

시각 장애인을 위한 스마트 옷장

옷의 종류와 색상 판별을 중심으로



제출일	2021.12.19	전공	컴퓨터공학과
과목	기계학습	학번	19102203 위성률 18100429 박희재
담당교수	한지형	이름	19101996 변현수

■ 서론

모든 사람이 오늘 입을 옷을 눈으로 보고 고를 수 있는 것은 아닙니다. 시각 장애인들에게는 옷을 고를 수 있는 기회가 주어지지 않습니다. <그림 5>는 한국 일보의 “시각장애인이라고 장례식서 빨간 옷 입을 순 없죠.”라는 기사의 일부를 발췌한 글입니다. 해당 기사는 옷의 왼쪽 아랫쪽에 오선지 모양의 라벨을 달아 시각장애인들이 점자를 읽듯이 옷의 색을 파악할 수 있도록 하는 프로젝트를 소개했습니다.

본격 작업에 돌입한 박 대표는 지난해 서울시각장애인 복지관의 도움을 받아 프로젝트에 관심 있는 시각장애인을 소개받았다. 이들의 집을 방문해 평소에는 어떤 방식으로 옷을 구별하는지 세세하게 관찰했다. 딱 한가지 종류의 셔츠나 바지만 사는 사람, 옷 안쪽의 케어 라벨을 특정한 모양으로 오려 구분하는 사람, 옷장에 두는 위치를 고정해 구분하는 사람 등 제각각이었다.

다만 한 가지 공통점이 있었다. "양말은 한 가지 색만 신는다고 하더라고요. 여러 색의 양말을 가지고 있으면 짹짹으로 신을 가능성이 높기 때문이래요. 그 때, 색을 구분하는 방법 만이라도 찾으면 참 좋겠다고 생각했죠."

<그림 5>

저희는 해당 기사에서 영감을 받아 옷을 들고 옷장 앞에 서면 카메라로 대상을 인식해 자켓, 코트, 치마 등 옷의 종류와 옷의 색상을 알려주는 스마트 옷장을 개발하고자 합니다. 이번 프로젝트는 단기 프로젝트인 만큼 실제 상용화를 시킬 수 있을 정도의 기술로 발전시키는 것은 어렵겠지만 인공지능이 탑재된 가전제품이 점점 많아지는 추세인 만큼 충분히 상용화가 가능한 아이디어이고 의미 있는 프로젝트라고 생각합니다.

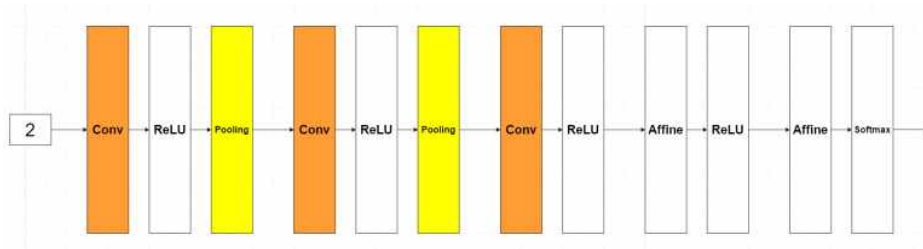
■ 본론

1. 적용 알고리즘에 대한 이론적 이해

1. CNN 알고리즘

뇌의 시각 피질에 대한 연구에서 시작되어 Convolutional neural networks, CNN이라고 불리는 이 알고리즘은 1980년대부터 이미지 인식에 사용되었습니다. 시각적 인식뿐 아니라 음성 인식 및 자연어 처리와 같은 분야에서도 많이 사용되고 있습니다. CNN을 활용하여 신경망과 같이 계층을 조합하는 모델을 또한 만들 수 있는데, 이 때 사용하는 계층은 합성곱 계층(Convolution layer)과 풀링 계층(Pooling layer)이 있습니다. 계층 모델에서 인접하는 계층의 모든 뉴런과 결합이 되어있는 완전연결(fully connected) 계층을 Affine 계층이라고 하는데, CNN의 convolution

layer의 뉴런은 입력 이미지의 모든 픽셀에 연결되는 것이 아니라 receptive fields의 pixel에만 연결이 됩니다.



