Intel Image Classification

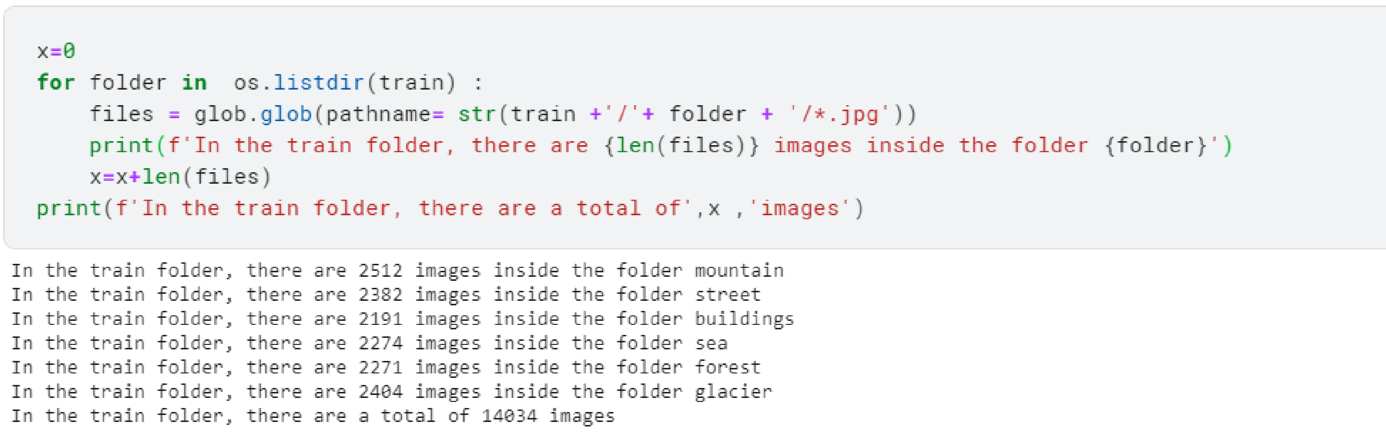
1. Problem description

The dataset that I am using has nacural scenes all around the world of various places. It is distributed in 25000 images of the size of 150x150 in 6 classes (building, forest, glacier, mountain, sea and street). The dataset is already seperated in the train, test and prediction folders. The goal of this project was to train a model to predict which of the natural scene was being shown of the image. I used CNN to solve this problem. CNN is a Convolutional Neural Network which specilizes in image recognition.

2. Methodology

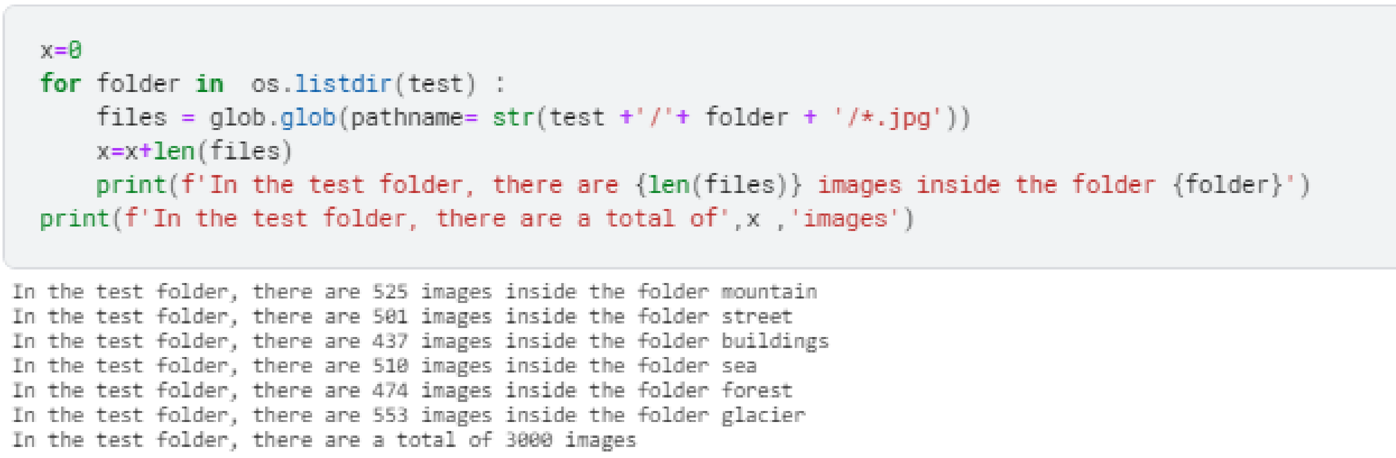
```
images = []
for folder in os.listdir(train):
    for image in os.listdir(train + '/' + folder):
        images.append(os.path.join(train, folder, image))

plt.figure(figsize=(20,20))

for i in range(16):
    random_img = random.choice(images)
    imgs = mpimg.imread(random_img)
    plt.subplot(4, 4, i+1)
    plt.imshow(imgs)
plt.show()
```

