Lab 6B - Custom Dataset and Scheduler

In this lab, we shall learn to implement the following two things:

1. Build a custom dataset with your own data
2. Perform learning rate scheduling

In [ ]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

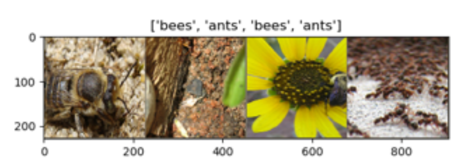
In [ ]:
```
cd '/content/gdrive/My Drive/UCCD3074_Lab6'
```

In [1]:
```
import os
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.models as models
from torch.utils.data import Dataset

from PIL import Image
```

---

1. The Hymenoptera Dataset

The problem we're going to solve today is to train a model to classify **ants** and **bees**. We have about 120 training images each for ants and bees. There are 75 validation images for each class. Usually, this is a very small dataset to generalize upon, if trained from scratch. Since we are using transfer learning, we should be able to generalize reasonably well. This dataset is a very small subset of imagenet.



In [ ]:
```
!wget https://download.pytorch.org/tutorial/hymenoptera_data.zip
!unzip -q hymenoptera_data.zip
rm 'hymenoptera_data/train/ants/imageNotFound.gif'
```

Take a look at the folder `hymenoptera_data` . It has the following directory structure:

```
hymenoptera_data\
   train\
      ants\
      bees\
   val\
      ants\
      bees\
```

2. Writing custom dataset

2.1 The Abstract `Dataset` Class

PyTorch provides `torch.utils.data.Dataset` to allow you create your own custom dataset. `Dataset` is an abstract class representing a dataset. Your custom dataset should inherit `Dataset` and override the following methods:

- **`__len__`** so that len(dataset) returns the size of the dataset.
- **`__getitem__`** to support the indexing such that dataset[i] can be used to get ith sample

The following code creates a dataset class for the hymenoptera dataset.

In [8]:
```python
class HymenopteraDataset(Dataset):

    def __init__(self, root, transform=None):
        self.data = []
        self.labels = []
        self.transform = transform
        self.classes = ['ants', 'bees']

        # get the training samples
        for class_id, cls in enumerate(self.classes):

            cls_folder = os.path.join(root, cls)

            # get the training samples for the class 'cls'
            for img_name in os.listdir(cls_folder):
                self.data.append(os.path.join(cls_folder, img_name))
                self.labels.append(class_id)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):

        # get the image
        image = Image.open(self.data[idx])

        # perform transformation
        if self.transform is not None:
            image = self.transform(image)

        # get the label
        label = self.labels[idx]

        # return sample
        return image, label
```

- **`__init__`** : Get the filenames of all training samples (`self.data`) and their corresponding labels (`self.labels`)
  - Line 10:
    If `transform` is passed by the user, all images would be transformed using this pipeline when they are read in `__getitem__` later.
  - Line 11:
    There are 2 classes in the dataset (0: ants, 1: bees)
  - Line 14-21:
    For each of the class (line 14), get the names of all the files in their class directories (line 19) and update `self.data` (line 20) and `self.labels` (line 21).
- **`__getitem__`** : Read the image and label. Transform the image if required. Return the transformed image and label.

Notes: *While it is possible to load all images in the `__init__` , we have choosen to read the images only when requested by the user in `__getitem__` . This is more memory efficient because all the images are not stored in the memory at once but read as required. This is the normal setup when the dataset is huge.*

2.2 Instantiating `HymenopteraDataset`

Let's instantiate the HymenopteraDataset and look into one of its sample.

In [9]:
```
trainset = HymenopteraDataset('./hymenoptera_data/train', transform=None)

print('Number of samples in dataset:', len(trainset))
print('Number of classes:', trainset.classes)
```

```
Number of samples in dataset: 244
Number of classes: ['ants', 'bees']
```

- `Line 1` : When creating `trainset` , the function `__init__` will be called to populate `trainset.data` and `trainset.labels` .

Next, we look into the first sample in the label. Since we did not transform the image, we can still display the image without undoing the transformation.

