Computer Vision

Assignment 1

Sahil Goyal

2020326

1)

1.

We have downloaded the SVHN dataset and extracted the data by using scipy.io.loadmat(). We have created our custom dataset class. We have also initialised Weights and Biases.

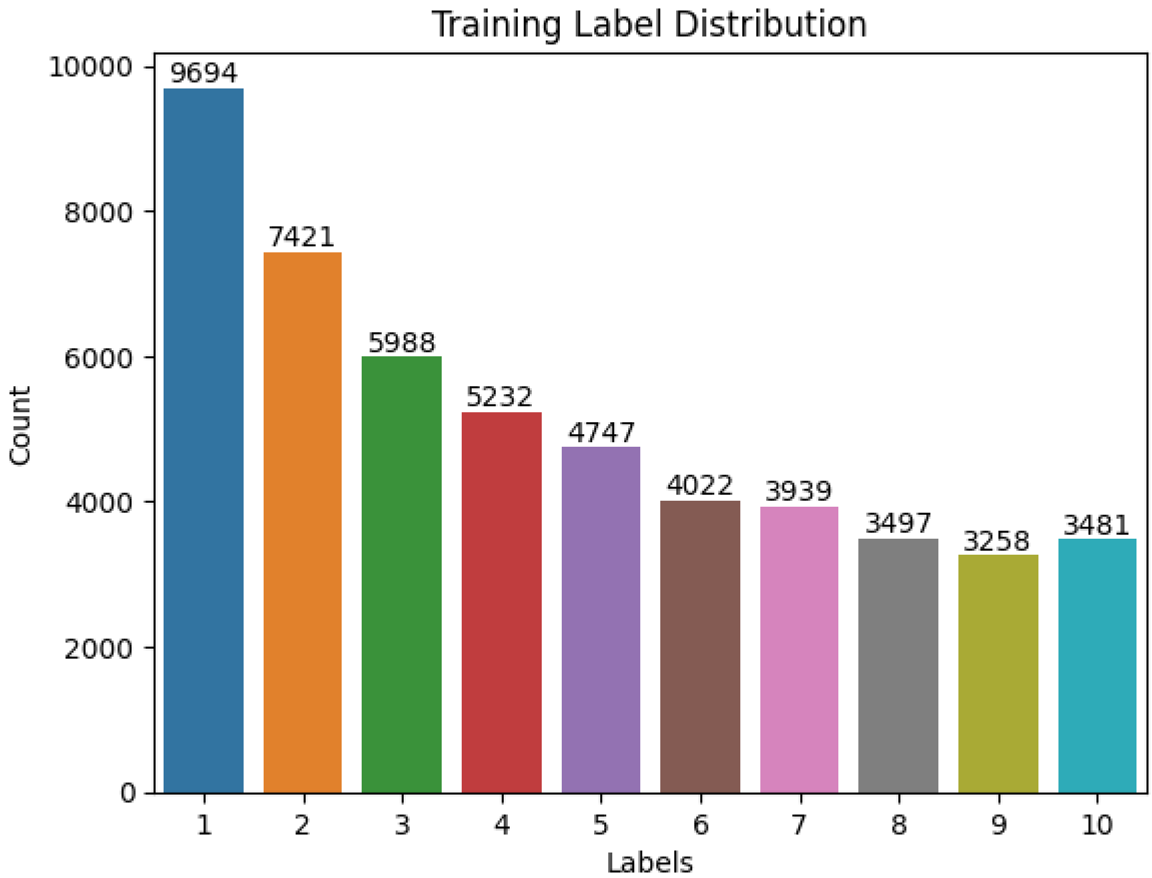
```
class custom(Dataset):
    def __init__(self, data, transform=None):
        self.images = data['X']
        self.labels = data['y']
        self.transform = transform
    def __len__(self):
        return self.images.shape[3]

    def __getitem__(self, idx):
        image=self.images[:,:,:,idx]
        label = self.labels[idx]
        if label[0] == 10:
            label = 0
        else:
            label = label[0]
        if self.transform:
            image = self.transform(image)
        return image, label
```

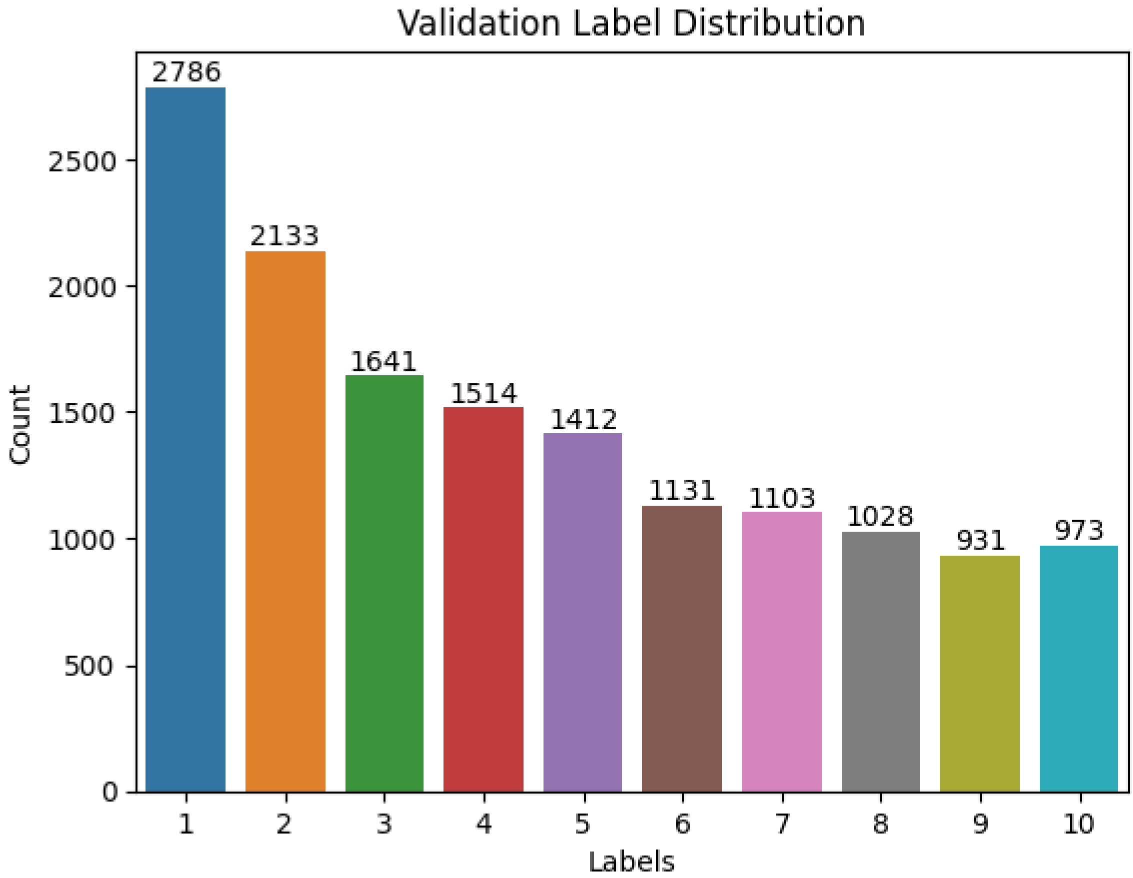
After that we made an object of this class that has all of the data, and then we use random_split() function in order to split the data into a 70:20:10 ratio (train,val,test). After that we have created custom dataloaders for all the data. We are using a batch size of 64.

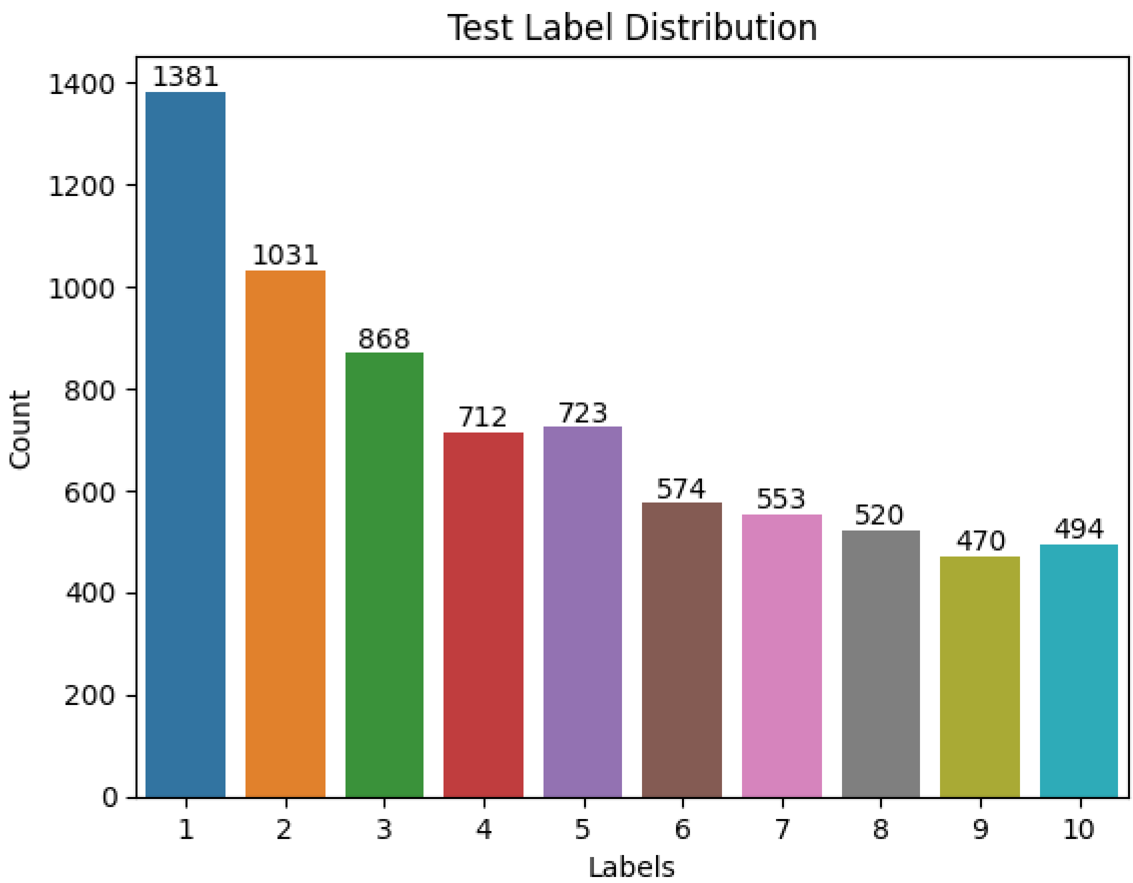
```
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
val_loader = DataLoader(val_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=True)
print(len(train_loader))
print(type(train_loader.dataset))
```

```
802
<class 'torch.utils.data.dataset.Subset'>
```

After this, we visualise the data by plotting the classwise distribution of the training, validation and testing data.




Validation Label Distribution
Count
Labels
2786
2133
1641
1514
1412
1131
1103
1028
931
973
0
500
1000
1500
2000
2500
1
2
3
4
5
6
7
8
9
10


Test Label Distribution
Count
Labels
1381
1031
868
712
723
574
553
520
470
494
0
200
400
600
800
1000
1200
1400
1
2
3
4
5
6
7
8
9
10

2.

We have created a custom CNN class with 2 Convolution Layers having a kernel size of 3×3 and padding of 1. We have used 32 feature maps for the first layer and 64 for the second.

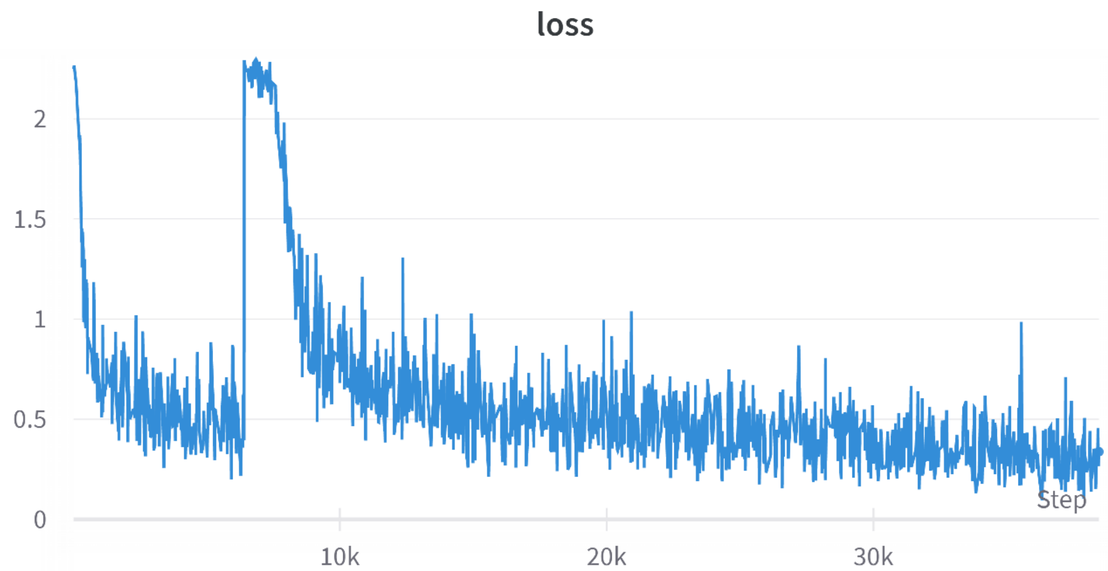
```
class cnn(nn.Module):
    def __init__(self):
        super(cnn, self).__init__()
        self.con1 = nn.Conv2d(3, 32,kernel_size=3, stride=1, padding=1)
        self.con2 = nn.Conv2d(32, 64,kernel_size=3, stride=1, padding=1)
        self.fc=nn.Linear(32*32*64, 10)
    def forward(self,x):
        return self.fc(torch.flatten(F.relu(self.con2(F.relu(self.con1(x)))),1))
```

After this, we create an object of this class. We are using cross entropy loss

```
model=cnn()
print(model)
model.to("cuda")
optimizer=optim.Adam(model.parameters(),lr=0.001)
criterion=nn.CrossEntropyLoss()
```

```
cnn(
  (con1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (con2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc): Linear(in_features=65536, out_features=10, bias=True)
)
```

We have used wandb to log the losses on the training set.

