final_25

March 26, 2023

[1]:
```
#!pip install torch torchvision
%matplotlib inline
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
```

Prepare for Dataset

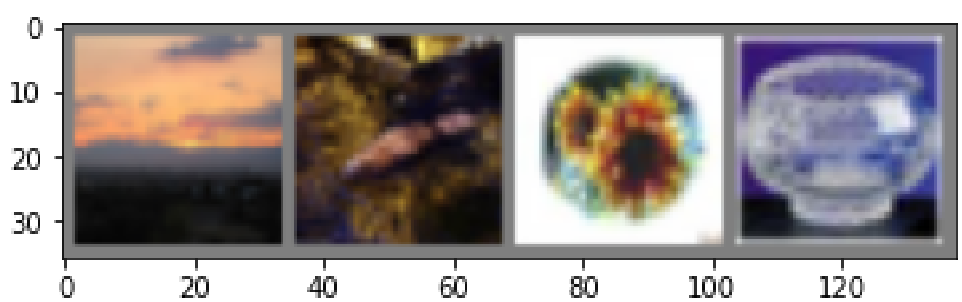
[2]:
```
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

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

trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR100(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('apple', 'aquarium_fish', 'baby', 'bear', 'beaver',
           'bed', 'bee', 'beetle', 'bicycle', 'bottle', 'bowl',
           'boy', 'bridge', 'bus', 'butterfly', 'camel', 'can',
           'castle', 'caterpillar', 'cattle', 'chair',
           'chimpanzee', 'clock', 'cloud', 'cockroach', 'couch',
```

```
           'crab', 'crocodile', 'cup', 'dinosaur', 'dolphin',
           'elephant', 'flatfish', 'forest', 'fox', 'girl',
           'hamster', 'house', 'kangaroo', 'keyboard', 'lamp',
           'lawn_mower', 'leopard', 'lion', 'lizard', 'lobster',
           'man', 'maple_tree', 'motorcycle', 'mountain', 'mouse',
           'mushroom', 'oak_tree', 'orange', 'orchid', 'otter',
           'palm_tree', 'pear', 'pickup_truck', 'pine_tree', 'plain',
           'plate', 'poppy', 'porcupine', 'possum', 'rabbit', 'raccoon',
           'ray', 'road', 'rocket', 'rose', 'sea', 'seal', 'shark',
           'shrew', 'skunk', 'skyscraper', 'snail', 'snake', 'spider',
           'squirrel', 'streetcar', 'sunflower', 'sweet_pepper', 'table',
           'tank', 'telephone', 'television', 'tiger', 'tractor', 'train',
           'trout', 'tulip', 'turtle', 'wardrobe', 'whale', 'willow_tree',
           'wolf', 'woman', 'worm')

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

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

[3]:

```
# The function to show an image.
def imshow(img):
    img = img / 2 + 0.5     # Unnormalize.
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# Get some random training images.
dataiter = iter(trainloader)
images, labels = next(dataiter)
# Show images.
imshow(torchvision.utils.make_grid(images))
# Print labels.
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```



```
cloud beaver sunflower  bowl
```

Choose a Device

```
[4]: # If there are GPUs, choose the first one for computing. Otherwise use CPU.
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
# If 'cuda:0' is printed, it means GPU is available.
```

```
cuda:0
```

