ECE 57000 Assignment 3 Exercise

Your Name: Cole Richardson

Prepare the pacakge we will use.

In [29]:
```
import time
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
```

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 [30]:
```
print(f'Can I can use CUDA now? -- {torch.cuda.is_available()}')
print(f'Can I use Metal Performance Shaders now? -- {torch.backends.mps.is_available(
```

```
Can I can use CUDA now? -- False
Can I use Metal Performance Shaders 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 [31]:
```
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.type, param.size()] for param in simp
print(f'output is on {simple_model(rand_tensor).device}')

# Use CUDA or Metal Performance Shaders (mps) if available for hardware acceleration
device = torch.device('cpu')
if torch.cuda.is_available():
    device = torch.device('cuda')
elif torch.backends.mps.is_available():
    device = torch.device('mps')

# ----------- <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.type, param.size()] for param in simp
print(f'output is on {simple_model(rand_tensor).device}')
```

```
input is on cpu
model parameters are on [['cpu', torch.Size([10, 2])], ['cpu', torch.Size([10])], ['cp
u', torch.Size([1, 10])], ['cpu', torch.Size([1])]]
output is on cpu
input is on mps:0
model parameters are on [['mps', torch.Size([10, 2])], ['mps', torch.Size([10])], ['mp
s', torch.Size([1, 10])], ['mps', torch.Size([1])]]
output is on mps:0
```

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

- 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 [32]:
```
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> ---------------
    train_loss = []

    # Set model to training mode
    model.train()

    # Loop over the data in batches for a single epoch
    for batch_idx, (images, targets) in enumerate(train_loader):
        # Zero the gradient
        optimizer.zero_grad()
        # move the data to the device
        images, targets = images.to(device), targets.to(device)
        # Model forward evaluation
        output = model(images)
        # Calculate loss
        loss = loss_fn(output, targets)
        # Gradient of loss function
        loss.backward()
        # Gradient evaluation and backward propagation
```

```
        optimizer.step()

        train_loss.append(loss.item()) # item() is to get the value of the tensor dir
        if batch_idx % 100 == 0: # We log our output every 100 batches
            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> ---------------
    # Set the model to evaluation mode (i.e. not training)
    model.eval()

    test_loss = 0
    predictions = []
    correct = 0

    with torch.no_grad():
        for images, targets in test_loader:
            # Move the data to the device
            images, targets = images.to(device), targets.to(device)
            # Evaluate the model on the batch
            output = model(images)
            test_loss += loss_fn(output, targets).item() * images.size(0) # sum up ba
            pred = output.data.max(1, keepdim=True)[1] # we get the estimate of our r
            correct += pred.eq(targets.data.view_as(pred)).sum() # sum up the correct
            predictions.append(pred)

    # Number of total test samples
    total_num = len(test_loader.dataset)
    # Create output dictionary
    test_stat = {}
    # Average loss over epoch
    test_stat['loss'] = test_loss / total_num
    # Average accuracy
    test_stat['accuracy'] = correct / total_num
    # Predictions
    test_stat['prediction'] = torch.cat(predictions)

    print(f"Test result on epoch {epoch}: total samples: {total_num}, Avg loss: {test
    # ----------- <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**
- 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 [33]:
```
# 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, transfo
test_dataset = torchvision.datasets.MNIST('data', train=False, download=True, transfo

print(train_dataset)
# ----------- <End Your code> -------------
```

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

In [34]:
```
# Build OurFC class and OurCNN class.
# ----------- <Your code> ---------------
import torch.nn as nn
import torch.nn.functional as F

class OurFC(nn.Module):

  def __init__(self):
    super().__init__()

    # 1x28x28 = 784
    self.fc1 = nn.Linear(784, 1024)
    self.fc2 = nn.Linear(1024, 128)
    self.fc3 = nn.Linear(128, 10)

  def forward(self, x):
    x = x.view(-1, 784)         # flatten the input (batchsize x 784)
    x = F.relu(self.fc1(x))     # x now has shape (batchsize x 300)
```

```
    x = F.relu(self.fc2(x))      # x has shape (batchsize x 100)
    x = self.fc3(x)              # x has shape (batchsize x 10)
    return F.log_softmax(x,-1)


class OurCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # Input size = batchsize x 1 x 28 x 28
        self.conv1 = nn.Conv2d(1, 10, 5, 1, padding=2) # output size = batchsize x 10
        self.pool1 = nn.MaxPool2d(2, 1) # output size = batchsize x 10 x 27 x 27
        self.conv2 = nn.Conv2d(10, 20, 5, 1) # output size = batchsize x 20 x 23 x 23
        self.pool2 = nn.MaxPool2d(2, 2) # output size = batchsize x 20 x 11 x 11
        # 20 * 11 * 11 = 2420
        self.fc1 = nn.Linear(2420, 128)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        output = F.log_softmax(x, dim=1)
        return output

# ----------- <End Your code> -------------
```

In [35]:

```
# Let's first train the FC model. Below are there common hyperparameters.
criterion = nn.CrossEntropyLoss()

start = time.time()
max_epoch = 3
# ----------- <Your code> ---------------

# Create the data loaders for training and testing
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, s

classifier = OurFC()
optimizer = optim.SGD(classifier.parameters(), lr=0.01, momentum=0.8)

# Move the model to the device
classifier.to(device)

for epoch in range(1, max_epoch+1):
  train(classifier, criterion, optimizer, train_loader, epoch)
  test(classifier, criterion, test_loader, epoch)

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

