SIT789 – Robotics, Computer Vision and Speech Processing

Credit Task 5.1: Deep learning for Computer Vision

Objectives

The objectives of this lab include:

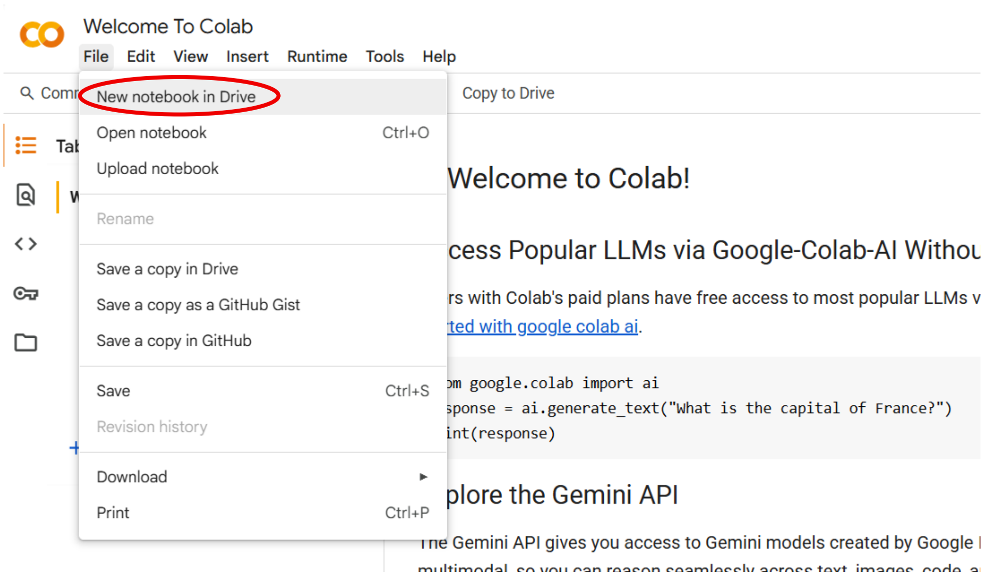
- Getting started with PyTorch in Google Colab
- Practising deep learning with PyTorch for solving fundamental Computer Vision tasks (image recognition and object detection)
- Experimenting with common practice of deep learning (e.g., fine-tuning, customised training)

Tasks

1. GPU usage in Google Colab

GPU (Graphics Processing Unit) is a specialised computing technology originally designed for image processing and computer graphics tasks but then extended to support parallel computations. GPUs have been intensively used in the fields of Artificial Intelligence, Machine Learning and Computer Vision. In this lab, we will explore the computational power of GPUs available in Google Colab. To use Google Colab, you need to register a free google account and then log-in using your Google account username and password.

To access Google Colab notebook, you can navigate to the following site from your browser: https://colab.research.google.com/notebooks/intro.ipynb. Firefox/Chrome are strongly recommended.
On the menu bar, select **File** and **New notebook in Drive** as follows,



Welcome to Colab!

