
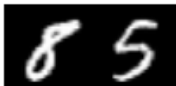



CMP684
Neural Network
Midterm Project Report

Osman Onur KUZUCU
N21131025

- Introduction:

In this report, we have worked on a paper which is “Performing Arithmetic Using a Neural Network Trained on Digit Permutation Pairs”. In this paper, researchers want to train the network for summation 2 numbers which have 1 digit. You can see the example in Picture 1 which is below.

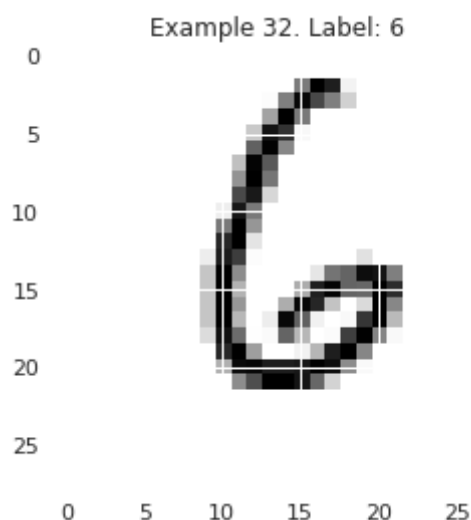
Image	Interpretation	Label
	$5 + 0$	5
	$8 + 5$	13
	$9 + 9$	18

Picture 1

Researchers used the mnist data from [Mnist Database](#). They use database's datas which are [train-images](#), [train-labels](#), [test-images](#) and [test-labels](#).

- About MNIST Database:

Mnist database is a database that contains images and labels. Images are handwritten one digit numbers. Labels are the value of the image. Example(Picture 2) mnist data is below.



Picture 2

Images' resolution are 28*28. You can see pixel size in Picture 2. Some information about picture 2 is below:

- Image's size is 28*28
 - Image's position is 32 in train-images data.
 - Image's label is 6.
- **Download the MNIST Database:**
We downloaded the mnist database from the source which mentioned above. Downloading data from source's code is below.

```
#1
#download source data from internet
from requests import get

def download_file(url, file_name):
    with open(file_name, "wb") as file:
        response = get(url)
        file.write(response.content)

download_file('http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz', 'train-images-idx3-ubyte.gz')
download_file('http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz', 'train-labels-idx1-ubyte.gz')
download_file('http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz', 't10k-images-idx3-ubyte.gz')
download_file('http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz', 't10k-labels-idx1-ubyte.gz')
```

Picture 3

- **Read Mnist Data from source data:**

The read_mnist function is below. read_mnist wants to get images and labels. if we give the images' path and labels' path, function gives features and labels. Features data is image data. Function read mnist data, after that return the 28*28 uint8 type numpy array. Labels is returned from function the value of 28*28 images.

```
#3
#read mnist data from source
def read_mnist(images_path: str, labels_path: str):
    with gzip.open(labels_path, 'rb') as labelsFile:
        labels = np.frombuffer(labelsFile.read(), dtype=np.uint8, offset=8)

    with gzip.open(images_path, 'rb') as imagesFile:
        length = len(labels)
        # Load flat 28x28 px images (784 px), and convert them to 28x28 px
        features = np.frombuffer(imagesFile.read(), dtype=np.uint8, offset=16) \
            .reshape(length, 784) \
            .reshape(length, 28, 28, 1)

    return features, labels
```

Picture 4

```
#4
#implementation test and train data
train = {}
test = {}

train['features'], train['labels'] = read_mnist('train-images-idx3-ubyte.gz', 'train-labels-idx1-ubyte.gz')
test['features'], test['labels'] = read_mnist('t10k-images-idx3-ubyte.gz', 't10k-labels-idx1-ubyte.gz')
```

Picture 5

In picture 5, train and test data is generated.

- train['features'] is a list of images for the train network.
- train[labels] is a list of labels for the train network.
- test['features'] is a list of images for a test network.
- test[labels] is a list of labels for a test network.

• Data Separation with using Label

Train images separated by using train labels. You can see the code in Picture 6. We use the numpy.where function for separating the images.

```
#6
#seperation of zeros, ones, ... , nines
resultOfZeros = np.where(train['labels'] == 0)
print('len(resultOfZeros): ', len(resultOfZeros[0]))
resultOfOnes = np.where(train['labels'] == 1)
print('len(resultOfOnes): ', len(resultOfOnes[0]))
resultOfTwos = np.where(train['labels'] == 2)
print('len(resultOfTwos): ', len(resultOfTwos[0]))
resultOfThrees = np.where(train['labels'] == 3)
print('len(resultOfThrees): ', len(resultOfThrees[0]))
resultOfFours = np.where(train['labels'] == 4)
print('len(resultOfFours): ', len(resultOfFours[0]))
resultOfFives = np.where(train['labels'] == 5)
print('len(resultOfFives): ', len(resultOfFives[0]))
resultOfSixes = np.where(train['labels'] == 6)
print('len(resultOfSixes): ', len(resultOfSixes[0]))
resultOfSevens = np.where(train['labels'] == 7)
print('len(resultOfSevens): ', len(resultOfSevens[0]))
resultOfEights = np.where(train['labels'] == 8)
print('len(resultOfEights): ', len(resultOfEights[0]))
resultOfNines = np.where(train['labels'] == 9)
print('len(resultOfNines): ', len(resultOfNines[0]))

len(resultOfZeros): 5923
len(resultOfOnes): 6742
len(resultOfTwos): 5958
len(resultOfThrees): 6131
len(resultOfFours): 5842
len(resultOfFives): 5421
len(resultOfSixes): 5918
len(resultOfSevens): 6265
len(resultOfEights): 5851
len(resultOfNines): 5949
```

Picture 6

- Display 2 Images:

We want to display 2 images horizontally combined. So we use the `display_image2` function which is below.

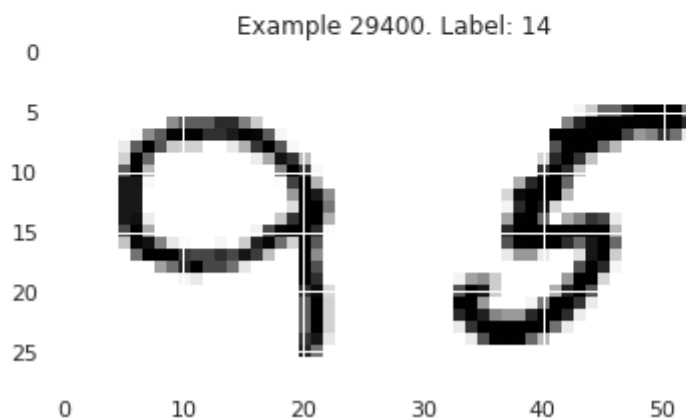
```
#7
#display image which combined 2 image in horizontally and labeled
def display_image2(position,position2):  ##combined 2 array in numpy and display
    image = train['features'][position].squeeze()
    image2 = train['features'][position2].squeeze()
    print(type(image[0][0]))
    print(type(train['labels'][position]))
    image3 = (np.hstack((image, image2)))
    #plt.title('Example %d. Label: %d' % (position, train['labels'][position]))
    print(train['labels'][position]*10+train['labels'][position2])
    plt.title('Example %d. Label: %d' % (position, train['labels'][position]+train['labels'][position2]))
    plt.imshow(image3, cmap=plt.cm.gray_r)
```

Picture 7

Example output of the `display_image2` is below.

```
#8
#try display_image2()
display_image2(29400,22329)
```

```
<class 'numpy.uint8'>
<class 'numpy.uint8'>
95
```



Picture 8

In the example picture, we can see the image size update. The new image size is 28*56*1. You can see the 95 in this picture. 95 is the label combination of two images in 2 digit number pairs. but the research of this report wants to summation of the two numbers. Because of that you can see the label value which is 14.

