ECE 47300 Assignment 5

Your Name:Tina Xu

Prepare the pacakges we will use.

In [22]:
```
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)

For some tasks in this assignment, it can take a long time if you run it on CPU. For example, based on our test on Exercise 3 Task 4, it will take roughly 2 hours to train the full model 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 [23]:
```
print(f'Can I can use GPU now? -- {torch.cuda.is_available()}')
```

```
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 [24]:
```
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
```

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 [25]:
```
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()
```

```
    train_loss = []
    train_counter = []
    for batch_idx, (images, label) in enumerate(train_loader):
      images, label = images.to(device), label.to(device)
      label = label.squeeze()
      optimizer.zero_grad()
      output = model(images)
      loss = loss_fn(output, label)
      loss.backward()
      optimizer.step()

      #if batch_idx % 10 == 0: # We record our output every 10 batches
      train_loss.append(loss.item())
      if batch_idx % 100 == 0:
        print(f'Epoch {epoch}: [{batch_idx*len(images)}/{len(train_loader.dataset)}] L
    # ----------- <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> ---------------
    total_num = len(test_dataset)
    model.eval()
    test_loss = 0
    correct = 0
    pred_list = []
    with torch.no_grad():
      for images, targets in test_loader:
        images, targets = images.to(device), targets.to(device)
        output = model(images)
        test_loss += loss_fn(output, targets)
        pred = output.data.max(1, keepdim=True)[1]
        pred_list.extend(pred.view(-1).cpu().numpy())
        correct += pred.eq(targets.data.view_as(pred)).sum()

    test_loss /= len(test_loader.dataset)
    accuracy = correct / len(test_loader.dataset)
    test_stat = {
        'loss': test_loss,
        'accuracy': accuracy,
        'prediction': torch.tensor(pred_list)
    }

    print(f"Test result on epoch {epoch}: total sample: {total_num}, Avg loss: {test_s
    # ----------- <End 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 need to add one `nn.Linear(*, 256)` where one of the dimension is `256` and decide how many other layers and hidden dimensions you want in your network (apart from this).
- Your final accuracy on the test dataset should lie roughly around 97% ($\pm$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

- You should have one `nn.Conv2d(*, *, kernel_size=5)` convolutional layer with `kernel_size=5`, and again, you should decide how many layers and channels you want for each layer.
- Your final accuracy on the test dataset should lie roughly around 97% ($\pm$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 [26]:
```
# Download MNIST and transformation
# ----------- <Your code> ---------------
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

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 [27]:
```
# Build OurFC class and OurCNN class.
# ----------- <Your code> ---------------
class OurFC(nn.Module):
  def __init__(self):
    super(OurFC, self).__init__()
```

```
    self.fc1 = nn.Linear(28 * 28, 256)
    self.fc2 = nn.Linear(256, 128)
    self.fc3 = nn.Linear(128, 64)
    self.fc4 = nn.Linear(64, 10)

  def forward(self, x):
    x = x.view(-1, 28 * 28)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = F.relu(self.fc3(x))
    x = self.fc4(x)
    return x

class OurCNN(nn.Module):
  def __init__(self):
    super(OurCNN, self).__init__()
    self.conv = nn.Conv2d(1, 3, kernel_size=5)
    self.conv2 = nn.Conv2d(3, 6, kernel_size=3)
    self.fc = nn.Linear(432, 10)

  def forward(self, x):
    x = self.conv(x)        # x now has shape (batchsize x 3 x 24 x 24)
    x = F.relu(F.max_pool2d(x,2))  # x now has shape (batchsize x 3 x 12 x 12)
    x = x.view(-1, 432)      # x now has shape (batchsize x 432)
    x = F.relu(self.fc(x))     # x has shape (batchsize x 10)
    return F.log_softmax(x,-1)
# ----------- <End Your code> ---------------
```