In [12]:
```
image, label = trainset[1]
display(image)
print("Class =", trainset.classes[label])
```

```
Class = ants
```

3.1 Customizing ResNet18 for Binary Classification

Now, customize ResNet18 (`torchvision.models.resnet18`) to build a classifier to differentiate between *ants* vs *bees*.

Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2 $\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

The ResNet18 receives an input of size 18x18 and it outputs a vector of 1000 dimensions since it was pretrained on the ImageNet with 1000 object categories.

In PyTorch implementation, the layers are defined as:

Layer Name	Name in `torchvision.models.resnet18`
conv1	conv1, bn1, relu, maxpool
conv2_x	layer 1
conv3_x	layer 2
conv4_x	layer 3
conv5_x	layer 4
average pool	avgpool
fully connected	fc
softmax	Not implemented. Softmax is implemented in `CrossEntropy` loss

Exercise:

Customize *resnet18* for a binary classification task. Replace the *fc* layer with the following layers with the following two layers:

- nn.Linear(512, 1)
- nn.Sigmoid()

You may group them into a `nn.Sequential` module.

Expected result:

```
ResNet(
    (conv1): ...
    (bn1): ...
    (relu): ...
    (maxpool): ...
    (layer1): ...
    (layer2): ...
    (layer3): ...
    (layer4): ...
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Sequential(
        (0): Linear(in_features=512, out_features=1, bias=True)
        (1): Sigmoid()
    )
)
```

In [13]:
```
################################################################################
##
# START OF CODE: Customize resnet18 for a binary classification task
################################################################################
##
model = models.resnet18(pretrained=True)
model.fc = nn.Sequential(nn.Linear(512, 1), nn.Sigmoid())
################################################################################
##
#       END OF CODE
################################################################################
##

print(model)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), b
ias=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_m
ode=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_runn
ing_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_runn
ing_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_runn
ing_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_runn
ing_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_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_run
ning_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
    (0): Linear(in_features=512, out_features=1, bias=True)
    (1): Sigmoid()
  )
)
```

To train the model, we shall finetune the the top layers, namely `layer4` and `fc` layers. Freeze all other layers.

In [14]:
```
freeze_layers = ["layer4", "fc"]

#################################################################
# Freeze all the layers except for the layers defined in
# freeze_layers  (5 lines)
#################################################################
for name, param in model.named_parameters():
    if np.any([name.startswith(layer) for layer in freeze_layers]):
        param.requires_grad = True
    else:
        param.requires_grad = False
#################################################################
#                END OF CODE
#################################################################
```

Use the following code to check if you have set your model correctly. If you have configured the layers correctly, all the weights and biases for `layer4` and `fc` would be `True` whereas those for the remaining layers would be `False` .

In [16]:
```
for name, param in model.named_parameters():
    print(name, ":", param.requires_grad)
```

```
conv1.weight : False
bn1.weight : False
bn1.bias : False
layer1.0.conv1.weight : False
layer1.0.bn1.weight : False
layer1.0.bn1.bias : False
layer1.0.conv2.weight : False
layer1.0.bn2.weight : False
layer1.0.bn2.bias : False
layer1.1.conv1.weight : False
layer1.1.bn1.weight : False
layer1.1.bn1.bias : False
layer1.1.conv2.weight : False
layer1.1.bn2.weight : False
layer1.1.bn2.bias : False
layer2.0.conv1.weight : False
layer2.0.bn1.weight : False
layer2.0.bn1.bias : False
layer2.0.conv2.weight : False
layer2.0.bn2.weight : False
layer2.0.bn2.bias : False
layer2.0.downsample.0.weight : False
layer2.0.downsample.1.weight : False
layer2.0.downsample.1.bias : False
layer2.1.conv1.weight : False
layer2.1.bn1.weight : False
layer2.1.bn1.bias : False
layer2.1.conv2.weight : False
layer2.1.bn2.weight : False
layer2.1.bn2.bias : False
layer3.0.conv1.weight : False
layer3.0.bn1.weight : False
layer3.0.bn1.bias : False
layer3.0.conv2.weight : False
layer3.0.bn2.weight : False
layer3.0.bn2.bias : False
layer3.0.downsample.0.weight : False
layer3.0.downsample.1.weight : False
layer3.0.downsample.1.bias : False
layer3.1.conv1.weight : False
layer3.1.bn1.weight : False
layer3.1.bn1.bias : False
layer3.1.conv2.weight : False
layer3.1.bn2.weight : False
layer3.1.bn2.bias : False
layer4.0.conv1.weight : True
layer4.0.bn1.weight : True
layer4.0.bn1.bias : True
layer4.0.conv2.weight : True
layer4.0.bn2.weight : True
layer4.0.bn2.bias : True
layer4.0.downsample.0.weight : True
layer4.0.downsample.1.weight : True
layer4.0.downsample.1.bias : True
layer4.1.conv1.weight : True
layer4.1.bn1.weight : True
layer4.1.bn1.bias : True
```

```
layer4.1.conv2.weight : True
layer4.1.bn2.weight : True
layer4.1.bn2.bias : True
fc.0.weight : True
fc.0.bias : True
```

