Final Report for Term Project

2021227387 김동희

12/12/2021

목표

- Cifar10 image classification
- Training set : 50000
- Test set : 10000
- Top - 1 rank accuracy

Code Review

- 기본적으로는 교수님께서 github에 올려주신 코드 중 resnet.ipynb 코드를 사용했음.

1. Import

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.autograd import Variable
from torchvision import models
import numpy as np
import torchvision
from torchvision import datasets, models, transforms

import time
import os
```

- Deep Learning을 위한 pytorch package를 import하고 필요한 module들 각각 import하였음.
- 시간 측정을위한 내장함수 time과 path manage를 위한 os 내장함수도 import 하였음.

2. HyperParameter & Dataloader

```
batch_size = 50
learning_rate = 0.01
root_dir = './'
default_directory = './saved'
# Data Augmentation
transform = transforms.Compose([
    transforms.Resize(224,interpolation=2),
    #transforms.RandomCrop(32, padding=4),
    #transforms.RandomHorizontalFlip(),
    #transforms.RandomRotation(10),
    #transforms.RandomAffine(0, shear=10, scale=(0.8, 1.2)),
    #transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.4914, 0.4824, 0.4467),std=(0.2471, 0.2436, 0.2616))
])
```

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(dataset=trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=0)

testset=torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(dataset=testset, batch_size=batch_size,
                                         shuffle=False, num_workers=0)


class_names = ('plane', 'car', 'bird', 'cat','deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

- HyperParameter : Batch size & learning rate
- Directory 변수 정의
- Data Augmentation : 많은 Augmentation방법을 썼봤지만 Resize, Normalization을 한 것이 가장 간단하면서도 학습이 잘되었음. (후자에 서술)
- Traninset & Testset DataLoader : Transform을 적용시킨 trainset과 testset을 만들고 이는 torch에 미리 있던 Cifar10 데이터를 사용.

3. Device Configuration

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

- GPU spec : Geforce 1650
- Cuda version : 11.x

4. Model Definition

```
# Get Pretrained Model

#model_ft = models.resnet50(pretrained=True)
#model_ft = models.densenet169(pretrained=True)
model_ft = models.resnet18(pretrained=True)

# Change Fc Layer or classifier layer for our problem

'''model_ft.classifier = nn.Sequential(
    nn.Linear(25088, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5,inplace=False),
    nn.Linear(4096, 1024,bias=True),
    nn.ReLU(inplace=True),
    nn.Dropout(0.4,inplace=False),
    nn.Linear(1024, 10,bias=True),
)'''
model_ft.fc =  nn.Linear(512,10,bias=True)

# Freezing

'''
for param in model_ft.parameters():
    param.requires_grad = False

for param in model_ft.fc.parameters():
    param.requires_grad = True
'''

# To GPU
```

```
model_ft = model_ft.to(device)

# Optimizer
criterion = nn.CrossEntropyLoss().to(device)

# Resnet 18
optimizer_ft = optim.SGD(model_ft.parameters(), lr=learning_rate,momentum=0.9)
# for resnet50 or other
optimizer_ft = optim.Adam(model_ft.parameters(), lr=learning_rate)
```

- Get pretrained Model : 주석친 부분의 모델들도 실험해봤었지만 실질적으로 결과가 어느정도 나왔던 모델은 resnet 50과 resnet 18이다. 그 중 가장 정확도가 높게 나왔던 것은 resnet 18이다.
- Fc layer을 여러가지로 조정해보았다. Fc layer자체를 deep하게 만들기도 해보았고, 가장 기초적인 single layer로 하는 것이 가장 정확도가 높았다.
- Freezing 부분은 Transfer Learning시 layer의 파라미터를 pretrained된 그대로 사용할 지, 아니면 내 학습 데이터셋을 학습을 시킬지를 정하는 부분이었다. fc layer밖에도 앞단인 convolutional layer의 파라미터도 끄고 닫을 수 있다.
- Optimizer는 처음 여러가지 모델에 대해서는 Adam으로 진행했다가 마지막 Resnet 18모델에서는 SGD를 썼다.

5. Train & Test & save checkpoint & load checkpoint function

