assignment3

November 11, 2019

1 Assignment 3 - Transfer Learning

1.0.1 Name: Anirudh Swaminathan

1.0.2 PID: A53316083

1.0.3 Email ID: aswamina@eng.ucsd.edu

Notebook created by Anirudh Swaminathan from ECE department majoring in Intelligent Systems, Robotics and Control for the course ECE285 Machine Learning for Image Processing for Fall 2019

1.1 Getting Started

[52]:
```
# The below line does NOT work as the PDFs generated do not have the Images
# %matplotlib notebook
%matplotlib inline

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
```

[2]:
```
# select the relevant device
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
cuda
```

1.2 Data Loader

Question 1

[3]:
```
dataset_root_dir = '/datasets/ee285f-public/caltech_ucsd_birds/'
```

[31]:
```
# Trying out getpass.getuser() and socket.gethostname()
import getpass
import socket

user = getpass.getuser()
hostname = socket.gethostname()
print(user)
print(hostname)
```

```
aswamina
aswamina-13662
```

We have created the *dataset_root_dir* and made it point to the Bird dataset directory

Question 2

[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

        # data is a pandas DataFrame
        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):
        # For the idxth entry, choose the value that is in the column file_path
        img_path = os.path.join(self.images_dir, self.data.
↪iloc[idx]['file_path'])

        # the bounding box coordinates are at the x1, y1, x2, and y2 columns
        bbox = self.data.iloc[idx][['x1', 'y1', 'x2', 'y2']]

        # DEBUG
```

```python
        # print(img_path)
        # print(bbox)

        # open the image
        img = Image.open(img_path).convert('RGB')
        img = img.crop([bbox[0], bbox[1], bbox[2], bbox[3]])
        transform = tv.transforms.Compose([
            # resize the image to image_size
            tv.transforms.Resize(self.image_size),

            # convert to torch tensor
            tv.transforms.ToTensor(),

            # Normalize each channel from [-1, 1]
            tv.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ])

        # apply the transform on the image
        x = transform(img)

        # access the data from the panda DataFrame at the idxth row and the␣
↪class column
        d = self.data.iloc[idx]['class']

        # DEBUG
        # print(d)
        return x, d

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

Completed the torchvision transforms compose function. I resize the image using $tv.transforms.Resize()$ function. Then the image is converted to a torch tensor using the $tv.transforms.ToTensor()$ function.

NOTE:-The $tv.trasforms.ToTensor()$ converts the PIL image from range $(0, 255)$ to a tensor of range $(0, 1)$

Finally, I normalize the image using the $tv.transforms.Normalize()$ function. This function takes means and standard deviations for each channel as the input. Since each channel has been transformed to $(0, 1)$ by the $tv.transforms.ToTensor()$ function, we have the mean for each channel is 0.5 and the standard deviation is 0.5. As given in the $PyTorch$ source code and documentation, the $tv.transforms.Normalize()$ function subtracts the mean for each channel from the image, and then divides by the standard deviation, so now the tensor in the range from $(0, 1)$ is converted to $\left(\frac{(0-0.5)}{0.5}, \frac{(1-0.5)}{0.5}\right)$, which is $(-1, 1)$.

Question 3

[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
```

[53]:
```
# train_set is an instance of BirdDataset
train_set = BirdsDataset(root_dir=dataset_root_dir)

# access the element at the 10th index
x, d_x = train_set.__getitem__(10)

# myimshow to display the obtained image
myimshow(x)
```

[53]: <matplotlib.image.AxesImage at 0x7f3d117a4cf8>

[32]:
```
print(type(x), x.min(), x.max(), x.dtype)
print(x.shape)
print(d_x, type(d_x), d_x.dtype)
```

```
<class 'torch.Tensor'> tensor(-0.9843) tensor(0.9294) torch.float32
torch.Size([3, 224, 224])
```

```
0 <class 'numpy.int64'> int64
```

Created the object *train_set* as an instance of *BirdsDataset*. I sampled the element at index 10 and stored it in the variable x. I finally used the *myimshow*() function that was defined to display the image x.

Question 4

```
[33]: train_loader = td.DataLoader(train_set, batch_size=16, shuffle=True, pin_memory=True)
```

```
[34]: print(type(train_loader), len(train_loader))
```

```
<class 'torch.utils.data.dataloader.DataLoader'> 47
```

I created *train_loader* that is defined to load the dataset. The *train_loader* object acts on the *train_set*.
I set the *batch_size* as 16 to sample minibatches of size 16. I set shuffle = *True* to have the data reshuffled at ea
The *pin_memory* was also set to *True* . Setting *pin_memory* to *True* enables fast data transfer to CUDA-enab
As given in the *PyTorch* documentation, host to GPU copies are much faster when they originate from pinned (
CPU tensors and storages expose a *pin_memory*() method, that returns a copy of the object, with data put in a
Since we have set the *drop_last* argument as *False* by default, the last minibatch is allowed to be of lower size t

Number of minibatches:-

$$\text{Number of training minibatches per epoch} = \frac{\text{Total number of training images}}{\text{Batch size}}$$

$$\text{Number of training minibatches per epoch} = \frac{743}{16}$$

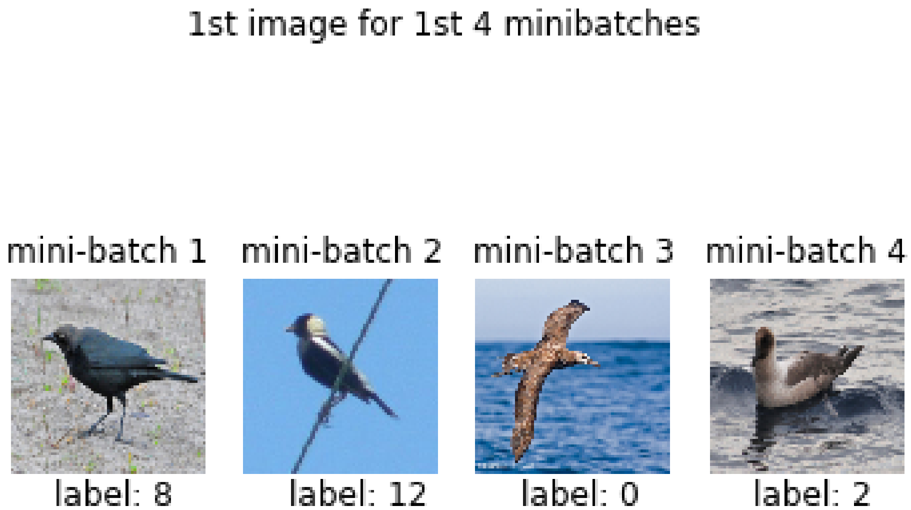
$$\text{Number of training minibatches per epoch} = 46 + 1 = 47$$

In one training epoch, we have 46 minibatches each of size 16 and the 47^{th} minibatch is of size 7.

Question 5

