HW3_yj2737

April 6, 2023

1 Homework 3: Object Recognition

> **Submission Instructions:** Before the deadline, export the completed notebook to PDF and upload it to GradeScope. The PDF should clearly show your code, and the result of running the code. Check the PDF to ensure that it is readable, the font-size is not small, and no information is cut-off. There will be no make-ups or extensions for corrupted/damaged/unreadable PDFs.

Names of Collaborators:

In this homework, we will investigate learning a neural network with PyTorch. This will give you some familarity with PyTorch and modern deep learning libraries. First, let's load in PyTorch and several functions that we will use throughout the homework.

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

device = "cuda" if torch.cuda.is_available() else "cpu"

def imshow(img):
    img = img / 2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

For this homework, having access to a GPU will be very useful. To enable a GPU, follow the instructions from the previous homework. If the code below outputs `cuda`, then you are using a GPU.

[ ]:
```
print(device)
```

```
cuda
```

2 1. Problem 1: Building Neural Network

2.1 Introduction to PyTorch

PyTorch is similar to numpy. For the most part, if there is a numpy operation, there is an equivalent PyTorch operation. However, the advantage of PyTorch is that it will automatically calculate gradients through back-propagation and the algorithms are implemented on the GPU.

2.1.1 Automatic Differentiation

You can view PyTorch as Numpy with gradient calculation built in. After you have finished computing your program, there is a `.backward()` function that calculates the gradients for all of the operations in the program. You no longer need to analytically calculate the gradients, code it up, and write a gradient checker.

Let's look at a basic example. We will perform matrix multiplication between `a` and `b`, followed by element-wise multiplication with `c`, and finally we sum the result. Since it is computed in PyTorch, we can simply call `result.backward()` to have all of the gradients calculated throughout the computational graph.

[ ]:
```
a = torch.rand(2,2, requires_grad=True, device=device)
b = torch.rand(2,2, requires_grad=True, device=device)
c = torch.rand(2,2, device=device)

result = torch.matmul(a, b) * c
result = result.sum()

result.backward() # calculate the gradients with back-propagation to the input

print(f'Result: {result.cpu().item()}')
print(f'Gradient a:\n {a.grad}')
print(f'Gradient b:\n {b.grad}')
print(f'Gradient c:\n {c.grad}')
```

```
Result: 0.32349035143852234
Gradient a:
 tensor([[0.6161, 0.0629],
        [0.3089, 0.0457]], device='cuda:0')
Gradient b:
 tensor([[0.4461, 0.1300],
        [0.2934, 0.2578]], device='cuda:0')
Gradient c:
 None
```

Note that only the tensors explicitly marked with `requires_grad=True` will have gradients calculated. Consequently, the gradient for `c` is `None` in the above computation. Automatic differenation makes it possible to implement very creative and complex deep learning algorithms. There has been extensive research and development to create many differentiable functions.

PyTorch makes it easy to transfer variables between the CPU and the GPU. When the variable `a` is constructed, the `device=device` specifies whether to store it on the CPU or GPU, depending on the value of the `device` variable. There is also a `.cpu()` function to bring a variable back to the CPU, and a `.to(device)` function to transfer a variable between devices.

[ ]:
```
cpu_tensor = torch.rand(2,2)
gpu_tensor = cpu_tensor.to("cuda")

cpu_tensor_2 = gpu_tensor.to("cpu")
cpu_tensor_3 = gpu_tensor.cpu()
```

2.1.2 Neural Network Layers

PyTorch also has a large library of deep learning layers. These layers allow you to operate at a higher-level of abstraction than low-level code. For example, one of the most basic layers in deep learning modules are linear layers, which is a matrix multiplication followed by a vector addition. PyTorch has layers that handle this for you, and automatically create the parameter vectors that need to be learned.

Below is an example. Notice how we first create the layer, then we call the layer. Creating the layer is like creating the function, which you can then later call. When the layer is created, the weights for the matrix multiplication (and addition) are automatically created, and initialized automatically with random numbers.

