DIGIT RECOGNITION REPORT

BASICS OF MACHINE LEARNING

Ahmet Oral

December, 2021

Abstract

In this project I created a program that can identify handwritten digits with more than %99 accuracy (on test dataset). I used MNIST dataset for training our program. I created 3 .py files; 'saveModel.py', 'evaluateModel.py' and 'predictGUI.py'. In 'saveModel.py' we preprocess the data, normalize images, define and save the model. In 'evaluateModel.py', model can be evaluated. 'predictGUI.py' contains a GUI that user can draw digits with mouse and get predictions.

1 Introduction

Machines can read the digits written with specific fonts but can't read handwritten digits because each one is different then each other. Goal of this project is to create a Digit Recognition program which can predict output corresponding to handwritten images. I want this program to be not limited by datasets test images so I implemented a GUI for users to draw their digits and make predictions according to those images.

2 Dataset

MNIST is a widely used dataset for the hand-written digit classification task. The MNIST stands for the Modified National Institute of Standards and Technology. It consists of 70,000 labelled 28x28 pixel gray scale images of hand-written digits. The dataset is split into 60,000 training images and 10,000 test images. There are 10 classes (one for each of the 10 digits).

3 Creating and saving the Model (saveModel.py)

3.1 Loading and Preprocessing Dataset:

I loaded the data using 'mnist.load_data()' function. The shape of our training data is (60000, 28, 28).

To avoid,
ValueError: Error when checking input: expected 4 dimensions, but got array with shape (60000, 28, 28)

I had to reshape our data to have 4 dimensions. The 2D convolution layer in Keras expects the number of channels (by default as the last dimension). The images are 28x28 and 1 color channel so we reshaped the data arrays to have a single color channel.

To able to operate on label data, I used 'One-Hot Code' technique. So all input and output variables are converted to a numerical form.

```
#Loading and preprocessing dataset.
def load_dataset():
    #Loading mnist dataset
    (X_train, y_train), (X_test, y_test) = mnist.load_data()

    #Reshaping the data
    X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
    X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))

    #Applying One-Hot Code technique to convert values into numerical form.
    y_train = to_categorical(y_train)
    y_test = to_categorical(y_test)

    return X_train, y_train, X_test, y_test
```

Figure 1: load_dataset function

Pixel values in images have to be scaled to provide the images as input to our CNN model. I needed to normalize inputs from 0–255 to 0–1 as to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of value. To do this, we convert the data type from integers to floats, then divide pixel values by maximum value (which is 255).It helps the model to better learning of features by decreasing computational complexities

```
#Preparing Pixel Data by normalizing inputs from 0-255 to 0-1
def scale_pixels(train, test):
    #Convert integers to float
    newTrain = train.astype('float32')
    newTest = test.astype('float32')

    #Normalize it to range 0-1
    newTrain = newTrain / 255.0
    newTest = newTest / 255.0

    #Return normalized images
    return newTrain, newTest
```

Figure 2: scale_pixels function

3.2 Creating Model

```
#Defining a baseline Convolutional Neural Network (CNN) model.
def create_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))

    #Compiling the model
    opt = SGD(lr=0.01, momentum=0.9)
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Figure 3: scale_pixels function

I defined a baseline convolutional neural network model for the problem. I used Sequential model because it is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.

There are 2 main aspects in the model, a front end feature extraction comprised of convolutional and pooling layers, and a classifier backend that going to make a prediction.

For the convolutional front-end, I started with a single convolutional layer with a small filter with '3,3' size, and a '32' number of filters followed by a max pooling layer. To increase model depth I added more convolutional and pooling layers with the same sized filter but increased the number of filters to '64'.

Our problem is multi-class classification task, and we need an output layer with 10 nodes so that we can predict the probability of distribution of an image that belongs to one of the 10 classes. Because of this, I used softmax activation function. We added a dense layer with 100 nodes to interpret the features between the feature extractor and output layer.

I used 'ReLU' activation function and 'he_uniform' kernel initializer in all layers because they work the best for this problem.

For optimizer, I used SGD with learning rate of 0.01 and momentum of 0.9. We optimized 'categorical_crossentropy' loss function because it is suitable for multi-class classification. I monitored the classification accuracy metric because we have the same number of examples in each of the 10 classes.

3.3 Main Function

Main is our last function which combines all other functions for preprocessing and creating model. We start by loading the preprocessed dataset using 'load_dataset()' function. Then process pixel data using 'scale_pixels()' function. We load our mode using 'create_model()' function. After that we fit training dataset to our model and save it as 'model.h5'.

```
def main():
    #Loading the dataset
    X_train, y_train, X_test, y_test = load_dataset()

    #Normalizing images by processing pixel data
    X_train, X_test = scale_pixels(X_train, X_test)

    #Loading model
    model = create_model()

    #Fitting X_train and y_train into our model
    #model.fit(X_train, y_train, epochs=10, batch_size=32)
    model.fit(X_train, y_train, epochs=5, batch_size=32)

    #Saving the model as 'model.h5'
    model.save('model.h5')
```

Figure 4: main function

While fitting the training dataset into our model, I used 5 epochs and 32 batch size. I decided to left the verbose option as default so each epoch can be observed. The reason I used 5 epoch is because after around algorithm runs for 4 times, accuracy improvement is so little that we don't need to run it more times. To save on time we stop after 5 epochs. The reason I used batch size 32 is because I saw that 32 is balanced value between time and accuracy.