```
[55]: # Display 1st image and label pair for the 1st 4 minibatches
fig, axes = plt.subplots(ncols=4)
fig.suptitle("1st image for 1st 4 minibatches")

for bind, mbat in enumerate(train_loader):
    # print(len(mbat))
    # print(type(mbat[0]), type(mbat[1]))
    # print(mbat[0].size(), mbat[1].size())

    # The dataloader returns both the image and the label
    # image is in the 0th index, label is 1st index
    img = mbat[0][0, :, :, :]
    lab = mbat[1][0]
    # print(lab.item())
    myimshow(img, ax=axes[bind])
```

```
    axes[bind].text(50, 250, "label: {}".format(lab.item()), size=12, verticalalignment='center')
    # axes[bind].set_ylabel("label: {}".format(lab.item()))
    axes[bind].set_title("mini-batch {}".format(bind+1))
    if bind == 3:
        break
```



[56]:

```
# Display 1st image and label pair for the 1st 4 minibatches
fig, axes = plt.subplots(ncols=4)
fig.suptitle("1st image for 1st 4 minibatches")

for bind, mbat in enumerate(train_loader):
    # print(len(mbat))
    # print(type(mbat[0]), type(mbat[1]))
    # print(mbat[0].size(), mbat[1].size())

    # The dataloader returns both the image and the label
    # image is in the 0th index, label is 1st index
    img = mbat[0][0, :, :, :]
    lab = mbat[1][0]
    # print(lab.item())
    myimshow(img, ax=axes[bind])
    axes[bind].text(50, 250, "label: {}".format(lab.item()), size=12, verticalalignment='center')
    # axes[bind].set_ylabel("label: {}".format(lab.item()))
    axes[bind].set_title("mini-batch {}".format(bind+1))
    if bind == 3:
        break
```

I have displayed the 1^{st} image and label pair for the 1^{st} 4 mini-batches. I re-evaluated my cell, i.e., displayed the same information again in another cell. I obtained different results compared to running it for the 1^{st} time. This is because the *train_loader* creates a random minibatch shuffle every epoch.

Question 6

[7]:
```
val_set = BirdsDataset(root_dir=dataset_root_dir, mode="val")
```

[8]:
```
val_loader = td.DataLoader(val_set, batch_size=16, pin_memory=True)
```

I have created *val_set* as an instance of BirdsDataset using *mode* = "*val*". I then created *val_loader* to load the *val_set*. Shuffle is not required for the validation dataset as the parameters of the network are not affected by the validation set. Since the trained weights are just going to forward propagate the validation set images through the network just once to produce the outputs, we do not need to shuffle it. On the other hand, shuffling the data is required for the training set of images. This is because the network has to backpropogate the errors and learn the optimal network parameters using SGD. When using SGD, it is best to randomly shuffle the data to avoid local optima and to avoid training the network to recognize a particular sequence of inputs.

1.3 Abstract Neural Network Model

[9]:
```
import nntools as nt
```

[37]:
```
help(nt.NeuralNetwork)
```

```
Help on class NeuralNetwork in module nntools:

class NeuralNetwork(torch.nn.modules.module.Module, abc.ABC)
```

```
 |  An abstract class representing a neural network.
 |
 |  All other neural network should subclass it. All subclasses should override
 |  ``forward``, that makes a prediction for its input argument, and
 |  ``criterion``, that evaluates the fit between a prediction and a desired
 |  output. This class inherits from ``nn.Module`` and overloads the method
 |  ``named_parameters`` such that only parameters that require gradient
 |  computation are returned. Unlike ``nn.Module``, it also provides a property
 |  ``device`` that returns the current device in which the network is stored
 |  (assuming all network parameters are stored on the same device).
 |
 |  Method resolution order:
 |      NeuralNetwork
 |      torch.nn.modules.module.Module
 |      abc.ABC
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Initializes internal Module state, shared by both nn.Module and
ScriptModule.
 |
 |  criterion(self, y, d)
 |
 |  forward(self, x)
 |      Defines the computation performed at every call.
 |
 |      Should be overridden by all subclasses.
 |
 |      .. note::
 |          Although the recipe for forward pass needs to be defined within
 |          this function, one should call the :class:`Module` instance
afterwards
 |          instead of this since the former takes care of running the
 |          registered hooks while the latter silently ignores them.
 |
 |  named_parameters(self, recurse=True)
 |      Returns an iterator over module parameters, yielding both the
 |      name of the parameter as well as the parameter itself.
 |
 |      Args:
 |          prefix (str): prefix to prepend to all parameter names.
 |          recurse (bool): if True, then yields parameters of this module
 |              and all submodules. Otherwise, yields only parameters that
 |              are direct members of this module.
 |
 |      Yields:
```

```
|           (string, Parameter): Tuple containing the name and parameter
|
|       Example::
|
|           >>> for name, param in self.named_parameters():
|           >>>    if name in ['bias']:
|           >>>        print(param.size())
|
|   ----------------------------------------------------------------------
|   Data descriptors defined here:
|
|   device
|
|   ----------------------------------------------------------------------
|   Data and other attributes defined here:
|
|   __abstractmethods__ = frozenset({'criterion', 'forward'})
|
|   ----------------------------------------------------------------------
|   Methods inherited from torch.nn.modules.module.Module:
|
|   __call__(self, *input, **kwargs)
|       Call self as a function.
|
|   __delattr__(self, name)
|       Implement delattr(self, name).
|
|   __dir__(self)
|       Default dir() implementation.
|
|   __getattr__(self, name)
|
|   __repr__(self)
|       Return repr(self).
|
|   __setattr__(self, name, value)
|       Implement setattr(self, name, value).
|
|   __setstate__(self, state)
|
|   add_module(self, name, module)
|       Adds a child module to the current module.
|
|       The module can be accessed as an attribute using the given name.
|
|       Args:
|           name (string): name of the child module. The child module can be
|               accessed from this module using the given name
```

