EE655 Assignment 01

Ques1.

```
import numpy as np

def euclidean_distance(p1, p2):
   return np.linalg.norm(np.array(p1) - np.array(p2))

def detect_smile(mouth_points):
   # Assuming mouth_points = [(x1, y1), (x2, y2), (x3, y3), (x4, y4)]
   left_corner, top_middle, bottom_middle, right_corner = mouth_points

   # Feature 1: Mouth Aspect Ratio (MAR)
   vertical_dist = euclidean_distance(top_middle, bottom_middle)
   horizontal_dist = euclidean_distance(left_corner, right_corner)
   mar = vertical_dist / horizontal_dist

   # Feature 2: Lip Curvature
   curvature = ((top_middle[1] + bottom_middle[1]) / 2) - ((left_corner[1] +
right_corner[1]) / 2)

   # Feature 3: Corner Distance Change
   mouth_width = horizontal_dist  # Track changes over time for smile detection

   return mar, curvature, mouth_width

# Example mouth key points: (x1, y1), (x2, y2), (x3, y3), (x4, y4)
mouth_keypoints = [(30, 60), (40, 50), (40, 70), (50, 60)]
features = detect_smile(mouth_keypoints)
print("Mouth Aspect Ratio (MAR):", features[0])
print("Lip Curvature:", features[1])
print("Mouth Width:", features[2])
```

A Brief Explanation of the above Code

`euclidean_distance(p1, p2)`

- Computes the Euclidean distance between two points using the formula: d=(x2−x1)2+(y2−y1)2d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}d=(x2−x1)2+(y2−y1)2
- This function is used to measure distances between key facial points.

`detect_smile(mouth_points)`

- Takes four key points representing the mouth: left corner, top middle, bottom middle, and right corner.

- Extracts three features to detect a smile:
    - **Mouth Aspect Ratio (MAR):** Measures how open the mouth is by dividing vertical distance by horizontal distance.
    - **Lip Curvature:** Checks how much the middle of the lips is above or below the corners, indicating a smile or neutral expression.
    - **Mouth Width:** Simply records the horizontal distance between the mouth corners, useful for tracking changes over time.

Ques4.

```
import cv2
import numpy as np

def count_objects(binary_image):
   rows, cols = binary_image.shape
   visited = np.zeros((rows, cols), dtype=bool)
   object_count = 0

   # 8-directional movement
   directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

   def dfs(x, y):
       stack = [(x, y)]
       while stack:
           cx, cy = stack.pop()
           if visited[cx, cy]:
               continue
           visited[cx, cy] = True
           for dx, dy in directions:
               nx, ny = cx + dx, cy + dy
               if 0 <= nx < rows and 0 <= ny < cols and binary_image[nx, ny] == 1 and not visited[nx, ny]:
                   stack.append((nx, ny))

   # Scan the image
   for i in range(rows):
       for j in range(cols):
           if binary_image[i, j] == 1 and not visited[i, j]:
               object_count += 1
               dfs(i, j)

   return object_count

# Example: Read a binary image
```

```
binary_image = cv2.imread('objects.png', cv2.IMREAD_GRAYSCALE)
_, binary_image = cv2.threshold(binary_image, 128, 1, cv2.THRESH_BINARY)

print("Number of objects:", count_objects(binary_image))
```

A Brief Explanation of the above code:

count_objects(binary_image): Counts distinct objects in a binary image using depth-first search (DFS).

- Creates a **visited** matrix to track checked pixels.
- Defines 8-directional movement for connected components.
- Uses DFS to explore and mark all connected pixels of an object.
- Iterates through the image, calling DFS when a new object is found and increments the count.
- Returns the total number of detected objects.

dfs(x, y) (Nested inside **count_objects**): Performs depth-first search to mark all pixels of a connected object.

- Uses a stack to explore neighboring pixels iteratively.
- Marks visited pixels to prevent recounting.

Binary Image Processing (Before Function Call):

- Reads a grayscale image using OpenCV.
- Converts it into a binary image using thresholding.

Number of objects: 9

Ques2.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

class Swish(nn.Module):
    def forward(self, x):
        return x * torch.sigmoid(x)

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, padding=1)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.conv3 = nn.Conv2d(16, 120, 3)
        self.pool = nn.MaxPool2d(2, 2)
```

EE655 Assignment 01

Ques1.