Network Definition

```
[5]: class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3,␣
 ↪padding=1)
        self.bn1 = nn.BatchNorm2d(128)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(256)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.relu2 = nn.ReLU()
        self.conv3 = nn.Conv2d(256, 512, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(512)
        self.pool3 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.relu3 = nn.ReLU()
        self.conv4 = nn.Conv2d(512, 1024, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(1024)
        self.pool4 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.relu4 = nn.ReLU()
        self.fc1 = nn.Linear(1024 * 2 * 2, 2048)
        self.bn5 = nn.BatchNorm1d(2048)
        self.relu5 = nn.ReLU()
        self.fc2 = nn.Linear(2048, 1024)
        self.bn6 = nn.BatchNorm1d(1024)
        self.relu6 = nn.ReLU()
        self.fc3 = nn.Linear(1024, 100)

    def forward(self, x):
        x = self.pool1(self.relu1(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu2(self.bn2(self.conv2(x))))
        x = self.pool3(self.relu3(self.bn3(self.conv3(x))))
        x = self.pool4(self.relu4(self.bn4(self.conv4(x))))
        x = x.flatten(start_dim=1)
        x = self.relu5(self.bn5(self.fc1(x)))
        x = self.relu6(self.bn6(self.fc2(x)))
```

```
        x = self.fc3(x)
        return x

net = Net()      # Create the network instance.
net.to(device)  # Move the network parameters to the specified device.
```

[5]:
```
Net(
  (conv1): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool1): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (relu1): ReLU()
  (conv2): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (relu2): ReLU()
  (conv3): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool3): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (relu3): ReLU()
  (conv4): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool4): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (relu4): ReLU()
  (fc1): Linear(in_features=4096, out_features=2048, bias=True)
  (bn5): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu5): ReLU()
  (fc2): Linear(in_features=2048, out_features=1024, bias=True)
  (bn6): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu6): ReLU()
  (fc3): Linear(in_features=1024, out_features=100, bias=True)
)
```

Optimizer and Loss Function

[6]:
```
# We use cross-entropy as loss function.
loss_func = nn.CrossEntropyLoss()
# We use stochastic gradient descent (SGD) as optimizer.
#opt = optim.SGD(net.parameters(), lr=0.0001, momentum=0.9)
opt = optim.Adam(net.parameters(), lr=0.0001)
```

