Lab 6A: CNN Architectures and Transfer Learning (Guide)

The learning objectives for this lab exercise are as follows:

1. Customize the standard CNN Network to a targeted task
2. Perform different kinds of transfer learning:
    A. Train from scratch
    B. Finetune the whole model
    C. Finetune the upper layers of the model
    D. As a feature extractor

In practice, it is common to use a **standard CNN architectures** such that ResNet, MNASNet, ResNeXt, EfficientNet, etc. to build a model. The effectiveness of these network architectures has been well attested for a wide range of applications.

Rather than training from scratch, it is advisable to use **transfer learning** by training on top of a standard model that has been **pretrained** on the ImageNet dataset. Transfer learning reduces overfitting and improves the generalization performance of the trained model, especially when the training set for the targeted task is small. The `torchvision.models` package contains these different network models that have been pre-trained on ImageNet.

We perform transfer learning in two ways:

1. *Finetuning the convnet*: Instead of random initialization, initialize the network with the pretrained network.

2. *Fixed feature extractor*: Freeze the weights for all of the layers of the network except for the final fully connected (fc) layer. Replace the last fc layer so that the output size is the same as the number of classes for the new task. The new layer is initialized with random weights and only this layer is trained.

Mount google drive onto virtual machine

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

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

Change current directory to Lab 6

In [ ]:
```
cd "/content/gdrive/My Drive/UCCD3074_Labs/UCCD3074_Lab6"
```

```
/content/gdrive/My Drive/UCCD3074_Labs/UCCD3074_Lab6
```

Load required libraries

In [1]:
```
import numpy as np
import torchvision.models as models

import torch, torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision.transforms as transforms

from cifar10 import CIFAR10
```

Helper Functions

Define the train function

In [2]:
```
loss_iter = 1

def train(net, num_epochs, lr=0.1, momentum=0.9, verbose=True):

    history = []

    loss_iterations = int(np.ceil(len(trainloader)/loss_iter))

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

    # set the optimizer
    optimizer = optim.SGD(net.parameters(), lr=lr, momentum=momentum)

    # 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):

            # 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.cross_entropy(outs, labels)

            # backpropagation to get dw
            loss.backward()

            # update w
            optimizer.step()

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

             # display the averaged loss value
            if i % loss_iterations == loss_iterations-1 or i == len(trainloader) - 1:
                train_loss = running_loss / running_count
                running_loss = 0.
                running_count = 0.
                if verbose:
                    print(f'[Epoch {e+1:2d}/{num_epochs:d} Iter {i+1:5d}/{len(trainloader)}]: train_loss = {train_loss:.4f}')

                history.append(train_loss)

    return history
```

Define the evaluate function

In [3]:
```
def evaluate(net):
    # set to evaluation mode
    net.eval()

    # running_correct
    running_corrects = 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 = net(inputs)
            _, predicted = torch.max(outputs, 1)
            running_corrects += (targets == predicted).double().sum()

    print('Accuracy = {:.2f}%'.format(100*running_corrects/len(testloader.dataset)))
```

1. Load CIFAR10 dataset

Here, we use a sub-sample of CIFAR10 where we use a sub-sample of 1000 training and testing samples. The sample size is small and hence is expected to face overfitting issue. Using a pretrained model alleviates the problem.

In [4]:
```
# transform the model
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))
])

# dataset
trainset = CIFAR10(train=True, download=True, transform=transform, num_samples=1000)
testset  = CIFAR10(train=False, download=True, transform=transform, num_samples=1000)

# dataloader]
trainloader = DataLoader(trainset, batch_size=32, shuffle=True, num_workers=2)
testloader  = DataLoader(testset, batch_size=128, shuffle=True, num_workers=2)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

2. The ResNet50 model

In this section, we shall build our network using a standard network architectures. We customize a pre-trained ResNet50 by replacing its classifier layer, i.e., the last fully connected layer with our own. The original ImageNet classifier is designed to classify 1000 output classes whereas our CIFAR10 classifier handles only 10 classes.

Using the pre-trained models

First, let's learn how to load and use a pre-trained model as it is. The following table lists the pretrained models for ResNet50 together their reported accuries on ImageNet-1K with single crops.

weight	Acc@1	Acc@5	Params
ResNet50_Weights.IMAGENET1K_V1	76.13	92.862	25.6MB
ResNet50_Weights.IMAGENET1K_V2	80.858	95.434	25.6MB

where IMAGENET1K_V2 improves upon IMAGENET1K_V1 by using a new training recipe

To specify the pretrained model, you can use the predefined constant:

In [5]:
```
from torchvision.models import resnet50, ResNet50_Weights
```

In [6]:
```
net = resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)
```

