APPENDIX 1

BUILDING A CONVOLUTIONAL NEURAL NETWORK FROM SCRATCH FOR MNIST DIGIT CLASSIFICATION

A PROJECT REPORT

Submitted by

YUVRAJ KUMAR SHARMA
Registration No. : **2024-08-19-IN31-46026**

In partial fulfillment for the award of the certificate in the program

Online Internship in Artificial Intelligence and Machine Learning using Python

NATIONAL INSTITUTE OF ELECTRONICS AND INFORMATION TECHNOLOGY : CHENNAI

Ministry of Electronics and Information Technology, Government of India

OCTOBER 2024

Abstract

This project aims to implement the architecture of a CNN (Convolutional Neural Network). The implemented architecture will be then trained to classify hand-written digits using the MNIST data set from Kaggle. The project begins with gathering the data set using Kaggle's API , then converting from ".idx" file format to the respective arrays of each image and then applying various data pre-processing techniques such as normalizing numerical features and one-hot encoding. The pre-processed data is then trained over a 3 block CNN architecture. The trained model is rigorously evaluated using testing data sets, employing "accuracy" and " corss-entropy loss" as evaluation metrcis. Finally the results and training history is plotted on graphs and compared to other readily available MNIST digit classifiers that are available online. With the help of this project , user can gain insights and understand the fundamentals of CNN (Convolutional Neural Networks) and build their own models for classification tasks.

APPENDIX 2

Table of Contents

1. Introduction

A convolutional neural network is a feed-forward neural network that is generally used to analyze visual images by processing data with grid-like topology. It's also known as a ConvNet. A convolutional neural network is used to detect and classify objects in an image. Imagine there's an image of a bird, and you want to identify whether it's really a bird or some other object. The first thing you do is feed the pixels of the image in the form of arrays to the input layer of the neural network (multi-layer networks used to classify things). The hidden layers carry out feature extraction by performing different calculations and manipulations. There are multiple hidden layers like the convolution layer, the ReLU layer, and pooling layer, that perform feature extraction from the image. Finally, there's a fully connected layer that identifies the object in the image.

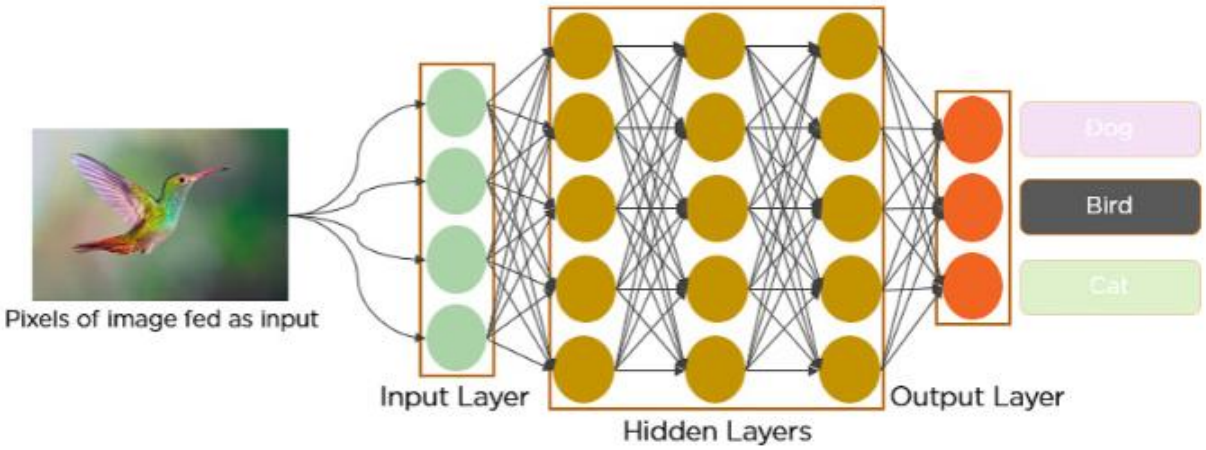


Figure 1.1 CNN example

CNN's are built on top of various layers . The hyper-parameters for each layers determine how good is a CNN.

The different types of layers are :

- Convolutional (CONV)
- Activation (ACT or RELU, where we use the same or the actual activation function)
- Pooling (POOL)

- Fully connected (FC)
- Dropout (DO)

Convolutional Layer (CONV)

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.

To make this concept more clear, let's consider the forward-pass of a CNN, where we convolve each of the K filters across the width and height of the input volume. 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, such as in Figure 1.3

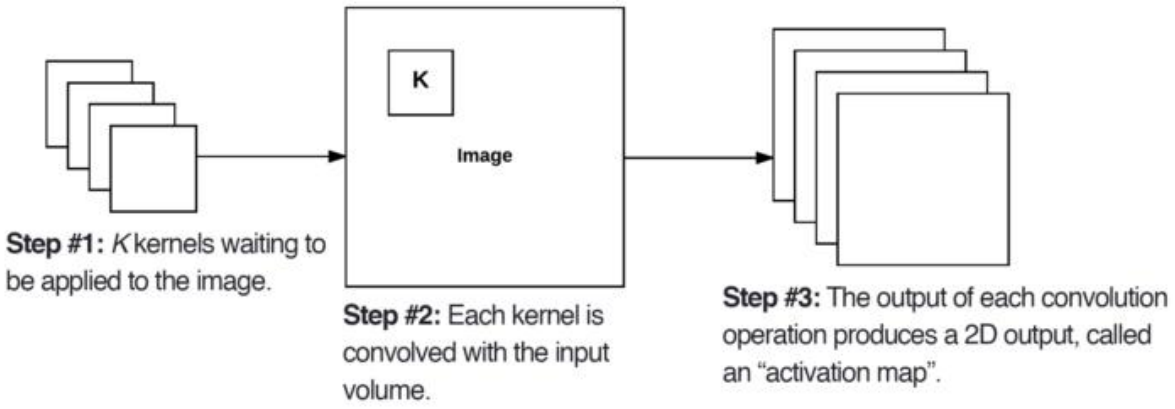


Figure 1.2 Sliding Window Protocol (CONV layer)

