Context

This is the Original data provided by MIT .

Indoor scene recognition is a challenging open problem in high level vision. Most scene recognition models that work well for outdoor scenes perform poorly in the indoor domain. The main difficulty is that while some indoor scenes (e.g. corridors) can be well characterized by global spatial properties, others (e.g., bookstores) are better characterized by the objects they contain. More generally, to address the indoor scenes recognition problem we need a model that can exploit local and global discriminative information.

Content

The database contains 67 Indoor categories, and a total of 15620 images. The number of images varies across categories, but there are at least 100 images per category. All images are in jpg format. The images provided here are for research purposes only.

Import Data from Kaggle

In [ ]:

```
! pip install -q kaggle
from google.colab import files
files.upload()                          # Upload kaggle.json file
! mkdir ~/.kaggle
! cp kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
```

In [ ]:

```
! kaggle datasets download -d itsahmad/indoor-scenes-cvpr-2019
! unzip indoor-scenes-cvpr-2019.zip
```

Import Modules

In [3]:

```
# Usual modules
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import time
import random
from tqdm.notebook import tqdm

# Data preparation modules
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from sklearn.model_selection import train_test_split

# Image Augmentation modules
# !pip install albumentations==0.4.6
# Restart runtime after installing this version
import albumentations as A
from albumentations.pytorch import ToTensorV2
```

```
import cv2

# Data learning modules
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
import torch.optim as optim
from torch.optim import lr_scheduler
import copy
```

Preprocessing Data

The data contains images and their annotations within a list. The text file 'TrainImages.txt' contains annotations to the images used in the training set, while the 'TestImages.txt' contains the same for the testing set. The image folders correspond to the classes. There are 67 image classes. The first step is to get the information about images into variables that can be used during the model training. Furthermore, we can split the training set into the training and validation sets.

In [4]:

```
# Create list of classes
classes = os.listdir('/content/indoorCVPR_09/Images')
labels2keys = {value: key for (key,value) in enumerate(classes)}
keys2labels = {key:value for (key,value) in enumerate(classes)}
print('Number of classes: ',len(classes))

# Create train and test datasets
train_annot = open('/content/TrainImages.txt').read().splitlines()
train_annot = [x for x in train_annot if "gif" not in x]
X = ['/content/indoorCVPR_09/Images/' + annot for annot in train_annot]
X_labels = [annot.split('/')[0] for annot in train_annot]
y = [labels2keys[label] for label in X_labels]

test_annot = open('/content/TestImages.txt').read().splitlines()
test_annot = [x for x in test_annot if "gif" not in x]
X_test = ['/content/indoorCVPR_09/Images/' + annot for annot in test_annot]
X_test_labels = [annot.split('/')[0] for annot in test_annot]
y_test = [labels2keys[label] for label in X_test_labels]

# Split train dataset into training and validation datasets
X_train, X_val, y_train, y_val = train_test_split(X,y,random_state=42,test_size=0.25,shu
ffle=True)
dataset_sizes = {'train' : len(X_train), 'val' : len(X_val), 'test' : len(X_test)}
print(dataset_sizes)
```

```
Number of classes:  67
{'train': 4011, 'val': 1337, 'test': 1334}
```

Image Augmentation

Thus, we have divided the dataset into three parts: train, validation and test sets. The ratio is 3:1:1. The next step is to create a class that processes each image and modifies it according to a set of rules. This is an important step since new images are not always clean, and the model must be prepared to tackle them. This class allows the model to gain robustness. The Albumentations module provides many options for modifying the images, of which few are used here. The images are all processed and transformed into a dataset, and batches of 64 are loaded for training.

