assignment3

November 12, 2019

1 A53299801 - SIDDARTH MEENAKSHI SUNDARAM

[4]:
```
import os
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
import torch.utils.data as td
import torchvision as tv
import pandas as pd
from PIL import Image
from matplotlib import pyplot as plt
```

[5]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
cuda
```

$ **Q1**$

[6]:
```
#1
dataset_root_dir = '/datasets/ee285f-public/caltech_ucsd_birds'
```

$ **Q2**$

[7]:
```
#2
class BirdsDataset(td.Dataset):

    def __init__(self, root_dir, mode="train", image_size=(224, 224)):
        super(BirdsDataset, self).__init__()
        self.image_size = image_size
        self.mode = mode
        self.data = pd.read_csv(os.path.join(root_dir, "%s.csv" % mode))
        self.images_dir = os.path.join(root_dir, "CUB_200_2011/images")

    def __len__(self):
        return len(self.data)
    def __repr__(self):
        return "BirdsDataset(mode={}, image_size={})". \
               format(self.mode, self.image_size)

    def __getitem__(self, idx):
        img_path = os.path.join(self.images_dir, \
                                self.data.iloc[idx]['file_path'])
        bbox = self.data.iloc[idx][['x1', 'y1', 'x2', 'y2']]
        img = Image.open(img_path).convert('RGB')
        img = img.crop([bbox[0], bbox[1], bbox[2], bbox[3]])
        transform = tv.transforms.Compose([
            tv.transforms.Resize(self.image_size),
            tv.transforms.ToTensor(),
            tv.transforms.Normalize((.5, .5, .5), (.5, .5, .5))
            ])
        x = transform(img)
        d = self.data.iloc[idx]['class']
        return x, d

    def number_of_classes(self):
        return self.data['class'].max() + 1
```

[8]:
```
print(tv.__version__)
```

```
0.4.2
```

$ **Q3**$

[9]:
```
#3
def myimshow(image, ax=plt):
    image = image.to('cpu').numpy()
    image = np.moveaxis(image, [0, 1, 2], [2, 0, 1])
    image = (image + 1) / 2
    image[image < 0] = 0
    image[image > 1] = 1
    h = ax.imshow(image)
    ax.axis('off')
    return h
```

[10]:
```
train_set = BirdsDataset(dataset_root_dir)
x = train_set.__getitem__(10)
```

[11]:
```
plt.figure()
myimshow(x[0])
```

[11]:
```
<matplotlib.image.AxesImage at 0x7fe8900b2780>
```

$ **Q4**$

[12]:
```
#4
train_loader = td.DataLoader(train_set, batch_size=16, shuffle=True,␣
↪pin_memory=True)
```

The data loader will copy tensors into CUDA pinned memory before returning them.

We see from CUDA documentation on using pinning that, “This makes the transfer faster. The pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory.”

We also see from documentation that, Transfers between the host and device are the slowest link of data movement involved in GPU computing, so you should take care to minimize transfers.

Hence, use of pin_memory makes the host to device transfers faster.

[13]:
```
len(train_loader)
```

[13]:
```
47
```

The number of minibatches : 47,

This can be verified manually as ceil (743/16) = 47

$ **Q5**$

[14]:
```
#5
for batch_no, batch in enumerate(train_loader):
    if batch_no == 4:
        break

plt.figure()
myimshow(batch[0][0])
print("Label",batch_no+1,":", batch[1][0])
```

