CISC3024
Pattern Recognition Project

Title :

Pytorch Satellite image classification using neural networks

Members :

1) Wong Kai Yuan (DC026157)
2) Guan Jia Xi (CC029721)

Table of Content :

1) Introduction :

- Pytorch description : PyTorch is an open source machine learning library primarily used for Deep Learning applications, computer vision and natural language processing using GPUs and CPUs. It can be implemented in Python, mainly developed by the Facebook AI Research team. other Machine learning libraries similar to pytorch are TensorFlow and Keras. It makes use of tensors and can be implemented with numpy.

- Dataset description : Satellite image Dataset from Kaggle, contains four classes of satellite images which are: water, desert, cloudy and green area, with 5631 images in total and around 1500 images of each class.

- Neural Network model : We will be using Multilayer Perceptron (MLP) & DenseNet to classify our dataset.

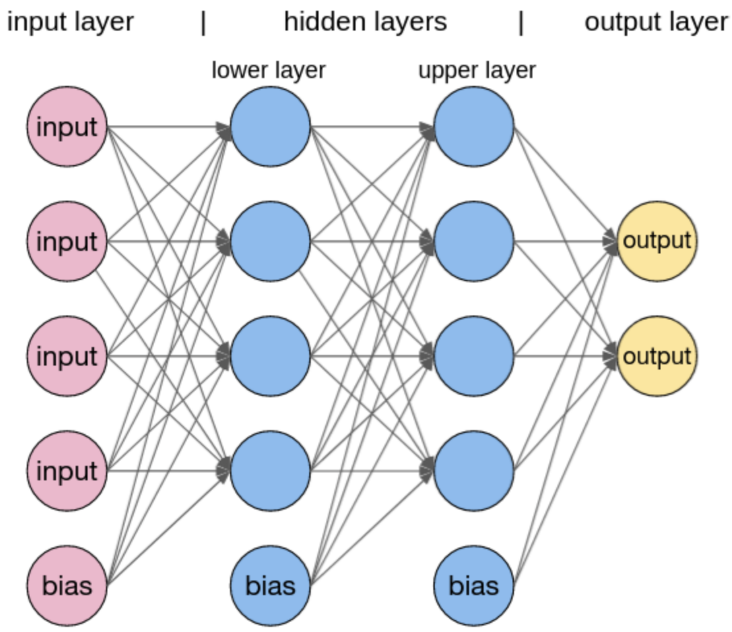
2) Goal :

To develop a deep learning or neural network model that can predict or classify satellite images into four classes: green, desert, cloudy and green area using pytorch. This model was also trained on a cpu enabled computer.

3) **Project Body :**

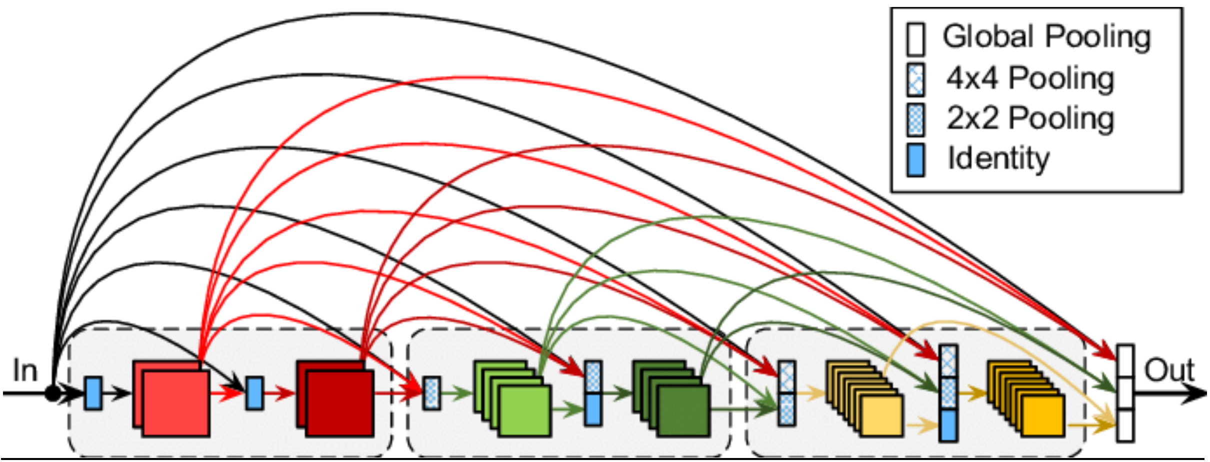
3.1 What is the Multilayer Perceptron (MLP) ?

A multilayer perceptron is a fully connected class of feedforward artificial neural network. The term MLP is used ambiguously, sometimes loosely to mean any feedforward ANN, sometimes strictly to refer to networks composed of multiple layers of perceptrons. It is one of the most basic neural network architectures.



3.2 What is Denesnet ?

A DenseNet is a type of convolutional neural network that utilises dense connections between layers, through the Dense Blocks, where we connect all layers(with matching feature-map sizes) directly with each other. To preserve the feed-forward nature, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers.



3.3 Implementation of MLP

3.3.1 Data Processing

- Import all the modules needed

```
import os
import torch
import torchvision
import tarfile
import torch.nn as nn
import numpy as np
import torch.nn.functional as F
from torchvision.datasets.utils import download_url
from tqdm.notebook import trange, tqdm
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torchvision.transforms as tt
from torch.utils.data import random_split
from torchvision.utils import make_grid
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

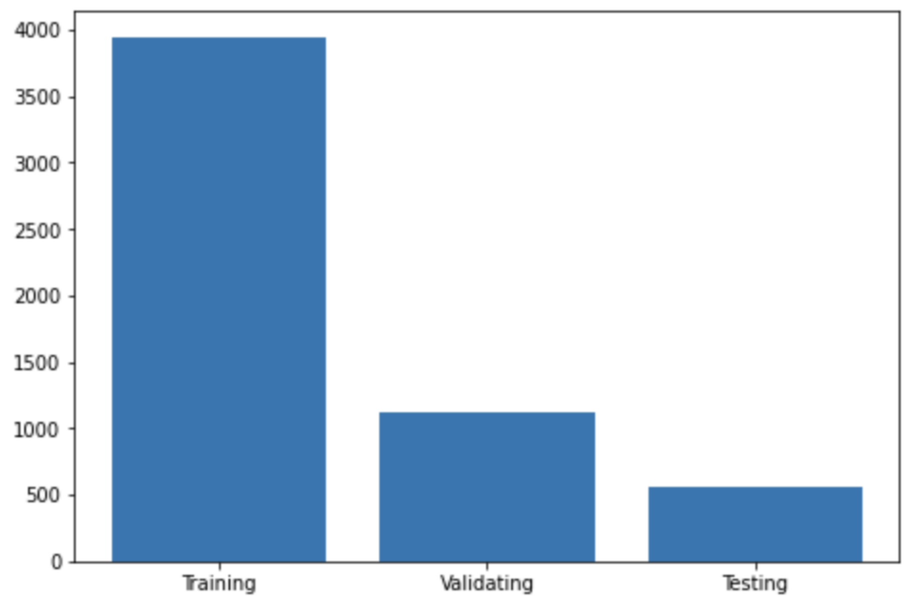
- Known stats : mean & standard decoration for 3 different channels in RGB of dataset

```
stats = ((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
```

- Define transform with known stats
- Load dataset with transform
- Split data into Training data, Validation data and Testing data in ratio of 7:2:1

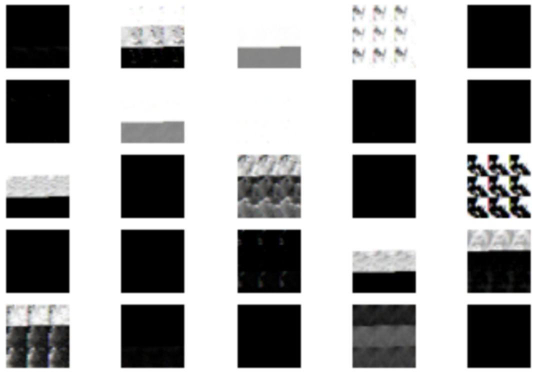
```
transform = transforms.Compose([    transforms.Resize(64),
                                transforms.ToTensor(),
                                    transforms.Normalize(*stats,inplace=True)])
dataset = datasets.ImageFolder('../input/satellite-image-classification/data', transform=transform)
train_data, val_data,test_data = torch.utils.data.random_split(dataset, [3950, 1120,561])
```

- Bar Chart of data after splitting



- Visualise & plot some images from training data

```
N_IMAGES = 25

images = [image for image, label in [train_data[i+200] for i in range(N_IMAGES)]]

plot_images(images)
```



- Define data loader or each dataset with batch size of 124

```
batch_size = 124
num_workers=0

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
valid_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)
```

3.3.2 Defining the model

- Our model will be a neural network, specifically a multilayer perceptron (MLP) with four hidden layers.
- Input dimension : 64*64*3
- Output dimension : 4 (total 4 class for classification)
- Hidden Layer 1 : 5000
- Hidden Layer 2 : 1500
- Hidden Layer 3 : 250
- Hidden Layer 4 : 50

```python
class MLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.input_fc = nn.Linear(input_dim, 5000)
        self.hidden_fc = nn.Linear(5000, 1500)
        self.hidden2_fc = nn.Linear(1500, 250)
        self.hidden3_fc = nn.Linear(250, 50)
        self.output_fc = nn.Linear(50, output_dim)

    def forward(self, x):

        # x = [batch size, height, width]

        batch_size = x.shape[0]

        x = x.view(batch_size, -1)

        h_1 = F.relu(self.input_fc(x))

        h_2 = F.relu(self.hidden_fc(h_1))

        h_3 = F.relu(self.hidden2_fc(h_2))

        h_4 = F.relu(self.hidden3_fc(h_3))

        y_pred = self.output_fc(h_4)


        return y_pred, h_4
```

```python
INPUT_DIM = 64 * 64 * 3
OUTPUT_DIM = 4

model = MLP(INPUT_DIM, OUTPUT_DIM)
```

- Create a small function to calculate the number of trainable parameters (weights and biases) in our model - in case all of our parameters are trainable.

```python
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 69,334,504 trainable parameters
```

3.3.3 Training the model

- Define Optimizer : We will be using Adam optimizer to update our parameters.
- Define Criterion : We will be using CrossEntropyLoss as loss function

```python
import torch.optim as optim
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()
```

- Define device : used to place model and data on to a GPU if having one
- Place model and criterion on the device using .to()

```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = criterion.to(device)
```

- Define a function to calculate the accuracy of our model

```python
def calculate_accuracy(y_pred, y):
#     print(y_pred, y)
#     print(y_pred.shape, y.shape)
    top_pred = y_pred.argmax(1, keepdim=True)

    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc
```

- Define Training Loop

Each Loop having :

- put our model into train mode
- iterate over our dataloader, returning batches of (image, label)
- place the batch on to our GPU, if we have one
- clear the gradients calculated from the last batch
- pass our batch of images, x, through to model to get predictions, y_pred
- calculate the loss between our predictions and the actual labels
- calculate the accuracy between our predictions and the actual labels
- calculate the gradients of each parameter
- update the parameters by taking an optimizer step
- update our metrics

```python
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in tqdm(iterator, desc="Training", leave=False):

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

- Define Evaluating Loop

The evaluation loop is similar to the training loop. The differences are:
- we put our model into evaluation mode with `model.eval()`
- we wrap the iterations inside a `with torch.no_grad()`
- we do not zero gradients as we are not calculating any
- we do not calculate gradients as we are not updating parameters

- we do not take an optimizer step as we are not calculating gradients

```
def evaluate(model, iterator, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for (x, y) in tqdm(iterator, desc="Evaluating", leave=False):

            x = x.to(device)
            y = y.to(device)

            y_pred, _ = model(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

3.3.4 Result of the training model

- Define epoch time function to tell us how each epoch took

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

- Results of the training with each epoch error & accuracy

```
Epoch: 01 | Epoch Time: 0m 52s
	Train Loss: 6.100 | Train Acc: 57.70%
	 Val. Loss: 1.149 |  Val. Acc: 70.97%

Epoch: 02 | Epoch Time: 0m 31s
	Train Loss: 0.871 | Train Acc: 74.67%
	 Val. Loss: 0.514 |  Val. Acc: 75.97%

Epoch: 03 | Epoch Time: 0m 30s
	Train Loss: 0.465 | Train Acc: 79.27%
	 Val. Loss: 0.536 |  Val. Acc: 76.29%

Epoch: 04 | Epoch Time: 0m 30s
	Train Loss: 0.455 | Train Acc: 80.31%
	 Val. Loss: 0.424 |  Val. Acc: 82.90%

Epoch: 05 | Epoch Time: 0m 30s
	Train Loss: 0.461 | Train Acc: 79.52%
	 Val. Loss: 0.453 |  Val. Acc: 77.34%

Epoch: 06 | Epoch Time: 0m 31s
	Train Loss: 0.420 | Train Acc: 81.74%
	 Val. Loss: 0.422 |  Val. Acc: 78.39%

Epoch: 07 | Epoch Time: 0m 30s
	Train Loss: 0.366 | Train Acc: 84.21%
	 Val. Loss: 0.377 |  Val. Acc: 83.55%

Epoch: 08 | Epoch Time: 0m 30s
	Train Loss: 0.386 | Train Acc: 83.31%
	 Val. Loss: 0.376 |  Val. Acc: 82.74%

Epoch: 09 | Epoch Time: 0m 30s
	Train Loss: 0.362 | Train Acc: 84.11%
	 Val. Loss: 0.367 |  Val. Acc: 84.03%

Epoch: 10 | Epoch Time: 0m 30s
	Train Loss: 0.337 | Train Acc: 84.94%
	 Val. Loss: 0.350 |  Val. Acc: 85.16%
```

3.3.5 Final Result on the testing dataset

- load our the parameters of the model that achieved the best validation loss and then use this to evaluate our model on the test set.

```
model.load_state_dict(torch.load('tut1-model.pt'))

test_loss, test_acc = evaluate(model, test_loader, criterion, device)
```

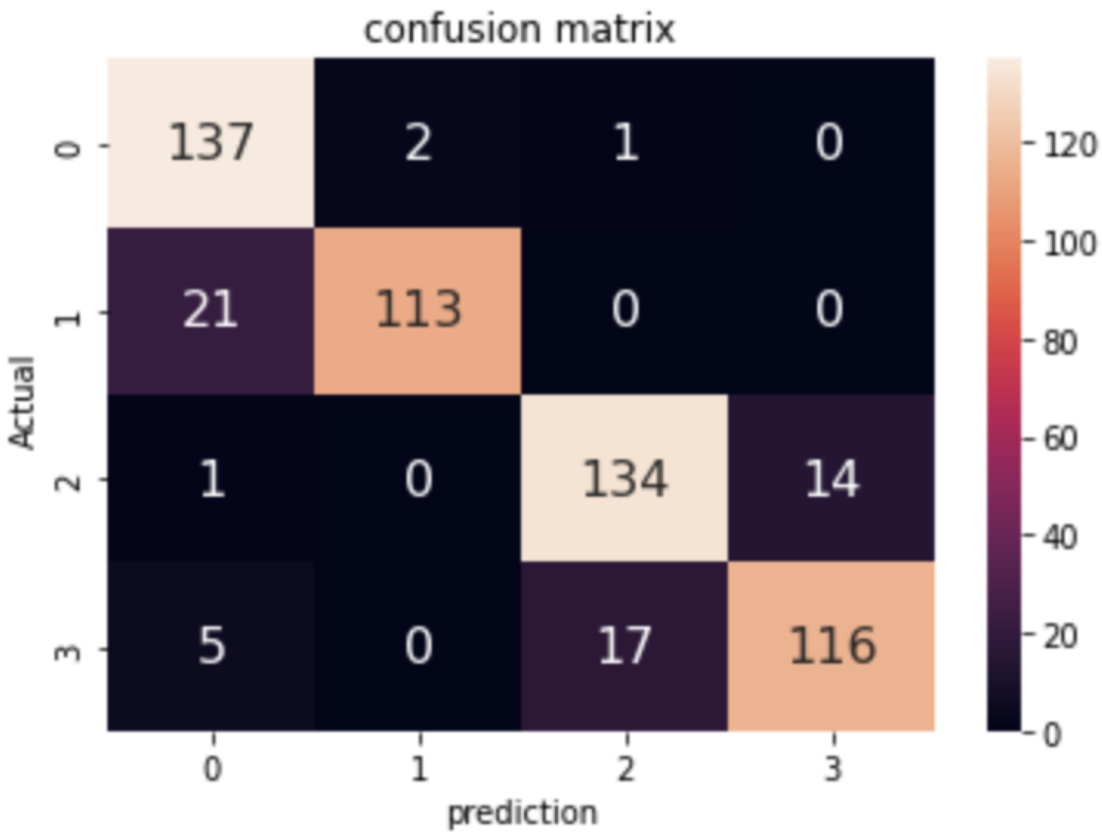
```
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.265 | Test Acc: 88.26%
```

3.3.6 Others

- Plot **confusion matrix** of our model

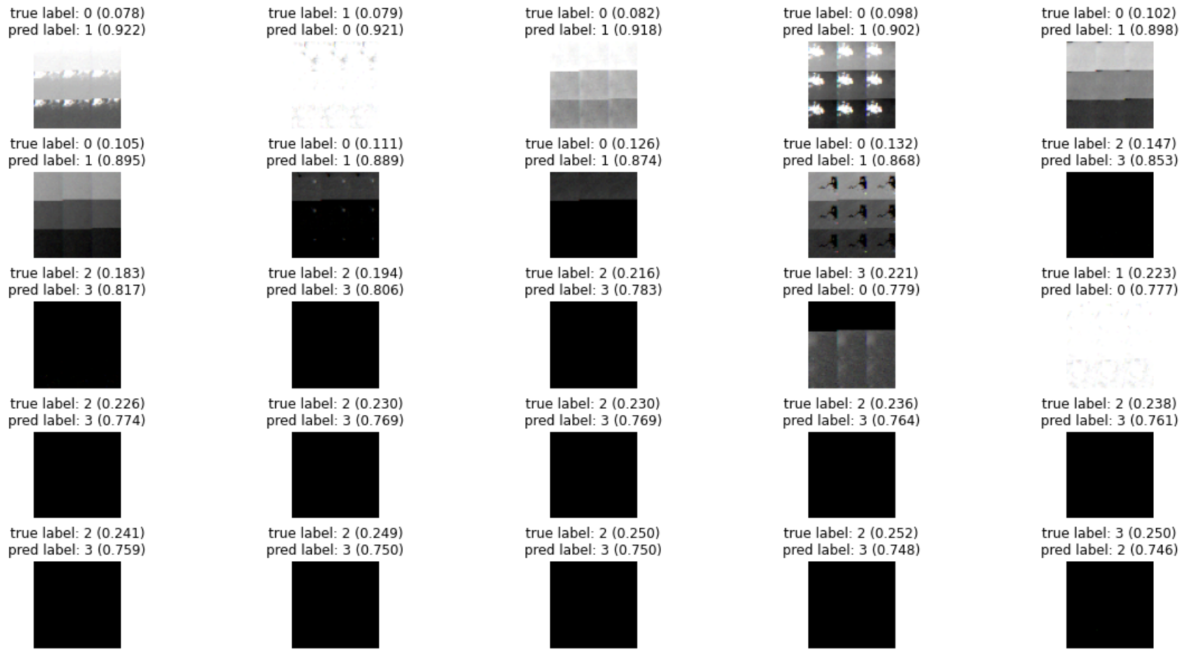
```
from sklearn.metrics import confusion_matrix
import pandas as pd
import seaborn
cm  = confusion_matrix(pred_labels,labels)
df_cm = pd.DataFrame(cm, index = [i for i in range(4)],
                  columns = [i for i in range(4)])
seaborn .heatmap(df_cm, annot=True, annot_kws={"size": 16}, fmt='d')
plt.title('confusion matrix')
plt.xlabel('prediction')
plt.ylabel('Actual');
```



- The results seem reasonable enough, the most confused predictions-actuals are: 0 -1
- Then, We can then plot the incorrectly predicted images along with how confident they were on the actual label and how confident they were at the incorrect label.

```
def plot_most_incorrect(incorrect, n_images):

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize=(20, 10))
    for i in range(rows*cols):
        ax = fig.add_subplot(rows, cols, i+1)
        image, true_label, probs = incorrect[i]
        true_prob = probs[true_label]
        incorrect_prob, incorrect_label = torch.max(probs, dim=0)
        ax.imshow(image.view(64, 64,3).cpu().numpy(), cmap='bone')
        ax.set_title(f'true label: {true_label} ({true_prob:.3f})\n'
                     f'pred label: {incorrect_label} ({incorrect_prob:.3f})')
        ax.axis('off')
    fig.subplots_adjust(hspace=0.5)
```

```
N_IMAGES = 25

plot_most_incorrect(incorrect_examples, N_IMAGES)
```



3.4 Implementation of DenseNet

3.4.1 Data Processing

- Import al the library needed

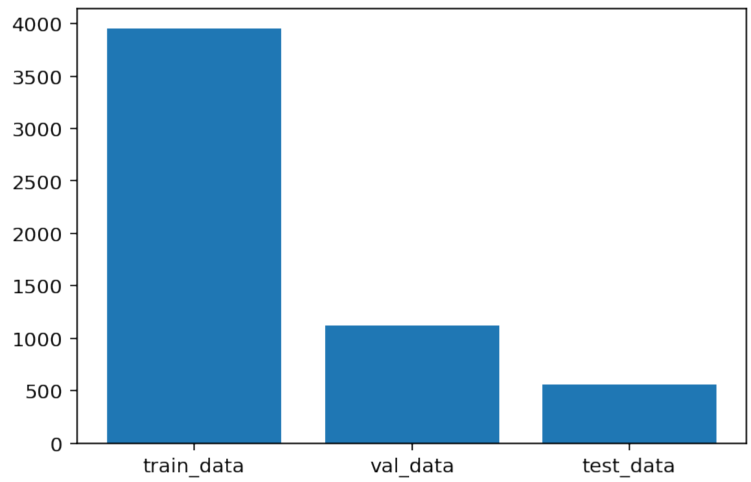
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F
import time
from torchvision import datasets, transforms, models
import torchvision.transforms as tt
from tqdm.notebook import trange, tqdm
from torchvision import datasets, transforms, models
from sklearn.metrics import confusion_matrix
import seaborn
```

- Define transform with known stats
- Load dataset with transform
- Split data into Training data, Validation data and Testing data in ratio of 7:2:1
- Define dataset and data loader with batch size of 64

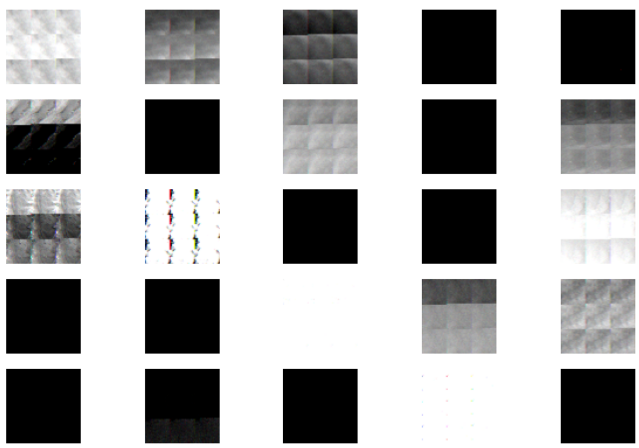
```
transform = transforms.Compose([   transforms.Resize(64),
                                transforms.ToTensor(),
                                   transforms.Normalize(*stats,inplace=True)])

dataset = datasets.ImageFolder('/kaggle/input/satellite-image-classification/data', transform = transform)

train_data, val_data, test_data = torch.utils.data.random_split(dataset,[3950,1120,561])

print("Split data to training, val and test in 7:2:1 ")
print('Num training images: ', len(train_data))
print('Num validating images: ', len(val_data))
print('Num test images: ', len(test_data))
trainloader = torch.utils.data.DataLoader(train_data,batch_size = 64, shuffle = True)
testloader = torch.utils.data.DataLoader(test_data,batch_size = 64)
Valloader = torch.utils.data.DataLoader(val_data, batch_size = 64)
```

```
Split data to training, val and test in 7:2:1
Num training images:  3950
Num validating images:  1120
Num test images:  561
```

- Bar chart of data after splitting



- Visualise & plot some images from training data



3.4.2 Defining the model

- Model will be a neural network, DenseNet-121. 121 means the depth of each layer in Dense Block.
- Using the pytorch.nn module define the neural network using relu activation functions, and softmax. Note there is no need through the descent weight.

```
#print out the model architecture
model = models.densenet121(pretrained=True)
model
```

```
# Use GPU if it's available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = models.densenet121(pretrained=True)

# Freeze parameters so we don't backprop through them
for param in model.parameters():
    param.requires_grad = False

#DEFINE NEURAL NETWORK

model.classifier = nn.Sequential(nn.Linear(1024, 512),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(512, 256),
                                 nn.ReLU(),
                                 nn.Dropout(0.1),
                                 nn.Linear(256, 4),
                                 nn.LogSoftmax(dim=1))

criterion = nn.NLLLoss()

# Only train the classifier parameters, feature parameters are frozen
optimizer = optim.Adam(model.classifier.parameters(), lr=0.003)

model.to(device);
```

3.4.3 Training the model

- Define the epochs, and inputs to device, then backward propagate and apply optimizer.
- calculate the loss between our predictions and the actual labels
- calculate the accuracy between our predictions and the actual labels

```
traininglosses = []
testinglosses = []
testaccuracy = []
totalsteps = []
epochs = 1
steps = 0
running_loss = 0
print_every = 5
for epoch in range(epochs):
    for inputs, labels in trainloader:
        steps += 1
        # Move input and label tensors to the default device
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
```

```
        #backward propagation and optimizer step to update the weights
        logps = model.forward(inputs)
        loss = criterion(logps, labels)
        loss.backward
        optimizer.step()


        running_loss += loss.item()

        if steps % print_every == 0:
            test_loss = 0
            accuracy = 0
            model.eval()
            with torch.no_grad():
                for inputs, labels in Valloader:
                    inputs, labels = inputs.to(device), labels.to(device)
                    logps = model.forward(inputs)
                    batch_loss = criterion(logps, labels)

                    test_loss += batch_loss.item()

                    # Calculate accuracy
                    ps = torch.exp(logps)
                    top_p, top_class = ps.topk(1, dim=1)
                    equals = top_class == labels.view(*top_class.shape)
                    accuracy += torch.mean(equals.type(torch.FloatTensor)).item()

            traininglosses.append(running_loss/print_every)
            testinglosses.append(test_loss/len(Valloader))
            testaccuracy.append(accuracy/len(Valloader))
            totalsteps.append(steps)
            print(f"Device {device}.."
                  f"Epoch {epoch+1}/{epochs}.. "
                  f"Step {steps}.. "
                  f"Train loss: {running_loss/print_every:.3f}.. "
                  f"Test loss: {test_loss/len(Valloader):.3f}.. "
                  f"Test accuracy: {accuracy/len(Valloader):.3f}")
            running_loss = 0
            model.train()
```

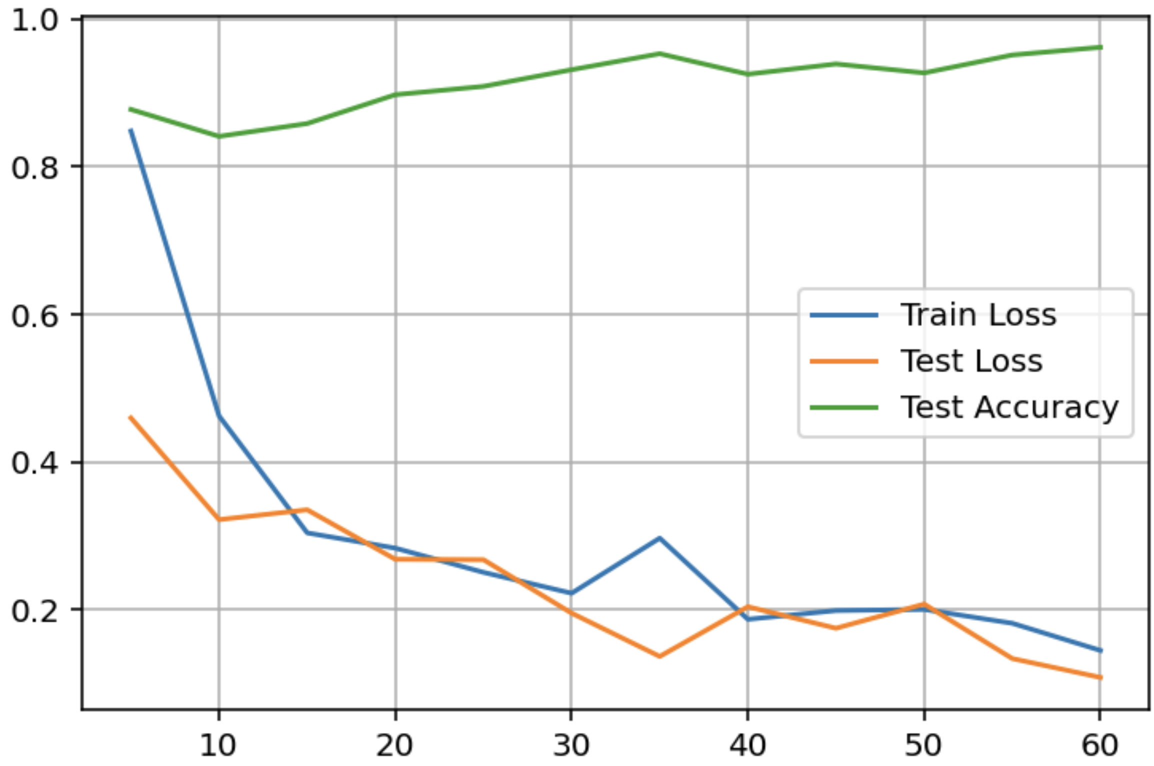
3.4.4 Result of the model

- Output of data training and evaluation

```
Device cuda..Epoch 1/1.. Step 5.. Train loss: 1.314.. Test loss: 1.037.. Test accuracy: 0.616
Device cuda..Epoch 1/1.. Step 10.. Train loss: 0.614.. Test loss: 1.092.. Test accuracy: 0.671
Device cuda..Epoch 1/1.. Step 15.. Train loss: 0.422.. Test loss: 0.450.. Test accuracy: 0.839
Device cuda..Epoch 1/1.. Step 20.. Train loss: 0.436.. Test loss: 0.378.. Test accuracy: 0.853
Device cuda..Epoch 1/1.. Step 25.. Train loss: 0.335.. Test loss: 0.158.. Test accuracy: 0.946
Device cuda..Epoch 1/1.. Step 30.. Train loss: 0.257.. Test loss: 0.190.. Test accuracy: 0.939
Device cuda..Epoch 1/1.. Step 35.. Train loss: 0.235.. Test loss: 0.158.. Test accuracy: 0.941
Device cuda..Epoch 1/1.. Step 40.. Train loss: 0.218.. Test loss: 0.104.. Test accuracy: 0.966
Device cuda..Epoch 1/1.. Step 45.. Train loss: 0.259.. Test loss: 0.127.. Test accuracy: 0.957
Device cuda..Epoch 1/1.. Step 50.. Train loss: 0.190.. Test loss: 0.132.. Test accuracy: 0.955
Device cuda..Epoch 1/1.. Step 55.. Train loss: 0.226.. Test loss: 0.094.. Test accuracy: 0.971
Device cuda..Epoch 1/1.. Step 60.. Train loss: 0.218.. Test loss: 0.123.. Test accuracy: 0.962
```

```python
print(f'Test Loss: {test_loss/len(Valloader):.3f} | Test accuracy: {accuracy/len(Valloader):.3f}%' )
```

```
Test Loss: 0.123 | Test accuracy: 0.962%
```



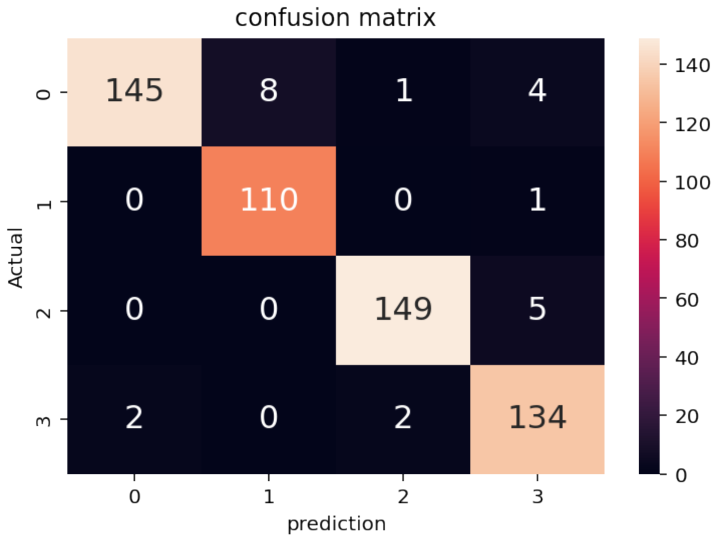
3.4.5 Final Result on Testing

- Load the test_data into the trained model and calculate the accuracy rate, turn the accuracy rate into a confusion matrix.

```python
def get_predictions(model, iterator, device):

    model.eval()

    images = []
    labels = []
    probs = []

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)

            y_pred = model(x)

            y_prob = F.softmax(y_pred, dim=-1)

            images.append(x.cpu())
            labels.append(y.cpu())
            probs.append(y_prob.cpu())

    images = torch.cat(images, dim=0)
    labels = torch.cat(labels, dim=0)
    probs = torch.cat(probs, dim=0)

    return images, labels, probs
```

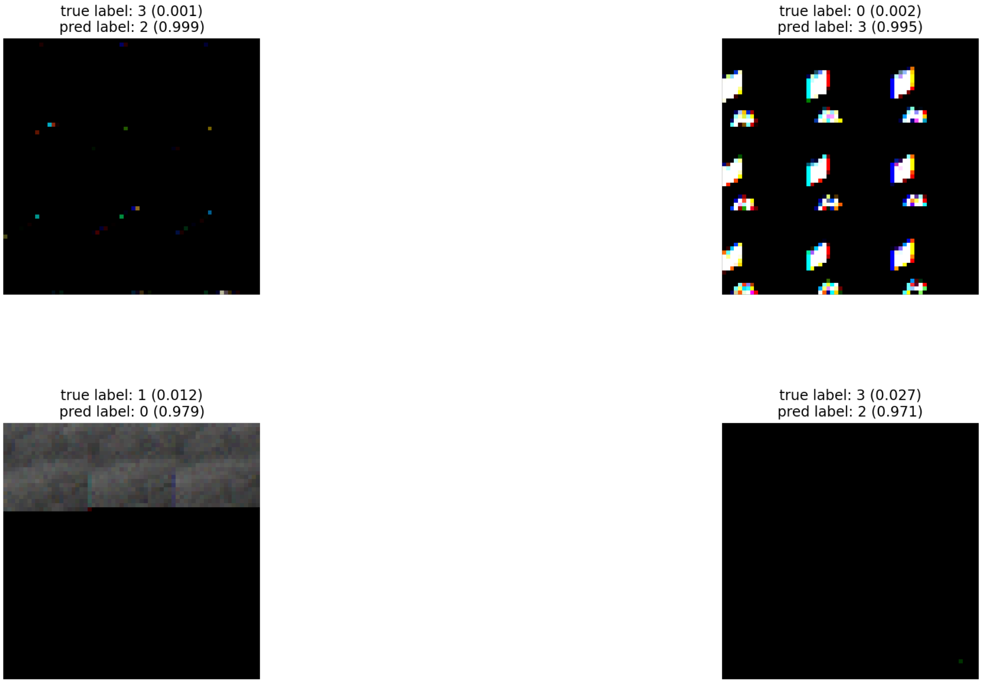
```python
images, labels, probs = get_predictions(model, test_loader, device)

pred_labels = torch.argmax(probs, 1)
```

```python
cm  = confusion_matrix(pred_labels,labels)
df_cm = pd.DataFrame(cm, index = [i for i in range(4)],
                  columns = [i for i in range(4)])
seaborn .heatmap(df_cm, annot=True, annot_kws={"size": 16}, fmt='d')
plt.title('confusion matrix')
plt.xlabel('prediction')
plt.ylabel('Actual');
```



3.3.6 Others

- Show incorrect images



4) <u>Conclusion</u>

<u>The testing results on MLP is :</u>
Error : 0.265
Accuracy : 88.26%
<u>The testing results on DenseNet is :</u>
Error : 0.123
Accuracy : 96.2%

This project explained the process of predicting a satellite image class with the pytorch library, with a comparison of MLP and DenseNet.

5) <u>contribution</u>

1. Contribution of Wong Kai Yuan :
- Implementation of Multilayer Perceptron (MLP) model on dataset
- 50% of report writing
- 50% of Presentation PPT

2. Contribution of Guan Jiaxi:
- Implementation of DenseNet model on dataset
- 50% of report writing
- 50% of Presentation PPT

Each member first does implementation on their own model, then integrates together during the writing report & PPT stage.