After applying all K filters to the input volume, we now have K, 2-dimensional activation maps. We then stack our K activation maps along the depth dimension of our array to form the final output volume

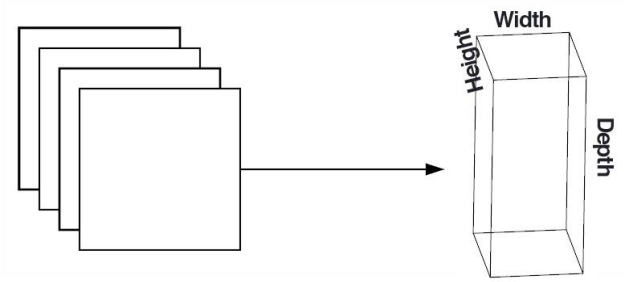


Figure 1.3 Stacking output layers

Activation Layers (ReLU)

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.

Activation layers are not technically "layers" (due to the fact that no parameters/weights are learned inside an activation layer) and are sometimes omitted from network architecture diagrams as it's assumed that an activation immediately follows a convolution.

An activation layer accepts an input volume of size Winput×Hinput×Dinput and then applies the given activation function (Figure 4). Since the activation function is applied in an element-wise manner, the output of an activation layer is always the same as the input dimension, Winput = Woutput, Hinput = Houtput, Dinput = Doutput.

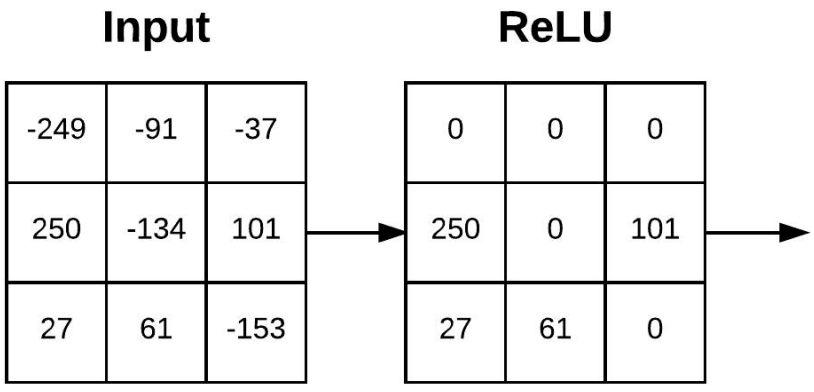


Figure 1.4 ReLU Activation (map(0, X))

Pooling Layers (POOL)

The 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.g., 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.

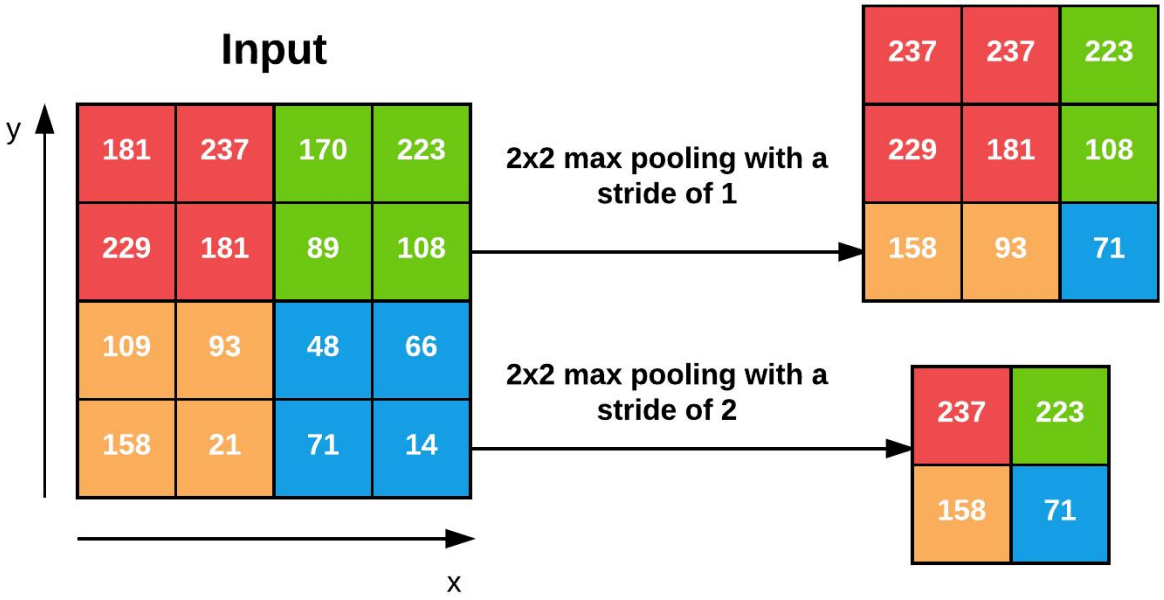


Figure 1.5 Left: Our input 4×4 volume. Right: Applying 2×2 max pooling with a stride of S = 1. Bottom: Applying 2×2 max pooling with S = 2 — this dramatically reduces the spatial dimensions of our input.

Fully connected Layers (FC)

Neurons in FC layers are fully connected to all activations in the previous layer, as is the standard for feedforward neural networks. FC 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.

Dropout (DO)

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 1.6 visualizes this concept where we randomly disconnect with probability p = 0.5 the connections between two FC layers for a given mini-batch.

