" as

SHA1 - 7 bytes - SHA1("content"),

which is 0x1107636f6e74656e74 in hex. 0x11 is the codec for SHA1, 0x07 is the length of the hash digest and 0x636f6e74656e74 is the SHA1 hash of "content."

A node contains its node ID and the corresponding public and private key pair. The node data structure is defined as follows:

```
type Multihash []byte
type PublicKey []byte
type PrivateKey []byte
type NodeID Multihash

type Node struct {
    nodeID NodeID
    pubKey PublicKey
    priKey PrivateKey
}
```

We define NodeID type as Multihash type. Multihash, PublicKey, and PrivateKey types are arrays of bytes. Node type consists of nodeID of type NodeID, pubKey of type PublicKey, and priKey of type PrivateKey.

To verify each other's identities, upon connecting to the other node, a node checks

whether the other node's ID matches the hash of its public key.

7.2 NETWORK

IDN peers communicate regularly with other peers in the IDN network. *libp2p* [90] is used to implement the network layer. It supports multiple transport protocols including UDP [91], TCP [92] and WebRTC [93].

Each node has an address in Multiaddr [94] format. Multiaddr is a self-described address. A Multiaddr itself contains transport protocols and network addresses. For example, a TCP connection over IPv4 [95] can be expressed in Multiaddr format as

/ip4/10.20.30.40/tcp/8080/.

Multiaddr also supports encapsulation. For example, a TCP/IPv4 connection proxied over a TCP/IPv4 connection can be expressed in Multiaddr format as

/ip4/5.6.7.8/tcp/8080/ip4/1.2.3.4/sctp/4756/.

With Multiaddr address, a node knows how to connect to the other node via its Multiaddr. For example, /ip4/10.20.30.40/tcp/8080/ describes a connection by IPv4 at an address 10.20.30.40 using the TCP protocol on port 8080.

Multiaddr allows the flexibility of addressing peers without any assumption on the underlying network (e.g., IP address) or transport layers (e.g., TCP). This allows IDN to be used in any type of network, including overlay networks.

7.3 ROUTING

IDN peers need to look up other peers' network addresses by their node IDs and look up peers who provide a particular service by the service ID. This is achieved by using a distributed hash table (DHT) [96, 97].

The interface for routing is derived from *libp2p* [90] and is defined as follows.

```
type Routing interface{
    findPeer(nodeID NodeID) []Multiaddr
    provide(key Multihash)
    findProviders(key Multihash, numPeers int) []NodeID

    getValue(key Multihash) []byte
    setValue(key Multihash, value []byte)
}
```

findPeer(*nodeID*) looks up a particular peer's network address by its node ID *nodeID*. The function returns an array of a peer's network addresses of type Multiaddr.

provide(*key*) broadcasts to other peers that this node is servicing a service with service ID *key*.

findProviders(*key, numPeers*) finds *numPeers* number of peers that provide services

with service ID *key*. This is used for a node to look up peers that provide a particular inference service that it needs. The function returns an array of node IDs.

getValue(*key*) gets *value* stored in DHT at *key*. It is used to retrieve small meta data about a particular key. The function returns *value* as an array of bytes.

setValue(*key, value*) stores *value* in DHT at *key*. It is used to store small meta data about a particular key.

7.4 Inference Service Engine (ISE)

Each service on IDN has a service ID. A service ID is an IPFS hash of a service descriptor specified in JavaScript Object Notation (JSON) format. The service descriptor contains information such as the name of the service, the description of the service, the type of inputs, the type of outputs, the link to download the model, and the model hash for integrity verification. The service ID is used as an identifier of the service. It is used as a key in the DHT to store and retrieve peers that provide the service (See Section 7.3).

A node decides what to service based on the compatibility of the device and the demands from its peers. A priority queue ranked by a service's priority score is main-

tained. The service priority score s is defined as follows:

$$s = \sum_{i=1}^{N} r_i,$$

where N is the total number of peers requesting the service $\mathcal{S}$, and r_i is the debt-to-credit ratio which is the ratio of debt the node owes for its peer i over the credit the node gives to its peer i (See Equation 7.1 for the exact definition).

A node services the top k services according to the priority score s in the priority queue. k is set by the node's operator in accordance with the capacity and capability of the device in terms of the number of CPUs, the number of GPUs, the amount of memory, etc.

When a service is deployed, there is a minimum service period T_s. The service will be run for the minimum service period before it can be swapped out. This is to prevent oscillation between services that are in the top k and ones that are not, and to minimize the time spent to bootstrap and tear down services when the services are swapped.

When the priority score updates and the top k services change, a node tear downs the services that are no longer in the top k (provided that its minimum service period T_s has passed) and bootstraps the services that are currently in the top k.

7.5 COMPUTESWAP

In IDN, peers exchange inference computation with other peers using a protocol called ComputeSwap. ComputeSwap is the core component of IDN. It governs efficient distribution of inference services. ComputeSwap is an adaptation of a BitSwap protocol in IPFS [73].

