IMPLEMENTATION

- Here I have used Google colab to write and run the code
- Question 2 has 3 parts
- Image is taken from my mounted google drive rather than uploading directly to colab
- Pytorch library is used for all the deep learning applications

1st Part:

- For this part data_loaders are created of batch size 1 as I am using it as a feature extractor and hence batch size does not matter here.
- Pre trained Resnet18 model is then initialised and last fc-layer is removed.
- Model is then brought to eval-mode (a very important step because there are Batch-Normalisation layers and this ensures they take static value during feature extraction).
- For train images the features are extracted using the above modified model and are stacked row-wise in a numpy array (x_train), similarly labels are also stacked row wise in a numpy array (y_train).
- Same above thing is repeated for test images and test dataset created.
- Now I have used the KNN of sklearn library with k value as 5(which is the default value) and trained it on training dataset created.
- Now it is evaluated for test dataset and the accuracy noted was 91.67%.

2nd Part:

- For this part I have taken batch size for train_dataloader as 64(else it would be a SGD optimiser) and batch size 1 for test_dataloader as during testing we donot update gradient.
- Pre trained Resnet18 model is then initialised and weights of all layers except last are freezed.
- The last layer output is changed to 6 instead of 1000 because there are 6 classes to classify.
- The following parameters are set for training the model:
    1. Optimiser – Adam

2. Loss function – Cross Entropy Loss
3. Number of epochs - 50

- The number of epoch is chosen after less number epochs like 5 or 10 did not give good accuracy on both train and test data. I had trained for 20 epochs and for 50 epochs to see the difference and there was not so much difference in test accuracy. I have only included 50 epochs in my final code.

```
epoch 0 loss 14.903
epoch 1 loss 4.711
epoch 2 loss 2.085
epoch 3 loss 1.101
epoch 4 loss 0.688
epoch 5 loss 0.483
epoch 6 loss 0.349
epoch 7 loss 0.279
epoch 8 loss 0.217
epoch 9 loss 0.176
epoch 10 loss 0.156
epoch 11 loss 0.125
epoch 12 loss 0.105
epoch 13 loss 0.095
epoch 14 loss 0.083
epoch 15 loss 0.067
epoch 16 loss 0.063
epoch 17 loss 0.057
epoch 18 loss 0.057
epoch 19 loss 0.049
```

Decreasing losses with each epoch(for 20 epoch case)

- Then after training model is brought back to evaluation mode and then evaluated on test images and the accuracy I got was 95% for 20 epochs and 96.67% accuracy for 50 epochs.

3rd part:

- Here I have used 1 flatten layer and 1 fully connected layer.
- Batch size of train_dataloader is kept 64 and that of test dataloader is 1.
- I have also trained a comparatively complex model consisting of few convolution layers, max pooling layers etc but I have replaced the simple model consisting of only 2 layers in my final code because the term "*simple*" was mentioned in the question.

- The following parameters are set for training the model:
    1. Optimiser – Adam
    2. Loss function – Cross Entropy Loss
    3. Number of epochs - 20
- In my complex model I was getting an accuracy of 41.67% and in my simple model an accuracy of 24.16%

```
[ ] class simple_model(nn.Module):
      def __init__(self):
        super(simple_model,self).__init__()
        self.conv1 = nn.Conv2d(3,64,kernel_size=5,padding =0)
        self.maxpool1 = nn.MaxPool2d(2,stride = 2)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(64,128,kernel_size=5,padding =1)
        self.maxpool2 = nn.MaxPool2d(2,stride = 2)
        self.relu2 = nn.ReLU()
        self.Avg = nn.AdaptiveAvgPool2d(output_size=(1,1))
        self.fc = nn.Linear(128,6)
      def forward(self,x):
        x=self.conv1(x)
        #print(x.shape)
        x=self.relu1(self.maxpool1(x))
        #print(x.shape)
        x=self.conv2(x)
        x=self.relu2(self.maxpool2(x))
        x=self.Avg(x)
        x=x.view(-1,128)
        x=self.fc(x)
        return(x)
```

Complex Model

```python
class simple_model(nn.Module):
  def __init__(self):
    super(simple_model,self).__init__()
    self.layer1 = nn.Flatten()
    self.layer2 = nn.Linear(150528,6)
  def forward(self,x):
    x=self.layer1(x)
    x=self.layer2(x)
    return(x)
```

Simple Model

```
epoch 0 loss 462.625
epoch 1 loss 154.181
epoch 2 loss 72.376
epoch 3 loss 43.484
epoch 4 loss 22.141
epoch 5 loss 13.067
epoch 6 loss 6.856
epoch 7 loss 6.536
epoch 8 loss 7.613
epoch 9 loss 6.072
epoch 10 loss 4.256
epoch 11 loss 2.541
epoch 12 loss 1.955
epoch 13 loss 1.087
epoch 14 loss 1.591
epoch 15 loss 0.830
epoch 16 loss 1.038
epoch 17 loss 0.697
epoch 18 loss 1.065
epoch 19 loss 1.123
```

Decreasing losses with epoch in simple model

LEARNINGS AND PROBLEMS FACED

Part 1:

The major problem I faced in part 1 is converting the feature vector that is in tensor form(because of pytorch) to numpy array to be used by KNN. Various function like squeeze (to remove dimensions) and detach (to turn computing gradient false) were used to overcome this difficulty.

<u>Part2:</u>

The main problem faced in this part is how to freeze the weights of other layers and not the last layer. It took a while for me to get that after freezing if I reinitialise a layer again then it unfreezes that layers weight and I used this to update weight of last layer. To freeze we have to switch off requires_grad function of each parameter to be frozen.

<u>Part3:</u>

The layers in pytorch usually takes an extra dimension in input that is of batch size. Hence when I was testing output for a single image I was getting error later I understood the mistake and added an extra dimension of one for a single image for testing. In train/test dataloader this is not required because the pytorch functions take care of this.

<u>COMPARISIONS:</u>

The main difference between the 3 parts is the way pre-trained resnet18 model is handled.

In part 1 no changes are made to the resnet18 model except for deleting the last fc layer to make it work as a feature extractor. Here no training is done to this model instead KNN is used as classifier and thus KNN is trained.

In part 2 resnet18 is tuned or changed. The last layer is replaced by another layer and is trained, although weights of previous layers are kept fixed (one may not fix it) but here the last layer is trained on training images and thus it acts as a classifier.

In part 3 there is no pre trained model and the model is built from scratch and trained.