[ ]:
```
test_input = torch.randn(3)

layer = nn.Linear(3, 2) # create the layer
output = layer(test_input) # call the layer

print('Weights of the Layer:')
print(layer.weight)
```

```
Weights of the Layer:
Parameter containing:
tensor([[-0.1839,  0.2116, -0.4551],
        [-0.5188,  0.0717,  0.1533]], requires_grad=True)
```

This allows us to chain layers together in order to create a neural network. For example, the below code creates a neural network similar to the previous homework. And the gradients can be easily calculated through back-propagation throughout all of the layers.

[ ]:
```
test_input = torch.randn(3)

layer1 = nn.Linear(3, 20)
layer2 = nn.ReLU()
layer3 = nn.Linear(20, 1)

out = layer1(test_input)
out = layer2(out)
out = layer3(out)
```

```
print(f'Result: {out.item()}')

out.backward()

print('Gradient to weights in layer 1:')
print(layer1.weight.grad)
```

```
Result: -0.14415761828422546
Gradient to weights in layer 1:
tensor([[ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [ 9.2860e-02,  4.4548e-04, -2.6681e-01],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [ 5.3429e-02,  2.5631e-04, -1.5352e-01],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [-1.6060e-03, -7.7045e-06,  4.6145e-03],
        [-2.9704e-02, -1.4250e-04,  8.5347e-02],
        [-1.8732e-02, -8.9861e-05,  5.3821e-02],
        [-1.8511e-02, -8.8803e-05,  5.3187e-02],
        [-8.6179e-02, -4.1343e-04,  2.4762e-01],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [-4.5856e-02, -2.1998e-04,  1.3176e-01],
        [-8.3285e-03, -3.9954e-05,  2.3930e-02],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [-6.5090e-02, -3.1226e-04,  1.8702e-01],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [ 0.0000e+00,  0.0000e+00, -0.0000e+00],
        [-2.2001e-03, -1.0554e-05,  6.3214e-03]])
```

Let's use this knowledge to create a neural network for object recognition.

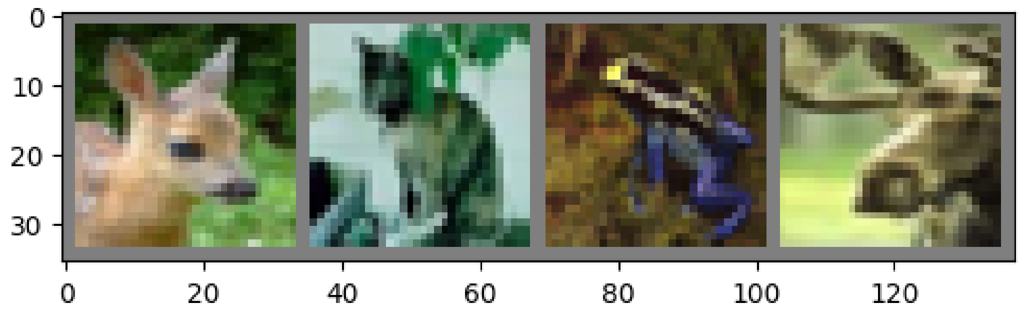
2.2 Loading Image Datasets

We are going to work with the CIFAR10 dataset, which is a small image dataset consisting of just ten object categories. Most datasets today are many orders of magnitude larger in size, but the smaller dataset will allow us to work on commodity computers. The code below will download both the train/test splits of the CIFAR10 dataset, and visualize some of the images.

[ ]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
```

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images))
```

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



2.3 Building the Neural Network

Create a convolutional neural network that classifies the category of the images in the CIFAR10 dataset. In the class below, there are two functions: `__init__` and `forward()`. In the constructor, instantiate the layers that you will need. In the `forward()` function, call these layers in order to run the neural network forwards.