```
Downloading: "https://download.pytorch.org/models/resnet50-11ad3fa6.pth" to C:\Users\hungk/.cache\torch\hub\checkpoints\resnet50-11ad3fa6.pth
  0%|          | 0.00/97.8M [00:00<?, ?B/s]
```

or a string argument.

In [7]:
```
net = resnet50(weights="IMAGENET1K_V2")
```

Inferencing with the pretrained model

Some pretrained model needs specific preprocessing steps (e.g., resize into a specific resolution / rescale the values, etc.). The preprocessing steps vary depending on how the model was trained. The necessary information for inference transforms are provided on the weight documentation. But to simplify inference, `TorchVision` also bundle a `transform` utility into `ResNet.Weights`.

In [8]:
```
weight = ResNet50_Weights.IMAGENET1K_V2
preprocess = weight.transforms()
```

In [9]:
```
from torchvision.io import read_image
img = read_image('img1.jpg')
print('Shape of x (before preprocessing)', img.shape)

x  = preprocess(img)
print('Shape of x after preprocessing:', x.shape)

x  = x.unsqueeze(0)
print('Shape of x after unsqueezing:', x.shape)
```

```
Shape of x (before preprocessing) torch.Size([3, 162, 288])
Shape of x after preprocessing: torch.Size([3, 224, 224])
Shape of x after unsqueezing: torch.Size([1, 3, 224, 224])
```

Perform inference with the pretrained model. The classes of the pretrained model can be found at `weights.meta['categories']`.

In [10]:
```
net.eval()
with torch.no_grad():
    score = net(x)
    predicted = score.argmax(axis=1)[0]

print('Predicted label =', weight.meta['categories'][predicted])
```

```
Predicted label = tabby
```

Customizing ResNet50

In the following, we shall replace the last layer with a new classifier layer. The pre-trained model is designed to classify ImageNet's 1000 image categories. In the following, we shall customize it to classify Cifar10's 10 classes. First, let's look at how ResNet50 is implemented in PyTorch.

In [11]:
```
print(net)
```

```
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): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (relu): ReLU(inplace=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): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
```

```
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=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)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

In [ ]:

```
for name, _ in net.named_children():
    print(name)
```

```
conv1
bn1
relu
maxpool
layer1
layer2
layer3
layer4
avgpool
fc
```

Here are some observations:

- `conv1`, `bn1`, `relu` and `maxpool` are the *stem* network
- There are 4 *blocks* in the network, namely `layer1`, `layer2`, `layer3` and `layer4`.
- Each of the block contains two convolutional layers.
- The second last layer (`avgpool`) performs *global average pooling* to average out the spatial dimensions.
- The last layer (`fc`) is a linear layer that functions as a classifier. **This is the layer that we want to replace to fit our model**.

To customize the network, we need to replace the `fc` layer with our own classifier layer.

In [ ]:
```
def build_network(weights=None):
    net = resnet50(weights=weights)
    in_c = net.fc.in_features
    net.fc = nn.Linear(in_c, 10)
    return net
```

Let's visualize what we have built. Note that the last layer of the network (`fc`) now has 10 instead of 1000 neurons.

In [ ]:
```
print(build_network())
```

Model 1: Training from scratch

Let's build the network **without** loading the pretrained model. To do this, we set `weights=None`.

In [ ]:
```
net1 = build_network(weights=None)
```

Train the model and save the training loss history into `history1`.

