Homework 10: ResNet

In this homework you will be building a simplified ResNet model to classify images from the Cifar-10 dataset. You will need to change the runtime type to GPU to train in a reasonable amount of time on this homework.

Note that google limits your GPU usage somewhat, so be efficient with your use! The final training run will take around half an hour. We recommend "babysitting" it: if it looks like its producing different results than our output that we provide to you, stop it and go try to debug.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn

import torchvision
from torchvision import transforms, datasets
# Make sure to change your runtime type to GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Download and prepare the data

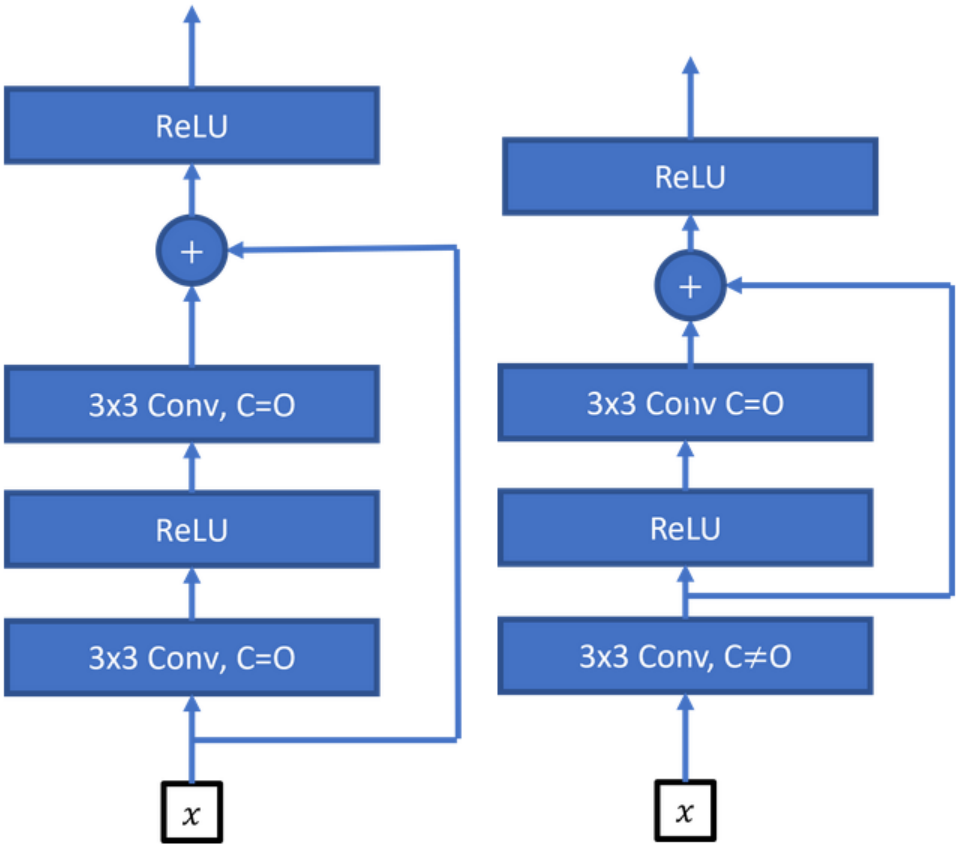
```
best_acc = 0  # best test accuracy
start_epoch = 0  # start from epoch 0 or last checkpoint epoch

# DO NOT CHANGE
print('==> Preparing data..')
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=128, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=100, shuffle=False, num_workers=2)

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