```
import numpy as np

def euclidean_distance(p1, p2):
   return np.linalg.norm(np.array(p1) - np.array(p2))

def detect_smile(mouth_points):
   # Assuming mouth_points = [(x1, y1), (x2, y2), (x3, y3), (x4, y4)]
   left_corner, top_middle, bottom_middle, right_corner = mouth_points

   # Feature 1: Mouth Aspect Ratio (MAR)
   vertical_dist = euclidean_distance(top_middle, bottom_middle)
   horizontal_dist = euclidean_distance(left_corner, right_corner)
   mar = vertical_dist / horizontal_dist

   # Feature 2: Lip Curvature
   curvature = ((top_middle[1] + bottom_middle[1]) / 2) - ((left_corner[1] +
right_corner[1]) / 2)

   # Feature 3: Corner Distance Change
   mouth_width = horizontal_dist  # Track changes over time for smile detection

   return mar, curvature, mouth_width

# Example mouth key points: (x1, y1), (x2, y2), (x3, y3), (x4, y4)
mouth_keypoints = [(30, 60), (40, 50), (40, 70), (50, 60)]
features = detect_smile(mouth_keypoints)
print("Mouth Aspect Ratio (MAR):", features[0])
print("Lip Curvature:", features[1])
print("Mouth Width:", features[2])
```

A Brief Explanation of the above Code

`euclidean_distance(p1, p2)`

- Computes the Euclidean distance between two points using the formula: d=(x2−x1)2+(y2−y1)2d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}d=(x2−x1)2+(y2−y1)2
- This function is used to measure distances between key facial points.

`detect_smile(mouth_points)`

- Takes four key points representing the mouth: left corner, top middle, bottom middle, and right corner.

- Extracts three features to detect a smile:
    - **Mouth Aspect Ratio (MAR):** Measures how open the mouth is by dividing vertical distance by horizontal distance.
    - **Lip Curvature:** Checks how much the middle of the lips is above or below the corners, indicating a smile or neutral expression.
    - **Mouth Width:** Simply records the horizontal distance between the mouth corners, useful for tracking changes over time.

Ques4.

```
import cv2
import numpy as np

def count_objects(binary_image):
   rows, cols = binary_image.shape
   visited = np.zeros((rows, cols), dtype=bool)
   object_count = 0

   # 8-directional movement
   directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1,
1)]

   def dfs(x, y):
       stack = [(x, y)]
       while stack:
           cx, cy = stack.pop()
           if visited[cx, cy]:
               continue
           visited[cx, cy] = True
           for dx, dy in directions:
               nx, ny = cx + dx, cy + dy
               if 0 <= nx < rows and 0 <= ny < cols and binary_image[nx, ny] == 1
and not visited[nx, ny]:
                   stack.append((nx, ny))

   # Scan the image
   for i in range(rows):
       for j in range(cols):
           if binary_image[i, j] == 1 and not visited[i, j]:
               object_count += 1
               dfs(i, j)

   return object_count

# Example: Read a binary image
```

```
binary_image = cv2.imread('objects.png', cv2.IMREAD_GRAYSCALE)
_, binary_image = cv2.threshold(binary_image, 128, 1, cv2.THRESH_BINARY)

print("Number of objects:", count_objects(binary_image))
```

A Brief Explanation of the above code:

count_objects(binary_image): Counts distinct objects in a binary image using depth-first search (DFS).

- Creates a **visited** matrix to track checked pixels.
- Defines 8-directional movement for connected components.
- Uses DFS to explore and mark all connected pixels of an object.
- Iterates through the image, calling DFS when a new object is found and increments the count.
- Returns the total number of detected objects.

dfs(x, y) (Nested inside **count_objects**): Performs depth-first search to mark all pixels of a connected object.

- Uses a stack to explore neighboring pixels iteratively.
- Marks visited pixels to prevent recounting.

Binary Image Processing (Before Function Call):

- Reads a grayscale image using OpenCV.
- Converts it into a binary image using thresholding.

Number of objects: 9

Ques2.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

class Swish(nn.Module):
    def forward(self, x):
        return x * torch.sigmoid(x)

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, padding=1)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.conv3 = nn.Conv2d(16, 120, 3)
        self.pool = nn.MaxPool2d(2, 2)
```

```
        self.fc1 = nn.Linear(120 * 4 * 4, 84)
        self.fc2 = nn.Linear(84, 10)
        self.act = Swish()

    def forward(self, x):
        x = self.pool(self.act(self.conv1(x)))
        x = self.pool(self.act(self.conv2(x)))
        x = self.act(self.conv3(x))
        x = torch.flatten(x, 1)
        x = self.act(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)

transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])
train_loader = DataLoader(datasets.MNIST('./data', train=True,
download=True, transform=transform), batch_size=64, shuffle=True)
test_loader = DataLoader(datasets.MNIST('./data', train=False,
transform=transform), batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LeNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    model.train()
    for i, (x, y) in enumerate(train_loader):
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        loss = criterion(model(x), y)
        loss.backward()
        optimizer.step()
        if i % 100 == 0:
            print(f"Epoch {epoch+1}/5, Batch {i}, Loss:
{loss.item():.4f}")

model.eval()
corr, tot = 0, 0
with torch.no_grad():
    for x, y in test_loader:
        x, y = x.to(device), y.to(device)
        _, pred = torch.max(model(x), 1)
```

```
        tot += y.size(0)
        corr += (pred == y).sum().item()

print(f"Test Accuracy: {100 * corr / tot:.2f}%")
```

