Lab #10: Image Classification with PyTorch

Fiseha B.

Objective

To understand and implement a basic image classification pipeline using the PyTorch library. This lab will cover data loading, preprocessing, model definition, training, and evaluation.

Prerequisites

- Basic understanding of Python programming.
- Familiarity with fundamental machine learning concepts (e.g., classification, training, testing).
- Basic knowledge of PyTorch tensors and modules (recommended).
- Access to a Python environment with PyTorch, torchvision, and matplotlib installed. You can install them using pip:

```
    pip install torch torchvision matplotlib  torchsummary
```

Listing 1: Installation

Dataset

We will use the CIFAR-10 dataset, a widely used dataset for image classification. It consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

Lab Structure

1. **Setting up the Environment and Loading Data:**
   - Import necessary libraries.
   - Download and load the CIFAR-10 dataset using `torchvision`.
   - Explore the dataset structure and visualize some images.
2. **Data Preprocessing and Augmentation:**
   - Understand the need for data preprocessing (normalization).
   - Apply necessary transformations to the images using `torchvision.transforms`.
   - (Optional) Implement basic data augmentation techniques.
   - Create DataLoaders for efficient batching of the data.
3. **Defining the Neural Network Model:**
   - Implement a simple Convolutional Neural Network (CNN) using `torch.nn`.
   - Understand the basic building blocks of a CNN (Conv2d, ReLU, MaxPool2d, Linear).
   - Define the forward pass of the model.
4. **Training the Model:**
   - Define the loss function (e.g., Cross-Entropy Loss).
   - Choose an optimizer (e.g., Adam, SGD).
   - Implement the training loop:

  - Iterate through the training DataLoader.
  - Perform the forward pass.
  - Calculate the loss.
  - Perform backpropagation to compute gradients.
  - Update model parameters using the optimizer.
- Monitor training progress (e.g., loss and accuracy on a validation set).

5. **Evaluating the Model:**
   - Load the test dataset.
   - Implement the evaluation loop:
     - Iterate through the test DataLoader.
     - Perform the forward pass.
     - Calculate the accuracy on the test set.
   - (Optional) Visualize predictions and understand model performance on different classes.

Detailed Steps

1. Setting up the Environment and Loading Data:

```python
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# Check if CUDA is available (for GPU acceleration)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Define the path to store the dataset
data_path = './data'

# Download and load the training dataset
trainset = torchvision.datasets.CIFAR10(root=data_path, train=True,
                                        download=True, transform=transforms.ToTensor())
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

# Download and load the test dataset
testset = torchvision.datasets.CIFAR10(root=data_path, train=False,
                                       download=True, transform=transforms.ToTensor())
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

# Define the class labels
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')

# Function to show some images
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# Get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# Show images
imshow(torchvision.utils.make_grid(images))
# Print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))
```

Listing 2: Setting up and Loading Data

Exercises:

- Print the size of the `trainset` and `testset`.
- What is the shape of a single image tensor in the loaded dataset?
- Experiment with different `batch_size` values in the `DataLoader`. What are the implications of changing the batch size?

2. Data Preprocessing and Augmentation:

```python
# Define transformations
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]) # Normalize to [-1, 1]

# Load the datasets with the defined transformations
trainset = torchvision.datasets.CIFAR10(root=data_path, train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root=data_path, train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

# (Optional) Data Augmentation Example: Random Horizontal Flip
transform_augmented = transforms.Compose(
    [transforms.RandomHorizontalFlip(),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset_augmented = torchvision.datasets.CIFAR10(root=data_path, train=True,
                                                  download=True, transform=transform_augmented)
trainloader_augmented = torch.utils.data.DataLoader(trainset_augmented, batch_size=4,
                                                    shuffle=True, num_workers=2)

# Visualize a few augmented images (optional)
dataiter_augmented = iter(trainloader_augmented)
images_augmented, _ = next(dataiter_augmented)
imshow(torchvision.utils.make_grid(images_augmented))
```

Listing 3: Data Preprocessing and Augmentation

Exercises:

- Explain the purpose of the `transforms.Normalize` operation. What are the mean and standard deviation values used?
- Research and implement other data augmentation techniques from `torchvision.transforms` (e.g., `RandomCrop`, `RandomRotation`). How might these augmentations improve model performance?
- Why is data augmentation typically applied only to the training set and not the test set?

