Problem 3 - Transfer learning: Shallow learning vs Finetuning

In this problem we will train a convolutional neural network for image classification using transfer learning. Transfer learning involves training a base network from scratch on a very large dataset (e.g., Imagenet1K with 1.2 M images and 1K categories) and then using this base network either as a feature extractor or as an initialization network for target task. Thus two major transfer learning scenarios are as follows:

• Finetuning the base model: Instead of random initialization, we initialize the network with a pretrained network, like the one that is trained on Imagenet dataset. Rest of the training looks as usual however the learning rate schedule for transfer learning may be different.

• Base model as fixed feature extractor: Here, we will freeze the weights for all of the network except that of the final fully connected layer.

This last fully connected layer is replaced with a new one with random weights and only this layer is trained. For this problem the following resources will be helpful. References • Pytorch blog. Transfer Learning for Computer Vision Tutorial by S. Chilamkurthy Available at https://pytorch.org/tutorials/beginner/transfer learning tutorial.html • Notes on Transfer Learning. CS231n Convolutional Neural Networks for Visual Recognition. Available at https://cs231n.github.io/transfer-learning/ • Visual Domain Decathlon

1. For fine-tuning you will select a target dataset from the Visual-Decathlon challenge. Their web site (link below) has several datasets which you can download. Select any one of the visual decathlon dataset and make it your target dataset for transfer learning. Important : Do not select Imagenet1K as the target dataset.

In [ ]:
```
import tensorflow_datasets as tfds
from torchvision import models,datasets,transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch

stats = ((0.507, 0.487, 0.441), (0.267, 0.256, 0.276))

data_transforms = {
    'train': transforms.Compose([transforms.RandomCrop(32, padding=4, padding
                         transforms.RandomHorizontalFlip(),
                         transforms.ToTensor(),
                         transforms.Normalize(*stats,inplace=True)]),
    'val': transforms.Compose([
        transforms.ToTensor(),
         transforms.Normalize(*stats)

    ]),
}


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

data = {'train':datasets.CIFAR100(root='./data', train=True, download=True,
transform=data_transforms['train']),
```

```
    "val":datasets.CIFAR100(root='./data', train=False, download=True,
transform=data_transforms['val'])}
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz to ./dat
a/cifar-100-python.tar.gz

Extracting ./data/cifar-100-python.tar.gz to ./data
Files already downloaded and verified
```

In [ ]:
```
batch_size = 64
dataloaders = {
    'train': DataLoader(data['train'], batch_size=batch_size, shuffle=True,nu
    'val': DataLoader(data['val'], batch_size=batch_size*2, shuffle=True,num_
}
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:566: Use
rWarning: This DataLoader will create 3 worker processes in total. Our suggest
ed max number of worker in current system is 2, which is smaller than what thi
s DataLoader is going to create. Please be aware that excessive worker creatio
n might get DataLoader running slow or even freeze, lower the worker number to
avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

In [ ]:
```
trainiter = iter(dataloaders['train'])
features, labels = next(trainiter)
features.shape, labels.shape
```

Out[ ]:
```
(torch.Size([64, 3, 32, 32]), torch.Size([64]))
```

(a) Finetuning: You will first load a pretrained model (Resnet50) and change the final fully connected layer output to the number of classes in the target dataset. Describe your target dataset features, number of classes and distribution of images per class (i.e., number of images per class). Show any 4 sample images (belonging to 2 different classes) from your target dataset. (2+2)

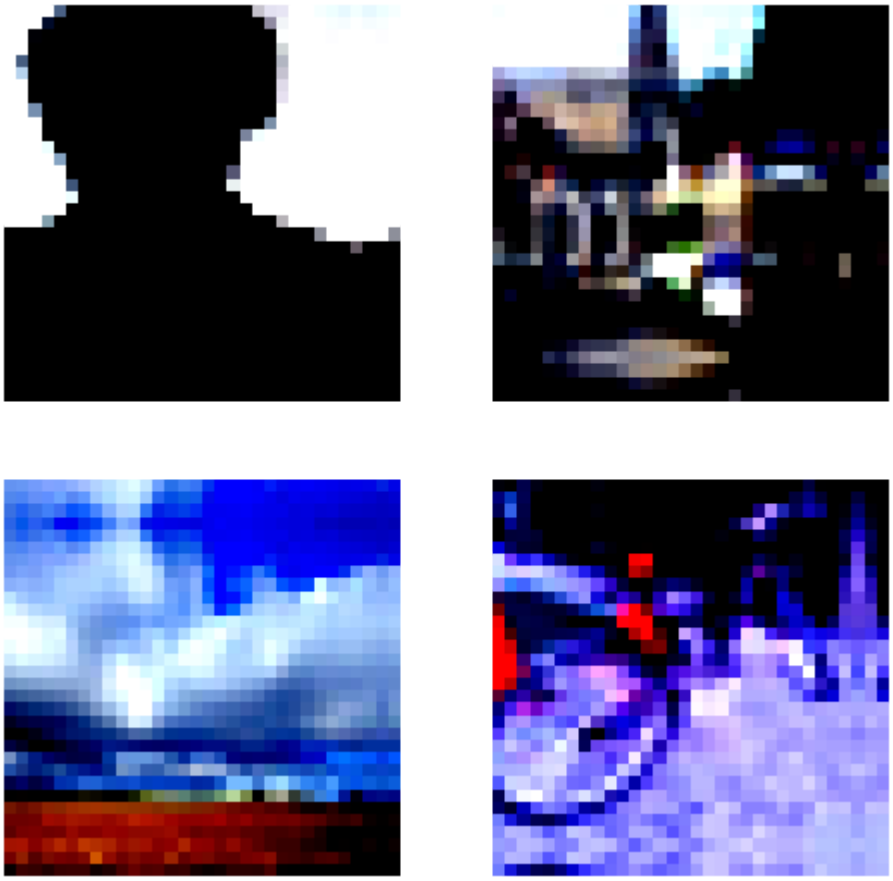
Answer:

The CIFAR-100 dataset contains 50,000 training and 10,000 test images of 20 object classes, along with 100 object subclasses. It is traditional to train on the 100 object subclasses. Each image is an RGB image of size 32x3

In [ ]:
```
import torch
import matplotlib.pyplot as plt
import numpy as np

figure = plt.figure(figsize=(8, 8))
cols, rows = 2, 2
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(dataloaders['train']), size=(1,)).item()
    img, label = data['train'][sample_idx]
    figure.add_subplot(rows, cols, i)
    npimg = img.numpy()
    # plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
h RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
h RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
h RGB data ([0..1] for floats or [0..255] for integers).
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
h RGB data ([0..1] for floats or [0..255] for integers).
```



In [ ]:
```
# pretrained model.
resnet = models.resnet50(pretrained=True)

num_features = resnet.fc.in_features

# the fully connected layer with to classify the 100 classes

resnet.fc = nn.Linear(num_features,100)

# sending to device , Following tutorial.
resnet.to(device)
```

```
/usr/local/lib/python3.7/dist-packages/torchvision/models/_utils.py:209: UserW
arning: The parameter 'pretrained' is deprecated since 0.13 and will be remove
d in 0.15, please use 'weights' instead.
  f"The parameter '{pretrained_param}' is deprecated since 0.13 and will be re
moved in 0.15, "
/usr/local/lib/python3.7/dist-packages/torchvision/models/_utils.py:223: UserW
arning: Arguments other than a weight enum or `None` for 'weights' are depreca
ted since 0.13 and will be removed in 0.15. The current behavior is equivalent
to passing `weights=ResNet50_Weights.IMAGENET1K_V1`. You can also use `weights
=ResNet50_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /r
oot/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth
```

Out[ ]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bi
as=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mo
de=False)
  (layer1): Sequential(
```

```
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
```

```
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
```

```
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=Fals
e)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=100, bias=True)
)
```

