IMAGE CLASSIFICATION WITH ENERGY EFFICIENT HADAMARD NEURAL NETWORKS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL AND ELECTRONICS ENGINEERING

By

Tuba Ceren Deveci

January 2018

IMAGE CLASSIFICATION WITH ENERGY EFFICIENT HADAMARD NEURAL NETWORKS

By Tuba Ceren Deveci

January 2018

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

A. Enis Çetin(Advisor)

Ömer Morgül

Emre Akbaş

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

IMAGE CLASSIFICATION WITH ENERGY EFFICIENT HADAMARD NEURAL NETWORKS

Tuba Ceren Deveci
M.S. in Electrical and Electronics Engineering
Advisor: A. Enis Çetin
January 2018

Deep learning has made significant improvements at many image processing tasks in recent years, such as image classification, object recognition and object detection. Convolutional neural networks (CNN), which is a popular deep learning architecture designed to process data in multiple array form, show great success to almost all detection & recognition problems and computer vision tasks. However, the number of parameters in a CNN is too high such that the computers require more energy and larger memory size. In order to solve this problem, we investigate the energy efficient network models based on CNN architecture. In addition to previously studied energy efficient models such as Binary Weight Network (BWN), we introduce novel energy efficient models. Hadamard-transformed Image Network (HIN) is a variation of BWN, but uses compressed Hadamard-transformed images as input. Binary Weight and Hadamard-transformed Image Network (BWHIN) is developed by combining BWN and HIN as a new energy efficient model. Performances of the neural networks with different parameters and different CNN architectures are compared and analyzed on MNIST and CIFAR-10 datasets. It is observed that energy efficiency is achieved with a slight sacrifice at classification accuracy. Among all energy efficient networks, our novel ensemble model outperforms other energy efficient models.

Keywords: Image classification, deep learning, convolutional neural networks, energy efficiency, ensemble models.

ÖZET

VERİMLİ ENERJİLİ HADAMARD SİNİR AĞLARI İLE GÖRÜNTÜ SINIFLANDIRMASI

Tuba Ceren Deveci
Elektrik Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: A. Enis Çetin
Ocak 2018

Derin öğrenim, görüntü sınıflandırması, nesne tanıma, nesne algılama gibi görüntü işleme görevlerinde son yıllarda önemli başarılar elde etmiştir. Çoklu dizi şeklindeki verileri işlemek üzerine tasarlanmış popüler bir derin öğrenim mimarisi olan evrişimli sinir ağları (CNN), algılama ve tanıma problemleri ve bilgisayarla görme görevlerinin neredeyse tamamında büyük başarı göstermiştir. Ancak CNN'deki yüksek parametre sayısı bilgisayarlar için daha fazla enerji ve daha büyük bellek boyutu gerektirmektedir. Bu sorunu çözmek amacıyla enerji verimli ağ modellerini inceliyoruz. Daha önceden ortaya atılmış olan İkili Ağırlık Katsayılı Ağlar (BWN) gibi enerji tasarruflu modellere ek olarak yeni enerji tasarruflu modeller sunuyoruz. Hadamard-dönüşümlü görüntü ağları (HIN), BWN'nin bir varyasyonu olup, girdi olarak Hadamard dönüşümü ile sıkıştırılmış görüntüleri kullanmaktadır. İkili Ağırlık ve Hadamard-dönüşümlü Görüntü Ağı (BWHIN) BWN ve HIN'i birleştirilmesiyle özgün bir verimli enerjili model olarak geliştirilmiştir. Farklı parametreler ve farklı CNN mimarileri ile sinir ağlarının performansları karşılaştırılmış ve MNIST ve CIFAR-10 veri setleri üzerinde analiz edilmiştir. Enerji verimliliğinin sınıflandırma doğruluğunda küçük bir fedakarlık yapılarak sağlandığı gözlenmiştir. Verimli enerjili ağlar arasında, yeni topluluk modelimiz diğer modellerden daha iyi performans göstermiştir.

Anahtar sözcükler: Görüntü sınıflandırması, derin öğrenme, evrişimli sinir ağları, enerji verimliliği, topluluk modelleri.

Acknowledgement

First and the foremost, I would like to express my gratitude and sincere thanks to my supervisor Prof. Dr. A. Enis Çetin for his suggestions, guidance and support throughout the development of this thesis.

I also would like to thank Prof. Ömer Morgül and Asst. Prof. Dr. Emre Akbaş for accepting to be a member of my thesis committee and reviewing my thesis.

I would like to thank Tübitak Bilgem İltaren for enabling to complete my M.Sc. study and my colleague for supporting me in every possible way.

I am also thankful to Damla Sebhan Bozbay, who is my best friend of all time, and Selin Yücesoy, who is always there to listen and share everything, for their support and love for all those years since high school. I want to thank Tuba Kesten, who is the best colleague and the best travelling companion in my life, for her encouragement and understanding. I would love to thank Damla Sarıca, my precious working-out friend, and Ecem Bozkurt for making me love this university more. I want to thank Elmas Soyak for her friendship and our precious, enjoyable dialogues since bachelor years. I also would like to thank Merve Kayaduvar and Gökçe Öztürk Türker for giving me new perspectives and helping me get through my hard times. I am thankful to Güneş Sucu for helping me in Python language and Tensorflow library with a great knowledge as computer engineer.

Last but not least, I am and I always will be grateful to my parents and my brother for their life-long guidance, patience and love.

Contents

1 Introduction **1**

2 Literature Review & Background **5**

2.1 Basics of Neural Network 5

2.1.1 Activation Functions 6

2.2 Training of Neural Networks 8

2.2.1 Forward Propagation 9

2.2.2 Backpropagation 10

2.3 Regularization 11

2.4 Optimizers 13

2.5 Convolutional Neural Networks (CNN) 16

2.5.1 Convolutional Layer 19

2.5.2 Nonlinearity Stage 20

2.5.3 Pooling Layer 20

2.5.4 Fully Connected Layer 21

2.5.5 Softmax Layer 21

3 Energy Efficient Neural Networks 23

3.1 Introduction 23

3.1.1 Binary Weight Networks (BWN) 24

3.1.2 Hadamard-transformed Image Networks (HIN) 25

3.1.3 Combination of Models: Binary Weight & Hadamard Transformed Image Network (BWHIN) 28

3.2 Neural Network Architecture and Hyperparamaters 31

3.2.1 CNN Architectures 31

3.2.2 Weight and Bias Initialization 34

3.2.3 Mini-Batch Size 36

3.2.4 Learning Rate 37

3.2.5 Momentum 38

3.3 Implementation of the Architectures 39

4 Simulation and Results 42

4.1 Experiments on MNIST 43

4.1.1 Effect of Optimizers 44

4.1.2 Effect of Dropout 45

4.1.3 Effect of Activation Function on FC Layer 47

4.2 Experiments on CIFAR-10 48

4.3 Effect of Architectures 50

4.4 Comparison of Energy Efficient Neural Networks 51

5 Conclusion and Future Work **53**

A MNIST Results **62**

A.1 Test Accuracies 62

A.2 Training Accuracies 62

A.3 Training and Test Losses 62

B CIFAR-10 Results **66**

List of Figures

2.1 Perceptron model. 6

2.2 Activation functions sigmoid, tangent hyperbolic and rectified linear unit. 7

2.3 An example of a neural network with one hidden layer. 9

2.4 *Left:* A classical neural network with 2 hidden layers. *Right:* A thinned network after dropout is applied. 13

2.5 An example of 2-D convolution operation without kernel flipping [1]. 17

2.6 An example of standard CNN with its major components. 19

2.7 Examples of non-overlapping pooling. *Top:* Max-pooling operation. *Bottom:* Average-pooling operation. 21

3.1 Fast Hadamard Transform algorithm of a vector of length 8. . . . 28

3.2 Our approach to combine BWN and HIN: The architecture of BWHIN [2]. 30

3.3 *Top:* Strided convolution with a stride of 2. *Bottom:* Convolution with unit stride followed by downsampling. 32

3.4 Effects of different learning rates [3]. 37

3.5 Sample images of MNIST database. 39

3.6 Sample images of CIFAR-10 database [4]. 40

4.1 Test accuracy results for different optimizers. 45

4.2 Test accuracy results for different dropout probabilities. 47

4.3 Test accuracy results for different activation functions at FC layer. 49

4.4 Test accuracy results for different activation functions at FC layer. 50

List of Tables

1.1 List of Abbreviations 4

3.1 Model description of the two architectures for MNIST dataset. . . 33

3.2 Model description of the two architectures for CIFAR-10 dataset. 34

4.1 Test accuracy results (in percentage) for different optimizers. . . . 44

4.2 Test accuracy results (in percentage) for different dropout probabilities. 46

4.3 Test accuracy results (in percentage) for different activation functions at FC layer. 48

4.4 Test accuracy results (in percentage) for CIFAR-10 dataset. . . . 49

A.1 Test accuracy results (in percentage) for MNIST database. 63

A.2 Training accuracy results. 64

A.3 Results for training and test loss 65

B.1 Overall results for CIFAR-10 dataset. (Note that test accuracies are in precentages.) 67

Chapter 1

Introduction

Machine learning techniques have gained widespread use on digital image processing area with the revival of neural networks. Nowadays, artificial neural networks (ANN) have various applications on image processing, such as image classification, feature extraction, segmentation, object recognition and detection [5]. Deep learning is a more advanced and particular form of machine learning, which enables us to build complex models composed of multiple layers for large datasets. Deep learning methods have enhanced the state-of-the-art performance in object recognition & detection and computer vision tasks. Deep learning is also advantageous for processing raw data such that it can automatically find a suitable representation for detection or classification [6].

Convolutional neural network (CNN) is a specific deep learning architecture for processing data which is composed of multiple arrays. Images can be a good example of input to CNN with its 2D grid of pixels. Convolutional Neural Networks have become popular with the introduction of its modern version *LeNet-5* for the recognition of handwritten numbers [7]. Besides, *AlexNet*, the winner of ILSVRC object recognition challenge in 2012, aroused both commercial and scientific interest in CNN and it is the main reason of the intense popularity of CNN architectures for deep learning applications [8]. The usage of CNN in AlexNet obtained remarkable results such that the network halved the error rate

of its previous competitors. Thanks to this great achievement, CNN is the most preferred approach for most detection and recognition problems and computer vision tasks.

Although CNNs are suitable for efficient hardware implementations such as in GPUs or FPGAs, the training is computationally expensive due to the high number of parameters. As a result, excessive amount of energy consumption and memory usage make the implementation of neural networks ineffective. According to [9], especially matrix multiplications at the layers of a neural network consume too much energy compared to addition or activation function and becomes a major problem for mobile devices with limited batteries. As a result, replacing the multiplication operation becomes the main concern in order to achieve energy efficiency.

Many solutions are proposed in order to handle the energy efficiency problem. An energy efficient ℓ_1-norm based operator is introduced in [10]. This multiplier-less operator is first used in image processing tasks such as cancer cell detection and licence plate recognition in [11, 12]. Multiplication-free neural networks (MFNN) based on this operator are studied in [13–15]. This operator achieved good performance especially at image classification on MNIST dataset with multi-layer perceptron (MLP) models [14]. Han et al. reduces both the computation and storage in three steps: First, the network is trained to learn the important connections. Then, the redundant connections are discarded for a sparser network. Finally, the remaining network is retrained [9]. Using neuromorphic processors with its special chip architecture is another solution for energy efficiency [16]. In order to improve energy consumption, Sarwar et al. exploits the error resiliency of artificial neurons and approximates the multiplication operation and defines a Multiplier-less Artificial Neuron (MAN) by using Alphabet Set Multiplier (ASM). In ASM, the multiplication is approximated as shifting and adding in bitwise manner with some previously defined alphabets [17]. Binary Weight Networks are energy efficient neural networks whose filters at the convolutional layers are approximated as binary weights. With these binary weights, convolution operation can be computed only with addition and subtraction [18]. There is also a computationally inexpensive method called distillation [19]. A

very large network or an emsemble model is first trained and transfers its knowledge to a much smaller, *distilled* network. Using this small and compact model is much more advantageous in mobile devices in terms of speed and memory size. This method shows promising results at image processing tasks such as facial expression recognition [20].

In this thesis, we investigate novel energy efficient neural networks as well as previously studied energy efficient models. We firstly analyze the performance of Binary Weight Network (BWN) proposed in [18], whose weights at the convolutional layers are approximated to binary values, $+1$ or -1. As another energy efficient network model, we modify BWN so that the network has compressed images as inputs rather than original images. This network is called Hadamard-transformed Image Network (HIN). In order to preserve the energy efficiency, Hadamard transform is implemented by Fast Walsh-Hadamard Transform algorithm which requires only addition or subtraction [21]. Our main contribution is the combination of BWN and HIN models: Binary Weight and Hadamard-transformed Image Network (BWHIN). The combination is carried out after energy efficient layers, i.e. convolutional layers with two different averaging techniques. All of the energy efficient models are also examined with different CNN architectures. One of them (ConvoPool-CNN) contains pooling layers along with convolutional layers, while the other (All-CNN [22]) uses strided convolution instead of pooling layer [22]. We analyze the performance of the models on two famous image datasets MNIST and CIFAR-10. While working on MNIST, we also study the effects of certain hyperparameters on the classification accuracy of energy efficient neural networks.

This thesis includes five chapters and the outline of the thesis is as follows: In Chapter 1, the necessity for energy saving neural networks is already explained and related work is mentioned. Chapter 2 describes the basics of machine learning and gives an explanation of convolutional neural network. The conventional and our proposed models are introduced in Chapter 3 and the crucial hyper-parameter selections are also demonstrated. Chapter 4 presents the simulations and results based on the proposed networks and previously determined criteria. In chapter 5, thesis concludes with overall results and future work is mentioned.

Acronym	Definition
ADAM	Adaptive Momentum
ANN	Artifical Neural Network
BWN	Binary Weight Network
BWHIN	(Combined) Binary Weight & Hadamard-transformed Image Network
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
HIN	Hadamard-transformed Image Network
ILSVRC	ImageNet Large Scale Visual Recognition Competition
lr	Learning Rate
MFNN	Multiplication-Free Neural Network
MLP	Multi-Layer Perceptron
NN	Neural Network
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent

Table 1.1: List of Abbreviations

List of abbreviations which are commonly used in this thesis is given in Table 1.1.

Chapter 2

Literature Review & Background

This chapter describes the theoretical basis of the neural networks and the training procedure in detail. Regularization methods and optimization techniques are also described. Afterwards, one of the most popular deep learning architectures, convolutional neural networks (CNN), are introduced to give a better understanding of both deep neural networks and energy efficient network models.

