In [1]:

```
#%matplotlib notebook

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
import socket
import getpass
from matplotlib import pyplot as plt
```

In [2]:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
cuda
```

Question 1

In [3]:

```
dataset_root_dir = "/datasets/ee285f-public/caltech_ucsd_birds/"

print(dataset_root_dir)
```

```
/datasets/ee285f-public/caltech_ucsd_birds/
```

Question 2

In [4]:

```
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((224,224)),            #resize PIL image
        tv.transforms.ToTensor(),                  #convert numpy array to tensor
        tv.transforms.Normalize([0.5, 0.5, 0.5],[0.5, 0.5, 0.5])
         ])
        x = transform(img)
        d = self.data.iloc[idx]['class']
        return x, d
    def number_of_classes(self):
        return self.data['class'].max() + 1
```

Question 3

In [5]:

```
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
```

Question 4

In [6]:

```
train_set = BirdsDataset(dataset_root_dir , mode = "train" , image_size =(224,224))

x = train_set[10][0]

xlabel = train_set[10][1]     #label of x

#x = train_set.__getitem__(10)

myimshow(x)

print(xlabel)      #bird belongs to class 0

print(train_set.__len__())
```

```
0
743
```

The training data is loaded as shuffled minibatches in the cell below. (Shuffle = True) . We have then displayed the number of minibatches.

In [7]:

```
train_loader = td.DataLoader(train_set, batch_size=16, shuffle=True, pin_memory=True, d
rop_last=True)
```

In [8]:

```
no_of_mini_batches = np.ceil(train_set.__len__() / 16 )

print("Number of minibatches : " , no_of_mini_batches)
```

```
Number of minibatches :  47.0
```

Advantage of pin memory

By setting the pin memory to true, when you load your samples present in the Dataset on CPU and transfer to GPU , you can speed up the host to device transfer by enabling pin_memory. Setting it true will automatically put the tensors in the pinned memory space.The transfer of data from the pinned memory space to the device is faster and hence helps in fast data transfer.

Question 5

In [9]:

```
# i = batch index
# batch = image label pairs of batch size
# print(train_loader.__len__())
# print(train_loader.dataset[0])

fig, axes = plt.subplots(ncols=4)

i = 0

for i,data_batch in enumerate (train_loader):

     if i == 4:
        break


     myimshow(data_batch[0][0], ax =axes[i])

    #myimshow(image)

     print(data_batch[1][0])
```

```
tensor(14)
tensor(18)
tensor(9)
tensor(18)
```

We see that everytime we re-evaluate the cell , different images are produced. The images are shuffled each time train_loader is called.

Question 6

In [14]:

```
val_set = BirdsDataset(dataset_root_dir , mode = "val" , image_size =(224,224))
```

In [15]:

```
val_loader = td.DataLoader(train_set, batch_size=16, sampler=None,
                           batch_sampler=None, num_workers=0, collate_fn=None,
                           pin_memory=True, drop_last=False, timeout=0,
                           worker_init_fn=None)
```

Why do you think we need to shuffle the dataset for training but not for validation?

The traning data is shuffled and passed as mini-batches to avoid overfitting of the model , since shuffling data reduces variance and makes sure that the model remains general and overfits less. It prevents the model from learning the order of training.

Whereas validation is performed to simply evaluate the performance of the model on random data and to fine tune the hyperparameters. Here, the model is not being trained or it is not learning anything as in the case of the training set. Hence, we need not shuffle the data.

Question 7

In [69]:

```
import nntools as nt
```

In [17]:

```
net = nt.NeuralNetwork()
```

```
TypeErrorTraceback (most recent call last)
<ipython-input-17-2b13b95b774d> in <module>
----> 1 net = nt.NeuralNetwork()

TypeError: Can't instantiate abstract class NeuralNetwork with abstract me
thods criterion, forward
```

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods.

We observe that the class NeuralNetwork cannot be instantiated since it is an abstract class and it can only be inherited by another class or require a subclass to provide implementation in which the methods are defined to the neural network that is being used here.

In [18]:

```
#help(nt.NeuralNetwork)
```

Question 8

In [71]:

```
vgg = tv.models.vgg16_bn(pretrained=True)
```

In [20]:

```
for name, param in vgg.named_parameters():
    print(name, param.size(), param.dtype, param.requires_grad)
```