In [28]:
```
# Let's first train the FC model. Below are there common hyperparameters.
criterion = nn.CrossEntropyLoss()

start = time.time()
max_epoch = 4
# ----------- <Your code> ---------------
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

max_epoch = 3
classifier = OurFC()
classifier = classifier.to(device)
optimizer = optim.SGD(classifier.parameters(), lr=0.01, momentum=0.8)
train_loss = []
test_losses = []

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: 2.3225107192993164
Epoch 1: [6400/60000] Loss: 0.9476959705352783
Epoch 1: [12800/60000] Loss: 0.6406949162483215
Epoch 1: [19200/60000] Loss: 0.35390621423721313
Epoch 1: [25600/60000] Loss: 0.19334815442562103
Epoch 1: [32000/60000] Loss: 0.38602402806282043
Epoch 1: [38400/60000] Loss: 0.29433339834213257
Epoch 1: [44800/60000] Loss: 0.19536735117435455
Epoch 1: [51200/60000] Loss: 0.24671359360218048
Epoch 1: [57600/60000] Loss: 0.3474026918411255
Test result on epoch 1: total sample: 10000, Avg loss: 0.000, Acc: 94.600%
Epoch 2: [0/60000] Loss: 0.06289239972829819
Epoch 2: [6400/60000] Loss: 0.12653708457946777
Epoch 2: [12800/60000] Loss: 0.13868169486522675
Epoch 2: [19200/60000] Loss: 0.09734631329774857
Epoch 2: [25600/60000] Loss: 0.16340899467468262
Epoch 2: [32000/60000] Loss: 0.22193831205368042
Epoch 2: [38400/60000] Loss: 0.04736936092376709
Epoch 2: [44800/60000] Loss: 0.10074757784605026
Epoch 2: [51200/60000] Loss: 0.15586017072200775
Epoch 2: [57600/60000] Loss: 0.06606201827526093
Test result on epoch 2: total sample: 10000, Avg loss: 0.000, Acc: 96.290%
Epoch 3: [0/60000] Loss: 0.1526002436876297
Epoch 3: [6400/60000] Loss: 0.1196659579873085
Epoch 3: [12800/60000] Loss: 0.06038293242454529
Epoch 3: [19200/60000] Loss: 0.06527547538280487
Epoch 3: [25600/60000] Loss: 0.05436798185110092
Epoch 3: [32000/60000] Loss: 0.26116129755973816
Epoch 3: [38400/60000] Loss: 0.1038118377327919
Epoch 3: [44800/60000] Loss: 0.1607113480567932
Epoch 3: [51200/60000] Loss: 0.1048082560300827
Epoch 3: [57600/60000] Loss: 0.03551391884684563
Test result on epoch 3: total sample: 10000, Avg loss: 0.000, Acc: 97.120%
Finished Training after 46.12486934661865 s
```

