Lab Assignment 10

Objective

Your task is to classify images from the CIFAR-10 dataset into 10 different classes (e.g., airplane, automobile, bird, etc.) using a custom implementation of the VGGNet architecture in PyTorch. The goal is to achieve high accuracy on both the validation and test datasets while learning the process of implementing deep learning models from scratch.

Guidelines and Hints:

Framework

Use PyTorch to implement the model, as it provides easy-to-use tools for building, training, and testing deep learning models.

Dataset

CIFAR-10 consists of 60,000 images (32x32 in size) categorized into 10 classes, with 50,000 images for training and 10,000 images for testing. The dataset is available in torchvision.datasets.CIFAR10.

Approach

- Preprocess the dataset (resize images, normalize, apply data augmentation).
- Implement the VGGNet architecture (VGG16) using torch.nn.
- Use appropriate training techniques such as dropout and data augmentation to reduce overfitting.
- Train the model using cross-entropy loss and an optimizer of your choice (e.g., SGD, Adam).
- Evaluate the model on the validation dataset after each epoch and test it after training.

Pseudo Code

Here’s a step-by-step outline to guide your implementation:

- Import Libraries:
    - Import necessary PyTorch libraries like torch, torchvision, torch.nn, and torch.optim.
- Data Loading and Preprocessing:
    - Define transformations using torchvision.transforms for training and testing datasets.
    - Load the CIFAR-10 dataset using torchvision.datasets.CIFAR10 and split it into training, validation, and test datasets.
    - Use torch.utils.data.DataLoader to create data loaders for each dataset.

- Define the VGGNet Model:
  - Create a class VGG16_NET that inherits from torch.nn.Module.
  - Define convolutional layers, max pooling layers, and fully connected layers following the VGG16 architecture.
  - Use nn.ReLU for activation, nn.MaxPool2d for pooling, and nn.Linear for fully connected layers.
- Model Training:
  - Define a loss function using torch.nn.CrossEntropyLoss.
  - Use an optimizer like torch.optim.Adam or torch.optim.SGD.
  - Iterate over the training dataset for multiple epochs. For each batch:
  - Move the data to the device (CPU/GPU).
  - Forward propagate the data through the model.
  - Compute the loss and backpropagate to update the weights.
  - Log training progress.
- Model Evaluation:
  - After each epoch, evaluate the model on the validation dataset.
  - Use `torch.no_grad()` to avoid computing gradients during evaluation.
  - Compute accuracy by comparing predictions with true labels.
- Test the Model:
  - Evaluate the final trained model on the test dataset and report the test accuracy.

Hints:

- Transformations: Use `torchvision.transforms` to resize images to 224x224 and normalize pixel values using mean and standard deviation of ImageNet (as VGG was trained on ImageNet). You can also apply augmentations like `RandomHorizontalFlip`.

- Data Loading: Use `torch.utils.data.DataLoader` to create data loaders. Set `shuffle=True` for training data to ensure batches are randomized.

- Training Loop: Use the following functions for essential operations:
  - `model(images)` for forward propagation.
  - `loss.backward()` for backpropagation.
  - `optimizer.step()` to update model parameters.
- Model Evaluation: Use `outputs.max(1)` to get the class with the highest probability from the model's output.

- Device Handling: Check for GPU availability using torch.cuda.is_available() and move data and models to the appropriate device using .to(device).

In [ ]:
```
### WRITE CODE HERE ###
import torch
import torch.nn as nn
import torch.optim as optim
```

```python
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.utils.data import random_split

# Define data transformations
transform_train = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.7),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

transform_test = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# Load CIFAR-10 dataset and split into train and validation sets
torch.manual_seed(2024)
train = torchvision.datasets.CIFAR10("data/", train=True, download=True,
                                     transform=transform_train)


val_size = 10000
train_size = len(train) - val_size
train, val = random_split(train, [train_size, val_size])

test = torchvision.datasets.CIFAR10("data/", train=False, download=True,
                                    transform=transform_test)

# Create data loaders
BATCH_SIZE = 64
train_loader = DataLoader(train, batch_size=BATCH_SIZE, shuffle=True)

val_loader = DataLoader(val, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test, batch_size=8, shuffle=False)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to data/cifar-10-python.tar.gz
100%|██████████| 170M/170M [00:03<00:00, 44.1MB/s]
Extracting data/cifar-10-python.tar.gz to data/
Files already downloaded and verified
```

