Deep Learning Model for Image Classification

DB927491 Ang Jian Hwee

DB927482 Emily Choo Hue Che

University of Macau

CISC3023 Pattern Recognition

Prof. Zhou Yicong

November 11, 2022

Contents

1. Introduction

Image Classification is an important problem in Computer Science and has many real-world applications such as Face recognition, Reverse Image Search, and other aspects. Many scholars have invented different models and approaches to deal with this problem such as LeNet, AlexNet, GoogLeNet, etc.

In this project, we face a typical image classification problem with 4 classes of satellite images. We experimented on 3 different models: Custom CNN (small model), transfer learning on ResNet18 (medium model), and transfer learning on VGG19 (large model) to investigate their performance and the effects of different model sizes. The results show that custom CNN models are able to achieve 96% accuracy after 50 epochs, while ResNet and VGG are able to achieve 98.49% and 98.23% just after 5 epochs. VGG takes a significantly longer time to train, therefore we propose that transfer learning on ResNet is the most suitable model.

We then investigate the possibilities of the underfitting problem. We conclude that transfer learning on ResNet is possible to encounter an overfitting problem. Eventually, we claim that epoch 3 from ResNet is our proposed model with 98.76% accuracy on the testing dataset.

In conclusion, our proposed model outperforms other baselines and is able to achieve around 99% accuracy with extremely small training epochs (~5 epochs). Our model is also medium-sized (~20MB), which makes it easier to deploy and not resource-demanding to train.

Keywords: *CNN, ResNet, VGG, Image Classification, Satellite*

2. Preprocessing

First, we import the library that we need.

```
# Import library
import os
import shutil
import random
import time

import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torchvision import transforms, datasets
import torchvision.models as models

# Set SEED
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

2.1 Import Dataset and Train Test Split

Set the training ratio to 0.8 and the test ratio to 0.2. We then set the ‘training_path’, ‘testing_path’, and ‘data_location’, which is originally provided by Kaggle.

```
train_ratio = 0.8
test_ratio = 0.2

train_path = r'./train/'
test_path = r'./test/'
data_location = r'/kaggle/input/satellite-image-classification/data'
```

We first list ‘data_location’ to get a list of all class names, which are ‘cloudy’, ‘desert’, ‘green_area’, and ‘water’.

```
class_name = os.listdir(data_location)
print(f"Class: {class_name}\n")
```

```
Class: ['cloudy', 'desert', 'green_area', 'water']
```

Then, we create a folder using the os.mkdir() method. If the directory that is being created already exists, this function raises a FileExistsError which will be captured.

```
try:
    os.mkdir(train_path)
    os.mkdir(test_path)
except FileExistsError:
    pass

for x in class_name:
    try:
        os.mkdir(os.path.join(train_path,x))
        os.mkdir(os.path.join(test_path,x))
    except FileExistsError:
        pass

for dirname, _, filenames in os.walk('./'):
        print(dirname)
```

And we print out the directory file names to check if there are no errors. We use os.walk() and get the result below.

```
./
./test
./test/green_area
./test/desert
./test/water
./test/cloudy
./.virtual_documents
./train
./train/green_area
./train/desert
./train/water
./train/cloudy
```

We remove any images that were previously saved in this location if exist. We use os.path.join() method to join the ‘training_path’ and current class, and set it as ‘cur_dir’. We then use os.remove() method to remove any files under that path. The same process is repeated for ‘testing_path’ and each class.

```
for x in class_name:
    cur_dir = os.path.join(train_path, x)
    [os.remove(os.path.join(cur_dir,x_)) for x_ in os.listdir(cur_dir)]
    cur_dir = os.path.join(test_path, x)
    [os.remove(os.path.join(cur_dir,x_)) for x_ in os.listdir(cur_dir)]
```

We use os.path.join() to join the original image folder and current ‘class_name’, and set this as ‘cur_dir’. Then use os.listdir() method to get the list of all the images in the directory ‘cur_dir’. Then, we used train_test_split from sklearn to split the images individually and shuffle them, extracting 80% of the images as the training data where the remaining images will be the testing data.

Then we use the shutil.copyfile() method to copy the split images to their relative folder. And we print out the count of images for training and testing. This process is then repeated for each class.

```
for x in class_name:
    cur_dir = os.path.join(data_location,x)
    print(f"{len(os.listdir(cur_dir))} image - {cur_dir}")

    train, test = train_test_split(os.listdir(cur_dir),train_size = 0.8,shuffle = True)

    for train_ in train:
        shutil.copyfile(os.path.join(cur_dir, train_),os.path.join(train_path, x, train_))

    for test_ in test:
        shutil.copyfile(os.path.join(cur_dir, test_),os.path.join(test_path, x, test_))

    print(f"train : {len(os.listdir(os.path.join(train_path,x)))} images")
    print(f"test  : {len(os.listdir(os.path.join(test_path,x)))} images")
    print()
```

We get the result below.

```
1500 image - /kaggle/input/satellite-image-classification/data/cloudy
train : 1200 images
test  : 300 images

1131 image - /kaggle/input/satellite-image-classification/data/desert
train : 904 images
test  : 227 images

1500 image - /kaggle/input/satellite-image-classification/data/green_area
train : 1200 images
test  : 300 images

1500 image - /kaggle/input/satellite-image-classification/data/water
train : 1200 images
test  : 300 images
```

Then, to see if the categorization of the image is accurate, we randomly print out the file names created by the os.walk() method in a directory tree. To simplify the output, we only print the first 3 file name as the output if there are more than 3 files in the folder. It should be noted that this will not affect the number of subfolders printed; all fsubolders will be printed.

```
for dirname, _, filenames in os.walk('./'):
    for x in filenames[0:3]:
        print(os.path.join(dirname,x))
    if len(filenames) > 3:
        print(dirname, "...")
    print()
```

We get the result below.

```
./test/green_area/Forest_2242.jpg
./test/green_area/Forest_319.jpg
./test/green_area/Forest_2995.jpg
./test/green_area ...

./test/desert/desert(7).jpg
./test/desert/desert(346).jpg
./test/desert/desert(839).jpg
./test/desert ...

./test/water/SeaLake_853.jpg
./test/water/SeaLake_462.jpg
./test/water/SeaLake_2863.jpg
./test/water ...

./test/cloudy/train_13047.jpg
./test/cloudy/train_21151.jpg
./test/cloudy/train_21872.jpg
./test/cloudy ...
```

```
./train/green_area/Forest_194.jpg
./train/green_area/Forest_495.jpg
./train/green_area/Forest_715.jpg
./train/green_area ...

./train/desert/desert(238).jpg
./train/desert/desert(221).jpg
./train/desert/desert(875).jpg
./train/desert ...

./train/water/SeaLake_2424.jpg
./train/water/SeaLake_1676.jpg
./train/water/SeaLake_2974.jpg
./train/water ...

./train/cloudy/train_16787.jpg
./train/cloudy/train_6105.jpg
./train/cloudy/train_5216.jpg
./train/cloudy ...
```

2.2 Normalization and Load Data using DataLoader

We use DataLoader to keep the data manageable and help to simplify the machine learning pipeline. We set the data's batch size to 64. We pre-calculate the mean of each RGB channel to be (0.4914, 0.4822, 0.4465) and the standard deviation to be (0.2023, 0.1994, 0.2010).

Next, we define the training and validation transforms pipeline. We first convert and save the images to tensor using transform.ToTensor() from torchvision, then we resize it to the targeted dimension 224 by 224. We then normalize it with our pre-calculated mean and standard deviation.

```
batch_size = 64
target_image_dim = 224
data_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Resize(target_image_dim),
        transforms.Normalize(mean=(0.4914, 0.4822, 0.4465),
                             std=(0.2023, 0.1994, 0.2010))
    ])
```

We used datasets.ImageFolder() from torchvision to prepare the dataset. It enables us to feed the image data into the model without manually loading labels. After processing, the image_folder object can then be fed into data loaders. We then use the DataLoader() to load our data, where the batch size was 64. We set the shuffle to be True to have the data reshuffled at every epoch and set the num_workers to be 4 which means 4 subprocesses to use. We repeat the same steps for loading the test data.

```
print(train_path)

train_dataset = datasets.ImageFolder(root=train_path,
                                     transform=data_transform)

train_dataset_loader = torch.utils.data.DataLoader(train_dataset,
                                       batch_size=batch_size, shuffle=True,
                                       num_workers=4)

print(train_dataset_loader.dataset.classes)

print(test_path)

test_dataset = datasets.ImageFolder(root=test_path,
                                    transform=data_transform)

test_dataset_loader = torch.utils.data.DataLoader(test_dataset,
                                       batch_size=batch_size, shuffle=True,
                                       num_workers=4)

print(test_dataset_loader.dataset.classes)
```

Then, we check if the classes are all correctly labeled. The classes for training data and test data should be the same, which are [‘cloudy’, ‘desert’, ‘green_area’, ‘water’].

```
./train/
['cloudy', 'desert', 'green_area', 'water']
./test/
['cloudy', 'desert', 'green_area', 'water']
```

```
count = 0
for x in list(enumerate(train_dataset_loader)):
    count += x[1][0].shape[0]
print(f"{count:5d} train images, each with dimension {list(x[1][0].shape)[1:]}")

count = 0
for x in list(enumerate(test_dataset_loader)):
    count += x[1][0].shape[0]
print(f"{count:5d} test images, each with dimension {list(x[1][0].shape)[1:]}")
```

```
4504 train images, each with dimension [3, 224, 224]
1127 test images, each with dimension [3, 224, 224]
```

3. Model

3.1 Custom CNN (Small Model)

Convolutional Neural Network (CNN) is a type of neural network whose main purpose is to perform image processing, image classification, and processing of other image-like data. A CNN has 3 types of layers: convolutional layers, pooling layers, and fully-connected layers. CNN usually has 2 modules, the Feature Extraction module, and the Classification module.

3.1.1 Model Architecture

Our model has two modules: feature extraction module and classification module.

For the feature extraction module, we first use a 5 by 5 convolution layer with 3 channel input and 6 channel output, then pass it through a 2 by 2 max pooling layer. And next, pass it through another 5 by 5 convolution layer with 6 channel input and 16 channel output.

For the classification module, we flatten the output from the feature extraction module and pass it through 3 fully connected layers, with a final out_feature of 4.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # Feature Extraction module
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # Classification module
        self.fc1 = nn.Linear(212*212, 120)
        self.fc2 = nn.Linear(120, 84)

        # Only 4 class at the end
        self.fc3 = nn.Linear(84, 4)
```

Now, we define the feed-forward method.

After all convolution and max pooling operations, we apply Rectified Linear Unit(ReLU) and pass the convolution layer 1 through the ReLU function, then into the max pooling layer, and again pass the convolution layer 2 through the ReLU function, then into the max pooling layer. Then we use flatten() to flatten all the dimensions from a multi-dimensional array to a 1-dimensional array.

Next, pass it through the fully connect layered 1 defined above, and then into the ReLU function. Then we repeat this step for the fully connected layer 2 and stop after passing through fully connected layer 3.

```
def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = torch.flatten(x, 1) # flatten all dimensions except batch
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

Next, we define the loss function as CrossEntropyLoss()and define the optimizer as Adam(). Note that these criteria and optimizers are constant across all models. This is to make sure the speed of convergence is measured under the same condition across all models.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters())
```

3.1.2 Model Training

First, we feed forward and calculate the loss, then use it to do the backpropagation for backpropagation. Then, we use the optimizer to update the parameters as follows.

```
running_loss = 0.0
for i, data in enumerate(train_dataset_loader, 0):
    # get the inputs; data is a list of [inputs, labels]
    inputs, labels = data

    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

To check on the status of the epoch, we set i=16 so that it will print out the current cumulative loss within the epoch every 16 mini batches run.

```
# Cumulative loss
running_loss += loss.item()
if i % 16 == 0:    # print every 16 mini-batches

    # Logging
    log += (f'Epoch {epoch + 1} | batch {i:5d} | cumulative loss within epoch: {running_loss / 20:.3f}\n')

    # Only print last 5 epoch
    if epoch >=45:
        print(f'Epoch {epoch + 1} | batch {i:5d} | cumulative loss within epoch: {running_loss / 20:.3f}')
```

Then we log it and later save this log to ‘train_log_CNN.txt’.

```
# Logging
log += (f'Saving model as "./model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt"\n\n')
```

```
# Save Model
torch.save(net.state_dict(), f'./model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt')
```

```
with open("train_log_CNN.txt", "w+") as file:
    file.write(log)
```

Here is part of the logs for epoch 49 and epoch 50.

```
Epoch 49 | batch     0 | cumulative loss within epoch: 0.001
Epoch 49 | batch    16 | cumulative loss within epoch: 0.058
Epoch 49 | batch    32 | cumulative loss within epoch: 0.104
Epoch 49 | batch    48 | cumulative loss within epoch: 0.134
Epoch 49 | batch    64 | cumulative loss within epoch: 0.183
Saving model as "./model_00049_loss_0.196.pt"

Epoch 50 | batch     0 | cumulative loss within epoch: 0.006
Epoch 50 | batch    16 | cumulative loss within epoch: 0.034
Epoch 50 | batch    32 | cumulative loss within epoch: 0.073
Epoch 50 | batch    48 | cumulative loss within epoch: 0.090
Epoch 50 | batch    64 | cumulative loss within epoch: 0.132
Saving model as "./model_00050_loss_0.140.pt"
```

Then we do a quick verification of our trained model. We print out the overall accuracy on train with accuracy_score(), recall_score(), precision_score(), and f1_score() from sklearn.metrics. And repeat these steps to print out the overall accuracy of the test.

```
main_labels, main_outputs = [], []
for i, data in enumerate(train_dataset_loader, 0):
    inputs, labels = data
    outputs = net(inputs)
    main_labels.extend(labels.tolist())
    main_outputs.extend(torch.argmax(outputs,axis = 1).tolist())

print("Overall Accuracy on train")
print("accuracy_score  :",accuracy_score(main_labels, main_outputs).round(4))
print("recall_score    :",recall_score(main_labels, main_outputs,average='macro').round(4))
print("precision_score :",precision_score(main_labels, main_outputs,average='macro').round(4))
print("f1_score        :",f1_score(main_labels, main_outputs,average='macro').round(4))
```

```
main_labels, main_outputs = [], []
for i, data in enumerate(test_dataset_loader, 0):
    inputs, labels = data
    outputs = net(inputs)
    main_labels.extend(labels.tolist())
    main_outputs.extend(torch.argmax(outputs,axis = 1).tolist())

print("Overall Accuracy on test")
print("accuracy_score  :",accuracy_score(main_labels, main_outputs).round(4))
print("recall_score    :",recall_score(main_labels, main_outputs,average='macro').round(4))
print("precision_score :",precision_score(main_labels, main_outputs,average='macro').round(4))
print("f1_score        :",f1_score(main_labels, main_outputs,average='macro').round(4))
```

And we get the result below.

```
Overall Accuracy on train
accuracy_score  : 0.9858
recall_score    : 0.9858
precision_score : 0.9859
f1_score        : 0.9858
```

```
Overall Accuracy on test
accuracy_score  : 0.9645
recall_score    : 0.9651
precision_score : 0.9674
f1_score        : 0.966
```

We can see that the final model accuracy is around 0.98 on training data and around 0.96 on testing data.

3.2 Transfer Learning using ResNet (Medium Model)

Residual Neural Network (ResNet) is a variant of the HighwayNet which is gateless and is usually implemented with double or triple-layer skips, for which the main purpose is to avoid vanishing gradient problems. It is first introduced in "Deep Residual Learning for Image Recognition" by He et al., the Year 2016.

The model we used is ResNet18 which has 18 layers and 11 million trainable parameters.

3.2.1 Model Architecture

First, we download the pre-trained weights from torchvision.models.resnet18 to use. Then we set the 'requires_grad' to false to freeze all layers from ResNet to avoid changing its parameters. We then create a single-layer fully connected network with 1000 in_features and 4 out_features. Then use the nn.Sequential() to run the layers sequentially.

```
# Download Weights (this will initiate a download if not cached)
resnet_model = models.resnet18(pretrained=True)

# Set all layer to be frozen
for param in resnet_model.parameters():
    param.requires_grad = False

# add another fc layer at the end
resnet_model = nn.Sequential(resnet_model, nn.Linear(in_features=1000, out_features=4))
```

Now we print the 'requires_grad' for all layers to make sure that only the last two layers are trainable.

```
# requires_grad to be True only for last layer
# Check, last 2 should be 1
print([int(x.requires_grad) for x in resnet_model.parameters()])
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
```

This is our final model architecture.

```
Sequential(
  (0): 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): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=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)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=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)
      )
    )

  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=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)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=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)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=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)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=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)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=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)
      (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): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=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)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
(1): Linear(in_features=1000, out_features=4, bias=True)
)
```

Then we define the loss function as CrossEntropyLoss() and define the optimizer as Adam(), the same as the Custom CNN model.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet_model.parameters())
```

3.2.2 Model Training

First, we fit forward and calculate the loss, then use it to do the backpropagation for fitting backward. Then, we use the optimizer to update the parameters as follows.

```
log = ""
for epoch in range(5):  # 5 Epoch is enough
    running_loss = 0.0
    for i, data in enumerate(train_dataset_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = resnet_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Cumulative loss
        running_loss += loss.item()
```

The experiment shows that 5 epochs are enough for ResNet, so we print all logs for each epoch.

```
# Cumulative loss
running_loss += loss.item()

print(f'Epoch {epoch + 1} | batch {i:5d} | cumulative loss within epoch: {running_loss / 20:.3f}')

# Logging
log += (f'Epoch {epoch + 1} | batch {i:5d} | cumulative loss within epoch: {running_loss / 20:.3f}\n')
```

Then we write it into a log and save it into a file named ‘train_log_ResNet.txt’.

```
log += (f'Saving model as "./resnet_model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt"\n\n')
print(f'Saving model as "./resnet_model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt"\n')
torch.save(resnet_model.state_dict(), f'./resnet_model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt')
```

```
with open("train_log_ResNet.txt", "w+") as file:
    file.write(log)
```

F

Here is part of the output logs.

```
Epoch 4 | batch     0 | cumulative loss within epoch: 0.001
Epoch 4 | batch    16 | cumulative loss within epoch: 0.021
Epoch 4 | batch    32 | cumulative loss within epoch: 0.042
Epoch 4 | batch    48 | cumulative loss within epoch: 0.054
Epoch 4 | batch    64 | cumulative loss within epoch: 0.087
Saving model as "./resnet_model_00004_loss_0.096.pt"

Epoch 5 | batch     0 | cumulative loss within epoch: 0.001
Epoch 5 | batch    16 | cumulative loss within epoch: 0.020
Epoch 5 | batch    32 | cumulative loss within epoch: 0.041
Epoch 5 | batch    48 | cumulative loss within epoch: 0.062
Epoch 5 | batch    64 | cumulative loss within epoch: 0.080
Saving model as "./resnet_model_00005_loss_0.087.pt"
```

Similarly, we do a quick analysis of model performance. We print out the overall accuracy on train with accuracy_score(), recall_score(), precision_score(), and f1_score() from sklearn.metrics. And repeat these steps to print out the overall accuracy of the test.

```
main_labels, main_outputs = [], []
for i, data in enumerate(train_dataset_loader, 0):
    inputs, labels = data
    outputs = resnet_model(inputs)
    main_labels.extend(labels.tolist())
    main_outputs.extend(torch.argmax(outputs,axis = 1).tolist())

print("Overall Accuracy on train")
print("accuracy_score  :",accuracy_score(main_labels, main_outputs).round(4))
print("recall_score    :",recall_score(main_labels, main_outputs,average='macro').round(4))
print("precision_score :",precision_score(main_labels, main_outputs,average='macro').round(4))
print("f1_score        :",f1_score(main_labels, main_outputs,average='macro').round(4))
```

```
main_labels, main_outputs = [], []
for i, data in enumerate(test_dataset_loader, 0):
    inputs, labels = data
    outputs = resnet_model(inputs)
    main_labels.extend(labels.tolist())
    main_outputs.extend(torch.argmax(outputs,axis = 1).tolist())

print("Overall Accuracy on test")
print("accuracy_score  :",accuracy_score(main_labels, main_outputs).round(4))
print("recall_score    :",recall_score(main_labels, main_outputs,average='macro').round(4))
print("precision_score :",precision_score(main_labels, main_outputs,average='macro').round(4))
print("f1_score        :",f1_score(main_labels, main_outputs,average='macro').round(4))
```

And we get the result below.

```
Overall Accuracy on train
accuracy_score  : 0.9918
recall_score    : 0.992
precision_score : 0.9923
f1_score        : 0.9921
```

```
Overall Accuracy on test
accuracy_score  : 0.9911
recall_score    : 0.9917
precision_score : 0.9918
f1_score        : 0.9917
```

ResNet model has around 0.99 accuracies on both the training dataset and testing dataset.

3.3 Transfer Learning using VGG (Large Model)

Visual Geometry Group (VGG) is a standard deep Convolutional Neural Network. It is a large-scale CNN model with multiple layers. It is first introduced in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" by Simonyan, K., & Zisserman, A., the Year 2014 in the submission to the Large Scale Visual Recognition Challenge 2014 (ILSVRC2014).

The model we used is VGG19 which has 19 layers and 138 million trainable parameters.

3.3.1 Model Architecture

First, we download the pre-trained weights from torchvision.models.vgg19 to use. Then we set the 'requires_grad' to false to freeze the original models to avoid updating the originally fitted parameters.

We then create a single-layer fully connected network with 1000 inputs and 4 outputs, then use the nn.Sequential() to run the layers sequentially.

```
# Download Weights (this will initiate a download if not cached)
vgg_model = models.vgg19(pretrained=True)

# Set all layer to be frozen
for param in vgg_model.parameters():
    param.requires_grad = False

# add another fc layer at the end
vgg_model = nn.Sequential(vgg_model, nn.Linear(in_features=1000, out_features=4))
```

We then print to check if only the last two layers are trainable.

```
# requires_grad to be True only for last layer
# Check, last 2 should be 1
print([int(x.requires_grad) for x in vgg_model.parameters()])
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
```

This is our final model architecture.

```
Sequential(
  (0): VGG(
    (features): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace=True)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (6): ReLU(inplace=True)
      (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (8): ReLU(inplace=True)
      (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (13): ReLU(inplace=True)
      (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (15): ReLU(inplace=True)
      (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (17): ReLU(inplace=True)
      (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace=True)
      (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (24): ReLU(inplace=True)
      (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (26): ReLU(inplace=True)
      (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (31): ReLU(inplace=True)
      (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (33): ReLU(inplace=True)
      (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (35): ReLU(inplace=True)
      (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
  (1): Linear(in_features=1000, out_features=4, bias=True)
  )
```

We define the loss function as CrossEntropyLoss()and define the optimizer as Adam(), the same as before.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(vgg_model.parameters())
```

3.3.2 Model Training

First, we fit forward and calculate the loss, then use it to do the backpropagation. Then, we use the optimizer to update the parameters as follows.

```
log = ""
for epoch in range(5):  # 5 Epoch is enough
    running_loss = 0.0
    for i, data in enumerate(train_dataset_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = vgg_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Cumulative loss
        running_loss += loss.item()
```

We found that the VGG model can achieve a fairly high accuracy after 5 epochs. Therefore, we print out all the logs in these 5 epochs.

```
# Cumulative loss
running_loss += loss.item()

print(f'Epoch {epoch + 1} | batch {i:5d} | cumulative loss within epoch: {running_loss / 20:.3f}')

# Logging
log += (f'Epoch {epoch + 1} | batch {i:5d} | cumulative loss within epoch: {running_loss / 20:.3f}\n')
```

Then we write it into a log that will later be saved as a file named 'train_log_VGG.txt'.

```
log += (f'Saving model as "./vgg_model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt"\n\n')
print(f'Saving model as "./vgg_model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt"\n')
torch.save(vgg_model.state_dict(), f'./vgg_model_{epoch+1:05d}_loss_{running_loss / 20:.3f}.pt')
```

```
with open("train_log_VGG.txt", "w+") as file:
    file.write(log)
```

Here are some of the training logs.

```
Epoch 4 | batch     0 | cumulative loss within epoch: 0.006
Epoch 4 | batch    16 | cumulative loss within epoch: 0.082
Epoch 4 | batch    32 | cumulative loss within epoch: 0.167
Epoch 4 | batch    48 | cumulative loss within epoch: 0.246
Epoch 4 | batch    64 | cumulative loss within epoch: 0.314
Saving model as "./vgg_model_00004_loss_0.344.pt"

Epoch 5 | batch     0 | cumulative loss within epoch: 0.003
Epoch 5 | batch    16 | cumulative loss within epoch: 0.069
Epoch 5 | batch    32 | cumulative loss within epoch: 0.128
Epoch 5 | batch    48 | cumulative loss within epoch: 0.200
Epoch 5 | batch    64 | cumulative loss within epoch: 0.270
Saving model as "./vgg_model_00005_loss_0.297.pt"
```

Then we rapidly validate model performance by calculating the overall performance on train data with accuracy_score(), recall_score(), precision_score(), and f1_score() from sklearn.metrics. These steps are repeated for testing data to print out the overall accuracy of the test.

```
main_labels, main_outputs = [], []
for i, data in enumerate(train_dataset_loader, 0):
    inputs, labels = data
    outputs = vgg_model(inputs)
    main_labels.extend(labels.tolist())
    main_outputs.extend(torch.argmax(outputs,axis = 1).tolist())

print("Overall Accuracy on train")
print("accuracy_score  :",accuracy_score(main_labels, main_outputs).round(4))
print("recall_score    :",recall_score(main_labels, main_outputs,average='macro').round(4))
print("precision_score :",precision_score(main_labels, main_outputs,average='macro').round(4))
print("f1_score        :",f1_score(main_labels, main_outputs,average='macro').round(4))
```

```
main_labels, main_outputs = [], []
for i, data in enumerate(test_dataset_loader, 0):
    inputs, labels = data
    outputs = vgg_model(inputs)
    main_labels.extend(labels.tolist())
    main_outputs.extend(torch.argmax(outputs,axis = 1).tolist())

print("Overall Accuracy on test")
print("accuracy_score  :",accuracy_score(main_labels, main_outputs).round(4))
print("recall_score    :",recall_score(main_labels, main_outputs,average='macro').round(4))
print("precision_score :",precision_score(main_labels, main_outputs,average='macro').round(4))
print("f1_score        :",f1_score(main_labels, main_outputs,average='macro').round(4))
```

And we get the result below.

```
Overall Accuracy on train
accuracy_score  : 0.974
recall_score    : 0.9725
precision_score : 0.974
f1_score        : 0.9732
```

```
Overall Accuracy on test
accuracy_score  : 0.9796
recall_score    : 0.9784
precision_score : 0.9798
f1_score        : 0.9791
```

Here we can see that the VGG model can achieve 0.97 accuracies on train and around 0.98 on test.

4. Performance Analysis and Model Selection

4.1 Calculation of Accuracy, Recall, Precision, and F1-score

We have all of the trained models (.pt) saved at https://www.kaggle.com/datasets/megahwee/cisc3024db927491angjianhwee-trained-models.

First, we add the data to the current notebook using “Add Data” at the right panels. Then use os.path.join() to get the complete path of all trained models. Here we filtered out those which start with “model” (by default model means CNN).

```
# For CNN
trained_model_path = r"../input/cisc3024db927491angjianhwee-trained-models"
models_path = sorted([os.path.join(trained_model_path,x) for x in os.listdir(trained_model_path) if x.startswith("model")])
models_path
```

And we will get the result below, which is saved models for CNN across all epochs.

```
['../input/cisc3024db927491angjianhwee-trained-models/model_00001_loss_2.040.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00002_loss_1.843.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00003_loss_1.245.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00004_loss_1.222.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00005_loss_1.100.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00006_loss_1.068.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00007_loss_0.965.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00008_loss_1.072.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00009_loss_0.975.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00010_loss_0.983.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00011_loss_0.960.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00012_loss_0.888.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00013_loss_0.734.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00014_loss_0.783.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00015_loss_0.740.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00016_loss_0.657.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00017_loss_0.560.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00018_loss_0.366.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00019_loss_0.467.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00020_loss_0.403.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00021_loss_0.380.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00022_loss_0.308.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00023_loss_0.298.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00024_loss_0.401.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00025_loss_0.265.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00026_loss_0.390.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00027_loss_0.351.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00028_loss_0.349.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00029_loss_0.393.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00030_loss_0.861.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00031_loss_0.281.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00032_loss_0.234.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00033_loss_0.403.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00034_loss_0.273.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00035_loss_0.223.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00036_loss_0.406.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00037_loss_0.270.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00038_loss_0.353.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00039_loss_0.211.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00040_loss_0.174.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00041_loss_0.162.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00042_loss_0.205.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00043_loss_0.266.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00044_loss_0.341.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00045_loss_0.270.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00046_loss_0.190.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00047_loss_0.263.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00048_loss_0.284.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00049_loss_0.196.pt',
 '../input/cisc3024db927491angjianhwee-trained-models/model_00050_loss_0.140.pt']
```

Now we initialize a fresh CNN model, load the weights for each epoch, do the prediction, and calculate ‘accuracy’, ‘recall’, ‘precision’, ‘F1 score’, and the total time is taken to predict all testing data.

```
# Initialize fresh model
net = Net()

# Get loss from filename
loss = [x[x.find("loss")+5:x.find("loss")+10] for x in models_path]

# Initialize array
test_accuracy_score = []
test_recall_score = []
test_precision_score = []
test_f1_score = []
test_prediction_time_taken = []

for x in models_path:
    # Record start time
    start = time.time()

    # Print every 5 epoch after evaluate
    if models_path.index(x) % 5 == 0:
        print(f"Working on model {models_path.index(x):3d}: {x}")

    # Load model
    net.load_state_dict(torch.load(x))

    # Model Prediction
    main_labels, main_outputs = [], []
    for i, data in enumerate(test_dataset_loader, 0):
        inputs, labels = data
        outputs = net(inputs)
        main_labels.extend(labels.tolist())
        main_outputs.extend(torch.argmax(outputs,axis = 1).tolist())

    # Calculate Stat
    test_accuracy_score.append(accuracy_score(main_labels, main_outputs).round(4))
    test_recall_score.append(recall_score(main_labels, main_outputs,average='macro').round(4))
    test_precision_score.append(precision_score(main_labels, main_outputs,average='macro').round(4))
    test_f1_score.append(f1_score(main_labels, main_outputs,average='macro').round(4))
    test_prediction_time_taken.append(time.time() - start)
```

We create a new pandas DataFrame() to store all the data.

```
df = pd.DataFrame([["CNN" for _ in models_path],
                   [models_path.index(x)+1 for x in models_path],
                   loss,
                   test_accuracy_score,
                   test_recall_score,
                   test_precision_score,
                   test_f1_score,
                   test_prediction_time_taken]).T
df.columns = ["type","epoch", "loss", "test_accuracy_score","test_recall_score",
              "test_precision_score", "test_f1_score","test_prediction_time_taken"]

df
```

Here is the result of the first and last 5 rows.

	type	epoch	loss	test_accuracy_score	test_recall_score	test_precision_score	test_f1_score	test_prediction_time_taken
0	CNN	1	2.040.	0.5138	0.5423	0.5813	0.4813	5.561126
1	CNN	2	1.843.	0.8571	0.8642	0.8657	0.8643	5.582082
2	CNN	3	1.245.	0.8589	0.8659	0.8719	0.8644	5.622688
3	CNN	4	1.222.	0.8625	0.8687	0.8744	0.8683	5.573508
4	CNN	5	1.100.	0.8713	0.8776	0.8817	0.878	5.559339
…								
45	CNN	46	0.190.	0.9255	0.9287	0.9405	0.9283	5.431942
46	CNN	47	0.263.	0.9556	0.9575	0.9566	0.9568	5.533185
47	CNN	48	0.284.	0.9796	0.9806	0.9796	0.9799	5.449032
48	CNN	49	0.196.	0.9627	0.9602	0.966	0.9625	5.513555
49	CNN	50	0.140.	0.9645	0.9651	0.9674	0.966	5.465121

We do the same for ResNet and VGG and get the saved model location.

```
Working on model   1: ../input/cisc3024db927491angjianhwee-trained-models/resnet_model_00001_loss_0.634.pt
Working on model   2: ../input/cisc3024db927491angjianhwee-trained-models/resnet_model_00002_loss_0.173.pt
Working on model   3: ../input/cisc3024db927491angjianhwee-trained-models/resnet_model_00003_loss_0.109.pt
Working on model   4: ../input/cisc3024db927491angjianhwee-trained-models/resnet_model_00004_loss_0.096.pt
Working on model   5: ../input/cisc3024db927491angjianhwee-trained-models/resnet_model_00005_loss_0.087.pt

Working on model   1: ../input/cisc3024db927491angjianhwee-trained-models/vgg_model_00001_loss_1.086.pt
Working on model   2: ../input/cisc3024db927491angjianhwee-trained-models/vgg_model_00002_loss_0.462.pt
Working on model   3: ../input/cisc3024db927491angjianhwee-trained-models/vgg_model_00003_loss_0.378.pt
Working on model   4: ../input/cisc3024db927491angjianhwee-trained-models/vgg_model_00004_loss_0.344.pt
Working on model   5: ../input/cisc3024db927491angjianhwee-trained-models/vgg_model_00005_loss_0.297.pt
```

Similarly, we calculate ‘accuracy’, ‘recall’, ‘precision’, ‘F1 score’, and the total time taken to predict all testing data again for both models and append them into the original DataFrame we created for Custom CNN. Here is the final result.

	type	epoch	loss	test_accuracy_score	test_recall_score	test_precision_score	test_f1_score	test_prediction_time_taken
0	CNN	1	2.040.	0.5138	0.5423	0.5813	0.4813	5.561126
…								
49	CNN	50	0.140.	0.9645	0.9651	0.9674	0.966	5.465121
0	ResNet	1	0.634.	0.9752	0.9764	0.9757	0.976	41.773079
1	ResNet	2	0.173.	0.9823	0.9833	0.9826	0.9829	40.566936
2	ResNet	3	0.109.	0.9876	0.9883	0.9879	0.988	40.256614
3	ResNet	4	0.096.	0.9876	0.9883	0.9876	0.9879	40.305889
4	ResNet	5	0.087.	0.9849	0.9856	0.9857	0.9856	42.658427
0	VGG	1	1.086.	0.9565	0.9551	0.9586	0.9562	321.051824
1	VGG	2	0.462.	0.9698	0.9693	0.9702	0.9697	333.236939
2	VGG	3	0.378.	0.9734	0.9721	0.9735	0.9727	328.008091
3	VGG	4	0.344.	0.9725	0.9718	0.9735	0.9725	332.433401
4	VGG	5	0.297.	0.9823	0.9815	0.982	0.9817	324.494083

4.2 Calculation of time taken to predict one batch of images

Since in section 4.1 we only recorded the time taken to predict the entire epoch, which includes the calculation of evaluation matrices, we are interested in the time required for each model to predict only one batch of images.

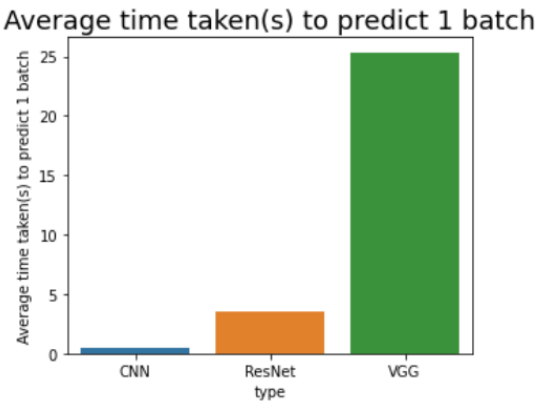
We stimulate the prediction process 10000 times and calculate the average time taken.

```
inputs = [(i, data) for i, data in enumerate(test_dataset_loader, 0)][0][1][0]
sample_size = int(1e4)
print(f"========== Average of {sample_size} sample ==========")

start = time.time()
for _ in range(sample_size):
    net(inputs)
print(f"(On CPU) Average time taken to predict 1 batch in CNN   : {(time.time()-start)/sample_size:5.4f}s")

start = time.time()
for _ in range(sample_size):
    resnet_model(inputs)
print(f"(On CPU) Average time taken to predict 1 batch in ResNet: {(time.time()-start)/sample_size:5.4f}s")

start = time.time()
for _ in range(sample_size):
    vgg_model(inputs)
print(f"(On CPU) Average time taken to predict 1 batch in VGG   : {(time.time()-start)/sample_size:5.4f}s")
```

```
========== Average of 10000 batch prediction ==========

(On CPU) Average time taken to predict 1 batch in CNN   : 0.4158s

(On CPU) Average time taken to predict 1 batch in ResNet: 3.5403s

(On CPU) Average time taken to predict 1 batch in VGG   : 25.3409s
```

And we visualize it as follows. We can see that it takes Custom CNN 0.41s to predict one batch (64 images), while ResNet takes 3.54s (around 8x of CNN), which is still tolerable. However, VGG takes 25.34s (around 60x of CNN), which is unacceptable for model deployment. This prediction time should be taken into account when we perform model selection in the next section.

```
df2 = pd.DataFrame(np.array([[0.4158,3.5403,25.3409],['CNN','ResNet','VGG']]).T,
                   columns = ['Average time taken(s) to predict 1 batch', 'type'])
df2['Average time taken(s) to predict 1 batch'] = df2['Average time taken(s) to predict 1 batch'].astype('float64')
df2['ratio'] = (df2['Average time taken(s) to predict 1 batch'] / df2['Average time taken(s) to predict 1 batch'].min()).round(2)

plt.figure(figsize=(5,4))
plt.title('Average time taken(s) to predict 1 batch',fontsize=18)
sns.barplot(data=df2, x="type", y="Average time taken(s) to predict 1 batch",)
plt.show()

df2
```

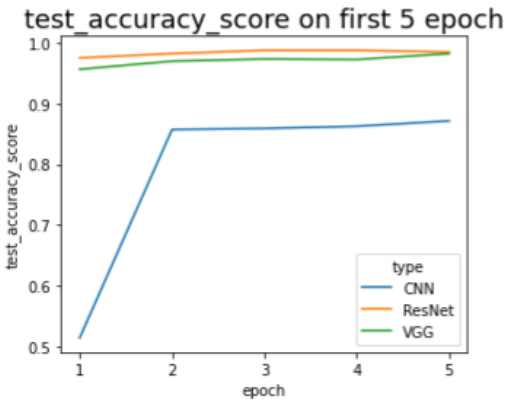


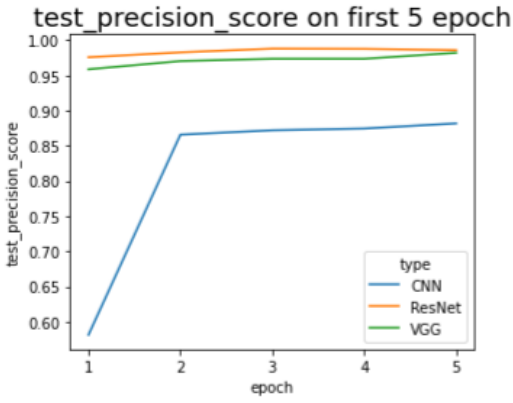
	Average time taken(s) to predict 1 batch	type	ratio
0	0.4158	CNN	1.00
1	3.5403	ResNet	8.51
2	25.3409	VGG	60.94

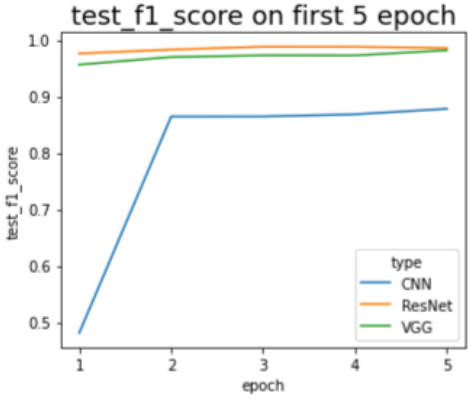
4.3 Model Selection

Initially, we train all models for 5 epochs. Here is the result.

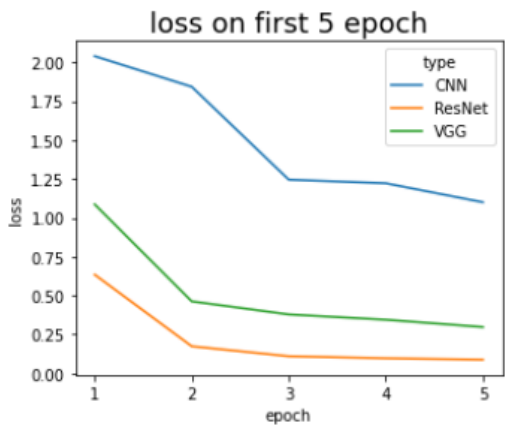
```
# Plot for first 5 epoch
plt.figure(figsize=(5,4))
plt.title('loss on first 5 epoch',fontsize=18)
sns.lineplot(data=df[df['epoch']<=5], x="epoch", y="loss", hue="type")
plt.xticks(list(range(1,5+1)))
plt.show()

plt.figure(figsize=(5,4))
plt.title('test_accuracy_score on first 5 epoch',fontsize=18)
sns.lineplot(data=df[df['epoch']<=5], x="epoch", y="test_accuracy_score", hue="type")
plt.xticks(list(range(1,5+1)))
plt.show()

plt.figure(figsize=(5,4))
plt.title('test_recall_score on first 5 epoch',fontsize=18)
sns.lineplot(data=df[df['epoch']<=5], x="epoch", y="test_recall_score", hue="type")
plt.xticks(list(range(1,5+1)))
plt.show()

plt.figure(figsize=(5,4))
plt.title('test_precision_score on first 5 epoch',fontsize=18)
sns.lineplot(data=df[df['epoch']<=5], x="epoch", y="test_precision_score", hue="type")
plt.xticks(list(range(1,5+1)))
plt.show()

plt.figure(figsize=(5,4))
plt.title('test_f1_score on first 5 epoch',fontsize=18)
sns.lineplot(data=df[df['epoch']<=5], x="epoch", y="test_f1_score", hue="type")
plt.xticks(list(range(1,5+1)))
plt.show()
```

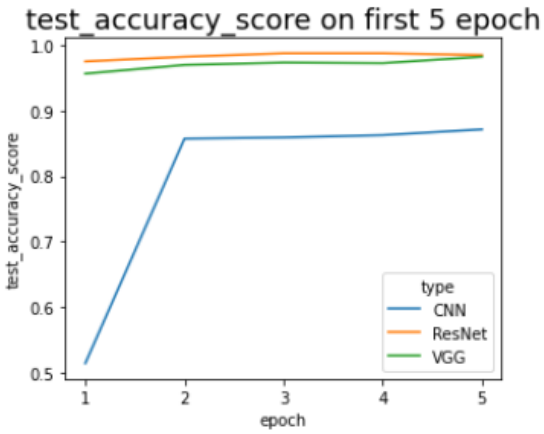




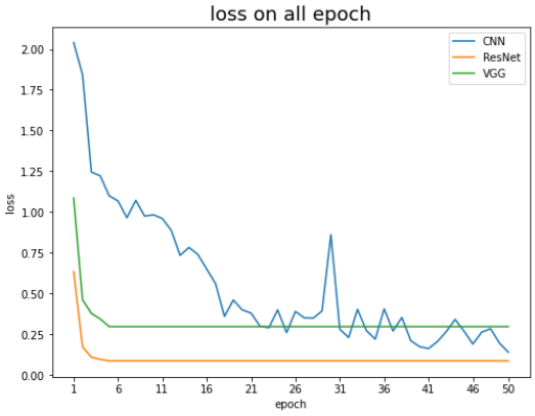


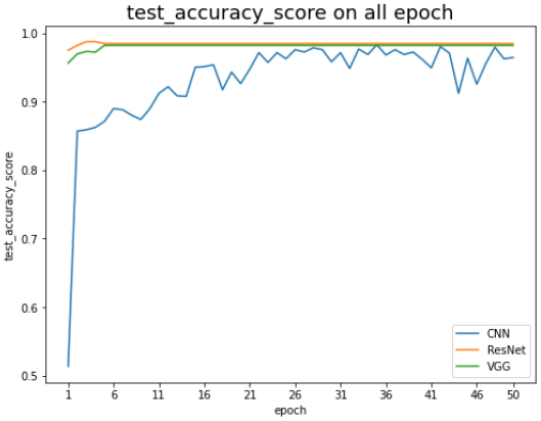
We noticed that the graphs look similar for all 4 evaluation matrices. In fact, this is all the case in our overall experiment. Therefore, we only focus on accuracy and prediction loss.





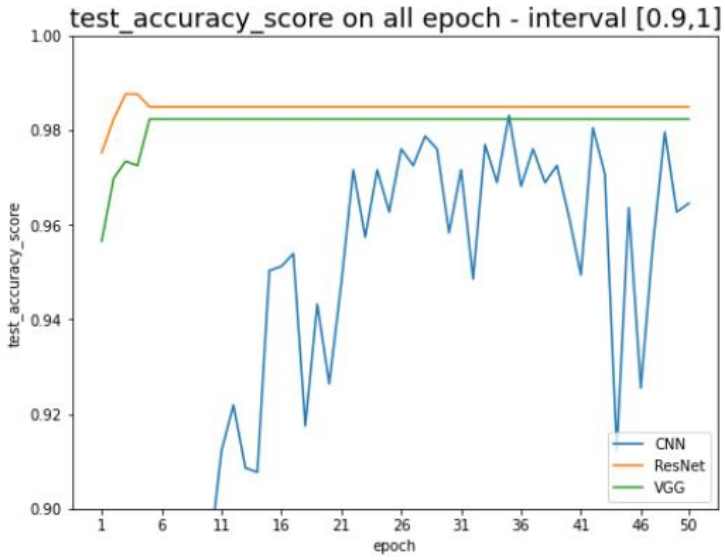
We notice that ResNet and VGG have a loss of less than 0.5, and both preserve a high accuracy, which is 98.49% and 98.23%. However, for Custom CNN the accuracy is around 89% only. We are interested in the limit that the Custom CNN model can achieve, therefore we train it for another 45 epochs.





We can see that both Custom CNN's loss and accuracy fluctuate heavily. We may also observe that Custom CNN is highly unstable and may not reach the same level of performance as the other two. Therefore, it is obvious to exclude this model from the model selection process.

Next, we take a closer look at the accuracy score at the interval [0.9, 1.0].



We can see that ResNet has a slightly higher accuracy than VGG. From the previous section, we also have the conclusion that VGG prediction time is significantly longer than ResNet ($\frac{60.94}{8.51} = 7.16$x slower). Another important factor in the model selection process is the model's training time. As VGG19 is a very large model with 138 million trainable parameters, while ResNet18 has only 11 million, training the VGG model is extremely resource-demanding. Therefore, we conclude that ResNet is our best model.

However, if we take a look at the graph closely, from epoch 3 to epoch 5, ResNet's loss decrease while its accuracy also decreases. This could be a sign of overfitting where the model tends to remember all the training data instead of learning the feature representation and implicit distribution of the population itself. Therefore, we conclude that our best model is ResNet epoch 3 with 98.76% of accuracy.

	type	epoch	loss	test_accuracy_score	test_recall_score	test_precision_score	test_f1_score	test_prediction_time_taken
2	ResNet	3	0.109.	0.9876	0.9883	0.9879	0.988	40.256614

5. Demo

Here is a demo of our trained model. We first create a fresh model of ResNet18, and a fully connected layer with 1000 features in and 4 features out. Then we load our weights with resnet_model.load_state_dict() method.

Select **resnet_model_00003_loss_0.109.pt** as best model

```
# Fresh model
resnet_model = models.resnet18(pretrained=False)
resnet_model = nn.Sequential(resnet_model, nn.Linear(in_features=1000, out_features=4))

# Load weights
resnet_model.load_state_dict(torch.load(r"../input/cisc3024db927491angjianhwee-trained-models/resnet_model_00003_loss_0.109.pt"))
```

Now we load input from DataLoader, predict it, and compare it with ground truth. We can see that our model predicts the first 20 images correctly.

```
# Load data to be predicted
X, y_true = list(enumerate(test_dataset_loader, 0))[0][1]
y_true = y_true.tolist()


y_pred = resnet_model(X)    # Predict
y_pred = torch.argmax(y_pred,axis = 1).tolist()    # Argmax and to list

# Print first 20
print(f"y_true :{y_true[0:20]}")
print(f"y_pred :{y_pred[0:20]}")
```

```
y_true :[1, 0, 2, 2, 0, 3, 0, 3, 3, 2, 0, 0, 0, 3, 2, 3, 0, 0, 0, 1]
y_pred :[1, 0, 2, 2, 0, 3, 0, 2, 3, 2, 0, 0, 0, 3, 2, 3, 0, 0, 0, 1]
```

The link to the entire notebook is
https://www.kaggle.com/code/megahwee/cisc3024-ang-jian-hwee-emily-choo-hue-che

The link of the trained model weights is

https://www.kaggle.com/datasets/megahwee/cisc3024db927491angjianhwee-trained-models

6. Conclusion and Future Work Suggestions

6.1 Conclusion

The first conclusion is that in the image classification task, our experiment proved that transfer learning on a pre-trained model has a better performance than training a model from scratch. Transfer learning from scratch can achieve much higher accuracy in a few epochs and is way more stable than a model from scratch. Some recent publications also support this theory.

The second conclusion is that our proposed model outperforms other baselines in different measures. Not only does it has higher accuracy, but it also converges quickly within 5 epochs and it is easy to train and deploy with weight filesize less than 20 MB.

The third conclusion is that solely increasing model size will not always increase model performance. Intuitively, a deeper neural network can better stimulate the complex relationship between variables. However, as the model gets larger, the marginal effect on the increase of accuracy will become smaller while the resources required to train such a model grows exponentially. Another potential risk is the gradient vanishing problem.

6.2 Future Work Suggestions

There are two possible improvements to our project. The first one is to increase the dataset size. The total dataset size including training and testing data is 5631, which is very small for a neural network. The second possible improvement is the imbalance problem. The class ‘desert’ has fewer samples compared with the other three classes, which could introduce bias in the experiment. We can deal with this problem using SMOTE or upsampling method.

7. Reference and Contribution

7.1 Reference

1. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition (Version 1). *arXiv.* https://doi.org/10.48550/ARXIV.1512.03385

2. Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition (Version 6). *arXiv.* https://doi.org/10.48550/ARXIV.1409.1556

3. O'Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks (Version 2). *arXiv.* https://doi.org/10.48550/ARXIV.1511.08458

4. https://pytorch.org/tutorials/

5. https://seaborn.pydata.org/

6. https://scikit-learn.org/

7.2 Contribution

Emily Choo Hue Che	Data Preprocessing, Custom CNN, and the corresponding part in PPT and report
Ang Jian Hwee	ResNet, VGG, Performance Analysis, and the corresponding part in PPT and report