3. Defining the Neural Network Model:

```python
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)  # Input channels: 3 (RGB), Output channels: 6, Kernel
            size: 5x5
        self.pool = nn.MaxPool2d(2, 2)   # Kernel size: 2x2, Stride: 2
        self.conv2 = nn.Conv2d(6, 16, 5) # Input channels: 6, Output channels: 16, Kernel size:
             5x5
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # Input features: 16 * (32-4)/2 - 4)/2 = 16 * 5 *
             5, Output features: 120
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)    # Output features: 10 (number of classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)          # Flatten the tensor for the fully connected layers
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net().to(device) # Move the model to the specified device (GPU if available)
# Generate summary
inputs = (3,32,32)
summary(net, input_size=inputs)
```

Listing 4: Defining the Neural Network Model

Exercises:

- Calculate the output shape after each layer in the `forward` pass for an input image of size 3x32x32.
- Experiment with different numbers of convolutional layers, filter sizes, and fully connected layer sizes. How do these changes affect the model's complexity and the number of parameters?
- Research and implement other activation functions (e.g., `nn.Sigmoid`, `nn.Tanh`) and observe their impact on training.

4. Training the Model:

```
import torch.optim as optim

# Define the loss function
criterion = nn.CrossEntropyLoss()

# Choose an optimizer
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) # Stochastic Gradient Descent

# Training loop
num_epochs = 2
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # Print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')

# Save the trained model (optional)
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

Listing 5: Training the Model

Exercises:

- Experiment with different optimizers (e.g., `optim.Adam`) and learning rates. How do these choices affect the training process and the final model performance?
- Implement a validation loop to monitor the model's performance on a separate validation set during training. This helps in detecting overfitting.
- Increase the number of training epochs. Observe how the training loss and potentially validation accuracy change. When might you stop training?

5. Evaluating the Model:

```python
# Load the saved model (if you saved it)
net = Net().to(device)
net.load_state_dict(torch.load(PATH))

# Evaluate on the test set
correct = 0
total = 0
with torch.no_grad(): # Disable gradient calculation during evaluation
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct / total:.2f} %')

# Class-wise accuracy
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # Collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1

# Print class-wise accuracy
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.2f} %')

# Visualize some predictions (optional)
dataiter_test = iter(testloader)
images_test, labels_test = next(dataiter_test)
imshow(torchvision.utils.make_grid(images_test))
print('GroundTruth: ', ' '.join(f'{classes[labels_test[j]]:5s}' for j in range(4)))

outputs_test = net(images_test.to(device))
_, predicted_test = torch.max(outputs_test, 1)
print('Predicted:   ', ' '.join(f'{classes[predicted_test[j]]:5s}' for j in range(4)))
```

Listing 6: Evaluating the Model

Exercises:

- What does `torch.no_grad()` do? Why is it important during evaluation?
- Analyze the class-wise accuracy. Are there any classes where the model performs significantly better or worse? Can you hypothesize why?
- Modify the code to predict the class probabilities for a few test images.
- (Advanced) Explore techniques to improve the model's performance, such as adding more convolutional layers, using different network architectures (e.g., ResNet, VGG), or employing regularization techniques (e.g., dropout, batch normalization).

Further Exploration

- Try training on a different image classification dataset available in `torchvision.datasets`.
- Implement techniques for visualizing the learned filters in the convolutional layers.
- Explore the use of pre-trained models (transfer learning) for image classification tasks with smaller datasets.

This lab material provides a foundational understanding of image classification using PyTorch. By completing the exercises and exploring the suggested extensions, you willgain practical experience in building and evaluating deep learning models for image recognition. Remember to experiment and have fun!