EXPERIMENT 1: DOCUMENTATION

Delhi Technological University(DTU)

(Department of Computer Science and Technology)

(Deep learning: CO328)

Submitted by:-

Aryan Gahlaut (2K22/CO/102)

6th semester(3rd year)

Submitted to:-

Dr. Anil Parihar

EXPERIMENT 1

Aim:- To train Artificial Neural Network on MNIST dataset to precisely classify handwritten digits.

MNIST dataset: The MNIST (Modified National Institute of Standards and Technology) dataset is a widely used dataset in machine learning, particularly for image classification tasks. It consists of 70,000 grayscale images of handwritten digits (0 to 9), where each image is 28x28 pixels. The dataset is split into two parts: 60,000 images for training and 10,000 images for testing. MNIST is often used as a benchmark to evaluate the performance of different machine learning models, from basic classifiers to advanced deep learning architectures. The simplicity and accessibility of MNIST make it an ideal starting point for beginners and a standard dataset for testing algorithms in computer vision and image processing.

Platform:- Google colab

Google Colab or Colaboratory is a free cloud-based computing platform built by Google to let its users write and execute python code in a web browser without the need for installing any local software.

It builds on the Jupyter Notebooks, which allow you to create and share documents with live code, equations, visualizations, and narrative text.. Colab is especially popular in machine learning, deep learning, for several good reasons.Free access to powerful hardware accelerators, like GPUs and TPUs, which serve a critical purpose of training huge machine learning models and deep-networks buyers, has been made possible only with an increment set on the hardware. It supports several of the most widely used libraries in machine learning, such as TensorFlow, PyTorch, and Keras, to try different algorithms easily.

What is more, is that Colab offers seamless integration with Google Drive, so storing, sharing, and collaborating on a project is quite easy and trouble-free. It is an attractive tool for beginners and hard-core professionals in the realms of data science, machine learning, and deep learning, thanks to its user-friendly interface, strong computational capabilities, and cloud-based architecture.

Libraries used:- The various Deep learning libraries and framework used for tasks are-

1). torch:
This library is the core of PyTorch, providing functionalities that facilitate tensor operations

and deep learning tools. Tensors are the basic data structure in PyTorch and are like NumPy arrays but capable of acceleration using a GPU.

2). torchvision:
Torchvision is an extension of PyTorch that includes various utilities to work with image data. It is equipped with tools to process images, predefined datasets, and pretrained models. You import torchvision.datasets and torchvision.transforms to help with the MNIST dataset and apply the required transformations to images.

3). MNIST (from torchvision.datasets):
This is a predefined dataset of handwritten digits (from 0-9), often used in image classification tasks. It has become a very good standard for use in training and evaluating models.
The MNIST is made up of 60,000 training images and 10,000 test images of 28x28 pixels of image of a handwritten digit.

4). ToTensor (from torchvision.transforms):
ToTensor is a transformation function that converts images (generally, those stored as a PIL Image or NumPy array) into PyTorch tensors. This is important given that inputs and outputs of PyTorch models work on tensors.
ToTensor also works to normalize that image along dimension [0, 1].

5). matplotlib.pyplot:
matplotlib is the library for plotting, while pyplot is a module in that. It is used for the visualization part of image dataset. matplotlib.plot works with graphs, shows images, and data visualizations.
When it comes to MNIST, it can be used to show images of handwritten digits before or after images are converted into tensors.

6). random_split (from torch.utils.data):
This function allows splitting a dataset randomly into non-overlapping subsets. It is used to split the train dataset into training data and validation data.
We also can now specify how much data goes into each subset (for example, 80% training, 20% validation).

7) DataLoader (torch.utils.data):
DataLoader is a PyTorch utility that allows us to load data in chunks or mini-batches. In other words, DataLoader allows the user to iterate through a dataset in batches so that all of

the data do not have to be held in memory at once. This is important for efficiency when training models. DataLoader also has some options for shuffling the data, allowing the model to see the data randomly ordered, which can help improve training.

8) torch.nn: torch.nn
It is a submodule of PyTorch that provides the building blocks for creating and training neural networks. It contains classes for the various layers (e.g., convolutional layers, linear layers, etc.), loss functions (e.g., cross-entropy loss), and optimization utilities. torch.nn would be used more likely by you to structure which neural network you would make (a simple fully connected or convolutional neural network for MNIST classification).

Implementation:

Importing necessary framework and libraries

```
import torch
import torchvision           # contains already functions and libraries to work with
images
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

dataset=MNIST(root='data/',download=True,)                    # downloading training
dataset and already converted into pytorch object
len(dataset)
```

The required libraries are imported . The MNIST dataset is imported from torchvision.datasets . The train data is then downloaded in the root directory folder ‘root’. The length of dataset is then printed which contains 60000 images .