- Combine 2 numbers' image function:

We use combine2Images function for combine 2 images and return the combined image and label. You can see the function in Picture 9 which is below.

```
#combination of 2 images and labeled
def combine2Images(firstImagePos,secondImagePos):    ##combined 2 array in numpy and display

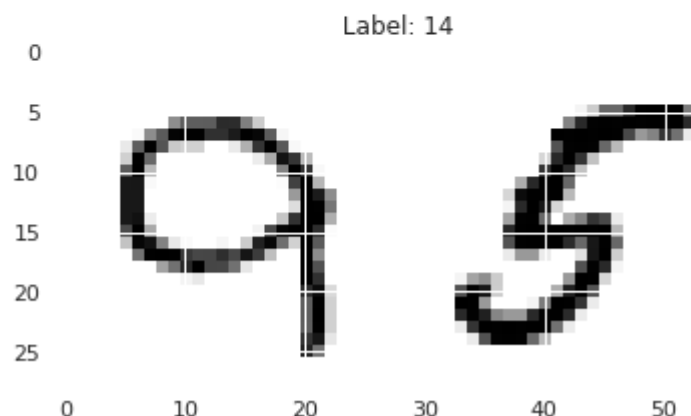
    image = train['features'][firstImagePos].squeeze()
    image2 = train['features'][secondImagePos].squeeze()
    combinedImage = (np.hstack((image, image2)))
    combinedLabel = train['labels'][firstImagePos]+train['labels'][secondImagePos]
    return combinedImage, combinedLabel
```

Picture 9

combine2Images is called in example which is below.

```
#10
#try combine2Images()
combinedImageWork, combinedLabelWork = combine2Images(29400,22329)
plt.title(' Label: %d' % ( combinedLabelWork))
plt.imshow(combinedImageWork, cmap=plt.cm.gray_r)
print(combinedImageWork.shape)
```

(28, 56)



Picture 10

- Generate 100000 images and labels

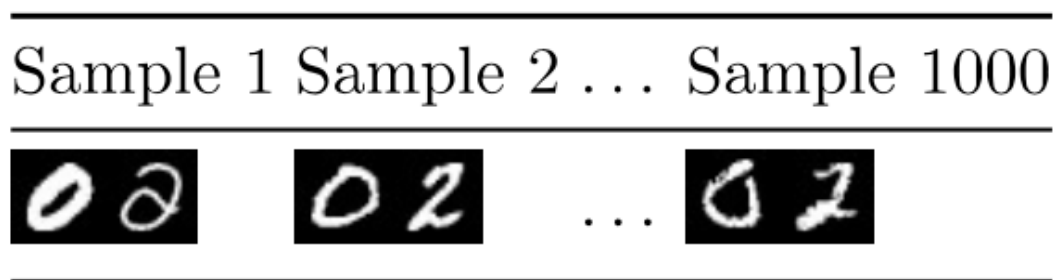
In this part we generate 100000 images and labels. We combine 2 images(2*(28*28)) to 1 image(1*(28*56)). We summate 2 images labeled to 1 image. We collected 100000 images in the foo array. We collected 100000 labels in fooLabel. You can see Picture 12 and Picture 13 below.

We generate 1000 pairs of each digit permutations. You can easily understand the example below.

First digit : 0

Second digit: 2

We want to generate 1000 unique combined images. Image size is 28*56. Image label is 2(0+2). You can see the example picture below. The picture is taken from a research paper.



Picture 11

```

i=0    #first digit of data
j=0    #second digit of data
k=0    #for 1000 sample
#foo = np.zeros((100,1000,2),dtype=np.ndarray)
foo = np.zeros((100000,28,56),dtype=np.uint8)
fooLabel = np.zeros((100000),dtype=np.uint8)
trainDataImage = np.zeros((90000,28,56),dtype=np.uint8)
trainDataLabel = np.zeros((90000),dtype=np.uint8)
testDataImage = np.zeros((10000,28,56),dtype=np.uint8)
testDataLabel = np.zeros((10000),dtype=np.uint8)
while(i<10):#first digit of pictures and labels
    #data selection for first digit
    if(i==0):
        dataSource=resultOfZeros
    elif(i==1):
        dataSource=resultOfOnes
    elif(i==2):
        dataSource=resultOfTwos
    elif(i==3):
        dataSource=resultOfThrees
    elif(i==4):
        dataSource=resultOfFours
    elif(i==5):
        dataSource=resultOfFives
    elif(i==6):
        dataSource=resultOfSixes
    elif(i==7):
        dataSource=resultOfSevens
    elif(i==8):
        dataSource=resultOfEights
    elif(i==9):
        dataSource=resultOfNines
    else:
        print("Error occured when data source initialize in i")

```

Picture 12


```

while(j<10):#second digit of pictures and labels
    #data selection for second digit
    if(j==0):
        dataSource2=resultOfZeros
    elif(j==1):
        dataSource2=resultOfOnes
    elif(j==2):
        dataSource2=resultOfTwos
    elif(j==3):
        dataSource2=resultOfThrees
    elif(j==4):
        dataSource2=resultOfFours
    elif(j==5):
        dataSource2=resultOfFives
    elif(j==6):
        dataSource2=resultOfSixes
    elif(j==7):
        dataSource2=resultOfSevens
    elif(j==8):
        dataSource2=resultOfEights
    elif(j==9):
        dataSource2=resultOfNines
    else:
        print("Error occured when data source initialize in j")

while(k<1000): #combined pictures and assign an array
    firstPosition = dataSource[0][random.randint(0, len(dataSource[0])-1)]
    secondPosition = dataSource2[0][random.randint(0, len(dataSource2[0])-1)]
    combinedImage, combinedLabel = combine2Images(firstPosition,secondPosition)
    foo[i*10000+j*1000+k] = combinedImage
    fooLabel[i*10000+j*1000+k] = combinedLabel
    k=k+1
k=0
j=j+1
j=0
i=i+1

```

Picture 13

- Generate Random 10 numbers between 0-99

We generated 10 numbers between 0-99 to separate train and test data. In a research paper, researchers use 90000 images and labels to train the network. 10000 images and labels are used to test network.

```

#13
#generate ten number for randnums
import random
randnums = random.sample(range(0,100), 10)
testIndex=0
trainIndex=0
print(randnums)
randnums = sorted(randnums, reverse=False)
print(randnums)

[41, 69, 63, 90, 83, 91, 17, 26, 60, 96]
[17, 26, 41, 60, 63, 69, 83, 90, 91, 96]

```

Picture 14

In Picture 14, we generated ten numbers to separate the test data. You can see an example about separating the data.

```

foo[16000-16999]->trainDataImage[16000-16999]
foo[17000-17999] -> testDataImage[0-999]
foo[18000-18999]->trainDataImage[17000-17999]

```

```

fooLabel[16000-16999]->trainDataLabel[16000-16999]
fooLabel[17000-17999] -> testDataLabel[0-999] 0
fooLabel[18000-18999]->trainDataLabel[17000-17999]

```

- Separate the train and test data:

We use 10 numbers which were generated randomly in the previous section, to separate the train and test data. You can see the code block in Picture 13 which is below.

```

#14
#separate foo array to trainDataImage,trainDataLabel and testl
randomIndex=0
k=0
fooIndex=0
trainIndex=0
testIndex=0
l=0
trainDataImage = np.zeros((90000,28,56),dtype=np.uint8)
trainDataLabel = np.zeros((90000),dtype=np.uint8)
testDataImage = np.zeros((10000,28,56),dtype=np.uint8)
testDataLabel = np.zeros((10000),dtype=np.uint8)

while(l<100000):
    if(k<10):
        print(str(randnums[k]*1000))
        if(randnums[k]*1000 <= fooIndex < (randnums[k]+1)*1000):
            testDataImage[testIndex]=foo[fooIndex]
            testDataLabel[testIndex]=fooLabel[fooIndex]
            testIndex=testIndex+1
            fooIndex=fooIndex+1
        else:
            trainDataImage[trainIndex]=foo[fooIndex]
            trainDataLabel[trainIndex]=fooLabel[fooIndex]
            trainIndex=trainIndex+1
            fooIndex=fooIndex+1
        if((randnums[k]+1)*1000==fooIndex):
            k=k+1
        else:
            trainDataImage[trainIndex]=foo[fooIndex]
            trainDataLabel[trainIndex]=fooLabel[fooIndex]
            trainIndex=trainIndex+1
            fooIndex=fooIndex+1
    #print(str(l)+ " :l+ k="+str(k) +" fooIndex: "+str(fooIndex))
    l=l+1

```

Picture 15

- Set the network parameters:

We can use the `modelTrain()` function to set the network parameters. Before setting the network parameters, we change the data type `uint8` to `float32`. After this operation, we work on a grayscale image.

```
#14
#from https://www.machinecurve.com/index.php/2020/02/18/how-to-use-k-fold-cross-validation-with-keras/

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np

# change type to uint8 to float32
trainDataImage = trainDataImage.astype('float32')
testDataImage = testDataImage.astype('float32')

#image to gray scale
trainDataImage /= 255
testDataImage /= 255

def modelTrain():
    # Define the K-fold Cross Validator
    #10-fold Cross Calidator
    kfold = KFold(n_splits=10, shuffle=True)

    # K-fold Cross Validation model evaluation
    fold_no = 1
    for train, test in kfold.split(trainDataImage, trainDataLabel):
        # Define the model architecture
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28,56,1)))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Flatten())
        model.add(Dense(256, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(1))
    return model
```