```
 |          module (Module): child module to be added to the module.
 |
 |  apply(self, fn)
 |      Applies ``fn`` recursively to every submodule (as returned by
``.children()``)
 |      as well as self. Typical use includes initializing the parameters of a
model
 |      (see also :ref:`torch-nn-init`).
 |
 |      Args:
 |          fn (:class:`Module` -> None): function to be applied to each
submodule
 |
 |      Returns:
 |          Module: self
 |
 |      Example::
 |
 |          >>> def init_weights(m):
 |          >>>     print(m)
 |          >>>     if type(m) == nn.Linear:
 |          >>>         m.weight.data.fill_(1.0)
 |          >>>         print(m.weight)
 |          >>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
 |          >>> net.apply(init_weights)
 |          Linear(in_features=2, out_features=2, bias=True)
 |          Parameter containing:
 |          tensor([[ 1.,  1.],
 |                  [ 1.,  1.]])
 |          Linear(in_features=2, out_features=2, bias=True)
 |          Parameter containing:
 |          tensor([[ 1.,  1.],
 |                  [ 1.,  1.]])
 |          Sequential(
 |            (0): Linear(in_features=2, out_features=2, bias=True)
 |            (1): Linear(in_features=2, out_features=2, bias=True)
 |          )
 |          Sequential(
 |            (0): Linear(in_features=2, out_features=2, bias=True)
 |            (1): Linear(in_features=2, out_features=2, bias=True)
 |          )
 |
 |  buffers(self, recurse=True)
 |      Returns an iterator over module buffers.
 |
 |      Args:
 |          recurse (bool): if True, then yields buffers of this module
 |              and all submodules. Otherwise, yields only buffers that
```

```
 |              are direct members of this module.
 |
 |      Yields:
 |          torch.Tensor: module buffer
 |
 |      Example::
 |
 |          >>> for buf in model.buffers():
 |          >>>     print(type(buf.data), buf.size())
 |          <class 'torch.FloatTensor'> (20L,)
 |          <class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
 |
 |  children(self)
 |      Returns an iterator over immediate children modules.
 |
 |      Yields:
 |          Module: a child module
 |
 |  cpu(self)
 |      Moves all model parameters and buffers to the CPU.
 |
 |      Returns:
 |          Module: self
 |
 |  cuda(self, device=None)
 |      Moves all model parameters and buffers to the GPU.
 |
 |      This also makes associated parameters and buffers different objects. So
 |      it should be called before constructing optimizer if the module will
 |      live on GPU while being optimized.
 |
 |      Arguments:
 |          device (int, optional): if specified, all parameters will be
 |              copied to that device
 |
 |      Returns:
 |          Module: self
 |
 |  double(self)
 |      Casts all floating point parameters and buffers to ``double`` datatype.
 |
 |      Returns:
 |          Module: self
 |
 |  eval(self)
 |      Sets the module in evaluation mode.
 |
 |      This has any effect only on certain modules. See documentations of
```

```
 |      particular modules for details of their behaviors in training/evaluation
 |      mode, if they are affected, e.g. :class:`Dropout`, :class:`BatchNorm`,
 |      etc.
 |
 |      This is equivalent with :meth:`self.train(False)
<torch.nn.Module.train>`.
 |
 |      Returns:
 |          Module: self
 |
 |  extra_repr(self)
 |      Set the extra representation of the module
 |
 |      To print customized extra information, you should reimplement
 |      this method in your own modules. Both single-line and multi-line
 |      strings are acceptable.
 |
 |  float(self)
 |      Casts all floating point parameters and buffers to float datatype.
 |
 |      Returns:
 |          Module: self
 |
 |  half(self)
 |      Casts all floating point parameters and buffers to ``half`` datatype.
 |
 |      Returns:
 |          Module: self
 |
 |  load_state_dict(self, state_dict, strict=True)
 |      Copies parameters and buffers from :attr:`state_dict` into
 |      this module and its descendants. If :attr:`strict` is ``True``, then
 |      the keys of :attr:`state_dict` must exactly match the keys returned
 |      by this module's :meth:`~torch.nn.Module.state_dict` function.
 |
 |      Arguments:
 |          state_dict (dict): a dict containing parameters and
 |              persistent buffers.
 |          strict (bool, optional): whether to strictly enforce that the keys
 |              in :attr:`state_dict` match the keys returned by this module's
 |              :meth:`~torch.nn.Module.state_dict` function. Default: ``True``
 |
 |      Returns:
 |          ``NamedTuple`` with ``missing_keys`` and ``unexpected_keys`` fields:
 |              * **missing_keys** is a list of str containing the missing keys
 |              * **unexpected_keys** is a list of str containing the unexpected
keys
 |
```

```
 |  modules(self)
 |      Returns an iterator over all modules in the network.
 |
 |      Yields:
 |          Module: a module in the network
 |
 |      Note:
 |          Duplicate modules are returned only once. In the following
 |          example, ``l`` will be returned only once.
 |
 |      Example::
 |
 |          >>> l = nn.Linear(2, 2)
 |          >>> net = nn.Sequential(l, l)
 |          >>> for idx, m in enumerate(net.modules()):
 |                  print(idx, '->', m)
 |
 |          0 -> Sequential(
 |            (0): Linear(in_features=2, out_features=2, bias=True)
 |            (1): Linear(in_features=2, out_features=2, bias=True)
 |          )
 |          1 -> Linear(in_features=2, out_features=2, bias=True)
 |
 |  named_buffers(self, prefix='', recurse=True)
 |      Returns an iterator over module buffers, yielding both the
 |      name of the buffer as well as the buffer itself.
 |
 |      Args:
 |          prefix (str): prefix to prepend to all buffer names.
 |          recurse (bool): if True, then yields buffers of this module
 |              and all submodules. Otherwise, yields only buffers that
 |              are direct members of this module.
 |
 |      Yields:
 |          (string, torch.Tensor): Tuple containing the name and buffer
 |
 |      Example::
 |
 |          >>> for name, buf in self.named_buffers():
 |          >>>    if name in ['running_var']:
 |          >>>        print(buf.size())
 |
 |  named_children(self)
 |      Returns an iterator over immediate children modules, yielding both
 |      the name of the module as well as the module itself.
 |
 |      Yields:
 |          (string, Module): Tuple containing a name and child module
```

