Deep Learning for Handwritten Digits Recognition Using MATLAB Toolbox

by

JiaCong Chen
B.Eng., University of Victoria, 2018

A Project Report Submitted in Partial fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

SUPERVISORY COMMITTEE

Deep Learning for Handwritten Digits Recognition
Using MATLAB Toolbox

by

JiaCong Chen

B.Eng., University of Victoria, 2018

Supervisory Committee

Dr. Wu-Sheng Lu, Department of Electrical and Computer Engineering, University of Victoria (Supervisor)

Dr. XiaoDai Dong, Department of Electrical and Computer Engineering, University of Victoria (Departmental Member)

ABSTRACT

In this report, we describe several neural network architectures for the classification of handwritten digits. In particular, our attention is focused on the class of convolutional neural networks (CNNs) for performance superiority. By using MATLAB deep learning toolbox, we provide the implementation details necessary for constructing and applying CNNs to a high-quality data set known as MNIST which collects as many as 60,000 handwritten digits for training and 10,000 digits for testing the CNNs.

This report also presents several variants of the original LeNet-5 architecture, which has been known for its excellent performance for classifying handwritten digits, for potential performance improvement. Using the deep learning toolbox, extensive simulation studies are conducted for performance evaluation and comparisons between various neural networks as well as two well-known classifiers that are not based on neural-networks.

Keywords: Deep learning, Convolutional Neural Networks, MATLAB.

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Wu-Sheng Lu for his continuous support and his invaluable suggestions without which I could not have completed this project. His patience and encouragement have backed me up the whole time, and he has always been open and honest to me, I would have never completed my degree without his supervision.

In addition, I would like to thank my parents and my friends for their motivation and support through my studies.

Abbreviations

AI	Artificial Intelligence
BA	Backpropagation Algorithm
CNN	Convolutional Neural Network
FM	Feature Map
GPU	Graphics Processor Unit
HOG	Histogram of Oriented Gradient
HWDR	Handwritten Digit Recognition
KKT	Karush-Kuhn-Tucker
LRF	Local Receptive Field
LSTM	Long Short-Term Memory
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
NN	Neural Network
PCA	Principal Component Analysis
QP	Quadratic Programming
ReLU	Rectified Linear Unit
RGB	Red-Green-Blue
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
Tanh	Hyperbolic Tangent

Chapter 1

Introduction

According to Wikipedia, machine learning (ML) is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead. It is seen as a subset of artificial intelligence (AI). Machine learning algorithms build a mathematical model based on sample data, known as *training data*, in order to make predictions or decisions without being explicitly programmed to perform the task. Machine learning algorithms are used in a wide variety of applications, such as email filtering and computer vision, where it is difficult or infeasible to develop a conventional algorithm for effectively performing the task.

1.1 A Brief History of ML

Arthur Samuel, a pioneer in the field of computer gaming and artificial intelligence, coined the term "machine learning" in 1959 while at IBM. A representative book of the machine learning research during 1960s was Nilsson's book on learning machines, dealing mostly with machine learning for pattern classification [1]. The interest of machine learning related to pattern recognition continued during 1970s, as described in the book of Duda and Hart in 1973 [2]. As a scientific endeavor, ML grew out of the quest for artificial intelligence. Already in the early days of AI as an academic discipline, some researchers were interested in having machines learn from data. They attempted to approach the problem with various symbolic methods, as well as what were then termed "neural networks"; these were mostly perceptrons and other models that were later found to be reinventions of the generalized linear models of statistics [3]. Probabilistic reasoning was also employed, especially in automated medical diagnosis [4].

However, an increasing emphasis on the logical, knowledge-based approach caused a rift between AI and ML. Probabilistic systems were plagued by theoretical and practical problems of data acquisition and representation [4]. By 1980, expert systems had come to dominate AI, and statistics was out of favor [5]. Work on symbolic/knowledge based

learning did continue within AI, leading to inductive logic programming, but the more statistical line of research was now outside the field of AI [4]. Neural networks research had been abandoned by AI and computer science around the same time. This line, too, was continued outside the AI/CS field, as "connectionism", by researchers from other disciplines including Hopfield, Rumelhart, and Hinton. Their main success came in the mid-1980s with the reinvention of backpropagation [4].

Machine learning, reorganized as a separate field, started to flourish in the 1990s. The field changed its goal from achieving artificial intelligence to tackling solvable problems of a practical nature. It shifted focus away from the symbolic approaches it had inherited from AI, and toward methods and models borrowed from statistics and probability theory [5]. It also benefited from the increasing availability of digitized information, and the ability to distribute it via the Internet.

More recent years have witnessed a surge of ML applications in practically every field of human activities and we are told that we now live in the “big data” era. The big-data era has been enabled by the rapid advances in data collection in terms of both quality and quantity, and computation technology, especially the development of powerful graphics processor units (GPUs) [6].

1.2 Neural Networks

1.2.1 Basic Architectures

A neural network, or more precisely artificial neural network, is a collection of connected and tunable units, which is called artificial neurons, analogous to neurons in an animal brain. Each connection, named a synapse, can pass signals from one unit to another, and network architecture refers the way neurons are connected to each other.

A single-layer neural network of S neurons is shown in Figure 1. The network receives R inputs $\{p_j, j = 1, 2, \ldots, R\}$ which, at the ith neuron are combined to produce a weighted sum plus a bias as

$$n_i = \sum_{j=1}^{R} w_{i,j} p_j + b_i \quad \text{for } i = 1, 2, \ldots, S$$

which is then fed into an activation function to generate ith output $a_i = f_i(n_i)$ for $i = 1, 2,$

…, *S*. The activation function may be linear or nonlinear. An example of linear activation is the identity function $f(z)=z$, examples of nonlinear activation functions include the sign function

$$f(z)=\text{sign}(z)=\begin{cases}1 & \text{if } z\geq 0\\ -1 & \text{if } z<0\end{cases}$$

and sigmoid function

$$f(z)=\frac{1}{1+e^{-z}}$$

which varies from 0 to 1 as *z* varies from $-\infty$ to ∞. The sign function is a discontinuous function and assumes only two output values which, as expected, is useful for data and pattern classifications. The sigmoid function on the other hand assumes continuous and differentiable output, and as such it is often interpreted as probability which explains why the sigmoid function is useful in both classification and regression. For several other types of activation functions, see Section 2.2.4.

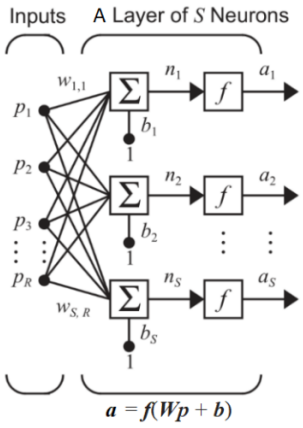

Figure 1. A single-layer network with identical activation functions.

If we let

$$\boldsymbol{p}=\begin{bmatrix}p_1\\ p_2\\ \vdots\\ p_R\end{bmatrix},\boldsymbol{w}_i=\begin{bmatrix}w_{i,1}\\ w_{i,2}\\ \vdots\\ w_{i,R}\end{bmatrix},\boldsymbol{W}=\begin{bmatrix}\boldsymbol{w}_1^T\\ \boldsymbol{w}_2^T\\ \vdots\\ \boldsymbol{w}_S^T\end{bmatrix},\boldsymbol{b}=\begin{bmatrix}b_1\\ b_2\\ \vdots\\ b_S\end{bmatrix},\boldsymbol{a}=\begin{bmatrix}a_1\\ a_2\\ \vdots\\ a_S\end{bmatrix}$$

the network structure can be characterized as

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{w}_1^T \boldsymbol{p} + b_1) \\ f(\boldsymbol{w}_2^T \boldsymbol{p} + b_2) \\ \vdots \\ f(\boldsymbol{w}_S^T \boldsymbol{p} + b_S) \end{bmatrix} = \boldsymbol{f}\left(\boldsymbol{W}\boldsymbol{p} + \boldsymbol{b}\right)$$

where $\boldsymbol{f}$ denotes a vector-in-vector-out function that acts as function f for each input.

An important special instance of the network in Fig. 1 is when the number of output is reduced to one. The input-output relationship in this case becomes a vector-in-scalar-out mapping

$$a = f(\boldsymbol{w}^T \boldsymbol{p})$$

and the network coincides with the classic notion of *perceptron* investigated by F. Rosenblatt in 1950s [7].

A neural network may have more than one layer, and each layer has its own weight matrix and bias. For example, Fig. 2 depicts a network with two hidden layers that sit between the input and output layers. The term "hidden layer" reflects the fact that it is not directly accessible from the environment outside the network.

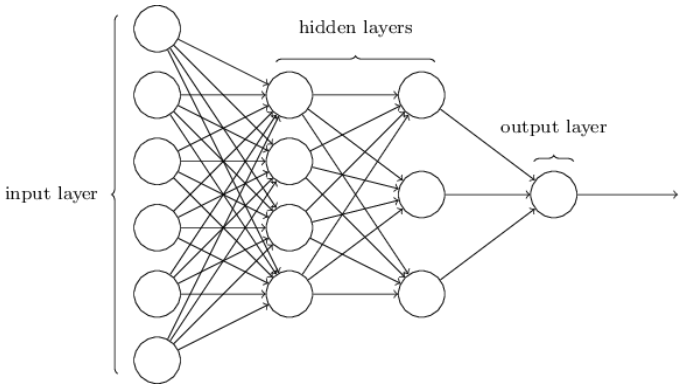

Figure 2. A neural network with two hidden layers and a single output.

In case the input is an image, then the pixels of the image may be arranged as a vector and each node in the input layer receives the value of an individual pixel. Figure 3 shows a network with a 15-neuron hidden layer and 10 outputs; the input is a 28×28 image that is arranged as a 784-dimesional vector. Obviously, the network is well suited for the

handwritten digits from MNIST.

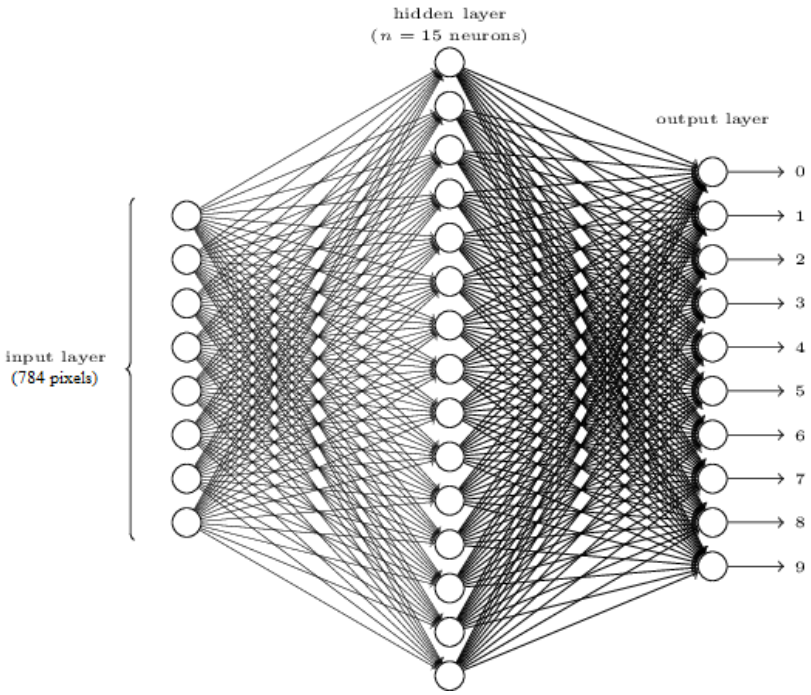

Figure 3. A simple network with one hidden layer and ten outputs for handwritten digits from MNIST.

For a general multilayer network with k hidden layers, there are $k + 1$ sets of parameters which are denoted by $\{\boldsymbol{W}_i, \boldsymbol{b}_i\}$ for $i = 1, 2, \ldots, k + 1$, and the input-output relationship of the network can be characterized by the recursive equations

$$\begin{aligned}
&\boldsymbol{a}_1 = \boldsymbol{f}_1(\boldsymbol{W}_1\boldsymbol{p} + \boldsymbol{b}_1) && \text{input to hidden layer 1}\\
&\boldsymbol{a}_{i+1} = \boldsymbol{f}_{i+1}(\boldsymbol{W}_{i+1}\boldsymbol{a}_i + \boldsymbol{b}_{i+1}) \quad \text{for } i = 1,\ldots,k-1 && \text{hidden layer } i \text{ to hidden layer } i+1\\
&\hat{\boldsymbol{y}} = \boldsymbol{f}_{k+1}(\boldsymbol{W}_{k+1}\boldsymbol{a}_k + \boldsymbol{b}_{k+1}) && \text{hidden layer } k \text{ to output layer}
\end{aligned}$$

where $\hat{\boldsymbol{y}}$ denotes the network's output. As expected, multilayer networks are more powerful than their single-layer counterparts, especially when the size of the dataset involved is sufficiently large. In effect, in this case even a two-layer network can be trained to approximate most functions with better accuracy than a single-layer network.

More specifically, an earlier version of the *universal approximation theorem*, proved by Cybenko for sigmoid activation functions [8], states that a multilayer perceptron is capable to approximate any continuous functions on a compact subset of $\mathbb{R}^n$. Later in 1991, Hornik showed that the theorem is not limited to the specific choice of the activation function [9].

The universal approximation theorem can analytically be described as follow. Let φ be a bounded and monotonically increasing continuous function, I_m be the m-dimensional unit hypercube $[0, 1]^m$, $C(I_m)$ be the space of continuous function on I_m, and $f \in C(I_m)$ be any continuous function on I_m. Then for any $\epsilon > 0$, there exists an integer N, real constants c_i, b_i, and real vectors $\boldsymbol{w}_i \in \mathbb{R}^m$ for $i = 1, 2, \ldots, N$ such that

$$\left| \sum_{i=1}^{N} c_i \varphi(\boldsymbol{w}_i^T \boldsymbol{p} + b_i) - f(\boldsymbol{p}) \right| < \epsilon$$

for all $\boldsymbol{p} \in I_m$.

1.2.2 Training a Neural Network

A neural network will not be of use until its parameters, namely the weights and biases, are properly tuned so that the network can respond to unseen inputs and predict their labels (i.e. true outputs) reasonably well. For neural networks for supervised learning, this tuning process is performed by the *backpropagation algorithm* (BA) using labeled training data [10].

However, to understand the BA there is one important issue to address that is the notion of *loss function*. Roughly speaking, given an input a loss function measures the performance of a neural network in terms of a "gap" which is a scalar indicating the closeness of the label predicted by the net to the true label of the given input, so we know the net performs better when the loss function gets smaller, and the learning is fulfilled by minimizing the loss function with respect to the net's adjustable parameters. Described below are two representative choices of loss function:

- L_2 (least squares) loss which is defined by

$$L(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{2} \sum_{\boldsymbol{p} \in \mathcal{D}} \| \hat{\boldsymbol{y}}(\boldsymbol{p}, \boldsymbol{W}, \boldsymbol{b}) - \boldsymbol{y}(\boldsymbol{p}) \|_2^2$$

where $\mathcal{D}$ is the training data, $\{\boldsymbol{W}, \boldsymbol{b}\}$ denotes the set of weights and biases involved in the entire net, $\hat{\boldsymbol{y}}(\boldsymbol{p}, \boldsymbol{W}, \boldsymbol{b})$ is the net's output given input $\boldsymbol{p}$ and net's parameters $\{\boldsymbol{W}, \boldsymbol{b}\}$, and $\boldsymbol{y}(\boldsymbol{p})$ is the ground-truth output associated with $\boldsymbol{p} \in \mathcal{D}$.

• *Cross-entropy* loss for *K*-category prediction. Consider a network for the classification of a *K*-class data set $\mathcal{D} = \{(\boldsymbol{p}_n, l_n), n = 1,2,...,N\}$ with class label $l_n \in \{1,2,...,K\}$. There are *K* nodes in the output layer of the network, where each activation function assumes the form

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{j=1}^{K} e^{a_j}} \qquad \text{for } i = 1, 2, \ldots, K$$

which are known as *softmax* functions. The $\{\hat{y}_i, i = 1,2,...,K\}$ correspond to the probabilities of the *K* classes. For a single instance $\boldsymbol{p}_n$ with class label $l_n \in \{1,2,...,K\}$, the cross-entropy loss is defined as $-\log(\hat{y}_{l_n})$ and hence the total cross-entropy loss over the entire training data $\mathcal{D}$ is given by

$$L(W,b) = -\sum_{n=1}^{N} \log(\hat{y}_{l_n})$$

There are several important features that the loss functions presented above as well as many others share:

• loss functions cannot be defined without training data;

• they assume the form $L = \sum_{n=1}^{N} L_n$ with a large *N*;

• L_n are similar to each other; and

• typically, each L_n involves only *one* input $\boldsymbol{p}_n$.

Having chosen an appropriate loss function for a given neural network, training the network is essentially a process of searching and secure a set of optimal parameters $\{\boldsymbol{W}^*, \boldsymbol{b}^*\}$ that solves the problem

$$\underset{\boldsymbol{W},\boldsymbol{b}}{\text{minimize}} \; L(\boldsymbol{W},\boldsymbol{b})$$

This is an unconstrained problem for which many algorithms are available [11]. Regardless of which algorithm is chosen, a key quantity that is always required to compute in order for the network to perform parameter tuning is the *gradient* of the loss function with respect

to parameters {$\boldsymbol{W}$, $\boldsymbol{b}$}, namely,

$$\nabla L = \sum_{n=1}^{N} \nabla L_n \text{ where } \nabla L_n = \begin{bmatrix} \nabla_{\boldsymbol{W}} L_n \\ \nabla_{\boldsymbol{b}} L_n \end{bmatrix}$$

Since the size of the training data N is usually large, computing ∇L is quite expensive. This problem is effectively addressed by employing the stochastic gradient descent (SGD) algorithm where the full-scale gradient $\nabla L = \sum_{n=1}^{N} \nabla L_n$ is replaced with a "partial" gradient obtained by computing and combing a small number of randomly selected component gradients $\{\nabla L_{n_i}, i = 1, 2, \ldots, k\}$, namely, $\sum_{i=1}^{k} \nabla L_{n_i}$.

