Simulating an Optical Neural Network for Deep Learning in Edge Computing

by

Jared Cochrane

B.S. Physics,
United States Military Academy at West Point (2020)

Submitted to the Institute for Data, Systems, and Society
in partial fulfillment of the requirements for the degree of

Master of Science in Technology and Policy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Institute for Data, Systems, and Society
May 6, 2022

Certified by .....
Dirk Englund
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Kenneth Oye
Professor of Political Science and Data Systems
Thesis Supervisor

Accepted by .....
Noelle E. Selin
Director, Technology and Policy Program
Professor, Institute for Data, Systems, and Society and Department of Earth, Atmospheric and Planetary Sciences

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Simulating an Optical Neural Network for Deep Learning in Edge Computing

by

Jared Cochrane

Submitted to the Institute for Data, Systems, and Society
on May 6, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science in Technology and Policy

Abstract

Deep learning has risen to prominence in fields from medicine to autonomous vehicles. This rise has been driven by improvements in parallel computing from graphics processing units (GPUs) as well as large data sets. Applying deep learning to edge computing is challenging because deep neural network (DNN) hardware must not only possess the needed computational power but must also satisfy size, weight, and power (SWaP) constraints for practical deployment. Many DNNs require a GPU or data center to run, both of which are too large to fit onto edge devices. Here, an optical neural network (ONN) accelerator called netcast is simulated on two real-world machine vision applications: MNIST digit classification and scene recognition. The netcast ONN enables large DNNs to run on SWaP-limited edge devices with significantly less energy needed to run inference compared to digital models. Software simulations are used to assess netcast's performance on MNIST classification and scene recognition relative to digital networks. Using an accuracy per energy consumption figure of merit (FOM), the simulations indicate that netcast is able to outperform digital electronics on average by over three orders of magnitude. Netcast's strong performance relative to its digital counterparts indicates that it will enable the novel deployment of large DNNs to edge applications in a way that would be infeasible using current digital electronics. Netcast's novel applications give rise to a host of policy challenges, one of which focuses on defining and applying acceptable performance metrics to optically enabled deep learning.

Thesis Supervisor: Dirk Englund
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Kenneth Oye
Title: Professor of Political Science and Data Systems

Acknowledgments

I would like to acknowledge and thank MIT Lincoln Labs Group 67 for their support and flexibility in allowing me to conduct research with the Quantum Photonics Group (QPG) on campus. Special thanks to Scott Hamilton, Catherine Lee, and Ben Dixon for their assistance and guidance throughout the research process.

I would also like to thank Alex Sludds, Ryan Hamerly, and Professor Dirk Englund for giving me the opportunity to research with QPG and for helping revise the thesis manuscript. Special thanks to Alex Sludds for his guidance and help regarding the technical aspects of the thesis- everything from the simulation theory to the netcast hardware architecture. Special thanks to Ryan Hamerly for his guidance and help regarding the simulation methodology employed in this thesis.

Finally, I would like to thank Professor Kenneth Oye for his assistance with regard to the policy chapter of this thesis as well as the broader considerations of netcast's social and policy implications.

Contents

List of Figures

List of Tables

Chapter 1

Applying Deep Neural Networks (DNNs) to Machine Vision Applications

The application of deep neural networks (DNNs) to help solve different problems including object classification, computational imaging, and depth sensing has evolved into a "third wave" of interest [67]. This heightened interest is largely due to increased training data size and computational capability from parallel computing. This chapter introduces the basic theory behind deep neural networks and then discusses how such networks can be applied to machine vision problems. Section 1.1 introduces deep learning and identifies where it fits into current artificial intelligence (AI) research. Section 1.2 discusses two common deep learning model architectures that this thesis will focus on: fully connected and convolutional networks. Section 1.3 discusses the two machine vision applications that netcast will be simulated on: MNIST digit classification and scene recognition for mobile robotics. Finally, section 1.4 summarizes the conclusions and main points of the chapter.

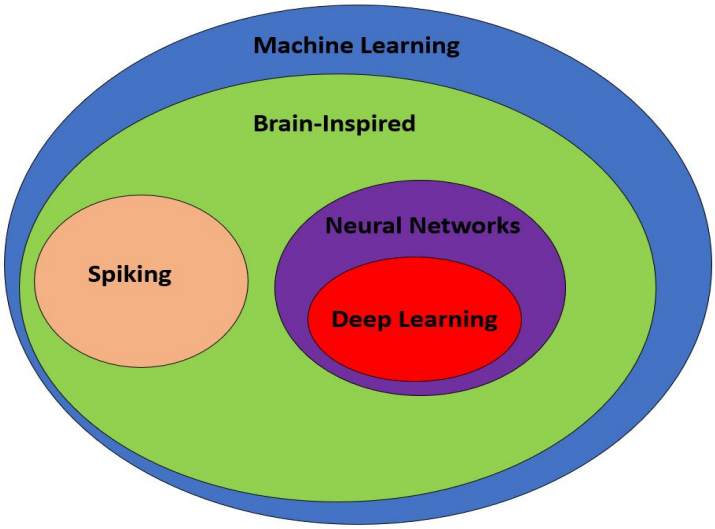


Figure 1-1: Overview of where deep neural networks (DNNs) fit into the taxonomy of machine learning research. Deep learning is a branch of brain-inspired machine learning with more than one hidden layer in its architecture.

1.1 Deep Learning: Introduction and Background

Before looking at deep learning's application to machine vision, it is first necessary to clearly define deep learning. As noted by [33], there is a lack of consensus regarding a universal definition of deep learning. This paper adopts the definition proposed by Sze et al. in [30], which is illustrated in figure 1-1. As can be seen in the figure, deep learning is a brain-inspired branch of machine learning. Machine learning is "a field of study that gives computers the ability to learn without being explicitly programmed", as articulated by Arthur Samuel who coined the term in 1959 [30]. While machine learning is a very broad field, a subset of machine learning technologies derives inspiration from the most sophisticated and complex computational machine ever created: the human brain. Figure 1-2 shows a neuron in the human brain, while figure 1-3 depicts an artificial neuron used in deep learning. Deep neural networks (DNNs) derive their inspiration from neurons in the human brain, which take inputs $[x_1, x_2, ... \ x_M]$ and perform a computation that weights the inputs with $[w_1, w_2, ... \ w_N]$. After performing this linear operation, each neuron (neuron j) will only fire if the weighted sum $y_j = \sum_{i=1}^{N} w_i x_j + b$ exceeds some threshold value. This introduces a

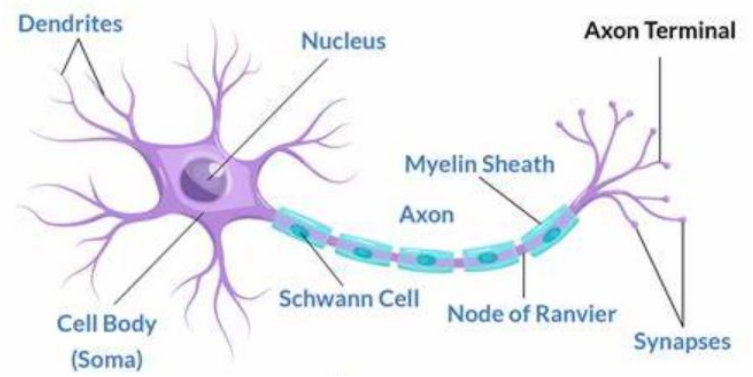


Figure 1-2: Structure of a neuron in the human brain. Incoming signals enter through the dendrites and leave through the axon. The synapses weight the relative importance of different input signals [9].

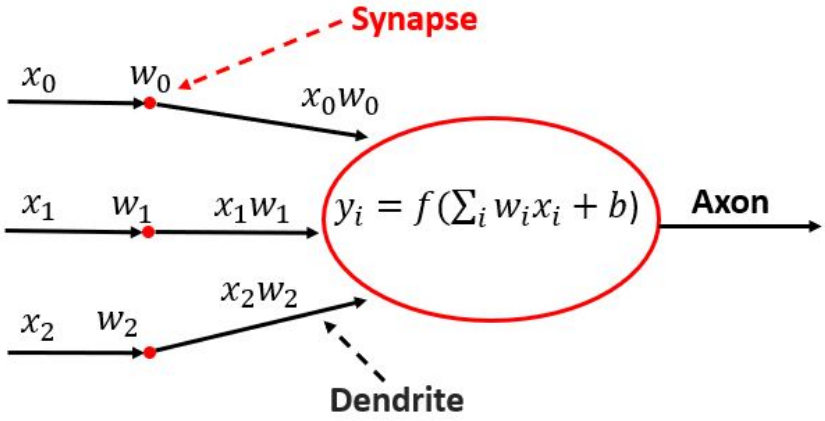


Figure 1-3: Artificial neuron structure. Similar to its biological counterpart, inputs travel through the dendrites while the synapses store the weight elements. The neuron applies a nonlinear function f to the weighted sum of the inputs, and the output exits through the axon.

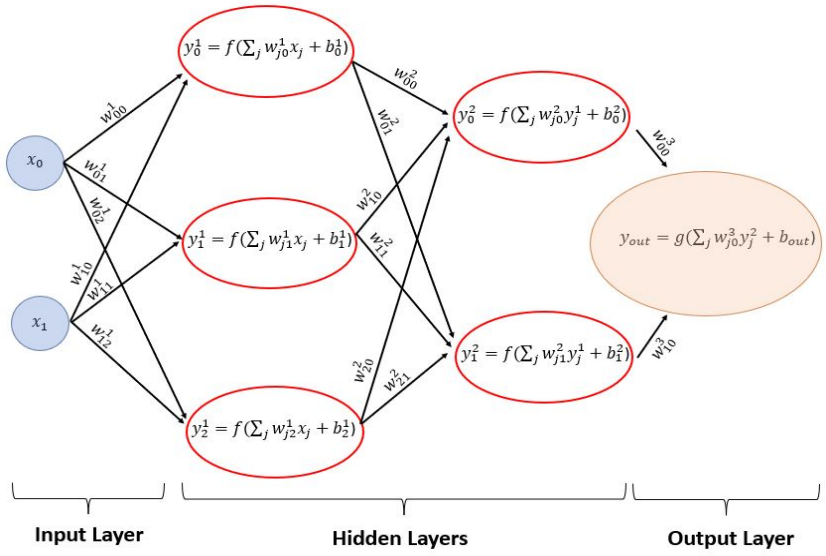


Figure 1-4: Fully Connected DNN. y_j^ℓ is the jth activation from the ℓth layer of the network. w_{ij}^ℓ is the weight element in layer ℓ that maps input x_i to activation y_j through multiplication. b_j^ℓ is a scalar bias term added to the jth output activation in layer ℓ. f and g denote nonlinear activation functions.

nonlinearity denoted by the function f. This nonlinearity plays a key role in allowing DNNs to learn complex causal relationships from data [48]. With this principle in mind, DNNs can be conceptualized as collections of artificial neurons that process input data in a series of layers where computations are performed in parallel at each layer and then sequentially passed through the network. An illustration of a simple DNN (called a fully connected network) is given in figure 1-4. The number of hidden layers (as seen in figure 1-4) distinguishes generic neural networks from deep neural networks. More specifically, DNNs are the subset of neural networks with more than one hidden layer [30]. Among the various possible DNN structures, this thesis will focus on two: fully connected and convolutional networks. These two architectures, along with the math that underlies them, are described in section 1.2.

1.2 Two Common Deep Neural Network Architectures: Fully Connected and Convolutional

Having introduced the basic motivation behind DNNs, this section describes the two DNN architectures that this thesis focuses on: fully connected and convolutional networks. These architectures will be used later in chapter 3 for the netcast simulations.

1.2.1 Fully Connected Networks

Figure 1-4 shows the structure of a fully connected DNN. In this architecture, each neuron takes a weighted sum of all inputs, applies a nonlinearity, and then feeds the output into each neuron of the next layer. In terms of notation, y_j^ℓ is the jth output activation from the ℓth layer of the network. w_{ij}^ℓ is the weight element in layer ℓ that maps input activation x_i to output activation y_j through scalar multiplication. b_j^ℓ is a scalar bias term added to the sum $\sum_j w_{ji}^\ell x_j$ before the nonlinear function (f) is applied. Note that g is also a nonlinear activation function where in general $f \neq g$ since g resides at the output layer and f resides in the hidden layers. It is important to note that the superscripts in w_{ij}^ℓ and y_j^ℓ do not denote exponents, but instead index layers in the network. This notation is canonically used because weights and activations in different layers of a DNN are almost always different from each other. Computation in a fully connected DNN can be succinctly formulated as iterative matrix multiplications. Equation 1.1 gives the general case where $\vec{x} \, \epsilon \, R^{Mx1}$ is the input activation to layer ℓ and $\vec{y} \, \epsilon \, R^{Nx1}$ is the output activation from layer ℓ.

$$\vec{y^\ell} = \begin{bmatrix} y_0^\ell \\ y_1^\ell \\ ... \\ y_N^\ell \end{bmatrix} = f(\begin{bmatrix} w_{00}^\ell & w_{10}^\ell & ... & w_{M0}^\ell \\ w_{01}^\ell & w_{11}^\ell & ... & w_{M1}^\ell \\ w_{02}^\ell & w_{12}^\ell & ... & w_{M2}^\ell \\ ... & ... & ... & ... \\ w_{0N}^\ell & w_{1N}^\ell & ... & w_{MN}^\ell \end{bmatrix} \begin{bmatrix} x_0^{\ell-1} \\ x_1^{\ell-1} \\ ... \\ x_M^{\ell-1} \end{bmatrix} + \begin{bmatrix} b_0^\ell \\ b_1^\ell \\ ... \\ b_N^\ell \end{bmatrix}) = f(\mathbf{W}^{(\ell)^T} x^{(\vec{\ell-1})} + \vec{b^\ell}) \tag{1.1}$$

Where, in the general case, $W \epsilon R^{MxN}$ so that $W^T \epsilon R^{NxM}$ as given in equation 1.1. Also, the superscript ℓ indexes the layer that is doing the computation. Note that W, b, and f are all specific to the layer that is performing the computation. Since fully connected networks consist of hidden layers (at least 2), equation 1.1 can be applied at each layer sequentially to calculate a forward pass through the DNN: $\vec{y} = DNN(\vec{x})$. An explicit equation to describe the forward pass is best understood as a recursion where the output at layer ℓ depends on the output from the previous layer $\ell - 1$. Equations 1.2 and 1.3 completely define a forward pass through a fully connected DNN. To be consistent with commonly used notation, the layer output $\vec{y^\ell}$ is often separated into two parts: Z^ℓ that represents the weighted sum before applying the nonlinearity, and A^ℓ that represents the nonlinearity output. Z^ℓ is often referred to as the pre-activation value, while A^ℓ is the output activation.

$$Z^{(\ell)} = \mathbf{W}^{(\ell)^T} A^{(\ell-1)} + \mathbf{W_0^\ell} \tag{1.2}$$

$$A^{(\ell)} = f(Z^{(\ell)}) \tag{1.3}$$

$\mathbf{W_0^\ell}$ is the bias term $\vec{b^\ell}$ and ℓ indexes the layers as before. The input to the first layer of the network is $Z^{(0)} = \vec{x}$ and the final network output is $A^{(\mathcal{L})} = \vec{y}$ for a fully connected DNN with "$\mathcal{L}$" layers in it. One common application that fully connected networks are well suited for is the object classification task that will be considered in more detail in chapter 3. Before proceeding, however, it is first important to fill in the function f with a specific form, which is done in section 1.2.2 below.

1.2.2 Two Common Nonlinear Functions: RELU and Softmax

Equation 1.3 uses a nonlinearity f at the output of each DNN layer. This section presents and describes two common nonlinear functions that will be used throughout the remainder of this thesis: RELU and Softmax. As mentioned in section 1.1, an important aspect of DNNs is the nonlinearity that resides between consecutive hidden

layers. If DNNs were entirely composed of linear functions that perform matrix multiplication, they would be unable to learn complex and nonlinear features in data. This is because a forward pass through a DNN without any nonlinear functions can be reduced to a single matrix multiplication [11]. To briefly demonstrate this, consider a two layer DNN that doesn't use nonlinear functions between layers. In this case, let the two layers have weight matrices W_1 and W_2. Then, $Z^L = A^L$ in equation 1.3, making f and g simple identity mappings. The forward pass in this case is simply...

$$A^{(2)} = f(\mathbf{W^{(2)T}} A^{(1)} + \mathbf{W_0^2}) \tag{1.4}$$

$$A^{(2)} = f(\mathbf{W^{(2)T}} f[\mathbf{W^{(1)T}} A^{(0)} + \mathbf{W_0^1}] + \mathbf{W_0^2}) \tag{1.5}$$

$$A^{(2)} = \mathbf{W^{(2)T}} \mathbf{W^{(1)T}} A^{(0)} + \mathbf{W^{(2)T}} \mathbf{W_0^1} + \mathbf{W_0^2} \tag{1.6}$$

$$A^{(2)} = \mathbf{W}^* A^{(0)} + \mathbf{W_0^*} \tag{1.7}$$

$\mathbf{W}^*$ is the product of the two layer weight matrices and $\mathbf{W_0^*}$ is a linear combination of the bias terms. This means that, without nonlinear functions, DNNs will only be able to learn very simple linear relationships in data that will severely limit their practical usefulness. Having motivated the importance of nonlinear functions in DNNs, two common nonlinearities are developed and described: RELU and Softmax.

In an attempt to mimic neurons firing in the human brain, early machine learning research used a simple step function as the nonlinearity, which is plotted in the left-most part of figure 1-5 and defined in equation 1.8.

$$f(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{1.8}$$

While the step function does provide the desired nonlinearity in the forward pass, it suffers from one major drawback: its derivative is undefined at $x = 0$. This discontinuity means that DNNs using a step function nonlinearity cannot be trained using derivative-based optimization algorithms such as gradient descent. In order to implement a nonlinearity similar to the step function that is differentiable, the

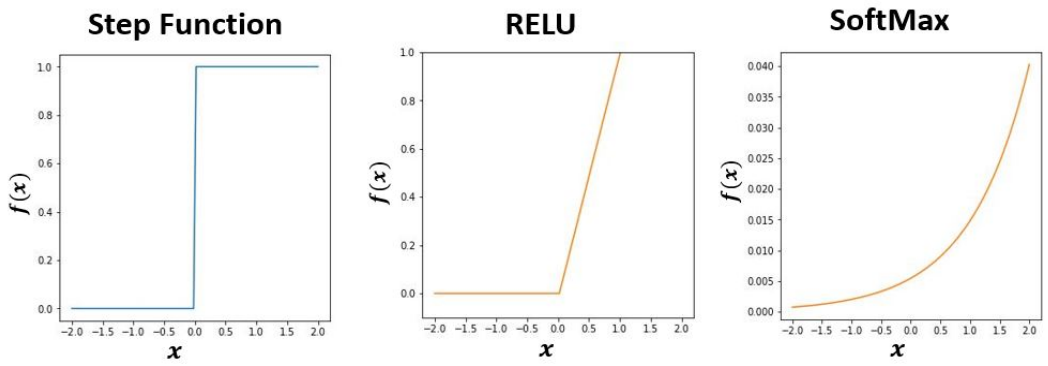


Figure 1-5: The step function (left) and two common nonlinear functions that will be used throughout this thesis: RELU (middle) and Softmax (right). RELU nonlinearity is commonly used between sequential hidden layers in a DNN and is better suited for deep learning than the step function because its derivative is well defined. Softmax is often used at the output layer because it maps a vector to a normalized probability distribution.

rectified linear unit (RELU) was developed. The RELU nonlinearity is plotted in the center part of figure 1-5 and given in equation 1.9.

$$RELU(x) = max(x, 0) \tag{1.9}$$

The derivative of RELU is well defined and is in fact just the step function.

$$\frac{\partial}{\partial x} RELU(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

RELU is commonly implemented between DNN layers and is usually not used at the output layer. Usually, the nonlinearity at the DNN output is the Softmax function. As will be discussed in section 1.3, one very useful application that DNNs are used for is classifying an image as belonging to a certain category within a fixed number of possible categories (classes). In this case, the DNN output is typically a vector of probabilities that are assigned to each possible category. This motivates the Softmax nonlinearity, which is plotted in the right panel of figure 1-5 and given by equation

1.10.

$$Softmax(x) = \frac{1}{\sum_{i=0}^{k} exp(x_k)} \begin{bmatrix} exp(x_1) \\ exp(x_2) \\ ... \\ exp(x_k) \end{bmatrix} \tag{1.10}$$

Softmax outputs a normalized probability distribution over a k-element vector that corresponds to a k-class classification problem. Thus, Softmax is usually the nonlinearity of choice at the output layer of image classification DNNs. These two nonlinear functions are commonly used in both fully connected as well as convolutional models, including those studied in chapter 3. Having covered the structure of fully connected DNNs, section 1.2.3 motivates and describes convolutional neural networks.

1.2.3 Convolutional Neural Networks

Convolution as a Linear Translation Invariant System

Before discussing the general structure of convolutional networks, it is first necessary to characterize the convolution operation as a system and discuss the properties that make it unique. A generic 2D linear system is defined by equation 1.11 below.

$$y[n,m] = f(x[k,l]) = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} h[n,m,k,l]x[k,l] \tag{1.11}$$

Where $x[k,l]$ is the input $h[n,m,k,l]$ is the weight matrix, and $y[n,m]$ is the output [72]. Linear systems are defined by two properties: $y(x_1 + x_2) = y(x_1) + y(x_2)$ and $y(ax_1) = ay(x_1)$ [72]. Of course, linear systems cover a broad range of application domains, while this thesis is primarily interested in image processing. One important characteristic of image processing systems is that they tend to be translation invariant so that the identity and properties of an object do not change upon translation in the image frame. For example, a bird should always be classified as such regardless of where it appears in any particular image. This means that, within the broad realm

of linear systems, image processing systems should be designed to apply the same transformation to each location in an image. One important linear system that is translation invariant is the convolution operation as defined in equation 1.12 below.

$$y[n,m] = f(x[k,l]) = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} h[m-k, n-l]x[k,l] = x * h \tag{1.12}$$

Where the symbol $*$ denotes the 2D convolution that extends naturally to higher dimensions. Notice how equation 1.12 is a translation invariant instance of equation 1.11 since the same filter elements from h are applied to multiple locations across the input x. To get a more intuitive picture of how convolutions are computed, consider the example illustrated in figure 1-6. In practice, convolutions are often

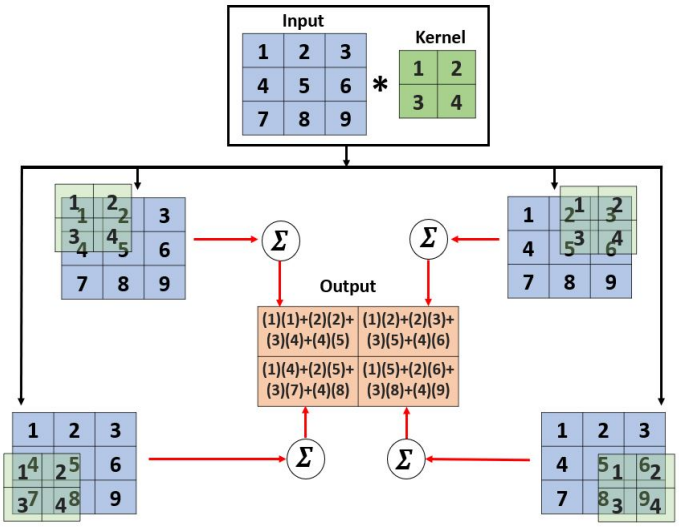


Figure 1-6: An example that shows the computation $y = x * k$ where x is the input, k is the kernel that is convolved with the input, and the resulting output is y.

performed with 3D and 4D inputs, but these are simple extensions of the 2D case. Notice how each 2x2 region of the image in figure 1-6 is multiplied and accumulated with the same weight kernel- this is how a translation invariant system operates. Another important note is that, in the example of figure 1-6, the filter is shifted only a single unit horizontally or vertically as it sweeps along the input image, but this need not be the case. The horizontal or vertical displacement of the filter at

different iterations (called the stride) controls the regions of the input image that are processed. If the stride equals or exceeds the filter size, then there will be some portions of the image that go unprocessed by the kernel. Because of this, the stride used in most convolutional networks tends to be fairly small. Also notice that the output in figure 1-6 is a 2x2 matrix whereas the input matrix is a 3x3 - meaning that the convolution has reduced the image resolution. Because this reduction in resolution is often undesirable, it is common to pad the input matrix with zeros on the edge so that the output resolution matches the input resolution. If the input in figure 1-6 is padded on the left with a 3x1 column of zeros and on the top with a 1x3 row of zeros, then the output resolution would be 3x3. The final important hyper-parameter in convolution is called the dilation, where the weight kernel is upsampled prior to being convolved with the input matrix [28]. Figure 1-7 illustrates dilation that maps a 2x2 weight kernel to an upsampled 3x3 weight kernel that is convolved with the 4x4 input matrix. Many modern convolutional neural networks (CNNs) use dilation to expand the patch size of the image that the kernel is able to process (called the receptive field). Being able to process a larger fraction of the image has led to a steady increase in image processing quality across a variety of applications, which has made dilation an important property in image networks ([25],[83],[22]).

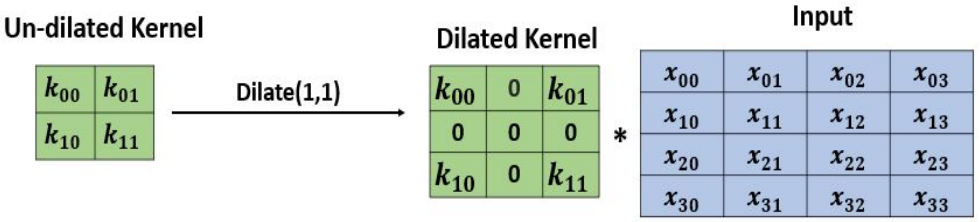


Figure 1-7: Example of dilation from a kernel size of 2x2 to an upsampled kernel size of 3x3 to increase the kernel's receptive field.

Convolutional Neural Network (CNN) Multi-Layer Architecture

The convolution operation forms the basis of convolutional neural networks (CNNs) just as neuron layers form the basis of fully connected networks. As illustrated in

figure 1-8, a convolutional neural network consists of multiple convolutional layers with elementwise nonlinearity and downsampling operations between each layer [59]. These convolutional layers are then followed by the fully connected architecture detailed in section 1.2.1 that yields the final network output: the class of the object shown in the input image ("car" in figure 1-8). The intuition behind CNNs comes from a "divide and conquer" approach where an arbitrarily large image is broken down into smaller patches, each of which can be processed identically and in parallel to yield an aggregated result over the entire input image. Convolutional networks are thus defined by two important properties: (1) each image patch is processed independently from all other patches and (2) each patch is processed identically [72]. Regarding property (1), processing image patches independently is an effective way to achieve image-level tasks because each patch by itself is much easier to operate on than the entire input image all at once. In addition, each patch can be processed in parallel, which means that a "divide and conquer" approach is much more efficient in terms of time and computational efficiency [72]. Regarding property (2), processing each patch in the same manner is an important aspect of CNN computation because of translation invariance [72]. Properties (1) and (2) are illustrated in figure 1-9, where the CNN's goal is to detect the presence of birds in the input image. By using translation invariant kernel filters, the CNN effectively divides the input image into the series of smaller square regions depicted in the figure. After partitioning the image, the CNN then processes each patch using both kernels and fully connected layers to classify each patch as either a bird or sky.

The Importance of Multiple Convolutional Layers

Since the computational burden of a CNN scales with the number of convolutional layers, it is important to understand why a series of multiple layers is typically needed to accurately process image patches. As described by [72], kernels act like feature detectors that extract both low-level information (edges, textures, basic patterns) as well as high-level features (semantic objects and scenes) from the image. Within a trained multi-layer architecture, different convolutional layers extract different types

of image information: early layers detect low-level features (ex. edges), middle layers use those low-level features to detect more complex patterns, and the final layers use those patterns to detect semantic objects, scenes, and other useful high-level features [86]. Therefore, in order to make sense of even a small patch in an image, it is important to use multiple layers to extract all the information needed to form accurate high-level features. Using only a single layer will cause the network to lose a lot of key information about the object that it is trying to classify.

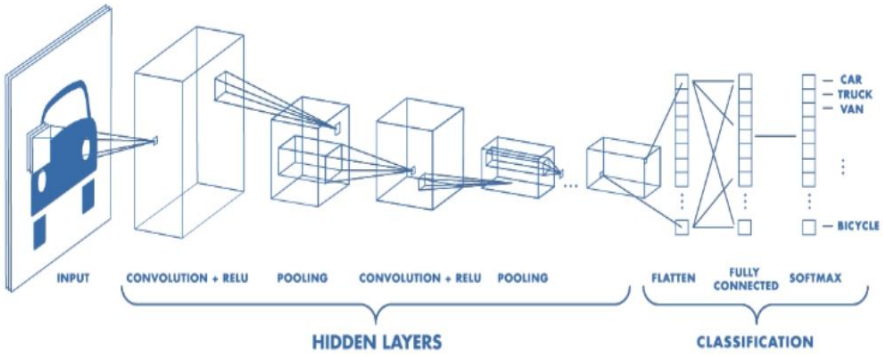


Figure 1-8: Architectural description of a CNN from [59]. The initial layers use convolutional kernels to extract important features from the image (edge locations and orientations, shadows, contact boundaries, etc...), while the final parts of the network are fully connected layers that map the flattened feature representations to a vector of probabilities that outputs the predicted class of the input image.