```
 |
 |      Example::
 |
 |          >>> for name, module in model.named_children():
 |          >>>     if name in ['conv4', 'conv5']:
 |          >>>         print(module)
 |
 |  named_modules(self, memo=None, prefix='')
 |      Returns an iterator over all modules in the network, yielding
 |      both the name of the module as well as the module itself.
 |
 |      Yields:
 |          (string, Module): Tuple of name and module
 |
 |      Note:
 |          Duplicate modules are returned only once. In the following
 |          example, ``l`` will be returned only once.
 |
 |      Example::
 |
 |          >>> l = nn.Linear(2, 2)
 |          >>> net = nn.Sequential(l, l)
 |          >>> for idx, m in enumerate(net.named_modules()):
 |                  print(idx, '->', m)
 |
 |          0 -> ('', Sequential(
 |            (0): Linear(in_features=2, out_features=2, bias=True)
 |            (1): Linear(in_features=2, out_features=2, bias=True)
 |          ))
 |          1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
 |
 |  parameters(self, recurse=True)
 |      Returns an iterator over module parameters.
 |
 |      This is typically passed to an optimizer.
 |
 |      Args:
 |          recurse (bool): if True, then yields parameters of this module
 |              and all submodules. Otherwise, yields only parameters that
 |              are direct members of this module.
 |
 |      Yields:
 |          Parameter: module parameter
 |
 |      Example::
 |
 |          >>> for param in model.parameters():
 |          >>>     print(type(param.data), param.size())
```

```
 |          <class 'torch.FloatTensor'> (20L,)
 |          <class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
 |
 |  register_backward_hook(self, hook)
 |      Registers a backward hook on the module.
 |
 |      The hook will be called every time the gradients with respect to module
 |      inputs are computed. The hook should have the following signature::
 |
 |          hook(module, grad_input, grad_output) -> Tensor or None
 |
 |      The :attr:`grad_input` and :attr:`grad_output` may be tuples if the
 |      module has multiple inputs or outputs. The hook should not modify its
 |      arguments, but it can optionally return a new gradient with respect to
 |      input that will be used in place of :attr:`grad_input` in subsequent
 |      computations.
 |
 |      Returns:
 |          :class:`torch.utils.hooks.RemovableHandle`:
 |              a handle that can be used to remove the added hook by calling
 |              ``handle.remove()``
 |
 |      .. warning ::
 |
 |          The current implementation will not have the presented behavior
 |          for complex :class:`Module` that perform many operations.
 |          In some failure cases, :attr:`grad_input` and :attr:`grad_output`
will only
 |          contain the gradients for a subset of the inputs and outputs.
 |          For such :class:`Module`, you should use
:func:`torch.Tensor.register_hook`
 |          directly on a specific input or output to get the required
gradients.
 |
 |  register_buffer(self, name, tensor)
 |      Adds a persistent buffer to the module.
 |
 |      This is typically used to register a buffer that should not to be
 |      considered a model parameter. For example, BatchNorm's ``running_mean``
 |      is not a parameter, but is part of the persistent state.
 |
 |      Buffers can be accessed as attributes using given names.
 |
 |      Args:
 |          name (string): name of the buffer. The buffer can be accessed
 |              from this module using the given name
 |          tensor (Tensor): buffer to be registered.
 |
```

```
 |      Example::
 |
 |          >>> self.register_buffer('running_mean', torch.zeros(num_features))
 |
 |  register_forward_hook(self, hook)
 |      Registers a forward hook on the module.
 |
 |      The hook will be called every time after :func:`forward` has computed an
output.
 |      It should have the following signature::
 |
 |          hook(module, input, output) -> None or modified output
 |
 |      The hook can modify the output. It can modify the input inplace but
 |      it will not have effect on forward since this is called after
 |      :func:`forward` is called.
 |
 |      Returns:
 |          :class:`torch.utils.hooks.RemovableHandle`:
 |              a handle that can be used to remove the added hook by calling
 |              ``handle.remove()``
 |
 |  register_forward_pre_hook(self, hook)
 |      Registers a forward pre-hook on the module.
 |
 |      The hook will be called every time before :func:`forward` is invoked.
 |      It should have the following signature::
 |
 |          hook(module, input) -> None or modified input
 |
 |      The hook can modify the input. User can either return a tuple or a
 |      single modified value in the hook. We will wrap the value into a tuple
 |      if a single value is returned(unless that value is already a tuple).
 |
 |      Returns:
 |          :class:`torch.utils.hooks.RemovableHandle`:
 |              a handle that can be used to remove the added hook by calling
 |              ``handle.remove()``
 |
 |  register_parameter(self, name, param)
 |      Adds a parameter to the module.
 |
 |      The parameter can be accessed as an attribute using given name.
 |
 |      Args:
 |          name (string): name of the parameter. The parameter can be accessed
 |              from this module using the given name
 |          param (Parameter): parameter to be added to the module.
```

```
 |
 |  requires_grad_(self, requires_grad=True)
 |      Change if autograd should record operations on parameters in this
 |      module.
 |
 |      This method sets the parameters' :attr:`requires_grad` attributes
 |      in-place.
 |
 |      This method is helpful for freezing part of the module for finetuning
 |      or training parts of a model individually (e.g., GAN training).
 |
 |      Args:
 |          requires_grad (bool): whether autograd should record operations on
 |                                parameters in this module. Default: ``True``.
 |
 |      Returns:
 |          Module: self
 |
 |  share_memory(self)
 |
 |  state_dict(self, destination=None, prefix='', keep_vars=False)
 |      Returns a dictionary containing a whole state of the module.
 |
 |      Both parameters and persistent buffers (e.g. running averages) are
 |      included. Keys are corresponding parameter and buffer names.
 |
 |      Returns:
 |          dict:
 |              a dictionary containing a whole state of the module
 |
 |      Example::
 |
 |          >>> module.state_dict().keys()
 |          ['bias', 'weight']
 |
 |  to(self, *args, **kwargs)
 |      Moves and/or casts the parameters and buffers.
 |
 |      This can be called as
 |
 |      .. function:: to(device=None, dtype=None, non_blocking=False)
 |
 |      .. function:: to(dtype, non_blocking=False)
 |
 |      .. function:: to(tensor, non_blocking=False)
 |
 |      Its signature is similar to :meth:`torch.Tensor.to`, but only accepts
 |      floating point desired :attr:`dtype` s. In addition, this method will
```

