CIFAR10_2layer_RELU

September 7, 2019

[2]:
```
import numpy as np
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
```

[3]:
```
# set random seeds for reproducibility
torch.manual_seed(12)
torch.cuda.manual_seed(12)
np.random.seed(12)
```

[4]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# If we are on a CUDA machine, then this should print a CUDA device, but we are␣
 ↪not, so it will run on CPU:
print(f'Working on device={device}')
```

```
Working on device=cpu
```

[5]:
```
# Hyper-parameters

# each CIFAR image is RGB 32x32, so it is an 3D array [3,32,32]
# we will flatten the image as vector dim=3*32*32
input_size = 3*32*32

hidden_size = 1024

# we have 10 classes
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

num_classes = 10

num_epochs = 5
batch_size = 16

learning_rate = 0.001
```

[6]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                           shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                          shuffle=False)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[7]:
```
class MultilayerNeuralNet(nn.Module):
    def __init__(self, input_size, num_classes):
        '''
        Fully connected neural network with 2 hidden layers
        '''
        super(MultilayerNeuralNet, self).__init__()

        # hidden layers sizes, you can play with it as you wish!
        hidden1 = 1024
        hidden2 = 1024

        # input to first hidden layer parameters
        self.fc1 = nn.Linear(input_size, hidden1)
        self.relu1 = nn.ReLU()

        # second hidden layer
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.relu2 = nn.ReLU()

        # last output layer
        self.output = nn.Linear(hidden2, num_classes)

    def forward(self, x):
        '''
        This method takes an input x and layer after layer compute network␣
↪states.
        Last layer gives us predictions.
        '''
        state = self.fc1(x)
        state = self.relu1(state)
```

```
        state = self.fc2(state)
        state = self.relu2(state)

        state = self.output(state)

        return state
```

[8]:
```
model = MultilayerNeuralNet(input_size, num_classes).to(device)
```

[9]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[13]:
```
#Train model

# set our model in the training mode
model.train()
for epoch in range(num_epochs):

    epoch_loss = 0
    # data loop, iterate over chunk of data(batch) eg. 32 elements
    # compute model prediction
    # update weights
    for i, batch_sample in enumerate(train_loader):

        # print(batch_sample)
        images, labels = batch_sample

        # flatten the image and move to device
        images = images.reshape(-1, input_size).to(device)
        labels = labels.to(device)

        # Forward pass, compute prediction,
        # method 'forward' is automatically called
        prediction = model(images)
        # Compute loss, quantify how wrong our predictions are
        # small loss means a small error
        loss = criterion(prediction, labels)
        epoch_loss += loss.item()

        # Backward and optimize
        model.zero_grad()
        loss.backward()
        optimizer.step()

    epoch_loss = epoch_loss / len(train_loader)
```

```
    # Test the model

    # set our model in the training mode
    model.eval()
    # In test phase, we don't need to compute gradients (for memory efficiency)
    with torch.no_grad():
        correct = 0
        total = 0

        for images, labels in test_loader:
            # reshape image
            images = images.reshape(-1, input_size).to(device)
            labels = labels.to(device)

            # predict classes
            prediction = model(images)

            # compute accuracy
            _, predicted = torch.max(prediction.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        acc = correct/total

        # Accuracy of the network on the 10000 test images
        print(
            f'Epoch [{epoch+1}/{num_epochs}]], Loss: {epoch_loss:.4f} Test acc:␣
↪{acc}')
```

```
Epoch [1/5]], Loss: 1.2684 Test acc: 0.5035
Epoch [2/5]], Loss: 1.2199 Test acc: 0.5083
Epoch [3/5]], Loss: 1.1677 Test acc: 0.5016
Epoch [4/5]], Loss: 1.1308 Test acc: 0.5068
Epoch [5/5]], Loss: 1.0841 Test acc: 0.4997
```