In [5]:

```
# Create Image Dataset with all augmentations

class AlbumentationsDataset(Dataset):
  def __init__(self, file_paths, labels, transform=None):
    self.file_paths = file_paths
    self.labels = labels
    self.transform = transform
```

```
  def __len__(self):
    return len(self.file_paths)

  def __getitem__(self, idx):
    label = self.labels[idx]
    file_path = self.file_paths[idx]

    # Read an image with OpenCV
    image = cv2.imread(file_path)
    if image is None:
      print(file_path)

    # By default OpenCV uses BGR color space for color images,
    # so we need to convert the image to RGB color space.
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    if self.transform:
      augmented = self.transform(image=image)
      image = augmented['image']
    return image, label


albumentations_transform = A.Compose([
  A.Resize(256, 256),
  A.RandomCrop(224, 224),
  A.HorizontalFlip(),
  A.Normalize(
      mean=[0.485, 0.456, 0.406],
      std=[0.229, 0.224, 0.225],
  ),
  ToTensorV2()
])

train_dataset = AlbumentationsDataset(file_paths = X_train, labels = y_train,
                                      transform = albumentations_transform)
val_dataset = AlbumentationsDataset(file_paths = X_val, labels = y_val,
                                      transform = albumentations_transform)
test_dataset = AlbumentationsDataset(file_paths = X_test, labels = y_test,
                                      transform = albumentations_transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=0)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=True, num_workers=0)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=True, num_workers=0)
```

Image Viewer

We need a function for viewing the images from the dataloader we have created in the last step.

In [6]:

```
# Function for viewing an image from DataLoader
def imshow(image, ax=None, title=None, normalize=True):
  if ax is None:
    fig, ax = plt.subplots()
  image = image.numpy().transpose((1, 2, 0))

  if normalize:
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    image = std * image + mean
    image = np.clip(image, 0, 1)

  ax.imshow(image)
  ax.spines['top'].set_visible(False)
  ax.spines['right'].set_visible(False)
  ax.spines['left'].set_visible(False)
  ax.spines['bottom'].set_visible(False)
  ax.tick_params(axis='both', length=0)
  ax.set_xticklabels('')
  ax.set_yticklabels('')
```

```
  ax.set_title(title)

  return ax

images, labels = next(iter(train_loader))
for i in range(5):
  imshow(images[i],title=keys2labels[labels[i].tolist()])
```

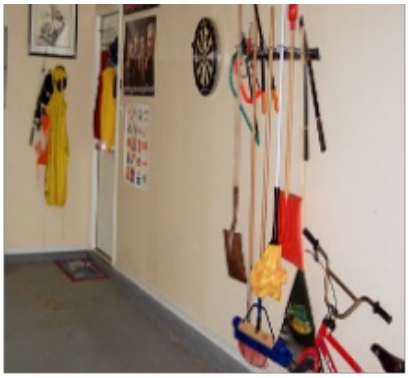
museum

airport_inside

bedroom

shoeshop

garage



Convolutional Neural Networks

Image Classification consists of building Convolutional Neural Networks. These contain hidden layers that transform the pixels in an image into data that can be processed into a label at the end. There are multiple CNNs already built for research purposes, such as ResNet, AlexNet, etc. These CNNs contain numerous hidden layers with predefined parameters that have been tested for large datasets. We can use them for our image classifier. However, the FC layer of these neural networks needs to be adjusted to fit our dataset, so that the output layer has neurons equal to the number of classes. Finally, we also use cross entropy loss minimization, Adam optimizer with a learning rate of 0.001, and a learning rate scheduler. The scheduler reduces the learning rate gradually so that the optimizer does not overshoot a local minima.