2.1 Basics of Neural Network

In 1958 [23], Rosenblatt proposed the perceptron as the first neuron model for supervised learning. This artifical model is inspired by the structure of a biological neuron cell and is still the basis for many neural network libraries [24]. The perceptron model is illustrated in Figure 2.1.

Input signals to this neuron k is denoted as $x_1, x_2, \ldots, x_m$ and output signal is y_k. Weight values are represented as $w_{k1}, w_{k2}, \ldots, w_{km}$ and b is the bias term. Perceptron sums the weighted input signal and the bias before the activation function. Since the output of the perceptron is a linear function of input, perceptron is considered as a linear classifier. A set of parameters for a neuron, $\boldsymbol{\theta}$ contains weight $\boldsymbol{w}$ and biases b. Parameters $\boldsymbol{\theta}$ is updated so that the neural

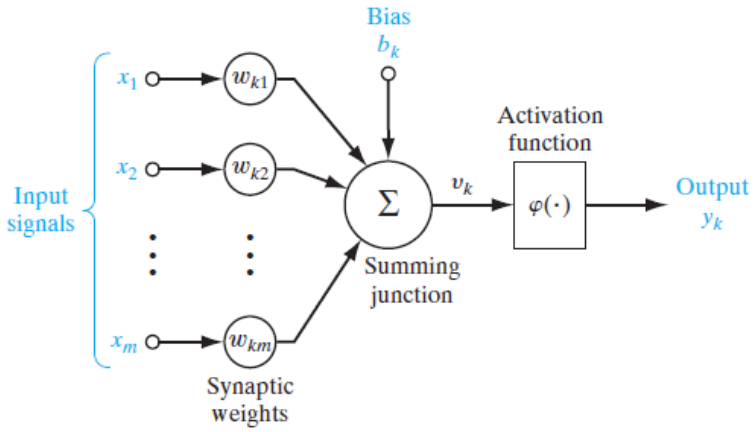

Figure 2.1: Perceptron model.[1]

network can learn to achieve a task. Equation 2.1 summarizes the perceptron model:

$$y = f(\sum_{j=1}^{m}(w_j x_j) + b) \tag{2.1}$$

where $f(\cdot)$ is a nonlinear activation function, which is described in detail with its examples in Section 2.1.1.

2.1.1 Activation Functions

The activation function, denoted as $f(x)$ in Equation 2.1, is a nonlinear function which computes the output of a neuron. There are various activation functions, some of the most popular activation functions will be mentioned here [24].

Sigmoid, also known as logistic function, $\sigma(x) = \frac{1}{1+e^{-x}}$ is a well-known activation function which was very popular in the 1980s when the neural networks are very small [1]. The output of the sigmoid is in the range of $[0, 1]$. The function saturates to 0 at large negative values and saturates to 1 at large positive values. However, saturated values cause the vanishing gradient problem. The gradient at

[1]Image retrieved from [24].

the saturated regions are almost zero and when the input to the activation function is too large, the gradient vanishes and the neuron "dies". Another drawback of the sigmoid is that it is not zero-centered [25].

Later, sigmoid is replaced by tangent hyperbolic function $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ which has better performance than sigmoid on neural networks. $Tanh$ is a scaled and shifted version of sigmoid, which can be also defined as $tanh(x) = 2\sigma(2x) - 1$. The output of $tanh$ is in the range of $[-1, 1]$. Unlike sigmoid, it is zero-centered and often converges faster than the standard logistic function [26]. Nevertheless, vanishing gradient problem still exists for $tanh$, since it also saturates at large positive and negative values.

A very popular activation function in modern deep learning architectures is rectified linear unit $ReLU(x) = max(0, x)$; which is a piecewise linear function. Although the exponential terms in sigmoid and tangent hyperbolic functions are computationally expensive, ReLU can be implemented very easily with a simple comparison. In practice, ReLU converges much faster (6 times faster) than both sigmoid and $tanh$ functions [8]. One of the flaws about ReLU is that it is zero for negative values; causes the zero gradient problem. If one chooses ReLU as the activation function, biases should be initialized with small positive numbers, such as 0.1, so that ReLU neurons will be activated at the beginning for the inputs in the training set. That could be the solution for the zero gradient problem.

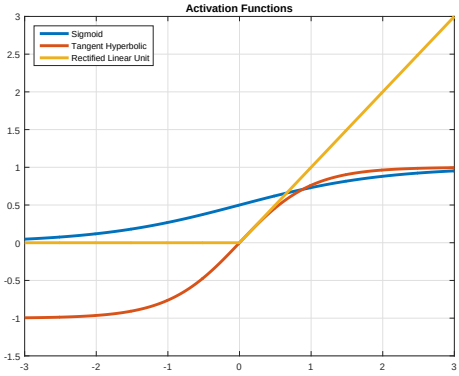

Figure 2.2: Activation functions sigmoid, tangent hyperbolic and rectified linear unit.

The variants of ReLU exist, such as softplus $f(x) = \ln(1+e^x)$ which is a smooth

approximation of ReLU, and leaky ReLU $f(x) = max(0.01x, x)$. Although softplus is differentiable for all values and has less saturating effect compared to ReLU, it has worse results than ReLU in practice [27]. Leaky ReLU is an alternative to ReLU in order to fix the zero gradient problem. The function has a small slope for negative values rather than being zero [28]. Leaky ReLU doesn't always give improved results over ReLU, hence one should be cautious about using this function.

Sigmoid, *tanh* and ReLU functions are shown in Figure 2.2. Their performance on energy-efficient neural networks will be investigated in Chapter 4.

2.2 Training of Neural Networks

Machine learning algorithms deal with many tasks such as classification, translation or detection. In order to solve such problems and learn a model, they first train the samples in the *training set*, then they evaluate the model on the *test set* which contains new and different samples from training set. Machine learning algorithms improve the parameters $\boldsymbol{\theta}$ such that the loss function between the correct output and the predicted output is minimized. The performance of the machine learning algorithm can be measured by its accuracy. Accuracy is determined by the proportion of the number of correct examples to the overall samples in the test set [1].

An example of an artifical neural network is illustrated in Figure 2.3. Neural networks involve input layer, output layer and hidden layers. While input layer is used to feed the input data to the network, the output layer generates the final output of the network. Hidden layers are placed between input and output layers and their number can be increased for a deeper network. The neurons in a layer behave like a perceptron which computes the affine transformation $f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$. Hence, such fully connected networks can be also called as multilayer perceptron (MLP) or deep feedforward networks. The number of neurons in a layer is optional and depends on the machine learning task.

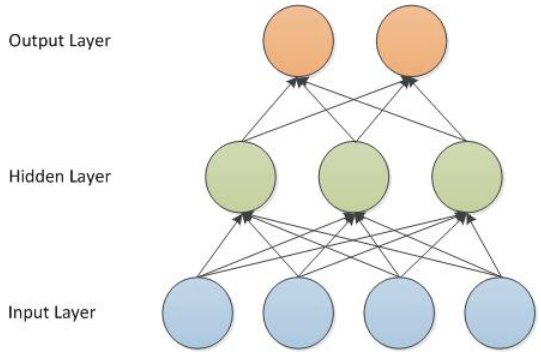

Figure 2.3: An example of a neural network with one hidden layer.

Training of a feedforward neural network consist of two main stages: Feed-forward phase and the back-propagation. The affine transformation is computed starting from the input layer and resulting signals pass through the hidden layers till the output layer at the forward propagation. At the output layer, a scalar cost is generated. In the backward pass, a gradient vector is computed by the aid of the cost function and calculates the error signals layer by layer in the backward direction. In backprop phase, parameters of the network (weights, biases) are successively update in the backward direction as well [24].

2.2.1 Forward Propagation

A vanilla network accepts an input $\boldsymbol{x}$ and computes the affine transformation described in Equation 2.1 through the network in the forward direction. It produces an output $\hat{\boldsymbol{y}}$ at the final layer. Let L be the number of layers and $l^{(i)}$ is the layer index of the i^{th} layer. Layer $l^{(i)}$ has $n^{(l)}$ neurons. If j represents the number of inputs to that layer and k is the number of output units, then the output of the first layer becomes:

$$\begin{aligned} v_k^{(1)} &= \sum_{j=1}^{m} W_{kj}^{(1)} x_j + b_k^{(1)} \\ y_k^{(1)} &= f(v_k^{(1)}) \end{aligned} \tag{2.2}$$

Output of the next hidden layers are computed similarly:

$$
\begin{aligned}
v_k^{(l)} &= \sum_{j=1}^{n^{(l-1)}} W_{kj}^{(l)} y_j^{(l-1)} + b_k^{(l)} \\
y_k^{(l)} &= f(v_k^{(l)})
\end{aligned}
\tag{2.3}
$$

After the feedforward activations are computed throughout the network, a scalar cost $L(\boldsymbol{\theta})$ is calculated in order to measure the error between the predicted output $\hat{\boldsymbol{y}}$ and correct output $\boldsymbol{y}$. There are two main functions to calculate the cost; mean squared error and cross-entropy cost function. When gradient-based optimization techniques are used, cross-entropy function gives better results than mean squared error in practice [1].

2.2.2 Backpropagation

Most learning algorithms use gradient-based optimization, which maximizes the likelihood $\mathcal{L}$, i.e. minimizes the negative log-likelihood $(-\ln(\mathcal{L}))$ by using gradients. This negative log-likelihood is per-example loss and denoted as $L(\boldsymbol{\theta})$. Thus, the objective of the gradient-based optimization becomes minimizing the cost function. Although traditional gradient-based algorithm calculates the loss over one sample, it is computationally more efficient to choose a minibatch from training data and average the loss function over the samples of the minibatch. Then the extended algorithm is called as Stochastic Gradient Descent Algorithm (SGD).

Gradients generates a vector which contains all the partial derivatives of a function with multiple variables. For example, if the partial derivative $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ is the derivative of $f(.)$ with respect to x_i at point $\boldsymbol{x}$, then the gradient is denoted by $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ and i^{th} element of the gradient is the partial derivative $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$. In order to minimize cost function, the gradient of the cost function is calculated with respect to parameters $\boldsymbol{\theta}$. $\boldsymbol{\theta}$ represents trainable parameters, mainly the weight and the bias. Backpropagation algorithm uses these gradients to update

the parameters and learn the model. The objective function of a minibatch $J(\boldsymbol{\theta})$ is calculated as:

$$L(\boldsymbol{\theta}) = \frac{1}{B}\nabla_{\boldsymbol{\theta}}(\sum_{i=1}^{B} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta})) \tag{2.4}$$

where B is the minibatch size. The gradient is estimated as:

$$\boldsymbol{g} = \frac{1}{B}\nabla_{\boldsymbol{\theta}}(\sum_{i=1}^{B} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta})) \tag{2.5}$$

The estimated gradient is used to update the parameters in the negative direction of the gradient. If the learning rate is denoted as ϵ, then SGD algorithm can be summarized as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g} \tag{2.6}$$

Initialization of the parameters is an important issue for the gradient-based algorithms. This topic will be analyzed in detail in Section 3.2.2. Other gradient-based optimizers will be also explained in Section 2.4.

2.3 Regularization

The performance of a machine learning algorithm can be understood by analyzing two major factors: First, it should be able to make the training error small. Second, the gap between training error and test error should be as small as possible. If these two factors cannot be achieved, the machine learning model will underfit or overfit, respectively. When the training error generated on the training set is not low enough, underfitting occurs. When the model cannot obtain a small generalization gap between the training and the test error, overfitting occurs [1].

As mentioned above, one of the central challenges about machine learning field is that to reduce the test error, while possibly getting an increased value of training error. In order to solve this problem and prevent the neural network from overfitting, there are many strategies known as regularization. Regularization is one of the most active research fields in machine learning and many forms of regularization techniques are already available for deep network models [1].

In the literature, many regularization methods are proposed. Some methods are based on limiting the capacity of the model by adding *penalty term* in the loss function. When the amount of data is limited, one can create additional data by shifting, scaling or rotating the original image and add those extra samples to the training set as a *dataset augmentation* technique. In addition, one can also add *noise* with infinitesimal variance to the inputs or to the weights in order to encourage the stability of the network. One can also *early stop* the model such that the algorithm stores the parameters obtained at the lowest validation error point and the model returns these parameters when the training algorithm is completed. Some forms of regularization combine several models as *ensemble neural networks* to reduce generalization error. *Unsupervised pre-training* can be also viewed as an unusual form of regularization [29]. *Batch normalization* is a major breakthrough in the regularization techniques. Minibatch of the activations in the input layer or hidden layers is normalized by substracting minibatch mean from each value in the minibatch and then dividing to the standard deviation of minibatch. As a result, the mean of minibatch becomes zero and its standard deviation becomes 1. This technique not only speeds up the convergence, but also makes the networks more robust to the parameter initialization and hyperparameter selection [30].

Dropout technique proposed by [31] is a powerful and computationally low-cost regularization technique which drops units (neurons) randomly. A visualization of the dropout approach is presented in Figure 2.4. As seen in the figure, the crossed units are dropped with all of its related connections. Which unit is going to be dropped out is chosen randomly. Dropout doesn't permanently remove a unit from the network. During training, a unit is present with probability p and has weight parameters $\boldsymbol{w}$. During testing, each unit in the layer is always present

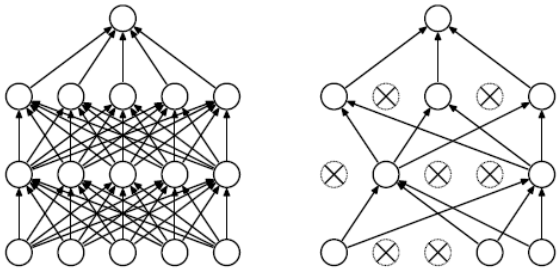

Figure 2.4: *Left:* A classical neural network with 2 hidden layers. *Right:* A thinned network after dropout is applied.[2]

but their weights become $p\boldsymbol{w}$. The probability of retention p can be determined by using a validation set or can be simply set at a value between $[0.5, 1]$. However, the optimum choice of p is usually closer to 1 [31].

We will use both ensemble models and dropout techniques for our algorithm. Ensemble models and model-averaging will be investigated in detail as one of the most crucial points in our algorithm in Section 3.1.3.

2.4 Optimizers

After the gradients are calculated in the backpropagation phase of the training, they are used to update parameters (weights and biases for a linear model). Optimizers update the parameters in the negative direction of the gradient so that the loss function is minimized. Various optimizers are introduced in the literature. Stochastic gradient descent (SGD), momentum algorithms, algorithms with adaptive learning rates and second-order methods are major optimization techniques for deep learning [1]. Gradient descent algorithm has already mentioned in Section 2.2. When a mini-batch is built by randomly choosing a certain number of training samples, then the gradient descent algorithm is given a new name as stochastic gradient descent (SGD). The parameters $\boldsymbol{\theta}$ are updated for SGD as shown in the equation 2.7:

[2]Image retrieved from [31].

