Assignment 3

● Graded

Student

Manish Kumar Krishne Gowda

Total Points

100 / 100 pts

Question 1

Exercise 1 — **40** / 40 pts

1.1 Task 1 — **20** / 20 pts

✔ **– 0 pts** Correct

1.2 Task 2 — **20** / 20 pts

✔ **– 0 pts** Correct

Question 2

Exercise 2 — **30** / 30 pts

2.1 Task 1 — **10** / 10 pts

✔ **– 0 pts** Correct

2.2 Task 2 — **15** / 15 pts

✔ **– 0 pts** Correct

2.3 Task 3 — **5** / 5 pts

✔ **– 0 pts** Correct

Question 3

Exercise 3 — **30** / 30 pts

3.1 **Task 2** — **10** / 10 pts

✔ **– 0 pts** Correct

3.2 **Task 3** — **Resolved** **5** / 5 pts

✔ **– 0 pts** Correct

Regrade Request — **Submitted on: Oct 22**

Accuracy is high happens because the resnet has been modified to include a fully connected layer with 10 output nodes (but still untrained at this point). Becasue of this the accuracy slightly increases since there can be a small probability of output matches in predicted and ground truth. If the fully connected layer is not added, then the accuracy will be low 0% due to more number of classes to match. Both methods are correct at this point, since adding the fc layer to an already trained model and tailor it to ones task will complete a part of the solution and get it ready to solve the problem in our hand. Once the training is done in task 4, the accuracy increases.

The model was to be tested solely for performance evaluation without any modifications or alterations. Please keep that in mind next time.

Reviewed on: Oct 29

3.3 **Task 4** — **15** / 15 pts

✔ **– 0 pts** Correct

Question 4

Admin only (for late penalty) - assign to the first page — **0** / 0 pts

✔ **– 0 pts** Correct

No questions assigned to the following page.

ECE 57000 Assignment 3 Exercise

Your Name: Manish Kumar Krishne Gowda

Prepare the pacakge we will use.

In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [3]:
```
!pwd
```

```
/content
```

In [4]:
```
import time
import os

from typing import List, Dict

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.models as models
import torchvision.transforms as transforms

import matplotlib.pyplot as plt
import pickle
```

Exercise 0: Train your model on GPU (0 points)

Some tasks in this assignment can take a long time if you run it on the CPU. For example, based on our solution of Exercise 3 Task 4 (Transfer Learning: finetuning of a pretrained model (resnet18)), it will take roughly 2 hours to train the model end-to-end (complete model and not only the last fc layer) for 1 epoch on CPU. Hence, we highly recommend you try to train your model on GPU.

To do so, first you need to enable GPU on Colab (this will restart the runtime). Click `Runtime` -> `Change runtime type` and select the `Hardware accelerator` there. You can then run the following code to see if the GPU is correctly initialized and available.

Note: If you would like to avoid GPU overages on Colab, we would suggest writing and debugging your code before switching on the GPU runtime. Otherwise, the time you spent debugging code will likely count against your GPU usage. Once you have the code running, you can switch on the GPU runtime and train the model much faster.

In [5]:
```
print(f'Can I can use GPU now? -- {torch.cuda.is_available()}')
```

No questions assigned to the following page.

```
Can I can use GPU now? -- True
```

You must manually move your model and data to the GPU (and sometimes back to the cpu)

After setting the GPU up on colab, then you should put your **model** and **data** to GPU. We give a simple example below. You can use `to` function for this task. See torch.Tensor.to to move a tensor to the GPU (probably your mini-batch of data in each iteration) or torch.nn.Module.to to move your NN model to GPU (assuming you create subclass torch.nn.Module). Note that `to()` of tensor returns a NEW tensor while `to` of a NN model will apply this in-place. To be safe, the best semantics are `obj = obj.to(device)`. For printing, you will need to move a tensor back to the CPU via the `cpu()` function.

Once the model and input data are on the GPU, everything else can be done the same. This is the beauty of PyTorch GPU acceleration. None of the other code needs to be altered.

To summarize, you need to 1) enable GPU acceleration in Colab, 2) put the model on the GPU, and 3) put the input data (i.e., the batch of samples) onto the GPU using `to()` after it is loaded by the data loaders (usually you only put one batch of data on the GPU at a time).

In [6]:
```
rand_tensor = torch.rand(5,2)
simple_model = nn.Sequential(nn.Linear(2,10), nn.ReLU(), nn.Linear(10,1))
print(f'input is on {rand_tensor.device}')
print(f'model parameters are on {[param.device for param in simple_model.parameters()]
print(f'output is on {simple_model(rand_tensor).device}')

# device = torch.device('cuda')

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# ----------- <Your code> ---------------
# Move rand_tensor and model onto the GPU device
rand_tensor = rand_tensor.to(device)
simple_model = simple_model.to(device)

# --------- <End your code> -------------
print(f'input is on {rand_tensor.device}')
print(f'model parameters are on {[param.device for param in simple_model.parameters()]
print(f'output is on {simple_model(rand_tensor).device}')
```

```
input is on cpu
model parameters are on [device(type='cpu'), device(type='cpu'), device(type='cpu'),
device(type='cpu')]
output is on cpu
input is on cuda:0
model parameters are on [device(type='cuda', index=0), device(type='cuda', index=0),
device(type='cuda', index=0), device(type='cuda', index=0)]
output is on cuda:0
```

In [7]:
```
total_params = sum(p.numel() for p in simple_model.parameters())
print(f'Number of parameters: {total_params}')
```

```
Number of parameters: 41
```

No questions assigned to the following page.

Exercise 1: Why use a CNN rather than only fully connected layers? (40 points)

In this exercise, you will build two models for the **MNIST** dataset: one uses only fully connected layers and another uses a standard CNN layout (convolution layers everywhere except the last layer is fully connected layer). Note, you will need to use cross entropy loss as your objective function. The two models should be built with roughly the same accuracy performance, your task is to compare the number of network parameters (a huge number of parameters can affect training/testing time, memory requirements, overfitting, etc.).

Task 1: Prepare train and test function

We will create our train and test procedure in these two functions. The train function should apply one epoch of training. The functions inputs should take everything we need for training and testing and return some logs.

Arguments requirement:

- For the `train` function, it takes the `model`, `loss_fn`, `optimizer`, `train_loader`, and `epoch` as arguments.
    - `model`: the classifier, or deep neural network, should be an instance of `nn.Module`.
    - `loss_fn`: the loss function instance. For example, `nn.CrossEntropy()`, or `nn.L1Loss()`, etc.
    - `optimizer`: should be an instance of `torch.optim.Optimizer`. For example, it could be `optim.SGD()` or `optim.Adam()`, etc.
    - `train_loader`: should be an instance of `torch.utils.data.DataLoader`.
    - `epoch`: the current number of epoch. Only used for log printing.(default: 1.)
- For the `test` function, it takes all the inputs above except for the optimizer (and it takes a test loader instead of a train loader).

Log requirement:

Here are some further requirements:

- In the `train` function, print the log 8-10 times per epoch. The print statement should be:

```
print(f'Epoch {epoch}:
[{batch_idx*len(images)}/{len(train_loader.dataset)}] Loss:
{loss.item():.3f}')
```

- In the `test` function, print the log after the testing. The print statement is:

```
print(f"Test result on epoch {epoch}: total sample: {total_num}, Avg
loss: {test_stat['loss']:.3f}, Acc: {100*test_stat['accuracy']:.3f}%")
```

Return requirement

Question assigned to the following page: 1.1

- The `train` function should return a list, which the element is the loss per batch, i.e., one loss value for every batch.
- The `test` function should return a dictionary with three keys: "loss", "accuracy", and "prediction". The values are the average loss of all the testset, average accuracy of all the test dataset, and the prediction of all test dataset.

Other requirement:

- In the `train` function, the model should be updated in-place, i.e., do not copy the model inside `train` function.

