Image Recognition with Convolutional Neural Networks

CASE STUDY
SEPTEMBER 2019

BY
YIXIN HUANGFU
CENTRE FOR MECHATRONICS & HYBRID TECHNOLOGIES
224-200 LONGWOOD ROAD SOUTH
HAMILTON, ON
L8P 0A6, CANADA

Table of Contents

Image Recognition with Convolutional Neural Networks

1 INTRODUCTION

Image recognition is the ability of computer software to recognize objects, such as people and animals, by analyzing the patterns in optical images. Its applications have gained a large amount of popularity in the recent decade, such as face recognition for photographing, internet content filtering, advanced driving assistance system, and self-driving vehicles. All these applications will not be feasible without the recent development of neural networks and the computational capable hardware. In this case study, we will dive into the fundamental of a popular image classification structure – the Convolutional Neural Networks (CNN) – and implement it in Matlab to recognize hand-written digits.

Section 2 of this document introduces neural networks, the convolution operation, a few critical machine learning concepts, and some state-of-the-art CNN models. In Section 3, a hands-on Matlab tutorial demonstrates the model configuration, training process, and performance evaluation. The digits dataset and Matlab code are attached to this document.

2 ALGORITHMS FOR IMAGE RECOGNITION

2.1 MULTILAYER PERCEPTRON

Artificial neural networks (ANN) is a subcategory of machine learning algorithms. They are inspired by the brain system and try to use artificial neurons to replicate how humans learn. Each neuron (a.k.a. perceptron) sums up multiple inputs and applies a non-linear function to the summation. The output of this non-linear function is called an *activation*. Figure 1 shows the diagram of an artificial neuron.

Connecting the inputs and outputs of multiple neurons constitutes a *neural network*. Typically, the neurons are organized by layers, as shown in Figure 2. The neurons in each layer only take inputs from the previous layer, and output to the next layer. This setup is

called multilayer perceptron (MLP), and this type of layer is called a *fully-connected layer*. From left to right, the three layers are called the input layer, hidden layer, and output layer, respectively. In the image classification case, the input layer represents an image in vector form, and the output layer represents the class of the object. The value of each output neuron is associated with the probability of the class it represents. The output neuron with the highest probability value is the recognition result.

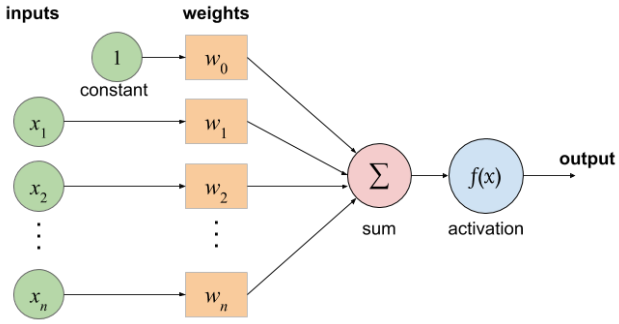


Figure 1: Illustration of one perceptron (change this)

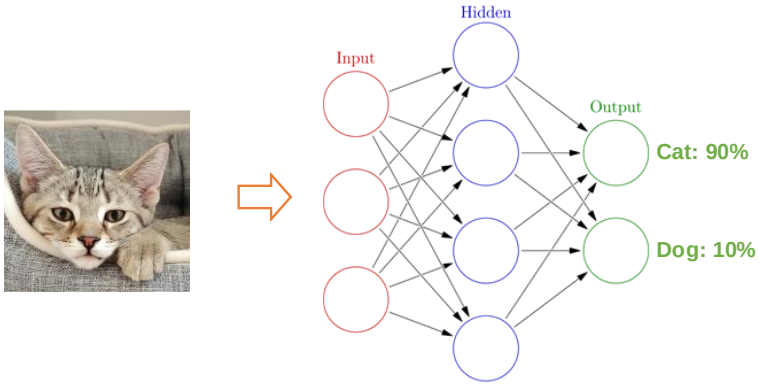


Figure 2: A 3-layer neural network

Each neuron has a series of weights that need to be set to proper values to propagate the correct output. The process of adjusting these weights is called training, and the algorithms used for training are referred to as *optimizers*. During training, the optimizer compares the network output with the true label to obtain an error. This error determines how the weights should be adjusted and by how much. Successful training should show the error decreasing and converging to a small value (if not zero), i.e., the network prediction converging to the true label.