(b) First finetune by setting the same value of hyperparameters (learning rate=0.001, momentum=0.9) for all the layers. Keep batch size of 64 and train for 50-60 epochs or until model converges well. You will use a multi-step learning rate schedule and decay by a factor of 0.1 (γ = 0.1 in the link below). You can choose steps at which you want to decay the learning rate but do 3 drops during the training. So the first drop will bring down the learning rate to 0.0001, second to 0.00001, third to 0.000001. For example, if training for 60 epochs, first drop can happen at epoch 15, second at epoch 30 and third at epoch 45. (6)

In [ ]:

```
import time
import os
import copy
dataset_sizes = {x: len(data[x]) for x in ['train', 'val']}
#dictiorany to store the data across epochs
training_data = {"epoch":[],'train_acc':[],"test_acc":[]}

def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    training_data["epoch"] = []
```

```
    training_data['train_acc'] = []
    training_data["test_acc"] = []
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    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
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
                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':
                scheduler.step()

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

            training_data['epoch'].append(epoch+1)
            if phase =='train':
              training_data['train_acc'].append(epoch_acc)
            else:
              training_data['test_acc'].append(epoch_acc)

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

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

        print()
```

```
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

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

In [ ]:
```
# Function created by me to print a
def print_history(data):
  import matplotlib.pyplot as plt

  new_data  = {}
  for k, v in data.items():
    if k is not 'epoch':
      v = [ float(i.detach().to("cpu").numpy()) for i in v]
      new_data[k] = v

  plt.plot(list(set(data['epoch'])),new_data['train_acc'],label= "Training Ac
  plt.plot(list(set(data['epoch'])),new_data['test_acc'],label= "Validation A
  plt.legend()
  plt.show()
```

In [ ]:
```
epochs = 40
step = 10

criterion = nn.CrossEntropyLoss()

# setting hyperparameter values.
optimizer = torch.optim.SGD(resnet.parameters(),lr = 0.001,momentum = 0.9)

# multistep learning rate schedule

step_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=step,

model_1 = train_model(resnet, criterion, optimizer, step_lr_scheduler, num_ep
```

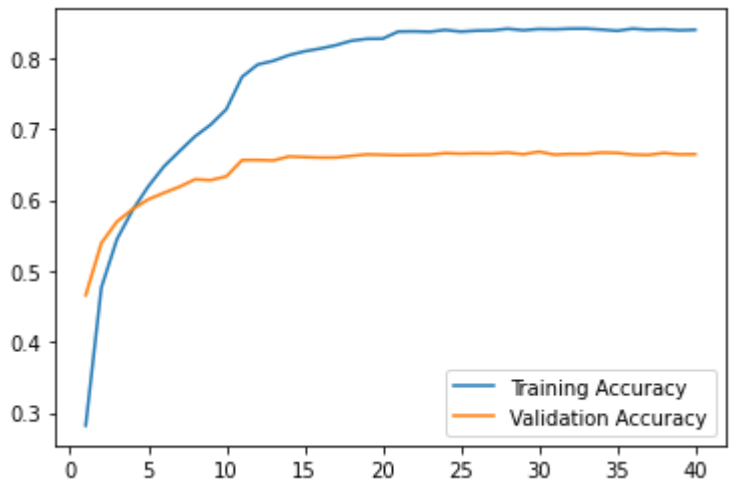
```
Epoch 0/39
----------
train Loss: 2.9805 Acc: 0.2821
val Loss: 1.9673 Acc: 0.4659

Epoch 1/39
----------
train Loss: 1.9197 Acc: 0.4765
val Loss: 1.6541 Acc: 0.5393

Epoch 2/39
----------
train Loss: 1.6344 Acc: 0.5448
val Loss: 1.5467 Acc: 0.5696

Epoch 3/39
----------
train Loss: 1.4637 Acc: 0.5858
val Loss: 1.4778 Acc: 0.5872

Epoch 4/39
----------
train Loss: 1.3276 Acc: 0.6191
val Loss: 1.4173 Acc: 0.6009

Epoch 5/39
----------
```

```
train Loss: 1.2237 Acc: 0.6472
val Loss: 1.4022 Acc: 0.6101

Epoch 6/39
----------
train Loss: 1.1355 Acc: 0.6689
val Loss: 1.3776 Acc: 0.6187

Epoch 7/39
----------
train Loss: 1.0574 Acc: 0.6899
val Loss: 1.3469 Acc: 0.6292

Epoch 8/39
----------
train Loss: 0.9913 Acc: 0.7064
val Loss: 1.3456 Acc: 0.6281

Epoch 9/39
----------
train Loss: 0.9206 Acc: 0.7280
val Loss: 1.3499 Acc: 0.6333

Epoch 10/39
----------
train Loss: 0.7562 Acc: 0.7736
val Loss: 1.2621 Acc: 0.6563

Epoch 11/39
----------
train Loss: 0.7004 Acc: 0.7910
val Loss: 1.2496 Acc: 0.6565

Epoch 12/39
----------
train Loss: 0.6808 Acc: 0.7963
val Loss: 1.2467 Acc: 0.6558

Epoch 13/39
----------
train Loss: 0.6589 Acc: 0.8042
val Loss: 1.2570 Acc: 0.6614

Epoch 14/39
----------
train Loss: 0.6346 Acc: 0.8097
val Loss: 1.2455 Acc: 0.6606

Epoch 15/39
----------
train Loss: 0.6253 Acc: 0.8136
val Loss: 1.2503 Acc: 0.6599

Epoch 16/39
----------
train Loss: 0.6111 Acc: 0.8183
val Loss: 1.2445 Acc: 0.6600

Epoch 17/39
----------
train Loss: 0.5881 Acc: 0.8246
val Loss: 1.2440 Acc: 0.6622

Epoch 18/39
----------
train Loss: 0.5769 Acc: 0.8273
val Loss: 1.2363 Acc: 0.6644

Epoch 19/39
```

```
----------
train Loss: 0.5722 Acc: 0.8275
val Loss: 1.2425 Acc: 0.6638

Epoch 20/39
----------
train Loss: 0.5488 Acc: 0.8376
val Loss: 1.2474 Acc: 0.6633

Epoch 21/39
----------
train Loss: 0.5457 Acc: 0.8379
val Loss: 1.2453 Acc: 0.6636

Epoch 22/39
----------
train Loss: 0.5428 Acc: 0.8372
val Loss: 1.2453 Acc: 0.6639

Epoch 23/39
----------
train Loss: 0.5427 Acc: 0.8399
val Loss: 1.2432 Acc: 0.6662

Epoch 24/39
----------
train Loss: 0.5431 Acc: 0.8375
val Loss: 1.2460 Acc: 0.6654

Epoch 25/39
----------
train Loss: 0.5421 Acc: 0.8388
val Loss: 1.2451 Acc: 0.6659

Epoch 26/39
----------
train Loss: 0.5431 Acc: 0.8393
val Loss: 1.2396 Acc: 0.6656

Epoch 27/39
----------
train Loss: 0.5340 Acc: 0.8415
val Loss: 1.2439 Acc: 0.6669

Epoch 28/39
----------
train Loss: 0.5347 Acc: 0.8394
val Loss: 1.2314 Acc: 0.6645

Epoch 29/39
----------
train Loss: 0.5339 Acc: 0.8412
val Loss: 1.2382 Acc: 0.6682

