Problem 2 (Neural networks with PyTorch)

In this problem, the goal is to utilize the PyTorch library to train a simple fully connected neural network to classify RGB color images from the CIFAR10 dataset. The dataset consist of 10 classes plane, car, bird, cat, deer, dog, frog, horse, ship, truck, where there are 5000 training and 1000 test images per class. Each image has dimensions $3 \times 32 \times 32$. The dataset also includes labels for each image.

In [16]:
```
import numpy as np
import os
import torch
from torch import nn
from torch.utils.data import DataLoader, Subset
import torchvision
import torchvision.transforms as transforms

device = (
    "cuda" if torch.cuda.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using cuda device
```

Load CIFAR10 train and test set

In [17]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
)

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
```

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

1

In this problem we only consider images of the classes cat, dog and ship and train/test on 3000/1000 images per class. Start by reducing the size of the training and test set accordingly. Then, initialize the train and test dataloaders. We use a batch size of $B = 4$.

In [18]:
```
class_indices = [3, 5, 8]

def build_subset(dataset, target_classes, samples):
    targets = np.array(dataset.targets)
    indices = []
    for i in target_classes:
        indices.extend(np.where(targets == i)[0][:samples])
    return Subset(dataset, indices)

trainset = build_subset(trainset, class_indices, 3000)
testset = build_subset(testset, class_indices, 1000)

class_mapping = {original_class: new_index for new_index, original_class in enumerate(class_indices)}
trainset.dataset.targets = [class_mapping[label] if label in class_indices else label for label in trains
testset.dataset.targets = [class_mapping[label] if label in class_indices else label for label in testset

trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True)
testloader = DataLoader(testset, batch_size=batch_size, shuffle=False)
```

2

Inheriting from nn.Module, implement a simple fully connected neural network with a single hidden layer of dimension 512, ReLU activations (not for the output layer) and an output dimension equal to the number of classes. Note that for the first linear layer to process a batch of input images the batch needs to be flattened across the color channels and spatial dimensions to a size of $B \times 3072$.

In [19]:
```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(3 * 32 * 32, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 3),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits


model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=3072, out_features=512, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=512, out_features=3, bias=True)
  )
)
```

3

Define a SGD optimizer with learning rate `0.001` and momentum `0.9` from torch.optim. With momentum the optimizer adds a weighted running average of the gradients per parameter to the current gradient in each step improving both stability and speed of convergence. Use the torch.nn.CrossEntropyLoss() loss and train the network for 10 epochs. Report the train and test accuracy after every epoch and save the best model.

In [20]:
```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
loss_fn = nn.CrossEntropyLoss()

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    train_loss, correct = 0, 0

    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        train_loss += loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    avg_trainloss = train_loss / len(dataloader)
    train_accuracy = correct / size
    print(f"Train Error: \n Accuracy: {(100*train_accuracy):>0.2f}%, Avg loss: {avg_trainloss:>5f} \n")
```

```
    return avg_trainloss, train_accuracy


def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    model.eval()
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    avg_testloss = test_loss / len(dataloader)
    test_accuracy = correct / size
    print(f"Test Error: \n Accuracy: {(100*test_accuracy):>0.2f}%, Avg loss: {avg_testloss:>5f} \n")

    return avg_testloss, test_accuracy


epochs = 10
best_accuracy = 0
best_model = None

for t in range(epochs):
    print(f"Epoch {t+1}")
    train_loss, train_accuracy = train(trainloader, model, loss_fn, optimizer)
    test_loss, test_accuracy = test(testloader, model, loss_fn)

    if test_accuracy > best_accuracy:
        best_accuracy = test_accuracy
        best_model = model.state_dict()

print("Done training")

if best_model:
    torch.save(best_model, "best_model.pth")
    print(f"Best model saved with accuracy: {best_accuracy*100:.2f}%")
```

```
Epoch 1
Train Error:
 Accuracy: 63.89%, Avg loss: 0.724714

Test Error:
 Accuracy: 66.77%, Avg loss: 0.688844

Epoch 2
Train Error:
 Accuracy: 68.60%, Avg loss: 0.650457

Test Error:
 Accuracy: 67.07%, Avg loss: 0.695679

Epoch 3
Train Error:
 Accuracy: 70.38%, Avg loss: 0.607824

Test Error:
 Accuracy: 65.47%, Avg loss: 0.713574

Epoch 4
Train Error:
 Accuracy: 73.38%, Avg loss: 0.568773

Test Error:
 Accuracy: 67.23%, Avg loss: 0.682848

Epoch 5
Train Error:
 Accuracy: 74.09%, Avg loss: 0.542213

Test Error:
 Accuracy: 68.40%, Avg loss: 0.704923

Epoch 6
Train Error:
 Accuracy: 76.64%, Avg loss: 0.506158

Test Error:
 Accuracy: 66.67%, Avg loss: 0.726577

Epoch 7
Train Error:
 Accuracy: 78.61%, Avg loss: 0.483516

Test Error:
 Accuracy: 68.40%, Avg loss: 0.759970

Epoch 8
Train Error:
 Accuracy: 80.32%, Avg loss: 0.448310

Test Error:
 Accuracy: 69.47%, Avg loss: 0.725484

Epoch 9
Train Error:
 Accuracy: 81.94%, Avg loss: 0.421566

Test Error:
 Accuracy: 68.30%, Avg loss: 0.784895

Epoch 10
Train Error:
 Accuracy: 83.04%, Avg loss: 0.395804

Test Error:
 Accuracy: 68.07%, Avg loss: 0.821584

Done training
Best model saved with accuracy: 69.47%
```

4

Load the best model and report the overall test accuracy and the test accuracy per class. On which class does our classifier perform best and why?

In [21]:
```
bestmodel = NeuralNetwork().to(device)
bestmodel.load_state_dict(torch.load("best_model.pth", weights_only=True))

bestmodel.eval()

total_correct, total_samples = 0, 0

class_correct = [0 for _ in range(len(class_indices))]
class_total = [0 for _ in range(len(class_indices))]

with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = bestmodel(inputs)
        _, predicted = torch.max(outputs, 1)

        total_samples += labels.size(0)
        total_correct += (predicted == labels).sum().item()

        for i in range(len(labels)):
            label = labels[i]
            class_correct[label] += (predicted[i] == label).item()
            class_total[label] += 1

overall_accuracy = total_correct / total_samples
class_accuracies = [class_correct[i] / class_total[i] for i in range(len(class_indices))]

print(f"Overall accuracy: {overall_accuracy*100:.2f}%")

classes = ["cat", "dog", "ship"]

for i, acc in enumerate(class_accuracies):
    print(f"Accuracy for class {classes[i]}: {acc*100:.2f}%")
```

```
Overall accuracy: 68.07%
Accuracy for class cat: 54.90%
Accuracy for class dog: 62.40%
Accuracy for class ship: 86.90%
```

The classifier performs best on the "ship" class with an accuracy of 86.90%, compared to around 60% for "cat" and "dog." This suggests that the model finds it easier to distinguish ships, likely because cats and dogs share similar animal features, making them harder to differentiate.