The first thing I did was print random images from each folder to see the types of images I have in my dataset. As we can see the dataset has images of each of the 6 classes said. It is also visable that every image is the size 150x150.



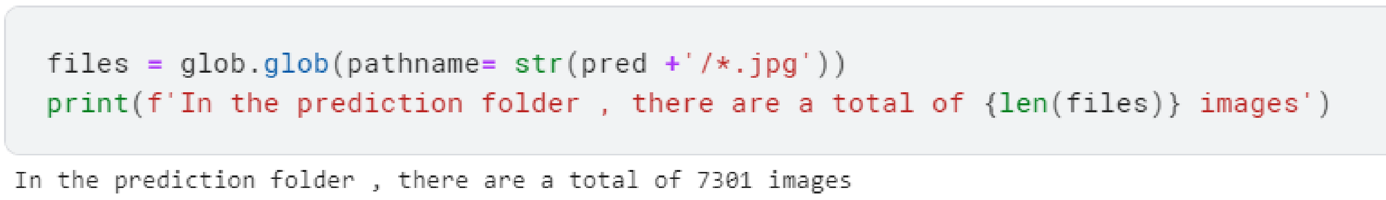
```
x=0
for folder in  os.listdir(train) :
    files = glob.glob(pathname= str(train +'/'+ folder + '/*.jpg'))
    print(f'In the train folder, there are {len(files)} images inside the folder {folder}')
    x=x+len(files)
print(f'In the train folder, there are a total of',x ,'images')
```

```
In the train folder, there are 2512 images inside the folder mountain
In the train folder, there are 2382 images inside the folder street
In the train folder, there are 2191 images inside the folder buildings
In the train folder, there are 2274 images inside the folder sea
In the train folder, there are 2271 images inside the folder forest
In the train folder, there are 2404 images inside the folder glacier
In the train folder, there are a total of 14034 images
```



```
x=0
for folder in  os.listdir(test) :
    files = glob.glob(pathname= str(test +'/'+ folder + '/*.jpg'))
    x=x+len(files)
    print(f'In the test folder, there are {len(files)} images inside the folder {folder}')
print(f'In the test folder, there are a total of',x ,'images')
```