Experiment with the below neural network layers to build a network that is able to classify the image: - nn.Conv2d() - nn.MaxPool2d() - nn.Linear() - nn.ReLU()

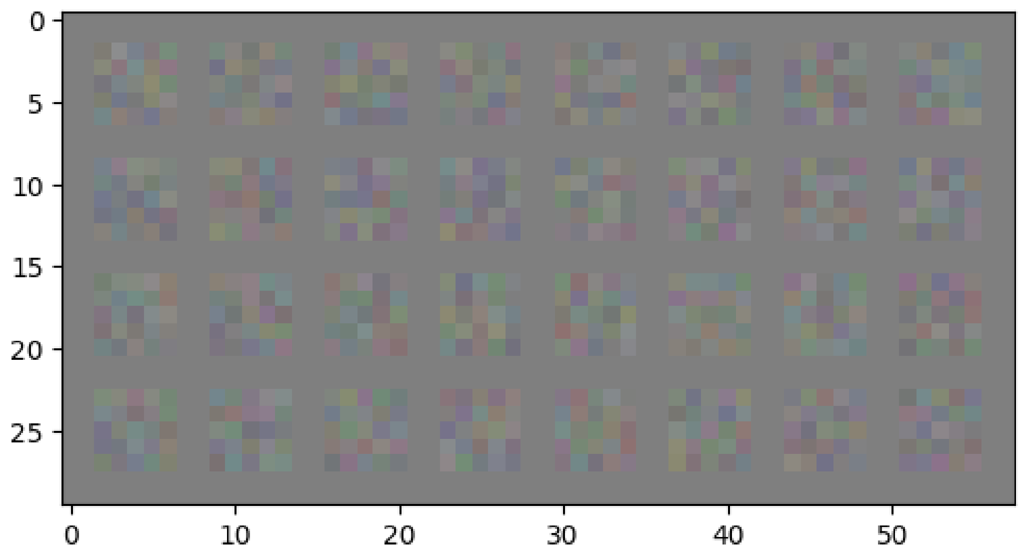
The input `x` will be an tensor of size `4x3x32x32`, which represents a batch of input images. The output should be a ten dimensional vector. You will most likely need to use other PyTorch operations as well, such as `torch.flatten()`. Feel free to use other layers and operations as you see fit.

We recommend first trying the following neural network: convolution, max pooling, convolution, max pooling, convolution, convolution, flattening, linear, linear, linear. Note that you should put the activation function in the right spots.

```
[ ]: class Net(nn.Module):
         def __init__(self):
             super().__init__()
             # TODO: Initialize network layers
             self.conv1 = nn.Conv2d(3, 32, 5)
             self.pool = nn.MaxPool2d(2, 2)
             self.conv2 = nn.Conv2d(32, 64, 5)
             self.conv3 = nn.Conv2d(64, 128, 5)
             self.fc1 = nn.Linear(128 * 1 * 1, 1024)
             self.fc2 = nn.Linear(1024, 512)
             self.fc3 = nn.Linear(512, 10)

         def forward(self, x):
             # TODO: Implement the forward pass with using the layers defined above
             #       and the proper activation functions
             x = self.pool(F.relu(self.conv1(x)))
             x = self.pool(F.relu(self.conv2(x)))
             x = F.relu(self.conv3(x))
             x = x.view(-1, 128 * 1 * 1)
             x = F.relu(self.fc1(x))
             x = F.relu(self.fc2(x))
             x = self.fc3(x)
             return x

     net = Net()
```

Before we proceed, let's visualize the weights of the first convolutional layer in the neural network. Modify the code below in order to plot the weights of the first convolutional layer. (You need to use `detach()` in order for this to work.)

```
[ ]: imshow(torchvision.utils.make_grid(net.conv1.weight.cpu().detach()))
```



2.4 Training the Network

Since the neural network is initialized with random noise, the filters visualized above are just random noise. In order to train them, we need to specify both a) a loss function and b) an optimization algorithm. We will use the cross entropy loss function with stochastic gradient descent. In PyTorch, we can specify these by creating the two objects below:

[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.0001)
```

Notice that the optimizer accepts the parameter `net.parameters()`. The call `net.parameters()` is a bit of magic. It will automatically determine which tensors are learnable inside the network, and pack them into a vector that is fed into the gradient descent method. In the previous homework, you needed to manually track these variables, but in PyTorch there is book-keeping underneath the API that does this for you automatically.

Now, we are ready to train the neural network.

[ ]:
```
def train_network(net, n_epochs=2):
    net.to(device)

    for epoch in range(n_epochs):  # loop over the dataset multiple times
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            inputs, labels = inputs.to(device), labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            if i % 1000 == 0:
                print(f'Epoch={epoch + 1} Iter={i + 1:5d} Loss={loss.item():.
↪3f}')
                running_loss = 0.0
    print('Finished Training')
    return net
```

If you want, you can train the network for longer too. This will help improve the performance.

[ ]:
```
train_network(net, n_epochs=8)
```

```
Epoch=1 Iter=    1 Loss=2.282
Epoch=1 Iter= 1001 Loss=1.547
Epoch=1 Iter= 2001 Loss=1.696
Epoch=1 Iter= 3001 Loss=1.544
Epoch=1 Iter= 4001 Loss=2.900
Epoch=1 Iter= 5001 Loss=1.321
Epoch=1 Iter= 6001 Loss=1.069
Epoch=1 Iter= 7001 Loss=1.908
Epoch=1 Iter= 8001 Loss=0.840
Epoch=1 Iter= 9001 Loss=2.004
Epoch=1 Iter=10001 Loss=1.539
Epoch=1 Iter=11001 Loss=1.209
Epoch=1 Iter=12001 Loss=1.565
Epoch=2 Iter=    1 Loss=1.564
Epoch=2 Iter= 1001 Loss=0.904
Epoch=2 Iter= 2001 Loss=0.697
Epoch=2 Iter= 3001 Loss=0.701
Epoch=2 Iter= 4001 Loss=0.713
Epoch=2 Iter= 5001 Loss=0.597
Epoch=2 Iter= 6001 Loss=1.981
Epoch=2 Iter= 7001 Loss=1.282
Epoch=2 Iter= 8001 Loss=1.209
Epoch=2 Iter= 9001 Loss=0.706
Epoch=2 Iter=10001 Loss=1.175
Epoch=2 Iter=11001 Loss=1.225
Epoch=2 Iter=12001 Loss=2.392
Epoch=3 Iter=    1 Loss=1.687
Epoch=3 Iter= 1001 Loss=1.417
```