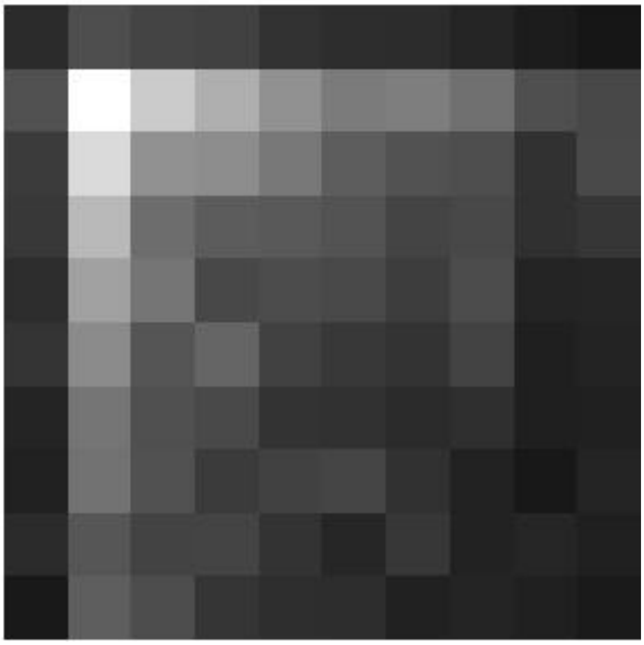


```
Epoch:  0  Loss:  2.223730884882577
Epoch:  1  Loss:  2.0041543050597137
Epoch:  2  Loss:  1.2196324353949388
Epoch:  3  Loss:  0.8706577594515094
Epoch:  5  Loss:  0.68386135653516
Epoch:  6  Loss:  0.6395862305773761
Epoch:  7  Loss:  0.6134442677373007
Epoch:  8  Loss:  0.5971916733574392
Epoch:  9  Loss:  0.5773981164518437
Epoch:  10  Loss:  0.5605208190190525
Epoch:  11  Loss:  0.547450689566403
Epoch:  12  Loss:  0.5345426653500209
Epoch:  13  Loss:  0.5217051561039285
Epoch:  14  Loss:  0.5101693595139463
Epoch:  15  Loss:  0.4989051696962846
Epoch:  16  Loss:  0.48996303613272096
Epoch:  17  Loss:  0.4814171576106043
Epoch:  18  Loss:  0.4706470612575883
Epoch:  19  Loss:  0.4626735669381898
Epoch:  20  Loss:  0.4527256051798414
Epoch:  21  Loss:  0.44521105140819217
Epoch:  22  Loss:  0.4363443310235504
Epoch:  23  Loss:  0.428937703371048
Epoch:  24  Loss:  0.41999247031950593
Epoch:  25  Loss:  0.41197028384541634
Epoch:  26  Loss:  0.40569621335687184
Epoch:  27  Loss:  0.3962647310107426
Epoch:  28  Loss:  0.39091962295354454
Epoch:  29  Loss:  0.38212295672728536
Epoch:  30  Loss:  0.37437014108955713
```

```
Epoch:  31  Loss:  0.36924910723717136
Epoch:  32  Loss:  0.36250510329654684
Epoch:  33  Loss:  0.35482114009057497
Epoch:  34  Loss:  0.34761740175305755
Epoch:  35  Loss:  0.34247695620257657
Epoch:  36  Loss:  0.33492145889863706
Epoch:  37  Loss:  0.32881460065704926
Epoch:  38  Loss:  0.32294202704642183
Epoch:  39  Loss:  0.3163109078000014
```

After this, we evaluate the model on the test set. The accuracy and F1 scores were:

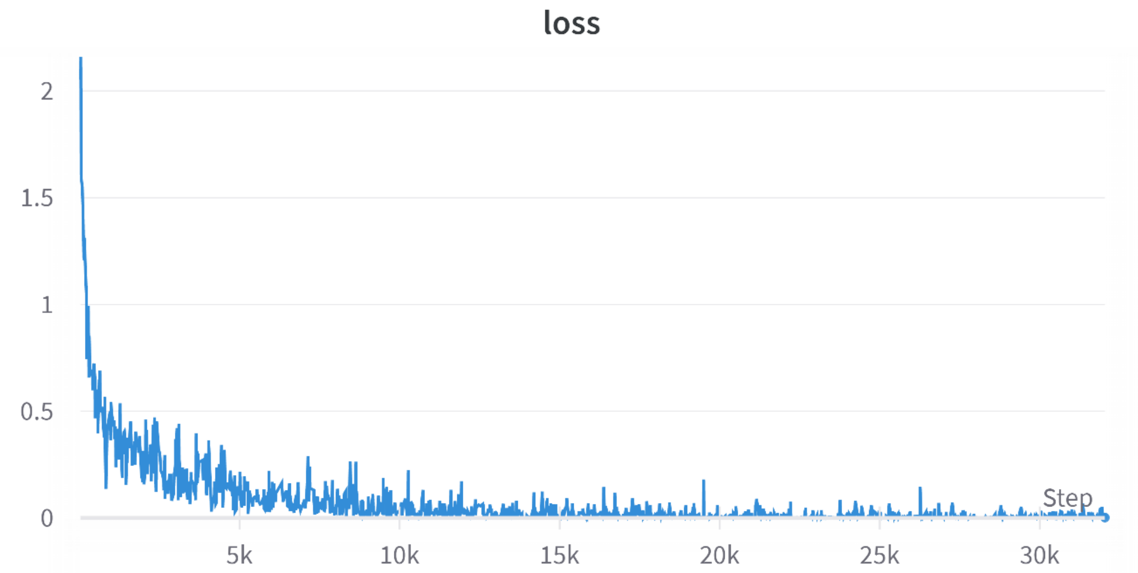
```
Accuracy :  0.8532623532623532
F1 Score :  0.7885640433632752
Confusion Matrix :
  [[ 43  78  68  65  50  45  44  37  28  22]
 [ 81 278 203 175 145 123 126 112  79  72]
 [ 59 218 145 141 120  93  82  78  49  73]
 [ 56 185 109  92  89  83  68  72  48  54]
 [ 45 161 117  72  76  73  61  76  36  37]
 [ 52 139  85 101  65  56  51  67  31  35]
 [ 36 117  80  73  51  49  43  47  31  32]
 [ 33 114  81  59  65  69  49  34  24  36]
 [ 43  86  67  68  51  38  55  34  38  32]
 [ 25  94  77  53  46  45  33  36  32  26]]
```



3.

We have used the pretrained model of resnet 18.

```
model2=torchvision.models.resnet18(pretrained=True)
model2.fc=nn.Linear(model2.fc.in_features,10)
model2.to("cuda")
optimizer2=optim.Adam(model2.parameters(),lr=0.001)
criterion2=nn.CrossEntropyLoss()
```

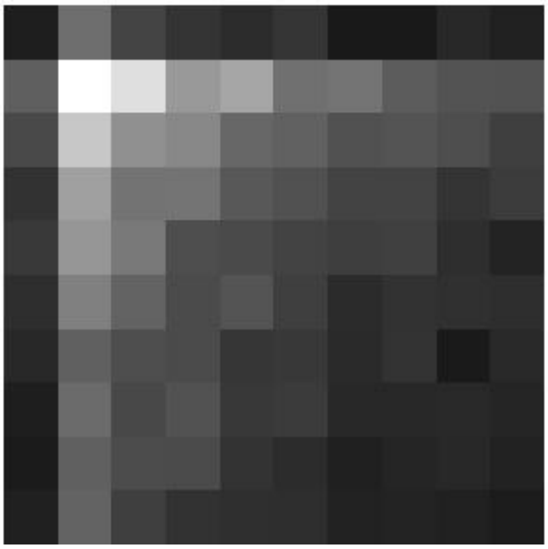



```
Epoch:  0  Loss:  0.8976613628745377
Epoch:  1  Loss:  0.3955819569621003
Epoch:  2  Loss:  0.2944721160897813
Epoch:  3  Loss:  0.23382415772041776
Epoch:  4  Loss:  0.19049941456442387
Epoch:  5  Loss:  0.15590826469122843
Epoch:  6  Loss:  0.12686529748857095
Epoch:  7  Loss:  0.11139837767640849
Epoch:  8  Loss:  0.09254216529113099
Epoch:  9  Loss:  0.0749725254294358
Epoch:  10  Loss:  0.06314013270716975
Epoch:  11  Loss:  0.054518551207170765
Epoch:  12  Loss:  0.04947874439500523
Epoch:  13  Loss:  0.04115947010409275
Epoch:  14  Loss:  0.03659638336788807
Epoch:  15  Loss:  0.03234873846332378
Epoch:  16  Loss:  0.030606816215844164
Epoch:  17  Loss:  0.02897314853730212
Epoch:  18  Loss:  0.03214405104351007
Epoch:  19  Loss:  0.020281399535831003
Epoch:  20  Loss:  0.02577418491910293
Epoch:  21  Loss:  0.020879494879930977
Epoch:  22  Loss:  0.016533842519407004
Epoch:  23  Loss:  0.015213110182076365
Epoch:  24  Loss:  0.01271740530530151
Epoch:  25  Loss:  0.011156602803954623
Epoch:  26  Loss:  0.009008767614658951
Epoch:  27  Loss:  0.007655284388367948
Epoch:  28  Loss:  0.007375530721888627
Epoch:  29  Loss:  0.00770081062321078
```

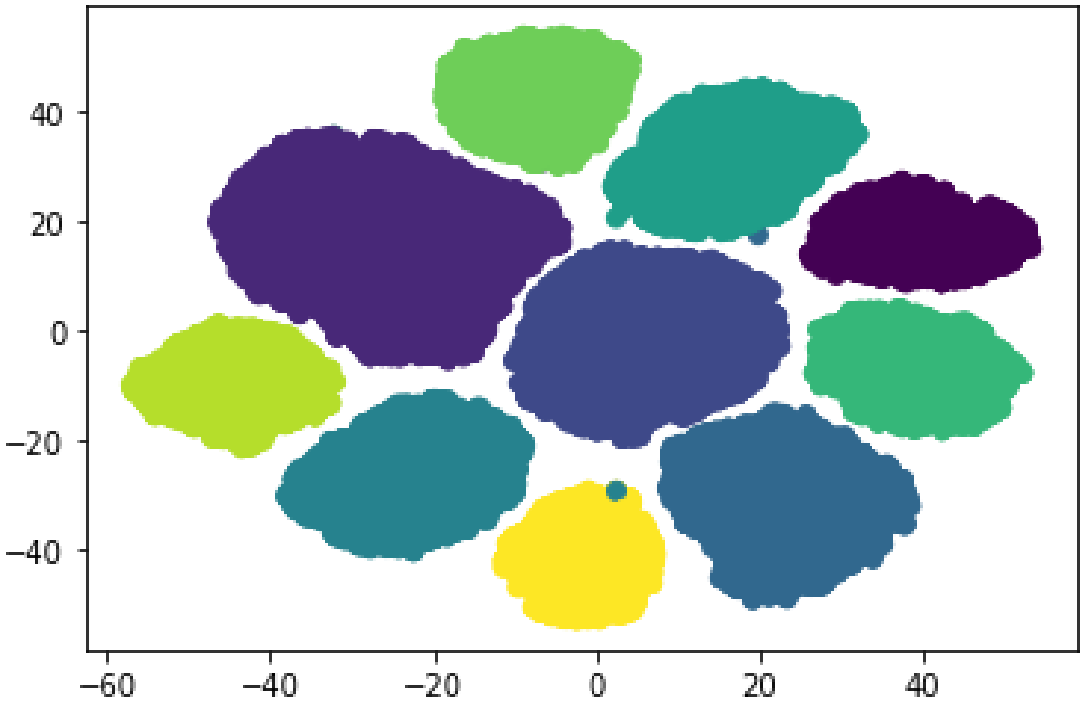
```
Epoch:  30  Loss:  0.008044710180181035
Epoch:  31  Loss:  0.008790599839116488
Epoch:  32  Loss:  0.007462119307278489
Epoch:  33  Loss:  0.00913835721791686
Epoch:  34  Loss:  0.01021281892037377
Epoch:  35  Loss:  0.00842513694428851
Epoch:  36  Loss:  0.009284549265221879
Epoch:  37  Loss:  0.008137257347516878
Epoch:  38  Loss:  0.007697554340532286
Epoch:  39  Loss:  0.010210868845916182
```

We have tested the model on the testing set. The accuracy and F1 score are as follows:

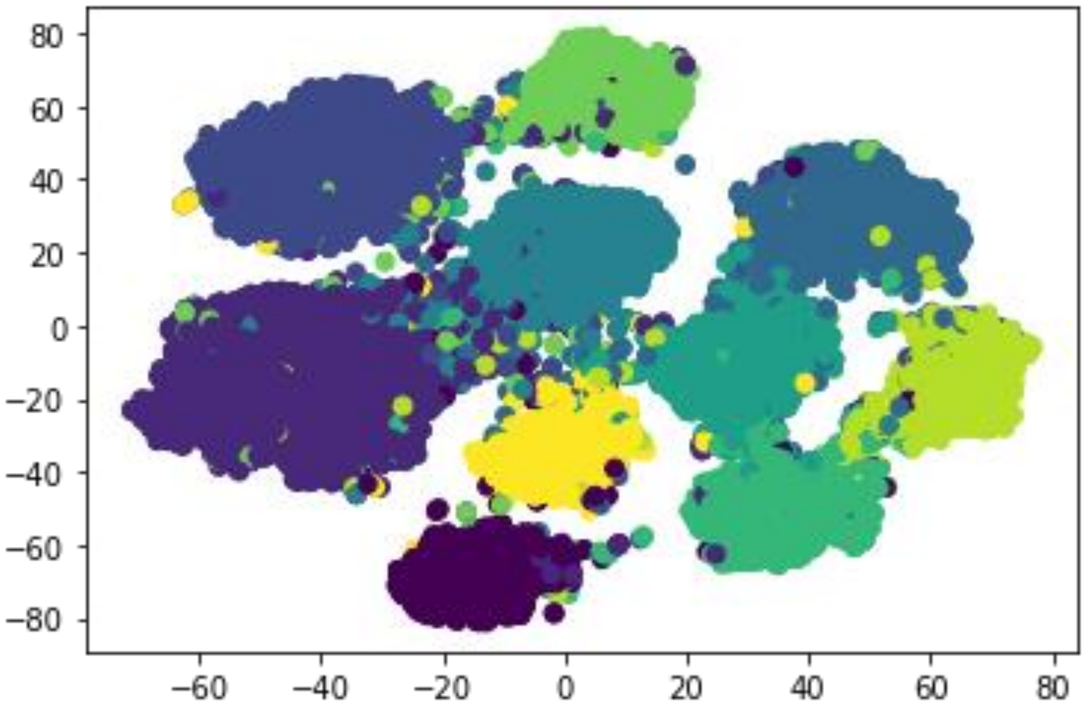
```
Accuracy :  0.9256074256074256
F1 Score :  0.80298252832351688
Confusion Matrix :
  [[ 30 110  68  52  44  53  25  25  40  33]
 [ 95 272 223 153 166 112 116  92  83  82]
 [ 73 199 144 136 103  97  81  84  78  63]
 [ 50 160 115 116  88  81  67  67  52  60]
 [ 57 150 121  78  74  67  62  64  46  35]
 [ 46 128  99  75  84  63  43  50  48  46]
 [ 40  96  78  75  54  56  42  51  26  41]
 [ 30 107  72  82  56  59  40  40  41  37]
 [ 27  96  76  75  51  44  32  37  40  34]
 [ 32  99  63  50  47  46  34  35  33  28]]
```



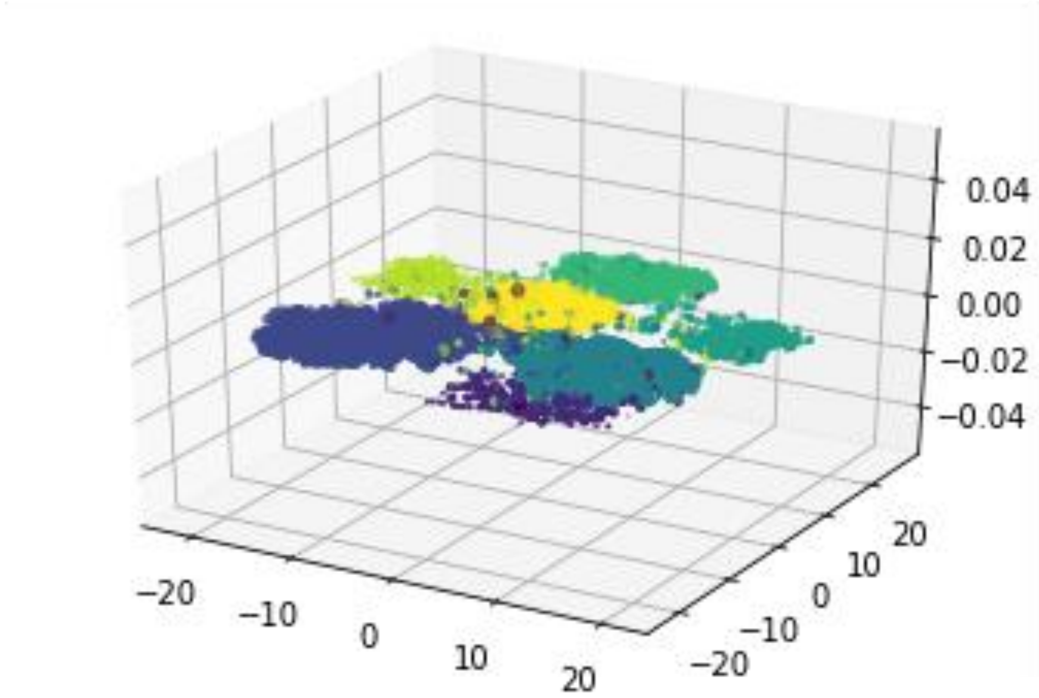
After this, we visualise TSNE on the training and validation sets in 2D space, and of the validation set in 3D space.



40
20
0
−20
−40
−60
−40
−20
0
20
40



80
60
40
20
0
−20
−40
−60
−80
−60
−40
−20
0
20
40
60
80



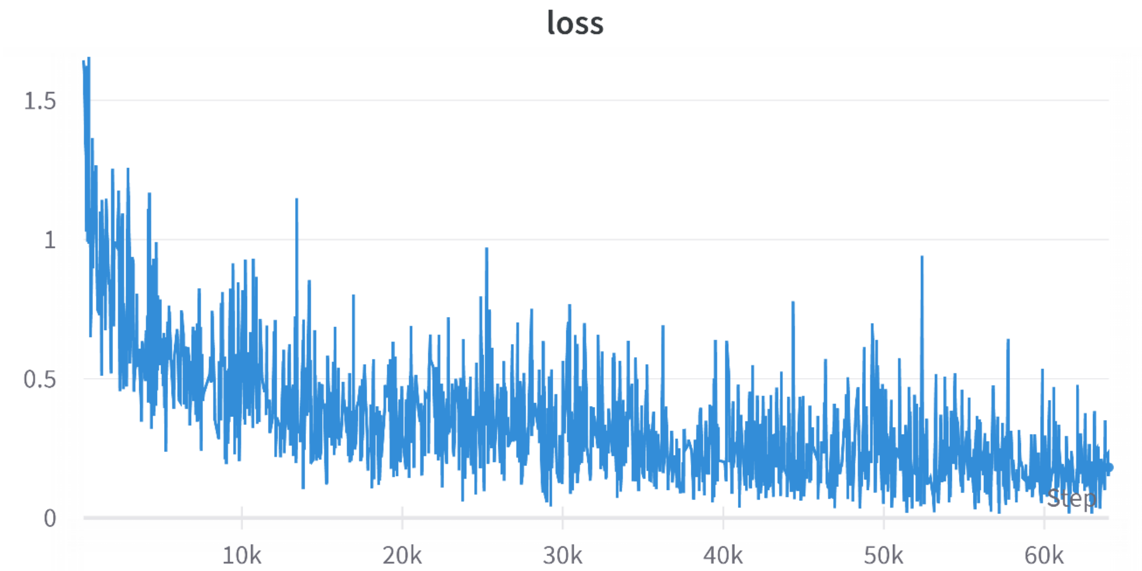
4.

We have created a function in order to perform data augmentation on the dataset.

```
import albumentations as A
transform1=A.Compose([
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.5),
    A.RandomBrightnessContrast(p=0.5),
    A.Sharpen(p=0.5)
])
```

These transformations are randomly performed on the training data and thus performs augmentation of the dataset.

After this, we train the model on the augmented data. We use the resnet 18 model again.

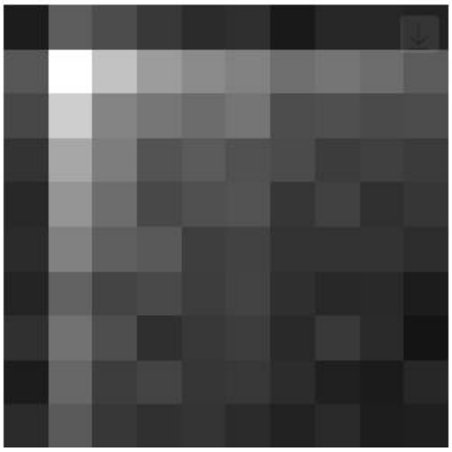



```
Epoch:  0  Loss:  1.0943912257645476
Epoch:  1  Loss:  0.7394927694361729
Epoch:  2  Loss:  0.6291500117004624
Epoch:  3  Loss:  0.5752015924516947
Epoch:  4  Loss:  0.5356643388532357
Epoch:  5  Loss:  0.4957097779845567
Epoch:  6  Loss:  0.47209971087170777
Epoch:  7  Loss:  0.4463212715411885
Epoch:  8  Loss:  0.42708434672074547
Epoch:  9  Loss:  0.41199839374960323
Epoch:  10  Loss:  0.39532545242395833
Epoch:  11  Loss:  0.3797515760677276
Epoch:  12  Loss:  0.36877202772008427
Epoch:  13  Loss:  0.35575855397389816
Epoch:  14  Loss:  0.3453312720464344
Epoch:  15  Loss:  0.335832828202669
Epoch:  16  Loss:  0.3249755284434657
Epoch:  17  Loss:  0.3148922325877086
Epoch:  18  Loss:  0.3061739420187116
Epoch:  19  Loss:  0.2975721187884928
Epoch:  20  Loss:  0.28884136302400143
Epoch:  21  Loss:  0.2818141157273494
Epoch:  22  Loss:  0.27117150543340096
Epoch:  23  Loss:  0.2640459497371504
Epoch:  24  Loss:  0.2581165056961518
Epoch:  25  Loss:  0.25169469211633994
Epoch:  26  Loss:  0.23909216125263918
Epoch:  27  Loss:  0.23277238043082074
Epoch:  28  Loss:  0.23228248675991167
Epoch:  29  Loss:  0.22302739371976124
```

```
Epoch:  30  Loss:  0.2173156779368669
Epoch:  31  Loss:  0.21230795067349822
Epoch:  32  Loss:  0.20551721849005278
Epoch:  33  Loss:  0.20255587528897212
Epoch:  34  Loss:  0.19358693820459819
Epoch:  35  Loss:  0.19250610709063715
Epoch:  36  Loss:  0.1817262490028405
Epoch:  37  Loss:  0.1782595339874346
Epoch:  38  Loss:  0.17545996316030613
Epoch:  39  Loss:  0.17001223136105587
```

We have tested the model on the testing set. The accuracy and F1 score are as follows:

```
Accuracy :  0.8633633633633634
F1 Score :  0.77754602475156112
Confusion Matrix :
  [[ 28  93  76  56  43  47  25  39  39  34]
 [ 87 259 194 156 139 130 111 117 109  92]
 [ 71 206 132 118 108 117  77  79  74  76]
 [ 51 167 125  83  91  81  75  60  64  59]
 [ 40 149 109  72  80  83  54  65  48  54]
 [ 43 129  95  89  62  66  51  51  51  45]
 [ 36  98  68  73  61  67  47  40  41  28]
 [ 48 114  78  47  56  61  41  57  42  20]
 [ 28 103  61  68  54  56  44  32  27  39]
 [ 43  90  55  48  52  43  34  42  29  31]]
```



6.

The performance of all three models is:

Accuracy:

Custom CNN Model: 85.32%

Pretrained resnet18: 92.56%

Pretrained resnet18 with augmentation: 86.33%

We can see that both the pretrained resnet 18 models have performed better than the custom cnn model we built. We can see that the accuracy has dropped a bit after performing augmentation. This may be because the model was overfitting previously and data augmentation has reduced that, which will prevent it from overfitting, but might have also caused the testing accuracy to drop a bit. Another reason might be because of an unfortunate split while splitting the data into training, validation, and testing data.

2)

First, we import all of the necessary modules. After that, we create two lists that will store the data, and then we use os.listdir() to iterate through the folder that contains the images and the masks, and then read them and append them into the list. We initialise wandb.