Like BitSwap, ComputeSwap operates as a persistent marketplace where peers barter and exchange inference computation with each other. Unlike BitSwap, ComputeSwap exchanges inference computation instead of blocks of data. ComputeSwap peers are looking to run inference computation on DNN model(s) given input data, and have a list of services that they could run inference computation to offer in exchange. ComputeSwap peers provide direct value to each other in the form of inference computation.

ComputeSwap's design goals are to:

1. Encourage peers to service inference computation when they do not need anything in particular.
2. Discourage free-loading peers that never service inference computation.

To accomplish these goals, ComputeSwap has a credit-like system where the probability that a node gets its request for inference computation serviced falls as its debt

increases. ComputeSwap peers run inference computation optimistically on credit, hoping that the debt will be repaid. This encourages peers to service inference computation when they do not need anything in particular, as other peers may be able to service inference computation it needs in exchange in the future. Since the the probability of servicing inference computation falls as debt increases, this discourages free-loading peers that never service inference computation, as their debts will be high and the probability of getting their inference computation requests serviced will be low.

ComputeSwap incentivizes a node to service an inference computation needed by its peers. If a node is currently not servicing inference computation its peers want, it will switch to service the inference computation its peers want in order to keep its debt on its peers low and therefore the probability of getting its own requests serviced on its peers high. This encourages the efficient distribution of inference services because more popular services in a particular area will be serviced by more peers.

We now introduce the details of ComputeSwap, including the ComputeSwap ledger and the ComputeSwap strategy; then, we give a detailed specification of ComputeSwap. The ComputeSwap ledger keeps track of the computation exchanged with other peers in terms of debt and credit. Any pair of peers maintain a common ledger. This allows peers to keep track of history and avoid tampering. The ComputeSwap strategy outlines a strategy each node uses to exchange inference computation.

7.5.1 COMPUTESWAP LEDGER

A node keeps ledgers to keep track of credits and debts of the inference computation exchanged and avoid tampering. Each pair of two connecting peers maintains a common ledger. A node keeps multiple ledgers, one for each peer. These ledgers allow nodes to keep track of a history of credits and debts and avoid tampering. A common ledger has to be agreed between two connecting peers.

The structure of a ComputeSwap ledger is defined as follows.

```
type Ledger struct {
    node1 NodeID
    node2 NodeID
    debt1 int
    debt2 int
}
```

Given a pair of connecting peers, their node IDs are sorted and stored in a common ledger. The first node ID is assigned to *node1*, and the second node ID is assigned to *node2*. *debt1* is the debt accumulated by *node1*, and *debt2* is the debt accumulated by *node2*. Note that *debt1* is the credit that *node2* gives to *node1* and *debt2* is the credit that *node1* gives to *node2*. These are used to compute the debt-to-credit ratio in Section 7.5.2.

The ledger is exchanged when a connection is activated between two connecting peers. If the ledger does not yet exist, it is initialized with zero debts. If a peer notices

a discrepancy in the ledger, it is free to hold the other node in contempt and refuse to exchange.

Peers may agree to create check points of their ledgers, thus allowing them to garbage collect the old history.

7.5.2 COMPUTESWAP STRATEGY

Each ComputeSwap peer uses a strategy to exchange its computation. Each peer may employ different strategies. The strategies used by peers dictate the performance of the network as a whole. In designing a strategy, we aim to:

1. Be lenient to trusted peers.
2. Maximize the efficiency and performance of the network.
3. Deter freeloaders from exploiting and degrading the network, and be resistant to other unknown strategies.

In this section, we outline a strategy that fits these criteria. To be lenient to trusted peers, peers should maintain a trusted peer list where the service probability is always one for those peers on the list. To maximize the efficiency and performance of the network, peers should serve service requests optimistically as the requests come in. To deter freeloaders and be resistant to other unknown strategies, peers should service the inference requests probabilistically, where the probability decreases as the

debt-to-credit ratio, or debt ratio for short, increases.

The debt-to-credit ratio or debt ratio (r) is defined for each pair of peers as:

$$r = \frac{\text{debt}}{\text{credit} + 1}, \tag{7.1}$$

where *debt* is the amount of inference computation a peer has requested for the other peer and *credit* is the amount of inference computation the peer has serviced for the other peer. The inference computation is counted as the number of operations the node has to do in order to service the request. For a floating point model, this is the total number of floating point operations. Note that one is added to the denominator to keep r well-defined when *credit* is zero.

