# Homework 11

## Problem 1

For a single step of convolution on an arbitary data point in an arbitrary channel, we have $k \times k$ many multiplications and $k \cdot (k-1)$ many additions, which means within a single channel, there are in total $(k^2 + k(k-1)) \cdot w \cdot h$ many calculations, and we have $c$ channels, which gives us $(k^2 + k(k-1)) \cdot w \cdot h \cdot c$ many calculations. In $\mathcal{O}$ notation, the computational cost should be $\mathcal{O}(k^2whc)$.

## Problem 2

### 1

- Load data set

In [ ]:

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset

import torch.nn as nn
import torch.nn.functional as F

import torch.optim as optim

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

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

- Filter the data set

In [ ]:

```
class_names = ['cat', 'dog', 'ship']
class_indices = {'cat': 0, 'dog': 1, 'ship': 2}

def filter_classes(dataset, classes):
    class_to_idx = {dataset.classes[i]: i for i in range(len(dataset.clas
    filtered_indices = []
    labels = []
    for i, (_, label) in enumerate(dataset):
        if dataset.classes[label] in classes:
            filtered_indices.append(i)
            labels.append(class_indices[dataset.classes[label]])
    return torch.utils.data.Subset(dataset, filtered_indices), torch.tens

trainset_filtered, train_labels = filter_classes(trainset, class_names)
testset_filtered, test_labels = filter_classes(testset, class_names)
```

- Remap the labels

In [ ]:

```
class remappedDataset(Dataset):
    def __init__(self, subset, labels):
        self.subset = subset
        self.labels = labels

    def __getitem__(self, idx):
        image, _ = self.subset[idx]
        return image, self.labels[idx]

    def __len__(self):
        return len(self.subset)

train_dataset = remappedDataset(trainset_filtered, train_labels)
test_dataset = remappedDataset(testset_filtered, test_labels)

trainloader = DataLoader(train_dataset, batch_size=4, shuffle=True, num_w
testloader = DataLoader(test_dataset, batch_size=4, shuffle=False, num_wo
```

- Implement the CNN

In [ ]:

```
class CNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = nn.Conv2d(3, 6, 5)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.pooling = nn.MaxPool2d(2, 2)
    self.linear1 = nn.Linear(5*5*16, 120)
    self.linear2 = nn.Linear(120, 84)
    self.linear3 = nn.Linear(84, 3)

  def forward(self, x):
    x = self.pooling(F.relu(self.conv1(x)))
    x = self.pooling(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.linear1(x))
    x = F.relu(self.linear2(x))
    x = self.linear3(x)
    return x

cnn = CNN()
```

- Define optimizer and loss function

In [ ]:

```
loss_function = nn.CrossEntropyLoss()
optimizer = optim.SGD(cnn.parameters(), lr=0.001, momentum=0.9)
```

- Train and Evaluate

In [ ]:

```
def train_and_evaluate(cnn, trainloader, testloader, optimizer,
                       loss_function, epochs=10):
  best_accuracy = 0
  best_model_state = None

  for epoch in range(epochs):
    cnn.train()
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
      inputs, labels = data
      # print("label: %s", labels)
      optimizer.zero_grad()

      outputs = cnn(inputs)
      loss = loss_function(outputs, labels)
      loss.backward()
      optimizer.step()

      running_loss += loss.item()

    # Evaluate on test data
    cnn.eval()
    correct = 0
    total = 0
    with torch.no_grad():
      for data in testloader:
        images, labels = data
        outputs = cnn(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    test_accuracy = 100 * correct / total
    print(f'Epoch {epoch + 1}, Loss: {running_loss / len(trainloader)},
          Test Accuracy: {test_accuracy}%')

    # Save the best model
    if test_accuracy > best_accuracy:
      best_accuracy = test_accuracy
      best_model_state = cnn.state_dict()

  print('Training Finished')
  return best_model_state

# Train the cnnwork and save the best model
best_model_state = train_and_evaluate(cnn, trainloader, testloader,
                                      optimizer, loss_function)

# Save the best model
torch.save(best_model_state, './best_model.pth')
```

```
Epoch 1, Loss: 0.5714861377806092, Test Accuracy: 72.36666666666666%
Epoch 2, Loss: 0.5367183759396275, Test Accuracy: 74.6%
Epoch 3, Loss: 0.5090775508440255, Test Accuracy: 73.76666666666667%
Epoch 4, Loss: 0.48291732497972745, Test Accuracy: 75.2%
Epoch 5, Loss: 0.4585593342268529, Test Accuracy: 77.4%
Epoch 6, Loss: 0.4344570583905714, Test Accuracy: 77.46666666666667%
Epoch 7, Loss: 0.40506579721325736, Test Accuracy: 75.4%
Epoch 8, Loss: 0.3847428083923165, Test Accuracy: 76.1%
Epoch 9, Loss: 0.35653543670056725, Test Accuracy: 75.43333333333334%
Epoch 10, Loss: 0.32142896284716793, Test Accuracy: 75.13333333333334%
Training Finished
```

- We can observe that the convolutional neural network performs better than the simple fully connected neural network.

