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

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_remount=Tru
```

```
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
```

```
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
```

Saved successfully!

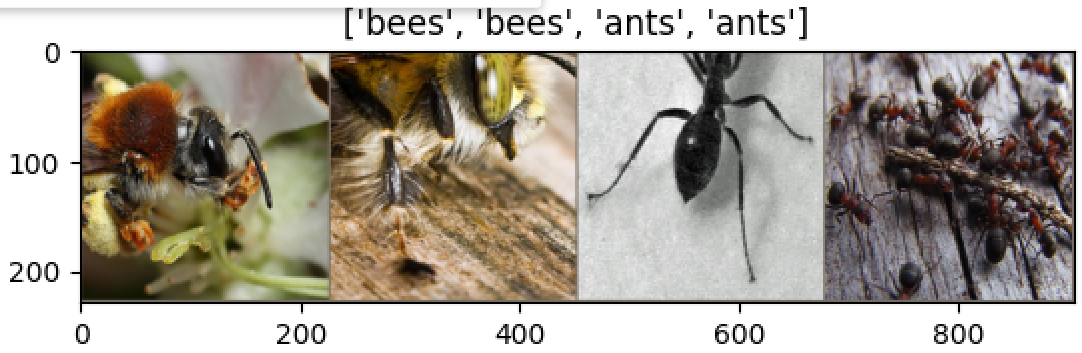
Helper functions

```
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.




```
1 !wget https://download.pytorch.org/tutorial/hymenoptera_data.zip
2 !unzip -q hymenoptera_data.zip
3 !rm 'hymenoptera_data/train/ants/imageNotFound.gif'
```

```
--2022-03-11 14:33:45--  https://download.pytorch.org/tutorial/hymenoptera_data.zip
Resolving download.pytorch.org (download.pytorch.org)... 108.156.120.107, 108.156.120.33, 108.156.120.103, ...
Connecting to download.pytorch.org (download.pytorch.org)|108.156.120.107|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 47286322 (45M) [application/zip]
Saving to: ‘hymenoptera_data.zip.2’

hymenoptera_data.zi 100%[===================>]  45.10M  57.7MB/s    in 0.8s

2022-03-11 14:33:46 (57.7 MB/s) - ‘hymenoptera_data.zip.2’ saved [47286322/47286322]

replace hymenoptera_data/train/ants/0013035.jpg? [y]es, [n]o, [A]ll, [N]one, [r]ename: N
```

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

```
hymenoptera_data\
   train\
      ants\
```

```
    val\
        ants\
        bees\
```

---

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.

```
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)
```

```
19     def __len__(self):
20         return len(self.data)
21 
22     def __getitem__(self, idx):
23 
24         # get the image
25         image = Image.open(self.data[idx])
26 
27         # perform transformation
28         if self.transform is not None:
29             image = self.transform(image)
30 
31         # get the label
32         label = self.labels[idx]
33 
34         # return sample
35         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.

```
1 trainset = HymenopteraDataset('./hymenoptera_data/train', transform=None)
2
3 print('Number of samples in dataset:', len(trainset))
4 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.

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

Transformation and Data Loader

```
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))
])
```

```
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 ResNet18 (`torchvision.models`) 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.

Saved successfully!

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

```
nn.Sequential(
    nn.Linear(512, 1)
    nn.Sigmoid()
)
```

```
def customize_network(pretrained=True):

    # replace the classification layer
    efficientNet = models.efficientnet_b0(pretrained=pretrained)
    in_c = efficientNet.classifier[1].in_features
    efficientNet.classifier[1] = nn.Sequential(
        nn.Linear(in_c, 1),
        nn.Sigmoid()
    )

    # freeze the top layers
    freeze_layers = ["features.6", "features.7", "features.8", "fc"]

    for name, param in efficientNet.named_parameters():
        if np.any([name.startswith(layer) for layer in freeze_layers]):
            param.requires_grad = True
        else:
            param.requires_grad = False

    return efficientNet
```

```
efficientNet = customize_network()
```

Saved successfully!

4. Train the Model

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

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

```
optimizer = optim.SGD(efficientNet.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 7 epochs.

```
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.

```
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()

```

```

        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.
```

Saved successfully!

```
um_epochs:d} : train_loss = {train_loss:.4f}')
```

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

    return
```

```
train (efficientNet, trainloader, optimizer, scheduler, num_epochs=30)
```

```
Epoch  1/30 : train_loss = 0.5958
Epoch  2/30 : train_loss = 0.3971
Epoch  3/30 : train_loss = 0.3288
Epoch  4/30 : train_loss = 0.2247
Epoch  5/30 : train_loss = 0.1929
Epoch  6/30 : train_loss = 0.2822
Epoch  7/30 : train_loss = 0.3054
Epoch  8/30 : train_loss = 0.2947
Epoch  9/30 : train_loss = 0.1716
Epoch 10/30 : train_loss = 0.1886
Epoch 11/30 : train_loss = 0.2310
Epoch 12/30 : train_loss = 0.2307
Epoch 13/30 : train_loss = 0.1559
Epoch 14/30 : train_loss = 0.2176
Epoch 15/30 : train_loss = 0.1665
Epoch 16/30 : train_loss = 0.1809
Epoch 17/30 : train_loss = 0.2038
Epoch 18/30 : train_loss = 0.1329
Epoch 19/30 : train_loss = 0.1762
Epoch 20/30 : train_loss = 0.1459
Epoch 21/30 : train_loss = 0.1408
Epoch 22/30 : train_loss = 0.1359
Epoch 23/30 : train_loss = 0.1670
Epoch 24/30 : train_loss = 0.1537
Epoch 25/30 : train_loss = 0.1546
Epoch 26/30 : train_loss = 0.1975
Epoch 27/30 : train_loss = 0.2395
Epoch 28/30 : train_loss = 0.1718
Epoch 29/30 : train_loss = 0.1849
Epoch 30/30 : train_loss = 0.1372
```

Evaluate the model

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

```
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(efficientNet, testloader)
```

```
.......................................
Accuracy = 90.85%
```

--- End of Lab ---

Saved successfully!

2s completed at 10:38 PM

Saved successfully!