For the service probability, we want to design a probability function that penalizes peers with a high debt ratio (r). The penalty must grow as the debt ratio increases to discourage peers to accumulate debt. When the debt ratio is less than one, i.e., the node has serviced requests more than requesting services, there should be no penalty. With these criteria in mind, the probability p of servicing a request is defined to be monotonic, starting from one and dropping to zero as r increases. Specifically, it is defined as follows:

$$p(r) = 1 - \mathrm{I}(\frac{r-1}{m-1}; n, \frac{1}{n}),$$

where m is a parameter indicating the upper bound of the debt ratio where the proba-

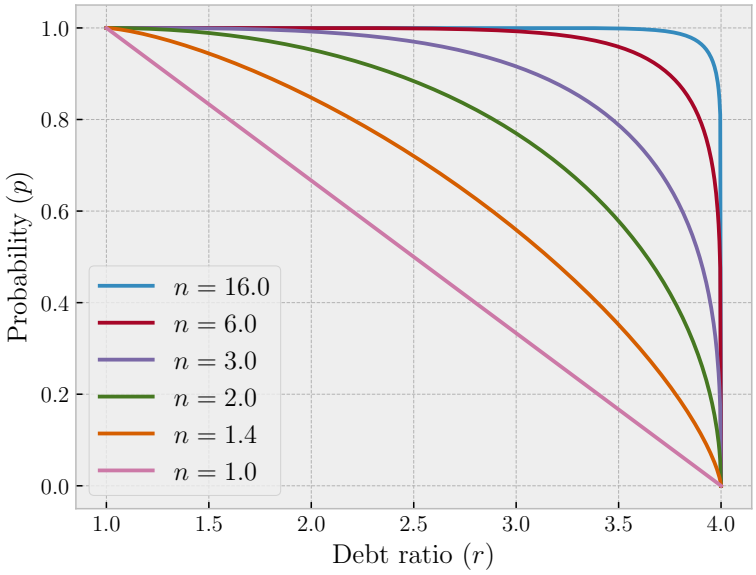

Figure 7.1: Probability of servicing a request as the debt ratio increases when $m = 4.0$ for varying n

bility is zero, n is a parameter indicating how p drops with r, and $\mathrm{I}(x; a, b)$ is the regularized incomplete beta function defined as:

$$\mathrm{I}(x; a, b) = \frac{\mathrm{B}(x;\, a, b)}{\mathrm{B}(a, b)},$$

where $\mathrm{B}(x;\, a, b)$ is the incomplete beta function, and $\mathrm{B}(a, b)$ is the beta function. Their functions are defined as:

$$\mathrm{B}(x;\, a, b) = \int_0^x t^{a-1}\,(1-t)^{b-1}\,dt,$$
$$\mathrm{B}(a, b) = \int_0^1 t^{a-1}\,(1-t)^{b-1}\,dt.$$

Figure 7.1 shows the probability p of servicing a request as the debt ratio r increases when $m = 4.0$ for varying n. We see that this definition of the probability p fits the criteria specified earlier: the probability drops as the debt ratio increases. How p drops depends on the configurable parameter n. As n increases, the function is more lenient to peers with high debt ratio r. For example, when $n = 16$ the penalty does not kick in until the debt ratio is very high; after a certain point, the probability drops significantly. When $n = 1$, the probability p drops linearly as the debt ratio r increases.

If a node decides not to service a peer, the node subsequently ignores the peer for a certain period of time T_g. This prevents the peers from trying to increase their success probability by increasing the number of requests. A node can also decide to add a peer to a blacklist and immediately terminate the connection if it detects undesirable behaviors from the peer.

7.5.3 COMPUTESWAP SPECIFICATION

The ComputeSwap data structure and interface are defined as follows:

```
type NeedList []Multihash
type HaveList []Multihash
type TrustedList []Multihash
type BlackList []Multihash
type State int
type ServiceID Multihash

type ComputeSwap struct {
```

```
    ledgers map[NodeID]Ledger
    actives map[NodeID]Peer
    needList NeedList
    haveList HaveList
    trustedList TrustedList
    blackList BlackList
}

type Peer struct {
    nodeID NodeID
    addrs []Multiaddr
    state State
    lastSeen Time
    ledger Ledger

    needList NeedList
    haveList HaveList
}

type Request struct {
    serviceID ServiceID
    inputs []byte
}

type Response struct {
    outputs []byte
}

type Peer interface {
    open(ledger Ledger) bool
    sendNeedList(needList NeedList)
    sendHaveList(haveList HaveList)
    infer(request Request) Response
    close(final bool)
}
```

A ComputeSwap node maintains a map of peer ID to active peers and a map of peer ID to associated ledgers. It also maintains various lists: a trusted peer list, a black-list, a need list and a have list. A trusted peer list is a list of peers for which the node will always service inference computation, regardless of the debt ratio the peer has. A

blacklist is a list of peers for which the node will never exchange inference computation. A need list is a list of services that the node needs from its peers. A have list is a list of services that it is currently servicing. When an application starts, joins IDN, and requests an inference computation service on a particular node, the service ID is added to the need list. When a node starts providing a particular service, the service ID is added to the have list.

For each connecting peer, a node also maintains the peer's need list and have list as well as other information such as the peer's ID, network addresses, state, last seen timestamp, and ledger. Each peer object also has the following available methods:

- **open(*ledger*)** opens a connection with the peer and sends the common ledger *ledger*. The method returns *true* if the peer accepts the ledger and activates the connection. Otherwise, it returns *false* and closes the connection.
- **sendNeedList(*needList*)** sends the list of services needed (*needList*) to the peer. The method returns *true* if the peer successfully receives the list. Otherwise, it returns *false*.
- **sendHaveList(*haveList*)** sends the list of services this node is providing (*haveList*) to the peer. The method returns *true* if the peer successfully receives the list. Otherwise, it returns *false*.
- **infer(*request, timeout*)** sends inference request of type Request to the peer and

waits for *timeout* period for a response from the peer. The peer executes the inference computation and returns the response of type Response. If the *timeout* period has passed and the peer has not yet responded, the method returns without a response.