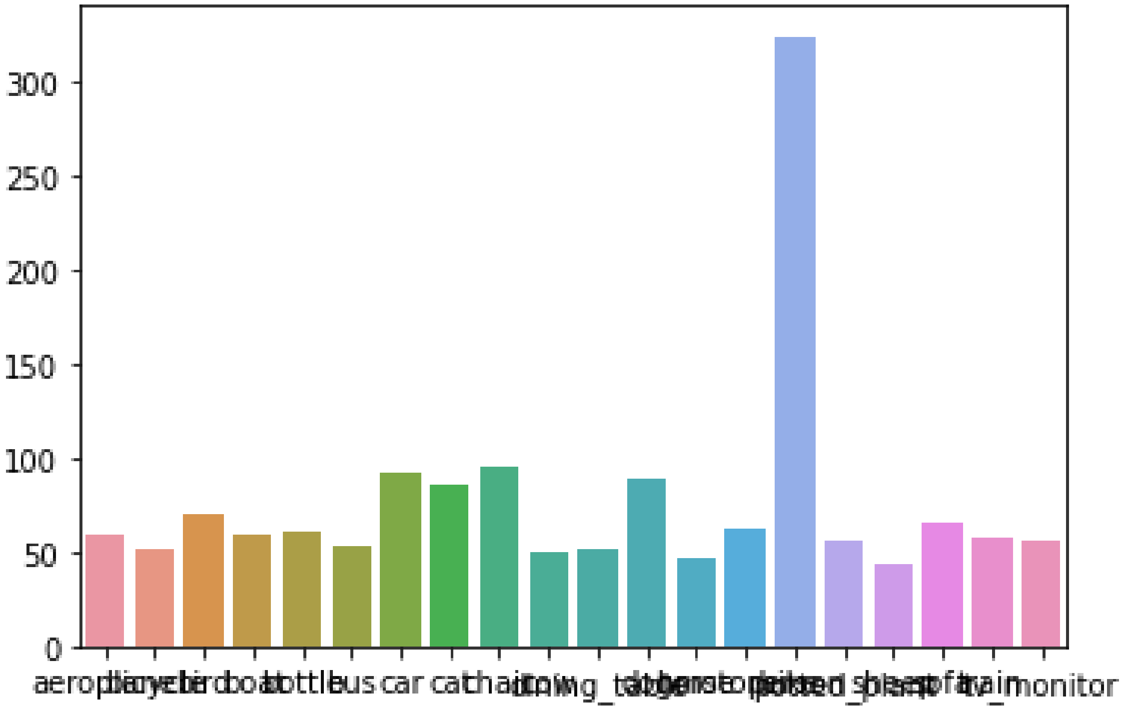
We have created a custom dataset for the data, and then create custom dataloaders for the training, validation and testing data.

```
class custom(Dataset):
    def __init__(self, image,mask,train,transform=None):
        self.transform=transform
        self.train=train
        if self.train=="train":
            self.image=image[:int(len(image)*0.7)]
            self.mask=mask[:int(len(mask)*0.7)]
        elif self.train=="val":
            self.image=image[int(len(image)*0.7):int(len(image)*0.9)]
            self.mask=mask[int(len(mask)*0.7):int(len(mask)*0.9)]
        elif self.train=="test":
            self.image=image[int(len(image)*0.9):]
            self.mask=mask[int(len(mask)*0.9):]
    def __len__(self):
        return len(self.image)
    def __getitem__(self, idx):
        img=self.image[idx]
        mask=self.mask[idx]
        if img.shape[0]<256:
            padding=np.zeros((256-img.shape[0],img.shape[1],img.shape[2]))
            img=np.concatenate((img,padding),axis=0)
            mask=np.concatenate((mask,padding),axis=0)
        if img.shape[1]<256:
            padding=np.zeros((img.shape[0],256-img.shape[1],img.shape[2]))
            img=np.concatenate((img,padding),axis=1)
```

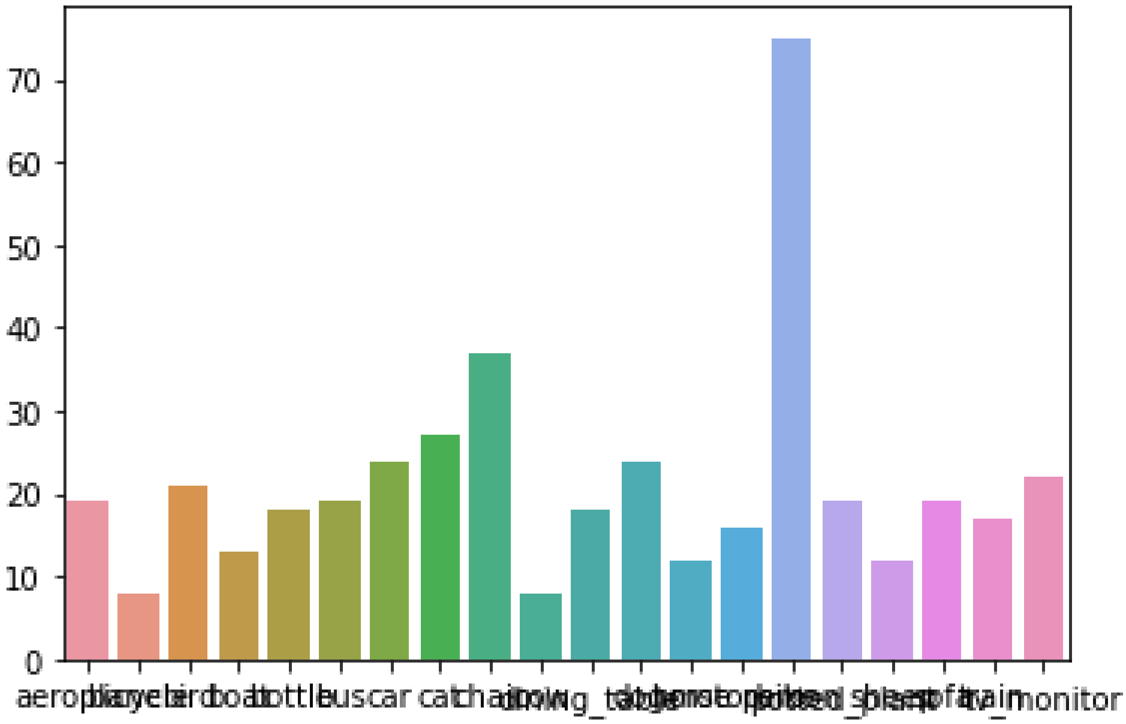
```python
        mask=np.concatenate((mask,padding),axis=1)
    height,width=img.shape[0],img.shape[1]
    upper=random.randint(0,height-256)
    left=random.randint(0,width-256)
    img=img[upper:upper+256,left:left+256]
    mask=mask[upper:upper+256,left:left+256]
    img=img.astype(np.double)
    mask=mask.astype(np.double)
    if self.transform:
        img=self.transform(img)
    lab=np.zeros((256,256))
    for i,color in enumerate(cmap):
        lab[(mask==color).all(axis=2)]=i
    return img,lab
```

```python
train_loader=DataLoader(train_custom,batch_size=4,shuffle=True)
val_loader=DataLoader(val_custom,batch_size=4,shuffle=True)
test_loader=DataLoader(test_custom,batch_size=4,shuffle=True)
```

After this, we have visualised the data distribution across class labels for the training and validation sets.




```
{'aeroplane': 59, 'bicycle': 51, 'bird': 70, 'boat': 60, 'bottle': 61, 'bus': 53, 'car': 92, 'cat': 86, 'chair':
96, 'cow': 50, 'dining_table': 51, 'dog': 89, 'horse': 47, 'motorbike': 62, 'person': 324, 'potted_plant': 56,
'sheep': 43, 'sofa': 66, 'train': 57, 'tv_monitor': 56}
```



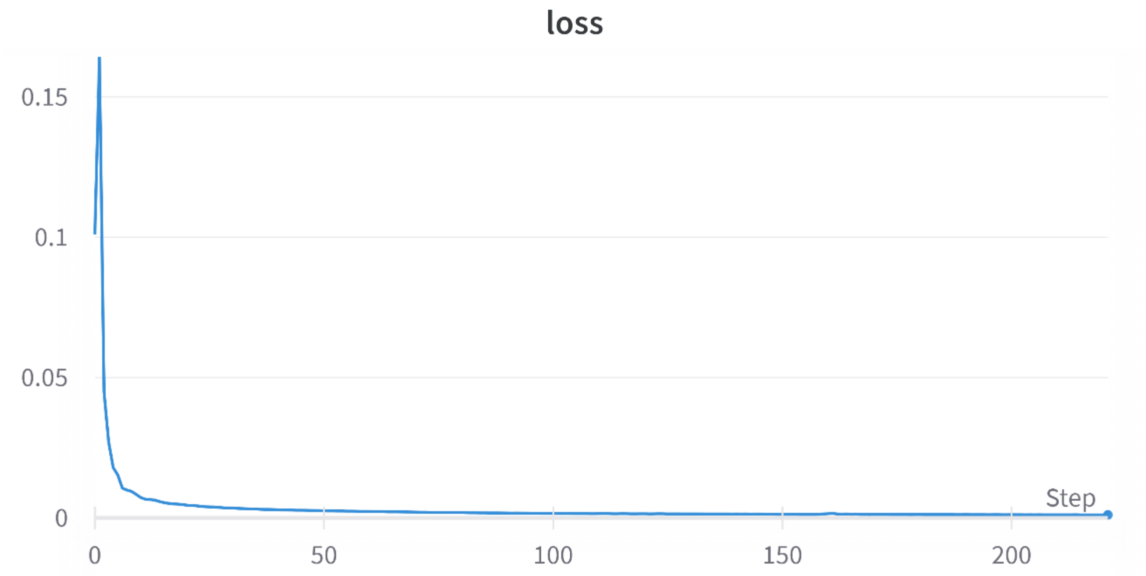

```
{'aeroplane': 19, 'bicycle': 8, 'bird': 21, 'boat': 13, 'bottle': 18, 'bus': 19, 'car': 24, 'cat': 27, 'chair':
37, 'cow': 8, 'dining_table': 18, 'dog': 24, 'horse': 12, 'motorbike': 16, 'person': 75, 'potted_plant': 19,
'sheep': 12, 'sofa': 19, 'train': 17, 'tv_monitor': 22}
```

2.

After this, we have trained a fcn_resnet50 model using pre-defined network weights.

```
model=torchvision.models.segmentation.fcn_resnet50(pretrained=True)
model=model.double()
model.train()
model=model.to("cuda")
criterion=nn.NLLLoss()
optimizer=optim.Adam(model.parameters(),lr=0.001)
wandb.watch(model,criterion,log="all",log_freq=10)
```

We have used wandb for logging the loss.




```
Epoch:  0  Loss:  0.0837076958753746
Epoch:  1  Loss:  0.0155444822342331
Epoch:  2  Loss:  0.011991728626011184
```

After this, we evaluate the performance of the model on the testing set. We report pixel wise accuracy, precision, recall, iou, F1 score, mean pixel wise accuracy and average precision.

```
Class:  background  Pixel Accuracy:  0.97248468
Precision:  0.9791269
Recall:  0.97801924
F1 Score 0.9785727565561926
IOU:  0.93472364
Class:  aeroplane  Pixel Accuracy:  0.98932576
Precision:  0.936456256
Recall:  0.93296261
F1 Score 0.93470616846614
IOU:  0.87981064
Class:  bicycle  Pixel Accuracy:  0.93410423
Precision:  0.68426865
Recall:  0.74059241
F1 Score 0.7113173106028268
IOU:  0.57467678
Class:  bird  Pixel Accuracy:  0.98592363
Precision:  0.94903613
Recall:  0.941883502
F1 Score 0.9454462881677111
IOU:  0.86704528
Class:  boat  Pixel Accuracy:  0.98928289
Precision:  0.92832337
Recall:  0.93471458
F1 Score 0.931508012377026
IOU:  0.85869137
Class:  bottle  Pixel Accuracy:  0.98284626
Precision:  0.93725125
Recall:  0.9255691
F1 Score 0.9313735443531901
IOU:  0.88924568
```

```
Class:  bus  Pixel Accuracy:  0.98426413
Precision:  0.98131566
Recall:  0.99156374
F1 Score 0.9864130832834166
IOU:  0.97924048
Class:  car  Pixel Accuracy:  0.97354311
Precision:  0.9807132
Recall:  0.846692366
F1 Score 0.9087882789949115
IOU:  0.78699581
Class:  cat  Pixel Accuracy:  0.98361513
Precision:  0.93452865
Recall:  0.93607512
F1 Score 0.9353012457493208
IOU:  0.8809525
Class:  chair  Pixel Accuracy:  0.98316463
Precision:  0.91502453
Recall:  0.77883897
F1 Score 0.8414571333167449
IOU:  0.72424492
Class:  cow  Pixel Accuracy:  0.97031378
Precision:  0.94228405
Recall:  0.84592366
F1 Score 0.8915075892784547
IOU:  0.79143564
Class:  dining_table  Pixel Accuracy:  0.98216527
Precision:  0.892543
Recall:  0.84671856
F1 Score 0.869027110215763
IOU:  0.76912532
```

```
Class:  dog  Pixel Accuracy:  0.98115414
Precision:  0.98012753
Recall:  0.9064618
F1 Score 0.9418564506281333
IOU:  0.87973
Class:  horse  Pixel Accuracy:  0.98355221
Precision:  0.946024535
Recall:  0.90149331
F1 Score 0.923222248387388
IOU:  0.8426232
Class:  motorbike  Pixel Accuracy:  0.9702693
Precision:  0.874591233
Recall:  0.89886391
F1 Score 0.8865614655651891
IOU:  0.78282684
Class:  person  Pixel Accuracy:  0.97258342
Precision:  0.90348021
Recall:  0.89531428
F1 Score 0.8993787096939558
IOU:  0.84655752
Class:  potted_plant  Pixel Accuracy:  0.96808413
Precision:  0.80534195
Recall:  0.8088138
F1 Score 0.8070741412393568
IOU:  0.66278624
Class:  sheep  Pixel Accuracy:  0.96457274
Precision:  0.96862342
Recall:  0.9802144
F1 Score 0.9743844405290205
IOU:  0.93575234
```

```
Class:  sofa  Pixel Accuracy:  0.9846626
Precision:  0.952825234
Recall:  0.84364552
F1 Score 0.894917702631358
IOU:  0.79245801
Class:  train  Pixel Accuracy:  0.99427825
Precision:  0.98923514
Recall:  0.93652524
F1 Score 0.9621588298591268
IOU:  0.94727336
Class:  tv_monitor  Pixel Accuracy:  0.98254322
Precision:  0.89122587
Recall:  0.93531345
F1 Score 0.9127375842080985
IOU:  0.84620824
Mean Pixel Accuracy:  0.9777492147619049
Average Precision:  0.9224927032380952
```

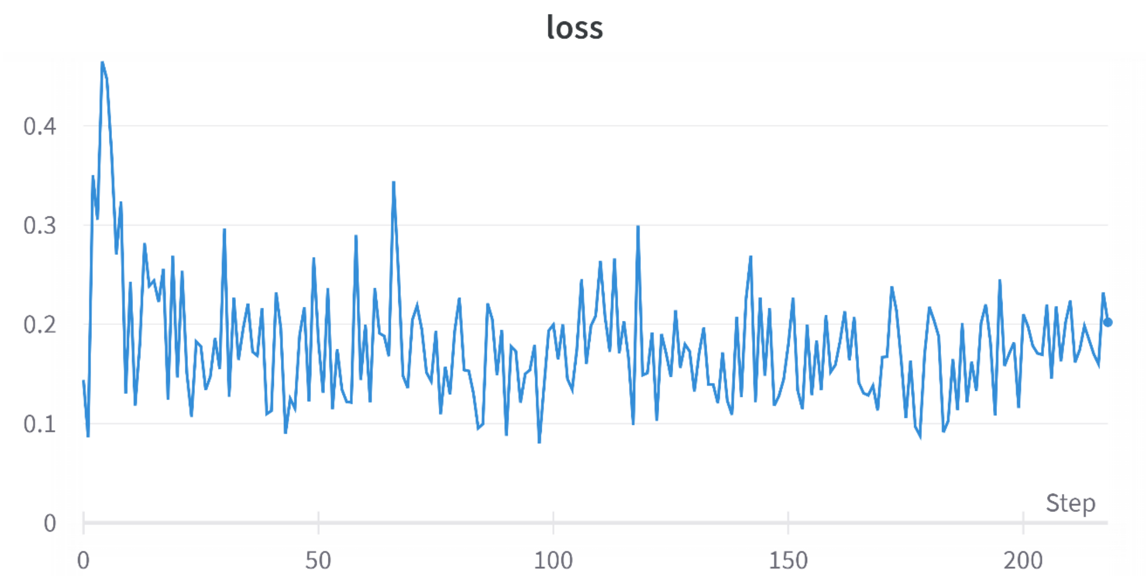
3.