```
Epoch 1: [0/60000] Loss: 2.3036627769470215
Epoch 1: [6400/60000] Loss: 0.507659375667572
Epoch 1: [12800/60000] Loss: 0.36319172382354736
Epoch 1: [19200/60000] Loss: 0.49460726976394653
Epoch 1: [25600/60000] Loss: 0.21799762547016144
Epoch 1: [32000/60000] Loss: 0.24595129489898682
Epoch 1: [38400/60000] Loss: 0.25214171409606934
Epoch 1: [44800/60000] Loss: 0.23277173936367035
Epoch 1: [51200/60000] Loss: 0.08823312073945999
Epoch 1: [57600/60000] Loss: 0.11263954639434814
Test result on epoch 1: total samples: 10000, Avg loss: 0.164, Acc: 95.340%
Epoch 2: [0/60000] Loss: 0.09985765814781189
Epoch 2: [6400/60000] Loss: 0.17845597863197327
Epoch 2: [12800/60000] Loss: 0.2772822678089142
Epoch 2: [19200/60000] Loss: 0.33599182963371277
Epoch 2: [25600/60000] Loss: 0.0626198872923851
Epoch 2: [32000/60000] Loss: 0.18426555395126343
Epoch 2: [38400/60000] Loss: 0.12364546209573746
Epoch 2: [44800/60000] Loss: 0.15872865915298462
Epoch 2: [51200/60000] Loss: 0.10262952744960785
Epoch 2: [57600/60000] Loss: 0.1131037026643753
Test result on epoch 2: total samples: 10000, Avg loss: 0.108, Acc: 96.740%
Epoch 3: [0/60000] Loss: 0.1368207335472107
Epoch 3: [6400/60000] Loss: 0.17074428498744965
Epoch 3: [12800/60000] Loss: 0.1047675758600235
Epoch 3: [19200/60000] Loss: 0.15148790180683136
Epoch 3: [25600/60000] Loss: 0.17269252240657806
Epoch 3: [32000/60000] Loss: 0.11413596570491791
Epoch 3: [38400/60000] Loss: 0.08393494039773941
Epoch 3: [44800/60000] Loss: 0.11729604005813599
Epoch 3: [51200/60000] Loss: 0.04852864146232605
Epoch 3: [57600/60000] Loss: 0.12906178832054138
Test result on epoch 3: total samples: 10000, Avg loss: 0.083, Acc: 97.410%
Finished Training after 18.9511981010437 s
```

In [36]:

```python
# Let's then train the OurCNN model.
start = time.time()
# ----------- <Your code> ---------------
model = OurCNN()
# Note: SGD doesn't work well here
optimizer = optim.Adadelta(model.parameters(), lr=1.0)

# Move the model to the device
model.to(device)

for epoch in range(1, max_epoch+1):
  train(model, criterion, optimizer, train_loader, epoch)
  test(model, criterion, test_loader, epoch)
# ----------- <End Your code> ---------------
end = time.time()
print(f'Finished Training after {end-start} s ')
```

```
Epoch 1: [0/60000] Loss: 4.821977615356445
Epoch 1: [6400/60000] Loss: 0.2862577438354492
Epoch 1: [12800/60000] Loss: 0.3254799544811249
Epoch 1: [19200/60000] Loss: 0.1868068277835846
Epoch 1: [25600/60000] Loss: 0.10389021039009094
Epoch 1: [32000/60000] Loss: 0.09858927130699158
Epoch 1: [38400/60000] Loss: 0.01736041158437729
Epoch 1: [44800/60000] Loss: 0.08421144634485245
Epoch 1: [51200/60000] Loss: 0.018595324829220772
Epoch 1: [57600/60000] Loss: 0.021651240065693855
Test result on epoch 1: total samples: 10000, Avg loss: 0.039, Acc: 98.710%
Epoch 2: [0/60000] Loss: 0.011565379798412323
Epoch 2: [6400/60000] Loss: 0.008629001677036285
Epoch 2: [12800/60000] Loss: 0.05675356835126877
Epoch 2: [19200/60000] Loss: 0.028941763564944267
Epoch 2: [25600/60000] Loss: 0.03557306528091431
Epoch 2: [32000/60000] Loss: 0.03628135100007057
Epoch 2: [38400/60000] Loss: 0.12786263227462769
Epoch 2: [44800/60000] Loss: 0.0820731595158577
Epoch 2: [51200/60000] Loss: 0.04045715928077698
Epoch 2: [57600/60000] Loss: 0.023317497223615646
Test result on epoch 2: total samples: 10000, Avg loss: 0.039, Acc: 98.820%
Epoch 3: [0/60000] Loss: 0.0066825347021222115
Epoch 3: [6400/60000] Loss: 0.05618815869092941
Epoch 3: [12800/60000] Loss: 0.014067328535020351
Epoch 3: [19200/60000] Loss: 0.012478461489081383
Epoch 3: [25600/60000] Loss: 0.03128328546881676
Epoch 3: [32000/60000] Loss: 0.08801764994859695
Epoch 3: [38400/60000] Loss: 0.03331048786640167
Epoch 3: [44800/60000] Loss: 0.07113268971443176
Epoch 3: [51200/60000] Loss: 0.05036716163158417
Epoch 3: [57600/60000] Loss: 0.004219552502036095
Test result on epoch 3: total samples: 10000, Avg loss: 0.033, Acc: 98.960%
Finished Training after 36.4839141368866 s
```

In [37]:
```python
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 936330 parameters
OurCNN has a total of 315168 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?

1. The fully connected neural network has ~3x the number of parameters
2. The CNN will likely be more computationally efficient simply because it has much fewer parameters
3. The CNN took ~2x the amount of time to trains

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.

