Hacettepe University

Department of Computer Engineering

BBM418 - Computer Vision Laboratory

# Programming Assignment 2

MELTEM TOKGÖZ
21527381
meltemtokgoz@cs.hacettepe.edu.tr

May 2019

# Contents

# 1 Introduction

In this assignment, I apply a further image classification method. I intend to learn how to do a transfer learning using CNNs and how a network is trained for a particular problem.

Transfer learning is a popular method in computer vision because it allows us to build accurate models in a timesaving way. With transfer learning, instead of starting the learning process from scratch, you start from patterns that have been learned when solving a different problem. This way you leverage previous learnings and avoid starting from scratch.

In computer vision, transfer learning is usually expressed through the use of pre-trained models. A pre-trained model is a model that was trained on a large benchmark dataset to solve a problem similar to the one that we want to solve.I used a pre-trained VGG-16 model on ImageNet for transfer learning in this assignment.

VGG-16 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network is 16 layers deep and can classify images into 1000 object categories.

# 2 Dataset Operations

Figure 1 shows the libraries I used for this assignment. I chose to use the torch library for this assignment.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
import itertools
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsRestClassifier
plt.ion()
```

Figure 1: Library

When we perform transfer learning, we have to shape our input data into the shape that the pre-trained model expects. VGG16 expects 224 -dim square images as input and so, we resize each image to fit this mold.We also normalize the pictures.ToTensor used to convert the numpy images to torch images (we need to swap axes). These operations are shown in Figure 2.

```
[2]  data_transforms = {
        'train': transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'validation': transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'test': transforms.Compose([
                transforms.Resize(256),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
    }
```

Figure 2: Data Transform

The dataset has 10 scene categories and it is splitted into train, validation and test sets.For each category you have 400 training, 25 validation and 25 test images.We get the data from our data set here with data-load.

```
directory = '/content/drive/My Drive/dataset'
i_dataset = {x: datasets.ImageFolder(os.path.join(directory, x),data_transforms[x])for x in ['train', 'validation', 'test']}

data_load = {x: torch.utils.data.DataLoader(i_dataset[x], batch_size=64, shuffle=True, num_workers=4)
              for x in ['train', 'validation' ,'test']}

data_size = {x: len(i_dataset[x]) for x in ['train', 'validation','test']}

print({x: len(i_dataset[x]) for x in ['train', 'validation','test']})
class_label = i_dataset['train'].classes
print(class_label)

device = "cuda:0" if torch.cuda.is_available() else "cpu"
print(device)
```

Figure 3: Dataset Loader

The output of the code in Figure 3 is shown in Figure 4.

```
{'train': 4000, 'validation': 250, 'test': 250}
['airport_inside', 'bar', 'bedroom', 'casino', 'inside_subway', 'kitchen', 'livingroom', 'restaurant', 'subway', 'warehouse']
cuda:0
```

Figure 4: Dataset Information

# 3 Part 1

Here we use the 1x4096 dimension vectors that we received after the fc7 layer of the vgg16 model. We make classification with the svm algorithm to these vectors.

## 3.1 Background Information

The idea here is that we keep all the convolutional layers, but replace the final fully-connected layer with our own classifier. This way we can use VGGNet as a fixed feature extractor for our images then easily train a simple classifier on top of that.I used the support vector machine algorithm as a classifier here.

In feature extraction, we start with a pretrained model and only update the final layer weights from which we derive predictions. It is called feature extraction because we use the pretrained CNN as a fixed feature-extractor, and only change the output layer.

To define a model for training i'll follow these steps:I loaded in a pre-trained VGG16 model. Freezed all the layers before training the network, except the FC layers. I removed the last layer. I replaced the last layer with a linear classifier of our own.I made classification using the svm algorithm after feature extraction.

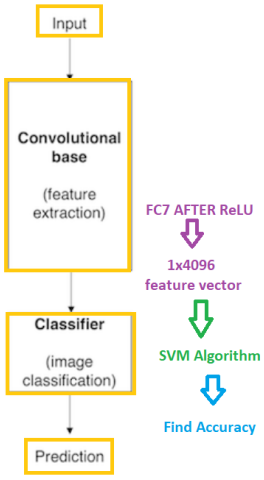



Figure 5: Part1 Implementation

## 3.2 Implementation Details

Firstly, I loaded the pretrained model from pytorch. Here, i need to freeze all the network except the FC layer.(Freezed layers act as feature vector extrac-

tor)We need to set requires-grad == False to freeze the parameters so that the gradients are not computed in backward().I then delete the fc8 layer from the model.Because it is the layer of classification.I don't want to do classification just want to get the feature vector.Figure 7 shows the layers of the model I used.

```
#Create Model
vgg16_model = models.vgg16(pretrained=True)

#Freeze model weights
for param in vgg16_model.features.parameters():
    param.requires_grad = False

#Delete fc8 layer
vgg16_model.classifier = nn.Sequential(*list(vgg16_model.classifier.children())[:-2])
#print(vgg16_model)
```

Figure 6: Create Model

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
  )
)
```

Figure 7: Model Layers

I used evaluation mode so that you are not training any weights. I created the feature vectors for both test and train images with the model I created and added them to the array. The X-extension arrays specify the property vector, while the arrays y represent the class to which the image belongs. After this procedure I made classification using the linear SVM as stated in the assignment sheet and I printed accuracy the console.In addition to these, I have calculated class based accuracy. This processes is shown in Figure 8. All these accuracies are shown in the result section.

```
vgg16_model.eval()
vgg16_model.to(device)

test_x = []
test_y =[]

tr_x = []
tr_y = []

with torch.no_grad():

    for i, (x,y) in enumerate(data_load['train']):
        x = x.to(device)
        features1 = vgg16_model(x)

        tr_y.extend(y.cpu().numpy())
        tr_x.extend(features1.cpu().numpy())

    for i, (a,b) in enumerate(data_load['test']):
        a = a.to(device)
        features2 = vgg16_model(a)

        test_y.extend(b.cpu().numpy())
        test_x.extend(features2.cpu().numpy())


len_test = len(i_dataset["test"].classes)
correct = np.zeros(len_test)
total = data_size['test']/ len_test

classify = LinearSVC(random_state=0, max_iter=500)
classifier =classify.fit(tr_x, tr_y)

pred_test =classify.predict(test_x)
for i in range(len(pred_test)):
  if(pred_test[i] == test_y[i]):
    correct[pred_test[i]] = correct[pred_test[i]] + 1

result_acc = 100*classify.score(test_x,test_y)
print('Part 1 Average Accuracy : {:.2f}'.format(result_acc))

for i in range(len(correct)):
  correct[i] = 100* correct[i]/total
  print("{} Class Accuracy = {:.2f}%".format(class_label[i], correct[i]))
```