In [56]:
```python
def train(model: nn.Module,
          loss_fn: nn.modules.loss._Loss,
          optimizer: torch.optim.Optimizer,
          train_loader: torch.utils.data.DataLoader,
          epoch: int=0)-> List:
    # ----------- <Your code> ---------------
    model.train()  # Set the model to training mode
    train_loss = []  # To store loss for each batch

    # Loop over batches
    for batch_idx, (images, targets) in enumerate(train_loader):
        # Zero the gradients before forward pass
        optimizer.zero_grad()

        # Forward pass: compute predictions and loss
        outputs = model(images)
        loss = loss_fn(outputs, targets)

        # Backward pass: compute gradients
        loss.backward()

        # Update weights using optimizer
        optimizer.step()

        # Store loss for this batch
        train_loss.append(loss.item())

        # Print log 8-10 times per epoch
        if batch_idx % (len(train_loader) // 10) == 0:
            print(f'Epoch {epoch}: [{batch_idx*len(images)}/{len(train_loader.dataset)
    # ----------- <End Your code> ---------------
    assert len(train_loss) == len(train_loader)
    return train_loss

def test(model: nn.Module,
         loss_fn: nn.modules.loss._Loss,
         test_loader: torch.utils.data.DataLoader,
         epoch: int=0)-> Dict:
    # ----------- <Your code> ---------------
    model.eval()  # Set the model to evaluation mode
    test_loss = 0.0
    correct = 0
    predictions = []
    # Disable gradient computation for testing
    with torch.no_grad():
        for images, targets in test_loader:
```

Question assigned to the following page: 1.1

```
            # Forward pass: compute predictions
            outputs = model(images)

            # Compute loss
            loss = loss_fn(outputs, targets)
            test_loss += loss.item()  # Accumulate the total loss

            # Get predictions: assuming classification task
            pred = outputs.argmax(dim=1, keepdim=True)  # Get index of the max log-prc
            predictions.append(pred.flatten())

            # Count correct predictions
            correct += pred.eq(targets.view_as(pred)).sum().item()

    # Convert predictions list to a 1D tensor
    predictions = torch.cat(predictions)

    # Average loss over all test data
    test_loss /= len(test_loader)

    # Accuracy calculation
    accuracy = correct / len(test_loader.dataset)

    # Create dictionary with test statistics
    test_stat = {
        "loss": test_loss,
        "accuracy": accuracy,
        "prediction": predictions
    }
    # ----------- <Your code> ---------------
    # dictionary should include loss, accuracy and prediction
    assert "loss" and "accuracy" and "prediction" in test_stat.keys()
    # "prediction" value should be a 1D tensor
    assert len(test_stat["prediction"]) == len(test_loader.dataset)
    assert isinstance(test_stat["prediction"], torch.Tensor)
    return test_stat
```

Task 2: Following the structure used in the instructions, you should create

- One network named `OurFC` which should consist with only fully connected layers
    - You should decide how many layers and how many hidden dimensions you want in your network
    - Your final accuracy on the test dataset should lie roughly around 97% (±2%)
    - There is no need to make the neural network unnecessarily complex, your total training time should no longer than 3 mins
- Another network named `OurCNN` which applys a standard CNN structure
    - Again, you should decide how many layers and how many channels you want for each layer.
    - Your final accuracy on the test dataset should lie roughly around 97% (±2%)
    - A standard CNN structure can be composed as **[Conv2d, MaxPooling, ReLU] x num_conv_layers + FC x num_fc_layers**

Question assigned to the following page: 1.2

- Train and test your network on MNIST data as in the instructions.
- Notice You can always use the `train` and `test` function you write throughout this assignment.
- The code below will also print out the number of parameters for both neural networks to allow comparison.
- (You can use multiple cells if helpful but make sure to run all of them to receive credit.)

In [8]:
```
# Download MNIST and transformation
# ----------- <Your code> ---------------
import torchvision

"""
Here the transform is a pipeline containing two seperate transforms:
1. Transform the data into tensor type
2. Normalize the dataset by a giving mean and std.
  (Those number is given as the global mean and standard deviation of MNIST dataset)
"""
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                      torchvision.transforms.Normalize((0.1307,),(0.3081,))])

train_dataset = torchvision.datasets.MNIST('data', train=True, download=True, transfor
test_dataset = torchvision.datasets.MNIST('data', train=False, download=True, transfor
# ----------- <End Your code> -------------
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
to data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 16300927.52it/s]
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
to data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 491829.72it/s]
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz t
o data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:00<00:00, 4476244.27it/s]
```

Question assigned to the following page: 1.2

```
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz t
o data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 5245189.64it/s]
Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw

Dataset MNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.1307,), std=(0.3081,))
           )
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.1307,), std=(0.3081,))
           )
```

In [11]:

```python
print(train_dataset)
print(test_dataset)

batch_size_train, batch_size_test = 64, 1000

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size_train,
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size_test, sh

print(train_loader)
batch_idx, (images, targets) = next(enumerate(train_loader))
print(f'current batch index is {batch_idx}')
print(f'images has shape {images.size()}')
print(f'targets has shape {targets.size()}')
```

Question assigned to the following page: 1.2

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.1307,), std=(0.3081,))
           )
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.1307,), std=(0.3081,))
           )
<torch.utils.data.dataloader.DataLoader object at 0x7d894c7e27a0>
current batch index is 0
images has shape torch.Size([64, 1, 28, 28])
targets has shape torch.Size([64])
```

In [10]:

```python
# Build OurFC class and OurCNN class.
# ----------- <Your code> ---------------
import torch.nn as nn
import torch.nn.functional as F

class OurCNN(nn.Module):
    def __init__(self):
        super(OurCNN, self).__init__()

        # Define two convolutional layers
        self.conv1 = nn.Conv2d(1, 3, kernel_size=3)  # First conv layer
        self.conv2 = nn.Conv2d(3, 3, kernel_size=3)  # Second conv layer

        # Define fully connected layer
        # After conv2, the image size is reduced to 11x11. 3 channels from conv2, and
        self.fc = nn.Linear(3 * 11 * 11, 10)

    def forward(self, x):
        # First convolutional layer with relu and max pooling
        x = self.conv1(x)                       # x shape: (batch_size, 3, 26, 26)
        x = F.relu(F.max_pool2d(x, 2))          # x shape: (batch_size, 3, 13, 13)

        # Second convolutional layer with relu (no pooling after this one)
        x = self.conv2(x)                       # x shape: (batch_size, 3, 11, 11)

        # Flatten the tensor for the fully connected layer
        x = x.view(-1, 3 * 11 * 11)             # x shape: (batch_size, 363)

        # Fully connected layer with relu
        x = F.relu(self.fc(x))                  # x shape: (batch_size, 10)

        # Log softmax for the output
        return F.log_softmax(x, dim=-1)

class OurFC(nn.Module):
    def __init__(self):
        super(OurFC, self).__init__()
```

Question assigned to the following page: 1.2

```
        # Define fully connected layers
        self.flatten = nn.Flatten()  # Flatten the 28x28 input images to a 1D vector
        self.fc1 = nn.Linear(28 * 28, 363)  # First hidden layer
        self.fc2 = nn.Linear(363, 363)      # Second hidden layer
        self.fc3 = nn.Linear(363, 363)      # Third hidden layer
        self.output = nn.Linear(363, 10)    # Output layer (10 classes for classificat

        # Define activation function (ReLU is commonly used for hidden layers)
        self.relu = nn.ReLU()

    def forward(self, x):
        # Forward pass through the network
        x = self.flatten(x)      # Flatten the input image
        x = self.relu(self.fc1(x))  # First hidden layer + activation
        x = self.relu(self.fc2(x))  # Second hidden layer + activation
        x = self.relu(self.fc3(x))  # Third hidden layer + activation
        x = self.output(x)          # Output layer (no activation because it's for cla
        return x
# ----------- <End Your code> -------------
```

In [18]:

```
# Let's first train the FC model. Below are there common hyperparameters.
# ----------- <Your code> ---------------
criterion = nn.CrossEntropyLoss()
max_epoch = 3
model = OurFC()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)  # SGD with momentum
# File paths for saving model and losses
model_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/model.pth'
loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/train_loss.pkl'

if os.path.exists(model_file) and os.path.exists(loss_file):
    model.load_state_dict(torch.load(model_file))  # Load model weights
    with open(loss_file, 'rb') as f:
        train_losses = pickle.load(f)  # Load training losses
else:
    train_losses = []  # Initialize list if not present

start = time.time()
for epoch in range(1, max_epoch + 1):
    epoch_loss = train(model, criterion, optimizer, train_loader, epoch)
    train_losses.append(epoch_loss)  # Append the loss for this epoch

    # Save the model after each epoch
    torch.save(model.state_dict(), model_file)

# Save the training losses to a file
with open(loss_file, 'wb') as f:
    pickle.dump(train_losses, f)
# ----------- <End Your code> ---------------
end = time.time()
print(f'Finished Training after {end-start} s')
```

Question assigned to the following page: 1.2