In [29]:

```python
criterion = nn.CrossEntropyLoss()

# Let's then train the OurCNN model.
start = time.time()
# ----------- <Your code> ---------------
max_epoch = 3
classifier = OurCNN()
classifier = classifier.to(device)
optimizer = optim.SGD(classifier.parameters(), lr=0.01, momentum=0.8)
train_loss = []
test_loss = []

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: 2.3162405490875244
Epoch 1: [6400/60000] Loss: 0.8069671988487244
Epoch 1: [12800/60000] Loss: 0.8876321315765381
Epoch 1: [19200/60000] Loss: 0.6190297603607178
Epoch 1: [25600/60000] Loss: 0.5085657835006714
Epoch 1: [32000/60000] Loss: 0.48061224818229675
Epoch 1: [38400/60000] Loss: 0.26104214787483215
Epoch 1: [44800/60000] Loss: 0.22581329941749573
Epoch 1: [51200/60000] Loss: 0.07480457425117493
Epoch 1: [57600/60000] Loss: 0.1471724659204483
Test result on epoch 1: total sample: 10000, Avg loss: 0.000, Acc: 96.150%
Epoch 2: [0/60000] Loss: 0.24099670350551605
Epoch 2: [6400/60000] Loss: 0.12015841901302338
Epoch 2: [12800/60000] Loss: 0.14905524253845215
Epoch 2: [19200/60000] Loss: 0.12678638100624084
Epoch 2: [25600/60000] Loss: 0.05848908796906471
Epoch 2: [32000/60000] Loss: 0.04194008186459541
Epoch 2: [38400/60000] Loss: 0.1255013644695282
Epoch 2: [44800/60000] Loss: 0.1321403831243515
Epoch 2: [51200/60000] Loss: 0.22423624992370605
Epoch 2: [57600/60000] Loss: 0.18203610181808472
Test result on epoch 2: total sample: 10000, Avg loss: 0.000, Acc: 96.610%
Epoch 3: [0/60000] Loss: 0.07076451927423477
Epoch 3: [6400/60000] Loss: 0.017751364037394524
Epoch 3: [12800/60000] Loss: 0.11676153540611267
Epoch 3: [19200/60000] Loss: 0.06384273618459702
Epoch 3: [25600/60000] Loss: 0.07712694257497787
Epoch 3: [32000/60000] Loss: 0.03436969593167305
Epoch 3: [38400/60000] Loss: 0.06634313613176346
Epoch 3: [44800/60000] Loss: 0.0653982013463974
Epoch 3: [51200/60000] Loss: 0.07489781081676483
Epoch 3: [57600/60000] Loss: 0.24033835530281067
Test result on epoch 3: total sample: 10000, Avg loss: 0.000, Acc: 97.430%
Finished Training after 45.245373487472534 s
```

In [30]:

```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 242762 parameters
OurCNN has a total of 4576 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?

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 [31]:
```
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> ---------------
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
 download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
 download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(trainset, batch_size=9,
 shuffle=True, num_workers=2)
test_loader = torch.utils.data.DataLoader(testset, batch_size=9,
 shuffle=False, num_workers=2)

# ----------- <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
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB d
ata ([0..1] for floats or [0..255] for integers).
```

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 [32]:
```
# Create CNN network.
# ----------- <Your code> ---------------
class Net(nn.Module):#This class is from lecture demo
 def __init__(self):
  super(Net, self).__init__()
  self.conv1 = nn.Conv2d(3, 6, 5)
  self.pool = nn.MaxPool2d(2, 2)
  self.conv2 = nn.Conv2d(6, 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):
  x = self.pool(F.relu(self.conv1(x))) # (N, 6, 14, 14)
  x = self.pool(F.relu(self.conv2(x))) # (N, 16, 5, 5)
  x = x.view(-1, 16 * 5 * 5) # (N, 400)
  x = F.relu(self.fc1(x)) # (N, 120)
  x = F.relu(self.fc2(x)) # (N, 84)
  x = self.fc3(x) # (N, 10)
  return x
net = Net()
net = net.to(device)

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

In [33]:
```
# Train your neural network here.
start = time.time()
max_epoch = 4
# ----------- <Your code> ---------------
net = net.to(device) # This function on Lecture demo
for epoch in range(max_epoch):
  running_loss = 0.0
  for i, data in enumerate(train_loader, 0):
    inputs, labels = data
    inputs, labels = inputs.to(device), labels.to(device)
    optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.8)
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
    if i % 2000 == 1999:
      print('[%d, %5d] loss: %.3f' %(epoch + 1, i + 1, running_loss / 2000))
      running_loss = 0.0


# ----------- <End Your code> ---------------
output = test(net, criterion, test_loader, epoch)
```

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

```
[1,  2000] loss: 2.127
[1,  4000] loss: 1.767
[2,  2000] loss: 1.511
[2,  4000] loss: 1.453
[3,  2000] loss: 1.337
[3,  4000] loss: 1.322
[4,  2000] loss: 1.242
[4,  4000] loss: 1.234
Test result on epoch 3: total sample: 10000, Avg loss: 0.142, Acc: 55.090%
Finished Training after 78.66084146499634 s
```

Task 3: Plot misclassified test images

Plot some misclassified images in your test dataset:

- select five images that are **misclassified** for `class_id in {1,3,5,7,9}` by your neural network, one image each (i.e., the true label is `class_id` but the predicted label is not `class_id`).
- label each images with true label and predicted label
- use `detach().cpu()` when plotting images if the image is in gpu