Picture 16

We use like Lenet-5 type network. We add the last layer with 1 neuron , instead of 10 neuron softmax layer. We use 10-fold cross validation for training the network.

```
def modelTrain():

    # Define the K-fold Cross Validation
    #10-fold Cross Calidator
    kfold = KFold(n_splits=1, shuffle=True)

    # K-fold Cross Validation model evaluation
    fold_no = 10
    for train, test in kfold.split(trainDataImage, trainDataLabel):
        # Define the model architecture
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=(28,56,1)))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
```

```

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
#one-neuron-layer for trained as a regression problem
model.add(Dense(1))
return model

```

- **Model Training and Parameter Setting**

We use the Adadelata optimizer. We use the mean squared error loss function. You can see in Picture 17

```

#15
#start working on model

#return model
model = modelTrain()

#optimizer ada's parameter is set
ada = tf.keras.optimizers.Adadelta(learning_rate=0.1, rho=0.95, epsilon=1e-08)

#model compile parameters is set
model.compile(optimizer=ada,
              loss=tf.keras.losses.mean_squared_error,
              metrics=['accuracy'])

#fitting model with trainData, testData and some parameters
history = model.fit(trainDataImage, trainDataLabel,
                    epochs=10,
                    batch_size=128,
                    validation_data=(testDataImage, testDataLabel))

```

```

Epoch 1/10
704/704 [=====] - 155s 219ms/step - loss: 9.1138 - accuracy: 0.0223 - val_loss: 7.6026 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 153s 218ms/step - loss: 3.8125 - accuracy: 0.0224 - val_loss: 3.7694 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 153s 218ms/step - loss: 3.2585 - accuracy: 0.0229 - val_loss: 2.9306 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 153s 218ms/step - loss: 2.8587 - accuracy: 0.0229 - val_loss: 2.8215 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 153s 218ms/step - loss: 2.5645 - accuracy: 0.0234 - val_loss: 2.7155 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 153s 217ms/step - loss: 2.3266 - accuracy: 0.0236 - val_loss: 2.4588 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 153s 217ms/step - loss: 2.1476 - accuracy: 0.0240 - val_loss: 4.0182 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 153s 218ms/step - loss: 2.0117 - accuracy: 0.0239 - val_loss: 3.2911 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 153s 217ms/step - loss: 1.8753 - accuracy: 0.0242 - val_loss: 2.1390 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 153s 217ms/step - loss: 1.7685 - accuracy: 0.0242 - val_loss: 2.0820 - val_accuracy: 0.0000e+00

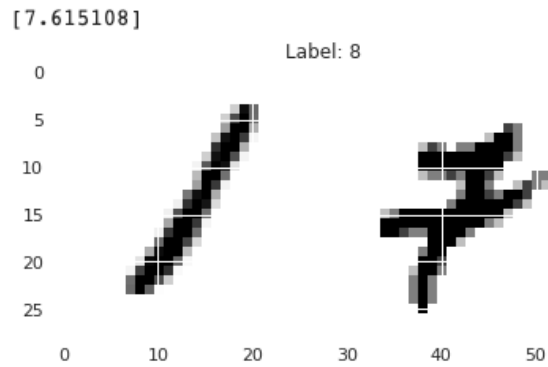
```

Picture 17

- **Check Some Prediction with Using Network**

- There are 10 example prediction results.
- ☐ First example:
 1. You can see Picture 18 which is below.
 2. Label is 8, but neural networks perform 7.615108

```
#16
#predictions checking
predictions = model.predict(testDataImage)
print(predictions[755])
plt.title('Label: %d' % testDataLabel[755])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[755].squeeze(), cmap=plt.cm.binary)
plt.show()
```

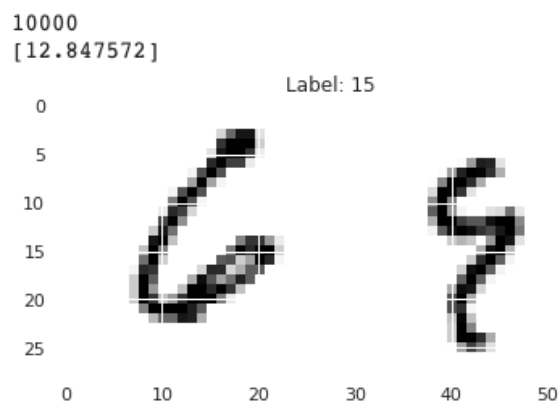


Picture 18

☐ Second Example:

1. You can see Picture 19 which is below.
2. Label is 15, but neural networks perform 12.847572

```
#17
#predictions checking
print(len(predictions))
print(predictions[5689])
plt.title('Label: %d' % testDataLabel[5689])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[5689].squeeze(), cmap=plt.cm.binary)
plt.show()
```



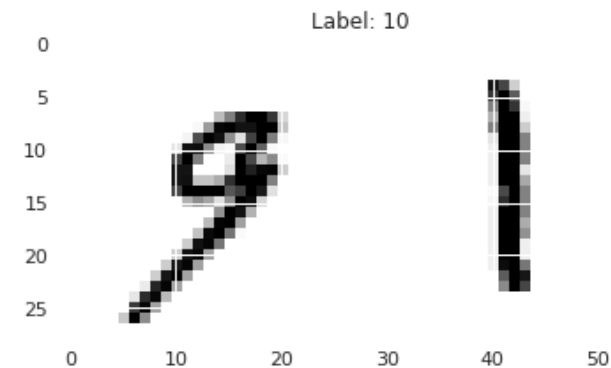
Picture 19

☐ Third Example:

1. You can see Picture 20 which is below.
2. Label is 10, but neural networks perform 9.415844

```
print(predictions[8500])
plt.title('Label: %d' % testDataLabel[8500])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[8500].squeeze(),cmap=plt.cm.binary)
plt.show()
```

[9.415844]



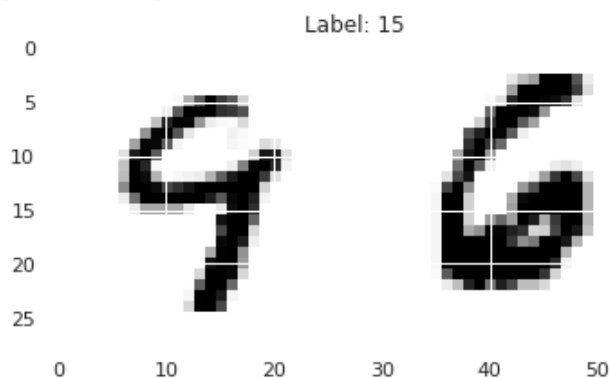
Picture 20

☐ Fourth Example

1. You can see Picture 21 which is below.
2. Label is 15, but neural networks perform 14.481272

```
print(predictions[9500])
plt.title('Label: %d' % testDataLabel[9500])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[9500].squeeze(),cmap=plt.cm.binary)
plt.show()
```

[14.481272]