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon(\frac{1}{B}\nabla_{\boldsymbol{\theta}}(\sum_{i=1}^{B} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}))) \tag{2.7}$$

where ϵ is the learning rate, B is the minibatch size. The input samples in the minibatch are denoted as $\boldsymbol{x}^{(i)}$ while $\boldsymbol{y}^{(i)}$ are their corresponding targets. SGD is the simplest form of optimization and it is still a commonly used optimization strategy.

Momentum algorithm is proposed as an improvement to the SGD algorithm in terms of learning speed. Learning with SGD can be slow for some cases. In momentum update, a variable v, is introduced in order to accelerate the learning. This variable v behaves like the velocity which indicates the speed and the direction of the parameters. v has a hyperparameter β, which is named as momentum. Momentum hyperparameter $\beta \in [0, 1)$ adjusts the decaying speed of the gradients. SGD with Momentum optimizer accelerates the learning speed $1\Big/(1-\beta)$. For example, when β is chosen as 0.9, SGD with momentum algorithm learns 10 times faster than SGD. Parameter update with this optimizer is shown in equation 2.8:

$$\begin{aligned} \boldsymbol{v} &\leftarrow \beta\boldsymbol{v} - \epsilon(\frac{1}{B}\nabla_{\boldsymbol{\theta}}(\sum_{i=1}^{B} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}))) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{v} \end{aligned} \tag{2.8}$$

Setting the learning rate is one of the most challenging tasks in deep learning field and it affects considerably the performance of neural network. Adaptive learning rate methods eases this task, since they tune the learning rate for each parameter. ADAM is an example of such optimizers. The name of the algorithm derives from adaptive moment estimation. The parameter update with ADAM is shown in equation 2.9 [32]:

$$
\begin{aligned}
\boldsymbol{g} &\leftarrow \frac{1}{B}\nabla_{\boldsymbol{\theta}}\left(\sum_{i=1}^{B} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta})\right) \\
t &\leftarrow t+1 \\
\boldsymbol{m} &\leftarrow \beta_1 \boldsymbol{m} + (1-\beta_1)\boldsymbol{m} \\
\boldsymbol{v} &\leftarrow \beta_2 \boldsymbol{v} + (1-\beta_2)\boldsymbol{g} \otimes \boldsymbol{g} \\
\hat{\boldsymbol{m}} &\leftarrow \frac{\boldsymbol{m}}{1-\beta_1^t} \\
\hat{\boldsymbol{v}} &\leftarrow \frac{\boldsymbol{v}}{1-\beta_2^t} \\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \epsilon \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}} + \delta}
\end{aligned}
\tag{2.9}
$$

After the gradient $\boldsymbol{g}$ is computed at time-step t, first moment estimate $\boldsymbol{m}$ and the second moment estimate $\boldsymbol{v}$ initialized as zero are updated. Here, $\boldsymbol{g} \otimes \boldsymbol{g}$ represents the elementwise multiplication. Afterwards, the moment estimates are bias-corrected by dividing them to terms which include exponential decay rates, β_1 and β_2. The parameters are updated by using corrected moment estimates ($\hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{v}}$), step size term ϵ and small stabilization constant δ. As Kingma and Ba suggested in [32], the default settings are $\epsilon = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\delta = 10^{-8}$. ADAM is computationally efficient and it requires little tuning for hyperparameters. It also performs well when the data is large and/or there are lots of parameters.

Learning rate is a very crucial hyperparameter for optimizers. Decision of using a fixed or decaying learning, choice of its initial value and selecting the learning rate decay type have great impacts on training performance. Learning rates and other hyperparameters such as momentum are studied in section 3.2.

There are also other optimizers such as Nesterov Momentum, AdaGrad and RMSProp. In Nesterov Momentum, a correction factor which includes the velocity term is added while the gradient is evaluated. AdaGrad [33] and its modified form RMSProp [34] are other adaptive learning rate methods. Since ADAM integrates the advantages of these two methods, it can be favored in the deep

neural networks. Second-order methods, such as Newton's method, are computationally expensive because it has to calculate second-order partial derivatives in order to build Hessian matrices. We will eventually evaluate the performance of three optimizer for energy-efficient neural networks in Chapter 4: SGD, SGD with momentum and ADAM.

2.5 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are one of the oldest examples of deep learning architectures, which have remarkable success at vision and signal processing tasks [35]. First inspiration to convolutional neural networks comes from Hubel and Wiesel's study on the visual cortex of a cat [36]. Later, Fukushima adapted this study to build the structure of a neural network and proposed the first CNN-like model named as neocognitron [37]. Afterwards, LeCun et al. applied backpropagation to the handwritten zip code recognition task and successfully trained the convolutional neural network [38]. LeCun et al.'s another handwritten number recognition study is a trademark in the machine learning history, which proposed famous convolutional neural network LeNet-5 [7]. In 2012, CNNs received great attention in deep learning area when Krizhevsky et al. won the ImageNet contest ILSVRC (ImageNet Large Scale Visual Recognition Challenge) by significantly improving the classification error rate from 26.2% to 15.3% [8].

Convolutional neural networks are usually used to process data with grid-like topology. They perform well especially on images, which can be regarded as a 2D-grid of pixels. While referring CNN, one has to mention about the convolution operation. A two-dimensional discrete convolution function is defined as [1]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.10}$$

where I is the 2D input image and K is the 2-D kernel whose width and height positions are i and j, respectively. The output S is sometimes referred as

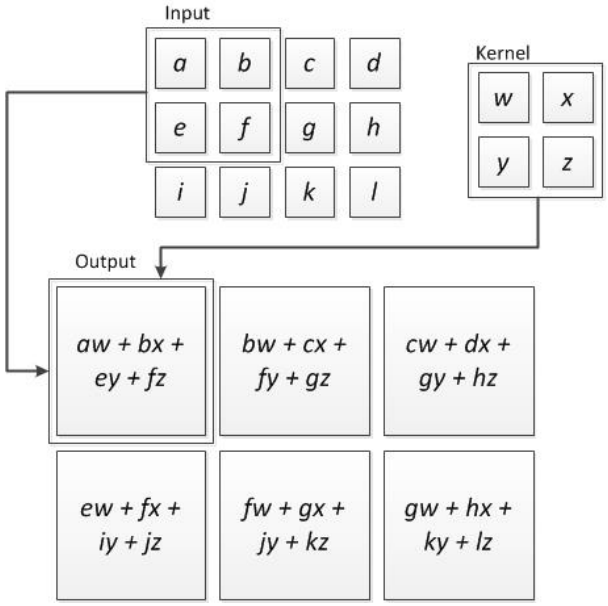

Figure 2.5: An example of 2-D convolution operation without kernel flipping [1].

feature map. According to this formula, the kernel is flipped relative to input. However, most machine learning libraries implement the convolution operation without flipping the kernel, which is, in fact, the cross-correlation formula in equation 2.11 and still call it convolution:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{2.11}$$

A visual representation of the convolution without flipping the kernel is shown in 2.5.

Convolutional neural network advantages from three main concepts which enhance the performance of a machine learning structure: sparse connectivity, shared parameters and equivariance to translation [1]. In traditional neural networks, such as multi-layer perceptron (MLP), every output is affected by every input unit. However, this is not the case for CNN. Since the kernel is much smaller compared to the input, **sparse connectivity** is accomplished. This means that a convolutional network is more efficient, since it has to deal with less parameters and requires less memory usage. **Parameter sharing**, as the name implies,

means that different models or different units in the model use the same set of parameters. While calculating the output of a layer, conventional neural networks use each element of weight matrix to multiply an input element and same weight value is never applied to the other input components. In a CNN, same kernel is used at every location of the input tensor as seen in 2.5. This implies that the network learns only one parameter set rather than learning separate sets every time. Although using shared parameters doesn't affect the computation time of the training, it reduces the memory requirements significantly. The term that a function is equivariant can be described as; if the input undergoes a change, the output also experiences that change in the same way. Convolution operation is **equivariant to translation**, i.e. shifting. CNN uses this different form of parameter sharing effectively. For instance, an image of a car is still an image of a car when each pixel in the image is shifted by one unit. CNN computes the same feature of the image over different positions of the input image. Hence, a car still can be detected even though the car is shifted. Unfortunately, convolution is not equivariant to other image transformations such as scaling or rotation [1].

A typical architecture of convolutional neural network contains several main layers: After input stage, *convolutional layer* performs convolution operations to compute the output of the neurons at that layer. Nonlinearity stage applies a nonlinear activation function to each element of the input of the layer. This activation function is rectified linear unit (ReLU) in general and this layer is commonly called as *ReLU layer*. *Pooling layer*, also known as downsampling stage, summarizes the input over a rectangular neighborhood according to a mathematical operation such as *max* or *average*. As a result, pooling reduces the dimensions (width and height) of the input. *Fully-connected (FC) layer* is the dense layer and computes the affine transform in equation 2.1 like ordinary neural networks. At last, the *softmax layer* can be set as the final layer of CNN where the distribution of the predicted class labels is produced with a more stabilized manner [39]. An example of typical CNN architecture is shown in Figure 2.6.

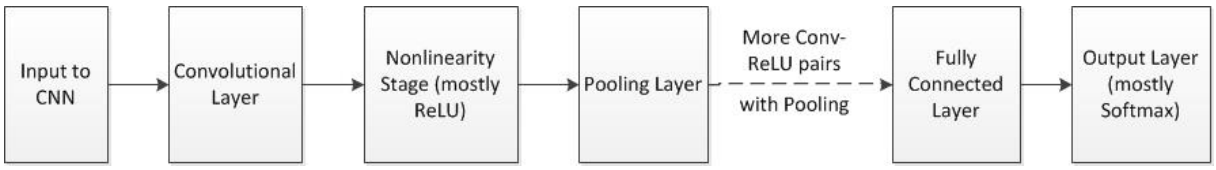

Figure 2.6: An example of standard CNN with its major components.

2.5.1 Convolutional Layer

A convolutional layer is the main component of the CNN and, as the name implies, performs convolutional operation. The neurons in this layer are connected to only a small portion of the input of this layer. This small portion is called kernel, or the filter. During the training, the filters are updated and learned by the neural network so that they can eventually detect some features of the image such as edges or colors [39].

The size of the filter is an important parameter of the convolutional layer. If the input data is an image, we can think that the input and also the neurons in the convolutional layer are in 3 dimensions (*width*, *height* and *depth*). For example, an image in the CIFAR-10 dataset has a size of 32*32*3 and the size of a filter at the first convolutional layer may be 4*4*3. While the depth of the filter is the same as the depth of the input of the layer, the number of filters used in a layer determines the depth of the output of the layer. The size of the output of a convolutional layer is calculated as shown in Equation 2.12, where N is the width/height of the image, F is the filter size, S is the stride an P is the size of zero-padding:

$$\frac{N - F + 2P}{S} + 1 \tag{2.12}$$

Stride (S) is a parameter which specifies the amount of the slide of filter in the image. In Figure 2.5, stride is $S = 1$. The values of stride are restricted by input size and other parameters F & P because the result of Equation 2.12 has to be an integer. Zero-padding term states that the input image is padded with zeros around the border. The size of the zero padding is denoted as P. Although

P can be set as an integer number, many machine learning libraries use *valid padding* and *same padding* approaches. Valid padding indicates that the image is not zero-padded, i.e. $(P = 0)$. Same padding ensures that the output size of the image is the same as the input size with $S = 1$; the total number of zeros padded to the image is $F - 1$. As stated before, the depth of the output of convolutional layer depends on how many filters are used in that layer. For example, if the size of the input image is 32*32*3 and 12 filters with size 4*4*3 are used with $P = 1$ and $S = 2$, then the output will be 16*16*12. $(\frac{32-4+2*1}{2} + 1 = 16)$

2.5.2 Nonlinearity Stage

At nonlinear stage, a nonlinear activation function is applied to the output of the convolutional and the fully-connected layers. This activation function performs an elementwise operation and the size on the input to this layer is not changed. ReLU is the most favorable function for nonlinearity stage, especially after convolutional layers.

2.5.3 Pooling Layer

Pooling layer essentially reduces the size of the input image. As a result, the number of parameters is decreased and it takes less time to compute in the network. Pooling layer also has filter size F and stride S parameters. A common choice for S is $S = 2$. F can be chosen as $F = 2$ or $F = 3$. $F = 2$ and $S = 2$ is the common selection for the pooling layer parameters and we can reduce the number of computations by 75% with this option. In Figure 2.7, downsampling operation is illustrated with different methods. One should note that only width/height of the image is reduced with the pooling operation and the depth of the image is still the same.

There are several pooling techniques such as max-pooling, average pooling, or L_2-norm pooling. Max-pooling is highly recommended and commonly preferred

Figure 2.7: Examples of non-overlapping pooling. *Top:* Max-pooling operation. *Bottom:* Average-pooling operation.

in practice, since it can make the network converge faster and improve generalization [40]. Pooling layer can be placed after a convolutional-ReLU layer pair or after multiple convolutional-ReLU pairs in the architecture.

2.5.4 Fully Connected Layer

Before fully-connected (FC) layer, the output of the convolutional/pooling layer just before FC layer is flattened. For example, the output of the previous layer is [a*b*c], then the input to the FC layer will be [1*1*(abc)]. FC layer uses this single vector and computes the output as regular neural networks (like Multilayer Perceptron). The outputs of the neurons at FC layer are affected by all inputs of FC layer; which implies that sparse connectivity no longer exists.

2.5.5 Softmax Layer

Softmax layer can be used as the output layer of the CNN and computes the softmax function for classification purposes. Softmax function takes a n-dimensional vector with arbitrary real values as input and produces a n-dimensional vector with values only in the interval of $[0, 1]$. Sum of the elements of output vector is 1. Softmax function produces the predicted class probabilities over n class labels and should be used for only the output layer of the neural network architectures. The function is shown in Equation 2.13.

$$f(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.13}$$

The neural networks with softmax layer are usually trained with a log loss function (cross-entropy). Since softmax function is differentiable, it is mostly preferred to compute the output of the networks trained with gradient descent based algorithms. In addition, softmax function makes easier to apply a threshold for the decision because the output vector of the softmax layer has values only between 0 and 1.