In [34]:
```
total_images = 5
predictions = output['prediction']
targets = torch.tensor(testset.targets)
# ----------- <Your code> ---------------
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru
import matplotlib.pyplot as plt
import numpy as np
misclassified_images = {}
misclassified_preds = {}
misclassified_targets = {}

net.eval()
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        for idx, (image, label) in enumerate(zip(predicted, labels)):
            if image != label and label.item() in [1, 3, 5, 7, 9] and label.item() not
                misclassified_images[label.item()] = images[idx]
                misclassified_preds[label.item()] = image.item()
                misclassified_targets[label.item()] = label.item()
            if len(misclassified_images) >= 5:
                break
        if len(misclassified_images) >= 5:
            break

fig, axes = plt.subplots(1, 5, figsize=(15, 3))
for ax, class_id in zip(axes, [1, 3, 5, 7, 9]):
    img = misclassified_images[class_id].cpu().numpy().transpose((1, 2, 0))
    img = np.clip(img, 0, 1)
    ax.imshow(img)
    ax.set_title(f'True: {classes[misclassified_targets[class_id]]}\nPred: {classes[mi
```

```
    ax.axis('off')

plt.show()


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

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

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 [35]:
```
resnet18 = models.resnet18(pretrained=True)
resnet18 = resnet18.to(device)
```

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 [54]:
```
# Create your dataloader here
# ----------- <Your code> ---------------
from torchvision.transforms import v2

transforms = v2.Compose([
    v2.RandomResizedCrop(size=(224, 224), antialias=True),
    v2.RandomHorizontalFlip(p=0.5),
    v2.PILToTensor(),
    v2.ToDtype(torch.float32, scale=True),
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, trar
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, trar
train_loader = torch.utils.data.DataLoader(trainset, batch_size=9, shuffle=True, num_w
test_loader = torch.utils.data.DataLoader(testset, batch_size=9, shuffle=False, num_wc

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

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

Task3: Classify test data on pretrained model

link textUse 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 [37]:
```
# ----------- <Your code> --------------
test(resnet18, criterion, test_loader, epoch)
# ----------- <End Your code> ---------------
```

```
Test result on epoch 3: total sample: 10000, Avg loss: 1.348, Acc: 0.030%
```

Out[37]:
```
{'loss': tensor(1.3482, device='cuda:0'),
 'accuracy': tensor(0.0003, device='cuda:0'),
 'prediction': tensor([653, 567, 147,  ..., 383, 392, 268])}
```