In [7]:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = models.resnet50(pretrained=True)
# print(model)

for param in model.parameters():           # Avoid backpropagation of pretrained layers
  param.requires_grad = False

model.fc = nn.Sequential(
    nn.Linear(2048,1024),                  # fc layer of resnet50 has 2048 inputs
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(1024, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512,len(classes))
)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
model.to(device)
learning_rate_scheduler = lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.75)
```

```
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/
torch/hub/checkpoints/resnet50-0676ba61.pth
```

Learning Model

Finally, we define the function for training the model. This consists of two steps: training and evaluation. We get the inputs from the dataloaders, optimize the gradients, and backpropagate. This is only done during the training phase. Every epoch, we note the running loss. At the end of the epochs, we can estimate the best accuracy that can be obtained from our model.

In [18]:

```
def train_model(model, criterion, optimizer, scheduler, num_epochs = 100):
  since = time.time()
  # Take backup of pretrained weights
  best_model_wts = copy.deepcopy(model.state_dict())
```

```
best_accuracy = 0.0
train_losses, val_losses = [], []

for epoch in range(num_epochs):
  print('Epoch {}/{}'.format(epoch, num_epochs - 1))
  print('-' * 10)



  # Each epoch has a training and validation phase
  for phase in ['train', 'val']:
    if phase == 'train':
      model.train()                     # Set model to training mode
      loader = train_loader
    else:
      model.eval()                      # Set model to evaluate mode
      loader = val_loader

    running_loss = 0
    running_corrects = 0

    # Iterate over data.
    for inputs, labels in loader:
      inputs = inputs.to(device)
      labels = labels.to(device)

      # zero the parameter gradients
      optimizer.zero_grad()

      # forward
      # track history if only in train
      with torch.set_grad_enabled(phase == 'train'):
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        loss = criterion(outputs, labels)

        # backward + optimize only if in training phase
        if phase == 'train':
          loss.backward()
          optimizer.step()

      # statistics
      running_loss += loss.item() * inputs.size(0)
      running_corrects += torch.sum(preds == labels.data)

    if phase == 'train':
      train_losses.append(running_loss)
    else:
      val_losses.append(running_loss)

    if phase == 'train':
      scheduler.step(running_loss)

    epoch_loss = running_loss / dataset_sizes[phase]
    epoch_acc = running_corrects.double() / dataset_sizes[phase]

    print('{} Loss: {:.4f} Acc: {:.4f}'.format(
      phase, epoch_loss, epoch_acc))

    # deep copy the model
    if phase == 'val' and epoch_acc > best_accuracy:
      best_accuracy = epoch_acc
      best_model_wts = copy.deepcopy(model.state_dict())

time_elapsed = time.time() - since
print('\nTraining complete in {:.0f}m {:.0f}s'.format(
time_elapsed // 60, time_elapsed % 60))
print('\nBest val Acc: {:4f}'.format(best_accuracy))

# load best model weights
model.load_state_dict(best_model_wts)
return model
```

Finally, we write a small function for checking the predicted label and the true label of a random set of images in the validation or the test set.

In [8]:

```
# def visualize_model(model, loader, num_images=6):
#   was_training = model.training
#   model.eval()
#   images_so_far = 0
#   fig = plt.figure()

#   with torch.no_grad():
#     for i, (inputs, labels) in enumerate(loader):
#       inputs = inputs.to(device)
#       labels = labels.to(device)

#       outputs = model(inputs)
#       _, preds = torch.max(outputs, 1)

#       for j in range(inputs.size()[0]):
#         images_so_far += 1
#         ax = plt.subplot(num_images//2, 2, images_so_far)
#         ax.axis('off')
#         ax.set_title('predicted: {}, actual: {}'.format(classes[preds[j]], classes[labe
ls[j]]))
#         imshow(inputs.cpu().data[j])

#         if images_so_far == num_images:
#             model.train(mode=was_training)
#             return
#     model.train(mode=was_training)

def predict_image(img, model):
    # Convert to a batch of 1
    xb = img.unsqueeze(0).to(device)
    # Get predictions from model
    yb = model(xb)
    # Pick index with highest probability
    prob, preds  = torch.max(yb, dim=1)
    # Retrieve the class label
    return classes[preds[0].item()]
```

Train Model

In [11]:

```
model = train_model(model, criterion, optimizer, learning_rate_scheduler, num_epochs=50)
```

