CV Project3 report

2018147558 김정주

1. Implementation

A. MyModel

```
class MyModel(nn.Module) :
    def __init__(self, res_block):
        super().__init__()
        #first convolve layer
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, 7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64), nn.ReLU(),
            nn.MaxPool2d(3, stride=2, padding=1)
        )
        #rest of residual block
        #depth 64 -> 128 -> 128
        self.conv2 = nn.Sequential(
            res_block(64, 64, 1),
            res_block(128, 64, 1)
        )
        #depth 128 -> 256 -> 256 -> 256
        self.conv3 = nn.Sequential(
            res_block(128, 128, 2),
            res_block(256, 128, 1),
            res_block(256, 128, 1)
        )
        #depth 256 -> 512 -> 512 -> 512 -> 512 -> 512
        self.conv4 = nn.Sequential(
            res_block(256, 256, 2),
            res_block(512, 256, 1),
            res_block(512, 256, 1),
            res_block(512, 256, 1),
            res_block(512, 256, 1)
        )
        #depth 512 -> 1024 -> 1024 -> 1024
        self.conv5 = nn.Sequential(
            res_block(512, 512, 2),
            res_block(1024, 512, 1),
            res_block(1024, 512, 1)
        )
        #avg pooling 1*1*1024
        self.pool = nn.AdaptiveAvgPool2d((1,1))
        #fully connected layer
        self.fc = nn.Linear(1024, 80)

    #forward function
    def forward(self,x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

i. Conv1 : 7*7, stride 2, padding 3 filter를 사용하여 전체 깊이를 64로 확장하고, 활성화 함수로를 ReLU함수를 채용하였고, 여기서는 max pool의 filter 크기를 3*3으로 설정하여 전체 크기를 줄였다.

ii. Conv2 : 여기부터는 res_block에서 구현된 conv 깊이를 점점 확장해가는 방식으

로 구현하였다. 여기서는 res_block의 연산이 2번 수행된다.

iii. Conv3 : 마찬가지로 res_block의 사용이며, 3번의 연산이 수행된다.

iv. Conv4 : res_block 5번의 연산이 수행된다.

v. Conv5 : res_block 3번의 연산이 수행된다.

vi. Pool : 마지막 계층에서 x*x*1024를 1*1*1024로 평균 pool를 해주는 계층으로 다음으로 나올 fc 계층에 연결된다.

vii. Fc : fully connected layer로서 80개의 class를 구별하는 데 사용된다.

viii. Forward : 앞에서 언급했던 과정을 차례로 수행해 준다.

B. Res_block

```
class res_block(nn.Module):
    def __init__(self, in_, out_, stride=1):
        super().__init__()

        #bottle neck layer
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_, out_, 1, stride=1, bias=False),
            nn.BatchNorm2d(out_), nn.ReLU()
        )
        #convolution layer(3*3)
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_, out_, 3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(out_), nn.ReLU()
        )
        #bottle neck and extension depth
        self.conv3 = nn.Sequential(
            nn.Conv2d(out_, out_ * 2, 1, stride=1, bias=False),
            nn.BatchNorm2d(out_ * 2)
        )
        #identity mapping
        self.shortway = nn.Sequential(
            nn.Conv2d(in_, out_ * 2, 1, stride=stride, bias=False),
            nn.BatchNorm2d(out_ * 2)
        ) if (stride != 1 or in_ != out_ * 2) else nn.Sequential()
        self.relu = nn.ReLU()

    def forward(self, x):
        output = x
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x += self.shortway(output)
        x = self.relu(x)
        return x
```

i. Conv1 : 주어진 input 크기와 output 크기에 따라 1*1의 bottle neck 연산을 수행해준다.

ii. Conv2 : 3*3 filter로 conv연산을 수행해준다.

iii. Conv3 : 1*1의 bottle neck 연산이나 깊이를 2배로 확장해준다. 깊이를 2배로 확장해줌으로써 conv 연산을 쌓을수록 깊이를 늘려갈 수 있다.

iv. Shortway : identity 계층을 말하며, 최적화 문제를 해결하기 위해 이와 같은 계층을 사용하여 직접 연결해준다.

v. Forward : 구현 방식이 위에서의 MyModel과 비슷하나 shortway 계층의 수행 값을 더해주는 부분으로 인해 residual으로서 작동하게 된다.

C. MyDataset(for train data)

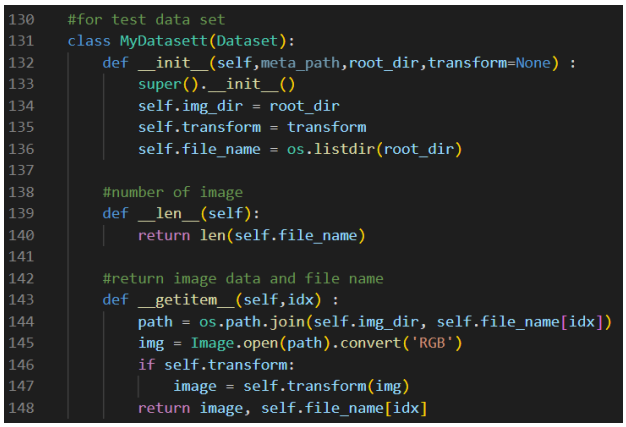
```
class MyDataset(Dataset):