cess Popular LLMs via Google-Colab-AI Withou

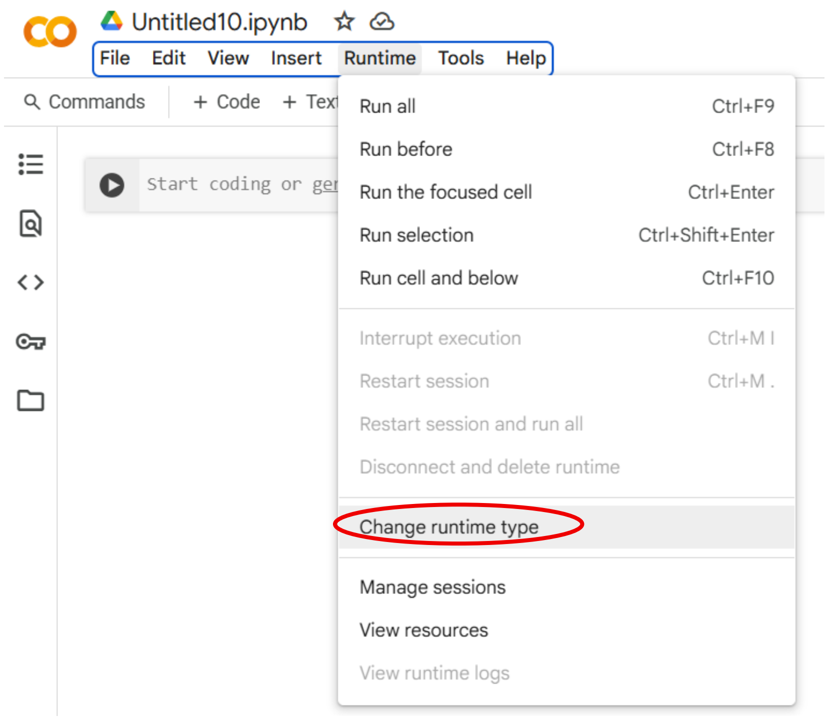
rs with Colab's paid plans have free access to most popular LLMs v
rted with google colab ai.

```
om google.colab import ai
sponse = ai.generate_text("What is the capital of France?")
int(response)
```

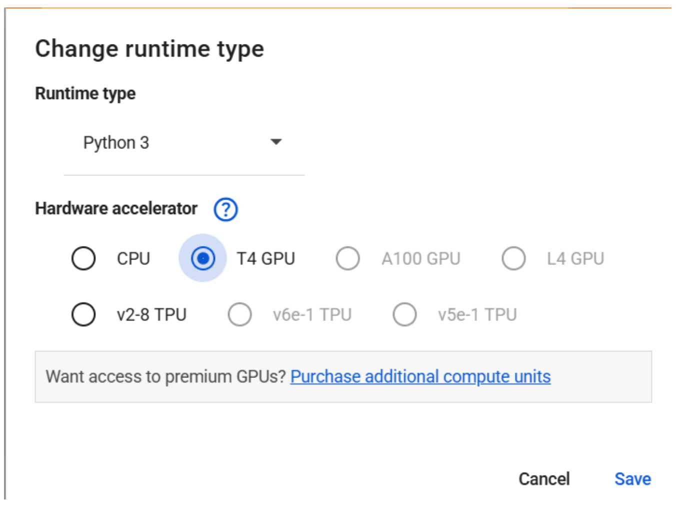
plore the Gemini API

The Gemini API gives you access to Gemini models created by Google I

We then set the runtime type of our new notebook as GPU by doing.



and then select **GPU** and **Save**



Change runtime type

Runtime type

Python 3

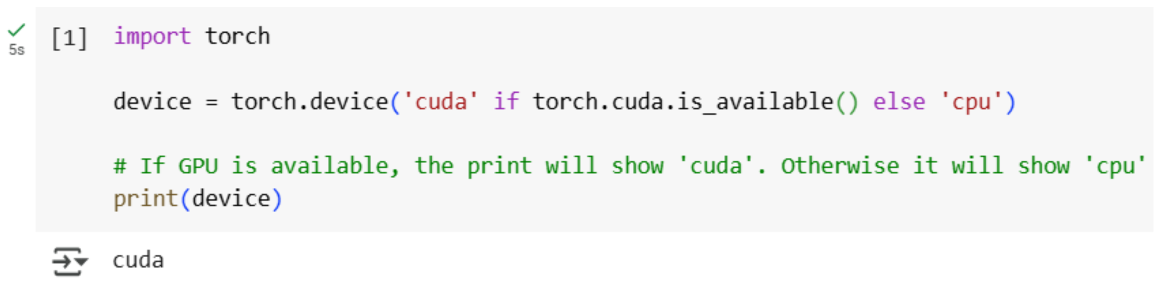
Hardware accelerator

CPU [✓ T4 GPU] A100 GPU L4 GPU

v2-8 TPU v6e-1 TPU v5e-1 TPU

Want access to premium GPUs? Purchase additional compute units

To test the GPU mode, we run the following code.



```
[1] import torch

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # If GPU is available, the print will show 'cuda'. Otherwise it will show 'cpu'
    print(device)
```

```
cuda
```

If ‘cuda’ is shown as above, you successfully access to GPU.