Figure 8: Feature Extraction and Applying SVM

In Figure 9, my confusion matrix drawing code is shown.

```
cm = confusion_matrix(test_y, pred_test)
normalize = False
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(cm)
fig.colorbar(cax)

fmt = '.2f' if normalize else 'd'
thresh = cm.max() / 2.
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        ax.text(j, i, format(cm[i, j], fmt),
                ha="center", va="center",
                color="black" if cm[i, j] > thresh else "white")
fig.tight_layout()

tick_marks = np.arange(len(class_label))
plt.xticks(tick_marks,class_label, rotation=90)
plt.yticks(tick_marks,class_label)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

Figure 9: Confusion Matrix

## 3.3 Parameters

### 3.3.1 Batch Size

Batch size is a term used in machine learning and computer vision and refers to which denotes the number of samples contained in each generated batch. Table 1 shows the accuracy according to the change in batch number.According to the table, the accuracy increases as the batch decreases.

| Batch Size | Max Iteration | Accuracy |
|---|---|---|
| 64 | 500 | % 81.60 |
| 32 | 500 | % 84.00 |
| 16 | 500 | % 84.40 |

Table 1: Accuracy according to the change in batch number

### 3.3.2 Max Iter

This parameter refers to the maximum number of iterations to run. Table 2 shows the accuracy according to the change in number of max-iter. Although the number of max-iter is large change, it doesn't affect accuracy very much.This observation can also be observed from table2.

| Batch Size | Max Iteration | Accuracy |
|---|---|---|
| 64 | 100 | % 84.00 |
| 64 | 1000 | % 84.00 |
| 64 | 3000 | % 83.20 |

Table 2: Accuracy according to the change in max-iter

### 3.4 Experimental Results and Analysis

Table 3 shows the results according to the different parameter values. Also for some parameters, confusion matrix and class-based accuracy are given.As far as I can observe from the results, the best accuracy was achieved with the 500 max-iter of 16 batch size.

When looking at the confusion matrices, it can be concluded that the class which gives the best result is inside-subway, the worst is the restaurant and living room.

| Batch Size | Max Iteration | Accuracy |
|---|---|---|
| 64 | 100 | % 84.00 |
| 32 | 100 | % 83.20 |
| 64 | 500 | % 81.60 |
| 16 | 500 | % 82.80 |
| 64 | 1000 | % 84.00 |
| 32 | 1000 | % 80.40 |
| 64 | 3000 | % 83.20 |
| 16 | 5000 | % 80.00 |
| 128 | 1000 | % 82.80 |

Table 3: Part 1 Result

Example results (Confusion matrix and accuracy ) for different parameters.

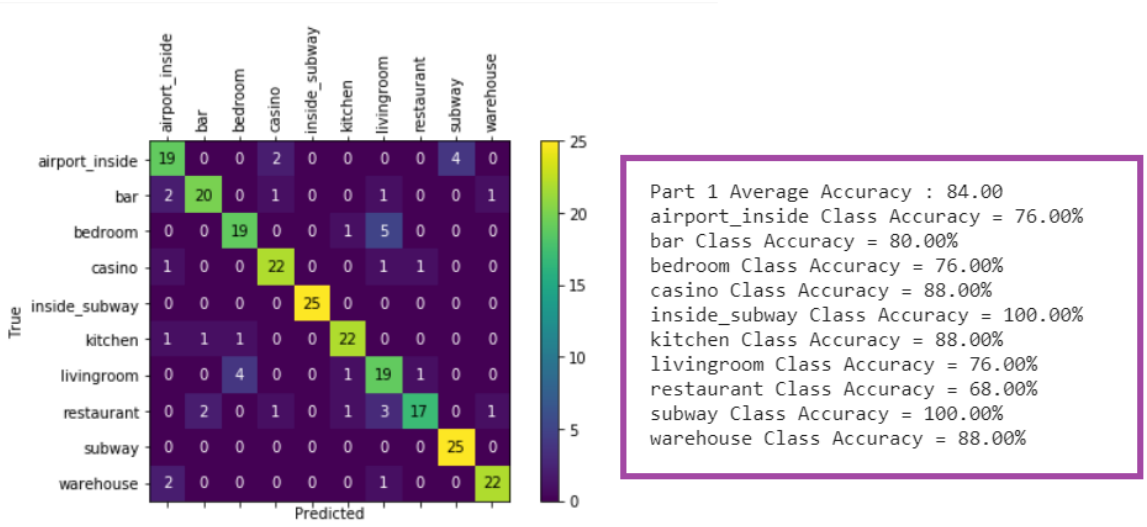



Figure 10: 64 Batch Size and 100 iteration Confusion Matrix and Accuracy

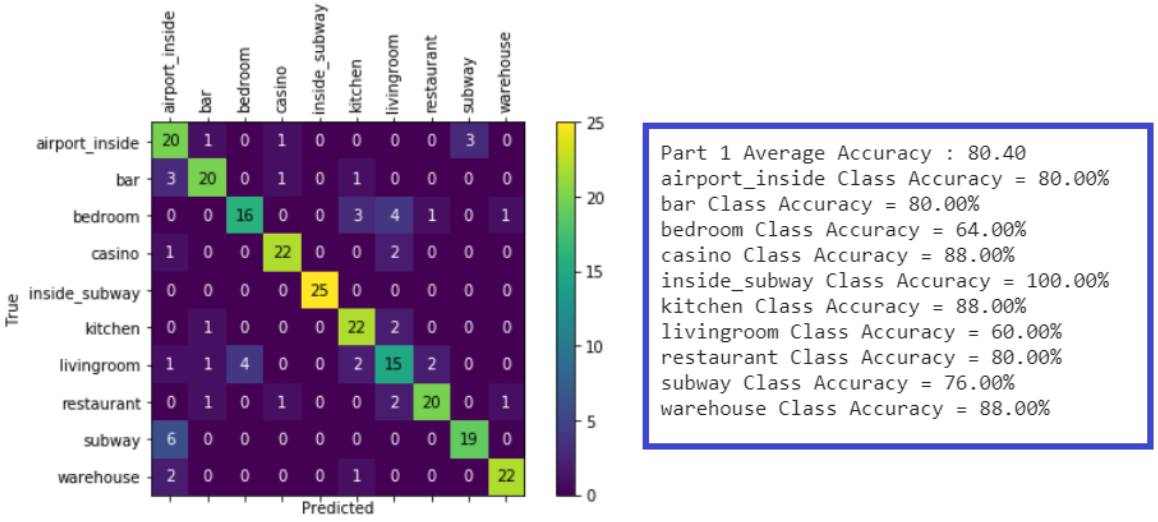
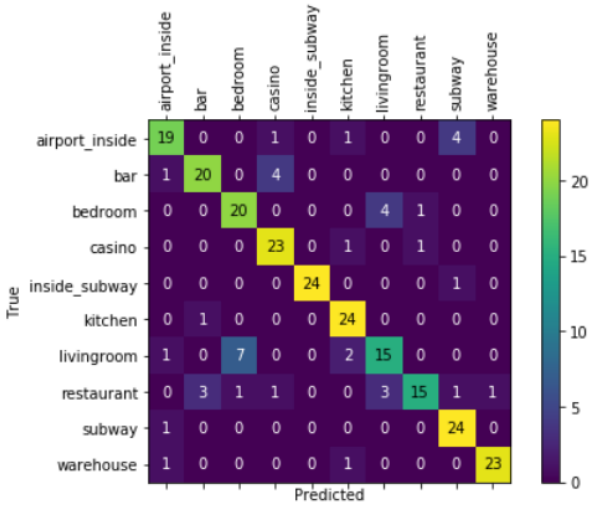
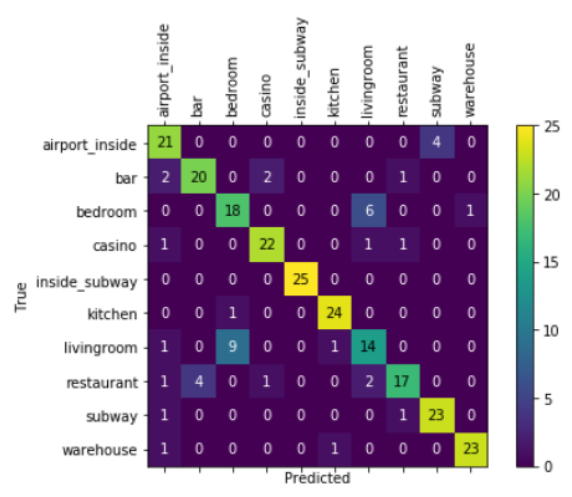



Figure 11: 32 Batch Size and 1000 iteration Confusion Matrix and Accuracy




