Lab 6A: CNN Architectures and Transfer Learning

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:
    1. Train from scratch
    2. Finetune the whole model
    3. Finetune the upper layers of the model
    4. 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. The `torchvision.models` subpackage contains these different network models that have been pre-trained on ImageNet. 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. 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.

> Training a deep architecture using a pre-trained model allows us to train on a small dataset with less overfitting.

Mount google drive onto virtual machine

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

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

Change current directory to Lab 6

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

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

Load required libraries

```
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 torchsummary import summary

from cifar10 import CIFAR10
```

Helper Functions

Define the train function

```
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

```
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():
```

```
        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.

```
# 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)
```

2. The ResNet18 model

In this section, we shall build our network using a standard network architectures. We customize resnet18 by replacing its classifier layer, i.e., the last fully connected layer with our own. The original classifier layer has 1000 outputs (ImageNet has 1000 output classes) whereas our model has only 10.

Network Architecture of ResNet18

We shall use resnet18 as our base network. Before we customize it, let's print out the summary of all layers of the model to view its architecture. Bear in mind that to customize the network, we need to replace the last linear layer.

First, let's review the resnet18 network architecture.

```
resnet18 = models.resnet18()
```

```
print(resnet18)
```

We can get the name of the first layer by accessing the `.named_children`.

```
for name, _ in resnet18.named_children():
    print(name)
```

```
conv1
bn1
relu
maxpool
layer1
layer2
layer3
```

```
    layer4
    avgpool
    fc
```

We will see that:

- `layer1` to `layer4` contains two blocks each. Each block is 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 and indeed, it 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.

Customizing ResNet18

In the following, we shall replace the last layer with a new classifier layer. The original layer is designed to classify ImageNet's 1000 image categories. The new layers will be used to classify Cifar10's 10 classes

```
def build_network(pretrained=True):
    resnet18 = models.resnet18(pretrained=pretrained)
    in_c = resnet18.fc.in_features
    resnet18.fc = nn.Linear(in_c, 10)
    return resnet18
```

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

```
print(build_network())
```

Model 1: Training from scratch

Let's build the network **without** loading the pretrained model. To do this, we set `pretrained=False` .

```
resnet18 = build_network(pretrained=False)
```

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

```
1 history1 = train(resnet18, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 2.1836
[Epoch  2/30 Iter    32/32]: train_loss = 1.9838
[Epoch  3/30 Iter    32/32]: train_loss = 1.9456
[Epoch  4/30 Iter    32/32]: train_loss = 1.7846
[Epoch  5/30 Iter    32/32]: train_loss = 1.7053
[Epoch  6/30 Iter    32/32]: train_loss = 1.6381
[Epoch  7/30 Iter    32/32]: train_loss = 1.5608
[Epoch  8/30 Iter    32/32]: train_loss = 1.5213
[Epoch  9/30 Iter    32/32]: train_loss = 1.4757
[Epoch 10/30 Iter    32/32]: train_loss = 1.4243
[Epoch 11/30 Iter    32/32]: train_loss = 1.3421
[Epoch 12/30 Iter    32/32]: train_loss = 1.3469
[Epoch 13/30 Iter    32/32]: train_loss = 1.2983
[Epoch 14/30 Iter    32/32]: train_loss = 1.2284
[Epoch 15/30 Iter    32/32]: train_loss = 1.1556
[Epoch 16/30 Iter    32/32]: train_loss = 1.1114
[Epoch 17/30 Iter    32/32]: train_loss = 1.1526
[Epoch 18/30 Iter    32/32]: train_loss = 1.1184
[Epoch 19/30 Iter    32/32]: train_loss = 0.9794
[Epoch 20/30 Iter    32/32]: train_loss = 1.0344
[Epoch 21/30 Iter    32/32]: train_loss = 0.9484
[Epoch 22/30 Iter    32/32]: train_loss = 0.9170
[Epoch 23/30 Iter    32/32]: train_loss = 0.9083
[Epoch 24/30 Iter    32/32]: train_loss = 0.7944
[Epoch 25/30 Iter    32/32]: train_loss = 0.8187
[Epoch 26/30 Iter    32/32]: train_loss = 0.7778
[Epoch 27/30 Iter    32/32]: train_loss = 0.7031
[Epoch 28/30 Iter    32/32]: train_loss = 0.6073
[Epoch 29/30 Iter    32/32]: train_loss = 0.6314
[Epoch 30/30 Iter    32/32]: train_loss = 0.6600
```

