CIFAR-10_Transfer_learning

January 28, 2026

1 Introduction

In this project, you will build a neural network of your own design to evaluate the CIFAR-10 dataset. Our target accuracy is 70%, but any accuracy over 50% is a great start. Some of the benchmark results on CIFAR-10 include:

78.9% Accuracy | Deep Belief Networks; Krizhevsky, 2010

90.6% Accuracy | Maxout Networks; Goodfellow et al., 2013

96.0% Accuracy | Wide Residual Networks; Zagoruyko et al., 2016

99.0% Accuracy | GPipe; Huang et al., 2018

98.5% Accuracy | Rethinking Recurrent Neural Networks and other Improvements for ImageClassification; Nguyen et al., 2020

Research with this dataset is ongoing. Notably, many of these networks are quite large and quite expensive to train.

1.1 Imports

[2]:
```
## This cell contains the essential imports you will need - DO NOT CHANGE THE␣
 ↪CONTENTS! ##
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import time
```

1.2 Load the Dataset

Specify your transforms as a list first. The transforms module is already loaded as `transforms`.

CIFAR-10 is fortunately included in the torchvision module. Then, you can create your dataset using the `CIFAR10` object from `torchvision.datasets` (the documentation is available here). Make sure to specify `download=True`!

Once your dataset is created, you'll also need to define a `DataLoader` from the `torch.utils.data` module for both the train and the test set.

[4]:
```
import torch
from torchvision import datasets, transforms

# Load your dataset (e.g., CIFAR10)
train_dataset = datasets.CIFAR10(root='./data', train=True, download=True,
                                 transform=transforms.ToTensor())

# Calculate the mean and standard deviation
mean = 0.0
std = 0.0
num_samples = len(train_dataset)

for images, labels in train_dataset:
    # Calculate mean and std for each image
    mean += images.mean(dim=(1, 2))
    std += images.std(dim=(1, 2))

# Average across all images
mean /= num_samples
std /= num_samples
print('mean = ', mean)
print('std deviation = ', std)


# Create a Normalize transform
normalize = transforms.Normalize(mean=mean, std=std)

# Apply the transform to your DataLoader
train_loader = torch.utils.data.DataLoader(
    datasets.CIFAR10(root='./data', train=True, download=True,
                     transform=transforms.Compose([
                         transforms.ToTensor(),
                         normalize
                     ])),
    batch_size=64, shuffle=True)
```

```
Files already downloaded and verified
mean =  tensor([0.4914, 0.4822, 0.4465])
std deviation =  tensor([0.2023, 0.1994, 0.2010])
Files already downloaded and verified
```

[5]:
```
#data_dir = '~/.pytorch/CIFAR10/'

# Define transforms
## YOUR CODE HERE ##
```

```
train_transforms = transforms.Compose([transforms.Resize(32),
                                       transforms.RandomRotation(15),
                                       transforms.RandomCrop(32, padding = 4),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ColorJitter(brightness=0.2,␣
 ↪contrast=0.2, saturation=0.2, hue=0.1),  # Random color jitter
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.4914, 0.4822, 0.
 ↪4465), (0.2203, 0.1994, 0.2010))]) #normalization for RGB images

test_transforms = transforms.Compose([transforms.Resize(32),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.4914, 0.4822, 0.
 ↪4465), (0.2023, 0.1994, 0.2010))]) #normalization for RGB images



# Create training set and define training dataloader
## YOUR CODE HERE ##
train_data = datasets.CIFAR10(root='./data', train = True, download = True,␣
 ↪transform=train_transforms)
test_data = datasets.CIFAR10(root='./data', train = False, download = True,␣
 ↪transform=test_transforms)

# Create test set and define test dataloader
## YOUR CODE HERE ##
trainloader = torch.utils.data.DataLoader(train_data, batch_size=64,␣
 ↪shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=64)

# The 10 classes in the dataset
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck')
```

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

1.3 Explore the Dataset

Using matplotlib, numpy, and torch, explore the dimensions of your data.

You can view images using the `show5` function defined below – it takes a data loader as an argument. Remember that normalized images will look really weird to you! You may want to try changing your transforms to view images. Typically using no transforms other than `toTensor()` works well for viewing – but not as well for training your network. If `show5` doesn't work, go back and check your code for creating your data loaders and your training/test sets.