Epoch 30/39
----------
train Loss: 0.5317 Acc: 0.8407
val Loss: 1.2467 Acc: 0.6640

Epoch 31/39
----------
train Loss: 0.5282 Acc: 0.8416
val Loss: 1.2496 Acc: 0.6650

Epoch 32/39
----------
train Loss: 0.5340 Acc: 0.8417
val Loss: 1.2454 Acc: 0.6649
```

```
Epoch 33/39
----------
train Loss: 0.5327 Acc: 0.8403
val Loss: 1.2433 Acc: 0.6669

Epoch 34/39
----------
train Loss: 0.5394 Acc: 0.8386
val Loss: 1.2432 Acc: 0.6665

Epoch 35/39
----------
train Loss: 0.5287 Acc: 0.8417
val Loss: 1.2391 Acc: 0.6641

Epoch 36/39
----------
train Loss: 0.5315 Acc: 0.8401
val Loss: 1.2396 Acc: 0.6636

Epoch 37/39
----------
train Loss: 0.5354 Acc: 0.8407
val Loss: 1.2377 Acc: 0.6664

Epoch 38/39
----------
train Loss: 0.5364 Acc: 0.8393
val Loss: 1.2493 Acc: 0.6643

Epoch 39/39
----------
train Loss: 0.5351 Acc: 0.8400
val Loss: 1.2393 Acc: 0.6647

Training complete in 30m 24s
Best val Acc: 0.668200
```

In [ ]:
```
print_history(training_data)
```



(c) Next keeping all the hyperparameters (including multi-step learning rate schedule) same as before, change the learning rate to 0.01 and 0.1 uniformly for all the layers. This means keep all the layers at same learning rate. So you will be doing two experiments, one keeping learning rate of all layers at 0.01 and one with 0.1. Again finetune the model and report the final accuracy. How does the accuracy with the three learning rates compare ? Which learning rate gives you the best accuracy on the target dataset ?

In [ ]:
```
# experiment with initial learning rate of 0.1
resnet = models.resnet50(pretrained=True)
```

```
num_features = resnet.fc.in_features

# the fully connected layer with to classify the 100 classes

resnet.fc = nn.Linear(num_features,100)

# sending to device , Following tutorial.
resnet.to(device)

# learning rate to 0.01
optimizer = torch.optim.SGD(resnet.parameters(),lr = 0.1,momentum = 0.9)

# multistep learning rate schedule

step_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=step,

model_2 = train_model(resnet, criterion, optimizer, step_lr_scheduler, num_ep
```

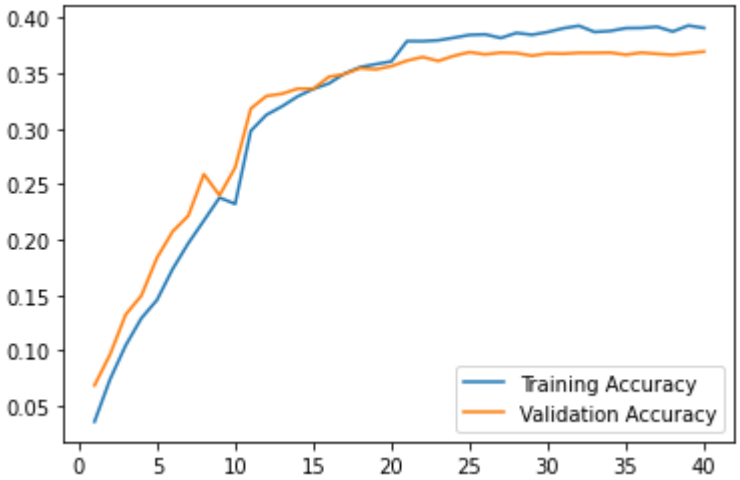
```
Epoch 0/39
----------
train Loss: 4.5729 Acc: 0.0360
val Loss: 4.1397 Acc: 0.0687

Epoch 1/39
----------
train Loss: 4.0441 Acc: 0.0743
val Loss: 3.9198 Acc: 0.0966

Epoch 2/39
----------
train Loss: 3.8253 Acc: 0.1048
val Loss: 3.7606 Acc: 0.1326

Epoch 3/39
----------
train Loss: 3.6899 Acc: 0.1292
val Loss: 3.6202 Acc: 0.1495

Epoch 4/39
----------
train Loss: 3.5889 Acc: 0.1456
val Loss: 3.4023 Acc: 0.1837

Epoch 5/39
----------
train Loss: 3.4295 Acc: 0.1735
val Loss: 3.3289 Acc: 0.2073

Epoch 6/39
----------
train Loss: 3.2909 Acc: 0.1966
val Loss: 3.1992 Acc: 0.2214

Epoch 7/39
----------
train Loss: 3.1721 Acc: 0.2172
val Loss: 3.0738 Acc: 0.2590

Epoch 8/39
----------
train Loss: 3.0759 Acc: 0.2378
val Loss: 3.1372 Acc: 0.2399

Epoch 9/39
----------
train Loss: 3.1035 Acc: 0.2322
val Loss: 2.9991 Acc: 0.2649
```

```
Epoch 10/39
----------
train Loss: 2.7728 Acc: 0.2980
val Loss: 2.7456 Acc: 0.3180

Epoch 11/39
----------
train Loss: 2.6924 Acc: 0.3124
val Loss: 2.7081 Acc: 0.3294

Epoch 12/39
----------
train Loss: 2.6592 Acc: 0.3201
val Loss: 2.6739 Acc: 0.3314

Epoch 13/39
----------
train Loss: 2.6221 Acc: 0.3291
val Loss: 2.6513 Acc: 0.3360

Epoch 14/39
----------
train Loss: 2.5908 Acc: 0.3357
val Loss: 2.6573 Acc: 0.3359

Epoch 15/39
----------
train Loss: 2.5648 Acc: 0.3407
val Loss: 2.5996 Acc: 0.3467

Epoch 16/39
----------
train Loss: 2.5332 Acc: 0.3496
val Loss: 2.6410 Acc: 0.3491

Epoch 17/39
----------
train Loss: 2.5008 Acc: 0.3555
val Loss: 2.5771 Acc: 0.3545

Epoch 18/39
----------
train Loss: 2.4780 Acc: 0.3581
val Loss: 2.6588 Acc: 0.3534

Epoch 19/39
----------
train Loss: 2.4537 Acc: 0.3606
val Loss: 2.6301 Acc: 0.3563

Epoch 20/39
----------
train Loss: 2.3948 Acc: 0.3789
val Loss: 2.5858 Acc: 0.3613

Epoch 21/39
----------
train Loss: 2.3792 Acc: 0.3788
val Loss: 2.5569 Acc: 0.3645

Epoch 22/39
----------
train Loss: 2.3748 Acc: 0.3796
val Loss: 2.5401 Acc: 0.3610

Epoch 23/39
----------
train Loss: 2.3712 Acc: 0.3819
```

```
val Loss: 2.5369 Acc: 0.3656

Epoch 24/39
----------
train Loss: 2.3670 Acc: 0.3844
val Loss: 2.5136 Acc: 0.3689

Epoch 25/39
----------
train Loss: 2.3603 Acc: 0.3849
val Loss: 2.5328 Acc: 0.3669

Epoch 26/39
----------
train Loss: 2.3629 Acc: 0.3817
val Loss: 2.5118 Acc: 0.3685

Epoch 27/39
----------
train Loss: 2.3568 Acc: 0.3862
val Loss: 2.5218 Acc: 0.3681

Epoch 28/39
----------
train Loss: 2.3505 Acc: 0.3846
val Loss: 2.5405 Acc: 0.3657

Epoch 29/39
----------
train Loss: 2.3469 Acc: 0.3870
val Loss: 2.5131 Acc: 0.3678

