**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**
2. **Goal**
3. **Project Body**

**3.1 What is MLP?**

**3.2 What is DenseNet?**

**3.3 Implementation on MLP**

**3.3.1 Data Processing**

**3.3.2 Defining the model**

**3.3.3 Training the model**

**3.3.4 Result of the model**

**3.3.5 Final Result on Testing**

**3.3.6 Others**

**3.4 Implementation on DenseNet**

**3.4.1 Data Processing**

**3.4.2 Defining the model**

**3.4.3 Training the model**

**3.4.4 Result of the model**

**3.4.5 Final Result on Testing**

**3.4.6 Others**

1. **Conclusion**
2. **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.

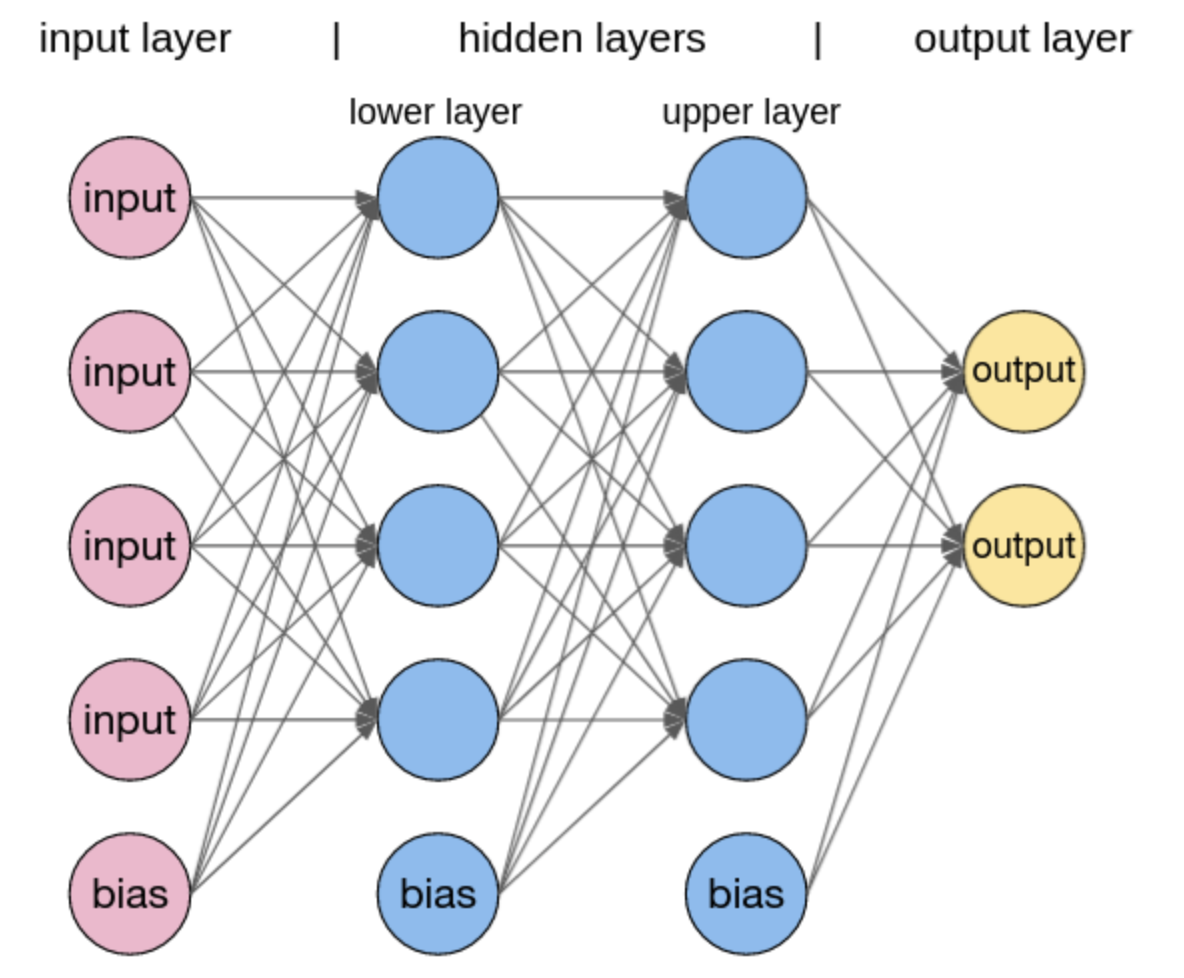
1. **Goal** :

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.

1. **Project Body :**

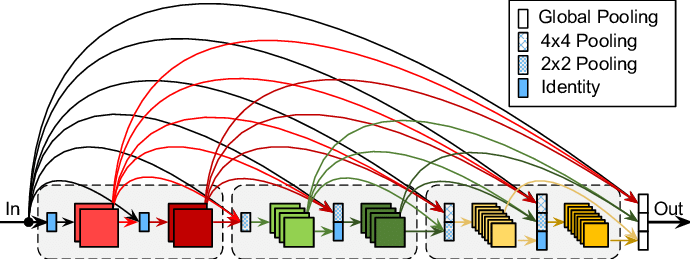
3.1 What is the Multilayer Perceptron (MLP) ?

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 ?

