P1

November 28, 2019

[ ]:
```python
# License: BSD
# Author: Sasank Chilamkurthy

from __future__ import print_function, division

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

plt.ion()   # interactive mode
```

0.1 Load Data

Target dataset- VGG Flowers #Classes : 102 #Total number of images: 1002 Approx 10 images per class

[2]:
```python
# Data augmentation and normalization for training
# Just normalization for validation
train_transforms = transforms.Compose([
        transforms.Resize((128,128)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

val_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

```
#image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
#                                           data_transforms[x])
#                   for x in ['train', 'val']}
#dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4,
#                                              shuffle=True, num_workers=4)
#               for x in ['train', 'val']}

trainset = torchvision.datasets.ImageFolder(root='train',␣
 ↪transform=train_transforms)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=2)

valset = torchvision.datasets.ImageFolder(root='val', transform=val_transforms)
valloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=False, num_workers=2)

#dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
#class_names = image_datasets['train'].classes

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

0.2 Training the model

[3]:
```
def train_model(model, criterion, optimizer, scheduler, num_epochs=200):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    accuracy = []

    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

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs,labels in trainloader:

                    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
                        loss.backward()
                        optimizer.step()

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

                scheduler.step()

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

                running_loss = 0.0
                running_corrects = 0

                # Iterate over data.
                for inputs,labels in valloader:

                    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)

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

```
                epoch_loss = running_loss / 1020
                epoch_acc = running_corrects.double() / 1020

                if epoch%25 == 0:
                    print("Epoch "+str(epoch)+" complete, accuracy:␣
↪"+str(epoch_acc))
                #print(epoch_acc)
                accuracy.append((epoch,epoch_acc.cpu().numpy()))

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

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

    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,accuracy)
```

0.3 Finetuning the convnet

Load a pretrained model and reset final fully connected layer.

[4]:
```
model_ft = models.resnet50(pretrained=True)
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 2.
# Alternatively, it can be generalized to nn.Linear(num_ftrs, len(class_names)).
model_ft.fc = nn.Linear(num_ftrs, 102)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=40, gamma=0.1)
```

```
[ ]: model_ft, accuracy = train_model(model_ft, criterion, optimizer_ft,␣
      ↪exp_lr_scheduler,
                             num_epochs=150)
```

```
Epoch 0 complete, accuracy: tensor(0.0588, device='cuda:0', dtype=torch.float64)
Epoch 25 complete, accuracy: tensor(0.0588, device='cuda:0',
dtype=torch.float64)
```

```
[7]: accuracy = np.array(accuracy).ravel().reshape(100,2)
     np.savetxt("finetune_train_128_0.001.csv", accuracy, delimiter=',')
```

0.4 ConvNet as fixed feature extractor

Here, we need to freeze all the network except the final layer. We need to set `requires_grad == False` to freeze the parameters so that the gradients are not computed in `backward()`.

You can read more about this in the documentation here `<https://pytorch.org/docs/notes/autograd.html#excluding-subgraphs-from-backward>`__.

```
[4]: model_conv = torchvision.models.resnet50(pretrained=True)
     for param in model_conv.parameters():
         param.requires_grad = False

     # Parameters of newly constructed modules have requires_grad=True by default
     num_ftrs = model_conv.fc.in_features
     model_conv.fc = nn.Linear(num_ftrs, 102)

     model_conv = model_conv.to(device)

     criterion = nn.CrossEntropyLoss()

     # Observe that only parameters of final layer are being optimized as
     # opposed to before.
     optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

     # Decay LR by a factor of 0.1 every 7 epochs
     exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=25, gamma=0.1)
```

Train and evaluate ^^^^^^^^^^^^^^^^^^

On CPU this will take about half the time compared to previous scenario. This is expected as gradients don't need to be computed for most of the network. However, forward does need to be computed.

```
[5]: model_conv, accuracy_conv = train_model(model_conv, criterion, optimizer_conv,
                              exp_lr_scheduler, num_epochs=100)
```

```
Epoch 0 complete, accuracy: tensor(0.0010, device='cuda:0', dtype=torch.float64)
Epoch 25 complete, accuracy: tensor(0.0510, device='cuda:0',
dtype=torch.float64)
Epoch 50 complete, accuracy: tensor(0.0578, device='cuda:0',
dtype=torch.float64)
Epoch 75 complete, accuracy: tensor(0.0569, device='cuda:0',
dtype=torch.float64)
Training complete in 2m 56s
Best val Acc: 0.057843
```

[9]:
```
accuracy_conv = np.array(accuracy_conv).ravel().reshape(100,2)
np.savetxt("featext_train_128_.00001.csv", accuracy_conv, delimiter=',')
```

0.5 Overall for both cases, the validation accuracy is poor (~6%). This might be because of the size of the dataset- each class has hardly 10 images to train on. Thus, the model is not accurate overall.

Finetuning: For the three starting learning rates explored, l = 0.0001 appears to give the best performance. Feature Extractor: For the three starting learning rates explored, l = 0.001 appears to give the best performance.

Comparison between FineTuning, Feature extraction: Feature extraction performs better than FineTuning. This can be attributed to the fact that while FineTuning, we propagate the loss through to all the layers. Thus, the features the model learnt while training on the original ImageNet dataset are lost. The new set of features do not have enough data to predict properly. However, while we use it as a feature extractor, the first n-1 layers are undisturbed- they effeciently extract the features, based on which the last layer acts similar to a regression model to train on. Thus, it performs better

[ ]:
```

```