Epoch 30/39
----------
train Loss: 2.3328 Acc: 0.3904
val Loss: 2.5459 Acc: 0.3675

Epoch 31/39
----------
train Loss: 2.3278 Acc: 0.3927
val Loss: 2.5104 Acc: 0.3683

Epoch 32/39
----------
train Loss: 2.3370 Acc: 0.3871
val Loss: 2.5563 Acc: 0.3683

Epoch 33/39
----------
train Loss: 2.3398 Acc: 0.3880
val Loss: 2.5171 Acc: 0.3685

Epoch 34/39
----------
train Loss: 2.3312 Acc: 0.3906
val Loss: 2.5301 Acc: 0.3666

Epoch 35/39
----------
train Loss: 2.3322 Acc: 0.3907
val Loss: 2.5151 Acc: 0.3685

Epoch 36/39
----------
train Loss: 2.3341 Acc: 0.3918
val Loss: 2.5198 Acc: 0.3674

Epoch 37/39
----------
```

```
train Loss: 2.3375 Acc: 0.3875
val Loss: 2.5184 Acc: 0.3665

Epoch 38/39
----------
train Loss: 2.3282 Acc: 0.3929
val Loss: 2.5608 Acc: 0.3680

Epoch 39/39
----------
train Loss: 2.3295 Acc: 0.3907
val Loss: 2.5090 Acc: 0.3694

Training complete in 30m 8s
Best val Acc: 0.369400
```

In [ ]:
```
print_history(training_data)
```



In [ ]:
```
# experiment with initial learning rate of 0.01
resnet = models.resnet50(pretrained=True)

num_features = resnet.fc.in_features

# the fully connected layer with to classify the 100 classes

resnet.fc = nn.Linear(num_features,100)

# sending to device , Following tutorial.
resnet.to(device)

# learning rate to 0.01
optimizer = torch.optim.SGD(resnet.parameters(),lr = 0.01,momentum = 0.9)

# multistep learning rate schedule

step_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=step,

model_3 = train_model(resnet, criterion, optimizer, step_lr_scheduler, num_ep
```

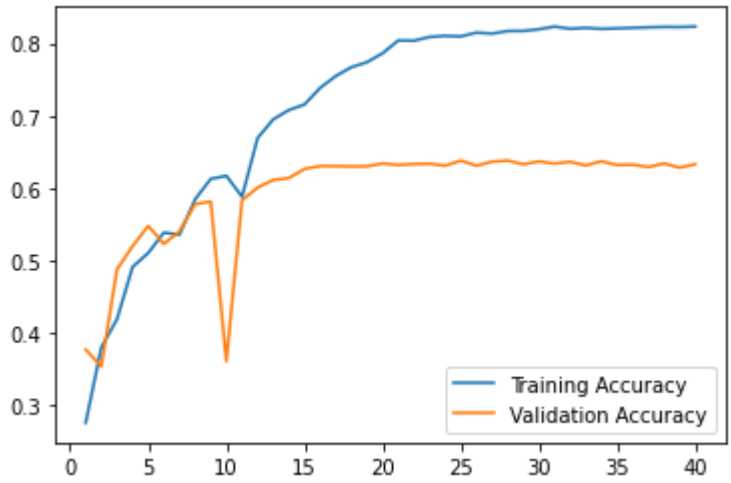
```
Epoch 0/39
----------
train Loss: 2.9122 Acc: 0.2767
val Loss: 3.0357 Acc: 0.3775

Epoch 1/39
----------
train Loss: 2.3793 Acc: 0.3805
val Loss: 2.5077 Acc: 0.3541

Epoch 2/39
```

```
----------
train Loss: 2.1850 Acc: 0.4193
val Loss: 1.9344 Acc: 0.4881

Epoch 3/39
----------
train Loss: 1.8426 Acc: 0.4915
val Loss: 1.8211 Acc: 0.5201

Epoch 4/39
----------
train Loss: 1.7610 Acc: 0.5107
val Loss: 1.7134 Acc: 0.5480

Epoch 5/39
----------
train Loss: 1.6488 Acc: 0.5388
val Loss: 1.8854 Acc: 0.5231

Epoch 6/39
----------
train Loss: 1.6694 Acc: 0.5361
val Loss: 1.9526 Acc: 0.5407

Epoch 7/39
----------
train Loss: 1.4535 Acc: 0.5847
val Loss: 1.5632 Acc: 0.5781

Epoch 8/39
----------
train Loss: 1.3451 Acc: 0.6130
val Loss: 1.5652 Acc: 0.5816

Epoch 9/39
----------
train Loss: 1.3354 Acc: 0.6172
val Loss: 3.1480 Acc: 0.3615

Epoch 10/39
----------
train Loss: 1.4642 Acc: 0.5883
val Loss: 1.5323 Acc: 0.5834

Epoch 11/39
----------
train Loss: 1.1412 Acc: 0.6693
val Loss: 1.6528 Acc: 0.6010

Epoch 12/39
----------
train Loss: 1.0321 Acc: 0.6951
val Loss: 1.6473 Acc: 0.6114

Epoch 13/39
----------
train Loss: 0.9907 Acc: 0.7079
val Loss: 1.5326 Acc: 0.6141

Epoch 14/39
----------
train Loss: 0.9608 Acc: 0.7155
val Loss: 1.4206 Acc: 0.6266

Epoch 15/39
----------
train Loss: 0.8828 Acc: 0.7384
val Loss: 1.5003 Acc: 0.6306
```

```
Epoch 16/39
----------
train Loss: 0.8198 Acc: 0.7548
val Loss: 1.6732 Acc: 0.6305

Epoch 17/39
----------
train Loss: 0.7791 Acc: 0.7670
val Loss: 1.6892 Acc: 0.6301

Epoch 18/39
----------
train Loss: 0.7411 Acc: 0.7739
val Loss: 1.7297 Acc: 0.6303

Epoch 19/39
----------
train Loss: 0.7072 Acc: 0.7862
val Loss: 1.5988 Acc: 0.6342

Epoch 20/39
----------
train Loss: 0.6528 Acc: 0.8039
val Loss: 1.9491 Acc: 0.6322

Epoch 21/39
----------
train Loss: 0.6440 Acc: 0.8035
val Loss: 1.8972 Acc: 0.6334

Epoch 22/39
----------
train Loss: 0.6271 Acc: 0.8087
val Loss: 1.8644 Acc: 0.6337

Epoch 23/39
----------
train Loss: 0.6215 Acc: 0.8102
val Loss: 1.8876 Acc: 0.6312

Epoch 24/39
----------
train Loss: 0.6189 Acc: 0.8094
val Loss: 1.6324 Acc: 0.6381

Epoch 25/39
----------
train Loss: 0.6125 Acc: 0.8146
val Loss: 1.8531 Acc: 0.6311

Epoch 26/39
----------
train Loss: 0.6102 Acc: 0.8133
val Loss: 1.7204 Acc: 0.6367

Epoch 27/39
----------
train Loss: 0.5976 Acc: 0.8169
val Loss: 1.5952 Acc: 0.6381

Epoch 28/39
----------
train Loss: 0.5971 Acc: 0.8169
val Loss: 1.8094 Acc: 0.6331

Epoch 29/39
----------
train Loss: 0.5881 Acc: 0.8193
val Loss: 1.8564 Acc: 0.6370
```

```
Epoch 30/39
----------
train Loss: 0.5794 Acc: 0.8230
val Loss: 1.9580 Acc: 0.6341

Epoch 31/39
----------
train Loss: 0.5834 Acc: 0.8199
val Loss: 1.9625 Acc: 0.6364