```
Part 1 Average Accuracy : 82.80
airport_inside Class Accuracy = 76.00%
bar Class Accuracy = 80.00%
bedroom Class Accuracy = 80.00%
casino Class Accuracy = 92.00%
inside_subway Class Accuracy = 96.00%
kitchen Class Accuracy = 96.00%
livingroom Class Accuracy = 60.00%
restaurant Class Accuracy = 60.00%
subway Class Accuracy = 96.00%
warehouse Class Accuracy = 92.00%
```

Figure 12: 16 Batch Size and 500 iteration Confusion Matrix and Accuracy




```
Part 1 Average Accuracy : 82.80
airport_inside Class Accuracy = 84.00%
bar Class Accuracy = 80.00%
bedroom Class Accuracy = 72.00%
casino Class Accuracy = 88.00%
inside_subway Class Accuracy = 100.00%
kitchen Class Accuracy = 96.00%
livingroom Class Accuracy = 56.00%
restaurant Class Accuracy = 68.00%
subway Class Accuracy = 92.00%
warehouse Class Accuracy = 92.00%
```

Figure 13: 128 Batch Size and 1000 iteration Confusion Matrix and Accuracy

# 4 Part 2

In this step, I fine-tuned the network in the previously trained VGG-16 network where you loaded all the weights from the lantern library, and I ran the network.

## 4.1 Background Information

Several pre-trained models used in transfer learning are based on large convolutional neural networks (CNN). Its high performance and its easiness in training are two of the main factors driving the popularity of CNN over the last years.

A typical CNN has two parts:

Convolutional base, which is composed by a stack of convolutional and pooling layers. The main goal of the convolutional base is to generate features from the image.

Classifier, which is usually composed by fully connected layers. The main goal of the classifier is to classify the image based on the detected features. A fully connected layer is a layer whose neurons have full connections to all activation in the previous layer.

***Fine-tuning***

First pretrain a deep net on a large-scale dataset, like ImageNet. Then, given a new dataset, we can start with these pretrained weights when training on our new task. This process is commonly called fine-tuning. We update all of the network's weights for the new task.

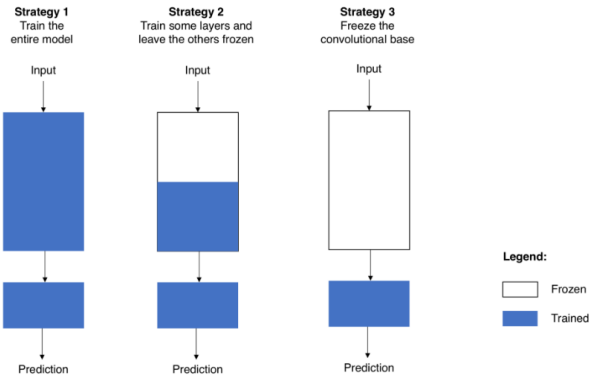



Figure 14: Fine tunning Strategies

Fine-tuning the ConvNet is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet, or it's possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier

features of a ConvNet contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset.

When and how to fine-tune? How do you decide what type of transfer learning you should perform on a new dataset?

This is a function of several factors, but the two most important ones are the size of the new dataset (small or big), and its similarity to the original dataset (e.g. ImageNet-like in terms of the content of images and the classes, or very different, such as microscope images). Keeping in mind that ConvNet features are more generic in early layers and more original-dataset-specific in later layers, here are some common rules of thumb for navigating the 4 major scenarios:

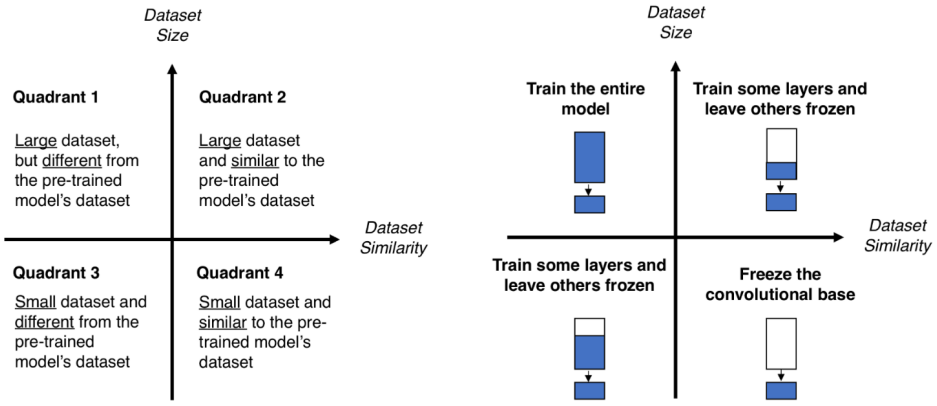



Figure 15: Fine tunning

## 4.2 Implementation Details

The model-of-training function (in figure 13) handles the training and validation of a given model. As input, it takes a PyTorch model, a dictionary of dataloaders, a loss function, an optimizer, a specified number of epochs to train and validate for. The function trains for the specified number of epochs and after each epoch runs a full validation step. It also keeps track of the best performing model (in terms of validation accuracy), and at the end of training returns the best performing model.