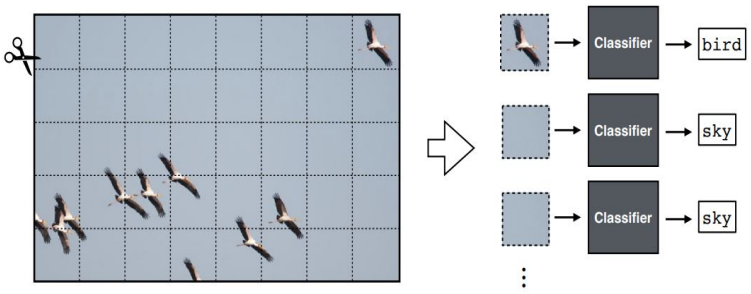


Figure 1-9: Illustration of the two CNN properties from [72]: (1) processing each image patch independently and (2) processing each patch identically. Under these two properties, a CNN takes a "divide and conquer" approach where the input image is partitioned into patches, each of which is run through the same series of convolutional kernels and fully connected layers to classify each patch as either a bird or sky [72].

1.3 Applying Deep Learning to Machine Vision: Object Classification and Scene Recognition

Deep learning has had a tremendous impact in advancing the accuracy of machine vision systems over the past few years [72]. Among the numerous machine vision areas that are currently being researched, this thesis focuses on two: object classification and scene recognition. Since the creation of AlexNet in 2012, the task of object classification has become a canonical application in deep learning vision problems [1]. In this thesis, the object classification data set used is MNIST handwritten digits (0-9).

The second machine vision application considered in this thesis is scene recognition. This application area remains very challenging because effective scene recognition systems must incorporate both localized image features (ex. specific objects in the scene) as well as global features (ex. color) [62]. MNIST digit classification and scene recognition are motivated and described in sections 1.3.1 and 1.3.2 below.

1.3.1 MNIST Handwritten Digit Classification

One common use for convolutional networks is to classify input images as belonging to a certain object category [64]. In the case of MNIST digit classification, a visual image of a handwritten number is mapped to a semantic label (class), where the possible classes in this case are the integers 0-9. Figure 1-10 shows how a CNN uses a combination of convolutional layers and fully connected layers to map an input MNIST image to a class label. Each convolutional kernel acts as a feature detector, and stacking several convolutional layers in sequence allows the network to generate feature maps that are then used by the fully connected layers to generate the image's semantic label. To get an idea of how convolutional layers act as feature detectors, figure 1-11 shows the intermediate convolutional outputs from the hidden layers in a simple 2-layer CNN. Each image represents a different channel of the intermediate feature maps, and it is interesting to notice how each filter extracts different image

features. Once these features have been extracted and processed by the convolutional layers, the fully connected layers can then be used to generate the class prediction. Typically, most of the nonlinearities used between convolutional and fully connected layers are RELU, while the final nonlinearity used is Softmax.

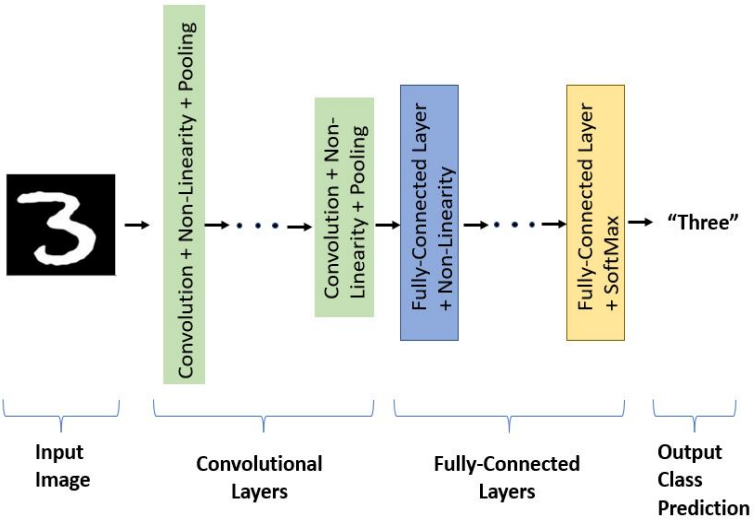


Figure 1-10: Applying a CNN to learn a mapping from an input image to the class label of the object in the image. In this case, the input image is a handwritten digit from the MNIST data set.

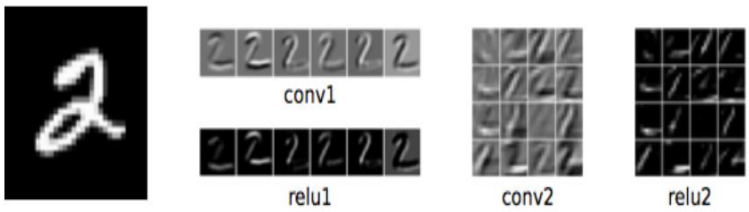


Figure 1-11: Visualizing the intermediate feature maps from a 2-layer CNN applied to MNIST classification. Notice how different filters extract different information from the image. Image credit: `https://stackoverflow.com/questions/45678473/convolution-neural-networks-all-feature-maps-are-blackpixel-value-is-0`.

Object classification networks typically use cross entropy loss (equation 1.13) in order to train the network and evaluate it during inference.

$$L_{CE}(y, \hat{y}) = -\sum_{k=1}^{K} y_k \ln \hat{y}_k \tag{1.13}$$

Where $\hat{y}$ is the network output and y is the corresponding ground truth. An intuitive way to conceptualize MNIST classification is as a probability distribution matching problem. Under this framework, each ground truth label corresponds to a "one-hot" encoded vector, which is a sparse Kx1 vector where K=10 is the number of classes. The label of class k has a 1 at the kth position and zeros everywhere else. For example, the label "three" would correspond to the vector $\begin{bmatrix}0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$ for possible output classes of 0-9. This is essentially a probability distribution with all of the probability mass concentrated at the kth bin. The final CNN layer is a Softmax that generates a normalized probability distribution over the possible image classes. To have an accurate network, the bulk of the probability mass at the output should reside in the kth bin if the ground truth class is k. Figure 1-12 shows how object classification with a Softmax activation layer can be thought of as a distribution matching problem. In chapter 3, netcast will be simulated on the task of MNIST digit classification to see how well it performs relative to digital networks. The simulation will use the theory presented in this chapter to train and test MNIST classifiers - both fully connected and convolutional.

1.3.2 A More Challenging Application: Scene Recognition for Robotic Localization

The second application that this thesis focuses on is scene recognition for robotic localization. Over the past few years, advances in deep learning have given robots an increased prevalence in a variety of tasks - everything from home applications to automated assembly to driving ([24], [32]). One important high-level vision task that robots must perform in order to effectively operate in any environment is simultaneous localization and mapping (SLAM). That is, the robot must have an accurate map of its environment as well as an accurate estimate of its position within that map. This motivates the use of CNNs to map pictures of different scenes to the proper semantic labels that will allow the robot to both identify what scene it is at as well as learn a topological map for its operating environment. For example, figure 1-13 depicts a

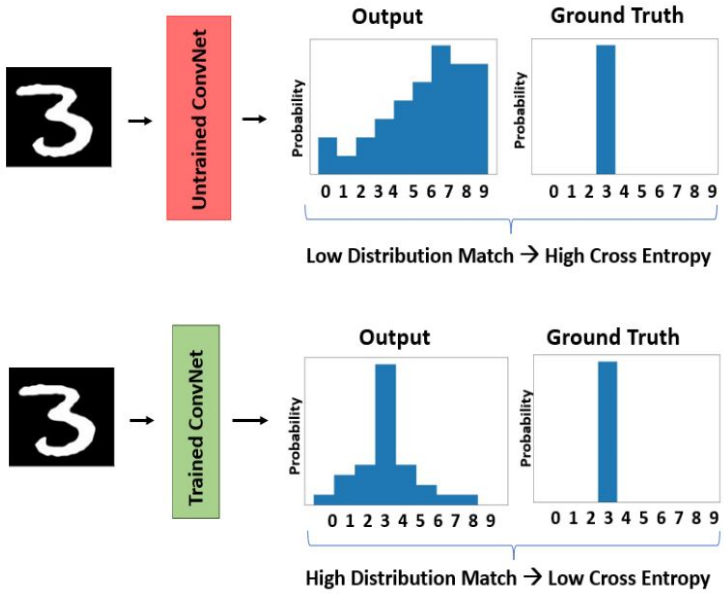


Figure 1-12: Object classification can be formulated as a probability distribution matching problem. An untrained CNN will output a randomly distributed probability mass function from the given input image, while a trained CNN will output a probability distribution that more accurately matches the ground truth distribution. From this, the goal of MNIST classification is to tune the CNN parameters to concentrate probability mass at the correct bin.

house floor plan where a robot is to operate, while figure 1-14 shows the topological map that the robot needs to learn in order to navigate and operate in the house environment. Since this thesis focuses on the machine vision aspect of the localization problem, it is assumed that the robot is able to plan and execute trajectories between the different node locations. This means that the major high-level task that the robot must perform is identifying what room it is in.

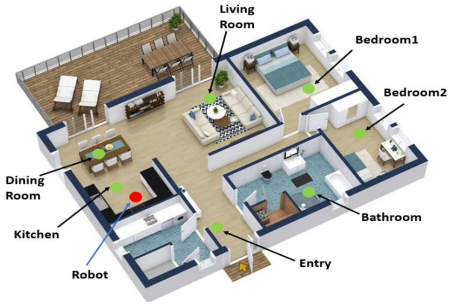


Figure 1-13: Operational environment that a robot will operate in. The type of tasks that the robot will be required to perform depends on where the robot is.

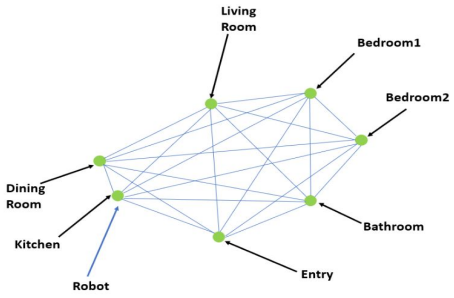


Figure 1-14: Given the environment depicted in figure 1-13, the robot will need to learn the topological map depicted above in order to successfully navigate and operate. Notice that the nodes of the map are the distinct locations in the house, while the light blue lines connecting the nodes represent paths between the distinct locations.

While scene recognition may seem like a simple re-brand of object recognition, this is not the case. As noted by [78], scene recognition is a challenging open question in machine vision because successfully recognizing a scene requires effective inference about both the local features (ex. individual objects or edges in the scene) as well as global features that span the entire scene (ex. color). The challenges associated with scene recognition and the methods currently being used are described in more detail in chapter 3.

1.4 Chapter Conclusion and Summary

This chapter introduced the basic theory behind deep learning- how biological neurons can be approximated by artificial neurons in DNNs, which implement pointwise nonlinearity to emulate the way in which biological neurons fire. After introducing deep learning, two common DNN architectures that will be used throughout this thesis (fully connected and convolutional networks) were introduced. In order to understand the intuition behind convolutional networks, the convolution operation was formulated as a translation invariant linear system that processes individual image patches in parallel using a "divide and conquer" approach. After establishing an intuitive picture of convolution, the application of DNNs to the problems of MNIST classification and scene recognition were discussed. While MNIST digit classification is a canonical machine vision task, scene recognition remains challenging because it

requires a learned understanding of both local and global scene features. Up until this point, only the software aspects of deep learning have been considered. In chapter 2, the application of DNNs to edge computing will be discussed along with the challenges and current methods of implementing DNNs in hardware.

Chapter 2

Applying DNNs to Edge Computing: Challenges and Techniques

2.1 Overview

One important application of DNNs is edge computing, where data is processed close to the source of collection (sensors) to facilitate high bandwidth computation in real-time. As DNNs have been used to solve increasingly complicated tasks, their size (number of parameters) has dramatically increased. Because edge device hardware is limited by size, weight, and power (SWaP), many modern DNNs are too large to be used in edge applications due to prohibitively high energy cost. This has given rise to an increased interest in developing hardware accelerators that allow arbitrarily large networks to be deployed and used on SWaP-limited edge devices.

Currently used Von Neumann computing architectures separate the locations of memory and computation. Not only are these architectures slow, but they also impose a very high energy cost that makes them unsuitable for DNN-based edge applications [29] . In order to transition away from Von Neumann computing in DNNs, current work uses three general methodologies: (1) data flow optimization, (2) co-locating the computation and memory, and (3) model-based energy optimization. In terms of organization, section 2.2 gives an overview of edge computing along with its benefits and challenges. Section 2.3 describes the three methods listed above, while section 2.4

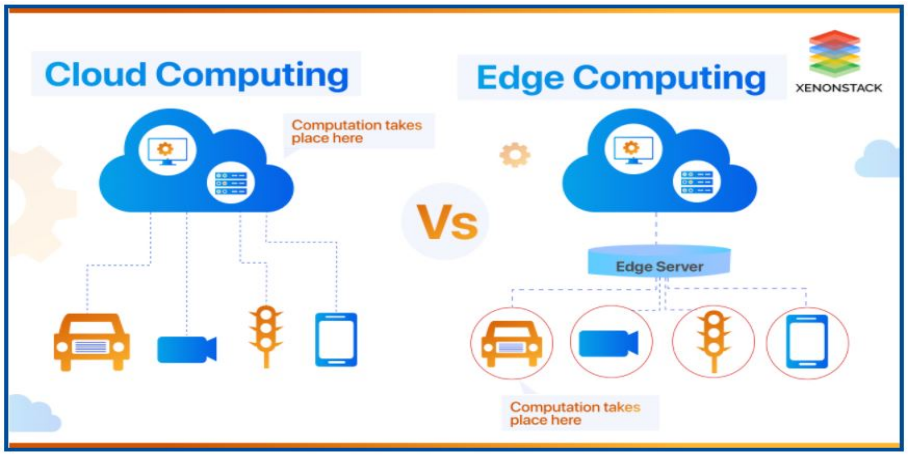


Figure 2-1: Comparison between edge computing and cloud computing [80].

lists current hardware architectures that use these methods. Section 2.5 summarizes the chapter's main points and conclusions.

2.2 Edge Computing

Edge computing is a paradigm in which data processing occurs in physical proximity to the location where data is collected - usually a sensor. Unlike cloud computing where data must be transferred between local devices and the centralized cloud, edge computing is done locally on the device itself. Figure 2-1 depicts the difference between cloud computing and edge computing [80]. Notice that cloud computing keeps processing power centralized so that devices must transmit their data and then wait for it to be processed and sent back [70]. While this is the predominant computing paradigm currently in use, it is not suitable for applications that require real-time data processing. Transmitting data between devices and the cloud incurs a time lag since the devices have to wait for data to be processed in the cloud and then sent back. This time lag makes cloud computing infeasible for applications like autonomous vehicles or mobile robotics, whose ability to operate critically depends on low latency, real-time data processing (ex. real-time collision avoidance).

The time lag that plagues cloud computing has motivated the development of

edge computing, which distributes computing power from the centralized cloud onto local devices as shown in figure 2-1. For example, instead of having a mobile robot transmit visual inputs to a data center, edge computing equips the robot with the computational resources needed to process images by itself with minimal latency. Edge computing introduces both new opportunities as well as new challenges, both of which are described below.

2.2.1 Benefits of Edge Computing

Distributing data processing capabilities onto edge devices has several benefits. First, it reduces processing response times by eliminating the need to transfer data between the edge device and the cloud [61]. Second, it is much more energy-efficient because device $\leftrightarrow$ cloud wireless communication tends to be an energy-hungry process [31]. Third, edge computing helps preserve consumer privacy by keeping personal data geographically confined and not exposing it to the risk of being compromised or stolen on its way to and from the cloud. For example, a smart house that relies on cloud computing is vulnerable to hacking during data transmission, but this would not be an issue with edge computing since the data would never physically leave the house [17]. Fourth, edge computing enables the deployment of DNNs that facilitate smart applications and the Internet of Things (IoT). Cloud computing does not lend itself to deep inference at the edge because transferring large volumes of data to and from the cloud is too slow to be practically useful. For example, autonomous vehicles generate one gigabyte of data every second, which is too much data volume to be processed by the cloud without latency effects [31]. By placing DNNs in close proximity to the data that they process, edge computing alleviates the issue of data transmission bottlenecks [61].

Because of the above benefits, edge computing has been applied to a variety of fields. Current work looks at applications including consumer services [63], search and rescue [63], social virtual reality [26], and IoT [19]. Along with these benefits, however, edge computing also introduces significant and novel challenges, which are described in section 2.2.2 below.

2.2.2 Energy Costs: The Challenge of DNN-Based Edge Computing

While deploying DNNs to edge computing opens the door for new and advanced technological applications, applying DNNs to edge computing introduces several challenges. In order to be practically useful, edge computing DNNs must not only be accurate but must also operate on small edge devices that can only source a limited amount of power. Edge computing DNNs must therefore optimize multiple different objectives including energy costs, bandwidth, accuracy, and size.

The main obstacle hindering high bandwidth and low energy costs in current DNN accelerators is memory access [30]. In order to see how memory access limits DNN hardware performance, consider the multiply and accumulate (MAC) operation, which forms the basis of DNN matrix-vector operations. Figure B-1 in the appendix shows an example of a simple vector-vector multiplication decomposed into MAC operations. As shown in figure 2-2, each MAC operation in a network requires three memory reads - two reads to access the elements being multiplied and one read to access the partial sum to which the product will be added. In addition, each MAC requires a memory write to update the running partial sum. Von Neumann architectures are infeasible candidates for running edge computing DNNs because these architectures maximize the energy cost per memory read by separating the processing units from the memory. For example, AlexNet (724 million parameters) requires nearly 3 million reads from external memory, which consumes significantly more energy than what small edge devices can source [41].

Although edge computing helps to minimize undesired latency effects and enables real-time data processing, practically deploying DNNs to the edge requires minimizing the amount of energy consumed while retrieving parameters and activations from memory [30]. This is a challenging problem, and the following section describes the methods that are currently used to minimize DNN memory access costs.

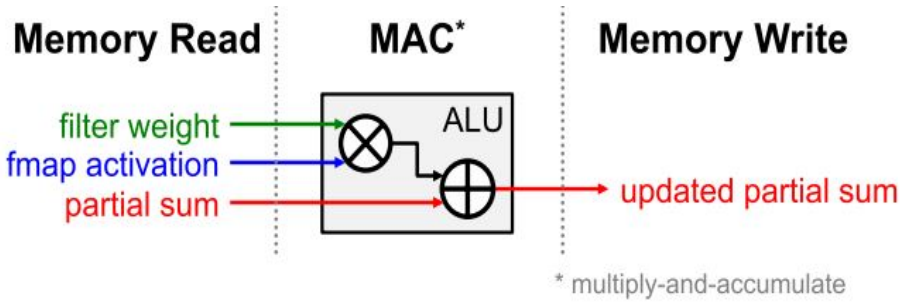


Figure 2-2: Memory access challenges association with DNN inference. Each MAC operation requires four different memory operations: three reads from memory and one write to memory [30].

2.3 Minimizing the Energy Cost of Memory Access: Current Methods

Three common methods used to solve the memory access problem are (1) data flow optimization, (2) co-location of memory and computation, and (3) model-based energy optimization. Data flow optimization distributes memory from off-chip dynamic random access memory (DRAM) to on-chip processing units that reuse weights and activations as many times as possible. In contrast, co-location of memory and computation tries to physically place the memory on-chip, while model-based energy optimization modifies a DNN's computational structure in order to minimize its energy consumption cost. Each method is described in greater detail below.

2.3.1 Method 1: Data Flow Optimization

The data flow optimization approach uses a memory hierarchy that decentralizes off-chip DRAM and distributes it locally to the on-chip units that perform computation, which are called processing engines (PEs). As shown in figure 2-3, a memory hierarchy helps optimize hardware's energy efficiency by allowing local data access within each PE register file (RF). This means that processors don't have to go to off-chip DRAM to perform MAC computations. As seen in figure 2-3, utilizing local memory significantly decreases the energy costs of accessing weights and activations. For example, accessing

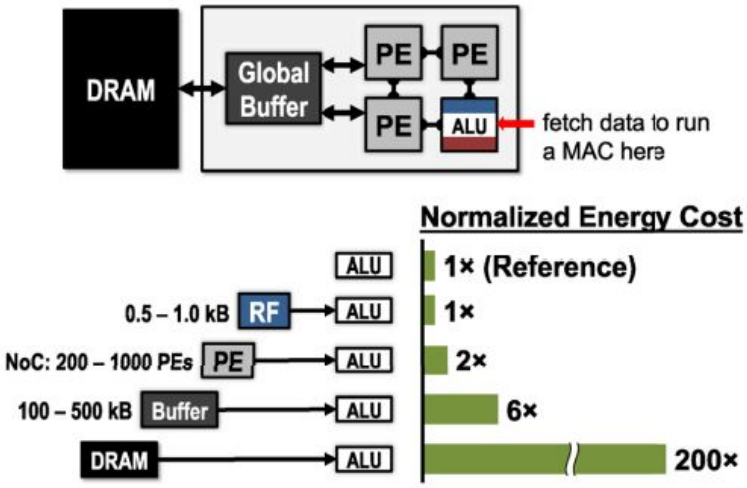


Figure 2-3: Memory access hierarchy used to improve energy efficiency in DNN hardware. This hierarchy enhances energy efficiency by distributing memory from DRAM to the processing engines that perform the computation. The bottom part of the figure shows the normalized energy costs of accessing memory from different locations [30].

memory locally from a PE's register file consumes 200 times less energy than using off-chip DRAM.

In order to take full advantage of locally distributed memory, it is necessary to maximize reuse in both network parameters and activations while they reside in local memory close to the processing units. This way, low-cost memory is accessed many times while high-cost DRAM is accessed as few times as possible. Since local memories have limited capacity, it is important to organize the order of computations (the data flow) to maximize local memory reuse. Three memory reuse techniques commonly used in neural networks are convolutional reuse, feature map (Fmap) reuse, and filter reuse. These techniques are illustrated in figure 2-4

As shown in the figure, convolutional reuse sweeps the same filter over the same input, so that both the filter and the input are reused over multiple different MACs. Fmap reuse applies multiple different filters to the same input and thus only reuses the input feature map. Finally, filter reuse is similar to convolutional reuse except the filter does not move in the lateral dimension of each input but rather is applied

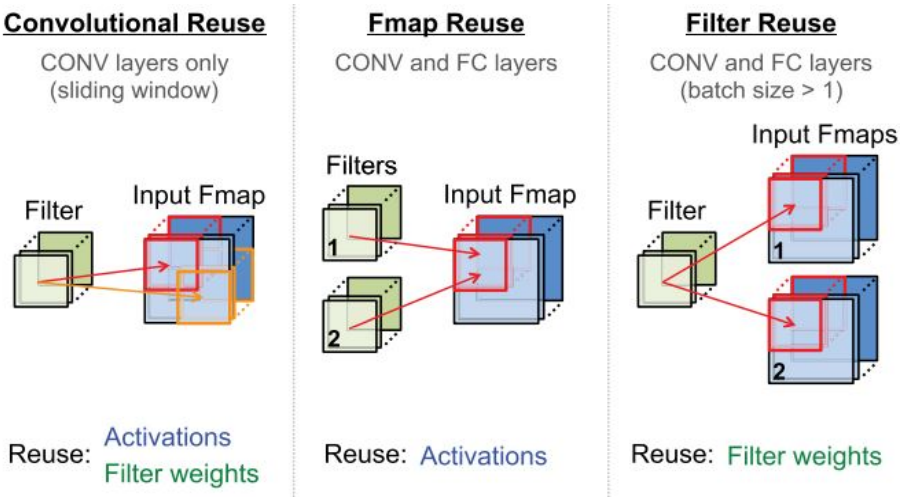


Figure 2-4: Three commonly used methods for data reuse in neural networks. Convolutional reuse slides the same filter over multiple subregions of the input. Fmap reuse applies the same input to multiple filters, while filter reuse applies the same filter to multiple inputs [30].

to one lateral location. This reuses the filter weights since the same filter is applied to all inputs in the batch tensor. The next four sections describe four common data flow optimization techniques in current work that leverage data reuse: (1) weight stationary, (2) output stationary, (3) row stationary, and (4) no-local reuse.

Weight Stationary

The weight stationary data flow maximizes filter reuse by (1) storing weight elements locally within PE register files and then (2) mapping all MACs that use the same weight element to the correct PE where that weight lives [84]. A high-level overview of a weight stationary data flow is given in figure 2-5 where the weight elements $W_0 ... W_7$ are stored locally in the PE register files while the input activations are broadcast to each PE. The partial products are computed within each PE locally and the partial sums are spatially accumulated over the PE array.

Figure B-2 in the appendix gives an example where a weight stationary data flow is used to perform a simple 1D convolution. Notice that each PE stores its assigned weight throughout the duration of the computation, which illustrates how the weight

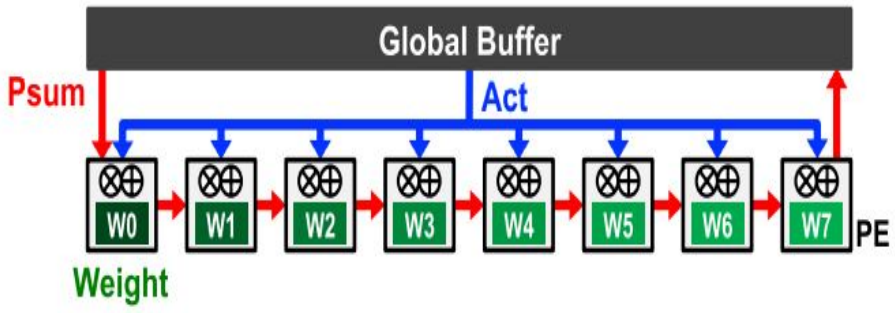


Figure 2-5: High-level architecture of the weight stationary data flow. The weight elements W_0 ... W_7 are loaded from external DRAM into the local register files of the PEs in the array. Then, the input elements (Act) are broadcast to each PE, and the partial products are computed. The partial sums (Psum) are spatially accumulated over the PE array to compute the final output [30].

stationary data flow leverages filter reuse in order to minimize expensive memory reads from DRAM.

Output Stationary

While weight stationary data flows minimize the energy cost of reading weights, output stationary data flows focus on minimizing the cost of reading and writing partial sums [43]. The high-level structure of the output stationary data flow is given in figure 2-6. Instead of storing weights locally in each PE register file, output stationary data flows store accumulated partial sums in the PE arrays and broadcast both the inputs and the weights. This means that each MAC operation will be able to read from and write to partial sums with minimal energy cost since the partial sums live in local memory.

Similar to the weight stationary case, figure B-3 illustrates an example where a 1D convolution is computed using an output stationary data flow. In this case, the three PEs store the intermediate partial sums while the weights and inputs move along the PE array in opposite directions. By keeping the partial sums in local memory, the output stationary data flow minimizes the energy needed to access partial sums during MAC operations.

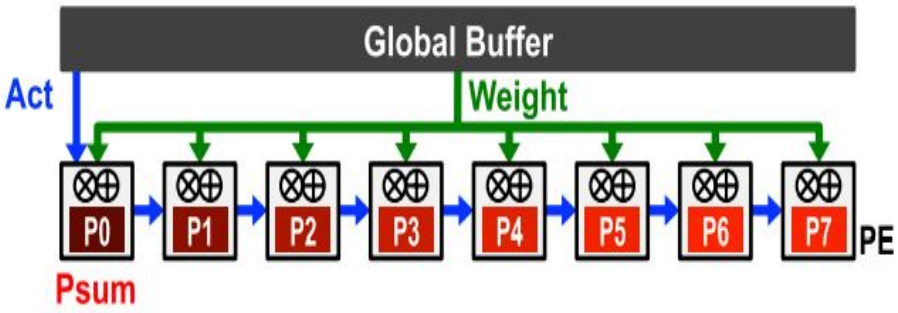


Figure 2-6: High-Level architecture for the output stationary data flow. Instead of locally storing weights, the partial sums are stored in each PE while the weights and inputs are broadcast to each PE in the array. The purpose of this data flow is to minimize the energy cost of reading and writing partial sums [30].

Row Stationary

While weight stationary and output stationary data flows are designed to minimize the energy costs of accessing weights and partial sums respectively, the row stationary data flow uses both filter reuse and convolutional reuse to optimize overall energy consumption for all data types (weights, input Fmaps, and partial sums) [85]. Row stationary data flows use 1D convolutions called row primitives to perform DNN computations. Figure 2-7 illustrates how a 1D convolutional primitive is computed. Note how the PE register file stores the weights (filter reuse) but also stores the inputs so that overlapping input elements are reused (convolutional reuse). The 1D convolutional primitives shown in figure 2-7 can be used with a 2D array of PEs to perform a 2D convolution that scales to higher dimensions as depicted in figure B-4 in the appendix.