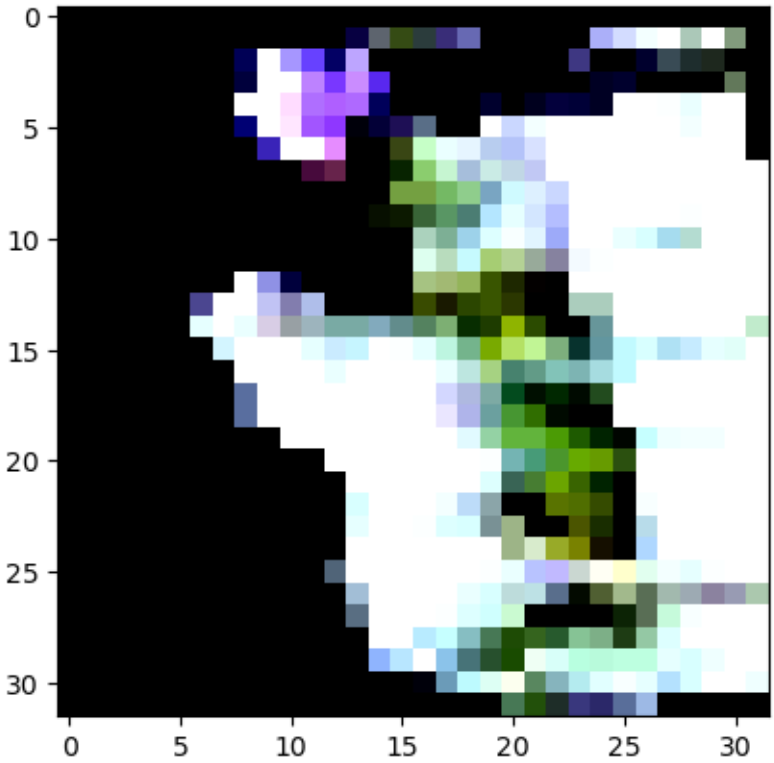
[7]:
```
num_samples = len(train_data)

# Get the shape of a single image (CIFAR-10 images are 32x32 with 3 color␣
 ↪channels)
image, label = train_data[0]
image_shape = image.shape

# Print the results
print(f"Number of training samples: {num_samples}")
print(f"Number of test samples: {len(test_data)}")
print(f"Shape of a single image: {image_shape}")
print('len trainloader ', len(trainloader))
print('len test ', len(testloader))

def show5(img_loader):
    dataiter = iter(img_loader)

    batch = next(dataiter)
    labels = batch[1][0:5]
    images = batch[0][0:5]
    for i in range(5):
        print(classes[labels[i]])

        image = images[i].numpy()
        plt.imshow(image.T)
        plt.show()
```

```
Number of training samples: 50000
Number of test samples: 10000
Shape of a single image: torch.Size([3, 32, 32])
len trainloader  782
len test  157
```

[8]:
```
# Explore data
## YOUR CODE HERE ##

show5(trainloader)
```

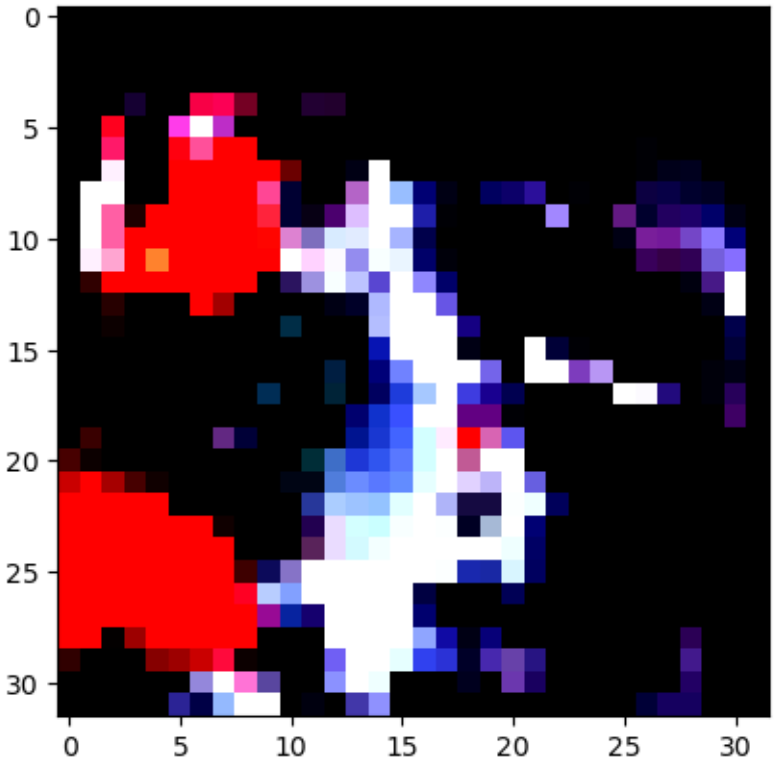
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
horse
```



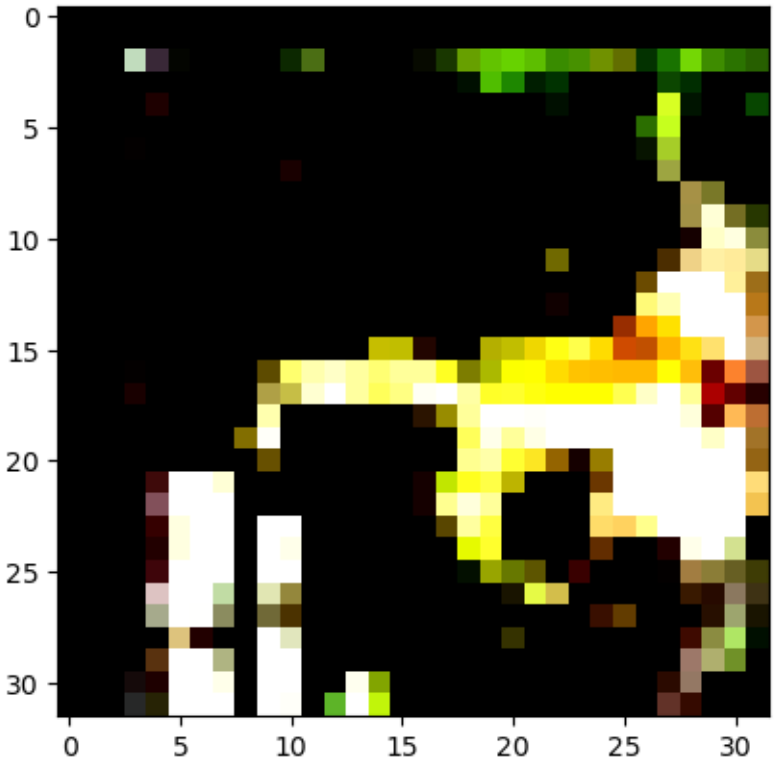

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
car
```