Chapter 3

Energy Efficient Neural Networks

3.1 Introduction

As stated earlier, convolutional neural networks (CNN) is a very successful example of deep learning architecture on vision and object recognition tasks. Although this type of deep neural network has very reliable results on object recognition and detection, it requires large amount of energy and computational time. Especially on mobile devices or any other small portable machines, memory limitation and restricted battery power become a huge problem while implementing such machine learning tasks. Hence, we study different energy efficient networks in this thesis. Firstly, we investigate the efficieny of Binary-Weight-Network (BWN) proposed in [18] which approximates the weights to binary values. Similar to BWN, we propose a Hadamard-transformed Image Network (HIN) which uses the Hadamard-transformed images with binarized weights. Lastly, a combined network is introduced and its superiority to both BWN and HIN is investigated.

While describing the algorithms, the nomenclature used in this section is as follows: $\boldsymbol{I}$ is the input tensor, $\boldsymbol{W}$ is the weight (filter), L is the number of layers, K is the number of filters in the l^{th} layer of the CNN. ϵ is the learning rate and β is the momentum parameter.

3.1.1 Binary Weight Networks (BWN)

Binary-Weight Network (BWN) is proposed in [18] as an efficient approximation to standard convolutional neural networks. In BWNs, the filters, i.e. weights of the CNN are approximated to binary values $+1$ and -1. While a conventional convolutional neural network needs multiplication, addition and subtraction for convolution operation, convolution with binary weights can be estimated by only addition and subtraction.

Convolution operation can be approximated as follows:

$$\boldsymbol{I} * \boldsymbol{W} \approx (\boldsymbol{I} \oplus \boldsymbol{B})\alpha \tag{3.1}$$

where $\boldsymbol{B}$ is the binary weight tensor which has the same size with $\boldsymbol{W}$ and $\alpha \in \mathbb{R}^+$ is the scaling factor such that $\boldsymbol{W} \approx \alpha\boldsymbol{B}$. $\oplus$ operation indicates convolution only with addition and subtraction. Since the weight values are only $+1$ and -1, convolution operation can be implemented in a multiplier-less manner. After solving an optimization problem to estimate $\boldsymbol{W}$, $\boldsymbol{B}$ and α is found as:

$$\boldsymbol{B} = sign(\boldsymbol{W}) \tag{3.2}$$

$$\alpha = \frac{\boldsymbol{W}^T\boldsymbol{B}}{n} = \frac{\boldsymbol{W}^T sign(\boldsymbol{W})}{n} = \frac{\sum |\boldsymbol{W}_i|}{n} = \frac{1}{n}||\boldsymbol{W}||_{\ell 1} \tag{3.3}$$

In Equation 3.3, $n = c \times w \times h$ where c is the channel, h is the height and w is the width of weight tensor $\boldsymbol{W}$, and of $\boldsymbol{B}$ as well. Equations 3.2 and 3.3 show that binary weight filter is simply the sign of weight values and scaling factor is the average of absolute weight values.

While training a CNN with binary weights, the weights are only binarized in forward pass and back propagation steps of the training. At the parameter-update stage, the real-valued weights (not binarized) are used. Training procedure for a BWN is demonstrated in Algorithm 1. $\mathcal{W}$ is the set of weight filters where $\mathcal{W}_{lk}$ is

the k^{th} weight filter in the l^{th} layer of CNN. $\mathcal{B}$ is the set of binary tensors where $\mathcal{B}_{lk}$ is a binary filter in this set and $\mathcal{A}$ is the set of positive real scalars where each element of this set is a scaling factor.

Algorithm 1 One step parameter update during the training of a CNN with binary weights

$\boldsymbol{I}$ is the input and $\boldsymbol{Y}$ is the target. $\widetilde{\mathcal{W}}$ is the binarized weight.
$C(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$: cost function, $\mathcal{W}^t$: current weight, ϵ^t: current learning rate.
$\mathcal{W}^{t+1}$: updated weight, ϵ^{t+1}: updated learning rate.

Binarize the weights in each corresponding layer
for l from 1 to L **do**
  for k from 1 to K **do**
    $\mathcal{A}_{lk} = \frac{1}{n}||\mathcal{W}^t_{lk}||_{\ell 1}$
    $\mathcal{B}_{lk} = sign(\mathcal{W}^t_{lk})$
    $\widetilde{\mathcal{W}_{lk}} = \mathcal{A}_{lk}\mathcal{B}_{lk}$
  end for
end for
Forward propagation with $\boldsymbol{I} * \boldsymbol{W} \approx (\boldsymbol{I} \oplus \boldsymbol{B})\alpha$
$\hat{\boldsymbol{Y}}$ =BinaryForward$(\boldsymbol{I}, \mathcal{B}, \mathcal{A})$
Backward propagation with binarized weights
$\frac{\partial C}{\partial \widetilde{\mathcal{W}}}$ =BinaryBackward$(\frac{\partial C}{\partial \hat{\boldsymbol{Y}}}, \widetilde{\mathcal{W}})$
Update parameters with SGD (with momentum) or ADAM
$\mathcal{W}^{t+1}$=UpdateParameters$(\mathcal{W}^t, \frac{\partial C}{\partial \widetilde{\mathcal{W}}}, \epsilon^t)$
Update learning rate
ϵ^{t+1}=UpdateLearningRate(ϵ^t, t)

One should take into account a very significant point: It is assumed that the convolutional filters here don't have bias terms, and this convolution approximation is only held in convolutional layers. Fully connected layers still do have bias terms and standard multiplication.

3.1.2 Hadamard-transformed Image Networks (HIN)

Independent from this thesis, compressed domain data is also used as input in deep learning structures. Discrete Cosine Transform (DCT) domain data as the input data can outperform the state-of-the-art results as shown in [41]. Compressed domain video frames as input to the convolutional neural networks are

preferred rather than RGB frames, since data decompression requires extra computation time and energy. As a result; simpler implementation, effective computation and improved model accuracy are achieved. Wu et al. uses DCT because JPEG, MPEG video coding standards are based on DCT. In this thesis, we adopt a similar approach. We also use transform domain images and feed them into the our CNN model. This network model is called Hadamard-transformed Image Networks (HIN).

Before introducing the Hadamard-transformed Image Networks (HIN), we describe Hadamard Transform first. Hadamard Transform, also called as Hadamard-ordered Walsh-Hadamard Transform, is an image transform technique which is also used to compress images in 1970s [42]. Transform coefficients are only +1 and -1. Thus, Hadamard Transform can be considered as a simpler alternate of other image transforms such that it can be implemented without any multiplication and division [43].

1-D Hadamard Transform is defined with the Hadamard matrix $\boldsymbol{H}_m$ whose size is $2^m \times 2^m$:

$$\underline{T} = \boldsymbol{H}_m \underline{g} \tag{3.4}$$

where $\underline{g}$ is 1-D array with 2^m elements and $\underline{T}$ is the transformed array. $\boldsymbol{H}_m$ is a real, symmetric and unitary matrix with orthonormal columns and rows such that:

$$\boldsymbol{H}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}, \boldsymbol{H}_m = (\frac{1}{\sqrt{2}})^m \begin{bmatrix} \boldsymbol{H}_{m-1} & \boldsymbol{H}_{m-1} \\ -\boldsymbol{H}_{m-1} & \boldsymbol{H}_{m-1} \end{bmatrix}$$

1-D Hadamard Transform can also be expressed by Equation 3.5. In this formula, $g(x)$ is the elements of 1-D array $\underline{g}$ and $b_i(x)$ is the i^{th} bit (from right to left) in the binary representation of x. The scaling factor $(\frac{1}{\sqrt{2}})^m$ is used to make the Hadamard matrix orthonormal, hence it is mostly kept in the calculations.

$$T(u) = (\frac{1}{\sqrt{2}})^m \sum_{x=0}^{2^m-1} g(x)(-1)^{\sum_{i=0}^{m-1} b_i(x)b_{m-1-i}(u)} \tag{3.5}$$

2-D Hadamard Transform is a straightforward extension of 1-D Hadamard Transform [44]:

$$T(u,v) = (\frac{1}{2})^m \sum_{x=0}^{2^m-1} \sum_{y=0}^{2^m-1} g(x,y)(-1)^{\sum_{i=0}^{m-1} (b_i(x)p_i(u)+b_i(y)p_i(v))} \tag{3.6}$$

In Equation 3.6, $p_i(u)$ is computed using:

$$\begin{aligned} p_0(u) &= b_{m-1}(u) \\ p_1(u) &= b_{m-1}(u) + b_{m-2}(u) \\ &\vdots \\ p_{m-1}(u) &= b_1(u) + b_0(u) \end{aligned} \tag{3.7}$$

2-D Hadamard Transform is separable and symmetric, hence it can be implemented by using row-column or column-row passes of the corresponding 1-D transform.

There is an algorithm called Fast Walsh-Hadamard Transform ($FWTH_h$) which requires less storage and is fast and efficient to compute Hadamard Transform [21]. The implementation of this algorithm is so simple that it can be achieved with only addition and subtraction which can be summarized in a butterfly structure. This structure is illustrated in Figure 3.1 for a vector of length 8. While the complexity of Hadamard Transform is $O(N^2)$, complexity of fast algorithm is $O(Nlog_2N)$ where $N = 2^m$.

As seen from above, Hadamard Transform is only for 1-D array whose length is a power of 2 because only Hadamard matrices whose order is a power of 2 exists. If the length of the 1-D array is less than a power of 2, the array is padded with zeros to the next greater power of two. Since 2-D Hadamard Transform is separable, we can treat columns and rows of the 2-D array as separate 1-D arrays.

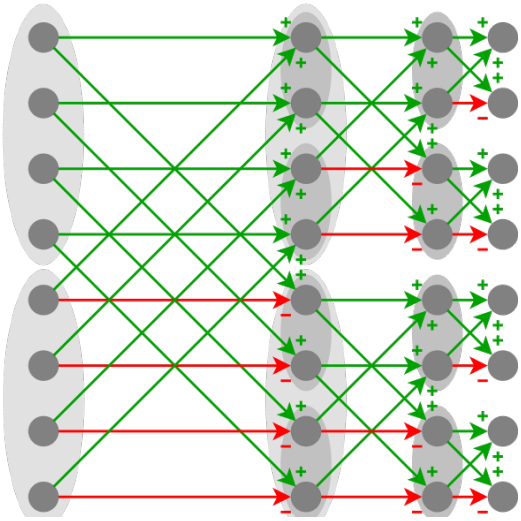

Figure 3.1: Fast Hadamard Transform algorithm of a vector of length 8.[1]

Training of HIN is quite equivalent to BWN; the only difference is that the input images are Hadamard-transformed as explained above. Training proceeds as explained in Algorithm 1, but at the beginning Hadamard-transformed input data $\widetilde{\boldsymbol{\mathcal{I}}}$ is fed in to the network rather than ordinary input $\boldsymbol{I}$. As in BWN, binarized weights are used and no bias terms are defined.

3.1.3 Combination of Models: Binary Weight & Hadamard Transformed Image Network (BWHIN)

Combination of the neural networks can improve the performance of the neural networks by a few percent. Since combining the neural networks reduces the test error and tends to keep the training error the same, it can be viewed as a regularization technique. One of the popular techniques of the combination is called "model ensembles" which combines the multiple hypotheses that explain the same training data [1,3]. As an example of ensemble methods, several different models are trained separately, then their predictions are averaged at test time. This method is called "model averaging". In model averaging, different models will probably make different errors on the test set and if the members of the

[1]Image retrieved from "Fast Walsh-Hadamard Transform - Wikipedia"

ensemble make independent errors, the ensemble will perform significantly better than its members. Even if all models are trained on the same dataset, differences in hyperparameters, mini-batch selections or different random initialization etc. cause the members of ensemble to produce partially independent errors.

In model ensembles, the error made by averaging prediction of all models in the ensemble decreases linearly with the ensemble size, i.e. the number of models in the ensemble. However, since they need longer time and increased memory to evaluate on test example, we try to avoid increasing the size in terms of energy efficiency. Speaking of the energy efficiency, since we want to build the entire model as efficient as possible, we don't need to wait the models in the ensemble train completely. Instead, different networks can be trained independently and separately until some point and they can be combined with a combination layer where locates somewhere before the output layer. Bilinear CNNs [2] is a good example for such models. In bilinear CNN, there are two sub-networks which are standard CNN units. After these CNN units, the image regions which extract features are combined with a matrix dot product and then average pooled to obtain the bilinear vector. In order to perform these operations properly, those image regions have to be of the same size. This vector is passed through a fully-connected and softmax layer to obtain class predictions.

Our approach to combine BWN and HIN is quite similar to Bilinear-CNN, but simpler. After convolutional, ReLU and pooling layers, the output tensor is reshaped for fully connected layer as a 1-D tensor. Afterwards, these same sized 1-D tensors of each sub-network will be averaged instead of dot product. Since multiplication consumes power, dot product is avoided and averaging is preferred. Two averaging methods are used: Simple averaging and weighted averaging. [45]. Simple averaging is the conventional averaging technique which calculates the output by averaging the sum of outputs from each ensemble member. Weighted averaging technique assigns a weight to each ensemble member and calculated the output by taking these weights into account. The total weight of each ensemble is 1. In order to implement this technique, we define a random number which behaves like a weight. If YY_{binary} is defined as the 1-D tensor of BWN and $YY_{hadamard}$ is the 1-D tensor of HIN, the standard averaging is shown as:

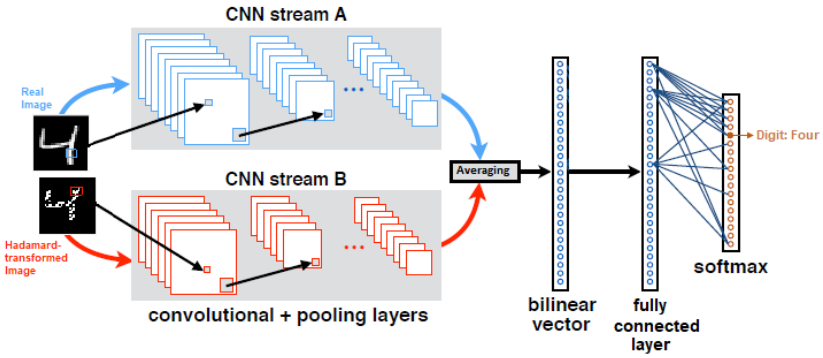

Figure 3.2: Our approach to combine BWN and HIN: The architecture of BWHIN [2].

$$YY_{combined} = 0.5 \times YY_{binary} + 0.5 \times YY_{hadamard} \tag{3.8}$$

while the new averaging method can be described as:

$$YY_{combined} = (W_{combined} \times YY_{binary}) + ((1 - W_{combined})) \times YY_{hadamard} \tag{3.9}$$

where $W_{combined}$ is the random number which can only take values in $[0, 1]$. This random number is generated according to truncated normal distribution whose mean is 0.5 and standard deviation is 0.25. We will also compare the performances of these two combination methods in Chapter 4. After the averaging operation, obtained 1-D tensor is followed by a fully connected and softmax layer. The architecture of our combined network Binary Weight & Hadamard-Transformed Image Network (BWHIN) is summarized in Figure 3.2. One should notice that the combination is applied after the convolutional layers of each network, which are energy efficient layers. With this combination model, we still want to maintain the energy efficieny of the entire network.