2. PyTorch

PyTorch is an open-source python-based computing package providing libraries for most of machine learning algorithms and frameworks. Many well-known network architectures with pre-trained models for various computer vision tasks are now available in PyTorch, e.g., ResNet, MobileNet, EfficientNet, etc. for image classification; Faster R-CNN, RetinaNet for object detection; DeepLab, U-Net for semantic segmentation; and, recently, Vision Transformers. See more at https://pytorch.org/vision/stable/models.html and https://pytorch.org/hub/research-models.

PyTorch is available in Google Colab. However, if you want to install PyTorch in your own computer. You can follow the instructions provided at https://pytorch.org/.

There are two important features supported by PyTorch including,

- a replacement for NumPy by Torch to exploit the power of GPUs
- a deep learning research platform that provides flexibility and high-speed computation

This section provides you with simple examples illustrating the aforementioned features. You are also recommended to further study PyTorch at https://pytorch.org/tutorials/.

PyTorch replaces arrays in NumPy by **Tensors** which optimise the computational speed on GPUs. For example, we can define a 5x3 array with randomly initialised values as follows.

```
import torch
x = torch.rand(5, 3)
print(x)
```

Like shape of NumPy, to check the size of a tensor, we can use size method as follow.

```
print(x.size())
```

Also like arrays, we can apply operators on tensors. For example, we can add two tensors as,

```
y = torch.rand(5, 3)
z = x + y
print(z)
```

We can convert a tensor to a NumPy array. For example,

```
t = x.numpy()
print(t)
```

Likewise, we can convert a NumPy array to a tensor. For example,

```
u = torch.from_numpy(t)
print(u)
```

As presented, another advanced feature of PyTorch is the convenience in constructing deep neural networks, in which torch.nn.Module is one of the most powerful functionalities of PyTorch. The following code shows you how to define a fully-connected neural network (i.e., two consecutive layers are fully connected) including one input layer, one hidden layer, and one output layer. More examples can be found at https://pytorch.org/tutorials/beginner/pytorch_with_examples.html.

In general, to define a neural network using torch.nn.Module, you need to

- Determine the architecture for your network (e.g., how many layers, their connections, etc.)
- Define a loss function
- Determine an optimizer and its parameters (e.g., learning rate, batch size, epoch)

You can make your network a subclass of torch.nn.Module and implement the forward method to fit with the architecture of your network as below

```python
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out): #D_in/H/D_out: dim of input/hidden/output layer
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Tensor of input data and we must return
        a Tensor of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Tensors.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
```

The network can be trained as follows.

```python
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Move the model to GPU
model.to(device)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x.to(device)) # Don’t forget x.to(device)

    # Compute and print loss
    loss = criterion(y_pred, y.to(device)) # Don’t forget y.to(device)
    if t % 100 == 99:
        print(t, loss.item())
    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

3. Image recognition

3.1. Image recognition using convolutional neural networks

In this section, we will learn how to build a convolutional neural network (CNN) for image recognition (image classification). We will use the CIFAR10 dataset as a case study. CIFAR10 includes 10 classes: ‘airplane’, ‘automobile’, ‘bird’, ‘cat’, ‘deer’, ‘dog’, ‘frog’, ‘horse’, ‘ship’, ‘truck’. The images in CIFAR10 are of size 3x32x32, i.e., 3-channel colour (RGB) images of 32x32 pixels in size. For more details, you are referred to https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

You are given the notebook “image_recognition_demo.ipynb” in the supplied resources. Here we will walk you through the supplied code. First, we need to download the training and test data. The following code will download the CIFAR10 dataset into 'data' folder in your current working directory. This code makes use of torchvision.datasets and torch.utils.data.DataLoader, which you can find more details at https://pytorch.org/vision/stable/datasets.html.