A Brief Explanation of the above code:

`Swish(nn.Module)`: Implements the Swish activation function, which is defined as x⋅sigmoid(x)x \cdot \text{sigmoid}(x)x⋅sigmoid(x), providing smooth, non-monotonic activation.
`LeNet(nn.Module)`: Defines a convolutional neural network (CNN) based on the LeNet-5 architecture.

- Contains three convolutional layers (`conv1`, `conv2`, `conv3`) with activation and pooling.
- Uses `MaxPool2d` to reduce spatial dimensions.
- Fully connected layers (`fc1`, `fc2`) for classification.
- Uses the Swish activation function and `softmax` for output probabilities.

Data Preprocessing & Loading:

- Applies transformations (tensor conversion and normalization) using `transforms.Compose`.
- Loads MNIST dataset using `DataLoader` for training and testing.

Model Training:

- Uses Cross-Entropy Loss (`nn.CrossEntropyLoss()`) as the criterion.
- Optimizes using Adam (`optim.Adam`).
- Runs for 5 epochs, updating weights using backpropagation.
- Prints loss every 100 batches.

Model Evaluation:

- Disables gradient computation (`torch.no_grad()`).
- Performs inference on the test set, calculating the accuracy.
- Prints the final test accuracy.

Epoch 1/5, Batch 0, Loss: 2.3023
Epoch 1/5, Batch 100, Loss: 1.5620
Epoch 1/5, Batch 200, Loss: 1.5059
Epoch 1/5, Batch 300, Loss: 1.5574
Epoch 1/5, Batch 400, Loss: 1.5626
Epoch 1/5, Batch 500, Loss: 1.4693
Epoch 1/5, Batch 600, Loss: 1.4847
Epoch 1/5, Batch 700, Loss: 1.5171
Epoch 1/5, Batch 800, Loss: 1.4787
Epoch 1/5, Batch 900, Loss: 1.4628
Epoch 2/5, Batch 0, Loss: 1.4814

Epoch 2/5, Batch 100, Loss: 1.4871
Epoch 2/5, Batch 200, Loss: 1.4945
Epoch 2/5, Batch 300, Loss: 1.4770
Epoch 2/5, Batch 400, Loss: 1.5028
Epoch 2/5, Batch 500, Loss: 1.5238
Epoch 2/5, Batch 600, Loss: 1.5085
Epoch 2/5, Batch 700, Loss: 1.5098
Epoch 2/5, Batch 800, Loss: 1.4919
Epoch 2/5, Batch 900, Loss: 1.4660
Epoch 3/5, Batch 0, Loss: 1.4784
Epoch 3/5, Batch 100, Loss: 1.4724
Epoch 3/5, Batch 200, Loss: 1.4861
Epoch 3/5, Batch 300, Loss: 1.4770
Epoch 3/5, Batch 400, Loss: 1.4780
Epoch 3/5, Batch 500, Loss: 1.4865
Epoch 3/5, Batch 600, Loss: 1.5024
Epoch 3/5, Batch 700, Loss: 1.4951
Epoch 3/5, Batch 800, Loss: 1.4770
Epoch 3/5, Batch 900, Loss: 1.4613
Epoch 4/5, Batch 0, Loss: 1.4768
Epoch 4/5, Batch 100, Loss: 1.4785
Epoch 4/5, Batch 200, Loss: 1.4940
Epoch 4/5, Batch 300, Loss: 1.4612
Epoch 4/5, Batch 400, Loss: 1.4720
Epoch 4/5, Batch 500, Loss: 1.4621
Epoch 4/5, Batch 600, Loss: 1.4740
Epoch 4/5, Batch 700, Loss: 1.4617
Epoch 4/5, Batch 800, Loss: 1.4612
Epoch 4/5, Batch 900, Loss: 1.4612
Epoch 5/5, Batch 0, Loss: 1.4887
Epoch 5/5, Batch 100, Loss: 1.4618
Epoch 5/5, Batch 200, Loss: 1.4621
Epoch 5/5, Batch 300, Loss: 1.4612
Epoch 5/5, Batch 400, Loss: 1.4771
Epoch 5/5, Batch 500, Loss: 1.4612
Epoch 5/5, Batch 600, Loss: 1.4850
Epoch 5/5, Batch 700, Loss: 1.4768
Epoch 5/5, Batch 800, Loss: 1.4691
Epoch 5/5, Batch 900, Loss: 1.4814
Test Accuracy: 98.79%

