Pattern Recognition and Machine Learning(CSL2050)

PA6-Report

Soumen Kumar
Roll No-B22ES006

Google Colab Notebook Link

Access my notebook: PA6-B22ES006.ipynb.

Neural Network

1 Task0:-Data Downloading and Splitting

The MNIST dataset is a widely used dataset for image classification tasks, consisting of 60,000 training images and 10,000 test images of handwritten digits from 0 to 9. Firstly, we download the MNIST dataset using PyTorch's torchvision library, and split the data into train, validation, and test sets, and apply various data augmentation techniques.

1.1 Import Statements

We begin by importing the necessary libraries and setting up the device (CPU or GPU) for computation.

1.2 Data Augmentation

We define two sets of transformations: one for the training set and another for the validation and test sets. The training set transformations include RandomRotation, RandomCrop, ToTensor, and Normalize, while the validation and test sets only involve ToTensor and Normalize.

```python
# Defining transformations for data augmentation (train set)
train_transform = transforms.Compose([
   transforms.RandomRotation(29),  # Randomly rotate images by up to 29 degrees
   transforms.RandomCrop(28, padding=4),  # Randomly crop images with padding of 4 pixels
   transforms.ToTensor(),  # Convert images to PyTorch tensors
   transforms.Normalize((0.1307,), (0.3081,))  # Normalize the images
])

# Defining transformation for validation and test sets (no augmentation)
test_transform = transforms.Compose([
   transforms.ToTensor(),
   transforms.Normalize((0.1307,), (0.3081,))
])
```

1.3 Dataset Loading and Splitting

We load the MNIST dataset from the torchvision library, applying the respective transformations for the training and test sets. We then split the training set into train and validation sets using a validation ratio of 0.1.

2 Task1:-MNIST Dataset Visualization and Creating Data Loaders

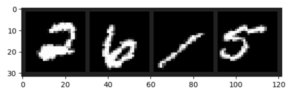
We extend our work with the MNIST dataset by visualizing sample images from each class and creating data loaders for the training, validation, and test sets.
Data visualization is essential for understanding the dataset and ensuring the correct application of transformations.
Data loaders facilitate efficient loading and batching of data during the training and evaluation process.

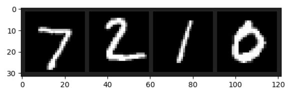
2.1 Plotting Sample Images

We define a function ‘imshow‘ to display an image after reversing the normalization applied during preprocessing.

```python
def imshow(img):
    img = img * 0.3081 + 0.1307  # Reverse normalization
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

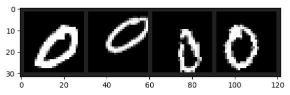
We then plot four random images from the training and test sets, along with their corresponding labels.

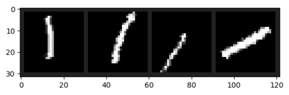

(a) Training Set

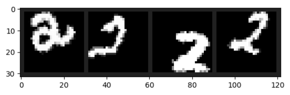

(b) Testing Set

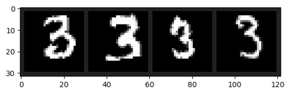
2.2 Visualizing Images from Each Class

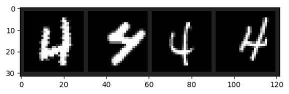
We define a function ‘*printclassimages*‘ to display four images from each class in the dataset. This function iterates over the data loader and collects four examples of each class.

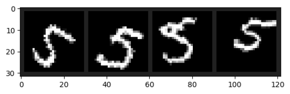

(a) 0

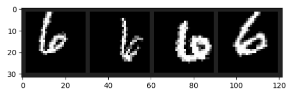

(b) 1

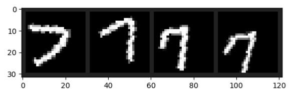

(c) 2

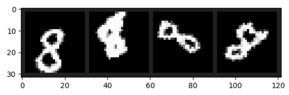

(d) 3


(e) 4


(f) 5


(g) 6


(h) 7


(i) 8

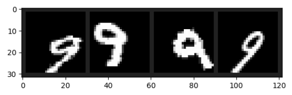

(j) 9

Figure 2: Images of each class

2.3 Creating Data Loaders

We create data loaders for the training, validation, and test sets using PyTorch’s ‘DataLoader‘ class. The data loaders handle batching and shuffling of data during training and evaluation.