```
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data',
                                        train=True, #for training
                                        download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=4, #process 4 images at a time
                                          shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data',
                                       train=False, #not for training
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset,
                                         batch_size=4, #process 4 images at a time
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Note: In the above code, transforms.Normalize is used to scale the colour values in images' channels w.r.t. their means and standard deviations. Those statistics must be pre-computed. More details of transforms can be found at https://pytorch.org/vision/stable/transforms.html.

After loading the training data, we show several training images using the following code.

```
import matplotlib.pyplot as plt
import numpy as np

# function to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images))

# print true labels of the images
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

We now define our CNN. Our network receives inputs as 3x32x32-dimensional tensors (i.e., 3-channel images in size 32x32) and has the following architecture.

Layer	Input dimension	Output dimension
Input image		3x32x32
Conv1 (#kernels:6, kernel size:5x5, stride:1)	3x32x32	6x28x28
Pool1 (max pooling with size:2x2)	6x28x28	6x14x14
Conv2 (#kernels:16, kernel size:5x5, stride:1)	6x14x14	16x10x10
Pool2 (max pooling with size:2x2)	16x10x10	16x5x5
Fc1	16x5x5	120
Fc2	120	84
Fc3	84	10

Conv: convolutional layer, Pool: pooling layer, Fc: fully-connected layer.

More information about convolutional layers, max pooling layers, and fully-connected layers can be found at https://pytorch.org/docs/master/generated/torch.nn.Conv2d.html, https://pytorch.org/docs/master/generated/torch.nn.MaxPool2d.html, and https://pytorch.org/docs/master/generated/torch.nn.Linear.html, respectively.

We use relu for the activation functions. The code below shows how to define such a network. Since Pool1 and Pool2 are exactly same (i.e., the same max pooling operation and the same pooling size: 2x2), we will use only one nn.MaxPool2d(2,2) to implement both Pool1 and Pool2. You should also pay attention on the forward method, where we define how to process input data in forward direction.

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        f = self.pool(F.relu(self.conv1(x)))  #f = self.conv1(x),
                                              #f = F.relu(f),
                                              #f = self.pool(f)
        f = self.pool(F.relu(self.conv2(f)))  #f = self.conv2(f),
                                              #f = F.relu(f),
                                              #f = self.pool(f)

        f = f.view(-1, 16 * 5 * 5) #make a 16x5x5 tensor to 16x5x5-dimensional vector
        f = F.relu(self.fc1(f))
        f = F.relu(self.fc2(f))
        f = self.fc3(f)

        return f

net = Net()
```

Next, we define a loss function and an optimiser for our network. We will use Cross-Entropy loss and Adam optimizer, where the learning rate (lr) is set to 0.0005.

```
import torch.optim as optim
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.AdamW(net.parameters(), lr=0.0005) #lr: learning rate
```

Suppose that we train our network with epoch = 50, i.e., we scan the whole training dataset 50 times (you can use more epoch, e.g., 100+ epochs). The network can be trained using the following code. Note that, during training, we also log the loss function to keep track of the learning loss.

```
import time
import matplotlib.pyplot as plt

net.to(device)
start_time = time.time()
loss_history = []
epoch = 50

for e in range(epoch):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # set the parameter gradients to zeros
        optimizer.zero_grad()

        # forward
        outputs = net(inputs)

        #calculate errors
        loss = criterion(outputs, labels) #calculate errors

        #backward
        loss.backward()
        optimizer.step()

        running_loss += loss.item() # update total loss
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('epoch: %d, batches: %5d, loss: %.3f' %
                  (e + 1, i + 1, running_loss / 2000))
            loss_history.append(running_loss)
            running_loss = 0.0
```

We can measure the training time as:

```
print('Training time in %s seconds ----' % (time.time()- start_time))
```

The following code is then used to plot the learning loss

```
plt.plot(loss_history, label = 'training loss', color = 'r')
plt.legend(loc = ‘upper left’)
plt.show()
```

Since training takes time, we should save our model after the training is completed by doing.

```
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

Next, we will test our network. We first show several images from our test set and their corresponding labels from the ground truth.