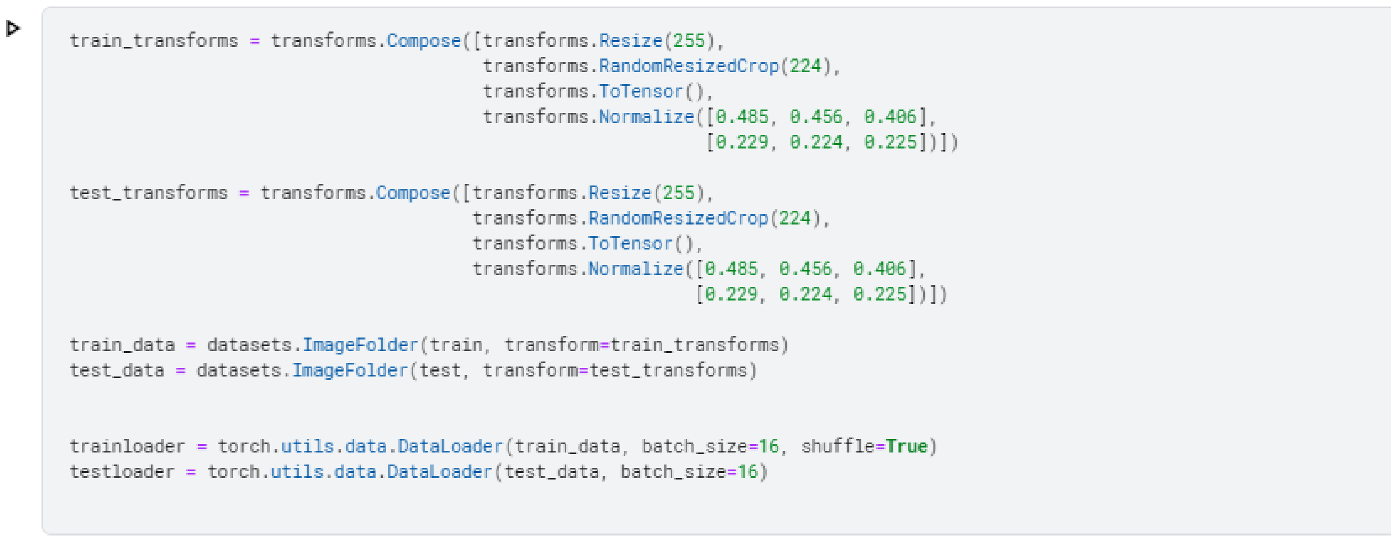
```
In the test folder, there are 525 images inside the folder mountain
In the test folder, there are 501 images inside the folder street
In the test folder, there are 437 images inside the folder buildings
In the test folder, there are 510 images inside the folder sea
In the test folder, there are 474 images inside the folder forest
In the test folder, there are 553 images inside the folder glacier
In the test folder, there are a total of 3000 images
```



```
files = glob.glob(pathname= str(pred +'/*.jpg'))
print(f'In the prediction folder , there are a total of {len(files)} images')
```

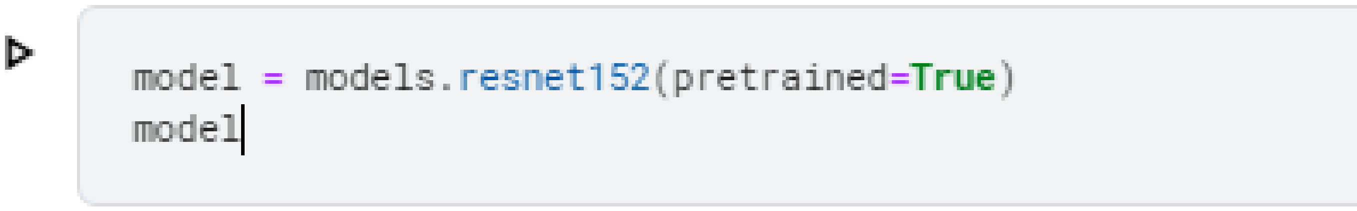
```
In the prediction folder , there are a total of 7301 images
```

Afterwards, I checked how many images are in each folder, and how they are seperated. It is really important for the dataset to have an equal amount of images in every folder so that the model can train itself properly and have the best result. From the closer look at the data, The train set has 14034 images, and each folder has around 2000- 2500, the test set has 3000 images, and each folder has around 450-550 images, and the prediction set has 7301 images. This is a balanced set so I did not have to do anything to make it better.



