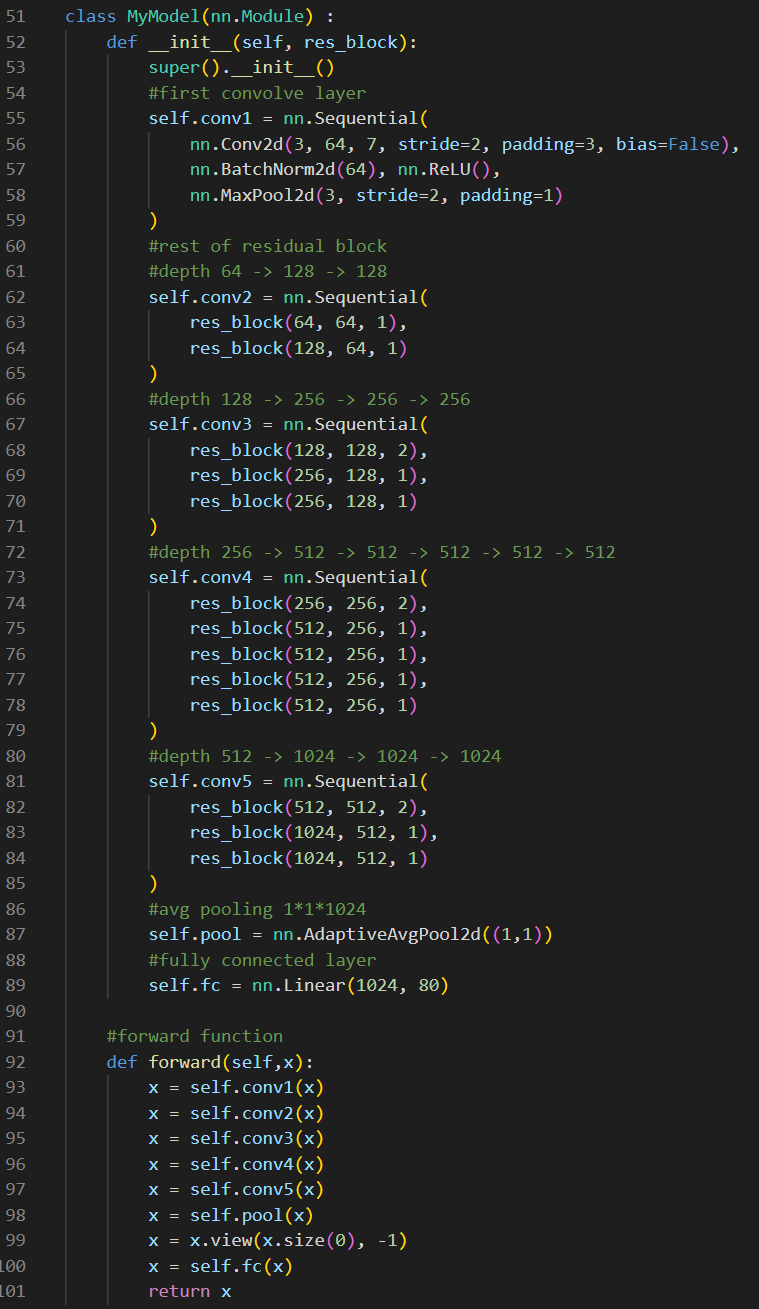
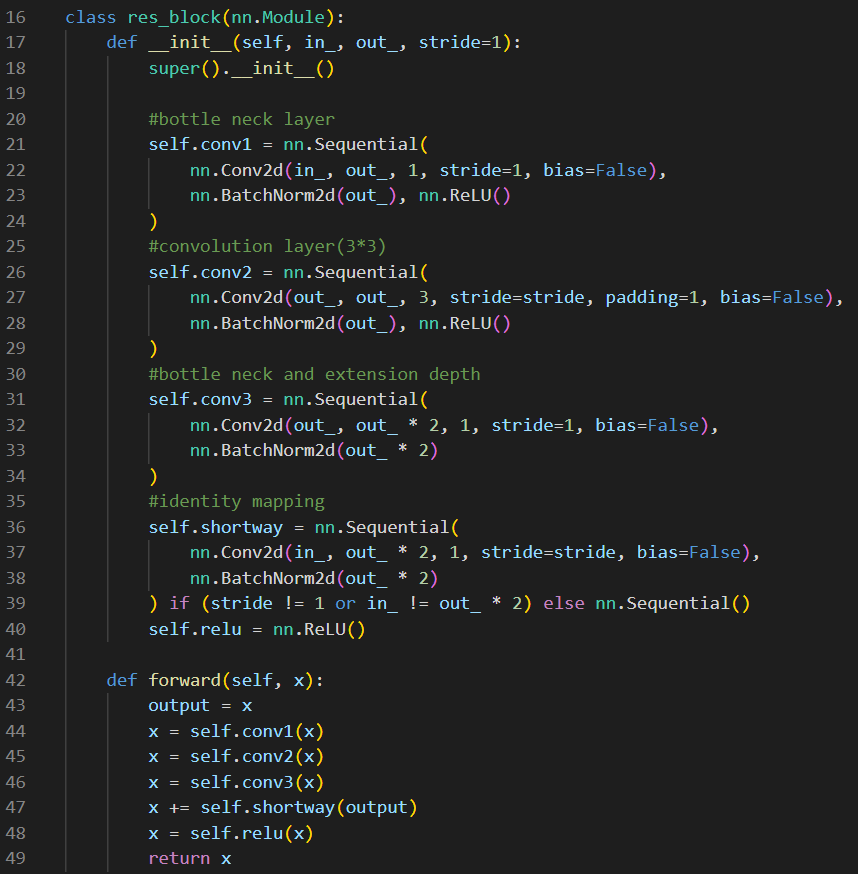
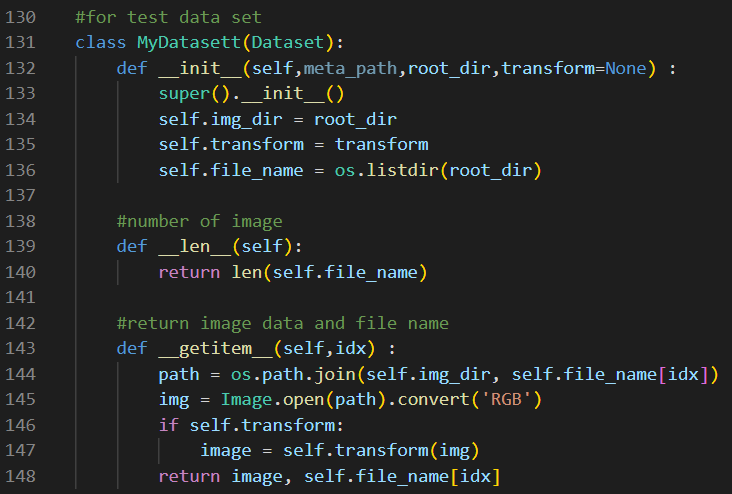