```
dataiter = iter(testloader)
images, labels = next(dataiter)

# show images and print ground-truth labels
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

We test our network with the above examples as follows.

```
net = Net() #create a network
PATH = './cifar_net.pth'
net.load_state_dict(torch.load(PATH)) #load the trained network
net.to(device)
outputs = net(images.to(device)) #test the network with inputs as images
```

Recall that our network returns 10 probabilities (or confidence scores) corresponding to 10 classes; the higher a returned probability/confidence score to a class is, the more probably the input image belongs to that class. Therefore, to find the image class for each test image, we get the class corresponding to the highest score returned by the network. The code below shows you how to do so.

```
_, predicted_labels = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted_labels[j]]
                              for j in range(4)))
```

To fully and quantitatively evaluate the network, we calculate the network's overall recognition accuracy across all image classes on the entire test set as follows.

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, groundtruth_labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted_labels = torch.max(outputs.data, 1)
        total += groundtruth_labels.size(0)
        correct += (predicted_labels == groundtruth_labels).sum().item()
print('Recognition accuracy on the 10000 test images: %d %%' % (100 * correct / total))
```

The network, if trained properly, achieves around 65% for the overall recognition accuracy. Better networks can be found at: https://paperswithcode.com/sota/image-classification-on-cifar-10. More models can be found at: https://pytorch.org/vision/stable/models.html.

To further investigate the above result, we evaluate the per-class accuracy of the network as follows.

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, groundtruth_labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted_labels = torch.max(outputs, 1)
        c = (predicted_labels == groundtruth_labels).squeeze()
        for i in range(4):
            label = groundtruth_labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

Your task now is to calculate the confussion_matrix of the network. **Hint:** In the above evaluation code, groundtruth_labels and predicted_labels contain the ground truth labels and predicted labels for only test images in a batch. To calculate the confusion matrix on the entire test set, you will need to store the ground truth labels and predicted labels for ALL images in the test set.

Finally, you are to develop a neural network to recognise the food images in Task 4.1P. You can use the same architecture for the CIFAR10 dataset. However, you are free to try with different architectures to maximise the performance of your model. You can use the following code to define the trainset, testset, and food classes.

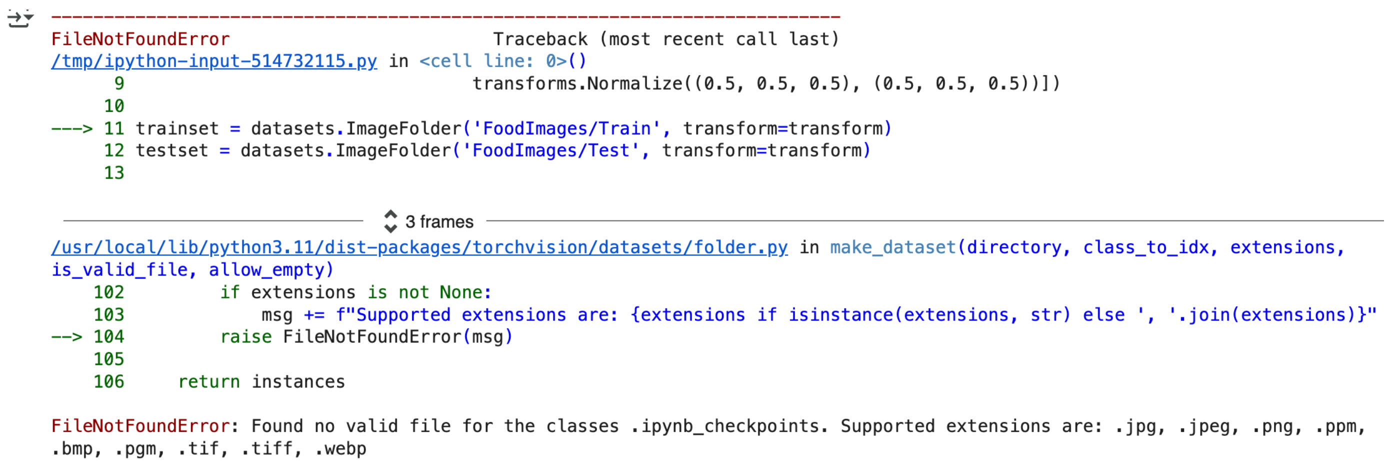
```
from torchvision import datasets

transform = transforms.Compose(
    [transforms.Resize((32, 32)), #you can change the image size, but need to change
                                  #the architecture accordingly
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = datasets.ImageFolder('FoodImages/Train', transform=transform)
testset = datasets.ImageFolder('FoodImages/Test', transform=transform)

classes = ('Cakes', 'Pasta', 'Pizza')
```

You may get the following error (in Google Colab):



```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
/tmp/ipython-input-514732115.py in <cell line: 0>()
      9                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
     10 
---> 11 trainset = datasets.ImageFolder('FoodImages/Train', transform=transform)
     12 testset = datasets.ImageFolder('FoodImages/Test', transform=transform)
     13 

                                  3 frames
/usr/local/lib/python3.11/dist-packages/torchvision/datasets/folder.py in make_dataset(directory, class_to_idx, extensions,
is_valid_file, allow_empty)
    102         if extensions is not None:
    103             msg += f"Supported extensions are: {extensions if isinstance(extensions, str) else ', '.join(extensions)}"
--> 104         raise FileNotFoundError(msg)
    105 
    106     return instances

FileNotFoundError: Found no valid file for the classes .ipynb_checkpoints. Supported extensions are: .jpg, .jpeg, .png, .ppm,
.bmp, .pgm, .tif, .tiff, .webp
```

This is because a file with extension "`.ipynb_checkpoints`" is created and added hiddenly besides the image class subfolders ("Cakes", "Pasta", "Pizza"). See more about this issue at https://www.programmersought.com/article/68145915110/.

You can remove such hidden files by using

```
!rm -R FoodImages/Train/.ipynb_checkpoints

!rm -R FoodImages/Test/.ipynb_checkpoints
```

Then, you can re-call the code lines for creating trainset and testset as usual.

Your task is to train your model and report the overall accuracy and confusion matrix of your developed deep learning model on the test set of the food image dataset. Note that you are also free to set learning parameters.
Your task is to train your model and report the overall accuracy and confusion matrix of your developed deep learning model on the test set of the food image dataset. Note that you are also free to set learning parameters.

3.2. Common practice

In this section, we explore other aspects of common practice of deep learning in image recognition, including fine-tuning and customised training.

3.2.1. Fine-tuning a CNN

Training a CNN as in 3.1 is called "training from scratch". However, one may want to adapt a CNN pre-trained from a source domain/dataset to a target domain/dataset. This type of training is called fine-tuning. Fine-tuning is often applied when the source dataset is rich and large while the target dataset is much smaller.

Here, suppose that we want to fine-tune the CNN, that has been pre-trained on CIFAR10, on the food image dataset in Task 4.1. We first need to load the CIFAR10 model using the following command.

```
PATH = './cifar_net.pth'
net = Net()
net.load_state_dict(torch.load(PATH))
```

Note that the CIFAR10 model has 10 output nodes corresponding to 10 categories while the food image dataset has only 3 food categories. We, therefore, need to replace the last fully-connected layer of the CIFAR10 model by a fully-connected layer with 3 output nodes as follows.