Evaluate the model

```
evaluate(resnet18)
```

```
Accuracy = 43.40%
```

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 resnet18 with the pretrained model and use it to **initialize** the network. To do this, we set `pretrained=True`.
- 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`.

```
resnet18 = build_network(pretrained=True)
```

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

```
for name, param in resnet18.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
layer1.1.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
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.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
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.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
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
```

Train the model and save into `history2`.

```
1 history2 = train(resnet18, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 1.6447
[Epoch  2/30 Iter    32/32]: train_loss = 0.7526
[Epoch  3/30 Iter    32/32]: train_loss = 0.4089
[Epoch  4/30 Iter    32/32]: train_loss = 0.2350
[Epoch  5/30 Iter    32/32]: train_loss = 0.1499
[Epoch  6/30 Iter    32/32]: train_loss = 0.1051
[Epoch  7/30 Iter    32/32]: train_loss = 0.1027
[Epoch  8/30 Iter    32/32]: train_loss = 0.0601
[Epoch  9/30 Iter    32/32]: train_loss = 0.0878
[Epoch 10/30 Iter    32/32]: train_loss = 0.0612
[Epoch 11/30 Iter    32/32]: train_loss = 0.0394
[Epoch 12/30 Iter    32/32]: train_loss = 0.0568
[Epoch 13/30 Iter    32/32]: train_loss = 0.0937
[Epoch 14/30 Iter    32/32]: train_loss = 0.0637
[Epoch 15/30 Iter    32/32]: train_loss = 0.1105
[Epoch 16/30 Iter    32/32]: train_loss = 0.1112
[Epoch 17/30 Iter    32/32]: train_loss = 0.0511
[Epoch 18/30 Iter    32/32]: train_loss = 0.0666
[Epoch 19/30 Iter    32/32]: train_loss = 0.1069
[Epoch 20/30 Iter    32/32]: train_loss = 0.0754
[Epoch 21/30 Iter    32/32]: train_loss = 0.0848
[Epoch 22/30 Iter    32/32]: train_loss = 0.0684
[Epoch 23/30 Iter    32/32]: train_loss = 0.0437
[Epoch 24/30 Iter    32/32]: train_loss = 0.0254
[Epoch 25/30 Iter    32/32]: train_loss = 0.0208
[Epoch 26/30 Iter    32/32]: train_loss = 0.0140
[Epoch 27/30 Iter    32/32]: train_loss = 0.0245
[Epoch 28/30 Iter    32/32]: train_loss = 0.0106
[Epoch 29/30 Iter    32/32]: train_loss = 0.0200
[Epoch 30/30 Iter    32/32]: train_loss = 0.0296
```

Evaluate the network

```
evaluate(resnet18)
```

```
Accuracy = 80.00%
```

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.

```
# Load the pretrained model
resnet18 = build_network(pretrained=True)
```

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

```
parameters = list(resnet18.parameters())
for param in parameters[:-2]:
    param.requires_grad = False
```

```
for name, param in resnet18.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 : 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.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
```

Train the model and save into `history3`.

```
history3 = train(resnet18, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 1.8724
[Epoch  2/30 Iter    32/32]: train_loss = 1.1564
[Epoch  3/30 Iter    32/32]: train_loss = 1.0614
[Epoch  4/30 Iter    32/32]: train_loss = 0.9169
[Epoch  5/30 Iter    32/32]: train_loss = 0.8257
[Epoch  6/30 Iter    32/32]: train_loss = 0.8087
[Epoch  7/30 Iter    32/32]: train_loss = 0.7397
[Epoch  8/30 Iter    32/32]: train_loss = 0.7282
[Epoch  9/30 Iter    32/32]: train_loss = 0.7335
[Epoch 10/30 Iter    32/32]: train_loss = 0.6554
[Epoch 11/30 Iter    32/32]: train_loss = 0.6712
[Epoch 12/30 Iter    32/32]: train_loss = 0.6205
[Epoch 13/30 Iter    32/32]: train_loss = 0.6171
[Epoch 14/30 Iter    32/32]: train_loss = 0.6021
[Epoch 15/30 Iter    32/32]: train_loss = 0.5648
[Epoch 16/30 Iter    32/32]: train_loss = 0.5623
[Epoch 17/30 Iter    32/32]: train_loss = 0.5568
[Epoch 18/30 Iter    32/32]: train_loss = 0.5751
[Epoch 19/30 Iter    32/32]: train_loss = 0.5802
[Epoch 20/30 Iter    32/32]: train_loss = 0.5615
[Epoch 21/30 Iter    32/32]: train_loss = 0.5329
[Epoch 22/30 Iter    32/32]: train_loss = 0.5644
[Epoch 23/30 Iter    32/32]: train_loss = 0.5365
[Epoch 24/30 Iter    32/32]: train_loss = 0.5710
[Epoch 25/30 Iter    32/32]: train_loss = 0.5519
```

```
[Epoch 26/30 Iter    32/32]: train_loss = 0.5390
[Epoch 27/30 Iter    32/32]: train_loss = 0.5258
[Epoch 28/30 Iter    32/32]: train_loss = 0.5146
[Epoch 29/30 Iter    32/32]: train_loss = 0.4891
[Epoch 30/30 Iter    32/32]: train_loss = 0.5192
```