In [ ]:

```python
# Define the VGG16-based model
import torch.nn.functional as F

class VGG16_NET(nn.Module):
    def __init__(self):
        super(VGG16_NET, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=64,
                               kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128,
                               kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=128,
                               kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256,
                               kernel_size=3, padding=1)
        self.conv6 = nn.Conv2d(in_channels=256, out_channels=256,
                               kernel_size=3, padding=1)
```

```
        self.conv7 = nn.Conv2d(in_channels=256, out_channels=256,
                               kernel_size=3, padding=1)
        self.conv8 = nn.Conv2d(in_channels=256, out_channels=512,
                               kernel_size=3, padding=1)
        self.conv9 = nn.Conv2d(in_channels=512, out_channels=512,
                               kernel_size=3, padding=1)
        self.conv10 = nn.Conv2d(in_channels=512, out_channels=512,
                                kernel_size=3, padding=1)
        self.conv11 = nn.Conv2d(in_channels=512, out_channels=512,
                                kernel_size=3, padding=1)
        self.conv12 = nn.Conv2d(in_channels=512, out_channels=512,
                                kernel_size=3, padding=1)
        self.conv13 = nn.Conv2d(in_channels=512, out_channels=512,
                                kernel_size=3, padding=1)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc14 = nn.Linear(25088, 4096)
        self.fc15 = nn.Linear(4096, 4096)
        self.fc16 = nn.Linear(4096, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.maxpool(x)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.maxpool(x)
        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = F.relu(self.conv7(x))
        x = self.maxpool(x)
        x = F.relu(self.conv8(x))
        x = F.relu(self.conv9(x))
        x = F.relu(self.conv10(x))
        x = self.maxpool(x)
        x = F.relu(self.conv11(x))
        x = F.relu(self.conv12(x))
        x = F.relu(self.conv13(x))
        x = self.maxpool(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc14(x))
        x = F.dropout(x, 0.5)
        x = F.relu(self.fc15(x))
        x = F.dropout(x, 0.5)
        x = self.fc16(x)
        return x
```

In [ ]:

```
# Initialize the model and move it to GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = VGG16_NET()
model = model.to(device=device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

# Training Loop
num_epochs = 3
for epoch in range(num_epochs):
    loss_var = 0
    for idx, (images, labels) in enumerate(train_loader):
        images = images.to(device=device)
        labels = labels.to(device=device)
        optimizer.zero_grad()
        scores = model(images)
```

```
        loss = criterion(scores, labels)
        loss.backward()
        optimizer.step()
        loss_var += loss.item()
        if idx % 64 == 63:
            print(f'Epoch [{epoch + 1}/{num_epochs}] ||
                  Step [{idx + 1}/{len(train_loader)}] ||
                     Loss:{loss_var / (idx+1)}')
    print(f"Loss at epoch {epoch + 1} || {loss_var / len(train_loader)}")

    with torch.no_grad():
        correct = 0
        samples = 0
        for idx, (images, labels) in enumerate(val_loader):
            images = images.to(device=device)
            labels = labels.to(device=device)
            outputs = model(images)
            _, preds = outputs.max(1)
            correct += (preds == labels).sum()
            samples += preds.size(0)
        print(f"Validation accuracy {float(correct) /
            float(samples) * 100:.2f} percentage ||
              Correct {correct} out of {samples} samples")
```

```
Epoch [1/3] || Step [64/625] || Loss:2.259603075683117
Epoch [1/3] || Step [128/625] || Loss:2.1456460868939757
Epoch [1/3] || Step [192/625] || Loss:2.056329576919476
Epoch [1/3] || Step [256/625] || Loss:1.9933002782054245
Epoch [1/3] || Step [320/625] || Loss:1.9439113698899746
Epoch [1/3] || Step [384/625] || Loss:1.9061478913451235
Epoch [1/3] || Step [448/625] || Loss:1.8702877235731907
Epoch [1/3] || Step [512/625] || Loss:1.841114955721423
Epoch [1/3] || Step [576/625] || Loss:1.8104227226641443
Loss at epoch 1 || 1.790208109664917
Validation accuracy 45.58 percentage || Correct 4558 out of 10000 samples
Epoch [2/3] || Step [64/625] || Loss:1.4544031880795956
Epoch [2/3] || Step [128/625] || Loss:1.458205558359623
Epoch [2/3] || Step [192/625] || Loss:1.4394093609104555
Epoch [2/3] || Step [256/625] || Loss:1.4279775535687804
Epoch [2/3] || Step [320/625] || Loss:1.4046219799667596
Epoch [2/3] || Step [384/625] || Loss:1.3835333770451446
Epoch [2/3] || Step [448/625] || Loss:1.3686555219548089
Epoch [2/3] || Step [512/625] || Loss:1.348092781379819
Epoch [2/3] || Step [576/625] || Loss:1.3325752795984347
Loss at epoch 2 || 1.3182448407173157
Validation accuracy 58.96 percentage || Correct 5896 out of 10000 samples
Epoch [3/3] || Step [64/625] || Loss:1.0976725611835718
Epoch [3/3] || Step [128/625] || Loss:1.06377385975793
Epoch [3/3] || Step [192/625] || Loss:1.049402781451742
Epoch [3/3] || Step [256/625] || Loss:1.0381128629669547
Epoch [3/3] || Step [320/625] || Loss:1.024192050471902
Epoch [3/3] || Step [384/625] || Loss:1.0159851419739425
Epoch [3/3] || Step [448/625] || Loss:1.0030659842970115
Epoch [3/3] || Step [512/625] || Loss:0.9929894460365176
Epoch [3/3] || Step [576/625] || Loss:0.9823403326380584
Loss at epoch 3 || 0.9753584115982056
Validation accuracy 68.77 percentage || Correct 6877 out of 10000 samples
```

In [ ]:

```
# Test the model on the test dataset
correct = 0
samples = 0
for idx, (images, labels) in enumerate(test_loader):
    images = images.to(device=device)
```

```
    labels = labels.to(device=device)
    outputs = model(images)
    _, preds = outputs.max(1)
    correct += (preds == labels).sum()
    samples += preds.size(0)
print(f"Test accuracy {float(correct) / float(samples) * 100:.2f} percentage
                || Correct {correct} out of {samples} samples")
```

```
Test accuracy 68.24 percentage || Correct 6824 out of 10000 samples
```

In [ ]:

```

```