In [ ]:
```
history1 = train(net1, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 4.4605
[Epoch  2/30 Iter    32/32]: train_loss = 2.8071
[Epoch  3/30 Iter    32/32]: train_loss = 2.6078
[Epoch  4/30 Iter    32/32]: train_loss = 2.2501
[Epoch  5/30 Iter    32/32]: train_loss = 2.1665
[Epoch  6/30 Iter    32/32]: train_loss = 2.0636
[Epoch  7/30 Iter    32/32]: train_loss = 2.0430
[Epoch  8/30 Iter    32/32]: train_loss = 1.9960
[Epoch  9/30 Iter    32/32]: train_loss = 1.9771
[Epoch 10/30 Iter    32/32]: train_loss = 1.8862
[Epoch 11/30 Iter    32/32]: train_loss = 1.9134
[Epoch 12/30 Iter    32/32]: train_loss = 1.8538
[Epoch 13/30 Iter    32/32]: train_loss = 1.8350
[Epoch 14/30 Iter    32/32]: train_loss = 1.8004
[Epoch 15/30 Iter    32/32]: train_loss = 1.7669
[Epoch 16/30 Iter    32/32]: train_loss = 1.6987
[Epoch 17/30 Iter    32/32]: train_loss = 1.7374
[Epoch 18/30 Iter    32/32]: train_loss = 1.6700
[Epoch 19/30 Iter    32/32]: train_loss = 1.7457
[Epoch 20/30 Iter    32/32]: train_loss = 1.6286
[Epoch 21/30 Iter    32/32]: train_loss = 1.6843
[Epoch 22/30 Iter    32/32]: train_loss = 1.5599
[Epoch 23/30 Iter    32/32]: train_loss = 1.5442
[Epoch 24/30 Iter    32/32]: train_loss = 1.5833
[Epoch 25/30 Iter    32/32]: train_loss = 1.5343
[Epoch 26/30 Iter    32/32]: train_loss = 1.5184
[Epoch 27/30 Iter    32/32]: train_loss = 1.5077
[Epoch 28/30 Iter    32/32]: train_loss = 1.4209
[Epoch 29/30 Iter    32/32]: train_loss = 1.4528
[Epoch 30/30 Iter    32/32]: train_loss = 1.4286
```

Evaluate the model

In [ ]:
```
evaluate(net1)
```

```
Accuracy = 40.50%
```

Model 2: Finetuning the pretrained model

Typically, a standard network come with a pretrained model trained on ImageNet's large-scale dataset for the image classification task.

- In the following, we shall load resnet50 with the pretrained model and use it to **initialize** the network. To do this, we set `weights='IMAGENET1K_V2'` .
- The training will update the parameters **all layers** of the network.

For Windows system, the pretrained model will be saved to the following directory: `C:\Users\<user name>\.cache\torch\checkpoints` . A PyTorch model has an extension of `.pt` or `.pth` .