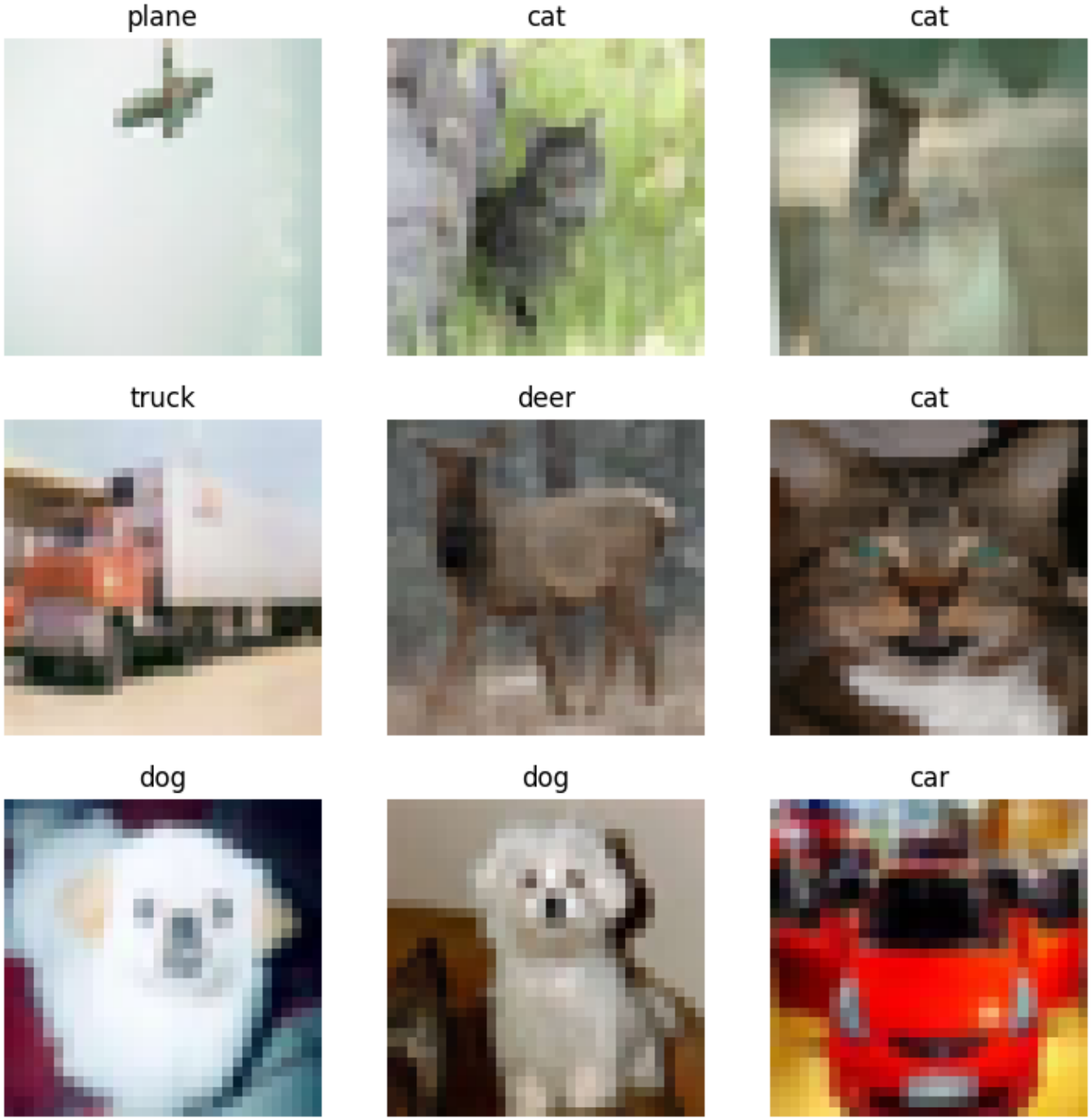
Task 1: Create data loaders

- 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)

In [38]:
```
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tr

# Create the appropriate transform, load/download CIFAR10 train and test datasets wit
# ----------- <Your code> ---------------
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                            torchvision.transforms.Normalize((0.5, 0.
trainset = torchvision.datasets.CIFAR10('data', train=True, download=True, transform=
testset = torchvision.datasets.CIFAR10('data', train=False, download=True, transform=
# ----------- <End Your code> ---------------

# Define trainloader and testloader
# ----------- <Your code> ---------------
train_loader = torch.utils.data.DataLoader(trainset, batch_size=9, shuffle=True, num_
test_loader = torch.utils.data.DataLoader(testset, batch_size=9, shuffle=False, num_w
# ----------- <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
```

```
/var/folders/xq/rx8kt7mx02xfd87r07v219bm0000gn/T/ipykernel_6825/1670839852.py:27: User
Warning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



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

In [39]:
```
# Create CNN network.
# ----------- <Your code> ---------------
class CfarCnn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 12, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(12, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```
    def forward(self, x):
        # input size = batchsize x 3 x 32 x 32
        x = self.pool(F.relu(self.conv1(x)))    # output size = batchsize x 6 x 14 x
        x = self.pool(F.relu(self.conv2(x)))    # output size = batchsize x 16 x 5 x
        x = torch.flatten(x, 1)                 # output size = batchsize x 400
        x = F.relu(self.fc1(x))                 # output size = batchsize x 120
        x = F.relu(self.fc2(x))                 # output size = batchsize x 84
        x = self.fc3(x)                         # output size = batchsize x 10
        return x
# ----------- <End Your code> ---------------
```

In [40]:

```
# Train your neural network here.
start = time.time()
max_epoch = 4
# ----------- <Your code> ---------------
# Define net
net = CfarCnn()
# Define loss function
criterion = nn.CrossEntropyLoss()
# Define optimizer
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# Move the model to the device
net.to(device)
# Training loop
for epoch in range(1, max_epoch+1):
  train(net, criterion, optimizer, train_loader, epoch)
# ----------- <End Your code> ---------------
output = test(net, criterion, test_loader, epoch)
end = time.time()
print(f'Finished Training after {end-start} s ')
```