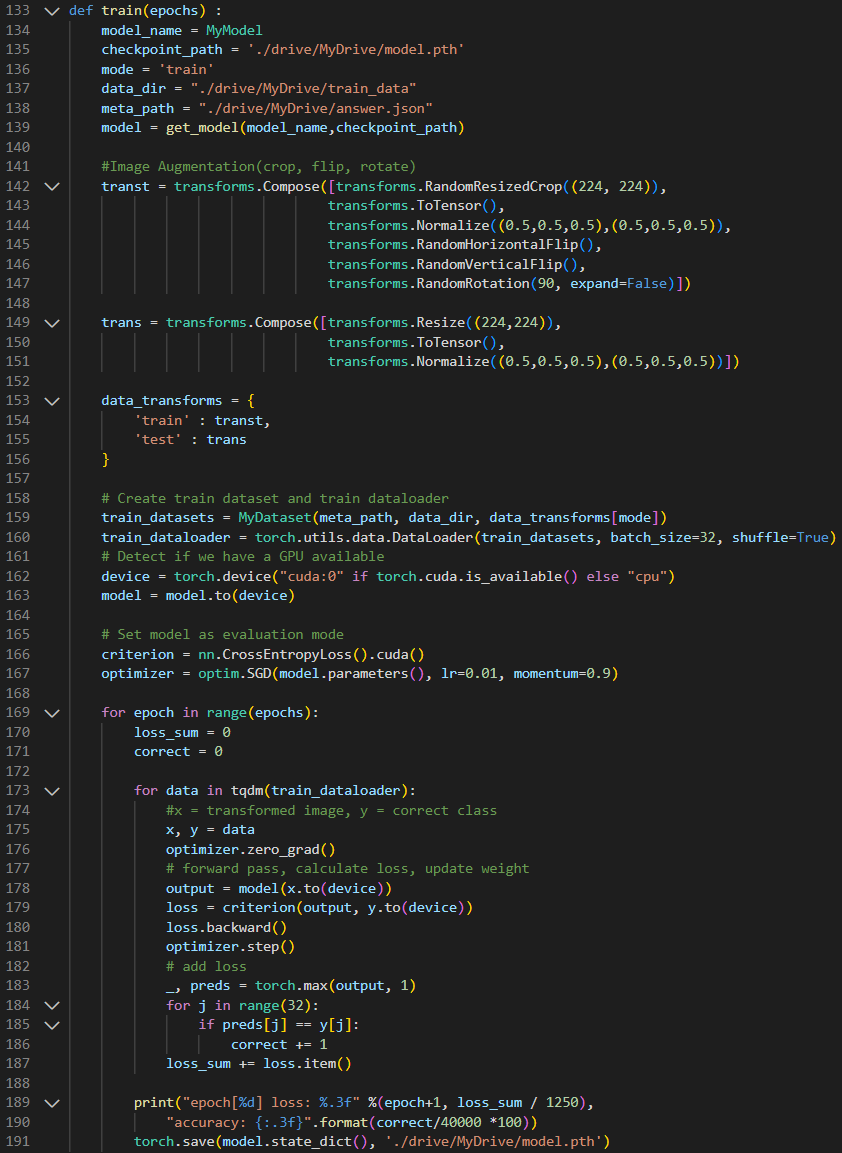
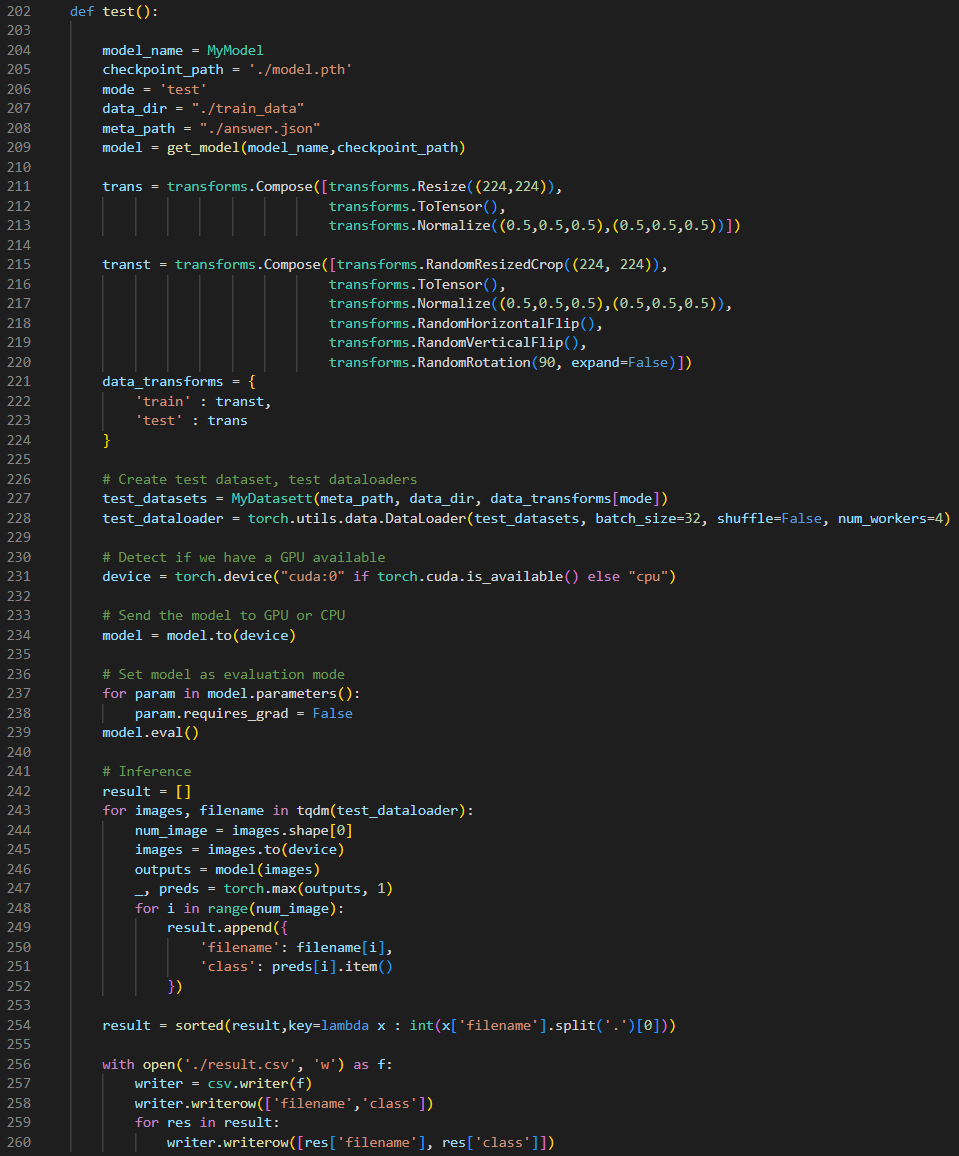
CV Project3 report

2018147558 김정주

1. Implementation
   1. MyModel
      1. 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.