No-Local Reuse

Weight stationary, output stationary, and row stationary data flows minimize memory-based energy costs by localizing memory within each PE register file. In contrast, the no-local reuse (NLR) data flow removes all local memory from PEs and instead puts all the on-chip memory into the global buffer [4]. While this data flow increases

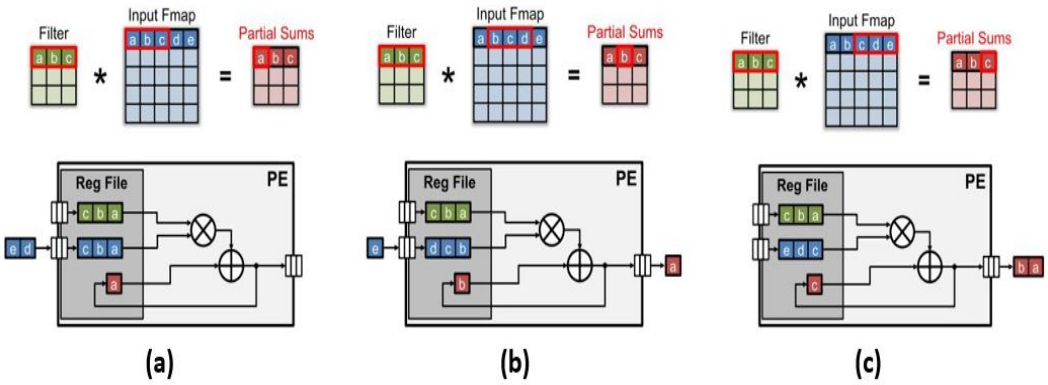


Figure 2-7: The 1D convolution primitive used in row stationary data flows. The weights are stored in the PE register file, while the inputs are streamed in and operated on. (a) time step 1, (b) time step 2, (c) time step 3 [30].

inter-PE data traffic, it enhances energy efficiency by drawing all memory access from the global buffer instead of DRAM. Looking back at figure 2-3 reveals that accessing the global buffer is about 33 times less energy expensive than accessing DRAM. Figure 2-8 shows how the no-local reuse data flow functions. Unlike figures 2-5 and 2-6, which depict the weight stationary and output stationary data flows, the NLR data flow stores no data whatsoever in the PE register files. This means that the global buffer must broadcast all the weights and inputs while accumulating the partial products over the PE array [30]. The motivation behind the NLR data flow is the fact that PE register files are inefficient when it comes to the required amount of on-chip area needed to store a given amount of data. It is much more efficient in terms of storage per chip area to access memory from the global buffer instead of using local PE memory [30].

2.3.2 Method 2: Co-Location of Memory and Computation

Since data movement from memory to chip is the primary obstacle facing energy-efficient DNN hardware, an alternative strategy to maximize energy efficiency is to integrate DRAM into the chip itself [30]. The two sections below describe two ways in which memory and computation can be co-located by bringing off-chip DRAM onto

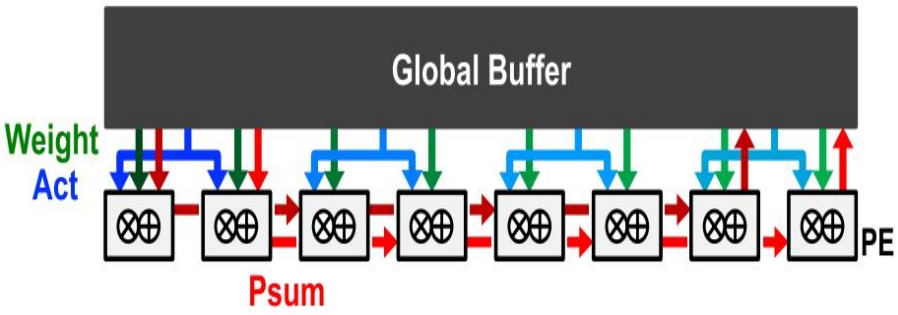


Figure 2-8: High-Level architecture of No-Local Reuse data flow. Instead of localizing memory within PE register files, NLR assigns all on-chip memory to the global buffer that broadcasts the weights and the inputs to each PE in the array [30].

the chip.

Embedded DRAM

Embedded DRAM (eDRAM) eliminates the need for any off-chip memory access by placing tens of megabytes of memory on-chip using electronic capacitors [30]. Since capacitors are often leaky, periodic refreshes are needed in order to prevent data decay [65]. Using eDRAM enhances DNN hardware energy efficiency because accessing on-chip eDRAM is 221 times less expensive than accessing off-chip memory [34]. However, the drawback of this approach is that eDRAM has a lower memory density (Megabytes/mm^2) as compared to off-chip memory. This lower memory density can result in higher chip financial costs since the chip area needed for a given amount of memory will be greater for eDRAM than for off-chip DRAM [30]. In addition, eDRAM requires standby power to facilitate the periodic refreshes to ensure that data doesn't leak from the on-chip capacitors [77].

Resistive Memory

Another way to co-locate memory with computation is to use resistive memory as illustrated in figure B-5 in the appendix. Resistive elements encode the weights as conductance values (G), while the activations are encoded by the voltage drops

over the resistors (V). MAC operations are performed using $I{=}VG$ and applying Kirchhoff's current law as seen in figure B-5. Here, I is the current leaving a resistor that encodes a partial product. Unlike eDRAM, resistive memory is much higher density and is comparable to DRAM in terms of the amount of memory per chip area [12]. Despite this advantage, resistive memories suffer from several drawbacks. In addition to suffering from low precision in floating point operations, resistive memories also require integrated digital-analog converters (DACs) and analog-digital converters (ADCs), which can consume a lot of energy [57]. In addition, the voltage drop across the resistive elements can degrade the accuracy of memory reads, while the amount of energy needed to write data to the resistive elements can be costly [57].

2.3.3 Method 3: Model-Based Energy Optimization

The previous section described common methods that are used to minimize energy costs by placing the compute and memory in proximity to each other. While this strategy has proven effective in various DNN hardware architectures, another common way of minimizing the energy cost of memory access is to alter the way in which DNN computations are performed. These strategies are discussed in the following sections.

Network Pruning

Originally proposed in 1989, network pruning utilizes the fact that most DNN models tend to be overparameterized in order to make the training process easier [82]. Increasing the number of DNN parameters makes the loss function more convex so that optimal accuracy values are easier to reach during training. This means that a trained DNN will likely have a large number of weights that have a negligible impact (salience) on the network accuracy. In order to reduce the number of model parameters while preserving accuracy, network pruning removes low-salience weights and then fine-tunes the remaining weights. Although this approach is often infeasible in large DNNs, a variation of this pruning method simply removes low-magnitude weights [58]. As an example, using network pruning can reduce the number of weights

in AlexNet by a factor of 9, which reduces the number of MACs by a factor of 3 [30]. Another commonly used pruning method is called structured pruning where entire groups of low-salience weights are removed [73]. Reducing the number of weights in a network means that fewer MACs need to be performed to run inference. Lowering the number of MACs needed, in turn, decreases the energy cost of running a DNN. As described in [71], figure B-6 shows the decrease in energy costs of inference for three common neural networks (GoogLeNet, SqueezeNet, and AlexNet) as a result of pruning.

Knowledge Distillation

Instead of pruning parameters directly from a large DNN model, a smaller DNN can be trained to mimic the output of its larger counterpart through knowledge distillation [20]. The idea here is to pre-train an over-parameterized DNN to output some level of accuracy for a given task. Then, a smaller DNN called the student network can be trained to match the output of the larger teacher network. This method allows the student network to achieve an accuracy level that would have been unattainable had the student been directly trained on the same data set as the teacher. This process is illustrated in figure B-7 in the appendix. Note that the student network is trained to match the raw, unnormalized output scores instead of the normalized class probabilities. This is because Softmax removes information in low-probability and high-probability class scores by driving them to zero and one respectively [35].

Exploiting Sparsity

Another way to increase energy efficiency in DNN hardware is to exploit the sparsity that arises from nonlinearity in the intermediate output activations. For example, the RELU nonlinearity zeros out all negative elements of an input, which results in a sparse matrix as depicted in figure B-8. As noted by Chen et al., DNN hardware can be configured to skip any MACs that map to zero-valued activations, which can nearly halve the amount of energy needed for a DNN to run inference [81].

2.4 Current Work: Hardware Architectures That Minimize Energy Access Costs

The architectures listed in this section describe how the techniques from 2.3.1, 2.3.2, and 2.3.3 are implemented in hardware in order to reduce the energy costs of running DNNs.

2.4.1 Eyeriss

The authors of [81] developed an architecture called Eyeriss that is based on a row-stationary data flow. Eyeriss uses a 1D convolutional primitive where each convolution is mapped to a single processing unit in a PE array. This means that each PE operates on a single filter row and a single input feature map row. 2D convolutions are composed from repeated 1D convolutions using the procedure previously outlined in figure B-4 [81] . Eyeriss employs convolutional reuse since both the filter elements and the input feature map elements are reused. Eyeriss also uses filter reuse since each filter element is repeated over the entire N>1 batch size [81].

2.4.2 MobileNet

The authors in [14] used convolutional factorization to minimize DRAM energy costs by minimizing the number of MACs needed to perform the convolution operation. Figure B-9 depicts how the standard convolution operation is factored into two parts: depthwise convolution that operates on the individual channels followed by a point-wise convolution that produces the final output through a linear combination of the convolved channels. Figure B-10 gives an architectural description of the MobileNet structure, while figure B-11 shows the number of MAC operations used in MobileNet compared to another DNN used for the same application called FaceNet. As can be seen in figure B-11, the use of depthwise separable convolutions significantly reduces the number of MACs required to implement convolution. By decreasing the required number of MACs, MobileNet decreases the energy costs of implementing

convolutional DNNs in hardware.

2.4.3 SqueezeNet

In [18], the authors developed an architecture called SqueezeNet to run inference through AlexNet with 50 times less parameters while maintaining stable output accuracy. The SqueezeNet architecture is based on a series of 1x1 and 3x3 convolutions called the fire module, which is depicted in figure B-12. As shown in the figure, three hyper-parameters define the structure of each fire module: s_{1x1} (the number of 1x1 convolution kernels in the squeeze layer), e_{1x1} (the number of 1x1 convolution kernels in the expansion layer), and e_{3x3} (the number of 3x3 convolution kernels in the expansion layer). As noted by the authors, SqueezeNet decreases the number of parameters needed to run inference by replacing standard convolutional layers with fire modules and decreasing the number of channels going into the 3x3 filters [18]. Using the fire module as a computational primitive, SqueezeNet employs the high-level architecture depicted in figure B-13. As shown in the figure, a forward pass through SqueezeNet consists of a standard convolution followed by a series of fire modules and max pooling for downsampling. The forward pass ends with a standard convolution and a Softmax activation layer. Figure B-14 compares the magnitude of model compression achieved by SqueezeNet versus the amount of model compression achieved by other comparable methods. Note how SqueezeNet is able to compress the AlexNet architecture much more than comparable works while suffering minimal losses in accuracy.

2.5 Chapter Conclusion and Summary

This chapter began by offering a high-level introduction to the concept of edge computing- including its basic structure, motivation, and benefits. Section 2.2 described the benefits and challenges of applying DNNs to edge computing applications, and it was noted that the primary obstacle facing energy-efficient DNNs is the energy cost of memory access. In section 2.3, three common approaches for optimizing en-

ergy efficiency were analyzed: (1) data flow optimization, (2) co-location of memory and computation, and (3) model-based energy optimization. Finally, in section 2.4, current hardware architectures that use the above techniques were listed and analyzed. While the architectures thus far show improvement in energy efficiency, they are not optimized for the task of edge computing because they require local storage of the DNN weight matrices on or near edge devices. Chapter 3 introduces the netcast architecture, which leverages optical parallelism to minimize energy costs and eliminates the need to store weight matrices on edge devices.

Chapter 3

Simulating the Netcast Optical Neural Network (ONN)

Having introduced DNNs in chapter 1 as well as currently used DNN hardware in chapter 2, this chapter focuses on an optically based edge computing architecture called netcast. The netcast optical neural network (ONN) leverages parallelism in both the time domain as well as the frequency domain to minimize energy costs. Netcast also avoids the need for edge processors to store large DNN weight matrices on SWaP-limited hardware. Section 3.1 below gives a broad overview of the netcast server-client protocol where weight elements are encoded at the server and transmitted to the client. Section 3.2 discusses the three predominant sources of error in the netcast hardware: thermal noise, shot noise, and calibration error. Section 3.3 describes the advantages of using netcast for edge computing compared to digital electronics. Before detailing the techniques used to simulate netcast in software, section 3.4 defines the figure of merit that will be used to compare netcast to digital networks: output accuracy normalized by energy consumption. Section 3.5 describes how the netcast optical hardware is simulated using a computational technique called stacked convolution. Section 3.6 gives the simulation results for MNIST classification, while section 3.7 gives the simulation results for scene recognition. Finally, section 3.8 concludes with a short summary of the main points discussed throughout the chapter.

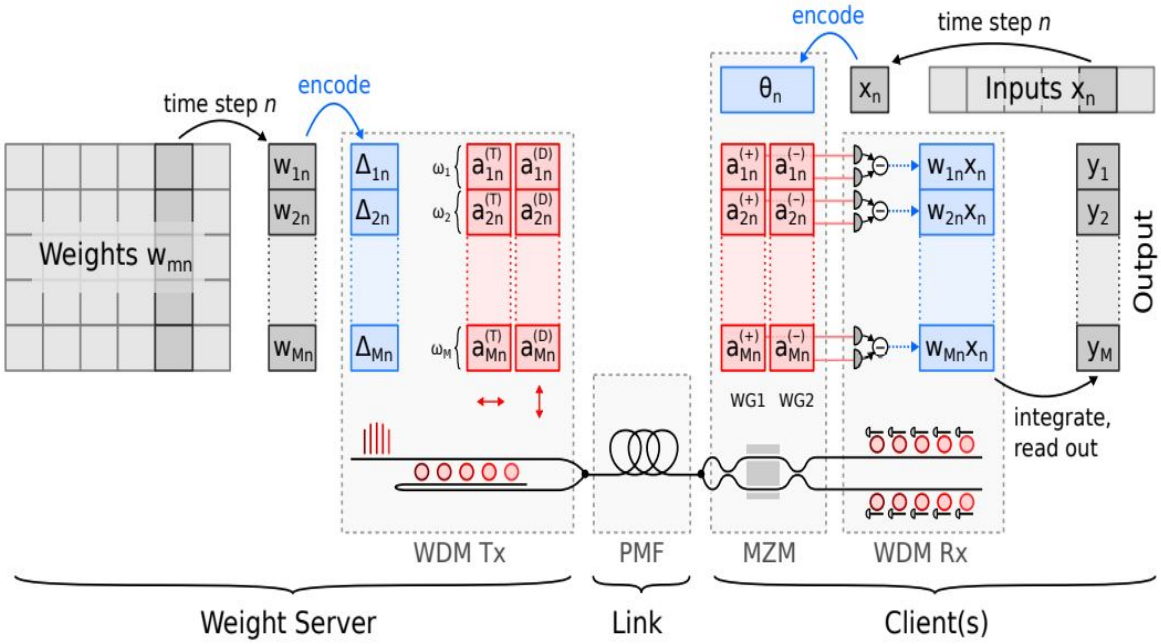


Figure 3-1: The netcast ONN architecture that uses time multiplexing and wavelength division multiplexing (WDM). The architecture performs a matrix-vector product over N time steps using M different wavelength channels [38].

3.1 Overview of the Netcast Optical Neural Network

Figure 3-1 shows an overview of the netcast architecture that leverages both time multiplexing as well as wavelength division multiplexing (WDM). During each time step (indexed by n), a column vector $w_{:,n}$ is extracted from the weight matrix, and each extracted weight element w_{mn} is mapped to a transmission coefficient and a reflection coefficient through a bank of M micro-ring modulators. The weight value w_{mn} is encoded by the detuning of the mth cavity at time step n: Δ_{mn}. At each micro-ring, the transmission and reflection coefficients are given by the following equations.

$$t_{mn} = \frac{i\Delta_{mn}}{i\Delta_{mn} + \kappa_{abs}/2} \tag{3.1}$$

$$r_{mn} = \frac{-\sqrt{\kappa_1\kappa_2}}{i\Delta_{mn} + \kappa_{abs}/2} \tag{3.2}$$

Where Δ_{mn} is the detuning of cavity m at time step n, while κ_1 and κ_2 are the ring coupling coefficients for the through (T) and drop (D) ports respectively. κ_{abs} is the

rate of absorption and scattering in the ring modulator. The notational convention used in this chapter is that N represents the number of time steps and M is the number of wavelengths, which equals the number of micro-ring modulators. Given the encoded transmission and reflection coefficients, the through-port and drop-port optical powers are the final outputs from the micro-ring modulators as given by the equations below.

$$a_{mn}^{(T)} = t_{mn} a_0 \tag{3.3}$$

$$a_{mn}^{(D)} = r_{mn} a_0 \tag{3.4}$$

In these equations, the known incident optical power is a_0, T refers to the through-port, and D refers to the drop-port. The through and drop-port output signals from the modulator are then fed through a polarizing beamsplitter (PBS) that maps to orthogonal polarizations on a polarization maintaining fiber (PMF). Then, the M wavelength channels are wavelength division multiplexed and transmitted from the server to the client over the optical fiber. Once at the client, another polarizing beamsplitter separates out the through-port and drop-port signals, while a phase shifter is used to correct for the relative phase differences induced by the server $\rightarrow$ client transmission. The signals $a_{mn}^{(T)}$ and $a_{mn}^{(D)}$ enter a Mach Zehnder Modulator (MZM) that encodes the activation at time n (x_n) as a voltage θ_n. The equation below gives the MZM output.

$$\begin{bmatrix} a_{mn}^{(+)} \\ a_{mn}^{(-)} \end{bmatrix} = \begin{bmatrix} \cos\theta_n & -\sin\theta_n \\ \sin\theta_n & -\cos\theta_n \end{bmatrix} \begin{bmatrix} a_{mn}^{(T)} \\ a_{mn}^{(D)} \end{bmatrix} \tag{3.5}$$

Finally, the M different wavelength channels are demultiplexed and fed into a bank of M photo-detectors, each of which maps input optical powers $a_{mn}^{(+)}$ and $a_{mn}^{(-)}$ to current values $I_{mn(+)}$ and $I_{mn(-)}$. The photocurrent difference at each wavelength channel

encodes the partial product $w_{mn}x_n$ as shown in the equations below [38].

$$\Delta I_{mn} = I_{mn(+)} - I_{mn(-)} \tag{3.6}$$

$$\Delta I_{mn} = |a_{mn}^{(+)}|^2 - |a_{mn}^{(-)}|^2 \tag{3.7}$$

$$\Delta I_{mn} = [|t_{mn}\cos\theta_n - r_{mn}\sin\theta_n|^2 - |t_{mn}\sin\theta_n + r_{mn}\cos\theta_n|^2]|a_0|^2 \tag{3.8}$$

$$\Delta I_{mn} = [(|t_{mn}|^2 - |r_{mn}|^2)\cos 2\theta_n - Re[t_{mn}^* r_{mn}]\sin 2\theta_n]|a_0|^2 \tag{3.9}$$

$$\Delta I_{mn} = [(|t_{mn}|^2 - |r_{mn}|^2)\cos 2\theta_n]|a_0|^2 \tag{3.10}$$

Where the last line follows from the assumption that the two MZM output optical fields, $a_{mn}^{(+)}$ and $a_{mn}^{(-)}$, have an absolute phase difference of $\pi/2$ [38]. Then, the weight element w_{mn} can encoded by the term $(|t_{mn}|^2 - |r_{mn}|^2)$, while the input activation value can be encoded by $\cos 2\theta_n$. This means that equation 3.10 can be rewritten as the desired scalar product.

$$\frac{\Delta I_{mn}}{|a_0|^2} = w_{mn}x_n \tag{3.11}$$

The above partial product is the result of computation at time step n. In order to form the desired matrix-vector product, a time integrator is used to obtain the final output at the client, which is given by the equation below.

$$\sum_n \frac{\Delta I_{mn}}{|a_0|^2} = \sum_n w_{mn}x_n \tag{3.12}$$

Using a smart transceiver to encode weight matrix column vectors over M different wavelengths in conjunction with a time integrator that adds over N time steps, the netcast hardware has demonstrated a significant reduction in the energy cost of inference [13]. The hardware is able to perform computation at a total (client) energy cost of about 10 fJ/MAC, which is three orders of magnitude lower than what is possible with digital electronics, including those mentioned in section 2.4 [13].

While netcast's ability to minimize energy costs has been experimentally shown, this alone does not demonstrate its practical usefulness in edge computing tasks. In

order to assess netcast's usefulness in edge computing, the ONN hardware must be accurately simulated in software and compared to digital DNNs using some figure of merit (FOM) that takes into account both network accuracy as well as energy costs. Before describing the FOM and simulation details, however, it is first necessary to understand netcast's sources of error as well as how netcast is able to outperform digital electronics by over three orders of magnitude. Both of these topics are discussed in the sections below.

3.2 Noise Sources in Netcast

At a high-level, netcast is able to perform computations more efficiently than digital electronics by effectively mapping float point MAC operations onto optical hardware. Because netcast uses optical and electronic hardware, several noise sources must be considered and accounted for. These noise sources include both fundamental noise (shot noise and thermal noise) as well as hardware-based error (calibration error). The importance of these error sources as well as their mathematical formulations are given in the sections below.

3.2.1 The Importance of Noise Sources

Before getting into the details of noise sources in netcast, it is first important to clearly establish why such sources are important. In order for any computing hardware (including netcast) to output reasonably accurate results, the signals traveling through the hardware must have an adequately high signal-to-noise ratio (SNR). Given some signal, s, to be measured that is corrupted by noise, n, the SNR is simply the ratio $\frac{Power(s)}{Power(n)}$. A low SNR means that the signal being used for computation is effectively lost in the background noise, which results in highly inaccurate computing architectures. In contrast, large SNRs yield more accurate and useful computing systems because the signal can be more easily distinguished from the interfering noise. As noted by [13], DNN hardware SNR values typically lie in the range 10-100, which means that the netcast SNR must lie in this range in order for the ONN to oper-

ate accurately. The challenge is that various noise sources (both fundamental and hardware-based) work to decrease the SNR and thus negatively impact netcast's output accuracy. Having motivated the importance of accounting for noise sources, the following sections discuss each noise source in greater detail.

3.2.2 Thermal Noise

Thermal noise originates from the fact that random thermal excitations in charge carriers give rise to a small but measurable current, which manifests itself as broadband white noise [46]. In order to see how the netcast hardware can be used to minimize the effects of thermal noise, it is first necessary to write down an equation that describes thermal noise. While many different expressions are used to define thermal noise in different units, one of the more intuitive expressions measures thermal noise in units of electrons using the equation below.

$$\sigma_{thermal} = \frac{\sqrt{kTC}}{q} \tag{3.13}$$

Intuitively, equation 3.13 describes the fluctuation in the number of electrons that are read out from the time integrator and subsequently digitized. In the equation, k is the Boltzmann constant, T is the temperature, C is the capacitance of the electronic device, and q is the fundamental electron charge.

Using the equation for thermal noise, the netcast SNR can be computed assuming that thermal noise dominates. This computation will give a picture of how the thermal noise negatively affects netcast's ability to compute accurately. Assume that the energy consumed by a single MAC operation is E_{mac}, and suppose further that the number of time steps needed to digitize and read out MAC results is N. Then, let η represent the quantum efficiency of the photodiodes used at the time integrator. Here, quantum efficiency describes the number of electrons emitted from an incident group of photons at a photodiode. For ideal systems, $\eta = 1$, meaning that each incident photon on the photodetector results in the emission of a single electron. Finally, let q be the fundamental electron charge as stated earlier, and let the quantum energy

of a single photon be given by the Plank law: $E = hv$. Calculating the signal from the above variables is then a simple matter of unit conversion from energy ($E_{mac}N$) to number of photons ($E_{mac}N/hv$) to number of electrons ($\frac{E_{mac}N}{hv}\eta$). From this, the SNR (assuming that thermal noise dominates the system) is given by equation 3.14 below.

$$SNR = \eta \frac{E_{mac}Nq}{hv\sqrt{kTC}} \tag{3.14}$$

As shown in the equation, increasing the amount of thermal noise in the system decreases the SNR by scaling the magnitude of the denominator term.

3.2.3 Shot Noise

Shot noise is fundamental in all optical systems, meaning that shot noise errors are inevitable and cannot be eliminated even in the most well calibrated hardware. As stated by [46], shot noise originates from the fact that quantized photons in a light beam do not arrive at evenly-distributed points in time. Rather, the distribution of photon arrival times at a fixed point is described by the Poisson distribution with a probability mass function given by the equation below.

$$P(q) = \frac{exp(-n_p)n_p^q}{q!} \tag{3.15}$$

Where n_p is the mean. Because the mean of the Poisson distribution equals its variance, the SNR can be expressed simply as $\sqrt{n_p}$, as noted by [46]. In terms of the number of photons per MAC, the shot noise limit for netcast exists at approximately 1 photon/MAC [13]. As noted by [13], the hardware optimization that can be used to amortize thermal noise (increasing N) cannot be applied to mitigate shot noise, because shot noise is physics-limited.

3.2.4 Calibration Errors

While the above fundamental noise sources (thermal and shot) decrease netcast's accuracy by diminishing the SNR, calibration errors also impact the final output accuracy of the network. Calibration errors occur because the hardware mapping between floating point values and optical intensities has a certain amount of error associated with it. As detailed in [13], netcast uses an encoding function to map floating point values to optical intensities that are then mapped to voltage values and finally back to floating point outputs. The float to intensity mapping and intensity to voltage mapping are performed using linear interpolation and a third-order polynomial fit respectively, both of which introduce an amount of calibration error. To measure the amount of calibration error that exists in the system experimentally, figure 3-2 was generated by performing 100,000 scalar-scalar multiplications and plotting the floating point versus optical results [13]. A perfect calibration would be represented in the figure as an identity function where the optical products exactly equal the floating point results. As seen in the figure, netcast's experimental scalar products don't exactly follow an identity function (although they are close), which gives rise to the scalar product error distribution depicted in the bottom panel of figure 3-2. This scalar product error distribution means that, for any MAC operation performed by netcast, there will be some error value that arises from mapping floating point computation onto optical hardware. This per-product calibration error motivates the simulation of netcast in software as described in section 3.5.

3.3 Advantages of Netcast Compared to Digital Electronics

As noted by [13], netcast is designed to minimize the energy consumption at the client in two ways: (1) eliminating weight data movement on chip and (2) exploiting optical parallelism in both the time domain as well as the frequency domain. Each of these benefits is discussed in greater detail in the two sections below.

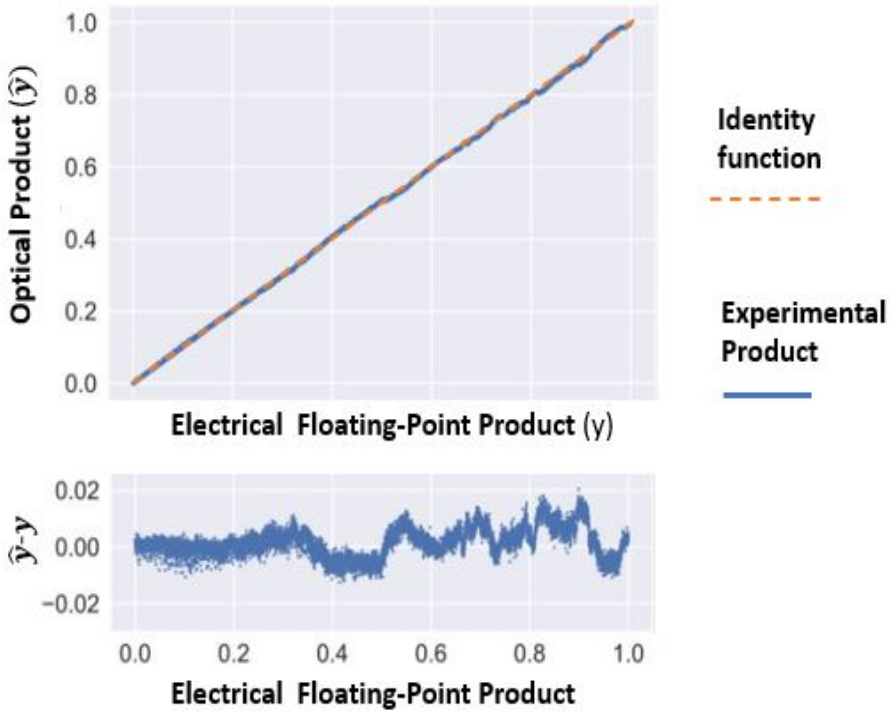


Figure 3-2: Calibration error from netcast [13].