```
def train(epoch):
    model_ft.train()
    train_loss = 0
    total = 0
    correct = 0
    for batch_idx, (data, target) in enumerate(trainloader):
        data, target = Variable(data.to(device)), Variable(target.to(device))

        optimizer_ft.zero_grad()
        output = model_ft(data).to(device)
        loss = criterion(output, target).to(device)

        loss.backward()
        optimizer_ft.step()

        train_loss += loss.item()
        _, predicted = torch.max(output.data, 1)

        total += target.size(0)
        correct += predicted.eq(target.data).cpu().sum()

        if batch_idx % 100 == 0:
            print('Epoch: {} | Batch_idx: {} |  Loss: ({:.4f}) | Acc: ({:.2f}%) ({}/{})'
                  .format(epoch, batch_idx, train_loss / (batch_idx + 1), 100. * correct / total, correct, total))


def save_checkpoint(directory, state, filename='latest.tar.gz'):

    if not os.path.exists(directory):
        os.makedirs(directory)

    model_filename = os.path.join(directory, filename)
    torch.save(state, model_filename)
    print("=> saving checkpoint")

def load_checkpoint(directory, filename='latest.tar.gz'):

    model_filename = os.path.join(directory, filename)
    if os.path.exists(model_filename):
        print("=> loading checkpoint")
        state = torch.load(model_filename)
        return state
    else:
        return None
```

```
def test():
    model_ft.eval()
    test_loss = 0
    correct = 0
    total = 0
    for batch_idx, (data, target) in enumerate(testloader):
        data, target = Variable(data.to(device)), Variable(target.to(device))
        outputs = model_ft(data)
        loss = criterion(outputs, target)

        test_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += target.size(0)
        correct += predicted.eq(target.data).cpu().sum()
    print('# TEST : Loss: ({:.4f}) | Acc: ({:.2f}%) ({}/{})'
          .format(test_loss / (batch_idx + 1), 100. * correct / total, correct, total))

    return 100 * correct/total
```

- 기본적으로 pytorch를 심화학습의 기초 과목에서 처음 배웠기 때문에 이 부분에 어려움을 겪어 교수님께서 제공해주신 resnet.ipynb 파일의 train과 test부분을 그대로 사용하였다.

6. Main

```
torch.cuda.empty_cache()
start_epoch = 0
start_time = time.time()
checkpoint = load_checkpoint(default_directory)

if not checkpoint:
    pass
else:
    start_epoch = checkpoint['epoch'] + 1
    model_ft.load_state_dict(checkpoint['state_dict'])
    optimizer_ft.load_state_dict(checkpoint['optimizer'])

best_acc = float(0)
for epoch in range(start_epoch, 20):
    torch.cuda.empty_cache()
    if epoch < 5:
        lr = learning_rate
    elif epoch < 10:
        lr = learning_rate * 0.5
    else:
        lr = learning_rate * 0.1
    for param_group in optimizer_ft.param_groups:
        param_group['lr'] = lr

    train(epoch)
    save_checkpoint(default_directory, {
        'epoch': epoch,
        'model': model_ft,
        'state_dict': model_ft.state_dict(),
        'optimizer': optimizer_ft.state_dict(),
    })

    now = test()
    if now > best_acc:
        save_checkpoint(default_directory,{
        'epoch': epoch,
        'model': model_ft,
        'state_dict': model_ft.state_dict(),
        'optimizer': optimizer_ft.state_dict(),
    },filename='best_acc_with'+str(now)+'.tar.gz')
        best_acc=now
```