Ques3.

```
import numpy as np
from glob import glob
from skimage import io, color, transform
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```python
def load_image(path, size=(128, 128)):
   return transform.resize(color.rgb2gray(io.imread(path)), size,
anti_aliasing=True)

def roberts_edge_detector(img):
   gx = img[:-1, :-1] - img[1:, 1:]
   gy = img[:-1, 1:] - img[1:, :-1]
   return np.pad(np.hypot(gx, gy), ((0, 1), (0, 1)), mode='edge')

def modified_hog(img, bins=9, cell_size=(16, 16), block_size=(2, 2)):
   gy, gx = np.gradient(img)
   magnitude = np.hypot(gx, gy)
   orientation = np.degrees(np.arctan2(gy, gx)) % 180
   sy, sx = img.shape
   cy, cx = cell_size
   ny, nx = sy // cy, sx // cx
   hist = np.zeros((ny, nx, bins))

   for i in range(ny):
       for j in range(nx):
           m = magnitude[i * cy:(i + 1) * cy, j * cx:(j + 1) * cx]
           a = orientation[i * cy:(i + 1) * cy, j * cx:(j + 1) * cx]
           hist[i, j, :], _ = np.histogram(a, bins=bins, range=(0, 180), weights=m)

   features = []
   by, bx = block_size
   for i in range(ny - by + 1):
       for j in range(nx - bx + 1):
           block = hist[i:i + by, j:j + bx, :].ravel()
           features.append(block / (np.linalg.norm(block) + 1e-6))

   return np.concatenate(features)

cat_images = glob('test_set/cats/*.jpg')
dog_images = glob('test_set/dogs/*.jpg')
image_paths = cat_images + dog_images
labels = np.array([0] * len(cat_images) + [1] * len(dog_images))
features = np.array([modified_hog(roberts_edge_detector(load_image(p))) for p in
image_paths])

X_train, X_test, y_train, y_test = train_test_split(features, labels,
test_size=0.2, random_state=42)
classifier = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
classifier.fit(X_train, y_train)
```

```python
accuracy = accuracy_score(y_test, classifier.predict(X_test))
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

A Brief Explanation of the above code

- **`load_image(path, size=(128, 128))`**
  This function reads an image from the given path, converts it to grayscale, and resizes it to 128×128 pixels using anti-aliasing.
- **`roberts_edge_detector(img)`**
  It applies the Roberts Cross Edge Detector to find edges in the image. It calculates horizontal and vertical gradients, computes the edge magnitude, and pads the result to maintain the original size.
- **`modified_hog(img, bins=9, cell_size=(16, 16), block_size=(2, 2))`**
  This function extracts Histogram of Oriented Gradients (HoG) features. It calculates gradient magnitudes and orientations, divides the image into small cells, creates histograms for gradient directions, and normalizes feature blocks to improve accuracy. It returns a feature vector for classification.
- **Loading and Labeling Dataset**
  The script collects all cat and dog image paths. It assigns label 0 to cats and label 1 to dogs.
- **Feature Extraction**
  Each image undergoes edge detection using the Roberts operator, followed by HoG feature extraction. The extracted features are stored for classification.
- **Train-Test Split**
  The dataset is split into 80% training and 20% testing to evaluate the model's performance.
- **Training the Classifier**
  A Random Forest classifier with 100 decision trees is trained on the extracted features.
- **Evaluating the Model**
  The trained model predicts labels for the test data, and the accuracy is computed and displayed.

