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 [2]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

In [3]:
```
cd "/content/gdrive/My Drive/UCCD3074_Labs/UCCD3074_Lab7"
```

```
[Errno 2] No such file or directory: '/content/gdrive/My Drive/UCCD3074_Labs/UCCD3074_Lab7'
/content
```

In [4]:
```
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 torch.optim import lr_scheduler

from PIL import Image
```

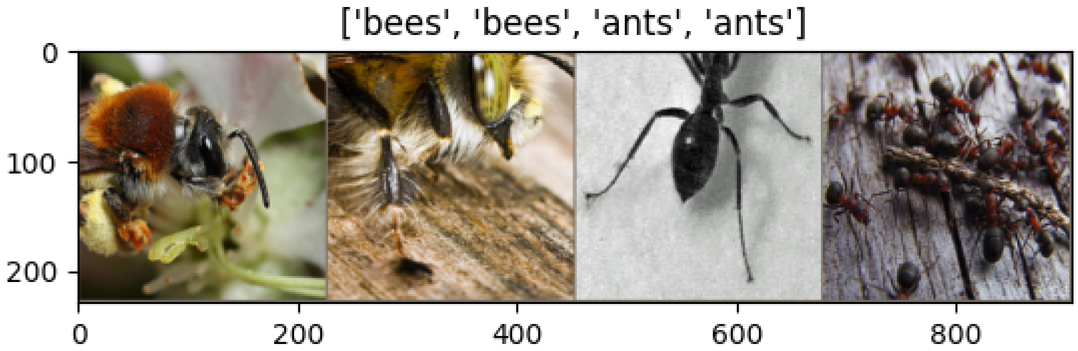
Helper functions

In [5]:
```python
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))
```

---

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 [6]:
```
!wget https://download.pytorch.org/tutorial/hymenoptera_data.zip
!unzip -q hymenoptera_data.zip
!rm 'hymenoptera_data/train/ants/imageNotFound.gif'
```

```
--2022-07-31 22:17:45--  https://download.pytorch.org/tutorial/hymenoptera_data.zip (https://download.pytorch.org/tutorial/hymenoptera_data.zip)
Resolving download.pytorch.org (download.pytorch.org)... 13.224.250.7, 13.224.250.113, 13.224.250.24, ...
Connecting to download.pytorch.org (download.pytorch.org)|13.224.250.7|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 47286322 (45M) [application/zip]
Saving to: 'hymenoptera_data.zip'

hymenoptera_data.zi 100%[===================>]  45.10M  28.1MB/s    in 1.6s

2022-07-31 22:17:48 (28.1 MB/s) - 'hymenoptera_data.zip' saved [47286322/47286322]
```

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

2. Implementing a custom dataset

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 [7]:

```
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.

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.

Instantiating `HymenopteraDataset`

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

In [8]:
```python
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 dataset. Since we did not transform the image, we can still display the image without undoing the transformation.

In [9]:
```
image, label = trainset[1]
display(image)
print("Class =", trainset.classes[label])
```

```
Class = ants
```

Transformation and Data Loader

In [10]:
```
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))
])
```

In [11]:
```
trainset = HymenopteraDataset("./hymenoptera_data/train", transform=train_transform)
trainloader = DataLoader(trainset, batch_size=4, shuffle=True, num_workers=0)
```

---

3 Customizing EfficientNet for Binary Classification

Now, customize EfficientNet B0 (`torchvision.models.efficientnet_b0`) to build a classifier to differentiate between *ants* vs *bees*. We shall build our model using pre-trained model from ImageNet to build our model.

In [26]:
```
net = models.efficientnet_b0(weights='IMAGENET1K_V1')
in_c = net.classifier[1].in_features
net.classifier[1] = nn.Sequential(
    nn.Linear(in_c, 1),
    nn.Sigmoid()
)
```

---

4. Train the Model

Set up the optimizer with momentum. Set the learning rate `lr` to 0.01 and `momentum` to 0.9.

In [27]:
```
optimizer = optim.SGD(net.parameters(), lr=0.01, 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 10 epochs.

In [28]:
```
scheduler = lr_scheduler.StepLR(optimizer, step_size=10, 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

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

In [29]:
```python
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} : train_loss = {train_loss:.4f}')

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

    return
```

In [30]:
```
train (net, trainloader, optimizer, scheduler, num_epochs=50)
```

```
Epoch  1/50 : train_loss = 0.4818
Epoch  2/50 : train_loss = 0.4880
Epoch  3/50 : train_loss = 0.5629
Epoch  4/50 : train_loss = 0.5404
Epoch  5/50 : train_loss = 0.6086
Epoch  6/50 : train_loss = 0.4520
Epoch  7/50 : train_loss = 0.3490
Epoch  8/50 : train_loss = 0.3814
Epoch  9/50 : train_loss = 0.3917
Epoch 10/50 : train_loss = 0.2900
Epoch 11/50 : train_loss = 0.3547
Epoch 12/50 : train_loss = 0.2646
Epoch 13/50 : train_loss = 0.1799
Epoch 14/50 : train_loss = 0.1785
Epoch 15/50 : train_loss = 0.1947
Epoch 16/50 : train_loss = 0.1506
Epoch 17/50 : train_loss = 0.1890
Epoch 18/50 : train_loss = 0.1458
Epoch 19/50 : train_loss = 0.1480
Epoch 20/50 : train_loss = 0.1409
Epoch 21/50 : train_loss = 0.1138
Epoch 22/50 : train_loss = 0.1366
Epoch 23/50 : train_loss = 0.1621
Epoch 24/50 : train_loss = 0.1503
Epoch 25/50 : train_loss = 0.1524
Epoch 26/50 : train_loss = 0.1204
Epoch 27/50 : train_loss = 0.1151
Epoch 28/50 : train_loss = 0.1772
Epoch 29/50 : train_loss = 0.1402
Epoch 30/50 : train_loss = 0.1096
Epoch 31/50 : train_loss = 0.1319
Epoch 32/50 : train_loss = 0.1640
Epoch 33/50 : train_loss = 0.1727
Epoch 34/50 : train_loss = 0.1418
Epoch 35/50 : train_loss = 0.0933
Epoch 36/50 : train_loss = 0.1974
Epoch 37/50 : train_loss = 0.1622
Epoch 38/50 : train_loss = 0.1467
Epoch 39/50 : train_loss = 0.1397
Epoch 40/50 : train_loss = 0.1377
Epoch 41/50 : train_loss = 0.1543
```

```
Epoch 42/50 : train_loss = 0.1336
Epoch 43/50 : train_loss = 0.0962
Epoch 44/50 : train_loss = 0.1142
Epoch 45/50 : train_loss = 0.1637
Epoch 46/50 : train_loss = 0.1344
Epoch 47/50 : train_loss = 0.1454
Epoch 48/50 : train_loss = 0.1587
Epoch 49/50 : train_loss = 0.1259
Epoch 50/50 : train_loss = 0.1249
```

Evaluate the model

The following code then evaluates the model. The expected accuracy is around 86%.

In [33]:
```python
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)
```

In [34]:
```python
evaluate(net, testloader)
```

```
......................................
Accuracy = 86.93%
```

--- End of Lab ---