```python
def model_of_training(model, criterion, optimizer, epoch_num):

    best_model = copy.deepcopy(model.state_dict())
    train_acc = 0.0
    train_acc_hist = list()
    val_acc = 0.0
    val_acc_hist = list()
    loss_train_hist = list()
    loss_val_hist = list()

    for epoch in range(epoch_num):
        for section in ['train', 'validation']:
            if section == 'train':
                model.train()
            else:
                model.eval()

            loss_now = 0.0
            correct_out_now = 0

            for x, y in data_load[section]:
                x = x.to(device)
                y = y.to(device)
                optimizer.zero_grad()

                #forward
                with torch.set_grad_enabled(section == 'train'):
                    out = model(x)
                    _, preds = torch.max(out, 1)
                    loss = criterion(out, y)
                    # Backpropagation and Opitimization Training Section
                    if section == 'train':
                        loss.backward()
                        optimizer.step()

                loss_now += loss.item() * x.size(0)
                correct_out_now += torch.sum(preds == y.data)

            loss_now = loss_now / data_size[section]
            accuracy_epoch = correct_out_now.double() / data_size[section]

            if section == 'validation':
                val_acc_hist.append(accuracy_epoch)
                loss_val_hist.append(loss_now)
            else:
                train_acc_hist.append(accuracy_epoch)
                loss_train_hist.append(loss_now)

            if section == 'train' and accuracy_epoch > train_acc:
                train_acc = accuracy_epoch
            if section == 'validation' and accuracy_epoch > val_acc:
                val_acc = accuracy_epoch

                best_model = copy.deepcopy(model.state_dict())


    print('Train Accuracy: {:.2f}'.format(100*train_acc))
    print('Validation Accuracy: {:.2f}'.format(100*val_acc))

    # load best model weights
    model.load_state_dict(best_model)
```

Figure 16: train function

The test-acc function calculates the accuracy for the test data using the model that is being trained. It also calculates the accuracy of the top1 and top5 of the test dataset and prints them console.

```
def test_acc(trained_model, data_load, data_size):

  correct = 0
  top_five = 0
  top_one = 0
  len_test = data_size['test']
  cm = torch.zeros(10, 10)
  np.set_printoptions(precision=2)

  with torch.no_grad():
    for i, (x, y) in enumerate(data_load['test']):
        x = x.to(device)
        y = y.to(device)
        out = trained_model(x)
        _, preds = torch.max(out, 1)
        correct  += torch.sum(preds == y.data)


        for t, p in zip(y.view(-1), preds.view(-1)):
            cm[t.long(), p.long()] += 1


        probabilities, labels = out.topk(5,dim=1)
        for p in range(labels.size(0)):
          if y[p] in labels[p]:
            top_five +=1

        probabilities, labels = out.topk(1,dim=1)
        for p in range(labels.size(0)):
          if y[p] in labels[p]:
            top_one +=1


    top1_s = 100*top_one / len_test
    top5_s = 100*top_five / len_test
    test_acc =100*correct.double() / len_test

    print('Test acc: {:.2f}'.format(test_acc))
    print('Top5 acc: {:.2f}'.format(top5_s))
    print('Top1 acc: {:.2f}'.format(top1_s))

  return cm
```

Figure 17: Calculate accuracy

Here, our model is installed and created with vgg16.Then the required layers are frozen. ADAM is used as optimizer and loss of cross entropy and all these parameters are called with the function of the train.Depending on the model from the Train function, the function where we calculate the test accuracy is called.