## 2

- Shuffle the dataset

In [ ]:
```python
class shuffledDataset(Dataset):
    def __init__(self, original_dataset):
        self.dataset = original_dataset
        self.permutation = torch.randperm(3 * 32 * 32)

    def __getitem__(self, index):
        image, label = self.dataset[index]
        # Convert image to 1D tensor and apply the same permutation to al
        image_1d = image.view(-1)
        shuffled_image_1d = image_1d[self.permutation]
        # Convert back to original shape
        shuffled_image = shuffled_image_1d.view(3, 32, 32)
        return shuffled_image, label

    def __len__(self):
        return len(self.dataset)


# Create shuffled datasets
trainset_shuffled = shuffledDataset(trainset_filtered)
testset_shuffled = shuffledDataset(testset_filtered)

trainset_shuffled_remapped = remappedDataset(trainset_shuffled, train_lab
testset_shuffled_remapped = remappedDataset(testset_shuffled, test_labels



# DataLoader
trainloader_shuffled = DataLoader(trainset_shuffled_remapped, batch_size=
                                  shuffle=True, num_workers=2)
testloader_shuffled = DataLoader(testset_shuffled_remapped, batch_size=4,
                                 shuffle=False, num_workers=2)
```

- Train with shuffled data

In [ ]:
```python
best_model_state = train_and_evaluate(cnn, trainloader_shuffled,
                                      testloader_shuffled, optimizer,
                                      loss_function)
```

```
Epoch 1, Loss: 0.7743579309908052, Test Accuracy: 31.966666666666665%
Epoch 2, Loss: 0.6789236864373088, Test Accuracy: 32.666666666666664%
Epoch 3, Loss: 0.6392491559597353, Test Accuracy: 32.5%
Epoch 4, Loss: 0.6065750461059312, Test Accuracy: 32.8%
Epoch 5, Loss: 0.5770396108590998, Test Accuracy: 31.833333333333332%
Epoch 6, Loss: 0.5474726777335008, Test Accuracy: 32.7%
Epoch 7, Loss: 0.52086586153411, Test Accuracy: 33.36666666666667%
Epoch 8, Loss: 0.49159315741838266, Test Accuracy: 32.43333333333333%
Epoch 9, Loss: 0.4616542441578349, Test Accuracy: 33.266666666666666%
Epoch 10, Loss: 0.4337282149097669, Test Accuracy: 33.43333333333333%
Training Finished
```

- Evaluation

We found that the performance is lot worse, because by shuffling the pixels, we distorted the shape of original object

## 3

- Implement fully connected layer

In [ ]:

```
class FCNN(nn.Module):
    def __init__(self):
        super(FCNN, self).__init__()
        # Hidden layer with d=3*32*32 and 512 neurons
        self.layer1 = nn.Linear(3 * 32 * 32, 512)
        # Output layer maps number of neurons to the output dimension
        self.layer2 = nn.Linear(512, len(class_names))

    def forward(self, x):
        x = x.view(-1, 3 * 32 * 32)
        x = F.relu(self.layer1(x))
        x = self.layer2(x)
        return x

fcnn = FCNN()
```

- Train on shuffled set

In [ ]:

```
best_model_state = train_and_evaluate(fcnn, trainloader_shuffled,
                                      testloader_shuffled, optimizer,
                                      loss_function)
```

```
Epoch 1, Loss: 1.1030349195480347, Test Accuracy: 33.7%
Epoch 2, Loss: 1.1030349198500315, Test Accuracy: 33.7%
Epoch 3, Loss: 1.1030349203745524, Test Accuracy: 33.7%
Epoch 4, Loss: 1.1030349208990733, Test Accuracy: 33.7%
Epoch 5, Loss: 1.103034920056661, Test Accuracy: 33.7%
Epoch 6, Loss: 1.1030349197069804, Test Accuracy: 33.7%
Epoch 7, Loss: 1.103034920835495, Test Accuracy: 33.7%
Epoch 8, Loss: 1.1030349195162454, Test Accuracy: 33.7%
Epoch 9, Loss: 1.1030349198818208, Test Accuracy: 33.7%
Epoch 10, Loss: 1.1030349203109742, Test Accuracy: 33.7%
Training Finished
```

- Evaluation

From test accuracies we noticed that the classifier is accturally randomly labeling the input, which means it cannot work properly, the reason is the same as mentioned in subtask 2