Importing train and test data

```
train_data=MNIST(root='data/', train=True, transform=ToTensor())         # reads the train
data and converts it into pytorch tensor
test_data=MNIST(root='data/', train=False, transform=ToTensor())
len(train_data), len(test_data)
train_data[0]
```

It loads the MNIST dataset and runs through a few important tasks. The code is loading the MNIST dataset and performing a couple of essential tasks. First, it loads the training and test data using the MNIST() function from the torchvision.datasets module. The argument train=True loads 60,000 training images (same as in above code ...but this time it also

converts it into tensors on the go), while train=False loads 10,000 test images. These images are objects of PIL, so the argument transform=ToTensor() is applied to transform them into PyTorch tensors, which are necessary for model training and evaluation. Tensors are the fundamental data structure in PyTorch, and this transformation also scales the pixel values to a range of [0, 1].

The expression len(train_data), len(test_data) will return the number of images in training and test datasets, which for MNIST should be 60 000 and 10 000 respectively. When you access train_data[0] it returns the first image with its corresponding label as a tuple. The image itself is a 28x28 tensor since MNIST images are 28x28 pixels, and the label is a digit between 0 and 9, which represents the digit in the image. For example, train_data[0] might return a tensor of shape [1, 28, 28] (the image) and a label, for instance, 5, meaning the digit appearing in the image.

We can see the first image by unrolling the tensor back into something you can view, like a NumPy array, and plot it with matplotlib.

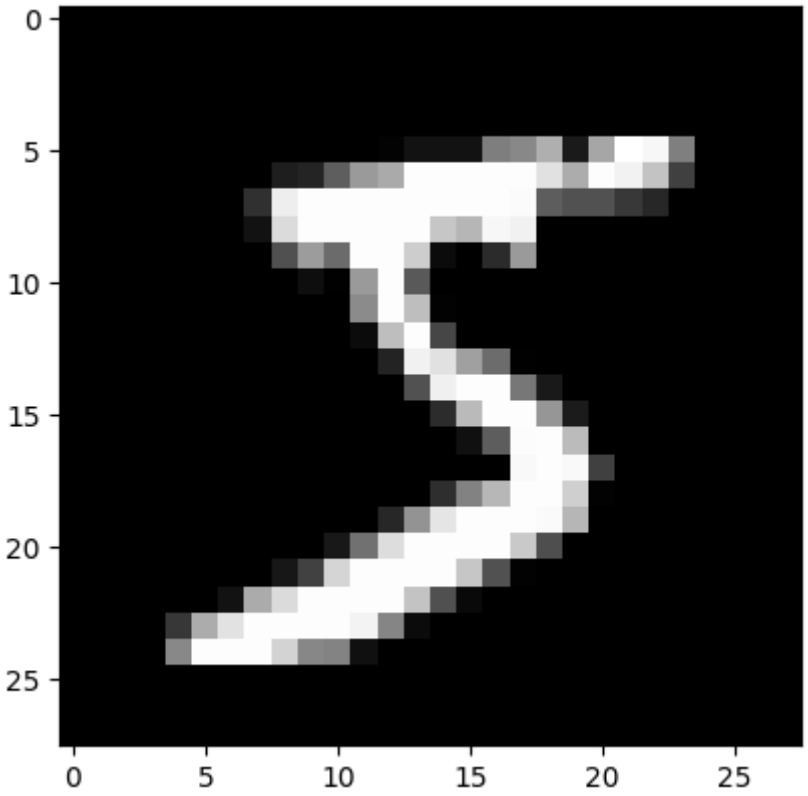
Data Visualization

```
train=MNIST(root='data/', train=True)
test=MNIST(root='data/', train=False)
train[0]

%matplotlib inline
image, label=train[0]
plt.imshow(image, cmap='gray')    # imshow() to visualise the image
print('label:', label)
```

In the code, train = MNIST(root='data/', train=True) loads the train data consisting of 60,000 images from the MNIST dataset, whereas test = MNIST(root='data/', train=False) loads the test data of 10,000 images.

The first image-label pair from the training set is achieved by accessing train[0]. The image is then displayed by using the plt.imshow(image, cmap='gray'), hence presenting the image in grayscale. Finally, the label, i.e., the digit depicted in the image, is printed with print('label:', label). This allows you to see the first image in the dataset and its corresponding label.



Splitting training data into training and validation data

```
from torch.utils.data import random_split
train_ds, val_ds=random_split(train_data, [50000, 10000])#randomly splitting training data into training and validation dataset
len(train_ds), len(val_ds)
```

The code will use random_split from torch.utils.data to randomly split the train_data into two subsets: a training dataset of 50,000 images and a validation dataset of 10,000 images. The function ensures that data is divided in non-overlapping subsets but also maintains randomness. After splitting, len(train_ds) will return 50,000 (for the new training set), and len(val_ds) will return 10,000 (for the validation set), giving you two datasets to train and validate your model with.