```
Epoch 0/49
----------
```

```
/usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:718: UserWarning: Named ten
sors and all their associated APIs are an experimental feature and subject to change. Ple
ase do not use them for anything important until they are released as stable. (Triggered
internally at  /pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
```

```
train Loss: 4.1220 Acc: 0.0379
val Loss: 3.7116 Acc: 0.1997
Epoch 1/49
----------
train Loss: 3.1992 Acc: 0.1835
val Loss: 2.4924 Acc: 0.3553
Epoch 2/49
----------
train Loss: 2.5041 Acc: 0.2994
val Loss: 2.1195 Acc: 0.4286
Epoch 3/49
----------
```

```
train Loss: 2.1572 Acc: 0.3877
val Loss: 1.8207 Acc: 0.4779
Epoch 4/49
----------
train Loss: 1.9324 Acc: 0.4508
val Loss: 1.6826 Acc: 0.5064
Epoch 5/49
----------
train Loss: 1.7612 Acc: 0.4877
val Loss: 1.5634 Acc: 0.5415
Epoch 6/49
----------
train Loss: 1.6359 Acc: 0.5201
val Loss: 1.5245 Acc: 0.5587
Epoch 7/49
----------
train Loss: 1.5543 Acc: 0.5395
val Loss: 1.4744 Acc: 0.5654
Epoch 8/49
----------
train Loss: 1.4896 Acc: 0.5547
val Loss: 1.4717 Acc: 0.5625
Epoch 9/49
----------
train Loss: 1.4356 Acc: 0.5717
val Loss: 1.4329 Acc: 0.5812
Epoch 10/49
----------
train Loss: 1.3280 Acc: 0.5981
val Loss: 1.3797 Acc: 0.5961
Epoch 11/49
----------
train Loss: 1.2701 Acc: 0.6223
val Loss: 1.3355 Acc: 0.6163
Epoch 12/49
----------
train Loss: 1.2522 Acc: 0.6240
val Loss: 1.3004 Acc: 0.6171
Epoch 13/49
----------
train Loss: 1.2164 Acc: 0.6313
val Loss: 1.3382 Acc: 0.6111
Epoch 14/49
----------
train Loss: 1.1717 Acc: 0.6440
val Loss: 1.3537 Acc: 0.6036
Epoch 15/49
----------
train Loss: 1.0838 Acc: 0.6674
val Loss: 1.2718 Acc: 0.6260
Epoch 16/49
----------
train Loss: 1.0630 Acc: 0.6759
val Loss: 1.2705 Acc: 0.6148
Epoch 17/49
----------
train Loss: 1.0673 Acc: 0.6789
val Loss: 1.2589 Acc: 0.6268
Epoch 18/49
----------
train Loss: 1.0321 Acc: 0.6791
val Loss: 1.2775 Acc: 0.6298
Epoch 19/49
----------
train Loss: 1.0055 Acc: 0.6938
val Loss: 1.2957 Acc: 0.6200
Epoch 20/49
----------
train Loss: 0.9737 Acc: 0.7001
val Loss: 1.2615 Acc: 0.6200
Epoch 21/49
----------
```