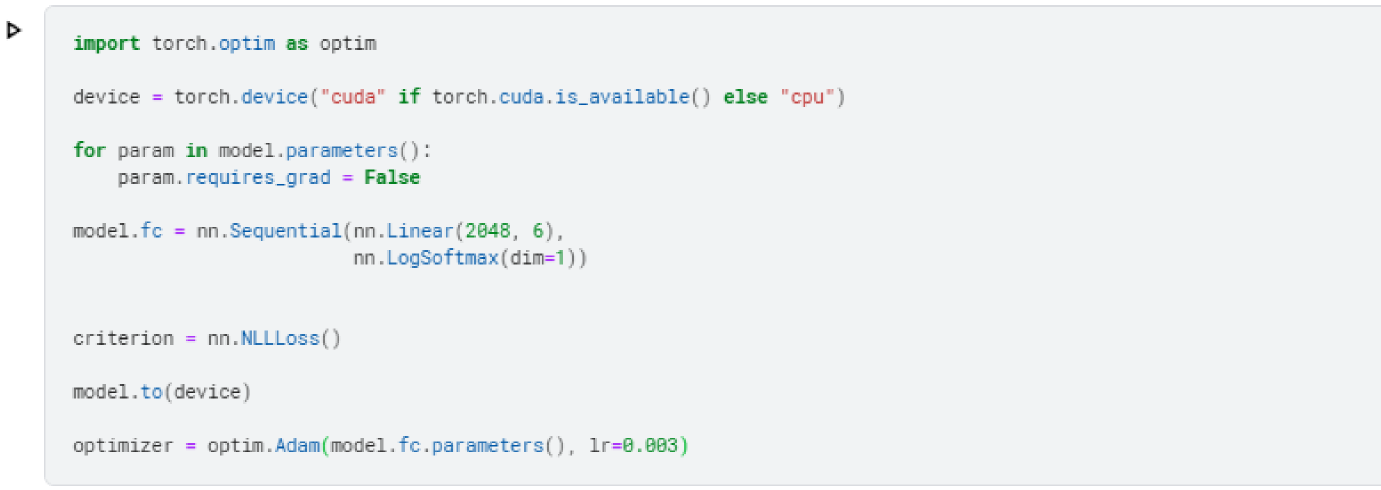
```
train_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.RandomResizedCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.RandomResizedCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

train_data = datasets.ImageFolder(train, transform=train_transforms)
test_data = datasets.ImageFolder(test, transform=test_transforms)


trainloader = torch.utils.data.DataLoader(train_data, batch_size=16, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=16)
```

The transofrmation is pretty basic, I tried some other methods like RandomRotation and simmilar to improve my accurarcy, but in the end it just made it worse. For the same reasoning I used batch size 16 because it gave the best accurarcy. For normalize I used the standard mean and std of Imagenet.



```
model = models.resnet152(pretrained=True)
model
```