We have used data augmentation techniques that are suitable for the given problem. We have used a horizontal flip and rotation data augmentation and altered the code of the custom dataset such that data augmentation is included.

```
if self.train=="train":
    randr=random.random()
    randg=random.random()
    if randr>=0.5:
        img=cv2.flip(img,1)
        mask=cv2.flip(mask,1)
    if randg>=0.5:
        ang=random.randint(-10,10)
        height,width=img.shape[0],img.shape[1]
        rotn=cv2.getRotationMatrix2D((width/2,height/2),ang,1)
        img=cv2.warpAffine(img,rotn,(width,height))
        mask=cv2.warpAffine(mask,rotn,(width,height))
```

After this, we have trained our fcn_resnet50 model on the augmented data.

```
Epoch:  0  Loss:  0.19637889024190591
Epoch:  1  Loss:  0.1660757751903744
Epoch:  2  Loss:  0.16600251999396214
```



After this, we evaluate the performance of the model on the testing set. We report pixel wise accuracy, precision, recall, iou, F1 score, mean pixel wise accuracy and average precision.

```
Class:  background
Pixel Accuracy:  0.93874191
Precision:  0.97319174
Recall:  0.96702663
F1 Score 0.9700993900764234
IOU:  0.93672047
Class:  aeroplane
Pixel Accuracy:  0.94131644
Precision:  0.94183562
Recall:  0.81752891
F1 Score 0.8752908617724312
IOU:  0.74724368
Class:  bicycle
Pixel Accuracy:  0.77944105
Precision:  0.75982341
Recall:  0.95215905
F1 Score 0.8451870893973534
IOU:  0.701024289
Class:  bird
Pixel Accuracy:  0.96861405
Precision:  0.90574608
Recall:  0.8771843
F1 Score 0.8912364162671836
IOU:  0.79372144
Class:  boat
Pixel Accuracy:  0.96182681
Precision:  0.90313571
Recall:  0.85205136
F1 Score 0.876850135376243
IOU:  0.78689414
```

```
Class:  bottle
Pixel Accuracy:  0.97814234
Precision:  0.94209374
Recall:  0.95355611
F1 Score 0.94779027041281
IOU:  0.88786517
Class:  bus
Pixel Accuracy:  0.9583311
Precision:  0.9621419
Recall:  0.96614589
F1 Score 0.9641397379607853
IOU:  0.91487375
Class:  car
Pixel Accuracy:  0.97224179
Precision:  0.93619322
Recall:  0.9162362
F1 Score 0.9261072072139342
IOU:  0.84225784
Class:  cat
Pixel Accuracy:  0.95824651
Precision:  0.88919543
Recall:  0.88925512
F1 Score 0.8892252739983145
IOU:  0.80641547
Class:  chair
Pixel Accuracy:  0.9125164
Precision:  0.9322414
Recall:  0.86855783
F1 Score 0.8992735602404295
IOU:  0.81010478
```

```
Class:  cow
Pixel Accuracy:  0.942504174
Precision:  0.94294674
Recall:  0.86761436
F1 Score 0.9037133652536623
IOU:  0.82146316
Class:  dining_table
Pixel Accuracy:  0.91404132
Precision:  0.87213155
Recall:  0.8459352
F1 Score 0.8588336596067178
IOU:  0.73261413
Class:  dog
Pixel Accuracy:  0.9656416
Precision:  0.95705488
Recall:  0.93443591
F1 Score 0.9456101530504845
IOU:  0.88751274
Class:  horse
Pixel Accuracy:  0.91343702
Precision:  0.92162245
Recall:  0.872193497
F1 Score 0.8962269612147757
IOU:  0.80573168
Class:  motorbike
Pixel Accuracy:  0.93697315
Precision:  0.83262618
Recall:  0.9203189
F1 Score 0.874279084760376
IOU:  0.75106411
```

```
Class:  person
Pixel Accuracy:  0.96824389
Precision:  0.91663511
Recall:  0.93272516
F1 Score 0.9246101406043157
IOU:  0.81610314
Class:  potted_plant
Pixel Accuracy:  0.98633103
Precision:  0.83130135
Recall:  0.97394585
F1 Score 0.8969879581388049
IOU:  0.79341341
Class:  sheep
Pixel Accuracy:  0.95134232
Precision:  0.91725196
Recall:  0.912352
F1 Score 0.9147954185778217
IOU:  0.80528426
Class:  sofa
Pixel Accuracy:  0.95662493
Precision:  0.8789512
Recall:  0.87612344
F1 Score 0.8775350419696429
IOU:  0.75144172
Class:  train
Pixel Accuracy:  0.9652354
Precision:  0.96452056
Recall:  0.878482665
F1 Score 0.9194933362051957
IOU:  0.85641319
```

```
Class:  tv_monitor
Pixel Accuracy:  0.93671324
Precision:  0.935245741
Recall:  0.84059827
F1 Score 0.8853997840348242
IOU:  0.78253231
Mean Pixel Accuracy:  0.9431669749523809
Average Precision:  0.9102802843333334
```

4.

We can see that the average precision of the first model is 92.24 % and the average precision of the second model is 91.02 %.

We can see that the accuracy has dropped a bit after performing augmentation. This may be because the model was overfitting previously and data augmentation has reduced that, which will prevent it from overfitting, but might have also caused the testing accuracy to drop a bit. Another reason might be because of an unfortunate split while splitting the data into training, validation, and testing data.

3)

1.

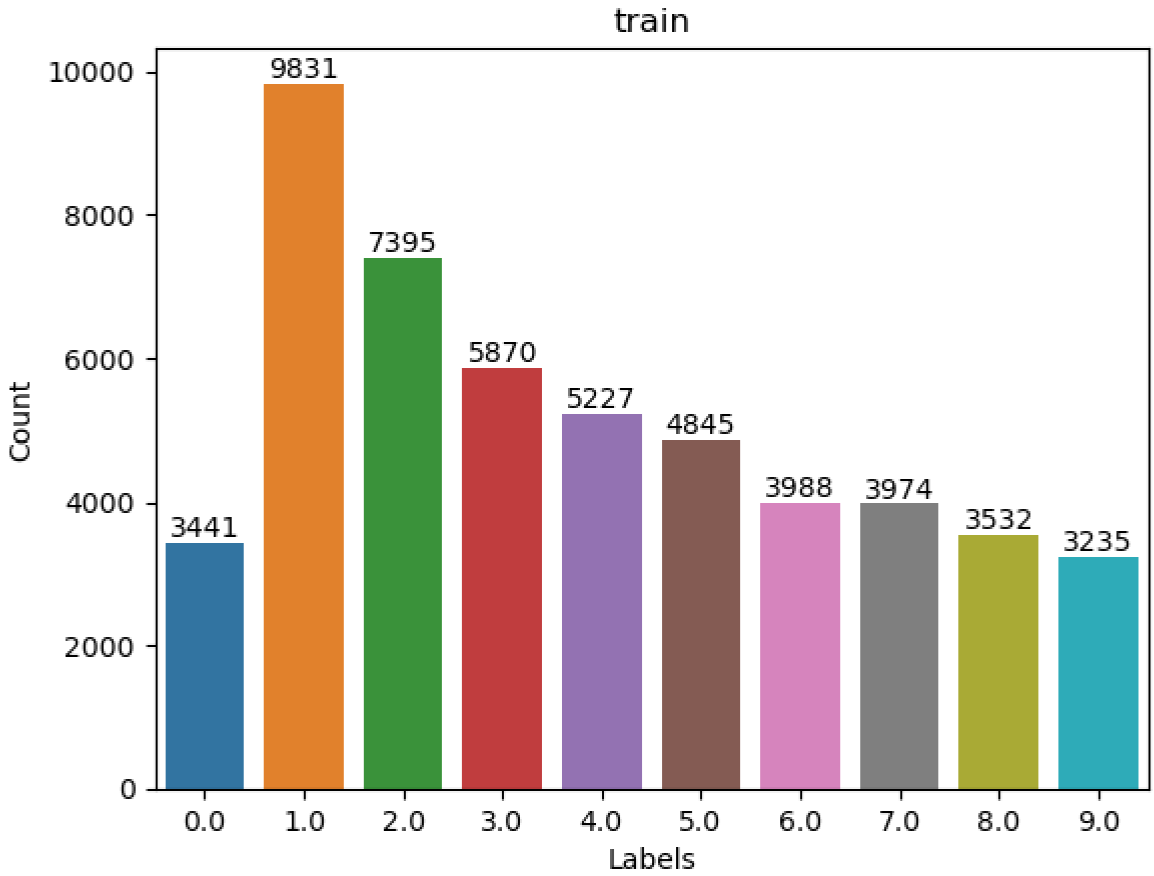
Explanation of Code:

We have downloaded the dataset (SVHN dataset in YOLO format) and split it into training, validation, and testing sets in the ratio of 70:20:10. We have also initialised Weights and Biases. We have created data loaders for training, validation, and testing sets.

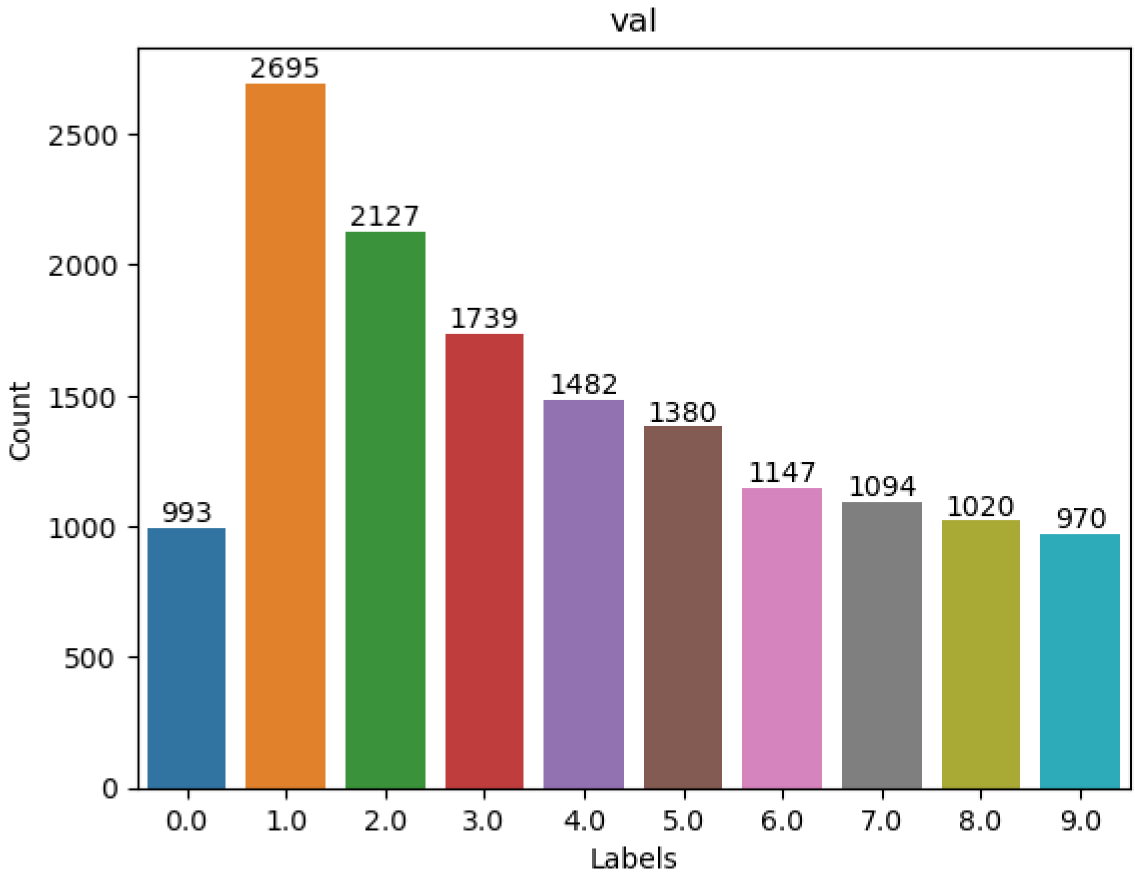
After this, we have plotted bar graphs of the different numbers present in each image. We have done this by iterating through all the labels, and adding their frequency to a dictionary and then we have plotted the dictionary.

