University of Regina

Faculty of Engineering and Applied Science

ENIN 880 CA

Assignment #4

Behnam Moradi – 200 433 555

1 Introduction

Artificial neural networks are advancing rapidly, and researchers in this field are working hard to ensure that neural algorithms can learn and analyze problems like the human brain. In this report, we first introduce the Convolutional Neural Networks. These networks are particularly useful in the field of machine vision. Then we introduce the idea and programming algorithm behind these networks. Finally, using the TensorFlow library and the MINIST dataset, we implement these particular networks with the help of Python programming. The primary purpose of this exercise is to implement convolutional neural networks (CNNs) to detect handwritten digits in a picture.

2 Convolutional Neural Networks

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm that can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods, filters are hand-engineered, with enough training, ConvNets can learn these filters/characteristics.

A ConvNet is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better. Figure (1) presents the overall idea behind the CNNs.

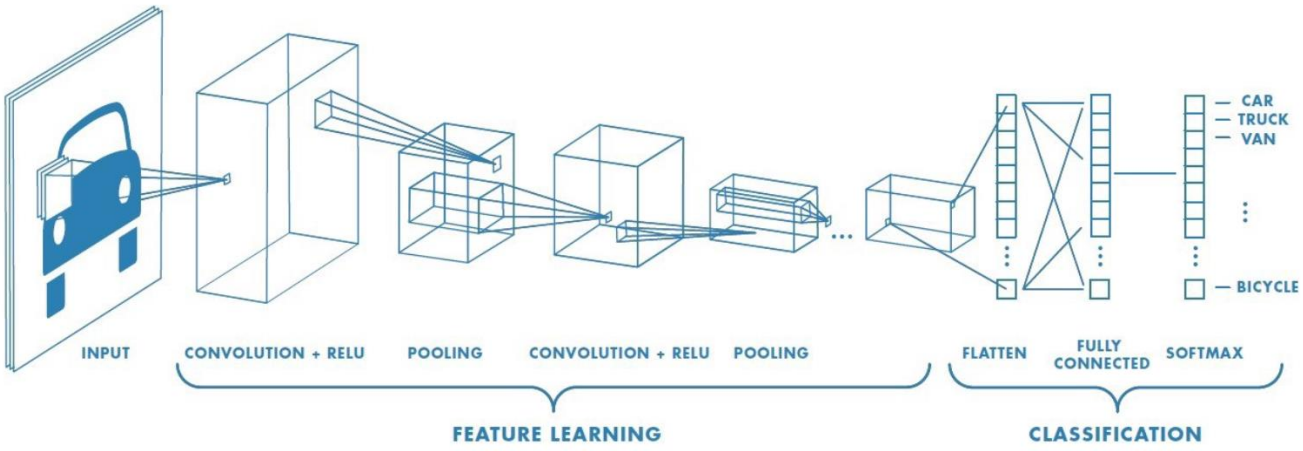


Figure 1. The big picture of Convolutional Neural Networks

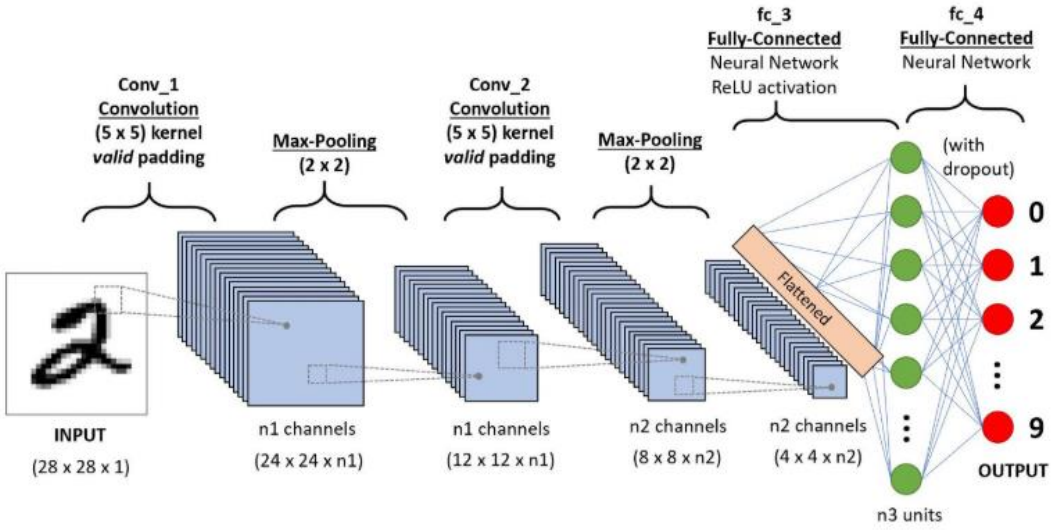


Figure 2 A CNN sequence to classify a handwritten digit

In the figure (3), the presented RGB image that has been separated by its three color channels of Red, Green, and Blue. One can imagine how computationally intensive things would get once the images reach dimensions, say 8K (7680×4320). The role of the ConvNet is to reduce the images into a form that is easier to process without losing features, which are critical for getting the right prediction. It is crucial when designing an architecture that is not only good at learning features but also scalable to massive datasets.