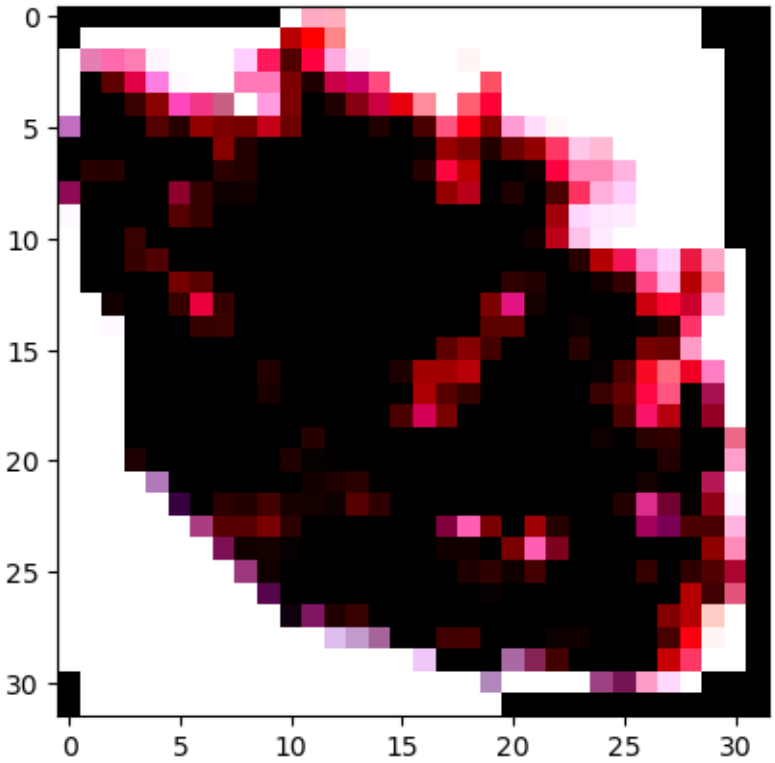

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
cat
```



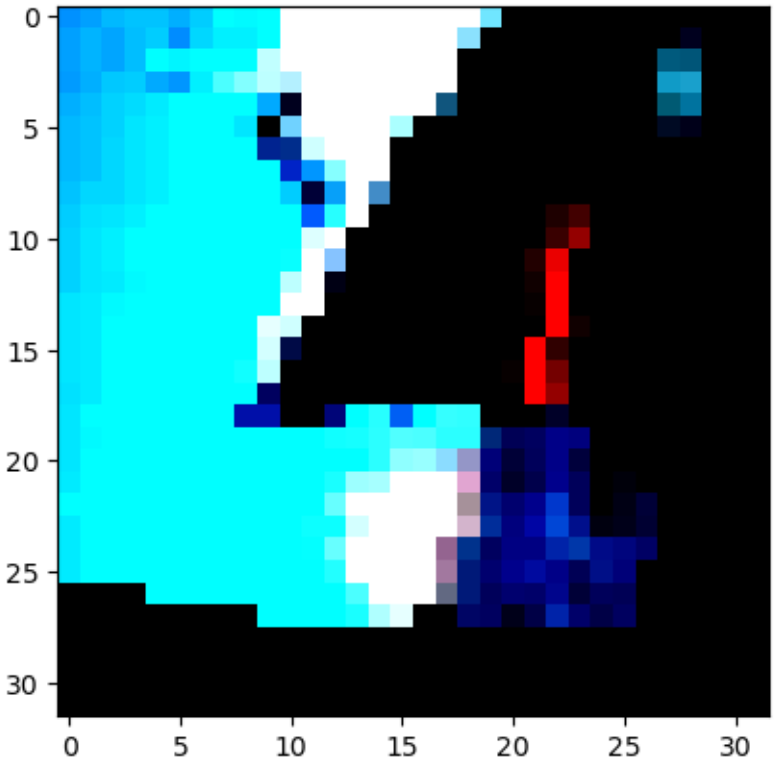

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
frog
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
ship
```