3.3.1 Eliminating On-Chip Weight Data Movement

As demonstrated in chapter 2, accessing weight elements on a chip imposes a prohibitively large energy cost that many edge devices are unable to facilitate. One of netcast's key innovations is that, instead of storing large-memory DNN weights at the client, the weights are optically broadcast to the edge device from a server. In order to see how eliminating on-chip weight storage significantly lowers energy consumption, it is helpful to use figure 3-1 and determine the energy consumption requirements needed to operate the client. The client contains four hardware components that consume energy. First, a digital-to-analog converter (DAC) is used to encode the input activations as shown by the "encode" arrow in figure 3-1. Second, a Lithium Niobate modulator (labeled MZM in figure 3-1) is used to modulate the received weight elements with the encoded input activations. Third, a bank of time integrators is used to accumulate the partial products at the "integrate and read out" arrow of figure 3-1. Finally, an analog-to-digital converter (ADC) is used to

digitize the output of the time integrators. The key point to note here is that all of the client components that consume energy are being used for computation in the optical domain (the MZM), computation in the electronic domain (time integrator), or data encoding and reading (DAC and ADC). In terms of memory, no resources are needed for weight storage - only activation storage. This is a significant improvement because activations require significantly less memory to store than weight parameters [38]. Eliminating the need to store DNN weights on-chip helps netcast operate at 1000 times less energy than its digital electronic counterparts. The calculations that support this are given in section 3.3.2 below.

3.3.2 Exploiting Optical Parallelism

Offloading weight storage from the client to the server helps netcast operate in an energy regime that is three orders of magnitude below current digital electronics, which currently operate at 1 pJ/MAC [13]. The goal of this section is to calculate netcast's energy consumption from the four client devices listed in section 3.3.1. First, consider the DAC that encodes the input activations to be modulated with the weight values. Encoding a single activation element x_j at time j requires 1 pJ of energy, which is expensive. However, the energy cost per MAC can be scaled down dramatically by exploiting the parallel nature of optical signals. Specifically, the DAC output is fanned out to M different optical channels where M is the number of optical wavelengths that the system uses. This means that, for an energy cost of 1 pJ, the DAC is able to facilitate M MAC operations in parallel, which scales down the per-MAC energy cost to $(1/M)$ pJ [13]. Similar to the DAC, the client MZM also requires 1 pJ of energy to perform M MACs in parallel for an energy/MAC cost of only $(1/M)$ pJ [13]. Just as the DAC and MZM exploit M-way optical fan-out in the frequency domain, so also the time integrators and ADC are able to exploit fan-out in the time domain. One key note is that using time integration means that the integrators and ADC only need to read out and digitize the computation results after N time steps [13]. Considering the time integrator energy consumption, this means that each of the M integrators performs N MACs over N time steps, where each integrator requires 1 fJ

of energy to perform integration. Thus, the M time integrators, after N time steps have performed NM MACs and consumed M fJs of energy, for a per-MAC energy cost of $(1/N)$ fJ. Finally, the ADC requires 1 pJ of energy per read but only needs to read once for every N time steps and therefore N MACs of computation. Similar to the time integrators, the ADC energy cost per MAC is $(1/N)$ pJ [13]. Using the per-component energy costs allows us to write down the total energy consumption for netcast using equation 3.16.

$$E_{mac}^{(total)} = E_{mac}^{(MZM)} + E_{mac}^{(DAC)} + E_{mac}^{(Integrator)} + E_{mac}^{(ADC)} = (\frac{1}{M})pJ + (\frac{1}{M})pJ + (\frac{1}{N})fJ + (\frac{1}{N})pJ \tag{3.16}$$

Following the example of [13], realistic near-term values for N and M are $N = M = 100$, which yields a total energy consumption per MAC of $E_{mac}^{(total)} \approx 10fJ/MAC$, which is three orders of magnitude lower than digital electronics. The energy consumption calculations for netcast illustrate how optical parallelism in both the frequency domain (M) as well as the time domain (N) allows netcast to downscale its energy cost to levels that would be otherwise unachievable. Before moving on, it is important to note that optical parallelism not only minimizes netcast's energy consumption, but also minimizes the level of thermal noise that affects the output accuracy. This can be clearly seen in equation 3.14 where the SNR is given as a function of the number of time steps used for time integration, N. Recall that modern computing architectures require a SNR value of somewhere between 10-100 in order to operate with a useful level of accuracy. Here, the netcast hardware performs at 98 percent accuracy with a SNR of 20 [13]. Using time integration allows netcast to leverage parallelism in the time domain where the value of N can be scaled to boost the SNR to the desired level. Thus, as seen in equations 3.14 and 3.16, netcast exploits parallelism in the frequency and time domains to both amortize the energy cost of inference as well as achieve a high SNR.

3.4 Figure of Merit to Compare Netcast vs. Digital Networks: Energy-Normalized Accuracy (ENA)

Before going into further detail regarding the netcast simulations, it is important to clearly define the FOM that will be used to compare the results of netcast with those of digital DNNs. While many machine learning papers only look at network accuracy, this valuation of a network's quality completely ignores the energy cost needed to actually implement that network in hardware. Even the most accurate network may not be practically useful if it is prohibitively expensive to run. The importance of energy cost as a FOM is especially important in edge computing because the ability to deploy neural networks to the edge assumes that the network in question is energy-efficient enough to satisfy size, weight, and power (SWaP) constraints. Because both energy efficiency and network accuracy should be jointly optimized for edge computing applications, the FOM used in this thesis is the network accuracy normalized by the energy cost of inference. This FOM, called energy normalized accuracy (ENA), is given by the equation below.

$$ENA(A, E) = \frac{A}{E} \tag{3.17}$$

Where the accuracy A is a scalar value in the range $[0, 1]$, and E is the energy cost of running a single forward pass through the DNN. With this FOM in mind, the netcast optical hardware can now be simulated in software to compare netcast's energy normalized accuracy to that of digital networks. Netcast simulations are performed for both MNIST digit classification and scene recognition. Section 3.6 details the MNIST classification task, while section 3.7 describes the scene recognition task. Before describing the task-specific results and comparisons, however, it is first necessary to understand how the netcast hardware can be simulated. Both MNIST classification and scene recognition use the same method, which propagates the per-product calibration errors onto an arbitrary network's final classification output using stacked convolution. This method is described in section 3.5 below.

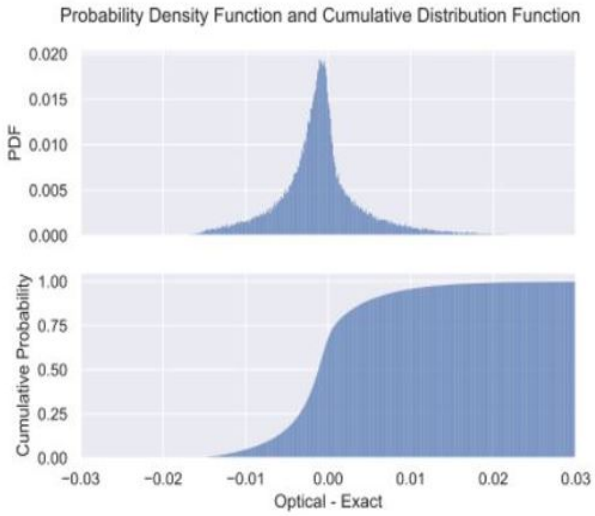


Figure 3-3: Netcast error distribution for partial products of the form $w_{mn}x_n$ [13].

3.5 Simulating Netcast: Theory and Methods

As described in section 3.2, calibration error introduces a per-product error distribution into the netcast computations. Figure 3-3 takes the error elements from figure 3-2 and depicts them as a probability density function (PDF) and cumulative distribution function (CDF) of the (optical-target) error. This partial product error is helpful because it gives a sense of how accurate netcast can be on machine learning tasks, which all consist of MAC operations. However, knowing the distribution of partial product errors by itself does not give clear evidence that the netcast hardware outperforms digital networks in terms of energy normalized accuracy (equation 3.17). While running netcast directly on large DNNs may seem like a straightforward solution to this issue, the bandwidth limitations inherent in lab equipment makes this method infeasible in terms of runtime. In order to directly compare netcast's accuracy and energy consumption to that of currently used DNNs, two quantities must be determined. First, the partial product errors from figure 3-3 must be propagated through a DNN architecture in order to simulate how accurately netcast would perform computations within that architecture. Second, the energy consumption of netcast must be computed for the chosen DNN architecture. These two quantities can then be used to compute netcast's ENA, which can then be compared to the

average ENA of DNNs run using digital electronics.

3.5.1 Netcast Activation and Weight Mapping in Software

Netcast maps floating point operations to optical intensities through a series of encoders and decoders illustrated in the left panel of figure 3-4. Floating point values are normalized to the range $[0, 1]$ and mapped to optical intensities through the transmission and reflection coefficients as given in equations 3.1, 3.2, 3.3, and 3.4. The optical intensities are then mapped to voltage values at the output of the photodetector (PD), which can then be converted back into floating point values through re-scaling by the appropriate multiplication factor. The float $\rightarrow$ intensity $\rightarrow$ voltage $\rightarrow$ float mapping that occurs in the netcast hardware can be simulated in software using the fact that netcast downscales its computations to the $[0, 1]$ range, performs the MAC operations, and then re-scales back to the appropriate range. The float mapping for both the activation value (x) as well as the weight value (w) is illustrated in the right side of figure 3-4. Given some x in the range $[0, b]$ and some w in the range $[c, d]$, the activation is scaled to the range $[0, 1]$, while the weight is scaled to the range $[-1, 1]$ and factored into the w_+ and w_- terms that are both in the desired range of $[0, 1]$. It is safe to assume that the activation values are greater than or equal to zero because the input to hidden layers in the architectures considered in this thesis use RELU, which zeros out all negative activation values. In addition, the two data sets used to simulate netcast both lie in the range $[0, b]$, so the first DNN layer will see activation values in that range. Multiplying $x \ \epsilon \ [0, 1]$ and $w \ \epsilon \ [0, 1]$ yields the product $xw \ \epsilon \ [0, 1]$, which can then be upscaled to the original range $[0, b]$. Having established the parallel between netcast's mapping in hardware and the corresponding mapping in software, section 3.5.2 below gives a high-level description of the methodology used to simulate netcast.

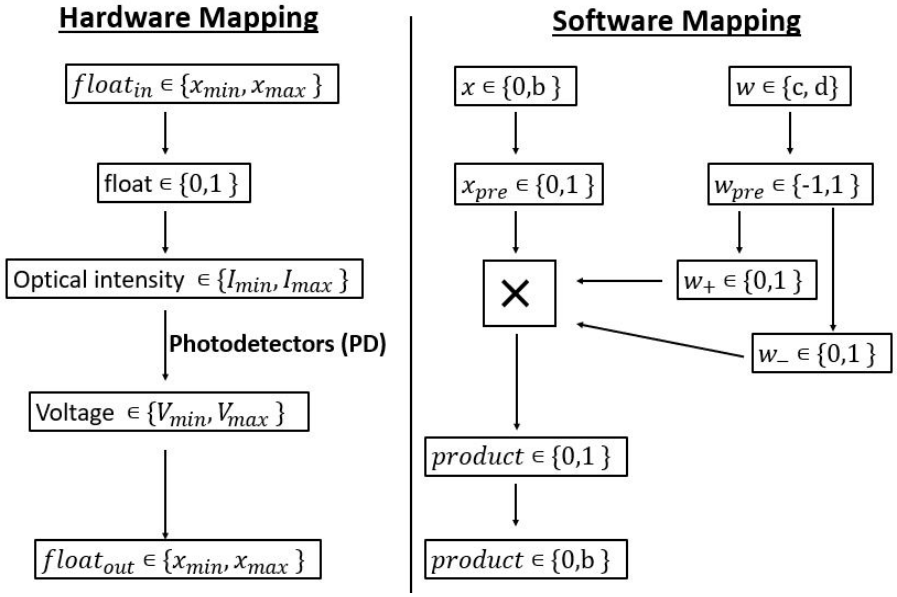


Figure 3-4: A side-by-side comparison between the float → voltage mapping that occurs in the netcast hardware and the corresponding input and weight mappings that occur in the simulated netcast software. The hardware mapping consists of four steps: re-scaling the input float to the range [0,1], mapping the scaled float value to an optical intensity defined on the range $[I_{min},I_{max}]$, converting the optical intensity to a voltage, and converting the voltage value to an output float value. In order to accurately simulate netcast in software, the following mapping is performed. First, the input activation is scaled to the range [0,1], while the weight matrix is scaled to the range [-1,1]. Next, the weights are factored into W_+ and W_- matrices that are both in the range [0,1]. The MAC operation between the activation and weight values is computed with the added error elements and scaled to the same range as the input activations.

3.5.2 Simulating Netcast: A High-Level Overview

With the product error distribution in hand along with a method that simulates optical computations in netcast, the partial product errors can now be directly integrated into each MAC operation that netcast performs. The way to accomplish this is given by the simple expression below.

$$xw \rightarrow xw + \Delta \tag{3.18}$$

This expression means that simulating netcast on a DNN workload requires sampling from the netcast error distribution (figure 3-3) to obtain Δ and then perturbing the partial product xw. Sampled error values must be added to each partial product in each MAC that occurs within the DNN. This strategy can be applied to both fully connected and convolutional networks. Figure 3-5 illustrates a high-level overview of how the netcast error data from figure 3-3 can be used to simulate netcast on a fully connected DNN architecture, while figure 3-6 illustrates the netcast simulation process for a convolutional network. Note that the basic process for simulating netcast is essentially the same for both network architectures. First, the activations X and weights W are preprocessed to convert X to the range $[0, 1]$ and W to the range $[-1, 1]$. Then, W is factored into $W^{(*,+)}$ and $W^{(*,-)}$, which are both in the range $[0, 1]$. The preprocessed activations and factored weight matrices are multiplied elementwise to give the Hadamard products H^+ and H^-, to which the error elements from figure 3-3 are added. Finally, the Hadamard products are post-processed and scaled to give the desired output: a matrix multiplication for fully connected networks or a convolution for CNNs.

On its face, this may seem like an easy process to implement, but there are three main challenges that this approach introduces. First, modern CNNs contain millions of MACs, while the error distribution only has 100,000 elements in it. Therefore, the netcast simulation cannot simply draw directly from the error distribution because the error probability mass is not evenly distributed. Direct error draws result in skewed errors being simulated, which artificially drops the network accuracy. Second, neural

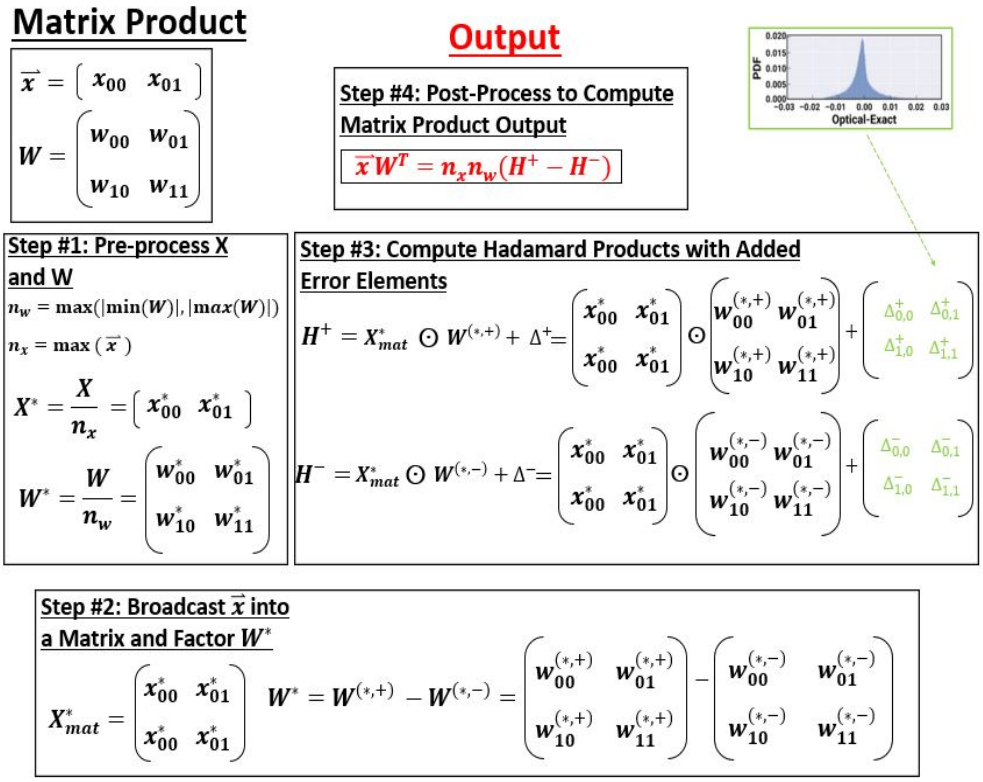


Figure 3-5: High-level illustration of how the netcast optical hardware can be simulated a fully connected network that performs matrix multiplication. The activation X and weights W are preprocessed, and the activation vector is broadcast into a matrix X^*_{mat}. The weight matrix is factored into two terms: $W^{(*,+)}$ and $W^{(*,-)}$ that lie on the range $[0, 1]$. The Hadamard products H^+ and H^- are formed, the error distribution elements are added, and the perturbed products are then post-processed to obtain the desired matrix-vector product, $\vec{x}W^T$. Note that the error elements must be added to the Hadamard products before post-processing in order to implement the update rule given in equation 3.18.

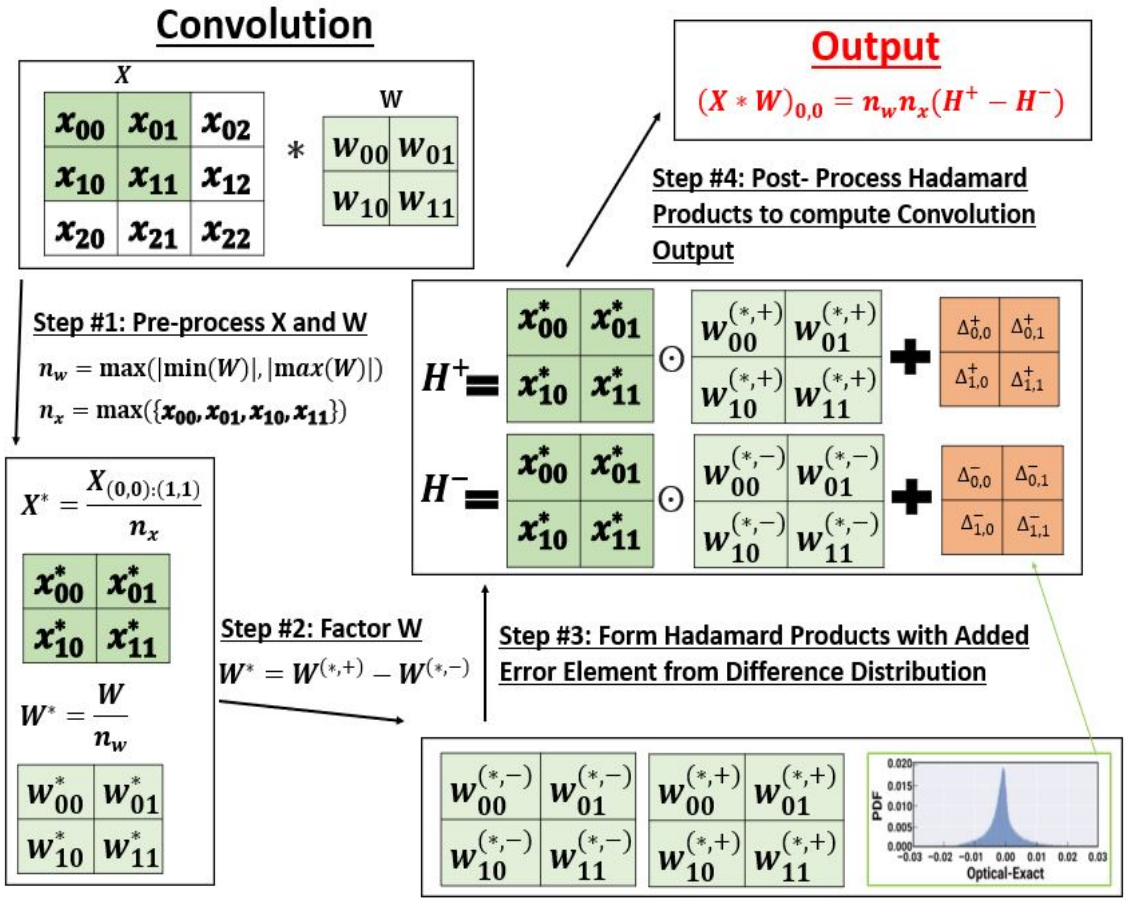


Figure 3-6: High-level illustration of how the netcast optical hardware can be simulated in a convolutional network. Note that the process here is very similar to the fully connected case illustrated in figure 3-5, with the one major difference being that the activation values are already in matrix form and don't have to be broadcast from a vector into a matrix.

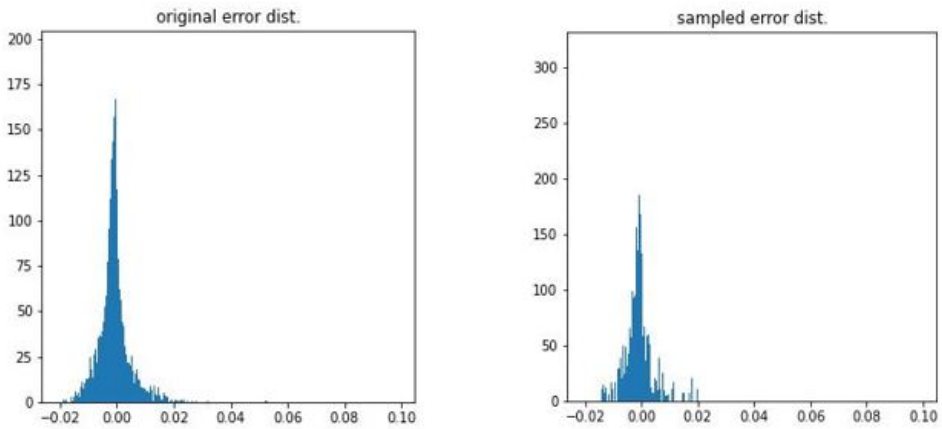
network libraries like Pytorch can perform convolutions and matrix multiplications very quickly, but Pytorch convolutional and fully connected layers do not allow internal modifications of the partial products before the accumulation part of the MAC computation. While this is not a significant issue for matrix multiplication, it is a challenge for convolutional networks because, in order to implement the update rule given in equation 3.18, custom convolution functions must be constructed and implemented. The third and final challenge is computation time. While Pytorch is great at quickly running its own convolutional layers, implementing custom convolutions takes significantly longer and ends up making it infeasible to run large convolutional networks that implement equation 3.18. In order to address this challenge, a custom convolution function called stacked convolution is developed, which derives inspiration from the principle of convolutional reuse mentioned in chapter 2. Section 3.5.3 addresses the first challenge (sampling), while section 3.5.4 addresses the other two challenges (custom convolution functions and computation time).

3.5.3 Netcast Error Distribution Sampling

The first challenge of simulating netcast - drawing error elements with the correct distribution - is a relatively straightforward problem to solve. In order to generate random draws from the difference distribution in figure 3-3, the netcast simulation samples uniformly from the distribution's inverse cumulative distribution function (CDF). A proof that uniformly sampling from a distribution's inverse CDF generates random draws is given in appendix A and is based off of [2]. As shown in figure 3-7, the sampled distribution with 1,000,000 elements is very similar to the original distribution of 100,000 elements and has the same mean and range.

3.5.4 Implementing Stacked Convolution

The convolutional networks used for scene recognition, as detailed in section 3.7, require hundreds of millions to billions of MACs to run a single forward pass. This introduces a significant challenge in creating a netcast simulation that can implement



Distribution	Mean	Min	Max
original	-8.8e-4	-0.02	0.099
sampled	-8.8e-4	-0.02	0.099

Figure 3-7: Comparison between the original netcast error distribution (N=100,000) and a sampled distribution (N=1,000,000) obtained by uniformly sampling the original distribution's inverse CDF. Note the similarity between the two distributions.

equation 3.18 in a reasonable amount of time, since custom convolutional layers take significantly longer to run than Pytorch's nn.Conv2d layer. One potential solution would be to simply reduce the resolution of the input images. However, this detracts from the purpose of the simulation- to see how netcast performs on networks that are actually used in practice. Since modern CNNs take high resolution images as their input, the netcast simulation should do likewise. Without modifying the image size or the depth of the network, the principle of convolutional reuse can be leveraged to implement a custom convolution function that simulates netcast's per-product errors accurately in a reasonable amount of time. Since the coding environment used for this thesis is google colab, a "reasonable amount of time" is defined as $<$12 hours, which is google colab's runtime limit. In order to leverage convolutional reuse to implement an efficient custom convolution function, it is important to understand how convolution is typically implemented in software. As described by [68], 4D convolution on a batch tensor consists of a series of nested for loops that iterate over each (2D input)-(2D kernel) pair, as illustrated in figure 3-8. As seen in the figure, the standard method for

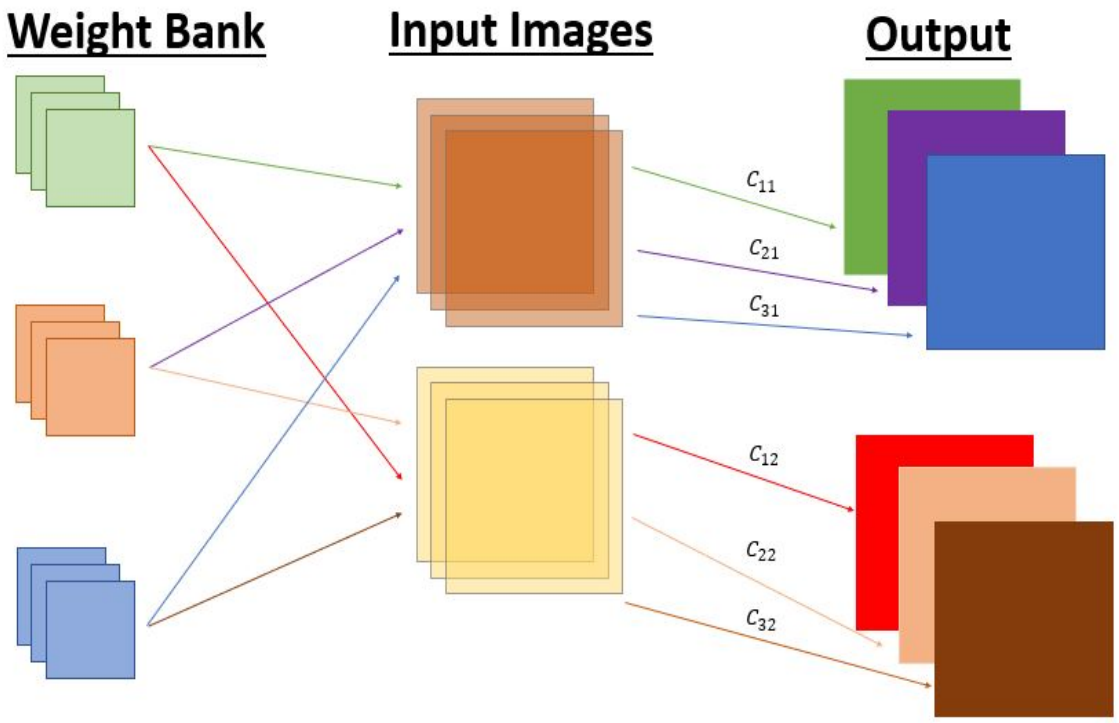


Figure 3-8: Standard method used to implement convolution. Each 3D input image is paired with each 3D filter in the weight bank. Then, within each input-filter pair, each channel is extracted and the 3D convolution is composed of a series of nested 2D convolutions. In this case, each 3D convolution would consist of 3 2D convolutions since there are three channels. While this method is straightforward to implement in software, it results in prohibitively long run times and is therefore unsuitable for the netcast simulation.

convolution looks at each 3D input image in the 4D image batch tensor. Each input image is then paired with each 3D kernel in the 4D kernel bank. Because each image-kernel pair has the same number of channels, the 3D convolution is composed from a series of 2D convolutions, one for each channel. It is important to note that there are advantages to using this convolutional method: it is conceptually simple, easy to implement in software, and is able to finish computations in a reasonable amount of time for smaller networks (2 convolutional layers or less). However, simulating netcast on modern DNNs requires a more efficient convolution method because the CNNs that netcast will be simulated on have much more than just 2 convolutional layers. For example, the RESNET-50 network has 50 layers and requires over 2 billion MAC operations for a single forward pass. This motivates the application of convolutional reuse to construct a custom function that stacks the input activations and weights in a way that implements 4D convolution as a stacked 3D convolution. The process of mapping 4D convolution onto 3D convolution is accomplished in three steps: (1) stack the input feature map (activations), (2) stack weight filters, and (3) use the stacked input feature map and stacked weights to compute the 4D convolution as a 3D stacked convolution. These three steps are illustrated in figures 3-9, 3-10, and 3-11 respectively.