```python
# Defining data loaders for train, validation, and test sets
train_loader = torch.utils.data.DataLoader(train_set, batch_size=64, sampler=train_sampler, shuffle=False)
val_loader = torch.utils.data.DataLoader(train_set, batch_size=64, sampler=val_sampler, shuffle=False)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=64, shuffle=False)
```

3 Task2:- 3-Layer MLP using PyTorch

In this we implement a 3-layer Multi-Layer Perceptron (MLP) using PyTorch for the task of classifying handwritten digits from the MNIST dataset. We define the MLP architecture, forward pass, and calculate the number of trainable parameters.

3.1 MLP Class Definition

We define the MLP class as a subclass of nn.Module from PyTorch’s nn module.

3.1.1 init Method

The init method initializes the linear layers of the network based on the provided input, hidden, and output sizes.

```python
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_size)
```

3.1.2 Forward Pass

The ‘forward‘ method defines the forward pass of the MLP. It first reshapes the input tensor to a flattened vector using the ‘num flat features‘ helper function. Then, it applies ReLU activations and linear transformations
to the input, passing through the three linear layers.

```
    def forward(self, x):
        x = x.view(-1, self.num_flat_features(x))
        x = nn.functional.relu(self.fc1(x))
        x = nn.functional.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # All dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

3.2 MLP Initialization and Parameter Counting

We initialize the MLP with the appropriate input, hidden, and output sizes for the MNIST dataset. Then, we create an instance of the ‘MLP‘ class and calculate the total number of trainable parameters.

```
# Dimensions of the MLP
input_size = 28 * 28  # MNIST image size
hidden_size1 = 128
hidden_size2 = 64
output_size = 10  # Classes in MNIST

# Instance of the MLP
model = MLP(input_size, hidden_size1, hidden_size2, output_size)

# The number of trainable parameters of the model
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Number of trainable parameters:", total_params)
```

4 Task3:-Training 3-Layer MLP on MNSIT Dataset

In this, we train a 3-layer Multi-Layer Perceptron (MLP) on the MNIST dataset using PyTorch. We utilize the Adam optimizer and CrossEntropyLoss as the loss function. During training, we evaluate the model's performance on the validation set after each epoch and save the best model based on the validation accuracy. Additionally, we log the training and validation losses and accuracies for each epoch.

4.1 Code

4.1.1 Setting up the Loss Function and Optimizer

We define the loss function and optimizer for the training process.

4.1.2 Training Loop

We set the number of epochs to 5 and initialize lists to store the training and validation losses and accuracies for each epoch.
The training loop iterates over the specified number of epochs. For each epoch, we perform the following steps:
1. Set the model to training mode using ‘model.train()‘.
2. Iterate over the training data loader and perform forward and backward passes.
3. Compute the training loss and accuracy for the current epoch.
4. Set the model to evaluation mode using ‘model.eval()‘.
5. Iterate over the validation data loader and compute the validation loss and accuracy without updating the model parameters.
6. Print the training and validation losses and accuracies for the current epoch.

7. If the validation accuracy is higher than the best validation accuracy so far, update the ‘best val acc‘ and ‘best model‘ variables.
8. Save the best model to a file named ‘best model.pth‘.

4.2 Observations

4.2.1 Decreasing Loss

Both the training and validation losses are decreasing with each epoch. This is an expected behavior, indicating that the model is learning and improving its performance on the training and validation data.