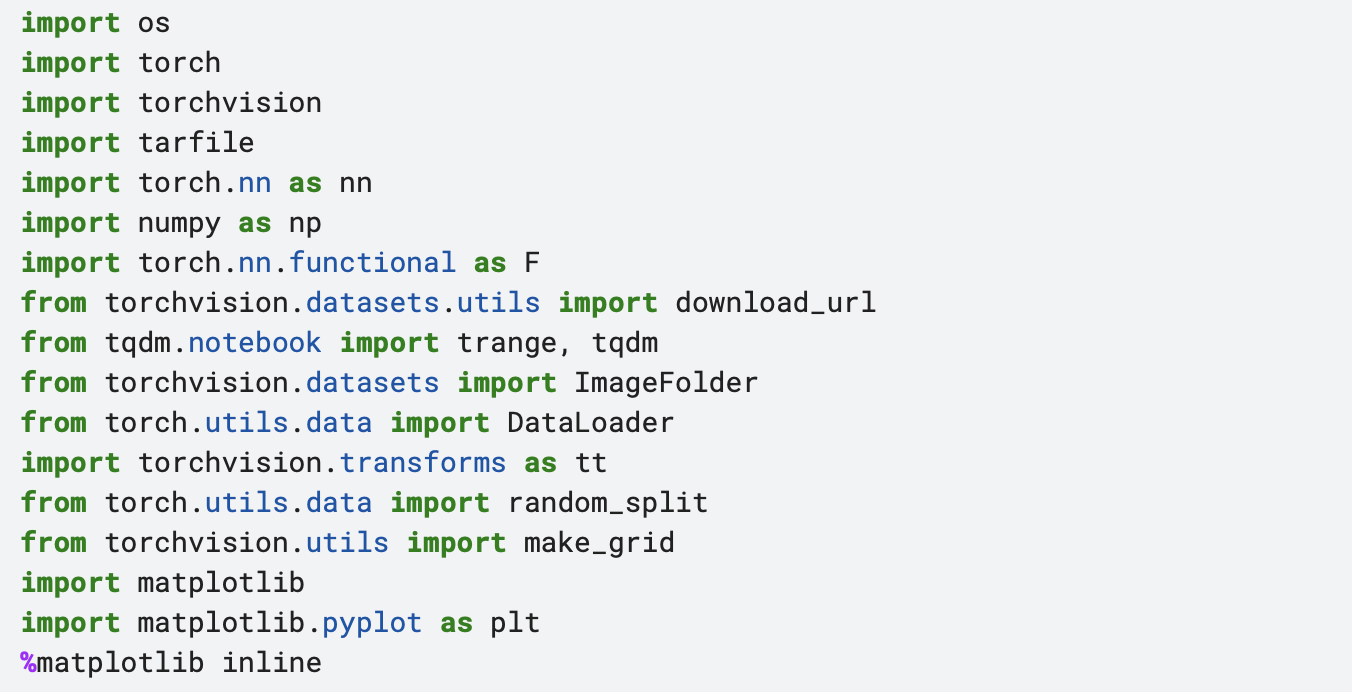
A DenseNet is a type of convolutional neural network that utilises dense connections between layers, through the Dense Blocks, where we connect all layers(with matching feature-map sizes) directly with each other. To preserve the feed-forward nature, each layer obtains additional inputs from all preceding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



**3.3 Implementation of MLP**

3.3.1 Data Processing

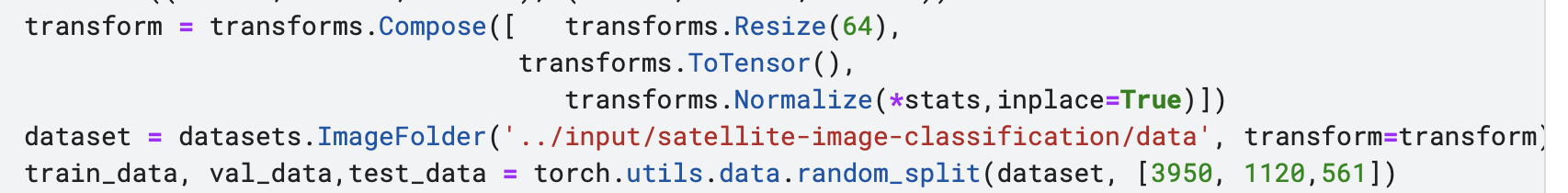
* Import all the modules needed



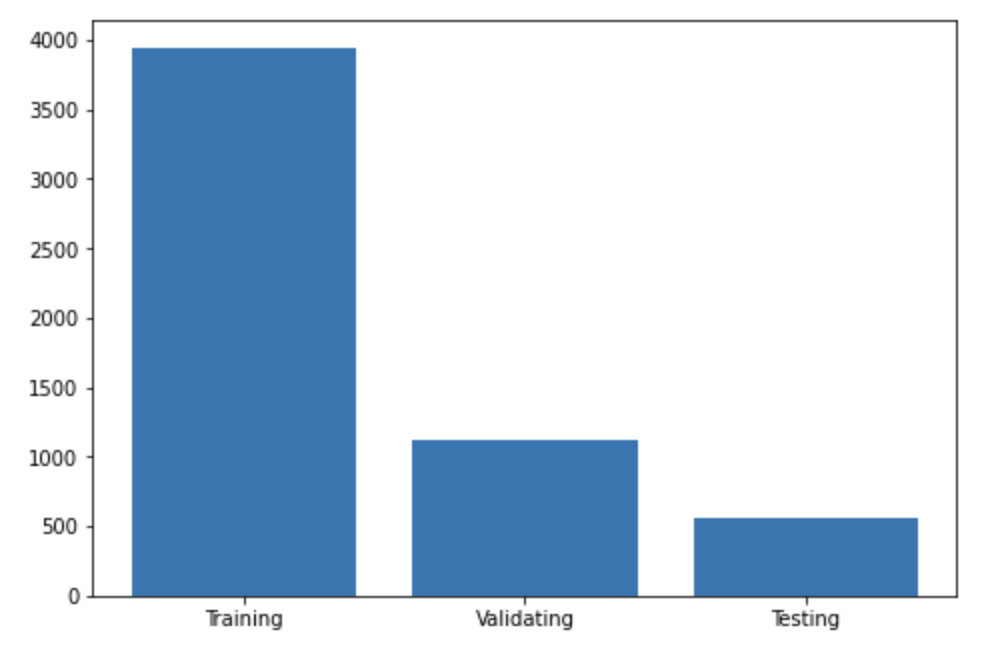
* Known stats : mean & standard decoration for 3 different channels in RGB of dataset