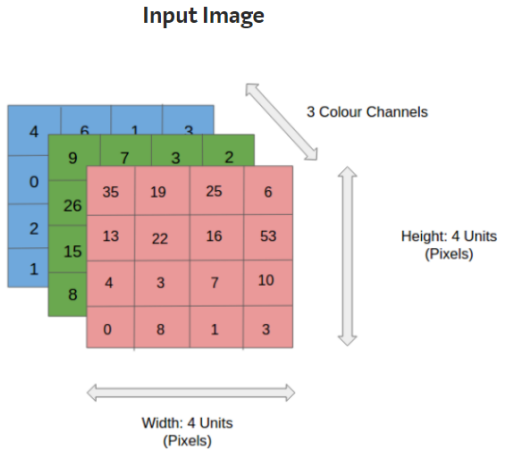


Figure 3. The mathematical model of an RGB image

2.1 CNN Building blocks

using Convolutional Neural Networks (CNNs) one is able to take advantage of the input image structure and define a network architecture in a sensible way. Unlike a standard neural network, layers of a CNN are arranged in a 3D volume in three dimensions: width, height, and depth (where depth refers to the third dimension of the volume, such as the number of channels in an image or the number of filters in a layer). To make this example more concrete, again consider the CIFAR-10 dataset: the input volume will have dimensions 32×32×3 (width, height, and depth, respectively). Neurons in subsequent layers will only be connected to a small region of the layer before it (rather than the fully-connected structure of a standard neural network) – we call this local connectivity which enables us to save a huge amount of parameters in our network. Finally, the output layer will be a 1×1×N volume which represents the image distilled into a single vector of class scores. In the case of CIFAR-10, given ten classes, N = 10, yielding a 1×1×10 volume.

2.2 CNN - Layer Structure and Strategy

A very general way to build a layer structure for a CNN is as follows:

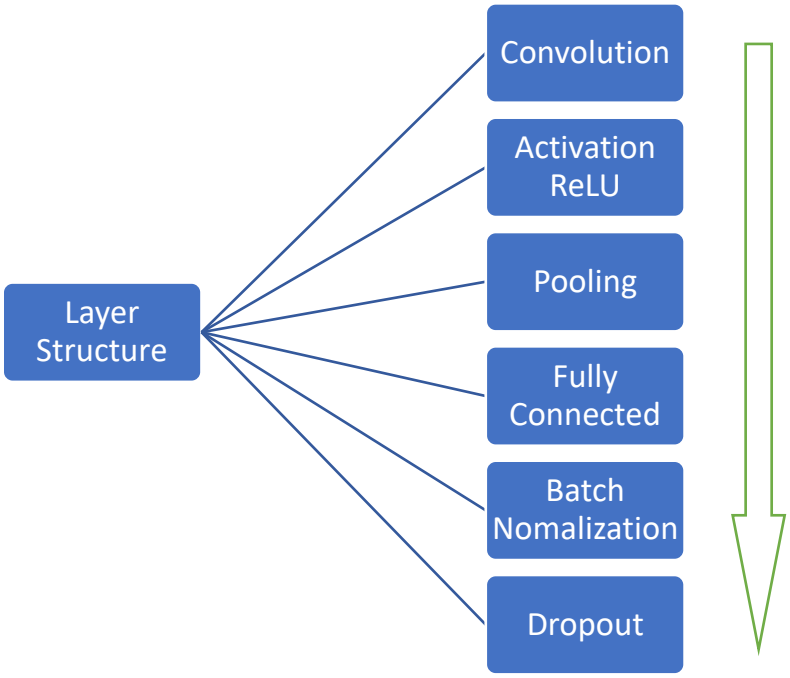


Figure 4. The general layer structure for Convolutional Neural Networks

Of these layer types, CONV and FC, (and to a lesser extent, BN) are the only layers that contain parameters that are learned during the training process. Convolution, Pooling, Activation ReLU, and Fully Connected are the most important when defining your actual network architecture.

2.2.1 Convolution Layers

The CONV layer is the core building block of a Convolutional Neural Network. The CONV layer parameters consist of a set of K learnable filters (i.e., "kernels"), where each filter has a width and a height, and are nearly always square. These filters are small (in terms of their spatial dimensions) but extend throughout the full depth of the volume.

For inputs to the CNN, the depth is the number of channels in the image (i.e., a depth of three when working with RGB images, one for each channel). For volumes deeper in the network, the depth will be the number of filters applied in the previous layer.

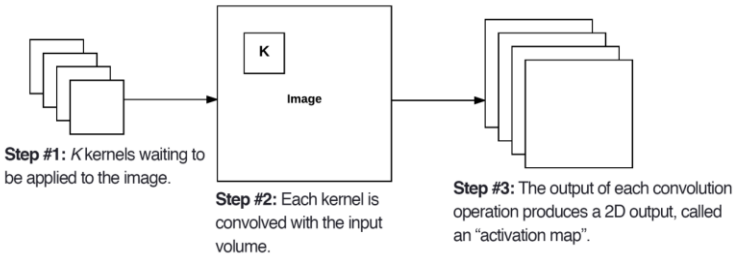


Figure 5. More simply, we can think of each of our K kernels sliding across the input region, computing an element-wise multiplication, summing, and then storing the output value in a 2- dimensional activation map

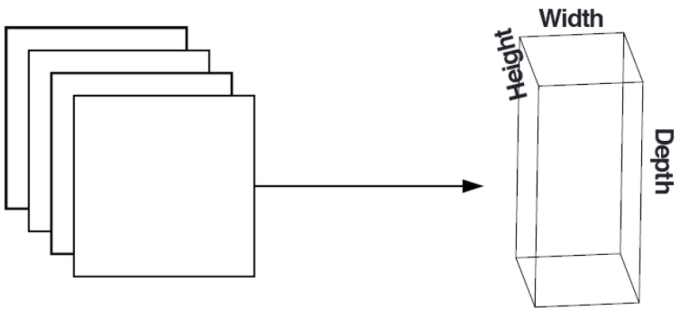


Figure 6. After obtaining the K activation maps, they are stacked together to form the input volume to the next layer in the network.