Training Procedure

[7]:
```python
import sys
from tqdm.notebook import tqdm

avg_losses = []   # Avg. losses.
epochs = 8        # Total epochs.
print_freq = 500  # Print frequency.

for epoch in range(epochs):  # Loop over the dataset multiple times.
    running_loss = 0.0       # Initialize running loss.
    for i, data in enumerate(tqdm(trainloader), 0):
        # Get the inputs.
        inputs, labels = data

        # Move the inputs to the specified device.
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients.
        opt.zero_grad()

        # Forward step.
        outputs = net(inputs)
        loss = loss_func(outputs, labels)

        # Backward step.
        loss.backward()

        # Optimization step (update the parameters).
        opt.step()

        # Print statistics.
        running_loss += loss.item()
        if i % print_freq == print_freq - 1: # Print every several mini-batches.
            avg_loss = running_loss / print_freq
            print('[epoch: {}, i: {:5d}] avg mini-batch loss: {:.3f}'.
↪format(epoch, i, avg_loss),flush=True)
            sys.stdout.flush()
            avg_losses.append(avg_loss)
            running_loss = 0.0

print('Finished Training.')
```

```
  0%|          | 0/12500 [00:00<?, ?it/s]

[epoch: 0, i:   499] avg mini-batch loss: 4.521
[epoch: 0, i:   999] avg mini-batch loss: 4.379
[epoch: 0, i:  1499] avg mini-batch loss: 4.295
[epoch: 0, i:  1999] avg mini-batch loss: 4.186
[epoch: 0, i:  2499] avg mini-batch loss: 4.200
```

```
[epoch: 0, i:  2999] avg mini-batch loss: 4.135
[epoch: 0, i:  3499] avg mini-batch loss: 4.042
[epoch: 0, i:  3999] avg mini-batch loss: 4.025
[epoch: 0, i:  4499] avg mini-batch loss: 3.955
[epoch: 0, i:  4999] avg mini-batch loss: 3.939
[epoch: 0, i:  5499] avg mini-batch loss: 3.890
[epoch: 0, i:  5999] avg mini-batch loss: 3.875
[epoch: 0, i:  6499] avg mini-batch loss: 3.895
[epoch: 0, i:  6999] avg mini-batch loss: 3.841
[epoch: 0, i:  7499] avg mini-batch loss: 3.732
[epoch: 0, i:  7999] avg mini-batch loss: 3.760
[epoch: 0, i:  8499] avg mini-batch loss: 3.680
[epoch: 0, i:  8999] avg mini-batch loss: 3.648
[epoch: 0, i:  9499] avg mini-batch loss: 3.653
[epoch: 0, i:  9999] avg mini-batch loss: 3.618
[epoch: 0, i: 10499] avg mini-batch loss: 3.567
[epoch: 0, i: 10999] avg mini-batch loss: 3.566
[epoch: 0, i: 11499] avg mini-batch loss: 3.575
[epoch: 0, i: 11999] avg mini-batch loss: 3.490
[epoch: 0, i: 12499] avg mini-batch loss: 3.472
```

```
  0%|          | 0/12500 [00:00<?, ?it/s]
```

```
[epoch: 1, i:   499] avg mini-batch loss: 3.371
[epoch: 1, i:   999] avg mini-batch loss: 3.334
[epoch: 1, i:  1499] avg mini-batch loss: 3.398
[epoch: 1, i:  1999] avg mini-batch loss: 3.410
[epoch: 1, i:  2499] avg mini-batch loss: 3.297
[epoch: 1, i:  2999] avg mini-batch loss: 3.324
[epoch: 1, i:  3499] avg mini-batch loss: 3.245
[epoch: 1, i:  3999] avg mini-batch loss: 3.279
[epoch: 1, i:  4499] avg mini-batch loss: 3.228
[epoch: 1, i:  4999] avg mini-batch loss: 3.222
[epoch: 1, i:  5499] avg mini-batch loss: 3.194
[epoch: 1, i:  5999] avg mini-batch loss: 3.213
[epoch: 1, i:  6499] avg mini-batch loss: 3.190
[epoch: 1, i:  6999] avg mini-batch loss: 3.144
[epoch: 1, i:  7499] avg mini-batch loss: 3.114
[epoch: 1, i:  7999] avg mini-batch loss: 3.169
[epoch: 1, i:  8499] avg mini-batch loss: 3.066
[epoch: 1, i:  8999] avg mini-batch loss: 3.091
[epoch: 1, i:  9499] avg mini-batch loss: 3.155
[epoch: 1, i:  9999] avg mini-batch loss: 3.095
[epoch: 1, i: 10499] avg mini-batch loss: 3.062
[epoch: 1, i: 10999] avg mini-batch loss: 3.036
[epoch: 1, i: 11499] avg mini-batch loss: 3.018
[epoch: 1, i: 11999] avg mini-batch loss: 3.049
[epoch: 1, i: 12499] avg mini-batch loss: 2.981
```

```
  0%|          | 0/12500 [00:00<?, ?it/s]

[epoch: 2, i:   499] avg mini-batch loss: 2.899
[epoch: 2, i:   999] avg mini-batch loss: 2.918
[epoch: 2, i:  1499] avg mini-batch loss: 2.915
[epoch: 2, i:  1999] avg mini-batch loss: 2.854
[epoch: 2, i:  2499] avg mini-batch loss: 2.887
[epoch: 2, i:  2999] avg mini-batch loss: 2.881
[epoch: 2, i:  3499] avg mini-batch loss: 2.834
[epoch: 2, i:  3999] avg mini-batch loss: 2.923
[epoch: 2, i:  4499] avg mini-batch loss: 2.874
[epoch: 2, i:  4999] avg mini-batch loss: 2.891

IOPub message rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_msg_rate_limit`.

Current values:
NotebookApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
NotebookApp.rate_limit_window=3.0 (secs)


[epoch: 4, i:  7999] avg mini-batch loss: 2.351
[epoch: 4, i:  8499] avg mini-batch loss: 2.342
[epoch: 4, i:  8999] avg mini-batch loss: 2.379
[epoch: 4, i:  9499] avg mini-batch loss: 2.392
[epoch: 4, i:  9999] avg mini-batch loss: 2.337
[epoch: 4, i: 10499] avg mini-batch loss: 2.368
[epoch: 4, i: 10999] avg mini-batch loss: 2.352
[epoch: 4, i: 11499] avg mini-batch loss: 2.371
[epoch: 4, i: 11999] avg mini-batch loss: 2.276
[epoch: 4, i: 12499] avg mini-batch loss: 2.304

  0%|          | 0/12500 [00:00<?, ?it/s]

[epoch: 5, i:   499] avg mini-batch loss: 2.174
[epoch: 5, i:   999] avg mini-batch loss: 2.202
[epoch: 5, i:  1499] avg mini-batch loss: 2.215
[epoch: 5, i:  1999] avg mini-batch loss: 2.134
[epoch: 5, i:  2499] avg mini-batch loss: 2.205
[epoch: 5, i:  2999] avg mini-batch loss: 2.199
[epoch: 5, i:  3499] avg mini-batch loss: 2.199
[epoch: 5, i:  3999] avg mini-batch loss: 2.261
[epoch: 5, i:  4499] avg mini-batch loss: 2.159
[epoch: 5, i:  4999] avg mini-batch loss: 2.185
[epoch: 5, i:  5499] avg mini-batch loss: 2.182
[epoch: 5, i:  5999] avg mini-batch loss: 2.252
[epoch: 5, i:  6499] avg mini-batch loss: 2.176
```