1.4 Build your Neural Network

Using the layers in `torch.nn` (which has been imported as `nn`) and the `torch.nn.functional` module (imported as `F`), construct a neural network based on the parameters of the dataset. Feel free to construct a model of any architecture – feedforward, convolutional, or even something more advanced!

[10]:
```
import torchvision.models as models

#model = models.densenet121(pretrained=True)
#model = models.resnet18(pretrained=True)
#model = models.mobilenet_v2(pretrained=True)
#model = models.inception_v3(pretrained=True)
#model = models.wide_resnet50_2(pretrained=True)
model = models.resnext50_32x4d(pretrained=True)
#model = models.shufflenet_v2_x1_0(pretrained=True) # focus on speed

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


num_ftrs = model.fc.in_features
```

```
model.fc = torch.nn.Linear(num_ftrs, 10)

#model
```

```
C:\Users\Cristi\AppData\Roaming\Python\Python311\site-
packages\torchvision\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(
C:\Users\Cristi\AppData\Roaming\Python\Python311\site-
packages\torchvision\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 behavior is equivalent to passing
`weights=ResNeXt50_32X4D_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNeXt50_32X4D_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

1.5 NIVIDIA CUDA acceleration

[12]:
```
print('Do we have NVIDIA CUDA acceleration?', torch.cuda.is_available())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print (device)
torch.cuda.empty_cache()


import subprocess

def get_gpu_info():
    try:
        # Run the `nvidia-smi` command and capture its output
        result = subprocess.run(['nvidia-smi', '--query-gpu=name',␣
 ↪'--format=csv,noheader'],
                                stdout=subprocess.PIPE, stderr=subprocess.PIPE,␣
 ↪universal_newlines=True)

        # Check if the command was successful
        if result.returncode == 0:
            gpu_name = result.stdout.strip()  # Get the name of the GPU
            return gpu_name
        else:
            return "No NVIDIA GPU found or `nvidia-smi` not available."
    except FileNotFoundError:
        return "nvidia-smi not found. Please ensure you have the NVIDIA drivers␣
 ↪installed."

# Example usage:
```

```
gpu_name = get_gpu_info()
print(f"Your NVIDIA GPU: {gpu_name}")
```

```
Do we have NVIDIA CUDA acceleration? True
cuda
Your NVIDIA GPU: NVIDIA GeForce RTX 3050 Ti Laptop GPU
```

Specify a loss function and an optimizer, and instantiate the model.

If you use a less common loss function, please note why you chose that loss function in a comment.

[14]:
```
## YOUR CODE HERE ##

#criterion = nn.NLLLoss()
criterion = nn.CrossEntropyLoss()
optimizer  = optim.Adam(model.parameters(), lr=0.001)
#optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9)
#optimizer = optim.RMSprop(model.parameters(), lr=0.001, alpha=0.99)
#optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.01)
#optimizer = optim.Adagrad(model.parameters(), lr=0.01)
```

1.6 Running your Neural Network

Use whatever method you like to train your neural network, and ensure you record the average loss at each epoch. Don't forget to use `torch.device()` and the `.to()` method for both your model and your data if you are using GPU!

If you want to print your loss during each epoch, you can use the `enumerate` function and print the loss after a set number of batches. 250 batches works well for most people!

[42]:
```
from torch import nn, optim
import torchvision.models as models
import torch

# Load the pretrained Inception V3 model
model = models.inception_v3(pretrained=True)

# Adjust the final fully connected layer to output 2 classes (cats vs dogs)
num_ftrs = model.fc.in_features
model.fc = torch.nn.Linear(num_ftrs, 2)


# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Set number of epochs
num_epochs = 30
train_losses = []
```

```
test_losses = []

for epoch in range(num_epochs):
    running_loss = 0
    start_time = time.time()

    # Set model to training mode
    model.train()

    # Training loop
    for images, labels in trainloader:
        # Move data to the device (GPU/CPU)
        images, labels = images.to(device), labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs, aux_outputs = model(images)  # Inception outputs two outputs
        loss = criterion(outputs, labels) + 0.4 * criterion(aux_outputs,␣
↪labels)  # Combine main and auxiliary loss

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    # Evaluation loop (testing data)
    test_loss = 0
    accuracy = 0
    end_time = time.time()

    print(f'Time to train Epoch {epoch+1}/{num_epochs} on {device}:␣
↪{(end_time-start_time):.2f} seconds')