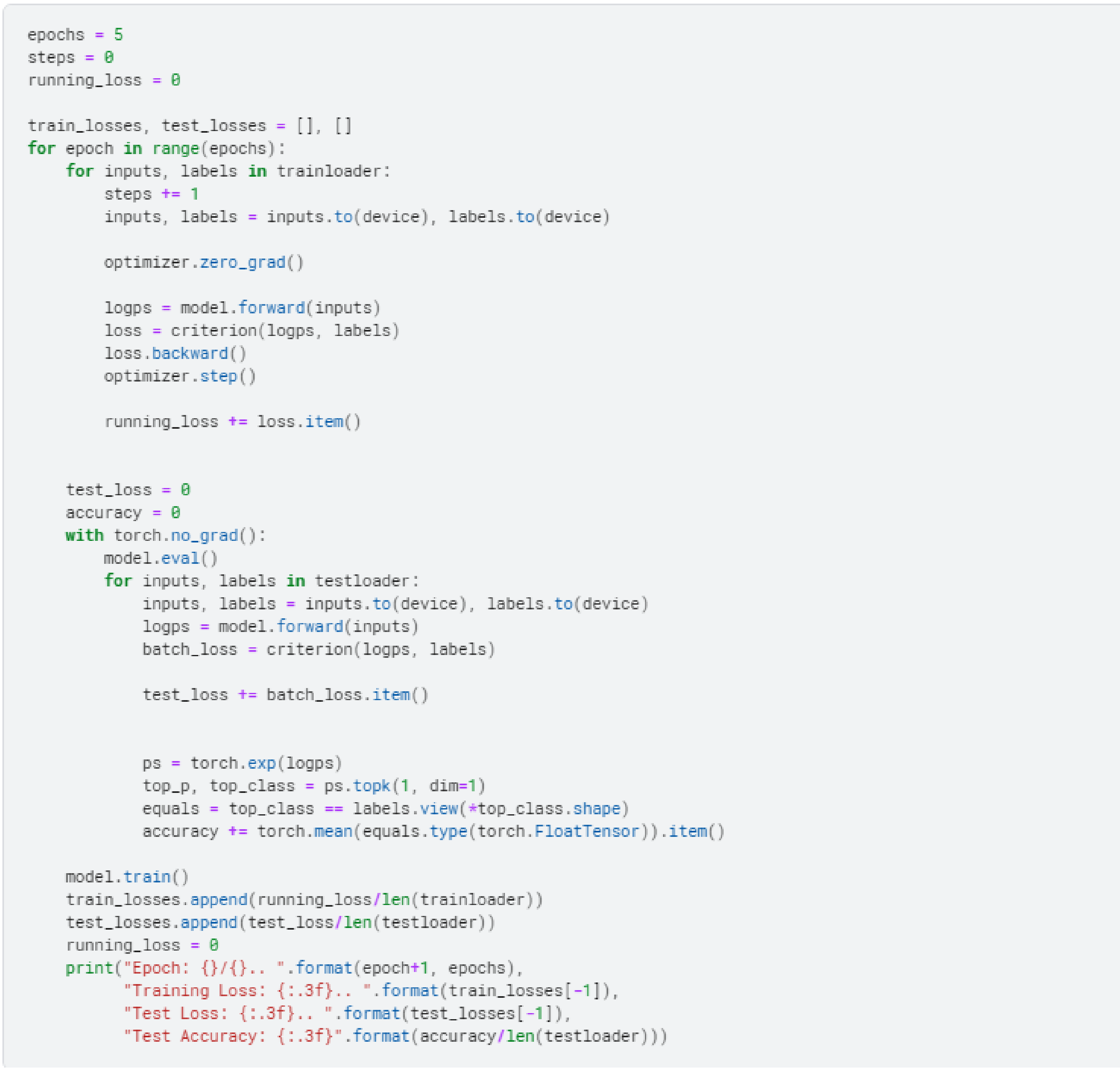
For the model I decided to go with resnet152 because it gave better results than vgg16. I checked what other people did and most of them used either of these pretrained models with some people making their own models.



```
import torch.optim as optim

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

for param in model.parameters():
    param.requires_grad = False

model.fc = nn.Sequential(nn.Linear(2048, 6),
                         nn.LogSoftmax(dim=1))


criterion = nn.NLLLoss()

model.to(device)

optimizer = optim.Adam(model.fc.parameters(), lr=0.003)
```

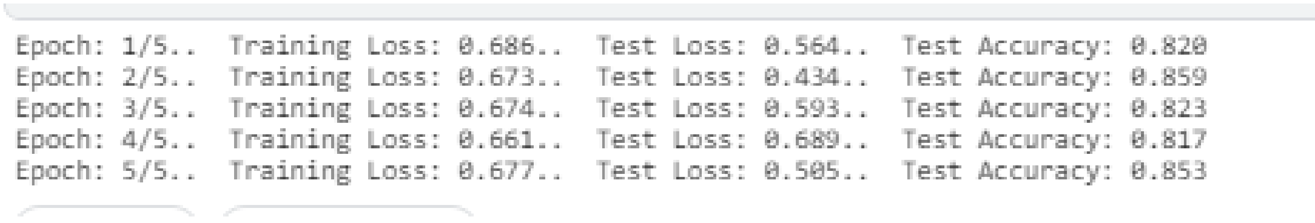
For the optimizer here I decided to go with Adam, and after messing around with the learning rate I came to the realization that the best results I've gotten were with 0.003. I tried going higher and lower but the results were always worse.