```
Label 1 : tensor(8)
Label 2 : tensor(3)
Label 3 : tensor(2)
Label 4 : tensor(1)
```

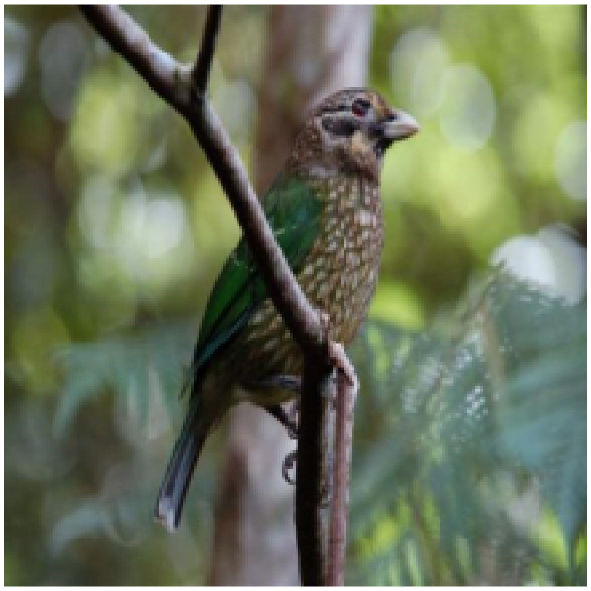
ARKive
www.arkive.org
© Graham Robertson / Auscape International

```
[17]: for batch_no, batch in enumerate(train_loader):
          if batch_no == 4:
              break

          plt.figure()
          myimshow(batch[0][0])
          print("Label",batch_no+1,":", batch[1][0])
```

```
Label 1 : tensor(6)
Label 2 : tensor(12)
Label 3 : tensor(6)
Label 4 : tensor(17)
```

HEATHER FORCIER
WWW.HFORCIER.COM



As the shuffle is set to true, we get different images of birds each time this is run.

$ **Q6**$

```
[18]: #6
val_set = BirdsDataset(dataset_root_dir, mode='val')
val_loader = td.DataLoader(val_set, batch_size=16, pin_memory=True)
```

$ **Shuffling Validation and training sets:** $

Shuffling of training data leads to a a more uniform distribution of data, in case of unshuffled data, you might create batches that are not representative of the whole dataset and the estimate will be wrong. Hence, to create a general model, it is useful to shuffle training data.

In case of validation set, the loss/performance has to be evaluated and evaluation doesn't depend on the parameters of the system or the order. Hence, shuffling of validation dataset is not needed.

[28]:
```
import nntools as nt
```

$ **Q7**$

[15]:
```
#7
net = nt.NeuralNetwork()
```

```
        TypeErrorTraceback (most recent call last)

        <ipython-input-15-045728e9f2a4> in <module>
          1 #7
    ----> 2 net = nt.NeuralNetwork()


        TypeError: Can't instantiate abstract class NeuralNetwork with abstract␣
 ↪methods criterion, forward
```

An error occurs because, NeuralNetwork is an abstract class and the methods criterion and forward has not yet been defined.

```
@abstractmethod

    def forward(self, x):
    pass

@abstractmethod

    def criterion(self, y, d):
    pass
```

We can see from the code - pass in nntools.py that the forward and criterion methods must be defined before using them.

[27]:
```
class NNClassifier(nt.NeuralNetwork):

    def __init__(self):
        super(NNClassifier, self).__init__()
        self.cross_entropy = nn.CrossEntropyLoss()

    def criterion(self, y, d):
        return self.cross_entropy(y, d)
```

Criterion has been implemented but forward is not, Hence NNClassifier is an abstract class.

$ **Q8**$

[22]:
```
#8
vgg = tv.models.vgg16_bn(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/vgg16_bn-6c64b313.pth" to
/tmp/xdg-cache/torch/checkpoints/vgg16_bn-6c64b313.pth
100%|      | 528M/528M [00:08<00:00, 65.6MB/s]
```

[24]:
```
vgg #Printing vgg net
```

[24]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (36): ReLU(inplace=True)
    (37): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (39): ReLU(inplace=True)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

[25]:
```python
for name, param in vgg.named_parameters():
    print(name, param.size(), param.requires_grad)
```