```
vgg16_model = models.vgg16(pretrained=True)
#Freeze model weights
for param in vgg16_model.features.parameters():
    param.requires_grad = False

num_ftrs = vgg16_model.classifier[6].in_features
vgg16_model.classifier[6] = nn.Linear(num_ftrs ,10)

#print(vgg16_model)
vgg16_model  =vgg16_model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(vgg16_model.parameters(), lr=0.0001)

vgg16_model = model_of_training(vgg16_model, criterion, optimizer ,epoch_num=30)

matrix = test_acc_calculate(vgg16_model,data_load,data_size)

plt.figure(figsize = (10,10))
conf_mat_plot(matrix , label=class_label)
plt.show()
```

Figure 18: Create Model and called functions

```
def conf_mat_plot(cm, label):

    cmap=plt.cm.Blues
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.colorbar()
    tick_marks = np.arange(len(label))
    plt.xticks(tick_marks, label, rotation=45)
    plt.yticks(tick_marks, label)

    threshold = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j]),horizontalalignment="center",color="white" if cm[i, j] > threshold else "black")

    plt.xlabel('Predicted label')
    plt.ylabel('True label')
```

Figure 19: Plot Confusion Matrix

## 4.3 Parameters

### 4.3.1 Learning Rate

The learning rate is perhaps one of the most import hyperparameters which has to be set for enabling your deep neural network to perform better on train/val data sets.

If the learning rate is too small—training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny. If the learning rate is too big—then training may not converge or even diverge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse.

In this experiment, I have clearly observed that the learning rate increases, the accuracy is decreasing. To get a high accuracy I had to choose the low learning rate. I think the optimum learning rate parameter is 0.0001.We can observe this situation table 4 and 5.

| Batch Size | Learning Rate | Epoch | Train Accuracy | Validation Accuracy | Test Accuracy |
