final_5

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
```

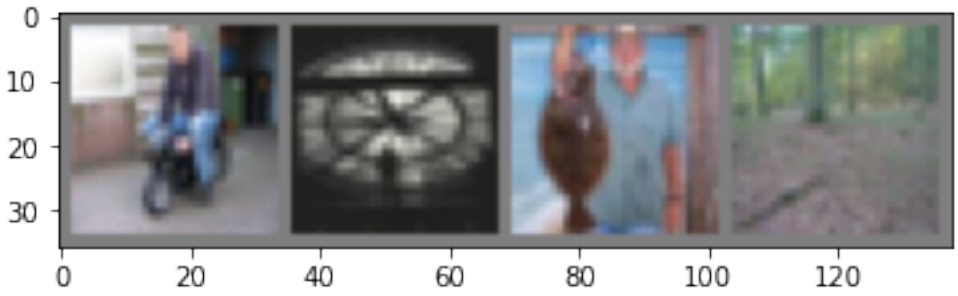
Prepare for Dataset

[2]:
```
transform = 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)))
```




```
motorcycle clock flatfish forest
```

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 This modified version of the Net class adds a fourth convolutional layer with 512 output channels, followed by a dropout layer with a dropout rate of 0.5. It also increases the size of the fully connected layer to 1024.

[5]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3,␣
 ↪padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.relu1 = nn.ELU()
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.relu2 = nn.ELU()
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.pool3 = nn.MaxPool2d(2, 2)
        self.relu3 = nn.ELU()
        self.fc1 = nn.Linear(256 * 4 * 4, 1024)
        self.bn4 = nn.BatchNorm1d(1024)
        self.relu4 = nn.ELU()
        self.fc2 = nn.Linear(1024, 100)

    def forward(self, x):
        x = self.pool1(self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = self.pool3(self.relu3(self.conv3(x)))
        x = x.view(-1, 256 * 4 * 4)
        x = self.relu4(self.fc1(x))
        x = self.fc2(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, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (relu1): ELU(alpha=1.0)
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (relu2): ELU(alpha=1.0)
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (relu3): ELU(alpha=1.0)
  (fc1): Linear(in_features=4096, out_features=1024, bias=True)
  (bn4): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu4): ELU(alpha=1.0)
  (fc2): 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.Adam(net.parameters(), lr=0.0005)

#scheduler = optim.lr_scheduler.StepLR(opt, step_size=2, gamma=0.1)
```

Training Procedure added learning rate scheduler

A learning rate scheduler adjusts the learning rate during training, based on a pre-defined schedule. The learning rate determines how quickly the model parameters are updated during training. If the learning rate is too high, the optimization process may overshoot the optimal parameters and fail to converge. If the learning rate is too low, the optimization process may take a long time to converge.

A learning rate scheduler can help address this problem by gradually decreasing the learning rate during training, allowing the model to converge more slowly and potentially to a better solution. There are different types of learning rate schedules, such as step decay, exponential decay, and cosine annealing, among others. Each type has its own formula for computing the learning rate at each epoch or iteration.

Another way to improve accuracy is by using learning rate scheduling. You can decrease the learning rate over time, so that the model can converge to a better minimum in the optimization landscape.

we define a StepLR scheduler with a step size of 2 epochs and a gamma of 0.1. This means that the learning rate will be multiplied by 0.1 every 2 epochs. We then use the scheduler to update the learning rate after each optimization step with the line scheduler.step().