```
Epoch 1: [0/50000] Loss: 2.321708917617798
Epoch 1: [900/50000] Loss: 2.299635648727417
Epoch 1: [1800/50000] Loss: 2.2978734970092773
Epoch 1: [2700/50000] Loss: 2.290391683578491
Epoch 1: [3600/50000] Loss: 2.298146963119507
Epoch 1: [4500/50000] Loss: 2.295351982116699
Epoch 1: [5400/50000] Loss: 2.296144723892212
Epoch 1: [6300/50000] Loss: 2.28908634185791
Epoch 1: [7200/50000] Loss: 2.2900102138519287
Epoch 1: [8100/50000] Loss: 2.2798256874084473
Epoch 1: [9000/50000] Loss: 2.258164167404175
Epoch 1: [9900/50000] Loss: 2.1937971115112305
Epoch 1: [10800/50000] Loss: 2.2859363555908203
Epoch 1: [11700/50000] Loss: 2.2640109062194824
Epoch 1: [12600/50000] Loss: 2.221187114715576
Epoch 1: [13500/50000] Loss: 2.2097768783569336
Epoch 1: [14400/50000] Loss: 1.8516954183578491
Epoch 1: [15300/50000] Loss: 1.589654803276062
Epoch 1: [16200/50000] Loss: 2.2060043811798096
Epoch 1: [17100/50000] Loss: 1.932188630104065
Epoch 1: [18000/50000] Loss: 1.7408777475357056
Epoch 1: [18900/50000] Loss: 1.8206384181976318
Epoch 1: [19800/50000] Loss: 1.9106794595718384
Epoch 1: [20700/50000] Loss: 1.8558152914047241
Epoch 1: [21600/50000] Loss: 1.4823869466781616
Epoch 1: [22500/50000] Loss: 2.1817240715026855
Epoch 1: [23400/50000] Loss: 2.3003766536712646
Epoch 1: [24300/50000] Loss: 2.025707721710205
Epoch 1: [25200/50000] Loss: 1.513455867767334
Epoch 1: [26100/50000] Loss: 1.4090296030044556
Epoch 1: [27000/50000] Loss: 1.6202081441879272
Epoch 1: [27900/50000] Loss: 1.9843851327896118
Epoch 1: [28800/50000] Loss: 2.168372392654419
Epoch 1: [29700/50000] Loss: 1.6755865812301636
Epoch 1: [30600/50000] Loss: 1.7604559659957886
Epoch 1: [31500/50000] Loss: 1.9732818603515625
Epoch 1: [32400/50000] Loss: 1.700818419456482
Epoch 1: [33300/50000] Loss: 1.5055220127105713
Epoch 1: [34200/50000] Loss: 1.38328218460083
Epoch 1: [35100/50000] Loss: 1.9576326608657837
Epoch 1: [36000/50000] Loss: 1.7075934410095215
Epoch 1: [36900/50000] Loss: 1.516133189201355
Epoch 1: [37800/50000] Loss: 1.9411119222640991
Epoch 1: [38700/50000] Loss: 1.2252225875854492
Epoch 1: [39600/50000] Loss: 1.7805311679840088
Epoch 1: [40500/50000] Loss: 1.6912747621536255
Epoch 1: [41400/50000] Loss: 1.7717127799987793
Epoch 1: [42300/50000] Loss: 1.1294394731521606
Epoch 1: [43200/50000] Loss: 1.7092843055725098
Epoch 1: [44100/50000] Loss: 1.9570494890213013
Epoch 1: [45000/50000] Loss: 1.6713416576385498
Epoch 1: [45900/50000] Loss: 1.5023714303970337
Epoch 1: [46800/50000] Loss: 1.5392595529556274
Epoch 1: [47700/50000] Loss: 1.3173969984054565
Epoch 1: [48600/50000] Loss: 1.1450313329696655
Epoch 1: [49500/50000] Loss: 2.0282881259918213
Epoch 2: [0/50000] Loss: 1.8778436183929443
Epoch 2: [900/50000] Loss: 1.6356370449066162
Epoch 2: [1800/50000] Loss: 1.5758452415466309
Epoch 2: [2700/50000] Loss: 1.347749948501587
Epoch 2: [3600/50000] Loss: 0.9537296295166016
Epoch 2: [4500/50000] Loss: 1.6455636024475098
Epoch 2: [5400/50000] Loss: 1.3506660461425781
Epoch 2: [6300/50000] Loss: 1.8846770524978638
Epoch 2: [7200/50000] Loss: 1.649356484413147
```

```
Epoch 2: [8100/50000] Loss: 1.0162817239761353
Epoch 2: [9000/50000] Loss: 1.1660022735595703
Epoch 2: [9900/50000] Loss: 1.8109067678451538
Epoch 2: [10800/50000] Loss: 1.7444498538970947
Epoch 2: [11700/50000] Loss: 2.026499032974243
Epoch 2: [12600/50000] Loss: 1.5067088603973389
Epoch 2: [13500/50000] Loss: 1.495421290397644
Epoch 2: [14400/50000] Loss: 1.5104856491088867
Epoch 2: [15300/50000] Loss: 1.243625521659851
Epoch 2: [16200/50000] Loss: 1.0046625137329102
Epoch 2: [17100/50000] Loss: 1.5360863208770752
Epoch 2: [18000/50000] Loss: 1.2957812547683716
Epoch 2: [18900/50000] Loss: 0.9973399043083191
Epoch 2: [19800/50000] Loss: 1.5260810852050781
Epoch 2: [20700/50000] Loss: 1.508498191833496
Epoch 2: [21600/50000] Loss: 1.8290362358093262
Epoch 2: [22500/50000] Loss: 1.1669437885284424
Epoch 2: [23400/50000] Loss: 1.3006129264831543
Epoch 2: [24300/50000] Loss: 0.9979870915412903
Epoch 2: [25200/50000] Loss: 1.0801255702972412
Epoch 2: [26100/50000] Loss: 1.2413626909255981
Epoch 2: [27000/50000] Loss: 1.7255421876907349
Epoch 2: [27900/50000] Loss: 1.142411231994629
Epoch 2: [28800/50000] Loss: 1.4030568599700928
Epoch 2: [29700/50000] Loss: 1.081099271774292
Epoch 2: [30600/50000] Loss: 0.8693921566009521
Epoch 2: [31500/50000] Loss: 1.2664532661437988
Epoch 2: [32400/50000] Loss: 1.612056016921997
Epoch 2: [33300/50000] Loss: 1.4307501316070557
Epoch 2: [34200/50000] Loss: 1.8679612874984741
Epoch 2: [35100/50000] Loss: 0.9278923869132996
Epoch 2: [36000/50000] Loss: 1.0833483934402466
Epoch 2: [36900/50000] Loss: 1.4514213800430298
Epoch 2: [37800/50000] Loss: 1.9627469778060913
Epoch 2: [38700/50000] Loss: 2.0789670944213867
Epoch 2: [39600/50000] Loss: 1.2360973358154297
Epoch 2: [40500/50000] Loss: 1.0814067125320435
Epoch 2: [41400/50000] Loss: 1.2454876899719238
Epoch 2: [42300/50000] Loss: 1.3109993934631348
Epoch 2: [43200/50000] Loss: 1.7124848365783691
Epoch 2: [44100/50000] Loss: 0.8191314339637756
Epoch 2: [45000/50000] Loss: 1.2850565910339355
Epoch 2: [45900/50000] Loss: 1.1655189990997314
Epoch 2: [46800/50000] Loss: 1.6276681423187256
Epoch 2: [47700/50000] Loss: 1.8680506944656372
Epoch 2: [48600/50000] Loss: 1.571911096572876
Epoch 2: [49500/50000] Loss: 1.3375544548034668
Epoch 3: [0/50000] Loss: 0.5831671357154846
Epoch 3: [900/50000] Loss: 1.1166114807128906
Epoch 3: [1800/50000] Loss: 1.0220412015914917
Epoch 3: [2700/50000] Loss: 1.6020028591156006
Epoch 3: [3600/50000] Loss: 1.463404655456543
Epoch 3: [4500/50000] Loss: 1.4514434337615967
Epoch 3: [5400/50000] Loss: 1.4257850646972656
Epoch 3: [6300/50000] Loss: 0.5743331909179688
Epoch 3: [7200/50000] Loss: 1.3003188371658325
Epoch 3: [8100/50000] Loss: 1.1367636919021606
Epoch 3: [9000/50000] Loss: 1.3123995065689087
Epoch 3: [9900/50000] Loss: 1.2242037057876587
Epoch 3: [10800/50000] Loss: 1.4720933437347412
Epoch 3: [11700/50000] Loss: 1.397192120552063
Epoch 3: [12600/50000] Loss: 1.597601056098938
Epoch 3: [13500/50000] Loss: 1.6822606325149536
Epoch 3: [14400/50000] Loss: 1.300584316253662
Epoch 3: [15300/50000] Loss: 1.7020220756530762
```