3.2 Neural Network Architecture and Hyper-paramaters

In order to investigate the energy efficieny of the neural networks, the neural network architectures are formed as very simple models with small capacity as possible. Convolutional Neural Networks are used as an efficient deep neural networks model. Hyperparameters are chosen according to the state-of-the-art solutions in the literature.

3.2.1 CNN Architectures

In deep neural networks (DNN), the size of the layers determines the capacity. A model's capacity is an important model parameter so that it controls the model's ability to fit a wide variety of functions. In case of low capacity, the model may struggle to fit the training set and produce large training errors and the model underfits. Models with high capacity can memorize aspects of the training set which may not function properly on the test set. As a result, overfitting occurs and a large difference is produced between training and test error [1].

In case some regularization techniques are used, it is important to choose the number of neurons in a layer large enough so that the generalization will not be damaged. Yet, larger number of neurons requires longer computation time as expected. As mentioned in [46], the size of all layers can be the same, or can be selected as a decreasing size (pyramid-like) or increasing size (upside down pyramid). Naturally, this selection depends on the data. We will choose the neuron numbers with an increasing manner from first convolutional layer to the fully connected layer.

In order to implement the proposed networks and analyze their performances, two well-known datasets, MNIST and CIFAR-10 are chosen. In addition, two different CNN architectures are investigated to observe the energy efficiency. First type of architecture for MNIST database is very similar to LeNet-5 in [7] with

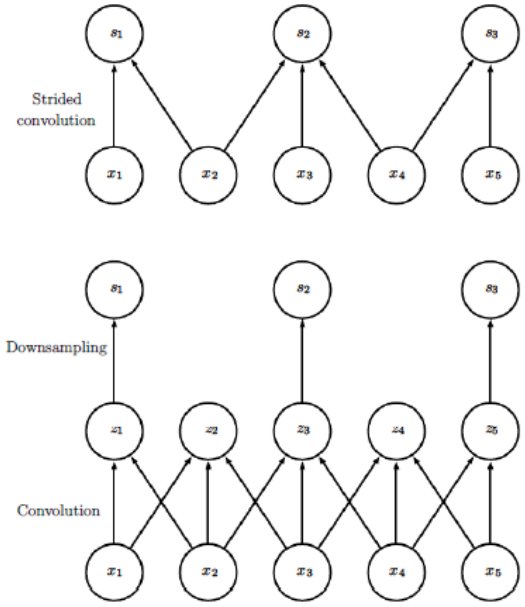

Figure 3.3: *Top:* Strided convolution with a stride of 2. *Bottom:* Convolution with unit stride followed by downsampling.[2]

convolutional and pooling layers. Second architecture is built according to All-Convolutional-Neural-Network [22] with strided convolution. Strided convolution is that some positions of the kernel are skipped over in order to reduce the computational burden while implementing the convolution operation. Strided convolution is equivalent to downsampling the output of the full convolution function. This is illustrated in Figure 3.3. The reason is to investigate the effect of the pooling layer and strided convolution on energy efficiency and test accuracy. Both neural network architectures used for MNIST are summarized in Table 3.1.

First architecture is built as [Conv-ReLU-Conv-ReLU-Pool-Conv-ReLU-Pool-FC-Softmax] while second architecture is built as [Conv-ReLU-StridedConv-ReLU-StridedConv-ReLU-FC-Softmax]. For 3 convolutional layers and 1 fully connected layer, the sizes of each layer is determined as 6, 12, 24, 200; with an increasing manner as mentioned before. These numbers are set by trial and error. If the sizes of the layers were too low, the model would encounter with the low capacity problems. On the other hand, the network with a big capacity would not only overfit, also it could cause hardware problems such that the training

[2]Image retrieved from [1].

<table>
<tr><td>ConvPool-CNN</td><td>All-CNN</td></tr>
<tr><td colspan="2">Input 28*28 gray-scale image</td></tr>
<tr><td>6*6 conv. 6 ReLU</td><td>6*6 conv. 6 ReLU</td></tr>
<tr><td>5*5 conv. 12 ReLU</td><td rowspan="2">5*5 conv. 12 ReLU
with stride 2</td></tr>
<tr><td>2*2 max-pooling, stride 2</td></tr>
<tr><td>4*4 conv. 24 ReLU</td><td rowspan="2">4*4 conv. 24 ReLU
with stride 2</td></tr>
<tr><td>2*2 max-pooling, stride 2</td></tr>
<tr><td colspan="2">fully connected layer with 200 neurons, dropout</td></tr>
<tr><td colspan="2">10-way softmax layer</td></tr>
</table>

Table 3.1: Model description of the two architectures for MNIST dataset.

and/or test process would fail. Both pooling and strided convolutional operations are used to shrink the input size by a factor of two in order to reduces the computational and statistical burden on the next layer.

Filter sizes are determined heuristically. Since 5*5 filters are used in LeNet-5, filter sizes are selected to be close to this size. In order to preserve the input size for conventional convolutional layers, stride is chosen as 1 and zero padding is used accordingly. For strided convolutional layers, stride is 2 to decrease the height and width of the image by a factor of 2. According to [39], common choice for non-overlapping max-pooling operation is with 2*2 filters and stride 2. This size is preferred, otherwise pooling with larger receptive fields would be too harmful.

Architectures applied to CIFAR-10 dataset are described in Table 3.2. Since CIFAR-10 has color images and larger images than MNIST, models with higher capacity is preferred. Model capacity is expanded by increasing both the number of layers and the number of neurons at the hidden layers. The architecture with pooling layers is built as [Conv-ReLU-Conv-ReLU-Pool-Conv-ReLU-Conv-ReLU-Pool-FC-Softmax], while all-CNN architecture is build as [Conv-ReLU-StridedConv-ReLU-Conv-ReLU-StridedConv-ReLU-FC-Softmax]. Since we want to preserve the energy efficieny as far as possible, we use more convolutional layers, which can be modified as energy efficient layers, and only one fully-connected layer. The sizes of these 4 convolutional layers and 1 fully-connected layer are 32, 32, 64, 64, 512, respectively. The number of neurons in a layer and the filter

ConvPool-CNN	All-CNN
Input 32*32 RGB image	
3*3 conv. 32 ReLU	3*3 conv. 32 ReLU
3*3 conv. 32 ReLU	3*3 conv. 32 ReLU
2*2 max-pooling, stride 2	with stride 2
Dropout	
3*3 conv. 64 ReLU	3*3 conv. 64 ReLU
3*3 conv. 64 ReLU	3*3 conv. 64 ReLU
2*2 max-pooling, stride 2	with stride 2
Dropout	
fully connected layer with 512 neurons, dropout	
10-way softmax layer	

Table 3.2: Model description of the two architectures for CIFAR-10 dataset.

sizes are selected empirically. A critical point in CIFAR-10 architectures is that more dropout is used due to the increased capacity.

Note that the size of an image in the MNIST dataset is altered from 28*28*1 to 32*32*1 after Hadamard transform. As a result, the outputs of the BWN and HIN will not compatible in the combined model. In order to overcome this problem in the MNIST architectures, the filter size in the first convolutional layer whose input is Hadamard-transformed image is modified as 5*5 and zero-padding is not used. On the other hand, we will not have that issue for CIFAR-10 database. Since the width & height of an image in CIFAR-10 is 32, a power of 2, the size will remain unchanged (32*32*3) after Hadamard transform. No minor changes in the neural network architecture will be required.

3.2.2 Weight and Bias Initialization

Parameter initialization plays an important role for the deep neural networks to converge and achieve reasonable results in an acceptable amount of time. Especially weight initialization is still a popular and active research area because it has an strong effect on the training of the neural network. In general, weights are initialized as small random values while the biases are initialized to zero or

to small constant positive values [1].

Although biases can be initialized as zero, weights should be initialized differently to break the symmetry between different hidden units of the same layer. In case of symmetry, if two hidden units with same activation function are connected to same input and output, the model will update both of these units in same way and these units will have the same output and compute the same gradient. Symmetry wastes the capacity, since some input patterns may be lost in the null space of forward propagation and some gradient patterns may be lost in the null space of back- propagation as well. Hence the weights need to be initialized with different initial parameters [46].

The weights are usually initialized with small random numbers with uniform or Gaussian distribution. Large initial weights result in extreme values during forward propagation and that may cause the activation function to saturate and makes the gradient lost completely through saturated hidden units. Small initial weights are usually preferable due to regularization concerns. Some heuristic initialization strategies use uniformly distributed random numbers such as

$$W \sim U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$$

for a fully connected layer with m inputs and n outputs. As suggested in [47], Xavier initialization is another option for weight initialization:

$$W \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}})$$

We prefer Xavier initialization for CIFAR-10 architectures. Since we use smaller neural networks for MNIST database, we initialize weights for these models as:

$$W \sim \mathcal{N}(0, 0.1)$$

According to [48], zero-mean Gaussian with a small standard deviation around 0.1 or 0.01 works well.

The approach for initializing the biases is mostly to set them to zero. However, biases can be initialized with small positive numbers when the ReLU nonlinearity

is used. This makes the ReLU initially active for most inputs so that ReLU units can obtain some gradient and propagate. Since we use ReLU for both convolutional layers and fully connected layers, we set the bias of all ReLU hidden units to 0.1 rather than 0.

3.2.3 Mini-Batch Size

Mini-batch size, B is an important parameter for gradient-based training algorithms. Instead of training whole samples in the training set, only a small portion of the training set is selected to compute the gradient. On each step of the training algorithm, a minibatch of examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(B)}\}$ is drawn uniformly or randomly from the training set. Parameter update is performed based on an average of the gradients inside each block of B examples according to equation 3.10:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon(\frac{1}{B}\nabla_{\boldsymbol{\theta}}(\sum_{i=1}^{B} L(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}, \boldsymbol{\theta}))) \tag{3.10}$$

where ϵ is the learning rate and L is the loss function. This training algorithm is called as stochastic gradient descent algorithm (SGD) as also mentioned in Section 2.4.

The mini-batch size B is chosen as a relatively small number compared to the size of the training set; mostly in the range of 1 and few hundred. However, it is crucial that mini-batch size must be kept fixed during the training [1]. When B=1, the algorithm becomes ordinary gradient descent and when B is equal to training set size, SGD is now standard gradient descent. As B increases, there will be more multiply-add operations per second because these multiply-add operations will be parallelized and multiplication process will be more efficient. Nevertheless, with an increased B, it will take more time to converge, since one update on the batch will take longer time and the number of updates per computation time decreases. When B is a very small, more steps per epoch will be needed

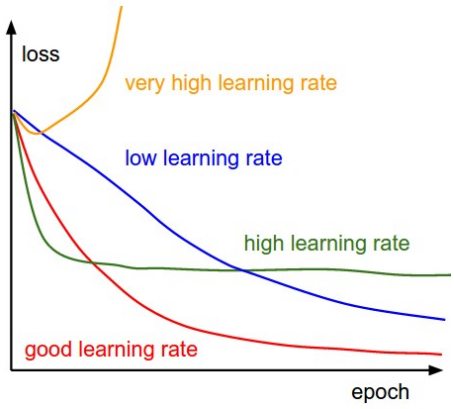

Figure 3.4: Effects of different learning rates [3].

to train whole set and the total run time will be very high [46]. Considering all these factors, B is chosen as 100 for all of our models.

3.2.4 Learning Rate

A crucial hyperparameter for many optimizers, perhaps the most crucial one, is the learning rate ϵ, which is a positive constant determining the step size of the gradient. According to [46], typical values for standardized learning rates are in the interval of $(10^{-6}, 1)$, but one has to note that this is not the strict range and learning rate highly depends on the parametrization of the model. Choice of the (initial) learning rate is very critical. Loss increases with too high learning rate and the model cannot even be trained. Too low learning rate is also problematic, because the training is going to be so slow that the cost function will never decrease and it may even have stuck at high values. Although the learning rate can be chosen as a fixed number, a good learning rate should decay over time as seen in Figure 3.4. While loss starts to decay exponentially with high learning rates, the improvement is almost linear with lower learning rates at the beginning. Although it is useful to have a decaying learning rate, one should be careful about the decay rate. If the decay is too slow, it will take too much time to achieve a reasonable and low cost. If the decay is too fast, the model will be trained too fast, and unable to find the local minimum.

In order to implement the learning rate decay, there are three common methods [3]:

- Step decay: After keeping the learning rate constant for a certain number of steps, it decreases by a certain factor according to a pre-determined rule. For example, one may reduce the learning rate by 0.5 every 10 epochs. These numbers vary according to the problem or model.
- Exponential decay: This decay is performed according to formula $\epsilon = \epsilon_0 e^{-kt}$ where ϵ_0 is the initial learning rate, k is the decay rate and t is the iteration number.
- $1/t$ decay: This type of decay has the mathematical formula $\epsilon = \epsilon_0/(1+kt)$ where ϵ_0 is the initial learning rate, k is the decay rate and t is the iteration number.

An exponentially decay learning rate is used as suggested in [49], since dropout technique can also be used to finetune the model along with an exponentially decaying learning rate as in this paper. For MNIST database, the maximum and minimum learning rate can be found empirically along with the decay speed k. Our learning rate starts at 0.003 and ends at 0.0001 with a decay rate of 0.0005. For CIFAR-10 dataset, the initial learning rate is selected as 0.0001 and the decay rate is set to 10^{-6}. No lower bound is specified for the learning rate used to train CIFAR-10.

3.2.5 Momentum

As described in Section 2.4, momentum is an important hyperparameter which accelerates the learning of gradient-based networks. The momentum hyperparameter, β which determines the exponential decay rate of the past gradients should be a number $\beta \in [0, 1)$. Although β can be adapted over time like learning rate, it is mostly chosen as a fixed number in the literature. In practice, β is

commonly chosen as 0.5, 0.9 or 0.99 [1]. Our choice of momentum used in SGD with Momentum optimizer is 0.9 as used in AlexNet [8] and ResNet [50].

3.3 Implementation of the Architectures

The MNIST (Modified National Institute of Standards and Technology) database of handwritten digit images is a very popular digit database for implementing learning techniques and pattern recognition methods [51]. It contains 60,000 training images and 10,000 test images. These black and white images are with 28*28 pixels, which means that the dimensionality of each image is 784. Pixel values of the images in this database varies from 0 to 255. Since the database consists of digits from 0 to 9, it has 10 classes. MNIST database is preferable since it requires less effort on preprocessing and formatting while dealing with real-world data. Sample images from each class is shown in Figure 3.5.