[7]:
```
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()

        # Update the learning rate.
        #scheduler.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: 4.407
[epoch: 0, i:   999] avg mini-batch loss: 4.048
[epoch: 0, i:  1499] avg mini-batch loss: 3.750
[epoch: 0, i:  1999] avg mini-batch loss: 3.643
[epoch: 0, i:  2499] avg mini-batch loss: 3.531
[epoch: 0, i:  2999] avg mini-batch loss: 3.425
[epoch: 0, i:  3499] avg mini-batch loss: 3.325
[epoch: 0, i:  3999] avg mini-batch loss: 3.258
[epoch: 0, i:  4499] avg mini-batch loss: 3.170
[epoch: 0, i:  4999] avg mini-batch loss: 3.187
[epoch: 0, i:  5499] avg mini-batch loss: 3.136
[epoch: 0, i:  5999] avg mini-batch loss: 3.120
[epoch: 0, i:  6499] avg mini-batch loss: 3.099
[epoch: 0, i:  6999] avg mini-batch loss: 3.058
[epoch: 0, i:  7499] avg mini-batch loss: 2.996
[epoch: 0, i:  7999] avg mini-batch loss: 2.937
[epoch: 0, i:  8499] avg mini-batch loss: 2.906
[epoch: 0, i:  8999] avg mini-batch loss: 2.920
[epoch: 0, i:  9499] avg mini-batch loss: 2.912
[epoch: 0, i:  9999] avg mini-batch loss: 2.939
[epoch: 0, i: 10499] avg mini-batch loss: 2.958
[epoch: 0, i: 10999] avg mini-batch loss: 2.904
[epoch: 0, i: 11499] avg mini-batch loss: 2.818
[epoch: 0, i: 11999] avg mini-batch loss: 2.770
[epoch: 0, i: 12499] avg mini-batch loss: 2.815
  0%|          | 0/12500 [00:00<?, ?it/s]
[epoch: 1, i:   499] avg mini-batch loss: 2.446
[epoch: 1, i:   999] avg mini-batch loss: 2.509
[epoch: 1, i:  1499] avg mini-batch loss: 2.451
[epoch: 1, i:  1999] avg mini-batch loss: 2.551
[epoch: 1, i:  2499] avg mini-batch loss: 2.498
[epoch: 1, i:  2999] avg mini-batch loss: 2.485
[epoch: 1, i:  3499] avg mini-batch loss: 2.474
[epoch: 1, i:  3999] avg mini-batch loss: 2.501
[epoch: 1, i:  4499] avg mini-batch loss: 2.509
[epoch: 1, i:  4999] avg mini-batch loss: 2.574
[epoch: 1, i:  5499] avg mini-batch loss: 2.557
[epoch: 1, i:  5999] avg mini-batch loss: 2.537
[epoch: 1, i:  6499] avg mini-batch loss: 2.589
[epoch: 1, i:  6999] avg mini-batch loss: 2.550
[epoch: 1, i:  7499] avg mini-batch loss: 2.513
[epoch: 1, i:  7999] avg mini-batch loss: 2.479
[epoch: 1, i:  8499] avg mini-batch loss: 2.526
[epoch: 1, i:  8999] avg mini-batch loss: 2.611