```
features.0.weight torch.Size([64, 3, 3, 3]) True
features.0.bias torch.Size([64]) True
features.1.weight torch.Size([64]) True
features.1.bias torch.Size([64]) True
features.3.weight torch.Size([64, 64, 3, 3]) True
features.3.bias torch.Size([64]) True
features.4.weight torch.Size([64]) True
features.4.bias torch.Size([64]) True
features.7.weight torch.Size([128, 64, 3, 3]) True
features.7.bias torch.Size([128]) True
features.8.weight torch.Size([128]) True
features.8.bias torch.Size([128]) True
features.10.weight torch.Size([128, 128, 3, 3]) True
features.10.bias torch.Size([128]) True
features.11.weight torch.Size([128]) True
features.11.bias torch.Size([128]) True
features.14.weight torch.Size([256, 128, 3, 3]) True
features.14.bias torch.Size([256]) True
features.15.weight torch.Size([256]) True
features.15.bias torch.Size([256]) True
features.17.weight torch.Size([256, 256, 3, 3]) True
features.17.bias torch.Size([256]) True
features.18.weight torch.Size([256]) True
features.18.bias torch.Size([256]) True
features.20.weight torch.Size([256, 256, 3, 3]) True
features.20.bias torch.Size([256]) True
features.21.weight torch.Size([256]) True
features.21.bias torch.Size([256]) True
features.24.weight torch.Size([512, 256, 3, 3]) True
features.24.bias torch.Size([512]) True
features.25.weight torch.Size([512]) True
features.25.bias torch.Size([512]) True
features.27.weight torch.Size([512, 512, 3, 3]) True
features.27.bias torch.Size([512]) True
features.28.weight torch.Size([512]) True
features.28.bias torch.Size([512]) True
features.30.weight torch.Size([512, 512, 3, 3]) True
features.30.bias torch.Size([512]) True
features.31.weight torch.Size([512]) True
features.31.bias torch.Size([512]) True
features.34.weight torch.Size([512, 512, 3, 3]) True
features.34.bias torch.Size([512]) True
features.35.weight torch.Size([512]) True
features.35.bias torch.Size([512]) True
features.37.weight torch.Size([512, 512, 3, 3]) True
features.37.bias torch.Size([512]) True
features.38.weight torch.Size([512]) True
features.38.bias torch.Size([512]) True
features.40.weight torch.Size([512, 512, 3, 3]) True
features.40.bias torch.Size([512]) True
features.41.weight torch.Size([512]) True
features.41.bias torch.Size([512]) True
classifier.0.weight torch.Size([4096, 25088]) True
classifier.0.bias torch.Size([4096]) True
classifier.3.weight torch.Size([4096, 4096]) True
classifier.3.bias torch.Size([4096]) True
classifier.6.weight torch.Size([1000, 4096]) True
classifier.6.bias torch.Size([1000]) True
```

We can see that above parameters are learnable.

The learnable parameters are :

1. Weights of all features and classifiers
2. Biases of all features and classifiers

$ **Q9**$

[29]:
```
#9
class VGG16Transfer(NNClassifier):

    def __init__(self, num_classes, fine_tuning=False):
        super(VGG16Transfer, self).__init__()
        vgg = tv.models.vgg16_bn(pretrained=True)
        for param in vgg.parameters():
            param.requires_grad = fine_tuning
        self.features = vgg.features
        self.avgpool = vgg.avgpool
        self.classifier = vgg.classifier
        num_ftrs = vgg.classifier[6].in_features
        self.classifier[6] = nn.Linear(num_ftrs, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        f = torch.flatten(x, 1)
        y = self.classifier(f)
        return y
```

$ **Q10**$

[30]:
```
#10
num_classes = train_set.number_of_classes()
print (num_classes)
```

```
20
```

[31]:
```
vgg16 = VGG16Transfer(num_classes)
```

[32]:
```
vgg16
```

[32]:
```
VGG16Transfer(
  (cross_entropy): CrossEntropyLoss()
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (14): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (36): ReLU(inplace=True)
    (37): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (39): ReLU(inplace=True)
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=20, bias=True)
  )
)
```

[34]:
```
for name, param in vgg16.named_parameters():
    print(name, param.size(), param.requires_grad)
```