Evaluate the model

```
evaluate(resnet18)
```

```
Accuracy = 67.30%
```

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.

```
# Load the pretrained model
resnet18 = build_network(pretrained=True)
```

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

```
for name, param in resnet18.named_parameters():
    if not any(name.startswith(ext) for ext in ['layer4', 'fc']):
        param.requires_grad = False
```

```
for name, param in resnet18.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
```

Train the model and save into `history4`.

```
1 history4 = train(resnet18, num_epochs=30, lr=0.01, momentum=0.8)
```

```
[Epoch  1/30 Iter    32/32]: train_loss = 1.5520
[Epoch  2/30 Iter    32/32]: train_loss = 0.6816
[Epoch  3/30 Iter    32/32]: train_loss = 0.4301
[Epoch  4/30 Iter    32/32]: train_loss = 0.3384
[Epoch  5/30 Iter    32/32]: train_loss = 0.2128
[Epoch  6/30 Iter    32/32]: train_loss = 0.1631
[Epoch  7/30 Iter    32/32]: train_loss = 0.1401
[Epoch  8/30 Iter    32/32]: train_loss = 0.0918
[Epoch  9/30 Iter    32/32]: train_loss = 0.0801
[Epoch 10/30 Iter    32/32]: train_loss = 0.0634
[Epoch 11/30 Iter    32/32]: train_loss = 0.0616
[Epoch 12/30 Iter    32/32]: train_loss = 0.0706
[Epoch 13/30 Iter    32/32]: train_loss = 0.0577
[Epoch 14/30 Iter    32/32]: train_loss = 0.0368
[Epoch 15/30 Iter    32/32]: train_loss = 0.0304
[Epoch 16/30 Iter    32/32]: train_loss = 0.0354
[Epoch 17/30 Iter    32/32]: train_loss = 0.0305
[Epoch 18/30 Iter    32/32]: train_loss = 0.0242
[Epoch 19/30 Iter    32/32]: train_loss = 0.0182
[Epoch 20/30 Iter    32/32]: train_loss = 0.0197
[Epoch 21/30 Iter    32/32]: train_loss = 0.0302
[Epoch 22/30 Iter    32/32]: train_loss = 0.0199
```

```
[Epoch 23/30 Iter    32/32]: train_loss = 0.0129
[Epoch 24/30 Iter    32/32]: train_loss = 0.0197
[Epoch 25/30 Iter    32/32]: train_loss = 0.0300
[Epoch 26/30 Iter    32/32]: train_loss = 0.0159
[Epoch 27/30 Iter    32/32]: train_loss = 0.0180
[Epoch 28/30 Iter    32/32]: train_loss = 0.0121
[Epoch 29/30 Iter    32/32]: train_loss = 0.0234
[Epoch 30/30 Iter    32/32]: train_loss = 0.0329
```

Evaluate the model

```
evaluate(resnet18)
```

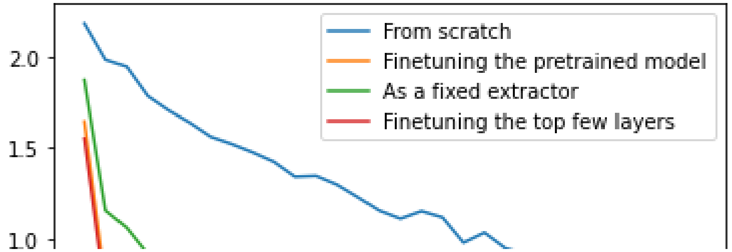
```
Accuracy = 78.20%
```

Plotting training loss

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

```
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.

```
def build_network(pretrained=True):
    ...
    return efficientNet
```

```
efficientNet = build_network()
```

```
history5 = train(efficientNet, num_epochs=30, lr=0.01, momentum=0.8)
```

```
evaluate(efficientNet)
```