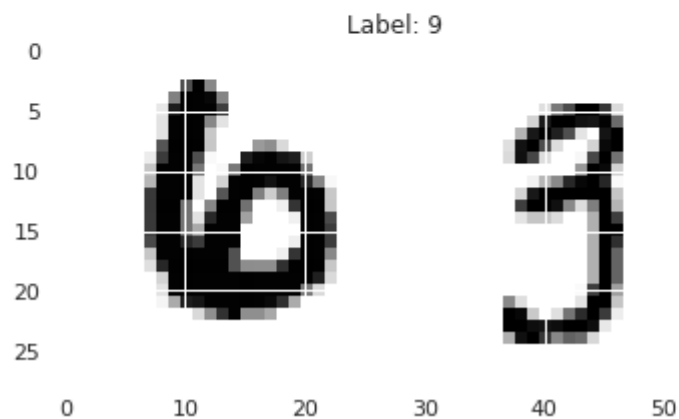
Picture 21

☐ Fifth Example

1. You can see Picture 22 which is below.
2. Label is 9, but neural networks perform 9.684742

```
print(predictions[4444])
plt.title('Label: %d' % testDataLabel[4444])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[4444].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[9.684742]



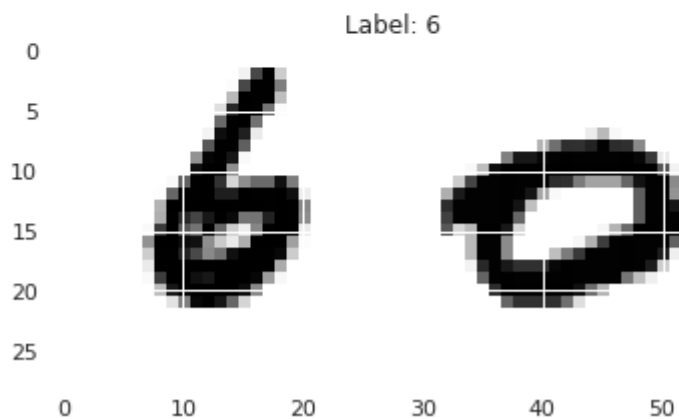
Picture 22

☐ Sixth Example

1. You can see Picture 23 which is below.
2. Label is 6, but neural networks perform 8.625708

```
print(predictions[3333])
plt.title('Label: %d' % testDataLabel[3333])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[3333].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[8.625708]

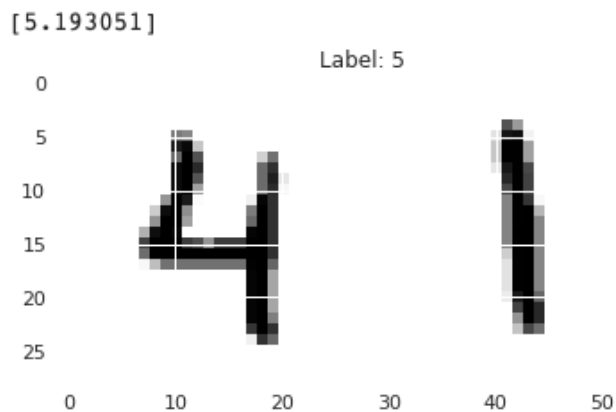


Picture 23

☐ Seventh Example

1. You can see Picture 24 which is below.
2. Label is 5, but neural networks perform 5.193051

```
print(predictions[2222])
plt.title('Label: %d' % testDataLabel[2222])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[2222].squeeze(), cmap=plt.cm.binary)
plt.show()
```

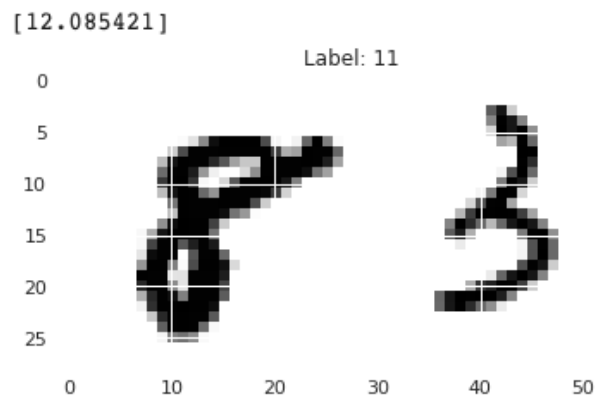


Picture 24

☐ Eighth Example

1. You can see Picture 25 which is below.
2. Label is 11, but neural networks perform 12.085421

```
print(predictions[6666])
plt.title('Label: %d' % testDataLabel[6666])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[6666].squeeze(), cmap=plt.cm.binary)
plt.show()
```



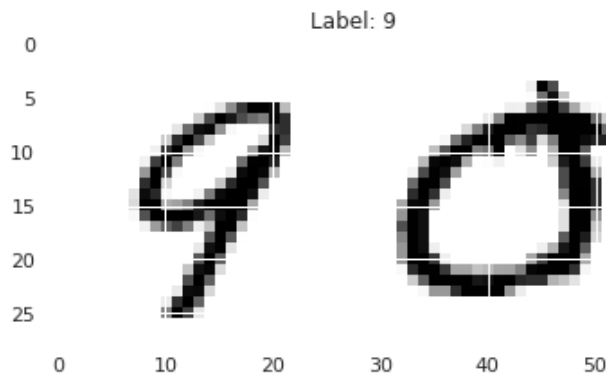
Picture 25

☐ Ninth Example

1. You can see Picture 26 which is below.
2. Label is 9, but neural networks perform 9.595124

```
print(predictions[7777])
plt.title('Label: %d' % testDataLabel[7777])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[7777].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[9.595124]



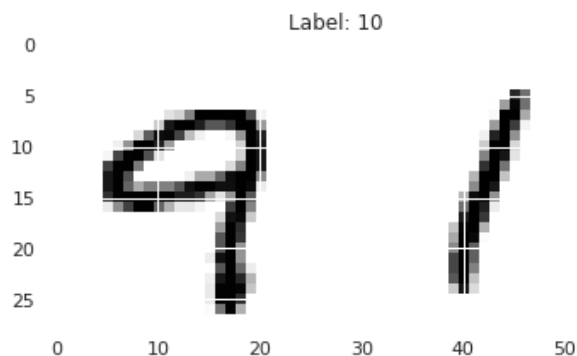
Picture 26

☐ Tenth Example

1. You can see Picture 27 which is below.
2. Label is 10, but neural networks perform 9.9429

```
print(predictions[8888])
plt.title('Label: %d' % testDataLabel[8888])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[8888].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[9.9429]



Picture 27

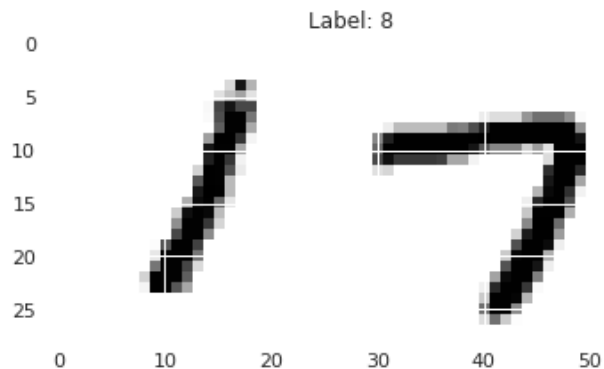
- Check some prediction from same label with different digit pairs

- ☐ First Example (Label:8)

1. You can see Picture 28 which is below.
2. Label is 8, but neural networks perform 7.9031167

```
print(predictions[488])
plt.title('Label: %d' % testDataLabel[488])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[488].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[7.9031167]



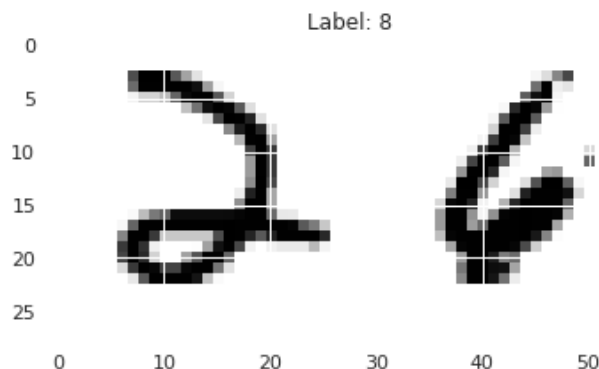
Picture 28

- ☐ Second Example (Label:8)

1. You can see Picture 29 which is below.
2. Label is 8, but neural networks perform 7.265822

```
print(predictions[1245])
plt.title('Label: %d' % testDataLabel[1245])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[1245].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[7.265822]



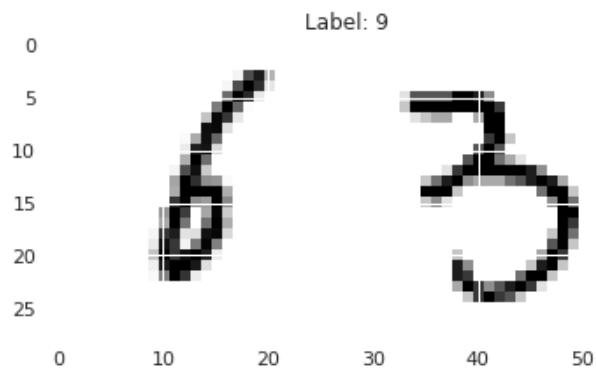
Picture 29

☐ Third Example (Label: 9)