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 [41]:
```
# Directly train the whole model.
start = time.time()
#----------- <Your code> ---------------
max_epoch = 4
optimizer = optim.SGD(filter(lambda p: p.requires_grad, resnet18.parameters()), lr=0.0
for epoch in range(1, max_epoch+1):
  train(resnet18, criterion, optimizer, train_loader, epoch)
# ----------- <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: 0.9849302768707275
Epoch 1: [900/50000] Loss: 0.6891946196556091
Epoch 1: [1800/50000] Loss: 0.547964334487915
Epoch 1: [2700/50000] Loss: 0.8184277415275574
Epoch 1: [3600/50000] Loss: 0.5266135334968567
Epoch 1: [4500/50000] Loss: 0.8284181356430054
Epoch 1: [5400/50000] Loss: 0.9807007312774658
Epoch 1: [6300/50000] Loss: 0.5416517853736877
Epoch 1: [7200/50000] Loss: 0.4476836025714874
Epoch 1: [8100/50000] Loss: 0.4538692831993103
Epoch 1: [9000/50000] Loss: 0.66401606798172
Epoch 1: [9900/50000] Loss: 0.9981821775436401
Epoch 1: [10800/50000] Loss: 0.6260478496551514
Epoch 1: [11700/50000] Loss: 0.6677668690681458
Epoch 1: [12600/50000] Loss: 0.7393826246261597
Epoch 1: [13500/50000] Loss: 0.7403638362884521
Epoch 1: [14400/50000] Loss: 0.5619577169418335
Epoch 1: [15300/50000] Loss: 0.8158794045448303
Epoch 1: [16200/50000] Loss: 1.4424934387207031
Epoch 1: [17100/50000] Loss: 0.45140504837036133
Epoch 1: [18000/50000] Loss: 0.6159303188323975
Epoch 1: [18900/50000] Loss: 0.28817957639694214
Epoch 1: [19800/50000] Loss: 1.1235251426696777
Epoch 1: [20700/50000] Loss: 0.23907096683979034
Epoch 1: [21600/50000] Loss: 0.8489562273025513
Epoch 1: [22500/50000] Loss: 1.2779484987258911
Epoch 1: [23400/50000] Loss: 0.6469547748565674
Epoch 1: [24300/50000] Loss: 1.697003960609436
Epoch 1: [25200/50000] Loss: 0.5200656652450562
Epoch 1: [26100/50000] Loss: 0.19058328866958618
Epoch 1: [27000/50000] Loss: 0.3811015784740448
Epoch 1: [27900/50000] Loss: 0.7860597372055054
Epoch 1: [28800/50000] Loss: 0.3740847110748291
Epoch 1: [29700/50000] Loss: 0.40207087993621826
Epoch 1: [30600/50000] Loss: 0.36634576320648193
Epoch 1: [31500/50000] Loss: 0.16625672578811646
Epoch 1: [32400/50000] Loss: 1.1789175271987915
Epoch 1: [33300/50000] Loss: 0.50594162940979
Epoch 1: [34200/50000] Loss: 1.1102519035339355
Epoch 1: [35100/50000] Loss: 0.2945215702056885
Epoch 1: [36000/50000] Loss: 0.6855146884918213
Epoch 1: [36900/50000] Loss: 0.7727756500244141
Epoch 1: [37800/50000] Loss: 0.35939958691596985
Epoch 1: [38700/50000] Loss: 0.4412614107131958
Epoch 1: [39600/50000] Loss: 0.5531730055809021
Epoch 1: [40500/50000] Loss: 0.5961208343505859
Epoch 1: [41400/50000] Loss: 0.3412613272666931
Epoch 1: [42300/50000] Loss: 0.48015096783638
Epoch 1: [43200/50000] Loss: 0.7083131074905396
Epoch 1: [44100/50000] Loss: 1.476594090461731
Epoch 1: [45000/50000] Loss: 0.3266851305961609
Epoch 1: [45900/50000] Loss: 0.6296382546424866
Epoch 1: [46800/50000] Loss: 0.24130858480930328
Epoch 1: [47700/50000] Loss: 1.169346570968628
Epoch 1: [48600/50000] Loss: 0.34805166721343994
Epoch 1: [49500/50000] Loss: 0.6282298564910889
Epoch 2: [0/50000] Loss: 0.20323872566223145
Epoch 2: [900/50000] Loss: 0.22967730462551117
Epoch 2: [1800/50000] Loss: 0.38426321744918823
Epoch 2: [2700/50000] Loss: 0.8651399612426758
```

```
Epoch 2: [3600/50000] Loss: 0.4233931005001068
Epoch 2: [4500/50000] Loss: 0.5420306324958801
Epoch 2: [5400/50000] Loss: 1.1341089010238647
Epoch 2: [6300/50000] Loss: 0.21644242107868195
Epoch 2: [7200/50000] Loss: 0.8338583707809448
Epoch 2: [8100/50000] Loss: 0.8227366209030151
Epoch 2: [9000/50000] Loss: 0.7115365862846375
Epoch 2: [9900/50000] Loss: 0.5526877641677856
Epoch 2: [10800/50000] Loss: 0.7473070025444031
Epoch 2: [11700/50000] Loss: 1.6697274446487427