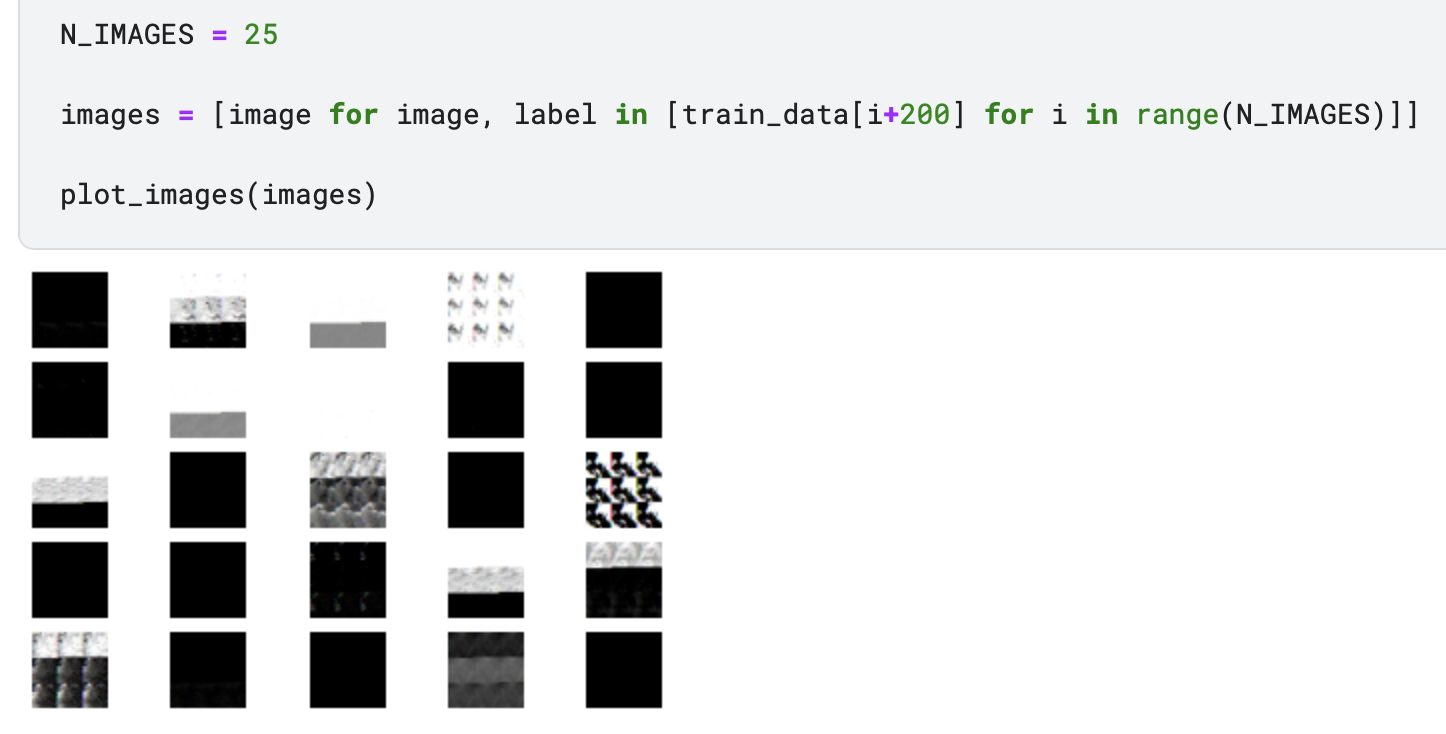
* Define transform with known stats
* Load dataset with transform
* Split data into Training data, Validation data and Testing data in ratio of 7:2:1



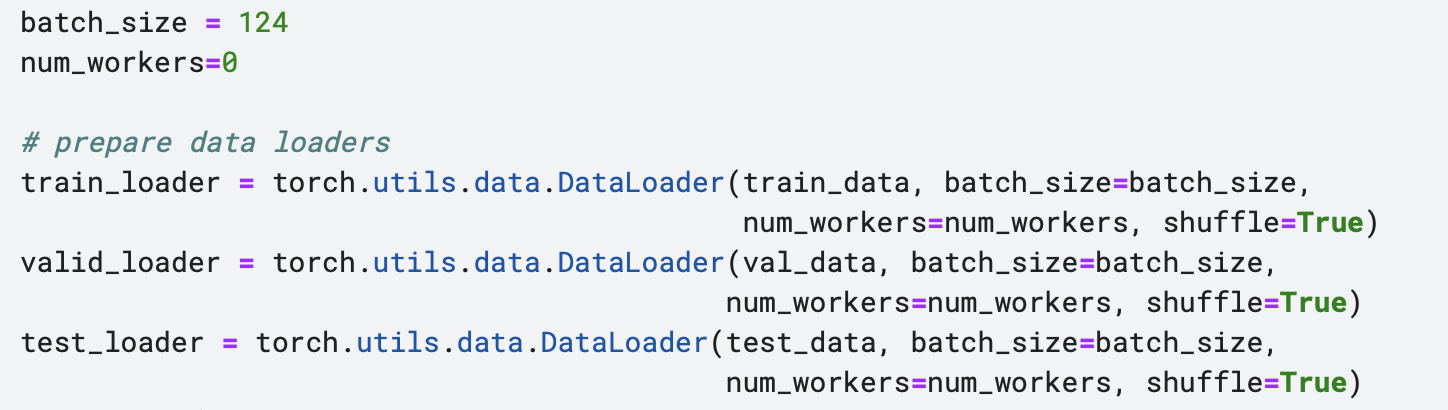
* Bar Chart of data after splitting



* Visualise & plot some images from training data

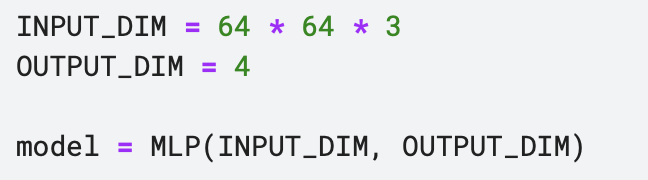


* Define data loader or each dataset with batch size of 124

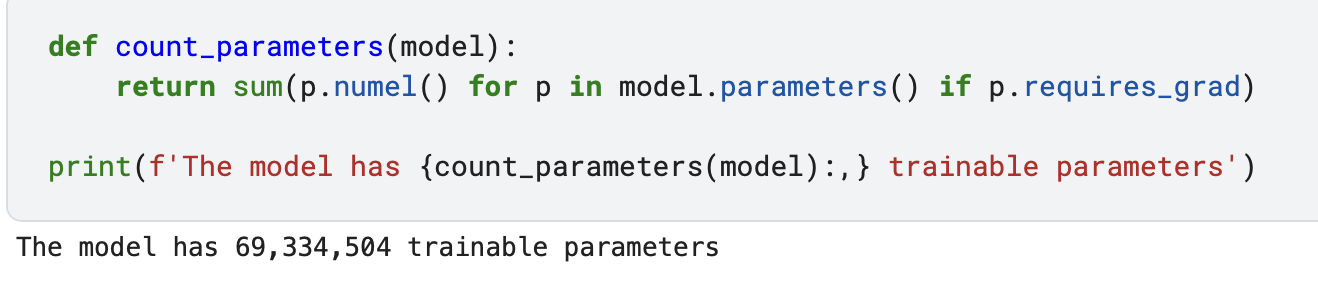


3.3.2 Defining the model

* 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

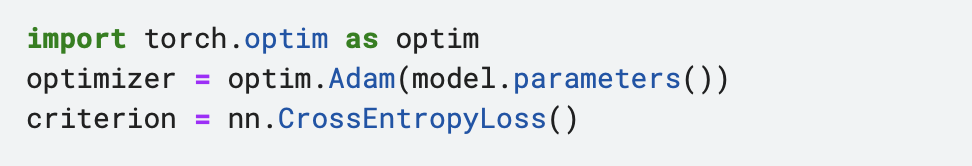


* Create a small function to calculate the number of trainable parameters (weights and biases) in our model - in case all of our parameters are trainable.