|---|---|---|---|---|---|
| 64 | 0.0001 | 10 | %86.92 | %86.00 | %86.00 |
| 64 | 0.001 | 10 | %76.70 | %82.60 | %84.00 |
| 64 | 0.01 | 10 | %26.07 | %40.40 | % 38.80 |

Table 4: Accuracy according to different learning rate

| Batch Size | Learning Rate | Epoch | Train Accuracy | Validation Accuracy | Test Accuracy |
|---|---|---|---|---|---|
| 32 | 0.001 | 15 | % 74.80 | % 81.60 | % 82.40 |
| 32 | 0.003 | 15 | % 52.35 | % 72.40 | % 73.60 |

Table 5: Accuracy according to different learning rate

### 4.3.2 Epoch Number

While the model is being trained, all of the data is not included in the training at the same time. They take part in a number of parts. The first part is trained, the performance of the model is tested and the weights are updated with backpropagation. Then the model is re-trained with the new training set and the weights are updated again. This process is repeated in each training step to calculate the most suitable weight values for the model. Each of these training steps is called “epoch”.

As the most suitable weight values were calculated step by step in my assignment, the performance was lower in the first epochs, and as the number of epoch increased, the performance increased. However, after a certain step, the learning status of my model will be considerably reduced.

In the model with the same parameters, the number of epoch increases with accuracy. In Table 6 we can observe this situation.

| Batch Size | Learning Rate | Epoch | Train Accuracy | Validation Accuracy | Test Accuracy |
|---|---|---|---|---|---|
| 64 | 0.0001 | 10 | % 86.92 | % 86.00 | % 86.00 |
| 64 | 0.0001 | 30 | % 91.03 | % 86.00 | % 87.20 |
| 32 | 0.003 | 15 | % 88.08 | % 86.80 | % 88.80 |
| 32 | 0.001 | 40 | %79.60 | %85.20 | % 82.40 |
| 16 | 0.001 | 20 | % 71.53 | % 81.60 | % 83.60 |
| 16 | 0.001 | 50 | % 76.17 | % 84.00 | % 84.40 |

Table 6: Accuracy according to different number of epoch