However, there are two problems when directly applying an MLP to image recognition. First, converting an image into an input vector uses a method called *flattening*. Flattening concatenates the rows of a multi-dimensional matrix to produce a vector. This process loses the spatial relationship across multiple rows. For example, two images may contain the same object but with slightly varied sizes and positions. To the human eyes, it is easy to spot the similarity, but their vector representations can be vastly different. Another problem appears as the size of the network grows. A typical image size could be around 480×480 pixels, giving an input layer with a size of 691,200. Fully connecting all these neurons generates a massive number of trainable weights, making the training process extremely slow.

2.2 Convolutional Neural Networks

The introduction of CNN addressed both problems. First, the input of a CNN is matrix rather than a vector, preserving spatial information. The forward propagation is applied directly to the matrix instead of individual neurons. In addition, the weights in a convolution layer are shared across the whole image, reducing the parameters.

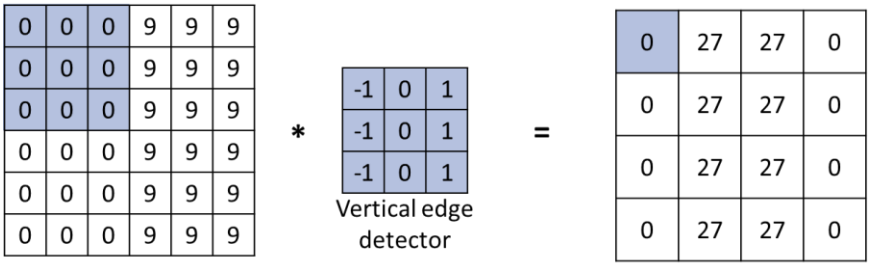


Figure 3: Convolving operation

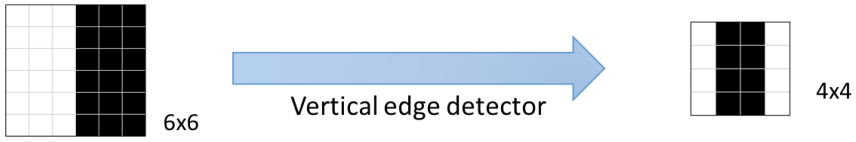


Figure 4: Convolution effect on images

The convolution operation between two matrices is shown in Figure 3. The small 3×3 matrix maps itself to the large matrix (gray area), performing element-wise multiplication,

and then summing the 9 elements to produce one output. The small matrix then scans through the whole large matrix, producing the result matrix. In neural networks, this small matrix is called a kernel or *filter*.

If the input matrix comes from a 6x6 square image and values represent the darkness of the pixels (0~9), this convolution operation works effectively as an edge detector. As illustrated in Figure 4, the input image is half white, half black, with a clear vertical edge in between, so the output highlights the edge in the middle (the value of 27 can be normalized to the scale of 0~9). With different filter configurations, the convolution can extract various features from the original image, such as curves and corners. In practice, the filter values are trainable and get optimized during the training process. The training data determine the most effective filter configurations.

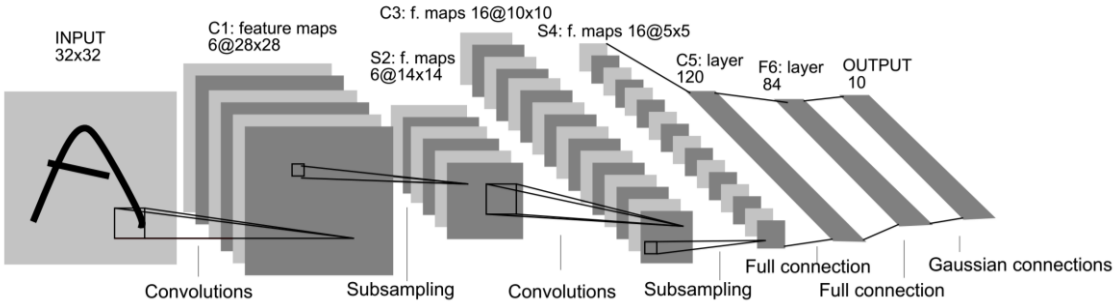


Figure 5: A typical CNN model (LeNet-5)

The whole picture of a CNN model may look like Figure 5 [1]. In this setting, the input layer has a size of 32×32, which is processed by 6 convolutional filters. The size of the second layer is, therefore, 28×28×6. A subsampling operating is simply down sampling the image, picking the dominant value in a subsample window. This process is more commonly known as *pooling*. The same convolution-pooling process repeats, giving the 5^{th} layer with size 5×5×16. The matrix is then flattened and goes through two fully-connected layers described in the previous section. In this particular model, its goal is to recognize 10 digits; therefore, the output layer has a size of 10. Compared to an MLP with the same number of layers, this CNN has significantly less trainable weights. Thanks to the convolutional filters, the spatial information is well preserved in the form of convoluted features.