```
classifier.6.weight torch.Size([20, 4096]) True
classifier.6.bias torch.Size([20]) True
```

The weights and biases of the final classifier - final fc layer are learnable.

$ **Q11**$

[35]:
```
#11
class ClassificationStatsManager(nt.StatsManager):

    def __init__(self):
        super(ClassificationStatsManager, self).__init__()

    def init(self):
        super(ClassificationStatsManager, self).init()
        self.running_accuracy = 0

    def accumulate(self, loss, x, y, d):
        super(ClassificationStatsManager, self).accumulate(loss, x, y, d)
        _, l = torch.max(y, 1)
        self.running_accuracy += torch.mean((l == d).float())

    def summarize(self):
        loss = super(ClassificationStatsManager, self).summarize()
        accuracy = 100 * self.running_accuracy / self.number_update
        return {'loss': loss, 'accuracy': accuracy}
```

$ **Ans. 12.**$

self.net is set to eval() mode to notify all layers that you are in eval mode.

We see from the eval() module documentation that it has effect on certain modules like BatchNorm and Dropout. We use it in this case because vggnet uses BatchNorm2d module for finding feautures. Also, vggnet uses Dropout module while pooling.

The Dropout and BatchNorm behave differently during training and evaluation. We can see from documentation that they work only in evaluation mode.

$ **Q13**$

[36]:
```
#13
lr = 1e-3
net = VGG16Transfer(num_classes)
net = net.to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = ClassificationStatsManager()
exp1 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
               output_dir="birdclass1", perform_validation_during_training=True)
```

birdclass1 has been created.

What does `checkpoint.pth.tar` correspond to?

We can see the following piece of code in nntools.py:

```
        print("Epoch {} (Time: {:.2f}s)".format(

            self.epoch, time.time() - s))
        self.save()
```

We can observe from the above code that the net is stored after each epoch. Also, we can see from torch.save documentation that torch.save saves an object to a disk file.

After each epoch, `checkpoint.pth.tar`, a binary file containing the state of the experiment will be stored in the directory `output_dir`

If `output_dir` does not exist, it will be created. Otherwise, the last checkpoint `checkpoint.pth.tar` containing state of the experiment will be loaded, and loaded experiment will be continued from where it stopped (settings {hyperparameters, batch size, etc.} known from `config.txt` file).

$ **Q14**$

[27]:
```
#14
lr = 1e-4
net = VGG16Transfer(num_classes)
net = net.to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = ClassificationStatsManager()
exp1 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
               output_dir="birdclass1", perform_validation_during_training=True)
```

```
        ValueErrorTraceback (most recent call last)

        <ipython-input-27-c8070f025878> in <module>
          6 stats_manager = ClassificationStatsManager()
          7 exp1 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
    ----> 8                output_dir="birdclass1",␣
↪perform_validation_during_training=True)


        /datasets/home/home-01/33/133/simeenak/nntools.py in __init__(self, net,␣
↪train_set, val_set, optimizer, stats_manager, output_dir, batch_size,␣
↪perform_validation_during_training)
        168                 if f.read()[:-1] != repr(self):
        169                     raise ValueError(
    --> 170                         "Cannot create this experiment: "
        171                         "I found a checkpoint conflicting with the␣
↪current setting.")
        172             self.load()