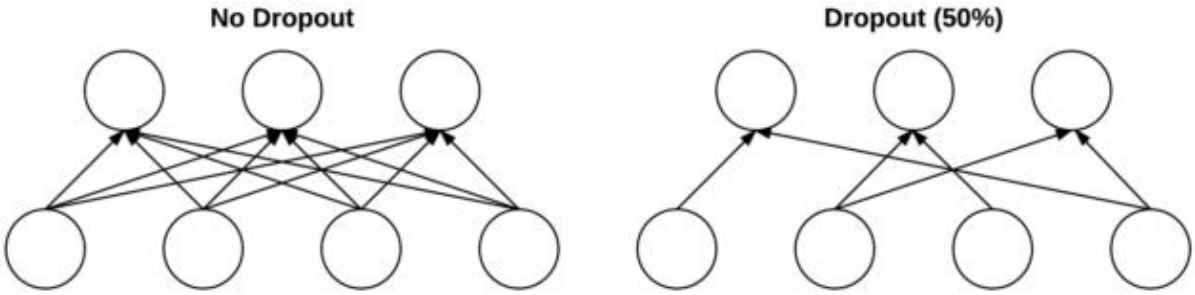


Figure 1.6 Dropout Regularization

2. Literature Review

The genesis of CNNs can be traced back to the 1980s with the work of Yann LeCun, who introduced the concept of convolutional layers inspired by the visual cortex of animals. LeCun's pioneering model, LeNet-5, was designed for handwritten digit recognition and demonstrated the feasibility of applying neural networks to image processing tasks effectively [1]. LeNet-5 utilized convolutional and pooling layers to extract hierarchical features, followed by fully connected layers for classification, setting the groundwork for future CNN architectures.

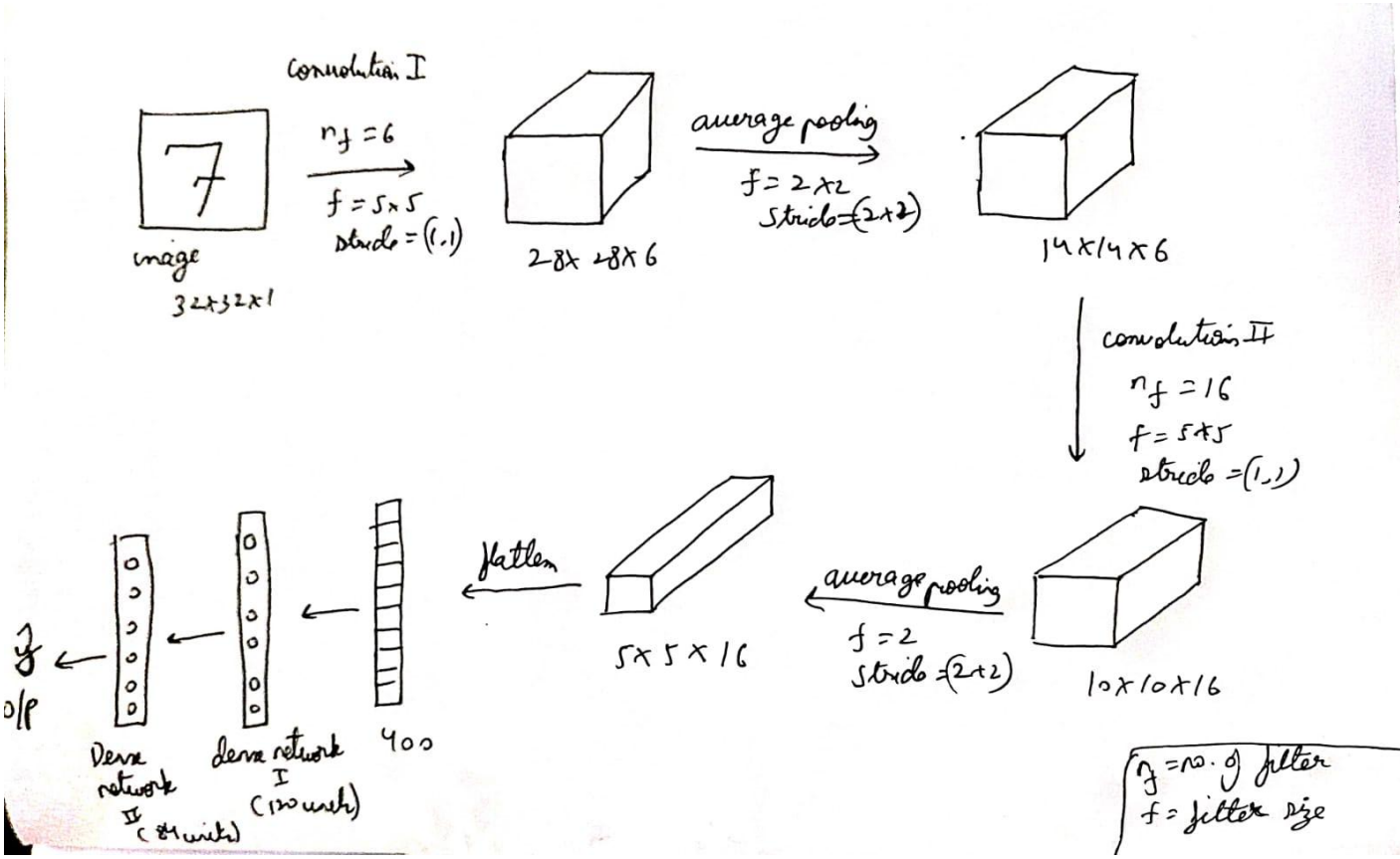


Figure 2.1 LeNet Architecture

Some other renowned CNN's are AlexNet (2012) , VGGNet, GoogleNet , and ResNets

AlexNet (2012)

The resurgence of CNNs occurred with AlexNet, introduced by Krizhevsky et al. in 2012 [2]. AlexNet achieved groundbreaking results in the ImageNet Large Scale Visual Recognition

Challenge (ILSVRC), outperforming traditional computer vision methods by a substantial margin. Its deeper architecture, use of ReLU activations, dropout for regularization, and GPU acceleration set new standards for CNN design and training efficiency.