[epoch: 1, i:  9499] avg mini-batch loss: 2.563
```

```
[epoch: 1, i:  9999] avg mini-batch loss: 2.545
[epoch: 1, i: 10499] avg mini-batch loss: 2.474
[epoch: 1, i: 10999] avg mini-batch loss: 2.422
[epoch: 1, i: 11499] avg mini-batch loss: 2.571
[epoch: 1, i: 11999] avg mini-batch loss: 2.524
[epoch: 1, i: 12499] avg mini-batch loss: 2.544
  0%|          | 0/12500 [00:00<?, ?it/s]
[epoch: 2, i:   499] avg mini-batch loss: 1.747
[epoch: 2, i:   999] avg mini-batch loss: 1.672
[epoch: 2, i:  1499] avg mini-batch loss: 1.690
[epoch: 2, i:  1999] avg mini-batch loss: 1.874
[epoch: 2, i:  2499] avg mini-batch loss: 1.902
[epoch: 2, i:  2999] avg mini-batch loss: 1.872
[epoch: 2, i:  3499] avg mini-batch loss: 1.859
[epoch: 2, i:  3999] avg mini-batch loss: 1.874
[epoch: 2, i:  4499] avg mini-batch loss: 1.834
[epoch: 2, i:  4999] avg mini-batch loss: 2.008
[epoch: 2, i:  5499] avg mini-batch loss: 1.935
[epoch: 2, i:  5999] avg mini-batch loss: 1.992
[epoch: 2, i:  6499] avg mini-batch loss: 2.025
[epoch: 2, i:  6999] avg mini-batch loss: 2.005
[epoch: 2, i:  7499] avg mini-batch loss: 1.985
[epoch: 2, i:  7999] avg mini-batch loss: 1.998
[epoch: 2, i:  8499] avg mini-batch loss: 2.048
[epoch: 2, i:  8999] avg mini-batch loss: 2.075
[epoch: 2, i:  9499] avg mini-batch loss: 2.085
[epoch: 2, i:  9999] avg mini-batch loss: 2.036
[epoch: 2, i: 10499] avg mini-batch loss: 2.070
[epoch: 2, i: 10999] avg mini-batch loss: 2.124
[epoch: 2, i: 11499] avg mini-batch loss: 2.075
[epoch: 2, i: 11999] avg mini-batch loss: 2.032
[epoch: 2, i: 12499] avg mini-batch loss: 2.027
  0%|          | 0/12500 [00:00<?, ?it/s]
[epoch: 3, i:   499] avg mini-batch loss: 0.994
[epoch: 3, i:   999] avg mini-batch loss: 1.031
[epoch: 3, i:  1499] avg mini-batch loss: 1.009
[epoch: 3, i:  1999] avg mini-batch loss: 1.055
[epoch: 3, i:  2499] avg mini-batch loss: 1.110
[epoch: 3, i:  2999] avg mini-batch loss: 1.203
[epoch: 3, i:  3499] avg mini-batch loss: 1.161
[epoch: 3, i:  3999] avg mini-batch loss: 1.236
[epoch: 3, i:  4499] avg mini-batch loss: 1.213
[epoch: 3, i:  4999] avg mini-batch loss: 1.258
[epoch: 3, i:  5499] avg mini-batch loss: 1.343
[epoch: 3, i:  5999] avg mini-batch loss: 1.317
[epoch: 3, i:  6499] avg mini-batch loss: 1.328
```