        ValueError: Cannot create this experiment: I found a checkpoint␣
↪conflicting with the current setting.
```

We can see that for learning rate = 1e-4, an error occurs. This is because when loading, it checks with the setting of the model from init() which checks the settings from the stored `config.txt` file, which has a value of 1e-3 which produces a conflict and hence the error occurs.

[37]:
```
lr = 1e-3
net = VGG16Transfer(num_classes)
net = net.to(device)
adam = torch.optim.Adam(net.parameters(), lr=lr)
stats_manager = ClassificationStatsManager()
exp1 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
               output_dir="birdclass1", perform_validation_during_training=True)
```

When we change the learning rate back to 1e-3, the setting matches and the experiment runs smoothly.

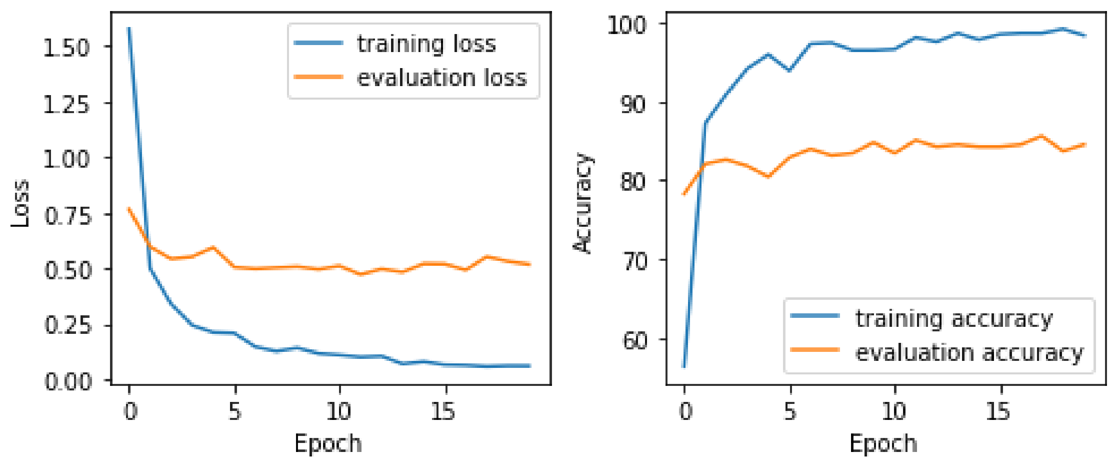
$ **Q15**$

[63]:
```
#15
def plot(exp, fig, axes):
    axes[0].clear()
    axes[1].clear()
    axes[0].plot([exp.history[k][0]['loss'] for k in range(exp.epoch)],
                 label="training loss")

    axes[0].plot([exp.history[k][1]['loss'] for k in range(exp.epoch)],
                 label="evaluation loss")
    axes[0].set_xlabel('Epoch')
    axes[0].set_ylabel('Loss')
    axes[0].legend(('training loss', 'evaluation loss'))
    axes[1].plot([exp.history[k][0]['accuracy'] for k in range(exp.epoch)],
                 label="training accuracy")
    # evaluation accuracy
    axes[1].plot([exp.history[k][1]['accuracy'] for k in range(exp.epoch)],
                 label="evaluation accuracy")
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('Accuracy')
    axes[1].legend(('training accuracy', 'evaluation accuracy'))
    plt.tight_layout()
    fig.canvas.draw()
```

[64]:
```
fig, axes = plt.subplots(ncols=2, figsize=(7, 3))
exp1.run(num_epochs=20, plot=lambda exp: plot(exp, fig=fig, axes=axes))
```

```
Start/Continue training from epoch 20
Finish training for 20 epochs
```



$ **Q16**$

[49]:
```
resnet = tv.models.resnet18(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to
/tmp/xdg-cache/torch/checkpoints/resnet18-5c106cde.pth
100%|      | 44.7M/44.7M [00:00<00:00, 56.0MB/s]
```

[50]:
```
resnet
```

[50]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
```

```
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

[51]:
```
for name, param in resnet.named_parameters():
    print(name, param.size(), param.requires_grad)
```