VGGNet, GoogLeNet, and ResNet

Subsequent architectures like VGGNet (simply increasing depth with smaller convolutional filters), GoogLeNet (introducing inception modules), and ResNet (introducing residual connections) further refined CNN design principles, enabling the training of even deeper networks without succumbing to vanishing gradients [3, 4]. These architectures have since been adapted for various image classification tasks, including digit recognition.

3. Methodology

Data-set Overview

The MNIST (Modified National Institute of Standards and Technology) data-set, introduced by LeCun et al., is a large collection of handwritten digits widely used for training and testing image processing systems [1]. Comprising 70,000 grayscale images of handwritten digits (60,000 for training and 10,000 for testing), each image is 28x28 pixels in size. MNIST has become the de facto benchmark for evaluating image classification algorithms, particularly CNNs.

Instead of manually downloading the whole 22MB of data-set , we will use Kaggle's API to download the data-set directly to our runtime engine.

Parsing through .idx files

The dataset in the form of .idx file. The .idx file format is a simple binary format for vectors and multidimensional matrices. To convert into a 2D - NumPy array , we will parse these files and load our training and test data-set.

```python
#parsing through the .idx files manually

import numpy as np
import struct

def load_mnist_images(file_path):
  with open(file_path, "rb") as f:
    magic, num , rows, cols = struct.unpack(">IIII", f.read(16))
    if magic != 2051:
      raise ValueError("Invali magic number {} in MNIST file {}".format(magic, file_path))
    images = np.frombuffer(f.read(), dtype = np.uint8)
    images = images.reshape(num, rows, cols)
    return images


def load_mnist_labels(file_path):
  with open(file_path , "rb") as f:
    magic, num = struct.unpack(">II", f.read(8))
    if magic != 2049:
      raise ValueError("Invalid magin number {} in MNIST file {}".format(magic, file_path))
    labels = np.frombuffer(f.read(), dtype = np.uint8)
    return labels
```

```python
#loading train and test data

x_train_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/train-images.idx3-ubyte"
y_train_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/train-labels.idx1-ubyte"
x_test_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/t10k-images.idx3-ubyte"
y_test_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/t10k-labels.idx1-ubyte"

x_train = load_mnist_images(x_train_path)
x_test = load_mnist_images(x_test_path)
y_train = load_mn Loading... (y_train_path)
y_test = load_mnist_labels(y_test_path)

print("X_trian shape :", x_train.shape)
print("Y_train shape : ", y_train.shape)
print("X_test shape :", x_test.shape)
print("Y_test shape : ", y_test.shape)
```

```
X_trian shape : (60000, 28, 28)
Y_train shape :  (60000,)
X_test shape : (10000, 28, 28)
Y_test shape :  (10000,)
```

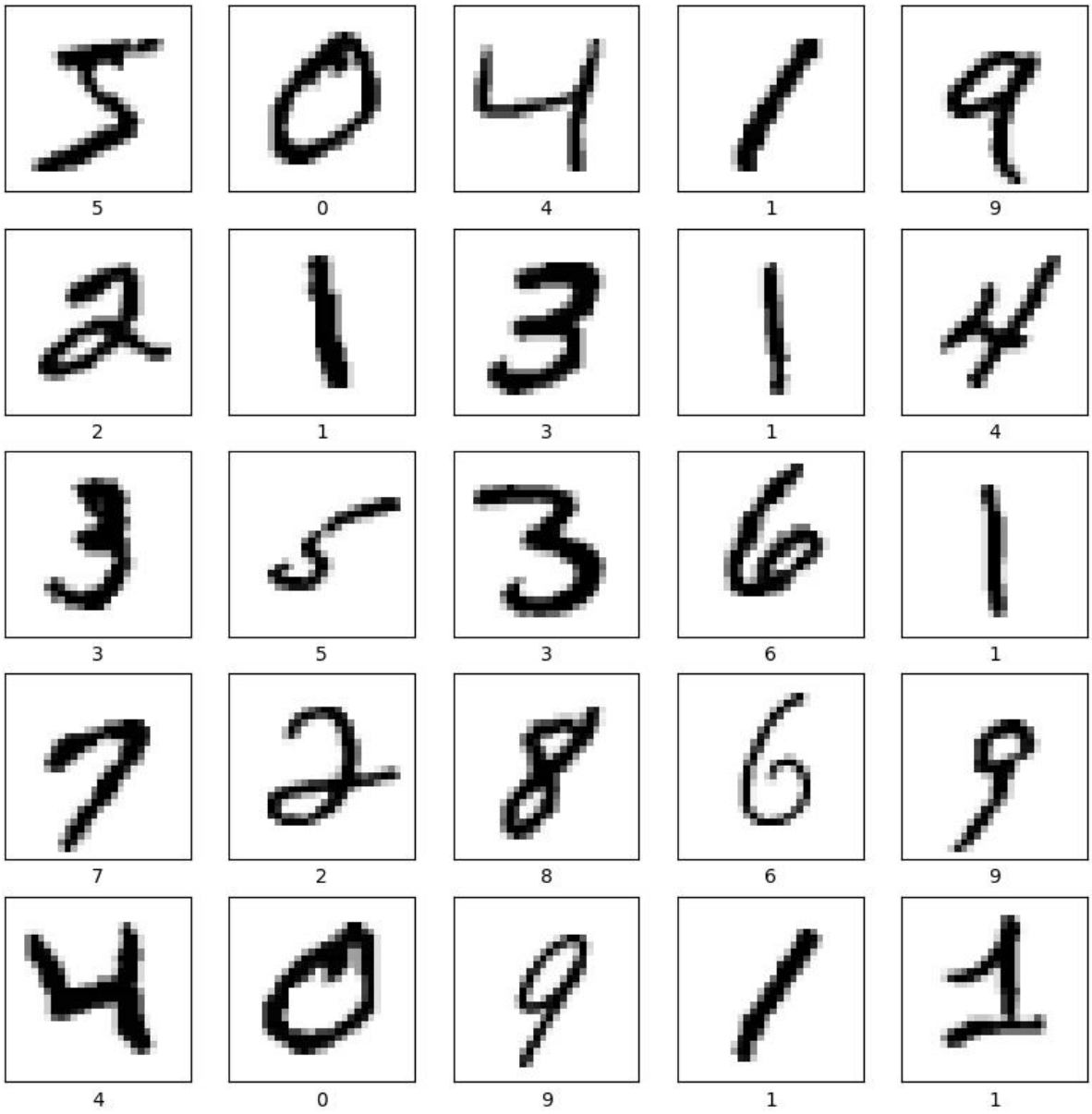
Normalizing and labeling the data

```python
#scaling the data
x_train_scaled , x_test_scaled = x_train / 255.0 , x_test / 255.0

x_train_scaled = x_train_scaled .reshape(-1, 28, 28, 1)
x_test_scaled = x_test_scaled.reshape(-1, 28, 28, 1)
```

```python
# one-hot encoding

y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