3. Experiments and evaulation



```python
epochs = 5
steps = 0
running_loss = 0

train_losses, test_losses = [], []
for epoch in range(epochs):
    for inputs, labels in trainloader:
        steps += 1
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        logps = model.forward(inputs)
        loss = criterion(logps, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()


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

            test_loss += batch_loss.item()


            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()

    model.train()
    train_losses.append(running_loss/len(trainloader))
    test_losses.append(test_loss/len(testloader))
    running_loss = 0
    print("Epoch: {}/{}.. ".format(epoch+1, epochs),
          "Training Loss: {:.3f}.. ".format(train_losses[-1]),
          "Test Loss: {:.3f}.. ".format(test_losses[-1]),
          "Test Accuracy: {:.3f}".format(accuracy/len(testloader)))
```

This code is modified from the exercise „Interference and Validation (Solution)“. It writes the training/test loss and accurarcy of the pretrained model.

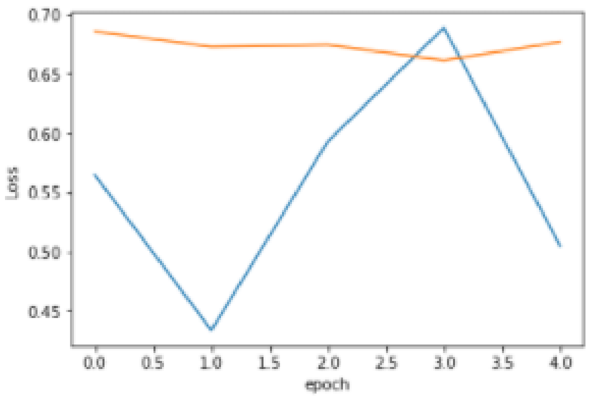


```
Epoch: 1/5..  Training Loss: 0.686..  Test Loss: 0.564..  Test Accuracy: 0.820
Epoch: 2/5..  Training Loss: 0.673..  Test Loss: 0.434..  Test Accuracy: 0.859
Epoch: 3/5..  Training Loss: 0.674..  Test Loss: 0.593..  Test Accuracy: 0.823
Epoch: 4/5..  Training Loss: 0.661..  Test Loss: 0.689..  Test Accuracy: 0.817
Epoch: 5/5..  Training Loss: 0.677..  Test Loss: 0.505..  Test Accuracy: 0.853
```

As we can see the accurarcy is around 82-86%.

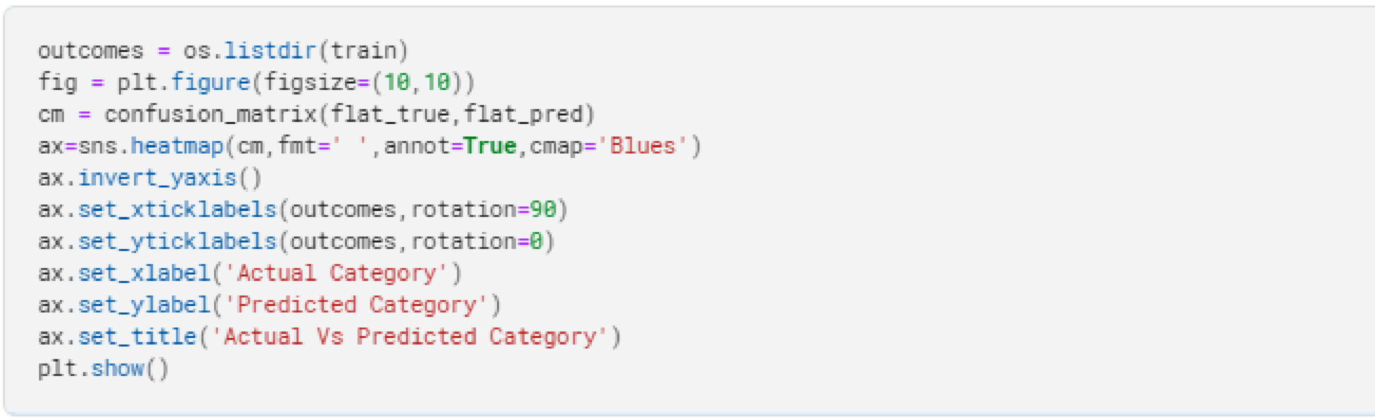
```python
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(test_losses)
plt.plot(train_losses)
plt.xlabel("epoch")
plt.ylabel("Loss")
```