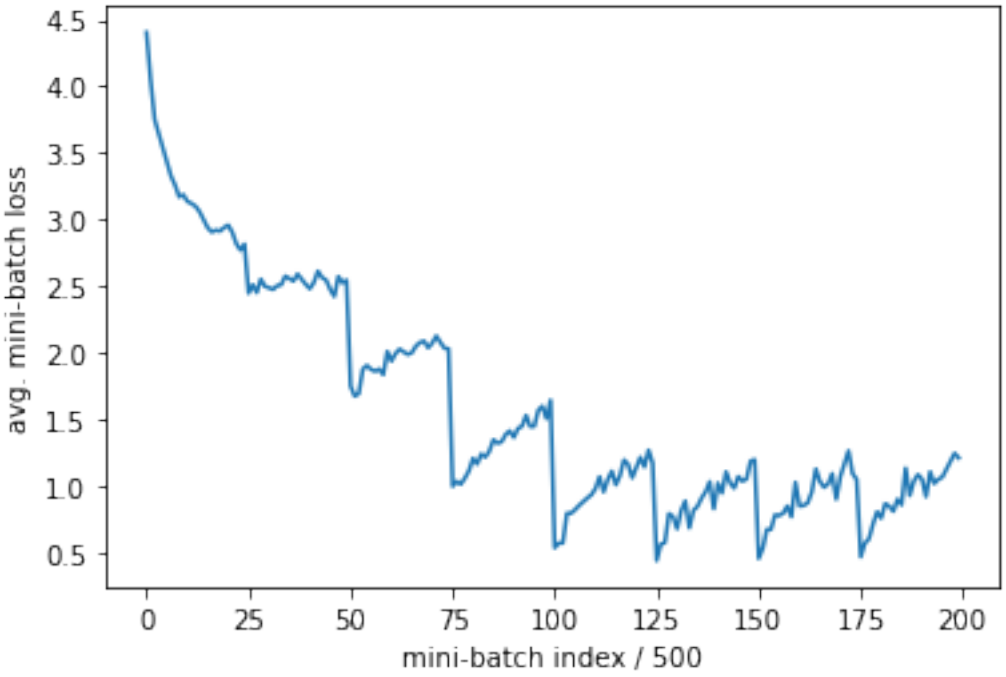
```
[epoch: 3, i:  6999] avg mini-batch loss: 1.383
[epoch: 3, i:  7499] avg mini-batch loss: 1.410
[epoch: 3, i:  7999] avg mini-batch loss: 1.361
[epoch: 3, i:  8499] avg mini-batch loss: 1.429
[epoch: 3, i:  8999] avg mini-batch loss: 1.445
[epoch: 3, i:  9499] avg mini-batch loss: 1.527
[epoch: 3, i:  9999] avg mini-batch loss: 1.443
[epoch: 3, i: 10499] avg mini-batch loss: 1.445
[epoch: 3, i: 10999] avg mini-batch loss: 1.566
[epoch: 3, i: 11499] avg mini-batch loss: 1.596
[epoch: 3, i: 11999] avg mini-batch loss: 1.502
[epoch: 3, i: 12499] avg mini-batch loss: 1.641
  0%|          | 0/12500 [00:00<?, ?it/s]
[epoch: 4, i:   499] avg mini-batch loss: 0.531
[epoch: 4, i:   999] avg mini-batch loss: 0.566
[epoch: 4, i:  1499] avg mini-batch loss: 0.567
[epoch: 4, i:  1999] avg mini-batch loss: 0.790
[epoch: 4, i:  2499] avg mini-batch loss: 0.790
[epoch: 4, i:  2999] avg mini-batch loss: 0.817
[epoch: 4, i:  3499] avg mini-batch loss: 0.850
[epoch: 4, i:  3999] avg mini-batch loss: 0.878
[epoch: 4, i:  4499] avg mini-batch loss: 0.906
[epoch: 4, i:  4999] avg mini-batch loss: 0.930
[epoch: 4, i:  5499] avg mini-batch loss: 0.976
[epoch: 4, i:  5999] avg mini-batch loss: 1.065
[epoch: 4, i:  6499] avg mini-batch loss: 0.952
[epoch: 4, i:  6999] avg mini-batch loss: 1.044
[epoch: 4, i:  7499] avg mini-batch loss: 1.107
[epoch: 4, i:  7999] avg mini-batch loss: 1.005
[epoch: 4, i:  8499] avg mini-batch loss: 1.074
[epoch: 4, i:  8999] avg mini-batch loss: 1.192
[epoch: 4, i:  9499] avg mini-batch loss: 1.158
[epoch: 4, i:  9999] avg mini-batch loss: 1.058
[epoch: 4, i: 10499] avg mini-batch loss: 1.137
[epoch: 4, i: 10999] avg mini-batch loss: 1.211
[epoch: 4, i: 11499] avg mini-batch loss: 1.140
[epoch: 4, i: 11999] avg mini-batch loss: 1.265
[epoch: 4, i: 12499] avg mini-batch loss: 1.169
  0%|          | 0/12500 [00:00<?, ?it/s]
[epoch: 5, i:   499] avg mini-batch loss: 0.435
[epoch: 5, i:   999] avg mini-batch loss: 0.558
[epoch: 5, i:  1499] avg mini-batch loss: 0.569
[epoch: 5, i:  1999] avg mini-batch loss: 0.788
[epoch: 5, i:  2499] avg mini-batch loss: 0.766
[epoch: 5, i:  2999] avg mini-batch loss: 0.674
[epoch: 5, i:  3499] avg mini-batch loss: 0.811
```

```
[epoch: 5, i:  3999] avg mini-batch loss: 0.882
[epoch: 5, i:  4499] avg mini-batch loss: 0.681
[epoch: 5, i:  4999] avg mini-batch loss: 0.811
[epoch: 5, i:  5499] avg mini-batch loss: 0.843
[epoch: 5, i:  5999] avg mini-batch loss: 0.905
[epoch: 5, i:  6499] avg mini-batch loss: 0.953
[epoch: 5, i:  6999] avg mini-batch loss: 1.028
[epoch: 5, i:  7499] avg mini-batch loss: 0.825
[epoch: 5, i:  7999] avg mini-batch loss: 1.018
[epoch: 5, i:  8499] avg mini-batch loss: 0.943
[epoch: 5, i:  8999] avg mini-batch loss: 1.102
[epoch: 5, i:  9499] avg mini-batch loss: 1.018