Figure 3.5: Sample images of MNIST database.

CIFAR-10 (Canadian Institute for Advanced Research-10) is also a famous dataset used for image classification tasks [4]. It consists of 50,000 training images and 10,000 test images. These 60,000 color images with size 32*32 is collected in

10 classes of objects (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). Each class has 6000 images and test set is generated by selecting 1000 images randomly from each class. Like in MNIST dataset, the pixel values are in the range of [0,255]. One of the unique properties of this dataset is that the classes are mutually exclusive. For example, "automobile" class doesn't include trucks or pickup trucks while "truck" class contains only big trucks, but not pickup trucks. In Figure 3.6, sample images from each class are illustrated.

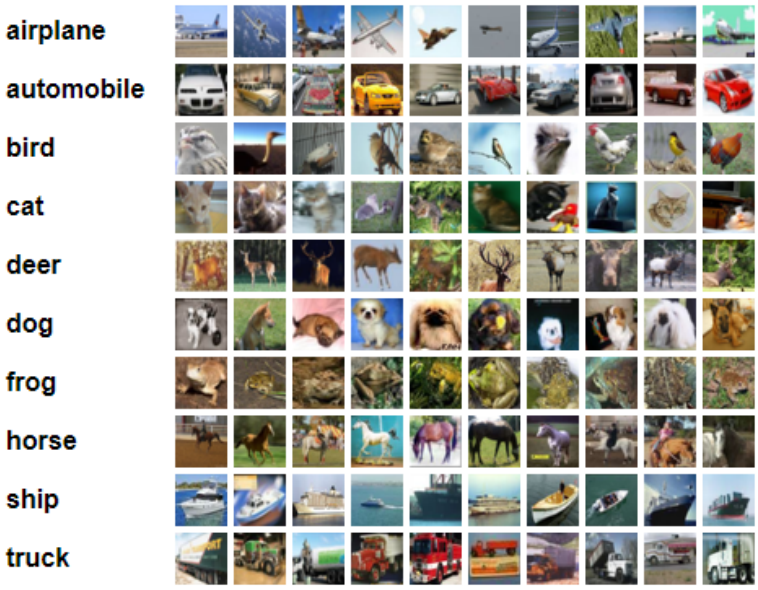

Figure 3.6: Sample images of CIFAR-10 database [4].

Tensorflow is chosen to implement all of the deep neural networks for this thesis work. Tensorflow is an open-source software library for machine intelligence, which is developed by Google Brain Team in Python language [52]. It is a very flexible library so that various algorithms for deep neural network models can be expressed easily for both research purposes and for deploying machine learning systems into production. One of its main advantages is that it can be easily programmed for multi-GPU and multi-node usage.

Since our main purpose is to investigate the energy efficieny of the proposed neural network models, simple architectures are chosen and no pretraining such as feature extraction or unsupervised learning techniques is performed. Images in both datasets are normalized such that the pixel values are in $[0, 1]$. All

experiments are carried out on a single GPU, which is NVIDIA GeForce 940M. Thanks to our CUDA-enabled GPU, we are able to run Tensorflow with GPU support and we achieve faster computation.

Chapter 4

Simulation and Results

As previously stated in Chapter 3, six CNN models with two different architectures are trained on MNIST and CIFAR-10 database. These CNN models include the standard CNN, previously studied BWN and our energy efficient neural networks. The performance of the neural networks is evaluated based on test accuracies and their performance is also compared with different specific hyperparameters. Networks trained on MNIST dataset has 10000 iterations. Considering the mini-batch size is chosen as 100, this number of iterations corresponds to after the completion of 16^{th} epoch. Since MNIST database contains 60000 training images, there are 600 iterations per epoch. Networks of CIFAR-10 dataset are trained in 150000 iterations. This means the number of epochs for CIFAR-10 training is chosen as 300 (500 iterations per epoch). Before learning procedure is performed, the input images are normalized such that the pixels of the images have the values only in $[0, 1]$. No other data preprocessing or unsupervised learning as pretraining is implemented at the beginning of the training. Cross validation is not required for now.

Test accuracy is decided as the main criterion to analyze the models in the following sections. Training accuracy results, training losses and test losses for MNIST database are shown in Appendix A. Other results for CIFAR-10 dataset

are demonstrated in Appendix B. The abbreviations used in the tables are already explained in Table 1.1 while CNN corresponds to the standard CNN here. BWHIN-Normal represents the combined model with standard average operation, while BWHIN-Random is the model with our "random-average" technique.

4.1 Experiments on MNIST

Three parameters are chosen in order to examine the behavior of the networks. These parameters are optimizers, the probability of keeping the hidden units for dropout technique and the activation function at the Fully Connected (FC) layer. SGD, SGD with momentum and ADAM are chosen as the optimization algorithms. SGD is a very simple learning algorithm and it was a very popular choice from the 1970s to late 1990s [6]. SGD with momentum is also frequently preferred by ILSVRC winners [8, 50] and it has better performance than SGD on deep networks. ADAM is also another practical optimizer which is fairly robust to the choice of hyperparameters [1]. As regularization technique, dropout is chosen. In case of dropout, a neuron is kept active with a fixed probability of p independent of other units. p can be considered as a hyperparameter and it can be set to a number or determined by cross-validation. Although $p = 0.5$ is a reasonable choice for hidden units, the optimal probability is usually closer to 1 than to 0.5 [31]. Here, we investigate no dropout case ($p = 1.0$), $p = 0.75$ and $p = 0.5$. Sigmoid function, tangent hyperbolic and ReLU are preferred as activation functions at FC layer. Prior to the introduction of ReLU, sigmoid and tangent hyperbolic functions are used by most neural networks, but nowadays ReLU is the most popular nonlinear function [1, 6]. While evaluating the performance of one parameter, other parameters are kept fixed. ADAM is selected as optimizer due to its robust nature while investigating dropout and activation functions. The probability of retention is chosen as $p = 0.75$ while analyzing the behavior of optimizers and activation functions. ReLU is preferred as the activation function due to its piecewise linearity while studying dropout and the optimizers.

	ConvoPool-CNN					
	SGD		SGD-momentum		ADAM	
	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy
CNN	99.26	99.17	11.35	11.35	99.48	99.41
BWN	98.46	98.36	98.92	98.79	98.88	98.75
HIN	11.35	11.35	11.35	11.35	98.32	98.23
BWHIN-NormalAvg	98.27	98.15	99.04	98.91	98.79	98.67
BWHIN-RandomAvg	98.9	98.78	98.77	98.71	98.96	98.88

(a) ConvoPool-CNN

	All-CNN					
	SGD		SGD-momentum		ADAM	
	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy
CNN	99.12	99.09	11.35	11.35	99.31	99.24
BWN	98.11	98.05	98.48	98.44	98.37	98.23
HIN	96.48	96.35	11.35	11.35	97.84	97.71
BWHIN-NormalAvg	97.24	97.13	98.47	98.38	98.4	98.23
BWHIN-RandomAvg	98.09	98.07	10.09	9.8	98.61	98.5

(b) All-CNN

Table 4.1: Test accuracy results (in percentage) for different optimizers.

4.1.1 Effect of Optimizers

The gradients computed in backpropagation stage of the training are used for parameter update. Optimizers are the learning algorithms which have different strategies for parameter update and acceleration techniques to speed up the training. Optimization is still a very active research area, but for this section we only use the common optimizers to investigate the models. SGD, SGD with momentum and ADAM are chosen. The common hyperparameters for these three optimizers is the learning and mini-batch size which are already determined in Chapter 3. Momentum parameter for SGD with momentum is chosen as 0.9 as stated in Section 3.2.5. The moment estimates β_1 and β_2 for ADAM optimizer are left as default Tensorflow values $\beta_1 = 0.9$ and $\beta_2 = 0.999$. These values are also suggested by [32]. There is also another constant ρ required for ADAM. This small constant is used for numerical stabilization and recommended value $\rho = 10^{-8}$ is used. Test accuracy results are reported in Table 4.1 and their variation throughout the training is shown in Figure 4.1.

ADAM optimizer is the best choice for both architectures and most of the models. SGD with momentum gives the best result only for BWN. As seen from

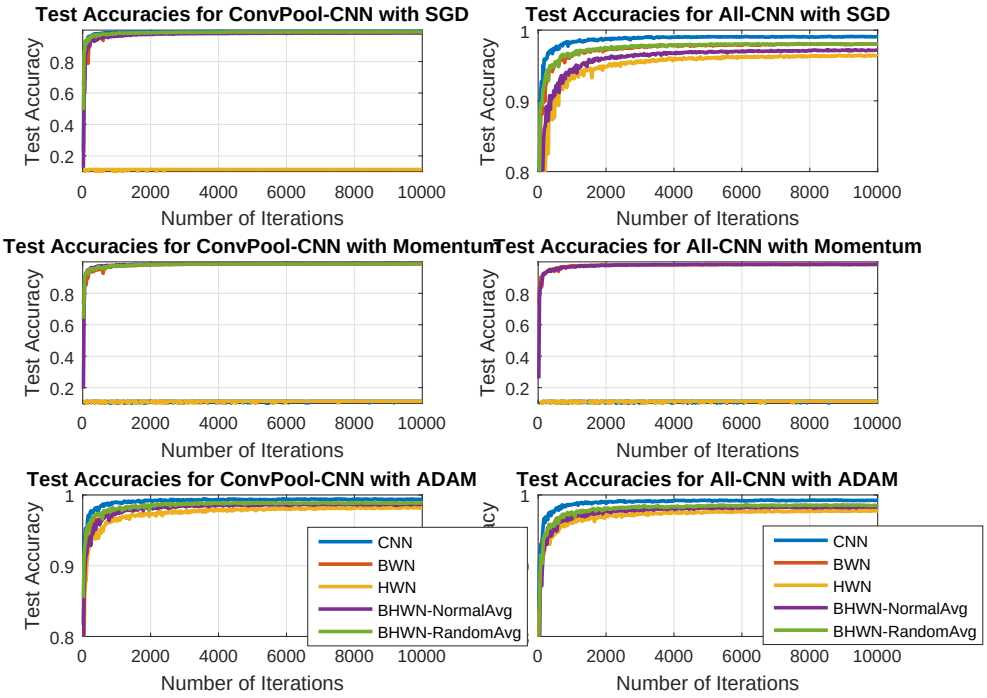

Figure 4.1: Test accuracy results for different optimizers.

tables and figures, SGD and SGD with momentum algorithms cannot train some networks, especially HIN. SGD with momentum optimizer struggles to train even the standard CNN. SGD based algorithms may be slow for these networks. The reasons might be the learning rate or the low capacity. In order to avoid these problems, a higher initial learning rate can be chosen or another learning rate decay technique can be applied. Increasing the size of the hidden layers is another option. The training failure which is caused by the SGD based algorithms also affects the performance of combined BWHIN. If HIN cannot be trained, it may probably affect the averaging negatively and the network may fail to perform classification.

4.1.2 Effect of Dropout

As mentioned in Section 2.3, dropout is chosen as the regularization technique for our models in order to reduce overfitting. Since a fully connected layer in a CNN owns most of the parameters, individual nodes are dropped out at this

	ConvoPool-CNN					
	$p = 1.0$		$p = 0.75$		$p = 0.5$	
	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy
CNN	99.46	99.35	99.48	99.41	99.48	99.38
BWN	98.89	98.85	98.88	98.75	98.83	98.77
HIN	98.27	98.18	98.32	98.23	98.02	97.91
BWHIN-NormalAvg	99.01	98.85	98.79	98.67	98.66	98.61
BWHIN-RandomAvg	98.73	98.64	98.96	98.88	98.74	98.59

(a) ConvoPool-CNN

	All-CNN					
	$p = 1.0$		$p = 0.75$		$p = 0.5$	
	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy
CNN	99.2	99.13	99.31	99.24	99.3	99.15
BWN	98.43	98.38	98.37	98.23	98.49	98.36
HIN	97.73	97.65	97.84	97.71	98.01	98.01
BWHIN-NormalAvg	98.32	98.21	98.4	98.23	97.95	97.94
BWHIN-RandomAvg	98.38	98.3	98.61	98.5	98.55	98.48

(b) All-CNN

Table 4.2: Test accuracy results (in percentage) for different dropout probabilities.

layer with probability $(1 - p)$ where p is the probability of the nodes kept in the neural network [31]. For example, if $p = 1.0$, none of the units in the layer will be dropped out. If $p = 0.5$, only the half of the units at the layer are kept. In Table 4.2, the results for with different p probabilities are shown. One should pay attention that dropout is performed only at the training stage, while there is no dropout at test time; i.e. $p = 1.0$ while testing. The removed nodes of the hidden layer at the training stage has to be reinserted at the test stage. The results are also illustrated in Figure 4.2.

Dropout technique is highly recommended in terms of energy efficiency and its regularization effect. Considering ConvPool-CNN architecture, most networks are more successful at classification with $p = 0.75$. $p = 1.0$ gives the best results only for BWN and BWHIN-Normal which is affected by the training of BWN. Considering All-CNN architecture, when $p = 0.75$, most networks give better results. Only BWN and HIN has the best results for $p = 0.5$. Yet, both combined network models work better than BWN and HIN when the probability of retention is equal to 0.75. From Figure 4.2, the probabilities $p = 1.0$ and $p = 0.75$ seem to train with high speed. However, when the table A.3 in Appendix A.3 which shows the loss results is analyzed, it can be observed that the generalization gap

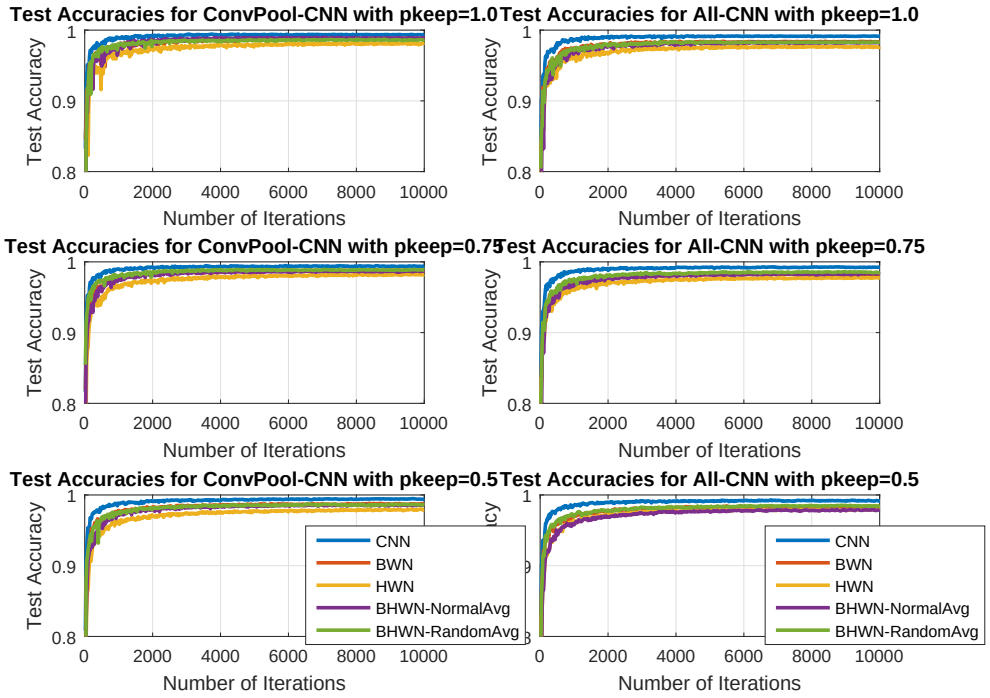

Figure 4.2: Test accuracy results for different dropout probabilities.

between training and test loss is higher for $p = 1.0$. Hence $p = 0.75$ is preferred over $p = 1.0$. As also seen from Figure 4.2, the models with $p = 0.5$ has slower training speed than the models with higher p.