```
Epoch=3 Iter= 2001 Loss=0.884
Epoch=3 Iter= 3001 Loss=1.912
Epoch=3 Iter= 4001 Loss=1.742
Epoch=3 Iter= 5001 Loss=0.882
Epoch=3 Iter= 6001 Loss=0.879
Epoch=3 Iter= 7001 Loss=1.221
Epoch=3 Iter= 8001 Loss=0.857
Epoch=3 Iter= 9001 Loss=0.703
Epoch=3 Iter=10001 Loss=1.027
Epoch=3 Iter=11001 Loss=0.547
Epoch=3 Iter=12001 Loss=1.318
Epoch=4 Iter=    1 Loss=0.927
Epoch=4 Iter= 1001 Loss=0.467
Epoch=4 Iter= 2001 Loss=0.376
Epoch=4 Iter= 3001 Loss=1.044
Epoch=4 Iter= 4001 Loss=0.182
Epoch=4 Iter= 5001 Loss=0.225
Epoch=4 Iter= 6001 Loss=0.327
Epoch=4 Iter= 7001 Loss=1.667
Epoch=4 Iter= 8001 Loss=0.007
Epoch=4 Iter= 9001 Loss=0.773
Epoch=4 Iter=10001 Loss=0.087
Epoch=4 Iter=11001 Loss=1.312
Epoch=4 Iter=12001 Loss=0.644
Epoch=5 Iter=    1 Loss=0.526
Epoch=5 Iter= 1001 Loss=0.119
Epoch=5 Iter= 2001 Loss=0.901
Epoch=5 Iter= 3001 Loss=0.145
Epoch=5 Iter= 4001 Loss=0.634
Epoch=5 Iter= 5001 Loss=0.107
Epoch=5 Iter= 6001 Loss=0.292
Epoch=5 Iter= 7001 Loss=0.813
Epoch=5 Iter= 8001 Loss=0.700
Epoch=5 Iter= 9001 Loss=0.412
Epoch=5 Iter=10001 Loss=0.711
Epoch=5 Iter=11001 Loss=0.182
Epoch=5 Iter=12001 Loss=0.239
Epoch=6 Iter=    1 Loss=0.319
Epoch=6 Iter= 1001 Loss=0.825
Epoch=6 Iter= 2001 Loss=1.674
Epoch=6 Iter= 3001 Loss=0.571
Epoch=6 Iter= 4001 Loss=0.877
Epoch=6 Iter= 5001 Loss=0.615
Epoch=6 Iter= 6001 Loss=0.702
Epoch=6 Iter= 7001 Loss=0.122
Epoch=6 Iter= 8001 Loss=0.628
Epoch=6 Iter= 9001 Loss=0.547
Epoch=6 Iter=10001 Loss=0.312
```

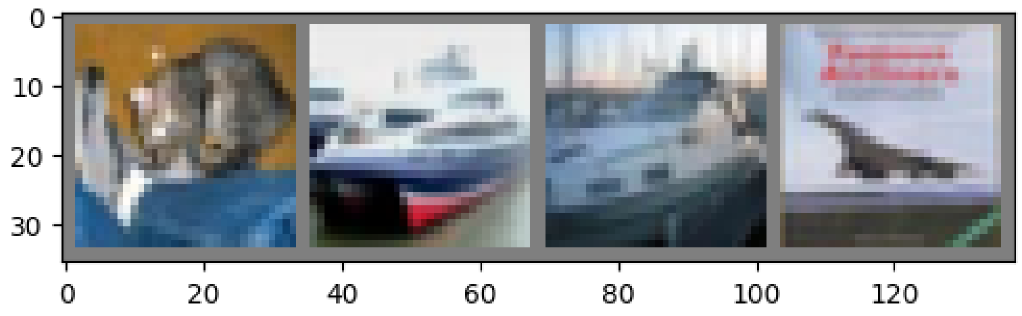
```
Epoch=6 Iter=11001 Loss=0.563
Epoch=6 Iter=12001 Loss=1.294
Epoch=7 Iter=    1 Loss=0.291
Epoch=7 Iter= 1001 Loss=0.090
Epoch=7 Iter= 2001 Loss=1.247
Epoch=7 Iter= 3001 Loss=0.071
Epoch=7 Iter= 4001 Loss=3.548
Epoch=7 Iter= 5001 Loss=0.773
Epoch=7 Iter= 6001 Loss=0.143
Epoch=7 Iter= 7001 Loss=0.378
Epoch=7 Iter= 8001 Loss=0.234
Epoch=7 Iter= 9001 Loss=0.739
Epoch=7 Iter=10001 Loss=1.042
Epoch=7 Iter=11001 Loss=0.333
Epoch=7 Iter=12001 Loss=0.042
Epoch=8 Iter=    1 Loss=0.940
Epoch=8 Iter= 1001 Loss=0.303
Epoch=8 Iter= 2001 Loss=0.697
Epoch=8 Iter= 3001 Loss=0.212
Epoch=8 Iter= 4001 Loss=0.024
Epoch=8 Iter= 5001 Loss=0.012
Epoch=8 Iter= 6001 Loss=0.412
Epoch=8 Iter= 7001 Loss=0.152
Epoch=8 Iter= 8001 Loss=0.362
Epoch=8 Iter= 9001 Loss=1.137
Epoch=8 Iter=10001 Loss=0.734
Epoch=8 Iter=11001 Loss=0.277
Epoch=8 Iter=12001 Loss=0.209
Finished Training
```