```
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified
```

1.a Define a convolutional block and simplified ResNet model (80 pts)

Fill in the forward passes for the basic block and resnet modules below. The module should have two different behaviors depending on whether `in_channels == out_channels`. If `in_channels==out_channels`, then the module should wrap two conv layers in a

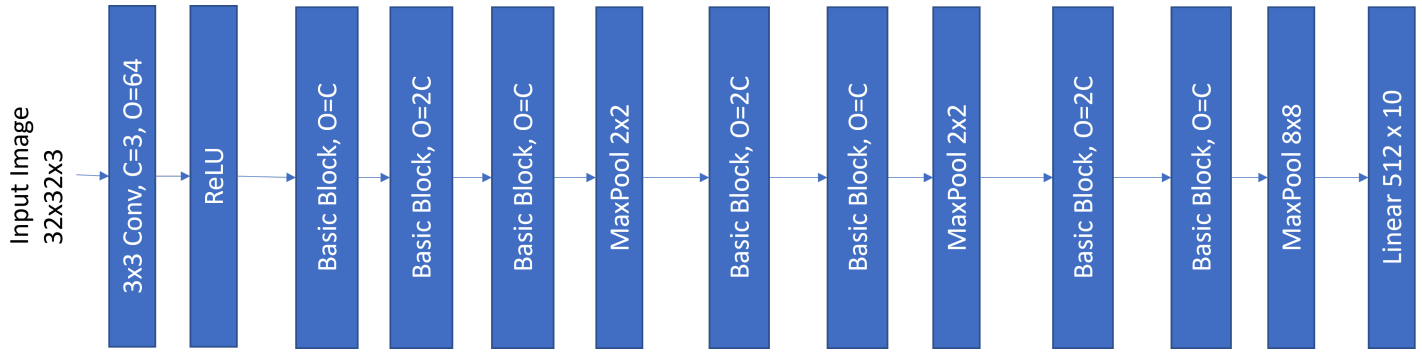
residual connection as shown on the left below. Otherwise, the module should only wrap the second conv layer in a residual connection as shown on the right below.



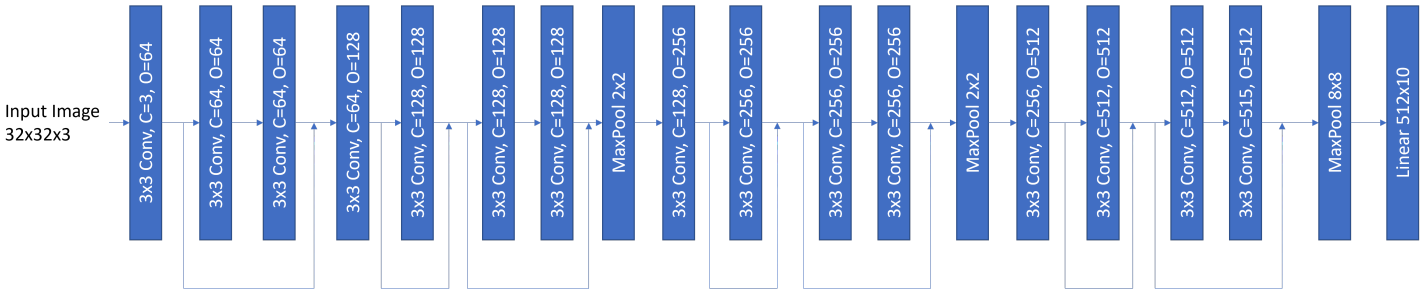

```
class BasicBlock(nn.Module):

    def __init__(self, in_channels, out_channels):
        super(BasicBlock, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.conv1 = nn.Conv2d(
            in_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)

    def forward(self, x):
        '''
        Pass the input through the layers as depicted in the diagram above
        If the input channels are the same as the output channels, add a residual connection before the fin
        Note that you will need to change the the location of the residual connection depending on the inpu
        '''
        #ADD CODE HERE

        if self.in_channels != self.out_channels:
            match_conv = nn.Conv2d(self.in_channels, self.out_channels, kernel_size=1, stride=1, bias=False
            residual = match_conv(x)
        else:
            residual = x


        out = self.conv1(x)
        out = F.relu(out)

        out = self.conv2(out)
        out += residual

        out = F.relu(out)
```

```
        return out
```

The Resnet18 model will be connected as shown in the following diagrams. The first diagram below shows the connections in terms of the "BasicBlock" module you implemented above. The second diagram shows the connections with all the conv layers included, but omits ReLUs to simplify the diagram a bit.



Model Architecture (ReLUs omitted)