4.2.2 Increasing Accuracy

The training and validation accuracies are increasing with each epoch, which is a positive sign. The model is becoming better at classifying the handwritten digits correctly.

4.2.3 Validation Accuracy Higher than Training Accuracy

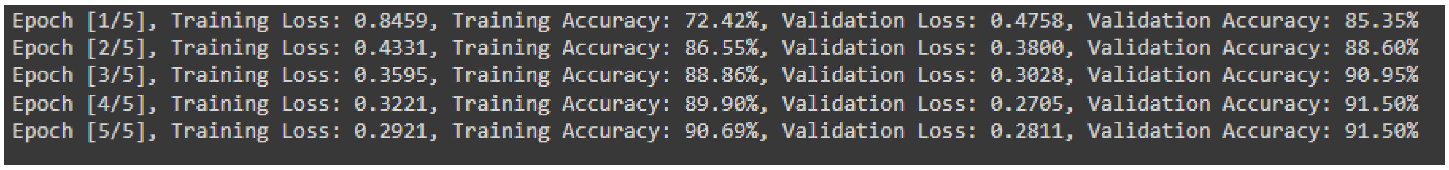
It is interesting to note that the validation accuracy is consistently higher than the training accuracy. This is an unusual observation, as typically, the training accuracy is expected to be higher than the validation accuracy due to the model's tendency to overfit to the training data.

4.2.4 Small Gap between Training and Validation Metrics

The gap between the training and validation losses and accuracies is relatively small. This suggests that the model is not overfitting significantly to the training data and is generalizing well to the validation data.

4.2.5 Convergence

The training and validation metrics seem to be converging towards the end of the training process. The improvements in loss and accuracy become smaller in the later epochs, indicating that the model is approaching its optimal performance on the given data.



```
Epoch [1/5], Training Loss: 0.8459, Training Accuracy: 72.42%, Validation Loss: 0.4758, Validation Accuracy: 85.35%
Epoch [2/5], Training Loss: 0.4331, Training Accuracy: 86.55%, Validation Loss: 0.3800, Validation Accuracy: 88.60%
Epoch [3/5], Training Loss: 0.3595, Training Accuracy: 88.86%, Validation Loss: 0.3028, Validation Accuracy: 90.95%
Epoch [4/5], Training Loss: 0.3221, Training Accuracy: 89.90%, Validation Loss: 0.2705, Validation Accuracy: 91.50%
Epoch [5/5], Training Loss: 0.2921, Training Accuracy: 90.69%, Validation Loss: 0.2811, Validation Accuracy: 91.50%
```

5 Task4:-Visualizing Predictions and Performance Metrics for the MNIST MLP

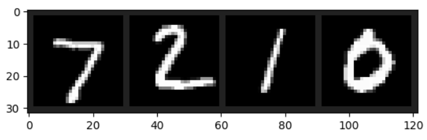
In this, we explore various techniques for visualizing the predictions and performance metrics of the 3-layer Multi-Layer Perceptron (MLP) trained on the MNIST dataset. Specifically, we visualize correct and incorrect predictions, plot the loss and accuracy curves over epochs, and examine correct and incorrect predictions class-wise for both the training and testing datasets.

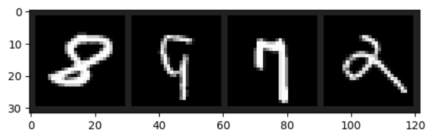
5.1 Visualizing Correct and Incorrect Predictions