The first step (illustrated in figure 3-9) maps the input 4D batch tensor to a stacked 3D tensor that can be convolved with a stacked filter. In order to stack the 4D input, each 3D tensor in the input batch is repeated M times, where M is the number of 3D weights in the 4D filter bank. If there are N 3D images in the input batch, this means that we will get N groups of M 3D input tensors. Next, each group of M input tensors is stacked in the width dimension to form a single stacked 3D tensor to replace each tensor group. These N stacked tensors are themselves stacked in the width dimension to yield a single 3D tensor called I_s that replaces the 4D input.

The second step stacks the weight bank and is illustrated in figure 3-10. First, the weight bank is repeated N times where N is the number of 3D tensors in the 4D input batch. Next, each repeated weight bank is padded with zeros to ensure that each of

the M filters maps correctly to each of the N 3D inputs. After padding, each repeated weight bank is then stacked in the width dimension, which results in N padded and stacked 3D weight tensors. Finally, the N stacked weights are concatenated in the width dimension so that the original 4D weight bank is replaced by a stacked $3D$ weight tensor called K_s.

With the stacked input and stacked weight tensors in hand, the 4D convolution between the input and weights can be implemented as a 3D convolution between the stacked input and stacked weights, as illustrated in figure 3-11. The first part of this step consists of taking a standard 3D convolution between the stacked input I_s and stacked weight K_s. Since netcast error elements need to be added to each partial product, 3D convolution can be implemented using equation 3.19 below.

$$(C_{F,X})_{i,j} = \sum_{H} \sum_{W} \sum_{CH} n_x n_w (I^*_{s,(:,i:i+K_s[0],j:j+K_s[1])} K^*_s + B\mathbf{\Delta}) \tag{3.19}$$

In the term $(C_{F,X})_{i,j}$, F indexes the 3D filters in the filter bank, X indexes the 3D inputs in the input batch, and (i, j) index the height and width dimensions respectively. The summation indices H, W, and CH represent the height dimension, the width dimension, and the channel dimension respectively. n_x and n_w are normalization factors derived from the stacked input and stacked weights. The star operator (*) denotes the series of preprocessing steps illustrated in figure 3-6, which map I_s and K_s to the range $[0, 1]$. The indexing convention used in the stacked input term is the standard (channel, height, width) convention, so that $I^*_{s,(:,i:i+K_s[0],j:j+K_s[1])}$ is a 3D tensor that includes all channels over the receptive field of the kernel at the location (i, j). $K_s[0]$ and $K_s[1]$ are the height and width of the weight kernel. $\mathbf{\Delta}$ is the error matrix drawn from the netcast difference distribution in figure 3-3, and B is a 3D Bernoulli mask whose shape matches that of the error matrix. The purpose of multiplying the error matrix by the Bernoulli mask is to map the error elements to the correct spots in the preprocessed Hadamard product. Because the weight bank was padded before being convolved with the input, the stacked weight tensor is sparse and thus adding a raw error matrix Δ to the Hadamard product will introduce excess

error at the padded locations. Multiplying by the Bernoulli mask ensures that partial product error elements are only added to points in the Hadamard product where input and weight products are actually calculated. Using equation 3.19 to convolve the stacked input and the stacked weight results in a 2D matrix whose rows contain the flattened convolutional outputs between the filters (indexed by F) and inputs (indexed by X). Each row of this matrix can be reshaped from a 1D vector to a 2D square tensor through row-wise stacking. Finally, each 2D row tensor can be stacked and ordered in the channel dimension to produce the final 4D convolutional output tensor. In order to validate that this method works properly, random 4D inputs were convolved with random 4D weight banks using the three steps described above with an error matrix of all zeros so that $\mathbf{\Delta} = \mathbf{0}$. The stacked convolutional results were then compared to the ground truth output from Pytorch's nn.Conv2D layer, and the results were found to be the same.

By leveraging the principle of convolutional reuse, this custom convolution function allows the netcast hardware to be efficiently and accurately simulated for large CNNs with high resolution images, which is a critical step in validating netcast's usefulness in real-world machine vision tasks.

3.5.5 Calculating Energy Consumption

Having addressed the three main challenges associated with simulating netcast (accurately drawing from the error distribution, constructing a custom convolution function, and optimizing the runtime efficiency) the numerator term of the energy normalized accuracy FOM (equation 3.17) can now be computed. Calculating the denominator term of equation 3.17 is much more straightforward because the amount of MACs needed to run a forward pass through modern CNNs is well known. In the case of custom CNNs and fully connected networks, the MAC count needed for a forward pass is not well known but can be easily calculated. A convolutional layer ℓ that convolves an input with C_{in}^{ℓ} channels uses a weight bank containing 3D kernels of size $(C_{in}^{\ell}, \kappa^{\ell}, \kappa^{\ell})$ since the number of channels in the kernel must match the number of channels in the input. This means that a single 3D kernel in the weight bank will

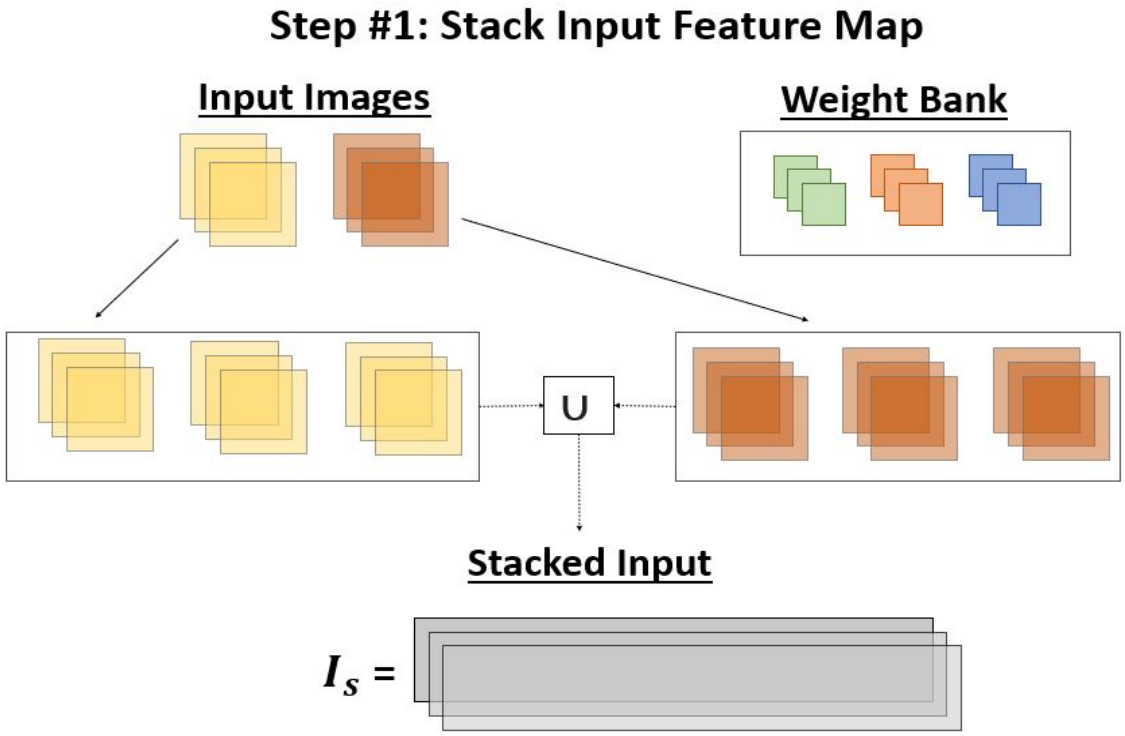


Figure 3-9: The first step of constructing the stacked convolution function. In this step, the 3D input images in the 4D input batch tensor are repeated M times, where M is the number of 3D weights in the 4D weight bank. In the figure, $M = 3$. The groups of repeated inputs are stacked in the width dimension, and these stacked inputs are then concatenated in the width dimension as illustrated by the U function. The concatenated, stacked inputs form a 3D tensor that replaces the original 4D batch tensor.

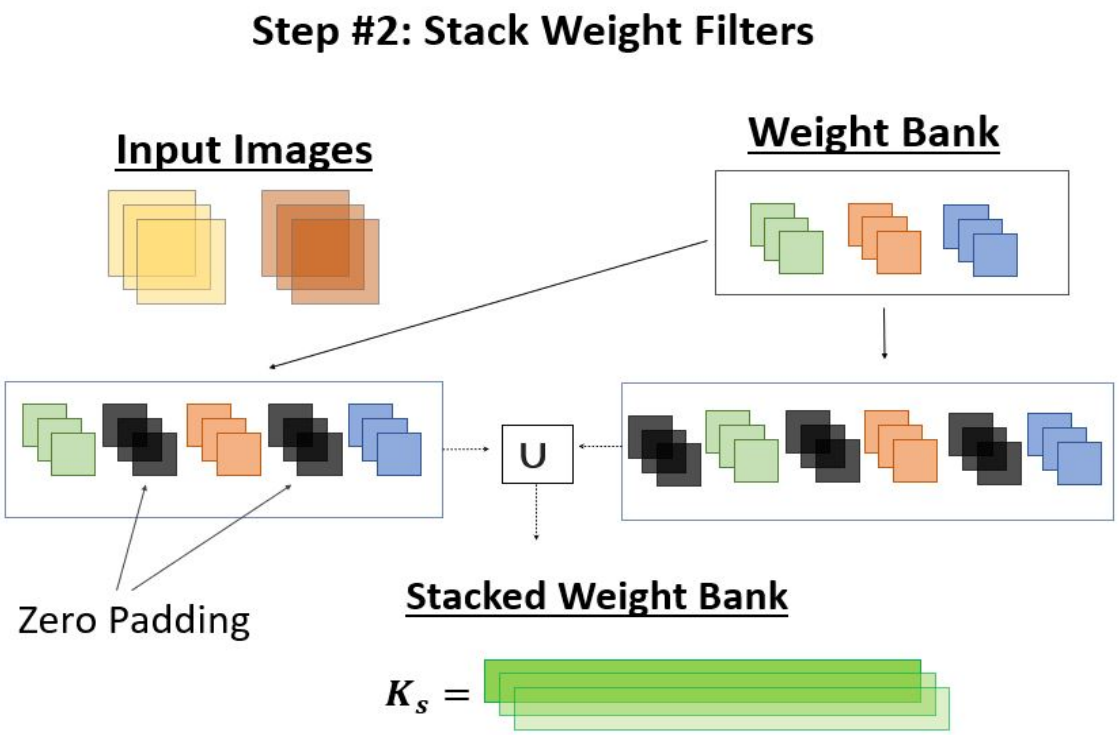


Figure 3-10: The second step used to construct the stacked convolution function. In this step, the 4D weight bank consists of M 3D weight kernels and the input consists of N 3D images. First, the entire 4D weight bank is repeated N times- once for each input image. Next, zero-value tensors are inserted between the 3D kernels in each repeated weight bank in order to map the appropriate weight kernel to the appropriate 3D input tensor. Finally, the N padded and stacked filters are concatenated in the width dimension so that the original 4D weight bank is replaced by a padded 3D weight tensor.

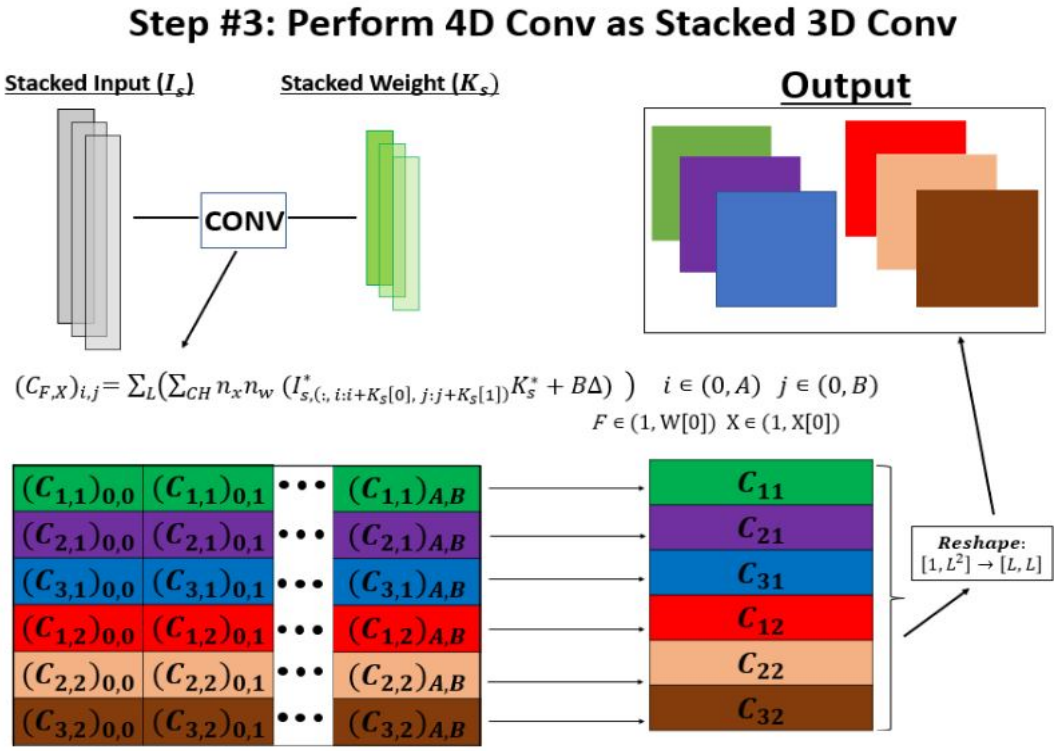


Figure 3-11: The third step used to construct the stacked convolution function. In this step, the stacked input I_s and the stacked weight K_s are convolved to produce a 2D matrix where each row contains the flattened elements of $C_{F,X}$, which represents the convolution between 3D filter F and 3D input X. Note here that $W[0]$ is the number of 3D filters in the filter bank, $X[0]$ is the number of 3D inputs in the batch tensor, and A and B are the output height and width respectively. Each row of the 2D matrix can be reshaped into a square 2D tensor, which can be ordered and stacked into the 4D tensor that corresponds to the 4D convolutional output. Note that the netcast errors are added to the elementwise product $I^*_{s,(:,i:i+K_s[0],j:j+K_s[1])} K^*_s$ where $K_s[0]$ and $K_s[1]$ are the height and width of the weight kernel. Also note that $\sum_{CH} ...$ denotes a sum in the channel dimension and the matrix B denotes a Bernoulli mask that maps error elements to locations in the elementwise product where input-weight partial products occur. Finally, $\sum_L ...$ represents the sum over the width and height dimensions of each input-kernel patch of the stacked product: $\sum_{CH} n_x n_w (I^*_{s,(:,i:i+K_s[0],j:j+K_s[1])} K^*_s + B\Delta))$. The star (*) operator in this case denotes preprocessing that scales I_s and K_s to the range $[0, 1]$, and n_x and n_w are normalization factors.

Table 3.1: Energy/MAC values for Digital Networks and Netcast

Network	**Energy/MAC**
Digital	1 pJ/MAC
Netcast	$\approx$ 10 fJ/MAC

perform $(\kappa^\ell)^2 C_{in}^\ell$ MACs at each location where the kernel is swept over the input. If the output tensor has height H_{out}^ℓ and width W_{out}^ℓ, then there are $(H_{out}^\ell W_{out}^\ell)(\kappa^\ell)^2 C_{in}^\ell$ total MACs performed for a single 3D input - 3D kernel convolution. The number of kernels in the weight bank controls the number of output channels, so the weight bank must have C_{out}^ℓ kernels, where C_{out}^ℓ is the number of channels in the output tensor. Since the weight bank has C_{out}^ℓ kernels, and each kernel performs $(H_{out}^\ell W_{out}^\ell)(\kappa^\ell)^2 C_{in}^\ell$ MACs, the total MAC count for the convolutional layer is $(H_{out}^\ell W_{out}^\ell)(\kappa^\ell)^2 C_{in}^\ell C_{out}^\ell$. Since a convolutional network is a series of convolutional layers, the total MAC count for a CNN can be computed as the sum given in equation 3.20 below.

$$MAC_{total} = \sum_\ell (H_{out}^\ell W_{out}^\ell)(\kappa^\ell)^2 C_{in}^\ell C_{out}^\ell \tag{3.20}$$

For a fully connected network, the number of MACs per layer is simply the product of the weight matrix height and width. The per-layer MACs can be summed to obtain the following equation for a fully connected network.

$$MAC_{total} = \sum_\ell W_\ell[0] W_\ell[1] \tag{3.21}$$

Where $W_\ell[0]$ and $W_\ell[1]$ are the height and width of the weight matrix for layer ℓ. Converting from MAC count to energy consumption requires knowing the energy/MAC cost for both digital networks and netcast. These two values, given by [27] and [13], are shown in table 3.1. Using the information in the table along with equations 3.20 and 3.21 allows us to calculate the energy consumption for a given DNN architecture. In section 3.6, netcast is simulated on the task of MNIST handwritten digit classification, and its energy normalized accuracy is compared to that of its digital

counterparts. In section 3.7, netcast is simulated on an open problem in machine vision: scene recognition for robotic localization.

3.6 Simulating Netcast on MNIST Digit Classification

As a baseline test, the netcast hardware is simulated on a relatively simple machine vision task: classifying handwritten digits from the MNIST data set. Simulating netcast on this task consists of three steps. First, train and test a digital network to obtain the weight matrix $\mathbf{W}$. Second, using $\mathbf{W}$, simulate a forward pass computed by netcast using the stacked convolution function and the per-product error distribution. Finally, compare the energy normalized accuracy of the netcast output to that of the digital model. These steps are described in more detail below.

As shown in figure 1-10, an MNIST classifier maps input images of handwritten digits to the correct category (0 through 9). The training set consists of 60,000 images, while the testing set consists of 10,000 images. As a first step, the training and testing data were grouped into batches of 50 images and resized to a resolution of 56x56. After processing the data, two DNN architectures were constructed: a fully connected network (FC3) and a convolutional network (Conv3). The FC3 architecture consists of three fully connected layers of sizes 1000, 100, and 10, with RELU nonlinearities between the first two layers and a Softmax layer at the end. The Conv3 architecture consists of two convolutional layers, each with a RELU nonlinearity, followed by a dropout layer. Dropout layers zero out random input elements with some probability (in this case $p = 0.5$) in order to prevent model overfitting. After the dropout layer, the convolutional feature maps are flattened into a vector and fed through a fully connected layer with 20 neurons and then finally through a Softmax layer.

The FC3 network was trained for 30 epochs using stochastic gradient descent (SGD) with a learning rate of 0.01 and momentum of 0.9. Momentum can be thought of as a moving gradient average that helps the model train more quickly [54]. Cross

entropy, as given in equation 1.13, was used as the loss function to be optimized. The Conv3 network also used the cross entropy loss function, but was trained for only 10 epochs through the data. The Conv3 network was trained using the Adam optimizer, which is a gradient-based optimizer that uses momentum and exponential moving gradient averages to slightly outperform SGD [87]. After training, FC3 and Conv3 were tested and found to have accuracy values of 0.979 and 0.980 respectively.

After training and testing both FC3 and Conv3, the parameters from both trained networks along with the stacked convolution function and netcast error distribution were used to simulate netcast in both architectures. For both FC3 and Conv3, the netcast error distribution elements were added to each partial product for each forward pass through the test set. Note that, since FC3 only implements matrix multiplication, the stacked convolution method was not used for this architecture. Instead, matrix multiplication at each layer was implemented using a series of preprocessing steps to map the inputs and weights to the range $[0, 1]$. This preprocessing was followed by elementwise multiplication between the inputs and weights, to which the error distribution elements were added, as depicted in figure 3-5. The elementwise products were then summed and post-processed to yield the correct matrix multiplication. Since the Conv3 network contained a combination of convolutional and fully connected layers, the above procedure was used to simulate the fully connected layers, while the stacked convolution method from section 3.5.4 was used in the convolutional layers.

After simulating the netcast hardware in both FC3 and Conv3, the energy consumption of both networks was calculated using equations 3.21 and 3.20. The MAC counts were then converted to energy consumption using the energy/MAC values in table 3.1. With these energy consumption values in hand, the energy normalized accuracy (ENA) of the digital networks (FC3 and Conv3) and the netcast simulations (NetcastFC3 and NetcastConv3) were calculated and compared. The results are given in table 3.2. These results indicate that the netcast hardware is able to outperform its digital counterparts by three orders of magnitude in terms of accuracy per energy consumed. This provides strong evidence that netcast is much better

Table 3.2: Energy Normalized Accuracy for Digital Network and Netcast

Network	**Accuracy**	**Energy Consumption**	**ENA**
FC3	0.979	3.24 μJ	0.300
Conv3	0.980	64.5 μJ	0.150e-1
NetcastFC3	0.975	32.4e-4 μJ	300.930
NetcastConv3	0.972	64.5e-3 μJ	15.070

suited to run DNN workloads in edge computing applications compared to digital electronics. The code used to simulate netcast on MNIST digit classification can be found here: `https://github.com/jmcochrane1998/ONN-Simulation-Code.git`. While these results look promising, it could be argued that MNIST digit classification is a relatively simple task that does not accurately represent the state of the art when it comes to machine vision. This is true, and section 3.7 simulates netcast on a much more advanced application that lies at the cutting edge of current machine vision research: scene recognition for localization in mobile robotics.

3.7 Simulating Netcast on Scene Recognition For Robotic Localization

3.7.1 Introduction and Overview

Recall from section 1.3.2 the second machine vision task that netcast is to be simulated on: training a mobile robotic system to recognize the scene at which it is currently located. When integrated into a full stack system, this vision task allows the robot to perform simultaneous localization and mapping (SLAM) to generate an accurate topological map of its operating environment (similar to figure 1-14) and know its current position in that environment. As described by [60], localization and mapping for mobile robotics has both a quantitative and a qualitative component. While the quantitative component focuses on accurate pose estimation for low-level trajectory planning, the qualitative component focuses on understanding what topo-

logical node (location) the robot is in. Similar to [60], this thesis focuses on the qualitative aspect of localization, which is formulated as a k-way classification problem between different types of scenes. Unlike MNIST digit classification discussed in section 3.6, scene recognition is challenging for several reasons. First, accurately identifying scenes requires a weighted balance of both local image information (specific objects or shapes) as well as global image information (color or texture) [78]. For example, identifying a kitchen might be done by detecting the presence of a specific object class (dishes), whereas trying to identify a parking spot might call for reliance on color and texture features instead of objects. In addition, [69] notes that mobile robotics tasks including place recognition, SLAM, and pose estimation are subject to high degrees of uncertainty, which can accumulate as the robot continues to operate. As will be shown in section 3.7.2, recent advances in deep learning have significantly increased the accuracy of mobile robot vision networks. However, many modern deep learning approaches have optimized accuracy at the expense of energy consumption, which leads to feasibility issues [78]. Edge devices (including small robotic platforms) can typically source milliwatts of power, whereas large CNNs require watts of power to run [13]. For example, using a VGG19 CNN to run a live video image recognition system at 100 Hz speed requires 9.84 millijoules of energy per forward pass, which corresponds to $(9.84 * 10^{-3}J)(100s^{-1}) \approx 1$ watt of power. While the VGG19 might be very accurate, its energy cost lies beyond what the robot hardware can source. The high energy cost of running large-scale CNNs on mobile robots motivates the application of netcast to compute the CNN workload instead of digital electronics.

3.7.2 Previous Work

The previous work related to scene recognition can be roughly divided into three categories: (1) laser range data classifiers, (2) hand-crafted image descriptors, and (3) deep learning methods. Major works in each category are given below.

Laser Range Data Classifiers

Early work in the field of scene recognition focused on the use of laser range data to estimate the geometry of a scene and then semantically classify it. In [53], Mozoz et al. input laser range data into a simple classifier model to determine the identity of the corresponding scene. Zender et al. combine both visual and range data as well as a linguistic framework to help the robot perform SLAM with multiple integrated data sources [21]. The authors use a consensus-like algorithm called a voting matrix formed from a receptive field occurrence histogram, which measures how often certain filter responses and colors occur in an input image [21]. While laser-based methods can achieve high accuracy in certain environments, their ability to generalize to a variety of different scene types is limited. As noted by [60], classifying a scene from laser scanning fails when two different environments have similar geometric structures but different visual appearances. This limitation has motivated the use of visual data captured from RGB cameras to perform scene recognition tasks. The next section describes the major works that develop hand-crafted features from visual data to perform scene recognition more robustly.

Hand-Crafted Image Descriptors

In [45], Lowe et al. developed the Scale Invariant Feature Transformation (SIFT), which describes an image using gradients at and around a group of key points that are calculated using a Difference of Gaussians (DOG) function [66]. Building off of SIFT, the authors of [23] developed the Speeded-Up Robust Features (SURF) descriptor, which uses integral images to make the key point detection step more efficient. Instead of using a DOG function, SURF finds key points using the determinants of the Hessian matrix at each location of the integral image. The locations that have maximal Hessian determinants are selected as key points [66]. In [7], Dallal and Triggs develop the Histogram of Oriented Gradients (HOG) descriptor, which partitions an input image into small patches called cells. Within each cell, the gradient directions are quantized and binned, and a histogram of gradient magnitudes is calculated using

the cell's orientation bins. The histograms for the cells are stacked into a vector that forms the HOG descriptor for that image [66]. Another major work in the field is [52], in which Torralba and Olivia leverage the representational power of the frequency domain to encode image spectral information using the discrete Fourier transform (DFT) of an image. The authors use the DFT to encode and compress the input image, and their work demonstrates a reasonable level of accuracy.

The above works helped move the field of scene recognition from a heavy reliance on laser range data to methods that leveraged a scene's rich optical information. However, the rise of AlexNet in the 2012 ImageNet challenge demonstrated that the path towards more reliable and adaptable scene recognition methods could be found in the field of deep learning. Deep learning methods for scene recognition are discussed next.

Deep Learning Methods for Scene Recognition

Instead of hand-crafting image features to classify a scene, convolutional neural networks use the deep learning methods discussed in chapter 1 to learn the optimal filters that allow accurate and generalizable scene recognition. In [78], Wozniak et al. posit the use of a VGG CNN architecture to perform scene recognition as an 11 class classification problem. The authors use transfer learning to fine tune a VGG network on custom data using stochastic gradient descent and a slight structural modification of the fully connected layers. In [62], the authors consider the task of 3D scene recognition using a custom DNN called ScanNet as well as a RESNET architecture.

While deep learning methods applied to scene recognition have demonstrated <10 percent error rates, [78] notes one major challenge associated with applying deep learning to mobile robotics: energy costs. Like other SWaP-limited applications, mobile robotics is constrained by limited battery life, and running large DNNs on a mobile robot can incur significant energy costs [78]. Here, mobile robotics can be conceptualized as a specific instance of edge computing. As discussed in section 2.2.2, edge device SWaP constraints mean that feasibly deploying DNNs to mobile robotics requires a joint optimization of accuracy and energy efficiency. This motivates the

application of netcast to mobile robotic scene recognition. Using the procedure detailed in section 3.5, netcast is simulated on a scene recognition CNN and its energy normalized accuracy (ENA) is compared to the average digital electronic ENA over a group of selected CNNs. The details of the methodology are given in section 3.7.3 below.

3.7.3 Methods: Simulating Netcast on Scene Recognition

Because section 3.5 already gives a complete description of how netcast can be simulated using stacked convolution and the netcast error distribution, those details will not be repeated here. Instead, figure 3-12 gives a high-level description of the methodology used for the netcast scene recognition simulation. This methodology consists of six steps:

(1) Acquire scene recognition training and testing data

(2) Select a group of CNN architectures to use (called $\mathcal{S}$)

(3) Train each CNN ϵ $\mathcal{S}$ on the training data to get trained parameter matrices W_i where i indexes the individual CNNs in $\mathcal{S}$

(4) Instantiate the netcast simulation on one of the CNN architectures in $\mathcal{S}$ and compute netcast's energy normalized accuracy (called $ENA_{netcast}$)

(5) Run testing on each digital CNN in $\mathcal{S}$, record each network's ENA, and then average the individual digital ENAs to find $ENA_{digital}$

(6) Compare $ENA_{netcast}$ to $ENA_{digital}$

Each step is described in more detail below.