```
Epoch 1: [0/60000] Loss: 2.298
Epoch 1: [5952/60000] Loss: 0.520
Epoch 1: [11904/60000] Loss: 0.214
Epoch 1: [17856/60000] Loss: 0.068
Epoch 1: [23808/60000] Loss: 0.148
Epoch 1: [29760/60000] Loss: 0.240
Epoch 1: [35712/60000] Loss: 0.338
Epoch 1: [41664/60000] Loss: 0.099
Epoch 1: [47616/60000] Loss: 0.195
Epoch 1: [53568/60000] Loss: 0.139
Epoch 1: [59520/60000] Loss: 0.132
Epoch 2: [0/60000] Loss: 0.117
Epoch 2: [5952/60000] Loss: 0.103
Epoch 2: [11904/60000] Loss: 0.175
Epoch 2: [17856/60000] Loss: 0.109
Epoch 2: [23808/60000] Loss: 0.231
Epoch 2: [29760/60000] Loss: 0.109
Epoch 2: [35712/60000] Loss: 0.048
Epoch 2: [41664/60000] Loss: 0.073
Epoch 2: [47616/60000] Loss: 0.062
Epoch 2: [53568/60000] Loss: 0.165
Epoch 2: [59520/60000] Loss: 0.093
Epoch 3: [0/60000] Loss: 0.107
Epoch 3: [5952/60000] Loss: 0.048
Epoch 3: [11904/60000] Loss: 0.060
Epoch 3: [17856/60000] Loss: 0.076
Epoch 3: [23808/60000] Loss: 0.025
Epoch 3: [29760/60000] Loss: 0.028
Epoch 3: [35712/60000] Loss: 0.027
Epoch 3: [41664/60000] Loss: 0.136
Epoch 3: [47616/60000] Loss: 0.095
Epoch 3: [53568/60000] Loss: 0.059
Epoch 3: [59520/60000] Loss: 0.052
Finished Training after 45.72572946548462 s
```

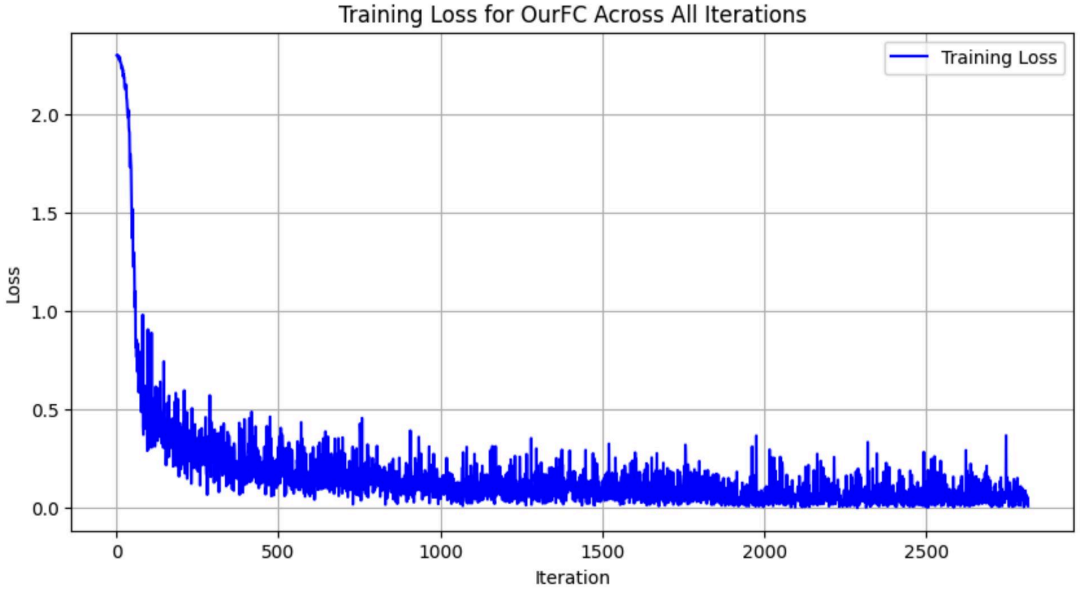
In [35]:

```python
# File paths for loading the model and losses
model_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/model.pth'
loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/train_loss.pkl'

# Initialize the model
model = OurFC()

# Load the trained model weights
model.load_state_dict(torch.load(model_file))
model.eval()  # Set the model to evaluation mode

# Load training losses
with open(loss_file, 'rb') as f:
    train_losses = pickle.load(f)

# Flatten the training losses across all epochs
flattened_losses = [loss for epoch_loss in train_losses for loss in epoch_loss]

# Calculate the total number of iterations (assuming batch size is known)
# You may need to replace `batch_size` with your actual batch size
total_iterations = len(flattened_losses)

# Generate x-axis values (iterations)
x_values = range(1, total_iterations + 1)
```

Question assigned to the following page: 1.2

```
# Plot the training losses
plt.figure(figsize=(10, 5))
plt.plot(x_values, flattened_losses, label='Training Loss', color='blue')
plt.title('Training Loss for OurFC Across All Iterations')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.grid()
plt.legend()
plt.show()
```

```
<ipython-input-35-968b78338ac9>:9: FutureWarning: You are using `torch.load` with `we
ights_only=False` (the current default value), which uses the default pickle module i
mplicitly. It is possible to construct malicious pickle data which will execute arbit
rary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURIT
Y.md#untrusted-models for more details). In a future release, the default value for `
weights_only` will be flipped to `True`. This limits the functions that could be exec
uted during unpickling. Arbitrary objects will no longer be allowed to be loaded via
this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.add_safe_globals`. We recommend you start setting `weights_only=True` for any use c
ase where you don't have full control of the loaded file. Please open an issue on Git
Hub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_file))
```



In [29]:

```
# File paths for loading model and losses
model_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/model.pth'
# Initialize the model
criterion = nn.CrossEntropyLoss()
model = OurFC()
# Load the trained model for testing
model.load_state_dict(torch.load(model_file))
model.eval()  # Set the model to evaluation mode

# Testing the model
test_stat  = test(model, criterion, test_loader)

# Print the testing loss and accuracy
print(f'Test Loss: {test_stat["loss"]:.3f}')
print(f'Test Accuracy: {test_stat["accuracy"]*100:.2f}%')
```

Question assigned to the following page: 1.2

```
<ipython-input-29-146aafd2fb8d>:7: FutureWarning: You are using `torch.load` with `we
ights_only=False` (the current default value), which uses the default pickle module i
mplicitly. It is possible to construct malicious pickle data which will execute arbit
rary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURIT
Y.md#untrusted-models for more details). In a future release, the default value for `
weights_only` will be flipped to `True`. This limits the functions that could be exec
uted during unpickling. Arbitrary objects will no longer be allowed to be loaded via
this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.add_safe_globals`. We recommend you start setting `weights_only=True` for any use c
ase where you don't have full control of the loaded file. Please open an issue on Git
Hub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_file))
Test Loss: 0.075
Test Accuracy: 97.64%
```

In [19]:

```python
# Let's then train the OurCNN model.
# ----------- <Your code> ---------------
model_cnn = OurCNN()
# Hyperparameters
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_cnn.parameters(), lr=0.01, momentum=0.9)  # SGD with momer
max_epoch = 3

# File paths for saving model and losses
model_cnn_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cnn_model.pth'
cnn_loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cnn_train_loss.pkl'

# Load model and losses if they exist
if os.path.exists(model_cnn_file) and os.path.exists(cnn_loss_file):
    model_cnn.load_state_dict(torch.load(model_cnn_file))  # Load model weights
    with open(cnn_loss_file, 'rb') as f:
        cnn_train_losses = pickle.load(f)  # Load training losses
else:
    cnn_train_losses = []  # Initialize list if not present

start = time.time()

for epoch in range(1, max_epoch + 1):
    epoch_loss = train(model_cnn, criterion, optimizer, train_loader, epoch)
    cnn_train_losses.append(epoch_loss)  # Append the loss for this epoch

    # Save the model after each epoch
    torch.save(model_cnn.state_dict(), model_cnn_file)

# Save the training losses to a file
with open(cnn_loss_file, 'wb') as f:
    pickle.dump(cnn_train_losses, f)
# ----------- <End Your code> ---------------
end = time.time()
print(f'Finished Training after {end-start} s ')
```

Question assigned to the following page: 1.2