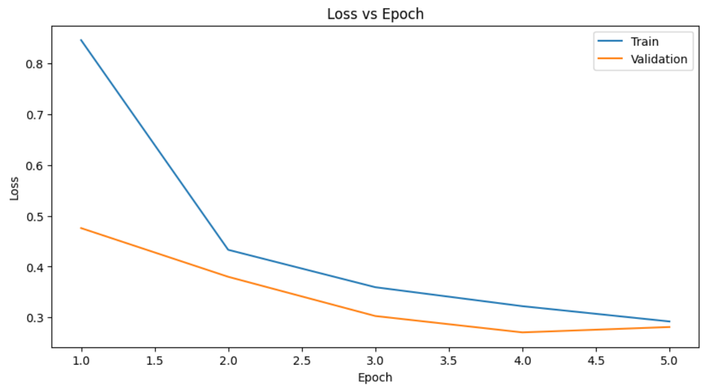
The ‘visualize predictions‘ function takes a data loader and the trained model as input. It iterates over the data loader, makes predictions using the model, and separates the input images into two lists: correct predictions and incorrect predictions. The function then displays a grid of four images from each list, allowing visual inspection of the model's performance.

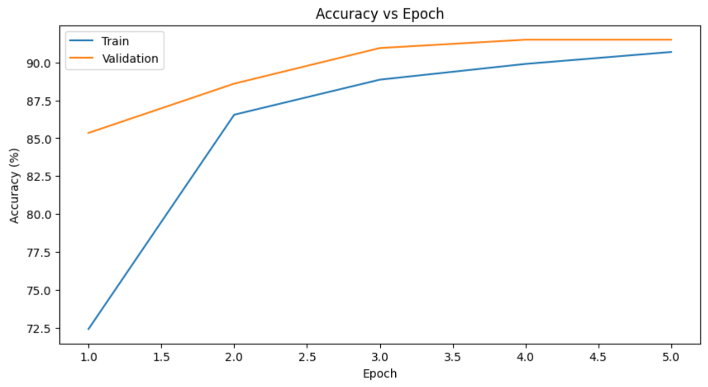
5.2 Plotting Loss and Accuracy Curves

The ‘plot metrics‘ function takes the lists of training and validation losses and accuracies as input and generates two separate plots: one for loss versus epoch, and another for accuracy versus epoch. This allows for visual analysis of the model's performance during training and validation.


(a) Correct Predictions

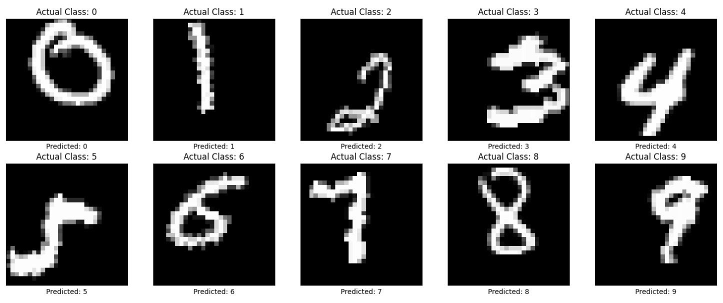

(b) Incorrect Predictions



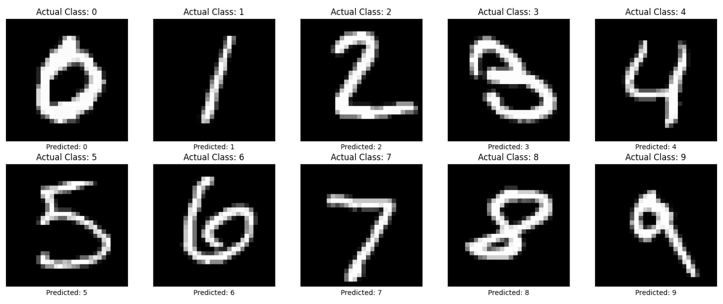


5.3 Visualizing Class-wise Correct and Incorrect Predictions

The ‘print classwise predictions‘ function takes a data loader and the trained model as input. It iterates over the data loader, makes predictions using the model, and separates the input images into two lists of lists: one for correct predictions and one for incorrect predictions, with each inner list corresponding to a specific class label. The function then displays a grid of four images for each class, showing correct and incorrect predictions separately.
For Example:



(a) Correct Predictions



(b) Incorrect Predictions