Acquire Scene Recognition Training and Testing data

The data set used is the MIT Indoor Scenes data set, which can be found at `https://www.kaggle.com/datasets/itsahmad/indoor-scenes-cvpr-2019/discussion`. The raw data set consists of 67 indoor scene classes with at least 100 images per class, for a total image count of 15620. Following the example of [78], scene recognition is formulated as an 11 class classification problem, where the 11 scene categories sampled

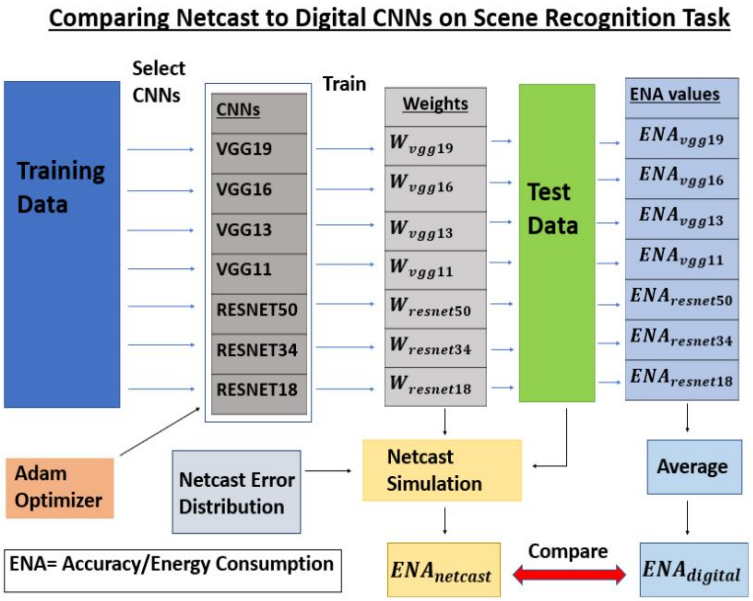


Figure 3-12: Method used to simulate netcast on scene recognition. From a given training set, train a selected group of CNNs using the Adam optimizer. The CNN architectures chosen are depicted in the figure, and these specific architectures were chosen because previous work uses VGG and RESNET ([78], [62]). Each trained network's parameter matrix is denoted by W_i where i indexes the CNNs in the group. To run the netcast simulation, the RESNET18 architecture was chosen because it is small enough to be simulated relatively quickly ($\approx$ 1.5 hour) but also deep enough to obtain a reasonable network accuracy. Using the netcast error elements from figure 3-3 and the test data withheld from training, the netcast procedure outlined in section 3.5 gives the simulated netcast accuracy, which can be divided by the netcast energy consumption on RESNET18 to obtain netcast's energy normalized accuracy, $ENA_{netcast}$. Each digital CNN is tested on the same test data, and the output test accuracies/energy are averaged to yield the digital accuracy per energy value, $ENA_{digital}$. Finally, $ENA_{netcast}$ and $ENA_{digital}$ are compared to see how well the netcast hardware performs relative to digital electronics.

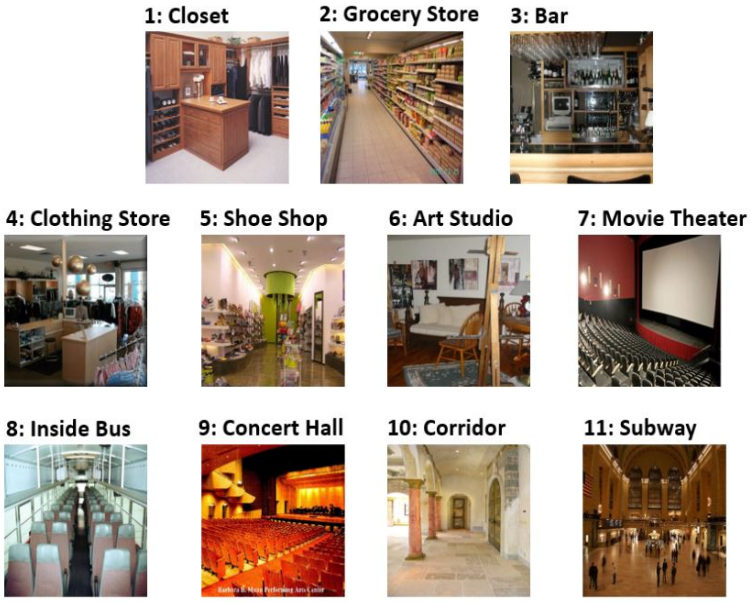


Figure 3-13: Eleven scene classes sampled from the MIT Indoor Scenes Dataset.

from the raw data are shown in figure 3-13. To process the data, the images were all converted to grayscale tensors and divided into training, validation, and testing sets prior to mixing. This ensures that each data set contains images from each scene class. The training set contained 2237 images, the validation set contained 161 images, and the testing set contained 160 images. Each ground truth label was mapped to a one-hot encoded vector as is typically done in classification problems. After the (image, label) pairs were processed and divided into train, validation, and test sets, the data within each set was mixed and batched at 32. All images were resized to a resolution of 168x168.

Select a Group of CNN Architectures

With the training and testing data in hand, the next step is to select a group of convolutional networks that will be trained and tested on the data. Because prior work uses the VGG and RESNET architectures, the selected group of CNNs used ($\mathcal{S}$) is shown in figure 3-12. The VGG architecture consists of multiple convolutional layers with RELU nonlinearities and max pool layers that iteratively downsample the image resolution as the image goes deeper into the network [78]. After the series of

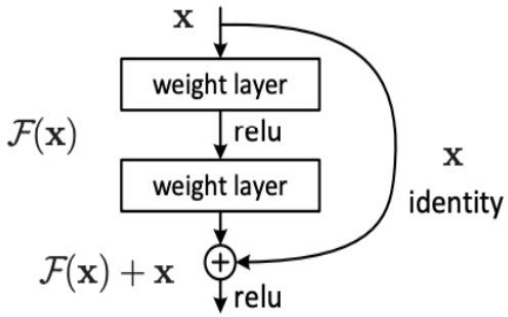


Figure 3-14: The residual block implemented in RESNET. Standard convolutional networks like VGG learn the mapping $F(x)$, which is susceptible to vanishing or exploding gradients as the network depth increases. RESNET architectures instead learn the mapping $F(x) + x$, which allows gradient information to flow through the identity connection even if the gradient from the previous layer vanishes to zero or explodes to infinity.

convolutional layers extracts the image features, these features are then run through a series of fully connected layers that output a probability distribution over the 11 scene classes [78]. Like the VGG architecture, RESNETs have a series of convolutional layers followed by a series of fully connected classification layers. The main difference between the RESNET and VGG architectures is that a RESNET employs what are often called skip connections to help solve the vanishing/exploding gradient problem [39]. The vanishing/exploding gradient problem states that, as data passes deeper into CNNs, the numerical values of the gradients used for training tend to either drop to zero or explode to infinity. As the gradients either vanish or explode, the model training fails to reach a reasonable level of accuracy. In order to address this problem, RESNETs provide a skip connection that allows gradient information to flow through the network during training in a way that is robust to vanishing or exploding gradients [39]. The RESNET skip connection structure is illustrated in figure 3-14.

Train Each Digital CNN

The next step is to take the selected CNNs and train them. All of the chosen models were trained using the Adam optimizer with a learning rate of 0.001. The loss function optimized during training is cross entropy as defined in equation 1.13. Training each

CNN model for 30 epochs resulted in the parameter matrices represented in figure 3-12.

Run the Netcast Simulation

In order to use the procedure described in section 3.5, it is first necessary to select a model architecture to use. From $\mathcal{S}$, the RESNET18 was selected because it is both small enough to simulate in a relatively quick amount of time ($\approx$ 1.5 hours) and large enough to provide good image features of the scenes. Using the RESNET18 trained weight matrix, $W_{resnet18}$, in conjunction with the netcast error distribution and the stacked convolution function, the netcast hardware was simulated on the test data, and the output accuracy was found to be 60 percent. Next, the energy consumption must be calculated. From [49], RESNET18 requires 0.91 GMACs (giga-MACs) to compute a single forward pass. From table 3.1, this MAC count corresponds to an energy consumption cost of about 0.91μJ. Thus, netcast's energy normalized accuracy is $\frac{0.6}{0.91} = ENA_{netcast} = \mathbf{0.6593}\ \mu J^{-1}$. The code used to simulate netcast on the scene recognition task can be found here: `https://github.com/jmcochrane1998/ONN-Simulation-Code.git`.

Test Each Digital CNN and Find the Average Digital ENA

Each digital CNN (including RESNET18) was run on the test data as shown in figure 3-12. Table 3.3 shows the results. Averaging the per-network ENAs from the table gives a digital energy normalized accuracy of $ENA_{digital} = \mathbf{0.2007 mJ^{-1}}$.

Compare the Netcast Energy Normalized Accuracy to the Digital Energy Normalized Accuracy

Comparing the accuracy per energy of netcast to that of the digital networks indicates that netcast outperforms its digital counterparts on average by a factor of 3000 as

Table 3.3: Energy Normalized Accuracy (ENA) of Each Digital CNN Model. Note that the star (*) here denotes a model with batch normalization.

Digital Model	**Accuracy**	**Energy Consumption[mJ]**	**ENA[mJ^{-1}]**
RESNET50	0.731	2.06	0.3549
RESNET34	0.675	1.84	0.3669
RESNET18	0.725	0.91	0.7967
VGG19	0.238	9.84	0.0242
VGG16	0.238	7.75	0.0307
VGG16*	0.500	7.77	0.0644
VGG13	0.238	5.67	0.0420
VGG13*	0.406	5.68	0.0715
VGG11	0.588	3.82	0.1539
VGG11*	0.388	3.82	0.1016

seen by the calculation below.

$$\frac{ENA_{netcast}}{ENA_{digital}} = \frac{0.6593(\mu J)^{-1}}{0.2007(mJ)^{-1}} = \frac{0.6593e6J^{-1}}{0.2007e3J^{-1}} \approx 3000 \tag{3.22}$$

This calculation, $\frac{ENA_{netcast}}{ENA_{digital}}$, represents an improvement of over three orders of magnitude, the importance of which is discussed in the next section.

3.7.4 Results and Discussion: Netcast Versus Digital Electronics

How can the numerical result from equation 3.22 be interpreted? Here, it is important to note that netcast already outperforms any digital network by three orders of magnitude when it comes to energy consumption, as indicated by the energy/MAC values in table 3.1. Because of netcast's head start in energy consumption, the standard used to compare netcast to digital electronics is the following: netcast's ENA must exceed the digital ENA by at least three orders of magnitude. For example, if the results from subsection 3.7.3 had been $\frac{ENA_{netcast}}{ENA_{digital}} = 10$ (one order of magnitude improvement), these results would indicate very poor performance from netcast. This is because an ENA improvement of 10 with an energy improvement of 1000 means

that netcast's raw network accuracy was only 1 percent of its digital counterpart. Regardless of the energy efficiency, such a small accuracy value (≈ 0.005) is well below the threshold of practical usefulness. On the other hand, having an ENA improvement, $\frac{ENA_{netcast}}{ENA_{digital}} = 1000$ (three orders of magnitude) lies at the threshold of good performance since it indicates that netcast's accuracy matched that of its average digital counterpart with significantly less energy consumed. Therefore, with an ENA improvement factor of 3000, netcast takes a significant lead over its digital electronic counterparts in the task of scene recognition.

Before concluding, it is important to address the raw accuracy of the models in table 3.3 - namely the fact that even RESNET isn't as accurate as current work in the field. While this is true, the point of this thesis was not to develop a novel scene recognition network but rather to simulate a novel ONN accelerator on relevant CNN workloads that are commonly used in scene recognition. The comparison that this thesis focuses on is netcast performance vs. digital performance for any given parameter matrix. Another important note is that the data size used in this application is significantly smaller than that used by current work to include [78] and [62]. Since the FOM used in this thesis is not just network accuracy (but rather ENA), all of the models in table 3.3 were trained from scratch, whereas current work commonly uses transfer learning with much larger data sets. Therefore, data augmentation and transfer learning are two short-term techniques that can be used to increase the raw network accuracies used to simulate netcast. The key point here is that the simulated netcast ENA depends on the per-product error distribution from figure 3-3 for any given DNN weight matrix. Therefore, even if transfer learning and data augmentation were applied to increase the raw DNN accuracies, this would not significantly affect the 3000 factor ENA improvement demonstrated by netcast because the numerator and denominator terms of $\frac{ENA_{netcast}}{ENA_{digital}}$ would increase by similar amounts. Therefore, regardless of the raw accuracies in table 3.3, the netcast hardware is able to significantly outperform modern digital electronics and thus enables the energy-efficient deployment of large DNNs to SWaP-constrained edge applications like mobile robotics.

3.8 Conclusion and Summary

As discussed in section 3.3, netcast leverages frequency-domain and time-domain parallelism to run DNNs at 1000 times lower energy costs as well as minimize thermal noise (section 3.2). While this improvement in energy efficiency is important, energy costs alone do not demonstrate netcast's effectiveness on real-life machine vision tasks. In order to determine whether or not netcast can outperform digital CNNs, two tasks were chosen: MNIST classification and scene recognition. In section 3.5, a custom convolution function was constructed that leveraged the principle of convolutional reuse. This stacked convolution function allows the netcast error distribution to be propagated through any DNN architecture, thus allowing netcast to be simulated in software. Using the stacked convolution method, both machine vision use cases were considered separately. The first task of MNIST classification is a canonical vision problem, and it is not very surprising that netcast was able to perform well. After validating netcast's baseline performance, a more challenging application area was considered: scene recognition. This use case is challenging because accurate scene recognition requires a balanced understanding of both local image features as well as global image properties. Previous work in this field has tried to use laser range data, hand constructed features, and image Fourier representations. The recent advances in deep learning and the application of CNNs to vision problems have dramatically increased the localization accuracy of robotic systems. However, as noted by [78], one major obstacle that hinders the scalability of large DNNs in mobile robotics is that large networks require billions of MACs to run inference, which in turn requires a significant amount of energy. This motivates the simulation of netcast on the task of scene recognition. Comparing netcast's energy normalized accuracy to that of its digital counterparts indicates that the ONN is able to outperform digital networks by well over three orders of magnitude. The simulation results provide strong evidence that netcast is able to jointly optimize both accuracy and energy consumption in large DNNs applied to mobile robotics. Netcast opens the door to novel applications that allow powerful AI networks to be employed by small edge devices, and the range

of possibilities for such technology is virtually limitless.

Chapter 4

Establishing Acceptable Performance Metrics: Policy Implications of Netcast

Having demonstrated netcast's ability to integrate DNNs into edge computing hardware, this chapter considers specific applications that netcast could enable. These applications are then used to motivate a policy analysis that considers how acceptable performance standards should be distributed over netcast's application space. While the policy analysis in section 4.3 applies to all potential netcast applications, this chapter starts with five specific applications to establish a clear starting point. Before discussing applications, section 4.1 looks at the functional limitations of applying netcast to edge computing. Section 4.2 then focuses on five specific applications, which are divided into two general categories: civilian (section 4.2.1) and military (section 4.2.2). Section 4.3 then taxonomizes the netcast application space, proposes two metrics to quantify acceptable performance standards, and considers how these acceptable performance metrics should be applied to netcast's application space.

4.1 Netcast Limitations

As discussed in chapter 3, netcast is able to perform computation with about 1000 times less energy than currently used digital electronics. In addition, netcast's energy normalized accuracy (ENA) for scene recognition is 3000 times greater than the digital ENA. While these results are promising, it is important to acknowledge that netcast has two major limitations: (1) susceptibility to electrostatic discharge (ESD) and (2) the inability of C-band optical signals to travel through arbitrary media.

Netcast's first limitation comes from the fact that it is fabricated using Complementary Metal–Oxide–Semiconductor (CMOS) technology. CMOS is a popular fabrication technique used to make integrated circuits (ICs), including those used in netcast. Although CMOS is the technique of choice used in modern CPUs, microprocessors, and cell phones, devices fabricated using CMOS can be damaged or destroyed with as little as three volts of electrostatic discharge [55]. While this is not a significant issue in the lab, many industrial and military applications are prone to ESD. This means that organizations desiring to use the netcast technology should pay close attention to how their proposed system shields netcast's ICs from ESD. This will likely require additional costs in terms of time and money spent designing ESD shields, but such costs may be worth it depending on the application.

Netcast's second limitation comes from the fact that a C-band optical beam connects the server and client. In all the applications discussed in section 4.2 below, a linear (or piecewise-linear) path connects the DNN weight server to the edge device client. This necessity for a linear server-client path comes from the fact that optical light can't go through solid matter such as walls. While [13] proposes a radio frequency (RF) version of netcast that would be able to penetrate solid barriers, the technology's current version uses exclusively optical signals. This means that potential applications must have a relatively clear visual path between the weight server and the client. This is a significant operational limitation because it limits the possible server-client configurations that can be practically employed in the short-term. As a result, future work should focus on developing an RF version of netcast where

the server-client signal is capable of traveling through solid obstacles.

4.2 Potential Netcast Applications

Although the netcast hardware was designed with no specific use case in mind, the technology opens the door to a virtually endless list of possible novel applications. This diverse application space is centered around the idea of deploying computationally powerful DNNs to small hardware devices that can process data in real-time. Section 4.2.1 lists three potential civilian applications of netcast: (1) deploying microdrones to augment search and rescue operations, (2) autonomously exploring the ocean floor, and (3) automating commercial facility safety inspections. Section 4.2.2 discusses two possible military applications: (1) autonomous targeting using unmanned aerial vehicles (UAVs) for ground operations and (2) naval autonomous targeting using autonomous underwater vehicles (AUVs). Each specific application is discussed in more detail below.

4.2.1 Civilian Applications

This section describes how the netcast hardware can be applied to search and rescue operations as well as autonomous ocean exploration and autonomous safety inspections.

Smart Microdrones for Search and Rescue (SAR)

Search and rescue (SAR) operations are challenging, complex, and often disheartening. SAR missions are generally characterized by two major challenges: (1) time is critical because human lives depend on fast action and (2) the operating environment is often hazardous [74]. When a disaster occurs, SAR teams are suddenly inundated with a massive amount of problems and decisions with an unforgiving timeline. With only a limited amount of information, rescuers need to quickly determine survivors' locations, conditions, and the safest way to provide help [3]. The past decade has seen an increased interest from researchers and SAR teams regarding the use of robotics

and AI to aid search and rescue operations. For example, two unmanned aerial vehicles (UAVs) were used by SAR teams in 2006 to search for survivors after Hurricane Katrina [74]. Also, [3] considers the application of small robotic systems to SAR operations in places that human teams cannot access such as collapsed buildings. The authors of [10] develop an image classification network that uses a series of handcrafted image features to map a disaster area in order to better inform SAR ground teams. The application of robotic and AI systems to SAR operations comes with several technical challenges, however [74]. First, the environments of many disaster areas make the process of high quality sensing and data acquisition difficult [74]. In addition, SAR robotic systems and UAVs have limited size and payload capacities, which make it very difficult for such systems to source enough energy and onboard memory to run large DNNs [74]. As a result of these technical limitations, current work uses small convolutional networks that are only able to output low-dimensional information (ex. a simple binary classification between a person being injured or not) [15]. Such simple binary classifications provide very limited data to SAR teams and give no indication of how to prioritize different survivors that are injured or how to optimally navigate to them. Using larger DNNs to provide high-dimensional information to SAR teams would provide a much more complete and clear picture of a disaster area.

The limitations of current methods motivate the application of netcast to the SAR problem. The netcast ONN is well suited for applications like SAR because the hardware (1) significantly reduces the amount of energy needed to operate large image processing DNNs and (2) eliminates the need for small UAVs to store weights locally. While netcast could be used in a variety of SAR devices, this section focuses on the integration of netcast into a small UAV. Figure 4-1 shows one potential way in which a netcast-based system could be used to autonomously generate a high-dimensional map of a disaster area. As shown in the figure, the DNN weight server lives on a satellite that broadcasts the DNN parameters through free space to the UAV. Using the weights from the server, the UAV then autonomously generates a high-dimensional map of the disaster area that is transmitted to a nearby SAR

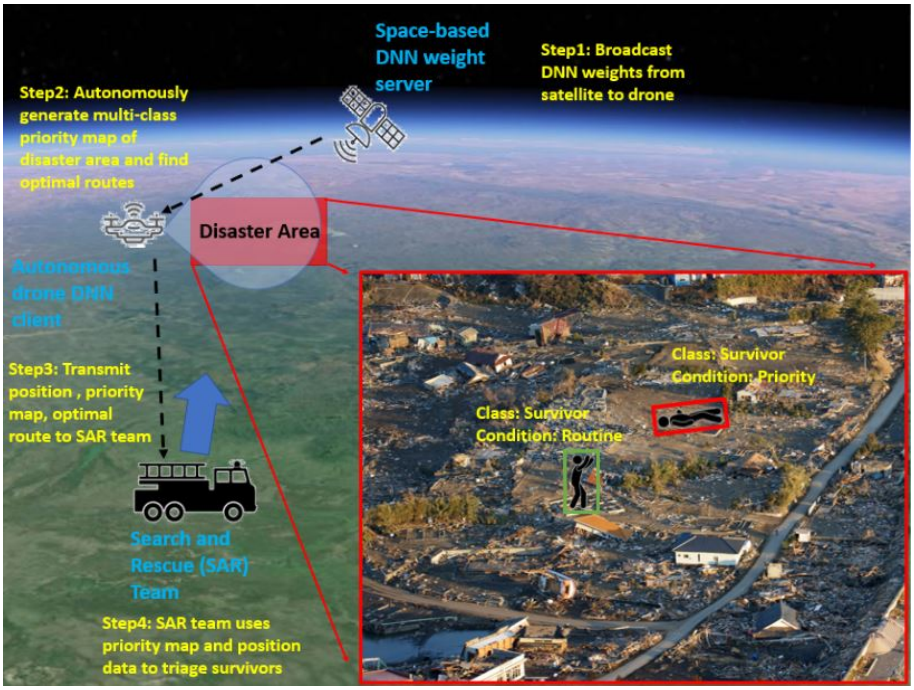


Figure 4-1: Applying the netcast ONN to SAR operations. The proposed system integrates a space-based DNN weight server that contains the parameters that will be used by the image processing networks located on the SAR drone. The weights are broadcast from the satellite to the drone, which then uses a large CNN to autonomously generate a high-dimensional priority map of the disaster area for the SAR team. Such data could include information about the survivor locations, medical status, as well as the optimal route to each person. The generated map is then transmitted from the UAV to the ground team, which then uses the data to inform how they choose to conduct their SAR ground operations.

team. This map would contain information including the location of survivors, their status, and the optimal route to each survivor. This would give the SAR team enough information to formulate an optimal rescue plan and help as many survivors as possible.

AI-Enabled Autonomous Underwater Vehicles (AUVs) for Exploring the Ocean Floor

Another possible netcast application would be using AI in small autonomous underwater vehicles (AUVs) to help scientists generate a detailed map of the ocean floor. Exploring the seafloor has many benefits, one of which is the extraction of minerals and metals - including copper, zinc, nickel, gold, silver, and phosphorus - which are

used in many everyday applications such as modern cell phones and electric vehicles [5]. Despite the importance of seafloor mapping, only 20 percent of the ocean floor has been mapped in detail after nearly a century of effort [50]. In large part, the slow progress is due to the fact that data taken over massive regions of the ocean floor takes months to process by hand [42]. In the past few years, autonomous underwater vehicles (AUVs) have been applied to this large-scale data collection task and have generated very promising results. For example, AUVs were applied in [42] to collect 1.3 million high resolution images of the Hydrate Ridge seafloor. While the scientists in [42] were able process the image data in a few days, the team still had to generate the seafloor map manually by clustering the AUV data. The processing time for similar expeditions could be reduced further still if AUVs were equipped with DNN technology that would enable quick and accurate image processing with minimal human supervision. This is where a netcast-based system can be developed and applied, as illustrated in figure 4-2. In the proposed system, an expedition team would come up with an initial set of object classes of interest (ex. mineral deposit, biological site, etc...) that would be classified using a pod of AUVs. Each AUV would be assigned a sub-area within the region being explored and would map that area using image semantic segmentation with a DNN. Semantic segmentation produces a map where pixels corresponding to the object of interest are distinguished from background pixels that do not contain the object of interest. The subregion semantic maps would then be combined into a map of the entire region, which could then be used to conduct more targeted sampling at certain sites (ex. mineral deposits). Netcast is well suited for this application because it allows small AUVs to operate with minimal energy cost and no need to store onboard DNN weights.

Automated Facility Safety Inspections

Another potential netcast application is in the field of safety inspections for large commercial and industrial companies. In this case, autonomous drones (UAVs) can be paired with a mobile DNN weight server to conduct autonomous safety inspections of potentially hazardous facilities (ex. pressurized chemical storage units). Facility

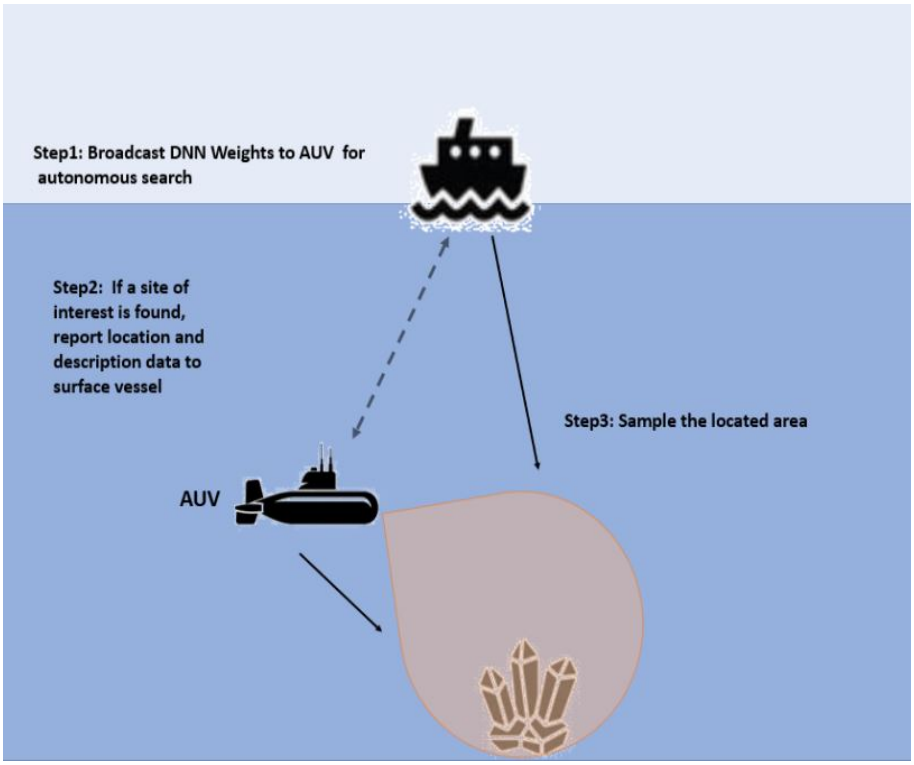


Figure 4-2: Applying netcast to mapping the ocean floor. In this system, a surface vessel transmits the DNN weights to a fleet of AUVs, each of which is assigned a subregion within the total area being explored. Each AUV uses an onboard DNN to perform semantic segmentation within its subregion. The individual subregion semantic maps are combined into a single map that shows where different sites of interest are located. This map allows scientists to then conduct more targeted and time-efficient sampling of selected sites (ex. mineral deposits).

safety regulations have been developed by both the Environmental Protection Agency (EPA) as well as the Occupational Safety and Health Administration (OSHA). While the EPA is primarily concerned with protecting local communities, OSHA focuses on employee health and safety [47]. The main body of legislation governing safety standards for hazardous chemicals in the U.S. is the Toxic Substances Control Act (TSCA). This legislation defines the safety standards required of U.S. companies storing hazardous chemicals as well as the methodology used by the EPA to conduct on-site inspections to monitor compliance [51]. In the past few years, companies have begun to research how drones can be applied to make the inspection process less expensive and risky for employees. Using drones saves companies the costs associated with scaffold construction and maintenance, and it eliminates any safety liabilities faced by company workers [56]. The application of netcast to automated inspection is depicted in figure 4-3. Here, it is important to note that reliably imaging the interior regions of storage tanks will require the DNN weight signal to take a piecewise-linear path from the server to the client. This path can be facilitated through the use of a secondary UAV to reroute the weight signal from the server to the client UAV that actually performs the fault detection. This automates the safety inspection process in a way that both saves costs for businesses and minimizes the risk faced by company workers.

Using autonomous UAVs for safety inspections provides a significant financial and safety benefit to companies, but it could also paint an inaccurate picture of how safe a company's facilities actually are. Regardless of who controls the inspection technology (the company or the EPA), the autonomous system is subject to some degree of error that could have significant impacts on the lives and welfare of both employees and the public. In this case, the autonomous fault detection system could either miss a fault (false negative) or incorrectly report a fault that does not actually exist (false positive). A false negative means that an inspection incorrectly determines that an unsafe chemical facility is safe. This error rate is not just a number - it could impose significant health and safety hazards on the public and workers. The negative impacts associated with different error types are discussed later on in section 4.3.