```
Epoch 1/5
1875/1875 [==============================] - 27s 14ms/step - loss: 0.1212 - accuracy: 0.9622
Epoch 2/5
1875/1875 [==============================] - 27s 15ms/step - loss: 0.0413 - accuracy: 0.9876
Epoch 3/5
1875/1875 [==============================] - 27s 14ms/step - loss: 0.0282 - accuracy: 0.9913
Epoch 4/5
1875/1875 [==============================] - 27s 14ms/step - loss: 0.0207 - accuracy: 0.9935
Epoch 5/5
1875/1875 [==============================] - 27s 15ms/step - loss: 0.0153 - accuracy: 0.9952

Process finished with exit code 0
```

Figure 5: Fitting the training data in the model

It takes around 2 minutes to fit the model and as seen in the picture above, we reached accuracy values more than %99. After the process we save the model as 'model.h5' so running it one time is enough for the program to work. In our repository we will include our .h5 file if you choose not to wait for this process, you can use our pre fitted model.

4 Evaluating the Model (evaluateModel.py)

We load and preprocess the data as we did in 'saveModel.py'. Inside the main function, we load the model we crated before. I used model.evaluate() function to evaluate the model on the test dataset. Then printed the accuracy and loss.

After running evaluateModel.py the you can see the accuracy and loss of the model.

```
313/313 [==============================] - 1s 4ms/step - loss: 0.0261 - accuracy: 0.9912

Accuracy of 'model.h5' is : 0.9911999702453613
Loss of 'model.h5' is : 0.026126554235816002

Process finished with exit code 0
```

Figure 6: Loss and Accuracy of the model

5 Creating GUI and Making Predictions (predictGUI.py)

I used tkinter library to create our gui.

I created 4 functions called 'load_prepare_predict()', 'painting(event)', 'btnPredict()' and 'btnClear()'.

5.1 load_prepare_predict() Function:

```python
#Load and prepare the image and return the prediction for that image
def load_prepare_predict():
    #Loading our model
    model = tf.keras.models.load_model('model.h5')

    #Loading the image (image name is 'image.png' because we captured users drawing and saved is as 'image.png')
    img = cv2.imread('image.png', 0)
    img = cv2.bitwise_not(img)

    #Showing captured image to user. This was mainly for test but we didn't delete it because it looks good.
    cv2.imshow('img', img)

    #Resizing and reshaping to match images in mnist data
    img = cv2.resize(img, (28, 28))
    img = img.reshape(1, 28, 28, 1)

    #Preparing pixels to normalize inputs
    img = img.astype('float32')
    img = img / 255.0

    #taking and returning the prediction
    pred = model.predict(img)
    return pred
```

Figure 7: load_prepare_predict() Function

In this function, we load our model and image. Image name is 'image.png' because when we call this function we will be already saved our image as 'image.png'. We process the image by resizing and reshaping. Then as we did in our saveModel.py file, we prepare the pixels by converting it to float and normalizing it to range 0-1. Then I use '.predict()' function to get prediction of the image using our model and return it.

5.2 btnPredict() function:

```
#Saves the image user drew, gets the prediction and inserts it to GUI
def btnPredict():
    #Saving the digit user drew as 'image.png'
    filename = "image.png"
    image1.save(filename)

    #Getting the prediction of the image
    pred = load_prepare_predict()

    #Printing the prediction and accuracy
    print('argmax', np.argmax(pred[0]), '\n', pred[0][np.argmax(pred[0])], '\n', classes[np.argmax(pred[0])])

    #Inserting the prediction and accuracy to GUI
    txt.insert(tk.INSERT, "{}\nAccuracy: {}%".format(classes[np.argmax(pred[0])],
                                                     round(pred[0][np.argmax(pred[0])] * 100, 3)))
```

Figure 8: btnPredict() function

This function is called when user clicks the 'Predict' button on GUI. We save the digit user drew as 'image.png' and get the prediction by calling 'load_prepare_predict()' function. Then we print(in console) and display(in GUI) prediction of the image.

5.3 btnClear() function:

```
#Clears GUI screen
def btnClear():
    cv.delete('all')
    draw.rectangle((0, 0, 500, 500), fill=(255, 255, 255, 0))
    txt.delete('1.0', END)
```

Figure 9: btnClear() function

This function is called when user clicks 'Clear' button. It clears the canvas where user drew, and text area where predictions displayed.

5.4 painting() function:

```
#Allowing user to draw black ovals with width:40
def painting(event):
    x1, y1 = (event.x - 10), (event.y - 10)
    x2, y2 = (event.x + 10), (event.y + 10)
    #Width 40 is ideal because lower values causes image to be distorted and it reduces the accuracy.
    cv.create_oval(x1, y1, x2, y2, fill="black", width=40)
    draw.line([x1, y1, x2, y2], fill="black", width=40)
```

Figure 10: painting() function

This function is called when user uses mouse1. It creates black ovals with 40 width to where user points the mouse. Width 40 look a little big when drawing. But lower width values causes our image to be distorted and reduces the accuracy.

5.5 GUI Design