    def __init__(self,meta_path,root_dir,transform=None) :
        super().__init__()
        self.img_labels = []
        with open(meta_path, 'r') as f:
            json_data = json.load(f)
        for l in json_data['annotations']:
            elemt = []
            elemt.append(l['file_name'])
            elemt.append(l['category'])
            self.img_labels.append(elemt)
        self.img_dir = root_dir
        self.transform = transform

    #number of image
    def __len__(self):
        return len(self.img_labels)

    #return image data and label
    def __getitem__(self,idx) :
        label = int(self.img_labels[idx][1])
        path = os.path.join(self.img_dir, self.img_labels[idx][0])
        img = Image.open(path).convert('RGB')
        if self.transform: image = self.transform(img)
        return image, label
```

i. __init__ : 생성자로서 디렉토리를 설정해주고, transform을 객체의 데이터로 저장하고 meta path로부터 json파일을 읽어 file_name과 category를 분류하여 데이터로 유지한다.

ii. __len__ : 전체 이미지의 개수를 return한다.

iii. __getitem__ : idx에 맞는 이미지 데이터를 불러들여 transform을 적용하여 label과 함께 return한다.

D. MyDatasett(for test data)



```
#for test data set
class MyDatasett(Dataset):
    def __init__(self,meta_path,root_dir,transform=None) :
        super().__init__()
        self.img_dir = root_dir
        self.transform = transform
        self.file_name = os.listdir(root_dir)

    #number of image
    def __len__(self):
        return len(self.file_name)

    #return image data and file name
    def __getitem__(self,idx) :
        path = os.path.join(self.img_dir, self.file_name[idx])
        img = Image.open(path).convert('RGB')
        if self.transform:
            image = self.transform(img)
        return image, self.file_name[idx]
```

i. __init__ : 생성자로서 디렉토리를 설정해주고, transform을 객체의 데이터로 저장하고 root_dir에 있는 모든 이미지 파일의 이름을 리스트화하여 저장한다.

ii. __len__ : 전체 이미지의 개수를 return한다.

iii. __getitem__ : idx에 맞는 이미지 데이터를 불러들여 transform을 적용하여 해당 이미지의 이름과 함께 return 한다.

E. Train

```python
def train(epochs) :
    model_name = MyModel
    checkpoint_path = './drive/MyDrive/model.pth'
    mode = 'train'
    data_dir = "./drive/MyDrive/train_data"
    meta_path = "./drive/MyDrive/answer.json"
    model = get_model(model_name,checkpoint_path)

    #Image Augmentation(crop, flip, rotate)
    transt = transforms.Compose([transforms.RandomResizedCrop((224, 224)),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5)),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.RandomVerticalFlip(),
                                 transforms.RandomRotation(90, expand=False)])

    trans = transforms.Compose([transforms.Resize((224,224)),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])

    data_transforms = {
        'train' : transt,
        'test' : trans
    }

    # Create train dataset and train dataloader
    train_datasets = MyDataset(meta_path, data_dir, data_transforms[mode])
    train_dataloader = torch.utils.data.DataLoader(train_datasets, batch_size=32, shuffle=True)
    # Detect if we have a GPU available
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = model.to(device)

    # Set model as evaluation mode
    criterion = nn.CrossEntropyLoss().cuda()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    for epoch in range(epochs):
        loss_sum = 0
        correct = 0

        for data in tqdm(train_dataloader):
            #x = transformed image, y = correct class
            x, y = data
            optimizer.zero_grad()
            # forward pass, calculate loss, update weight
            output = model(x.to(device))
            loss = criterion(output, y.to(device))
            loss.backward()
            optimizer.step()
            # add loss
            _, preds = torch.max(output, 1)
            for j in range(32):
                if preds[j] == y[j]:
                    correct += 1
            loss_sum += loss.item()