    # Turn off gradients for evaluation, saves memory and computations
    with torch.no_grad():
        model.eval()  # Set model to evaluation mode
        for images, labels in testloader:
            images, labels = images.to(device), labels.to(device)

            # Forward pass for evaluation (use only the main output for testing)
            outputs = model(images)
            test_loss += criterion(outputs, labels).item()

            # Calculate accuracy
```

```
            _, top_class = outputs.topk(1, dim=1)
            equals = top_class == labels.view(*top_class.shape)
            accuracy += torch.mean(equals.type(torch.FloatTensor)).item()  #␣
↪Accuracy calculation

    # Append losses for tracking
    train_losses.append(running_loss / len(trainloader))
    test_losses.append(test_loss / len(testloader))

    # Timing information for evaluation
    finish_time = time.time()
    print(f'Time to evaluate Epoch {epoch+1}/{num_epochs} on {device}:␣
↪{(finish_time - end_time):.2f} seconds')

    # Print epoch summary
    print(f"Epoch: {epoch+1}/{num_epochs}.. "
          f"Training Loss: {train_losses[-1]:.3f}.. "
          f"Test Loss: {test_losses[-1]:.3f}.. "
          f"Test Accuracy: {accuracy / len(testloader):.3f}")
```

```
Time to train Epoch 1/70 on cuda: 111.39 seconds
Time to evaluate Epoch 1/70 on cuda: 6.48 seconds
Epoch: 1/70..  Average Training Loss: 0.347..  Average Test Loss: 0.459..
Average Test Accuracy: 86.216
Time to train Epoch 2/70 on cuda: 106.27 seconds
Time to evaluate Epoch 2/70 on cuda: 6.44 seconds
Epoch: 2/70..  Average Training Loss: 0.225..  Average Test Loss: 1.024..
Average Test Accuracy: 87.172
Time to train Epoch 3/70 on cuda: 112.30 seconds
Time to evaluate Epoch 3/70 on cuda: 6.46 seconds
Epoch: 3/70..  Average Training Loss: 0.243..  Average Test Loss: 0.740..
Average Test Accuracy: 87.978
Time to train Epoch 4/70 on cuda: 112.97 seconds
Time to evaluate Epoch 4/70 on cuda: 6.38 seconds
Epoch: 4/70..  Average Training Loss: 0.222..  Average Test Loss: 0.386..
Average Test Accuracy: 87.908
Time to train Epoch 5/70 on cuda: 109.28 seconds
Time to evaluate Epoch 5/70 on cuda: 7.16 seconds
Epoch: 5/70..  Average Training Loss: 0.229..  Average Test Loss: 0.392..
Average Test Accuracy: 87.838
Time to train Epoch 6/70 on cuda: 108.12 seconds
Time to evaluate Epoch 6/70 on cuda: 6.43 seconds
Epoch: 6/70..  Average Training Loss: 0.228..  Average Test Loss: 0.392..
Average Test Accuracy: 88.057
Time to train Epoch 7/70 on cuda: 110.53 seconds
Time to evaluate Epoch 7/70 on cuda: 6.38 seconds
Epoch: 7/70..  Average Training Loss: 0.226..  Average Test Loss: 0.383..
```

```
Average Test Accuracy: 88.535
Time to train Epoch 8/70 on cuda: 110.96 seconds
Time to evaluate Epoch 8/70 on cuda: 6.37 seconds
Epoch: 8/70..  Average Training Loss: 0.224..  Average Test Loss: 0.466..
Average Test Accuracy: 87.440
Time to train Epoch 9/70 on cuda: 106.95 seconds
Time to evaluate Epoch 9/70 on cuda: 6.57 seconds
Epoch: 9/70..  Average Training Loss: 0.228..  Average Test Loss: 0.397..
Average Test Accuracy: 87.898
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[42], line 17
     14 optimizer.zero_grad()
     16 # Forward pass
---> 17 outputs = model(images)
     18 loss = criterion(outputs, labels)
     20 # Backward pass and optimization

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py
 ↪1553, in Module._wrapped_call_impl(self, *args, **kwargs)
   1551     return self._compiled_call_impl(*args, **kwargs)  # type: ␣
 ↪ignore[misc]
   1552 else:
-> 1553     return self._call_impl(*args, **kwargs)

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py
 ↪1562, in Module._call_impl(self, *args, **kwargs)
   1557 # If we don't have any hooks, we want to skip the rest of the logic in
   1558 # this function, and just call forward.
   1559 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1560         or _global_backward_pre_hooks or _global_backward_hooks
   1561         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1562     return forward_call(*args, **kwargs)
   1564 try:
   1565     result = None

File ~\AppData\Roaming\Python\Python311\site-packages\torchvision\models\resnet
 ↪py:285, in ResNet.forward(self, x)
    284 def forward(self, x: Tensor) -> Tensor:
--> 285     return self._forward_impl(x)