```
now = time.gmtime(time.time() - start_time)
print('{} hours {} mins {} secs for training'.format(now.tm_hour, now.tm_min, now.tm_sec))
```

- 먼저 많은 코드 실행으로 'out of memroy'라는 GPU memory에 데이터를 너무 많이 올렸고 해제해주지 않아서 쌓이는 문제가 발생했다. 그래서 cache를 비워주는 코드를 매 epoch 실행마다 하였다.
- 두 번째 부분으로는 check point에서 loading하는 부분이며 이 부분도 역시 resnet.ipynb파일을 그대로 사용하였다. 이 부분은 중간에 계속 epoch와 model을 저장하면서 혹시 종료되더라도 다시 실행시 해당 epoch에서 그대로 다시 시작할 수 있도록 하고, 중요했던 점은 saving하면서 중간에 overfitting으로 빠지면 model configuration을 살짝 변경시켜서 학습을 더 잘 진행할 수 있도록 할 수 있었다.
- 마지막 부분은 실제로 epoch를 돌렸던 부분이다. 이 부분의 코드 자체는 마지막에 학습시켰던 resnet18 기준으로 작성되었다.
- scheduler를 쓰지 않고 단순히 if문으로 learning rate를 epoch이 진행함에 따라 낮추어주었다.

Model Hyperparameter

1. DenseNet 169 (result not saved)

- Batch size : 128
- DataAugmentation : 윗 코드의 주석친 부분 모두 다 train에 대해서 적용. test는 Normalization만 적용하였었음.
- Epoch : 200
- Learning rate :
  - ~ 80 : 0.001
  - ~ 130 : 0.0005
  - ~ 200 : 0.0001
- Freezing : fc layer도 cifar10 data에 맞게 10개의 class 분류로 하였음.

** 제일 첫 번째로 실행했던 코드여서 결과를 저장하지 못했었음.

2. Resnet152 & Resnet 50

- Batch size : 128
- DataAugmentation : 위와 동일
- Learning rate : 위와 동일
- Epoch : 200
- Freezing :
  - Fc layer : 주석화 시킨 부분처럼 2줄의 layer를 convolutional layer output size를 fc layer의 input으로 한 번 넣고 다른 Linear layer를 제작한 후 마지막에 10class로 분류를 하였음.
  - Layer 3~4 : Resnet 152와 Resnet 50은 Residual Block을 사용하여 큰 layer가 layer 1~4로 분류됨. 이 중 Layer1 ~2는 Freeze, 나머지 fc layer과 layer 3, 4는 cifar10에 대하여 학습시켰음.

** github file 중 Restnet152_layer3_true.tar.gz file이 가장 test accuracy가 높았던 시점의 model

** github file 중 resnet50.tar.gz file이 test accuracy가 가장 높았던 시점의 model

3. Resnet18

- Batchsize : 50
- DataAugmentation : Reisze, Normalization만
- Learning rate :
  - ~ 5 : 0.01
  - ~ 10 : 0.005
  - ~ 20 : 0.001
- Epoch : 20
- No Freezing : 모든 pretrained된 파라미터를 학습에 사용.

** github file 중 'best_acc~ ' file이 resnet 18을 활용한 file임.

Result

1. DenseNet 169

- test accuracy : 80%근방
- train accuracy : 93%
- epoch 50 → overfitting

2. Resnet 50 & Resnet 152

- test accuracy : 86~88%(resnet50), 82%(Resnet152)
- train accuracy : 95%
- epoch 100 → overfitting

3. Resnet18 (*)

- best test accuracy : 95.9(epoch : 10)
- train accuracy : 99~100%