```
class SimplifiedResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(SimplifiedResNet, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.layer1 = self._make_layer(block, 64, 64)
        self.layer2 = self._make_layer(block, 64, 128)
        self.layer3 = self._make_layer(block, 128, 256)
        self.layer4 = self._make_layer(block, 256, 512)
        self.linear = nn.Linear(512, num_classes)

    def _make_layer(self, block, in_planes, out_planes):
        layers = [
            block(in_planes, out_planes),
            block(out_planes, out_planes)
        ]
        return nn.Sequential(*layers)

    def forward(self, x):
        '''
        Pass the input through each of the layers above as shown in the diagram.
        The make layer function already defines how many blocks of each kind so you only need to call each functi
        You will also need to call F.relu and F.avg_pool2d for the layers not defined above.
        '''
        # ADD CODE HERE

        x = self.conv1(x)
        x = F.relu(x)

        x = self.layer1(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        x = self.layer2(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        x = self.layer3(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        x = self.layer4(x)
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        x = F.avg_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = self.linear(x)

        return x
```

Now, we initialize your model, and define the optimizer. We will use SGD with momentum. The `weight_decay` option also adds a small amount of L2 regularization.

```
print('==> Building model..')
net = SimplifiedResNet(BasicBlock, [2, 2, 2, 2])
net = net.to(device)
if device == 'cuda':
    net = torch.nn.DataParallel(net)
    cudnn.benchmark = True

optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
```

```
==> Building model..
```

1.b Finish train and test functions in `train_epoch` and `test` (20 pts)

```
def train_epoch(epoch, net, optimizer, trainloader):
    '''
    Function should iterate through the trainloader, pass the inputs through the network, compute the loss given
    call backward() on the loss, and step the optimizer.

    arguments:
      epoch: current epoch, integer.
      net: torch.nn.Module object for convnet to train.
      optimizer: optimizer class for training.
      trainloader: pytorch DataLoader for the training set

    no return value
    '''
    print('\nEpoch: %d' % epoch)
    # net.train() below is good practice when performing testing or evaluation.
    # In this case, it has no effect, but there are more complicated networks
    # for which the architecture is deliberately different during training vs evaluation.
    net.train()
    # use the crossentropyloss as your loss function.
    criterion = nn.CrossEntropyLoss()
    # ADD CODE HERE

    loss = 0.0
    total = 0
    correct = 0


    for batch_idx, (inputs, targets) in enumerate(trainloader):
      inputs, targets = inputs.to(device), targets.to(device)

      optimizer.zero_grad()

      outputs = net(inputs)
      curr_loss = criterion(outputs, targets)
      curr_loss.backward()
      optimizer.step()

      loss += curr_loss.item()
      _, predicted = outputs.max(1)
      total += targets.size(0)
      correct += predicted.eq(targets).sum().item()
```

Your test function should loop through the test data and return the % accuracy of the model.

```
from contextlib import AsyncContextDecorator
def test(net, testloader):
    '''
    Function should iterate through testloader and return a variable acc that is the total test accuracy score fo
    arguments:
      net: torch.nn.Module object for convnet to evaluate.
      testloader: pytorch DataLoader for the testing set.

    returns:
      float value from 0 to 100 holding the accuracy on the test set for this model.
    '''
    # net.eval() below is good practice when performing testing or evaluation.
    # In this case, it has no effect, but there are more complicated networks
    # for which the architecture is deliberately different during training vs evaluation.
    net.eval()
    # ADD CODE HERE

    correct = 0
    total = 0

    with torch.no_grad():
      for images, labels in testloader:
        images, labels = images.to(device), labels.to(device)

        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    acc = 100. * correct / total


    return acc
```

1.c Train the model (10 pts)

For each epoch, train the model, call the test function to get the accuracy score for the epoch, append that accuracy to the list `accs`, and step the scheduler to update the learning rate. Running through 30 epochs should take ~30 minutes and your final accuracy should be ~89%.

We have left the output of our implementation in the cell below **DO NOT WAIT 30 minutes to find out you did not obtain high accuracy: if your output looks very different from ours in the early epochs, stop your training run early and go debug!**

Note that more advanced optimization schemes (or even just running longer with a simple scheme like this one) can improve this accuracy.

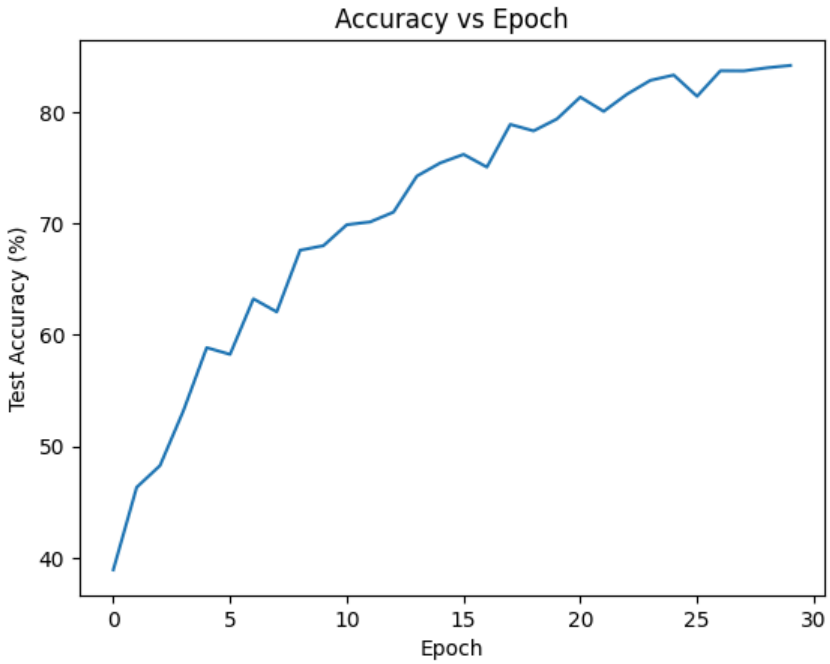
```
accs = []
for epoch in range(30):
    # ADD CODE HERE
    # it is recommended to print out the Test accuracy after each epoch so that you can compare your progress
    # to the reference output.

    train_epoch(epoch, net, optimizer, trainloader)

    accuracy = test(net, testloader)
    accs.append(accuracy)

    print(f'Test accuracy: {accuracy:.2f}')
```

```
Epoch: 0
Test accuracy: 38.92

Epoch: 1
```

```
Test accuracy: 46.32

Epoch: 2
Test accuracy: 48.27

Epoch: 3
Test accuracy: 53.18

Epoch: 4
Test accuracy: 58.85

Epoch: 5
Test accuracy: 58.25

Epoch: 6
Test accuracy: 63.23

Epoch: 7
Test accuracy: 62.06

Epoch: 8
Test accuracy: 67.59

Epoch: 9
Test accuracy: 68.00

Epoch: 10
Test accuracy: 69.88

Epoch: 11
Test accuracy: 70.13

Epoch: 12
Test accuracy: 71.01

Epoch: 13
Test accuracy: 74.24

Epoch: 14
Test accuracy: 75.42

Epoch: 15
Test accuracy: 76.19

Epoch: 16
Test accuracy: 75.05

Epoch: 17
Test accuracy: 78.88

Epoch: 18
Test accuracy: 78.30
```

1.d: Plot the accuracy over time (10 pts)

```
import matplotlib.pyplot as plt
import numpy as np
# Produce a plot of the test accuracy on the y-axis and epoch count on the x-axis.
# ADD CODE HERE
epochs = np.arange(len(accs))

plt.plot(epochs, accs)
plt.title('Accuracy vs Epoch')
plt.xlabel('Epoch')
plt.ylabel('Test Accuracy (%)')

plt.show()
```

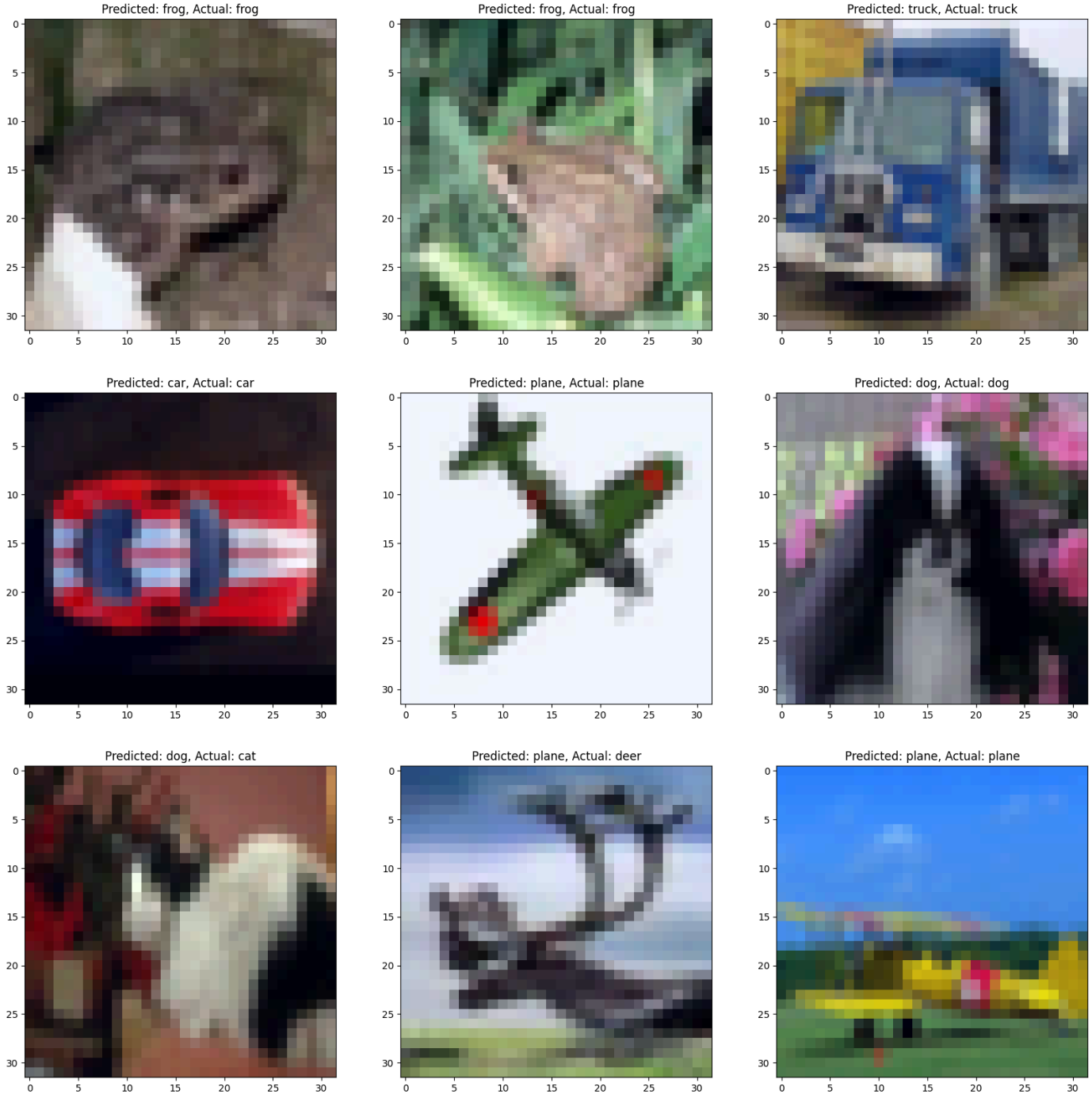


Now, we get the predicted labels for a batch of the testing data and plot 9 of the images with their labels. From looking at the pictures you can get an idea of the difficulty of this task. One anecdotal human-level classification accuracy is only 94% (http://karpathy.github.io/2011/04/27/manually-classifying-cifar10/), which is about the same (actually slightly worse) than what a real non-simplified ResNet model would achieve.

```
iterator = iter(testloader)
x_batch, y_batch = next(iterator)

with torch.no_grad():
    inputs, targets = x_batch.to(device),y_batch.to(device)
    outputs = net(inputs)
    _,predicted = outputs.max(1)
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was called. os.fork() is incor
  self.pid = os.fork()
```

```
fig, axs = plt.subplots(3,3,figsize = (20,20))

inds = np.random.randint(1,inputs.shape[0],size = 9)
count = 0

for i in np.arange(3):
  for j in np.arange(3):
    image = x_batch[inds[count]].reshape(3,32,32).numpy()
    image = image.transpose(1,2,0)
    image -= image.min()
    image /= image.max()
    axs[i,j].imshow(image)
    title = "Predicted: " + classes[predicted[inds[count]]] + ", Actual: " + classes[y_batch[inds[count]]]
    axs[i,j].title.set_text(title)
    count += 1
```


Predicted: frog, Actual: frog
Predicted: frog, Actual: frog
Predicted: truck, Actual: truck
Predicted: car, Actual: car
Predicted: plane, Actual: plane
Predicted: dog, Actual: dog
Predicted: dog, Actual: cat
Predicted: plane, Actual: deer
Predicted: plane, Actual: plane