```
net.fc3 = nn.Linear(84, 3)
```

To check the architecture of the model, you can call:

```
print(net)
```

Your task now is to train and test the model, and report its overall accuracy and confusion matrix on the food image dataset. You are then to compare two training approaches: "training from scratch" and "fine-tuning" on the food image dataset, and draw conclusions.

3.2.2. Customised training

When fine-tuning a pre-trained model on a new dataset, one can choose to freeze some layers in the model and only train the remaining layers. This can be done by setting the requires_grad attribute of the parameters in the layers chosen to freeze to false. For instance, given the model which has been pre-trained on the CIFAR10 dataset in Section 3.1, we want to freeze the convolutional layer conv1 while letting the other layers in this model be trainable during fine-tuning the model on the food image dataset. We can run the following code prior to the fine-tuning. See more at this discussion post.

```
#freeze parameters of conv1
for param in net.conv1.parameters():
    param.requires_grad = False
```

To avoid mistake in asking the optimizer to optimize parameters without gradients, one is recommended to explicitly tell the optimizer to update parameters they want. For instance, if we just want to update parameters from layers other than conv1, we can apply a filter to parameters that need to be updated as below. Read more at this discussion post.

```
#update the optimizer
optimizer = optim.AdamW(filter(lambda p: p.requires_grad, net.parameters()), lr=0.0005)
```

You can double check the effect of requires_grad by printing out the parameters of conv1 before and after the fine-tuning when requires_grad is set to false and true respectively. The following code can be used to get the parameters of conv1.

```
print(net.conv1.weight)
print(net.conv1.bias)
```

Your task is to freeze both conv1 and conv2 during the fine-tuning of the model in Section 3.1 (i.e., only the fully-connected layers are fine-tuned) on the food image dataset.

4. Object detection

In this section, we will practice with Faster RCNN. Specifically, we will build a detector to detect sea creatures from the Aquarium dataset at https://public.roboflow.com/object-detection/aquarium.

First, you need to download the Aquarium dataset at the above link into your working directory. The dataset is organised into three folders: train, valid and test. The "train" folder contains images used to train the architecture of Faster RCNN. The "valid" folder contains images used to validate the detector. The validations in the "valid" folder are used to determine if the detector is overfitted (e.g., to determine which epoch should be used). The "test" folder contains images used to test the detector. Each "train", "valid", "test" folder contains an annotation file (.jason) for all the images included in the folder. The annotation file is written in the COCO format. See more about the COCO format at https://www.v7labs.com/blog/coco-dataset-guide.

You also need to download the notebook "object_detection_demo.ipynb" and code files (.py) in the supplied resources into your working directory. You need to walk through all the steps and instructions presented in the notebook. Your tasks include training and testing Faster RCNN with two different backbones: ResNet50 and MobileNet_v3. Pre-trained models of these backbones are available in PyTorch and can be called (see Step 5 in "object_detection_demol.ipynb") as,

- ResNet50:

```
from torchvision.models.detection.faster_rcnn import FasterRCNN_ResNet50_FPN_Weights

model = torchvision.models.detection.fasterrcnn_resnet50_fpn(
        weights=FasterRCNN_ResNet50_FPN_Weights.DEFAULT)
```

- MobileNetv3:

```
from torchvision.models.detection.faster_rcnn import FasterRCNN_MobileNet_V3_Large_320_
FPN_Weights

model = torchvision.models.detection.fasterrcnn_mobilenet_v3_large_320_fpn(
        weights=FasterRCNN_MobileNet_V3_Large_320_FPN_Weights.DEFAULT)
```

Submission instructions

1. Perform tasks required in Section 1, 2, 3 and 4.
2. Complete the supplied answer sheet with required results
3. Submit the answer sheet (.pdf) and code (.py) to OnTrack.