In [ ]:

```
net2 = build_network(weights='IMAGENET1K_V2')
```

By default, all the layers are set to `requires_grad=True`

In [ ]:

```
for name, param in net2.named_parameters():
    print(name, ':', param.requires_grad)
```

```
conv1.weight : True
bn1.weight : True
bn1.bias : True
layer1.0.conv1.weight : True
layer1.0.bn1.weight : True
layer1.0.bn1.bias : True
layer1.0.conv2.weight : True
layer1.0.bn2.weight : True
layer1.0.bn2.bias : True
layer1.0.conv3.weight : True
layer1.0.bn3.weight : True
layer1.0.bn3.bias : True
layer1.0.downsample.0.weight : True
layer1.0.downsample.1.weight : True
layer1.0.downsample.1.bias : True
layer1.1.conv1.weight : True
layer1.1.bn1.weight : True
layer1.1.bn1.bias : True
layer1.1.conv2.weight : True
layer1.1.bn2.weight : True
layer1.1.bn2.bias : True
layer1.1.conv3.weight : True
layer1.1.bn3.weight : True
layer1.1.bn3.bias : True
layer1.2.conv1.weight : True
layer1.2.bn1.weight : True
layer1.2.bn1.bias : True
layer1.2.conv2.weight : True
layer1.2.bn2.weight : True
layer1.2.bn2.bias : True
layer1.2.conv3.weight : True
layer1.2.bn3.weight : True
layer1.2.bn3.bias : True
layer2.0.conv1.weight : True
layer2.0.bn1.weight : True
layer2.0.bn1.bias : True
layer2.0.conv2.weight : True
layer2.0.bn2.weight : True
layer2.0.bn2.bias : True
layer2.0.conv3.weight : True
layer2.0.bn3.weight : True
layer2.0.bn3.bias : True
layer2.0.downsample.0.weight : True
layer2.0.downsample.1.weight : True
layer2.0.downsample.1.bias : True
layer2.1.conv1.weight : True
```

```
layer2.1.bn1.weight : True
layer2.1.bn1.bias : True
layer2.1.conv2.weight : True
layer2.1.bn2.weight : True
layer2.1.bn2.bias : True
layer2.1.conv3.weight : True
layer2.1.bn3.weight : True
layer2.1.bn3.bias : True
layer2.2.conv1.weight : True
layer2.2.bn1.weight : True
layer2.2.bn1.bias : True
layer2.2.conv2.weight : True
layer2.2.bn2.weight : True
layer2.2.bn2.bias : True
layer2.2.conv3.weight : True
layer2.2.bn3.weight : True
layer2.2.bn3.bias : True
layer2.3.conv1.weight : True
layer2.3.bn1.weight : True
layer2.3.bn1.bias : True
layer2.3.conv2.weight : True
layer2.3.bn2.weight : True
layer2.3.bn2.bias : True
layer2.3.conv3.weight : True
layer2.3.bn3.weight : True
layer2.3.bn3.bias : True
layer3.0.conv1.weight : True
layer3.0.bn1.weight : True
layer3.0.bn1.bias : True
layer3.0.conv2.weight : True
layer3.0.bn2.weight : True
layer3.0.bn2.bias : True
layer3.0.conv3.weight : True
layer3.0.bn3.weight : True
layer3.0.bn3.bias : True
layer3.0.downsample.0.weight : True
layer3.0.downsample.1.weight : True
layer3.0.downsample.1.bias : True
layer3.1.conv1.weight : True
layer3.1.bn1.weight : True
layer3.1.bn1.bias : True
layer3.1.conv2.weight : True
layer3.1.bn2.weight : True
layer3.1.bn2.bias : True
layer3.1.conv3.weight : True
layer3.1.bn3.weight : True
```

```
layer3.1.bn3.bias : True
layer3.2.conv1.weight : True
layer3.2.bn1.weight : True
layer3.2.bn1.bias : True
layer3.2.conv2.weight : True
layer3.2.bn2.weight : True
layer3.2.bn2.bias : True
layer3.2.conv3.weight : True
layer3.2.bn3.weight : True
layer3.2.bn3.bias : True
layer3.3.conv1.weight : True
layer3.3.bn1.weight : True
layer3.3.bn1.bias : True
layer3.3.conv2.weight : True
layer3.3.bn2.weight : True
layer3.3.bn2.bias : True
layer3.3.conv3.weight : True
layer3.3.bn3.weight : True
layer3.3.bn3.bias : True
layer3.4.conv1.weight : True
layer3.4.bn1.weight : True
layer3.4.bn1.bias : True
layer3.4.conv2.weight : True
layer3.4.bn2.weight : True
layer3.4.bn2.bias : True
layer3.4.conv3.weight : True
layer3.4.bn3.weight : True
layer3.4.bn3.bias : True
layer3.5.conv1.weight : True
layer3.5.bn1.weight : True
layer3.5.bn1.bias : True
layer3.5.conv2.weight : True
layer3.5.bn2.weight : True
layer3.5.bn2.bias : True
layer3.5.conv3.weight : True
layer3.5.bn3.weight : True
layer3.5.bn3.bias : True
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.conv3.weight : True
layer4.0.bn3.weight : True
layer4.0.bn3.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
layer4.1.conv3.weight : True
layer4.1.bn3.weight : True
layer4.1.bn3.bias : True
layer4.2.conv1.weight : True
layer4.2.bn1.weight : True
layer4.2.bn1.bias : True
layer4.2.conv2.weight : True
layer4.2.bn2.weight : True
layer4.2.bn2.bias : True
layer4.2.conv3.weight : True
layer4.2.bn3.weight : True
layer4.2.bn3.bias : True
fc.weight : True
fc.bias : True
```

Train the model and save into `history2`.