1. You can see Picture 30 which is below.
2. Label is 9, but neural networks perform 8.191742

```
print(predictions[4212])
plt.title('Label: %d' % testDataLabel[4212])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[4212].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[8.191742]



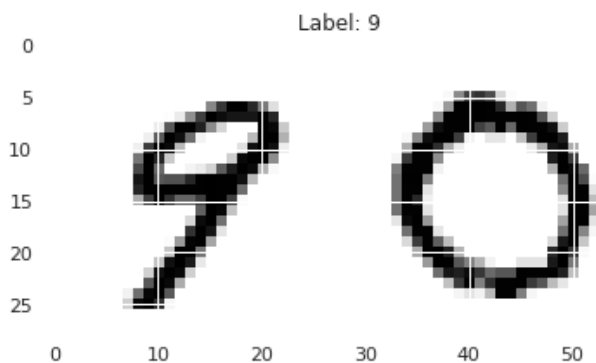
Picture 30

☐ Fourth example (Label: 9)

1. You can see Picture 31 which is below.
2. Label is 9, but neural networks perform 9.537987

```
print(predictions[7648])
plt.title('Label: %d' % testDataLabel[7648])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[7648].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[9.537987]



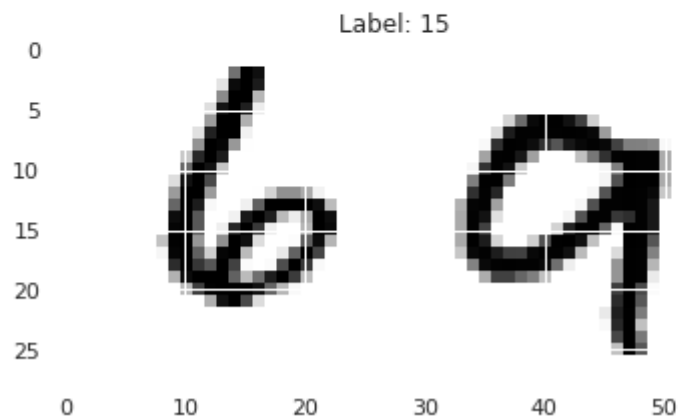
Picture 31

☐ Fifth Example(Label: 15)

1. You can see Picture 32 which is below.
2. Label is 15, but neural networks perform 14.675799

```
print(predictions[5421])
plt.title('Label: %d' % testDataLabel[5421])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[5421].squeeze(),cmap=plt.cm.binary)
plt.show()
```

[14.675799]



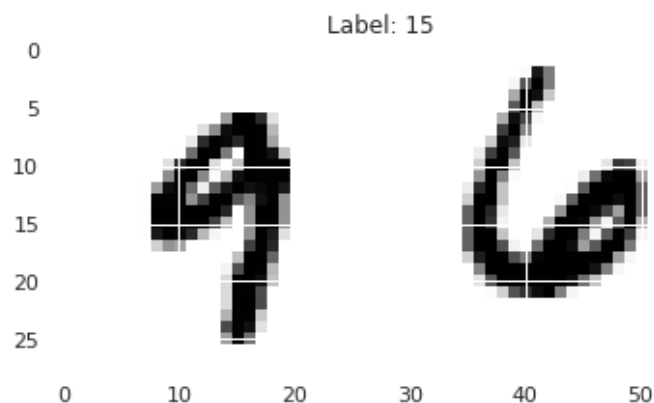
Picture 32

☐ Sixth Example (Label: 15)

1. You can see Picture 33 which is below.
2. Label is 15, but neural networks perform 14.675799

```
print(predictions[9845])
plt.title('Label: %d' % testDataLabel[9845])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[9845].squeeze(),cmap=plt.cm.binary)
plt.show()
```

[14.241512]



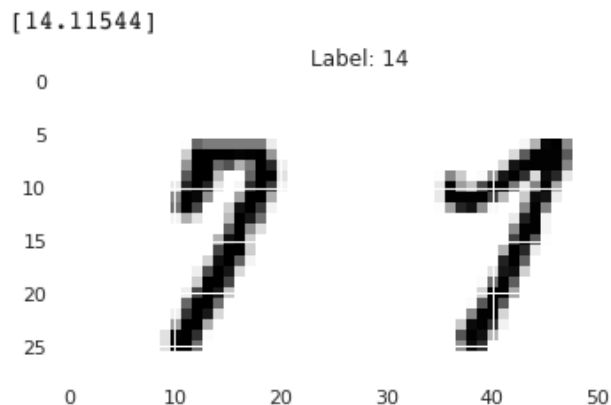
Picture 33

- **Some prediction example with same label result different permutations**

- ☐ First Example

1. You can see Picture 34 which is below.
2. Label is 14, but neural networks perform 14.11544

```
print(predictions[8500])
plt.title('Label: %d' % testDataLabel[8500])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[8500].squeeze(), cmap=plt.cm.binary)
plt.show()
```

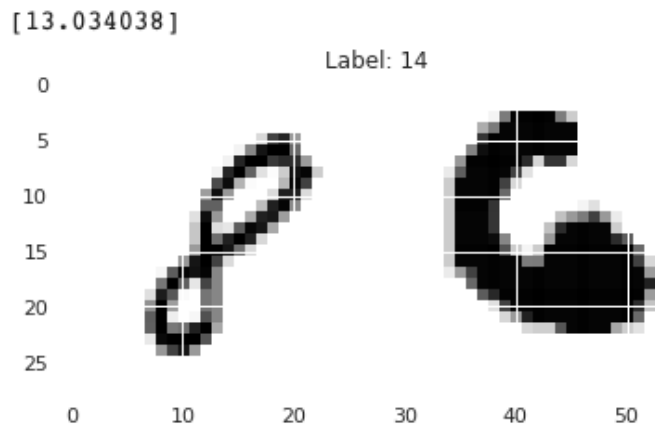


Picture 34

- ☐ Second Example

1. You can see Picture 35 which is below.
2. Label is 14, but neural networks perform 13.034038

```
print(predictions[9758])
plt.title('Label: %d' % testDataLabel[9758])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[9758].squeeze(), cmap=plt.cm.binary)
plt.show()
```



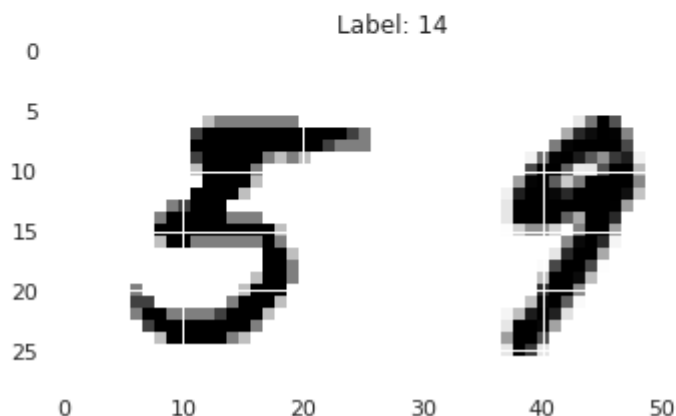
Picture 35

☐ Third Example:

1. You can see Picture 36 which is below.
2. Label is 14, but neural networks perform 12.278471

```
print(predictions[4444])
plt.title('Label: %d' % testDataLabel[4444])
#plt.imshow(foo[56664].squeeze(), cmap=plt.cm.gray_r)
plt.imshow(testDataImage[4444].squeeze(), cmap=plt.cm.binary)
plt.show()
```

[12.278471]



Picture 36

● Floor/Ceil and Round Accuracy:

☐ kfold = 10 :

```
Epoch 1/10
704/704 [=====] - 146s 206ms/step - loss: 9.4867 - accuracy: 0.0222 - val_loss: 4.7626 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 144s 204ms/step - loss: 3.8573 - accuracy: 0.0225 - val_loss: 3.9320 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 144s 205ms/step - loss: 3.2550 - accuracy: 0.0230 - val_loss: 3.7974 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 144s 204ms/step - loss: 2.9052 - accuracy: 0.0232 - val_loss: 4.5653 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 144s 204ms/step - loss: 2.6510 - accuracy: 0.0232 - val_loss: 3.2153 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 144s 205ms/step - loss: 2.4524 - accuracy: 0.0233 - val_loss: 3.1896 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 144s 205ms/step - loss: 2.2920 - accuracy: 0.0233 - val_loss: 3.7958 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 144s 205ms/step - loss: 2.1639 - accuracy: 0.0234 - val_loss: 2.9443 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 144s 205ms/step - loss: 2.0435 - accuracy: 0.0234 - val_loss: 2.8314 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 144s 205ms/step - loss: 1.9297 - accuracy: 0.0240 - val_loss: 2.7178 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 5702 accuracy in 10000 samples is %57.02
 Rounded prediction is correct = 3133 accuracy in 10000 samples is %31.33

☐ kfold = 9 :