- best test Loss : 0.1639 (이 때의 test accuracy 95.79, epoch : 6)
- Epoch 10에서 overfitting이 일어난 것 같아서 강제종료.

** Resnet18은 assignment_Cifar10.ipynb file의 main train부분을 참조.

** 나머지는 해당 model의 정확도와 해당 모델을 save하였음.

Conclusion

앞선 DenseNet 169와 Resnet50,Resnet152의 차이는 freezing부분이고, Resnet50,152와 가장 학습이 잘 되었던 Resnet18과의 차이는 DataAugmentation부분과 model size, batch size, lr 차이이다.

먼저 첫 번째 DenseNet같은 경우에는 cifar10의 현재 approach를 볼 수 있는, 과제 설명 부분에 링크된 사이트에서 정확도가 가장 높아보여서 사용을 하였었다. 하지만 여러가지 논문이나 github를 돌아다니면서 다른 연구자들이 했던 모델 구조를 확인하고 구현해 보았지만 정확도가 그리 높게 나오지는 않았다. 그 이유를 추측하는데, resnet 50과 resnet 152에서의 차이에서처럼 모델의 복잡도가 큰 요소로 작용한 것이 아닐까 한다. cifar10같은 경우에는 기존에 densenet과 resnet이 타게팅했던 ImageNet dataset처럼 큰 size의 dataset이 아니므로 학습한 데이터셋에 대해서 너무 잘 학습을 하기 때문에 오히려 모든 파라미터를 학습하기어렵고 일반화 성능이 떨어지므로 test dataset에 대하여는 낮은 정확도를 보여주었다.

그래서 마지막으로는 가장 model size가 작고, transfer learning에 대하여 좋은 pretrained된 파라미터를 제공하는 resnet18로 정하고 이에 따라 batch size를 128에서 50정도까지 줄이고, learning rate를 0.01로 늘리면서 좀 더 학습이 빠르게 될 수 있도록 하였다. 그럼에도 불구하고 그렇게까지 높은 정확도를 보이지 않았다(기존의 resnet 18 accuracy는 80~90% 대였음). 여기서 나는 고민을 해본 결과 batch size 및 lr 그리고 여러가지 모델에 대한 hyper parameter tuning은 현재 거의 효과가 없는 것 같아서 Data Loader부분의 transformer부분을 건드려보기로 했다. 그 결과 오히려 Data Augmentation을 많이 추가하여 일반화 성능을 높이려고했던 부분을 주석처리하고 transform에 대하여 Resize후 Normalization만 진행하니 점점 test accuracy가 앞도적으로 높아졌다. 이는 많은 Data Augmentation방법을 적용하는 것도 그렇게까지 좋은 방법은 아니었음을 시사한다.

결론적으로는 나는 이번 심화학습의 기초 Term project를 통해 3가지를 배우게 된것 같다. 첫 번째는 pytorch이다. 나는 기존에 항상 tensorflow를 사용해왔었고 그조차도 tutorial수준이었다. 그래서 이번에 pytorch를 통해서 term project를 통해서 dataloader의 중요성에 관한부분, 여러가지 model의 구조를 쌓는 부분을 배우게 되었다. 두 번째로는 바로 Transfer Learning과 그 불러온 모델의 미세조정하는 부분(freezing)이다. 엄청나게 많이 공을 들여서 layer을 쌓는 부분과 가중치 초기화에 대한 부분을 미리 좋게 튜닝하고 파라미터를 학습한 모델을 가져와서 나에게 맞는 문제를 해결하는 점에서 코드의 간결화 및 편리성이 내가 직접 모델을 일일히 만드는 것 보다 훨씬 더 효율적이라 생각한다. 물론 직접 한번 해 보는 것이 실력향상에는 도움되지만 미리 만들어놓은 모델에서 모델 구조를 개선해 나가는 것도 좋은 방법이라고 생각된다. 마지막으로 hyper parameter모다 data 전처리 부분인 transformer부분을 어떻게 쓰냐도 성능 향상에 아주 큰 요소로 작용한다는 것이다. 나는 일반화 성능을 높이려고 transform부분에서 엄청나게 많은 기법들을 한번에 넣고 처음에 진행을 했지만 별로 성능이 높지 않았고(학습조차 epoch 50~100까지 90퍼대를 못넘길정도로 잘 되지 않았다.) 이를 수정하니 오히려 간단한 data인 cifar10에서는 Data Augmentation을 하지 않는 것이 학습속도도 빠르고 일반화 성능도 높게 나왔다.