```
Epoch 3: [16200/50000] Loss: 1.2736577987670898
Epoch 3: [17100/50000] Loss: 1.4700736999511719
Epoch 3: [18000/50000] Loss: 0.779529869556427
Epoch 3: [18900/50000] Loss: 1.005055546760559
Epoch 3: [19800/50000] Loss: 1.0191023349761963
Epoch 3: [20700/50000] Loss: 1.3922312259674072
Epoch 3: [21600/50000] Loss: 1.583356261253357
Epoch 3: [22500/50000] Loss: 0.8319861888885498
Epoch 3: [23400/50000] Loss: 1.4574971199035645
Epoch 3: [24300/50000] Loss: 0.9409314393997192
Epoch 3: [25200/50000] Loss: 1.3658194541931152
Epoch 3: [26100/50000] Loss: 0.984150230884552
Epoch 3: [27000/50000] Loss: 1.4271997213363647
Epoch 3: [27900/50000] Loss: 1.1687589883804321
Epoch 3: [28800/50000] Loss: 1.2601910829544067
Epoch 3: [29700/50000] Loss: 0.6778614521026611
Epoch 3: [30600/50000] Loss: 1.5910258293151855
Epoch 3: [31500/50000] Loss: 1.3889461755752563
Epoch 3: [32400/50000] Loss: 0.6454980373382568
Epoch 3: [33300/50000] Loss: 0.859140157699585
Epoch 3: [34200/50000] Loss: 0.9297245740890503
Epoch 3: [35100/50000] Loss: 1.4766268730163574
Epoch 3: [36000/50000] Loss: 1.2971596717834473
Epoch 3: [36900/50000] Loss: 1.406444787979126
Epoch 3: [37800/50000] Loss: 1.1855318546295166
Epoch 3: [38700/50000] Loss: 0.9315974116325378
Epoch 3: [39600/50000] Loss: 1.0091110467910767
Epoch 3: [40500/50000] Loss: 1.1702919006347656
Epoch 3: [41400/50000] Loss: 1.0228756666183472
Epoch 3: [42300/50000] Loss: 1.1787364482879639
Epoch 3: [43200/50000] Loss: 1.3652215003967285
Epoch 3: [44100/50000] Loss: 1.0535550117492676
Epoch 3: [45000/50000] Loss: 1.711564302444458
Epoch 3: [45900/50000] Loss: 0.9721580147743225
Epoch 3: [46800/50000] Loss: 1.3176251649856567
Epoch 3: [47700/50000] Loss: 1.340333104133606
Epoch 3: [48600/50000] Loss: 1.4332070350646973
Epoch 3: [49500/50000] Loss: 1.2777760028839111
Epoch 4: [0/50000] Loss: 1.0615335702896118
Epoch 4: [900/50000] Loss: 1.4185848236083984
Epoch 4: [1800/50000] Loss: 0.6444305181503296
Epoch 4: [2700/50000] Loss: 0.6919461488723755
Epoch 4: [3600/50000] Loss: 0.6487124562263489
Epoch 4: [4500/50000] Loss: 0.7217295169830322
Epoch 4: [5400/50000] Loss: 1.4574980735778809
Epoch 4: [6300/50000] Loss: 0.5107339024543762
Epoch 4: [7200/50000] Loss: 0.7577032446861267
Epoch 4: [8100/50000] Loss: 1.6369519233703613
Epoch 4: [9000/50000] Loss: 1.300873875617981
Epoch 4: [9900/50000] Loss: 1.1880792379379272
Epoch 4: [10800/50000] Loss: 1.491941213607788
Epoch 4: [11700/50000] Loss: 0.961120069026947
Epoch 4: [12600/50000] Loss: 1.2385045289993286
Epoch 4: [13500/50000] Loss: 0.899552583694458
Epoch 4: [14400/50000] Loss: 1.3693711757659912
Epoch 4: [15300/50000] Loss: 0.5101864337921143
Epoch 4: [16200/50000] Loss: 0.5591734647750854
Epoch 4: [17100/50000] Loss: 1.2177468538284302
Epoch 4: [18000/50000] Loss: 1.418099284172058
Epoch 4: [18900/50000] Loss: 0.8075178265571594
Epoch 4: [19800/50000] Loss: 1.0885252952575684
Epoch 4: [20700/50000] Loss: 0.6750888824462891
Epoch 4: [21600/50000] Loss: 1.3219999074935913
Epoch 4: [22500/50000] Loss: 1.3941826820373535
Epoch 4: [23400/50000] Loss: 0.752391517162323
```