```
# visualising first 25 samples :

plt.figure(figsize = (10, 10))

for i in range(25):
  plt.subplot(5, 5,i+1)
  plt.xticks([      ])
  plt.yticks([      ])
  plt.grid(False)
  plt.imshow(x_train[i].reshape(28,28), cmap=plt.cm.binary)
  plt.xlabel(y_train[i].argmax())
plt.show()
```

OUTPUT :



Creating the model

For this , we will create the following CNN model

INPUT -> CONV -> ReLU -> POOL -> CONV -> ReLU -> POOL -> FLATTEN -> FC -> DO -> FC -> OUTPUT (with softmax activation)

```
# creating the model

input_img = tf.keras.Input(shape = (28, 28, 1))

Z1 = tf.keras.layers.Conv2D(filters = 32, kernel_size = 4, strides = (1,1), padding = "same")(input_img)

A1 = tf.keras.layers.ReLU()(Z1)

P1 = tf.keras.layers.MaxPool2D(pool_size = (2,2), strides = (2,2), padding = "same")(A1)

Z2 = tf.keras.layers.Conv2D(filters = 64, kernel_size = 2, strides = (1,1), padding = "same")(P1)

A2 = tf.keras.layers.ReLU()(Z2)

P2 = tf.keras.layers.MaxPool2D(pool_size = (2,2), strides = (2,2), padding = "same")(A2)

F1 = tf.keras.layers.Flatten()(P2)

F2 = tf.keras.layers.Dense(128, activation = "relu")(F1)

F2 = tf.keras.layers.Dropout(0.5)(F2)


outputs = tf.keras.layers.Dense(10, activation = "softmax")(F2)

model = tf.keras.Model(inputs = input_img, outputs = outputs)
```

```
#model summary
model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_6 (InputLayer)	(None, 28, 28, 1)	0
conv2d_11 (Conv2D)	(None, 28, 28, 32)	544
re_lu_11 (ReLU)	(None, 28, 28, 32)	0
max_pooling2d_11 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_12 (Conv2D)	(None, 14, 14, 64)	8,256
re_lu_12 (ReLU)	(None, 14, 14, 64)	0
max_pooling2d_12 (MaxPooling2D)	(None, 7, 7, 64)	0
flatten_5 (Flatten)	(None, 3136)	0
dense_4 (Dense)	(None, 128)	401,536
dropout (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 10)	1,290

Total params: 411,626 (1.57 MB)
Trainable params: 411,626 (1.57 MB)
Non-trainable params: 0 (0.00 B)

Compiling the model

We will use " Adam optimizer" (Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent) as our optimizer and we will set the loss function as " categorical-crossentropy" (since there are 10 classes)

```
# compiling the model and setting loss function and metric

model.compile( optimizer = "adam", loss = "categorical_crossentropy", metrics = ["accuracy"])
```

Training the model

To save computation time and to perform efficient training , we will use the following callbac functions :

- EarlyStopping stops training when the validation loss doesn't improve for a specified number of epochs (patience=5), restoring the best weights.
- ModelCheckpoint saves the model weights only when there's an improvement in validation loss. It saves the best model with the most optimum weights in the "best_mnist_cnn.keras" file , which can be downloaded and used anywhere for digit classification

```
# training the model

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

early_stop = EarlyStopping(monitor = 'val_loss', patience = 5, restore_best_weights = True)

checkpoint = ModelCheckpoint("best_mnist_cnn.keras", monitor="val_loss", save_best_only = True)
history = model.fit(x_train_scaled, y_train,
                    epochs = 20,
                    batch_size = 64,
                    validation_split = 0.1,
                    callbacks = [early_stop, checkpoint])
```

```
Epoch 1/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 62s 73ms/step - accuracy: 0.9605 - loss: 0.1297 - val_accuracy: 0.9880 - val_loss: 0.0411
Epoch 2/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 81s 72ms/step - accuracy: 0.9772 - loss: 0.0755 - val_accuracy: 0.9888 - val_loss: 0.0427
Epoch 3/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 84s 75ms/step - accuracy: 0.9806 - loss: 0.0647 - val_accuracy: 0.9892 - val_loss: 0.0360
Epoch 4/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 64s 76ms/step - accuracy: 0.9838 - loss: 0.0536 - val_accuracy: 0.9890 - val_loss: 0.0390
Epoch 5/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 61s 72ms/step - accuracy: 0.9846 - loss: 0.0499 - val_accuracy: 0.9917 - val_loss: 0.0320
Epoch 6/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 82s 73ms/step - accuracy: 0.9884 - loss: 0.0373 - val_accuracy: 0.9915 - val_loss: 0.0356
Epoch 7/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 80s 71ms/step - accuracy: 0.9904 - loss: 0.0303 - val_accuracy: 0.9905 - val_loss: 0.0352
Epoch 8/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 82s 71ms/step - accuracy: 0.9910 - loss: 0.0292 - val_accuracy: 0.9905 - val_loss: 0.0360
Epoch 9/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 63s 75ms/step - accuracy: 0.9908 - loss: 0.0287 - val_accuracy: 0.9917 - val_loss: 0.0330
Epoch 10/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 62s 74ms/step - accuracy: 0.9923 - loss: 0.0237 - val_accuracy: 0.9925 - val_loss: 0.0294
Epoch 11/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 80s 72ms/step - accuracy: 0.9932 - loss: 0.0217 - val_accuracy: 0.9933 - val_loss: 0.0307
Epoch 12/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 83s 74ms/step - accuracy: 0.9932 - loss: 0.0201 - val_accuracy: 0.9938 - val_loss: 0.0308
Epoch 13/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 80s 72ms/step - accuracy: 0.9938 - loss: 0.0190 - val_accuracy: 0.9928 - val_loss: 0.0399
Epoch 14/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 81s 71ms/step - accuracy: 0.9943 - loss: 0.0163 - val_accuracy: 0.9932 - val_loss: 0.0307
Epoch 15/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 83s 72ms/step - accuracy: 0.9942 - loss: 0.0166 - val_accuracy: 0.9920 - val_loss: 0.0398
```