- **close(*final*)** closes the connection with the *final* parameter indicating whether it is the final message and the node is shutting down the connection.

7.5.3.1 States and Lifecycle

The following are the different states of a peer during its lifetime:

1. Open: peers negotiate their ledgers until they agree.
2. Online: a peer is online and ready to exchange lists and inference computation.
3. Close: the connection to the peer is deactivated.
4. Ignore: a peer is ignored for an ignore period T_g.

Figure 7.2 shows different states of a peer during its lifecycle. We now describe each state in detail.

7.5.3.2 Open

When a connection is established, a peer is in an Open state. A node then sends an *open* message with the common ledger, either stored from the previous connection or

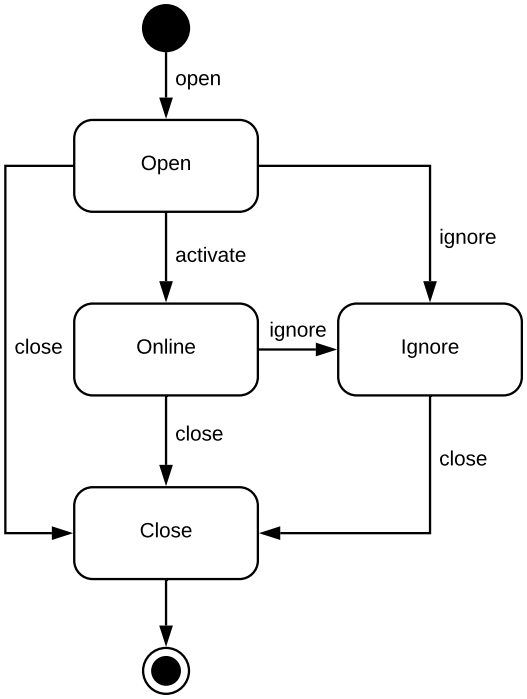

Figure 7.2: ComputeSwap Peer State Machine. It consists of four states: Open, Online, Ignore and Close

a new one if it does not exist, to the peer.

Upon receiving an *open* message, a peer chooses whether to activate the connection or not depending on whether the peer is in the trusted peer list, the blacklist, or neither. A node may choose to ignore the *open* message if it detects undesirable behaviors from the peer. This will transition the state of the peer to Ignore state and start an ignore period T_g, where the node will ignore this peer for a T_g period of time. A node may also choose to add the peer to a blacklist and proceed to close the connection.

If a node decides to activate the connection, it initializes a Peer object with the local version of the ledger or a new one if it has not yet existed. The node then sets the last seen timestamp and compares the received ledger with its own. If they match exactly, the connection is open and the peer's state is transitioned to Online. Otherwise, the peer creates a new ledger, and sends it back to the sender to open the connection.

7.5.3.3 ONLINE

When a peer is in the Online state, a node sends out its need list and have list to the peer under the following conditions:

1. the connection is first activated,
2. after a randomized periodic timeout,
3. after a change in node's need list and have list.

Upon receiving a need list or have list, a node stores it. It checks whether it can service any of the services in the need list and has capacity for it. This decision depends on the state of the service priority queue. If the node can provide the service, it bootstraps the service and waits for the inference requests.

Upon receiving an incoming inference request, the node simply executes the inference computation given the input data. Upon receiving the response, the node option-

ally verifies the response and returns the receipt confirmation.

A node can occasionally send a challenge where it knows the expected response. Upon receiving the response, the node verifies if the response of the challenge is correct and returns receipt confirmation. If the challenge is not responded to correctly, a node is free to ignore the node for a certain period of time or blacklist the node and close the connection.

Upon receiving the receipt confirmation, both the receiver and sender update their ledgers to reflect the new debt ratio.

7.5.3.4 CLOSE

A node can send a *close* message to a peer to indicate that it wants to close the connection. A *close* message has a *final* parameter to indicate the final message and the node's intention to tear down the connection.

A *close* message with the *final* parameter set to *false* is sent when the node does not receive any messages from the peer and has waited for a certain period of time T_c. A *close* message with the *final* parameter set to *true* is sent when the node is shutting down.

A peer transitions to a Close state when a node sends a *close* message to the peer to tear down the connection. After a *close* message is sent, the node tears down the con-

nection to the peer, clearing any state stored except the ledger which is stored for future accountability.

Upon receiving a *close* message, if the *final* parameter is set to *false*, a node may reopen the connection immediately. Otherwise, it tears down the connection to the peer, clearing any state stored except the ledger which is stored for future accountability.

7.5.3.5 IGNORE

When a peer is transitioned into the Ignore state, a node ignores all messages from that peer. The peer is transitioned out of the Ignore state after the ignore period T_g has passed.