```
|       only cast the floating point parameters and buffers to :attr:`dtype`
|       (if given). The integral parameters and buffers will be moved
|       :attr:`device`, if that is given, but with dtypes unchanged. When
|       :attr:`non_blocking` is set, it tries to convert/move asynchronously
|       with respect to the host if possible, e.g., moving CPU Tensors with
|       pinned memory to CUDA devices.
|
|       See below for examples.
|
|       .. note::
|           This method modifies the module in-place.
|
|       Args:
|           device (:class:`torch.device`): the desired device of the parameters
|               and buffers in this module
|           dtype (:class:`torch.dtype`): the desired floating point type of
|               the floating point parameters and buffers in this module
|           tensor (torch.Tensor): Tensor whose dtype and device are the desired
|               dtype and device for all parameters and buffers in this module
|
|       Returns:
|           Module: self
|
|       Example::
|
|           >>> linear = nn.Linear(2, 2)
|           >>> linear.weight
|           Parameter containing:
|           tensor([[ 0.1913, -0.3420],
|                   [-0.5113, -0.2325]])
|           >>> linear.to(torch.double)
|           Linear(in_features=2, out_features=2, bias=True)
|           >>> linear.weight
|           Parameter containing:
|           tensor([[ 0.1913, -0.3420],
|                   [-0.5113, -0.2325]], dtype=torch.float64)
|           >>> gpu1 = torch.device("cuda:1")
|           >>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
|           Linear(in_features=2, out_features=2, bias=True)
|           >>> linear.weight
|           Parameter containing:
|           tensor([[ 0.1914, -0.3420],
|                   [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
|           >>> cpu = torch.device("cpu")
|           >>> linear.to(cpu)
|           Linear(in_features=2, out_features=2, bias=True)
|           >>> linear.weight
|           Parameter containing:
```

```
 |          tensor([[ 0.1914, -0.3420],
 |                  [-0.5112, -0.2324]], dtype=torch.float16)
 |
 |  train(self, mode=True)
 |      Sets the module in training mode.
 |
 |      This has any effect only on certain modules. See documentations of
 |      particular modules for details of their behaviors in training/evaluation
 |      mode, if they are affected, e.g. :class:`Dropout`, :class:`BatchNorm`,
 |      etc.
 |
 |      Args:
 |          mode (bool): whether to set training mode (``True``) or evaluation
 |                       mode (``False``). Default: ``True``.
 |
 |      Returns:
 |          Module: self
 |
 |  type(self, dst_type)
 |      Casts all parameters and buffers to :attr:`dst_type`.
 |
 |      Arguments:
 |          dst_type (type or string): the desired type
 |
 |      Returns:
 |          Module: self
 |
 |  zero_grad(self)
 |      Sets gradients of all model parameters to zero.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from torch.nn.modules.module.Module:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from torch.nn.modules.module.Module:
 |
 |  dump_patches = False
```

Question 7

[38]:
```
net = nt.NeuralNetwork()
```

```
    TypeErrorTraceback (most recent call last)

    <ipython-input-38-2b13b95b774d> in <module>
----> 1 net = nt.NeuralNetwork()


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

I observe that on running the above line of code, I am getting a *TypeError*. The exact error message is that I can't instantiate abstract class NeuralNetwork with abstract methods criterion and forward. As given in the question, an abstract class does not implement all of its methods and cannot be instantiated.

[10]:
```
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)
```

We have also defined the *NNClassifier* that inherits from *NeuralNetwork*, and defines only the *criterion*. Here, the *criterion* method is implemented to be the cross-entropy loss. But this class is still abstract as it does not implement the *forward* method.

1.4 VGG-16 Transfer Learning

Question 8

[39]:
```
vgg = tv.models.vgg16_bn(pretrained=True)
```

[40]:
```
# print the network
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_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)
  )
)
```

I have thus printed the network architecture using the $print(vgg)$ function.

[41]:
```python
# print the named parameters of the network
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
```

I inspected all the named parameters of the network, and I find that all of the parameters are learnable parameters, as the *requires_grad* attribute is set to *True* for all of them.

Question 9

[11]:
```
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():
            # if we do not want to fine tune the network, then fine_tuning=False
            # this means that we need to freeze these VGG16 pretrained layers
            # and NOT train them when not fine-tuning
            param.requires_grad = fine_tuning
        self.features = vgg.features

        # COMPLETE
        # the average pooling is the same
        self.avgpool = vgg.avgpool
        # the classifier is also the same
        self.classifier = vgg.classifier

        # CODE to change the final classifier layer
        num_ftrs = vgg.classifier[6].in_features
        self.classifier[6] = nn.Linear(num_ftrs, num_classes)

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

I have created the subclass $VGG16Transfer$ that inherits from $NNClassifier$. I copied the layers of the $VGG16$ pretrained network to my classifier. This included the Sequential features, the average pooling layer, and the Sequential classifier. I modified the final layer of my classifier to be trained specific to my task. I finally implemented the $forward()$ method of my network and thus, this class is no longer an abstract class.

Question 10

[12]:
```
num_classes = train_set.number_of_classes()
print(num_classes)
```

```
20
```

[13]:
```
net = VGG16Transfer(num_classes)
```

```
Downloading: "https://download.pytorch.org/models/vgg16_bn-6c64b313.pth" to
/tmp/xdg-cache/torch/checkpoints/vgg16_bn-6c64b313.pth
100%|     | 528M/528M [00:12<00:00, 43.5MB/s]
```

[42]:
```python
# print the network
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_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)
  )
)
```

I have thus printed the network architecture using the $print(net)$ function. The difference between this network and the VGG pretrained net is that there is a *cross_entropy* criterion for the loss. Also, the final layer in the classifier is defined to be connecting 4096 to 20, which is the number of classes instead of 1000 as defined in the original VGG network.

[43]:
```
# print the named parameters of the network
for name, param in net.named_parameters():
    print(name, param.size(), param.requires_grad)
```

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

I inspected all the named parameters of the network. The only learnable parameters are for the final Fully Connected (FC) layer of the classifier, which is specific to our given task, and is different from the pre-trained VGG network. This is because we had explicitly set the *requires_grad* attribute of all the other parameters of the network to *False* to avoid training the entire network and to leverage the pre-trained weights for these layers. Since we modified the final Fully Connected Layer of the classifier, we created a new *nn.Linear* layer, whose *requires_grad* is by default set to *True*, and thus only that layer is learnable for our network.

1.5 Training experiment and checkpoints

Question 11

[14]:
```python
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)