File ~\AppData\Roaming\Python\Python311\site-packages\torchvision\models\resnet
 ↪py:275, in ResNet._forward_impl(self, x)
    273 x = self.layer1(x)
    274 x = self.layer2(x)
```

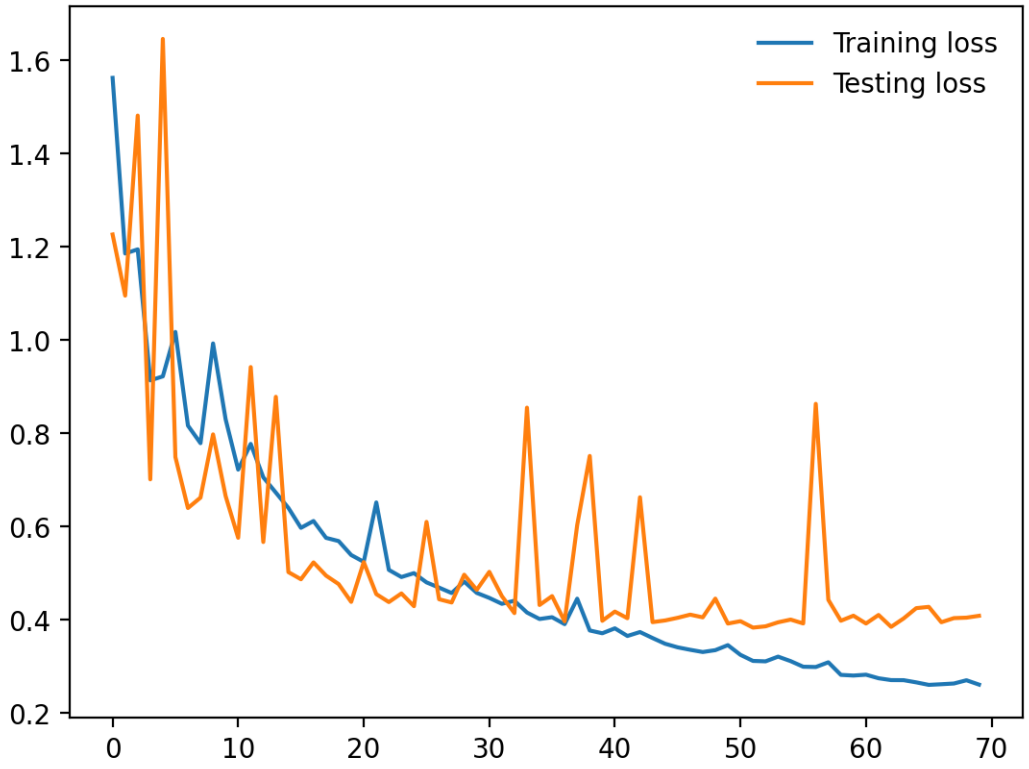
```
--> 275 x = self.layer3(x)
    276 x = self.layer4(x)
    278 x = self.avgpool(x)

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py:
 ↪1553, in Module._wrapped_call_impl(self, *args, **kwargs)
   1551     return self._compiled_call_impl(*args, **kwargs)  # type:␣
 ↪ignore[misc]
   1552 else:
-> 1553     return self._call_impl(*args, **kwargs)

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py:
 ↪1562, in Module._call_impl(self, *args, **kwargs)
   1557 # If we don't have any hooks, we want to skip the rest of the logic in
   1558 # this function, and just call forward.
   1559 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1560         or _global_backward_pre_hooks or _global_backward_hooks
   1561         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1562     return forward_call(*args, **kwargs)
   1564 try:
   1565     result = None

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\container.
 ↪py:219, in Sequential.forward(self, input)
    217 def forward(self, input):
    218     for module in self:
--> 219         input = module(input)
    220     return input

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py:
 ↪1553, in Module._wrapped_call_impl(self, *args, **kwargs)
   1551     return self._compiled_call_impl(*args, **kwargs)  # type:␣
 ↪ignore[misc]
   1552 else:
-> 1553     return self._call_impl(*args, **kwargs)

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py:
 ↪1562, in Module._call_impl(self, *args, **kwargs)
   1557 # If we don't have any hooks, we want to skip the rest of the logic in
   1558 # this function, and just call forward.
   1559 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1560         or _global_backward_pre_hooks or _global_backward_hooks
   1561         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1562     return forward_call(*args, **kwargs)
   1564 try:
   1565     result = None
```

```
File ~\AppData\Roaming\Python\Python311\site-packages\torchvision\models\resnet
 ↪py:148, in Bottleneck.forward(self, x)
    146 out = self.conv1(x)
    147 out = self.bn1(out)
--> 148 out = self.relu(out)
    150 out = self.conv2(out)
    151 out = self.bn2(out)

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py:
 ↪1553, in Module._wrapped_call_impl(self, *args, **kwargs)
   1551     return self._compiled_call_impl(*args, **kwargs)  # type:␣
 ↪ignore[misc]
   1552 else:
-> 1553     return self._call_impl(*args, **kwargs)