Epoch 2: [12600/50000] Loss: 0.865280032157898
Epoch 2: [13500/50000] Loss: 0.49589619040489197
Epoch 2: [14400/50000] Loss: 0.1453690379858017
Epoch 2: [15300/50000] Loss: 0.43032947182655334
Epoch 2: [16200/50000] Loss: 0.5002106428146362
Epoch 2: [17100/50000] Loss: 0.5027185678482056
Epoch 2: [18000/50000] Loss: 0.8088065981864929
Epoch 2: [18900/50000] Loss: 0.5112119317054749
Epoch 2: [19800/50000] Loss: 0.5504079461097717
Epoch 2: [20700/50000] Loss: 0.7669381499290466
Epoch 2: [21600/50000] Loss: 0.29870861768722534
Epoch 2: [22500/50000] Loss: 0.6886396408081055
Epoch 2: [23400/50000] Loss: 0.6568844318389893
Epoch 2: [24300/50000] Loss: 0.3270705044269562
Epoch 2: [25200/50000] Loss: 0.9025945663452148
Epoch 2: [26100/50000] Loss: 0.05027858540415764
Epoch 2: [27000/50000] Loss: 0.44731923937797546
Epoch 2: [27900/50000] Loss: 0.4595891535282135
Epoch 2: [28800/50000] Loss: 0.41101211309432983
Epoch 2: [29700/50000] Loss: 0.34626272320747375
Epoch 2: [30600/50000] Loss: 0.5789348483085632
Epoch 2: [31500/50000] Loss: 0.5101833343505859
Epoch 2: [32400/50000] Loss: 0.2846188545227051
Epoch 2: [33300/50000] Loss: 0.6277625560760498
Epoch 2: [34200/50000] Loss: 0.878325879573822
Epoch 2: [35100/50000] Loss: 0.4823019802570343
Epoch 2: [36000/50000] Loss: 1.2615007162094116
Epoch 2: [36900/50000] Loss: 0.21819984912872314
Epoch 2: [37800/50000] Loss: 0.615347146987915
Epoch 2: [38700/50000] Loss: 0.40865838527679443
Epoch 2: [39600/50000] Loss: 0.3973275423049927
Epoch 2: [40500/50000] Loss: 0.8778776526451111
Epoch 2: [41400/50000] Loss: 0.3870818614959717
Epoch 2: [42300/50000] Loss: 0.031020402908325195
Epoch 2: [43200/50000] Loss: 0.4159247875213623
Epoch 2: [44100/50000] Loss: 0.6430480480194092
Epoch 2: [45000/50000] Loss: 0.578813374042511
Epoch 2: [45900/50000] Loss: 0.03151388093829155
Epoch 2: [46800/50000] Loss: 1.8487226963043213
Epoch 2: [47700/50000] Loss: 0.44276708364486694
Epoch 2: [48600/50000] Loss: 1.1476013660430908
Epoch 2: [49500/50000] Loss: 0.6635501384735107
Epoch 3: [0/50000] Loss: 0.8140881061553955
Epoch 3: [900/50000] Loss: 1.3710519075393677
Epoch 3: [1800/50000] Loss: 0.5944159030914307
Epoch 3: [2700/50000] Loss: 1.125868320465088
Epoch 3: [3600/50000] Loss: 0.150893896818161
Epoch 3: [4500/50000] Loss: 0.3372439742088318
Epoch 3: [5400/50000] Loss: 0.31924259662628174
Epoch 3: [6300/50000] Loss: 0.3049107789993286
```

```
Epoch 3: [7200/50000] Loss: 0.36053428053855896
Epoch 3: [8100/50000] Loss: 0.2757761776447296
Epoch 3: [9000/50000] Loss: 1.0924931764602661
Epoch 3: [9900/50000] Loss: 0.5736520886421204
Epoch 3: [10800/50000] Loss: 1.0072927474975586
Epoch 3: [11700/50000] Loss: 0.6616535782814026
Epoch 3: [12600/50000] Loss: 0.5944721102714539
Epoch 3: [13500/50000] Loss: 0.5050820112228394
Epoch 3: [14400/50000] Loss: 0.46904468536376953
Epoch 3: [15300/50000] Loss: 0.5426397323608398
Epoch 3: [16200/50000] Loss: 0.7912757992744446
Epoch 3: [17100/50000] Loss: 0.47037798166275024
Epoch 3: [18000/50000] Loss: 1.0176444053649902
Epoch 3: [18900/50000] Loss: 1.0751872062683105
Epoch 3: [19800/50000] Loss: 1.4605355262756348
Epoch 3: [20700/50000] Loss: 1.0882437229156494
Epoch 3: [21600/50000] Loss: 0.3398069441318512
Epoch 3: [22500/50000] Loss: 0.6517194509506226
Epoch 3: [23400/50000] Loss: 0.15191291272640228
Epoch 3: [24300/50000] Loss: 0.6915101408958435
Epoch 3: [25200/50000] Loss: 1.0269641876220703
Epoch 3: [26100/50000] Loss: 0.09979817271232605
Epoch 3: [27000/50000] Loss: 0.8432350158691406
Epoch 3: [27900/50000] Loss: 0.3602488040924072
Epoch 3: [28800/50000] Loss: 1.089316725730896
Epoch 3: [29700/50000] Loss: 1.0595890283584595
Epoch 3: [30600/50000] Loss: 0.1678881049156189
Epoch 3: [31500/50000] Loss: 0.33670851588249207