2.2.2 Activation Layer

After each CONV layer in a CNN, we apply a nonlinear activation function, such as ReLU, ELU, or any of the other Leaky ReLU variants. We typically denote activation layers as RELU in network diagrams as since ReLU activations are most commonly used, we may also simply state ACT – in either case, we are making it clear that an activation function is being applied inside the network architecture.

2.2.3 Pooling Layer

he primary function of the POOL layer is to progressively reduce the spatial size (i.e., width and height) of the input volume. Doing this allows us to reduce the amount of parameters and computation in the network – pooling also helps us control overfitting. POOL layers operate on each of the depth slices of an input independently using either the max or average function. Max pooling is typically done in the middle of the CNN architecture to reduce spatial size, whereas average pooling is normally used as the final layer of the network (e.x., GoogLeNet, SqueezeNet, ResNet) where we wish to avoid using FC layers entirely. The most common type of POOL layer is max pooling, although this trend is changing with the introduction of more exotic micro-architectures. Typically we'll use a pool size of 2×2. Technically, involving the pooling layer, it is easy to visualize the flow of the network in this manner:

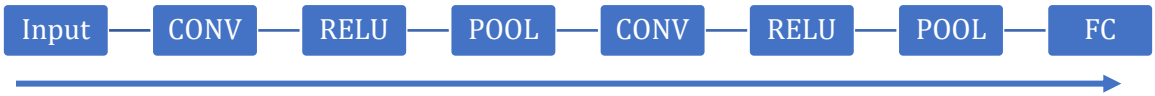


Figure 7. CNN Layer Structurer - Considering Pooling Layer

2.2.4 Fully-Connected Layer

Neurons in FC layers are fully-connected to all activations in the previous layer, as is the standard for feedforward neural networks. Layers are always placed at the end of the network (i.e., we don't apply a CONV layer, then an FC layer, followed by another CONV) layer. It's common to use one or two FC layers prior to applying the softmax classifier, as the following (simplified) architecture demonstrates:

Figure 8. CNN Layer Structurer - Considering two FC Layer

Here we apply two fully-connected layers before our (implied) softmax classifier which will compute our final output probabilities for each class.

2.2.5 Dropout Layer

The last layer type we are going to discuss is dropout. Dropout is actually a form of regularization that aims to help prevent overfitting by increasing testing accuracy, perhaps at the expense of training accuracy. For each mini-batch in our training set, dropout layers, with probability p, randomly disconnect inputs from the preceding layer to the next layer in the network architecture.

Figure (9) visualizes this concept where we randomly disconnect with probability p = 0:5 the connections between two FC layers for a given mini-batch. Again, notice how half of the connections are severed for this mini-batch. After the forward and backward pass are computed for the minibatch, we re-connect the dropped connections, and then sample another set of connections to drop

The reason we apply dropout is to reduce overfitting by explicitly altering the network architecture at training time. Randomly dropping connections ensures that no single node in the network is responsible for "activating" when presented with a given pattern. Instead, dropout ensures there are multiple, redundant nodes that will activate when presented with similar inputs – this in turn helps our model to generalize.

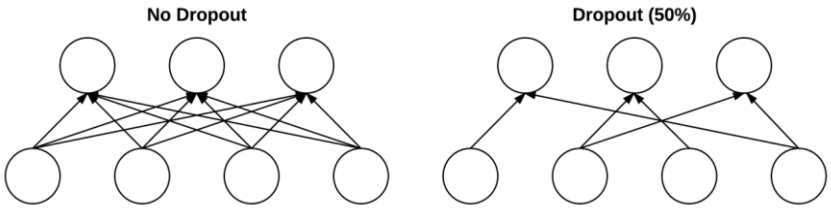
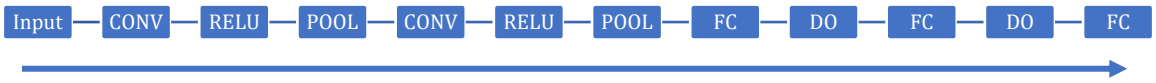


Figure 9.CNN – Dropout Layer Structure

It is most common to place dropout layers with p = 0:5 in-between FC layers of an architecture where the final FC layer is assumed to be our softmax classifier:



3 Recognizing Handwritten Digits

The LeNet architecture is straightforward and small (in terms of memory footprint), making it perfect for teaching the basics of CNNs. In this section, we'll start by reviewing the LeNet architecture and then implement the network using Keras. Finally, we'll evaluate LeNet on the MNIST dataset for handwritten digit recognition.

The combination of LeNet + MNIST is able to be easily run on the CPU, making it easy for beginners to take their first step in deep learning and CNNs. In many ways, LeNet + MNIST is the "Hello, World" equivalent of deep learning applied to image classification. The LeNet architecture consists of the following layers, using a pattern of CONV => ACT => POOL.

