Handwritten Digit Recognition Report Using TF

Himanshu Gangwal

July 3, 2023

1 Introduction

This report presents an implementation of a handwritten digit recognition system using TensorFlow and the MNIST dataset.

2 Data Preparation

The MNIST dataset, which consists of grayscale images of handwritten digits and their corresponding labels, is imported using the `tf.keras.datasets.mnist` module.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = [1/256] * x_train
x_test = [1/256] * x_test
```

The input data is then normalized by dividing each pixel value by 256.

3 Model Overview

The implemented model is a sequential neural network with the following layers:

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

The first layer flattens the input images, and two hidden layers with 128 units each and ReLU activation are added. The final layer consists of 10 units with softmax activation for multi-class classification.

4 Model Architecture

The implemented model is a sequential neural network with the following layers:

4.1 Sequential Model

The choice of the `Sequential` model from the `tf.keras.models` module is appropriate for our task. The `Sequential` class allows us to build a sequential model in TensorFlow, which is a linear stack of layers where the output of one layer serves as the input for the next layer. In our case, we are designing a feedforward neural network where the data flows through the layers from input to output.

To create an empty neural network model, we instantiate a `Sequential` object as follows:

```
model = tf.keras.models.Sequential()
```

With this empty model, we can add layers to it using the `add()` method.

4.2 Flatten Layer

The first layer we add to the model is the `Flatten` layer. This layer is responsible for transforming the input data from a 2-dimensional shape (28x28 pixels) to a 1-dimensional shape (784 pixels). The subsequent layers in the model expect a 1-dimensional input. The `Flatten` layer does not have any trainable parameters but is necessary to make the input compatible with the fully connected layers.

We add the `Flatten` layer to the model as follows:

```
model.add(tf.keras.layers.Flatten(input_shape=(28, 28)))
```

Here, we specify the `input_shape` parameter as (28, 28), which represents the dimensions of the input images.

4.3 Dense Layers

After the `Flatten` layer, we add two `Dense` layers to the model. These layers are fully connected layers, where each neuron is connected to every neuron in the previous and following layers. The `Dense` layers have trainable weights and biases.

We add the `Dense` layers to the model as follows:

```
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='relu'))
```

Here, we specify the number of units (neurons) in each layer as 128, and we use the ReLU activation function. ReLU is a commonly used activation function in neural networks, known for its ability to introduce non-linearity to the model.

These `Dense` layers enable the model to learn complex patterns and relationships in the data, leading to better performance in the handwritten digit recognition task.

The next step is to add the final output layer to the model, compile it, and train it using appropriate settings.

5 Model Training

The model is compiled with the Adam optimizer, sparse categorical crossentropy loss, and accuracy as the evaluation metric.

```
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model.fit(x_train, y_train, epochs=6)
```

The model is trained on the normalized training data for 6 epochs.

6 Saving The Model

After training, the model's performance can be evaluated using the test dataset. Finally, the trained model is saved.

```
model.evaluate(x_test, y_test)
model.save('digrec.model')
```

7 Testing($\_run.pyfile$)

The code snippet provided demonstrates the evaluation of a trained model using the test dataset. Let's break down the code and understand its functionality.

7.1 Loading the Trained Model

The next line of code loads a pre-trained model from the file `digrec.model`. The function `tf.keras.models.load_model()` is used to load the model and assign it to the `model` variable.

7.2 Model Evaluation

To evaluate the model's performance, the code snippet uses the `evaluate()` function of the `model` object. It takes the test data (`x_test`) and corresponding labels (`y_test`) as inputs and returns the loss and accuracy of the model on the test dataset.

The loss and accuracy values are then printed to the console using the `print()` function.

It's important to note that this code assumes that the trained model file `digirec.model` is already present in the working directory.

7.3 Testing

This code snippet demonstrates how to load a pre-trained model and evaluate its performance on a test dataset. The model is loaded from a file, and the test data is prepared and evaluated using the loaded model. The output includes the loss and accuracy values of the model on the test dataset.

8 Conclusion

In this project, we implemented a handwritten digit recognition system using TensorFlow and the MNIST dataset. We began by importing the necessary libraries and loading the dataset, which consists of grayscale images of handwritten digits and their corresponding labels. We then normalized the input data by dividing each pixel value by 256.

Next, we constructed a sequential neural network model using TensorFlow's `Sequential` class. The model consisted of a `Flatten` layer to transform the input images from a 2-dimensional shape to a 1-dimensional shape, followed by two fully connected `Dense` layers with ReLU activation. The final output layer used the softmax activation function for multi-class classification.

We compiled the model using the Adam optimizer and sparse categorical crossentropy loss, and trained it on the normalized training data for a specified number of epochs. After training, we evaluated the model's performance on the test dataset and obtained the loss and accuracy values.

Overall, our implemented model achieved a certain level of accuracy on the test dataset, demonstrating its capability to recognize handwritten digits. Further improvements could be explored by adjusting hyperparameters, increasing the complexity of the model architecture, or applying additional preprocessing techniques.

Handwritten digit recognition has various practical applications, including optical character recognition, automated form processing, and signature verification. The techniques and concepts employed in this project can be extended to more complex image recognition tasks and serve as a foundation for building more sophisticated machine learning models.