Epoch 3: [32400/50000] Loss: 1.0444612503051758
Epoch 3: [33300/50000] Loss: 0.30274391174316406
Epoch 3: [34200/50000] Loss: 1.1024484634399414
Epoch 3: [35100/50000] Loss: 0.1848711520433426
Epoch 3: [36000/50000] Loss: 0.39340299367904663
Epoch 3: [36900/50000] Loss: 0.8191666603088379
Epoch 3: [37800/50000] Loss: 0.6369274258613586
Epoch 3: [38700/50000] Loss: 0.6730743050575256
Epoch 3: [39600/50000] Loss: 0.3020130395889282
Epoch 3: [40500/50000] Loss: 0.7070930004119873
Epoch 3: [41400/50000] Loss: 1.0670198202133179
Epoch 3: [42300/50000] Loss: 0.5216058492660522
Epoch 3: [43200/50000] Loss: 0.23438532650470734
Epoch 3: [44100/50000] Loss: 0.6135627627372742
Epoch 3: [45000/50000] Loss: 0.349598228931427
Epoch 3: [45900/50000] Loss: 0.26562485098838806
Epoch 3: [46800/50000] Loss: 0.8983584642410278
Epoch 3: [47700/50000] Loss: 1.1109256744384766
Epoch 3: [48600/50000] Loss: 0.1894942969083786
Epoch 3: [49500/50000] Loss: 0.36178258061408997
Epoch 4: [0/50000] Loss: 0.3306840658187866
Epoch 4: [900/50000] Loss: 0.30540451407432556
Epoch 4: [1800/50000] Loss: 0.8343163728713989
Epoch 4: [2700/50000] Loss: 0.30153611302375793
Epoch 4: [3600/50000] Loss: 0.9777965545654297
Epoch 4: [4500/50000] Loss: 0.18115361034870148
Epoch 4: [5400/50000] Loss: 1.1445307731628418
Epoch 4: [6300/50000] Loss: 0.28653040528297424
Epoch 4: [7200/50000] Loss: 0.5045759677886963
Epoch 4: [8100/50000] Loss: 0.6300418972969055
Epoch 4: [9000/50000] Loss: 1.1178699731826782
Epoch 4: [9900/50000] Loss: 0.6410528421401978
```

```
Epoch 4: [10800/50000] Loss: 0.5219218730926514
Epoch 4: [11700/50000] Loss: 0.3320443332195282
Epoch 4: [12600/50000] Loss: 0.23285868763923645
Epoch 4: [13500/50000] Loss: 1.387982726097107
Epoch 4: [14400/50000] Loss: 0.8041008114814758
Epoch 4: [15300/50000] Loss: 0.19343164563179016
Epoch 4: [16200/50000] Loss: 0.2519959807395935
Epoch 4: [17100/50000] Loss: 0.5802217721939087
Epoch 4: [18000/50000] Loss: 0.5105597376823425
Epoch 4: [18900/50000] Loss: 0.7861250638961792
Epoch 4: [19800/50000] Loss: 1.0933032035827637
Epoch 4: [20700/50000] Loss: 0.467133104801178
Epoch 4: [21600/50000] Loss: 0.8308685421943665
Epoch 4: [22500/50000] Loss: 0.35176241397857666
Epoch 4: [23400/50000] Loss: 0.3832857310771942
Epoch 4: [24300/50000] Loss: 0.38329702615737915
Epoch 4: [25200/50000] Loss: 1.3424688577651978
Epoch 4: [26100/50000] Loss: 1.9836993217468262
Epoch 4: [27000/50000] Loss: 0.4520920217037201
Epoch 4: [27900/50000] Loss: 1.003834843635559
Epoch 4: [28800/50000] Loss: 0.6368727684020996
Epoch 4: [29700/50000] Loss: 1.8800686597824097
Epoch 4: [30600/50000] Loss: 0.15716761350631714
Epoch 4: [31500/50000] Loss: 0.597868025302887
Epoch 4: [32400/50000] Loss: 0.9818665981292725
Epoch 4: [33300/50000] Loss: 0.9539517164230347
Epoch 4: [34200/50000] Loss: 1.2056217193603516
Epoch 4: [35100/50000] Loss: 1.1459869146347046
Epoch 4: [36000/50000] Loss: 0.2410164624452591
Epoch 4: [36900/50000] Loss: 1.340208888053894
Epoch 4: [37800/50000] Loss: 0.7258832454681396
Epoch 4: [38700/50000] Loss: 0.28668299317359924
Epoch 4: [39600/50000] Loss: 0.9338753819465637
Epoch 4: [40500/50000] Loss: 1.2484253644943237
Epoch 4: [41400/50000] Loss: 0.27167269587516785
Epoch 4: [42300/50000] Loss: 0.6388657689094543
Epoch 4: [43200/50000] Loss: 0.2011478692293167
Epoch 4: [44100/50000] Loss: 1.4204493761062622
Epoch 4: [45000/50000] Loss: 1.1570276021957397
Epoch 4: [45900/50000] Loss: 0.37552720308303833
Epoch 4: [46800/50000] Loss: 0.1384652554988861
Epoch 4: [47700/50000] Loss: 0.16978037357330322
Epoch 4: [48600/50000] Loss: 0.35698598623275757
Epoch 4: [49500/50000] Loss: 0.8229844570159912
Test result on epoch 4: total sample: 10000, Avg loss: 0.061, Acc: 81.720%
Finished Training after 294.74332642555237 s
```