Epoch 32/39
----------
train Loss: 0.5821 Acc: 0.8212
val Loss: 1.7892 Acc: 0.6316

Epoch 33/39
----------
train Loss: 0.5813 Acc: 0.8198
val Loss: 1.7495 Acc: 0.6372

Epoch 34/39
----------
train Loss: 0.5822 Acc: 0.8205
val Loss: 2.1009 Acc: 0.6321

Epoch 35/39
----------
train Loss: 0.5832 Acc: 0.8213
val Loss: 1.8463 Acc: 0.6325

Epoch 36/39
----------
train Loss: 0.5816 Acc: 0.8220
val Loss: 2.2940 Acc: 0.6293

Epoch 37/39
----------
train Loss: 0.5773 Acc: 0.8226
val Loss: 1.7606 Acc: 0.6342

Epoch 38/39
----------
train Loss: 0.5802 Acc: 0.8224
val Loss: 2.3171 Acc: 0.6284

Epoch 39/39
----------
train Loss: 0.5748 Acc: 0.8230
val Loss: 1.9416 Acc: 0.6332

Training complete in 29m 56s
Best val Acc: 0.638100
```

In [ ]:
```
print_history(training_data)
```



Answer:

Using the initial learning rate of 0.001 gives me the best accuracy in this dataset.

1. When using a pretrained model as feature extractor, all the layers of the network are frozen except the final layer. Thus except the last layer, none of the inner layers' gradients are updated during backward pass with the target dataset. Since gradients do not need to be computed for most of the network, this is faster than finetuning.

(a) Now train only the last layer for 1, 0.1, 0.01, and 0.001 while keeping all the other hyperparameters and settings same as earlier for finetuning. Which learning rate gives you the best accuracy on the target dataset ? (6)

Answer:

Using 0.01 as an initial learning rate for this feature extracting technique is better than the other learning rates.

In [ ]:
```
# freeze resnet weights
# start with lr =1

epochs = 40
step = 10

resnet = models.resnet50(pretrained=True)


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


num_features = resnet.fc.in_features

# the fully connected layer with to classify the 100 classes

resnet.fc = nn.Linear(num_features,100)

# sending to device , Following tutorial.
resnet.to(device)

optimizer = torch.optim.SGD(resnet.parameters(),lr = 1,momentum = 0.9)

# multistep learning rate schedule
```

```
step_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=step,

model_4 = train_model(resnet, criterion, optimizer, step_lr_scheduler, num_ep
```

```
Epoch 0/39
----------
train Loss: 143.4770 Acc: 0.1329
val Loss: 162.8962 Acc: 0.1743

Epoch 1/39
----------
train Loss: 147.7955 Acc: 0.1704
val Loss: 149.1542 Acc: 0.1980

Epoch 2/39
----------
train Loss: 150.1334 Acc: 0.1837
val Loss: 164.4403 Acc: 0.2028

Epoch 3/39
----------
train Loss: 150.7827 Acc: 0.1924
val Loss: 180.3307 Acc: 0.1973

Epoch 4/39
----------
train Loss: 154.3752 Acc: 0.1976
val Loss: 199.1113 Acc: 0.1983

Epoch 5/39
----------
train Loss: 153.2598 Acc: 0.2020
val Loss: 171.8840 Acc: 0.2163

Epoch 6/39
----------
train Loss: 151.8256 Acc: 0.2091
val Loss: 154.5750 Acc: 0.2202

Epoch 7/39
----------
train Loss: 153.5606 Acc: 0.2135
val Loss: 168.4267 Acc: 0.2161

Epoch 8/39
----------
train Loss: 154.4218 Acc: 0.2143
val Loss: 167.7627 Acc: 0.2198

Epoch 9/39
----------
train Loss: 155.2577 Acc: 0.2165
val Loss: 158.8483 Acc: 0.2337

Epoch 10/39
----------
train Loss: 100.3730 Acc: 0.2744
val Loss: 103.2360 Acc: 0.2844

Epoch 11/39
----------
train Loss: 81.1296 Acc: 0.2933
val Loss: 82.9161 Acc: 0.2924

Epoch 12/39
----------
train Loss: 72.0054 Acc: 0.3014
val Loss: 79.7353 Acc: 0.2971
```

```
Epoch 13/39
----------
train Loss: 66.6975 Acc: 0.3006
val Loss: 77.6304 Acc: 0.2919

Epoch 14/39
----------
train Loss: 63.0025 Acc: 0.3056
val Loss: 71.3981 Acc: 0.2961

Epoch 15/39
----------
train Loss: 60.2794 Acc: 0.3007
val Loss: 64.8271 Acc: 0.2969

Epoch 16/39
----------
train Loss: 57.7845 Acc: 0.3045
val Loss: 64.3492 Acc: 0.2912

Epoch 17/39
----------
train Loss: 55.6869 Acc: 0.3047
val Loss: 60.0634 Acc: 0.2995

Epoch 18/39
----------
train Loss: 54.2679 Acc: 0.3054
val Loss: 61.3069 Acc: 0.2961

Epoch 19/39
----------
train Loss: 52.5448 Acc: 0.3062
val Loss: 59.7963 Acc: 0.2876

Epoch 20/39
----------
train Loss: 48.4841 Acc: 0.3160
val Loss: 57.9491 Acc: 0.3005

Epoch 21/39
----------
train Loss: 47.4499 Acc: 0.3180
val Loss: 56.7436 Acc: 0.3013

Epoch 22/39
----------
train Loss: 47.1501 Acc: 0.3164
val Loss: 55.2304 Acc: 0.3029

Epoch 23/39
----------
train Loss: 46.4051 Acc: 0.3173
val Loss: 57.0758 Acc: 0.3005

Epoch 24/39
----------
train Loss: 46.4972 Acc: 0.3164
val Loss: 55.4223 Acc: 0.3018

Epoch 25/39
----------
train Loss: 46.2899 Acc: 0.3190
val Loss: 54.4439 Acc: 0.3051

Epoch 26/39
----------
train Loss: 45.3302 Acc: 0.3255
val Loss: 55.0063 Acc: 0.3027
```

```
Epoch 27/39
----------
train Loss: 45.1974 Acc: 0.3208
val Loss: 54.8542 Acc: 0.3022

Epoch 28/39
----------
train Loss: 45.0135 Acc: 0.3211
val Loss: 53.6349 Acc: 0.3067

Epoch 29/39
----------
train Loss: 44.7260 Acc: 0.3228
val Loss: 58.3780 Acc: 0.2972

Epoch 30/39
----------
train Loss: 44.5888 Acc: 0.3244
val Loss: 51.1387 Acc: 0.3060

Epoch 31/39
----------
train Loss: 44.5101 Acc: 0.3213
val Loss: 54.0367 Acc: 0.3022

Epoch 32/39
----------
train Loss: 44.3815 Acc: 0.3238
val Loss: 53.6597 Acc: 0.3056

Epoch 33/39
----------
train Loss: 44.4679 Acc: 0.3225
val Loss: 53.2656 Acc: 0.3040

Epoch 34/39
----------
train Loss: 44.0807 Acc: 0.3245
val Loss: 52.7803 Acc: 0.3045

Epoch 35/39
----------
train Loss: 44.0033 Acc: 0.3235
val Loss: 52.9238 Acc: 0.3004

Epoch 36/39
----------
train Loss: 44.3294 Acc: 0.3224
val Loss: 52.5874 Acc: 0.3021

Epoch 37/39
----------
train Loss: 44.2153 Acc: 0.3232
val Loss: 51.8293 Acc: 0.3012