In [ ]:

```
history2 = train(net2, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 1.9645
[Epoch  2/30 Iter    32/32]: train_loss = 1.0289
[Epoch  3/30 Iter    32/32]: train_loss = 0.5807
[Epoch  4/30 Iter    32/32]: train_loss = 0.3406
[Epoch  5/30 Iter    32/32]: train_loss = 0.2320
[Epoch  6/30 Iter    32/32]: train_loss = 0.1732
[Epoch  7/30 Iter    32/32]: train_loss = 0.1001
[Epoch  8/30 Iter    32/32]: train_loss = 0.0991
[Epoch  9/30 Iter    32/32]: train_loss = 0.0664
[Epoch 10/30 Iter    32/32]: train_loss = 0.0696
[Epoch 11/30 Iter    32/32]: train_loss = 0.0682
[Epoch 12/30 Iter    32/32]: train_loss = 0.0461
[Epoch 13/30 Iter    32/32]: train_loss = 0.0342
[Epoch 14/30 Iter    32/32]: train_loss = 0.0549
[Epoch 15/30 Iter    32/32]: train_loss = 0.0681
[Epoch 16/30 Iter    32/32]: train_loss = 0.0402
[Epoch 17/30 Iter    32/32]: train_loss = 0.0547
[Epoch 18/30 Iter    32/32]: train_loss = 0.0390
[Epoch 19/30 Iter    32/32]: train_loss = 0.0977
[Epoch 20/30 Iter    32/32]: train_loss = 0.0600
[Epoch 21/30 Iter    32/32]: train_loss = 0.0325
[Epoch 22/30 Iter    32/32]: train_loss = 0.0353
[Epoch 23/30 Iter    32/32]: train_loss = 0.0356
[Epoch 24/30 Iter    32/32]: train_loss = 0.0212
[Epoch 25/30 Iter    32/32]: train_loss = 0.0112
[Epoch 26/30 Iter    32/32]: train_loss = 0.0105
[Epoch 27/30 Iter    32/32]: train_loss = 0.0176
[Epoch 28/30 Iter    32/32]: train_loss = 0.0373
[Epoch 29/30 Iter    32/32]: train_loss = 0.0305
[Epoch 30/30 Iter    32/32]: train_loss = 0.0128
```

Evaluate the network

In [ ]:

```
evaluate(net2)
```

```
Accuracy = 85.10%
```

Model 3: As a fixed feature extractor

When the dataset is too small, fine-tuning the model may still incur overfitting. In this case, you may want to try to use the pretrained as a fixed feature extractor where we train only the classifier layer (i.e., **last layer**) that we have newly inserted into the network.

In [ ]:
```
# Load the pretrained model
net3 = build_network(weights='IMAGENET1K_V2')
```

We set `requires_grad=False` for all parameters except for the newly replaced layer `fc`, i.e., the last two parameters in `resnet.parameters()`.

In [ ]:
```
parameters = list(net3.parameters())
for param in parameters[:-2]:
    param.requires_grad = False
```

In [ ]:
```
for name, param in net3.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
layer1.0.conv3.weight : False
layer1.0.bn3.weight : False
layer1.0.bn3.bias : False
layer1.0.downsample.0.weight : False
layer1.0.downsample.1.weight : False
layer1.0.downsample.1.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
layer1.1.conv3.weight : False
layer1.1.bn3.weight : False
layer1.1.bn3.bias : False
layer1.2.conv1.weight : False
layer1.2.bn1.weight : False
layer1.2.bn1.bias : False
layer1.2.conv2.weight : False
layer1.2.bn2.weight : False
layer1.2.bn2.bias : False
layer1.2.conv3.weight : False
layer1.2.bn3.weight : False
layer1.2.bn3.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.conv3.weight : False
layer2.0.bn3.weight : False
layer2.0.bn3.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
layer2.1.conv3.weight : False
layer2.1.bn3.weight : False
layer2.1.bn3.bias : False
layer2.2.conv1.weight : False
layer2.2.bn1.weight : False
layer2.2.bn1.bias : False
layer2.2.conv2.weight : False
layer2.2.bn2.weight : False
layer2.2.bn2.bias : False
layer2.2.conv3.weight : False
layer2.2.bn3.weight : False
layer2.2.bn3.bias : False
layer2.3.conv1.weight : False
layer2.3.bn1.weight : False
layer2.3.bn1.bias : False
layer2.3.conv2.weight : False
layer2.3.bn2.weight : False
layer2.3.bn2.bias : False
layer2.3.conv3.weight : False
layer2.3.bn3.weight : False
layer2.3.bn3.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.conv3.weight : False
layer3.0.bn3.weight : False
layer3.0.bn3.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
layer3.1.conv3.weight : False
layer3.1.bn3.weight : False
```

```
layer3.1.bn3.bias : False
layer3.2.conv1.weight : False
layer3.2.bn1.weight : False
layer3.2.bn1.bias : False
layer3.2.conv2.weight : False
layer3.2.bn2.weight : False
layer3.2.bn2.bias : False
layer3.2.conv3.weight : False
layer3.2.bn3.weight : False
layer3.2.bn3.bias : False
layer3.3.conv1.weight : False
layer3.3.bn1.weight : False
layer3.3.bn1.bias : False
layer3.3.conv2.weight : False
layer3.3.bn2.weight : False
layer3.3.bn2.bias : False
layer3.3.conv3.weight : False
layer3.3.bn3.weight : False
layer3.3.bn3.bias : False
layer3.4.conv1.weight : False
layer3.4.bn1.weight : False
layer3.4.bn1.bias : False
layer3.4.conv2.weight : False
layer3.4.bn2.weight : False
layer3.4.bn2.bias : False
layer3.4.conv3.weight : False
layer3.4.bn3.weight : False
layer3.4.bn3.bias : False
layer3.5.conv1.weight : False
layer3.5.bn1.weight : False
layer3.5.bn1.bias : False
layer3.5.conv2.weight : False
layer3.5.bn2.weight : False
layer3.5.bn2.bias : False
layer3.5.conv3.weight : False
layer3.5.bn3.weight : False
layer3.5.bn3.bias : False
layer4.0.conv1.weight : False
layer4.0.bn1.weight : False
layer4.0.bn1.bias : False
layer4.0.conv2.weight : False
layer4.0.bn2.weight : False
layer4.0.bn2.bias : False
layer4.0.conv3.weight : False
layer4.0.bn3.weight : False
layer4.0.bn3.bias : False
```

```
layer4.0.downsample.0.weight : False
layer4.0.downsample.1.weight : False
layer4.0.downsample.1.bias : False
layer4.1.conv1.weight : False
layer4.1.bn1.weight : False
layer4.1.bn1.bias : False
layer4.1.conv2.weight : False
layer4.1.bn2.weight : False
layer4.1.bn2.bias : False
layer4.1.conv3.weight : False
layer4.1.bn3.weight : False
layer4.1.bn3.bias : False
layer4.2.conv1.weight : False
layer4.2.bn1.weight : False
layer4.2.bn1.bias : False
layer4.2.conv2.weight : False
layer4.2.bn2.weight : False
layer4.2.bn2.bias : False
layer4.2.conv3.weight : False
layer4.2.bn3.weight : False
layer4.2.bn3.bias : False
fc.weight : True
fc.bias : True
```