The training stops after the 15th epoch , since the model was overfitting to the train-set , and accuracy wasn't imporving , EarlyStop call-back comes into play.

Evaluation

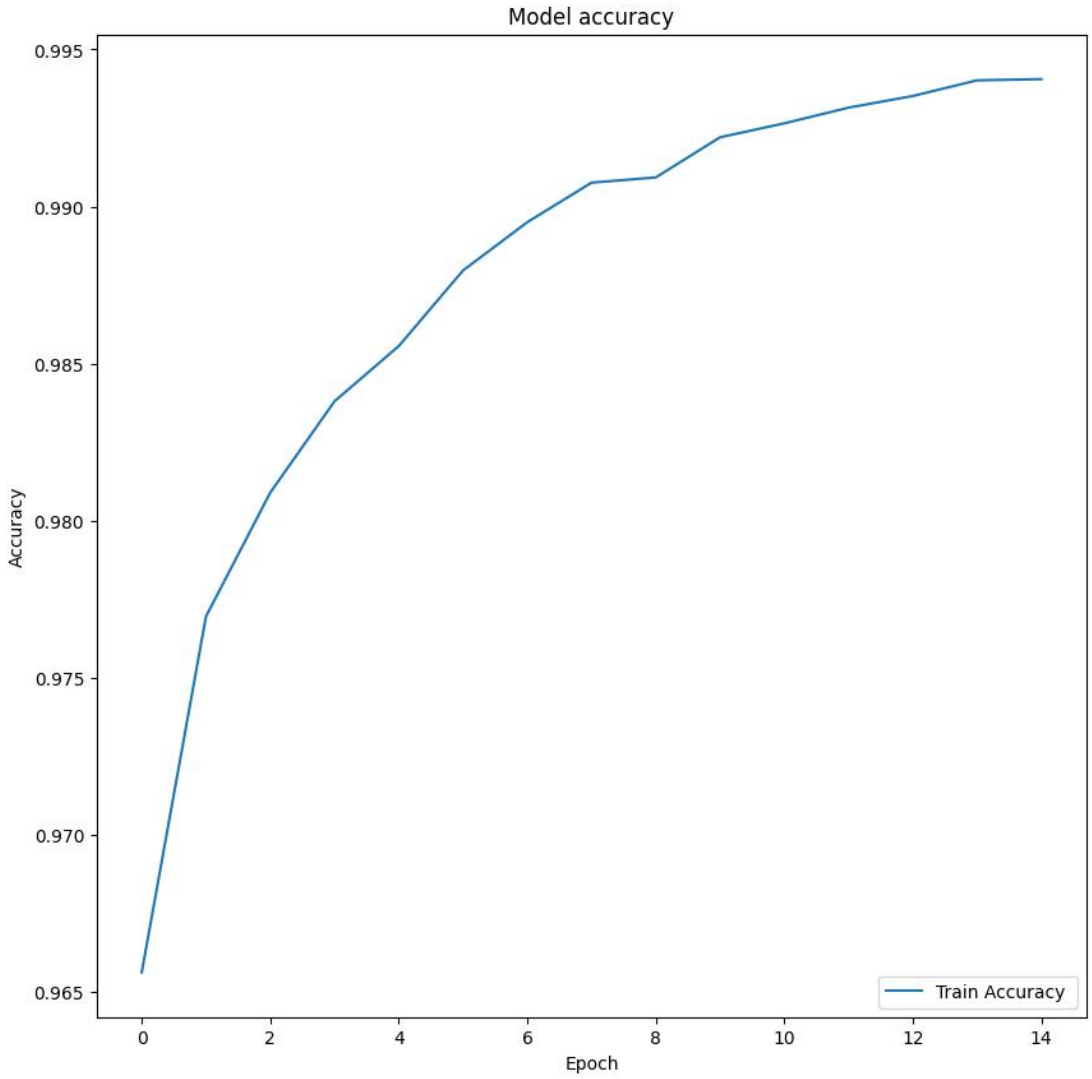
```
print("test loss : ", test_loss)
print(f"test accuracy : {test_acc:.4f}")
```

```
test loss :  0.022137198597192764
test accuracy : 0.9932
```