```
Epoch 1: [0/60000] Loss: 2.395
Epoch 1: [5952/60000] Loss: 0.381
Epoch 1: [11904/60000] Loss: 0.318
Epoch 1: [17856/60000] Loss: 0.413
Epoch 1: [23808/60000] Loss: 0.169
Epoch 1: [29760/60000] Loss: 0.565
Epoch 1: [35712/60000] Loss: 0.127
Epoch 1: [41664/60000] Loss: 0.120
Epoch 1: [47616/60000] Loss: 0.126
Epoch 1: [53568/60000] Loss: 0.211
Epoch 1: [59520/60000] Loss: 0.109
Epoch 2: [0/60000] Loss: 0.082
Epoch 2: [5952/60000] Loss: 0.281
Epoch 2: [11904/60000] Loss: 0.159
Epoch 2: [17856/60000] Loss: 0.178
Epoch 2: [23808/60000] Loss: 0.147
Epoch 2: [29760/60000] Loss: 0.065
Epoch 2: [35712/60000] Loss: 0.124
Epoch 2: [41664/60000] Loss: 0.097
Epoch 2: [47616/60000] Loss: 0.151
Epoch 2: [53568/60000] Loss: 0.048
Epoch 2: [59520/60000] Loss: 0.054
Epoch 3: [0/60000] Loss: 0.126
Epoch 3: [5952/60000] Loss: 0.152
Epoch 3: [11904/60000] Loss: 0.044
Epoch 3: [17856/60000] Loss: 0.046
Epoch 3: [23808/60000] Loss: 0.078
Epoch 3: [29760/60000] Loss: 0.079
Epoch 3: [35712/60000] Loss: 0.031
Epoch 3: [41664/60000] Loss: 0.077
Epoch 3: [47616/60000] Loss: 0.117
Epoch 3: [53568/60000] Loss: 0.220
Epoch 3: [59520/60000] Loss: 0.170
Finished Training after 49.88678979873657 s
```

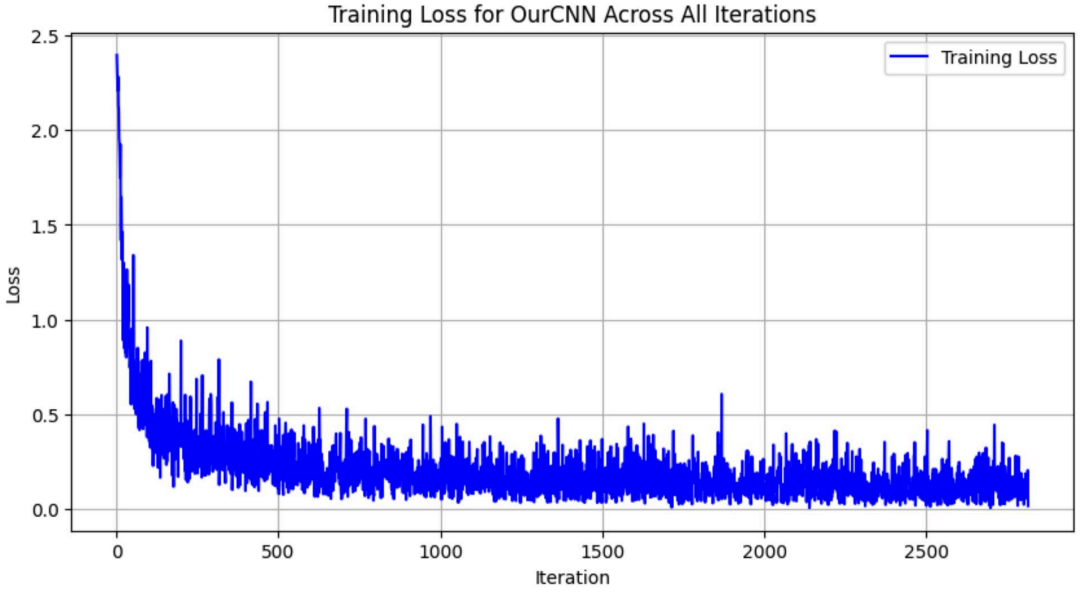
In [36]:

```python
# File paths for loading the model and losses
model_cnn_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cnn_model.pth'
cnn_loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cnn_train_loss.pkl'

# Initialize the model
model = OurCNN()

# Load the trained model weights
model.load_state_dict(torch.load(model_cnn_file))
model.eval()  # Set the model to evaluation mode

# Load training losses
with open(cnn_loss_file, 'rb') as f:
    train_losses = pickle.load(f)

# Flatten the training losses across all epochs
flattened_losses = [loss for epoch_loss in train_losses for loss in epoch_loss]

# Calculate the total number of iterations (assuming batch size is known)
# You may need to replace `batch_size` with your actual batch size
total_iterations = len(flattened_losses)

# Generate x-axis values (iterations)
x_values = range(1, total_iterations + 1)
```

Question assigned to the following page: 1.2

```
# Plot the training losses
plt.figure(figsize=(10, 5))
plt.plot(x_values, flattened_losses, label='Training Loss', color='blue')
plt.title('Training Loss for OurCNN Across All Iterations')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.grid()
plt.legend()
plt.show()
```

```
<ipython-input-36-07ef0f732239>:9: FutureWarning: You are using `torch.load` with `we
ights_only=False` (the current default value), which uses the default pickle module i
mplicitly. It is possible to construct malicious pickle data which will execute arbit
rary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURIT
Y.md#untrusted-models for more details). In a future release, the default value for `
weights_only` will be flipped to `True`. This limits the functions that could be exec
uted during unpickling. Arbitrary objects will no longer be allowed to be loaded via
this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.add_safe_globals`. We recommend you start setting `weights_only=True` for any use c
ase where you don't have full control of the loaded file. Please open an issue on Git
Hub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_cnn_file))
```



In [27]:
```
# File paths for loading model and losses
model_cnn_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cnn_model.pth'
criterion = nn.CrossEntropyLoss()
# Initialize the model
model = OurCNN()
# Load the trained model for testing
model.load_state_dict(torch.load(model_cnn_file))
model.eval()  # Set the model to evaluation mode

# Testing the model
test_stat  = test(model, criterion, test_loader)

# Print the testing loss and accuracy
print(f'Test Loss: {test_stat["loss"]:.3f}')
print(f'Test Accuracy: {test_stat["accuracy"]*100:.2f}%')
```

Questions assigned to the following page: 2.1 and 1.2

```
<ipython-input-27-ffb64fbbf39c>:7: FutureWarning: You are using `torch.load` with `we
ights_only=False` (the current default value), which uses the default pickle module i
mplicitly. It is possible to construct malicious pickle data which will execute arbit
rary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURIT
Y.md#untrusted-models for more details). In a future release, the default value for `
weights_only` will be flipped to `True`. This limits the functions that could be exec
uted during unpickling. Arbitrary objects will no longer be allowed to be loaded via
this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.add_safe_globals`. We recommend you start setting `weights_only=True` for any use c
ase where you don't have full control of the loaded file. Please open an issue on Git
Hub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_cnn_file))
Test Loss: 0.116
Test Accuracy: 96.53%
```

In [20]:
```
ourfc = OurFC()
total_params = sum(p.numel() for p in ourfc.parameters())
print(f'OurFC has a total of {total_params} parameters')

ourcnn = OurCNN()
total_params = sum(p.numel() for p in ourcnn.parameters())
print(f'OurCNN has a total of {total_params} parameters')
```

```
OurFC has a total of 552859 parameters
OurCNN has a total of 3754 parameters
```

Questions (0 points, just for understanding): Which one has more parameters? Which one is likely to have less computational cost when deployed? Which one took longer to train?

OurFC has more parameters (552,859) compared to OurCNN (3,754). OurCNN is likely to have less computational cost when deployed due to fewer parameters. OurCNN took longer to train, however usually CNNs take lesser times compared to FCNNs for same number of layers. The difference here is likely due to too few parameters and layers to make full utilisation of GPU capabilities

Exercise 2: Train classifier on CIFAR-10 data. (30 points)

Now, lets move our dataset to color images. CIFAR-10 dataset is another widely used dataset. Here all images have colors, i.e each image has 3 color channels instead of only one channel in MNIST. You need to pay more attention to the dimension of the data as it passes through the layers of your network.

Task 1: Create data loaders

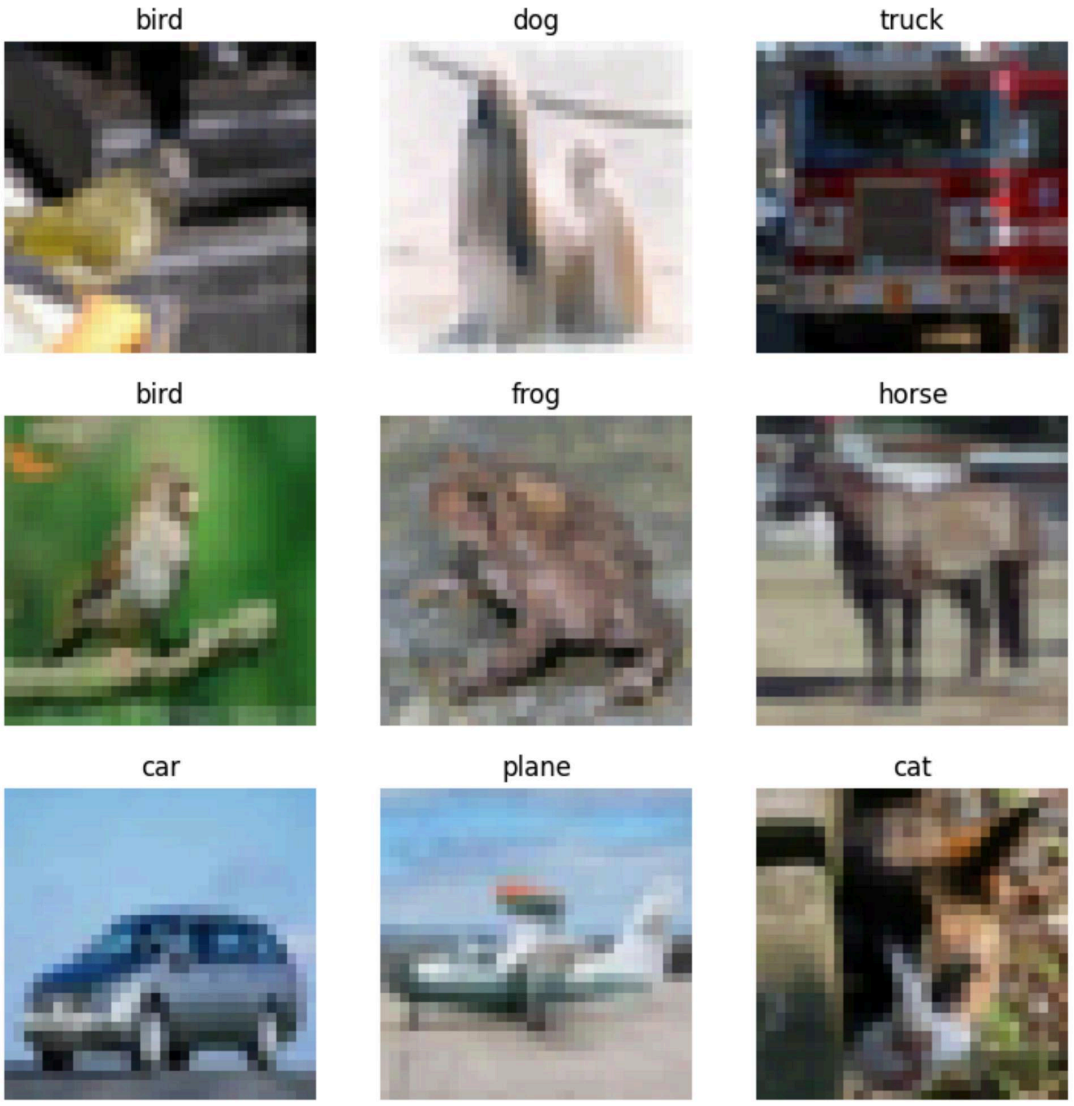
- Load CIFAR10 train and test datas with appropriate composite transform where the normalize transform should be `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])` .
- Set up a `train_loader` and `test_loader` for the CIFAR-10 data with a batch size of 9 similar to the instructions.
- The code below will plot a 3 x 3 subplot of images including their labels. (do not modify)

Question assigned to the following page: 2.1

In [33]:
```
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru

# Create the appropriate transform, load/download CIFAR10 train and test datasets with
# ----------- <Your code> ---------------
# Define the composite transform
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert images to PyTorch tensors
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize with mean and
])

# Load and download CIFAR10 train and test datasets
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
# ----------- <End Your code> ---------------

# Define trainloader and testloader
# ----------- <Your code> ---------------
batch_size = 9

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuff
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle
# ----------- <End Your code> ---------------

# Code to display images
batch_idx, (images, targets) = next(enumerate(train_loader)) #fix!!!!!
fig, ax = plt.subplots(3,3,figsize = (9,9))
for i in range(3):
    for j in range(3):
        image = images[i*3+j].permute(1,2,0)
        image = image/2 + 0.5
        ax[i,j].imshow(image)
        ax[i,j].set_axis_off()
        ax[i,j].set_title(f'{classes[targets[i*3+j]]}')
fig.show()
```

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

Questions assigned to the following page: 2.1 and 2.2



In [34]:
```
print(train_loader)
batch_idx, (images, targets) = next(enumerate(train_loader))
print(f'current batch index is {batch_idx}')
print(f'images has shape {images.size()}')
print(f'targets has shape {targets.size()}')
```

```
<torch.utils.data.dataloader.DataLoader object at 0x7c49d1191780>
current batch index is 0
images has shape torch.Size([9, 3, 32, 32])
targets has shape torch.Size([9])
```

Task 2: Create CNN and train it

Set up a convolutional neural network and have your data trained on it. You have to decide all the details in your network, overall your neural network should meet the following standards to receive full credit:

- You should not use more than three convolutional layers and three fully connected layers

Question assigned to the following page: 2.2

- Accuracy on the test dataset should be **above** 50%

In [15]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

# Create CNN network.
# ----------- <Your code> ---------------
class OurCIFARCNN(nn.Module):
    def __init__(self):
        super(OurCIFARCNN, self).__init__()

        # Define three convolutional layers
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3)  # First conv layer
        self.bn1 = nn.BatchNorm2d(16)                 # Batch normalization for the fi

        self.conv2 = nn.Conv2d(16, 32, kernel_size=3) # Second conv layer
        self.bn2 = nn.BatchNorm2d(32)                 # Batch normalization for the se

        self.conv3 = nn.Conv2d(32, 64, kernel_size=3) # Third conv layer
        self.bn3 = nn.BatchNorm2d(64)                 # Batch normalization for the th

        # Define fully connected layers
        self.fc1 = nn.Linear(64 * 4 * 4, 256)        # First fully connected layer
        self.fc2 = nn.Linear(256, 128)               # Second fully connected layer
        self.fc3 = nn.Linear(128, 10)                # Output layer

    def forward(self, x):
        # First convolutional layer with batch normalization, relu, and max pooling
        x = self.conv1(x)                            # x shape: (batch_size, 16, 30, 30)
        x = self.bn1(x)                              # Apply batch normalization
        x = F.relu(x)                                # ReLU activation
        x = F.max_pool2d(x, 2)                       # x shape: (batch_size, 16, 15, 15)

        # Second convolutional layer with batch normalization, relu, and max pooling
        x = self.conv2(x)                            # x shape: (batch_size, 32, 13, 13)
        x = self.bn2(x)                              # Apply batch normalization
        x = F.relu(x)                                # ReLU activation
        x = F.max_pool2d(x, 2)                       # x shape: (batch_size, 32, 6, 6)

        # Third convolutional layer with batch normalization and relu (NO max pooling
        x = self.conv3(x)                            # x shape: (batch_size, 64, 4, 4)
        x = self.bn3(x)                              # Apply batch normalization
        x = F.relu(x)                                # ReLU activation

        # Flatten the tensor for the fully connected layers
        x = x.view(-1, 64 * 4 * 4)                   # x shape: (batch_size, 1024)

        # First fully connected layer
        x = F.relu(self.fc1(x))                      # x shape: (batch_size, 256)

        # Second fully connected layer
        x = F.relu(self.fc2(x))                      # x shape: (batch_size, 128)

        # Output layer with log softmax
        return F.log_softmax(self.fc3(x), dim=-1) # x shape: (batch_size, 10)
# ----------- <End Your code> ---------------
```

Question assigned to the following page: 2.2

In [16]:

```python
# Initialize and train the OurCIFARCNN model.
# ----------- <Your code> ---------------
model_cnn = OurCIFARCNN()  # Initialize the model

# Hyperparameters
criterion = nn.CrossEntropyLoss()  # Loss function
optimizer = optim.SGD(model_cnn.parameters(), lr=0.01, momentum=0.9)  # SGD with momer
max_epoch = 4  # Number of epochs

# File paths for saving model and losses
model_cnn_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cifar_cnn_model.pth
cnn_loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cifar_cnn_train_loss

# Load model and losses if they exist
if os.path.exists(model_cnn_file) and os.path.exists(cnn_loss_file):
    model_cnn.load_state_dict(torch.load(model_cnn_file))  # Load model weights
    with open(cnn_loss_file, 'rb') as f:
        cnn_train_losses = pickle.load(f)  # Load training losses
else:
    cnn_train_losses = []  # Initialize list if not present

start = time.time()

# Training loop using the existing train function
for epoch in range(1, max_epoch + 1):
    epoch_loss = train(model_cnn, criterion, optimizer, train_loader, epoch)
    cnn_train_losses.append(epoch_loss)  # Append the loss for this epoch

    # Save the model after each epoch
    torch.save(model_cnn.state_dict(), model_cnn_file)

# Save the training losses to a file
with open(cnn_loss_file, 'wb') as f:
    pickle.dump(cnn_train_losses, f)

# ----------- <End Your code> ---------------
end = time.time()
print(f'Finished Training after {end-start} s ')
```