7.6 RELATED WORKS

In 1999, Napster [79], a pioneering peer-to-peer (p2p) MP3 [98] file sharing application, was born out of a college dormitory. It has made a tremendous impact on how the Internet was used in the following decades. In 2000, Clay Shirkey [99] defines peer-to-peer (p2p) as "a class of applications that take advantage of the resources – storage, cycles, content, human presence – available at the edges of the Internet." That definition remains true today. A peer in a p2p network can be both a resource

provider and a resource consumer. As more nodes join the network, the demand on and the resources of the system increase. In addition, since each peer in the network is a provider, the servicing resources of the network is distributed with redundancy. Thus, inherently, applications on a p2p network are low-cost, low-latency, fault tolerant, and scalable.

Since Napster, many other p2p systems have emerged. For example, SETI@Home [100] and Folding@Home [101] provide infrastructure for communities to volunteer their CPU cycles to help solve certain problems. Gnutella [102] and BitTorrent [103] are p2p file sharing systems. CoralCDN [97] is a p2p content distribution network (CDN). More recently, Storj [79] is a p2p cloud storage network and IPFS [73] is distributed file system. The successes of these p2p applications share a common trait: these p2p applications greatly reduce the cost of running the source server, which in turn allows for the growth of their usage. Built as a p2p network, IDN shares this common trait. IDN peers can economically run inference on DNN models in their applications using spare computing resources nearby. The growth of participating peers will increase the resources available in the network and in turn accelerate the growth of artificial intelligence applications.

7.7 Conclusion

In this chapter, we describe different components of IDN, including Identity, Network, Routing, Inference Service Engine (ISE), and ComputeSwap. The core component of IDN is ComputeSwap that manages the efficient distribution of inference computation among peers. In the strategy presented, a node services inference computation to a peer optimistically hoping that the debt will be repaid and probabilistically according to the debt ratio of that peer. This incentivizes a node to service DNN models that its peers need; thus, more popular DNN models are serviced by more peers. With the right incentive, peers are organized in a decentralized manner to dissipate the inference load of popular DNN models. A node also maintains a trusted peer list where it always services inference requests from these peers, and a blacklist where it never services inference requests from these peers.

8

Conclusion

Today the demand for artificial intelligence applications is ever-growing. Complex DNN models are large and require substantial computing resources. These models are trained and deployed on highly sophisticated and powerful computing devices on the cloud. This has led to related work aiming at reducing their computational complexity and fitting them on smaller and embedded devices. Intelligence Distribu-

tion Network (IDN) as introduced in this dissertation tackles this problem differently. Following in the footsteps of CDNs and learning from the successes of peer-to-peer (p2p) networks for media streaming, file sharing, and content distribution, IDN aims to provide tools and a p2p infrastructure to support this ever-growing demand by allowing end devices to tap into computing resources nearby. In summary, the benefits of IDN discussed here include:

- **Lower cost**. Using the spare computing resources of other devices nearby which would otherwise go unused, therefore lowering the cost of DNN inference.
- **Faster inference**. Using nearby computing devices brings computation closer to the devices that need it and therefore lowers the inference time. With BranchyNet (Chapter 3), the inference computation can exit early, further reducing the inference time.
- **Scalability**. With DDNN (Chapter 4), DNN models can utilize resources beyond a single device and scale across multiple devices, edges, and cloud servers.
- **Fault tolerance**. DNN models generated with ParallelNet (Chapter 5) can execute independently in parallel on devices of varying compute capacity. Not all models are required to return the result, making the inference more robust to faults.
- **Abundance of high-quality DNN models**. DaiMoN (Chapter 6) incentivizes peers to collaborate and share DNN models, improving the quality and increas-

ing the number of publicly available high-quality DNN models.

- **Efficient distribution of DNN inference**. ComputeSwap (Chapter 7) incentivizes peers to participate in the network, enabling efficient distribution of DNN models and inference computation.

This dissertation introduces both the frameworks and neural network architecture motifs that developers can utilize in order to adapt their models to optimally take advantage of the IDN platform. By using IDN and these associated motifs, users can tap into the computing resources of other peers nearby, democratizing neural network inference.

These frameworks introduced here for use with IDN include BranchyNet, Distributed Deep Neural Network (DDNN), and ParallelNet. BranchyNet (Chapter 3) proposes an augmentation to a DNN model by adding early exit branches to exit the computation early, reducing unnecessary communication and computation. Distributed Deep Neural Network (DDNN) (Chapter 4) takes this concept further by mapping a DNN model to a computing hierarchy consisting of end devices, edges, and clouds, allowing the computation to exit early at various tiers of the computing hierarchy and scale beyond the capacity and capabilities of a single device. ParallelNet (Chapter 5) builds upon this concept to allow devices of different capability to execute different-sized models independently in parallel, and to terminate the computation early when enough confidence is accumulated, increasing redundancy and fault tolerance of the

services as not every model is required to return the result. Since the introduction of these frameworks, people have used and adapted these frameworks to design efficient inference systems for various applications [104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114].