[45… Text(0, 0.5, 'Loss')

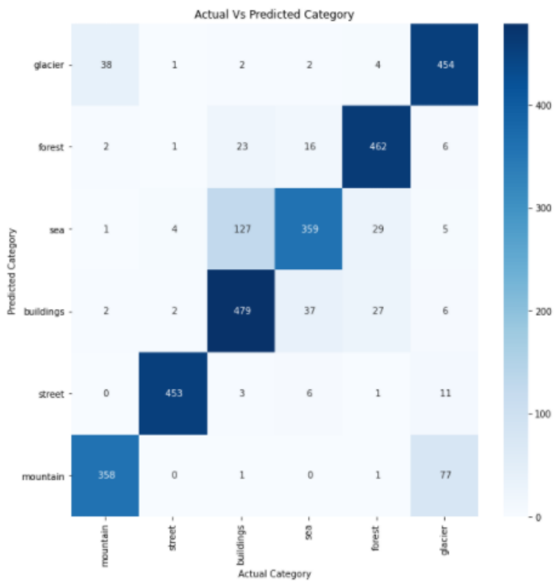


This graph shows the progress of training and test loss when it comes to training the model on each epoch. When googling the issue I tried lowering the learning rate to make the loss be more stable so it gets lowered with each epoch, but the accurarcy went down a lot so I decided against it.

```python
y_pred_list = []
y_true_list = []
with torch.no_grad():
    for inp, labels in testloader:
            inp, labels = inp.to(device), labels.to(device)
            y_test_pred = model(inp)
            _, y_pred_tag = torch.max(y_test_pred, dim = 1)
            y_pred_list.append(y_pred_tag.cpu().numpy())
            y_true_list.append(labels.cpu().numpy())

flat_pred = []
flat_true = []
for i in range(len(y_pred_list)):
    for j in range(len(y_pred_list[i])):
        flat_pred.append(y_pred_list[i][j])
        flat_true.append(y_true_list[i][j])
```



```
outcomes = os.listdir(train)
fig = plt.figure(figsize=(10,10))
cm = confusion_matrix(flat_true,flat_pred)
ax=sns.heatmap(cm,fmt=' ',annot=True,cmap='Blues')
ax.invert_yaxis()
ax.set_xticklabels(outcomes,rotation=90)
ax.set_yticklabels(outcomes,rotation=0)
ax.set_xlabel('Actual Category')
ax.set_ylabel('Predicted Category')
ax.set_title('Actual Vs Predicted Category')
plt.show()
```



This code was used to make the confusion matrix. As we can see it shows us that most of the pictures are predicted fairly correctly with the exception of the model mistaking buildings for seas and glaciers for mountains.

4. Conclusion

As we can see the model did its job fine and I am satisfied with the results. The part that gave me most issues was figuring out the optimal learning rate.

5. References

1. https://stackoverflow.com/questions/53290306/confusion-matrix-and-test-accuracy-for-pytorch-transfer-learning-tutorial

2. Transfer Learning (Homework Introduction)

3. Inference and Validation (Solution)