      2. Conv2 : 여기부터는 res\_block에서 구현된 conv 깊이를 점점 확장해가는 방식으로 구현하였다. 여기서는 res\_block의 연산이 2번 수행된다.
      3. Conv3 : 마찬가지로 res\_block의 사용이며, 3번의 연산이 수행된다.
      4. Conv4 : res\_block 5번의 연산이 수행된다.
      5. Conv5 : res\_block 3번의 연산이 수행된다.
      6. Pool : 마지막 계층에서 x\*x\*1024를 1\*1\*1024로 평균 pool를 해주는 계층으로 다음으로 나올 fc 계층에 연결된다.
      7. Fc : fully connected layer로서 80개의 class를 구별하는 데 사용된다.
      8. Forward : 앞에서 언급했던 과정을 차례로 수행해 준다.
   2. Res\_block
      1. Conv1 : 주어진 input 크기와 output 크기에 따라 1\*1의 bottle neck 연산을 수행해준다.
      2. Conv2 : 3\*3 filter로 conv연산을 수행해준다.
      3. Conv3 : 1\*1의 bottle neck 연산이나 깊이를 2배로 확장해준다. 깊이를 2배로 확장해줌으로써 conv 연산을 쌓을수록 깊이를 늘려갈 수 있다.
      4. Shortway : identity 계층을 말하며, 최적화 문제를 해결하기 위해 이와 같은 계층을 사용하여 직접 연결해준다.
      5. Forward : 구현 방식이 위에서의 MyModel과 비슷하나 shortway 계층의 수행 값을 더해주는 부분으로 인해 residual으로서 작동하게 된다.
   3. MyDataset(for train data)
      1. \_\_init\_\_ : 생성자로서 디렉토리를 설정해주고, transform을 객체의 데이터로 저장하고 meta path로부터 json파일을 읽어 file\_name과 category를 분류하여 데이터로 유지한다.
      2. \_\_len\_\_ : 전체 이미지의 개수를 return한다.
      3. \_\_getitem\_\_ : idx에 맞는 이미지 데이터를 불러들여 transform을 적용하여 label과 함께 return한다.
   4. MyDatasett(for test data)
      1. \_\_init\_\_ : 생성자로서 디렉토리를 설정해주고, transform을 객체의 데이터로 저장하고 root\_dir에 있는 모든 이미지 파일의 이름을 리스트화하여 저장한다.
      2. \_\_len\_\_ : 전체 이미지의 개수를 return한다.
      3. \_\_getitem\_\_ : idx에 맞는 이미지 데이터를 불러들여 transform을 적용하여 해당 이미지의 이름과 함께 return 한다.
   5. Train
      1. Epochs의 수만큼 데이터의 학습을 진행한다.
      2. Transt : train 데이터의 경우, 데이터 수를 늘리기 위해서 image augmentation을 사용했다. Random crop, random flip, random rotate를 사용하였다.
      3. Dataloader : train 데이터는 학습용도이기 때문에 shuffle = true로 설정했고, 소규모 데이터로 학습을 한 결과 최적의 batch size는 32이였다.
      4. Criterion : 가장 일반적으로 사용되는 cross entropy loss를 사용하였다.
      5. Optimizer : 일반적인 SGD를 사용하였고, 소규모 데이터로 학습 시 learning rate가 0.01일 때의 수행결과가 적합하다고 판단하여 0.01로 설정했다.
      6. 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 epoch마다 출력하고 model의 데이터를 저장하는 방식으로 구현했다.
   6. Test
      1. Trans : test 데이터는 image augmentation이 필요 없기에 resize와 정규화, tensor화만 사용하였다.
      2. Dataloader : 마찬가지로 shuffle의 의미가 없기에 false, train와 같은 batch size로 구현했다.
2. Improve performance
   1. Data augmentation : 주어진 데이터가 40000개 이지만 이 데이터에 대해서만 학습하면 overfitting이 될 것이라고 생각하여 data augmentation을 이미지에 적용하여, train하는 방식을 취했다.
   2. 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.
   3. Bottle neck : 내 계층은 50계층이 넘어간다. 그렇기 때문에 계산의 효율성을 위해서 bottle neck 계층을 사용하였고, 이 때문에 실행시간을 단축할 수 있었다.
   4. 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
   1. 먼저 5개의 batch에서 내가 구현한 모델이 overfitting되는지 확인하여, 잘 안되면 계층을 점점 늘려가는 방향으로 네트워크의 아키텍처를 구현했다. 그리고 이 과정에서 학습률도 같이 결정했다.
   2. 다음으로는 40000개의 이미지를 resize만 적용하여 train을 30번 수행하였다. 이 과정에서 train set에 대한 정확도가 100%를 달성했지만, train set에 overfitting이 되었을 것이라고 판단했다.
   3. 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.