2.3 KEY CONCEPTS

Here are some useful terminologies for implementing a CNN model.

Activation function. As mentioned in Section 2.1, activation refers to the non-linear function after the weighted sum within a neuron. In CNN, it refers to the non-linear function after convolving the filter and the input matrix. In MLP, sigmoid functions are often selected because they provide smooth gradients and saturation effects. Common examples include the logistic function and hyperbolic tangent (tanh) function. In CNN, the rectified linear unit (ReLU) function, or rectifier is preferred for its computational efficiency, despite it not being continuously differentiable.

$$\text{Logistic function: } f(x) = \frac{1}{1+e^{-x}}$$

$$\text{Rectifier: } f(x) = \max(0, x)$$

Softmax function takes an input vector with N values and normalizes it into probability distributions. The output probability values are proportional to the exponentials of the input values. Softmax is often used in the last layer of a classification network.

$$f(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}, \qquad for\ i = 1, \dots, N$$

Optimizer is the training algorithm to update neural network weights. While many optimization techniques are applicable, the most popular ones are gradient-based. This set of optimizers starts with the error between model prediction and true label and updating the weights by taking their derivatives of the error. Therefore, they are called gradient descent or backpropagation. Some popular variations of this algorithm are stochastic gradient descent, RMSprop, and Adam.

2.4 ADVANCED MODELS

The recent advancement of computing power enables the application of larger, more complex CNN models, which have boosted image processing performance. Here are some of the latest advanced models that have proved effective.

The AlexNet model [2] developed in 2012 incorporating 5 convolutional layers can process 224 x 224 RGB color images and recognize 1000 different objects. The research team also developed practical techniques such as dropout and data augmentation to accelerate the training process.

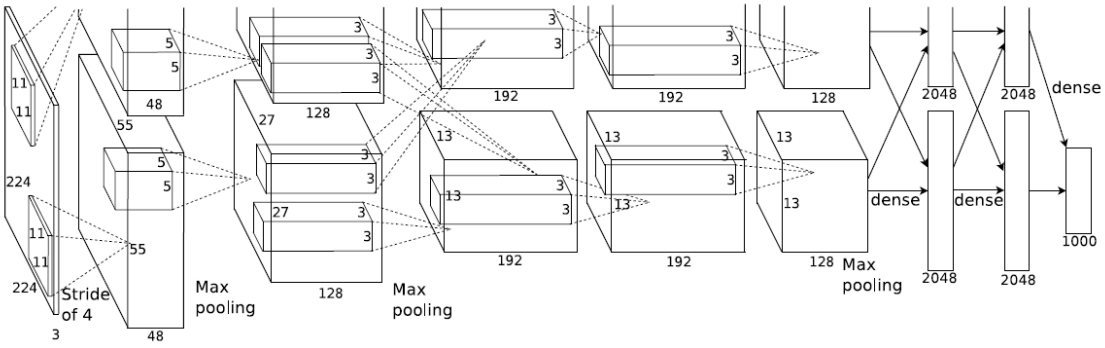


Figure 6: AlexNet model (2012)

The VGG-16/19 model [3] introduced in 2014 with 13/16 convolutional layers proves the effectiveness of a deeper network. The performance has significantly improved compared to AlexNet (error rate 16.4% to 7.3%), while its staggering 130 million trainable weights set a bottleneck for network training.