#### 4.3.3 Batch Size

Which denotes the number of samples contained in each generated batch. Table 7 shows the accuracy according to the change in batch number.When the other parameters are the same according to the table, the most suitable batch is set to 64.

| Batch Size | Learning Rate | Epoch | Train Accuracy | Validation Accuracy | Test Accuracy |
|---|---|---|---|---|---|
| 128 | 0.001 | 10 | % 80.80 | % 84.80 | % 83.20 |
| 64 | 0.001 | 10 | %86.92 | %86.00 | %86.00 |
| 32 | 0.001 | 10 | %73.33 | %81.20 | %83.20 |
| 16 | 0.001 | 10 | % 69.33 | % 80.00 | % 81.60 |

Table 7: Accuracy according to different batch size

### 4.4 Experimental Results and Analysis

The two highest accuracy I found as a result of my experiments and the parameters used in the model are shown in Table 8.The following shows the loss graph, the accuracy graph and the confusion matrix. When I looked at the loss charts, I observed that when it started from high values and the number of epochs decreased, it decreased loss and did not change much after a certain stage. On the other hand, i observed that the accuracy graph starts with low value and rises with the number of epoch.Here too, there is not much change after a certain epoch.

| Batch Size | Learning Rate | Epoch | Train Accuracy | Validation Accuracy | Test Accuracy |
|---|---|---|---|---|---|
| 64 | 0.0001 | 30 | % 91.08 | % 87.60 | % 88.80 |
| 16 | 0.0001 | 50 | % 91.85 | % 87.20 | % 88.40 |

Table 8: Best Accuracy

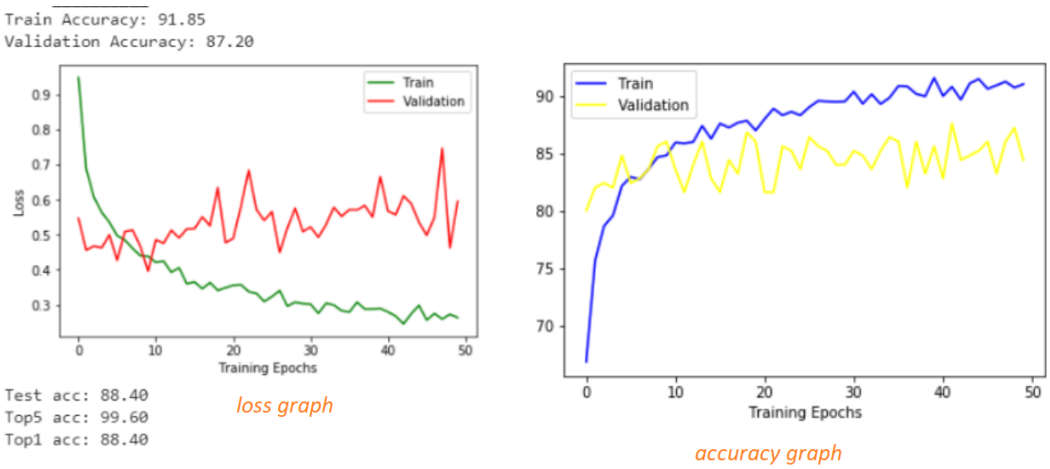



Figure 20: 16 Batch Size and 50 epoch and 0.0001 lr loss graph and accuracy graph

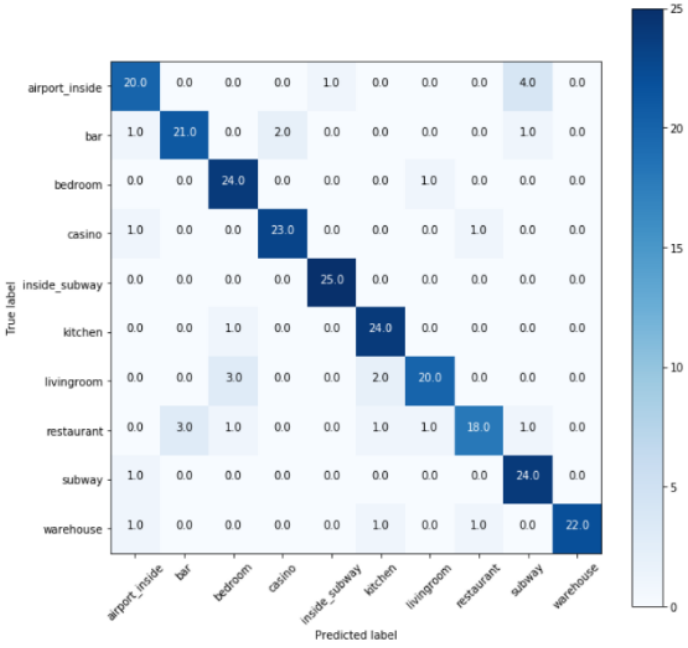



Figure 21: 16 Batch Size and 50 epoch and 0.0001 lr confusion matrix

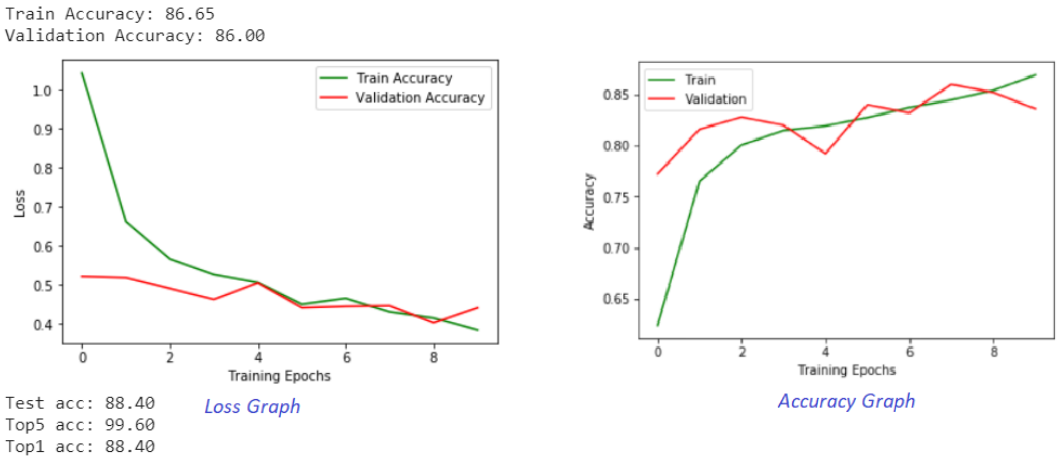



Figure 22: 64 Batch Size and 10 epoch and 0.0001 lr loss graph and accuracy graph

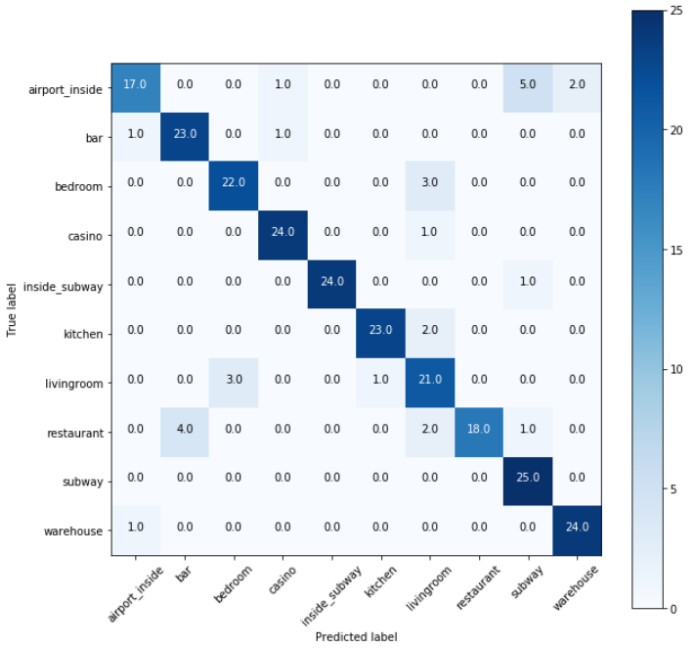



Figure 23: 64 Batch Size and 10 epoch and 0.0001 lr confusion matrix