        # Gets the indices of the maximum activation of softmax for each sample
        _, l = torch.max(y, 1)

        # count the running average fraction of correctly classified samples
        self.running_accuracy += torch.mean((l == d).float())

    def summarize(self):
        # this is the average loss when called
        loss = super(ClassificationStatsManager, self).summarize()

        # this is the average accuracy percentage when called
        accuracy = 100 * self.running_accuracy / self.number_update
        return {'loss' : loss, 'accuracy' : accuracy}
```

Created a subclass *ClassificationStatsManager* that inherits from *StatsManager* and overloads each method. The additional information apart from the running loss is the running accuracy that is also being tracked here. In *init*(), the running accuracy is set to 0. The *accumulate*() method adds the mean accuracy for each minibatch to the running accuracy. Finally, the *summarize*() method is overloaded to set the accuracy to the average over all the epochs/updates.

Question 12 I read the documentation for the *evaluate*() method. *self.net* is set to *eval* mode by the *eval*() function. This ensures that the model is in evaluation mode while testing. This is to ensure that only those modules like Dropout, Batchnorm, etc. that behave differently during training and testing behave correctly. For example, Dropout, as given in the documentation, during training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

This means that during evaluation the module simply computes an identity function. Hence, to ensure that this behaviour is followed, we call the *eval*() function on the module first. Once the *evaluate*() function is computed, we again set the network to the train mode using the *train*() method so that it can continue with the training.

Question 13

[15]:
```
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)
```

I have checked that the directory *birdclass*1 has been created. Inspecting its contents, I find that it contains 2 files, namely *checkpoint.pth.tar* and *config.txt*. Visualizing the *config.txt* file, and also from the code in the *nntools.py* class, I find that it corresponds to the settings of the experiment. So, *config.txt* contains the network parameters, the training set, the validation set, the optimizer, the stats manager I am using, the batch size, and the boolean value for *perform_validation_during_training*. The file *checkpoint.pth.tar* corresponds to saving the *state_dict*() of the model. It consists of the dictionary of key-value pairs. The 'Net' attribute corresponds to the networks *state_dict*(), the 'Optimizer' corresponds to the optimizers *state_dict*() and 'History' corresponds to the history of the training of the network. The *checkpoint.pth.tar* file was saved using the *torch.save*() function that saves the models *save_dict*() to the file with the name contained in the variable *checkpoint_path*.

Question 14

[44]:
```
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-44-c9e959da5e85> in <module>
      6 exp1 = nt.Experiment(net, train_set, val_set, adam,
      7                      stats_manager, output_dir="birdclass1",
----> 8                      perform_validation_during_training=True)
```

```
    /datasets/home/home-02/60/660/aswamina/ece285_mlip_fa19/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.
```

On running the same code with just a different learning rate (LR), we have accessed the same folder and the same checkpoint file. Since the settings here are different, this means that we are supposed to create a separate folder for the same, as it is a different experiment. Hence, a $ValueError$ is raised, with the error message **"Cannot create this experiment: I found a checkpoint conflicting with the current setting."**.

[45]:
```
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)
```

On changing the learning rate(LR) back to 1^{-3}, we can access the same checkpoint folder and file, as all the network settings are the same, including the learning rate. This means that the checkpoint is now just loaded onto the program, as there is no conflict with the settings of the network.

Question 15

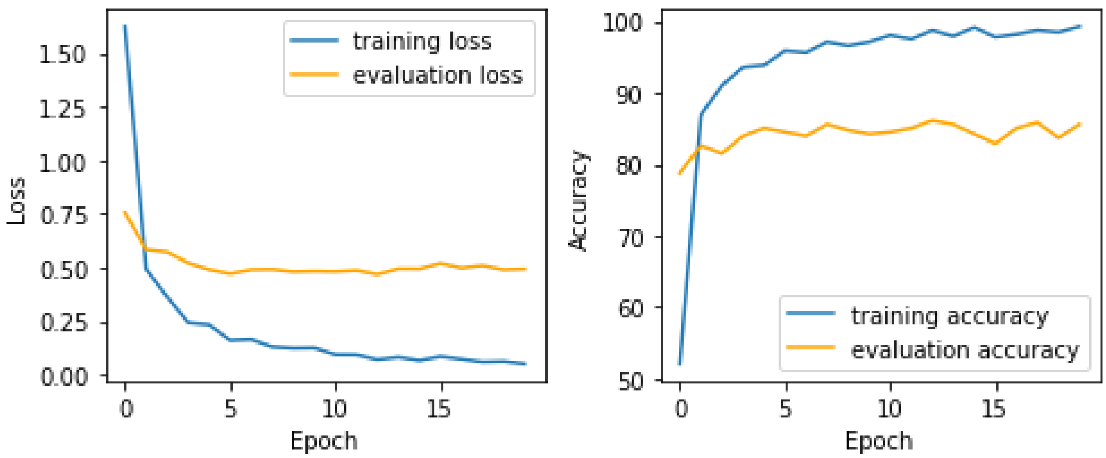
[16]:
```
def plot(exp, fig, axes):
    axes[0].clear()
    axes[1].clear()

    # Plot the training loss over the epochs
    axes[0].plot([exp.history[k][0]['loss'] for k in range(exp.epoch)],
                 label="training loss")

    # Plot the evaluation loss over the epochs
```

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

    # legend for the plot
    axes[0].legend()

    # xlabel and ylabel
    axes[0].set_xlabel("Epoch")
    axes[0].set_ylabel("Loss")

    # Plot the training accuracy over the epochs
    axes[1].plot([exp.history[k][0]['accuracy'] for k in range(exp.epoch)],
                 label="training accuracy")

    # Plot the evaluation accuracy over the epochs
    axes[1].plot([exp.history[k][1]['accuracy'] for k in range(exp.epoch)],
                 color='orange', label="evaluation accuracy")

    # legend for the plot
    axes[1].legend()

    # xlabel and ylabel
    axes[1].set_xlabel("Epoch")
    axes[1].set_ylabel("Accuracy")

    plt.tight_layout()

    # set the title for the figure
    # fig.suptitle("Loss and Accuracy metrics")
    fig.canvas.draw()
```