4.1.3 Effect of Activation Function on FC Layer

According to our proposed architectures, there is only one fully-connected layer before the output layer (softmax layer). Unlike the convolutional layers of the models, no energy efficient technique is applied to the FC layer, except from the dropout training as regularization. Hence the behavior of the activation function at this layer should be investigated. Sigmoid, tangent hyperbolic ($tanh$) and ReLU functions are used to evaluate the performance of neural networks. Although ReLU mostly outperforms other activation functions for MLPs and CNNs such as in [8], sigmoid and tanh are still common. The results for these activation functions are shown in Table 4.3 and the test accuracies throughout the training are demonstrated in Figure 4.3.

	ConvoPool-CNN					
	Sigmoid		Tanh		ReLU	
	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy
CNN	99.39	99.33	99.38	99.29	99.48	99.41
BWN	98.62	98.5	98.73	98.66	98.88	98.75
HIN	98.36	98.28	98.3	98.22	98.32	98.23
BWHIN-NormalAvg	98.95	98.85	98.92	98.81	98.79	98.67
BWHIN-RandomAvg	98.34	98.25	98.58	98.51	98.96	98.88

(a) ConvoPool-CNN

	All-CNN					
	Sigmoid		Tanh		ReLU	
	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy
CNN	99.12	99.09	99.33	99.26	99.31	99.24
BWN	98.11	97.95	98.41	98.3	98.37	98.23
HIN	97.71	97.64	97.61	97.55	97.84	97.71
BWHIN-NormalAvg	98.18	98.02	98.31	98.2	98.4	98.23
BWHIN-RandomAvg	98.34	98.28	98.27	98.17	98.61	98.5

(b) All-CNN

Table 4.3: Test accuracy results (in percentage) for different activation functions at FC layer.

ReLU has the highest test accuracy for both architectures and most of the models. As an exception, for ConvPool-CNN architecture, sigmoid function has better classification results for HIN and BWHIN-Normal whose one of the sub-networks is HIN. For All-CNN architecture, tangent hyperbolic gives best results only for standard CNN and BWN. As shown in Figure 4.3, ReLU is the fastest activation function. It is an expected result, since the definition function of ReLU $f(x) = max(0, x)$ only requires a comparison. On the other hand, sigmoid and its more practical counterpart *tanh* have more complex computation due to the exponential term in their formula. Hence ReLU is the best option for the energy efficient neural networks.

4.2 Experiments on CIFAR-10

After we observe the effects of three parameters on energy efficient neural networks experimented on MNIST dataset, we choose the most suitable parameters for our CIFAR-10 architectures. Thus, ReLU is chosen as the activation function

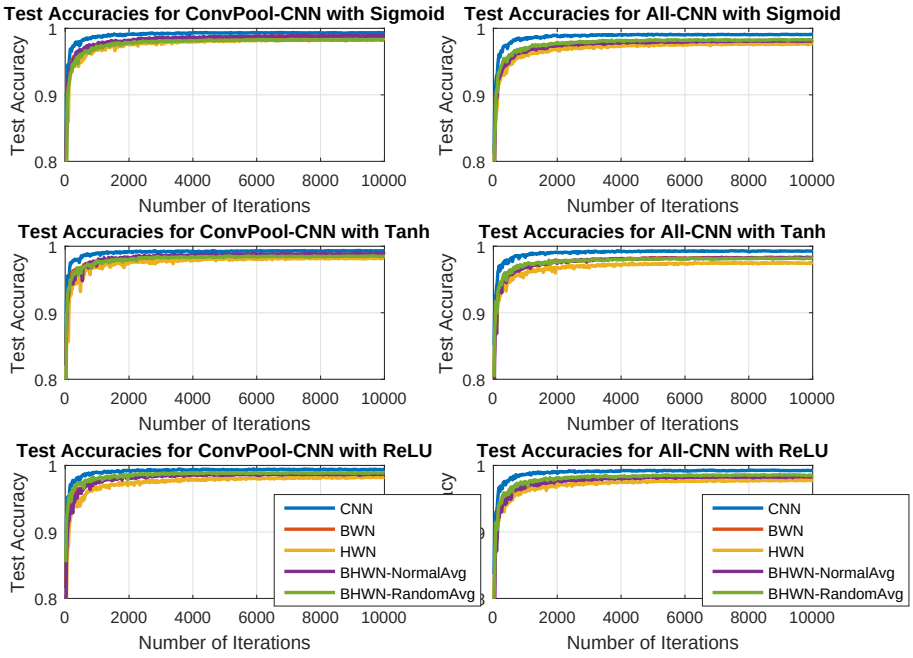

Figure 4.3: Test accuracy results for different activation functions at FC layer.

at the FC layer and ADAM algorithm is preferred as the optimizer. Since dropout is strongly encouraged for energy-efficient neural networks as examined in Section 4.1.2, we use once again dropout for CIFAR-10 models, but for three times. As illustrated in Table 3.2, the probability of retention at the convolutional layers is chosen as $p = 0.75$, while p is set as $p = 0.5$ at the FC layer. In Table 4.4, the results for CIFAR-10 database are demonstrated. The progress of the test accuracies is also shown in Figure 4.4.

	ConvoPool-CNN		All-CNN	
	Max Test Accuracy	Last Test Accuracy	Max Test Accuracy	Last Test Accuracy
CNN	82.64	81.2	77.32	75.32
BWN	68.72	64.92	65.36	62.88
HIN	61.36	58.72	11.76	9.92
BWHIN-NormalAvg	72.10	69.65	67.70	66.90
BWHIN-RandomAvg	72.65	71.15	67.30	63.80

Table 4.4: Test accuracy results (in percentage) for CIFAR-10 dataset.

According to the training accuracy results in Appendix B, the energy efficient neural networks are underfitting; HIN is not even trained for All-CNN architecture. If we increase the number of iterations without making the conventional

CNN overfit, the test accuracy results will be higher and closer to the standard CNN. Combined model BWHIN-Random has the best result for ConvPool architecture, while BWHIN-Normal has the best test accuracy with All-CNN architecture. Even though HIN is not even trained, combined models achieve better results than both BWN and HIN due to the robust nature of BWN.

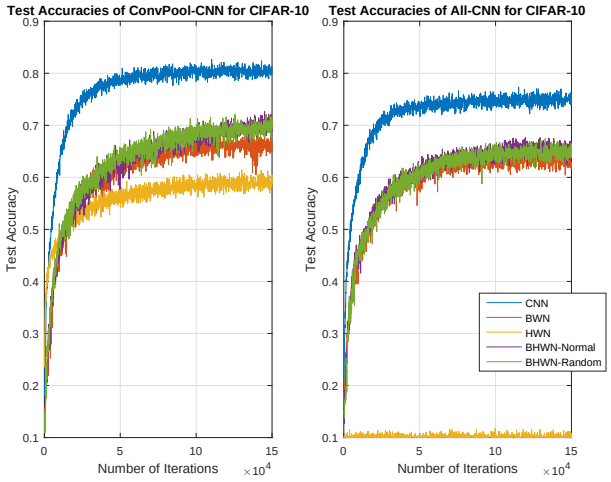

Figure 4.4: Test accuracy results for different activation functions at FC layer.

4.3 Effect of Architectures

Pooling layer is widely used in convolutional neural networks and its usage is highly recommended to handle the input sizes because it summarizes the responses over a whole neighborhood by reducing the dimensionality. Hence it can be said that pooling improves the computational efficiency [1]. As observed from the tables in the previous sections, it's clear that our CNN-based energy efficient neural networks with ConvPool architecture have better classification results than the All-CNN networks. Although networks with pooling layer have greater test accuracies, All-CNN can be preferred over ConvPool-CNN, since they are more energy-efficient. Here, both a convolutional layer with stride 1 and pooling layer is replaced by a convolutional layer which has stride greater than 1. Number of multiplications, multiplication-based operations in the convolutional layer is

decreased by a significant amount with strided convolution. While the down-sampling is performed along with full convolution, a large number of values are already computed in convolutional layer, then many of these values are discarded with pooling operation. This is computationally wasteful; it will take more time and use more memory than strided convolutional layer. If we take the risk of less test accuracy in order to achieve energy efficiency, strided convolution should be used instead of pooling.

A very important point must be stated about All-CNN. If we increase the capacity of All-CNN by increasing the size of the layers and/or making a deeper network, image classification can be performed without the loss of accuracy. In some cases, it may even give better results [22].

4.4 Comparison of Energy Efficient Neural Networks

We proposed four energy efficient neural networks and compared them with the conventional CNN and BWN which was already proposed in here [18]. The trade-off of the energy efficient neural networks is that we sacrifice the classification performance slightly in order to achieve energy efficiency.

Binary Weight Network is a very robust network to the changes of the hyperparameters. As seen from the tables 4.1 to 4.4, this model could train the network for any cases. On the other hand, Hadamard-transformed Image Network is slightly worse than BWN. The reason could be the slight change in the original architecture. If Hadamard transform is applied to an image whose width & height is not a power of 2, size of the image changes such that the width and height is increased to the next power of 2 of their original values. Hence, if we want to feed the Hadamard-transformed images to the network as input images, we need to modify the architecture of the network. For example, MNIST images have a size of 28*28*1 and they are zero-padded to the size of 32*32*1 in order to

perform Hadamard Transform. As a result, we change our original architecture for MNIST such that the output of the first convolutional layer is still 28*28*1. We modify the filter size as 5*5 and remove the zero-padding completely for the first convolutional layer. Hence, that might be the reason why HIN has worse performance and is a lossy network than BWN.

Combined models work as expected; they have better test accuracies than their sub-networks BWN and HIN. Our random average method seem to work for most cases, but for some cases it gives worse results than BWN. When the random averaging has the lower test accuracies than BWN, the conventional averaging is always better than both BWN and BWHIN-Random models. Hence both averaging techniques should be tried and it should be observed which one has better test accuracies even though the random-average works for most of the cases. Combined models are also a good solution to the failure of one sub-network. For example, when HIN is not trained, BWN compensates this failure and BWN-Normal/BWN-Random gives better results than both BWN and HIN. Hence it can be said that our combined model is also robust in case one of the subnetworks cannot be trained.

Chapter 5

Conclusion and Future Work

In recent years, artificial neural networks (ANN) are used widely to compute challenging tasks such as image classification, object detection and to solve real world problems in computer vision, robotics, natural language processing etc. Especially convolutional neural networks (CNN) achieve very satisfying results for these tasks. However, the number of parameters in these deep neural networks are so high that computers require high energy consumption and large memory size to handle complex problems. In case of mobile devices, this hardware and energy problem prevents us from implementing such algorithms effectively. In this thesis, we study CNN-based enery efficient neural networks for the image classification tasks of MNIST database. These models are BWN, HIN, BWHIN-Normal and BWHIN-Random. Their performances are compared with different parameters (optimizers, dropout probabilities and activation function at FC layer) and with different image datasets. Two different architectures are also proposed to investigate their energy efficiency, namely ConvPool-CNN and All-CNN. Their success at classification is evaluated by the test accuracy result.

According to our experiments on MNIST, ADAM optimizer seems to be best learning algorithm. In some cases, SGD and SGD with momentum couldn't train the model. As a solution, capacity of the neural networks can be increased

or a larger initial learning rate can be chosen. Different learning decay techniques might be helpful, too. Considering the regularization techniques, dropout is highly recommended because it is computationally inexpensive. Dropout with the correct probability of retention p not only improves the performance of the network, but also decreases the generalization gap between training and test loss. The optimal probability of the nodes kept in the layer should be between $[0.5, 1)$, but it is suggested that it should be closer to 1.0. It is also observed that ReLU is the best choice not only for the nonlinearity stages of the CNN, but also as the activation function at the FC layer. Since both sigmoid and tangent hyperbolic functions saturate and they involve computationally expensive operations like the exponential, ReLU can be preferred with its non-saturating form and simple implementation.

According to our observations on MNIST database, while the standard CNN has the highest test accuracy as 99.41% for ConvPool-CNN and 99.26% test accuracy for All-CNN, the best result for the energy efficient models is 98.88% and 98.5% for ConvPool-CNN and All-CNN architectures, respectively. As seen here, energy efficiency comes with a small loss of accuracy. This is the main drawback of the energy efficient neural networks. Both results are obtained by BWHIN-Random model with ADAM optimizer, dropout with the probability of keeping neurons $p = 0.75$ and ReLU activation at FC layer. Our CIFAR-10 architecture is also based on these parameters and verifies the MNIST results. Conventional CNN is almost trained in 300 epochs and has 81.20% test accuracy for ConvPool-CNN and 75.32% for All-CNN. On the other hand, energy efficient network models have the best test accuracies as 71.15% for ConvPool-CNN and 66.90% for All-CNN. We achieve the best results for CIFAR-10 with BWHIN-Random model and BWHIN-Normal models, respectively. Since CIFAR-10 models have larger capacity and more parameters than MNIST models, energy efficient models become slower than expected. As a result, the difference between the results of standard CNN and energy efficient CNN is larger. Slower training speed is another disadvantage of the energy efficient network models. These problems can be solved by increasing the number of epoch.

While comparing the architectures, ConvPool-CNN with convolutional-pooling

layer pair has slightly better results than All-CNN with strided convolutional layer. However, if we want to implement a energy efficient network, All-CNN should be chosen because it computes the convolutions with less parameters and it requires less memory storage than ConvoPool-CNN architecture. Once again there is a trade-off between better test accuracy results and more energy-efficient network. Test accuracies for All-CNN can be improved by increasing the layer size or the number of hidden layers. Considering the proposed networks, BWN gives really good results, while HIN sometimes struggles to train the network. The combined models BWHIN-Normal and BWHIN-Random, which are our principle contributions, certainly improves the performance of their sub-networks BWN and HIN. Hence as an energy efficient model, the combined model can be preferred with its superior classification performance.