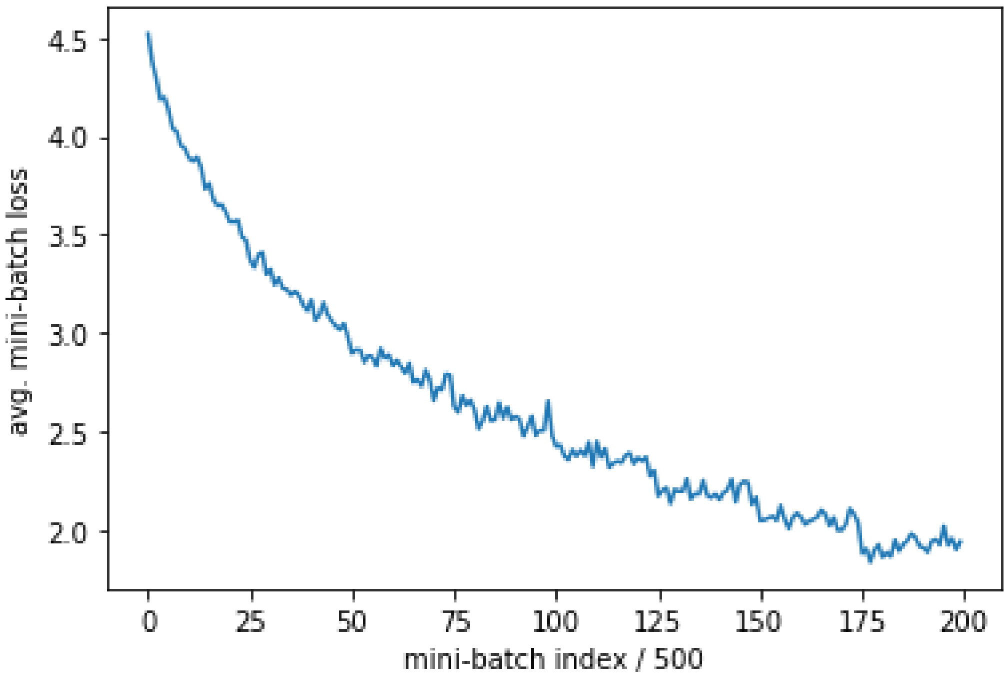
```
[epoch: 5, i:  6999] avg mini-batch loss: 2.165
[epoch: 5, i:  7499] avg mini-batch loss: 2.185
```

```
IOPub message rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_msg_rate_limit`.

Current values:
NotebookApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
NotebookApp.rate_limit_window=3.0 (secs)
```

```
[epoch: 7, i:  2999] avg mini-batch loss: 1.862
[epoch: 7, i:  3499] avg mini-batch loss: 1.885
[epoch: 7, i:  3999] avg mini-batch loss: 1.866
[epoch: 7, i:  4499] avg mini-batch loss: 1.950
[epoch: 7, i:  4999] avg mini-batch loss: 1.894
[epoch: 7, i:  5499] avg mini-batch loss: 1.927
[epoch: 7, i:  5999] avg mini-batch loss: 1.946
[epoch: 7, i:  6499] avg mini-batch loss: 1.982
[epoch: 7, i:  6999] avg mini-batch loss: 1.963
[epoch: 7, i:  7499] avg mini-batch loss: 1.919
[epoch: 7, i:  7999] avg mini-batch loss: 1.911
[epoch: 7, i:  8499] avg mini-batch loss: 1.888
[epoch: 7, i:  8999] avg mini-batch loss: 1.944
[epoch: 7, i:  9499] avg mini-batch loss: 1.952
[epoch: 7, i:  9999] avg mini-batch loss: 1.926
[epoch: 7, i: 10499] avg mini-batch loss: 2.021
[epoch: 7, i: 10999] avg mini-batch loss: 1.924
[epoch: 7, i: 11499] avg mini-batch loss: 1.957
[epoch: 7, i: 11999] avg mini-batch loss: 1.901
[epoch: 7, i: 12499] avg mini-batch loss: 1.941
Finished Training.
```