I have completed the $plot()$ function to plot the different metrics for 20 epochs. I access the k^{th} epoch using the $history[k]$ index. I then access the metrics evaluated on the training set using the 0^{th} index. I access the metrics evaluated on the validation set using the 1^{st} index. To access the loss, we use the loss as key value for the dictionary. To access the accuracy, we use accuracy as the key value for the dictionary. To get the orange colour for the plot, we use the $color = "orange"$ as another parameter in the $plot()$ function in the axes. To set the X Label and Y Label for each subplot, we use the $set\_xlabel()$ and the $set\_ylabel()$ methods respectively. To set the legend for the subplots, since we have already specified the $label$ for each subplot, we just need to call the $legend()$ function for each subplot.

[57]:
```
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
```



I have run the 1^{st} experiment using $VGG16$ for Transfer Learning on the GPU, with Adam Optimizer and with learning rate $= 1^{-3}$. The training has been completed for 20 epochs, and we have got 2 plots, one each for loss and accuracy for the training and the validation set with the number of epochs. For each epoch, it takes about 25 seconds to run on the GPU. The loss evolutions as well as the accuracy evolutions are found to be consistent with the ones given to us.

1.6 ResNet18 Transfer Learning

[46]:
```
resnet = tv.models.resnet18(pretrained=True)
```

[47]:
```
# print the network
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_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)
)
```

I have thus printed the network architecture using the $print(resnet)$ function.

[48]:
```python
# print the named parameters of the network
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

[18]:
```python
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():
            # if we do not want to fine tune the network, then fine_tuning=False
            # this means that we need to freeze these VGG16 pretrained layers
            # and NOT train them when not fine-tuning
            param.requires_grad = fine_tuning

        # network definition
        self.conv1 = resnet.conv1
        self.bn1 = resnet.bn1
```

```python
        self.relu = resnet.relu
        self.maxpool = resnet.maxpool

        # the "layers" now
        self.layer1 = resnet.layer1
        self.layer2 = resnet.layer2
        self.layer3 = resnet.layer3
        self.layer4 = resnet.layer4

        # the average pooling is the same
        self.avgpool = resnet.avgpool

        # the classifier is also the same
        self.fc = resnet.fc

        # CODE to change the FC layer
        num_ftrs = resnet.fc.in_features
        self.fc = nn.Linear(num_ftrs, num_classes)

    def forward(self, x):
        # forward prop through the network
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

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

        a = self.avgpool(f)
        a = torch.flatten(a, 1)
        y = self.fc(a)
        return y
```

I created the $Resnet18Transfer$ class that inherits from $NNClassifier$. I copied the layers of the $Resnet18$ pretrained network to my classifier. I modified the final fully connected (FC) layer of my classifier to be trained specific to my task. I finally implemented the $forward()$ method of my network and thus, this class is no longer an abstract class.

Question 17

[49]:
```python
re_net = Resnet18Transfer(num_classes)
```

[50]:
```python
print(re_net)
```

```
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)
)
```

I have thus printed the network architecture using the $print(re\_net)$ function. The difference between this network and the $Resnet18$ pretrained net is that there is a $cross\_entropy$ criterion for the loss. Also, the final layer in the classifier is defined to be connecting 512 to 20, which is the number of classes instead of 1000 as defined in the original Resnet18 network.

[51]:
```
# print the named parameters of the network
for name, param in re_net.named_parameters():
    print(name, param.size(), param.requires_grad)
```

```
fc.weight torch.Size([20, 512]) True
fc.bias torch.Size([20]) True
```

I inspected all the named parameters of the network. The only learnable parameters are for the final Fully Connected (FC) layer of the classifier, which is specific to our given task, and is different from the pre-trained $Resnet18$ network. This is because we had explicitly set the $requires\_grad$ attribute of all the other parameters of the network to $False$ to avoid training the entire network and to leverage the pre-trained weights for these layers. Since we modified the final Fully Connected Layer of the classifier, we created a new *nn.Linear* layer, whose $requires\_grad$ is by default set to $True$, and thus only that layer is learnable for our network.

[19]:
```
lr = 1e-3
re_net = Resnet18Transfer(num_classes)
```

```
re_net = re_net.to(device)
adam = torch.optim.Adam(re_net.parameters(), lr=lr)
stats_manager = ClassificationStatsManager()
exp2 = nt.Experiment(re_net, train_set, val_set, adam,
                     stats_manager, output_dir="birdclass2",
                     perform_validation_during_training=True)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to
/tmp/xdg-cache/torch/checkpoints/resnet18-5c106cde.pth
100%|      | 44.7M/44.7M [00:01<00:00, 30.4MB/s]
```

I have checked that the directory *birdclass*2 has been created. Inspecting its contents, I find that it contains 2 files, namely *checkpoint.pth.tar* and *config.txt*. Visualizing the *config.txt* file, and also from the code in the *nntools.py* class, I find that it corresponds to the settings of the experiment. So, *config.txt* contains the network parameters, the training set, the validation set, the optimizer, the stats manager I am using, the batch size, and the boolean value for *perform_validation_during_training*. The file *checkpoint.pth.tar* corresponds to saving the *state_dict*() of the model. It consists of the dictionary of key-value pairs. The 'Net' attribute corresponds to the networks *state_dict*(), the 'Optimizer' corresponds to the optimizers *state_dict*() and 'History' corresponds to the history of the training of the network. The *checkpoint.pth.tar* file was saved using the *torch.save*() function that saves the models *save_dict*() to the file with the name contained in the variable *checkpoint_path*.

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