Graphs:

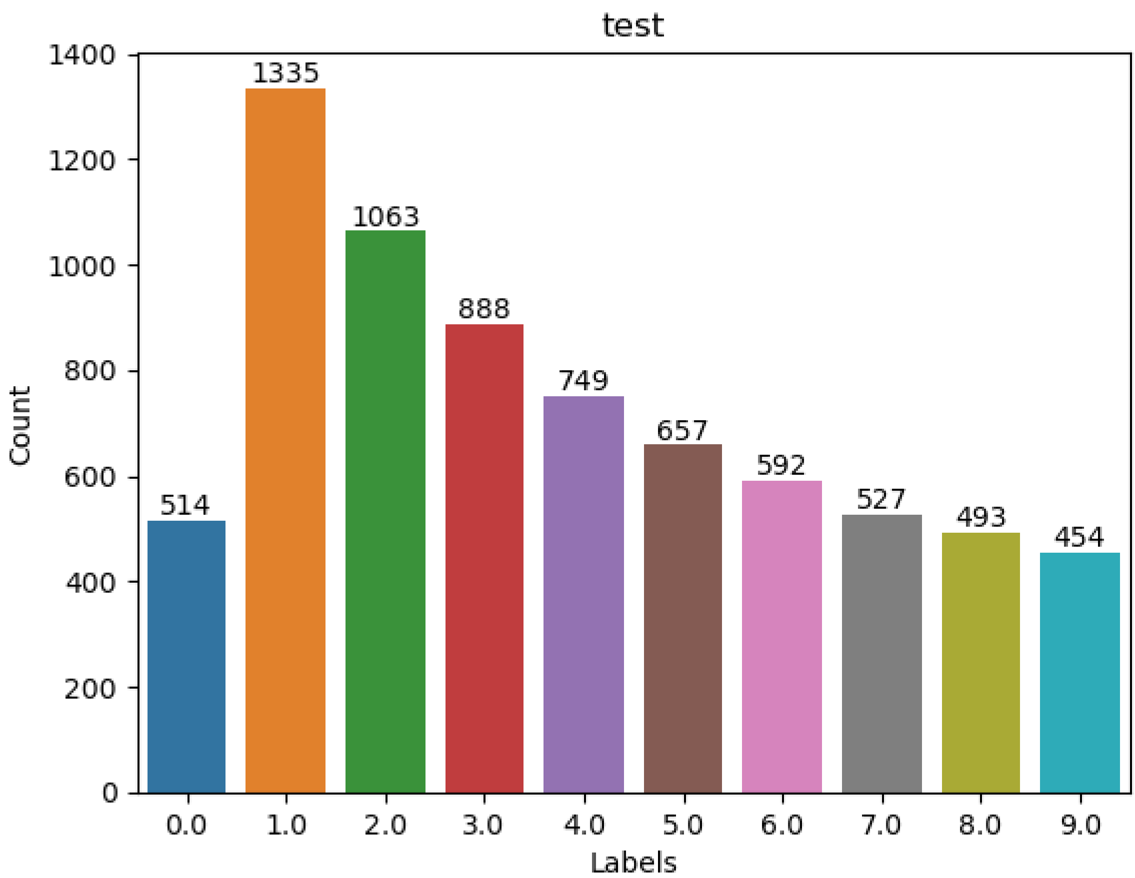
{2.0: 7395, 5.0: 4845, 3.0: 5870, 1.0: 9831, 9.0: 3235, 6.0: 3988, 0.0: 3441, 7.0: 3974, 4.0: 5227, 8.0: 3532}



{2.0: 2127, 0.0: 993, 4.0: 1482, 8.0: 1020, 9.0: 970, 5.0: 1380, 3.0: 1739, 1.0: 2695, 6.0: 1147, 7.0: 1094}



{2.0: 1063, 1.0: 1335, 3.0: 888, 5.0: 657, 8.0: 493, 9.0: 454, 0.0: 514, 7.0: 527, 6.0: 592, 4.0: 749}



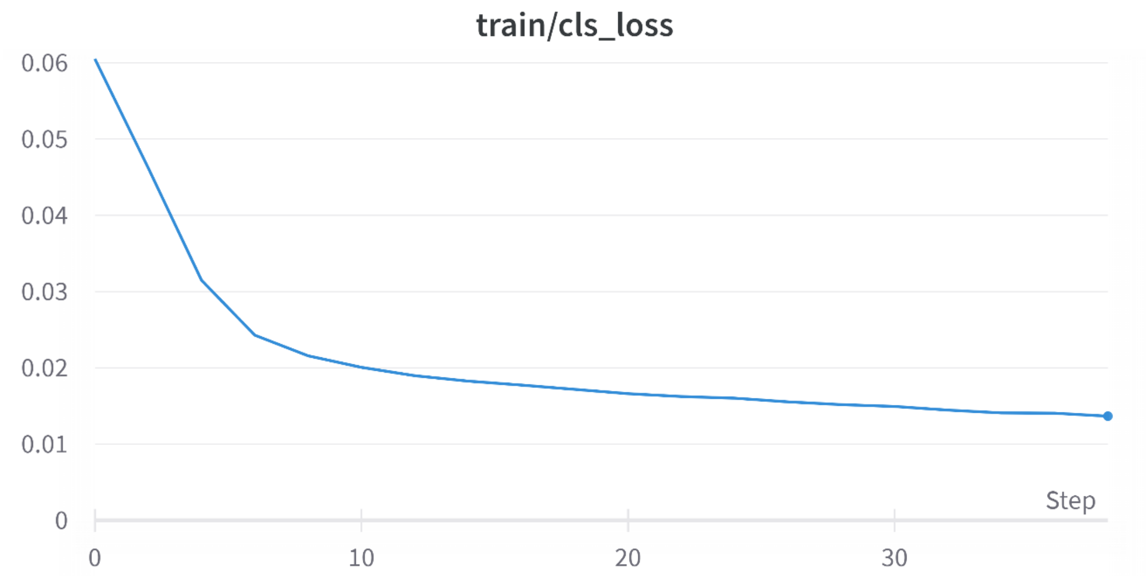
2.

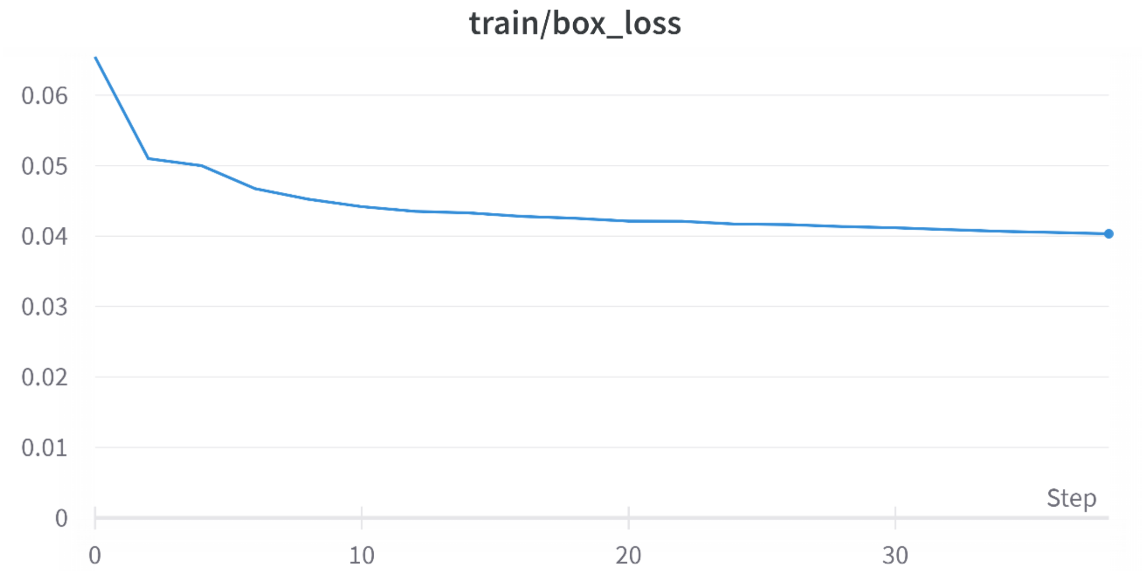
We have trained the yolo model successfully. We have used the yolov5n model. We have trained it for 20 epochs with a batch size of 32. The results are as follows:

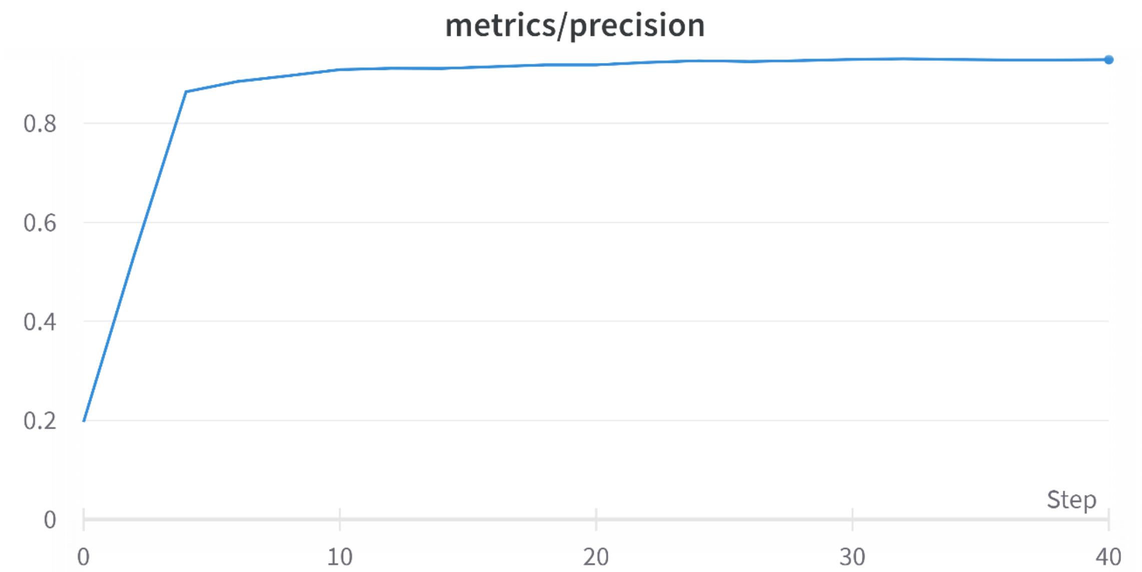
Model summary: 157 layers, 1772695 parameters, 0 gradients, 4.2 GFLOPs

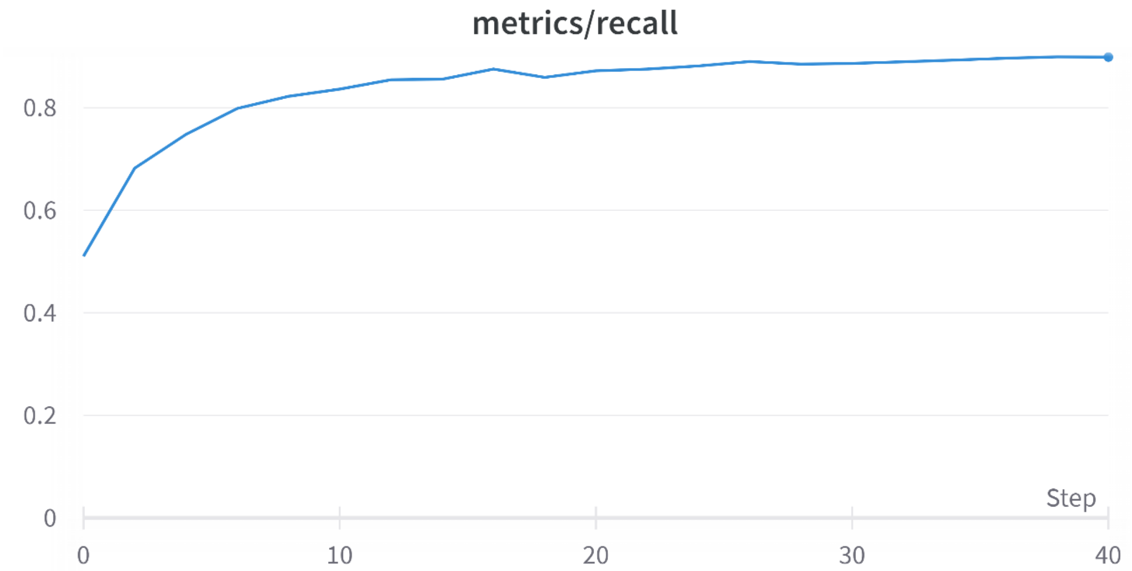
Class	Images	Instances	P	R	mAP50	mAP50-95:
all	6679	14645	0.929	0.899	0.935	0.49
0	6679	992	0.926	0.911	0.935	0.499
1	6679	2695	0.897	0.875	0.894	0.389
2	6679	2127	0.946	0.927	0.955	0.515
3	6679	1739	0.935	0.883	0.938	0.502
4	6679	1481	0.943	0.905	0.947	0.497
5	6679	1380	0.935	0.903	0.941	0.503
6	6679	1147	0.925	0.88	0.93	0.501
7	6679	1094	0.934	0.896	0.93	0.478
8	6679	1020	0.927	0.908	0.945	0.52
9	6679	970	0.919	0.902	0.936	0.491