As future work, proposed energy efficient neural networks can be used for the classification of other datasets such as CIFAR-100, Street View House Numbers (SVHN) dataset or ImageNet. The networks can be also implemented with bigger capacity by increasing the size of a hidden layer or with more hidden layers, i.e. as a deeper network. Other regularization or optimization techniques could be also investigated. Effects of data preprocessing or usage of unsupervised learning as pretraining on energy efficient NN might be another research topic.

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016.

[2] T.-Y. Lin, A. RoyChowdhury, and S. Maji, "Bilinear cnn models for fine-grained visual recognition," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1449–1457, 2015.

[3] F. F. Li, J. Johnson, and A. Karpathy, "Cs231n: Convolutional neural networks for visual recognition - neural networks part 3: Learning and evaluation," 2017. [Online; accessed 26-July-2017].

[4] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[5] M. Egmont-Petersen, D. de Ridder, and H. Handels, "Image processing with neural networks—a review," *Pattern recognition*, vol. 35, no. 10, pp. 2279–2301, 2002.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[9] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, pp. 1135–1143, 2015.

[10] H. Tuna, I. Onaran, and A. E. Cetin, "Image description using a multiplier-less operator," *IEEE Signal Processing Letters*, vol. 16, no. 9, pp. 751–753, 2009.

[11] C. E. Akbas, A. Bozkurt, M. T. Arslan, H. Aslanoglu, and A. E. Cetin, "L1 norm based multiplication-free cosine similarity measures for big data analysis," in *Computational Intelligence for Multimedia Understanding (IWCIM), 2014 International Workshop on*, pp. 1–5, IEEE, 2014.

[12] A. Suhre, F. Keskin, T. Ersahin, R. Cetin-Atalay, R. Ansari, and A. E. Cetin, "A multiplication-free framework for signal processing and applications in biomedical image analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 1123–1127, IEEE, 2013.

[13] C. E. Akbaş, A. Bozkurt, A. E. Çetin, R. Çetin-Atalay, and A. Üner, "Multiplication-free neural networks," in *Signal Processing and Communications Applications Conference (SIU), 2015 23th*, pp. 2416–2418, IEEE, 2015.

[14] A. Afrasiyabi, O. Yildiz, B. Nasir, F. T. Y. Vural, and A. E. Cetin, "Energy saving additive neural network," *arXiv preprint arXiv:1702.02676*, 2017.

[15] D. Badawi, E. Akhan, M. Mallah, A. Üner, R. Çetin-Atalay, and A. E. Çetin, "Multiplication free neural network for cancer stem cell detection in h-and-e stained liver images," in *SPIE Commercial+ Scientific Sensing and Imaging*, pp. 102110C–102110C, International Society for Optics and Photonics, 2017.

[16] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences*, p. 201604850, 2016.

[17] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 145–150, IEEE, 2016.

[18] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[19] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[20] İ. Çuğu, E. Şener, and E. Akbaş, "Microexpnet: An extremely small and fast model for expression recognition from frontal face images," *arXiv preprint arXiv:1711.07011*, 2017.

[21] B. J. Fino and V. R. Algazi, "Unified matrix treatment of the fast walsh-hadamard transform," *IEEE Transactions on Computers*, vol. 25, no. 11, pp. 1142–1146, 1976.

[22] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[23] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[24] S. S. Haykin, *Neural networks and learning machines*, vol. 3. Pearson, 2009.

[25] F. F. Li, J. Johnson, and A. Karpathy, "Cs231n: Convolutional neural networks for visual recognition - neural networks part 1: Setting up the architecture," 2017. [Online; accessed 12-December-2017].

[26] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*, pp. 9–50, Springer, 1998.

[27] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323, 2011.

[28] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. ICML*, vol. 30, 2013.

[29] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 625–660, 2010.

[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, pp. 448–456, 2015.

[31] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[32] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[34] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[35] Y. Bengio *et al.*, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[36] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[37] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*, pp. 267–285, Springer, 1982.

[38] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[39] F. F. Li, J. Johnson, and A. Karpathy, "Cs231n: Convolutional neural networks for visual recognition - convolutional neural networks: Architectures," 2017. [Online; accessed 24-July-2017].

[40] D. C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, p. 1237, Barcelona, Spain, 2011.

[41] C.-Y. Wu, M. Zaheer, H. Hu, R. Manmatha, A. J. Smola, and P. Krähenbühl, "Compressed video action recognition," *arXiv preprint arXiv:1712.00636*, 2017.

[42] M. Petrou and C. Petrou, *Image Processing: The Fundamentals*. John Wiley & Sons.

[43] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Pearson Education, 2007.

[44] W. K. Pratt, J. Kane, and H. C. Andrews, "Hadamard transform image coding," *Proceedings of the IEEE*, vol. 57, no. 1, pp. 58–68, 1969.

[45] S. Yang and A. Browne, "Neural network ensembles: combining multiple models for enhanced performance using a multistage approach," *Expert Systems*, vol. 21, no. 5, pp. 279–288, 2004.

[46] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade*, pp. 437–478, Springer, 2012.

[47] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[48] G. Hinton, "A practical guide to training restricted boltzmann machines," *Momentum*, vol. 9, no. 1, p. 926, 2010.

[49] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[50] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[51] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[52] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

Appendix A

MNIST Results

A.1 Test Accuracies

Overall test accuracy results for MNIST are shown in Table A.1 on page 63.

A.2 Training Accuracies

Results for the maximum training accuracy and the last training accuracy at the end of the iterations are shown in Table A.2 on page 64.

A.3 Training and Test Losses

Results for the training loss and the test loss at the end of the iterations are shown in Table A.3 on page 65.

		CNN		BWN		HIN		BWHIN-Normal		BWHIN-Random	
		Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.
Optimizers	SGD	99.26	99.17	98.46	98.36	11.35	11.35	98.27	98.15	98.9	98.78
	SGD-momentum	11.35	11.35	98.92	98.79	11.35	11.35	99.04	98.91	98.77	98.71
	ADAM	99.48	99.41	98.88	98.75	98.32	98.23	98.79	98.67	98.96	98.88
Dropout	1.0	99.46	99.35	98.89	98.85	98.27	98.18	99.01	98.85	98.73	98.64
	0.75	99.48	99.41	98.88	98.75	98.32	98.23	98.79	98.67	98.96	98.88
	0.5	99.48	99.38	98.83	98.77	98.02	97.91	98.66	98.61	98.74	98.59
FC Activation	Sigmoid	99.39	99.33	98.62	98.5	98.36	98.28	98.95	98.85	98.34	98.25
	Tanh	99.38	99.29	98.73	98.66	98.3	98.22	98.92	98.81	98.58	98.51
	ReLU	99.48	99.41	98.88	98.75	98.32	98.23	98.79	98.67	98.96	98.88

(a) Test accuracy results for ConvPool-CNN

		CNN		BWN		HIN		BWHIN-Normal		BWHIN-Random	
		Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.	Max Test Accu.	Last Test Accu.
Optimizers	SGD	99.12	99.09	98.11	98.05	96.48	96.35	97.24	97.13	98.09	98.07
	SGD-momentum	11.35	11.35	98.48	98.44	11.35	11.35	98.47	98.38	10.09	9.8
	ADAM	99.31	99.24	98.37	98.23	97.84	97.71	98.4	98.23	98.61	98.5
Dropout	1.0	99.2	99.13	98.43	98.38	97.73	97.65	98.32	98.21	98.38	98.3
	0.75	99.31	99.24	98.37	98.23	97.84	97.71	98.4	98.23	98.61	98.5
	0.5	99.3	99.15	98.49	98.36	98.01	98.01	97.95	97.94	98.55	98.48
FC Activation	Sigmoid	99.12	99.09	98.11	97.95	97.71	97.64	98.18	98.02	98.34	98.28
	Tanh	99.33	99.26	98.41	98.3	97.61	97.55	98.31	98.2	98.27	98.17
	ReLU	99.31	99.24	98.37	98.23	97.84	97.71	98.4	98.23	98.61	98.5

(b) Test accuracy results for All-CNN

Table A.1: Test accuracy results (in percentage) for MNIST database.

		CNN		BWN		HIN		BWHIN-Normal		BWHIN-Random	
		Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.
Optimizers	SGD	1	1	1	1	0.25	0.14	1	0.98	1	1
	SGD-momentum	0.2	0.08	1	1	0.2	0.16	1	0.99	1	1
	ADAM	1	1	1	1	1	0.99	1	1	1	1
Dropout	1.0	1	1	1	1	1	0.98	1	1	1	1
	0.75	1	1	1	1	1	0.99	1	1	1	1
	0.5	1	1	1	1	1	0.99	1	1	1	1
FC Activation	Sigmoid	1	1	1	1	1	1	1	1	1	1
	Tanh	1	1	1	1	1	0.99	1	1	1	1
	ReLU	1	1	1	1	1	0.99	1	1	1	1

(a) Training accuracy results for ConvPool-CNN

		CNN		BWN		HIN		BWHIN-Normal		BWHIN-Random	
		Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.	Max Training Accu.	Last Training Accu.
Optimizers	SGD	1	1	1	1	1	0.95	1	0.99	1	1
	SGD-momentum	0.21	0.11	1	1	0.22	0.16	1	1	0.19	0.06
	ADAM	1	1	1	1	1	0.99	1	1	1	1
Dropout	1.0	1	1	1	1	1	1	1	1	1	1
	0.75	1	1	1	1	1	0.99	1	1	1	1
	0.5	1	1	1	1	1	0.98	1	1	1	1
FC Activation	Sigmoid	1	1	1	0.99	1	1	1	1	1	0.98
	Tanh	1	1	1	1	1	0.97	1	1	1	1
	ReLU	1	1	1	1	1	0.99	1	1	1	1

(b) Training accuracy results for All-CNN

Table A.2: Training accuracy results.

		CNN		BWN		HIN		BWHIN-Normal		BWHIN-Random	
		Training Loss.	Test Loss.	Training Loss.	Test Loss.	Training Loss.	Test Loss.	Training Loss.	Test Loss.	Training Loss.	Test Loss.
Optimizers	SGD	0.193	2.7309	0.3306	4.859	229.4678	230.1058	4.8677	5.626	0.8196	3.7535
	SGD-momentum	230.9007	230.1091	0.793	4.0287	229.3678	230.1379	1.1557	3.9633	1.2483	5.7455
	ADAM	0.0024	2.3116	0.0375	4.7709	2.6586	5.7103	1.0923	3.7193	0.1579	4.187
Dropout	1.0	0.001	3.2069	0.0328	5.8298	4.8138	5.6002	0.8779	4.0245	0.3581	5.0914
	0.75	0.0024	2.3116	0.0375	4.7709	2.6586	5.7103	1.0923	3.7193	0.1579	4.187
	0.5	0.0529	2.1396	0.4998	4.1591	4.4832	6.7218	1.7438	4.3293	1.7236	4.2315
FC Activation	Sigmoid	0.0633	2.0488	0.4037	4.7846	1.7681	5.1843	1.1913	3.4413	2.3236	5.2174
	Tanh	0.0389	2.4278	0.4691	5.0157	2.2674	5.4531	1.2296	3.8486	1.1551	4.3545
	ReLU	0.0024	2.3116	0.0375	4.7709	2.6586	5.7103	1.0923	3.7193	0.1579	4.187

(a) Losses for ConvPool-CNN

		CNN		BWN		HIN		BWHIN-Normal		BWHIN-Random	
		Training Loss.	Test Loss.	Training Loss.	Test Loss.	Training Loss.	Test Loss.	Training Loss.	Test Loss.	Training Loss.	Test Loss.
Optimizers	SGD	0.1288	3.0332	1.4822	6.2232	16.3204	11.7929	3.1096	9.0029	1.297	6.4973
	SGD-momentum	230.1908	230.1297	0.0784	6.4308	229.5455	230.1171	0.3442	6.5361	NaN	NaN
	ADAM	0.0013	3.1021	0.0228	7.7545	1.6347	7.3568	0.4416	6.3945	0.0852	7.9288
Dropout	1.0	0.0032	5.1362	0.0116	10.3614	0.5211	7.7626	0.2133	8.1662	0.0111	11.4236
	0.75	0.0013	3.1021	0.0228	7.7545	1.6347	7.3568	0.4416	6.3945	0.0852	7.9288
	0.5	0.0795	2.6034	1.0097	6.3271	5.4895	6.9676	1.8281	7.3975	0.8113	6.027
FC Activation	Sigmoid	0.0572	2.8594	2.0648	6.5707	0.575	7.1807	0.5278	6.0022	4.5582	6.0218
	Tanh	0.0433	2.4753	0.5702	5.971	3.2749	8.3406	0.3445	5.6817	0.1256	6.5881
	ReLU	0.0013	3.1021	0.0228	7.7545	1.6347	7.3568	0.4416	6.3945	0.0852	7.9288

(b) Losses for All-CNN

Table A.3: Results for training and test loss .

Appendix B

CIFAR-10 Results

Test accuracy results, training accuracies and training & test losses for CIFAR-10 are shown in Table B.1 on page 67.

		CNN	BWN	HIN	BWHIN-Normal	BWHIN-Random
Test Accuracy	Max Test Accu.	82.64	68.72	61.36	72.10	72.65
	Last Test Accu.	81.20	64.92	58.72	69.65	71.15
Training Accuracy	Max Training Accu.	1.00	0.89	0.94	0.91	0.90
	Last Training Accu.	1.00	0.75	0.87	0.82	0.85
Loss	Training Loss	0.0578	68.7528	44.5814	56.2987	48.2561
	Test Loss	84.5149	101.9648	127.8482	92.6407	87.4764

(a) Results for ConvPool-CNN

		CNN	BWN	HIN	BWHIN-Normal	BWHIN-Random
Test Accuracy	Max Test Accu.	77.32	65.36	11.76	67.70	67.30
	Last Test Accu.	75.32	62.88	9.92	66.90	63.80
Training Accuracy	Max Training Accu.	1.00	0.99	1.00	1.00	0.98
	Last Training Accu.	1.00	0.97	0.98	0.91	0.90
Loss	Training Loss	0.2128	22.3079	15.4764	29.7083	32.9094
	Test Loss	111.700531	113.0176	518.2485	104.134033	108.7281

(b) Results for All-CNN

Table B.1: Overall results for CIFAR-10 dataset. (Note that test accuracies are in precentages.)