```
<IPython.core.display.Javascript object>
```

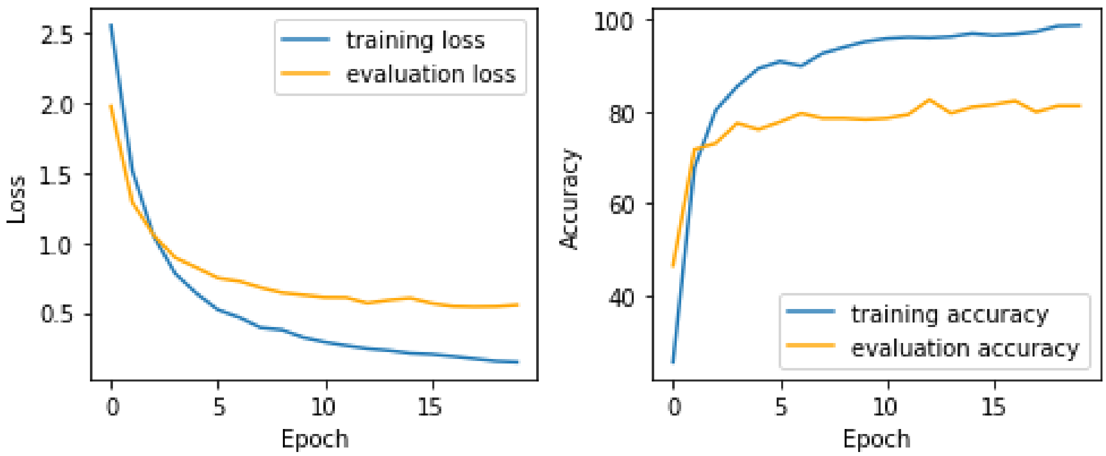
```
<IPython.core.display.HTML object>
```

```
Start/Continue training from epoch 0
Epoch 1 (Time: 26.46s)
Epoch 2 (Time: 25.85s)
Epoch 3 (Time: 26.23s)
Epoch 4 (Time: 26.25s)
Epoch 5 (Time: 25.78s)
Epoch 6 (Time: 25.83s)
Epoch 7 (Time: 25.86s)
Epoch 8 (Time: 25.78s)
Epoch 9 (Time: 26.17s)
Epoch 10 (Time: 25.87s)
Epoch 11 (Time: 26.13s)
Epoch 12 (Time: 26.42s)
Epoch 13 (Time: 26.36s)
Epoch 14 (Time: 26.34s)
Epoch 15 (Time: 25.88s)
Epoch 16 (Time: 26.11s)
```

```
Epoch 17 (Time: 26.39s)
Epoch 18 (Time: 26.16s)
Epoch 19 (Time: 26.34s)
Epoch 20 (Time: 25.99s)
Finish training for 20 epochs
```

[58]:
```
# I'm running this cell again because matplotlib inline does not work!
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
```



I have run the 2^{nd} experiment using $ResNet18$ for Transfer Learning on the GPU, with Adam Optimizer and with learning rate $= 1^{-3}$. The training has been completed for 20 epochs, and we have got 2 plots, one each for loss and accuracy for the training and the validation set with the number of epochs. For each epoch, it takes about 26 seconds to run on the GPU. The loss evolutions as well as the accuracy evolutions are found to be consistent with the ones given to us.

Question 18

[21]:
```
# Compute the validation performance obtained by exp1
vgg_val = exp1.evaluate()
print(vgg_val)
```

```
{'loss': 0.4929845600753375, 'accuracy': tensor(85.5978, device='cuda:0')}
```

[29]:
```
print("Loss of network on validation set using VGG16 as the frozen model after␣
↪20 epochs is %.3f" % vgg_val['loss'])
print("Accuracy of the network on validation set using VGG16 as the frozen␣
↪model after 20 epochs is %.3f" % vgg_val['accuracy'].item(), "%")
```

```
Loss of network on validation set using VGG16 as the frozen model after 20
epochs is 0.493
Accuracy of the network on validation set using VGG16 as the frozen model after
20 epochs is 85.598 %
```

[22]:
```python
# Compute the validation performance  obtained by exp2
resnet_val = exp2.evaluate()
print(resnet_val)
```

```
{'loss': 0.5576237161522326, 'accuracy': tensor(81.2500, device='cuda:0')}
```

[30]:
```python
print("Loss of network on validation set using Resnet18 as the frozen model␣
 ↪after 20 epochs is %.3f" % resnet_val['loss'])
print("Accuracy of the network on validation set using Resnet18 as the frozen␣
 ↪model after 20 epochs is %.3f" % resnet_val['accuracy'].item(), "%")
```

```
Loss of network on validation set using Resnet18 as the frozen model after 20
epochs is 0.558
Accuracy of the network on validation set using Resnet18 as the frozen model
after 20 epochs is 81.250 %
```

We used the $evaluate()$ method of $Experiment$ to compare the validation performance obtained by $exp1$ and $exp2$ using $VGG16Transfer$ and $Resnet18Transfer$ respectively.

1.6.1 Observations

Loss for the trained networks on the validation set

The loss for $VGG16$ pre-trained network on the validation set after 20 epochs is 0.493. The loss for $Resnet18$ pre-trained network on the validation set after 20 epochs is 0.558.

Accuracy for the trained networks on the validation set

The accuracy for $VGG16$ pre-trained network on the validation set after 20 epochs is 85.598%. The accuracy for $Resnet18$ pre-trained network on the validation set after 20 epochs is 81.250%.

1.6.2 Inferences

We infer that the $VGG16$ pre-trained network performs better for the same Learning rate and Adam optimization for our particular task of classifying birds. The $Resnet18$ pre-trained network on the other hand has worse loss on the validation set, i.e., its validation loss is higher compared to the validation loss of $VGG16$ pre-trained network on the same dataset. Likewise, it has worse classification accuracy on the validation set, i.e., its validation accuracy is lower compared to the validation accuracy of $VGG16$ pre-trained network on the same dataset.

1.6.3 Conclusion

Thus, we have explored a new dataset this homework, which is the Caltech-UCSD Birds Dataset. We have explored *PyTorch Dataset* class, *DataLoader*, and how to define and use abstract neural network module classes, and then implement a specific network for our own purposes. We then learnt how to load pre-trained models such as $VGG16$ and $Resnet18$ using *PyTorch*. We then used these models in transfer learning to specifically work for our limited dataset by freezing all the layers except the final one. Finally, we learnt how to create checkpoints to stop and resume model training.

Assignment completed by:-Name: Anirudh Swaminathan PID: A53316083 Email ID: aswamina@eng.ucsd.edu