```
[ ]: Net(
       (conv1): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
       (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
     ceil_mode=False)
       (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
       (conv3): Conv2d(64, 128, kernel_size=(5, 5), stride=(1, 1))
       (fc1): Linear(in_features=128, out_features=1024, bias=True)
       (fc2): Linear(in_features=1024, out_features=512, bias=True)
       (fc3): Linear(in_features=512, out_features=10, bias=True)
     )
```

2.5 Visualizing Predictions

Unless you train the neural network for a long time, the loss will most likely not go to zero. However, it should still go down, which means it has learned some association between visual patterns and the category labels in the dataset. Let's try the model on some images in the test set and see what it predicts for them.

```
[ ]: dataiter = iter(testloader)
     images, labels = next(dataiter)

     net.to(device)
     predictions = net(images.to(device)).argmax(axis=1).cpu().detach()
     accuracy = (labels==predictions).double().mean()

     # print images
     imshow(torchvision.utils.make_grid(images))
     print('GroundTruth: ', '\t'.join(f'{classes[labels[j]]:5s}' for j in range(4)))
     print('Predictions: ', '\t'.join(f'{classes[predictions[j]]:5s}' for j in␣
      ↪range(4)))
     print(f'Accuracy: {accuracy*100}%')
```



```
GroundTruth:  cat       ship    ship    plane
Predictions:  cat       ship    ship    plane
Accuracy: 100.0%
```

In our implementation, the predictions are not always correct, but they are often reasonable. This is impressive considering we have barely trained the neural network at all.

Let's calculate the accuracy on the full test set.

```
[ ]: dataiter = iter(testloader)

     running_accuracy = 0
     running_count = 0
     for images, labels in dataiter:
       images = images.to(device)
       predictions = net(images.to(device)).argmax(axis=1).cpu().detach()
       accuracy = (labels==predictions).double().mean()

       running_accuracy += accuracy
       running_count += 1

     print(f'Accuracy: {running_accuracy/running_count*100:.2f}%')
```

```
Accuracy: 71.64%
```

In our solution, we get 74% accuracy after training for about 10 minutes on the Colab GPU. Can you do better?

3 2. Problem 2: Building ResNet

3.1 Residual Network

Residual networks have become a standard architecture because they are able to efficiently scale to a large number of layers. While the state-of-the-art networks have thousands of layers, they would be too expensive to train in time for the homework deadline. Let's implement just a simple residual network.

Implement a ResNet block which contains convolutional layers with a skip connection across them. Note that the dimensions of the original input and the output of the convolutional layers may not match up for addition. Hint: one way to address this is to introduce a linear transformation (like a 1x1 kernel convolution) to resize the input when necessary. Another would be to 0-pad the input to match dimensions for addition.

[ ]:
```
class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        # TODO: Initialize Two Convolutional Layers in the Residual Block
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride␣
↪= stride, padding = 1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride = 1, padding = 1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()


        self.skip_connection = nn.Identity()
        if stride != 1 or in_channels != out_channels:
          # TODO: Use a Conv2d layer with kernel_size=1 to "resize" input
          self.skip_connection = nn.Conv2d(in_channels, out_channels,␣
↪kernel_size = 1,
                                          stride = stride)

    def forward(self, x):
        identity_x = self.skip_connection(x)
        x = self.conv1(x)
        x = self.bn1(x)
        x= self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)