Train the model and save into `history3`.

In [ ]:

```
history3 = train(net3, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 2.0736
[Epoch  2/30 Iter    32/32]: train_loss = 1.6097
[Epoch  3/30 Iter    32/32]: train_loss = 1.3621
[Epoch  4/30 Iter    32/32]: train_loss = 1.2497
[Epoch  5/30 Iter    32/32]: train_loss = 1.1367
[Epoch  6/30 Iter    32/32]: train_loss = 1.0983
[Epoch  7/30 Iter    32/32]: train_loss = 1.0367
[Epoch  8/30 Iter    32/32]: train_loss = 0.9963
[Epoch  9/30 Iter    32/32]: train_loss = 0.9157
[Epoch 10/30 Iter    32/32]: train_loss = 0.9017
[Epoch 11/30 Iter    32/32]: train_loss = 0.9024
[Epoch 12/30 Iter    32/32]: train_loss = 0.8587
[Epoch 13/30 Iter    32/32]: train_loss = 0.8654
[Epoch 14/30 Iter    32/32]: train_loss = 0.8663
[Epoch 15/30 Iter    32/32]: train_loss = 0.7999
[Epoch 16/30 Iter    32/32]: train_loss = 0.7995
[Epoch 17/30 Iter    32/32]: train_loss = 0.7702
[Epoch 18/30 Iter    32/32]: train_loss = 0.7750
[Epoch 19/30 Iter    32/32]: train_loss = 0.7872
[Epoch 20/30 Iter    32/32]: train_loss = 0.7529
[Epoch 21/30 Iter    32/32]: train_loss = 0.7194
[Epoch 22/30 Iter    32/32]: train_loss = 0.7264
[Epoch 23/30 Iter    32/32]: train_loss = 0.7533
[Epoch 24/30 Iter    32/32]: train_loss = 0.7054
[Epoch 25/30 Iter    32/32]: train_loss = 0.7212
[Epoch 26/30 Iter    32/32]: train_loss = 0.7063
[Epoch 27/30 Iter    32/32]: train_loss = 0.6903
[Epoch 28/30 Iter    32/32]: train_loss = 0.6863
[Epoch 29/30 Iter    32/32]: train_loss = 0.6490
[Epoch 30/30 Iter    32/32]: train_loss = 0.6389
```

Evaluate the model

In [ ]:
```
evaluate(net3)
```

```
Accuracy = 68.90%
```

Model 4: Finetuning the top few layers

We can also tune the top few layers of the network. The following tunes all the layers in the block `layer 4` as well as the `fc` layer.