IDN relies on the availability of various DNN models for peers to use. To tackle this problem, IDN introduces DaiMoN (Chapter 6), a decentralized artificial intelligence model network, which incentivizes peer collaboration in improving the accuracy of DNN models. A main feature of DaiMoN is that it allows peers to verify the accuracy improvement of submitted models without knowing the test labels. This is an essential component in order to mitigate intentional model overfitting by model-improving peers. As a result, DaiMoN will encourage peers to collaborate, increasing the quality and the number of publicly available models for use on IDN.

To manage how IDN is used, ComputeSwap (Chapter 7) is introduced to incentivize peers to participate in the network and efficiently distribute inference computation among these peer computing devices. In ComputeSwap, peers execute inference requests optimistically and keep ledgers of debts that other peers owe and credits that other peers give. Peers execute inference requests probabilistically where the probability of servicing reduces as the debt-to-credit ratio increases.

IDN makes an impact on the field of artificial intelligence in two ways. First, IDN provides a p2p network of computing resources that researchers and application de-

velopers can tap into. As the demand for artificial intelligence applications grows, so does the demand for computing resources. With IDN, researchers and application developers now have access to an ever-growing source of spare computing resources nearby that would otherwise go unused. Having a platform like IDN to build upon will help accelerate the research and development of novel artificial intelligence applications.

Second, IDN as a platform opens up a new frontier of research topics. As more applications run on IDN, new problems will arise. More tools and frameworks will be developed to support various applications on IDN. It is the hope that IDN will have an impact on how artificial intelligence applications are built and used in the decades to follow.

8.1 Future Work

Though this dissertation touches upon various topics that make up IDN, it is in no way exhaustive. A lot of future and continuing work is needed. However, it is the hope that IDN will provide a platform for future research and developments of DNN models and inference using nearby peer computing resources.

This dissertation explores a few criteria for exiting samples and provides some heuristics in selecting the exit thresholds. However, these are not exhaustive. Future work

may explore other criteria for exiting samples and other methods for finding the exit thresholds.

This dissertation applies the presented frameworks and neural network architecture motifs to only image classification problems. However, these frameworks and motifs are generic and could be applied to other application domains. Future work may explore the use of these frameworks and motifs in other application domains such as object detection, natural language processing, and reinforcement learning.

Topics that are not discussed in this dissertation but could be helpful are other techniques to improve the performance and inference time of DNN models. These techniques include, for example, computational graph optimization, neural network pruning, quantization, and compression [29, 43, 115, 116, 117, 118, 119, 120, 121]. These techniques could further be developed to take advantage of available nearby peer computing resources on IDN.

DaiMoN aims to solve the quality and availability problem of DNN models to encourage peers to work collaboratively in a decentralized manner to create DNN models that solve a particular task and make them available for use. Future research in this area includes different neural network architectures to learn the distance embedding function to further protect against future attacks. Theoretical foundation of those functions can also be explored.

There is still a lot to explore in the design of IDN itself. We only touched upon a single ComputeSwap strategy in this dissertation. Future work could explore different strategies including different penalty functions for peers with high debt ratios. IDN currently maintains a ledger for each pair of peers. Future work could use a common public ledger for all peers, which will allow the use of a virtual currency that peers can exchange for inference services. Future iterations of IDN can also improve the efficiency and performance of the network by including performance metrics such as availability, response time, and bandwidth in the decision of whether nodes can host a particular inference service.

IDN is a platform which can further grow to provide support for DNN training. Training techniques such as distributed deep learning and federated learning [51, 52, 54, 122, 123] can also be explored on the IDN platform. Federated learning on IDN can take advantage of the locality of the data from nearby devices. With federated learning, IDN nodes can learn from each other's data in the same vicinity and generate models that are specialized to the data of that particular area. For example, object detection services on IDN nodes in Australia where there are a lot of kangaroos will be able to detect kangaroos really well.

Another area that has not been touched upon much in this dissertation is privacy and security. Though certain applications do not require privacy and security, these could be important to other applications dealing with sensitive information. Work such as

differential privacy, zero-knowledge proofs, multi party computation, and homomorphic encryption [124, 125, 126, 127, 128] can further be researched and included in the future iterations of IDN.

References

[1] 1999 annual report/10k, Akamai, 1999.

[2] Annual report 2016, Akamai, 2016.

[3] Content delivery network market by type & region - Global forecast 2022, 2017.

[4] 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.

[5] Global AI market size 2016-2025, 2016.

[6] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[7] Wikimedia Commons. File:neuron-figure.svg. https://commons.wikimedia.org/wiki/File:Neuron-figure.svg, 2007.

[8] Wikimedia Commons. File:synapseschematic_lines.svg. https://commons.wikimedia.org/wiki/File:SynapseSchematic_lines.svg, 2015.

[9] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

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

[11] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[12] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for natural language processing. *arXiv preprint arXiv:1606.01781*, 2016.

[13] Rie Johnson and Tong Zhang. Convolutional neural networks for text categorization: Shallow word-level vs. deep character-level. *arXiv preprint arXiv:1609.00718*, 2016.

[14] 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.

[15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[16] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

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

[19] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[20] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$. In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[21] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE, 2013.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[24] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. *arXiv preprint arXiv:1706.04599*, 2017.

