Assignment 6

Problem statement

Object Detection using Transfer Learning of CNN Architecture

Details

1. Name : Akash Kulkarni
2. Branch : Information Technology
3. Division : BE 10
4. Batch : R-10
5. Roll Number : 43241
6. Course : Laboratory Practice 4 (Deep Learning)

```
import torch
import torchvision
import torch.nn as nn  # All neural network modules, nn.Linear, nn.Conv2d, BatchNo
import torch.optim as optim  # For all Optimization algorithms, SGD, Adam, etc.
import torch.nn.functional as F  # All functions that don't have any parameters
from torch.utils.data import (
    DataLoader,
)  # Gives easier dataset managment and creates mini batches
import torchvision.datasets as datasets  # Has standard datasets we can import in a
import torchvision.transforms as transforms  # Transformations we can perform on ou


# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Hyperparameters
num_classes = 10
learning_rate = 1e-3
batch_size = 1024
num_epochs = 50


# Simple Identity class that let's input pass without changes
class Identity(nn.Module):
    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        return x
```

1. Load in a pretrained model (VGG16)

```
model = torchvision.models.vgg16(pretrained=True)


print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

2. Freezing parameters in model's lower layers

```
# If you want to do finetuning then set requires_grad = False
for param in model.parameters():
```

```
    param.requires_grad = False

## Freezing the average pool layer of the model and add a custom classifier
model.avgpool = Identity()
```

3. Add custom classifier with several layers of trainable parameters to mode

```
model.classifier = nn.Sequential(
    nn.Linear(512, 100), nn.ReLU(),
    nn.Linear(100, num_classes)
)
model.to(device)
```

```
    VGG(
      (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace=True)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace=True)
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
    ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace=True)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace=True)
        (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
    ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace=True)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace=True)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace=True)
        (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
    ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace=True)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace=True)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace=True)
        (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
    ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace=True)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace=True)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace=True)
        (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
    ceil_mode=False)
      )
      (avgpool): Identity()
      (classifier): Sequential(
        (0): Linear(in_features=512, out_features=100, bias=True)
        (1): ReLU()
```

```
      (2): Linear(in_features=100, out_features=10, bias=True)
    )
  )
```

4. Train classifier layers on training data available for task

```
# Load Data
train_dataset = datasets.CIFAR10(
    root="dataset/", train=True, transform=transforms.ToTensor(), download=True
)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=Tr
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to datase
  0%|          | 0/170498071 [00:00<?, ?it/s]
Extracting dataset/cifar-10-python.tar.gz to dataset/
```

```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)


# Train Network
for epoch in range(num_epochs):
    losses = []

    for batch_idx, (data, targets) in enumerate(train_loader):
        # Get data to cuda if possible
        data = data.to(device=device)
        targets = targets.to(device=device)

        # forward
        scores = model(data)
        loss = criterion(scores, targets)

        losses.append(loss.item())
        # backward
        optimizer.zero_grad()
        loss.backward()

        # gradient descent or adam step
        optimizer.step()

    print(f"Cost at epoch {epoch} is {sum(losses)/len(losses):.5f}")
```

```
Cost at epoch 0 is 1.07481
Cost at epoch 1 is 1.06143
Cost at epoch 2 is 1.04898
Cost at epoch 3 is 1.03931
Cost at epoch 4 is 1.02822
Cost at epoch 5 is 1.02065
Cost at epoch 6 is 1.01188
Cost at epoch 7 is 1.00401
Cost at epoch 8 is 0.99593
```

```
Cost at epoch 9 is 0.98736
Cost at epoch 10 is 0.98071
Cost at epoch 11 is 0.97320
Cost at epoch 12 is 0.96478
Cost at epoch 13 is 0.95784
Cost at epoch 14 is 0.94961
Cost at epoch 15 is 0.94268
Cost at epoch 16 is 0.93464
Cost at epoch 17 is 0.92747
Cost at epoch 18 is 0.92085
Cost at epoch 19 is 0.91396
Cost at epoch 20 is 0.90671
Cost at epoch 21 is 0.89789
Cost at epoch 22 is 0.89123
Cost at epoch 23 is 0.88446
Cost at epoch 24 is 0.87809
Cost at epoch 25 is 0.87101
Cost at epoch 26 is 0.86366
Cost at epoch 27 is 0.85622
Cost at epoch 28 is 0.84890
Cost at epoch 29 is 0.84319
Cost at epoch 30 is 0.83885
Cost at epoch 31 is 0.83068
Cost at epoch 32 is 0.82383
Cost at epoch 33 is 0.81861
Cost at epoch 34 is 0.81195
Cost at epoch 35 is 0.80578
Cost at epoch 36 is 0.79985
Cost at epoch 37 is 0.79240
Cost at epoch 38 is 0.78767
Cost at epoch 39 is 0.78148
Cost at epoch 40 is 0.77561
Cost at epoch 41 is 0.76939
Cost at epoch 42 is 0.76440
Cost at epoch 43 is 0.75909
Cost at epoch 44 is 0.75379
Cost at epoch 45 is 0.74800
Cost at epoch 46 is 0.74199
Cost at epoch 47 is 0.73722
Cost at epoch 48 is 0.73185
Cost at epoch 49 is 0.72716
```

5. Checking accuracy and fine tuning if required.

```
def check_accuracy(loader, model):
    if loader.dataset.train:
        print("Checking accuracy on training data")
    else:
        print("Checking accuracy on test data")

    num_correct = 0
    num_samples = 0
    model.eval()

    with torch.no_grad():
        for x, y in loader:
```

```
            x = x.to(device=device)
            y = y.to(device=device)

            scores = model(x)
            _, predictions = scores.max(1)
            num_correct += (predictions == y).sum()
            num_samples += predictions.size(0)

        print(
            f"Got {num_correct} / {num_samples} with accuracy {float(num_correct)/
        )

    model.train()


check_accuracy(train_loader, model)
```

```
Checking accuracy on training data
Got 37926 / 50000 with accuracy 75.85
```