```
Test Accuracy: 73.83%
```

```
        self.fc1 = nn.Linear(120 * 4 * 4, 84)
        self.fc2 = nn.Linear(84, 10)
        self.act = Swish()

    def forward(self, x):
        x = self.pool(self.act(self.conv1(x)))
        x = self.pool(self.act(self.conv2(x)))
        x = self.act(self.conv3(x))
        x = torch.flatten(x, 1)
        x = self.act(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)

transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])
train_loader = DataLoader(datasets.MNIST('./data', train=True,
download=True, transform=transform), batch_size=64, shuffle=True)
test_loader = DataLoader(datasets.MNIST('./data', train=False,
transform=transform), batch_size=64, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LeNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    model.train()
    for i, (x, y) in enumerate(train_loader):
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        loss = criterion(model(x), y)
        loss.backward()
        optimizer.step()
        if i % 100 == 0:
            print(f"Epoch {epoch+1}/5, Batch {i}, Loss:
{loss.item():.4f}")

model.eval()
corr, tot = 0, 0
with torch.no_grad():
    for x, y in test_loader:
        x, y = x.to(device), y.to(device)
        _, pred = torch.max(model(x), 1)
```

```
        tot += y.size(0)
        corr += (pred == y).sum().item()

print(f"Test Accuracy: {100 * corr / tot:.2f}%")
```

A Brief Explanation of the above code:

`Swish(nn.Module)`: Implements the Swish activation function, which is defined as x⋅sigmoid(x)x \cdot \text{sigmoid}(x)x⋅sigmoid(x), providing smooth, non-monotonic activation.

`LeNet(nn.Module)`: Defines a convolutional neural network (CNN) based on the LeNet-5 architecture.

- Contains three convolutional layers (`conv1`, `conv2`, `conv3`) with activation and pooling.
- Uses `MaxPool2d` to reduce spatial dimensions.
- Fully connected layers (`fc1`, `fc2`) for classification.
- Uses the Swish activation function and `softmax` for output probabilities.

Data Preprocessing & Loading:

- Applies transformations (tensor conversion and normalization) using `transforms.Compose`.
- Loads MNIST dataset using `DataLoader` for training and testing.

Model Training:

- Uses Cross-Entropy Loss (`nn.CrossEntropyLoss()`) as the criterion.
- Optimizes using Adam (`optim.Adam`).
- Runs for 5 epochs, updating weights using backpropagation.
- Prints loss every 100 batches.

Model Evaluation:

- Disables gradient computation (`torch.no_grad()`).
- Performs inference on the test set, calculating the accuracy.
- Prints the final test accuracy.

Epoch 1/5, Batch 0, Loss: 2.3023
Epoch 1/5, Batch 100, Loss: 1.5620
Epoch 1/5, Batch 200, Loss: 1.5059
Epoch 1/5, Batch 300, Loss: 1.5574
Epoch 1/5, Batch 400, Loss: 1.5626
Epoch 1/5, Batch 500, Loss: 1.4693
Epoch 1/5, Batch 600, Loss: 1.4847
Epoch 1/5, Batch 700, Loss: 1.5171
Epoch 1/5, Batch 800, Loss: 1.4787
Epoch 1/5, Batch 900, Loss: 1.4628
Epoch 2/5, Batch 0, Loss: 1.4814

Epoch 2/5, Batch 100, Loss: 1.4871
Epoch 2/5, Batch 200, Loss: 1.4945
Epoch 2/5, Batch 300, Loss: 1.4770
Epoch 2/5, Batch 400, Loss: 1.5028
Epoch 2/5, Batch 500, Loss: 1.5238
Epoch 2/5, Batch 600, Loss: 1.5085
Epoch 2/5, Batch 700, Loss: 1.5098
Epoch 2/5, Batch 800, Loss: 1.4919
Epoch 2/5, Batch 900, Loss: 1.4660
Epoch 3/5, Batch 0, Loss: 1.4784
Epoch 3/5, Batch 100, Loss: 1.4724
Epoch 3/5, Batch 200, Loss: 1.4861
Epoch 3/5, Batch 300, Loss: 1.4770
Epoch 3/5, Batch 400, Loss: 1.4780
Epoch 3/5, Batch 500, Loss: 1.4865
Epoch 3/5, Batch 600, Loss: 1.5024
Epoch 3/5, Batch 700, Loss: 1.4951
Epoch 3/5, Batch 800, Loss: 1.4770
Epoch 3/5, Batch 900, Loss: 1.4613
Epoch 4/5, Batch 0, Loss: 1.4768
Epoch 4/5, Batch 100, Loss: 1.4785
Epoch 4/5, Batch 200, Loss: 1.4940
Epoch 4/5, Batch 300, Loss: 1.4612
Epoch 4/5, Batch 400, Loss: 1.4720
Epoch 4/5, Batch 500, Loss: 1.4621
Epoch 4/5, Batch 600, Loss: 1.4740
Epoch 4/5, Batch 700, Loss: 1.4617
Epoch 4/5, Batch 800, Loss: 1.4612
Epoch 4/5, Batch 900, Loss: 1.4612
Epoch 5/5, Batch 0, Loss: 1.4887
Epoch 5/5, Batch 100, Loss: 1.4618
Epoch 5/5, Batch 200, Loss: 1.4621
Epoch 5/5, Batch 300, Loss: 1.4612
Epoch 5/5, Batch 400, Loss: 1.4771
Epoch 5/5, Batch 500, Loss: 1.4612
Epoch 5/5, Batch 600, Loss: 1.4850
Epoch 5/5, Batch 700, Loss: 1.4768
Epoch 5/5, Batch 800, Loss: 1.4691
Epoch 5/5, Batch 900, Loss: 1.4814
Test Accuracy: 98.79%

Ques3.