```
train Loss: 0.9508 Acc: 0.7051
val Loss: 1.2740 Acc: 0.6290
Epoch 22/49
----------
train Loss: 0.9328 Acc: 0.7113
val Loss: 1.2516 Acc: 0.6350
Epoch 23/49
----------
train Loss: 0.9186 Acc: 0.7128
val Loss: 1.2535 Acc: 0.6328
Epoch 24/49
----------
train Loss: 0.8878 Acc: 0.7233
val Loss: 1.2269 Acc: 0.6432
Epoch 25/49
----------
train Loss: 0.8743 Acc: 0.7295
val Loss: 1.2491 Acc: 0.6358
Epoch 26/49
----------
train Loss: 0.8201 Acc: 0.7497
val Loss: 1.2448 Acc: 0.6380
Epoch 27/49
----------
train Loss: 0.8574 Acc: 0.7325
val Loss: 1.2429 Acc: 0.6328
Epoch 28/49
----------
train Loss: 0.8273 Acc: 0.7440
val Loss: 1.2555 Acc: 0.6313
Epoch 29/49
----------
train Loss: 0.8132 Acc: 0.7455
val Loss: 1.2127 Acc: 0.6447
Epoch 30/49
----------
train Loss: 0.7975 Acc: 0.7587
val Loss: 1.2377 Acc: 0.6372
Epoch 31/49
----------
train Loss: 0.7904 Acc: 0.7569
val Loss: 1.2176 Acc: 0.6447
Epoch 32/49
----------
train Loss: 0.7657 Acc: 0.7641
val Loss: 1.2673 Acc: 0.6328
Epoch 33/49
----------
train Loss: 0.7703 Acc: 0.7539
val Loss: 1.2054 Acc: 0.6530
Epoch 34/49
----------
train Loss: 0.7479 Acc: 0.7632
val Loss: 1.2451 Acc: 0.6395
Epoch 35/49
----------
train Loss: 0.7272 Acc: 0.7746
val Loss: 1.2356 Acc: 0.6417
Epoch 36/49
----------
train Loss: 0.7410 Acc: 0.7711
val Loss: 1.2436 Acc: 0.6350
Epoch 37/49
----------
train Loss: 0.7186 Acc: 0.7769
val Loss: 1.2502 Acc: 0.6477
Epoch 38/49
----------
train Loss: 0.7249 Acc: 0.7741
val Loss: 1.1977 Acc: 0.6477
Epoch 39/49
----------
```

```
train Loss: 0.7060 Acc: 0.7746
val Loss: 1.2392 Acc: 0.6328
Epoch 40/49
----------
train Loss: 0.7029 Acc: 0.7756
val Loss: 1.2349 Acc: 0.6485
Epoch 41/49
----------
train Loss: 0.6959 Acc: 0.7759
val Loss: 1.2287 Acc: 0.6530
Epoch 42/49
----------
train Loss: 0.6822 Acc: 0.7853
val Loss: 1.2119 Acc: 0.6545
Epoch 43/49
----------
train Loss: 0.7032 Acc: 0.7774
val Loss: 1.2286 Acc: 0.6425
Epoch 44/49
----------
train Loss: 0.7010 Acc: 0.7858
val Loss: 1.2488 Acc: 0.6395
Epoch 45/49
----------
train Loss: 0.6591 Acc: 0.7951
val Loss: 1.2285 Acc: 0.6372
Epoch 46/49
----------
train Loss: 0.6617 Acc: 0.8003
val Loss: 1.2361 Acc: 0.6455
Epoch 47/49
----------
train Loss: 0.6652 Acc: 0.7906
val Loss: 1.1961 Acc: 0.6455
Epoch 48/49
----------
train Loss: 0.6852 Acc: 0.7796
val Loss: 1.2239 Acc: 0.6492
Epoch 49/49
----------
train Loss: 0.6777 Acc: 0.7913
val Loss: 1.2024 Acc: 0.6537

Training complete in 44m 39s

Best val Acc: 0.654450
```

Check Predicted Labels

We see that the best validation set accuracy obtained was 65%. We can check some of the predicted labels and their true labels using the function we defined.

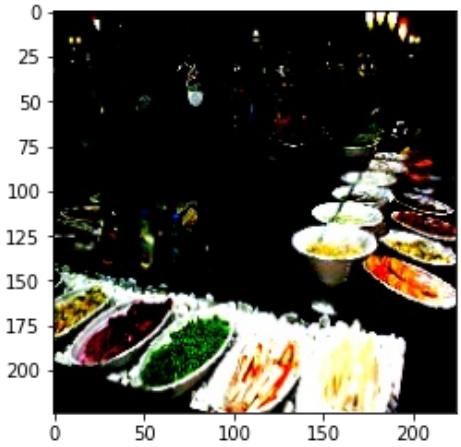
In [12]:

```
for i in range(5):
  n = random.randint(0,len(X_test))
  img, label = test_dataset[n]
  print('Label:', classes[label], ', Predicted:', predict_image(img, model))
  plt.imshow(img.permute(1, 2, 0))
  plt.show()


# visualize_model(model, test_loader)
```

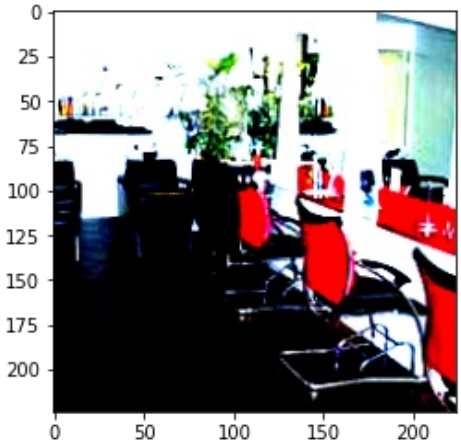
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
.255] for integers).
```

```
Label: buffet , Predicted: buffet
```



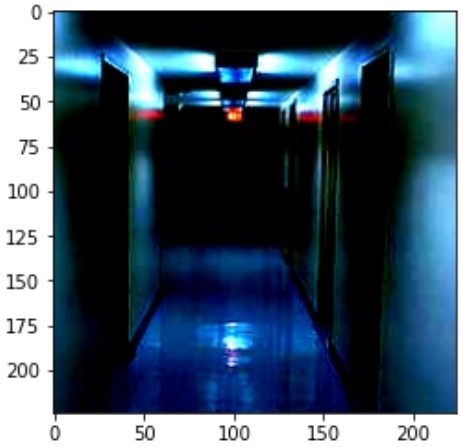
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
.255] for integers).
```

```
Label: hairsalon , Predicted: gym
```



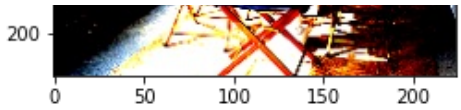
```
Label: corridor , Predicted: corridor
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
.255] for integers).
```



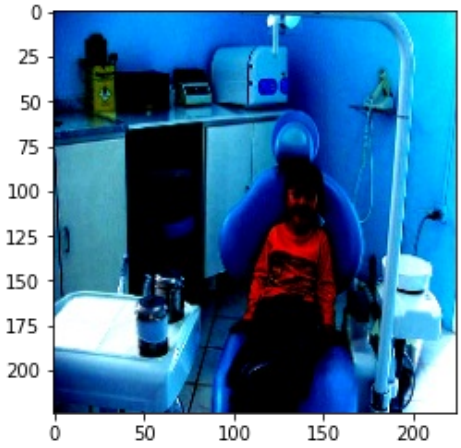
```
Label: dining_room , Predicted: classroom
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
.255] for integers).
```




```
Label: dentaloffice , Predicted: dentaloffice
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
.255] for integers).
```



Changing the Model

We can see that given the image augmentations, the model did decently good. We tested the model for five images from the test set, which it has never seen, and it still predicted three of them correctly. Coincidentally, that is 66% accuracy. The next job is to modify certain parameters of the model and observe if the performance changes. Here, we change the learning rate, the learning rate scheduler, and some of the layers in the FC layer of the model.

In [17]:

```
# Changing model parameters
model.fc = nn.Sequential(
    nn.Linear(2048,2000),                  # fc layer of resnet50 has 2048 inputs
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(2000, 1000),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(1000, 512),
    nn.ReLU(),
    nn.Dropout(0.1),
    nn.Linear(512,len(classes))
)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.0005)
model.to(device)
learning_rate_scheduler = lr_scheduler.ReduceLROnPlateau(optimizer)
```

As can be seen, we have changed some of the layers in the FC layer of the CNN, reduced the learning rate by half, and are using a different learning rate scheduler. The ReduceLROnPlateau allows dynamic learning rate reducing based on some validation measurements. We can run this model and check whether it performs better or worse than before.

In [19]:

```
model = train_model(model, criterion, optimizer, learning_rate_scheduler, num_epochs=50)
```

```
Epoch 0/49
----------
train Loss: 4.1821 Acc: 0.0267
val Loss: 3.9997 Acc: 0.0546
Epoch 1/49
```

```
----------
train Loss: 3.4462 Acc: 0.1169
val Loss: 2.9338 Acc: 0.2102
Epoch 2/49
----------
train Loss: 2.6824 Acc: 0.2545
val Loss: 2.3172 Acc: 0.3396
Epoch 3/49
----------
train Loss: 2.3389 Acc: 0.3321
val Loss: 2.0223 Acc: 0.4174
Epoch 4/49
----------
train Loss: 2.0062 Acc: 0.4064
val Loss: 1.9265 Acc: 0.4383
Epoch 5/49
----------
train Loss: 1.8681 Acc: 0.4438
val Loss: 1.8072 Acc: 0.4607
Epoch 6/49
----------
train Loss: 1.6956 Acc: 0.4964
val Loss: 1.6990 Acc: 0.5071
Epoch 7/49
----------
train Loss: 1.6210 Acc: 0.5004
val Loss: 1.6485 Acc: 0.5153
Epoch 8/49
----------
train Loss: 1.5051 Acc: 0.5497
val Loss: 1.5537 Acc: 0.5542
Epoch 9/49
----------
train Loss: 1.4357 Acc: 0.5642
val Loss: 1.4581 Acc: 0.5669
Epoch 10/49
----------
train Loss: 1.3637 Acc: 0.5799
val Loss: 1.5053 Acc: 0.5677
Epoch 11/49
----------
train Loss: 1.3471 Acc: 0.5931
val Loss: 1.5144 Acc: 0.5647
Epoch 12/49
----------
train Loss: 1.2158 Acc: 0.6382
val Loss: 1.5273 Acc: 0.5662
Epoch 13/49
----------
train Loss: 1.1734 Acc: 0.6387
val Loss: 1.3898 Acc: 0.5999
Epoch 14/49
----------
train Loss: 1.1216 Acc: 0.6587
val Loss: 1.3703 Acc: 0.6043
Epoch 15/49
----------
train Loss: 1.0757 Acc: 0.6672
val Loss: 1.4914 Acc: 0.5931
Epoch 16/49
----------
train Loss: 1.0408 Acc: 0.6836
val Loss: 1.4006 Acc: 0.6006
Epoch 17/49
----------
train Loss: 0.9863 Acc: 0.7003
val Loss: 1.4106 Acc: 0.6185
Epoch 18/49
----------
train Loss: 0.9648 Acc: 0.6948
val Loss: 1.3899 Acc: 0.6163
Epoch 19/49
```