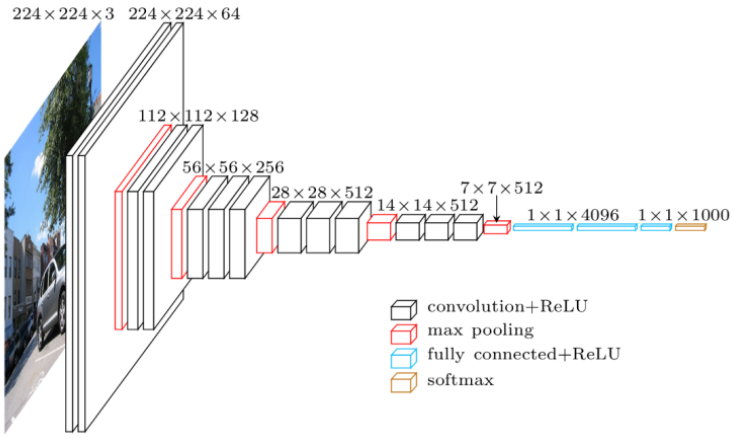


Figure 7: VGG-16 model (2014)

The GoogLeNet model [4] introduced in the same year takes a different approach with multiple paths between layers, letting the network choose the best path. It has significantly fewer weights (6.8M) and performs slightly better compared to VGG models.

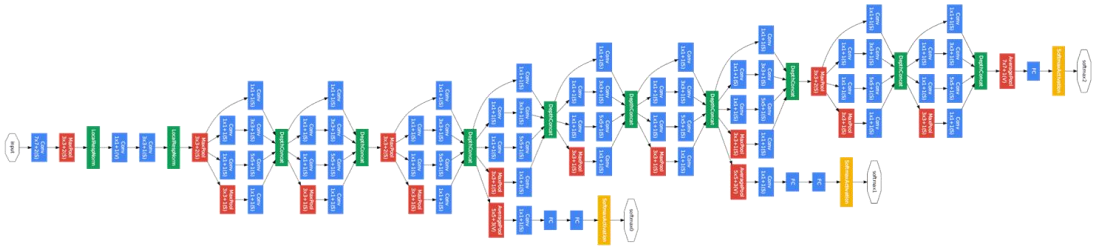


Figure 8: GoogLeNet model (2014)

The ResNet model [5] addresses the gradient vanish problem in deep network training, creating a network depth of 110 layers with 1.7M trainable parameters. Its image classification error reaches 3.57%.

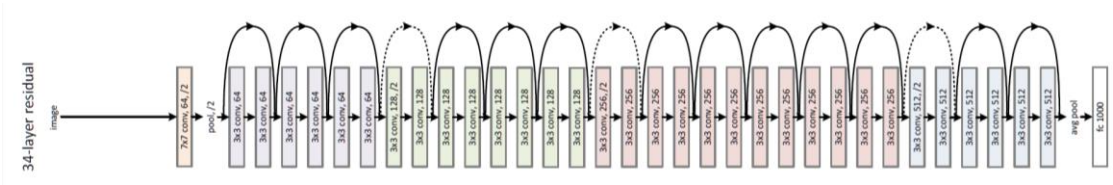


Figure 9: ResNet model (2015)

The performance of image classification using CNN models has reached a human level. Further reduction of the error rate is becoming trivial as the difficult image samples tend to be visually confusing. The accuracy of these state-of-the-art algorithms is high enough for many vision-based applications such as driving assist system and face recognition.

3 MATLAB DEMONSTRATION

In this section, a CNN will be implemented to recognize images of digits using Matlab. All files needed for the tutorial is attached:

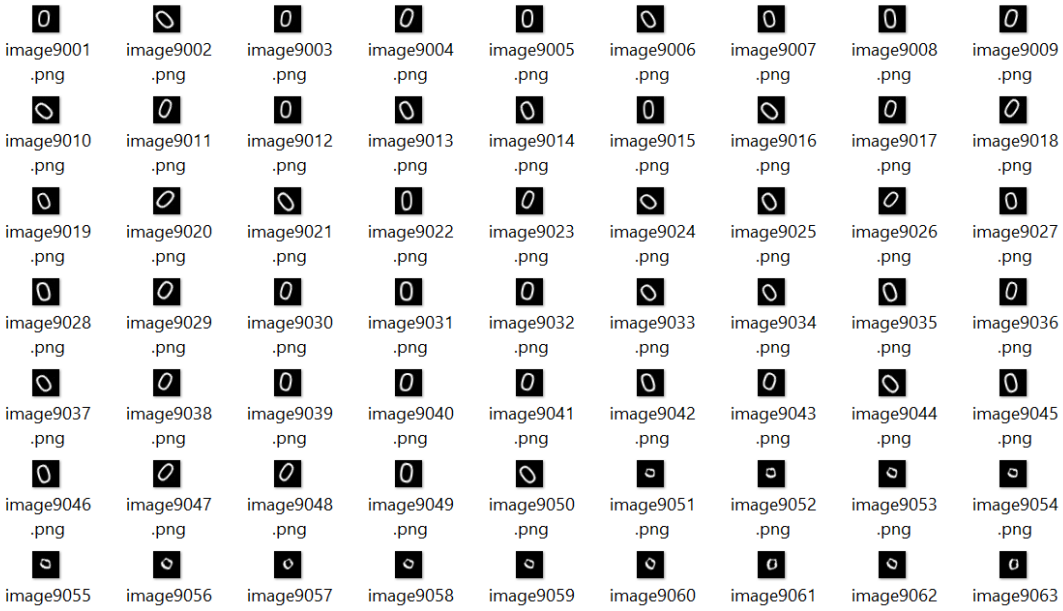
- digits_images.zip: a zip file containing digits images,
- digits_recognition_live.mlx: the Matlab live script.
    - Recommend to open this file in Matlab while going through this tutorial.
- digits_recognition.m: the Matlab script.

- This one is effectively the same as the .mlx file, but less elaborative. Use this unless the .mlx version does not work correctly.

3.1 EXPLORE THE DATASET

The original dataset is openly accessible online at http://yann.lecun.com/exdb/mnist/ [1]. It contains a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a 28×28px gray-scale image. Decoding the original dataset requires specific software skills that are out of the scope of this tutorial. Therefore, we provide the compressed file "digits_images.zip" containing 10,000 individual image files that are ready to use.

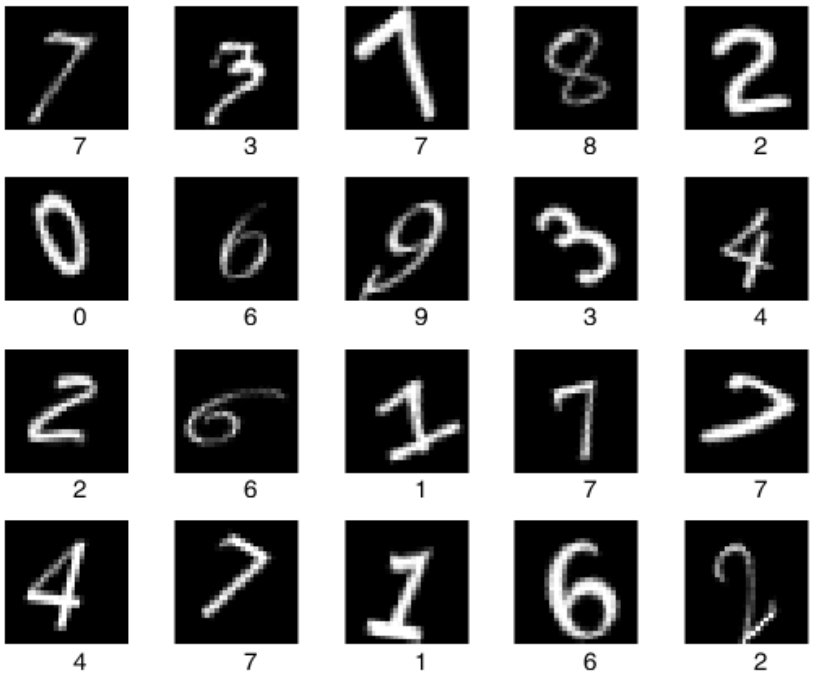
First, unzip the "digits_images.zip" file in the tutorial folder. This will give 10 folders with names from "0" to "9". Each folder contains 1,000 images of the same digit. For example, the content in folder "0" should look like the following:



Matlab provides a convenient built-in function `imageDatastore` to load all image data in a folder and infer the labels by their parent folder name:

```
datasetPath = '.\digits_images';
digitData = imageDatastore(digitDatasetPath,  ...
            'IncludeSubfolders',true,'LabelSource','foldernames');
```

After loading, randomly displaying 20 images and their labels will look like this:



3.2 Split into Training and Testing Dataset

Randomly splitting the dataset into training and testing set is a best practice to ensure the integrity of evaluation. The model does not see the testing data during training, so the test accuracy is more realistic and practical. In this case, we pick a ratio of 75%: 25%. Since we have 10,000 samples for 10 labels, the training set will contain 750 samples for each label:

```
trainNum = 750;  % for each label
[trainDigitData, testDigitData] = splitEachLabel(digitData, trainNum,'randomize');
```

3.3 Configure and Train the Network

In Matlab, the neural network model is defined as a list of layers from the input to the output. For simplicity, we will build a naïve CNN model containing only one convolutional layer and one fully connected layer. The following script shows how the model is constructed:

```
layers = [imageInputLayer([28 28 1])            % input size: 28*28*1
      convolution2dLayer(5,20)                  % output size: 24*24*20
      reluLayer
      maxPooling2dLayer(2,'Stride',2)           % output size: 12*12*20
      fullyConnectedLayer(10)                   % output size: 10
      softmaxLayer
      classificationLayer()];                   % output size: 1
```

The `imageInputLayer` tells Matlab this input type is a 3d matrix representing 28×28 grayscale images. Following the input, the convolutional layer function specifies 20 filters with a size of 5×5. The `reluLayer` is not an actual layer; it simply configures the convolutional layer's activation to be ReLU function. The pooling layer is used after convolution to subsample the image features. Here each 2×2 window on the image is subsampled to a scalar. This output matrix is flattened and fully connected to 10 neurons in the next layer. Each neuron in this layer corresponds to a label/digit. A softmax function is used to normalize the values across 10 neurons so that each output value represents a probability. The last `classificationLayer` picks the label with the highest probability score as the detection output.

Next step is to specify the training parameters using `trainingOptions` function:

```
options = trainingOptions('sgdm','MaxEpochs',15, 'InitialLearnRate',0.0001);
```

Here the optimizer '`sgdm`' means stochastic gradient descent with momentum, a variation of gradient descend algorithm. In this simple example, the choice of optimizer makes little difference to the training result. The '`InitialLearnRate`' specifies the gradient descend step size, whose practical choice is between 0.001 to 0.0001. This function also specifies the total number of epochs. An epoch means the training has gone through all the available data for one time. In practice, the number of epochs needed to reach convergence varies depending on the size of data, the configuration of the network, and training parameters. Therefore, this value often requires trial and error. In this study, a value of 15 fits the setting just fine.

With the training data, network configuration, and training parameters all set, the training process can start with this command:

```
convnet = trainNetwork(trainDigitData, layers, options);
```

The training will start with the specified optimizer to update the network parameters. Matlab provides a clean interface in the workspace to display the training process:

```
Training on single CPU.
Initializing input data normalization.
|=========================================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================================|
|       1 |           1 |       00:00:00 |        9.38% |      13.9089 |      1.0000e-04 |
|       1 |          50 |       00:00:04 |       56.25% |       2.7499 |      1.0000e-04 |
|       2 |         100 |       00:00:08 |       75.78% |       0.9251 |      1.0000e-04 |
|       3 |         150 |       00:00:14 |       80.47% |       0.9513 |      1.0000e-04 |
|       4 |         200 |       00:00:19 |       89.84% |       0.2437 |      1.0000e-04 |
|       5 |         250 |       00:00:23 |       87.50% |       0.4307 |      1.0000e-04 |
|       6 |         300 |       00:00:27 |       89.84% |       0.2872 |      1.0000e-04 |
|       7 |         350 |       00:00:30 |       95.31% |       0.2087 |      1.0000e-04 |
|       7 |         400 |       00:00:34 |       96.88% |       0.1312 |      1.0000e-04 |
|       8 |         450 |       00:00:38 |       93.75% |       0.2305 |      1.0000e-04 |
|       9 |         500 |       00:00:42 |       96.09% |       0.1327 |      1.0000e-04 |
|      10 |         550 |       00:00:46 |       97.66% |       0.0642 |      1.0000e-04 |
|      11 |         600 |       00:00:49 |       99.22% |       0.0453 |      1.0000e-04 |
|      12 |         650 |       00:00:55 |       98.44% |       0.0943 |      1.0000e-04 |
|      13 |         700 |       00:01:00 |       97.66% |       0.0863 |      1.0000e-04 |
|      13 |         750 |       00:01:04 |       97.66% |       0.1075 |      1.0000e-04 |
|      14 |         800 |       00:01:08 |       99.22% |       0.0475 |      1.0000e-04 |
|      15 |         850 |       00:01:13 |       98.44% |       0.0580 |      1.0000e-04 |
|      15 |         870 |       00:01:14 |       99.22% |       0.0367 |      1.0000e-04 |
|=========================================================================================|
```

If configured correctly, the accuracy will improve and the loss will decrease during the training process. At the end of the training, the accuracy struggles the increase, meaning the loss has converged, and the training should be stopped. We can now use this model to detect digits from images.

3.4 Evaluate on Testing Dataset

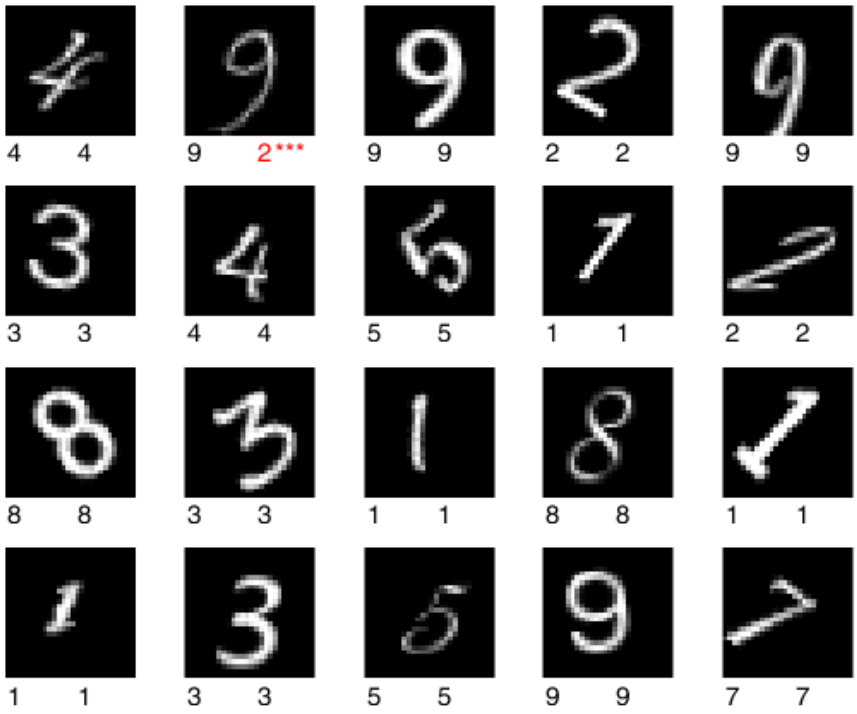
For overall evaluation of all 2500 test images, we can use this Matlab function to batch process and classify:

```
yTest = classify(convnet, testDigitData);
```

The result is a vector of 2500 predicted classes. It can be compared with the ground truths to obtain the accuracy score:

```
tTest = testDigitData.Labels;
accuracy = sum(yTest == tTest)/numel(tTest)
```

This should give a score of around 94.24%. The number may vary a little due to the randomness in the data splitting and model training process. To examine the result in detail, we randomly select 20 test images, as shown below. The number under each image indicates the ground truth label and model classification. There is only one image with digit 9 misclassified as 2. All the rest 19 images are all correctly recognized. This agrees with the test accuracy of ~94%.



At this point, we could conclude the model being effective at recognizing digits. However, there are rooms for improvement, as the testing accuracy is 5% lower than the training accuracy. The width and depth of the network can contribute to higher accuracy; more sophisticated optimizers can help accelerate the training process; and the original MNIST dataset is more inclusive than the one provided in this tutorial. The state-of-the-

art research has shown that CNN models can score more than 99% real-world test accuracy, achieving human-level performance.

4 CONCLUSION

This document and the MATLAB companion files provide a case study on an image classification problem: digits recognition. The theory of neural networks and CNN are briefly introduced. Then a simple CNN model is implemented with Matlab script. With a relatively simple configuration, the model can achieve high performance on the test dataset, demonstrating the capability of CNN. This implementation can be easily adapted to other image classification tasks with few changes in the code. More advanced CNN models can be found in the references.

5 REFERENCES

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[3] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556 [cs]*, Sep. 2014, Accessed: Apr. 09, 2018. [Online]. Available: http://arxiv.org/abs/1409.1556.

[4] C. Szegedy *et al.*, "Going Deeper with Convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.