In [ ]:
```
# Load the pretrained model
net4 = build_network(weights='IMAGENET1K_V2')
```

Then, we freeze all tha layers except for `layer4` and `fc` layers

In [ ]:

```
for name, param in net4.named_parameters():
    if not any(name.startswith(ext) for ext in ['layer4', 'fc']):
        param.requires_grad = False
```

In [ ]:

```
for name, param in net4.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
layer1.0.conv3.weight : False
layer1.0.bn3.weight : False
layer1.0.bn3.bias : False
layer1.0.downsample.0.weight : False
layer1.0.downsample.1.weight : False
layer1.0.downsample.1.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
layer1.1.conv3.weight : False
layer1.1.bn3.weight : False
layer1.1.bn3.bias : False
layer1.2.conv1.weight : False
layer1.2.bn1.weight : False
layer1.2.bn1.bias : False
layer1.2.conv2.weight : False
layer1.2.bn2.weight : False
layer1.2.bn2.bias : False
layer1.2.conv3.weight : False
layer1.2.bn3.weight : False
layer1.2.bn3.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.conv3.weight : False
layer2.0.bn3.weight : False
layer2.0.bn3.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
layer2.1.conv3.weight : False
layer2.1.bn3.weight : False
layer2.1.bn3.bias : False
layer2.2.conv1.weight : False
layer2.2.bn1.weight : False
layer2.2.bn1.bias : False
layer2.2.conv2.weight : False
layer2.2.bn2.weight : False
layer2.2.bn2.bias : False
layer2.2.conv3.weight : False
layer2.2.bn3.weight : False
layer2.2.bn3.bias : False
layer2.3.conv1.weight : False
layer2.3.bn1.weight : False
layer2.3.bn1.bias : False
layer2.3.conv2.weight : False
layer2.3.bn2.weight : False
layer2.3.bn2.bias : False
layer2.3.conv3.weight : False
layer2.3.bn3.weight : False
layer2.3.bn3.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.conv3.weight : False
layer3.0.bn3.weight : False
layer3.0.bn3.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
layer3.1.conv3.weight : False
layer3.1.bn3.weight : False
```

```
layer3.1.bn3.bias : False
layer3.2.conv1.weight : False
layer3.2.bn1.weight : False
layer3.2.bn1.bias : False
layer3.2.conv2.weight : False
layer3.2.bn2.weight : False
layer3.2.bn2.bias : False
layer3.2.conv3.weight : False
layer3.2.bn3.weight : False
layer3.2.bn3.bias : False
layer3.3.conv1.weight : False
layer3.3.bn1.weight : False
layer3.3.bn1.bias : False
layer3.3.conv2.weight : False
layer3.3.bn2.weight : False
layer3.3.bn2.bias : False
layer3.3.conv3.weight : False
layer3.3.bn3.weight : False
layer3.3.bn3.bias : False
layer3.4.conv1.weight : False
layer3.4.bn1.weight : False
layer3.4.bn1.bias : False
layer3.4.conv2.weight : False
layer3.4.bn2.weight : False
layer3.4.bn2.bias : False
layer3.4.conv3.weight : False
layer3.4.bn3.weight : False
layer3.4.bn3.bias : False
layer3.5.conv1.weight : False
layer3.5.bn1.weight : False
layer3.5.bn1.bias : False
layer3.5.conv2.weight : False
layer3.5.bn2.weight : False
layer3.5.bn2.bias : False
layer3.5.conv3.weight : False
layer3.5.bn3.weight : False
layer3.5.bn3.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.conv3.weight : True
layer4.0.bn3.weight : True
layer4.0.bn3.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
layer4.1.conv3.weight : True
layer4.1.bn3.weight : True
layer4.1.bn3.bias : True
layer4.2.conv1.weight : True
layer4.2.bn1.weight : True
layer4.2.bn1.bias : True
layer4.2.conv2.weight : True
layer4.2.bn2.weight : True
layer4.2.bn2.bias : True
layer4.2.conv3.weight : True
layer4.2.bn3.weight : True
layer4.2.bn3.bias : True
fc.weight : True
fc.bias : True
```

Train the model and save into `history4`.