What remains to be addressed is how each individual component gradient ∇L_{n_i} is calculated. For feedforward neural networks this is done by the backpropagation algorithm [10] which leverages the chain rule of differential calculus that calculate the partial derivatives of a composition function in terms of summations of "local-gradient" products over the various "paths". The BA does that efficiently using *dynamic programming*, which results in a computational procedure consisting of two phases, a forward phase and a backward phase [6]. In the forward phase, the inputs for a training instance are fed into the network, resulting in a forward cascade of computations across the layers by using the current set of weights. The output of the network is then compared with that of the training instance and the derivative of the loss function with respect to the output is computed. The computation now continues with the backward phase in order to get the gradient of the loss function. This is done by using the chain rule of calculus in the backward direction starting from the output layer, hence the name of the algorithm. We refer the reader to reference [6] and [12] for lucid exposition of the BA.

1.3 Recognition of Characters and Images

Given their ability to handle a large amount of inputs and process them to infer hidden as well as complex and nonlinear relationships, neural networks have played an important role in image processing, especially in image and character recognition which, among other things, find applications in fraud detection bank checks and general security assessments.

The primary challenge arising from the handwritten digits recognition (HWDR) problem lies in the fact that handwritten digits (within the same digit class) vary widely in terms of shape, line width, and style, even when they are normalized in size and properly centralized [13]-[16].

One of the well-known datasets used in the study of HWDR problem is MNIST which stands for Modified National Institute of Standards and Technology database. MNIST offers two separate datasets. The first dataset contains 60,000 training images and their corresponding digits from 0 to 9, and the second dataset contains 10,000 testing images and their corresponding digits. Each of the images is an 8-bit grayscale image of size 28 × 28, see Figure 4 for a sample image from MNIST representing digit 5.

The dataset is available from the MNIST website [17] which consists of four zipped files, namely **`train-images-idx3-ubyte.gz`** for training images (9912422 bytes); **`train-labels-idx1-ubyte.gz`** for the labels of the training images (28881 bytes); **`t10k-images-idx3-ubyte.gz`** for test images (1648877 bytes); and **`t10k-labels-idx1-ubyte.gz`** for the labels of the test images (4542 bytes).

Figure 4. A handwritten digit "5" from MNIST.

In the next chapter, we focus on a class of multilayer neural networks, known as convolutional neural networks (CNNs), which finds great success in addressing the HWDR problem. In Chapter 3, the performance of CNNs as applied to the HWDR problem is evaluated in comparison with a simple fully-connected multilayer neural network with a feedforward structure and sigmoid activations.

Chapter 2

Convolutional Neural Networks for Deep Learning

Deep learning is a branch of ML that assists computers to do what comes naturally to humans: learning from experiences. One of the widely recognized mechanisms for deep learning has been the class of *convolutional neural networks* (CNNs) which are proven especially suitable for signal, including text, speech, image, and video, recognition and classification. As such CNNs are the subject of exposition in this chapter.

2.1 Background

CNNs are inspired by the biological structure of visual cortex that contains arrangements of simple and complex cells [18]. In 1959, Hubel and Wiesel [18] reported that cells in animal visual cortex are responsible for detecting light in receptive fields. Inspired by this discovery, Fukushima proposed the neocognitron in 1980 [19], which was considered by many as the predecessor of CNN. Then in 1989, LeCun *et al*. published the seminal paper [20] building the modern framework of CNNs which was further enhanced ten years later [21]. In particular, LeCun and collaborators developed a multilayer neural network named LeNet-5 to classify handwritten digits that can be trained with backpropagation algorithm [21]. In [17], LeNet-5 is shown to be able to represent an original image effectively by identifying visual patterns of the image directly from raw pixels with little-to-none processing.

2.2 Basic Architecture of CNNs

The architecture of CNNs is designed to work with grid-structured inputs that have strong spatial dependencies in local regions of the grid and hence is well suited to deal with images. CNNs can also of use for other forms of sequential data such as text and time series [6].

2.2.1 Convolutional Layers

A defining characteristic of CNNs is the operation of *convolution*. For 2-dimensional input data like images, convolution is an operation involving a kernel, which in the present case is a grid-structured set of weights, and a grid-structured input array, where inner (dot) product of the kernel with a same-size set of local samples from the input, called *local*

receptive field, is performed, and the operation continues between the kernel and a slightly shifted set of input samples and so on, until the entire input is covered. The size of the kernel (hence the local receptive field) is small enough to ensure the convolution catches local features of the input. Let the input image be denoted by $\boldsymbol{X} = \{x_{i,j}\}$ and the set of weights involved in the convolution be rectangular of size L by M as $\boldsymbol{W} = \{w_{l,m}, l = 0,1,...,L-1; m = 0,1,...,M-1\}$. The full-scale 2-dimensional convolution $\boldsymbol{X} \otimes \boldsymbol{W}$ is a matrix whose (i, j)th component is given by

$$\sum_{l=0}^{L-1}\sum_{m=0}^{M-1} w_{l,m} x_{i+l,j+m}$$

The term "full-scale" here refer to the requirement that the above convolution is performed covering the entire image from upper-left corner of the image scanning from left to right, shifting by one pixel at a time (see Figures 5 and 6), then back to the left while shifting down by one-pixel position at a time, until it reaches the last image block at the bottom-right corner. The result of the convolution is an "image" with L – 1 less rows and M – 1 less columns relative to the input image.

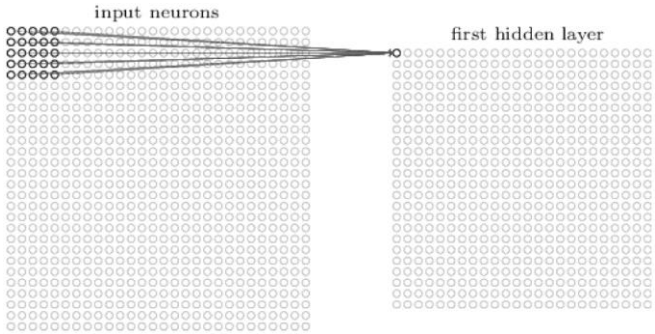

Figure 5. Local receptive field 1.

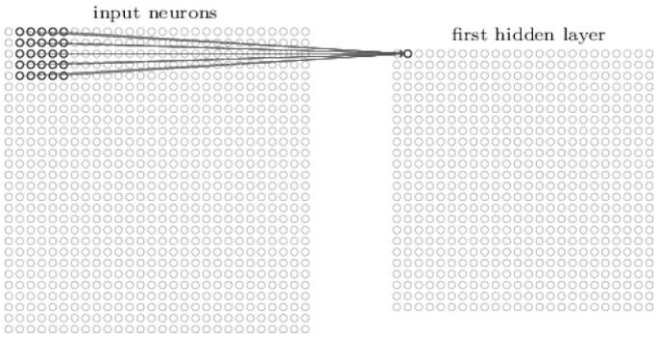

Figure 6. Local receptive field 2.

For an input digit from MNIST, for example, with $L = M = 5$ the convolution (plus a bias b) yields a matrix of size 24 by 24. Each components is then fed into an activation function f to produce a (local) output as

$$a_{i,j} = f\left(b + \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} w_{l,m} x_{i+l,j+m} \right) \tag{2.1}$$

and matrix $\{a_{i,j}\}$ is called a *feature map* because the matrix is expected to contain some kind of local features from the input when the weights $\boldsymbol{W} = \{w_{l,m}\}$ are properly chosen. It is important to stress that all local convolution operations are carried out using the *same* weight matrix $\boldsymbol{W}$ and bias b. This is referred to as convolution with *shared weights and biases*. In this way, the number of parameters in a CNN is drastically reduced relative to a fully connected network. On the other hand, since one set of $\{\boldsymbol{W}, b\}$ (called a *kernel*) can only extract one type of features from the input, a convolutional hidden layer of a CNN typically uses several small-size kernels $\{\boldsymbol{W}_i, b_i\}$ for $i = 1, 2, \ldots, J$ to generate multiple feature maps for extracting distinctive features of the input, see Figure 7 as an example where a hidden layer includes three feature maps [12].

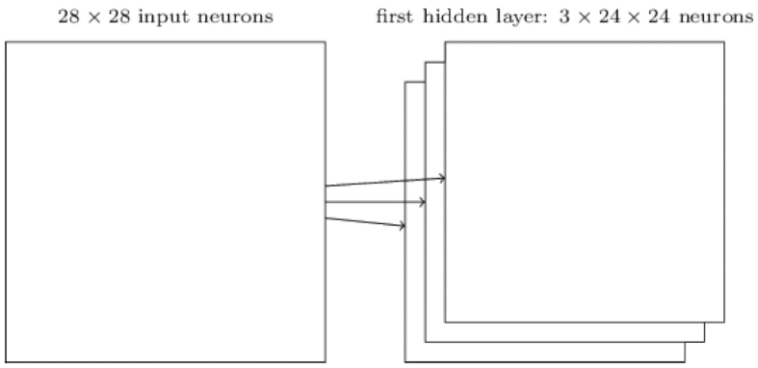

Figure 7. A hidden layer including three feature maps.

For illustration of complexity reduction by shared weights and biases, consider a convolutional layer from input (which is an image of size 28 by 28) to the first hidden layer, where a total of $J = 18$ kernels, each with $L = M = 7$, are used to generate 18 feature maps. The layer in this case involves $(7 \times 7 + 1) \times 18 = 900$ parameters. For comparison, a fully connected layer with 28×28 input nodes and a modest hidden layer with 30 neurons, the layer will have to use $(28 \times 28 + 1) \times 30 = 23550$ parameters that are 26 times more than

those involved in the CNN.

Finally, we remark that in practical implementation of a convolution layer, it is sometimes desirable to generate feature maps with the same size (or another specific size) as the input array. This can be achieved by padding appropriate number of zero rows at the top and bottom as well as zero columns on the left- and right-hand sizes of the input array. We call the number of zero rows or columns on each side *padding size*.

From (2.1), we see that generating a feature map requires an activation function *f* and its selection affects the network's performance in a significant manner. This issue will be addressed in Sec. 2.2.3.

2.2.2 Batch Normalization Layers

A convolutional layer is often followed by a *batch normalization* layer where the data from previous layer are normalized, scaled, and shifted mini-batch by mini-batch in order to improve the network's stability. The steps carried out by a batch normalization layer are summarized in Table 1.

Table 1. Batch Normalization Steps.

Input: a mini-batch of *m* data samples $\mathcal{B} = \{x_1, x_2, \ldots, x_m\}$; parameters to be leaned: γ, β.

Output: normalized, scaled, and shifted m data samples $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i), i = 1, 2, ..., m\}$.

Step 1. Compute mean of $\mathcal{B}$: $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$

Step 2. Compute variance of $\mathcal{B}$: $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$

Step 3. Normalize $\mathcal{B}$: $\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$ for $i = 1, 2, \ldots, m$.

Step 4. Scaling and shifting: $y_i = \gamma \hat{x}_i + \beta \triangleq \mathrm{BN}_{\gamma,\beta}(x_i)$ for $i = 1, 2, \ldots, m$.

From the Table, it is quite clear that the first three steps of the process normalize a mini-batch of data samples to have zero mean and unity variance. Step 4 involves two parameters, γ and β, which are to be learned for properly scaling and shifting the normalized data [22]. These parameters turn out to be useful in stabilizing the network's performance when they are tuned by an optimization (e.g., stochastic gradient decent) algorithm during the training phase.

2.2.3 Activation Layers

The notion of activation was introduced in Sec. 1.2.1 as a nonlinear single-input-single-output function such as sign and sigmoid functions to mimic what happens in a biological neuron when it was stimulated by a certain amount of input. In addition to these options, other types of activation functions are available, which include the *rectified linear units* (ReLUs) which responds to input x with $\max(0, x)$; hyperbolic tangent (tanh) defined by

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

which varies from –1 to 1 as x varies from $-\infty$ to ∞. These and several other popular activation functions are summarized in Fig. 8.

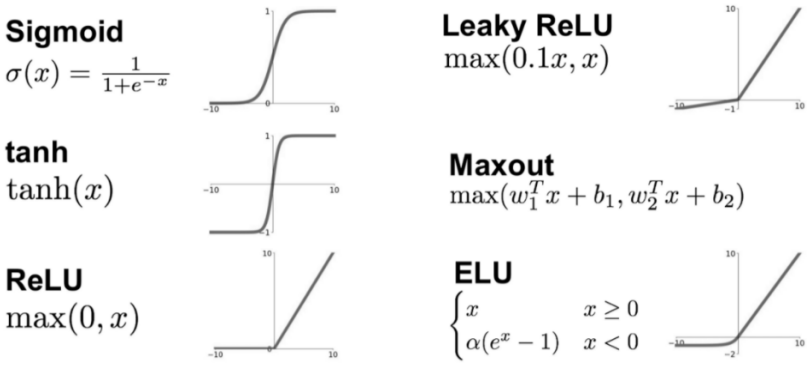

Figure 8. Activation functions.

By definition, for any negative inputs the ReLU yields a zero value hence the neuron remains inactive while preserves any positive inputs without modification. Relative to other popular activation functions such as sigmoid and tanh, ReLU is straightforward to compute and often improves the net's efficiency as few neurons would be activated simultaneously at any given time. In effect, a CNN employing ReLU may converge as six times faster as CNNs using sigmoids or tanh's.

On the other hand, note that the derivative of ReLU is identically zero for negative inputs, which implies that those weights that yield negative convolution outputs will not be updated during backpropagation. A simple way to fix the problem is to replace ReLU with a *leaky* ReLU, defined by $\max(0.1x, x)$, that partially responds to negative inputs instead of a complete shutdown.

As an example for illustration, consider a CNN for handwritten digits recognition with

inputs from MNIST, where the first hidden layer includes 18 feature maps of size 22×22 that are generated by convolving an input digit (of size 28×28) with 18 kernels of size 7×7, each is then fed into a ReLU. The values of the weights (and associated biases) in these kernels are optimized using the backpropagation algorithm, which are displayed in Figure 9 (after normalizing the values to interval [0, 1]) where each kernel is shown as a 7×7 image with a brighter square representing a weight value closer to 0. By inspecting the images in Fig. 9 as a visualized version of the optimal weights, it is hard to explain why they can do a good job in extracting important features of the input. Unfortunately, this is not a coincidence that only occurs in this example, but an unwelcoming characteristic of many deep supervised learning systems using CNNs [23]. In any event, we decide to take a realistic point of view by referring the resulting weights as optimal at a higher level of abstraction while verifying their performance as applied to real-world data sets. With an input digit 5, the feature maps created by the optimized convolutional kernels are depicted in Figure 10 where each feature map is shown as a 22×22 image. As can be seen, features of various parts of the input have been extracted. More details of this CNN will be presented in subsequent chapters.

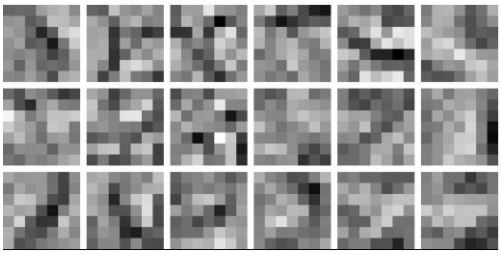

Figure 9. 18 optimized kernels that convolve with an input digit to extract its features.

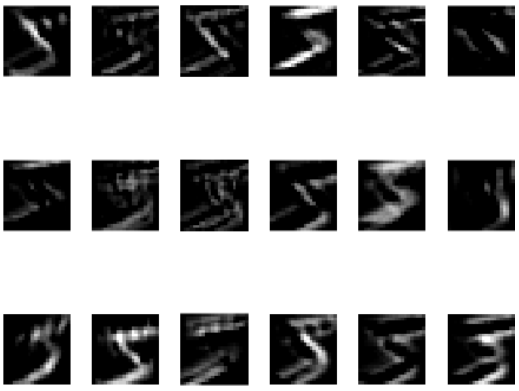

Figure 10. 18 feature maps in the first hidden layer for a digit 5 as input.
As expected, features in various parts of the input are extracted.

2.2.4 Pooling Layers

Pooling layers are usually used following convolutional layers, it is a process to further reduce the complexity as well as improve robustness of the network. Here the term "robustness" refers to network's reduced sensitivity to spatial variations of input. The most common types of pooling layers are *max pooling*, and *average pooling*.

To be more specific, the output feature map from a convolutional layer is processed by a subsampling, which divides a feature map into over-lapping or non-overlapping $k \times k$ subsets and produce *one* output from each subset by selecting the maximum of the subset or taking the average over the subset. The amount of spatial shift from one subset to the next subset is called *stride length*. In any event, a pooling layer yields an output whose size is considerably smaller than its input in each dimension while the output is expected to catch important features of the input. We reiterate that by subsampling together with the local operation of maximizing or averaging, local features are preserved however their exact locations are less critical and hence providing improved robustness of the network. As an example [12], Figure 11 shows a pooling layer with $k = 2$ and stride length of 2 for an input feature map of size 24×24 and the result is a 12×12 feature map.

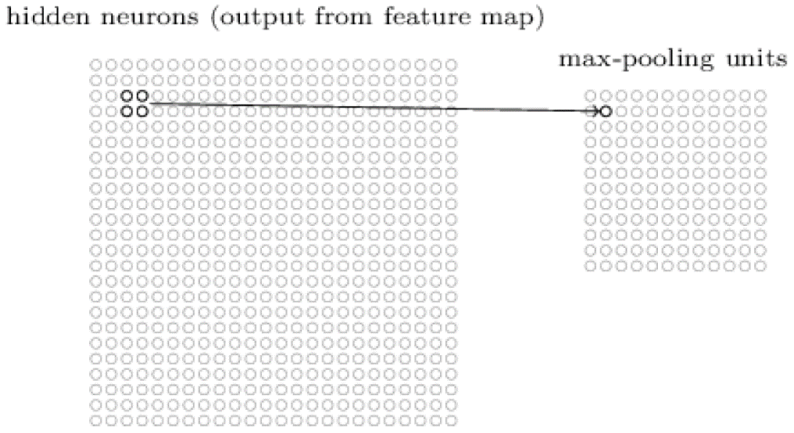

Figure 11. 2×2 max pooling as applied to a 24×24 feature map, yielding a 12×12 feature map.