```
Epoch 1/10
704/704 [=====] - 147s 207ms/step - loss: 21.0748 - accuracy: 0.0217 - val_loss: 10.6815 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 145s 206ms/step - loss: 17.2029 - accuracy: 0.0222 - val_loss: 10.2907 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 145s 207ms/step - loss: 17.1906 - accuracy: 0.0222 - val_loss: 10.4519 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 145s 207ms/step - loss: 17.1917 - accuracy: 0.0222 - val_loss: 10.3125 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 146s 207ms/step - loss: 17.1871 - accuracy: 0.0222 - val_loss: 10.5164 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 146s 207ms/step - loss: 17.1879 - accuracy: 0.0222 - val_loss: 10.1505 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 145s 206ms/step - loss: 17.1923 - accuracy: 0.0222 - val_loss: 10.2905 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 145s 207ms/step - loss: 17.1904 - accuracy: 0.0222 - val_loss: 10.9551 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 145s 206ms/step - loss: 17.1898 - accuracy: 0.0222 - val_loss: 10.4067 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 145s 206ms/step - loss: 17.1865 - accuracy: 0.0222 - val_loss: 10.4523 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 5702 accuracy in 10000 samples is %57.02
Rounded prediction is correct = 3133 accuracy in 10000 samples is %31.33

☐ kfold = 8 :

```
Epoch 1/10
704/704 [=====] - 147s 208ms/step - loss: 20.9447 - accuracy: 0.0220 - val_loss: 10.4862 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 145s 206ms/step - loss: 17.2108 - accuracy: 0.0222 - val_loss: 10.0639 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 145s 206ms/step - loss: 17.1926 - accuracy: 0.0222 - val_loss: 10.1224 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 145s 206ms/step - loss: 17.1920 - accuracy: 0.0222 - val_loss: 10.2413 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 145s 206ms/step - loss: 17.1918 - accuracy: 0.0222 - val_loss: 11.1046 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 145s 206ms/step - loss: 17.1926 - accuracy: 0.0222 - val_loss: 10.4438 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 145s 206ms/step - loss: 17.1905 - accuracy: 0.0222 - val_loss: 10.6011 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 145s 206ms/step - loss: 17.1883 - accuracy: 0.0222 - val_loss: 10.2821 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 145s 206ms/step - loss: 17.1897 - accuracy: 0.0222 - val_loss: 10.8974 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 145s 206ms/step - loss: 17.1921 - accuracy: 0.0222 - val_loss: 10.9760 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 4000 accuracy in 10000 samples is %40.0
Rounded prediction is correct = 2000 accuracy in 10000 samples is %20.0

☐ kfold = 7 :

```
Epoch 1/10
704/704 [=====] - 151s 214ms/step - loss: 21.3432 - accuracy: 0.0217 - val_loss: 11.1828 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 150s 214ms/step - loss: 17.2030 - accuracy: 0.0222 - val_loss: 10.7797 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 151s 214ms/step - loss: 17.1892 - accuracy: 0.0222 - val_loss: 10.2036 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 150s 214ms/step - loss: 17.1871 - accuracy: 0.0222 - val_loss: 10.5508 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 150s 213ms/step - loss: 17.1862 - accuracy: 0.0222 - val_loss: 10.0976 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 150s 214ms/step - loss: 17.1889 - accuracy: 0.0222 - val_loss: 10.1685 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 150s 213ms/step - loss: 17.1879 - accuracy: 0.0222 - val_loss: 10.9319 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 150s 213ms/step - loss: 17.1853 - accuracy: 0.0222 - val_loss: 10.5463 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 149s 212ms/step - loss: 17.1903 - accuracy: 0.0222 - val_loss: 10.9015 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 147s 208ms/step - loss: 17.1904 - accuracy: 0.0222 - val_loss: 10.6898 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 4000 accuracy in 10000 samples is %40.0
Rounded prediction is correct = 2000 accuracy in 10000 samples is %20.0

☐ kfold = 6 :

```
Epoch 1/10
704/704 [=====] - 145s 205ms/step - loss: 19.4621 - accuracy: 0.0216 - val_loss: 17.7998 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 143s 204ms/step - loss: 16.4210 - accuracy: 0.0222 - val_loss: 15.9970 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 145s 205ms/step - loss: 16.2119 - accuracy: 0.0222 - val_loss: 16.5359 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 145s 206ms/step - loss: 14.0293 - accuracy: 0.0222 - val_loss: 10.8031 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 146s 207ms/step - loss: 9.0361 - accuracy: 0.0224 - val_loss: 8.6163 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 145s 206ms/step - loss: 7.9287 - accuracy: 0.0228 - val_loss: 10.6158 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 143s 203ms/step - loss: 7.2936 - accuracy: 0.0229 - val_loss: 9.9803 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 143s 203ms/step - loss: 6.7527 - accuracy: 0.0230 - val_loss: 7.4049 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 143s 203ms/step - loss: 6.2420 - accuracy: 0.0231 - val_loss: 6.5724 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 143s 203ms/step - loss: 5.7880 - accuracy: 0.0233 - val_loss: 5.9642 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 3385 accuracy in 10000 samples is %33.85
Rounded prediction is correct = 1719 accuracy in 10000 samples is %17.19

☐ kfold = 5 :

```
Epoch 1/10
704/704 [=====] - 154s 218ms/step - loss: 9.5029 - accuracy: 0.0116 - val_loss: 5.2492 - val_accuracy: 0.1000
Epoch 2/10
704/704 [=====] - 154s 218ms/step - loss: 3.7985 - accuracy: 0.0124 - val_loss: 3.7584 - val_accuracy: 0.0985
Epoch 3/10
704/704 [=====] - 152s 217ms/step - loss: 3.2190 - accuracy: 0.0133 - val_loss: 3.9945 - val_accuracy: 0.1000
Epoch 4/10
704/704 [=====] - 153s 217ms/step - loss: 2.8740 - accuracy: 0.0138 - val_loss: 3.1495 - val_accuracy: 0.0948
Epoch 5/10
704/704 [=====] - 153s 217ms/step - loss: 2.5877 - accuracy: 0.0143 - val_loss: 3.0406 - val_accuracy: 0.0984
Epoch 6/10
704/704 [=====] - 153s 217ms/step - loss: 2.3574 - accuracy: 0.0145 - val_loss: 2.8123 - val_accuracy: 0.0999
Epoch 7/10
704/704 [=====] - 153s 218ms/step - loss: 2.1763 - accuracy: 0.0145 - val_loss: 3.0189 - val_accuracy: 0.0933
Epoch 8/10
704/704 [=====] - 153s 217ms/step - loss: 2.0136 - accuracy: 0.0148 - val_loss: 2.4551 - val_accuracy: 0.0985
Epoch 9/10
704/704 [=====] - 152s 216ms/step - loss: 1.9127 - accuracy: 0.0146 - val_loss: 2.5395 - val_accuracy: 0.0992
Epoch 10/10
704/704 [=====] - 153s 217ms/step - loss: 1.7835 - accuracy: 0.0148 - val_loss: 2.3150 - val_accuracy: 0.0996
```

Floored or ceiled prediction is correct = 5290 accuracy in 10000 samples is %52.9
Rounded prediction is correct = 2871 accuracy in 10000 samples is %28.71

✓ 0s completed at 11:01 AM

☐ kfold = 4 :