In [57]:

```python
# Load another resnet18 instance, only unfreeze the outer layers.
# ----------- <Your code> ---------------
criterion = nn.CrossEntropyLoss()
epoch = 0
for param in resnet18.parameters():
    param.requires_grad = False
num = resnet18.fc.in_features
resnet18.fc = nn.Linear(num, 10)
resnet18.layer4.requires_grad = True
#resnet18.layer3.requires_grad = True
resnet18.fc.requires_grad = True
# ----------- <End Your code> ---------------
```

In [58]:

```
# Train the model!!
start = time.time()
# ----------- <Your code> ---------------
resnet18 = resnet18.to(device)
num_epochs = 1
for epoch in range(num_epochs):
    resnet18.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer = optim.SGD(filter(lambda p: p.requires_grad, resnet18.parameters())
        optimizer.zero_grad()
        outputs = resnet18(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f'Epoch {epoch+1}, Loss: {running_loss/len(train_loader)}')
# ----------- <End Your code> ---------------
test(resnet18, criterion, test_loader)
end = time.time()
print(f'Finished Training after {end-start} s ')
```

```
Epoch 1, Loss: 1.0273716919381597
Test result on epoch 0: total sample: 10000, Avg loss: 0.072, Acc: 80.340%
Finished Training after 85.05671691894531 s
```