Question assigned to the following page: 2.2

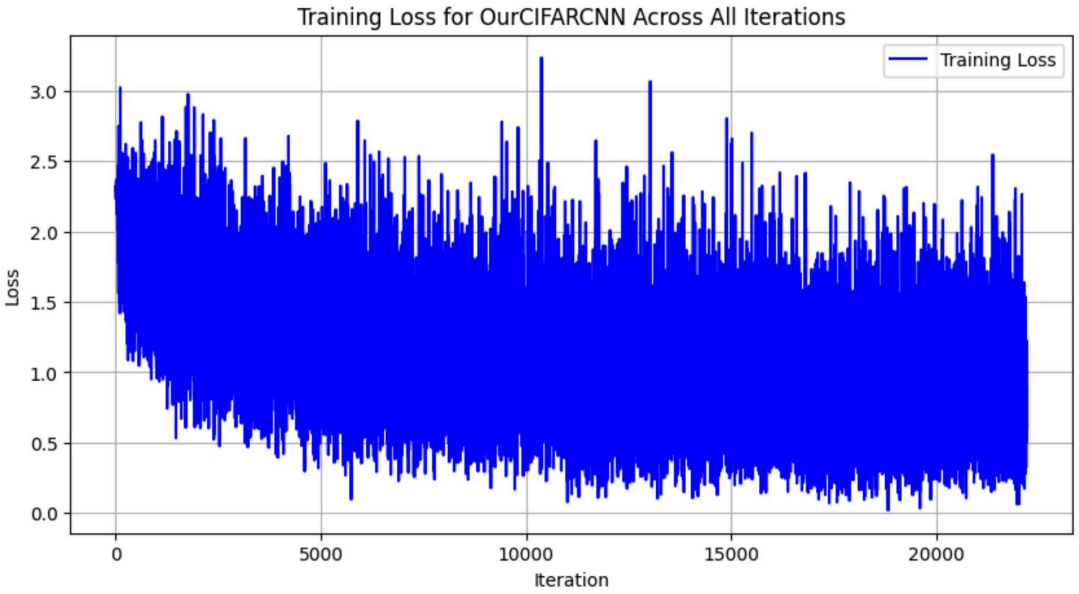
```
Epoch 1: [0/50000] Loss: 2.298
Epoch 1: [4995/50000] Loss: 1.601
Epoch 1: [9990/50000] Loss: 1.829
Epoch 1: [14985/50000] Loss: 1.241
Epoch 1: [19980/50000] Loss: 1.374
Epoch 1: [24975/50000] Loss: 1.321
Epoch 1: [29970/50000] Loss: 0.705
Epoch 1: [34965/50000] Loss: 1.259
Epoch 1: [39960/50000] Loss: 0.753
Epoch 1: [44955/50000] Loss: 1.458
Epoch 1: [49950/50000] Loss: 0.506
Epoch 2: [0/50000] Loss: 1.808
Epoch 2: [4995/50000] Loss: 1.681
Epoch 2: [9990/50000] Loss: 0.762
Epoch 2: [14985/50000] Loss: 1.067
Epoch 2: [19980/50000] Loss: 0.701
Epoch 2: [24975/50000] Loss: 0.737
Epoch 2: [29970/50000] Loss: 1.533
Epoch 2: [34965/50000] Loss: 1.489
Epoch 2: [39960/50000] Loss: 0.971
Epoch 2: [44955/50000] Loss: 1.012
Epoch 2: [49950/50000] Loss: 0.735
Epoch 3: [0/50000] Loss: 0.495
Epoch 3: [4995/50000] Loss: 1.000
Epoch 3: [9990/50000] Loss: 1.099
Epoch 3: [14985/50000] Loss: 1.071
Epoch 3: [19980/50000] Loss: 1.253
Epoch 3: [24975/50000] Loss: 1.036
Epoch 3: [29970/50000] Loss: 1.120
Epoch 3: [34965/50000] Loss: 0.618
Epoch 3: [39960/50000] Loss: 0.871
Epoch 3: [44955/50000] Loss: 0.708
Epoch 3: [49950/50000] Loss: 0.600
Epoch 4: [0/50000] Loss: 0.841
Epoch 4: [4995/50000] Loss: 1.845
Epoch 4: [9990/50000] Loss: 1.301
Epoch 4: [14985/50000] Loss: 0.237
Epoch 4: [19980/50000] Loss: 0.323
Epoch 4: [24975/50000] Loss: 0.418
Epoch 4: [29970/50000] Loss: 0.789
Epoch 4: [34965/50000] Loss: 0.950
Epoch 4: [39960/50000] Loss: 0.798
Epoch 4: [44955/50000] Loss: 0.717
Epoch 4: [49950/50000] Loss: 0.865
Finished Training after 228.8298945426941 s
Finished Training after 228.8307340145111 s
```

In [30]:

```python
# File paths for loading the model and losses
cnn_loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cifar_cnn_train_loss

# Load training losses
with open(cnn_loss_file, 'rb') as f:
    train_losses = pickle.load(f)

# Flatten the training losses across all epochs
flattened_losses = [loss for epoch_loss in train_losses for loss in epoch_loss]

# Calculate the total number of iterations (assuming batch size is known)
# You may need to replace `batch_size` with your actual batch size
total_iterations = len(flattened_losses)
```

Question assigned to the following page: 2.2

```
# Generate x-axis values (iterations)
x_values = range(1, total_iterations + 1)

# Plot the training losses
plt.figure(figsize=(10, 5))
plt.plot(x_values, flattened_losses, label='Training Loss', color='blue')
plt.title('Training Loss for OurCIFARCNN Across All Iterations')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.grid()
plt.legend()
plt.show()
```



In [35]:
```
# File paths for loading model and losses
model_cnn_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/cifar_cnn_model.pth
criterion = nn.CrossEntropyLoss()
# Initialize the model
model = OurCIFARCNN()
# Load the trained model for testing
model.load_state_dict(torch.load(model_cnn_file))
model.eval()  # Set the model to evaluation mode

# Testing the model
test_stat  = test(model, criterion, test_loader)

# Print the testing loss and accuracy
print(f'Test Loss: {test_stat["loss"]:.3f}')
print(f'Test Accuracy: {test_stat["accuracy"]*100:.2f}%')
```

Question assigned to the following page: 2.3

```
<ipython-input-35-893215ff9f85>:7: FutureWarning: You are using `torch.load` with `we
ights_only=False` (the current default value), which uses the default pickle module i
mplicitly. It is possible to construct malicious pickle data which will execute arbit
rary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURIT
Y.md#untrusted-models for more details). In a future release, the default value for `
weights_only` will be flipped to `True`. This limits the functions that could be exec
uted during unpickling. Arbitrary objects will no longer be allowed to be loaded via
this mode unless they are explicitly allowlisted by the user via `torch.serializatio
n.add_safe_globals`. We recommend you start setting `weights_only=True` for any use c
ase where you don't have full control of the loaded file. Please open an issue on Git
Hub for any issues related to this experimental feature.
  model.load_state_dict(torch.load(model_cnn_file))
Test Loss: 0.894
Test Accuracy: 69.99%
```

Task 3: Plot misclassified test images

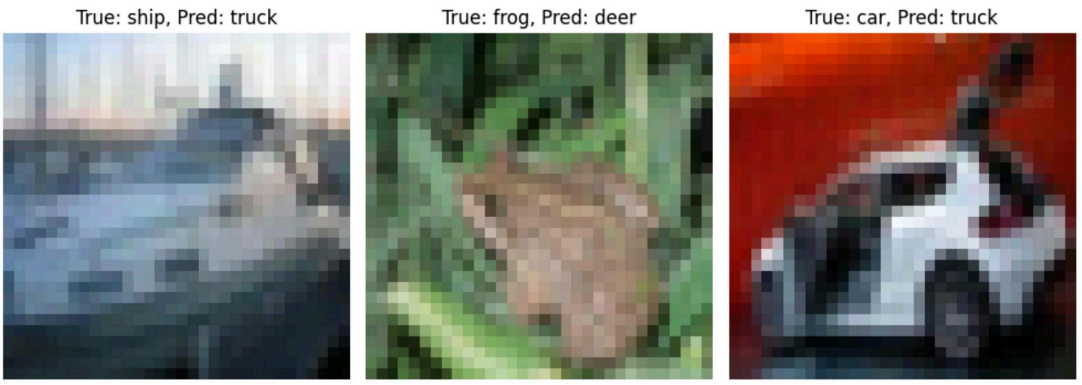
Plot some misclassified images in your test dataset:

- select three images that are **misclassified** by your neural network
- label each images with true label and predicted label
- use `detach().cpu()` when plotting images if the image is in gpu

In [20]:

```python
total_images = 3
predictions = test_stat['prediction']
targets = torch.tensor(test_dataset.targets)
# ----------- <Your code> ---------------
# Find misclassified images
# CIFAR-10 class names
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
misclassified = [(img, pred, true) for img, pred, true in zip(test_dataset.data, predi

# Select the first three misclassified images
misclassified_images = misclassified[:total_images]

# Create a plot for the misclassified images
fig, axes = plt.subplots(1, 3, figsize=(10, 5))
for i, (img, pred, true) in enumerate(misclassified_images):
    ax = axes[i]

    # If image is on GPU, detach it and move to CPU
    img = torch.tensor(img).detach().cpu().numpy() if isinstance(img, torch.Tensor) el

    # Plot image
    ax.imshow(img)
    ax.axis('off')

    # Set the title with true and predicted class names
    ax.set_title(f'True: {classes[true]}, Pred: {classes[pred]}')

# Display the plot
plt.tight_layout()
plt.show()
# ----------- <End Your code> ---------------
```

Question assigned to the following page: 2.3



Questions (0 points): Are the mis-classified images also misleading to human eyes?