```
Epoch 1/10
704/704 [=====] - 153s 216ms/step - loss: 19.3787 - accuracy: 0.0112 - val_loss: 16.9425 - val_accuracy: 0.1000
Epoch 2/10
704/704 [=====] - 152s 216ms/step - loss: 16.3644 - accuracy: 0.0111 - val_loss: 17.1461 - val_accuracy: 0.1000
Epoch 3/10
704/704 [=====] - 153s 217ms/step - loss: 16.2943 - accuracy: 0.0111 - val_loss: 17.1843 - val_accuracy: 0.1000
Epoch 4/10
704/704 [=====] - 152s 216ms/step - loss: 15.2971 - accuracy: 0.0111 - val_loss: 12.8217 - val_accuracy: 0.1000
Epoch 5/10
704/704 [=====] - 153s 217ms/step - loss: 8.9789 - accuracy: 0.0114 - val_loss: 10.6316 - val_accuracy: 0.0997
Epoch 6/10
704/704 [=====] - 152s 216ms/step - loss: 7.7209 - accuracy: 0.0116 - val_loss: 8.8721 - val_accuracy: 0.0999
Epoch 7/10
704/704 [=====] - 152s 217ms/step - loss: 7.1982 - accuracy: 0.0117 - val_loss: 8.7077 - val_accuracy: 0.1000
Epoch 8/10
704/704 [=====] - 152s 216ms/step - loss: 6.7660 - accuracy: 0.0119 - val_loss: 8.8714 - val_accuracy: 0.1000
Epoch 9/10
704/704 [=====] - 152s 216ms/step - loss: 6.4081 - accuracy: 0.0121 - val_loss: 7.8705 - val_accuracy: 0.1000
Epoch 10/10
704/704 [=====] - 152s 216ms/step - loss: 6.1200 - accuracy: 0.0122 - val_loss: 7.1543 - val_accuracy: 0.1000
```

Floored or ceiled prediction is correct = 2734 accuracy in 10000 samples is %27.34
Rounded prediction is correct = 1343 accuracy in 10000 samples is %13.43

☐ kfold = 3:

```
Epoch 1/10
704/704 [=====] - 149s 210ms/step - loss: 20.1142 - accuracy: 0.0110 - val_loss: 17.3561 - val_accuracy: 0.1000
Epoch 2/10
704/704 [=====] - 148s 210ms/step - loss: 16.4009 - accuracy: 0.0111 - val_loss: 17.9058 - val_accuracy: 0.1000
Epoch 3/10
704/704 [=====] - 148s 210ms/step - loss: 16.3910 - accuracy: 0.0111 - val_loss: 18.3865 - val_accuracy: 0.1000
Epoch 4/10
704/704 [=====] - 148s 211ms/step - loss: 16.3864 - accuracy: 0.0111 - val_loss: 17.9403 - val_accuracy: 0.1000
Epoch 5/10
704/704 [=====] - 148s 210ms/step - loss: 16.3883 - accuracy: 0.0111 - val_loss: 17.5473 - val_accuracy: 0.1000
Epoch 6/10
704/704 [=====] - 147s 209ms/step - loss: 16.3820 - accuracy: 0.0111 - val_loss: 19.0179 - val_accuracy: 0.1000
Epoch 7/10
704/704 [=====] - 148s 210ms/step - loss: 16.3865 - accuracy: 0.0111 - val_loss: 17.4211 - val_accuracy: 0.1000
Epoch 8/10
704/704 [=====] - 148s 210ms/step - loss: 16.3883 - accuracy: 0.0111 - val_loss: 18.1321 - val_accuracy: 0.1000
Epoch 9/10
704/704 [=====] - 148s 210ms/step - loss: 16.3884 - accuracy: 0.0111 - val_loss: 18.2607 - val_accuracy: 0.1000
Epoch 10/10
704/704 [=====] - 147s 208ms/step - loss: 16.3851 - accuracy: 0.0111 - val_loss: 17.2543 - val_accuracy: 0.1000
```

Floored or ceiled prediction is correct = 3000 accuracy in 10000 samples is %30.0
Rounded prediction is correct = 1000 accuracy in 10000 samples is %10.0

☐ kfold = 2 :

```
Epoch 1/10
704/704 [=====] - 149s 210ms/step - loss: 20.1561 - accuracy: 0.0110 - val_loss: 17.8855 - val_accuracy: 0.1000
Epoch 2/10
704/704 [=====] - 148s 210ms/step - loss: 16.4023 - accuracy: 0.0111 - val_loss: 17.9125 - val_accuracy: 0.1000
Epoch 3/10
704/704 [=====] - 148s 210ms/step - loss: 16.3927 - accuracy: 0.0111 - val_loss: 18.0665 - val_accuracy: 0.1000
Epoch 4/10
704/704 [=====] - 149s 211ms/step - loss: 16.3886 - accuracy: 0.0111 - val_loss: 17.3335 - val_accuracy: 0.1000
Epoch 5/10
704/704 [=====] - 149s 211ms/step - loss: 16.3854 - accuracy: 0.0111 - val_loss: 17.9749 - val_accuracy: 0.1000
Epoch 6/10
704/704 [=====] - 148s 211ms/step - loss: 16.3920 - accuracy: 0.0111 - val_loss: 17.6207 - val_accuracy: 0.1000
Epoch 7/10
704/704 [=====] - 147s 209ms/step - loss: 16.3877 - accuracy: 0.0111 - val_loss: 18.9056 - val_accuracy: 0.1000
Epoch 8/10
704/704 [=====] - 147s 209ms/step - loss: 16.3907 - accuracy: 0.0111 - val_loss: 18.0984 - val_accuracy: 0.1000
Epoch 9/10
704/704 [=====] - 147s 209ms/step - loss: 16.3875 - accuracy: 0.0111 - val_loss: 18.0775 - val_accuracy: 0.1000
Epoch 10/10
704/704 [=====] - 148s 210ms/step - loss: 16.3828 - accuracy: 0.0111 - val_loss: 17.6121 - val_accuracy: 0.1000
```

Floored or ceiled prediction is correct = 1000 accuracy in 10000 samples is %10.0
Rounded prediction is correct = 1000 accuracy in 10000 samples is %10.0

● Some Other Network Train Example:

□ Alternative 1:

```
#15
#from https://www.machinecurve.com/index.php/2020/02/18/how-to-use-k-fold-cross-validation-with-keras/

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np

# change type to uint8 to float32
trainDataImage = trainDataImage.astype('float32')
testDataImage = testDataImage.astype('float32')

#image to gray scale
trainDataImage /= 255
testDataImage /= 255

def modelTrain():
    # Define the K-fold Cross Validator
    #10-fold Cross Calidator
    kfold = KFold(n_splits=1, shuffle=True)

    # K-fold Cross Validation model evaluation
    for train, test in kfold.split(trainDataImage, trainDataLabel):
        # Define the model architecture
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28,56,1)))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Flatten())
        model.add(Dense(256, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(1))
    return model
```

```
Epoch 1/10
704/704 [=====] - 190s 269ms/step - loss: 9.9871 - accuracy: 0.0224 - val_loss: 4.4232 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 189s 268ms/step - loss: 3.8421 - accuracy: 0.0227 - val_loss: 3.5230 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 188s 267ms/step - loss: 3.2046 - accuracy: 0.0230 - val_loss: 3.8410 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 187s 266ms/step - loss: 2.8329 - accuracy: 0.0236 - val_loss: 2.9563 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 188s 267ms/step - loss: 2.5886 - accuracy: 0.0240 - val_loss: 2.7185 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 188s 267ms/step - loss: 2.3785 - accuracy: 0.0239 - val_loss: 3.9532 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 186s 264ms/step - loss: 2.2062 - accuracy: 0.0242 - val_loss: 2.4196 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 186s 264ms/step - loss: 2.0595 - accuracy: 0.0244 - val_loss: 2.4307 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 184s 261ms/step - loss: 1.9571 - accuracy: 0.0244 - val_loss: 2.7786 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 185s 262ms/step - loss: 1.8491 - accuracy: 0.0244 - val_loss: 2.2090 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 5487 accuracy in 10000 samples is %54.87
Rounded prediction is correct = 3050 accuracy in 10000 samples is %30.5