```
import numpy as np
from glob import glob
from skimage import io, color, transform
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```python
def load_image(path, size=(128, 128)):
   return transform.resize(color.rgb2gray(io.imread(path)), size,
anti_aliasing=True)

def roberts_edge_detector(img):
   gx = img[:-1, :-1] - img[1:, 1:]
   gy = img[:-1, 1:] - img[1:, :-1]
   return np.pad(np.hypot(gx, gy), ((0, 1), (0, 1)), mode='edge')

def modified_hog(img, bins=9, cell_size=(16, 16), block_size=(2, 2)):
   gy, gx = np.gradient(img)
   magnitude = np.hypot(gx, gy)
   orientation = np.degrees(np.arctan2(gy, gx)) % 180
   sy, sx = img.shape
   cy, cx = cell_size
   ny, nx = sy // cy, sx // cx
   hist = np.zeros((ny, nx, bins))

   for i in range(ny):
       for j in range(nx):
           m = magnitude[i * cy:(i + 1) * cy, j * cx:(j + 1) * cx]
           a = orientation[i * cy:(i + 1) * cy, j * cx:(j + 1) * cx]
           hist[i, j, :], _ = np.histogram(a, bins=bins, range=(0, 180), weights=m)

   features = []
   by, bx = block_size
   for i in range(ny - by + 1):
       for j in range(nx - bx + 1):
           block = hist[i:i + by, j:j + bx, :].ravel()
           features.append(block / (np.linalg.norm(block) + 1e-6))

   return np.concatenate(features)

cat_images = glob('test_set/cats/*.jpg')
dog_images = glob('test_set/dogs/*.jpg')
image_paths = cat_images + dog_images
labels = np.array([0] * len(cat_images) + [1] * len(dog_images))
features = np.array([modified_hog(roberts_edge_detector(load_image(p))) for p in
image_paths])

X_train, X_test, y_train, y_test = train_test_split(features, labels,
test_size=0.2, random_state=42)
classifier = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
classifier.fit(X_train, y_train)
```

```
accuracy = accuracy_score(y_test, classifier.predict(X_test))
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

<u>A Brief Explanation of the above code</u>

- **`load_image(path, size=(128, 128))`**
  This function reads an image from the given path, converts it to grayscale, and resizes it to 128×128 pixels using anti-aliasing.
- **`roberts_edge_detector(img)`**
  It applies the Roberts Cross Edge Detector to find edges in the image. It calculates horizontal and vertical gradients, computes the edge magnitude, and pads the result to maintain the original size.
- **`modified_hog(img, bins=9, cell_size=(16, 16), block_size=(2, 2))`**
  This function extracts Histogram of Oriented Gradients (HoG) features. It calculates gradient magnitudes and orientations, divides the image into small cells, creates histograms for gradient directions, and normalizes feature blocks to improve accuracy. It returns a feature vector for classification.
- **Loading and Labeling Dataset**
  The script collects all cat and dog image paths. It assigns label 0 to cats and label 1 to dogs.
- **Feature Extraction**
  Each image undergoes edge detection using the Roberts operator, followed by HoG feature extraction. The extracted features are stored for classification.
- **Train-Test Split**
  The dataset is split into 80% training and 20% testing to evaluate the model's performance.
- **Training the Classifier**
  A Random Forest classifier with 100 decision trees is trained on the extracted features.
- **Evaluating the Model**
  The trained model predicts labels for the test data, and the accuracy is computed and displayed.

```
Test Accuracy: 73.83%
```