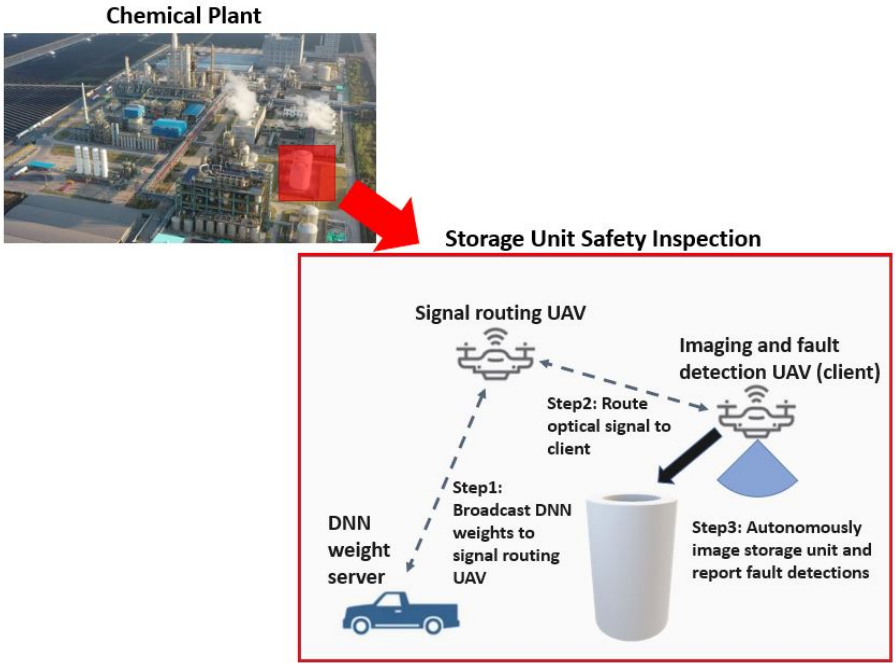


Figure 4-3: Application of the netcast ONN to plant safety inspection using autonomous UAVs. Note that, because netcast uses optical signals, line-of-sight limitations apply and this means that two drones will be needed to reliably image the inside of storage units like the one shown in the figure. DNN weights are broadcast from a mobile ground platform and are routed to the client using an intermediate routing drone that redirects the optical signal as needed. With the optical weight signal routed to the client, the imaging and fault detection UAV is then able to autonomously image the inside of a storage unit and identify any faults relative to a pre-defined detection threshold.

4.2.2 Military Applications

While the above applications use netcast in the civilian sphere, this section discusses possible ways in which netcast can be applied to support military operations. While the application of machine learning to military technology may seem like a far-off concept, autonomous weapons are projected to enter the modern battlefield in the 2026-2032 time frame [79]. As the U.S. military changes its focus from counter insurgency operations to great power competition, AI will play an increasingly important role in transforming the battlefield in every aspect from the front lines to the command centers [36]. AI's increased relevance to the modern battlefield is reflected in the recent budget demands from the military. Recently, the Pentagon has requested a tenfold increase in the Navy's autonomous systems spending along with a doubling of the Army's budget for developing autonomous ground systems [44]. Among the numerous possible military uses that netcast could be applied to, specific focus is given to autonomous targeting as presented in the two sections below.

UAV-Based Autonomous Targeting

As described in training circular 2-19.01, military intelligence units are responsible for providing accurate, relevant, and timely information to the battlefield commander, who is the final authority to authorize lethal force against targets [40]. Currently, military intelligence companies (MICOs) employ RQ-7B Shadow UAVs to monitor the battlefield and locate relevant enemy targets. While the Shadow UAV has proven effective, it is limited to a range of 125 km (line of sight) and does not possess any onboard AI capabilities. Because the RQ-7B uses signals in the same frequency range as netcast (the C-band), the netcast hardware can be integrated into the Shadow UAV to significantly increase its operating range and enable autonomous targeting within a defined area. This autonomous targeting system is illustrated and described in figure 4-4. As shown in the figure, the weight elements are broadcast from the space-based weight server to the UAV, which then uses onboard DNNs to generate a map of the battlefield that identifies enemy locations, capabilities, and current activities. This

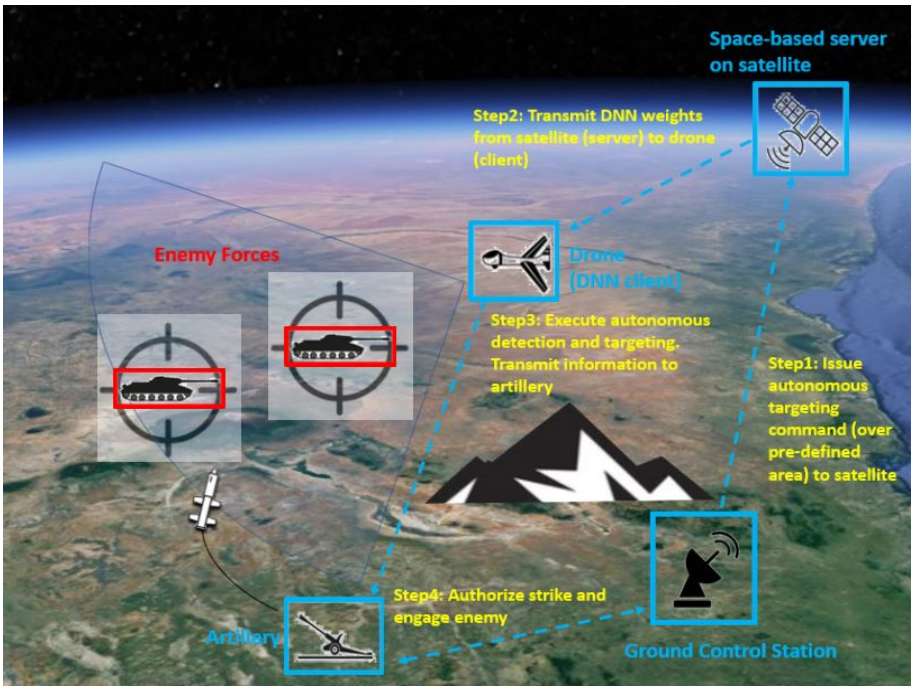


Figure 4-4: Applying netcast to autonomous targeting for military operations. The DNN weight server lives on a satellite that broadcasts DNN parameters to a UAV. Using onboard deep convolutional networks, the UAV is then able to identify and track targets autonomously in real-time. As the UAV tracks enemy targets, relevant target information including target description, activity, status, and location are sent to an artillery unit that verifies and engages the enemy with the appropriate munitions.

information is then sent to an artillery unit that confirms and engages the targeted enemy. Placing a DNN on a UAV will enable real-time targeting over a significantly longer range since the DNN weights are broadcast from a space-based weight server. This will help give U.S. forces the intelligence edge needed to match or outperform near peer military rivals in ground combat operations.

AI-Enabled Autonomous Underwater Vehicles (AUVs) for Naval Targeting Operations

The underwater analog of the previous military application is illustrated in figure 4-5. In this application, DNNs are integrated into a fleet of AUVs that can be used to autonomously target enemy submarines that would otherwise be undetectable to surface warships. Submarines are difficult to detect because they can hide under layers of cold water that act as shields against surface ship hull-mounted sonar (HMS) sensors [76].

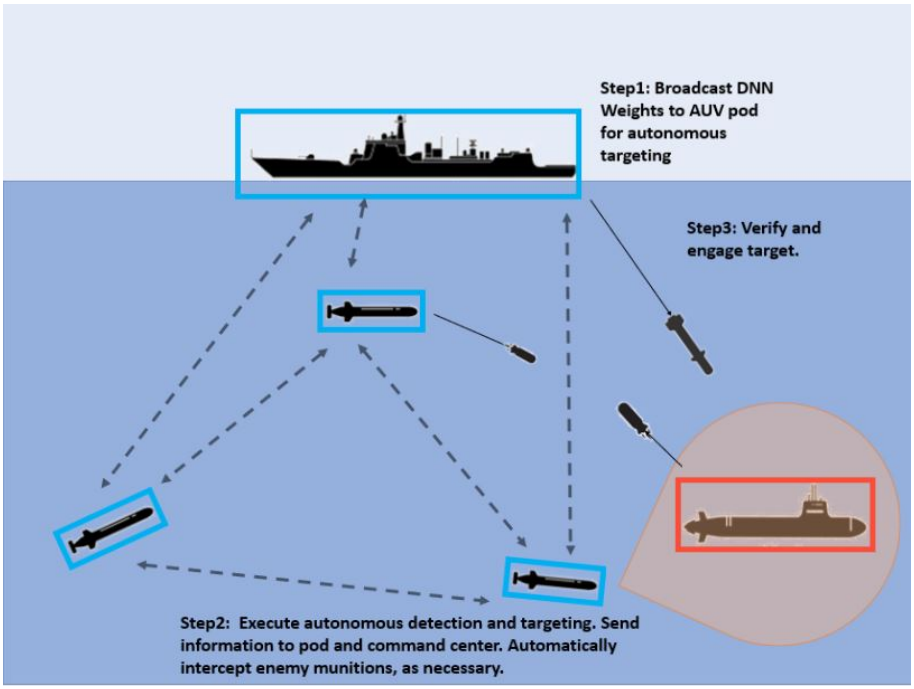


Figure 4-5: Applying netcast to autonomous targeting using AUVs. In this application, the DNN weight server resides in the surface vessel, and the DNN weights are broadcast to the fleet of AUVs. The AUV fleet is then able to autonomously detect, target, and track enemy submarines.

Even more advanced detection mechanisms like magnetic anomaly detection (MAD) are also unreliable since great power competitors like Russia have built submarines using non-magnetic titanium [76]. Recent methods have leveraged the use of optical signals in the blue-green region of the visible spectrum [75]. While this method provides a way to more reliably detect submarines, surface-based systems are limited in range to relatively shallow regions. Here, netcast can be applied to transmit DNN weights to AUVs that use optical signals to autonomously detect and target enemy submarines. Deploying a fleet of AUVs combines the computational power of DNNs with an autonomous targeting platform that is relatively small, fast, and difficult to detect or hit. Note that the AUVs operate in a similar manner as a swarm in that they share targeting information with both each other as well as the surface vessel that houses the command center.

4.3 Policy Implications of Netcast: Establishing Metrics of Acceptable Performance

The above applications are only a few examples of how netcast can be applied to next-generation AI for edge computing. While the netcast hardware demonstrates significant promise in helping AI solve increasingly complex problems, these technological promises come with concerns that should be carefully considered. One major concern associated with AI-enabled systems is that such systems make errors, even if they are able to outperform humans. Even the most accurate machine vision network for a simple task like MNIST classification will have some error rate greater than zero, and this error rate can have negative impacts depending on the application. For example, the search and rescue network proposed above might incorrectly classify an injured person as being uninjured, which might result in the person being denied life-saving treatment. On the military side, the autonomous targeting network might mislabel a friendly unit as an enemy one, which could result in friendly fire and fratricide. Given the concerns raised by DNN error rates in the applications proposed above, it is important to address the question: **how can acceptable performance metrics for netcast-based AI systems be defined, characterized, and applied?**. This question can be addressed by (1) looking at different types of error associated with netcast-based systems, (2) defining clear and tangible metrics that operationally define what "acceptable performance" means, (3) categorizing the different applications based on their error types and magnitudes, and (4) figuring out how the different performance metrics should be distributed over the different application types. Each of these questions is detailed in the sections that follow. Section 4.3.1 describes the types of error that a netcast-based AI system might make. Section 4.3.2 introduces and defines numerical detection threshold (P) and level of human control as two concrete metrics that describe acceptable performance in AI systems. Using the different error types, section 4.3.3 categorizes the netcast application space using two dimensions: (1) the magnitude of error impacts and (2) the distribution of negative impacts between different error types. Finally, section

4.3.4 proposes a distribution of the acceptable performance metrics over the different application categories.

4.3.1 Error Types in a Netcast-Based AI System

This section looks at how error rates in netcast can result in detrimental real-world impacts for each of the application areas discussed in section 4.2. Before delving deeper into the specific types of error, however, it is first necessary to map the complexity of modern machine vision classification to a simpler problem: binary (0-1) classification. One common thread among each of the applications presented in section 4.2 is that each maps input visual data (a picture of a disaster, ocean floor, chemical plant, or battlefield) to some semantic label that detects the presence or absence of some key property of the scene in question. For the search and rescue application, this label might be a high-dimensional vector that encodes the location, status, and optimal paths to various survivors. For autonomous targeting, the label might contain the target identity (enemy or friendly), weapon class (tank, artillery, etc...), and location. Regardless of the specifics, each application considered in this thesis is some sort of image classification problem, where the exact dimension and type of output is determined by user needs. In order to analyze the decisions made by an AI system in a simple way without sacrificing generality, the rest of this chapter considers the simplest and most fundamental vision task: binary classification. In this task, there are only two possible states: the positive class (ex. the weapon in question is an enemy) or the negative class (ex. the weapon in question is not an enemy). General image classification can be mapped to binary classification with no loss in generality because any k-way classification problem can be decomposed into a series of k binary classification problems. With this task in mind, figure 4-6 shows the two types of error that a binary classification network can make: false positive error and false negative error. A false positive occurs when a network incorrectly predicts the positive class, while a false negative happens when the network predicts a negative class incorrectly.

In some cases, a false positive and false negative may have similar or equal detri-

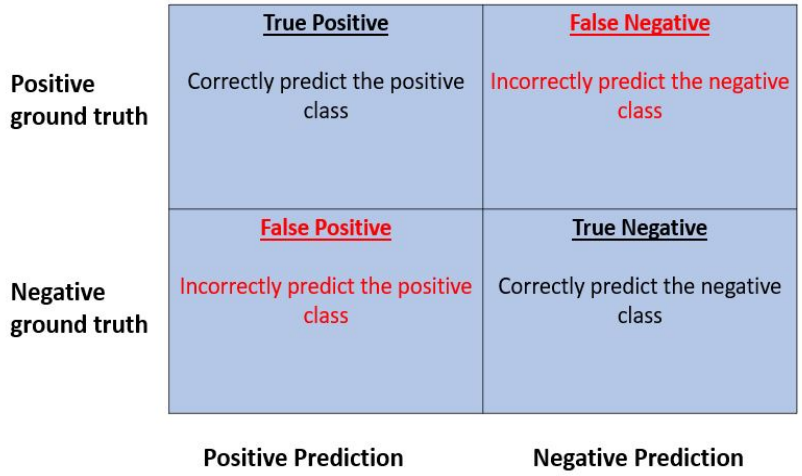


Figure 4-6: Two error types associated with a simple binary classification problem: false positives and false negatives.

mental impacts. For the autonomous search and rescue system, the positive class means that the person in question is injured and requires critical aid, while the negative class implies that the person does not require aid. Thus, a false positive means that the network classifies an uninjured person as needing critical aid. Assuming that the aid time and resources are limited, this likely will mean that other people who are actually in need of aid are less likely to receive the help they need. In contrast, in a false negative, the network determines that an injured person doesn't need aid. As a result, the person in need of help will likely not receive it. Assuming limited aid resources and time, false positive and false negative errors in this case both result in similar outcomes: those in need of aid do not receive it.

Another example where false positive and false negative errors have similar impacts is the application of autonomously mapping the ocean floor. Here, the positive class means that the region imaged contains material of interest (rare earth metals, biological hotspots, etc...), while the negative class indicates that the region contains no materials of interest. A false positive means that the netcast system scans an area without materials of interest but inaccurately reports the area as containing the material of interest. In a false negative, the netcast system sees an area that contains materials of interest, but incorrectly labels that area as not being of interest. While

the false positive and false negative error types in this case are again different, they both produce the same final result: the research team fails to acquire the material that they were searching for. While there may be edge cases where a false positive happens to be of some benefit to the researchers (ex. they find a useful material different than what they were searching for), these edge cases are by far the exception rather than the rule. Thus, similar to the search and rescue application, the final results produced by both error types in this case are very similar.

In contrast to the above examples, false positive and false negative errors can have very different impacts. The civilian application that illustrates this point is automated safety inspections. In this case, both false positive and false negative errors yield suboptimal outcomes for either the company or the local community, but the potential severity of the impacts varies greatly depending on the type of error that is made. In this case, the positive class is defined as the plant being unsafe, while the negative class corresponds to the plant being safe. Thus, a false negative corresponds to an error in which the netcast system images an unsafe facility and incorrectly classifies it as being safe. A false positive occurs when the system incorrectly concludes that a safe facility is not safe. In this case, a false positive will not be optimal for the company because it forces company executives to needlessly facilitate a human inspection and possibly conduct unnecessary repairs. While false positives are not ideal, false negatives in this case can be deadly. If the AI system incorrectly concludes that an unsafe plant is safe, then the undetected safety hazard could endanger the lives and welfare of the plant workers and the local community. Thus, in this case, a false negative carries significantly more weight than a false positive even if both error types are be equally likely to occur.

The errors discussed so far have analyzed the relative distribution of negative impacts between false positive and false negative errors. Another important consideration is that the sheer magnitude of impacts from error rates (regardless of the type) can also vary drastically between different applications. For example, error rates in the military application of autonomous targeting carry significant consequences: friendly fire. In contrast, error rates in the ocean floor mapping application do not

carry the same magnitude of impact because human lives and safety do not directly rely on the AI system.

4.3.2 Defining Acceptable Performance Metrics for Netcast AI Systems

The goal of this section is to define the metrics that determine acceptable levels of performance in netcast-based AI systems. While it is common in the deep learning community to define system performance solely as output accuracy, this metric often fails to give a holistic picture of how well systems actually perform [72]. Instead of using accuracy, this chapter uses a more nuanced metric that captures a network's confidence in its prediction as well as human operators' level of trust in the network's prediction. Recall from section 4.3.1 that this chapter has mapped generic image classification to binary classification. This means that a network's confidence level for its prediction is defined by a threshold probability (P), where any output probability greater than P yields a positive class prediction [11]. For example, if $P = 0.6$, then the network will need to output a probability greater than or equal to this value in order to predict the positive class. This means that the network must be at least 60 percent confident in its prediction. Having concretely defined the confidence of an AI system as its detection threshold (P), the degree to which human operators trust an AI system can be expressed through the level of human control. There are three regimes of human control over AI systems in the literature: human-in-the-loop, human-on-the-loop, and human-out-of-the-loop [16]. A human-in-the-loop system requires a human to have maximal control so that any decision that the AI makes can be manually overwritten. Human-on-the-loop allows AI to make low-level decisions autonomously, but any major decisions that have moral ramifications (ex. firing a weapon) require approval from the human operator. Human-out-of-the-loop leaves the human operator with no control of the AI system and allows the autonomous decision capabilities to progress unchecked [16].

These two metrics - detection threshold and human control - quantify the level

of acceptable performance that a netcast AI system must achieve in an application. For example, specifying a detection threshold of $P = 0.99$ with a human-in-the-loop means that the AI system in question needs to be at least 99 percent sure of the decisions that it makes, with all of these decisions being approved by the human operator. At the other extreme, a detection threshold of $P = 0.1$ with a human-out-of-the-loop means that the AI system only needs to be 10 percent sure of its decisions with no human involvement needed.

Although the detection threshold and level of human control aren't the only metrics that can be used, they can help policymakers get a general idea of how to regulate the architecture and human-machine interfaces associated with future tech applications that use netcast or similar ONN technology. As seen in section 4.3.1, different applications present different health and safety risks, each of which calls for a different level of human control and detection confidence. For example, human-in-the-loop control over AI is less important in a system designed for underwater exploration than in an autonomous military targeting system that could result in the taking of human life. While this thesis has focused on only five applications, the general space of potential netcast use cases is almost limitless. This motives a general taxonomy that categorizes netcast's diverse application space, which is done in section 4.3.3 below.

4.3.3 Using Error Types and Magnitudes to Categorize Netcast Applications

Having discussed the two error types in netcast applications (false positive and false negative) as well as the two metrics that will be used to define acceptable performance standards (threshold and level of human control), this section categorizes the netcast applications along two dimensions: impact of failure and distribution of impacts. The impact of failure describes the degree to which either a false positive or a false negative harms the lives and welfare of people. For example, the impact of failure associated with autonomous targeting is high because lives could be harmed by either false

I. High Impact, Equal Distribution	II. Low Impact, Equal Distribution
Example: Autonomous targeting, Autonomous search and rescue	**Example**: Autonomous mapping of ocean floor (mineral detection)
III. High Impact, Unequal Distribution	**IV. Low Impact, Unequal Distribution**
Example: Autonomous safety inspections	**Example**: Autonomous mapping of ocean floor (undersea cable detection)

Figure 4-7: Categorization matrix for netcast's application space. The two dimensions that sort this matrix are impact of failure and the distribution of impacts.

positive or false negative targeting errors. In contrast, the impact of failure associated with mapping the ocean floor is low because errors (whether they be false positives or false negatives) are unlikely to directly threaten the health and welfare of the researchers involved. The second dimension used to categorize netcast applications is the distribution of impacts, which measures the amount of harm potentially caused by a false positive versus a false negative error. For example, as discussed in section 4.3.1, autonomous search and rescue operations have a relatively equal distribution of impacts. This is because both false positive and false negative errors end in similar consequences: a person who needs aid doesn't get it. In contrast, autonomous safety inspections are an example with a highly unequal distribution of impacts. A false positive is an inconvenience for a company, while a false negative constitutes a public and employee safety hazard.

Using the two dimensions listed above allows the general space of netcast applications to be partitioned into the matrix shown in figure 4-7. While the matrix contains the netcast applications discussed in this chapter, it can be applied to any potential future application of either netcast or a similar ONN deep learning technology. As shown in the figure, the application space is partitioned according to the magnitude of impact that occurs from error rates as well as the impact distribution between false positive and false negative errors. Using this matrix to categorize netcast applica-

tions allows the performance metrics from section 4.3.2 to be applied to each different application category, which is done in section 4.3.4 below.

4.3.4 Applying Acceptable Performance Metrics to Netcast's Application Categories

Having defined the metrics of acceptable performance and partitioned netcast's application space using figure 4-7, the acceptable performance standards for each application type can now be determined. **The goal of this chapter is not to establish fixed, exact numerical standards for netcast-based systems but rather to give policymakers a general distribution of the performance metrics based on the application type**.

It is helpful to consider the two dimensions that sort the matrix that categorizes netcast's applications. Intuitively, netcast applications that are high-impact have significant effects on the people associated with those applications, which means that humans should exercise a high degree of control over the technology. Maintaining human-in-the-loop control over high-impact AI systems is important, not only for the moral reason that human lives are on the line, but also because of two pragmatic considerations: public support for AI and liability law.

An important element that influences how technology is brought from the lab into a functioning society is the way in which the technology is perceived by the public and its future users. An example that illustrates this point is autonomous weapons. Although the technology needed to develop and field autonomous weapons has been around for years, the technology has yet to make a major appearance on the battlefield largely due to the lack of trust from military commanders and the general public, some of whom view AI-enabled weapons as "killer robots" [8]. The attempt by 30 countries to preemptively ban autonomous weapons is largely rooted in distrust, which stems from the perceived lack of human control over the technology [37]. Those who are in favor of developing autonomous weapons have emphasized the critical importance of integrating meaningful human control that places lethal autonomous

decision making under careful human supervision [6]. This example illustrates a more general principle: people do not trust technologies that can affect human lives without appropriate human control. In order for the applications discussed in this chapter to progress from paper to practice, both the operators of the technology as well as the general public will need to possess a baseline level of trust in the technology. Establishing this trust will be much easier if the high-impact technology is monitored and controlled by human operators.

Another important reason why high-impact technology should be subject to strict human control is the way in which legal liability is assigned following a system error. Because it is impossible to assign legal liability to a machine, deploying a fully autonomous system that could impact human life gives rise to a very difficult legal problem: who to blame when the system makes an error. If a fully autonomous system makes a mistake, the operator usually can't be held liable because the system was functioning on its own. The designer usually can't be blamed assuming that the autonomous system was not developed with intentionally malign purposes. While tort law could be applied to recompense victims in a civilian context, this sidesteps the question of assigning legitimate moral blame to the relevant party. In addition, tort law does not apply to military applications, virtually all of which are likely to be high-impact. Having a human closely involved with high-impact AI systems makes the assignment of liability in the case of an error much more straightforward.

Having seen why high-impact AI systems should be subject to a high degree of human control, the second dimension of the matrix in figure 4-7 should be considered: the distribution of impacts. This dimension has a relatively straightforward relationship to the detection threshold metric. Namely, applications that have unequally distributed impacts between false negative errors and false positive errors should have a detection threshold that favors the class with the smaller negative impact. For example, in the autonomous safety inspection example, if the positive class means that the facility has a safety fault, then the AI detection system should have a threshold value that is small and therefore biased in favor of predicting the positive class. Biasing the detection threshold, in effect, avoids the highly negative impact

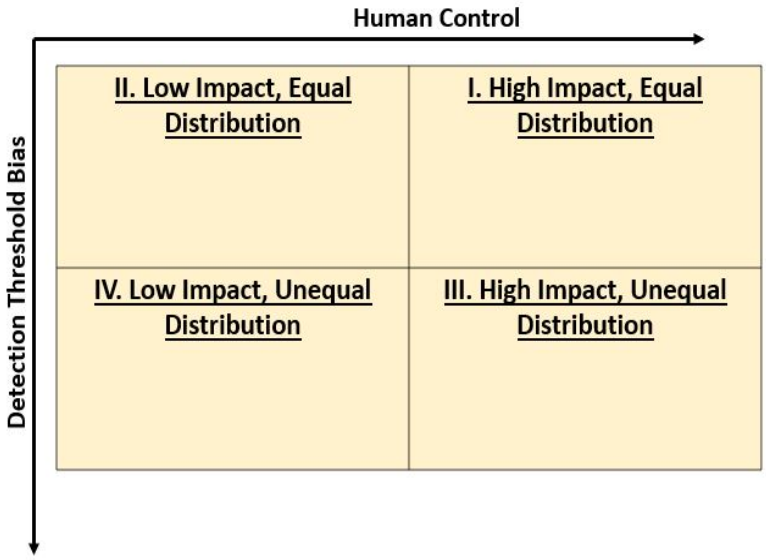


Figure 4-8: Applying acceptable performance metrics over netcast's four application categories.

associated with incorrectly labeling an unsafe facility as safe. Of course, the detection threshold should try to jointly optimize the network output accuracy (the number of true positives and true negatives) but should also place more probability mass on the side of a positive detection.

The acceptable performance levels for different netcast application types are illustrated in figure 4-8. Note that the detection threshold bias given on the vertical axis maps to an actual detection threshold value (P) depending on how the positive and negative classes are defined. Therefore, moving in the positive direction of the detection threshold bias axis can be thought of as stepping through a series of threshold values (P) that bias the predictions to favor either false positives or false negatives as desired. The point of the figure is two-fold. First, netcast applications with an unequal distribution of impacts should have their detection thresholds biased towards either a false positive or a false negative error, whereas such biasing is not as relevant to applications with an even distribution of impacts. Second, applications with high magnitude impacts of error (ex. military applications) should prioritize a high degree of human control regardless of how the impacts of different error types are distributed.

4.3.5 Netcast Policy Implications: Conclusion

While the netcast technology opens the door to a potentially infinite number of future applications, two general characteristics can be used to categorize netcast's application space: the impact of system error and the distribution of negative impacts associated with different error types. The metrics of detection threshold and human control can be used to get a more concrete picture of the performance standards that a netcast-based AI system can be held to. While the detection threshold describes how confident the system is with its own predictions, the level of human control defines how confident the system operator is regarding the AI's decision. Using the application categories from figure 4-7 as well as the two performance metrics, the standards that a netcast-based AI system should achieve can be described with two principles. First, AI systems that significantly impact human life and welfare should be subject to a high degree of human control. Second, systems with a skewed impact distribution between different error types should have detection threshold values that are biased to favor the less damaging outcome. ONN technology like netcast has the potential to transform the way in which AI is applied across numerous fields, and this technological promise should be accompanied by a careful consideration regarding the moral, ethical, and policy implications that accompany this cutting edge technology.

Appendix A

Proof that Uniform Samples From an Arbitrary Distribution's Inverse CDF Generate Random Draws From that Distribution

Let $X \sim$ Uniform[0,1] be a uniform random variable. We want to sample an arbitrary PDF, call it $f_Y(y)$ where the random variable Y is related to X through the deterministic function g. That is, $Y = g(X) \sim f_Y(y)$. We consider the function $g(x) = F_Y(x)^{-1}$ where x is an instantiated value of the random variable X and F is the inverse CDF of the random variable Y. If the function g satisfies the definition of the CDF - that $P(Y \leq y) = F_Y(y)$, then we know that g generates samples from the distribution of Y as desired. To see that g does indeed sample from $f_Y(y)$ as desired, the function can be substituted into the definition of the CDF to obtain the following.

$$y = g(x) = F_Y(x)^{-1} \rightarrow Y = g(X) = F_Y(X)^{-1} \tag{A.1}$$

$$P(Y \leq y) = P(F_Y(X)^{-1} \leq y) = P(X \leq F_Y(y)) \tag{A.2}$$

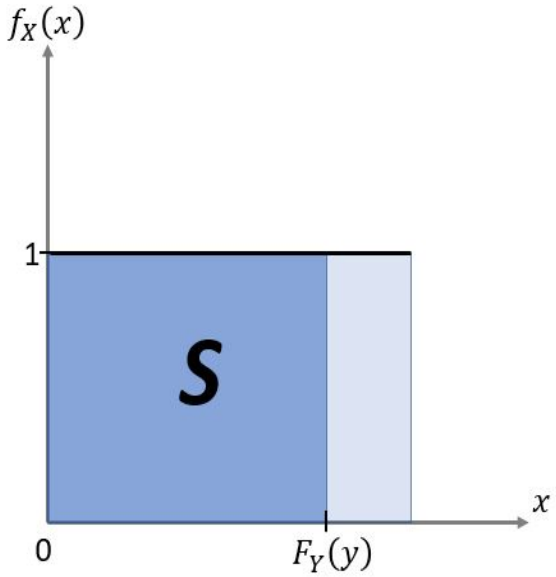


Figure A-1: Plot of the Probability Density Function of X, $f_X(x)$. The CDF is obtained by integrating over the region S. Note that $F_Y(y)$ is a deterministic number and thus is the upper limit of the integration in x.