[25] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. *online:* `https://www.cs.toronto.edu/~kriz/cifar.html`, 2014.

[26] Adam Paszke, Soumith Chintala, Ronan Collobert, Koray Kavukcuoglu, Clement Farabet, Samy Bengio, Iain Melvin, Jason Weston, and Johnny Marietho z. PyTorch: Tensors and dynamic neural networks in python with strong gpu acceleration, may 2017.

[27] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3, 2017.

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

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

[30] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[31] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[32] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, page 4, 2011.

[33] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[34] Andrew Lavin. Fast algorithms for convolutional neural networks. *arXiv preprint arXiv:1509.09308*, 2015.

[35] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[36] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[37] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[38] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. Deep networks with stochastic depth. *arXiv preprint arXiv:1603.09382*, 2016.

[39] Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 475–480. IEEE, 2016.

[40] 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.

[41] Karolj Skala, Davor Davidovic, Enis Afgan, Ivan Sovic, and Zorislav Sojat. Scalable distributed computing hierarchy: Cloud, fog and dew computing. *Open Journal of Cloud Computing (OJCC)*, 2(1):16–24, 2015.

[42] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

[43] Bradley McDanel, Surat Teerapittayanon, and HT Kung. Embedded binarized neural networks. In *International Conference on Embedded Wireless Systems and Networks*, 2017.

[44] Gemma Roig, Xavier Boix, Horesh Ben Shitrit, and Pascal Fua. Conditional random fields for multi-camera object detection. In *2011 International Conference on Computer Vision*, pages 563–570. IEEE, 2011.

[45] Xi Li, Weiming Hu, Chunhua Shen, Zhongfei Zhang, Anthony Dick, and Anton Van Den Hengel. A survey of appearance models in visual object tracking. *ACM transactions on Intelligent Systems and Technology (TIST)*, 4(4):58, 2013.

[46] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 73–78. IEEE, 2015.

[47] Michael Figurnov, Maxwell D. Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. *arXiv preprint arXiv:1612.02297*, 2016.

[48] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[49] Forrest N Iandola, Khalid Ashraf, Mattthew W Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *arXiv preprint arXiv:1511.00175*, 2015.

[50] Jeff Dean. Large scale deep learning. In *Keynote GPU Technical Conference*, volume 3, page 2015, 2015.

[51] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

[52] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated Learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[53] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191. ACM, 2017.

[54] Corentin Hardy, Erwan Le Merrer, and Bruno Sericola. Distributed deep learning on edge-devices: feasibility via adaptive compression. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2017.

[55] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. *arXiv preprint arXiv:1804.08333*, 2018.

[56] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. Federated learning with non-IID data. *arXiv preprint arXiv:1806.00582*, 2018.

[57] Pierre Baldi and Peter J Sadowski. Understanding dropout. In *Advances in neural information processing systems*, pages 2814–2822, 2013.

[58] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066, 2013.

[59] Saurabh Singh, Derek Hoiem, and David Forsyth. Swapout: Learning an ensemble of deep architectures. In *Advances in neural information processing systems*, pages 28–36, 2016.

[60] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[61] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

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

[63] Model zoo. https://github.com/BVLC/caffe/wiki/Model-Zoo.

[64] PyTorch model zoo. https://pytorch.org/docs/stable/torchvision/models.html.

[65] Models and examples built with TensorFlow. https://github.com/tensorflow/models.

[66] Model zoo. https://modelzoo.co/.

[67] Kaggle. https://www.kaggle.com.

[68] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[69] William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

[70] Kasper Green Larsen and Jelani Nelson. Optimality of the Johnson-Lindenstrauss lemma. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 633–638. IEEE, 2017.

[71] AJ Stam. Limit theorems for uniform distributions on spheres in high-dimensional euclidean spaces. *Journal of Applied probability*, 19(1):221–228, 1982.

[72] Shengqiao Li. Concise formulas for the area and volume of a hyperspherical cap. *Asian Journal of Mathematics and Statistics*, 4(1):66–70, 2011.

[73] Juan Benet. IPFS-content addressed, versioned, P2P file system. *arXiv preprint arXiv:1407.3561*, 2014.

[74] Ronald Rivest. The MD5 message-digest algorithm. Technical report, 1992.

[75] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[76] William Wesley Peterson and Daniel T Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[77] Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard, 2015. *online:* https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md, 2018.

[78] Juan Benet. Filecoin: A decentralized storage network. *online:* https://filecoin.io/filecoin.pdf, 2017.

[79] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. 2014.

[80] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[81] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *STOC*, volume 97, pages 618–625. Citeseer, 1997.

[82] Kang Zhao, Hongtao Lu, and Jincheng Mei. Locality preserving hashing. In *AAAI*, pages 2874–2881, 2014.

[83] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[84] Ben Goertzel, Simone Giacomelli, David Hanson, Cassio Pennachin, and Marco Argentieri. SingularityNET: A decentralized, open market and inter-network for AIs. 2017.

[85] Jesse Eisses, Laurens Verspeek, and Chris Dawe. Effect network: Decentralized network for artificial intelligence. *online:* http://effect.ai/download/effect_whitepaper.pdf, 2018.

