Appendix-6: Deep neural network without convolution layer (DNN):

This method uses a deep neural network architecture without convolutional layers to classify the CIFAR-10 dataset. The network consisted of several fully connected layers, each followed by a ReLU activation function, and a final softmax layer for classification.

We used PyTorch to implement the model and trained it using the cross-entropy loss function and the stochastic gradient descent (SGD) optimizer.

In [1]:
```
# import library dependencies
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
```

Import Data

In [2]:
```
ROOT_PATH='../'
```

In [3]:
```
BATCH_SIZE=16
```

In [4]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

In [5]:
```
train_dataset = CIFAR10(root=ROOT_PATH, download=True, train=True, transform
eval_dataset = CIFAR10(root=ROOT_PATH, train=False, transform=transform)
```

```
Files already downloaded and verified
```

Preprocess Data

In [6]:
```
train_data_loader = DataLoader(dataset=train_dataset, num_workers=4, batch_s
eval_data_loader = DataLoader(dataset=eval_dataset, num_workers=4, batch_siz
```

Define model and train

In [ ]:

```python
# define the DNN architecture
class DNN(nn.Module):
    def __init__(self):
        super(DNN, self).__init__()
        self.fc1 = nn.Linear(32*32*3, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = x.view(-1, 32*32*3)
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.fc2(x)
        return x
```

In [8]:

```python
# create an instance of the DNN
net = DNN()
```

In [9]:

```python
# define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

In [10]:

```python
num_epochs = 50
```

In [11]:

```python
# train the DNN
for epoch in range(num_epochs):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(train_data_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # 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('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 1.755
[2,  2000] loss: 1.495
[3,  2000] loss: 1.382
[4,  2000] loss: 1.306
[5,  2000] loss: 1.244
[6,  2000] loss: 1.195
[7,  2000] loss: 1.137
[8,  2000] loss: 1.082
[9,  2000] loss: 1.036
[10,  2000] loss: 0.992
[11,  2000] loss: 0.948
[12,  2000] loss: 0.908
[13,  2000] loss: 0.867
[14,  2000] loss: 0.823
[15,  2000] loss: 0.790
[16,  2000] loss: 0.752
[17,  2000] loss: 0.712
[18,  2000] loss: 0.685
[19,  2000] loss: 0.653
[20,  2000] loss: 0.621
[21,  2000] loss: 0.589
[22,  2000] loss: 0.551
[23,  2000] loss: 0.524
[24,  2000] loss: 0.499
[25,  2000] loss: 0.482
[26,  2000] loss: 0.447
[27,  2000] loss: 0.427
[28,  2000] loss: 0.397
[29,  2000] loss: 0.387
[30,  2000] loss: 0.361
[31,  2000] loss: 0.340
[32,  2000] loss: 0.320
[33,  2000] loss: 0.293
[34,  2000] loss: 0.281
[35,  2000] loss: 0.261
[36,  2000] loss: 0.249
[37,  2000] loss: 0.234
[38,  2000] loss: 0.218
[39,  2000] loss: 0.209
[40,  2000] loss: 0.189
[41,  2000] loss: 0.180
[42,  2000] loss: 0.174
[43,  2000] loss: 0.154
[44,  2000] loss: 0.142
[45,  2000] loss: 0.120
[46,  2000] loss: 0.120
[47,  2000] loss: 0.113
[48,  2000] loss: 0.120
[49,  2000] loss: 0.113
[50,  2000] loss: 0.092
Finished Training
```

Save and load model

In [12]:

```
# save the trained model
torch.save(net, 'dnn.pth')
```

In [13]:

```
# load the saved model
net = torch.load('dnn.pth')
```

Evaluate the model

In [14]:

```
# evaluate the DNN
net.eval()
correct = 0
total = 0
with torch.no_grad():
    for data in eval_data_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 53 %
```

In [15]:

```
class_names = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse'
```

In [16]:

```
# calculate class-wise accuracy
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in eval_data_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(BATCH_SIZE):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        class_names[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane : 55 %
Accuracy of   car : 61 %
Accuracy of  bird : 45 %
Accuracy of   cat : 42 %
Accuracy of  deer : 42 %
Accuracy of   dog : 38 %
Accuracy of  frog : 54 %
Accuracy of horse : 60 %
Accuracy of  ship : 68 %
Accuracy of truck : 61 %
```

In [17]:

```
TP = 0
FP = 0
TN = 0
FN = 0

with torch.no_grad():
    for data in eval_data_loader:
        images, labels = data
        images = images
        labels = labels
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        for i in range(len(labels)):
            if predicted[i] == labels[i]:
                if predicted[i] == 1:
                    TP += 1
                else:
                    TN += 1
            else:
                if predicted[i] == 1:
                    FP += 1
                else:
                    FN += 1

accuracy = 100 * (TP + TN) / (TP + TN + FP + FN)
precision = 100 * TP / (TP + FP)
recall = 100 * TP / (TP + FN)
f1_score = 2 * precision * recall / (precision + recall)

print('Accuracy: %.2f %%' % (accuracy))
print('Precision: %.2f %%' % (precision))
print('Recall: %.2f %%' % (recall))
print('F1 Score: %.2f %%' % (f1_score))
```

```
Accuracy: 53.07 %
Precision: 63.46 %
Recall: 12.38 %
F1 Score: 20.71 %
```