```
features.0.weight torch.Size([64, 3, 3, 3]) torch.float32 True
features.0.bias torch.Size([64]) torch.float32 True
features.1.weight torch.Size([64]) torch.float32 True
features.1.bias torch.Size([64]) torch.float32 True
features.3.weight torch.Size([64, 64, 3, 3]) torch.float32 True
features.3.bias torch.Size([64]) torch.float32 True
features.4.weight torch.Size([64]) torch.float32 True
features.4.bias torch.Size([64]) torch.float32 True
features.7.weight torch.Size([128, 64, 3, 3]) torch.float32 True
features.7.bias torch.Size([128]) torch.float32 True
features.8.weight torch.Size([128]) torch.float32 True
features.8.bias torch.Size([128]) torch.float32 True
features.10.weight torch.Size([128, 128, 3, 3]) torch.float32 True
features.10.bias torch.Size([128]) torch.float32 True
features.11.weight torch.Size([128]) torch.float32 True
features.11.bias torch.Size([128]) torch.float32 True
features.14.weight torch.Size([256, 128, 3, 3]) torch.float32 True
features.14.bias torch.Size([256]) torch.float32 True
features.15.weight torch.Size([256]) torch.float32 True
features.15.bias torch.Size([256]) torch.float32 True
features.17.weight torch.Size([256, 256, 3, 3]) torch.float32 True
features.17.bias torch.Size([256]) torch.float32 True
features.18.weight torch.Size([256]) torch.float32 True
features.18.bias torch.Size([256]) torch.float32 True
features.20.weight torch.Size([256, 256, 3, 3]) torch.float32 True
features.20.bias torch.Size([256]) torch.float32 True
features.21.weight torch.Size([256]) torch.float32 True
features.21.bias torch.Size([256]) torch.float32 True
features.24.weight torch.Size([512, 256, 3, 3]) torch.float32 True
features.24.bias torch.Size([512]) torch.float32 True
features.25.weight torch.Size([512]) torch.float32 True
features.25.bias torch.Size([512]) torch.float32 True
features.27.weight torch.Size([512, 512, 3, 3]) torch.float32 True
features.27.bias torch.Size([512]) torch.float32 True
features.28.weight torch.Size([512]) torch.float32 True
features.28.bias torch.Size([512]) torch.float32 True
features.30.weight torch.Size([512, 512, 3, 3]) torch.float32 True
features.30.bias torch.Size([512]) torch.float32 True
features.31.weight torch.Size([512]) torch.float32 True
features.31.bias torch.Size([512]) torch.float32 True
features.34.weight torch.Size([512, 512, 3, 3]) torch.float32 True
features.34.bias torch.Size([512]) torch.float32 True
features.35.weight torch.Size([512]) torch.float32 True
features.35.bias torch.Size([512]) torch.float32 True
features.37.weight torch.Size([512, 512, 3, 3]) torch.float32 True
features.37.bias torch.Size([512]) torch.float32 True
features.38.weight torch.Size([512]) torch.float32 True
features.38.bias torch.Size([512]) torch.float32 True
features.40.weight torch.Size([512, 512, 3, 3]) torch.float32 True
features.40.bias torch.Size([512]) torch.float32 True
features.41.weight torch.Size([512]) torch.float32 True
features.41.bias torch.Size([512]) torch.float32 True
classifier.0.weight torch.Size([4096, 25088]) torch.float32 True
classifier.0.bias torch.Size([4096]) torch.float32 True
classifier.3.weight torch.Size([4096, 4096]) torch.float32 True
classifier.3.bias torch.Size([4096]) torch.float32 True
classifier.6.weight torch.Size([1000, 4096]) torch.float32 True
classifier.6.bias torch.Size([1000]) torch.float32 True
```

The Learnable parameters are features.x.weight and features.x.bias. These are the parameters on which the loss is dependent. We minimize the loss with respect to the learnable parameters.

In [21]:

```
print(vgg)
```

```
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_runni
ng_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_runni
ng_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=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_runn
ing_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_run
ning_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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_run
ning_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_run
ning_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_run
ning_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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_run
ning_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_run
ning_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_run
ning_stats=True)
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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_run
```

```
ning_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_run
ning_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_run
ning_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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)
  )
)
```

Question 9

In [22]:

```
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)

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        #requires grad is false since we d
ont have to re train the network and calculate gradients for the previous layers
        self.features = vgg.features

        self.avgpool = vgg.avgpool
        self.flatten = nn.Flatten
        self.classifier = vgg.classifier

        num_ftrs = vgg.classifier[6].in_features
        self.classifier[6] = nn.Linear(num_ftrs, num_classes)

    def forward(self, x):

        f = self.features(x)
        #print(f.shape)
        f = self.avgpool(f)
        #print(f.shape)
        flat = nn.Flatten()
        f = flat(f)
        #print(f.shape)
        y = self.classifier(f)

        return y
```

Question 10

In [23]:

```
num_classes = train_set.number_of_classes()

net = VGG16Transfer(num_classes)

print(net)
```

```
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_runni
ng_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_runni
ng_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=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_runn
ing_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_run
ning_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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_run
ning_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_run
ning_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_run
ning_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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_run
ning_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_run
ning_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_run
ning_stats=True)
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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_run
ning_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_run
ning_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_run
ning_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=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)
  )
)
```

In [24]:

```
for name, param in net.named_parameters():
    print(name, param.size(), param.dtype, param.requires_grad)
```

```
classifier.6.weight torch.Size([20, 4096]) torch.float32 True
classifier.6.bias torch.Size([20]) torch.float32 True
```

The learnable parameters of the VGG16Transfer_net are just the last layer parameters (classifier layer parameters) - classifier.6.weight and classifier.6.bias.

Question 11

In [25]:

```
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}
```

Question 12

The eval method evaluates the experiment, i.e., forward propagates the validation set through the network and returns the statistics computed by the stats manager.

self.net is set to eval mode which means it sets the module in evaluation mode. This is done since all the layers behave differently in testing and eval modes. That is, during eval mode some of the fuctionalities are not required or are kept constant.

Hence we need to change operating mode. Here are different examples where each module functions differently in the two modes

1. GroupNorm - This module uses the statistics computed from input data in both training and evalaution mode.
2. BatchNorm - This module is kept running through the training mode by default and the mean and variance is computed. These values are then later used for normalization during evaluation.
3. Dropout - During training, randomly zeroes some of the elements of the input tensor (randomly sets some of its input to zero, which effectively erases them from the network) with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call. During evaluation the module computes an identity function.

Question 13

In [26]:

```
lr = 1e-3

net = VGG16Transfer(num_classes =  train_set.number_of_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)
```

Interpretting code :

The leaning rate is set to 1e - 3. The net is transfered to the GPU. The adam method is used for stochastic optimization.

Then the object of Experiment class is created having net,train_set,val_set,adam,stats_manager,output_dir,perform_validation_during_training as parameters.

The output_dir created 'birdclass1' in the working directory which has two files. 1. The config.txt file and 2. checkpoint.pth.tar file.

Inspecting the content of the birdclass :

Two files will be present:

1) `checkpoint.pth.tar` It contains the checkpoint ie. it is a binary file containing the state of the experiment. This enables us to continue from a checkpoint. We need not run/configure the model from scratch and rather continue from a checkpoint.

2) `config.txt` contains ASCII values which are the setting of the experiment.

Question 14

In [48]:

```
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-48-a68d98f79692> in <module>
     10
     11 exp1 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
---> 12 output_dir="birdclass1", perform_validation_during_training=True)

/datasets/home/home-00/09/609/spatange/nntools.py in __init__(self, net, t
rain_set, val_set, optimizer, stats_manager, output_dir, batch_size, perfo
rm_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 conflictin
g with the current setting.
```

When we set the learning rate to 1e-3 and run the code, the class Experiment creates an output directory and the config.txt file is already created with all the settings of the experiment. Hence when we initialize learning rate = 1e-4 and run the same code, an error is returned with an error message saying 'Conflicting Checkpoint'. This is because a config file is already created in the particular output directory with a different learning rate and the checkpoint is saved.

Thus to rectify this error and to change the learning rate to 1e-4 we have to clear the output directory and remove the already existing checkpoint and the config file with the already existing setting. or we can also create another seperate output directory with a another set of parameters to run the model with those parameters seperately.

When we switch the learning rate back to 1e-3 and if we run the code we see that it matches the parameters saved in the config file and hence the previous checkpoint is loaded. This is useful as it helps in loading parameters from these checkpoints without having to re-run everything again. This also ensures the intial parameter are set to be constant and the checkpoint is saved after each epoch once exp.run is initiated which means we need not run all the epochs at a time.

In [ ]:

```
lr = 1e-3

net = VGG16Transfer(num_classes =  train_set.number_of_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)
```

Question 15

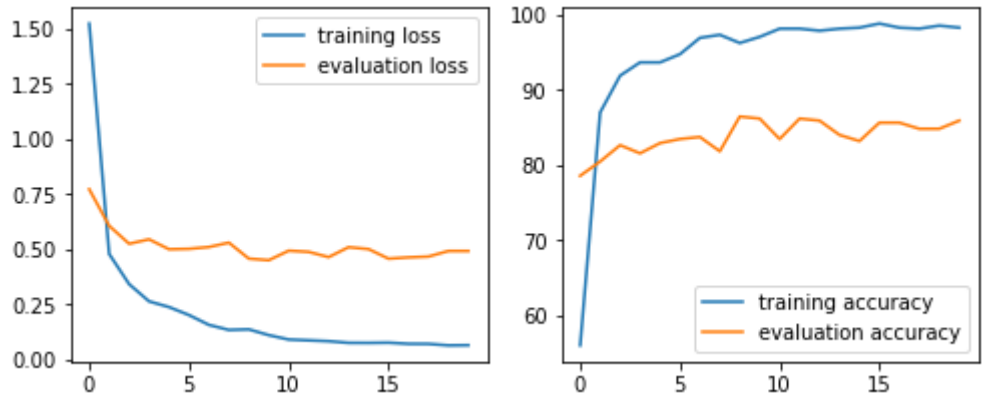
In [78]:

```
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[1].plot([exp.history[k][0]['accuracy'] for k in range(exp.epoch)],
    label="training accuracy")

    axes[0].plot([exp.history[k][1]['loss'] for k in range(exp.epoch)],
    label="evaluation loss")

    axes[1].plot([exp.history[k][1]['accuracy'] for k in range(exp.epoch)],
    label="evaluation accuracy")

    axes[0].legend()
    axes[1].legend()


    plt.tight_layout()
    fig.canvas.draw()
```

In [79]:

```
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
```



In [66]:

```
resnet = tv.models.resnet18(pretrained=True)
```

In [52]:

```
print(resnet)
```

```
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_runni
ng_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, cei
l_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_r
unning_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_r
unning_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_r
unning_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_r
unning_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=Fals
e)
        (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=Fals
e)
          (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=Fals
e)
          (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)
  )
```

In [67]:

```
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
```

Question 16

In [53]:

```
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        #requires grad is false since we d
ont have to re train the network and calculate gradients for the previous layers

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

    def forward(self, x):
        y = self.model(x)
        return y
```

In [73]:

```
net = Resnet18Transfer(num_classes = train_set.number_of_classes())

print(net)
```

```
Resnet18Transfer(
  (cross_entropy): CrossEntropyLoss()
  (model): 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_run
ning_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, c
eil_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), paddin
g=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=Fal
se)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, trac
```

```
k_running_stats=True)
      )
    )
    (layer3): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), paddi
ng=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=Fa
lse)
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
      )
    )
    (layer4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), paddi
ng=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=Fa
lse)
          (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), paddi
ng=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
```

```
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=512, out_features=20, bias=True)
  )
)
```

Question 17

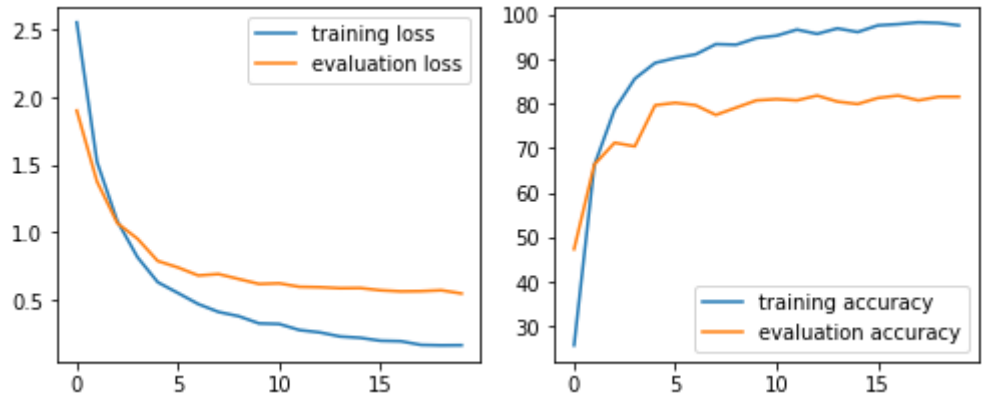
In [55]:

```
lr = 1e-3

net = Resnet18Transfer(num_classes =  train_set.number_of_classes() )

net = net.to(device)

adam = torch.optim.Adam(net.parameters(), lr=lr)

stats_manager = ClassificationStatsManager()

exp2 = nt.Experiment(net, train_set, val_set, adam, stats_manager,
output_dir="birdclass2", perform_validation_during_training=True)
```

In [80]:

```
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
```



Question 18

In [58]:

```
exp1.evaluate()     #VGG16 MODEL
```

Out[58]:

```
{'loss': 0.4920671442118676, 'accuracy': tensor(85.8696, device='cuda:0')}
```

In [77]:

```
exp2.evaluate()    #RESNET MODEL
```

Out[77]:

```
{'loss': 0.5473527072564416, 'accuracy': tensor(81.5217, device='cuda:0')}
```

We see that the validation accuracy of the VGG16 model is more compared to the validation accuracy of the RESNET18 model.