In [ ]:
```
history4 = train(net4, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 2.0133
[Epoch  2/30 Iter    32/32]: train_loss = 1.2797
[Epoch  3/30 Iter    32/32]: train_loss = 0.9507
[Epoch  4/30 Iter    32/32]: train_loss = 0.7146
[Epoch  5/30 Iter    32/32]: train_loss = 0.6157
[Epoch  6/30 Iter    32/32]: train_loss = 0.4763
[Epoch  7/30 Iter    32/32]: train_loss = 0.4160
[Epoch  8/30 Iter    32/32]: train_loss = 0.3392
[Epoch  9/30 Iter    32/32]: train_loss = 0.3121
[Epoch 10/30 Iter    32/32]: train_loss = 0.2908
[Epoch 11/30 Iter    32/32]: train_loss = 0.1961
[Epoch 12/30 Iter    32/32]: train_loss = 0.2080
[Epoch 13/30 Iter    32/32]: train_loss = 0.1907
[Epoch 14/30 Iter    32/32]: train_loss = 0.1722
[Epoch 15/30 Iter    32/32]: train_loss = 0.1200
[Epoch 16/30 Iter    32/32]: train_loss = 0.1057
[Epoch 17/30 Iter    32/32]: train_loss = 0.0885
[Epoch 18/30 Iter    32/32]: train_loss = 0.0946
[Epoch 19/30 Iter    32/32]: train_loss = 0.0897
[Epoch 20/30 Iter    32/32]: train_loss = 0.0970
[Epoch 21/30 Iter    32/32]: train_loss = 0.0947
[Epoch 22/30 Iter    32/32]: train_loss = 0.0927
[Epoch 23/30 Iter    32/32]: train_loss = 0.0659
[Epoch 24/30 Iter    32/32]: train_loss = 0.0653
[Epoch 25/30 Iter    32/32]: train_loss = 0.0803
[Epoch 26/30 Iter    32/32]: train_loss = 0.0768
[Epoch 27/30 Iter    32/32]: train_loss = 0.0575
[Epoch 28/30 Iter    32/32]: train_loss = 0.0389
[Epoch 29/30 Iter    32/32]: train_loss = 0.0831
[Epoch 30/30 Iter    32/32]: train_loss = 0.0705
```

Evaluate the model

In [ ]:

```
evaluate(net4)
```

```
Accuracy = 76.20%
```

Plotting training loss

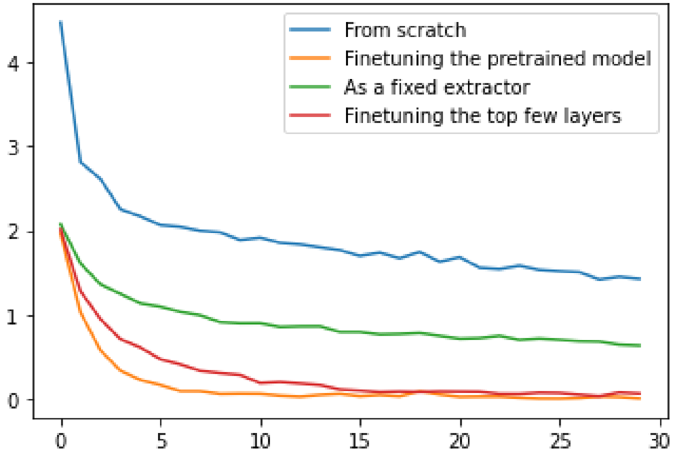
Lastly, we plot the training loss history for each of the training schemes above.

In [ ]:

```
import matplotlib.pyplot as plt

plt.plot(history1, label='From scratch')
```

```
plt.plot(history2, label='Finetuning the pretrained model')
plt.plot(history3, label='As a fixed extractor')
plt.plot(history4, label='Finetuning the top few layers')
plt.legend()
plt.show()
```



Exercise

You can try with different network architectures (e.g., EfficientNet-B0) and see if it results in higher test accuracy.

> *The list of all pre-trained models in PyTorch is listed in this Table.*

In [ ]:
```
def build_network(weights='IMAGENET1K_V1'):
    ...
    return net
```

In [ ]:
```
efficientNet = build_network()
```

In [ ]:
```
history5 = train(efficientNet, num_epochs=30, lr=0.01, momentum=0.8)
```

In [ ]:
```
evaluate(efficientNet)
```