        print("epoch[%d] loss: %.3f" %(epoch+1, loss_sum / 1250),
            "accuracy: {:.3f}".format(correct/40000 *100))
        torch.save(model.state_dict(), './drive/MyDrive/model.pth')
```

i. Epochs의 수만큼 데이터의 학습을 진행한다.

ii. Transt : train 데이터의 경우, 데이터 수를 늘리기 위해서 image augmentation을 사용했다. Random crop, random flip, random rotate를 사용하였다.

iii. Dataloader : train 데이터는 학습용도이기 때문에 shuffle = true로 설정했고, 소규모 데이터로 학습을 한 결과 최적의 batch size는 32이였다.

iv. Criterion : 가장 일반적으로 사용되는 cross entropy loss를 사용하였다.

v. Optimizer : 일반적인 SGD를 사용하였고, 소규모 데이터로 학습 시 learning rate가 0.01일 때의 수행결과가 적합하다고 판단하여 0.01로 설정했다.

vi. Learning loop : 모든 이미지에 대해서 MyModel에서의 forward를 수행하여 결과값과 차이를 통해, back propagation(loss.backward()) 계산, weight 업데이트(optimizer.step())를 수행하고, 전체 loss값의 합과 정확도를 계산하여 매 epoch 마다 출력하고 model의 데이터를 저장하는 방식으로 구현했다.

F. Test

```python
def test():

    model_name = MyModel
    checkpoint_path = './model.pth'
    mode = 'test'
    data_dir = "./train_data"
    meta_path = "./answer.json"
    model = get_model(model_name,checkpoint_path)

    trans = transforms.Compose([transforms.Resize((224,224)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])

    transt = transforms.Compose([transforms.RandomResizedCrop((224, 224)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5)),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomVerticalFlip(),
                                transforms.RandomRotation(90, expand=False)])
    data_transforms = {
        'train' : transt,
        'test' : trans
    }

    # Create test dataset, test dataloaders
    test_datasets = MyDatasett(meta_path, data_dir, data_transforms[mode])
    test_dataloader = torch.utils.data.DataLoader(test_datasets, batch_size=32, shuffle=False, num_workers=4)

    # Detect if we have a GPU available
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    # Send the model to GPU or CPU
    model = model.to(device)

    # Set model as evaluation mode
    for param in model.parameters():
        param.requires_grad = False
    model.eval()

    # Inference
    result = []
    for images, filename in tqdm(test_dataloader):
        num_image = images.shape[0]
        images = images.to(device)
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        for i in range(num_image):
            result.append({
                'filename': filename[i],
                'class': preds[i].item()
            })

    result = sorted(result,key=lambda x : int(x['filename'].split('.')[0]))

    with open('./result.csv', 'w') as f:
        writer = csv.writer(f)
        writer.writerow(['filename','class'])
        for res in result:
            writer.writerow([res['filename'], res['class']])
```

i. Trans : test 데이터는 image augmentation이 필요 없기에 resize와 정규화, tensor화만 사용하였다.

ii. Dataloader : 마찬가지로 shuffle의 의미가 없기에 false, train와 같은 batch size로 구현했다.

2. Improve performance

A. Data augmentation : 주어진 데이터가 40000개 이지만 이 데이터에 대해서만 학습하면 overfitting이 될 것이라고 생각하여 data augmentation을 이미지에 적용하여, train하는 방식을 취했다.

B. Res_block : 나의 모델은 기본적으로 resnet을 참고하여 만들었기에 res_block의 존재는 최적화를 위해서는 가장 중요한 것이다. 이 블록이 존재하지 않았을 때에는 평균 loss가 10배이상 이었고, train 속도도 많이 느렸다. Res_block을 사용함으로 이러한 문제를 해결하였다.

C. Bottle neck : 내 계층은 50계층이 넘어간다. 그렇기 때문에 계산의 효율성을 위해서 bottle neck 계층을 사용하였고, 이 때문에 실행시간을 단축할 수 있었다.

D. Depth extension : res_block를 사용할 때마다 depth는 2배씩 늘어나는 방식을 통해서 계층을 겹칠 때문에 depth를 늘려 classifier가 잘 분류할 수 있도록 하였다.

3. Run code

A. 먼저 5개의 batch에서 내가 구현한 모델이 overfitting되는지 확인하여, 잘 안되면 계층을 점점 늘려가는 방향으로 네트워크의 아키텍처를 구현했다. 그리고 이 과정에서 학습률도 같이 결정했다.

B. 다음으로는 40000개의 이미지를 resize만 적용하여 train을 30번 수행하였다. 이 과정에서 train set에 대한 정확도가 100%를 달성했지만, train set에 overfitting이 되었을 것이라고 판단했다.

C. Image augmentation을 통해 위에서 train한 이미지에 대해서 적용해보니 정확도가 50%로 낮아졌고, overfitting되었음을 어느정도 확신하였다. 그리고 augmentation을 적용한 이미지와 함께 다시 train을 50회가량 수행하니 정확도 85%의 그럴싸한 결과를 확인하였다.