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\module.py:
 ↪1562, in Module._call_impl(self, *args, **kwargs)
   1557 # If we don't have any hooks, we want to skip the rest of the logic in
   1558 # this function, and just call forward.
   1559 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1560         or _global_backward_pre_hooks or _global_backward_hooks
   1561         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1562     return forward_call(*args, **kwargs)
   1564 try:
   1565     result = None

File␣
 ↪~\AppData\Roaming\Python\Python311\site-packages\torch\nn\modules\activation.
 ↪py:104, in ReLU.forward(self, input)
    103 def forward(self, input: Tensor) -> Tensor:
--> 104     return F.relu(input, inplace=self.inplace)

File ~\AppData\Roaming\Python\Python311\site-packages\torch\nn\functional.py:
 ↪1498, in relu(input, inplace)
   1496     return handle_torch_function(relu, (input,), input, inplace=inplace
   1497 if inplace:
-> 1498     result = torch.relu_(input)
   1499 else:
   1500     result = torch.relu(input)

KeyboardInterrupt:
```

Plot the training loss (and validation loss/accuracy, if recorded).

[18]:
```
## YOUR CODE HERE ##
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import matplotlib.pyplot as plt

plt.plot(train_losses, label='Training loss')
plt.plot(test_losses, label='Testing loss')
plt.legend(frameon=False)
```

[18]:
```
<matplotlib.legend.Legend at 0x1c9d4dc9650>
```



1.7 Testing your model

Using the previously created `DataLoader` for the test set, compute the percentage of correct predictions using the highest probability prediction.

If your accuracy is over 70%, great work! This is a hard task to exceed 70% on.

If your accuracy is under 45%, you'll need to make improvements. Go back and check your model architecture, loss function, and optimizer to make sure they're appropriate for an image classification task.

[20]:
```
## TEST YOUR MODEL HERE ##

#dataiter = iter(testloader)
#images, labels = next(dataiter)
model.eval()
correct = 0
total = 0

# Disable gradient calculation for inference
with torch.no_grad():
    for images, labels in testloader:  # Assuming test_loader is your␣
 ↪DataLoader for the test set
        # Move images and labels to the appropriate device (CPU or GPU)
        images, labels = images.to(device), labels.to(device)

        # Get model predictions
        outputs = model(images)
        ps = torch.exp(outputs)
        top_probs, top_class = ps.topk(1, dim=1)  # dim=1 to get top k for each␣
 ↪row (each image)
        #equals = top_class == labels.view(*top_class.shape)
        top_class = top_class.view(1, -1)

        total += labels.size(0)  # Update total number of samples
        correct += (top_class.view(1, -1) == labels).sum().item()  # Count␣
 ↪correct predictions
        #print('top_class shape:',top_class.shape)
        #print('labels shape', labels.shape)

       # print ('top_class \n', top_class)
        #print('labels\n', labels)

# Calculate accuracy
accuracy = 100 * correct / total
print(f'Accuracy of the model on the test set: {accuracy:.2f}%')
```