Splitting training , testing and validation dataset into mini batches using dataloader

```
from torch.utils.data import DataLoader     #importing dataloader to split the dataset into small batches to feed at once to neural network
batch_size=128
train_dl=DataLoader(train_ds,batch_size,shuffle=True)          # size of each batch is 128
val_dl=DataLoader(val_ds,batch_size , shuffle=True)
test_dl=DataLoader(test_data ,200,shuffle=True)
```

Dataloader is used to split the datasets into groups of mini batches of fixed size. In this case the batch_size is 128. It means that there are 128 input label pairs in each batch. Train_ds which has length of 50000 is split into batches each of size 128 and assigned to train_dl. Exact same thing is done for testing and validating dataset. Shuffle is set to true to randomise the order of image-label pair.

Dynamically flattening the dimensions of each image in train_dl

```
import torch.nn as nn
input_size=784
hidden_size=128
output_size=10

'''for images , labels in train_dl:
  print(images.shape)
  break'''

for images , labels in train_dl:
  images=images.reshape(images.size(0), -1)           # reshaping each batch to flatten
image tensor into 1D tensor dynamically
```

Since we are using neural network , we need to flatten the dimensions of images. The input size is set to 784 , since after flattening the image of dimension 28x28 ...we get a vector of size 784. The neurons in hidden layer will be 128 and in case of output layer ,we need 10 neurons as number of classes to predict are 10 (0 to 9 digits).

Defining Artificial neural network layer using pytorch and python class

```
import torch

class fivelayernn(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(fivelayernn, self).__init__()



        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, hidden_size)
        self.layer4 = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()   # for hidden layers



    def forward(self, x):

        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        x = self.relu(self.layer3(x))
```

```
        x = self.layer4(x)
        return x



model=fivelayernn(input_size,hidden_size,output_size)
```

Here we have defined a class of five layered neural network (including input layer)

Layer1 will have 784 inputs neurons corresponding to each pixel of images and the output of the neurons of first layers are passed as input to hidden layer2 , having 128 neurons , after applying relu activation function.

Relu(x)=max(0,x).

Relu activation function simply clips the negative part , introducing non linearity to model, allowing it to learn complex non linear pattern as well.

Similarily , the output of first hidden layer is feed to second hidden layer3 , after applying relu.

Finally , the output of second hidden layer is feed into the final layer , again after applying relu activation function. The final layer contains 10 neurons each corresponding to digit from 0 to 9. No activation function is applied after final layer ....as crossentropyloss() internally applies softmax activation function to converts raw logits to probablity distribution to compute loss. All this forward propagation is defined in forward function in the class.

Training the model

```
loss_fn=nn.CrossEntropyLoss()          # for calculating loss in multi class
classification

opt=torch.optim.SGD(model.parameters(),lr=1e-3)       # optimiser (Stochastic gradient
optimizer)

def training(epoch ,model ,loss_fn ,opt,train_dl):
  for e in range(epoch):
    model.train()                                          # setting model to training
mode
    for images , labels in train_dl:
      images = images.reshape(images.size(0), -1)     # checking only
      opt.zero_grad()
      output=model(images)
      loss=loss_fn(output,labels)
      loss.backward()                                        # Back propagation
      opt.step()
      if (e+1) % 10 == 0:
        print('Epoch [{}/{}], Loss: {:.4f}'.format(e+1, epoch, loss.item()))
```

```
training(1500,model,loss_fn,opt,train_dl)
```

This is a training procedure for neural network. It defines a loss function, loss_fn, as the cross-entropy loss function typically used for multi-class classification problems. Then it defines an optimizer, opt, as stochastic gradient descent with a learning rate of 0.001, which helps in updating the model's parameters during training.

The function training accepts several arguments, including the number of epochs, the model to be trained, the loss function, the optimizer, and the training data loader, train_dl. Within the function, model.train() sets the model into training mode. Then, the training loop runs over the number of epochs. For each iteration in an epoch, it runs the training data in batches. Images are make sure to be flattened using the reshape command, and then the gradients of the optimizer are made zero using opt.zero_grad() so that it does not accumelate over these number of epochs . It makes predictions on images (output = model(images)) and computes loss by feeding predicted output along with true labels into the loss_fn (loss = loss_fn(output, labels)).

The loss is backpropagated through the network using loss.backward(), and the optimizer updates the model's parameters with opt.step(). Every 10 epochs, the current epoch and the loss are printed to track progress.