3.3.3 Training the model

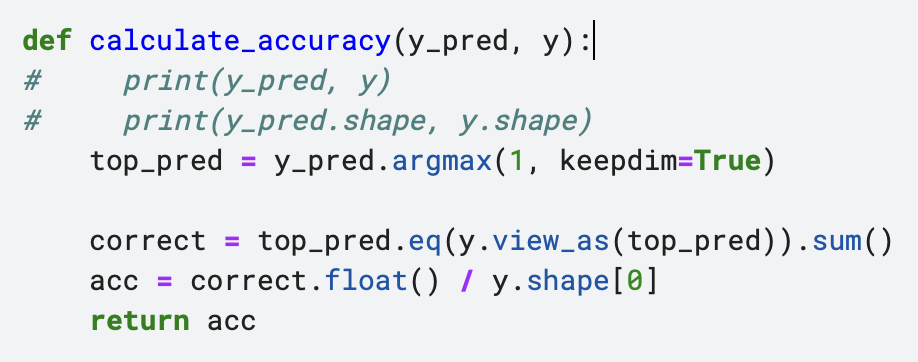
* Define Optimizer : We will be using Adam optimizer to update our parameters.
* Define Criterion : We will be using CrossEntropyLoss as loss function



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



* Define a function to calculate the accuracy of our model



* 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



* Define Evaluating Loop

The evaluation loop is similar to the training loop. The differences are:

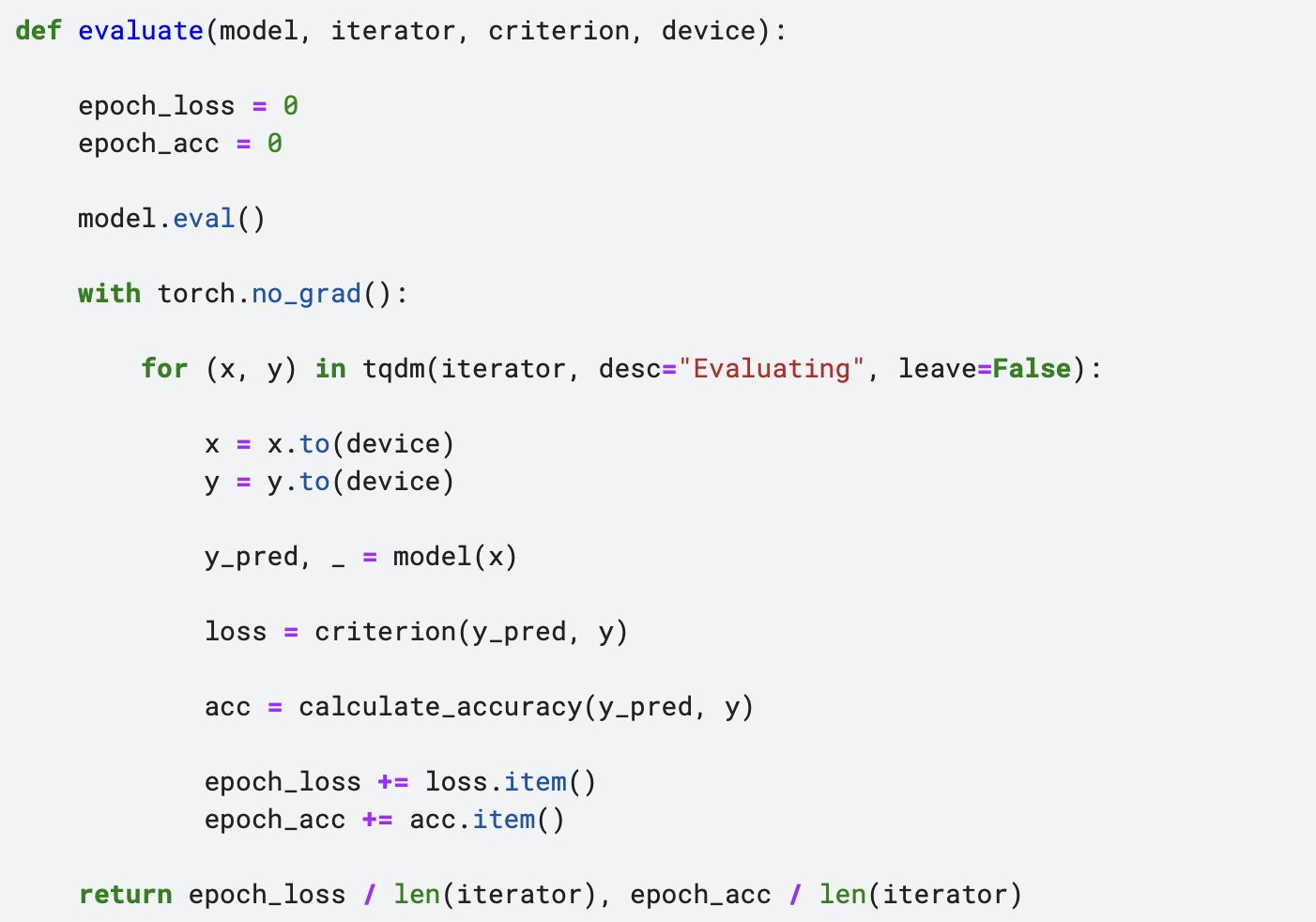
- we put our model into evaluation mode with `model.eval()`

- we wrap the iterations inside a `with torch.no\_grad()`

- we do not zero gradients as we are not calculating any

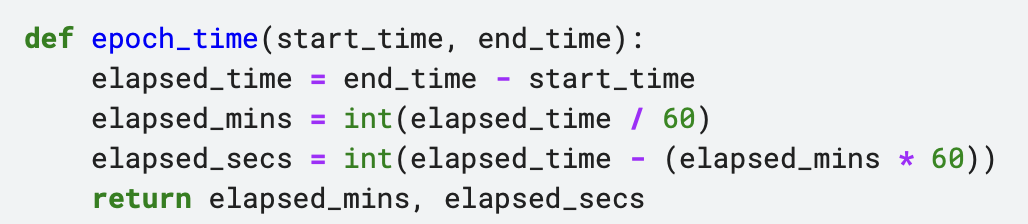
- we do not calculate gradients as we are not updating parameters