```
Epoch 4: [24300/50000] Loss: 1.230473518371582
Epoch 4: [25200/50000] Loss: 0.985683023929596
Epoch 4: [26100/50000] Loss: 2.1047449111938477
Epoch 4: [27000/50000] Loss: 1.0634948015213013
Epoch 4: [27900/50000] Loss: 1.088921070098877
Epoch 4: [28800/50000] Loss: 1.4579046964645386
Epoch 4: [29700/50000] Loss: 0.7629912495613098
Epoch 4: [30600/50000] Loss: 0.7970932722091675
Epoch 4: [31500/50000] Loss: 1.1444460153579712
Epoch 4: [32400/50000] Loss: 0.8442953824996948
Epoch 4: [33300/50000] Loss: 1.350523591041565
Epoch 4: [34200/50000] Loss: 0.6590205430984497
Epoch 4: [35100/50000] Loss: 1.0709158182144165
Epoch 4: [36000/50000] Loss: 1.0897608995437622
Epoch 4: [36900/50000] Loss: 1.3507342338562012
Epoch 4: [37800/50000] Loss: 0.5025115609169006
Epoch 4: [38700/50000] Loss: 1.5453457832336426
Epoch 4: [39600/50000] Loss: 1.1982403993606567
Epoch 4: [40500/50000] Loss: 1.1730624437332153
Epoch 4: [41400/50000] Loss: 0.7533056735992432
Epoch 4: [42300/50000] Loss: 1.0699715614318848
Epoch 4: [43200/50000] Loss: 0.7052624821662903
Epoch 4: [44100/50000] Loss: 0.7078002095222473
Epoch 4: [45000/50000] Loss: 0.8418609499931335
Epoch 4: [45900/50000] Loss: 0.9645633697509766
Epoch 4: [46800/50000] Loss: 0.7785659432411194
Epoch 4: [47700/50000] Loss: 1.2665808200836182
Epoch 4: [48600/50000] Loss: 0.48794025182724
Epoch 4: [49500/50000] Loss: 0.8028090000152588
Test result on epoch 4: total samples: 10000, Avg loss: 1.070, Acc: 61.870%
Finished Training after 158.98181295394897 s
```

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 [41]:

```python
total_images = 3
predictions = output['prediction']
targets = torch.tensor(testset.targets)
# ----------- <Your code> ---------------
# Compare predictions with the ground truth labels to generate a binary mask
correct = predictions.detach().cpu().eq(targets.data.view_as(predictions))

# Find the indices of the misclassified images
misclassified_indices = torch.where(~correct)[0]

# Select N misclassified images randomly
torch.manual_seed(0)
num_misclassified = 3
misclassified_indices = misclassified_indices[torch.randperm(len(misclassified_indice

# Load the first x misclassified images and their labels
misclassified_images = torch.stack([testset[i][0] for i in misclassified_indices])
misclassified_labels = torch.stack([targets[i] for i in misclassified_indices])

# Plot the images and display the true and predicted classes
fig, ax = plt.subplots(1, num_misclassified, figsize=(12, 6))
for i in range(num_misclassified):
    image = misclassified_images[i].permute(1,2,0)
    image = image/2 + 0.5
```

```
    ax[i].imshow(image)
    ax[i].set_title(f'True: {classes[misclassified_labels[i]]}, Predicted: {classes[p
    ax[i].axis('off')
# ----------- <End Your code> ---------------
```

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

Yes, although some of the correctly predicted ones are also misleading.

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 [42]:
```
resnet18 = models.resnet18(pretrained=True)
resnet18 = resnet18.to(device)
```

```
/Users/crich/Development/School/ECE5700-AI/venv/lib/python3.12/site-packages/torchvisi
on/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since
0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/Users/crich/Development/School/ECE5700-AI/venv/lib/python3.12/site-packages/torchvisi
on/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future. The current beha
vior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also u
se `weights=ResNet18_Weights.DEFAULT` 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 [43]:
```
# Create your dataloader here
# ----------- <Your code> ---------------
# Define the transform for cifar10 so that it is the correct size for resnet18
transform = torchvision.transforms.Compose([torchvision.transforms.Resize((224, 224))
                                            torchvision.transforms.ToTensor(),
                                            torchvision.transforms.Normalize((0.5, 0.5

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

train_loader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num
test_loader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_
# ----------- <End Your code> ---------------
```

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

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 [44]:
```
# ----------- <Your code> ---------------
criterion = nn.CrossEntropyLoss()
test(resnet18, criterion, test_loader);
# ----------- <End Your code> ---------------
```

```
Test result on epoch 0: total samples: 10000, Avg loss: 11.707, Acc: 0.030%
```