        # TODO: Implement Forward pass using 2 Conv Layers.
        # The addition of the original input is already done

        ##############################

        assert x.shape == identity_x.shape
        x = x + identity_x
        x = self.relu(x)
        return x
```

In the code block below, complete the class for a residual network. (PyTorch has a residual network built in, but you should not use it. Instead, create a residual network using the building blocks introduced above.) We recommend the following architecture: Convolution, Maxpooling, ResNetBlock, ResNetBlock, flatten, linear, linear, linear. Be sure to use ReLU activations where necessary.

[ ]:
```
class ResNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        # TODO: initialize network layers

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.res_block1 = ResNetBlock(64, 64, stride=1)
        self.res_block2 = ResNetBlock(64, 128, stride=2)

        self.flatten = nn.Flatten()

        self.linear1 = nn.Linear(8192, 512)
        self.relu2 = nn.ReLU()
        self.linear2 = nn.Linear(512, 128)
        self.relu3 = nn.ReLU()
        self.linear3 = nn.Linear(128, num_classes)

    def forward(self, x):
       # TODO: implement the forward pass
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.maxpool1(x)

        x = self.res_block1(x)
        x = self.res_block2(x)

        x = self.flatten(x)

        x = self.linear1(x)
        x = self.relu2(x)
        x = self.linear2(x)
        x = self.relu3(x)
        x = self.linear3(x)

        return x
```

[ ]:
```
res_net = ResNet()
```

[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(res_net.parameters(), lr=0.0001)
```

After you create the network, we can use the training loop from above in order to train it.

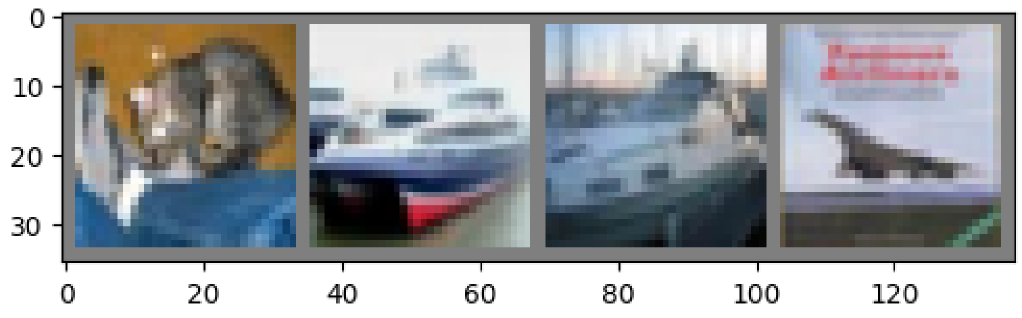
[ ]:
```
res_net = train_network(res_net, n_epochs=5)
```

```
Epoch=1 Iter=    1 Loss=2.300
Epoch=1 Iter= 1001 Loss=1.053
Epoch=1 Iter= 2001 Loss=0.851
Epoch=1 Iter= 3001 Loss=1.577
Epoch=1 Iter= 4001 Loss=0.643
Epoch=1 Iter= 5001 Loss=0.454
Epoch=1 Iter= 6001 Loss=0.491
Epoch=1 Iter= 7001 Loss=0.878
Epoch=1 Iter= 8001 Loss=1.140
Epoch=1 Iter= 9001 Loss=0.651
Epoch=1 Iter=10001 Loss=0.870
Epoch=1 Iter=11001 Loss=1.407
Epoch=1 Iter=12001 Loss=0.238
Epoch=2 Iter=    1 Loss=0.683
Epoch=2 Iter= 1001 Loss=0.179
Epoch=2 Iter= 2001 Loss=0.855
Epoch=2 Iter= 3001 Loss=1.065
Epoch=2 Iter= 4001 Loss=0.386
Epoch=2 Iter= 5001 Loss=1.488
Epoch=2 Iter= 6001 Loss=0.540
Epoch=2 Iter= 7001 Loss=1.490
Epoch=2 Iter= 8001 Loss=0.378
Epoch=2 Iter= 9001 Loss=0.065
Epoch=2 Iter=10001 Loss=1.551
Epoch=2 Iter=11001 Loss=0.147
Epoch=2 Iter=12001 Loss=1.597
Epoch=3 Iter=    1 Loss=0.220
Epoch=3 Iter= 1001 Loss=0.618
Epoch=3 Iter= 2001 Loss=0.263
Epoch=3 Iter= 3001 Loss=1.090
Epoch=3 Iter= 4001 Loss=0.214
```

```
Epoch=3 Iter= 5001 Loss=1.449
Epoch=3 Iter= 6001 Loss=1.161
Epoch=3 Iter= 7001 Loss=0.770
Epoch=3 Iter= 8001 Loss=0.680
Epoch=3 Iter= 9001 Loss=0.436
Epoch=3 Iter=10001 Loss=0.501
Epoch=3 Iter=11001 Loss=0.463
Epoch=3 Iter=12001 Loss=0.330
Epoch=4 Iter=    1 Loss=0.222
Epoch=4 Iter= 1001 Loss=0.942
Epoch=4 Iter= 2001 Loss=1.190
Epoch=4 Iter= 3001 Loss=0.849
Epoch=4 Iter= 4001 Loss=0.305
Epoch=4 Iter= 5001 Loss=1.295
Epoch=4 Iter= 6001 Loss=0.083
Epoch=4 Iter= 7001 Loss=0.076
Epoch=4 Iter= 8001 Loss=0.027
Epoch=4 Iter= 9001 Loss=0.182
Epoch=4 Iter=10001 Loss=0.104
Epoch=4 Iter=11001 Loss=0.373
Epoch=4 Iter=12001 Loss=0.018
Epoch=5 Iter=    1 Loss=0.005
Epoch=5 Iter= 1001 Loss=0.360
Epoch=5 Iter= 2001 Loss=0.081
Epoch=5 Iter= 3001 Loss=0.003
Epoch=5 Iter= 4001 Loss=0.127
Epoch=5 Iter= 5001 Loss=0.019
Epoch=5 Iter= 6001 Loss=0.008
Epoch=5 Iter= 7001 Loss=0.089
Epoch=5 Iter= 8001 Loss=0.018
Epoch=5 Iter= 9001 Loss=0.036
Epoch=5 Iter=10001 Loss=0.250
Epoch=5 Iter=11001 Loss=0.115
Epoch=5 Iter=12001 Loss=0.042
Finished Training
```