[epoch: 5, i:  9999] avg mini-batch loss: 0.980
[epoch: 5, i: 10499] avg mini-batch loss: 1.067
[epoch: 5, i: 10999] avg mini-batch loss: 1.033
[epoch: 5, i: 11499] avg mini-batch loss: 1.048
[epoch: 5, i: 11999] avg mini-batch loss: 1.182
[epoch: 5, i: 12499] avg mini-batch loss: 1.194

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

[epoch: 6, i:   499] avg mini-batch loss: 0.451
[epoch: 6, i:   999] avg mini-batch loss: 0.528
[epoch: 6, i:  1499] avg mini-batch loss: 0.669
[epoch: 6, i:  1999] avg mini-batch loss: 0.670
[epoch: 6, i:  2499] avg mini-batch loss: 0.778
[epoch: 6, i:  2999] avg mini-batch loss: 0.774
[epoch: 6, i:  3499] avg mini-batch loss: 0.791
[epoch: 6, i:  3999] avg mini-batch loss: 0.844
[epoch: 6, i:  4499] avg mini-batch loss: 0.765
[epoch: 6, i:  4999] avg mini-batch loss: 1.023
[epoch: 6, i:  5499] avg mini-batch loss: 0.846
[epoch: 6, i:  5999] avg mini-batch loss: 0.849
[epoch: 6, i:  6499] avg mini-batch loss: 0.869
[epoch: 6, i:  6999] avg mini-batch loss: 0.950
[epoch: 6, i:  7499] avg mini-batch loss: 1.123
[epoch: 6, i:  7999] avg mini-batch loss: 1.030
[epoch: 6, i:  8499] avg mini-batch loss: 0.987
[epoch: 6, i:  8999] avg mini-batch loss: 1.011
[epoch: 6, i:  9499] avg mini-batch loss: 1.088
[epoch: 6, i:  9999] avg mini-batch loss: 0.897
[epoch: 6, i: 10499] avg mini-batch loss: 1.066
[epoch: 6, i: 10999] avg mini-batch loss: 1.166
[epoch: 6, i: 11499] avg mini-batch loss: 1.260
[epoch: 6, i: 11999] avg mini-batch loss: 1.089
[epoch: 6, i: 12499] avg mini-batch loss: 1.045

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

[epoch: 7, i:   499] avg mini-batch loss: 0.463
```

```
[epoch: 7, i:   999] avg mini-batch loss: 0.566
[epoch: 7, i:  1499] avg mini-batch loss: 0.597
[epoch: 7, i:  1999] avg mini-batch loss: 0.710
[epoch: 7, i:  2499] avg mini-batch loss: 0.801
[epoch: 7, i:  2999] avg mini-batch loss: 0.754
[epoch: 7, i:  3499] avg mini-batch loss: 0.863
[epoch: 7, i:  3999] avg mini-batch loss: 0.843
[epoch: 7, i:  4499] avg mini-batch loss: 0.805
[epoch: 7, i:  4999] avg mini-batch loss: 0.896
[epoch: 7, i:  5499] avg mini-batch loss: 0.854
[epoch: 7, i:  5999] avg mini-batch loss: 1.130
[epoch: 7, i:  6499] avg mini-batch loss: 0.924
[epoch: 7, i:  6999] avg mini-batch loss: 1.031
[epoch: 7, i:  7499] avg mini-batch loss: 1.081
[epoch: 7, i:  7999] avg mini-batch loss: 1.048
[epoch: 7, i:  8499] avg mini-batch loss: 0.916
[epoch: 7, i:  8999] avg mini-batch loss: 1.102
[epoch: 7, i:  9499] avg mini-batch loss: 1.015
[epoch: 7, i:  9999] avg mini-batch loss: 1.046
[epoch: 7, i: 10499] avg mini-batch loss: 1.063
[epoch: 7, i: 10999] avg mini-batch loss: 1.121
[epoch: 7, i: 11499] avg mini-batch loss: 1.184
[epoch: 7, i: 11999] avg mini-batch loss: 1.242
[epoch: 7, i: 12499] avg mini-batch loss: 1.207
Finished Training.
```