Two of them (Ship and Frog) are clearly misleading to human eyes. Car is also misleading, but can be identified by careful attention

Exercise 3: Transfer Learning (30 points)

In practice, people won't train an entire CNN from scratch, because it is relatively rare to have a dataset of sufficient size (or sufficient computational power). Instead, it is common to pretrain a CNN on a very large dataset and then use the CNN either as an initialization or a fixed feature extractor for the task of interest.

In this task, you will learn how to use a pretrained CNN for CIFAR-10 classification.

Task1: Load pretrained model

`torchvision.models` (https://pytorch.org/vision/stable/models.html) contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection and video classification.

First, you should load the **pretrained** ResNet-18 that has already been trained on ImageNet using `torchvision.models`. If you are interested in more details about Resnet-18, read this paper https://arxiv.org/pdf/1512.03385.pdf.

In [77]:
```
resnet18 = models.resnet18(pretrained=True)
resnet18 = resnet18.to(device)
```

Question assigned to the following page: 3.1

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarnin
g: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the futu
re, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarnin
g: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.
13 and may be removed in the future. The current behavior is equivalent to passing `w
eights=ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet18_Weights.DE
FAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

Task2: Create data loaders for CIFAR-10

Then you need to create a modified dataset and dataloader for CIFAR-10. Importantly, the model you load has been trained on **ImageNet** and it expects inputs as mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be **at least** 224. So you need to preprocess the CIFAR-10 data to make sure it has a height and width of 224. Thus, you should add a transform when loading the CIFAR10 dataset (see `torchvision.transforms.Resize`). This should be added appropriately to the `transform` you created in a previous task.

In [78]:
```python
# Create your dataloader here
# ----------- <Your code> ---------------
# Set the device (GPU or CPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the transform to resize the images and normalize
transform = transforms.Compose([
    transforms.Resize((224, 224)),               # Resize CIFAR-10 images to 224x224
    transforms.ToTensor(),                       # Convert images to PyTorch tensors
    transforms.Normalize(                        # Normalize with mean and std for Ima
        mean=[0.485, 0.456, 0.406],              # ImageNet mean for 3 channels (RGB)
        std=[0.229, 0.224, 0.225]                # ImageNet std for 3 channels (RGB)
    )
])

# Load the CIFAR-10 dataset with the transform applied
train_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform
)
test_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform
)

# Create data loaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=9, shuffle=True,
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=9, shuffle=False, r
# ----------- <End Your code> ---------------
```

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

In [79]:
```python
# Modify the final layer to match the number of CIFAR-10 classes (10 classes)
num_ftrs = resnet18.fc.in_features
resnet18.fc = nn.Linear(num_ftrs, 10)
# Move the model to the appropriate device (GPU/CPU)
```

Question assigned to the following page: 3.1

```
resnet18 = resnet18.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(resnet18.parameters(), lr=0.001, momentum=0.9)
```

In [80]:

```
def train(model: nn.Module,
          loss_fn: nn.modules.loss._Loss,
          optimizer: torch.optim.Optimizer,
          train_loader: torch.utils.data.DataLoader,
          device: torch.device,
          epoch: int=0)-> List:
    # ----------- <Your code> ---------------
    model.train()  # Set the model to training mode
    train_loss = []  # To store loss for each batch

    # Loop over batches
    for batch_idx, (images, targets) in enumerate(train_loader):
        # Zero the gradients before forward pass
        optimizer.zero_grad()

        # Forward pass: compute predictions and loss
        outputs = model(images)
        loss = loss_fn(outputs, targets)

        # Backward pass: compute gradients
        loss.backward()

        # Update weights using optimizer
        optimizer.step()

        # Store loss for this batch
        train_loss.append(loss.item())

        # Print log 8-10 times per epoch
        if batch_idx % (len(train_loader) // 10) == 0:
            print(f'Epoch {epoch}: [{batch_idx*len(images)}/{len(train_loader.dataset)
    # ----------- <End Your code> ---------------
    assert len(train_loss) == len(train_loader)
    return train_loss

def test(model: nn.Module,
         loss_fn: nn.modules.loss._Loss,
         test_loader: torch.utils.data.DataLoader,
         device: torch.device,
         epoch: int=0) -> Dict:
    model.eval()  # Set the model to evaluation mode
    test_loss = 0.0
    correct = 0
    predictions = []

    # Disable gradient computation for testing
    with torch.no_grad():
        for images, targets in test_loader:
            # Move images and targets to the appropriate device
            images, targets = images.to(device), targets.to(device)

            # Forward pass: compute predictions
            outputs = model(images)
```

Questions assigned to the following page: 3.1, 3.2, and 3.3

```
            # Compute loss
            loss = loss_fn(outputs, targets)
            test_loss += loss.item()  # Accumulate the total loss

            # Get predictions: assuming classification task
            pred = outputs.argmax(dim=1, keepdim=True)  # Get index of the max log-pro
            predictions.append(pred.flatten())

            # Count correct predictions
            correct += pred.eq(targets.view_as(pred)).sum().item()

    # Convert predictions list to a 1D tensor
    predictions = torch.cat(predictions)

    # Average loss over all test data
    test_loss /= len(test_loader)

    # Accuracy calculation
    accuracy = correct / len(test_loader.dataset)

    # Create dictionary with test statistics
    test_stat = {
        "loss": test_loss,
        "accuracy": accuracy,
        "prediction": predictions
    }

    # Assertions for test statistics
    assert "loss" in test_stat.keys()
    assert "accuracy" in test_stat.keys()
    assert "prediction" in test_stat.keys()
    assert len(test_stat["prediction"]) == len(test_loader.dataset)
    assert isinstance(test_stat["prediction"], torch.Tensor)

    return test_stat
```

Task3: Classify test data on pretrained model

Use the model you load to classify the **test** CIFAR-10 data and print out the test accuracy. Don't be surprised if the accuracy is bad!

In [81]:
```
# ----------- <Your code> ---------------
# Set the model to evaluation mode
resnet18.eval()
# Test the model on CIFAR-10 test data
test_stat = test(resnet18, criterion, test_loader,device)
# ----------- <End Your code> ---------------
```

In [82]:
```
# Print the testing loss and accuracy
print(f'Test Loss: {test_stat["loss"]:.3f}')
print(f'Test Accuracy: {test_stat["accuracy"]*100:.2f}%')
```

```
Test Loss: 2.520
Test Accuracy: 7.63%
```

Task 4: Fine-tune (i.e., update) the pretrained model for CIFAR-10

Question assigned to the following page: 3.3

Now try to improve the test accuracy. We offer several possible solutions:

(1) You can try to directly continue to train the model you load with the CIFAR-10 training data.

(2) For efficiency, you can try to freeze part of the parameters of the loaded models. For example, you can first freeze all parameters by

```
for param in model.parameters():
    param.requires_grad = False
```

and then unfreeze the last few layers by setting `somelayer.requires_grad=True`.

You are also welcome to try any other approach you can think of.

Note: You must print out the test accuracy and to get full credits, the test accuracy should be at least **80%**.

In [83]:
```
# unfreeze the outer layers.
start = time.time()
#----------- <Your code> ---------------
for param in resnet18.parameters():
    param.requires_grad = False  # Freeze all pretrained layers

# Unfreeze the final layer (the last fully connected layer)
for param in resnet18.fc.parameters():
    param.requires_grad = True  # Only train the last layer

# Move the model to the appropriate device (GPU/CPU)
resnet18 = resnet18.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(resnet18.fc.parameters(), lr=0.001, momentum=0.9)  # Only optimi

# File paths for saving model and losses
model_resnet_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/resnet18_cifar.p
resnet_loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/resnet18_train_lo

# Initialize training losses list
if os.path.exists(model_resnet_file) and os.path.exists(resnet_loss_file):
    resnet18.load_state_dict(torch.load(model_resnet_file))  # Load model weights
    with open(resnet_loss_file, 'rb') as f:
        resnet_train_losses = pickle.load(f)  # Load training losses
else:
    resnet_train_losses = []  # Initialize list if not present

# Training parameters
max_epoch = 4  # Number of epochs

# Training loop
start = time.time()

for epoch in range(1, max_epoch + 1):
    # Training the last layer using the train function
    epoch_loss = train_transfer_load(resnet18, criterion, optimizer, train_loader,devi
```