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

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 [45]:
```
# Directly train the whole model.
start = time.time()
#----------- <Your code> ---------------
# Replace output layer to the resnet18 model for CIFAR10
```

```
num_ftrs = resnet18.fc.in_features
resnet18.fc = nn.Linear(num_ftrs, 10)
resnet18 = resnet18.to(device)
# Train
optimizer = optim.Adadelta(resnet18.parameters(), lr=2.0)
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.8)
max_epoch = 10
for epoch in range(1, max_epoch+1):
  train(resnet18, criterion, optimizer, train_loader, epoch)
  scheduler.step()

# ----------- <End Your code> ---------------
test(resnet18, criterion, test_loader, epoch)
end = time.time()
print(f'Finished Training after {end-start} s ')
```

```
Epoch 1: [0/50000] Loss: 2.455737352371216
Epoch 1: [6400/50000] Loss: 1.8351480960845947
Epoch 1: [12800/50000] Loss: 1.4284050464630127
Epoch 1: [19200/50000] Loss: 1.1470718383789062
Epoch 1: [25600/50000] Loss: 1.1097432374954224
Epoch 1: [32000/50000] Loss: 0.8831242918968201
Epoch 1: [38400/50000] Loss: 0.7232311367988586
Epoch 1: [44800/50000] Loss: 0.806818962097168
Epoch 2: [0/50000] Loss: 1.260986566543579
Epoch 2: [6400/50000] Loss: 0.6820101737976074
Epoch 2: [12800/50000] Loss: 0.5606314539909363
Epoch 2: [19200/50000] Loss: 0.4148138761520386
Epoch 2: [25600/50000] Loss: 0.7936642169952393
Epoch 2: [32000/50000] Loss: 0.8733022212982178
Epoch 2: [38400/50000] Loss: 0.7558035850524902
Epoch 2: [44800/50000] Loss: 0.3233976364135742
Epoch 3: [0/50000] Loss: 0.3677864670753479
Epoch 3: [6400/50000] Loss: 0.6645601987838745
Epoch 3: [12800/50000] Loss: 0.3766672611236572
Epoch 3: [19200/50000] Loss: 0.4428384304046631
Epoch 3: [25600/50000] Loss: 0.4343269467353821
Epoch 3: [32000/50000] Loss: 0.6250228881835938
Epoch 3: [38400/50000] Loss: 0.3310738205909729
Epoch 3: [44800/50000] Loss: 0.31316056847572327
Epoch 4: [0/50000] Loss: 0.23196682333946228
Epoch 4: [6400/50000] Loss: 0.3075266480445862
Epoch 4: [12800/50000] Loss: 0.36879056692123413
Epoch 4: [19200/50000] Loss: 0.29604414105415344
Epoch 4: [25600/50000] Loss: 0.17277595400810242
Epoch 4: [32000/50000] Loss: 0.1835363805294037
Epoch 4: [38400/50000] Loss: 0.3098328113555908
Epoch 4: [44800/50000] Loss: 0.4252855181694031
Epoch 5: [0/50000] Loss: 0.08815842866897583
Epoch 5: [6400/50000] Loss: 0.08945371210575104
Epoch 5: [12800/50000] Loss: 0.07851637899875641
Epoch 5: [19200/50000] Loss: 0.09542998671531677
Epoch 5: [25600/50000] Loss: 0.19645436108112335
Epoch 5: [32000/50000] Loss: 0.044943712651729584
Epoch 5: [38400/50000] Loss: 0.30096471309661865
Epoch 5: [44800/50000] Loss: 0.045926935970783234
Epoch 6: [0/50000] Loss: 0.3136380910873413
Epoch 6: [6400/50000] Loss: 0.07231145352125168
Epoch 6: [12800/50000] Loss: 0.09214847534894943
Epoch 6: [19200/50000] Loss: 0.07290101051330566
Epoch 6: [25600/50000] Loss: 0.04917777329683304
Epoch 6: [32000/50000] Loss: 0.056960996240377426
Epoch 6: [38400/50000] Loss: 0.05469515919685364
Epoch 6: [44800/50000] Loss: 0.022773031145334244
Epoch 7: [0/50000] Loss: 0.0035079626832157373
Epoch 7: [6400/50000] Loss: 0.06529614329338074
Epoch 7: [12800/50000] Loss: 0.010735446587204933
Epoch 7: [19200/50000] Loss: 0.0036503742448985577
Epoch 7: [25600/50000] Loss: 0.01040071714669466
Epoch 7: [32000/50000] Loss: 0.003284197300672531
Epoch 7: [38400/50000] Loss: 0.05746166780591011
Epoch 7: [44800/50000] Loss: 0.023919574916362762
Epoch 8: [0/50000] Loss: 0.08828301727771759
Epoch 8: [6400/50000] Loss: 0.009974498301744461
Epoch 8: [12800/50000] Loss: 0.0035578501410782337
Epoch 8: [19200/50000] Loss: 0.0002758373157121241
Epoch 8: [25600/50000] Loss: 0.045225657522678375
Epoch 8: [32000/50000] Loss: 0.002103306818753481
Epoch 8: [38400/50000] Loss: 0.005095007363706827
Epoch 8: [44800/50000] Loss: 0.005901057738810778
Epoch 9: [0/50000] Loss: 0.0007321221055462956
```

```
Epoch 9: [6400/50000] Loss: 0.0004380677128210664
Epoch 9: [12800/50000] Loss: 0.00013520810171030462
Epoch 9: [19200/50000] Loss: 0.0002504517906345427
Epoch 9: [25600/50000] Loss: 0.00814188364893198
Epoch 9: [32000/50000] Loss: 1.1301859558443539e-05
Epoch 9: [38400/50000] Loss: 0.0019423068733885884
Epoch 9: [44800/50000] Loss: 0.0013238276587799191
Epoch 10: [0/50000] Loss: 0.049948640167713165
Epoch 10: [6400/50000] Loss: 0.0002620444865897298
Epoch 10: [12800/50000] Loss: 0.0003274405898991972
Epoch 10: [19200/50000] Loss: 9.421687718713656e-05
Epoch 10: [25600/50000] Loss: 0.0003297037910670042
Epoch 10: [32000/50000] Loss: 0.0038742769975215197
Epoch 10: [38400/50000] Loss: 0.0022757539991289377
Epoch 10: [44800/50000] Loss: 0.00030583038460463285
Test result on epoch 10: total samples: 10000, Avg loss: 0.691, Acc: 89.890%
Finished Training after 2025.7652380466461 s
```

In [46]:

```python
# Load another resnet18 instance, only unfreeze the outer layers.
# ----------- <Your code> ---------------
# Helper function to print model layers
def print_model_layers(model: nn.Module):
    for name, layer in model.named_modules():
        print(name, layer)

# Define model with output layer replaced for CIFAR10
resnet18 = models.resnet18(pretrained=True)
num_ftrs = resnet18.fc.in_features
resnet18.fc = nn.Linear(num_ftrs, 10)
resnet18 = resnet18.to(device)
# print_model_layers(resnet18)

# Freeze all the layers in the model
for param in resnet18.parameters():
    param.requires_grad = False

# Unfreeze the last layer
for param in resnet18.fc.parameters():
    param.requires_grad = True
# ----------- <End Your code> ---------------
```