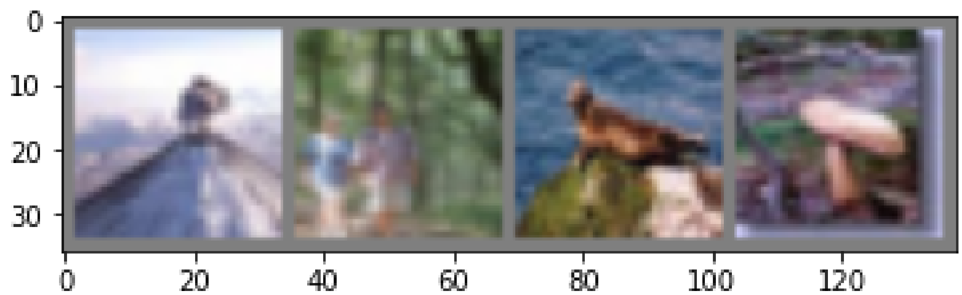
Training Loss Curve

[8]:
```
plt.plot(avg_losses)
plt.xlabel('mini-batch index / {}'.format(print_freq))
plt.ylabel('avg. mini-batch loss')
plt.show()
```



Evaluate on Test Dataset

[9]:
```
# Check several images.
dataiter = iter(testloader)
images, labels = next(dataiter)
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
outputs = net(images.to(device))
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```



```
GroundTruth:  mountain forest   seal mushroom
Predicted:  train rabbit dolphin mushroom
```

[10]:
```python
# Get test accuracy.
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        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()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 43 %
```

[11]:
```python
# Get test accuracy for each class.
class_correct = [0] * len(classes)
class_total = [0] * len(classes)
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(len(labels)):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(len(classes)):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of apple : 78 %
Accuracy of aquarium_fish : 60 %
Accuracy of  baby : 17 %
Accuracy of  bear : 12 %
Accuracy of beaver : 19 %
Accuracy of   bed : 44 %
Accuracy of   bee : 41 %
Accuracy of beetle : 41 %
Accuracy of bicycle : 67 %
```