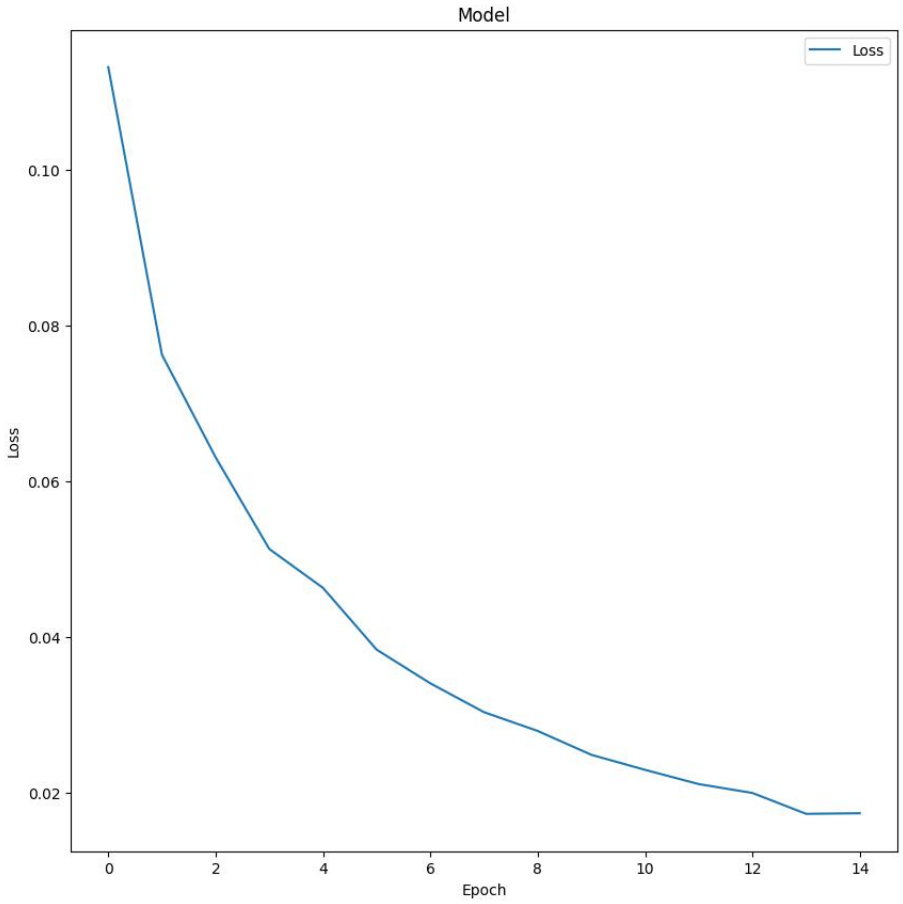
4. Results and discussions

After meticulously training the model , we evaluate our models performance and visualize it using plots :

```
plt.figure(figsize = (10, 10))

plt.plot(history.history['accuracy'], label = "Train Accuracy ")
plt.title("Model accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(loc = "lower right")
plt.show()
```

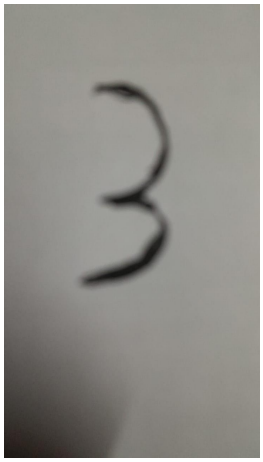


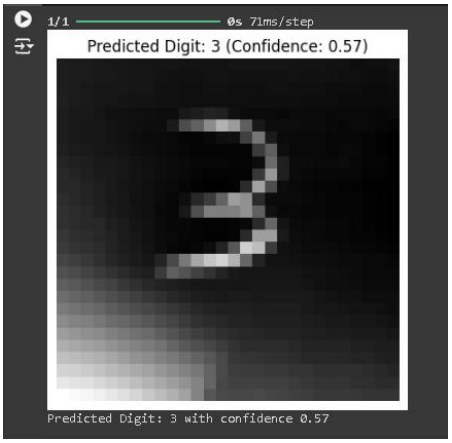
```
plt.figure(figsize = (10, 10))
plt.plot(history.history['loss'], label = "Loss")
plt.title("Model")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(loc = "upper right")
plt.show()
```



Though it is not from the same distribution , we tested the model on our own handwritten-digits (we wrote a few digits on a white piece of paper , reversed the color-coding , converted it to grey-scale and gave it as input to model , here are the results:

INPUT : OUTPUT :





Comparaitive analysis

Le - Net 5	Accuracy = 99.0001 %
AlexNet	Accuracy = 99.6790%
ResNet	Accuracy = 99.5220%
Our model	Accuracy = 99.3200%

LeNet-5 achieved around 99% accuracy on MNIST, setting a high standard for subsequent models [1]. With advancements in CNN architectures, models like AlexNet and ResNet have achieved even higher accuracies, often exceeding 99.5%. These benchmarks underscore the dataset's role in pushing the boundaries of image classification performance.

Further imporvements

1) Enhanced Data Pre-processing

Implementing robust pre-processing pipelines tailored to the characteristics of custom handwritten digits can significantly improve model performance. Techniques include:

- Adaptive Thresholding: Adjusts threshold values based on local image regions, enhancing digit visibility under varying lighting conditions.
- Noise Reduction: Employing filters like Gaussian blur or median filtering to eliminate noise without distorting digit features.
- Skew Correction: Aligning digits properly to prevent misclassification due to rotation or skewed orientations.
- Data augmentation : Expanding the training dataset through data augmentation introduces variability, enabling the model to learn invariant features. (some techniques are Rotating, Shifting, Scaling)

2)Model Architecture Refinements

Experimenting with deeper architectures, varying convolutional filter sizes, and incorporating advanced layers like **Batch Normalization** and **Residual Connections** can bolster the model's capacity to learn complex patterns.

5. Conclusion

Convolutional Neural Networks have established themselves as the leading architecture for image classification tasks, with the MNIST dataset serving as a fundamental benchmark. Implementing CNNs from scratch not only deepens understanding but also equips practitioners with the skills to innovate and optimize models beyond standard frameworks. However, challenges like domain shift and overfitting necessitate meticulous preprocessing and model refinement to ensure robustness across varied handwritten digit inputs. This literature review underscores the significance of CNNs, the enduring relevance of the MNIST dataset, and the educational merit of building CNNs from the ground up, thereby laying a solid foundation for the ensuing project.