In [47]:

```python
# Train the model!!
start = time.time()
# ----------- <Your code> ---------------
optimizer = optim.Adadelta(resnet18.parameters(), lr=2.0)
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.8)
max_epoch = 10
for epoch in range(1, max_epoch+1):
  train(resnet18, criterion, optimizer, train_loader, epoch)
  scheduler.step()
# ----------- <End Your code> ---------------
test(resnet18, criterion, test_loader)
end = time.time()
print(f'Finished Training after {end-start} s ')
```

```
Epoch 1: [0/50000] Loss: 2.4513375759124756
Epoch 1: [6400/50000] Loss: 0.8223788738250732
Epoch 1: [12800/50000] Loss: 1.011115550994873
Epoch 1: [19200/50000] Loss: 1.0958218574523926
Epoch 1: [25600/50000] Loss: 0.530171811580658
Epoch 1: [32000/50000] Loss: 0.9504736065864563
Epoch 1: [38400/50000] Loss: 0.6550635695457458
Epoch 1: [44800/50000] Loss: 0.865522027015686
Epoch 2: [0/50000] Loss: 1.84061598777771
Epoch 2: [6400/50000] Loss: 0.3884732127189636
Epoch 2: [12800/50000] Loss: 0.7712877988815308
Epoch 2: [19200/50000] Loss: 1.094550609588623
Epoch 2: [25600/50000] Loss: 0.9703587293624878
Epoch 2: [32000/50000] Loss: 0.5292966365814209
Epoch 2: [38400/50000] Loss: 0.5936389565467834
Epoch 2: [44800/50000] Loss: 0.43292999267578125
Epoch 3: [0/50000] Loss: 0.767737627029419
Epoch 3: [6400/50000] Loss: 0.690311074256897
Epoch 3: [12800/50000] Loss: 0.5836412906646729
Epoch 3: [19200/50000] Loss: 0.7930392026901245
Epoch 3: [25600/50000] Loss: 0.9157977104187012
Epoch 3: [32000/50000] Loss: 0.47803056240081787
Epoch 3: [38400/50000] Loss: 1.0779328346252441
Epoch 3: [44800/50000] Loss: 0.6509675979614258
Epoch 4: [0/50000] Loss: 0.7043559551239014
Epoch 4: [6400/50000] Loss: 0.3472605049610138
Epoch 4: [12800/50000] Loss: 0.5977930426597595
Epoch 4: [19200/50000] Loss: 0.5748199224472046
Epoch 4: [25600/50000] Loss: 0.6827872395515442
Epoch 4: [32000/50000] Loss: 0.8599730134010315
Epoch 4: [38400/50000] Loss: 0.6672967672348022
Epoch 4: [44800/50000] Loss: 0.7529177665710449
Epoch 5: [0/50000] Loss: 0.732818603515625
Epoch 5: [6400/50000] Loss: 0.44662195444107056
Epoch 5: [12800/50000] Loss: 0.680932879447937
Epoch 5: [19200/50000] Loss: 0.31240373849868774
Epoch 5: [25600/50000] Loss: 0.42053958773612976
Epoch 5: [32000/50000] Loss: 0.6184906959533691
Epoch 5: [38400/50000] Loss: 0.39910566806793213
Epoch 5: [44800/50000] Loss: 0.46631094813346863
Epoch 6: [0/50000] Loss: 0.7122711539268494
Epoch 6: [6400/50000] Loss: 0.6651718616485596
Epoch 6: [12800/50000] Loss: 0.5851793885231018
Epoch 6: [19200/50000] Loss: 0.4257611036300659
Epoch 6: [25600/50000] Loss: 0.5831985473632812
Epoch 6: [32000/50000] Loss: 0.4865630865097046
Epoch 6: [38400/50000] Loss: 0.533503532409668
Epoch 6: [44800/50000] Loss: 0.4893721640110016
Epoch 7: [0/50000] Loss: 0.5157039165496826
Epoch 7: [6400/50000] Loss: 0.551048755645752
Epoch 7: [12800/50000] Loss: 0.7604279518127441
Epoch 7: [19200/50000] Loss: 0.5399692058563232
Epoch 7: [25600/50000] Loss: 0.3798537850379944
Epoch 7: [32000/50000] Loss: 0.8176313638687134
Epoch 7: [38400/50000] Loss: 0.5513312220573425
Epoch 7: [44800/50000] Loss: 0.5545750856399536
Epoch 8: [0/50000] Loss: 0.44539377093315125
Epoch 8: [6400/50000] Loss: 0.6341750025749207
Epoch 8: [12800/50000] Loss: 0.4548236131668091
Epoch 8: [19200/50000] Loss: 0.567632257938385
Epoch 8: [25600/50000] Loss: 0.6433811187744141
Epoch 8: [32000/50000] Loss: 0.7074121236801147
Epoch 8: [38400/50000] Loss: 0.4494427740573883
Epoch 8: [44800/50000] Loss: 0.3641104996204376
Epoch 9: [0/50000] Loss: 0.3231915235519409
```

```
Epoch 9: [6400/50000] Loss: 0.7142308950424194
Epoch 9: [12800/50000] Loss: 0.5763132572174072
Epoch 9: [19200/50000] Loss: 0.3686150908470154
Epoch 9: [25600/50000] Loss: 0.7408024072647095
Epoch 9: [32000/50000] Loss: 0.5038802623748779
Epoch 9: [38400/50000] Loss: 0.40002381801605225
Epoch 9: [44800/50000] Loss: 0.6471608877182007
Epoch 10: [0/50000] Loss: 0.5599645376205444
Epoch 10: [6400/50000] Loss: 0.6520671844482422
Epoch 10: [12800/50000] Loss: 0.35117948055267334
Epoch 10: [19200/50000] Loss: 0.5856771469116211
Epoch 10: [25600/50000] Loss: 0.4719190299510956
Epoch 10: [32000/50000] Loss: 0.38351452350616455
Epoch 10: [38400/50000] Loss: 0.37984699010849
Epoch 10: [44800/50000] Loss: 0.9609031677246094
Test result on epoch 0: total samples: 10000, Avg loss: 0.570, Acc: 80.830%
Finished Training after 674.186439037323 s
```

In [50]:

```
# Close and clear widgets, required for export?
from ipywidgets import Widget
Widget.close_all()
```