In a bigger picture from the start, Figure 12 shows the structure of a CNN for a handwritten digit from MNIST, where the first convolutional layer uses three 5×5 kernels to generate three feature maps, followed by a pooling layer with 2×2 subsampling and stride length of 2 and max pooling to generate three feature maps with size reduced by half.

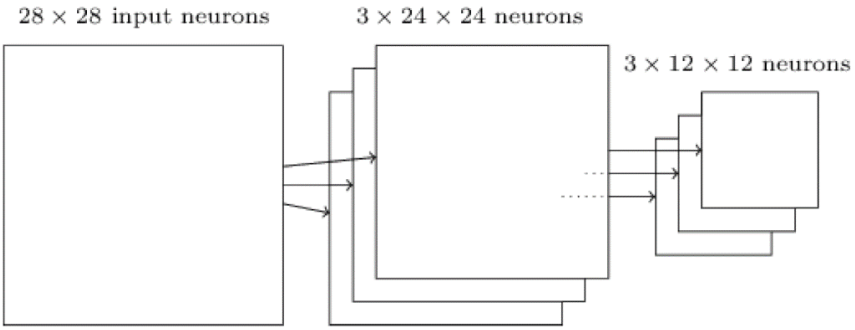

Figure 12. 3 pooling layer results.

2.2.5 Fully-Connected Layer and Output Layer

As shown in Figure 13, the second last layer of a CNN (for the purpose of handwritten digits recognition) is a fully-connected layer where every output neuron from a pooling layer is connected to every one of the 10 output neurons.

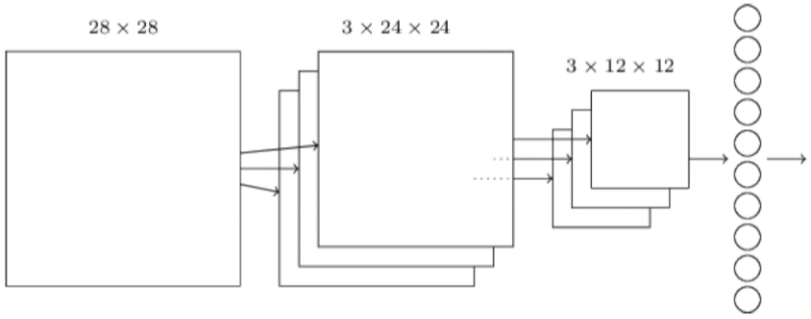

Figure 13. Convolutional neural network.

The weights and bias used in this layer are directly responsible for predicting the class to which the input belongs. Below we explain how these parameters are learned and then used in a set of decision functions that collectively predict the input digit. Let $\boldsymbol{x}$ be the vector collecting the outputs of the pooling layers and $\{\boldsymbol{w}_i, b_i\}$ be the weight and bias that connects $\boldsymbol{x}$ to the ith output for $i = 0, 1, \ldots, 9$, and define

$$\hat{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}, \ \hat{\boldsymbol{w}}_i = \begin{bmatrix} \boldsymbol{w}_i \\ b_i \end{bmatrix}, \text{ and } \hat{\boldsymbol{W}} = \begin{bmatrix} \hat{\boldsymbol{w}}_0 & \hat{\boldsymbol{w}}_1 & \cdots & \hat{\boldsymbol{w}}_9 \end{bmatrix}$$

It is important to note that although vector $\boldsymbol{x}$ itself is *not* the input digit, nevertheless $\boldsymbol{x}$ is produced exclusively by the digit and hence $\boldsymbol{x}$ can be regarded as a representative of the digit. With this in mind, we model the probabilities of an observed data vector $\boldsymbol{x}$ (hence the digit it represents) belonging to class C_j for $j = 0, 1, \ldots, 9$ by a *vector* logistic function

as

$$\begin{bmatrix} P(\hat{y}=0 \mid \boldsymbol{x}, \hat{\boldsymbol{W}}) \\ P(\hat{y}=1 \mid \boldsymbol{x}, \hat{\boldsymbol{W}}) \\ \vdots \\ P(\hat{y}=9 \mid \boldsymbol{x}, \hat{\boldsymbol{W}}) \end{bmatrix} = \frac{1}{\sum_{j=0}^{9} e^{\hat{\boldsymbol{w}}_j^T \hat{\boldsymbol{x}}}} \begin{bmatrix} e^{\hat{\boldsymbol{w}}_0^T \hat{\boldsymbol{x}}} \\ e^{\hat{\boldsymbol{w}}_1^T \hat{\boldsymbol{x}}} \\ \vdots \\ e^{\hat{\boldsymbol{w}}_9^T \hat{\boldsymbol{x}}} \end{bmatrix}$$

where $P(\hat{y} = j \mid \boldsymbol{x}, \hat{\boldsymbol{W}})$ denotes the conditional probability of data $\boldsymbol{x}$ belonging to class C_j given that sample $\boldsymbol{x}$ has been observed and $\hat{\boldsymbol{W}}$ is known and held fixed. We remark that the above model is a straightforward multi-class extension of the 2-class regression function.

Now suppose we are given a train data set with N samples which are exclusively represented by $\{ \boldsymbol{x}_n, n = 1, 2, \ldots, N\}$ and the labels associated with these samples are $\{y_n, n = 1, 2, \ldots, N\}$. By treating $\{ \boldsymbol{x}_n \}$ as i.i.d. random variables and given model parameter $\hat{\boldsymbol{W}}$, the probability of having observed the above data set is equal to $\prod_{n=1}^{N} P(y_n \mid \boldsymbol{x}_n, \hat{\boldsymbol{W}})$. where

$$P(y_n \mid \boldsymbol{x}_n, \hat{\boldsymbol{W}}) = \frac{e^{\hat{\boldsymbol{w}}_{y_n}^T \hat{\boldsymbol{x}}_n}}{\sum_{j=0}^{9} e^{\hat{\boldsymbol{w}}_j^T \hat{\boldsymbol{x}}_n}}$$

hence

$$\prod_{n=1}^{N} P(y_n \mid \boldsymbol{x}_n, \hat{\boldsymbol{W}}) = \prod_{n=1}^{N} \frac{e^{\hat{\boldsymbol{w}}_{y_n}^T \hat{\boldsymbol{x}}_n}}{\sum_{j=0}^{9} e^{\hat{\boldsymbol{w}}_j^T \hat{\boldsymbol{x}}_n}} \tag{2.2}$$

Expression (2.2) explicitly connects the conditional probability of having observed the data set to parameter $\hat{\boldsymbol{W}}$ and allows to optimize $\hat{\boldsymbol{W}}$ by maximizing (hence the name softmax regression) the conditional probability in (2.2) with respect to $\hat{\boldsymbol{W}}$. Since maximizing the conditional probability is equivalent to minimizing negative logarithm of the probability, the problem at hand can be formulated as the unconstrained problem

$$\underset{\hat{\boldsymbol{W}}}{\text{minimize}} \quad f(\hat{\boldsymbol{W}}) = -\frac{1}{N} \sum_{n=1}^{N} \ln \left(\frac{e^{\hat{\boldsymbol{w}}_{y_n}^T \hat{\boldsymbol{x}}_n}}{\sum_{j=0}^{9} e^{\hat{\boldsymbol{w}}_j^T \hat{\boldsymbol{x}}_n}} \right) \tag{2.3}$$

The minimizer of problem (2.3), $\hat{\boldsymbol{W}}^*$, can now be used to build a classifier for a test vector $\boldsymbol{x}$ which will be classified to class C_{j^*} if

$$j^* = \arg\left(\max_{0 \le j \le 9} e^{\hat{\boldsymbol{w}}_j^{*T} \hat{\boldsymbol{x}}} \right) \tag{2.4a}$$

which is equivalent to

$$j^* = \arg\left(\max_{0 \le j \le 9} \hat{\boldsymbol{w}}_j^{*T} \hat{\boldsymbol{x}} \right) \tag{2.4b}$$

where $\hat{\boldsymbol{w}}_j^*$ denotes the *j*th column of $\hat{\boldsymbol{W}}^*$.

In the next chapter, we will present an experimental study that evaluates a number of representative neural networks as applied to the handwritten digits recognition (HWDR) problem. In addition, the performance evaluation will be extended to several well-known solvers of the HWDR problem that are not based on neural nets.

Chapter 3

Performance Evaluation and Comparisons

This chapter is devoted to an experimental study of CNNs as applied to the HWDR problem. A key software supporting the study is the MATLAB Deep Learning Toolbox which has been available since early 2016. In effect, a large part of the chapter involves MATLAB functions that are available only from the above toolbox. The reader will be able to run the code discussed in this chapter smoothly as long as version R2019a (or later) of MATLAB is installed.

The objectives of this chapter are two-fold: (i) to implement and evaluate several neural networks for HWDR that includes a simple multi-neurons but non-convolutional neural network, a CNN called LeNet5 which has been well known for its excellent performance for HWDR, and several variants of LeNet5 for performance improvement; and (ii) to extend performance comparisons to several techniques that are not based on neural networks but are well known as solvers for the HWDR problem.

3.1 Data Preparation

For consistent performance evaluation, MATLAB version R2019a and the MNIST database are used throughout the simulations. Our computational platform is a Windows 10 PC with an Intel 6700k CPU, an Nvidia 1080 GPU, and 32 GB RAM.

MATLAB includes a Neural Network Toolbox for deep learning since 2016. The toolbox provides an effective framework for designing and implementing deep neural networks with many options for training algorithms and pre-trained models. The toolbox contains a variety of useful functions like **`nftool`** for function fitting, **`nprtool`** for pattern recognition, and **`nctool`** for data clustering, etc. Functions for implementing CNNs and long short-term memory (LSTM) networks for classification and regression for images, text data, and time series are also available.

For simplicity and clarity, in the rest of the chapter we shall take "function(s) or command(s)" to mean "MATLAB function(s) or command(s)", and the names of MATLAB functions, commands, and variables as well as MATLAB codes will be written in boldfaced **`Courier New`** font.

3.1.1 Loading the Data

Using functions **loadMNISTImages** and **loadMNISTLabels** at the site [24] or [25] directly, four data sets can be extracted from the database files with commands

```
Tr28  = loadMNISTImages('train-images.idx3-ubyte');
Ltr28 = loadMNISTLabels('train-labels.idx1-ubyte');
Te28  = loadMNISTImages('t10k-images.idx3-ubyte');
Lte28 = loadMNISTLabels('t10k-labels.idx1-ubyte');
```

where variables **Tr28** and **Ltr28** are two matrices of size 784×60000 and 60000×1, respectively, with each column of **Tr28** representing an image for a handwritten digit which is shaped to a column vector of length 784, and **Ltr28** being a column to represent the labels for the corresponding digits. To view the digits as images, it is required to reshape the columns of **Tr28** back to 28×28 matrices. Running the code below will display the first 100 digits from **Tr28**:

```
figure
for i = 1:100
      subplot(10,10,i)
      digit = reshape(Tr28(:,i),[28,28]);
      imshow(digit)
      title(num2str(labels(i)))
end
```

3.1.2 From MNIST Database to MATLAB Datastore

One of the key functions from the Deep Learning Toolbox is **trainedNet = trainNetwork(ds,layers,options)** which trains and returns a network **trainedNet** for a classification problem. The input **ds** is an **imageDatastore** with categorical labels or a **MiniBatchable Datastore** with responses; **layers** is an array of network layers or a **LayerGraph**; and **options** is a set of training options.

Alternatively, the function can also be used as **trainedNet = trainNetwork(X,Y, layers,options)** where the format for **X** depends on the input layer. For an image input layer, **X** is a numeric array of images arranged so that the first three dimensions are the width, height and channels, and the last dimension indexes the individual images. In a classification problem, **Y** specifies the labels for the images as a categorical vector. In a regression problem, **Y** contains the responses arranged as a matrix of size number of observations by number of responses, or a four dimensional numeric array, where the last dimension corresponds to the number of observations.

The third usage of the function is **`trainedNet = trainNetwork(tbl,layers,options)`** for networks with an image input layer, where **`tbl`** is a table containing predictors in the first column as either absolute or relative image paths or images. Responses must be in the second column as categorical labels for the images. In a regression problem, responses must be in the second column as either vectors or cell arrays containing 3-D arrays or in multiple columns as scalars. For networks with a sequence input layer, **`tbl`** is a table containing absolute or relative **`.mat`** file paths of predictors in the first column. For a sequence-to-label classification problem, the second column must be a categorical vector of labels. For a sequence-to-one regression problem, the second column must be a numeric array of responses or in multiple columns as scalars. For a sequence-to-sequence classification problem, the second column must be an absolute or relative file path to a **`.mat`** file with a categorical sequence. For a sequence-to-sequence regression problem, the second column must be an absolute or relative file path to a **`.mat`** file with a numeric response sequence.

To use the above function correctly, it is necessary to convert the raw MNIST dataset into appropriate formats.

3.1.2.1 *From MNIST database to image files*

In order to store image data (such as **`Tr28`**) into **`datastore`**, they need to be converted back to images (i.e., matrices rather than vectors), and this can be done using **`reshape`** or **`imwrite`**. The code below creates a main folder named **`tr`** and 10 separate subfolders within the main folder for 10 sets of MNIST digits according to their labels:

```
ltr = Ltr28';
len = length(ltr);
uni_ltr = unique(ltr);
cpath = pwd;
for i = 1:length(uni_ltr)
    label = num2str(uni_ltr(i));
    mkdir(fullfile(cpath,'tr',label));
end
```

Next, the input samples are reshaped into 28 × 28 images, and then stored in the respective subfolders in **`png`** format. The code below just does that:

```
count = 0;
cpath = pwd;
for n = 1:len
    count = count+1;
    digit = reshape(Tr28(:,n),[28 28]);
```

```
    label = num2str(ltr(n));
    count_str = num2str(count);
    fname = fullfile(cpath,'tr',label,[label '_' count_str '.png']);
    imwrite(digit,fname);
end
```

In effect, the code generates 10 subfolders, which are named from 0 to 9, and the training images are sorted into their respective subfolders. A similar piece of code was prepared to do the same for the testing data.

3.1.2.2 *From MNIST images to MATLAB datastore*

To create MATLAB datastore, it is necessary to get three data paths and set up some properties with commands:

```
cpath = pwd;
tr_path = fullfile(cpath,'tr',);
te_path = fullfile(cpath,'te',);
ds_path = fullfile(cpath);
verbose = true;
visualize = false;
```

Using function **`imageDatastore`**, the code below saves the training and testing data into respective **`datastore`**, and are named as **`trds`** and **`teds`**, respectively:

```
trds = imageDatastore(tr_path, 'IncludeSubfolders',true,...
'FileExtensions','.png','LabelSource','foldernames');
save(fullfile(ds_path,'trds.mat'),'trds');
teds = imageDatastore(te_path, 'IncludeSubfolders',true,...
'FileExtensions','.png','LabelSource','foldernames');
save(fullfile(ds_path,'teds.mat'),'teds');
```

3.2 Network #1: Fully-Connected One-Hidden Layer Network Without Convolution and Pooling

This section implements and evaluates a shallow neural network with one hidden layer where no convolutional and pooling operations are involved. We begin by reorganizing the data labels as row vectors for convenience of subsequent encoding.

```
ltr = Ltr28'; lte = Lte28';
```

At this point we need function **`dummyvar`** which returns a full set of dummy variables for each grouping variable. Note that **`dummyvar`** does not accept zero-valued entries, hence label 0 is converted into label 10 first:

```
ltr(ltr==0)=10; lte(lte==0)=10;
```

```
ltr = dummyvar(ltr); lte = dummyvar(lte);
```

To design a neural network, the training method must be specified. The toolbox provides many training functions for selection, such as **`trainbfg`** that implements the BFGS Quasi-Newton algorithm, **`traingd`** that implements the standard gradient descent algorithm, and **`traingdm`** that employs an accelerated gradient descent algorithm using momentum, etc. The network we implemented uses the scaled conjugate gradient algorithm and this is done by setting **`trainFcn = 'trainscg'`**. The network was evaluated by a total of five settings where it employs 10, 20, 30, 40, and 50 neurons in the hidden layer, respectively. For each setting, the entire training data was partitioned at random so that 80% of the available data are used for training while the rest 20% of the data are used for validation [26]. A cross-entropy loss function, which is equivalent to the softmax regression function in (2.3), is minimized in the training phase. The trained network was then applied to the test data set of 10,000 samples and the rate of success was calculated as

$$\text{rate of success} = 100 \times \left(\frac{\text{number of correctly classified digits}}{\text{number of digits tested}} \right) \%$$

The code listed below implements the training and evaluation of the network.

```
rand('state',state);
trainFcn = 'trainscg';
net.divideParam.trainRatio = 80/100;
net.divideParam.valRatio = 20/100;
net.divideParam.testRatio = 0/100;
net.performFcn = 'crossentropy';
net.plotFcns =
{'plotperform','plottrainstate','ploterrhist','plotconfusion','plotroc'
};
for i = 10:10:50
    t = cputime;
    net = patternnet(i,trainFcn);
    [net,~] = train(net,Tr28,ltr');
    time(i/10) = cputime - t;
    tt = cputime;
    tsty = net(Te28);
    tstt = cputime - tt;
    tind = vec2ind(lte');
    yind = vec2ind(tsty);
    percent = sum(tind == yind)/numel(tind);
    perc(i/10) = 100*percent;
end
```

We remark that the **`state`** in first line of the above code is an initial state which must be specified with an integer to run the code. The assignment of an initial sate ensures the code

to produce identical simulation results as long as the same initial state is used. Fig. 14 shows a training status window that pops up when the code is being executed.

Table 2 summarizes the recognition accuracy of the neural network using 20 different initial random states.