Question assigned to the following page: 3.3

```
    resnet_train_losses.append(epoch_loss)  # Append the loss for this epoch

    # Save the model after each epoch
    torch.save(resnet18.state_dict(), model_resnet_file)

# Save the training losses to a file
with open(resnet_loss_file, 'wb') as f:
    pickle.dump(resnet_train_losses, f)

end = time.time()
print(f'Finished Training after {end - start} s')
```

```
Epoch 1: [0/50000] Loss: 2.561
Epoch 1: [4995/50000] Loss: 1.158
Epoch 1: [9990/50000] Loss: 1.030
Epoch 1: [14985/50000] Loss: 1.050
Epoch 1: [19980/50000] Loss: 0.684
Epoch 1: [24975/50000] Loss: 0.417
Epoch 1: [29970/50000] Loss: 0.509
Epoch 1: [34965/50000] Loss: 0.671
Epoch 1: [39960/50000] Loss: 0.740
Epoch 1: [44955/50000] Loss: 0.381
Epoch 1: [49950/50000] Loss: 0.621
Epoch 2: [0/50000] Loss: 0.945
Epoch 2: [4995/50000] Loss: 0.769
Epoch 2: [9990/50000] Loss: 0.752
Epoch 2: [14985/50000] Loss: 0.665
Epoch 2: [19980/50000] Loss: 0.802
Epoch 2: [24975/50000] Loss: 1.325
Epoch 2: [29970/50000] Loss: 0.511
Epoch 2: [34965/50000] Loss: 1.191
Epoch 2: [39960/50000] Loss: 1.157
Epoch 2: [44955/50000] Loss: 0.387
Epoch 2: [49950/50000] Loss: 0.809
Epoch 3: [0/50000] Loss: 0.343
Epoch 3: [4995/50000] Loss: 1.019
Epoch 3: [9990/50000] Loss: 0.869
Epoch 3: [14985/50000] Loss: 0.250
Epoch 3: [19980/50000] Loss: 0.557
Epoch 3: [24975/50000] Loss: 0.456
Epoch 3: [29970/50000] Loss: 0.587
Epoch 3: [34965/50000] Loss: 0.865
Epoch 3: [39960/50000] Loss: 0.172
Epoch 3: [44955/50000] Loss: 0.487
Epoch 3: [49950/50000] Loss: 0.312
Epoch 4: [0/50000] Loss: 0.237
Epoch 4: [4995/50000] Loss: 0.696
Epoch 4: [9990/50000] Loss: 1.005
Epoch 4: [14985/50000] Loss: 0.494
Epoch 4: [19980/50000] Loss: 0.412
Epoch 4: [24975/50000] Loss: 0.662
Epoch 4: [29970/50000] Loss: 0.700
Epoch 4: [34965/50000] Loss: 0.644
Epoch 4: [39960/50000] Loss: 0.427
Epoch 4: [44955/50000] Loss: 1.558
Epoch 4: [49950/50000] Loss: 0.776
Finished Training after 444.06879138946533 s
```

In [84]:

```
# ----------- <Your code> ---------------
# Set the model to evaluation mode
```

Question assigned to the following page: 3.3

```
resnet18.eval()
# Test the model on CIFAR-10 test data
test_stat = test(resnet18, criterion, test_loader,device)
# ----------- <End Your code> ---------------
```

In [86]:
```
# Print the testing loss and accuracy
print(f'Test Loss: {test_stat["loss"]:.3f}')
print(f'Test Accuracy: {test_stat["accuracy"]*100:.2f}%')
```

```
Test Loss: 0.652
Test Accuracy: 81.02%
```

In [87]:
```
#load another instance, train full model
resnet18_full_train = models.resnet18(pretrained=True)
um_ftrs = resnet18_full_train.fc.in_features
resnet18_full_train.fc = nn.Linear(num_ftrs, 10)
# Move the model to the appropriate device (GPU/CPU)
# Modify the final layer to match the number of CIFAR-10 classes (10 classes)
num_ftrs = resnet18_full_train.fc.in_features
resnet18_full_train.fc = nn.Linear(num_ftrs, 10)
# Move the model to the appropriate device (GPU/CPU)
resnet18_full_train = resnet18_full_train.to(device)

start = time.time()
#----------- <Your code> ---------------
for param in resnet18_full_train.parameters():
    param.requires_grad = True  # Freeze all pretrained layers

# Unfreeze the final layer (the last fully connected layer)
for param in resnet18_full_train.fc.parameters():
    param.requires_grad = True  # Only train the last layer

# Move the model to the appropriate device (GPU/CPU)
resnet18_full_train = resnet18_full_train.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(resnet18_full_train.fc.parameters(), lr=0.001, momentum=0.9)  #

# File paths for saving model and losses
model_resnet_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/resnet18_full_tr
resnet_loss_file = '/content/drive/MyDrive/ECE57000_AI/Assignment_03/resnet18_full_tra

# Initialize training losses list
if os.path.exists(model_resnet_file) and os.path.exists(resnet_loss_file):
    resnet18_full_train.load_state_dict(torch.load(model_resnet_file))  # Load model w
    with open(resnet_loss_file, 'rb') as f:
        resnet_train_losses = pickle.load(f)  # Load training losses
else:
    resnet_train_losses = []  # Initialize list if not present

# Training parameters
max_epoch = 4  # Number of epochs

# Training loop
start = time.time()

for epoch in range(1, max_epoch + 1):
    # Training the full network using the train function
    epoch_loss = train(resnet18_full_train, criterion, optimizer, train_loader,device,
```

Question assigned to the following page: 3.3

```
    resnet_train_losses.append(epoch_loss)  # Append the loss for this epoch

    # Save the model after each epoch
    torch.save(resnet18_full_train.state_dict(), model_resnet_file)

# Save the training losses to a file
with open(resnet_loss_file, 'wb') as f:
    pickle.dump(resnet_train_losses, f)

end = time.time()
print(f'Finished Training after {end - start} s')
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarnin
g: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the futu
re, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarnin
g: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.
13 and may be removed in the future. The current behavior is equivalent to passing `w
eights=ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet18_Weights.DE
FAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

Question assigned to the following page: 3.3

```
Epoch 1: [0/50000] Loss: 2.846
Epoch 1: [4995/50000] Loss: 0.661
Epoch 1: [9990/50000] Loss: 1.412
Epoch 1: [14985/50000] Loss: 1.474
Epoch 1: [19980/50000] Loss: 0.871
Epoch 1: [24975/50000] Loss: 1.003
Epoch 1: [29970/50000] Loss: 0.273
Epoch 1: [34965/50000] Loss: 1.890
Epoch 1: [39960/50000] Loss: 0.956
Epoch 1: [44955/50000] Loss: 0.856
Epoch 1: [49950/50000] Loss: 1.031
Epoch 2: [0/50000] Loss: 0.817
Epoch 2: [4995/50000] Loss: 0.691
Epoch 2: [9990/50000] Loss: 0.424
Epoch 2: [14985/50000] Loss: 1.539
Epoch 2: [19980/50000] Loss: 0.445
Epoch 2: [24975/50000] Loss: 1.147
Epoch 2: [29970/50000] Loss: 0.650
Epoch 2: [34965/50000] Loss: 1.158
Epoch 2: [39960/50000] Loss: 0.999
Epoch 2: [44955/50000] Loss: 0.798
Epoch 2: [49950/50000] Loss: 0.375
Epoch 3: [0/50000] Loss: 0.880
Epoch 3: [4995/50000] Loss: 0.215
Epoch 3: [9990/50000] Loss: 0.651
Epoch 3: [14985/50000] Loss: 0.136
Epoch 3: [19980/50000] Loss: 1.523
Epoch 3: [24975/50000] Loss: 0.425
Epoch 3: [29970/50000] Loss: 1.304
Epoch 3: [34965/50000] Loss: 0.995
Epoch 3: [39960/50000] Loss: 0.308
Epoch 3: [44955/50000] Loss: 0.626
Epoch 3: [49950/50000] Loss: 0.890
Epoch 4: [0/50000] Loss: 1.107
Epoch 4: [4995/50000] Loss: 1.127
Epoch 4: [9990/50000] Loss: 1.762
Epoch 4: [14985/50000] Loss: 0.810
Epoch 4: [19980/50000] Loss: 0.638
Epoch 4: [24975/50000] Loss: 0.302
Epoch 4: [29970/50000] Loss: 0.858
Epoch 4: [34965/50000] Loss: 0.340
Epoch 4: [39960/50000] Loss: 0.475
Epoch 4: [44955/50000] Loss: 1.001
Epoch 4: [49950/50000] Loss: 0.720
Finished Training after 818.0664708614349 s
```

In [89]:

```python
# ----------- <Your code> ---------------
# Set the model to evaluation mode
resnet18.eval()
# Test the model on CIFAR-10 test data
test_stat = test(resnet18_full_train, criterion, test_loader, device)
# Print the testing loss and accuracy
print(f'Test Loss: {test_stat["loss"]:.3f}')
print(f'Test Accuracy: {test_stat["accuracy"]*100:.2f}%')
# ----------- <End Your code> ---------------
```

```
Test Loss: 0.642
Test Accuracy: 81.02%
```