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

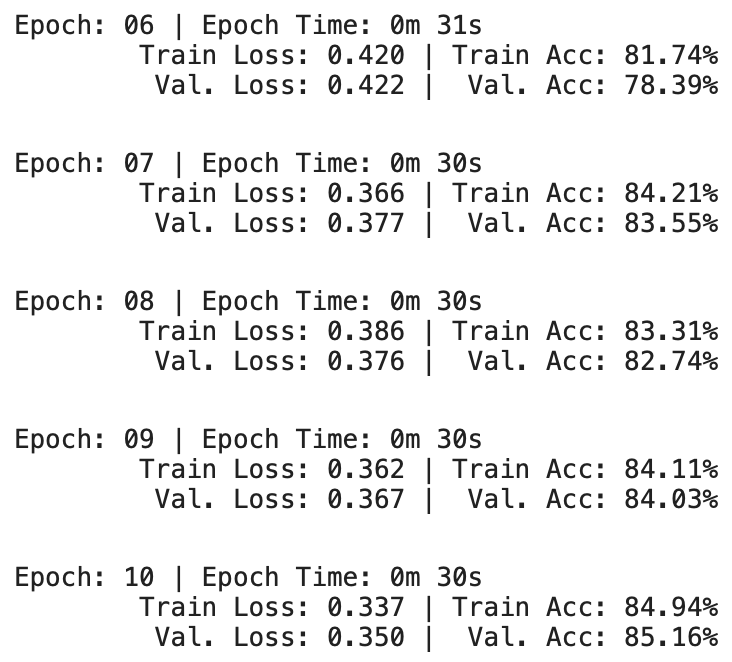
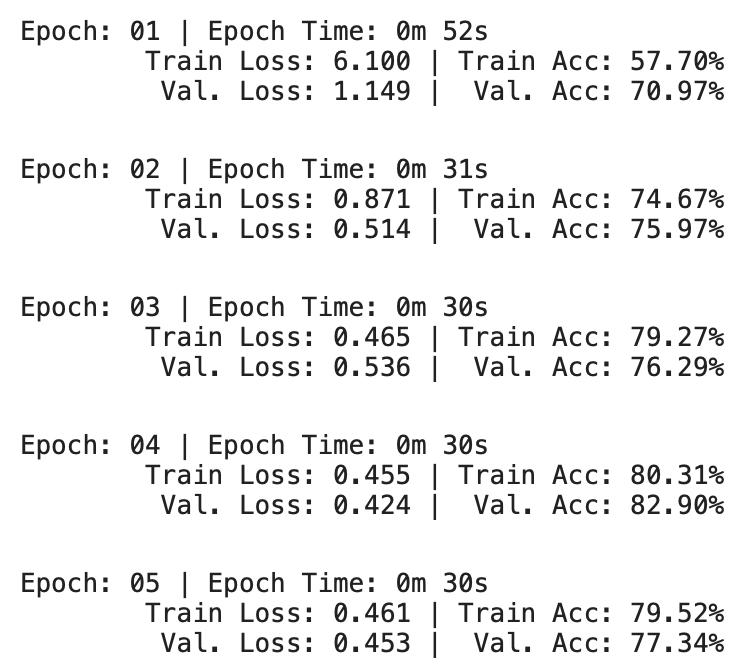


3.3.4 Result of the training model

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



* Results of the training with each epoch error & accuracy



3.3.5 Final Result on the testing dataset

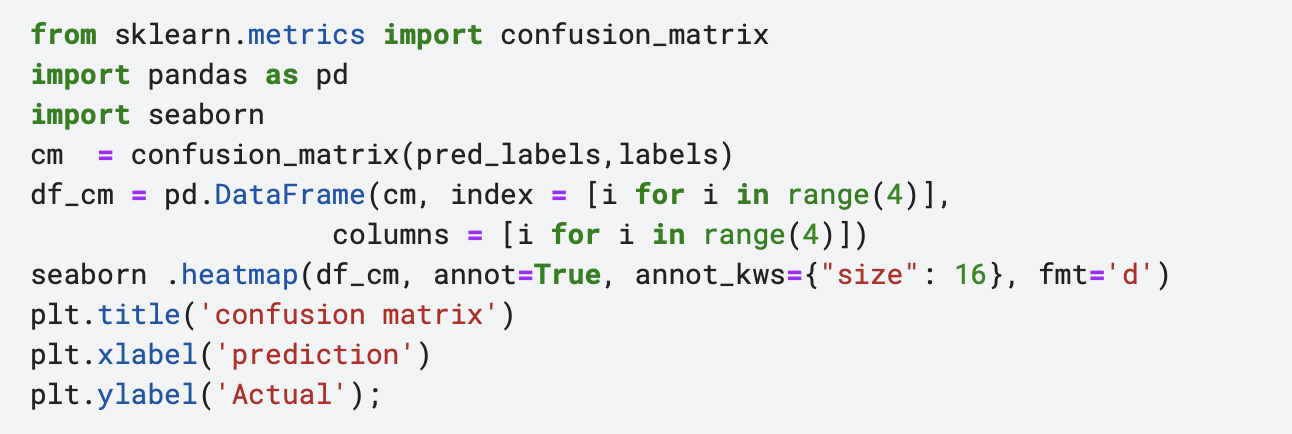
* load our the parameters of the model that achieved the best validation loss and then use this to evaluate our model on the test set.

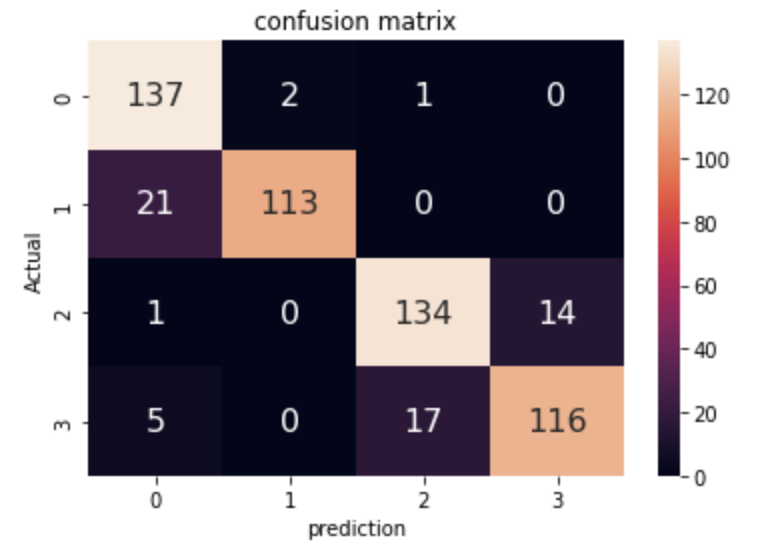




3.3.6 Others

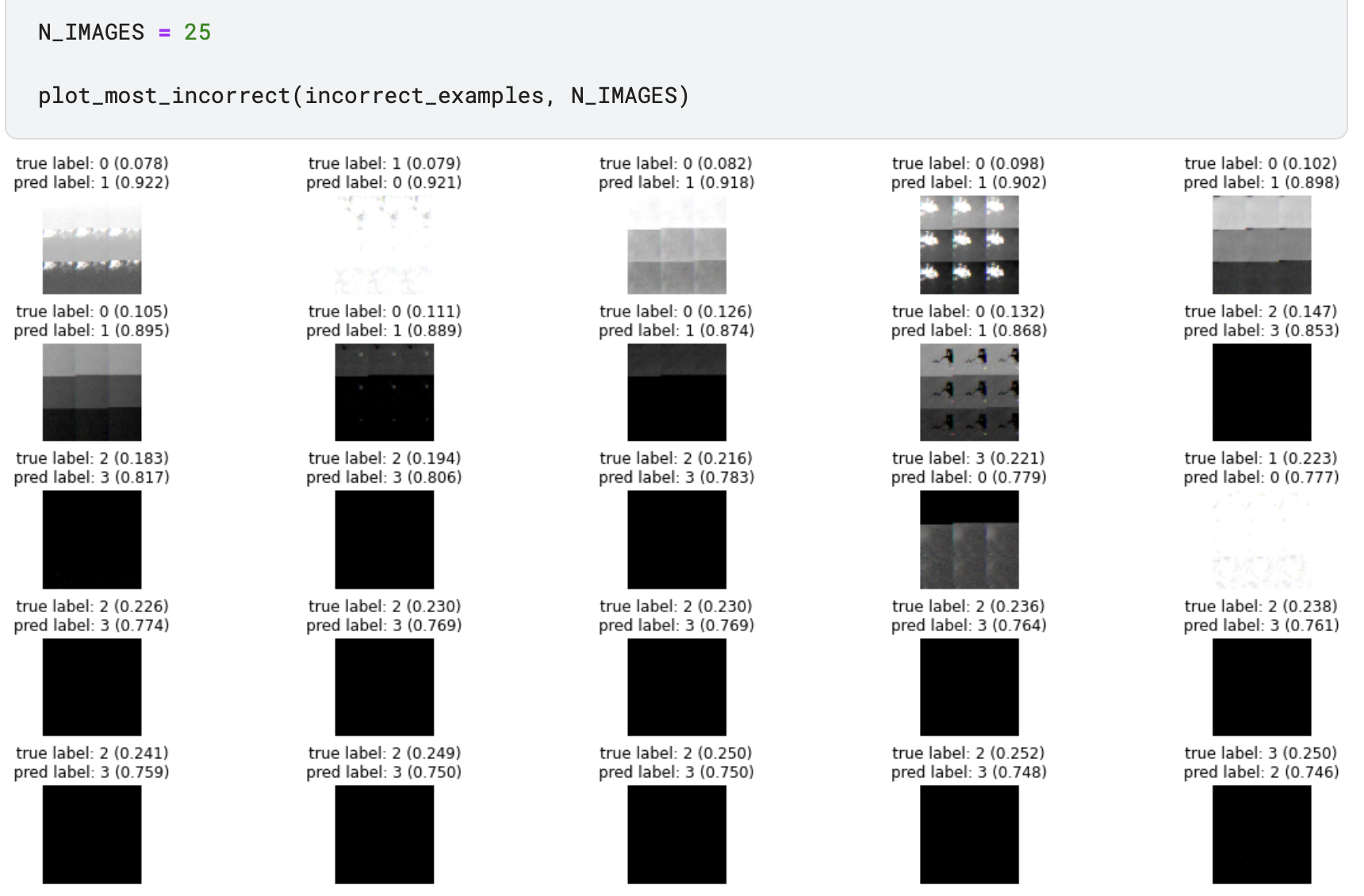
* Plot **confusion matrix** of our model





* 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.



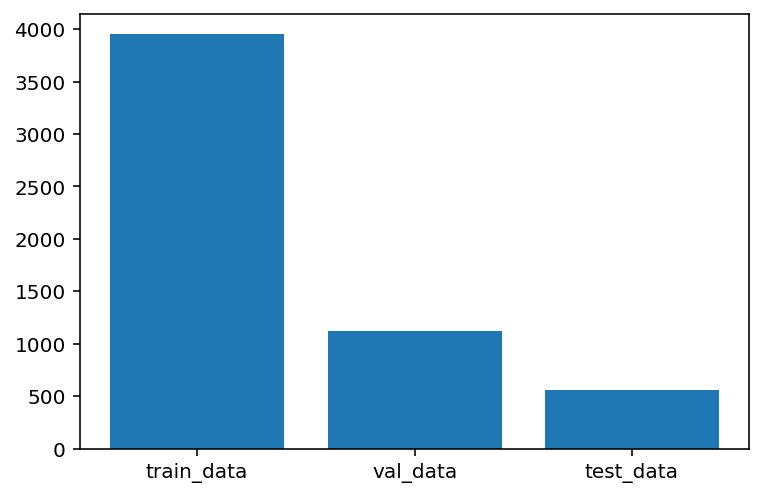


**3.4 Implementation of DenseNet**

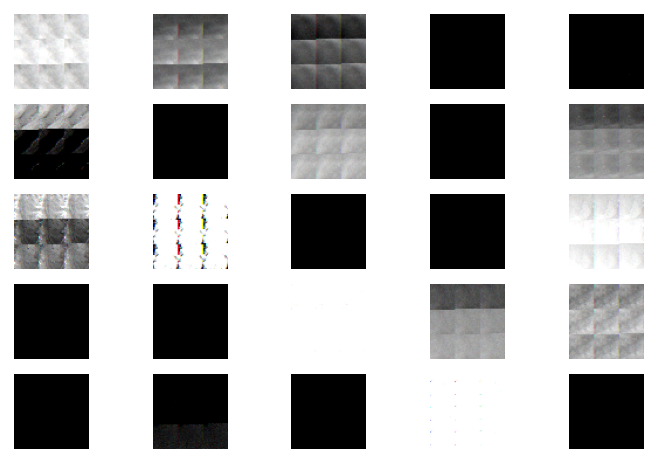
3.4.1 Data Processing

* Import al the library needed

|  |
| --- |
| * 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      * 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.