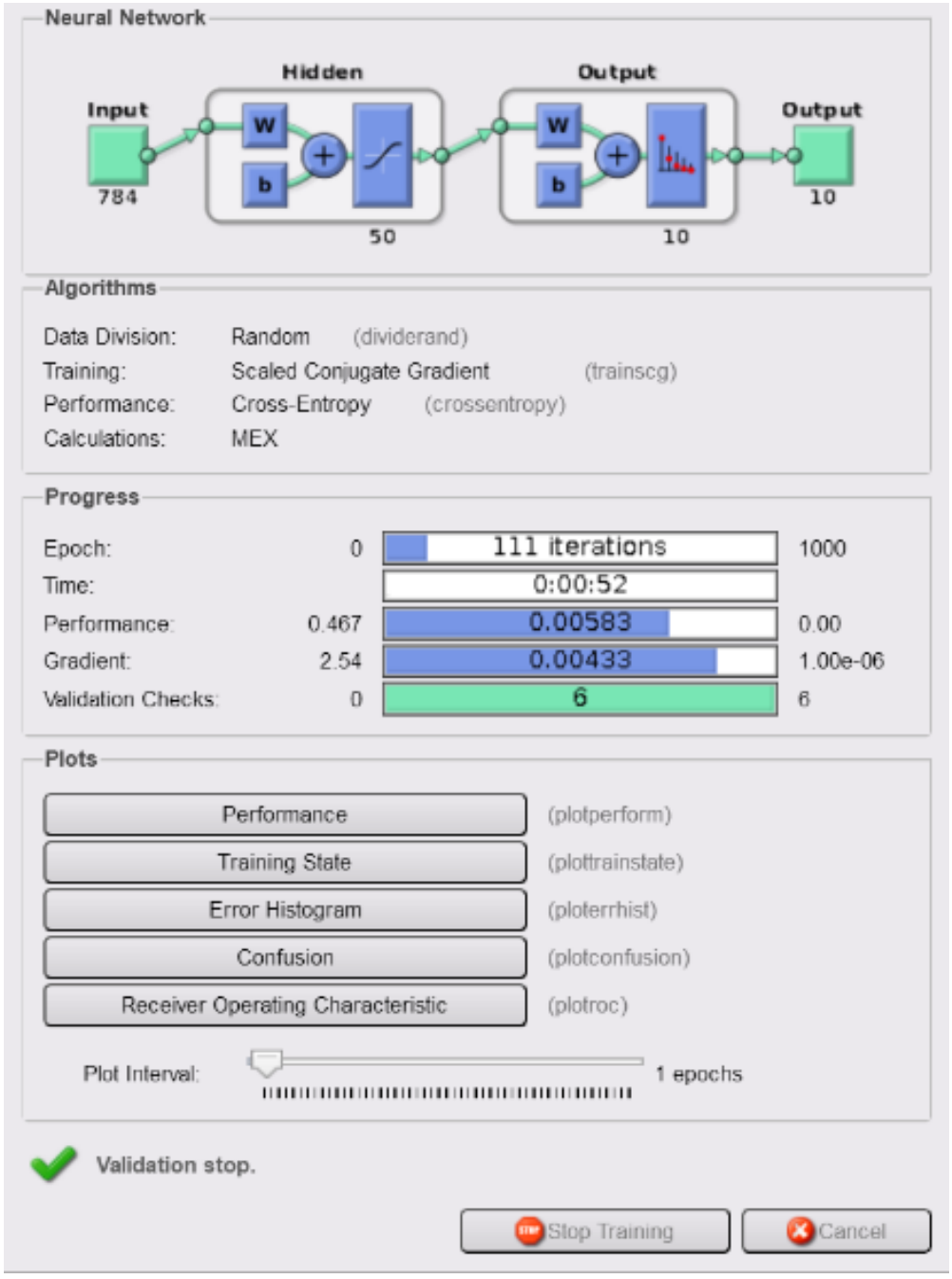

Figure 14. Training status of a fully-connected one-hidden-layer neural network.

Table 2. Rate of Success of the Neural Network.

Random State	Accuracy (%) with neurons in hidden layer varying from 10 to 50				
	10	20	30	40	50
1	92.13	93.82	94.35	94.85	95.60
2	87.60	93.63	94.95	95.23	95.22
3	92.28	93.39	94.96	94.80	95.29
4	92.04	94.01	94.61	95.32	**96.25**
5	91.15	93.94	94.92	95.26	95.60
6	92.17	93.51	94.55	95.32	95.49
7	92.15	92.99	94.05	95.09	95.72
8	91.71	94.40	94.78	95.72	95.49
9	90.60	93.33	94.88	94.84	95.12
10	90.73	93.91	94.52	94.89	95.44
11	92.22	92.87	94.54	95.52	96.10
12	91.99	93.84	94.84	95.18	96.07
13	90.71	93.42	94.55	95.14	96.23
14	92.55	93.66	94.43	94.91	95.65
15	91.62	93.42	94.28	94.76	95.50
16	91.74	94.34	94.81	95.41	95.71
17	92.06	93.34	94.86	95.52	95.63
18	91.81	92.55	94.76	94.94	95.43
19	92.29	93.74	94.31	95.15	95.74
20	91.92	93.88	94.79	94.83	95.75
Average	91.57	93.60	94.64	95.13	95.65

It is observed that in general the recognition accuracy increases with the number of neurons, and in average the net with 10, 20, 30, 40, and 50 neurons can achieve a rate of 91.57%, 93.60%, 94.64%, 95.13%, and 95.65%, respectively. The network achieved the best rate of 96.25% when it employed 50 neurons and initial random state was set to 4. Table 3 provides the training time that the network requires for various settings and initial

random states, while Table 4 shows the required testing time for the entire testing data set (of 10000 samples).

Table 3. Training Time of the Neural Network.

# times of training	Training time (in minutes) with neurons varying from 10 to 50				
	10	20	30	40	50
1	2.36	2.22	2.23	2.20	2.69
2	2.66	2.26	1.84	2.27	2.55
3	2.12	1.68	2.31	2.04	2.30
4	2.33	1.55	2.06	1.99	**2.94**
5	1.83	2.27	2.20	2.00	2.30
6	1.59	2.62	2.11	2.47	2.97
7	2.36	1.62	1.66	2.38	2.84
8	3.48	2.50	2.13	2.71	3.14
9	2.45	1.33	1.99	2.22	2.49
10	3.85	1.76	1.87	2.31	2.49
11	2.97	1.34	1.88	2.32	2.87
12	1.90	1.93	2.07	2.33	3.09
13	1.81	2.05	2.02	2.03	2.83
14	2.78	1.55	1.94	2.57	2.47
15	1.73	2.55	2.14	1.99	2.69
16	1.86	2.20	1.91	2.33	2.54
17	1.83	2.01	2.50	2.46	2.29
18	1.64	1.55	2.00	2.37	2.81
19	2.40	1.44	2.16	2.71	2.63
20	3.01	2.25	2.12	2.55	2.51
Average	2.35	1.93	2.06	2.31	2.67

From Table 4, we see that in average the network was able to recognize as many as 38,000 handwritten digits every second when the network uses 50 neurons in its hidden layer.

Table 4. Testing Time of the Neural Network.

# times of training	Testing time (in seconds) with neurons varying from 10 to 50				
	10	20	30	40	50
1	0.30	0.22	0.20	0.25	0.22
2	0.25	0.16	0.25	0.25	0.19
3	0.16	0.13	0.25	0.25	0.27
4	0.27	0.16	0.25	0.16	0.27
5	0.14	0.25	0.25	0.25	0.20
6	0.20	0.25	0.22	0.25	0.33
7	0.25	0.25	0.25	0.27	0.27
8	0.25	0.28	0.20	0.20	0.27
9	0.20	0.25	0.25	0.20	0.20
10	0.19	0.25	0.27	0.27	0.34
11	0.14	0.25	0.25	0.31	0.41
12	0.20	0.30	0.20	0.31	0.22
13	0.14	0.17	0.27	0.20	0.25
14	0.25	0.25	0.16	0.22	0.31
15	0.20	0.25	0.25	0.20	0.25
16	0.20	0.36	0.27	0.25	0.20
17	0.23	0.16	0.25	0.25	0.27
18	0.16	0.25	0.25	0.27	0.22
19	0.30	0.25	0.20	0.27	0.27
20	0.25	0.22	0.20	0.20	0.22
Average	**0.21**	**0.23**	**0.23**	**0.24**	**0.26**

Naturally one would be curious about what would happen when the number of neurons in the hidden layer continues to grow. It turns out that with 500 neurons the network achieves a 97.23% accuracy in 19.56 minutes, and with 1,000 neurons the accuracy reaches 97.38% in 41.82 minutes.

3.3 Convolutional Neural Network

The Deep Learning Toolbox provides many options for training a network. Options for traing algorithms include the stochastic gradient descent with momentum, root mean square propagation, and adaptive moment estimation. All these training algorithms adopts the same default initial weights, which is a Gaussian distribution with a mean of zero and a standard deviation of 0.01. The default initial bias is set to zero. However, if necessary these initial values can be reset manually through network setup.

For the sake of consistency, all CNNs in our simulations are trained employ stochastic gradient descent with momentum with default initial weights and bias values, and an initial learning rate is set to 0.01. Three choices of maximum epochs, namely 4 epochs, 10 epochs, and 30 epochs, are implemented for examining the training performance and efficiency, although with an increased number of epochs more stable results are expected. As can be seen from the code below, the training is executed in a single GPU and its progress is shown is a plot.

```
options = trainingOptions('sgdm', ...
    'MaxEpochs',epoch,...
    'InitialLearnRate',1e-2, ...
    'Shuffle','every-epoch',...
    'Verbose',false, ...
    'Plots','training-progress',...
    'ExecutionEnvironment','gpu');
```

To calculate the recognition accuracy, command **`classify`** is used as follows:

```
tep = classify(convnet,teds);
tev = teds.Labels;
acc = sum(tep == tev)/numel(tev);
fprintf('accuracy: %2.2f%%,error rate: %2.2f%%\n',acc*100,100-acc*100);
```

where **`convnet`** is the trained network, and **`teds`** is testing data in **`datastore`** format. The last two lines of the code compare the difference between the predicted testing data with their true labels and display the error rate.

3.3.1 Network #2: A Basic CNN with a Single Convolutional Layer

A basic CNN consists of an input layer, a convolutional layer, a normalization layer, an activation layer, a pooling layer, a fully-connected layer, and an output softmax layer to predict the label of the input. The code below implements such a basic network:

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,3)                 %C1
      batchNormalizationLayer
      reluLayer
    maxPooling2dLayer(2,'Stride',2)         %S2
    fullyConnectedLayer(10)                 %F3
    softmaxLayer
    classificationLayer];
```

To better illustrate the layers of the CNN, we execute function **`deepNetworkDesigner`** in the command window which produces a "workspace" as illustrated in Fig. 15 where the left column is **`layer library`**, providing available layers. On the right is **`properties`** bar where parameters values can be specified. Under **`properties`** bar is **`overview`** of the network. One can use **`ctrl`** and **`scroller`** to zoom out and in the details of the network as shown in Fig. 16. Once the layers are constructed, the network can be examined by clicking icon **`Analyze`**. If no error or warming showing, the network is ready to be exported to the workspace.

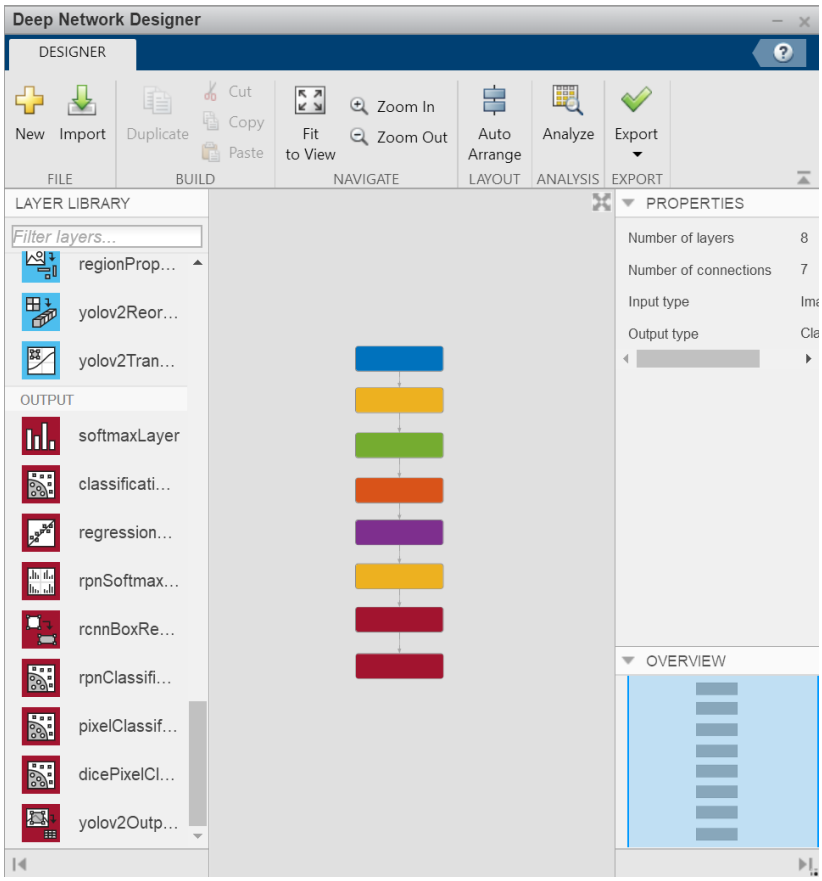

Figure 15. Deep network designer.

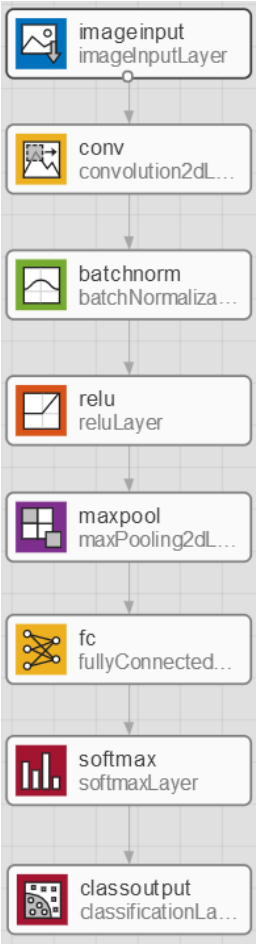

Figure 16. Basic CNN map.

In what follows, C_i represents a convolutional layer, B represents a batch normalization layer, A represents an activation layer, S_i represents a subsampling layer, and F_i represents a fully-connected layer, where i denotes the layer index. The batch normalization layer and the activation layer are usually not considered as CNN layers, so when counting the number of layers, they are not counted.

The input layer takes in an image of size of $28 \times 28 \times 1$, corresponding to the length, width color of the image. Since the MNIST digits are in grey scale, the dimension of color is set to 1 whereas for RGB images the dimension of color will be 3. In this basic network setting, only one convolutional layer (C_1) is used, the size of the local receptive field (which is the same that of the convolutional kernel) is 5×5 , and the layer produces three feature maps, each is followed by a batch normalization layer (B) to normalize the output of the convolutional layer. Since the deep learning toolbox does not have the built-in sigmoid layer, a ReLU layer (A) is used. The next layer is a max pooling layer (S_2) to perform $2 \times$

2 down-sampling and a step size of 2. The reduced feature maps are then fully connected to a 10-neuron layer (F_3) which is followed by a softmax layer and an output (classification) layer.

To test the network's performance, the size of local receptive field (LRF) are set to 3×3, 5×5, and 7×7 with appropriate padding sizes, in each case the convolutional layer generates 3, 6, or 8 feature maps (FM). Once the training of the network starts, a training progress plot pops up (see e.g. Fig. 17), this graphics window is defined in training options. The training progress also displays elapsed time, number of iterations, iterations per epoch, maximum iterations, hardware resource, and learning rate etc. as can be seen on the right-hand side of the plot.

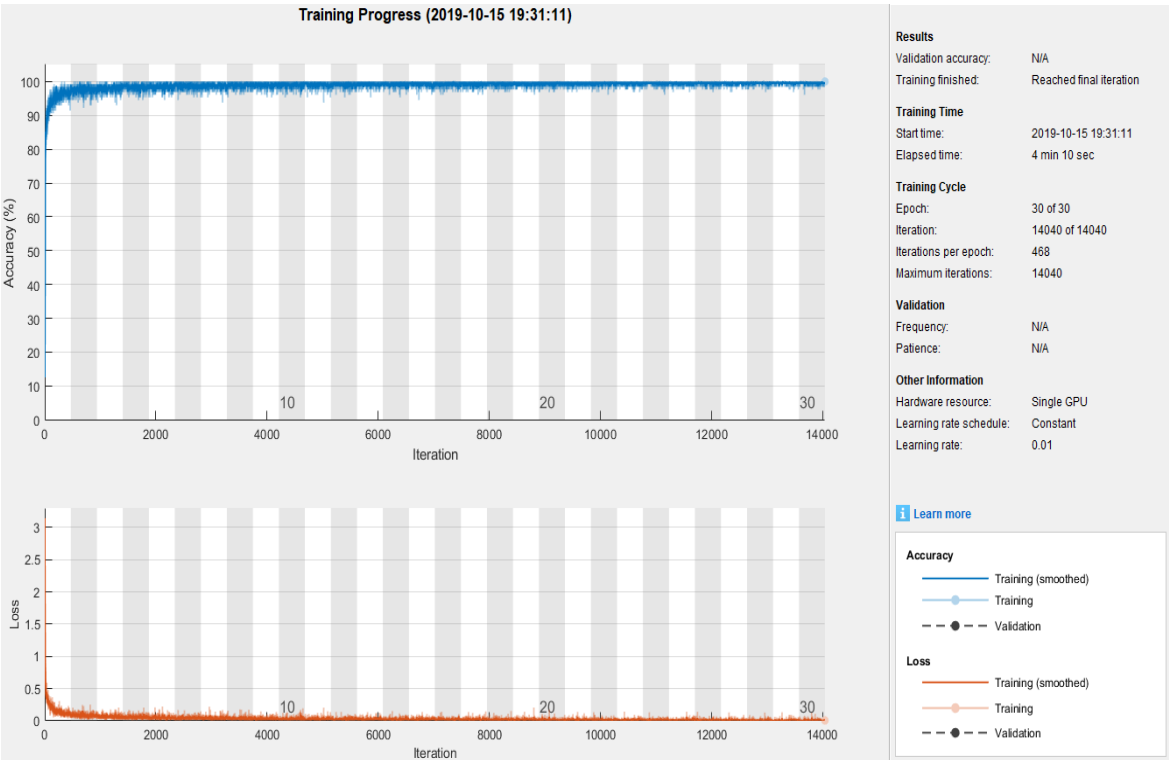

Figure 17. Training progress with 7 by 7 kernel and 8 feature maps.

The training results of this 3-layer CNN without padding and stride 2 are shown in Table 5, while the training results of the same CNN with padding size 1 and step size 1 are shown in Table 6.

Table 5. Performance of the Basic CNN with no Padding and Stride 2

LRF	FM	Accuracy (%)			Time					
					Training (minutes)			Testing (Seconds)		
Epoch		4	10	30	4	10	30	4	10	30
3 × 3	3	97.25	97.28	97.24	1.68	3.54	9.03	2.16	1.91	2.00
3 × 3	6	97.79	97.74	97.73	1.70	3.61	9.19	2.22	1.92	2.16
3 × 3	8	97.72	97.86	98.04	1.71	3.61	9.51	2.13	2.05	2.19
5 × 5	3	97.86	97.82	97.80	1.74	3.46	9.55	2.05	1.83	2.09
5 × 5	6	97.98	98.06	98.05	1.75	3.57	9.49	2.09	2.13	2.16
5 × 5	8	98.12	98.39	98.35	1.84	3.79	9.68	1.98	2.05	2.05
7 × 7	3	97.42	97.60	97.79	1.69	3.69	9.54	2.06	1.91	2.00
7 × 7	6	98.29	98.35	98.52	1.66	3.78	9.37	2.09	2.33	3.08
7 × 7	8	98.28	98.56	98.55	1.95	3.98	9.53	1.91	2.19	2.02

Table 6. Performance of the Basic CNN with Padding Size of 1 and Stride 1

LRF	FM	Accuracy (%)			Time					
					Training (minutes)			Testing (Seconds)		
Epoch		4	10	30	4	10	30	4	10	30
3 × 3	3	97.66	97.03	97.62	1.72	3.66	10.21	2.16	1.84	2.03
3 × 3	6	97.72	98.08	98.01	1.75	4.07	10.79	2.06	2.52	2.30
3 × 3	8	98.06	98.21	98.10	1.76	4.07	10.83	2.39	2.16	2.27
5 × 5	3	97.77	98.05	97.80	2.23	4.00	10.29	2.72	1.86	2.25
5 × 5	6	98.33	98.33	98.25	1.79	4.05	10.77	3.36	2.11	2.08
5 × 5	8	**98.49**	98.40	98.39	1.82	4.03	11.10	2.03	2.86	2.22
7 × 7	3	97.69	97.97	98.02	1.78	3.72	10.40	2.06	2.31	2.14
7 × 7	6	98.37	**98.55**	98.58	1.73	4.02	10.63	2.16	2.39	1.94
7 × 7	8	98.35	98.52	**98.61**	1.74	4.07	11.08	2.28	2.14	2.11

From Table 5 and Table 6, we see that the CNN offers its best performance with 98.61% prediction accuracy in 11.08 minutes when the LRF is set to 7 × 7, FM is set to 8, padding size is set to 1, and stride length is set to 1.

3.3.2 Network # 3: CNNs with Multiple Convolutional Layers

To improve the basic CNN, we consider adding a second convolutional layer (C_3) or even a third convolutional layer (C_5) as well as the respective pooling layers to the network. The same padding size and stride length as in the basic CNN are used, namely, the padding size is set to 1 and the stride length is set to 2.

The code shown below implements a CNN with two convolutional layers:

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(lrf1,fm1,'Padding',1)     %C1
      batchNormalizationLayer
      reluLayer
    maxPooling2dLayer(2,'Stride',s)              %S2
    convolution2dLayer(lrf2,fm2,'Padding',1)     %C3
      batchNormalizationLayer
      reluLayer
    fullyConnectedLayer(10)                      %F4
    softmaxLayer
    classificationLayer];
```

And the next code implements a CNN with three convolutional layers:

```
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(lrf1,fm1,'Padding',1)     %C1
      batchNormalizationLayer
      reluLayer
    maxPooling2dLayer(2,'Stride',s)              %S2
    convolution2dLayer(lrf2,fm2,'Padding',1)     %C3
      batchNormalizationLayer
      reluLayer
    maxPooling2dLayer(2,'Stride',s)              %S4
    convolution2dLayer(lrf3,fm3,'Padding',1)     %C5
      batchNormalizationLayer
      reluLayer
    fullyConnectedLayer(10)                      %F6
    softmaxLayer
    classificationLayer];
```

where `lrf1,lrf2,lrf3` are the 1st, 2nd, and 3rd local receptive field, `fm1,fm2,fm3` are the 1st, 2nd, and 3rd feature map, and `s` denotes stride length. The performance of the CNNs with two and three convolutional layers are shown in Tables 7 and 8, respectively.

Table 7. Performance of the CNN with Two Conv. Layers

LRF		FM		Accuracy (%)			Time					
1	2	1	2				Training (minutes)			Testing (Seconds)		
Epoch				4	10	30	4	10	30	4	10	30
3	3	3	3	97.56	97.65	97.99	1.69	3.62	9.99	3.13	2.17	2.91
3	3	3	9	98.60	98.45	98.08	1.67	3.58	10.03	2.17	2.23	2.28
3	7	3	3	98.11	98.25	98.20	1.88	3.81	9.65	2.05	2.20	2.58
3	7	3	9	98.63	98.65	98.86	1.92	3.80	9.58	2.11	2.42	3.45
3	3	9	3	98.26	98.28	98.16	1.93	3.81	9.68	2.00	2.41	2.13
3	3	9	9	98.39	98.62	98.78	1.94	3.84	9.80	2.11	2.52	2.41
3	7	9	3	98.75	98.54	98.57	1.95	3.77	9.88	2.20	2.33	2.36
3	7	9	9	98.70	98.98	99.17	1.96	3.89	9.99	2.00	2.41	2.30
5	3	3	3	97.92	97.91	98.24	1.90	3.78	9.76	2.20	2.17	2.50
5	3	3	9	98.49	98.43	98.55	1.88	3.96	9.82	2.19	2.08	3.14
5	7	3	3	97.94	98.20	98.29	1.82	3.90	9.73	2.20	2.34	2.47
5	7	3	9	98.62	98.66	98.87	1.87	3.96	9.81	1.97	2.09	2.48
5	3	9	3	98.08	98.58	98.31	1.89	4.01	9.91	2.00	2.27	2.28
5	3	9	9	98.53	98.86	99.07	1.88	4.10	9.92	2.09	2.61	3.66
5	7	9	3	98.30	98.48	98.69	1.91	4.15	9.98	2.33	2.34	2.63
5	7	9	9	**98.87**	**99.15**	**99.16**	1.91	4.15	10.06	2.14	2.45	2.33
7	3	3	3	98.05	98.17	97.88	1.86	4.06	9.94	2.31	2.53	3.11
7	3	3	9	98.43	98.84	98.79	1.85	4.08	10.00	2.47	2.34	2.73
7	7	3	3	97.90	98.15	98.42	1.85	4.04	10.00	2.83	2.56	2.67
7	7	3	9	98.73	98.80	98.96	1.87	4.10	10.04	2.25	2.20	3.13
7	3	9	3	98.39	98.48	98.37	1.91	4.11	10.17	2.20	2.25	2.69
7	3	9	9	98.97	98.96	98.98	1.87	4.01	10.19	2.03	2.89	2.94
7	7	9	3	98.20	98.81	98.65	1.88	4.18	10.35	2.48	2.58	3.05
7	7	9	9	98.81	98.94	99.13	1.84	4.13	10.21	2.73	2.38	3.17

Table 8. Performance of the CNN with Three Conv. Layers

LRF			FM			Accuracy (%)			Time					
1	2	3	1	2	3				Training (minutes)			Testing (seconds)		
Epoch						4	10	30	4	10	30	4	10	30
7	5	5	9	9	9	98.69	99.15	99.14	2.19	4.19	11.14	2.75	1.84	1.86
7	5	5	9	9	18	99.05	99.15	99.13	2.03	4.00	10.78	2.11	1.98	2.33
7	5	5	9	9	36	98.80	99.19	99.12	2.04	3.99	10.76	2.05	1.91	1.97
7	5	7	9	9	9	98.87	99.17	99.27	2.03	3.94	10.45	2.39	2.05	1.91
7	5	7	9	9	18	98.87	99.13	99.15	2.00	3.95	10.40	2.72	2.06	2.00
7	5	7	9	9	36	99.07	**99.31**	99.32	1.97	4.03	10.53	2.16	2.00	2.66
7	5	5	9	18	9	99.08	98.90	99.26	1.99	3.90	10.44	2.88	2.06	1.94
7	5	5	9	18	18	99.11	99.24	99.28	2.05	3.92	10.45	2.16	2.73	2.86
7	5	5	9	18	36	99.09	99.25	99.35	2.09	3.92	10.62	2.13	1.94	2.11
7	5	7	9	18	9	98.89	99.14	99.16	2.08	3.96	10.62	1.92	2.11	1.97
7	5	7	9	18	18	99.20	99.15	99.29	2.18	3.99	10.67	2.11	2.08	1.94
7	5	7	9	18	36	**99.27**	99.30	99.32	2.21	4.10	10.82	2.19	2.03	2.19
7	7	5	9	9	9	98.89	99.06	99.19	2.10	4.08	10.55	2.86	1.88	1.92
7	7	5	9	9	18	99.00	99.26	99.18	2.04	4.29	10.56	1.92	2.05	2.00
7	7	5	9	9	36	98.90	99.18	99.32	2.09	4.25	10.58	2.05	1.94	2.13
7	7	7	9	9	9	98.87	99.08	99.29	2.20	4.18	10.57	2.16	2.09	2.13
7	7	7	9	9	18	99.18	99.30	99.22	2.17	4.28	10.67	2.00	1.97	1.98
7	7	7	9	9	36	98.92	99.29	99.34	2.15	4.37	10.64	2.14	2.14	2.72
7	7	5	9	18	9	99.00	99.14	99.20	2.14	4.14	10.70	2.52	1.97	2.01
7	7	5	9	18	18	99.08	98.91	99.29	2.09	3.95	10.71	2.06	2.08	2.77
7	7	5	9	18	36	99.07	99.29	99.27	2.19	4.11	10.76	2.05	2.05	2.08
7	7	7	9	18	9	99.03	99.21	99.27	2.14	4.34	10.73	1.95	2.39	1.96
7	7	7	9	18	18	99.06	99.31	99.37	2.15	4.22	10.86	2.03	2.03	2.18
7	7	7	9	18	36	99.17	99.21	**99.43**	2.18	4.09	11.07	2.11	2.31	2.25

From Tables 7 and 8, it is observed that the CNN achieves its best performance with 99.43% prediction accuracy in 11.07 minutes when it employs three convolutional layers, each with 7×7 LRF and 9 FMs in *C1*, 18 FMs in *C2*, and 36 FMs in *C3*. We remark that the prediction accuracy achieved is a bit higher than the original LeNet-5 which offers a 99.05% accuracy. According to LeCun, actually LeNet-5 was able to achieve a 99.20% accuracy. Even then, the CNN constructed above does the job slightly better. In the next section, we implement the famous LeNet-5 and present a variant of it for performance improvement.

3.3.3 Network # 4: LeNet-5 and a Modified Version

Arguably the most original and well-known CNN architecture, LeNet-5 is a CNN made up with 7 layers that was specifically designed for handwritten digit and other image recognition problems [21]. Before getting into C_1, the input image is normalized. The first convolutional layer, C_1, contains 6 FMs, each is generated with a 5×5 local receptive field, and after that each FM is 2×2 subsampled and activated. Since the sigmoid function is not available from the toolbox, **`tanh`** function is used instead as it is the closest activation function to sigmoid in the toolbox. As a side note, it is known that multiple-layer CNNs with sigmoid activations generically suffer from "*the vanishing gradient problem*" meaning that the gradient components at the layers near the input tend to vanish that in turn leads to slow learning. Considering this matter, later on our simulations will cover CNNs using ReLU activations. Like LeNet-5, after C_1, 2×2 sub-sampling is applied and 2×2 average pooling is followed. The structures of C_3 and S_4 are similar to those of C_1 and S_2 except that in C_3 16 FMs are generated. In C_5, 120 FMs are generated and all of them are connected to a 84-neuron layer F_6, which in turn is fully connected to a 10-output layer F_7. The complete architecture of LeNet-5 is shown in Fig. 18 [21].

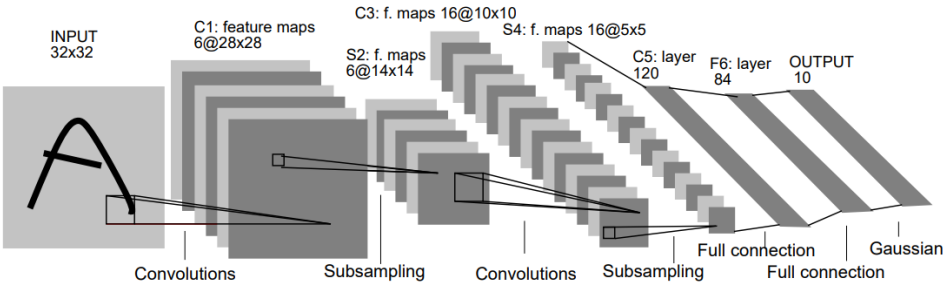

Figure 18. Architecture of LeNet-5.

The code below implements LeNet-5.

```
layers = [
    imageInputLayer([28 28 1])
      batchNormalizationLayer
    convolution2dLayer(5,6,'Padding','same')          %C1
      tanhLayer
    averagePooling2dLayer(2,'Stride',2)               %S2
    convolution2dLayer(5,16,'Padding',0)              %C3
      tanhLayer
    averagePooling2dLayer(2,'Stride',2)               %S4
    convolution2dLayer(5,120,'Padding',0)             %C5
      tanhLayer
    fullyConnectedLayer(84)                           %F6
    fullyConnectedLayer(10)                           %F7
    softmaxLayer
    classificationLayer];
```

As soon as the training starts, the training progress window pops up as shown in Fig. 19.

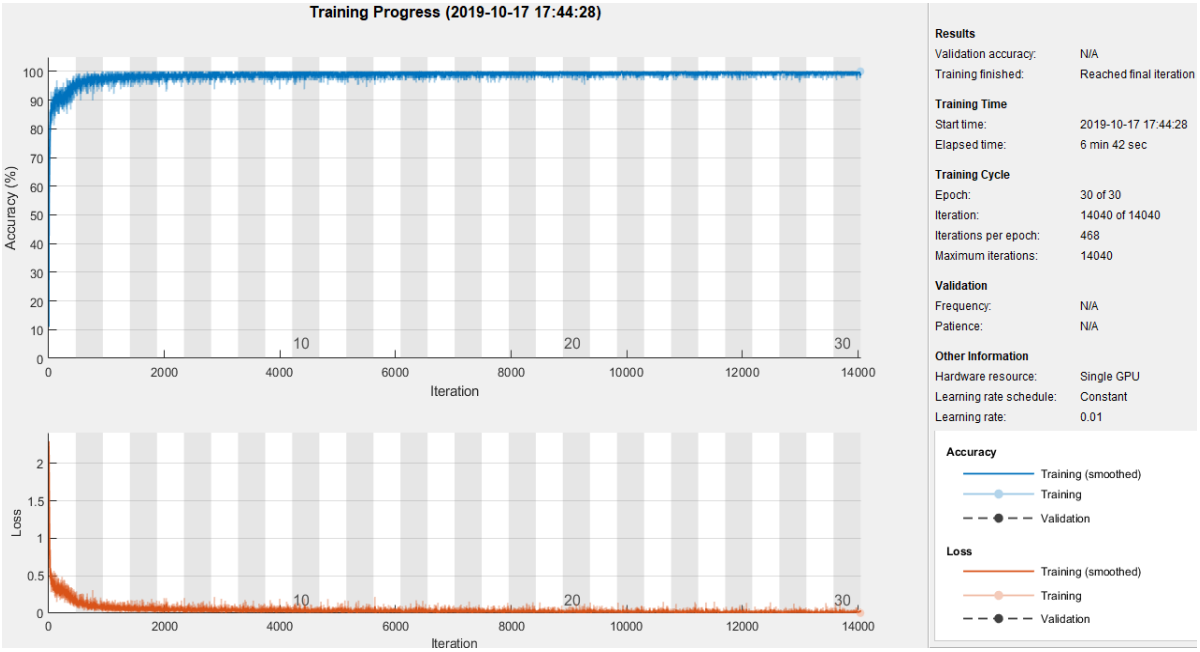

Figure 19. LeNet-5 training progress.

According to [21], the original LeNet-5 is capable of achieving an accuracy of 99.05%, whereas the above code, which employs **`tanh`** for activation, achieves a slightly better accuracy of 99.08% in 6.67 minutes.

We now present a modified version of LeNet-5 architecture that is shown to provide improved performance in terms of prediction rate. Instead of normalizing the input images, here the normalization is applied after each convolutional layer to reduce the internal covariate shift and accelerate the training.

A significant modification made in the CNN is the use of *rectified linear unit* (ReLU) instead of **`sigmoid`** and **`tanh`** for activation function, the reader is referred to Section 2.2.3 where ReLU was compared with several popular activation functions. Considering the fact that although the original LeNet-5 employs 5 × 5 LRF, the CNN described in Sec. 3.3.2 with 7 × 7 local receptive field in all C_1, C_3, and C_5 performs best (achieving an accuracy of 99.43%, see Table 8), it was decided to employ 7 × 7 LRF in all three convolutional layers. With this, the modified network has been pretty much defined except that the numbers of FMs in the convolutional layers remain to be specified. In our simulation studies the performance of the modified LeNet-5 with a variety of choices of the FM numbers has been examined and the results are shown in Table 9 when average pooling was used and Table 10 when max pooling was employed.

Table 9. Performance of the Modified LeNet-5 Using Average Pooling

FM			Accuracy (%)			Time					
1	2	3				Training (minutes)			Testing (seconds)		
Epoch			4	10	30	4	10	30	4	10	30
9	18	72	99.15	99.29	99.35	2.45	4.65	12.97	3.03	2.83	2.94
9	18	108	99.08	99.00	99.33	2.19	4.70	13.15	3.03	2.81	3.02
9	18	144	99.01	99.03	99.32	2.12	4.72	14.04	3.23	3.00	2.53
9	18	180	98.75	99.12	99.32	2.11	4.84	13.60	2.95	3.05	2.45
9	36	72	99.08	**99.40**	99.43	1.95	4.97	12.61	2.75	3.00	2.33
9	36	108	99.07	99.20	99.38	1.97	5.02	13.12	3.63	2.88	2.78
9	36	144	99.01	99.23	99.35	2.04	5.15	14.34	3.28	3.19	3.23
9	36	180	98.64	98.93	99.23	2.08	5.25	14.49	3.19	3.03	2.58
9	54	72	99.15	99.32	99.33	1.99	5.34	13.37	3.67	3.17	2.56
9	54	108	99.08	99.17	99.28	2.09	5.50	13.53	2.72	3.14	2.52
9	54	144	98.93	99.09	99.37	2.08	5.66	13.69	3.16	3.08	2.73
9	54	180	99.03	98.98	99.26	2.15	5.77	13.99	3.00	3.13	3.03
9	72	72	99.04	99.17	99.45	2.09	6.20	13.14	2.92	3.09	2.11
9	72	108	98.92	99.19	99.29	2.12	6.33	12.98	3.06	3.03	2.66
9	72	144	99.04	99.14	99.28	2.22	6.55	13.69	2.89	3.16	3.10

9	72	180	98.69	99.21	99.08	2.22	6.48	13.63	3.13	3.44	2.86
18	18	72	**99.16**	99.32	99.35	2.03	6.69	13.09	2.84	2.53	2.70
18	18	108	99.03	99.06	99.21	2.04	6.88	12.98	2.94	3.03	2.98
18	18	144	99.05	99.23	99.38	2.12	7.02	13.89	2.92	3.30	3.20
18	18	180	98.92	99.14	99.22	2.12	7.27	13.61	2.80	3.36	3.05
18	36	72	99.14	99.37	**99.45**	2.07	7.54	13.51	3.03	3.45	3.19
18	36	108	99.09	99.18	99.27	2.17	7.59	13.74	3.00	3.53	3.13
18	36	144	99.03	99.16	99.42	2.15	7.71	14.13	3.59	3.31	3.19
18	36	180	99.03	99.15	99.29	2.19	7.72	14.48	3.09	2.42	2.91
18	54	72	99.11	99.36	99.44	2.14	7.83	13.67	3.22	3.08	2.50
18	54	108	98.96	98.94	99.36	2.15	7.98	13.86	2.92	3.06	2.53
18	54	144	99.11	99.07	99.35	2.22	8.12	13.89	3.09	2.14	2.59
18	54	180	99.03	99.02	99.26	2.28	8.25	14.33	3.23	2.59	2.61
18	72	72	99.09	99.30	99.43	2.27	6.67	14.15	2.86	2.25	2.50
18	72	108	99.08	99.24	99.38	2.33	6.81	14.47	2.16	2.81	2.41
18	72	144	98.93	99.10	99.43	2.41	7.15	15.32	3.08	2.30	2.81
18	72	180	99.10	99.13	99.12	2.41	7.08	15.78	3.06	2.42	2.56

From Table 9, we see that the modified LeNet-5 achieves a prediction accuracy of 99.45% when 18, 36, and 72 FMs are used in C_1, C_3, and C_5 respectively, a 0.4% gain over the original LeNet-5. It should be noted that the performance gain was achieved at the cost of longer training time: it took 13.51 minutes (versus 6.67 minutes for the original LeNet-5) to complete. The training time can be reduced by using max pooling rather than average pooling.

Table 10. Performance of the Modified LeNet-5 Using Max Pooling

FM			Accuracy (%)			Time					
1	2	3				Training (minutes)			Testing (seconds)		
Epoch			4	10	30	4	10	30	4	10	30
9	18	72	98.96	99.37	99.36	1.93	1.93	4.17	2.36	2.56	2.52
9	18	108	98.86	99.23	99.24	1.94	4.19	11.25	2.39	2.38	2.28

9	18	144	98.98	99.21	99.33	1.95	4.38	11.65	2.38	2.89	2.52
9	18	180	98.99	99.12	99.16	1.99	4.33	11.84	2.77	2.88	2.13
9	36	72	99.21	99.20	99.40	2.05	4.01	11.30	3.25	3.55	2.88
9	36	108	99.20	99.19	99.46	2.11	4.08	11.64	2.48	2.64	2.39
9	36	144	99.04	99.36	99.38	2.14	4.34	11.84	3.69	3.08	2.13
9	36	180	98.81	99.04	99.36	2.14	4.51	12.30	2.55	2.75	3.16
9	54	72	99.05	99.33	99.34	2.08	4.22	11.27	2.17	2.77	2.49
9	54	108	99.10	99.13	99.49	2.18	4.19	11.69	2.42	2.20	2.63
9	54	144	98.97	99.22	99.33	2.21	4.44	12.04	2.52	2.05	2.50
9	54	180	98.85	99.27	99.21	2.21	4.45	12.33	2.78	2.05	2.66
9	72	72	99.23	99.38	**99.51**	2.04	4.39	11.75	2.95	2.78	2.99
9	72	108	98.92	99.31	99.35	2.08	4.44	12.44	2.69	2.72	2.27
9	72	144	99.20	99.27	99.43	2.22	4.68	12.92	3.47	3.42	2.49
9	72	180	98.77	99.18	99.21	2.21	4.55	12.82	2.75	2.17	3.09
18	18	72	99.17	99.38	99.31	1.93	4.01	11.51	3.23	2.80	2.43
18	18	108	98.95	99.18	99.37	2.01	4.02	11.78	2.47	2.39	2.31
18	18	144	98.87	99.17	99.40	2.10	4.10	12.19	2.64	2.69	2.96
18	18	180	99.16	99.31	99.11	2.11	4.16	12.17	3.06	3.30	2.19
18	36	72	**99.29**	99.30	99.31	2.02	4.00	11.57	2.42	2.45	2.55
18	36	108	99.24	**99.41**	99.34	2.05	4.10	11.80	2.95	2.55	2.40
18	36	144	99.28	99.24	99.39	2.09	4.35	12.08	3.13	2.84	2.83
18	36	180	99.03	98.99	99.35	2.15	4.40	12.51	3.20	3.11	2.88
18	54	72	98.98	99.36	99.45	1.99	4.11	11.76	2.52	2.63	2.93
18	54	108	99.07	99.33	99.45	2.06	4.44	12.10	2.77	3.20	3.12
18	54	144	99.15	99.21	99.31	2.11	4.50	12.52	2.55	2.39	3.12
18	54	180	98.92	99.26	99.26	2.14	4.72	12.86	2.94	3.31	2.49
18	72	72	99.11	99.40	**99.51**	2.11	4.52	12.88	2.61	2.84	2.11
18	72	108	99.14	99.14	99.49	2.14	4.47	13.16	2.45	2.41	2.63
18	72	144	99.16	99.16	99.34	2.28	4.80	13.65	3.19	2.63	2.91
18	72	180	99.07	99.14	99.39	2.33	4.69	14.13	3.02	3.17	2.97

From Table 10, we observe that the modified LeNet-5 achieves a 99.51% prediction accuracy when 9, 72, 72 FMs are used in C_1, C_3, and C_5, respectively, and the training took 11.75 minutes to complete. The highest rate occurs again when 18, 72, 72 FMs are used, but the training took a bit longer to finish. Based on these figures, max pooling appears to be a good choice for the HWDR problem using CNNs.

3.4 Comparisons with Other Techniques

3.4.1 Histogram of Oriented Gradients

Support vector machine (SVM) and principal component analysis (PCA) algorithms are widely used in classification problems and often demonstrate satisfactory performance. In this section, the CNNs constructed earlier are compared with HWDR solvers based on SVM and PCA.

Unlike deep neural networks such as CNNs that *learn* to extract features of an input, SVM and PCA are algorithms whose performance for a particular data set is largely dependent upon how effectively a pre-processing step, known as feature extraction, is done. Proposed in [27], the histogram of oriented gradients (HOG) is a feature descriptor for object detection in computer vision and image processing, and has since been popular for its simplicity, desirable invariance properties, and performance stability [28], [Chap. 9, 11].

To illustrate the concept, let the raw input from the MNIST data set assume the form $\mathcal{D}_j = \{(\boldsymbol{D}_i^{(j)}, l_j)$ for $i = 1, 2, ..., n_j\}$ where $\boldsymbol{D}_i^{(j)}$ is a digital image representing the ith digit in class $\mathcal{D}_j$ and l_j is the label of the digits in class $\mathcal{D}_j$. For the MNIST data set, we always have $l_j = j$, and each of the images is of dimension 28 × 28 pixels. Given an MNIST image $\boldsymbol{D}$, its HOG can be constructed by dividing $\boldsymbol{D}$ into a total of 196 non-overlapping sub-images, each of size 2 × 2 pixels, which can be referred to as the cells of the image so there are 14 rows and 14 columns of cells. We denote the cell in the ith row and jth column as $C(i, j)$.

Next, we form image blocks where each block is a 2 × 2 cell sub-image; in effect, an image block is a 4 × 4 sub-matrix. Unlike cells, image blocks overlap and are constructed as follows. Starting from the 4 cells in the upper left corner of the image, the blocks are formed row by row and the last block is constructed using the 4 cells at the bottom right corner. The first block denoted as $B(1, 1)$ is a square of size 4 × 4 which consists of the 4 cells

from the upper left corner of the image, namely, the 4 cells {$C(1, 1)$, $C(1, 2)$, $C(2, 1)$; $C(2, 2)$}. The next block denoted as $B(1, 2)$ consists of the 4 cells {$C(1, 2)$, $C(1, 3)$, $C(2, 2)$, $C(2, 3)$} and, as can be seen, it has 50% overlap with block $B(1, 1)$. The rest of the image blocks involving the first and second rows of cells can be similarly defined with the last block being $B(1, 13)$ which consists of the 4 cells {$C(1, 13)$, $C(1, 14)$, $C(2, 13)$, $C(2, 14)$} and has 50% overlap with block $B(1, 12)$. The second row of blocks involves the second and third rows of cells and it starts from block $B(2, 1)$ which has 50% overlap with block $B(1, 1)$ and consists of the 4 cells {$C(2, 1)$, $C(2, 2)$, $C(3, 1)$, $C(3, 2)$}. The remaining blocks $B(2, 2)$, …, $B(2, 13)$ are defined in a similar manner. The formation process continues until block $B(13, 13)$ consisting of cells {$C(13, 13)$, $C(13, 14)$, $C(14, 13)$, $C(14, 14)$} is formed. Evidently, there are 13 rows and 13 columns of overlapping blocks and, in effect, there is a total of 169 blocks. The manner in which the image blocks are formed immediately implies that each inner cell (i.e., a cell that does not contain boundary pixels) is involved in four blocks and each boundary cell is involved in two blocks. On the other hand, each of the cells in the corners of the image is involved in only one block.

The gradient $\nabla \boldsymbol{D}$ of image $\boldsymbol{D}$ is a vector field over the image's domain and is defined by

$$\nabla \boldsymbol{D} = \begin{bmatrix} \partial \boldsymbol{D} / \partial x \\ \partial \boldsymbol{D} / \partial y \end{bmatrix}$$

For digital images, however, the partial derivatives must be approximated by partial differences along the x and y directions, respectively, as

$$\frac{\partial \boldsymbol{D}}{\partial x} \approx \frac{\boldsymbol{D}(x+\delta, y) - \boldsymbol{D}(x-\delta, y)}{2\delta}$$

$$\frac{\partial \boldsymbol{D}}{\partial y} \approx \frac{\boldsymbol{D}(x, y+\delta) - \boldsymbol{D}(x, y-\delta)}{2\delta}$$

In practice, the approximated gradient over the entire image domain is evaluated by using two-dimensional discrete convolution as

$$\frac{\partial \boldsymbol{D}}{\partial x} \approx \boldsymbol{D} \otimes \boldsymbol{h}_x, \ \frac{\partial \boldsymbol{D}}{\partial y} \approx \boldsymbol{D} \otimes \boldsymbol{h}_y$$

where $\boldsymbol{h}_x$ and $\boldsymbol{h}_y$ are given by

$$\boldsymbol{h}_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix},\ \boldsymbol{h}_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

In what follows, the discrete gradient of image $\boldsymbol{D}$ is denoted by $\{\nabla_x \boldsymbol{D}, \nabla_y \boldsymbol{D}\}$. The gradient $\nabla \boldsymbol{D}$ provides two important pieces of information about the image's local structure, namely, the magnitude $\| \nabla \boldsymbol{D} \|_2$ and angle $\Theta(x, y)$ of the gradient, which can be computed as

$$\| \nabla \boldsymbol{D} \|_2 = \sqrt{(\nabla_x \boldsymbol{D})^2 + (\nabla_y \boldsymbol{D})^2}\ ,\quad \Theta = \tan^{-1}\left(\frac{\nabla_y \boldsymbol{D}}{\nabla_x \boldsymbol{D}}\right)$$

To facilitate the definition of the HOG of an image, we divide the angle range $(-\pi, \pi)$ evenly into β bins with the first and second bins associated with the ranges $-\pi$ to $-\pi + \Delta$ and $-\pi + \Delta$ to $-\pi + 2\Delta$ where $\Delta = 2\pi / \beta$, and so on. The 169 image blocks defined earlier can be arranged as a sequence {*B*(1, 1), …, *B*(1, 13), *B*(2, 1), …, *B*(2, 13), …, *B*(13, 1), …, *B*(13, 13)}. We can now associate each block *B*(*i*, *j*) with a column vector $\boldsymbol{h}_{i,j}$ of length β, which is initially set to $\boldsymbol{h}_{i,j} = \boldsymbol{0}$, and then associate each component of $\boldsymbol{h}_{i,j}$ with a bin defined above. Since *B*(*i*, *j*) is of size 4 × 4, it produces 16 pairs $\{\Theta, \| \nabla \boldsymbol{D} \|_2\}$. For each pair, the magnitude $\| \nabla \boldsymbol{D} \|_2$ is added to the corresponding component of $\boldsymbol{h}_{i,j}$. Vector $\boldsymbol{h}_{i,j}$ so obtained is called the HOG of block *B*(*i*, *j*) and the HOG of image $\boldsymbol{D}$ is defined by stacking all $\{\boldsymbol{h}_{i,j}\}$ in an orderly fashion as

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{h}_{1,1} \\ \vdots \\ \boldsymbol{h}_{1,13} \\ \boldsymbol{h}_{2,1} \\ \vdots \\ \boldsymbol{h}_{13,13} \end{bmatrix}$$

For an image of size 28 × 28, the length of $\boldsymbol{x}$ is equal to $169 \times \beta$. Typically the value of β (i.e., the number of bins per histogram) is in the range 7 to 9. With β = 7, for example, the vector $\boldsymbol{x}$ is of length 1183. In a supervised learning problem, for a class of training data that contains *N* images where each image $\boldsymbol{D}_i$ is associated with an HOG vector $\boldsymbol{x}_i$, the HOG

feature of the entire data set is a matrix of size 1183 by N in the form

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_N \end{bmatrix}$$

3.4.2 Support Vector Machines

SVMs are a set of related supervised learning methods used for classification and regression, which were first introduced in 1992 by Boser, Guyon, and Vapnik, and SVMs belong to the family of linear classifiers.

For binary classification of a linearly separable data set $\{(\boldsymbol{x}_n, y_n)$ for $n = 1,2, \ldots, N\}$ where $y_n = 1$ for the "positive" class and $y_n = -1$ for the "negative" class, a linear decision boundary can be found to separate the data set. Through appropriate scaling of parameters $\{\boldsymbol{w}, b\}$, the decision boundary of the data set assumes the form

$$\boldsymbol{w}^T \boldsymbol{x} + b = 0$$

which satisfies $|\boldsymbol{w}^T \boldsymbol{x}_n + b| = 1$ for the data samples that are the nearest to the decision boundary. SVM refers to an algorithm that identifies the optimal decision boundary that separates the two data classes with *maximum margin* for the data samples from both sides of the boundary, see Figure 20. Analytically the parameters $\{\boldsymbol{w}, b\}$ defining the optimal boundary can be characterized by the constraints

$$\boldsymbol{w}^T \boldsymbol{x}_n + b \geq 1 \quad \text{if } y_n = 1$$

$$\boldsymbol{w}^T \boldsymbol{x}_n + b \leq -1 \quad \text{if } y_n = -1$$

which are evidently equivalent to

$$y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1 \text{ for } n = 1, 2, \ldots N$$

With simple linear algebra, it can be shown that the distance between the decision boundary to the nearest data sample is inversely proportional to the 2-norm $\|\boldsymbol{w}\|_2$ (see Fig. 20) and hence the optimal parameters $\{\boldsymbol{w}, b\}$ can be obtained by solving the constrained problem

$$\begin{aligned} &\text{maximize} \quad \frac{1}{\|\boldsymbol{w}\|_2} \\ &\text{subject to:} \quad y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1 \quad \text{for } n = 1, 2, \ldots, N \end{aligned}$$

or equivalently,

$$\begin{aligned} &\text{minimize} \quad \tfrac{1}{2} \boldsymbol{w}^T \boldsymbol{w} \\ &\text{subject to:} \quad y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1 \quad \text{for } n = 1, 2, \ldots, N \end{aligned}$$

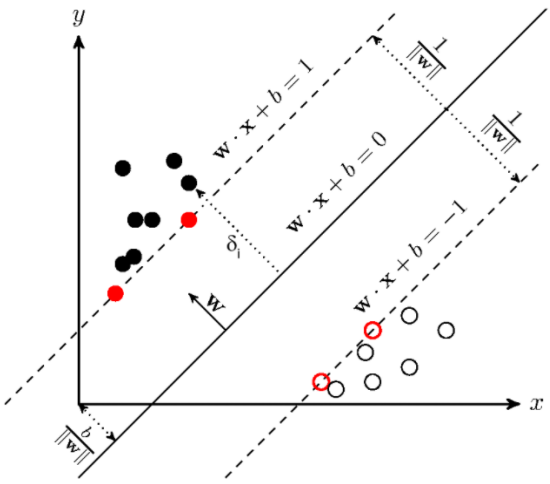

Figure 20. Support vector machine for two linearly separable data sets.

which is obviously a *convex quadratic programming* (QP) problem that can readily be solved using CVX [29].

In the literature, the QP problem described above is called the *primal* problem and it has a dual formulation. The dual problem can be derived via the Karush-Kuhn-Tucker (KKT) conditions of the primal problem [11]. Recall the Lagrangian of the primal problem given by

$$L(\boldsymbol{w}, b, \boldsymbol{\mu}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \sum_{n=1}^{N}\mu_n(1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b))$$

The KKT conditions for the solution of the primal problem are found to be

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{n=1}^{N}\mu_n y_n \boldsymbol{x}_n = \boldsymbol{0}$$

$$\frac{\partial L}{\partial b} = -\sum_{n=1}^{N}\mu_n y_n = 0$$

$$\mu_n(1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)) = 0 \quad \text{for } n = 1, \ldots, N$$

$$\mu_n \geq 0 \quad \text{for } n = 1, \ldots, N$$

and the Lagrange dual of the primal problem is given by [11]

$$\underset{\boldsymbol{\mu}}{\text{minimize}} \quad \frac{1}{2}\boldsymbol{\mu}^T(\boldsymbol{d}\cdot\boldsymbol{d}^T)\boldsymbol{\mu} - \boldsymbol{e}^T\boldsymbol{\mu}$$
$$\text{subject to:} \quad \boldsymbol{\mu}^T\boldsymbol{y} = 0, \quad \boldsymbol{\mu} \geq \boldsymbol{0}$$

where $\boldsymbol{d} = \begin{bmatrix} y_1\boldsymbol{x}_1 & y_2\boldsymbol{x}_2 & \cdots & y_N\boldsymbol{x}_N \end{bmatrix}^T$ and $\boldsymbol{y} = [y_1\, y_2 \ldots y_N]^T$.

It is important to note that the solutions of the primal and dual problems are closely related to each other. In effect, using the first KKT condition given above, the solution $\boldsymbol{\mu}$ of the dual problem is connected to the solution $\{\boldsymbol{w}, b\}$ of the primal problem by

$$\boldsymbol{w} = \sum_{n=1}^{N} \mu_n y_n \boldsymbol{x}_n$$

In addition, by using the third KKT condition with $\mu_m \neq 0$, we obtain

$$b = y_m - \boldsymbol{w}^T \boldsymbol{x}_m$$

Consequently, the solution $\{\boldsymbol{w}, b\}$ of the primal problem can be obtained by solving the dual problem and then using the two formulas shown above.

Since the dual is also a convex QP problem, using CVX to solve it becomes rather straightforward:

```
e = ones(N,1);
cvx_begin quiet
  variable u(N)
  v = X*(u.*y);
  minimize(0.5*v'*v - e'*u)
  subject to
  y'*u == 0;
  u >= 0;
cvx_end
```

In the above code, `u` denotes design variable $\boldsymbol{\mu}$ and `X` denotes data matrix [$\mathbf{x}_1$ $\mathbf{x}_2$ $\cdots$ $\mathbf{x}_N$].

The SVM algorithm described above can be extended to classify K data classes with $K > 2$ through the one-versus-the-rest technique, where a total of K rounds of the binary SVM are applied and in the jth round the jth class is regarded as class $\mathcal{P}$ while the rest of the data samples is regarded as class $\mathcal{N}$. In this way, a total of K decision boundaries are generated. A test sample is classified to class j if the jth decision boundary is the closest to the test sample.

3.4.3 Principal Component Analysis

Principal component analysis (PCA) is best known as a fast and effective feature extraction method in data processing. It can also be used for multi-category classification [28]. Below we describe a PCA-based classifier for the HWDR problem with the MNIST data set as input.

First, the given training data are sorted into 10 classes, each contains only the same digits. Let $\mathcal{D}_j = \{(\boldsymbol{x}_i^{(j)}, y_j), i = 1, 2, \ldots, n_j\}$ be the jth data class with identical labels $y_j \equiv j$.

Next, for each data class we compute its mean vector and covariance matrix

$$\boldsymbol{\mu}^{(j)} = \frac{1}{n_j}\sum_{i=1}^{n_j} \boldsymbol{x}_i^{(j)} \quad \text{and} \quad \boldsymbol{C}_j = \frac{1}{n_j - 1}\sum_{i=1}^{n_j}\left(\boldsymbol{x}_i^{(j)} - \boldsymbol{\mu}^{(j)}\right)\left(\boldsymbol{x}_i^{(j)} - \boldsymbol{\mu}^{(j)}\right)^T$$

and compute the q largest eigenvalues of $\boldsymbol{C}_j$ and the associated eigenvectors $\{\boldsymbol{u}_i^{(j)}, i = 1, 2, \ldots, q\}$. These eigenvectors are called the first *q principal axes* of $\mathcal{D}_j$, and used to construct matrix $\boldsymbol{U}_q^{(j)} = \left[\boldsymbol{u}_1^{(j)} \quad \boldsymbol{u}_2^{(j)} \quad \cdots \quad \boldsymbol{u}_q^{(j)}\right]$.

For a given test sample $\boldsymbol{x}$, its first *q principal components* associated with the jth class are the q components of the vector

$$\boldsymbol{f}^{(j)} = \boldsymbol{U}_q^{(j)T}\left(\boldsymbol{x} - \boldsymbol{\mu}^{(j)}\right) \quad \text{for } j = 0, 1, \ldots, 9$$

which are the projections of $\boldsymbol{x} - \boldsymbol{\mu}^{(j)}$ onto the q principal axes of $\mathcal{D}_j$.

Next, the q principal components are used to construct a projection in the original input space because it has the same dimensionality as that of the test sample $\boldsymbol{x}$:

$$\boldsymbol{z}_j = \boldsymbol{U}_q^{(j)}\boldsymbol{f}^{(j)} \quad \text{for } j = 0, 1, \ldots, 9$$

Finally, the distance between $\boldsymbol{x} - \boldsymbol{\mu}^{(j)}$ and its projection $\boldsymbol{z}_j$ is evaluated as

$$e_j = \| \boldsymbol{x} - \boldsymbol{\mu}^{(j)} - \boldsymbol{z}_j \|_2 \quad \text{for } j = 0, 1, \ldots, 9$$

The test sample $\boldsymbol{x}$ is classified to class j^* if the smallest distance is reached by e_{j^*}. This is to say, the classifier is built as

$$j^* = \arg\min_{0 \le j \le 9}\{e_j\}$$

3.4.4 Performance Comparisons

Classification techniques based on SVM and PCA were applied to the MINIST data set. Rather than using the raw MNIST data directly for training, in our simulation SVM was applied to the HOG of the MNIST data set with $\beta = 7$. Using only $N = 1600$ training samples (with 160 samples for each digit), it took the linear SVM 24.74 minutes to optimize ten decision boundaries. The SVM-based classifier offers a 96.59 % classification accuracy. The use of only a limited number of training samples in the SVM algorithm has to do with the fact that the size of the dual QP problem is determined by the number of training samples involved, namely *N*, and a larger N simply means a training time longer than 24.74 minutes which is already quite long relative to those required by the CNN-based techniques. Unfortunately, the use of small-size training data set has led to a prediction rate that fails to compete with those achieved using CNNs.

Like the SVM algorithm, the PCA classifier was also applied to the HOG of the MNIST data set with $\beta = 7$. Using a total of 16,000 training samples (with 1600 samples for each digit), the PCA-based algorithm achieves a 98.48% prediction accuracy in 2.78 minutes.

Simulation details in term of prediction accuracy and training/testing times of all the algorithms described in this report are summarized in Table 11. It is observed that with HOG as features of the input data, PCA offers good performance and fast training but the SVM seems to be less attractive within the given computing resource. In any account, however, the CNNs, especially the modified LeNet-5 presented in Sec. 3.3.3, remain to be superior practically in all aspects.

Table 11. Performance Summary

Classifier	Accuracy (%)	Time	
		Training (minutes)	Testing (seconds)
50 neurons neural network	96.25	2.94	0.27
Basic one convolution 3 layers convolutional neural network	98.49	1.82	2.03
	98.55	4.02	2.39
	98.61	11.08	2.11
Two convolution 4 layers convolutional neural network	98.87	1.91	2.14
	99.15	4.15	2.45
	99.16	10.06	2.33
Three convolution 6 layers convolutional neural network	99.27	2.21	2.19
	99.31	4.03	2.00
	99.43	11.07	2.15
LeNet-5 with tanh activation	99.08	6.67	2.21
Modified LeNet-5	99.29	2.02	2.42
	99.41	4.10	2.55
	99.51	11.75	2.11
Linear SVM with HOG	96.59	24.74	11.36
PCA with HOG	98.48	2.78	11.30

Chapter 4

Concluding Remarks

In closing, we would like to quote [6]: "*A key advantage of neural networks over traditional machine learning is that the former provides a higher-level abstraction of expressing semantic insights about data domains by architectural design choices in the computational graph. The second advantage is that neural networks provide a simple way to adjust the complexity of a model by adding or removing neurons from the architecture according to the availability of training data or computational power. A large part of the recent success of neural networks is explained by the fact that the increased data availability and computational power of modern computers has outgrown the limits of traditional machine learning algorithms, which fail to take full advantage of what is now possible. The performance of traditional machine learning remains better at times for smaller data sets because of more choices, greater ease of model interpretation, and the tendency to hand-craft interpretable features that incorporate domain-specific insights.*

With limited data, the best of a very wide diversity of models in machine learning will usually perform better than a single class of models (like neural networks). This is one reason why the potential of neural networks was not realized in the early years. The "big data" era has been enabled by the advances in data collection technology; virtually everything we do today, including purchasing an item, using the phone, or clicking on a site, is collected and stored somewhere. Furthermore, the development of powerful GPUs has enabled increasingly efficient processing on such large data sets. These advances largely explain the recent success of deep learning using algorithms that are only slightly adjusted from the versions that were available two decades back. Furthermore, these recent adjustments to the algorithms have been enabled by increased speed of computation, because reduced run-times enable efficient testing (and subsequent algorithmic adjustment). If it requires a month to test an algorithm, at most twelve variations can be tested in an year on a single hardware platform. This situation has historically constrained the intensive experimentation required for tweaking neural-network learning algorithms. The rapid advances associated with the three pillars of improved data, computation, and experimentation have resulted in an increasingly optimistic outlook about the future of

deep learning. By the end of this century, it is expected that computers will have the power to train neural networks with as many neurons as the human brain. Although it is hard to predict what the true capabilities of artificial intelligence will be by then, our experience with computer vision should prepare us to expect the unexpected."

From the point of the insightful view cited above, the report is merely a small-scale demonstration of CNNs as a powerful general tool to learn data. Our demonstration is made possible by using the current version of MATLAB deep learning toolbox for constructing various neural networks, especially CNNs.

On a more technical side, we presented several variants of the famous LeNet-5 to show that there are still rooms for small but useful performance improvement over LeNet-5 without sacrifices in training and testing times. Nevertheless, the modified LeNet-5 architectures presented in this report are merely a small number of instances and many CNNs with improved structures do exist in the literature [30].

Appendix MATLAB Codes

A. Code for Datastore Preparation

```
% Convert MNIST training data into PNG Image files
load Tr28.mat; load Ltr28.mat;
tr = Tr28;
ltr = Ltr28';
len = length(ltr);
uni_ltr = unique(ltr);
cpath = pwd;
for l = 1:length(uni_ltr)
    label = num2str(uni_ltr(l));
    mkdir(fullfile(cpath,'tr', label));
end
count = 0;
for n = 1:len
    count = count+1;
    digit = reshape(tr(:,n), [28 28]);
    label = num2str(ltr(n));
    count_str = num2str(count);
    fname = fullfile(cpath,'tr', label,[label '_' count_str '.png']);
    imwrite(digit, fname);
end

% Convert MNIST testing data into PNG Image files
load Te28.mat; load Lte28.mat;
te = Te28;
lte = Lte28';
len = length(lte);
uni_ltr = unique(lte);
cpath = pwd;
for l = 1:length(uni_ltr)
    label = num2str(uni_ltr(l));
    mkdir(fullfile(cpath,'te', label));
end
count = 0;
for n = 1:len
    count = count+1;
    digit = reshape(te(:,n), [28 28]);
    label = num2str(lte(n));
    count_str = num2str(count);
    fname = fullfile(cpath,'te', label,[label '_' count_str '.png']);
    imwrite(digit, fname);
end
% create image datastore
cpath = pwd;
tr_path = fullfile(cpath, 'tr');
te_path = fullfile(cpath, 'te');
ds_path = fullfile(cpath);
verbose = true;
visualize = false;
trds = imageDatastore(tr_path,...
'IncludeSubfolders',true,'FileExtensions','.png','LabelSource', ...
'foldernames');
```

```
save(fullfile(ds_path,'trds.mat'),'trds');
teds = imageDatastore(te_path,...
'IncludeSubfolders',true,'FileExtensions','.png','LabelSource', ...
'foldernames');
save(fullfile(ds_path,'teds.mat'),'teds');
```

B. Code for Basic Neural Network as Applied to MNIST Dataset

```
clear;clc;
load Tr28.mat; load Ltr28.mat; load Te28.mat; load Lte28.mat;
ltr = Ltr28';   lte = Lte28';
ltr(ltr==0)=10; lte(lte==0)=10;
ltr = dummyvar(ltr); lte = dummyvar(lte);
for state = 1:20
    fprintf('State: %2.0f\n',state);
    rand('state',state);
    trainFcn = 'trainscg';
    net.divideParam.trainRatio = 80/100;
    net.divideParam.valRatio = 20/100;
    net.divideParam.testRatio = 0/100;
    net.performFcn = 'crossentropy';
    net.plotFcns =...
{'plotperform','plottrainstate','ploterrhist','plotconfusion', ...
'plotroc'};
    for i = 10:10:50
        t=cputime;
        net = patternnet(i,trainFcn);
        [net,~] = train(net,Tr28,ltr');
        time(i/10) = cputime - t;
        tsty = net(Te28);
        tind = vec2ind(lte');
        yind = vec2ind(tsty);
        percent = sum(tind ~= yind)/numel(tind);
        perc(i/10) = percent;
    end
    for j = 1:5
        fprintf('%1.2fm    ',time(j)/60);
    end
    fprintf('\n');
    for k = 1:5
        fprintf('%2.2f%%   ',100-perc(k)*100);
    end
    fprintf('\n');
end
```

C. Code for CNN as Applied to MNIST Dataset

```
%   Basic CNN with only 1 convolutional layer
function CNN3
    clear; clc;
    load trds.mat; load teds.mat;
    s = 1;          % stride
    p = 1;          % padding
    for lrf = 3:2:7
```

```
        for f = 3:3:9
            if f ~= 9,
               fm = f;
            else
               fm = f-1;
            end
            for e = 1:3
                if e == 1
                   epoch=4;
                elseif e == 2
                   epoch=10;
                else
                   epoch=30;
                end
                fprintf('%2.0f:  %2.0f,%2.0f       ',epoch,lrf,fm);
                layers = [...
                    imageInputLayer([28 28 1])
                    convolution2dLayer(lrf,fm,'Padding',p)
                    batchNormalizationLayer
                    reluLayer
                    maxPooling2dLayer(2,'Stride',s)
                    fullyConnectedLayer(10)
                    softmaxLayer
                    classificationLayer];
                options = trainingOptions('sgdm', ...
                    'MaxEpochs',epoch,...
                    'InitialLearnRate',1e-2, ...
                    'Shuffle','every-epoch',...
                    'Verbose',false, ...
                    'Plots','training-progress',...
                    'ExecutionEnvironment','gpu');
                t = cputime;
                convnet = trainNetwork(trds,layers,options);
                time = cputime - t;
                tep = classify(convnet,teds);
                tev = teds.Labels;
                acc = sum(tep == tev)/numel(tev);
                fprintf('accuracy: %3.2f%%,  Training ...
                time: %2.2fm\n',acc*100,time/60);
             end
          end
       end

%   CNN with 2 convolutional layers
function CNN4
clear; clc;
load trds.mat; load teds.mat;
s = 1;      % stride
for e = 1:3
    if e == 1
       epoch = 4;
    elseif e == 2
       epoch = 10;
    else
       epoch = 30;
    end
    for lrf1 = 3:2:7
```

```
        if lrf1 ~= 3
           fprintf('\n');
        end
        for fm1 = 3:6:9
            for lrf2 = 3:4:7
                for fm2 = 3:6:9
            fprintf('%1.0f, %1.0f, %1.0f, %1.0f:\n',lrf1,lrf2,fm1,fm2);
            layers = [...
                      imageInputLayer([28 28 1])
                      convolution2dLayer(lrf1,fm1,'Padding',1)
                      batchNormalizationLayer
                      reluLayer
                      maxPooling2dLayer(2,'Stride',s)
                      convolution2dLayer(lrf2,fm2,'Padding',1)
                      batchNormalizationLayer
                      reluLayer
                      maxPooling2dLayer(2,'Stride',s)
                      fullyConnectedLayer(10)
                      softmaxLayer
                      classificationLayer];
            options = ...
trainingOptions('sgdm','MaxEpochs',epoch,'InitialLearnRate',1e-2,...
'Shuffle','every-epoch','Verbose',false,'ExecutionEnvironment','gpu');
                    t = cputime;
                    convnet = trainNetwork(trds,layers,options);
                    time = cputime - t;
                    tep = classify(convnet,teds);
                    tev = teds.Labels;
                    acc = sum(tep == tev)/numel(tev);
                    fprintf('Accuracy: %3.2f%%,  Training...
time: %2.2fm\n',acc*100,time/60);
                end
             end
          end
       end
    end

%   CNN with 6 layers, 3 convolutional layers
function CNN6
clear;clc;
load trds.mat; load teds.mat;
s = 1;      % stride
lrf1 = 7;
fm1 = 9;
for e = 1:3
    if e==1
       epoch = 4;
    elseif e==2
       epoch = 10;
    else
       epoch = 30;
    end
    for lrf2 = 5:2:7
        if lrf2 == 7
          fprintf('\n');
        end
        for fm2 = 9:9:18
```

```
            for lrf3 = 5:2:7
                for fmt = 1:3
                    if fmt == 1
                       fm3 = 9;
                    elseif fmt == 2
                           fm3 = 18;
                    else
                           fm3 = 36;
                    end
fprintf('%1.0f, %1.0f, %1.0f, %1.0f, %2.0f, %2.0f:\n',lrf1,lrf2,...
lrf3,fm1,fm2,fm3);
                    layers = [...
                        imageInputLayer([28 28 1])
                        convolution2dLayer(lrf1,fm1,'Padding',1)
                        batchNormalizationLayer
                        reluLayer
                        maxPooling2dLayer(2,'Stride',s)

                        convolution2dLayer(lrf2,fm2,'Padding',1)
                        batchNormalizationLayer
                        reluLayer
                        maxPooling2dLayer(2,'Stride',s)

                        convolution2dLayer(lrf3,fm3,'Padding',1)
                        batchNormalizationLayer
                        reluLayer
                        maxPooling2dLayer(2,'Stride',s)

                        fullyConnectedLayer(10)
                        softmaxLayer
                        classificationLayer];
                    options =...
trainingOptions('sgdm','MaxEpochs',epoch,'InitialLearnRate',1e-2,...
'Shuffle','every-epoch','Verbose',false,'ExecutionEnvironment','gpu');
                    t = cputime;
                    convnet = trainNetwork(trds,layers,options);
                    time = cputime - t;

                    tep = classify(convnet,teds);
                    tev = teds.Labels;
                    acc = sum(tep == tev)/numel(tev);
                    fprintf('Accuracy: %3.2f%%,  Training time: ...
                    %2.2fm\n',acc*100,time/60);
                end
             end
          end
       end
    end
%   Modified LeNet-5
function mln5
clear;clc;
load trds.mat;  load teds.mat;
p = 1;  s = 1;
for e = 1:3
    if e ~= 1
       fprintf('\n\n\n');
    end
```

```
    if e == 1
       epoch = 4;
    elseif e == 2
       epoch = 10;
    else
       epoch = 30;
    end
    for fm1 = 9:9:18
        for fm2 = 18:18:72
            for fm3 = 72:36:180
                fprintf('%2.0f:  %2.0f,%3.0f,%3.0f',epoch,fm1,fm2,...
                fm3);
                layers = [...
                    imageInputLayer([28 28 1])
                    convolution2dLayer(7,fm1,'Padding',p)            %C1
                    batchNormalizationLayer
                    reluLayer
                    maxPooling2dLayer(2,'Stride',s)                  %S2
                    convolution2dLayer(7,fm2,'Padding',p)            %C3
                    batchNormalizationLayer
                    reluLayer
                    maxPooling2dLayer(2,'Stride',s)                  %S4
                    convolution2dLayer(7,fm3,'Padding',p)            %C5
                    batchNormalizationLayer
                    reluLayer
                    fullyConnectedLayer(84)                          %F6
                    fullyConnectedLayer(10)                          %F7
                    softmaxLayer
                    classificationLayer];
                options =...
trainingOptions('sgdm','MaxEpochs',epoch,'InitialLearnRate',1e-2,...
'Shuffle','everyepoch','Verbose',false,'ExecutionEnvironment','gpu');
                t = cputime;
                convnet = trainNetwork(trds,layers,options);
                time = cputime - t;
                tep = classify(convnet,teds);
                tev = teds.Labels;
                acc = sum(tep == tev)/numel(tev);
                fprintf('     accuracy: %3.2f%%, time: %2.2fm\n',...
                acc*100,time/60);
            end
        end
    end
end
```

D. Code for HOG-based Feature Extraction

prep_hog_1.m

```
function Hg = prep_hog_1(Images, B) %SVM
[~,c] = size(Images);
for i = 1:c
    m = Images(:,i);
    Im = reshape(m,28,28);
    [H,~] =...
extractHOGFeatures(Im,'CellSize',[3,3],'BlockSize',[3,3],'NumBins',B);
    Hg(:,i) = double(H');
end
```

prep_hog_2.m

```
function H = prep_hog_2(Image, Bin) %PCA
Im = Image;
B = Bin;
[H,~] = extractHOGFeatures(Im,'CellSize',[3,3],'NumBins',B);
H = H';
```

E. Code for SVM as Applied to MNIST Dataset

main_SVM.m

```
%   Example:
%   [a,t] = main_svm(1600,7,2);
%   Execution type:
%   1:LSVM primal 2:LSVM dual
%   Input:
%   tn: number of training data used for training in each class
%   B:  HOG bin number
%   c:  1 for primal, 2 for dual
%   feature extraction: Bin size:
function [acc_rate,time] = main_SVM(tn,B,c)
    load Tr28.mat; load Ltr28.mat; load Te28.mat; load Lte28.mat;
    [Tr,Ltr,counter] = srt(Tr28,Ltr28,tn);
    Xtrain = prep_hog_1(Tr,B);
    ta = cputime;
    switch c
        %a) linear SVM by primal
        case 1
            [ws,bs] = lsvmp(Xtrain,Ltr);
        %b) linear SVM by dual
        case 2
            [ws,bs] = lsvmd(Xtrain,Ltr);
        otherwise
            fprintf('Error during training');
    end
    time = cputime-ta;
    %   test trained svm
    xte = prep_hog_1(Te28,B);
    l = unique(Ltr);
    acc_rate = mclassify(ws,bs,xte,Lte28,l);
    fprintf('\nAccuracy:%5.2f%%,Time:%2.2fs\n',mean(acc_rate),time);
```

srt.m

```
function [x,y,counter] = srt(testdataset,groundtruth,N)
    x = testdataset;
    y = groundtruth;
    c = length(y);
    counter = ones(10,1);
    for i = 1:c
        tmp = y(i);
        if tmp == 0
            if counter(1) <= N
                cgt0(counter(1),:) = tmp;
                cstds0(:,counter(1)) = x(:,i);
```

```
        counter(1) = counter(1)+1;
    end
elseif tmp == 1
    if counter(2) <= N
        cgt1(counter(2),:) = tmp;
        cstds1(:,counter(2)) = x(:,i);
        counter(2) = counter(2)+1;
    end
elseif tmp == 2
    if counter(3) <= N
        cgt2(counter(3),:) = tmp;
        cstds2(:,counter(3)) = x(:,i);
        counter(3) = counter(3)+1;
    end
elseif tmp == 3
    if counter(4) <= N
        cgt3(counter(4),:) = tmp;
        cstds3(:,counter(4)) = x(:,i);
        counter(4) = counter(4)+1;
    end
elseif tmp == 4
    if counter(5) <= N
        cgt4(counter(5),:) = tmp;
        cstds4(:,counter(5)) = x(:,i);
        counter(5) = counter(5)+1;
    end
elseif tmp == 5
    if counter(6) <= N
        cgt5(counter(6),:) = tmp;
        cstds5(:,counter(6)) = x(:,i);
        counter(6) = counter(6)+1;
    end
elseif tmp == 6
    if counter(7) <= N
        cgt6(counter(7),:) = tmp;
        cstds6(:,counter(7)) = x(:,i);
        counter(7) = counter(7)+1;
    end
elseif tmp == 7
    if counter(8) <= N
        cgt7(counter(8),:) = tmp;
        cstds7(:,counter(8)) = x(:,i);
        counter(8) = counter(8)+1;
    end
elseif tmp == 8
    if counter(9) <= N
        cgt8(counter(9),:) = tmp;
        cstds8(:,counter(9)) = x(:,i);
        counter(9) = counter(9)+1;
    end
elseif tmp == 9
    if counter(10) <= N
        cgt9(counter(10),:) = tmp;
        cstds9(:,counter(10)) = x(:,i);
        counter(10) = counter(10)+1;
    end
end
```

```
    end
    srtds = [cstds0,cstds1,cstds2,cstds3,cstds4];%
    sortedtds = [srtds,cstds5,cstds6,cstds7,cstds8,cstds9];
    sortedgt = [cgt0;cgt1;cgt2;cgt3;cgt4;cgt5;cgt6;cgt7;cgt8;cgt9];
    tot = 0;
    for i = 1:10
        tot = counter(i)-1+tot;
    end
    x = sortedtds(:,1:tot);
    y = sortedgt(1:tot);
    fprintf('Sort Completed!\n');
end
```

lsvmp.m

```
function [ws,bs] = lsvmp(x,y)
% This function trains SVM with linear SVM primal.
[row,~] = size(x);          % get training data size
list = unique(y);           % get the list of class numbers
numc = numel(list);         % get the number of classes
ws = zeros(row,numc);
bs = zeros(numc,1);
    for k = 1:numc
        t = cputime;                % start timer
        ytmp = y;
        cn = double(ytmp == list(k));
        ytmp(cn == 1) = 1;            % assign the current class 1
        ytmp(cn == 0) = -1/(numc-1); % assign rest classes -1/(numc-1)

        cvx_begin quiet
            variable w(row,1)
            variable b(1,1)
            minimize( 0.5*w'*w )
            subject to
                ytmp.*(x'*w + b) >= 1;
        cvx_end
        ws(:,k) = w;
        bs(k) = b;
        fprintf('Class %d trained in %3.2fm\n',k-1,(cputime-t)/60);
    end
```

lsvmd.m

```
function [ws,bs] = lsvmd(x,y)
% This function trains SVM with linear SVM dual.
[row,col] = size(x);                % get training data size
list = unique(y);                   % get the list of classes number
numc = numel(list);                 % get the number of classes
ws = zeros(row,numc);
bs = zeros(numc,1);
    for k = 1:numc
        t = cputime;                % start timer
        ytmp = y;
        cn = double(ytmp == list(k));
        ytmp(cn == 1) = 1;              % make the current class 1
        ytmp(cn == 0) = -1/(numc-1);    % make the rest to -1/(numc-1)
```

```
        e = ones(col,1);
        cvx_begin quiet
            variable u(col,1)
            dual variables de dp
            v = x*(ytmp.*u);                    % 784*N * (N*1 * N*1)
            minimize( 1/2*v'*v -e'*u )          % #ok<MHERM>
            subject to
                de : ytmp'*u == 0;              % #ok<EQEFF>
                dp : u >= 0;                    % #ok<VUNUS>
        cvx_end
        ws(:,k) = sum(x*(u.*ytmp),2);
        bs(k) = -de;
        fprintf('Class %d trained in %3.2f s\n',k-1,cputime-t);
    end
```

mclassify.m

```
function accuracy = mclassify(w,b,xt,yt,l)
    [~,col] = size(xt);
    numc = numel(l);
    err = 0;
    tmp = yt;
    ind = zeros(col,1);
    ys = ind;
    for j = 1:col
        tm = zeros(numc,1);
        for i = 1:numc
            tm(i) = w(:,i)'*xt(:,j)+b(i);
        end
        [~,ind(j)] = max(tm);
        ys(j) = l(ind(j));
        if l(ind(j)) ~= tmp(j)
            err = err+1;
        end
    end
    accuracy = (col-err)/col*100;
```

F. Code for PCA as Applied to MNIST Dataset

main_PCA.m

```
clear; clc;
load Tr28.mat; load Ltr28.mat; load Te28.mat; load Lte28.mat;
t = cputime;
Ht = gen_hog_digits(Tr28,Ltr28,7,1600,30);
[Js,er,cpt1,cpt2] = pca_hog_digits(Ht,1600,23,Te28,Lte28,7);
time = cputime - t;
fprintf('     accuracy: %2.2f%%,  time: %2.2fm\n',100-er*100,time/60);
```

gen_hog_digits.m

```
function Ht = gen_hog_digits(Tr,Ltr,B,nt,st)
N = 10*nt;
X = zeros(784,N);
for i = 1:10
    iw = i - 1;
    indw = find(Ltr == iw);
    Ti = Tr(:,indw);
    ni = length(indw);
```

```
    rand('state',st+i-1)
    nii = randperm(ni);
    Xi = Ti(:,nii(1:nt));
    X(:,((i-1)*nt+1):(i*nt)) = Xi;
end
hz = 169*B;
Ht = zeros(hz,N);
for i = 1:N
    xi = X(:,i);
    mi = reshape(xi,28,28);
    Ht(:,i) = prep_hog_2(mi,B);
End
```

pca_hog_digits.m

```
% Input:
% X: Ten classes of input data, each is of size 784 by nt.
% n: the actual size of each data matrix is 784 by n.
% Hence n is limited to n <= nt.
% K: the number of "eigen-digits" to be used in PCA. Typically
% K << m = 784.
% Te: testing data (here we use Te28 of size 784 by 10000).
% Lte: Labels for the testing data.
% Output:
% Js: the numerical values of testing digits obtained by
% PCA-based classification.
% er: classification error.
% Written by W.-S. Lu, University of Victoria.
% Last modified: July 4, 2015.
% Example: load Tr28; load Ltr28; load Te28; load Lte28;
% Ht = gen_hog_digits(Tr28,Ltr28,7,1600,30);
% [Js,er,cpt1,cpt2] = pca_hog_digits(Ht,1600,23,Te28,Lte28,7);
% (best error rate: 0.0161)
function [Js,er,cpt1,cpt2] = pca_hog_digits(Ht,n,K,Te,Lte,B)
[t1,t] = size(Ht);
U = [];
Xb = [];
nt = t/10;
t0 = cputime;
for i = 1:10
    Tw = Ht(:,((i-1)*nt+1):(i*nt));
    Ti = Tw(:,1:n);
    xbi = mean(Ti')';
    Xh = Ti - xbi*ones(1,n);
    Pw = (Xh*Xh')/(n-1);
    [u,~] = eig(Pw);
    U = [U u(:,(t1-K+1):t1)];
    Xb = [Xb xbi];
end
cpt1 = cputime - t0;
Lte = Lte(:);
M = length(Lte);
Js = zeros(M,1);
t0 = cputime;
for m = 1:M
    xm = Te(:,m);
    mm = reshape(xm,28,28);
```

```
    xw = prep_hog_2(mm,B);
    e = zeros(1,10);
    for j = 1:10
        Uj = U(:,((j-1)*K+1):(j*K));
        xbj = Xb(:,j);
        vj = xw - xbj;
        zj = Uj'*vj;
        e(j) = vj'*vj - zj'*zj;
    end
    [~,ind] = min(e);
    Js(m) = ind - 1;
end
cpt2 = (cputime - t0)/M;
E = (Lte ~= Js);
er = sum(E)/M;
```

References

[1] N. Nilsson, *Learning Machines*, McGraw Hill, 1965.

[2] R. Duda and P. Hart, *Pattern Recognition and Scene Analysis*, Wiley Interscience, 1973.

[3] W. Sarle, *Neural Networks and statistical models*, 1994. *CiteSeerX* 10.1.1.27.699.

[4] S. Russell and P. N*orvig, Artificial Intelligence*: *A Modern Approach*, 2nd ed., Prentice Hall, 2003.

[5] P. Langley, "The changing science of machine learning", *Machine Learning*, vol. 82, no. 3, pp. 275–279, 2011.

[6] C. C. Aggarwal, *Neural Networks and Deep Learning – A Textbook*, Springer, 2018.

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

[8] G. Cybenko, "Approximations by superpositions of sigmoidal functions," *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4, pp. 303-314, 1989.

[9] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251-257, 1991.

[10] D. Rumelhart, G. Hinton, and R. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323 (6088), pp. 533-536, 1986.

[11] A. Antoniou and W.-S. Lu, *Practical Optimization: Algorithms and Engineering Applications*, Springer, 2007.

[12] M. A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.

[13] Y. LeCun, L. D. Jackel, L. Bottou, A. Brunot, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard, and V. Vapnik, "Comparison of learning algorithms for handwritten digit recognition," *Int. Conf. on Artificial Neural Networks*, pp. 53–60, 1995.

[14] G. E. Hinton, P. Dayan, and M. Revow, "Modeling the manifolds of images of handwritten digits," *IEEE Trans. Neural Networks*, vol. 8, no.1, pp. 65–74, Jan. 1997.

[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, pp. 2278–2324, Nov. 1998.

[16] F. Lauer, C. Y. Suen, and G. Bloch, "A trainable feature extractor for handwritten digit recognition," *Pattern Recognition*, vol. 40, pp. 1816–1824, 2007.

[17] MNIST, http://yann.lecun.com/exdb/mnist/

[18] H. D. Hubel and T. N. Wiesel, " Receptive Fields of Single neurones in the Cat's Striate Cortex," *Journal of Physiology*, vol 148, pp. 574-591, 1959.

[19] 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, 1982.

[20] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Proc. Advances in Neural Information Processing Systems* (NIPS), pp. 396–404, 1989.

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

[22] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436-444, May 2015.

[24] UFLDL, Using the MNIST dataset, http://udl.stanford.edu/wiki/index.php/Using the MNIST Dataset.

[25] http://udl.stanford.edu/wiki/resources/mnistHelper.zip

[26] B. Kim, Y. H. Park, "Beginner's guide to neural networks for the MNIST dataset using MATLAB," *Korean J. Math.*, vol. 26, no 2, pp. 337-348, 2018.

[27] N. D. Dalal and B. Triggs, "Histogram of oriented gradients for human detection," in *Proc. CVPR*, San Diego, June 2005.

[28] W.-S. Lu, "Handwritten digits recognition using PCA of histogram of oriented gradient," *Proc. PacRim Conf.*, Victoria, BC, Canada, August 2017.

[29] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming," version 2.0 beta, http://cvxr.com/cvx, Sept. 2013.

[30] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," 2012, DOI: 10.1109/CVPR.2012.6248110.