□ Alternative 2:

```
#alternative version v1
#source : https://analyticsindiamag.com/complete-tutorial-on-lenet-5-guide-to-begin-with-cnns/

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np

# change type to uint8 to float32
trainDataImage = trainDataImage.astype('float32')
testDataImage = testDataImage.astype('float32')

#image to gray scale
trainDataImage /= 255
testDataImage /= 255

def modelTrain():
    # Define the K-fold Cross Validator
    #10-fold Cross Calidator
    kfold = KFold(n_splits=10, shuffle=True)

    # K-fold Cross Validation model evaluation
    for train, test in kfold.split(trainDataImage, trainDataLabel):
        # Define the model architecture
        model = Sequential()
        model.add(Conv2D(filters=6, kernel_size=(3, 3), activation='tanh', input_shape=(28,56,1)))
        model.add(AveragePooling2D())
        model.add(Conv2D(filters=16, kernel_size=(3, 3), activation='tanh'))
        model.add(AveragePooling2D())
        model.add(Flatten())
        model.add(Dense(units=128, activation='tanh'))
        model.add(Dense(units=1))
    return model
```

```
Epoch 1/10
704/704 [=====] - 47s 67ms/step - loss: 20.3185 - accuracy: 0.0218 - val_loss: 12.9683 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 45s 64ms/step - loss: 16.9897 - accuracy: 0.0222 - val_loss: 12.8715 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 45s 64ms/step - loss: 16.9864 - accuracy: 0.0222 - val_loss: 12.1850 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 45s 64ms/step - loss: 16.9865 - accuracy: 0.0222 - val_loss: 12.6067 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 45s 64ms/step - loss: 16.9818 - accuracy: 0.0222 - val_loss: 12.0796 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 45s 64ms/step - loss: 16.9860 - accuracy: 0.0222 - val_loss: 12.6889 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 45s 64ms/step - loss: 16.9840 - accuracy: 0.0222 - val_loss: 11.8750 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 45s 64ms/step - loss: 16.9856 - accuracy: 0.0222 - val_loss: 13.0163 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 45s 64ms/step - loss: 16.9832 - accuracy: 0.0222 - val_loss: 12.2088 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 45s 64ms/step - loss: 16.9857 - accuracy: 0.0222 - val_loss: 12.6497 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 5487 accuracy in 10000 samples is %54.87
Rounded prediction is correct = 3050 accuracy in 10000 samples is %30.5

□ Alternative 3:

```
#alternative version v1
#source : https://github.com/TaavishThaman/LeNet-5-with-Keras/blob/master/lenet\_5.py

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np

# change type to uint8 to float32
trainDataImage = trainDataImage.astype('float32')
testDataImage = testDataImage.astype('float32')

#image to gray scale
trainDataImage /= 255
testDataImage /= 255

def modelTrain():
    # Define the K-fold Cross Validator
    #10-fold Cross Calidator
    kfold = KFold(n_splits=10, shuffle=True)

    # K-fold Cross Validation model evaluation
    for train, test in kfold.split(trainDataImage, trainDataLabel):
        # Define the model architecture
        model = Sequential()
        #Layer 1
        #Conv Layer 1
        model.add(Conv2D(filters = 6, kernel_size = 5, strides = 1, activation = 'relu', input_shape = (28, 56, 1)))
        model.add(MaxPooling2D(pool_size = 2, strides = 2))
        model.add(Conv2D(filters = 16, kernel_size = 5, strides = 1, activation = 'relu'))
        model.add(MaxPooling2D(pool_size = 2, strides = 2))
        model.add(Flatten())
        model.add(Dense(units = 120, activation = 'relu'))
        model.add(Dense(units = 84, activation = 'relu'))
        model.add(Dense(units = 1))
    return model

Epoch 1/10
704/704 [=====] - 63s 89ms/step - loss: 21.5076 - accuracy: 0.0215 - val_loss: 10.8793 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 63s 89ms/step - loss: 16.9950 - accuracy: 0.0222 - val_loss: 12.1743 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 63s 89ms/step - loss: 16.9861 - accuracy: 0.0222 - val_loss: 11.9032 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 63s 89ms/step - loss: 16.9845 - accuracy: 0.0222 - val_loss: 12.8851 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 63s 90ms/step - loss: 16.9858 - accuracy: 0.0222 - val_loss: 12.4338 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 63s 90ms/step - loss: 16.9843 - accuracy: 0.0222 - val_loss: 12.3084 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 63s 90ms/step - loss: 16.9858 - accuracy: 0.0222 - val_loss: 12.5583 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 63s 89ms/step - loss: 16.9833 - accuracy: 0.0222 - val_loss: 12.6685 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 63s 90ms/step - loss: 16.9818 - accuracy: 0.0222 - val_loss: 12.3734 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 64s 90ms/step - loss: 16.9878 - accuracy: 0.0222 - val_loss: 13.0275 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 2000 accuracy in 10000 samples is %20.0
Rounded prediction is correct = 2000 accuracy in 10000 samples is %20.0

□ Alternative 4:

```
#alternative version v1
#source : https://www.kaggle.com/curiousprogrammer/lenet-5-cnn-with-keras-99-48

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np

# change type to uint8 to float32
trainDataImage = trainDataImage.astype('float32')
testDataImage = testDataImage.astype('float32')

#image to gray scale
trainDataImage /= 255
testDataImage /= 255

def modelTrain():
    # Define the K-fold Cross Validator
    #10-fold Cross Calidator
    kfold = KFold(n_splits=10, shuffle=True)

    # K-fold Cross Validation model evaluation
    for train, test in kfold.split(trainDataImage, trainDataLabel):
        # Define the model architecture
        model = Sequential()
        model.add(Conv2D(filters=32, kernel_size=(5,5), padding='same', activation='relu', input_shape=(28, 56, 1)))
        model.add(MaxPool2D(strides=2))
        model.add(Conv2D(filters=48, kernel_size=(5,5), padding='valid', activation='relu'))
        model.add(MaxPool2D(strides=2))
        model.add(Flatten())
        model.add(Dense(256, activation='relu'))
        model.add(Dense(84, activation='relu'))
        model.add(Dense(units = 1))
    return model

Epoch 1/10
704/704 [=====] - 223s 316ms/step - loss: 20.0472 - accuracy: 0.0219 - val_loss: 11.1103 - val_accuracy: 0.0000e+00
Epoch 2/10
704/704 [=====] - 219s 310ms/step - loss: 17.0073 - accuracy: 0.0222 - val_loss: 13.5779 - val_accuracy: 0.0000e+00
Epoch 3/10
704/704 [=====] - 227s 323ms/step - loss: 17.0031 - accuracy: 0.0222 - val_loss: 11.6235 - val_accuracy: 0.0000e+00
Epoch 4/10
704/704 [=====] - 224s 318ms/step - loss: 17.0009 - accuracy: 0.0222 - val_loss: 11.2437 - val_accuracy: 0.0000e+00
Epoch 5/10
704/704 [=====] - 225s 320ms/step - loss: 16.9946 - accuracy: 0.0222 - val_loss: 12.8459 - val_accuracy: 0.0000e+00
Epoch 6/10
704/704 [=====] - 226s 321ms/step - loss: 16.9967 - accuracy: 0.0222 - val_loss: 13.9736 - val_accuracy: 0.0000e+00
Epoch 7/10
704/704 [=====] - 226s 321ms/step - loss: 16.9972 - accuracy: 0.0222 - val_loss: 12.6352 - val_accuracy: 0.0000e+00
Epoch 8/10
704/704 [=====] - 226s 321ms/step - loss: 16.9971 - accuracy: 0.0222 - val_loss: 13.4112 - val_accuracy: 0.0000e+00
Epoch 9/10
704/704 [=====] - 227s 323ms/step - loss: 17.0018 - accuracy: 0.0222 - val_loss: 12.4508 - val_accuracy: 0.0000e+00
Epoch 10/10
704/704 [=====] - 225s 320ms/step - loss: 16.9966 - accuracy: 0.0222 - val_loss: 12.6053 - val_accuracy: 0.0000e+00
```

Floored or ceiled prediction is correct = 2000 accuracy in 10000 samples is %20.0
Rounded prediction is correct = 2000 accuracy in 10000 samples is %20.0

- References:

- <https://medium.com/@mgazar/lenet-5-in-9-lines-of-code-using-keras-ac99294c8086>
- <https://www.kaggle.com/curiousprogrammer/lenet-5-cnn-with-keras-99-48>
- <https://towardsdatascience.com/understanding-and-implementing-lenet-5-cnn-architecture-deep-learning-a2d531ebc342>
- <https://github.com/TaavishThaman/LeNet-5-with-Keras>
- <https://hackmd.io/@bouteille/S1WvJyqml>
- <https://www.pyimagesearch.com/2016/08/01/lenet-convolutional-neural-network-in-python/>
- <https://analyticsindiamag.com/complete-tutorial-on-lenet-5-guide-to-begin-with-cnns/>
- <https://github.com/udacity/CarND-LeNet-Lab/blob/master/LeNet-Lab-Solution.ipynb>
- https://keras.io/examples/vision/mnist_convnet/
- <https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/>
- https://www.tensorflow.org/datasets/keras_example
- <https://www.analyticsvidhya.com/blog/2021/06/mnist-dataset-prediction-using-keras/>
- <http://yann.lecun.com/exdb/mnist/>
- Reference Code attached with this report.