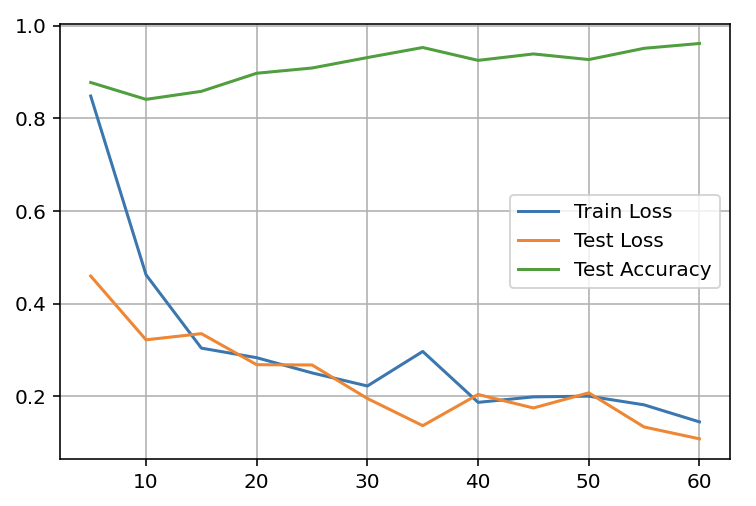
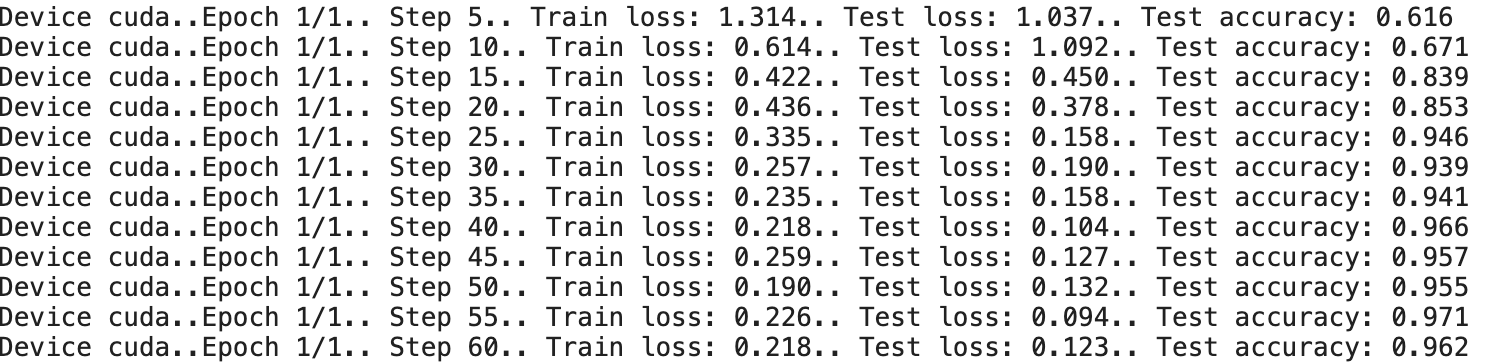
3.4.3 Training the model

* 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() |

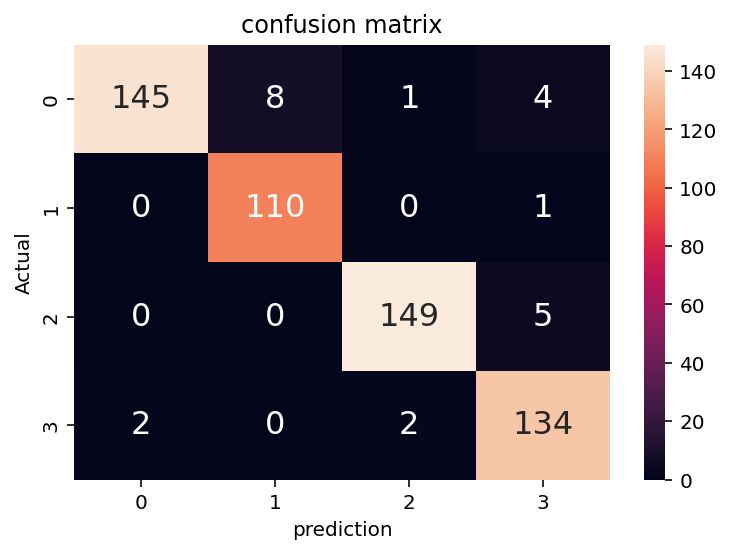
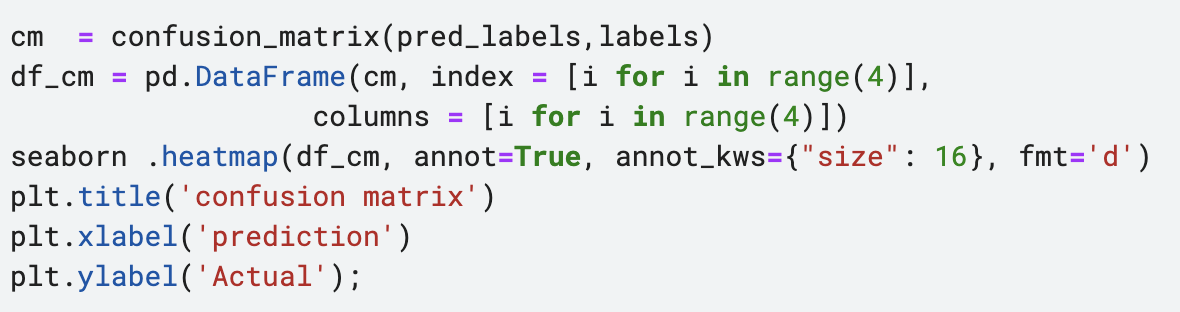
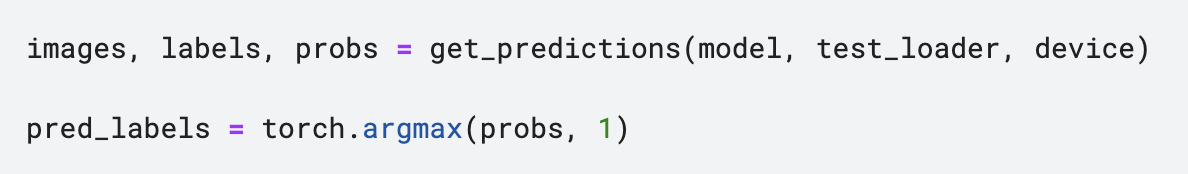
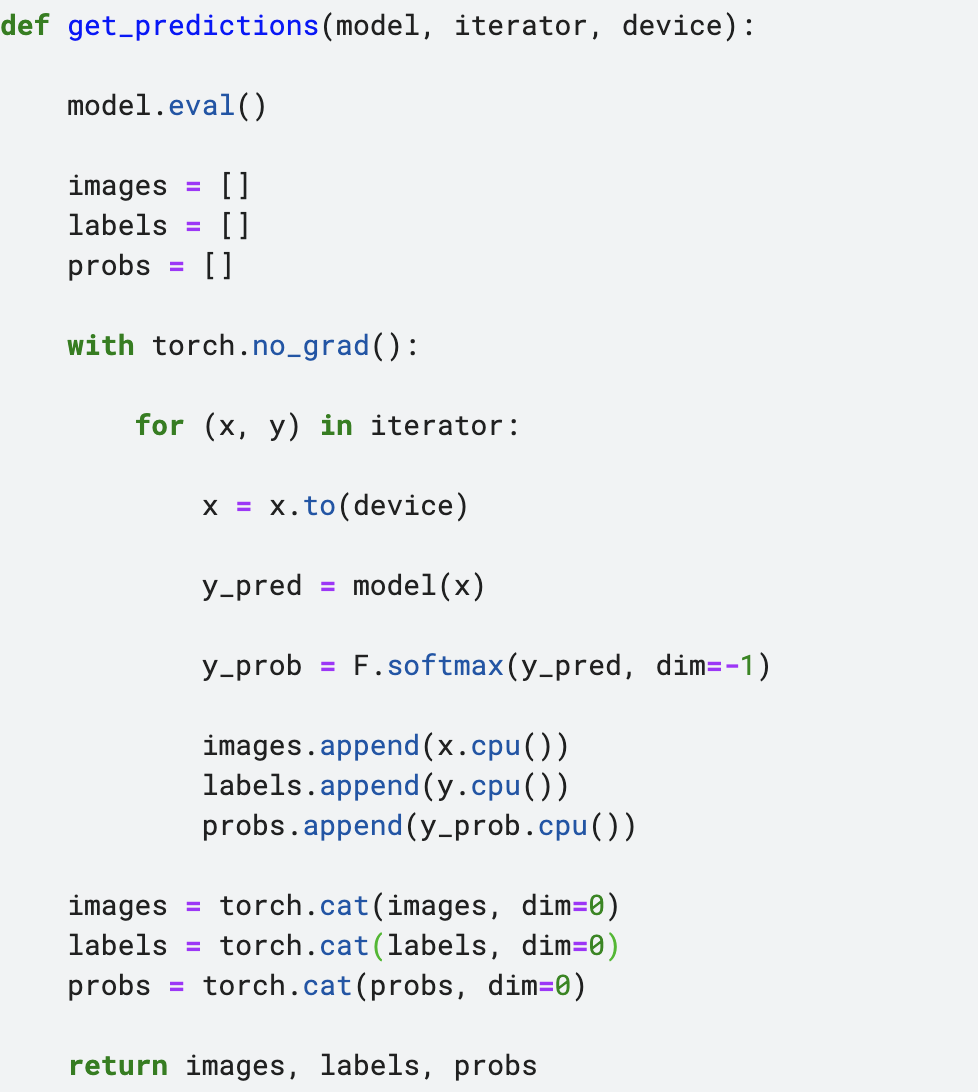
3.4.4 Result of the model