Finally, the training() function is called to train the model for 1500 epochs.

```
Streaming output truncated to the last 5000 lines.

Epoch [1380/1500], Loss: 0.0105

Epoch [1380/1500], Loss: 0.0083

Epoch [1380/1500], Loss: 0.0235

.

.

.

.

Epoch [1500/1500], Loss: 0.0048

Epoch [1500/1500], Loss: 0.0041

Epoch [1500/1500], Loss: 0.0035
```

Testing the Artificial Neural Network

```
for images , labels in test_dl:
  images=images.reshape(images.size(0), -1)
loss_fn=nn.CrossEntropyLoss()
```

```
def test(model , test_dl , loss_fn):                                    # Function to evaluate
the model
  model.eval()                                                  # setting model to evaluation
mode
  correct=0
  total=0
  test_loss=0
  with torch.no_grad():
    for images , labels in test_dl:
      images=images.reshape(images.size(0),-1)
      output=model(images)
      loss=loss_fn(output ,labels)

      test_loss+=loss.item()
      _, predicted=torch.max(output ,1)
      total += labels.size(0)
      correct += (predicted == labels).sum().item()

  accuracy = 100 * correct / total
  average_loss = test_loss / len(test_dl)

  print(f'Test Loss: {average_loss:.4f}')                                  #Printing
average loss per batch
  print(f'Test Accuracy: {accuracy:.2f}%')                               # printing
accuracy - total number of correct predictions/ total number of total cases

test(model, test_dl, loss_fn)
```

This is a test function to test the performance of a trained model on the test dataset. Inside this function, the model is set to evaluation mode using model.eval(), which adjusts behaviors like dropout and batch normalization that are only active during training. The function then initializes variables to track the number of correct predictions (correct), the total number of samples (total), and the cumulative test loss (test_loss).

The function iterates over the test data loader (test_dl), reshaping images to match the input shape that the model expects. The model then makes predictions on the reshaped images. The loss is calculated using loss_fn, and the total test loss is updated.

For each batch, the predicted class labels are compared to the true labels, and the number of correct predictions is accumulated. After all batches are processed, the accuracy is calculated as the percentage of correct predictions out of the total number of samples.

Finally, the average test loss per test batch and accuracy are printed, giving insight into how well the model performs on the test set. The function is called with the trained model, test_dl (test data loader), and the loss function (loss_fn) to evaluate the model's performance.

Validation of the model

```
for images , labels in val_dl:
```

```
  images=images.reshape(images.size(0), -1)
def validat(val_dl , model):
  for image , label in val_dl:
    image=image.reshape(image.size(0),-1)
    pred=model(image)
    _, predicted=torch.max(pred ,1)           # to igone the logit valueof highest
probablity class
    print('The predicted values are (in tensor form): ', predicted)
    print('\n')
    print('Actual values are (in tensor form): ', label)
    break
validat(val_dl,model)
```

This code snippet defines a function validat() for the validation of model predictions on a batch of data from the validation dataset, val_dl. In this function, each image that the validation data loader processes gets reshaped to the required input shape before being passed through the model for producing predictions. The torch.max() function is now used to select the predicted class with the highest probability with the logit values of other classes being ignored. The predicted values along with the actual labels from the validation set are printed out for comparison as well.

After defining the validat() function, we call it with val_dl, which is the validation data loader, and model, which is the trained model. This will output the predicted and actual labels for a single batch in the validation set so that you can check how well the model is performing on that batch. The break statement ensures that only one batch is processed for this validation check.

Results:

Training result after 1500 epoch:

The loss after each 10 epochs are printed below.

Streaming output truncated to the last 5000 lines.

Epoch [1380/1500], Loss: 0.0105

Epoch [1380/1500], Loss: 0.0083

Epoch [1380/1500], Loss: 0.0235

Epoch [1380/1500], Loss: 0.0127

.

.

.

.

.

Epoch [1500/1500], Loss: 0.0284

Epoch [1500/1500], Loss: 0.0048

Epoch [1500/1500], Loss: 0.0041

Epoch [1500/1500], Loss: 0.0035

Testing result:

Test Loss: 0.0956

Test Accuracy: 97.52%

Total test loss for test_dl which has batch_size of 128 comes out to be 0.0956.

Accuracy=(Total number of correct prediction)/(Total number of labels in test_dl) . It denotes ratio of correct prediction to total number of labels. The accuracy after training comes out to be 97.53%.

Conclusion

In conclusion, by implementing a 5-layer neural network with ReLU activation functions and using stochastic gradient descent (SGD) as the optimizer, I have achieved an impressive accuracy of **97.52%** on the MNIST dataset. This demonstrates the effectiveness of a relatively simple neural network architecture for image classification tasks. Despite the model's simplicity, it performs well on this benchmark dataset.

.ipynb executable file: MNIST digit classification

Github link: Github link