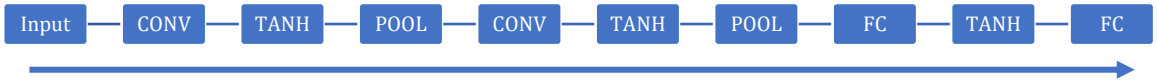


Figure 10. The LeNET layer Structure

The following table summarizes the parameters for the LeNet architecture. Our input layer takes an input image with 28 rows, 28 columns, and a single channel (grayscale) for depth (i.e., the dimensions of the images inside the MNIST dataset). We then learn 20 filters, each of which are 5×5. The CONV layer is followed by a ReLU activation followed by max pooling with a 2×2 size and 2×2 stride. The next block of the architecture follows the same pattern, this time learning 50 5×5 filters. It's common to see the number of CONV layers increase in deeper layers of the network as the actual spatial input dimensions decrease. We then have two FC layers. The first FC contains 500 hidden nodes followed by a ReLU activation. The final FC layer controls the number of output class labels (0-9; one for each of the possible ten digits). Finally, we apply a softmax activation to obtain the class probabilities.

Table 1. the LeNET Parameters

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$28 \times 28 \times 1$	
CONV	$28 \times 28 \times 20$	$5 \times 5, K = 20$
ACT	$28 \times 28 \times 20$	
POOL	$14 \times 14 \times 20$	2×2
CONV	$14 \times 14 \times 50$	$5 \times 5, K = 50$
ACT	$14 \times 14 \times 50$	
POOL	$7 \times 7 \times 50$	2×2
FC	500	
ACT	500	
FC	10	
SOFTMAX	10	

3.1 Implementing LeNET Using MNIST Dataset

In this section, we will be implementing the LeNET structure on MINIST dataset. First, we will train the network using 60,000 gray scale images of single handwritten digits (0-9). Then we will be evaluating the trained network using 290 gray scale handwritten images.

Before jumping into the python coding, let's define the folder structure as follows:

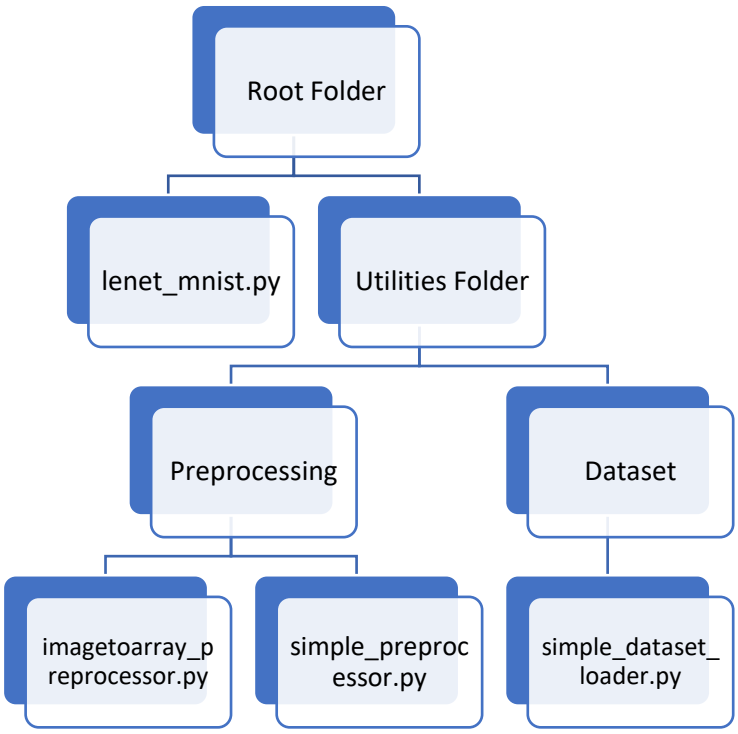
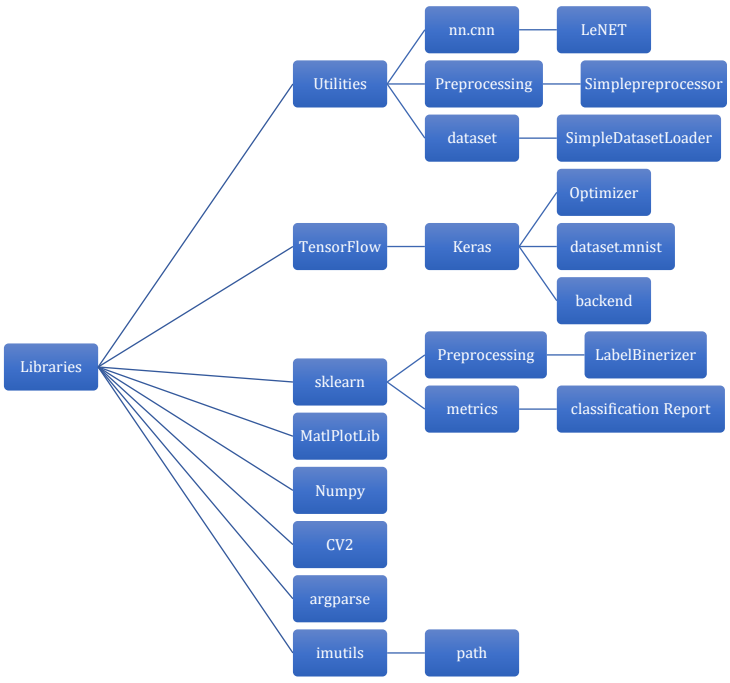