```
Accuracy of the model on the test set: 86.17%
```

1.8 Saving your model

Using `torch.save`, save your model for future loading.

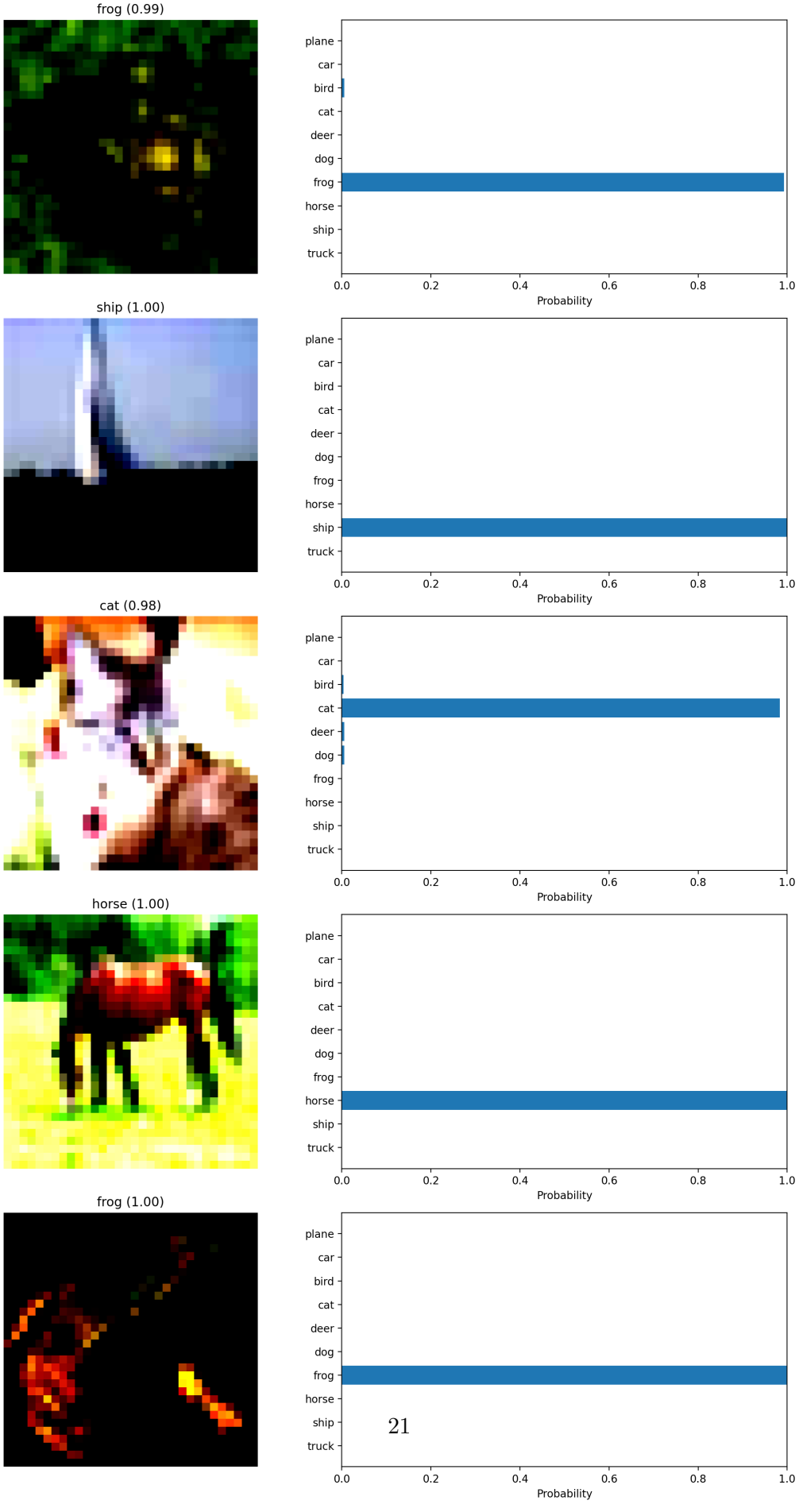
[22]:
```
## YOUR CODE HERE ##
torch.save(model.state_dict(), 'modelCIFAR10.pth')
```

1.9 Plotting some predictions!

[24]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import numpy as np
import torch
import matplotlib.pyplot as plt

# Check if CUDA is available and move the model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Function to get a random batch of images from the loader
def get_random_batch(loader):
    """ Get a random batch of images from the loader """
    dataiter = iter(loader)
    images, labels = next(dataiter)

    # Move images and labels to the GPU
    return images.to(device), labels.to(device)

# Get a random batch from the test loader
images, labels = get_random_batch(testloader)

# Select 5 random images from the batch
indices = torch.randperm(images.size(0))[:5]  # Randomly select 5 indices
random_images = images[indices]
random_labels = labels[indices]

# Calculate the class probabilities (softmax) for the selected images
model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    outputs = model(random_images)

# Convert output logits to probabilities
probs = torch.softmax(outputs, dim=1).cpu()

def view_classify(imgs, probs, classes):
    ''' Function for viewing 5 images, their predicted classes, and the␣
 ↪probability chart.
    '''
    fig, axes = plt.subplots(5, 2, figsize=(12, 20))  # 5 rows, 2 columns for␣
 ↪images and charts

    for idx in range(5):
        img = imgs[idx].cpu().numpy().transpose((1, 2, 0))
        prob = probs[idx].numpy()
```

```
        # Display image
        axes[idx, 0].imshow(img)
        axes[idx, 0].axis('off')

        # Get the class with the highest probability
        pred_class = classes[prob.argmax()]
        prob_str = f"{pred_class} ({prob.max():.2f})"
        axes[idx, 0].set_title(prob_str)

        # Display the probability chart
        axes[idx, 1].barh(np.arange(10), prob)
        axes[idx, 1].set_yticks(np.arange(10))
        axes[idx, 1].set_yticklabels(classes)
        axes[idx, 1].set_xlim(0, 1)
        axes[idx, 1].set_xlabel('Probability')
        axes[idx, 1].invert_yaxis()  # Highest probability at the top

    plt.tight_layout()

# Define CIFAR-10 class names
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']

# Plot the 5 randomly selected images and their predicted classes with␣
 ↪probability charts
view_classify(random_images, probs, classes)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```


frog (0.99)
ship (1.00)
cat (0.98)
horse (1.00)
frog (1.00)
plane
car
bird
cat
deer
dog
frog
horse
ship
truck
0.0
0.2
0.4
0.6
0.8
1.0
Probability

1.10 Make a Recommendation

Based on your evaluation, what is your recommendation on whether to build or buy? Explain your reasoning below.

1.10.1 Double click this cell to modify it