```
Accuracy of bottle : 56 %
Accuracy of  bowl : 29 %
Accuracy of   boy : 35 %
Accuracy of bridge : 42 %
Accuracy of   bus : 42 %
Accuracy of butterfly : 33 %
Accuracy of camel : 37 %
Accuracy of   can : 37 %
Accuracy of castle : 56 %
Accuracy of caterpillar : 46 %
Accuracy of cattle : 37 %
Accuracy of chair : 79 %
Accuracy of chimpanzee : 61 %
Accuracy of clock : 43 %
Accuracy of cloud : 73 %
Accuracy of cockroach : 64 %
Accuracy of couch : 33 %
Accuracy of  crab : 42 %
Accuracy of crocodile : 25 %
Accuracy of   cup : 66 %
Accuracy of dinosaur : 39 %
Accuracy of dolphin : 47 %
Accuracy of elephant : 50 %
Accuracy of flatfish : 22 %
Accuracy of forest : 36 %
Accuracy of   fox : 38 %
Accuracy of  girl : 33 %
Accuracy of hamster : 49 %
Accuracy of house : 46 %
Accuracy of kangaroo : 26 %
Accuracy of keyboard : 59 %
Accuracy of  lamp : 33 %
Accuracy of lawn_mower : 62 %
Accuracy of leopard : 47 %
Accuracy of  lion : 45 %
Accuracy of lizard : 17 %
Accuracy of lobster : 22 %
Accuracy of   man :  5 %
Accuracy of maple_tree : 50 %
Accuracy of motorcycle : 69 %
Accuracy of mountain : 57 %
Accuracy of mouse : 18 %
Accuracy of mushroom : 40 %
Accuracy of oak_tree : 58 %
Accuracy of orange : 75 %
Accuracy of orchid : 68 %
Accuracy of otter :  9 %
Accuracy of palm_tree : 63 %
```

```
Accuracy of  pear : 50 %
Accuracy of pickup_truck : 59 %
Accuracy of pine_tree : 36 %
Accuracy of plain : 70 %
Accuracy of plate : 44 %
Accuracy of poppy : 70 %
Accuracy of porcupine : 38 %
Accuracy of possum : 15 %
Accuracy of rabbit : 15 %
Accuracy of raccoon : 32 %
Accuracy of   ray : 28 %
Accuracy of  road : 82 %
Accuracy of rocket : 63 %
Accuracy of  rose : 28 %
Accuracy of   sea : 61 %
Accuracy of  seal :  7 %
Accuracy of shark : 37 %
Accuracy of shrew : 36 %
Accuracy of skunk : 65 %
Accuracy of skyscraper : 57 %
Accuracy of snail : 26 %
Accuracy of snake : 30 %
Accuracy of spider : 44 %
Accuracy of squirrel : 13 %
Accuracy of streetcar : 50 %
Accuracy of sunflower : 67 %
Accuracy of sweet_pepper : 48 %
Accuracy of table : 21 %
Accuracy of  tank : 48 %
Accuracy of telephone : 56 %
Accuracy of television : 58 %
Accuracy of tiger : 43 %
Accuracy of tractor : 44 %
Accuracy of train : 50 %
Accuracy of trout : 51 %
Accuracy of tulip : 23 %
Accuracy of turtle : 21 %
Accuracy of wardrobe : 74 %
Accuracy of whale : 40 %
Accuracy of willow_tree : 38 %
Accuracy of  wolf : 46 %
Accuracy of woman : 21 %
Accuracy of  worm : 34 %
```

[12]:
```
# One of the changes I made was that I added another layer in the network
# that takes the output from the second convolutional layer, applies ReLU␣
↪activation,
```

```
# and passes it through a 2x2 AvgPool layer to capture more relationships.
# I also reduced the learning rate of the optimizer to half (0.0005) which
# which allowed the optimizer to take smaller steps towards the minimum of
# the loss function which might allow for a better chance of finding the global␣
↪min of loss.
```