Table 2. Files and Folders Structure

3.1.1 Importing Libraries

In the section, as usual, we will be importing the essential libraries:



```
from utilities.nn.cnn import LeNet
from utilities.preprocessing import SimplePreprocessor
from utilities.datasets import SimpleDatasetLoader

from keras.optimizers import SGD
from tensorflow.keras.datasets import mnist
from tensorflow.keras import backend as K

from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report

import matplotlib.pyplot as plt
import numpy as np
import cv2
import argparse
from imutils import paths
```

3.2 Argument Parse

In this code block, we will set up an argument parser so that we can specify the location of evaluation data on the disk:

```
# Construct the argument parser and parse the arguments
# Use this command: python keras_cifar10.py --dataset ../datasets/animals
ap = argparse.ArgumentParser()
ap.add_argument('-d', '--dataset', required=True,
                help='Path to input dataset')
args = vars(ap.parse_args())
image_paths = list(paths.list_images(args['dataset']))
```

3.3 Loading training and evaluating data from disk

In this code block, we will be importing the following datasets:

1- Training Data: The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

2- Evaluating Data: 290 gray scaled images of handwritten digits

```
# grab the MNIST dataset (if this is your first time using this
# dataset then the 11MB download may take a minute)
print("[INFO] accessing MNIST...")
((trainData, trainLabels), (testData, testLabels)) = mnist.load_data()

# Loading Evaluation Data
sp = SimplePreprocessor(28, 28)
sdl = SimpleDatasetLoader(preprocessors=[sp])
(dataEv, labelsEv) = sdl.load(image_paths, verbose=500)
# if we are using "channels first" ordering, then reshape the
# design matrix such that the matrix is:
# num_samples x depth x rows x columns
if K.image_data_format() == "channels_first":
    trainData = trainData.reshape((trainData.shape[0], 1, 28, 28))
    testData = testData.reshape((testData.shape[0], 1, 28, 28))
    dataEv = dataEv.reshape((dataEv.shape[0], 1, 28, 28))

# otherwise, we are using "channels last" ordering, so the design
# matrix shape should be: num_samples x rows x columns x depth
else:
    trainData = trainData.reshape((trainData.shape[0], 28, 28, 1))
    testData = testData.reshape((testData.shape[0], 28, 28, 1))
    dataEv = dataEv.reshape((dataEv.shape[0], 28, 28, 1))
```

3.4 Image Scaling and Label Encoding

I this code block, we will be scaling the training dataset from (0, 255) to (0, 1). Also, we will be encoding Image labels to create a specific encoded map as follow:

Digit 0: [1 0 0 0 0 0 0 0 0 0]

Digit 1: [0 1 0 0 0 0 0 0 0 0]

.....

.....

Digit 9: [0 0 0 0 0 0 0 0 0 1]

```
50  # scale data to the range of [0, 1]
51  trainData = trainData.astype("float32") / 255.0
52  testData = testData.astype("float32") / 255.0
53  dataEv = dataEv.astype("float32") / 255.0
54
55  # convert the labels from integers to vectors
56  le = LabelBinarizer()
57  trainLabels = le.fit_transform(trainLabels)
58  testLabels = le.transform(testLabels)
59  labelsEv = le.transform(labelsEv)
60  labelsEv[:,9] = 1
```

3.5 Network Modeling and Optimizer Initialization

In this code blocks we will be defining and initializing:

1- The Stochastic Gradient Decent as the network optimizer
2- LeNet as our neural network structure

```
62  # initialize the optimizer and model
63  print("[INFO] compiling model...")
64  opt = SGD(lr=0.01)
65  model = LeNet.build(width=28, height=28, depth=1, classes=10)
66  model.compile(loss="categorical_crossentropy", optimizer=opt,
67                metrics=["accuracy"])
```

3.6 Network Training

In this code block, we will be training LeNet on MNIST for a total of 20 epochs using a mini-batch size of 128.

```
69  # train the network
70  print("[INFO] training network...")
71  H = model.fit(trainData, trainLabels,
72                validation_data=(dataEv, labelsEv), batch_size=128,
73                epochs=20, verbose=1)
```

3.7 Network Evaluating

In this code block we will be evaluating the Pre-trained LeNet using 290 28x28x1 single images of handwritten digits 0 -9.

```
# evaluate the network
print("[INFO] evaluating network...")
predictions = model.predict(dataEv, batch_size=128)
##print(classification_report(labelsEv.argmax(axis=1),
##                predictions.argmax(axis=1),
##                target_names=[str(x) for x in le.classes_]))
```

3.8 Plotting

Finally, we are implementing the final code block. In this code block we will be presenting some diagrams to investigate:

1- Training Loss
2- Training Accuracy
3- Evaluating Loss
4- Evaluating Accuracy

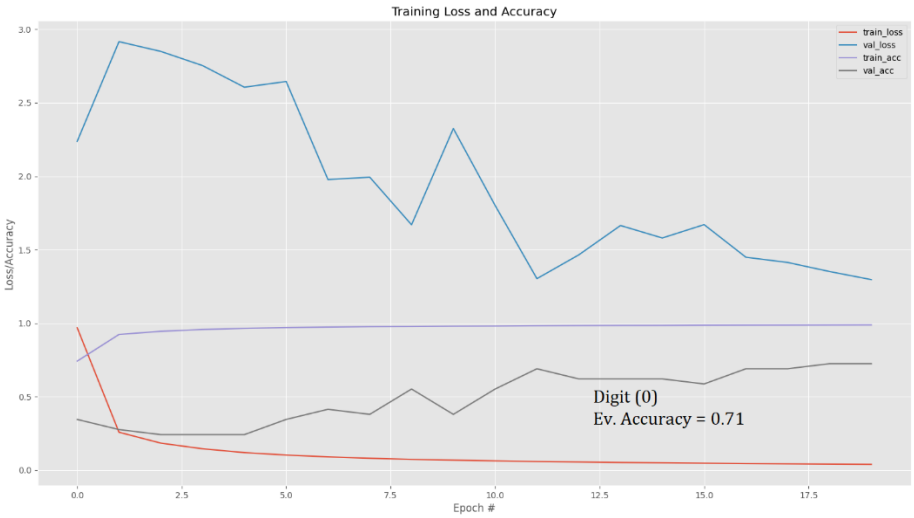
```
# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 20), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 20), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 20), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

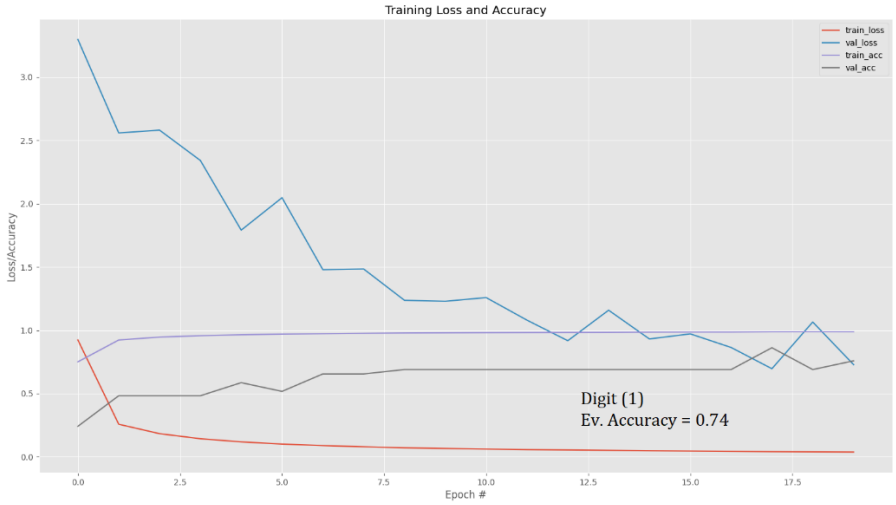
4 Results

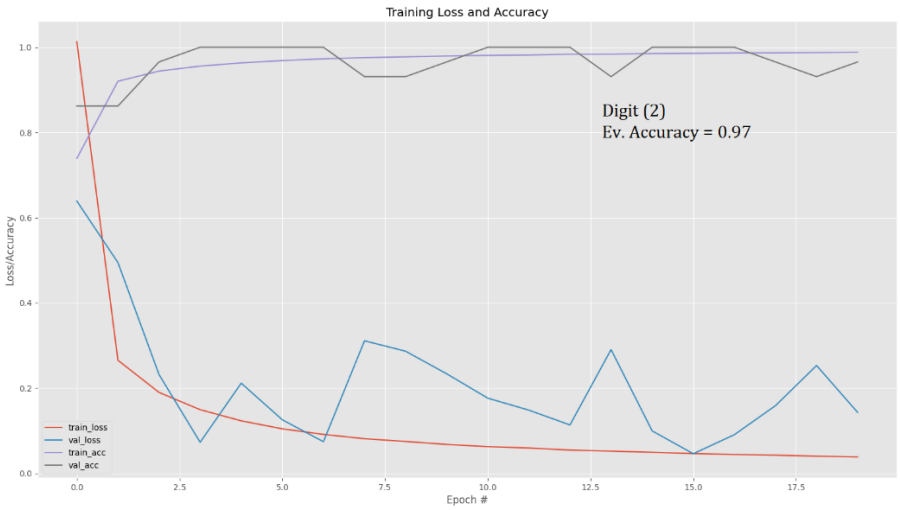
In this section we will be presenting the following results for training and evaluating process for each digit. We used 29 images per digits to evaluate the trained network.

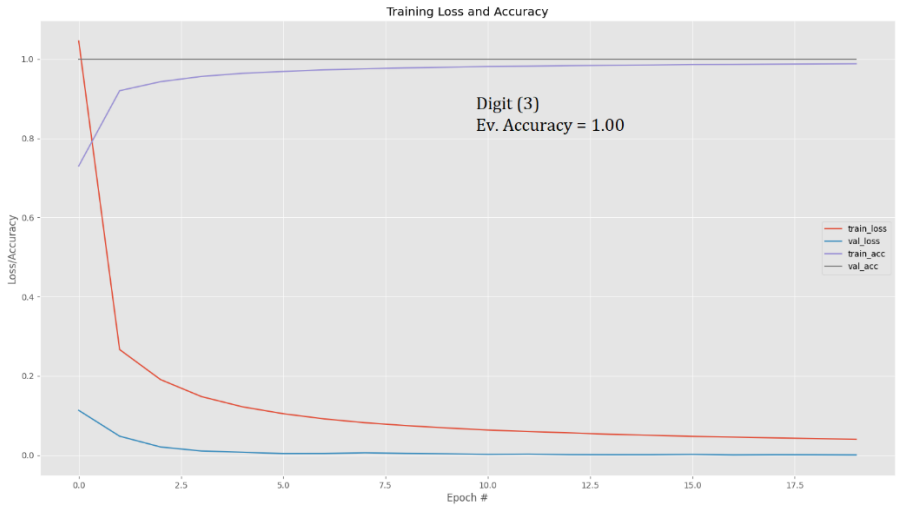
1- Training and Evaluating Loss
2- Training and Evaluating Accuracy

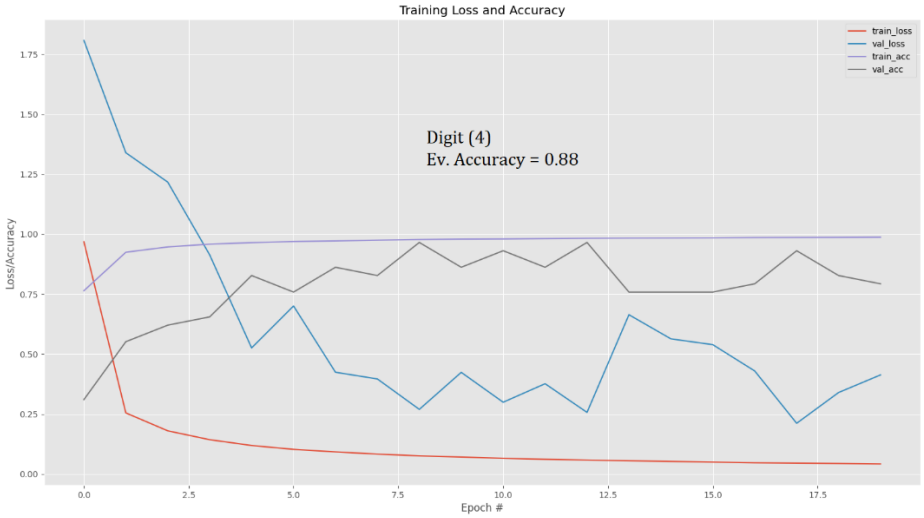
Digit	Evaluation Accuracy
0	0.71
1	0.74
2	0.97
3	1.00
4	0.88
5	0.58
6	0.21
7	0.51
8	0.95
9	0.26

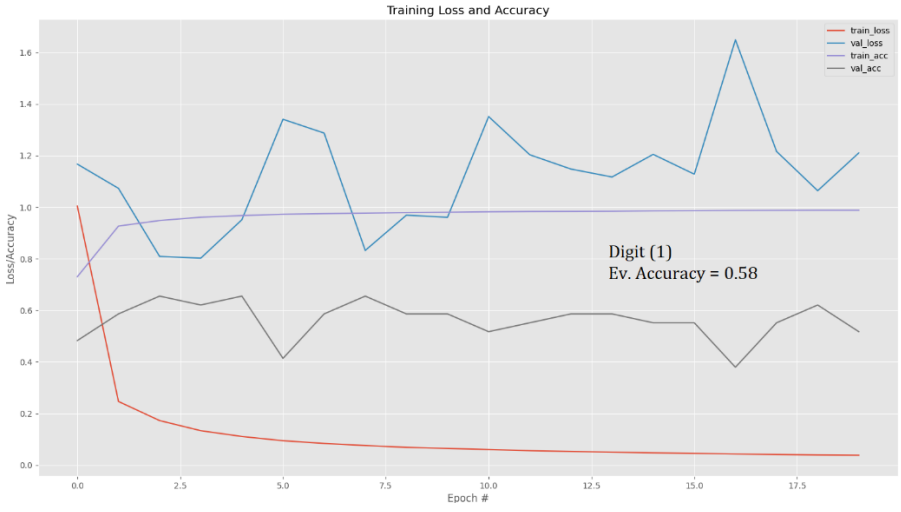

Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
Digit (0)
Ev. Accuracy = 0.71

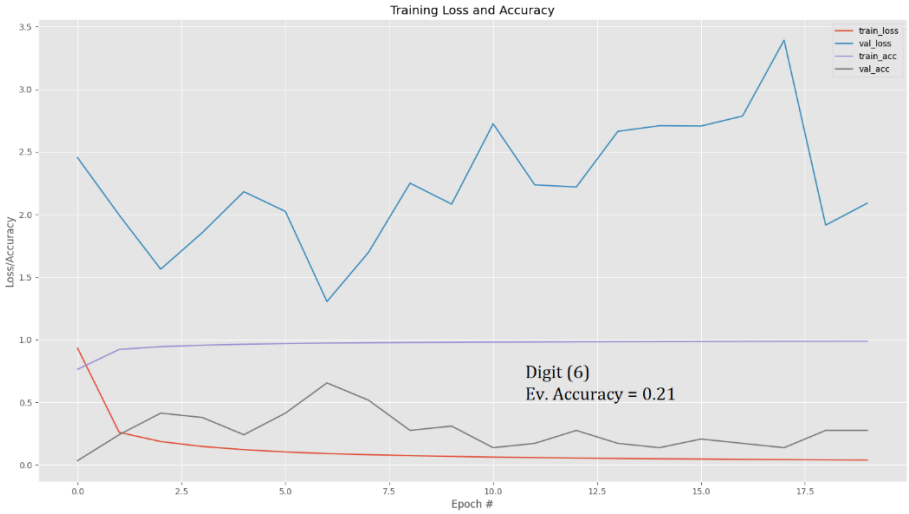

Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
Digit (1)
Ev. Accuracy = 0.74

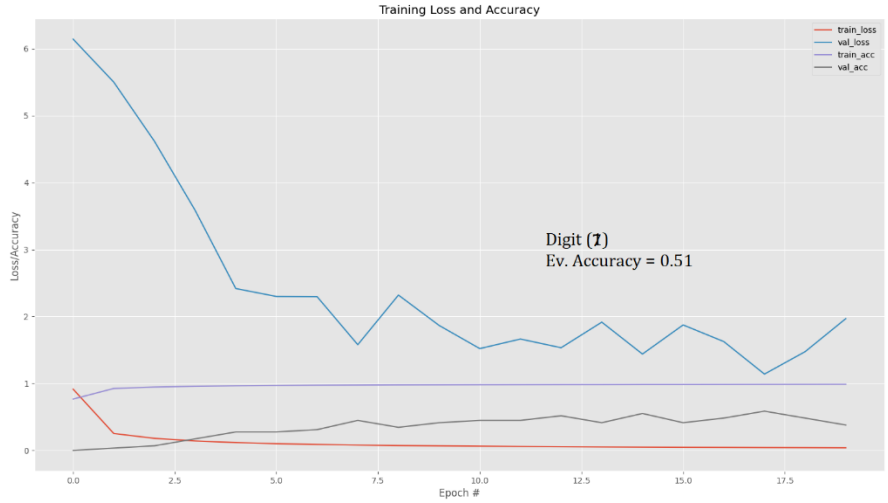

Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
Digit (2)
Ev. Accuracy = 0.97

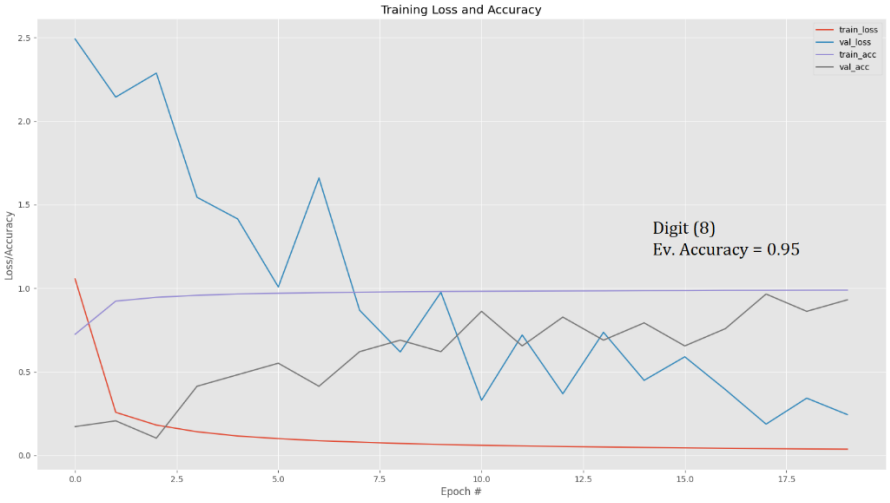