Since Y and X are continuous random variables, the CDF can be expressed as an integral over the region S that is bounded as shown in figure A-1. From the figure and using equation A.2, we get...

$$P(X \leq F_Y(y)) = \int_S f_X(x)dx = \int_0^{F_Y(y)} f_X(x)dx \tag{A.3}$$

$$P(X \leq F_Y(y)) = \int_0^{F_Y(y)} 1dx = F_Y(y)P(Y \leq y) = F_Y(y) \tag{A.4}$$

which is exactly the definition of the CDF. This means that the function g, which is the inverse CDF $F_Y(x)^{-1}$ samples correctly from the distribution $F_Y(y)$.

Appendix B

Chapter 2 Supplemental Figures

The figures below are supplemental material from chapter 2.

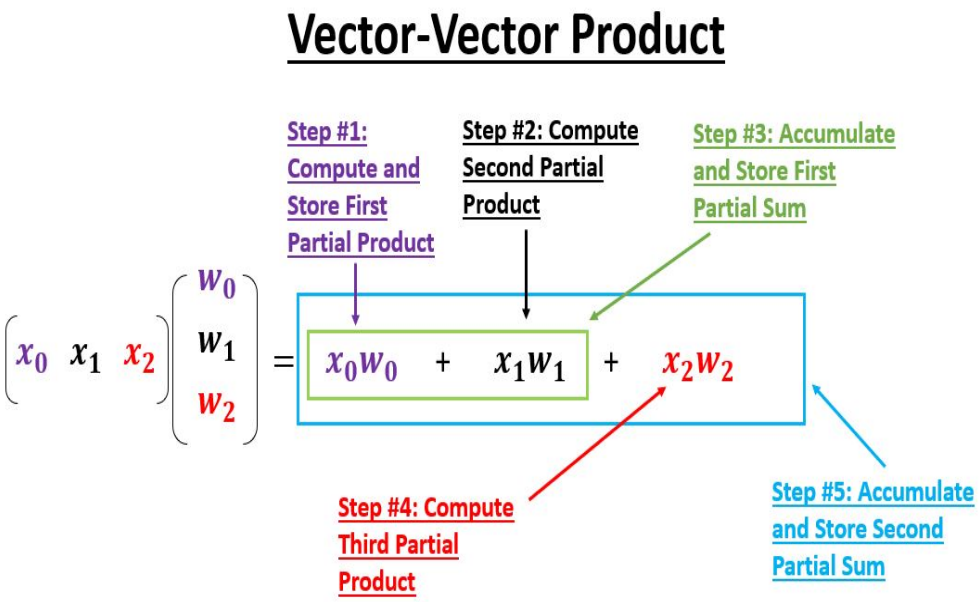
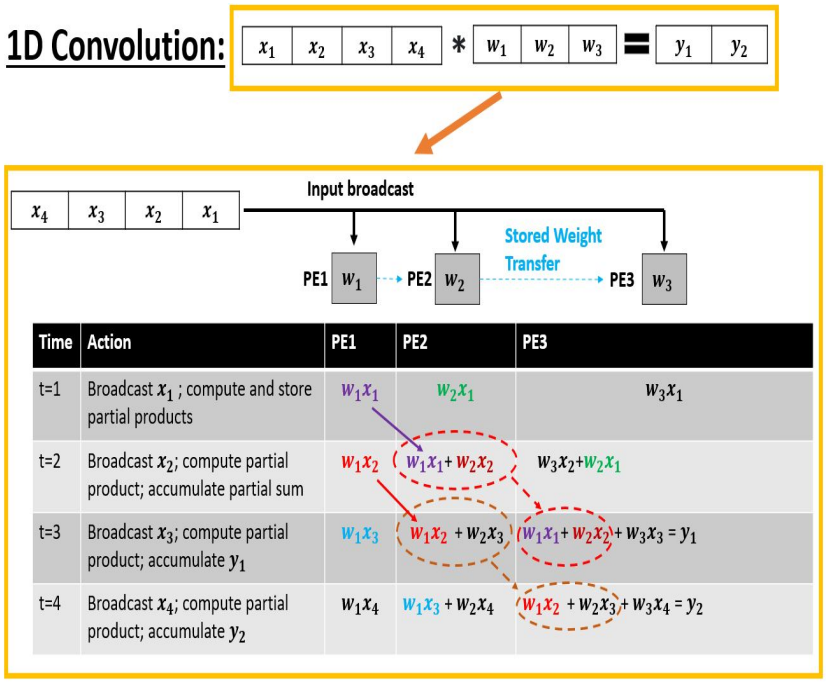


Figure B-1: Example illustrating the multiply-and-accumulate (MAC) operation in a vector-vector product.



Time	Action	PE1	PE2	PE3
t=1	Broadcast x_1 ; compute and store partial products	w_1x_1	w_2x_1	w_3x_1
t=2	Broadcast x_2; compute partial product; accumulate partial sum	w_1x_2	$w_1x_1+w_2x_2$	$w_3x_2+w_2x_1$
t=3	Broadcast x_3; compute partial product; accumulate y_1	w_1x_3	$w_1x_2+w_2x_3$	$w_1x_1+w_2x_2+w_3x_3=y_1$
t=4	Broadcast x_4; compute partial product; accumulate y_2	w_1x_4	$w_1x_3+w_2x_4$	$w_1x_2+w_2x_3+w_3x_4=y_2$

Figure B-2: Example of weight stationary computation. A 1d input vector x is convolved with a 1d weight kernel w. the table shows the running values in each PE as the computation is performed over time. During each time step i, input x_i is broadcast to each PE and the corresponding partial products are computed. From $t = 1$ onward, processing engine PE_j transfers its stored result from time step $i-1$ to PE_{j+1}. This results in a series of partial sums from which the desired output values y_1 and y_2 are computed as depicted.

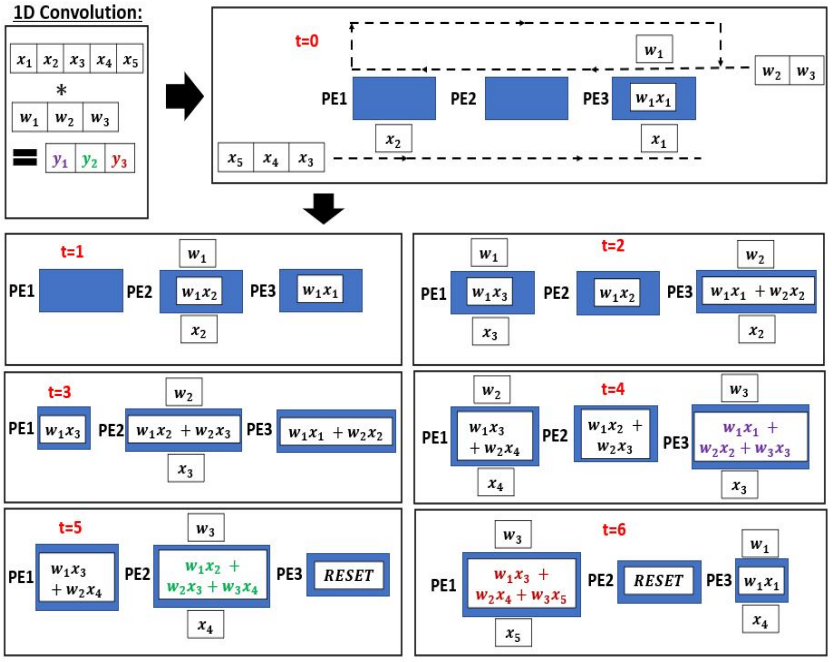


Figure B-3: Example computation using an output stationary dataflow. In this example, x is being convolved with w to form y (top left part of figure). The top right part of the figure shows the output stationary dataflow where weights and inputs move in opposite directions along the PE array. Each time a weight and input arrive at the same PE in the array, they are multiplied and accumulated to the current value being stored in that PE. The bottom part of the figure shows how the 1d convolution is performed over 6 time steps.

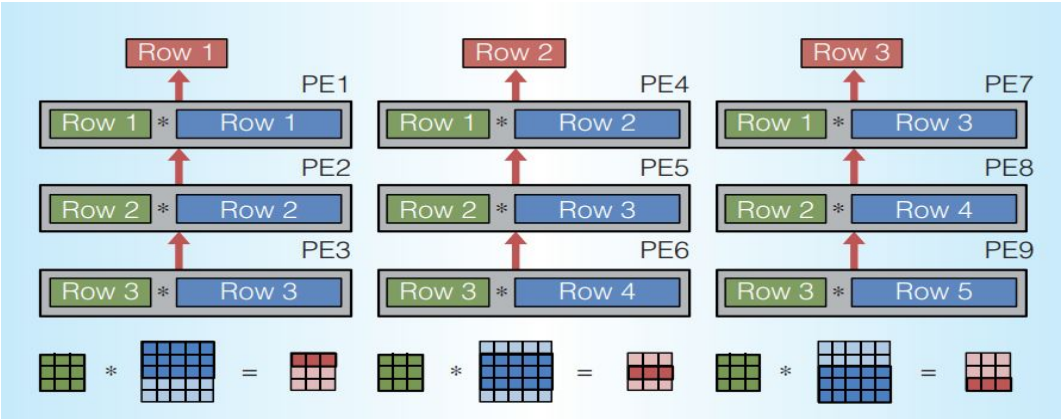


Figure B-4: Use of 1d convolutional primitives to compute a 2d convolution [85].

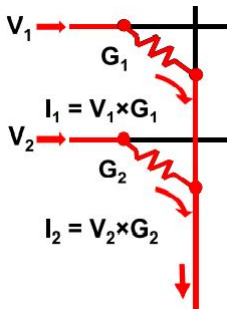


Figure B-5: Illustration of programmable resistive elements (memristors) for non-volatile, high density memory. Note the weight stationary nature of the memory where the resistors' conductances G_1, G_2 encode the weights that are multiplied by inputs encoded as voltages V_1, V_2. Applying Kirchhoff's current law, it can be seen that the output partial sum is given by $I_{out} = I_1 + I_2 = V_1G_1 + V_2G_2$ [30].

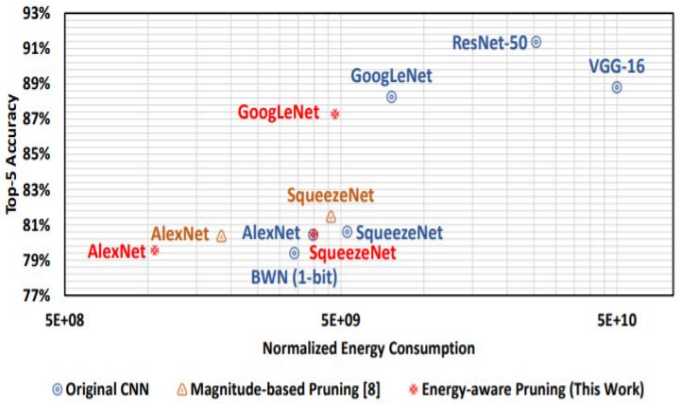


Figure B-6: Comparison between energy costs with and without pruning in GoogLeNet, SqueezeNet, and AlexNet [71].

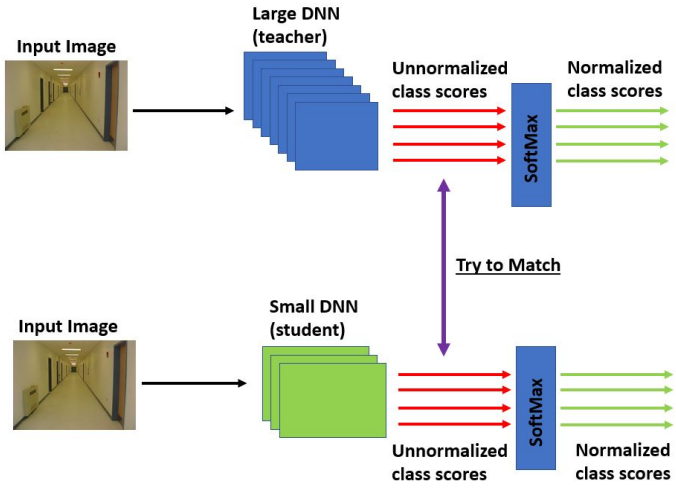


Figure B-7: Knowledge distillation is used to train a small DNN (student) to output the same accuracy as a much larger DNN (teacher). The key point in this method is that training the student network directly on the data without the teacher network results in a lower output accuracy.

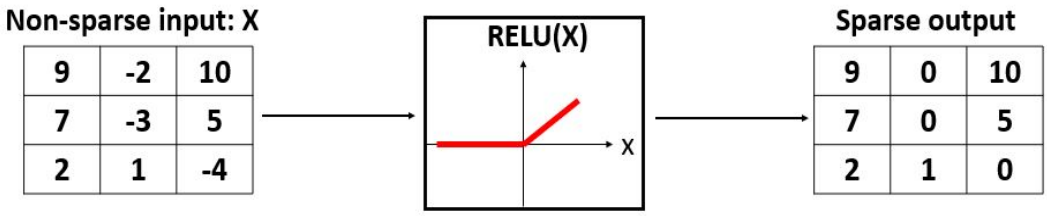


Figure B-8: Using a RELU nonlinearity maps non-sparse matrices to sparse ones, which helps increase the energy efficiency of DNN hardware. For example, the hardware can be configured such that any MACs that correspond to a zero element activation are skipped, thus allowing the network to run with fewer memory reads for weight access [81].

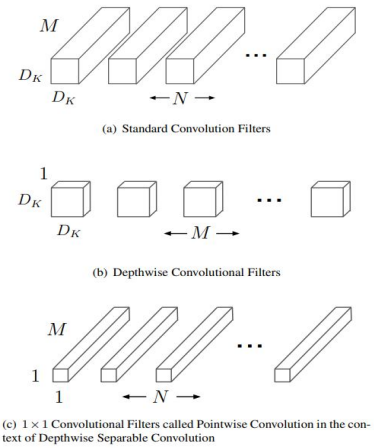


Figure B-9: MobileNet uses a convolution factorization technique where the standard convolution operation (as depicted in a) is broken up into two separate operations that decrease the computational burden of implementing convolution in hardware. This convolution method is called depth-wise separable convolution and consists of two parts: (1) depth-wise convolution as depicted in b that is applied to each channel separately followed by (2) a 1x1 point-wise convolution that forms a linear combination over the channels to form the final output of the convolution operation.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure B-10: A high-level architectural description of MobileNet. For each convolutional layer, the filter sizes and the intermediate feature map sizes are tracked.

Model	1e-4 Accuracy	Million Mult-Adds	Million Parameters
FaceNet [25]	83%	1600	7.5
1.0 MobileNet-160	79.4%	286	4.9
1.0 MobileNet-128	78.3%	185	5.5
0.75 MobileNet-128	75.2%	166	3.4
0.75 MobileNet-128	72.5%	108	3.8

Figure B-11: Comparing the number of MAC operations in various MobileNet variants versus the number of MAC operations required for a standard convolution operation in FaceNet.

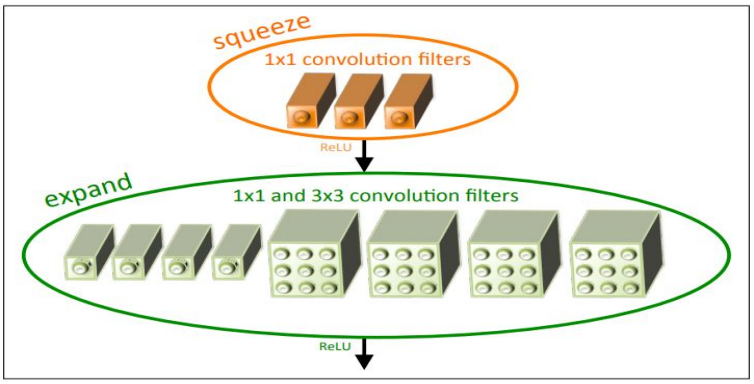


Figure B-12: The fire module that forms the computational basis for the SqueezeNet neural network. Three hyper-parameters define each fire module: s_{1x1} (the number of 1x1 convolutional kernels in the squeeze layer), e_{1x1} (the number of 1x1 convolutional kernels in the expansion layer), and e_{3x3} (the number of 3x3 convolutional kernels in the expansion layer).

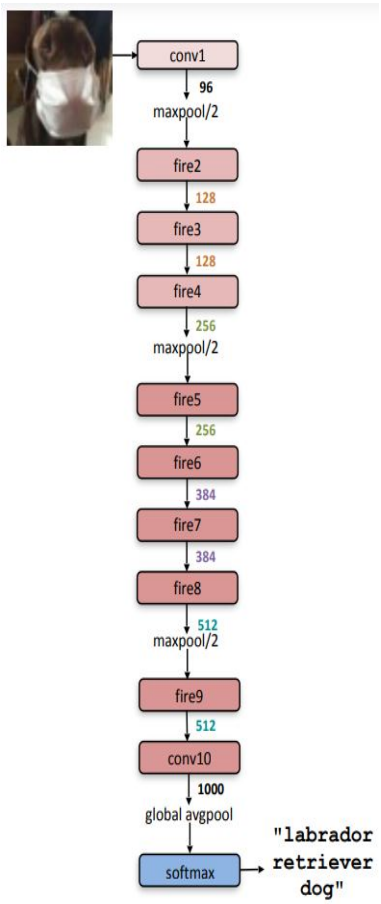


Figure B-13: Architectural description of SqueezeNet. The structure consists of a single standard convolution followed by a series of fire modules that ends with another standard convolutional layer and a Softmax activation layer. Max pooling and average pooling are used to downsample intermediate feature maps to the appropriate resolution.

CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	**50x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	**363x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	**510x**	57.5%	80.3%

Figure B-14: A side-by-side comparison between different methods of compressing AlexNet and the SqueezeNet method. As can be seen from the table, the SqueezeNet architecture offers a significant improvement in the level of model compression relative to other comparable methods while still maintaining model accuracy.

Bibliography

[1] Ilya Sutskever Alex Krizhevsky and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS)*, 2012.

[2] Dimitri P. Bertsekas and John N. Tsitsiklis. *Introduction to Probability, 2nd Edition*. Athena Scientific, 2008.

[3] John C. Blitch. Artificial intelligence technologies for robot assisted urban search and rescue. *Expert Systems with Applications*, 11(2):109, 1996.

[4] Kun Wang Bowen Zhang, Huaxi Gu and Yintang Yang. A novel conv acceleration strategy based on logical pe set segmentation for row stationary dataflow. *IEEE Transactions on Computers*, 2021.

[5] Coastal, Marine Hazards, and Resources Program. Seafloor minerals. Technical report, United States Geological Survey, 2019.

[6] Adam Cook. Taming killer robots- giving meaning to the 'meaningful human control' standard for lethal autonomous weapon systems.

[7] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005.

[8] Neil Davison. A legal perspective: Autonomous weapon systems under international humanitarian law.

[9] Biology Online Dictionary. Axon terminal url:`https://www.biologyonline.com/dictionary/axon-terminal`.

[10] Patrick Doherty and Piotr Rudol. A uav search and rescue scenario with human body detection and geolocalization. In *Proceedings of the 20th Australian Joint Conference on Artificial Intelligence*, 2007.

[11] MIT EECS. 6.862 applied machine learning course notes ch8 notes available on mit open course ware: `https://openlearning.mit.edu/courses-programs/mit-opencourseware`.

[12] Adam Teman et al. International technology roadmap for semiconductors (itrs). In *Proceedings of Semiconductor Industry Association*, 2013.

[13] Alexander Sludds et al. Wavelength multiplexed ultralow-power photonic edge computing. arXiv preprint, March 2022.

[14] Andrew G. Howard et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arxiv pre-print, April 2017.

[15] Balmukund Mishra et al. A hybrid approach for search and rescue using 3dcnn and pso. *Neural Computing and Applications*, 33, 2021.

[16] Doris Xin et al. Accelerating human-in-the-loop machine learning: Challenges and opportunities. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, 2018.

[17] Fang Liu et al. A survey on edge computing systems and tools. In *Proceedings of the IEEE Vol. 107, No. 8*, 2019.

[18] Forrest N. Iandola et al. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. arxiv pre-print, November 2016.

[19] Gopika Premsankar et al. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, 5(2):1275, 2018.

[20] Guobin Chen et al. Learning efficient object detection models with knowledge distillation. In *Proceedings of Conference on Neural Information Processing Systems*, 2017.

[21] H. Zender et al. Conceptual spatial representations for indoor mobile robots. *Robotics and Automation Systems*, 56(6):493, 2008.

[22] Hengshuang Zhao et al. Pyramid scene parsing network.

[23] Herbert Bay et al. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346, 2008.

[24] Krishna Shankar et al. A learned stereo depth system for robotic manipulation in homes. arxiv pre-print, 2021.

[25] Liang-Chieh Chen et al. Semantic image segmentation with deep convolutional nets and fully connected crfs. In *Proceedings of International Conference on Learning Representations*, 2015.

[26] Lin Wang et al. Service entity placement for social virtual reality applications in edge computing. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018.

[27] Ryan Hamerly et al. Large-scale optical neural networks based on photoelectric multiplication. *Physical Review X*, 9(021032), 2019.

[28] Ryuhei Hamaguchi et al. Effective use of dilated convolutions for segmenting small object instances in remote sensing imagery.

[29] Tommso Zanotti et al. Smart logic-in-memory architecture for low-power non-von neumann computing. *IEEE Journal of the Electron Devices Society*, 8, 2020.

[30] Vivienne Sze et al. Efficient processing of deep neural networks: A tutorial and survey. In *Proceedings of the IEEE (Volume: 105, Issue: 12)*, 2017.

[31] Weisong Shi et al. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5), 2016.

[32] William (Red) L. Whittaker et al. Robotics for assembly, inspection, and maintenance of space macrofacilities. In *Proceedings of AIAA Space Conference and Exposition*, 2000.

[33] W.J. Zhang et al. On definition of deep learning. In *World Automation Congress (WAC)*, 2018.

[34] Yunji Chen et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2014.

[35] O. Vinyals G. Hinton and J. Dean. Distilling the knowledge in a neural network. In *Proceedings of Conference on Neural Information Processing Systems*, 2014.

[36] Jim Garamone. Esper says artificial intelligence will change the battlefield.

[37] Stephen Goose and Mary Wareham. The growing international movement against killer robots. *Harvard International Review*, 37(4):28, 2016.

[38] Ryan Hamerly. Netcast: Low-power edge computing with optical neural networks via wdm weight broadcasting. MIT Quantum Photonics Group (QPG) Notes, August 2020.

[39] Kaiming et al He. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[40] Department of the Army Headquarters. Military intelligence (mi) company and platoon reference guide.

[41] M. Horowitz. Computing's energy problem (and what we can do about it),. *IEEE ISSCC Dig. Tech. Papers*, 2014.

[42] Schmidt Ocean Institute. Artificial intelligence guides rapid data-driven exploration of changing underwater habitats mapped onto one of the world's largest multiresolution 3d photogrammetric reconstruction of the seafloor.

[43] Somin Lee. Jaehyeong Sim and Lee-Sup Kim. An energy-efficient deep convolutional neural network inference processor with enhanced output stationary dataflow in 65-nm cmos. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):87, 2020.

[44] Michael Klare. Pentagon asks more for autonomous weapons.

[45] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 2004.

[46] Kent H. Lundberg. Noise sources in bulk cmos.

[47] Thomas O. McGarity. Substantive and procedural discretion in administrative resolution of science policy questions: Regulating carcinogens in epa and osha. *Georgetown Law Journal*, 67(3):729, 1979.

[48] Eugene M.Izhikevich. Resonate-and-fire neurons. *Neural Networks*, 14(6), 2001.

[49] online documentation `https://mmclassification.readthedocs.io/en/latest/index.html`.

[50] National Oceanic and Atmospheric Administration. How much of the ocean have we explored? Technical report.

[51] U.S. Environmental Protection Agency Office of Enforcement and Compliance Assurance. *Compliance Monitoring Strategy for the Toxic Substances Control Act (TSCA).*

[52] A. Olivia and A. Torralba. Modelling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145, 2001.

[53] Cyrill Stachniss Oscar Martinez-Mozos and Wolfram Burgard. Supervised learning of places from range data using adaboost. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005.

[54] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145, 1999.

[55] D.C. Pande Rajashree Narendra, M.L. Sudheer. Susceptibility of integrated circuits to electrostatic discharge. *International Journal of Advancements in Research Technology*, 1(4), 2012.

[56] Tarek Rakha and Alice Gorodetsky. Review of unmanned aerial system (uas) applications in the built environment: Towards automated building inspection procedures using drones. *Automation in Construction*, 93, 2018.

[57] E. Neftci W. Wan G. Cauwenberghs S. B. Eryilmaz, S. Joshi and H.-S. P. Wong. Neuromorphic architectures with electronic synapses. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*, 2016.

[58] J. Tran S. Han, J. Pool and W. J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of Conference on Neural Information Processing Systems*, 2015.

[59] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way. December 2018.

[60] Raghavemder Sahdev. Place recognition system for localization of mobile robots. Master's thesis, Birla Institute of Technology and Science, 2015.

[61] Mahadev Satyanarayanan. The emergence of edge computing. *IEEE Computer Journal*, 50(1):30, 2017.

[62] Mikhail Usvyatsov Shengyu Huang and Konrad Schindler. Indoor scene recognition in 3d. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[63] Weisong Shi and Schahram Dustdar. The promise of edge computing. *IEEE Computer Journal*, 49(5):78, 2016.

[64] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[65] Jeffrey S. Vetter Sparsh Mittal and Dong Li. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524, 2015.

[66] D. Srinivas and K. Hanumaji. Analysis of various image feature extraction methods against noisy image: Sift, surf and hog. *Journal of Engineering Sciences*, 10(2):32, 2019.

[67] University of Alabama Strategic Communications Directory. Developing 'third wave ai' to improve human performance. University of Alabama Article web article, November 2019.

[68] Vivienne Sze and Joel Emer. Course notes for 6.s082/6.888 hardware architecture for deep learning.

[69] Lei Tai and Ming Liu. Deep-learning in mobile robotics - from perception to control systems: A survey on why and why not. arXiv preprint, August 2016.

[70] Chen Wu Tharam Dillon and Elizabeth Chang. Cloud computing: Issues and challenges. In *24th IEEE International Conference on Advanced Information Networking and Applications*, 2010.

[71] Yu-Hsin Chen Tien-Ju Yang and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[72] Antonio Torralba, Phillip Isola, and William F. Freeman. *The Tiny Book of Computer Vision.* 2020.

[73] Y. Wang Y. Chen W. Wen, C. Wu and H. Li. Learning structured sparsity in deep neural networks. In *Proceedings of Conference on Neural Information Processing Systems*, 2016.

[74] Sonia Waharte and Niki Trigoni. Supporting search and rescue operations with uavs. In *Proceedings of the 2010 International Conference on Emerging Security Technologies*, 2010.

[75] Ashraf F. El-Sherifb Walid Gomaaa and Yasser H. El-Sharkawy. Underwater laser detection system. In *Proceedings of SPIE - The International Society for Optical Engineering*, 2015.

[76] Ryan White. Why are submarines so hard to find?

[77] L. Wilson. Energy versus data integrity trade-offs in embedded high-density logic compatible dynamic memories. In *Proceedings of 2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.

[78] Piotr Wozniak. Scene recognition for indoor localization of mobile robots using deep cnn. In *Proceedings of International Conference on Computer Vision and Graphics*, 2018.

[79] Austin Wyatt. Charting great power progress toward a lethal autonomous weapon system demonstration point. *Defence Studies*, 20(1):1, 2020.

[80] XenonStack. Difference between edge computing vs cloud computing? XenonStack web article, November 2021.

[81] J. Emer Y.-H. Chen, T. Krishna and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127, 2017.

[82] J. S. Denker Y. LeCun and S. A. Solla. Optimal brain damage. In *Proceedings of Conference on Neural Information Processing Systems*, 1989.

[83] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *Proceedings of International Conference on Learning Representations*, 2016.

[84] Joel Emer Yu-Hsin Chen and Vivienne Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators),. *IEEE Micro*, 37(3):12, 2017.

[85] Vivienne Sze Yu-Hsin Chen, Joel Emer. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37(3):12, 2017.

[86] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Proceedings of the European Conference on Computer Vision*, 2014.

[87] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 2018.