```
conv1.weight torch.Size([64, 3, 7, 7]) True
bn1.weight torch.Size([64]) True
bn1.bias torch.Size([64]) True
layer1.0.conv1.weight torch.Size([64, 64, 3, 3]) True
layer1.0.bn1.weight torch.Size([64]) True
layer1.0.bn1.bias torch.Size([64]) True
layer1.0.conv2.weight torch.Size([64, 64, 3, 3]) True
layer1.0.bn2.weight torch.Size([64]) True
layer1.0.bn2.bias torch.Size([64]) True
layer1.1.conv1.weight torch.Size([64, 64, 3, 3]) True
layer1.1.bn1.weight torch.Size([64]) True
layer1.1.bn1.bias torch.Size([64]) True
layer1.1.conv2.weight torch.Size([64, 64, 3, 3]) True
layer1.1.bn2.weight torch.Size([64]) True
layer1.1.bn2.bias torch.Size([64]) True
layer2.0.conv1.weight torch.Size([128, 64, 3, 3]) True
layer2.0.bn1.weight torch.Size([128]) True
layer2.0.bn1.bias torch.Size([128]) True
layer2.0.conv2.weight torch.Size([128, 128, 3, 3]) True
layer2.0.bn2.weight torch.Size([128]) True
layer2.0.bn2.bias torch.Size([128]) True
layer2.0.downsample.0.weight torch.Size([128, 64, 1, 1]) True
layer2.0.downsample.1.weight torch.Size([128]) True
layer2.0.downsample.1.bias torch.Size([128]) True
layer2.1.conv1.weight torch.Size([128, 128, 3, 3]) True
layer2.1.bn1.weight torch.Size([128]) True
layer2.1.bn1.bias torch.Size([128]) True
layer2.1.conv2.weight torch.Size([128, 128, 3, 3]) True
layer2.1.bn2.weight torch.Size([128]) True
layer2.1.bn2.bias torch.Size([128]) True
layer3.0.conv1.weight torch.Size([256, 128, 3, 3]) True
layer3.0.bn1.weight torch.Size([256]) True
layer3.0.bn1.bias torch.Size([256]) True
layer3.0.conv2.weight torch.Size([256, 256, 3, 3]) True
layer3.0.bn2.weight torch.Size([256]) True
layer3.0.bn2.bias torch.Size([256]) True
```

```
layer3.0.downsample.0.weight torch.Size([256, 128, 1, 1]) True
layer3.0.downsample.1.weight torch.Size([256]) True
layer3.0.downsample.1.bias torch.Size([256]) True
layer3.1.conv1.weight torch.Size([256, 256, 3, 3]) True
layer3.1.bn1.weight torch.Size([256]) True
layer3.1.bn1.bias torch.Size([256]) True
layer3.1.conv2.weight torch.Size([256, 256, 3, 3]) True
layer3.1.bn2.weight torch.Size([256]) True
layer3.1.bn2.bias torch.Size([256]) True
layer4.0.conv1.weight torch.Size([512, 256, 3, 3]) True
layer4.0.bn1.weight torch.Size([512]) True
layer4.0.bn1.bias torch.Size([512]) True
layer4.0.conv2.weight torch.Size([512, 512, 3, 3]) True
layer4.0.bn2.weight torch.Size([512]) True
layer4.0.bn2.bias torch.Size([512]) True
layer4.0.downsample.0.weight torch.Size([512, 256, 1, 1]) True
layer4.0.downsample.1.weight torch.Size([512]) True
layer4.0.downsample.1.bias torch.Size([512]) True
layer4.1.conv1.weight torch.Size([512, 512, 3, 3]) True
layer4.1.bn1.weight torch.Size([512]) True
layer4.1.bn1.bias torch.Size([512]) True
layer4.1.conv2.weight torch.Size([512, 512, 3, 3]) True
layer4.1.bn2.weight torch.Size([512]) True
layer4.1.bn2.bias torch.Size([512]) True
fc.weight torch.Size([1000, 512]) True
fc.bias torch.Size([1000]) True
```

[52]:
```
class Resnet18Transfer(NNClassifier):

    def __init__(self, num_classes, fine_tuning=False):
        super(Resnet18Transfer, self).__init__()
        resnet = tv.models.resnet18(pretrained=True)
        for param in resnet.parameters():
            param.requires_grad = fine_tuning