Epoch 38/39
----------
train Loss: 43.8193 Acc: 0.3251
val Loss: 52.4245 Acc: 0.3010

Epoch 39/39
----------
train Loss: 44.0528 Acc: 0.3262
val Loss: 52.8796 Acc: 0.3022

Training complete in 21m 11s
Best val Acc: 0.306700
```

In [ ]:
```
print_history(training_data)
```

In [ ]:
```
#frozen weights
#initial lr = 0.1

resnet = models.resnet50(pretrained=True)


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


num_features = resnet.fc.in_features

# the fully connected layer with to classify the 100 classes

resnet.fc = nn.Linear(num_features,100)

# sending to device , Following tutorial.
resnet.to(device)

# learning rate to 0.01
optimizer = torch.optim.SGD(resnet.parameters(),lr = 0.1,momentum = 0.9)

# multistep learning rate schedule

step_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=step,

model_5 = train_model(resnet, criterion, optimizer, step_lr_scheduler, num_ep
```

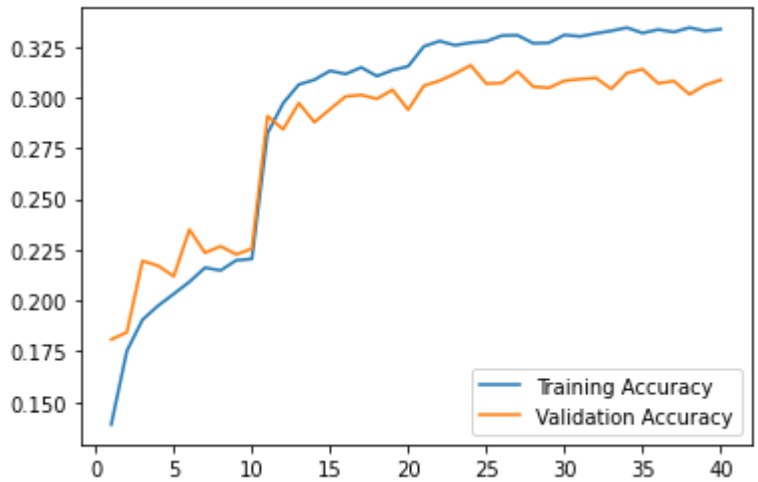
```
Epoch 0/39
----------
train Loss: 12.5057 Acc: 0.1389
val Loss: 13.9496 Acc: 0.1808

Epoch 1/39
----------
train Loss: 13.8288 Acc: 0.1752
val Loss: 15.5083 Acc: 0.1843

Epoch 2/39
----------
train Loss: 13.8938 Acc: 0.1905
val Loss: 14.1488 Acc: 0.2195

Epoch 3/39
----------
train Loss: 14.1809 Acc: 0.1974
val Loss: 15.1230 Acc: 0.2171
```

```
Epoch 4/39
----------
train Loss: 14.2566 Acc: 0.2032
val Loss: 14.9182 Acc: 0.2119

Epoch 5/39
----------
train Loss: 14.3325 Acc: 0.2092
val Loss: 13.7668 Acc: 0.2349

Epoch 6/39
----------
train Loss: 14.2649 Acc: 0.2162
val Loss: 15.1934 Acc: 0.2235

Epoch 7/39
----------
train Loss: 14.3252 Acc: 0.2148
val Loss: 15.5567 Acc: 0.2267

Epoch 8/39
----------
train Loss: 14.2849 Acc: 0.2198
val Loss: 15.4193 Acc: 0.2227

Epoch 9/39
----------
train Loss: 14.3379 Acc: 0.2205
val Loss: 15.9494 Acc: 0.2255

Epoch 10/39
----------
train Loss: 9.4718 Acc: 0.2823
val Loss: 9.1202 Acc: 0.2908

Epoch 11/39
----------
train Loss: 7.6862 Acc: 0.2971
val Loss: 8.9067 Acc: 0.2843

Epoch 12/39
----------
train Loss: 6.8867 Acc: 0.3063
val Loss: 7.3923 Acc: 0.2972

Epoch 13/39
----------
train Loss: 6.4151 Acc: 0.3087
val Loss: 7.3033 Acc: 0.2878

Epoch 14/39
----------
train Loss: 6.0812 Acc: 0.3131
val Loss: 7.1314 Acc: 0.2942

Epoch 15/39
----------
train Loss: 5.8378 Acc: 0.3115
val Loss: 6.5679 Acc: 0.3005

Epoch 16/39
----------
train Loss: 5.6018 Acc: 0.3148
val Loss: 6.4157 Acc: 0.3012

Epoch 17/39
----------
train Loss: 5.4932 Acc: 0.3105
val Loss: 6.1225 Acc: 0.2993
```

```
Epoch 18/39
----------
train Loss: 5.3247 Acc: 0.3134
val Loss: 6.0453 Acc: 0.3037

Epoch 19/39
----------
train Loss: 5.2236 Acc: 0.3153
val Loss: 6.2967 Acc: 0.2939

Epoch 20/39
----------
train Loss: 4.8374 Acc: 0.3251
val Loss: 5.7248 Acc: 0.3057

Epoch 21/39
----------
train Loss: 4.7872 Acc: 0.3277
val Loss: 5.7482 Acc: 0.3082

Epoch 22/39
----------
train Loss: 4.7587 Acc: 0.3257
val Loss: 5.5032 Acc: 0.3117

Epoch 23/39
----------
train Loss: 4.7334 Acc: 0.3270
val Loss: 5.4405 Acc: 0.3158

Epoch 24/39
----------
train Loss: 4.6964 Acc: 0.3276
val Loss: 5.4367 Acc: 0.3068

Epoch 25/39
----------
train Loss: 4.6921 Acc: 0.3304
val Loss: 5.4490 Acc: 0.3071

Epoch 26/39
----------
train Loss: 4.6517 Acc: 0.3306
val Loss: 5.3540 Acc: 0.3128

Epoch 27/39
----------
train Loss: 4.6529 Acc: 0.3267
val Loss: 5.4695 Acc: 0.3053

Epoch 28/39
----------
train Loss: 4.6266 Acc: 0.3268
val Loss: 5.3619 Acc: 0.3047

Epoch 29/39
----------
train Loss: 4.5875 Acc: 0.3308
val Loss: 5.4285 Acc: 0.3082

Epoch 30/39
----------
train Loss: 4.5748 Acc: 0.3300
val Loss: 5.2833 Acc: 0.3090

Epoch 31/39
----------
train Loss: 4.5469 Acc: 0.3315
```

```
val Loss: 5.5145 Acc: 0.3096

Epoch 32/39
----------
train Loss: 4.5406 Acc: 0.3328
val Loss: 5.3896 Acc: 0.3042

Epoch 33/39
----------
train Loss: 4.5206 Acc: 0.3343
val Loss: 5.2449 Acc: 0.3120

Epoch 34/39
----------
train Loss: 4.5279 Acc: 0.3317
val Loss: 5.2060 Acc: 0.3139

Epoch 35/39
----------
train Loss: 4.5264 Acc: 0.3334
val Loss: 5.4548 Acc: 0.3069

Epoch 36/39
----------
train Loss: 4.5268 Acc: 0.3322
val Loss: 5.4684 Acc: 0.3081

Epoch 37/39
----------
train Loss: 4.5018 Acc: 0.3344
val Loss: 5.4696 Acc: 0.3015

Epoch 38/39
----------
train Loss: 4.5008 Acc: 0.3327
val Loss: 5.4384 Acc: 0.3061

Epoch 39/39
----------
train Loss: 4.5391 Acc: 0.3336
val Loss: 5.3015 Acc: 0.3086