* Output of data training and evaluation



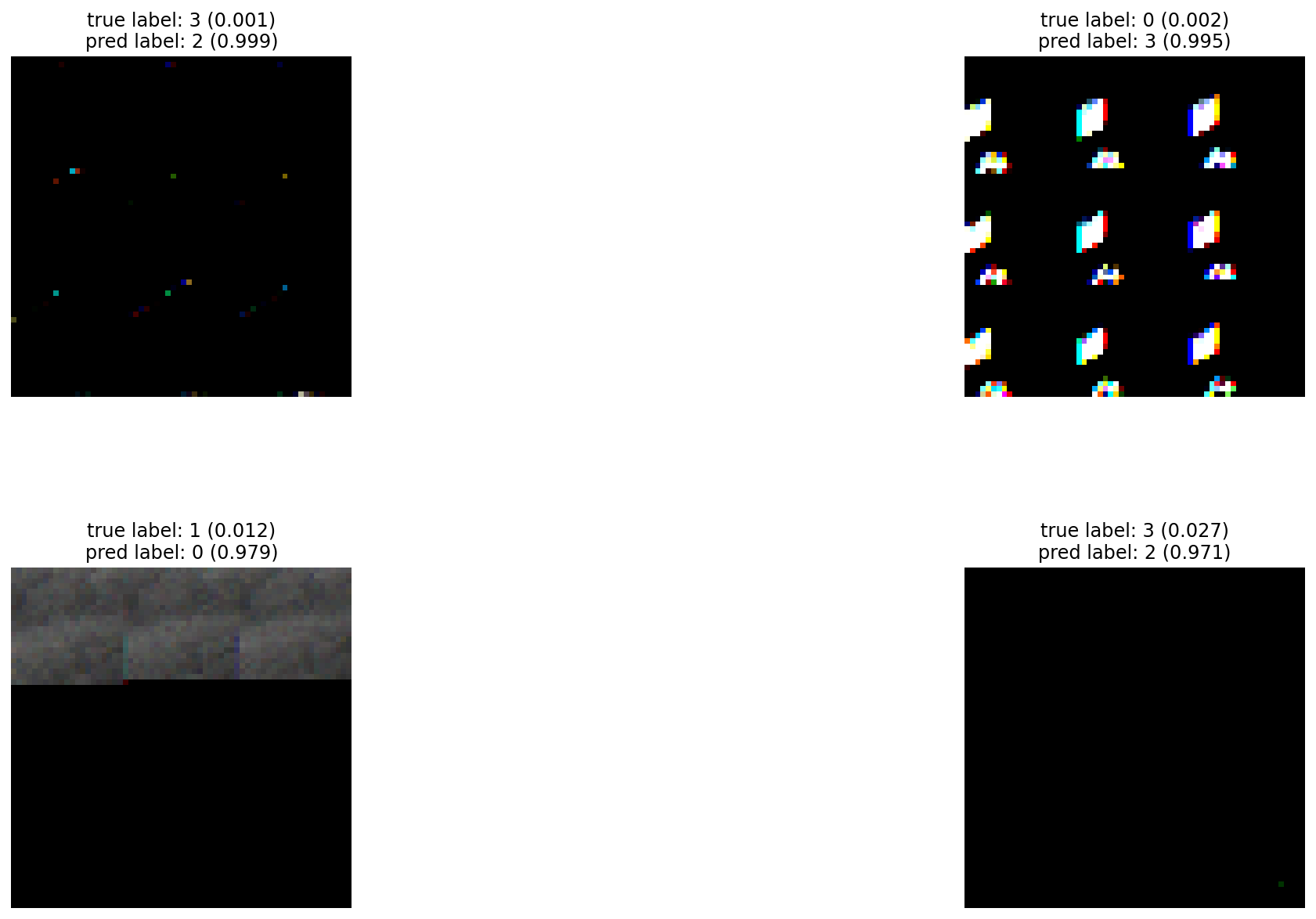
3.4.5 Final Result on Testing

* Load the test\_data into the trained model and calculate the accuracy rate, turn the accuracy rate into a confusion matrix.



3.3.6 Others

* Show incorrect images



1. **Conclusion**

The testing results on MLP is :

Error : 0.265

Accuracy : 88.26%

The testing results on DenseNet is :

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.

1. **contribution**
2. Contribution of Wong Kai Yuan :

* Implementation of Multilayer Perceptron (MLP) model on dataset
* 50% of report writing
* 50% of Presentation PPT

1. 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.