Let’s visualize some of the predictions from the trained network.

```
[ ]: dataiter = iter(testloader)
     images, labels = next(dataiter)

     res_net.to(device)
     predictions = res_net(images.to(device)).argmax(axis=1).cpu().detach()
     accuracy = (labels==predictions).double().mean()

     # print images
     imshow(torchvision.utils.make_grid(images))
     print('GroundTruth: ', '\t'.join(f'{classes[labels[j]]:5s}' for j in range(4)))
     print('Predictions: ', '\t'.join(f'{classes[predictions[j]]:5s}' for j in␣
       ↪range(4)))
     print(f'Accuracy: {accuracy*100}%')
```



```
GroundTruth:  cat       ship    ship    plane
Predictions:  cat       ship    ship    plane
Accuracy: 100.0%
```

Let's also calculate the accuracy on the full test set.

[ ]:
```
dataiter = iter(testloader)

running_accuracy = 0
running_count = 0
for images, labels in dataiter:
  images = images.to(device)
  predictions = res_net(images.to(device)).argmax(axis=1).cpu().detach()
  accuracy = (labels==predictions).double().mean()

  running_accuracy += accuracy
  running_count += 1

print(f'Accuracy: {running_accuracy/running_count*100:.2f}%')
```

```
Accuracy: 78.00%
```

In our solution, we get 76% accuracy.

[2]:
```
%%capture
from google.colab import drive
drive.mount('/content/drive');
!sudo apt-get update;
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
  ↪texlive-plain-generic;
!jupyter nbconvert --to pdf /content/drive/MyDrive/Co;
```