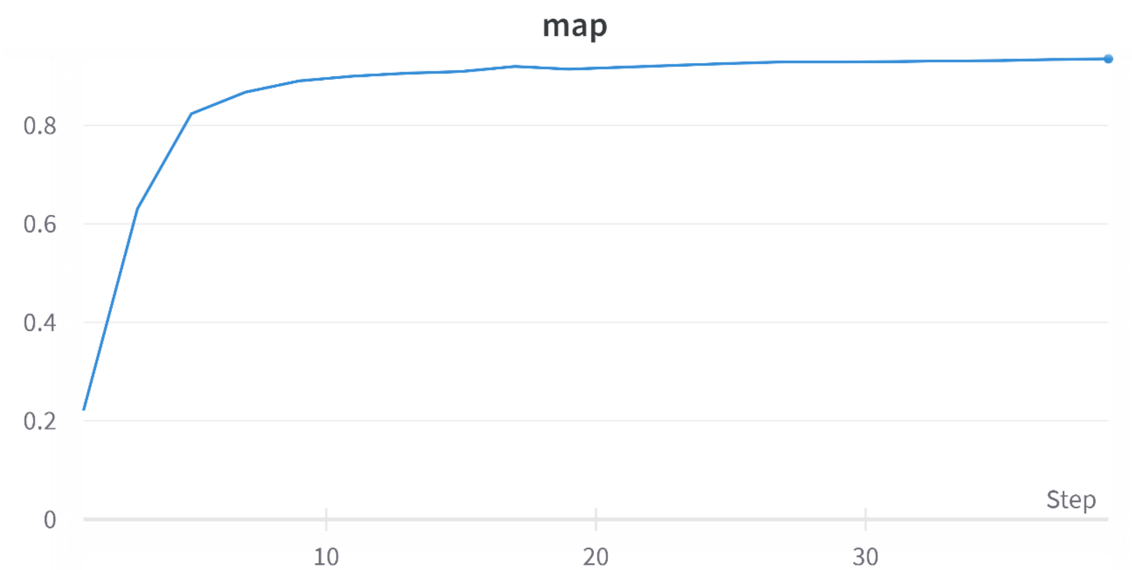

train/obj_loss
0.026
0.025
0.024
0.023
0.022
0.021
Step
0
10
20
30


train/cls_loss
0.06
0.05
0.04
0.03
0.02
0.01
0
Step
0
10
20
30

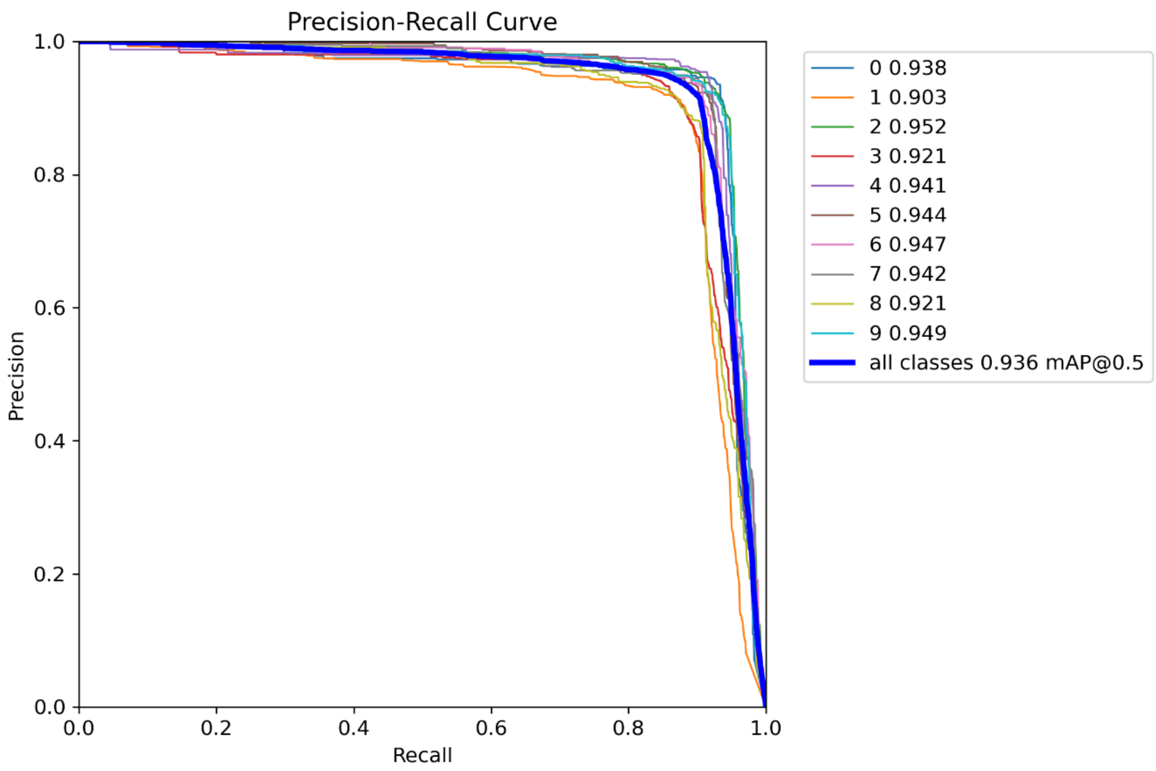

train/box_loss
0.06
0.05
0.04
0.03
0.02
0.01
0
Step
0
10
20
30

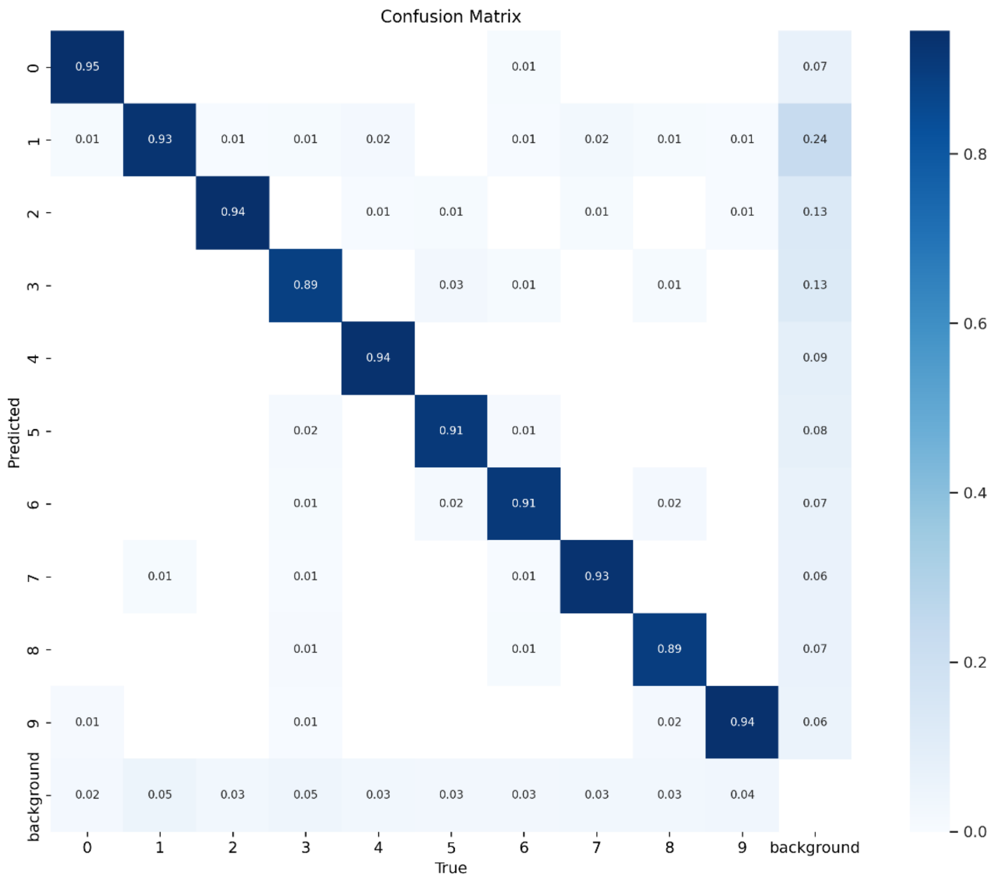

metrics/precision
0.8
0.6
0.4
0.2
0
Step
0
10
20
30
40





After this, we test the performance of the test set. We get the results as:





Classwise IOU's:

Class	IOU

0	0.81956
1	0. 76576
2	0. 82634
3	0. 7531
4	0. 82201
5	0. 78068
6	0. 7789
7	0. 80098
8	0. 76034
9	0. 78453

The mean IOU is: 0.78922060304692363

The classwise average precision is:

Classes	Average Precision
0	0. 93835
1	0. 90346
2	0. 95193
3	0. 92078
4	0. 9408
5	0. 94359
6	0. 9472
7	0. 942
8	0. 92091
9	0. 94882

The mean average precision is: 0.9357854956766183

c)

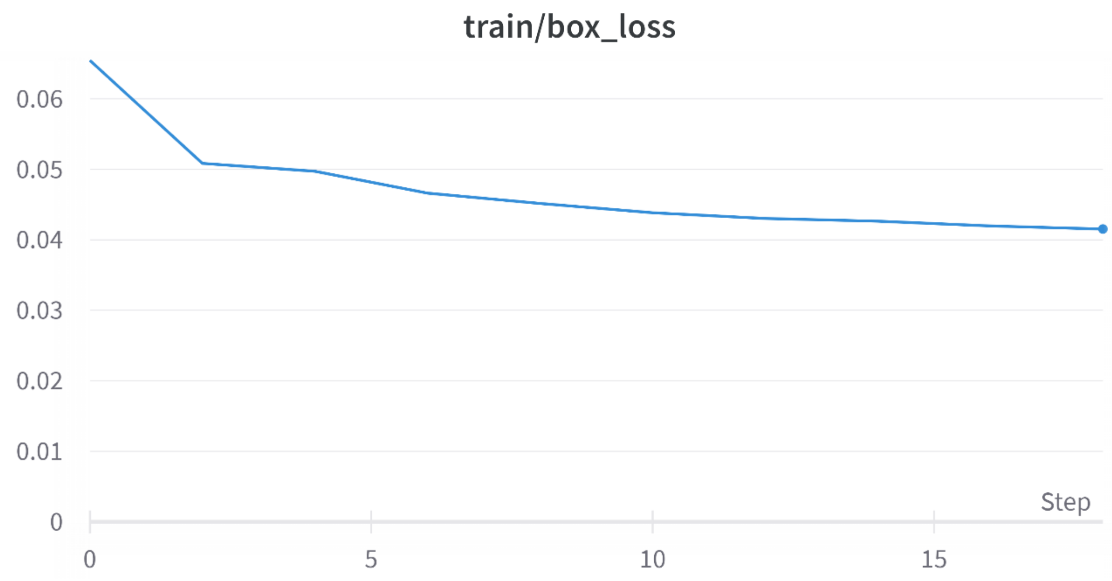
We apply data augmentation by altering the custom dataset class we made. We have applied flip, sharpen, and contrast transformations.

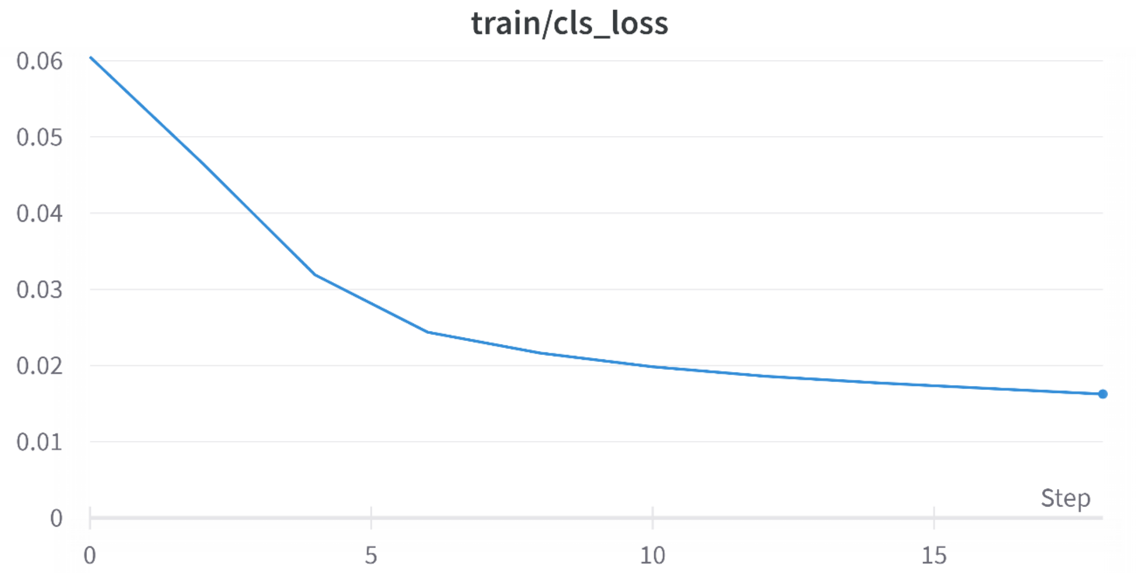
```
r=random.randint(0,1)
if r==0:
    img=cv2.flip(img,1)
    for i in lab:
        i[0]=1-i[0]
r=random.randint(0,1)
if r==0:
    kernel=np.array([[-1,-1,-1], [-1,9,-1], [-1,-1,-1]])
    img=cv2.filter2D(img,-1,kernel)
r=random.randint(0,1)
if r==0:
    img=cv2.cvtColor(img,cv2.COLOR_BGR2HSV)
    img=cv2.equalizeHist(img[:,:,2])
```

We have run the model after applying data augmentation. We have trained it for 20 epochs with a batch size of 32.