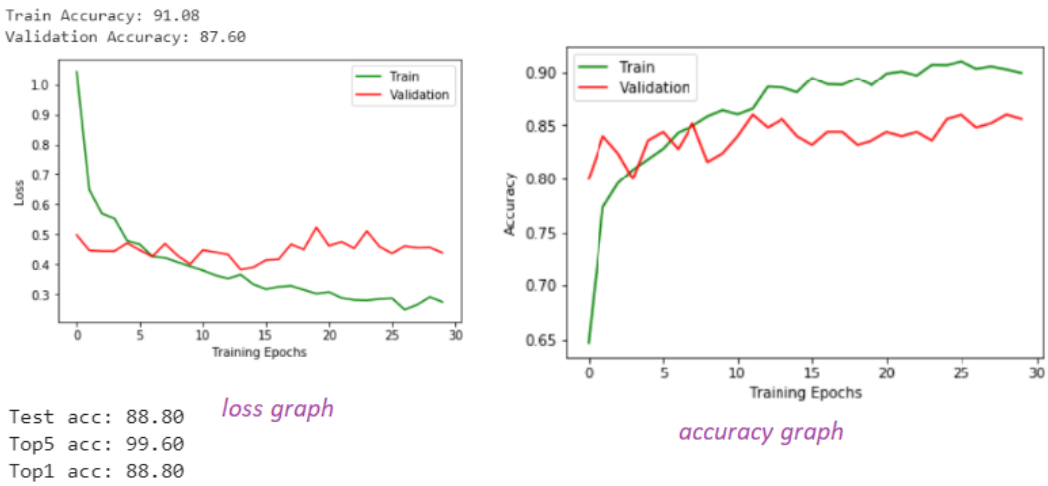



Figure 24: 64 Batch Size and 30 epoch and 0.0001 lr loss graph and accuracy graph

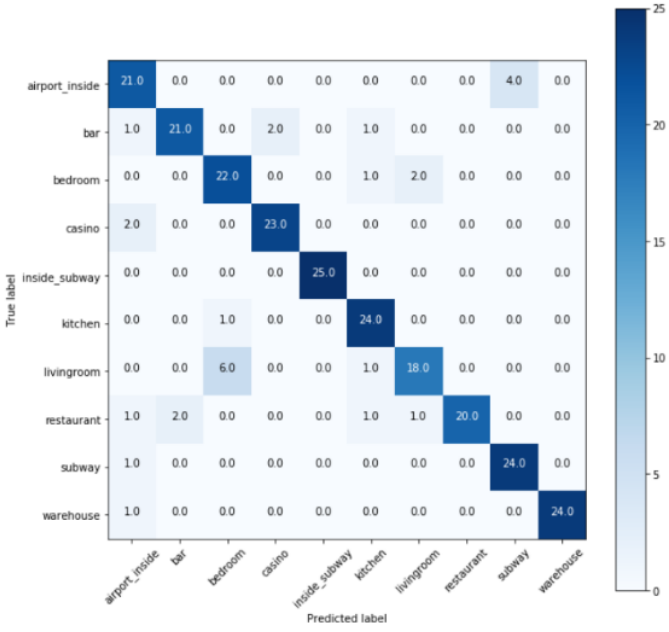



Figure 25: 64 Batch Size and 30 epoch and 0.0001 lr confusion matrix

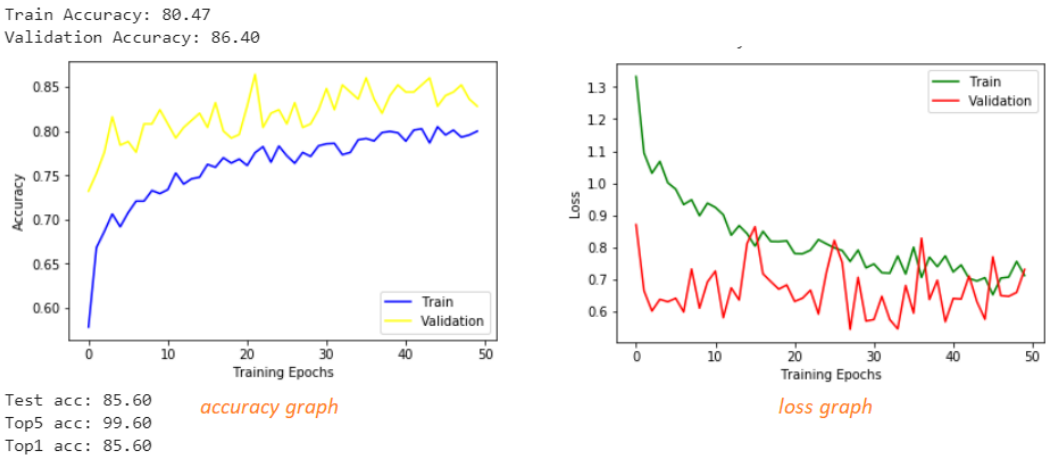



Figure 26: 32 Batch Size and 50 epoch and 0.001 lr loss graph and accuracy graph

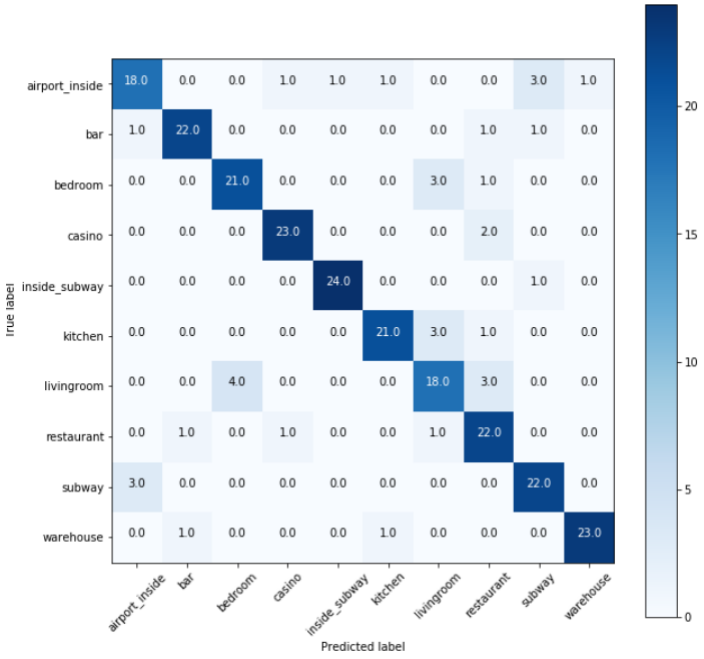



Figure 27: 32 Batch Size and 50 epoch and 0.001 lr confusion matrix

***Early Stopping***

A problem with training neural networks is in the choice of the number of training epochs to use.

Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

# 5 Part 3

In this step, I repeated Part1 and applied SVM again to get features over my fine-tuned network.

## 5.1 Implementation Details

The code for this step is the same as part 1. The only difference is that the model I sent to svm is which is the model that returns from the train function in part 2 .

## 5.2 Experimental Results and Analysis

I have tried part 3 with the highest accuracy model I have received in Part 2. The following are the sample results.When looking at the confusion matrices, it can be concluded that the classwhich gives the best result is inside-subway, the worst is living room.

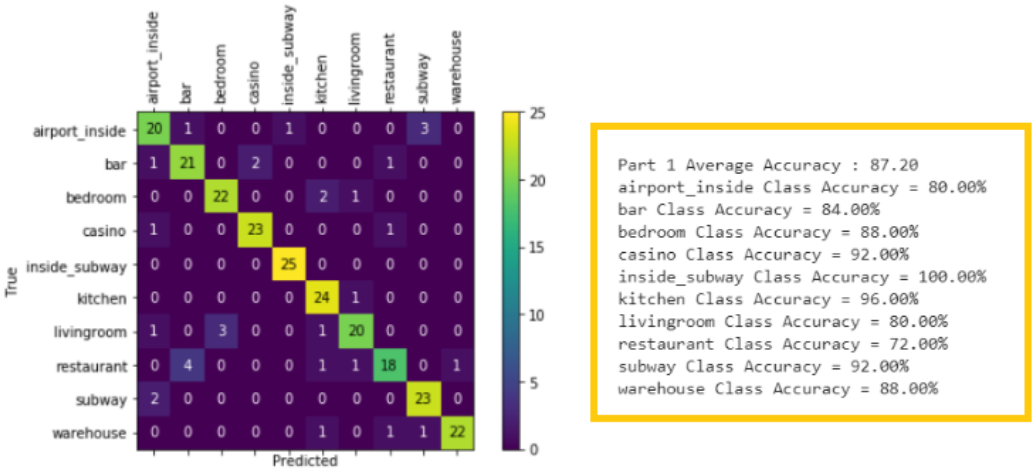



Figure 28: 16 batch size 0.0001 lr 50 epoch model 500 max-iter

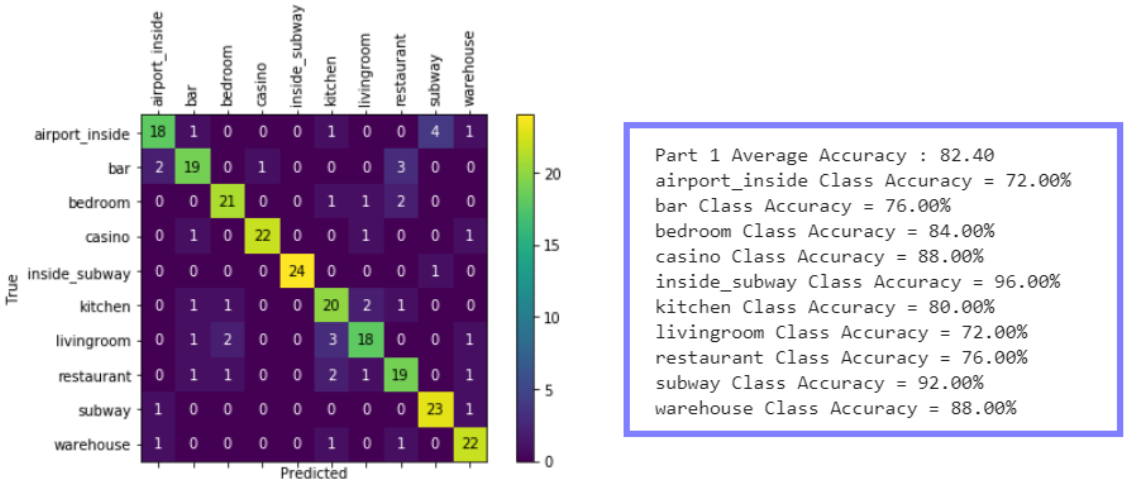



Figure 29: 64 batch size 0.001 lr 30 epoch model 3000 max-iter

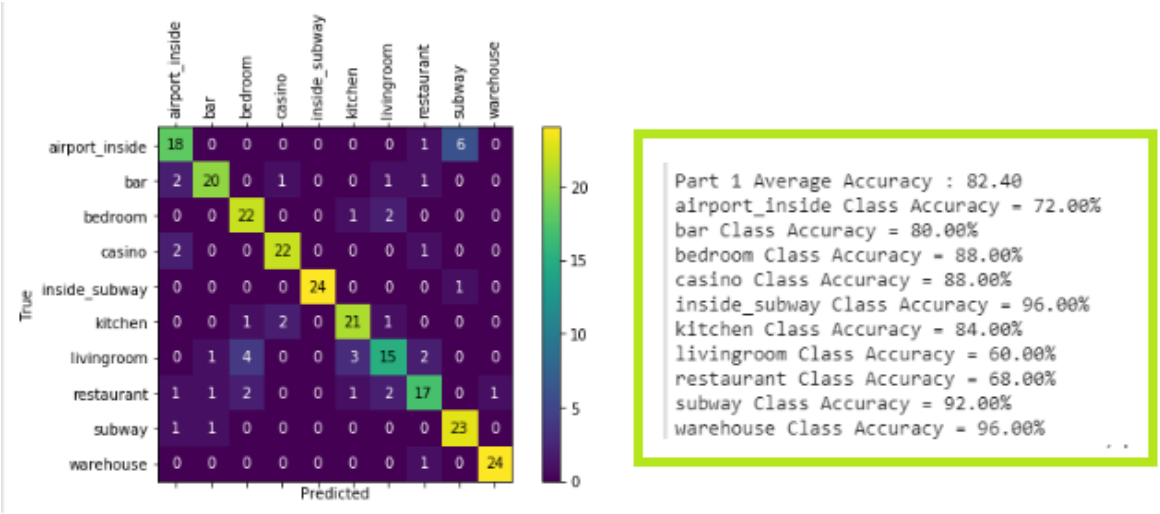



Figure 30: 128 batch size 0.001 lr 10 epoch model 1000 max-iter

# 6 Compare Result

When the results of Part 1 and Part 3 are compared, it can be observed that higher accuracy is obtained in part 3. The reason, for this is the use of the train model we train in part 2.In the other hand, The highest accuracy was obtained in part 2.Comparing in terms of time, part 1 is quite short compared to part 2 and 3.