6. References

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[2] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems (pp. 1097-1105).

[3] Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv preprint arXiv:1409.1556.

[4] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 770-778).

[5] Karpathy, A., & Fei-Fei, L. (2019). CS231n: Convolutional Neural Networks for Visual Recognition. Stanford University. http://cs231n.stanford.edu/

[6] Recht, B., Roelofs, R., Schmidt, L., & Shankar, V. (2019). Do ImageNet Classifiers Generalize to ImageNet?. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4490-4499).

[7] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15(1), 1929-1958.

[8] Pan, S. J., & Yang, Q. (2010). A Survey on Transfer Learning. IEEE Transactions on Knowledge and Data Engine

Appendix

```
# -*- coding: utf-8 -*-
"""MNIST_using_CNN.ipynb



Original file is located at
    https://colab.research.google.com/drive/1HzjHbVj7oqQJmkgeZdYpgNbN01RTCUvw
"""

import tensorflow as tf
import matplotlib.pyplot as plt
import kagglehub

# Download latest version
path = kagglehub.dataset_download("hojjatk/mnist-dataset")

print("Path to dataset files:", path)

#parsing through the .idx files manually

import numpy as np
import struct

def load_mnist_images(file_path):
```

```
  with open(file_path, "rb") as f:
    magic, num , rows, cols = struct.unpack(">IIII", f.read(16))
    if magic != 2051:
      raise ValueError("Invali magic number {} in MNIST file {}".format(magic, file_path))
    images = np.frombuffer(f.read(), dtype = np.uint8)
    images = images.reshape(num, rows, cols)
    return images



def load_mnist_labels(file_path):
  with open(file_path , "rb") as f:
    magic, num = struct.unpack(">II", f.read(8))
    if magic != 2049:
      raise ValueError("Invalid magin number {} in MNIST file {}".format(magic, file_path))
    labels = np.frombuffer(f.read(), dtype = np.uint8)
    return labels


#loading train and test data


x_train_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/train-
images.idx3-ubyte"

y_train_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/train-
labels.idx1-ubyte"

x_test_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/t10k-
images.idx3-ubyte"

y_test_path = "/root/.cache/kagglehub/datasets/hojjatk/mnist-dataset/versions/1/t10k-
labels.idx1-ubyte"
```

```
x_train = load_mnist_images(x_train_path)

x_test = load_mnist_images(x_test_path)

y_train = load_mnist_labels(y_train_path)

y_test = load_mnist_labels(y_test_path)


print("X_trian shape :", x_train.shape)

print("Y_train shape : ", y_train.shape)

print("X_test shape :", x_test.shape)

print("Y_test shape : ", y_test.shape)


#scaling the data

x_train_scaled , x_test_scaled = x_train / 255.0 , x_test / 255.0


x_train_scaled = x_train_scaled .reshape(-1, 28, 28, 1)

x_test_scaled = x_test_scaled.reshape(-1, 28, 28, 1)


# one-hot encoding


y_train = tf.keras.utils.to_categorical(y_train, 10)

y_test = tf.keras.utils.to_categorical(y_test, 10)


# visualising first 25 samples :


plt.figure(figsize = (10, 10))
```

```
for i in range(25):
  plt.subplot(5, 5,i+1)
  plt.xticks([      ])
  plt.yticks([      ])
  plt.grid(False)
  plt.imshow(x_train[i].reshape(28,28), cmap=plt.cm.binary)
  plt.xlabel(y_train[i].argmax())
plt.show()


# creating the model


input_img = tf.keras.Input(shape = (28, 28, 1))


Z1 = tf.keras.layers.Conv2D(filters = 32, kernel_size = 4, strides = (1,1), padding =
"same")(input_img)


A1 = tf.keras.layers.ReLU()(Z1)


P1 = tf.keras.layers.MaxPool2D(pool_size = (2,2), strides = (2,2), padding = "same")(A1)


Z2 = tf.keras.layers.Conv2D(filters = 64, kernel_size = 2, strides = (1,1), padding =
"same")(P1)


A2 = tf.keras.layers.ReLU()(Z2)
```

```
P2 = tf.keras.layers.MaxPool2D(pool_size = (2,2), strides = (2,2), padding = "same")(A2)

F1 = tf.keras.layers.Flatten()(P2)

F2 = tf.keras.layers.Dense(128, activation = "relu")(F1)

F2 = tf.keras.layers.Dropout(0.5)(F2)



outputs = tf.keras.layers.Dense(10, activation = "softmax")(F2)

model = tf.keras.Model(inputs = input_img, outputs = outputs)

#model summary

model.summary()

# compiling the model and setting loss function and metric

model.compile( optimizer = "adam", loss = "categorical_crossentropy", metrics =
["accuracy"])

# training the model

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

early_stop = EarlyStopping(monitor = 'val_loss', patience = 5, restore_best_weights = True)
```

```
checkpoint = ModelCheckpoint("best_mnist_cnn.keras", monitor="val_loss", save_best_only
= True)

history = model.fit(x_train_scaled, y_train,

                    epochs = 20,

                    batch_size = 64,

                    validation_split = 0.1,

                    callbacks = [early_stop, checkpoint])


# loading th3e best weights


print(model.load_weights("best_mnist_cnn.keras"))


test_loss, test_acc = model.evaluate(x_test_scaled, y_test)


print("test loss : ", test_loss)

print(f"test accuracy : {test_acc:.4f}")


plt.figure(figsize = (10, 10))


plt.plot(history.history['accuracy'], label = "Train Accuracy ")

plt.title("Model accuracy")

plt.xlabel("Epoch")

plt.ylabel("Accuracy")

plt.legend(loc = "lower right")

plt.show()
```

```
plt.figure(figsize = (10, 10))

plt.plot(history.history['loss'], label = "Loss")

plt.title("Model")

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.legend(loc = "upper right")

plt.show()
```