Training Loss and Accuracy
Digit (3)
Ev. Accuracy = 1.00
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #

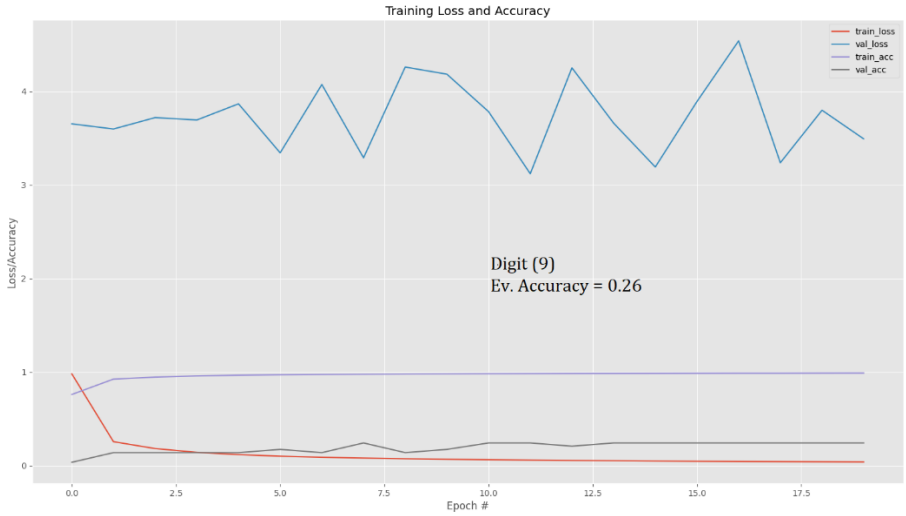

Training Loss and Accuracy
Digit (4)
Ev. Accuracy = 0.88
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #


Training Loss and Accuracy
Digit (1)
Ev. Accuracy = 0.58
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #


Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
Digit (6)
Ev. Accuracy = 0.21


Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
Digit (1)
Ev. Accuracy = 0.51


Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
Digit (8)
Ev. Accuracy = 0.95



The End	June 14, 2020