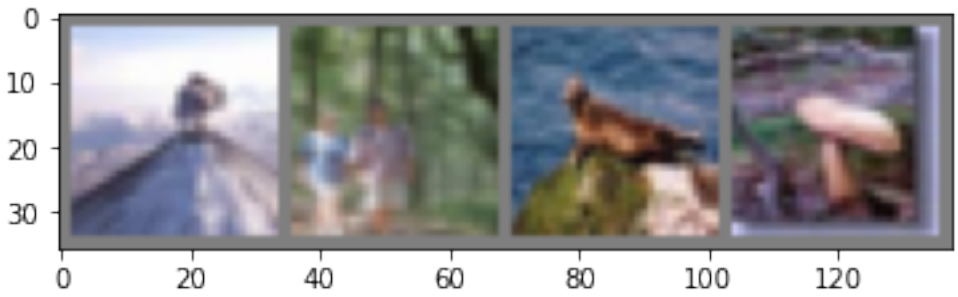
Training Loss Curve

[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:  bridge squirrel 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: 35 %
```

[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 : 69 %
Accuracy of aquarium_fish : 61 %
Accuracy of  baby : 30 %
Accuracy of  bear : 12 %
Accuracy of beaver : 21 %
Accuracy of   bed : 26 %
Accuracy of   bee : 48 %
Accuracy of beetle : 35 %
Accuracy of bicycle : 58 %
```

```
Accuracy of bottle : 58 %
Accuracy of  bowl : 34 %
Accuracy of   boy : 19 %
Accuracy of bridge : 45 %
Accuracy of   bus : 17 %
Accuracy of butterfly : 20 %
Accuracy of camel : 21 %
Accuracy of   can : 34 %
Accuracy of castle : 47 %
Accuracy of caterpillar : 28 %
Accuracy of cattle : 15 %
Accuracy of chair : 67 %
Accuracy of chimpanzee : 47 %
Accuracy of clock : 48 %
Accuracy of cloud : 64 %
Accuracy of cockroach : 45 %
Accuracy of couch : 23 %
Accuracy of  crab : 28 %
Accuracy of crocodile : 21 %
Accuracy of   cup : 51 %
Accuracy of dinosaur : 33 %
Accuracy of dolphin : 35 %
Accuracy of elephant : 31 %
Accuracy of flatfish : 35 %
Accuracy of forest : 26 %
Accuracy of   fox : 24 %
Accuracy of  girl : 22 %
Accuracy of hamster : 45 %
Accuracy of house : 35 %
Accuracy of kangaroo : 20 %
Accuracy of keyboard : 39 %
Accuracy of  lamp : 37 %
Accuracy of lawn_mower : 71 %
Accuracy of leopard : 25 %
Accuracy of  lion : 29 %
Accuracy of lizard :  8 %
Accuracy of lobster : 25 %
Accuracy of   man :  7 %
Accuracy of maple_tree : 41 %
Accuracy of motorcycle : 73 %
Accuracy of mountain : 29 %
Accuracy of mouse : 16 %
Accuracy of mushroom : 42 %
Accuracy of oak_tree : 60 %
Accuracy of orange : 55 %
Accuracy of orchid : 37 %
Accuracy of otter : 12 %
Accuracy of palm_tree : 59 %
```

```
Accuracy of  pear : 43 %
Accuracy of pickup_truck : 42 %
Accuracy of pine_tree : 16 %
Accuracy of plain : 78 %
Accuracy of plate : 43 %
Accuracy of poppy : 49 %
Accuracy of porcupine : 22 %
Accuracy of possum : 23 %
Accuracy of rabbit : 13 %
Accuracy of raccoon : 19 %
Accuracy of   ray : 28 %
Accuracy of  road : 54 %
Accuracy of rocket : 51 %
Accuracy of  rose : 21 %
Accuracy of   sea : 52 %
Accuracy of  seal :  8 %
Accuracy of shark : 25 %
Accuracy of shrew : 26 %
Accuracy of skunk : 48 %
Accuracy of skyscraper : 64 %
Accuracy of snail : 23 %
Accuracy of snake : 13 %
Accuracy of spider : 43 %
Accuracy of squirrel : 22 %
Accuracy of streetcar : 38 %
Accuracy of sunflower : 60 %
Accuracy of sweet_pepper : 20 %
Accuracy of table : 34 %
Accuracy of  tank : 52 %
Accuracy of telephone : 36 %
Accuracy of television : 40 %
Accuracy of tiger : 25 %
Accuracy of tractor : 35 %
Accuracy of train : 37 %
Accuracy of trout : 43 %
Accuracy of tulip : 39 %
Accuracy of turtle : 17 %
Accuracy of wardrobe : 72 %
Accuracy of whale : 29 %
Accuracy of willow_tree : 31 %
Accuracy of  wolf : 46 %
Accuracy of woman : 18 %
Accuracy of  worm : 32 %
```

[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.
```