        self.conv1 = resnet.conv1
        self.bn1 = resnet.bn1
        self.relu = resnet.relu
        self.maxpool = resnet.maxpool
        self.layer1 = resnet.layer1
        self.layer2 = resnet.layer2
        self.layer3 = resnet.layer3
        self.layer4 = resnet.layer4
        self.avgpool = resnet.avgpool
        self.fc = resnet.fc

        num_ftrs = self.fc.in_features
        self.fc = nn.Linear(num_ftrs, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        f = self.fc(x)

        return f
```

$ **Q17**$

[54]:
```
#17
num_classes= train_set.number_of_classes()
resnet_object  = Resnet18Transfer(num_classes) #instance of resnet
```

[62]:
```
resnet_object
```

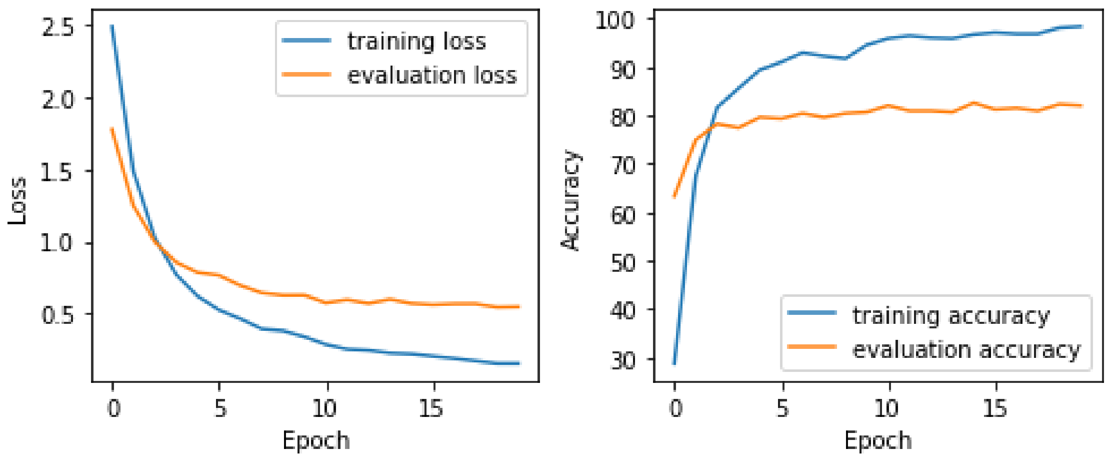
[62]:
```
Resnet18Transfer(
  (cross_entropy): CrossEntropyLoss()
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=20, bias=True)
)
```

[59]:
```
lr = 1e-3
net_res = Resnet18Transfer(num_classes)
net_res = net_res.to(device)
adam = torch.optim.Adam(net_res.parameters(), lr=lr)
stats_manager = ClassificationStatsManager()
exp2 = nt.Experiment(net_res, train_set, val_set, adam, stats_manager,
output_dir="birdclass2", perform_validation_during_training=True)
```

[65]:
```
fig, axes = plt.subplots(ncols=2, figsize=(7, 3))
exp2.run(num_epochs=20, plot=lambda exp: plot(exp, fig=fig, axes=axes))
```

```
Start/Continue training from epoch 20
Finish training for 20 epochs
```



$ **Q18**$

[58]:
```
exp1.evaluate() #Evaluation of VGGNet
```

[58]:
```
{'loss': 0.5181037927775279, 'accuracy': tensor(84.5109, device='cuda:0')}
```

[61]:
```
exp2.evaluate() #Evaluation of ResNet
```

```
[61]: {'loss': 0.5476801816536032, 'accuracy': tensor(82.0652, device='cuda:0')}
```

We can see that the vggnet produces a training accuracy of 84.51% and the Resnet18 has a training accuracy of 82.06%. In this case VGGNet16 produces a better accuracy than ResNet-18.