```
#Creating instance of TK
root = Tk()
#Disabling resizability.
root.resizable(0, 0)

#Creating a white canvas with width,height of 500. We also made the cursor: circle
cv = Canvas(root, width=500, height=500, bg='white', cursor='circle')
cv.pack(expand=YES, fill=BOTH)

#Creating an empty image and draw object to draw on. It is not visible.
image1 = PIL.Image.new("RGB", (500, 500), (255, 255, 255))
draw = ImageDraw.Draw(image1)

#Binding mouse 1 to painting function.
cv.bind("<B1-Motion>", painting)

#Creating the text window to display predictions and accuracy.
txt = tk.Text(root, bd=3, exportselection=0, bg='WHITE', font='Helvetica', padx=10, pady=10, height=5, width=20)

#Creating and binding predict and clear buttons.
btnPredict = Button(text="Predict", command=btnPredict, bg='green', fg='white')
btnClear = Button(text="Clear", command=btnClear, bg='red', fg='white')

#Packing buttons and text in order of predict>clear>txt area.
btnPredict.pack()
btnClear.pack()
txt.pack()

#Title of window
root.title('Digit Recognizer')
```

Figure 11: Creating GUI

I disabled option to resizing window because if user resizes the window image gets distorted and it causes problems.

I created a white canvas with 500 width and heigt (it looks best with 500) and changed the cursor to circle so it looks different in GUI.

To save the image user drew, I created an empty image and draw object to draw on. It is not visible to the user (User see it as they are drawing on GUI) and has the same width and height as the canvas we created.

We bind mouse1 to painting function so user can drew. Then created a text small text area to display predictions and accuracy,

I created 2 buttons called predict and clear. Predict button runs the 'btnPredict()' function which save the image user drew and calls 'load_prepare_predict()' function to get predictions. Then it displays the results in GUI. Clear button calls the 'btnClear' function which clears the canvas and text area. I also colored the buttons so they look nice.

Finally we pack our buttons, text are and set the title of our GUI.

5.6 GUI and Testing

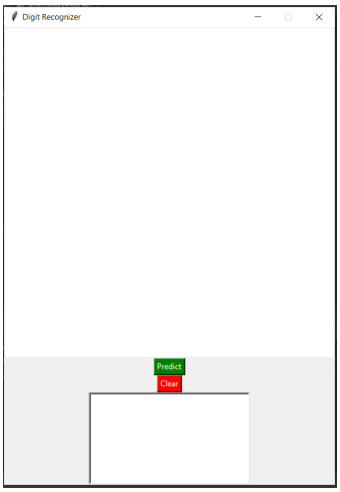


Figure 12: Empty GUI

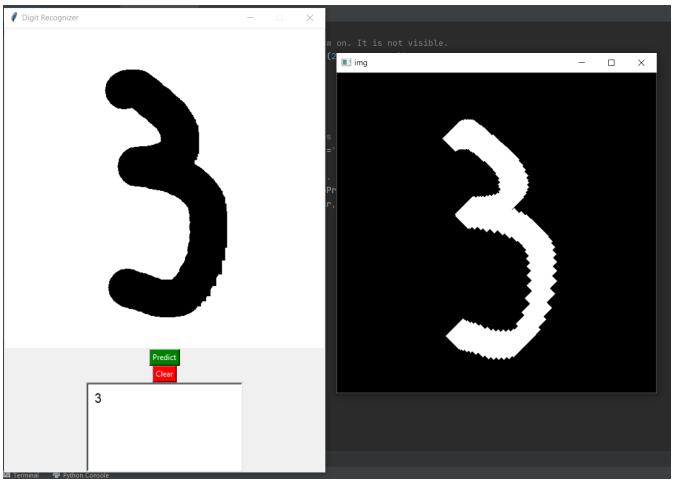
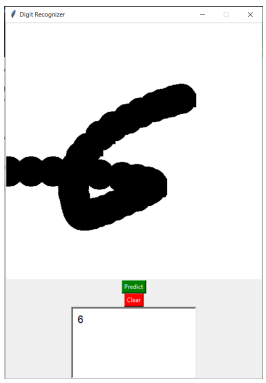


Figure 13: GUI Test

The GUI looks like in the figure12 when empty. It has 2 buttons which are predict and clear. Predict button displays the prediction and accuracy, clear button clears the canvas and text area.

On figure13 there is an example of drawing and getting prediction. As you can see, I drew 3 with my mouse and after clicking predict button, in the text area below you can see it predicts 3. The black and white image on the left is the image captured by the machine. It is processed and used to get predictions.



I did a lots of testing. If written clearly, program correctly reads the digit with really good accuracy. If handwritten is really bad, but still readable to humans, program still can accurately read most of the times(for ex. figure on left). If you randomly paint something it gets confused and fails with prediction and accuracy.