Training complete in 20m 51s
Best val Acc: 0.315800
```

In [ ]:
```
print_history(training_data)
```



In [ ]:
```
#frozen weights
#initial lr =  0.01

resnet = models.resnet50(pretrained=True)
```

```
for param in resnet.parameters():
    param.requires_grad = False


num_features = resnet.fc.in_features

# the fully connected layer with to classify the 100 classes

resnet.fc = nn.Linear(num_features,100)

# sending to device , Following tutorial.
resnet.to(device)

# learning rate to 0.01
optimizer = torch.optim.SGD(resnet.parameters(),lr = 0.01,momentum = 0.9)

# multistep learning rate schedule

step_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=step,

model_6 = train_model(resnet, criterion, optimizer, step_lr_scheduler, num_ep
```

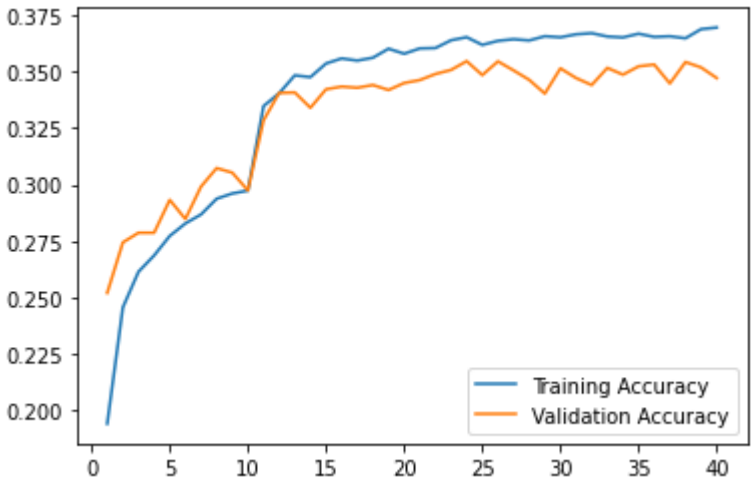
```
Epoch 0/39
----------
train Loss: 3.6214 Acc: 0.1938
val Loss: 3.3713 Acc: 0.2519

Epoch 1/39
----------
train Loss: 3.3294 Acc: 0.2455
val Loss: 3.2483 Acc: 0.2743

Epoch 2/39
----------
train Loss: 3.2322 Acc: 0.2613
val Loss: 3.2865 Acc: 0.2786

Epoch 3/39
----------
train Loss: 3.1951 Acc: 0.2685
val Loss: 3.3379 Acc: 0.2786

Epoch 4/39
----------
train Loss: 3.1509 Acc: 0.2772
val Loss: 3.1465 Acc: 0.2932

Epoch 5/39
----------
train Loss: 3.1085 Acc: 0.2827
val Loss: 3.2295 Acc: 0.2847

Epoch 6/39
----------
train Loss: 3.0741 Acc: 0.2867
val Loss: 3.1025 Acc: 0.2991

Epoch 7/39
----------
train Loss: 3.0586 Acc: 0.2937
val Loss: 3.1171 Acc: 0.3073

Epoch 8/39
----------
train Loss: 3.0400 Acc: 0.2960
val Loss: 3.1338 Acc: 0.3053
```

```
Epoch 9/39
----------
train Loss: 3.0236 Acc: 0.2973
val Loss: 3.2172 Acc: 0.2974

Epoch 10/39
----------
train Loss: 2.7576 Acc: 0.3348
val Loss: 2.8699 Acc: 0.3285

Epoch 11/39
----------
train Loss: 2.6943 Acc: 0.3405
val Loss: 2.7751 Acc: 0.3407

Epoch 12/39
----------
train Loss: 2.6511 Acc: 0.3485
val Loss: 2.7295 Acc: 0.3409

Epoch 13/39
----------
train Loss: 2.6473 Acc: 0.3477
val Loss: 2.8740 Acc: 0.3341

Epoch 14/39
----------
train Loss: 2.6247 Acc: 0.3538
val Loss: 2.7516 Acc: 0.3423

Epoch 15/39
----------
train Loss: 2.6108 Acc: 0.3560
val Loss: 2.7522 Acc: 0.3435

Epoch 16/39
----------
train Loss: 2.6065 Acc: 0.3551
val Loss: 2.7464 Acc: 0.3430

Epoch 17/39
----------
train Loss: 2.6022 Acc: 0.3563
val Loss: 2.7480 Acc: 0.3442

Epoch 18/39
----------
train Loss: 2.5855 Acc: 0.3603
val Loss: 2.7519 Acc: 0.3420

Epoch 19/39
----------
train Loss: 2.5834 Acc: 0.3581
val Loss: 2.7008 Acc: 0.3451

Epoch 20/39
----------
train Loss: 2.5768 Acc: 0.3604
val Loss: 2.7315 Acc: 0.3464

Epoch 21/39
----------
train Loss: 2.5679 Acc: 0.3606
val Loss: 2.6864 Acc: 0.3491

Epoch 22/39
----------
train Loss: 2.5637 Acc: 0.3641
```

```
val Loss: 2.7045 Acc: 0.3509

Epoch 23/39
----------
train Loss: 2.5567 Acc: 0.3654
val Loss: 2.6762 Acc: 0.3548

Epoch 24/39
----------
train Loss: 2.5691 Acc: 0.3620
val Loss: 2.7048 Acc: 0.3486

Epoch 25/39
----------
train Loss: 2.5692 Acc: 0.3638
val Loss: 2.6934 Acc: 0.3547

Epoch 26/39
----------
train Loss: 2.5643 Acc: 0.3645
val Loss: 2.6609 Acc: 0.3507

Epoch 27/39
----------
train Loss: 2.5580 Acc: 0.3640
val Loss: 2.7403 Acc: 0.3465

Epoch 28/39
----------
train Loss: 2.5601 Acc: 0.3659
val Loss: 2.8359 Acc: 0.3404

Epoch 29/39
----------
train Loss: 2.5559 Acc: 0.3654
val Loss: 2.7167 Acc: 0.3516

Epoch 30/39
----------
train Loss: 2.5525 Acc: 0.3667
val Loss: 2.6974 Acc: 0.3473

Epoch 31/39
----------
train Loss: 2.5520 Acc: 0.3672
val Loss: 2.7550 Acc: 0.3442

Epoch 32/39
----------
train Loss: 2.5535 Acc: 0.3657
val Loss: 2.6756 Acc: 0.3518

Epoch 33/39
----------
train Loss: 2.5466 Acc: 0.3653
val Loss: 2.7008 Acc: 0.3488

Epoch 34/39
----------
train Loss: 2.5533 Acc: 0.3670
val Loss: 2.6784 Acc: 0.3525

Epoch 35/39
----------
train Loss: 2.5475 Acc: 0.3656
val Loss: 2.7039 Acc: 0.3533

Epoch 36/39
----------
```

```
train Loss: 2.5516 Acc: 0.3658
val Loss: 2.6948 Acc: 0.3449

Epoch 37/39
----------
train Loss: 2.5487 Acc: 0.3650
val Loss: 2.6675 Acc: 0.3544

Epoch 38/39
----------
train Loss: 2.5467 Acc: 0.3690
val Loss: 2.6722 Acc: 0.3520

Epoch 39/39
----------
train Loss: 2.5405 Acc: 0.3697
val Loss: 2.6949 Acc: 0.3473

Training complete in 21m 24s
Best val Acc: 0.354800
```

In [ ]:
```
print_history(training_data)
```



In [ ]:
```
#frozen weights
#initial lr =  0.01

resnet = models.resnet50(pretrained=True)