[86] Richard Craib, Geoffrey Bradway, Xander Dunn, and J Krug. Numeraire: A cryptographic token for coordinating machine intelligence and preventing overfitting. *Retrieved*, 23:2018, 2017.

[87] Ronald L Rivest, Adi Shamir, and Leonard M Adleman. Cryptographic communications system and method, September 20 1983. US Patent 4,405,829.

[88] Multihash. https://github.com/multiformats/multihash.

[89] D Eastlake 3rd and Paul Jones. US secure hash algorithm 1 (SHA1). Technical report, 2001.

[90] Libp2p, a modular network stack. https://libp2p.io/.

[91] Jon Postel et al. Rfc 768: User datagram protocol, 1980.

[92] Jon Postel et al. Rfc 793: Transmission control protocol, 1981.

[93] Alan B Johnston and Daniel C Burnett. *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web*. Digital Codex LLC, 2012.

[94] Multiaddr. https://github.com/multiformats/multiaddr.

[95] Jon Postel et al. Rfc 791: Internet protocol. 1981.

[96] Michael J Freedman and David Mazieres. Sloppy hashing and self-organizing clusters. In *International Workshop on Peer-to-Peer Systems*, pages 45–55. Springer, 2003.

[97] Michael J Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with coral. In *NSDI*, volume 4, pages 18–18, 2004.

[98] Jonathan Sterne. *MP3: The meaning of a format*. Duke University Press, 2012.

[99] Clay Shirky. What is p2p... and what isn't. In *The O'Reilly P2P Conference*, 2000.

[100] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@Home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[101] Stefan M Larson, Christopher D Snow, Michael Shirts, and Vijay S Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *arXiv preprint arXiv:0901.0866*, 2009.

[102] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings first international conference on peer-to-peer computing*, pages 99–100. IEEE, 2001.

[103] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *International Workshop on Peer-to-Peer Systems*, pages 205–216. Springer, 2005.

[104] Yu-Yi Su, Yung-Chih Chen, Xiang-Xiu Wu, and Shih-Chieh Chang. Dynamic early terminating of multiply accumulate operations for saving computation cost in convolutional neural networks. 2018.

[105] Mohammad Saeed Shafiee, Mohammad Javad Shafiee, and Alexander Wong. Efficient inference on deep neural networks by dynamic representations and decision gates. *arXiv preprint arXiv:1811.01476*, 2018.

[106] Mou-Yue Huang, Ching-Hao Lai, and Sin-Horng Chen. Fast and accurate image recognition using deeply-fused branchy networks. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 2876–2880. IEEE, 2017.

[107] Chi Lo, Yu-Yi Su, Chun-Yi Lee, and Shih-Chieh Chang. A dynamic deep neural network design for efficient workload allocation in edge computing. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 273–280. IEEE, 2017.

[108] Hsi-Shou Wu. Energy-efficient neural network architectures. 2018.

[109] Yi Zhou, Yue Bai, Shuvra S Bhattacharyya, and Heikki Huttunen. Elastic neural networks for classification. *arXiv preprint arXiv:1810.00589*, 2018.

[110] Lijie Li, Yan Zhang, and Pengfei Wang. Compression of conditional deep learning network for fast and low power mobile applications. In *2nd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2017)*. Atlantis Press, 2017.

[111] Matthew Halpern, Behzad Boroujerdian, Todd Mummert, Evelyn Duesterwald, and Vijay Janapa Reddi. One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers.

[112] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services. *arXiv preprint arXiv:1801.08618*, 2018.

[113] Ji Wang, Bokai Cao, Philip Yu, Lichao Sun, Weidong Bao, and Xiaomin Zhu. Deep learning towards mobile applications. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1385–1393. IEEE, 2018.

[114] Dimitrios Stamoulis, Ting-Wu Rudy Chin, Anand Krishnan Prakash, Haocheng Fang, Sribhuvan Sajja, Mitchell Bognar, and Diana Marculescu. Designing adaptive neural networks for energy-constrained image classification. In *Proceedings of the International Conference on Computer-Aided Design*, page 23. ACM, 2018.

[115] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[116] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[117] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[118] 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.

[119] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016.

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

[121] Brad McDanel, Surat Teerapittayanon, and HT Kung. Incomplete dot products for dynamic computation scaling in neural network inference. In *2017*

16th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 186–193. IEEE, 2017.

[122] Karanbir Chahal, Manraj Singh Grover, and Kuntal Dey. A hitchhiker's guide on distributed training of deep neural networks. *arXiv preprint arXiv:1810.11787*, 2018.

[123] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. Federated multi-task learning. In *Advances in Neural Information Processing Systems*, pages 4424–4434, 2017.

[124] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 103–112. ACM, 1988.

[125] Cynthia Dwork. Differential privacy. *Encyclopedia of Cryptography and Security*, pages 338–340, 2011.

[126] Yehida Lindell. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining*, pages 1005–1009. IGI Global, 2005.

[127] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178, 2009.

[128] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of cryptology*, 1(2):77–94, 1988.