Training the Model

Now we are ready to train the model. In the following, we define the transformation, set up our optimizer and scheduler, and then define the training function before training the model.

Define the transformation function to augment the dataset.

In [ ]:
```
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# transform the model
train_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Load the dataset with the defined transformation

In [ ]:
```
trainset = HymenopteraDataset("./hymenoptera_data/train", transform=train_tran
sform)
trainloader = DataLoader(trainset, batch_size=4, shuffle=True, num_workers=0)
```

Set up the optimizer

In [ ]:
```
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Set up the scheduler. In the following, we are going to use the **step decay schedule**. We shall drop the learning rate by a factor of 0.1 every 7 epochs.

In [ ]:
```
from torch.optim import lr_scheduler

scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

Train the model. We pass both the dataloader, optimizer and scheduler into the function. In order to reduce the learning rate according to the schedule, you must `scheduler.step` at the end of every epoch

In [ ]:
```
def train(net, trainloader, optimizer, scheduler, num_epochs):

    history = []

    # transfer model to GPU
    if torch.cuda.is_available():
        net = net.cuda()

    # set to training mode
    net.train()

    # train the network
    for e in range(num_epochs):

        running_loss = 0.0
        running_count = 0.0

        for i, (inputs, labels) in enumerate(trainloader):

            labels = labels.reshape(-1, 1).float()

            # Clear all the gradient to 0
            optimizer.zero_grad()

            # transfer data to GPU
            if torch.cuda.is_available():
                inputs = inputs.cuda()
                labels = labels.cuda()

            # forward propagation to get h
            outs = net(inputs)

            # compute loss
            loss = F.binary_cross_entropy(outs, labels)

            # backpropagation to get dw
            loss.backward()

            # update the parameters
            optimizer.step()

            # get the loss
            running_loss += loss.item()
            running_count += 1

        # compute the averaged loss in each epoch
        train_loss = running_loss / running_count
        running_loss = 0.
        running_count = 0.
        print(f'[Epoch {e+1:2d}/{num_epochs:d} Iter {i+1:5d}/{len(trainloade
r)}]: train_loss = {train_loss:.4f}')

        # Update the scheduler's counter at the end of each epoch
        scheduler.step()

    return
```

Now we are ready to train our model. We should expect training loss of about 0.2.

In [ ]:
```
train(model, trainloader, optimizer, scheduler, num_epochs=25)
```

Evaluate the model

The following code then evaluates the model. The expected accuracy is around 93.4%.

In [ ]:
```
def evaluate(model, testloader):
    # set to evaluation mode
    model.eval()

    # running_correct
    running_corrects = 0
    running_count = 0

    for inputs, targets in testloader:

        # transfer to the GPU
        if torch.cuda.is_available():
            inputs = inputs.cuda()
            targets = targets.cuda()

        # perform prediction (no need to compute gradient)
        with torch.no_grad():
            outputs = model(inputs)
            predicted = outputs > 0.5
            running_corrects += (predicted.view(-1) == targets).sum().double()
            running_count += len(inputs)
            print('.', end='')

    print('\nAccuracy = {:.2f}%'.format(100*running_corrects/running_count))
```

In [ ]:
```
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# transform the model
val_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

testset = HymenopteraDataset("./hymenoptera_data/val", transform=val_transform
)
testloader = DataLoader(testset, batch_size=4, shuffle=True, num_workers=0)

evaluate(model, testloader)
```