```
----------
train Loss: 0.9363 Acc: 0.7110
val Loss: 1.4201 Acc: 0.6096
Epoch 20/49
----------
train Loss: 0.9133 Acc: 0.7133
val Loss: 1.4737 Acc: 0.5991
Epoch 21/49
----------
train Loss: 0.8513 Acc: 0.7375
val Loss: 1.4236 Acc: 0.6268
Epoch 22/49
----------
train Loss: 0.8646 Acc: 0.7273
val Loss: 1.3983 Acc: 0.6215
Epoch 23/49
----------
train Loss: 0.7970 Acc: 0.7517
val Loss: 1.3572 Acc: 0.6171
Epoch 24/49
----------
train Loss: 0.8091 Acc: 0.7447
val Loss: 1.4187 Acc: 0.6133
Epoch 25/49
----------
train Loss: 0.7692 Acc: 0.7579
val Loss: 1.4154 Acc: 0.6141
Epoch 26/49
----------
train Loss: 0.7093 Acc: 0.7731
val Loss: 1.4231 Acc: 0.6215
Epoch 27/49
----------
train Loss: 0.6904 Acc: 0.7796
val Loss: 1.4230 Acc: 0.6156
Epoch 28/49
----------
train Loss: 0.6978 Acc: 0.7751
val Loss: 1.4946 Acc: 0.6103
Epoch 29/49
----------
train Loss: 0.6475 Acc: 0.7866
val Loss: 1.4567 Acc: 0.6328
Epoch 30/49
----------
train Loss: 0.6648 Acc: 0.7846
val Loss: 1.5207 Acc: 0.6051
Epoch 31/49
----------
train Loss: 0.6377 Acc: 0.7901
val Loss: 1.4282 Acc: 0.6417
Epoch 32/49
----------
train Loss: 0.6157 Acc: 0.7956
val Loss: 1.4913 Acc: 0.6290
Epoch 33/49
----------
train Loss: 0.5956 Acc: 0.8080
val Loss: 1.5294 Acc: 0.6171
Epoch 34/49
----------
train Loss: 0.5576 Acc: 0.8277
val Loss: 1.4786 Acc: 0.6455
Epoch 35/49
----------
train Loss: 0.5199 Acc: 0.8310
val Loss: 1.4875 Acc: 0.6290
Epoch 36/49
----------
train Loss: 0.4938 Acc: 0.8469
val Loss: 1.5677 Acc: 0.6223
Epoch 37/49
```

```
----------
train Loss: 0.4931 Acc: 0.8382
val Loss: 1.5230 Acc: 0.6320
Epoch 38/49
----------
train Loss: 0.4954 Acc: 0.8382
val Loss: 1.6322 Acc: 0.6126
Epoch 39/49
----------
train Loss: 0.4958 Acc: 0.8417
val Loss: 1.5060 Acc: 0.6185
Epoch 40/49
----------
train Loss: 0.4613 Acc: 0.8479
val Loss: 1.4800 Acc: 0.6372
Epoch 41/49
----------
train Loss: 0.4538 Acc: 0.8546
val Loss: 1.5143 Acc: 0.6126
Epoch 42/49
----------
train Loss: 0.4285 Acc: 0.8661
val Loss: 1.4722 Acc: 0.6343
Epoch 43/49
----------
train Loss: 0.4321 Acc: 0.8624
val Loss: 1.5261 Acc: 0.6320
Epoch 44/49
----------
train Loss: 0.4282 Acc: 0.8616
val Loss: 1.5860 Acc: 0.6298
Epoch 45/49
----------
train Loss: 0.4304 Acc: 0.8601
val Loss: 1.5389 Acc: 0.6238
Epoch 46/49
----------
train Loss: 0.4015 Acc: 0.8736
val Loss: 1.5829 Acc: 0.6417
Epoch 47/49
----------
train Loss: 0.4114 Acc: 0.8681
val Loss: 1.6399 Acc: 0.6185
Epoch 48/49
----------
train Loss: 0.3828 Acc: 0.8753
val Loss: 1.5471 Acc: 0.6350
Epoch 49/49
----------
train Loss: 0.3763 Acc: 0.8746
val Loss: 1.6242 Acc: 0.6365

Training complete in 45m 3s

Best val Acc: 0.645475
```

Conclusion

We see that the model performed almost identically as earlier. Thus, there is not much improvement to be observed. However, there are many ways in which we can alter the process in order to enhance it, such as changing image augmentation parameters, using different CNNs or building one from scratch, and trying various optimizers and learning rate schedulers. As more and more of these parameters are tested, it becomes clearer which of them have an impact on the model performance. This allows optimizing those parameters specifically. In conclusion, we were able to achieve an accuracy of 65% using a rudimentary Deep Learning analysis on the MIT Indoor Scenes Classification.