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


num_features = resnet.fc.in_features

# the fully connected layer with to classify the 100 classes

resnet.fc = nn.Linear(num_features,100)

# sending to device , Following tutorial.
resnet.to(device)

# learning rate to 0.01
optimizer = torch.optim.SGD(resnet.parameters(),lr = 0.001,momentum = 0.9)

# multistep learning rate schedule

step_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=step,
```

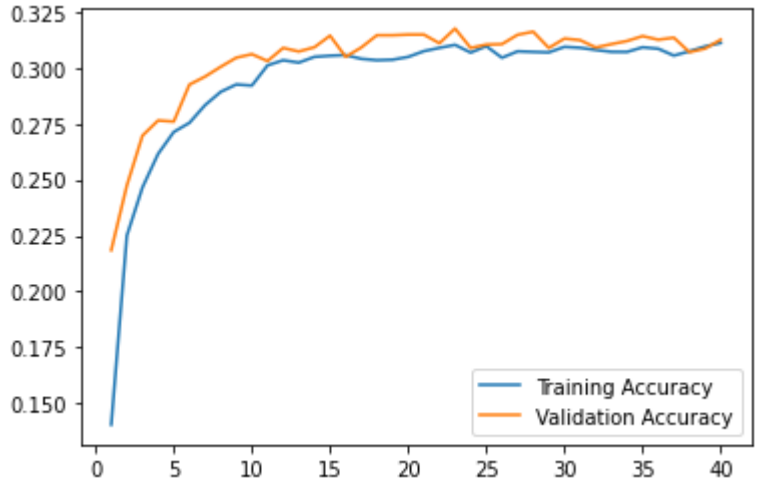
```
model_7 = train_model(resnet, criterion, optimizer, step_lr_scheduler, num_ep
```

```
Epoch 0/39
----------
train Loss: 4.0293 Acc: 0.1400
val Loss: 3.5791 Acc: 0.2184

Epoch 1/39
----------
train Loss: 3.4768 Acc: 0.2249
val Loss: 3.3250 Acc: 0.2475

Epoch 2/39
----------
train Loss: 3.2979 Acc: 0.2466
val Loss: 3.2536 Acc: 0.2699

Epoch 3/39
----------
train Loss: 3.1983 Acc: 0.2617
val Loss: 3.1240 Acc: 0.2767

Epoch 4/39
----------
train Loss: 3.1245 Acc: 0.2715
val Loss: 3.1349 Acc: 0.2762

Epoch 5/39
----------
train Loss: 3.0840 Acc: 0.2756
val Loss: 3.0203 Acc: 0.2928

Epoch 6/39
----------
train Loss: 3.0456 Acc: 0.2836
val Loss: 2.9879 Acc: 0.2964

Epoch 7/39
----------
train Loss: 3.0135 Acc: 0.2895
val Loss: 2.9827 Acc: 0.3008

Epoch 8/39
----------
train Loss: 2.9836 Acc: 0.2929
val Loss: 2.9382 Acc: 0.3049

Epoch 9/39
----------
train Loss: 2.9724 Acc: 0.2924
val Loss: 2.9271 Acc: 0.3065

Epoch 10/39
----------
train Loss: 2.9353 Acc: 0.3015
val Loss: 2.9683 Acc: 0.3034

Epoch 11/39
----------
train Loss: 2.9171 Acc: 0.3038
val Loss: 2.9344 Acc: 0.3093

Epoch 12/39
----------
train Loss: 2.9172 Acc: 0.3027
val Loss: 2.9348 Acc: 0.3077

Epoch 13/39
```

```
----------
train Loss: 2.9131 Acc: 0.3053
val Loss: 2.9104 Acc: 0.3096

Epoch 14/39
----------
train Loss: 2.9122 Acc: 0.3058
val Loss: 2.8823 Acc: 0.3148

Epoch 15/39
----------
train Loss: 2.9143 Acc: 0.3061
val Loss: 2.9834 Acc: 0.3053

Epoch 16/39
----------
train Loss: 2.9095 Acc: 0.3044
val Loss: 2.9511 Acc: 0.3095

Epoch 17/39
----------
train Loss: 2.9130 Acc: 0.3038
val Loss: 2.9054 Acc: 0.3150

Epoch 18/39
----------
train Loss: 2.9069 Acc: 0.3040
val Loss: 2.8824 Acc: 0.3150

Epoch 19/39
----------
train Loss: 2.9048 Acc: 0.3053
val Loss: 2.8793 Acc: 0.3153

Epoch 20/39
----------
train Loss: 2.9011 Acc: 0.3078
val Loss: 2.8868 Acc: 0.3153

Epoch 21/39
----------
train Loss: 2.8973 Acc: 0.3093
val Loss: 2.9031 Acc: 0.3114

Epoch 22/39
----------
train Loss: 2.8924 Acc: 0.3107
val Loss: 2.8785 Acc: 0.3180

Epoch 23/39
----------
train Loss: 2.8923 Acc: 0.3072
val Loss: 2.9201 Acc: 0.3093

Epoch 24/39
----------
train Loss: 2.8949 Acc: 0.3101
val Loss: 2.9341 Acc: 0.3108

Epoch 25/39
----------
train Loss: 2.9012 Acc: 0.3049
val Loss: 2.9150 Acc: 0.3110

Epoch 26/39
----------
train Loss: 2.8969 Acc: 0.3078
val Loss: 2.8871 Acc: 0.3152
```

```
Epoch 27/39
----------
train Loss: 2.8879 Acc: 0.3075
val Loss: 2.8742 Acc: 0.3166

Epoch 28/39
----------
train Loss: 2.9049 Acc: 0.3073
val Loss: 2.9292 Acc: 0.3093

Epoch 29/39
----------
train Loss: 2.8967 Acc: 0.3098
val Loss: 2.8895 Acc: 0.3135

Epoch 30/39
----------
train Loss: 2.8925 Acc: 0.3094
val Loss: 2.9097 Acc: 0.3128

Epoch 31/39
----------
train Loss: 2.9000 Acc: 0.3084
val Loss: 2.9157 Acc: 0.3095

Epoch 32/39
----------
train Loss: 2.8916 Acc: 0.3075
val Loss: 2.9124 Acc: 0.3110

Epoch 33/39
----------
train Loss: 2.8966 Acc: 0.3075
val Loss: 2.9012 Acc: 0.3124

Epoch 34/39
----------
train Loss: 2.8922 Acc: 0.3096
val Loss: 2.9019 Acc: 0.3146

Epoch 35/39
----------
train Loss: 2.8919 Acc: 0.3089
val Loss: 2.9353 Acc: 0.3130

Epoch 36/39
----------
train Loss: 2.8969 Acc: 0.3059
val Loss: 2.8946 Acc: 0.3139

Epoch 37/39
----------
train Loss: 2.8976 Acc: 0.3078
val Loss: 2.9694 Acc: 0.3074

Epoch 38/39
----------
train Loss: 2.8903 Acc: 0.3100
val Loss: 2.9060 Acc: 0.3090

Epoch 39/39
----------
train Loss: 2.8909 Acc: 0.3115
val Loss: 2.9231 Acc: 0.3130

Training complete in 21m 30s
Best val Acc: 0.318000
```

In [ ]:
```
print_history(training_data)
```



(b) For your target dataset find the best final accuracy (across all the learning rates) from the two transfer learning approaches. Which approach and learning rate is the winner? Provide a plausible explanation to support your observation

Answer:

Using the finetuning transfer learning approches we can achieve better accuracy values. Given that we are picking a goog starting learning rate.

We get better accuracy on the finetuning approach because we still get to training the neurons with our dataset, while with the feature extraction approach we are just training the last layer of the model.

In [ ]: