Homework 10

In [ ]:
```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

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

import torch.optim as optim
```

Load CIFAR10 train and test set

In [ ]:
```
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)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100.0%
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

Problem 1

To calculate the computational complexity of a single step of stochastic gradient we can look at 3 major steps

1. Forward Passes

Firstly, the computation from input layer to the first hiddden layer involves k many dot multiplication between vectors in $\mathbb{R}^d$, which gives us $\mathcal{O}(kd)$.

Then, in the calculation between 2 hidden layers, there are k^2 many scaler products, and there are q many hidden layers, which gives us $\mathcal{O}(qk^2)$.

Lastly, k neurons of the last layer will be mapped to a single scaler output, it costs $\mathcal{O}(k)$.

We can obtain a total cost by summing them together, which leads to $\mathcal{O}((d+1)k + qk^2)$.

2. Backpropagation

Firstly, the computation from output layer to an arbitary neuron in the last hidden layer contains 2 gradient, namely:

$$\frac{\partial y}{\partial x_{qi}} \quad \text{and} \quad \frac{\partial l}{\partial y},$$

it has a constant cost $\mathcal{O}(2)$, and there are k many neurons, so the cost is $\mathcal{O}(k)$.

Subsequently, between 2 hidden layers, we have k^2 many calculations interpreted above, the cost goes $\mathcal{O}(k^2)$, and multiplied with q layers, we have the cost in all hidden layers: $\mathcal{O}(qk^2)$.

By summing them together we have $\mathcal{O}(k + qk^2)$.

3. Parameter Updating

After the calculation, we need to update the parameters in the fcnnwork, in the input layer we have dk many parameters, in q layers there are $q \cdot k^2$ parameters, and in the input layer we have k parameters. To update them, the cost is $\mathcal{O}((d+1)k + qk^2)$.

Total Cost

The total cost can be calculated by adding all the 3 steps together, thus we have $\mathcal{O}(3qk^2 + 2(d+1)k + k)$, by ignoring constant coeffecients and lower order terms, it results in $\mathcal{O}(qk^2)$

Problem 2

1

Here, let's filter dataset by its name and initialize it in batch with size 4.

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.classes))}
    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.tensor(labels)

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

class CustomDataset(torch.utils.data.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 = CustomDataset(trainset_filtered, train_labels)
test_dataset = CustomDataset(testset_filtered, test_labels)
```

```
trainloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2)
testloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=2)
```

In [ ]:
```
# Modified to include only cat, dog, and ship classes
class_names = ['cat', 'dog', 'ship']
class_indices = {name: i for i, name in enumerate(class_names)}

# Function to filter out only the specified classes
def filter_classes(dataset, classes):
    class_to_idx = {dataset.classes[i]: i for i in range(len(dataset.classes))}
    indices = [i for i, (_, label) in enumerate(dataset) if dataset.classes[label] in classes]
    return torch.utils.data.Subset(dataset, indices)

# Apply the filter to the train and test datasets
trainset_filtered = filter_classes(trainset, class_names)
testset_filtered = filter_classes(testset, class_names)

# Initialize the dataloaders
trainloader = DataLoader(trainset_filtered, batch_size=batch_size, shuffle=True, num_workers=2)
testloader = DataLoader(testset_filtered, batch_size=batch_size, shuffle=False, num_workers=2)
```

2

The FCNN class is implemented as our fully connected neural fcnnwork

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()
```

3

Here, we define an SGD optimizer with CrossEntropyLoss, and perform training on our setup

In [ ]:
```
# Initialize optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(fcnn.parameters(), lr=0.001, momentum=0.9)
```

In [ ]:
```
def train_and_evaluate(fcnn, trainloader, testloader, optimizer, criterion, epochs=10):
  best_accuracy = 0
  best_model_state = None

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

      outputs = fcnn(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()

      running_loss += loss.item()

    # Evaluate on test data
    fcnn.eval()
    correct = 0
    total = 0
    with torch.no_grad():
      for data in testloader:
        images, labels = data
        outputs = fcnn(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 = fcnn.state_dict()

  print('Training Finished')
  return best_model_state

# Train the fcnnwork and save the best model
best_model_state = train_and_evaluate(fcnn, trainloader, testloader, optimizer, criterion)

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

```
Epoch 1, Loss: 0.257869926950691, Test Accuracy: 68.06666666666666%
Epoch 2, Loss: 0.24518481644349277, Test Accuracy: 68.1%
Epoch 3, Loss: 0.23455415490372966, Test Accuracy: 68.13333333333334%
Epoch 4, Loss: 0.23548380395160537, Test Accuracy: 67.6%
Epoch 5, Loss: 0.220462328650033７, Test Accuracy: 65.86666666666666%
Epoch 6, Loss: 0.21201846012310066, Test Accuracy: 68.6%
Epoch 7, Loss: 0.16910335472654575, Test Accuracy: 69.66666666666667%
Epoch 8, Loss: 0.22301156308879563, Test Accuracy: 67.7%
Epoch 9, Loss: 0.1792418297730261, Test Accuracy: 69.1%
Epoch 10, Loss: 0.1637681784330639, Test Accuracy: 69.8%
Training Finished
```

4

We shall load the best model and report the test accuracy for overall and per class

In [ ]:
```
def evaluate_model(fcnn, testloader, class_names):
  fcnn.eval()
  class_correct = list(0. for i in range(len(class_names)))
  class_total = list(0. for i in range(len(class_names)))
```

```
  with torch.no_grad():
    for data in testloader:
      images, labels = data
      outputs = fcnn(images)
      _, predicted = torch.max(outputs, 1)
      c = (predicted == labels).squeeze()
      for i in range(4):
        label = labels[i]
        class_correct[label] += c[i].item()
        class_total[label] += 1

  for i in range(len(class_names)):
    print(f'Accuracy of {class_names[i]} : {100 * class_correct[i] / class_total[i]}%')

# Load the best model
fcnn.load_state_dict(torch.load('./best_model.pth'))

# Evaluate the best model
evaluate_model(fcnn, testloader, class_names)
```

```
Accuracy of cat : 68.7%
Accuracy of dog : 52.7%
Accuracy of ship : 88.0%
```

We can justify that the model performs better when classifing a ship, the reason for it is highly likely that the different between ship and 2 animals we have is significantly larger, so it is easier to distinguish ships from the other. However, because of the similarity of cats and dogs, the model can make many error when labeling them.