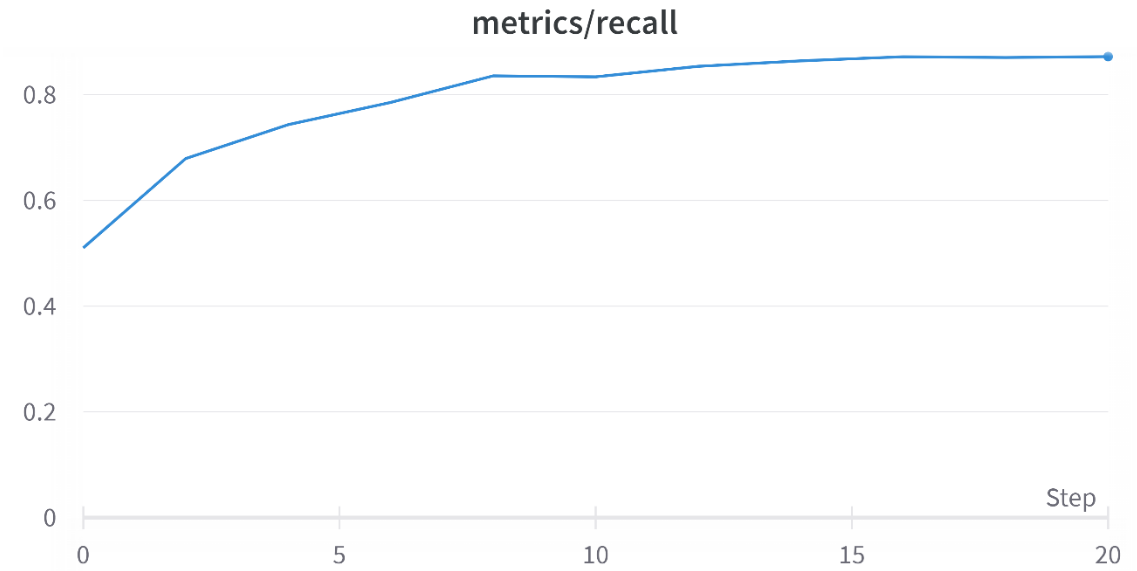
Results:

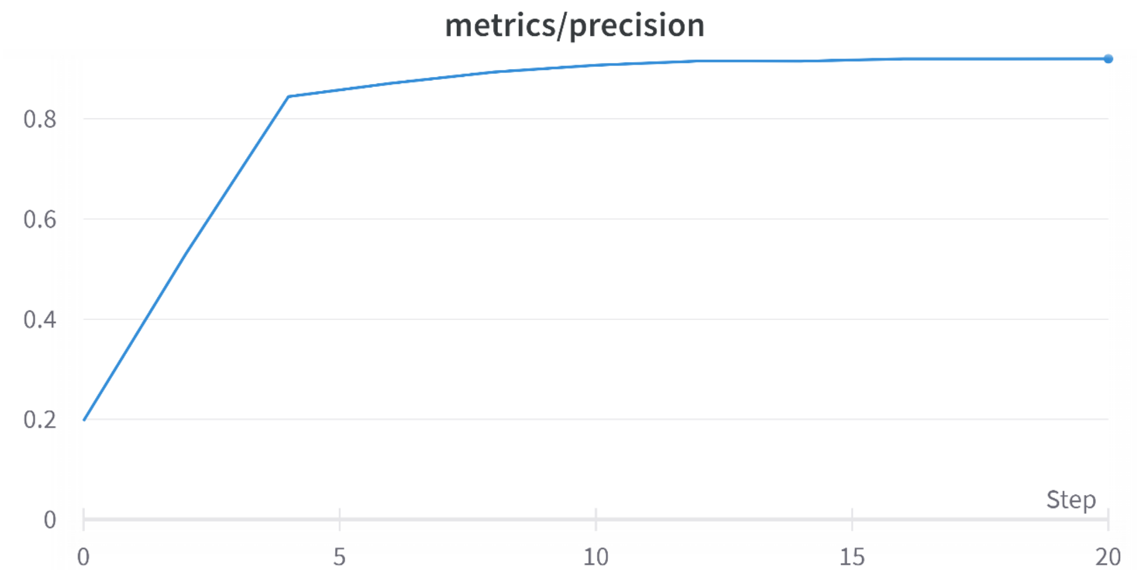
Class	Images	Instances	P	R	mAP50	mAP50-95:
all	6679	14645	0.92	0.872	0.919	0.472
0	6679	992	0.916	0.899	0.927	0.487
1	6679	2695	0.901	0.841	0.881	0.374
2	6679	2127	0.943	0.903	0.945	0.499
3	6679	1739	0.897	0.858	0.915	0.483
4	6679	1481	0.936	0.879	0.929	0.479
5	6679	1380	0.935	0.884	0.929	0.48
6	6679	1147	0.9	0.853	0.912	0.486
7	6679	1094	0.924	0.873	0.917	0.456
8	6679	1020	0.929	0.867	0.927	0.498
9	6679	970	0.92	0.862	0.912	0.481

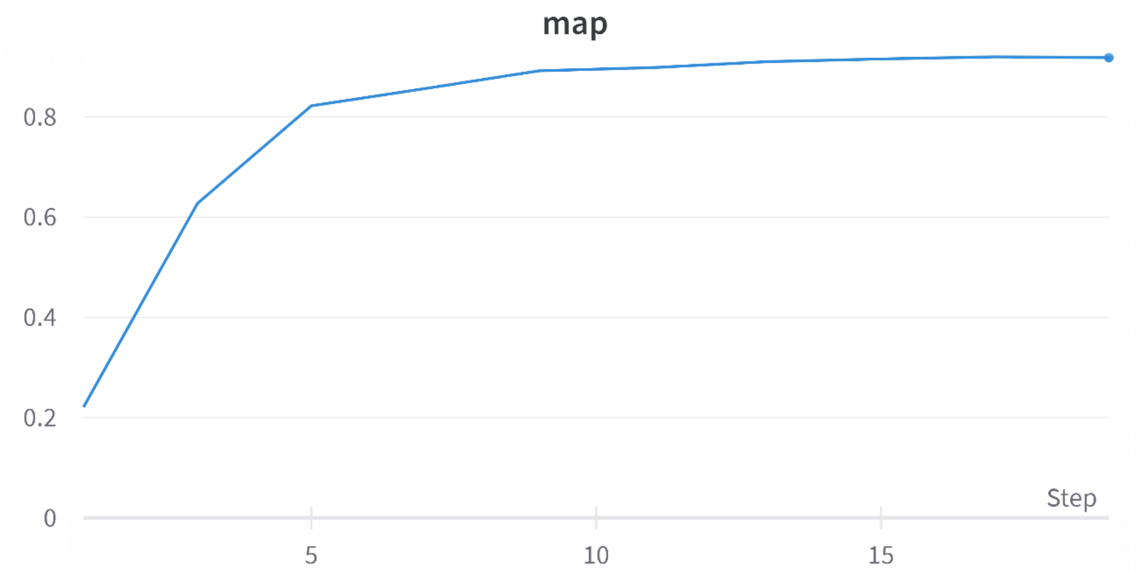


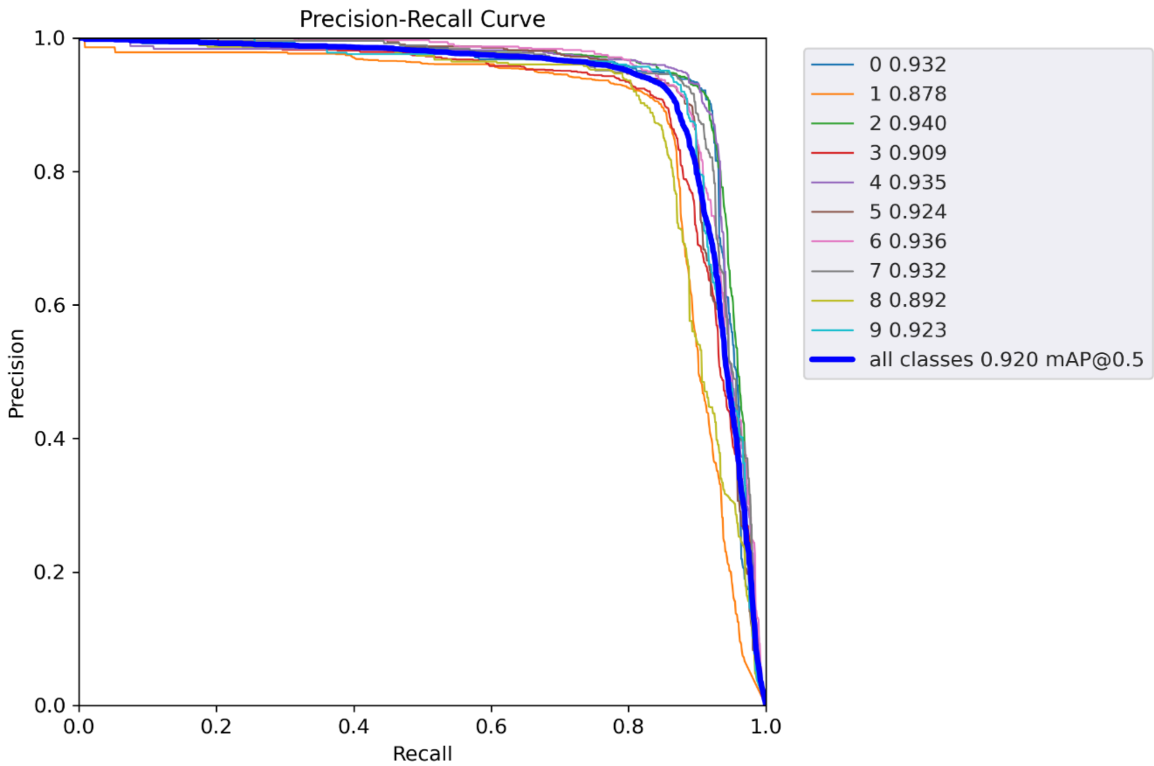

train/cls_loss
0.06
0.05
0.04
0.03
0.02
0.01
0
Step
0
5
10
15

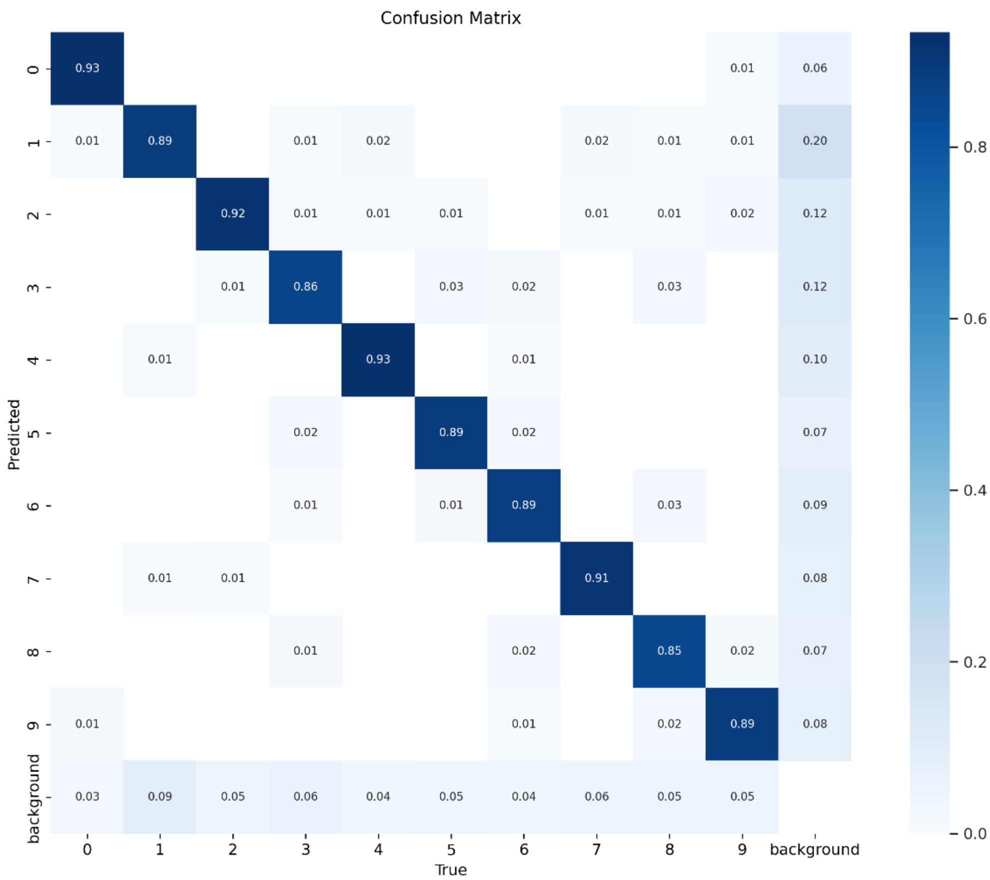

train/obj_loss
0.026
0.025
0.024
0.023
0.022
0.021
Step
0
5
10
15


metrics/recall
0.8
0.6
0.4
0.2
0
Step
0
5
10
15
20


metrics/precision
0.8
0.6
0.4
0.2
0
Step
0
5
10
15
20


map
0.8
0.6
0.4
0.2
0
Step
5
10
15


Precision-Recall Curve
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
0 0.932
1 0.878
2 0.940
3 0.909
4 0.935
5 0.924
6 0.936
7 0.932
8 0.892
9 0.923
all classes 0.920 mAP@0.5



After this, we test the performance of the test set. We get the results as:

Classwise IOU's:

Class	IOU
0	0. 80052
1	0. 75342
2	0. 81512
3	0. 73203
4	0. 77215
5	0. 75228
6	0. 7293
7	0. 75597
8	0. 73693
9	0. 70273

The mean IOU is: 0.75504597344967681

The classwise average precision is:

Classes	Average Precision
0	0. 92685
1	0. 88078
2	0. 88078
3	0. 91504

4	0. 92878
5	0. 92891
6	0. 91224
7	0. 91715
8	0. 92749
9	0. 91249

The mean average precision is: 0.9194264112970254

4.

We can see that the average precision of the first model is 93.578% and the average precision of the second model is 91.942 %.

We can see that the accuracy has dropped a bit after performing augmentation. This may be because the model was overfitting previously and data augmentation has reduced that, which will prevent it from overfitting, but might have also caused the testing accuracy to drop a bit. Another reason might be because of an unfortunate split while splitting the data into training, validation, and testing data.