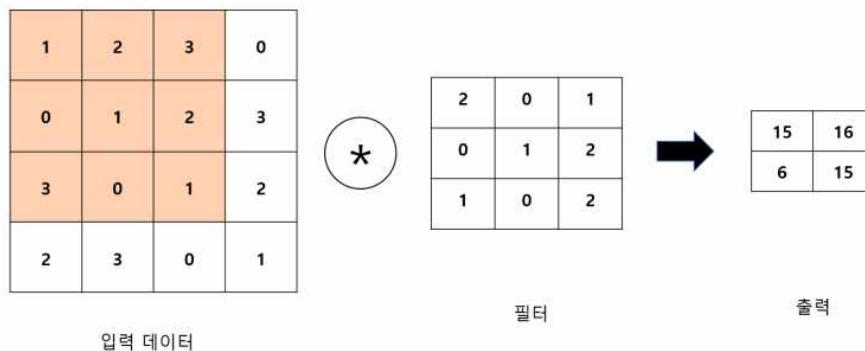
<그림 6> CNN Layer

<그림 6>은 CNN의 layer를 설명한 그림입니다. Conv-Relu-Pooling 흐름으로 연결이 되고 출력에 가까운 층은 Affine-ReLU 구성을 사용할 수도 있고 Affine-softmax 조합을 그대로 사용할 수도 있습니다.

CNN은 padding, stride를 사용하고 3차원 데이터같이 입체적인 데이터가 흐른다는 점에서 완전연결 신경망과 차이점을 갖습니다. 완전연결 계층에 데이터를 입력할 때에는 데이터의 차원을 1차원으로 낮춰줘야 하기 때문에 데이터의 형상이 무시된다는 단점이 있지만 CNN은 형상을 유지한 채 학습을 진행할 수 있습니다. 이미지를 3차원으로 입력받아 그대로 다음 계층으로 전달하기 때문에 이미지에 대한 이해도가 높습니다.

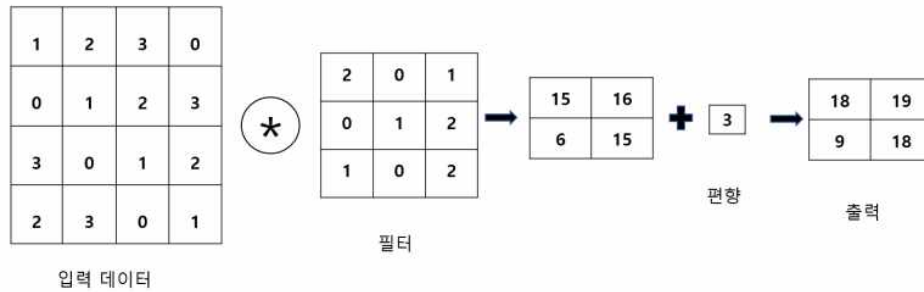
① 합성곱 연산

CNN의 합성곱 계층에서는 합성곱 연산을 수행합니다. 합성곱 연산은 입력 데이터에 커널이라고도 불리는 필터를 적용시키는데, 입력 데이터와 커널 모두 세로, 가로 방향의 차원을 갖습니다.



<그림 7> 합성곱 연산

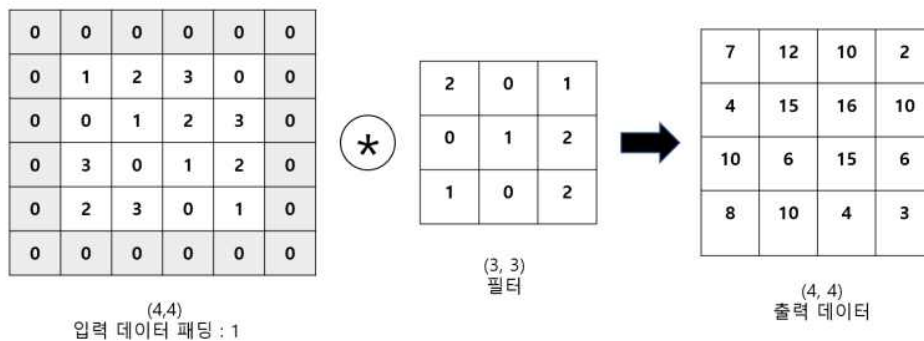
합성곱 연산은 입력 데이터와 대응하는 필터의 원소를 곱한 후 총 합을 구하는 과정으로 수행됩니다. 연산 결과를 출력의 해당 장소에 저장하고, 이 과정을 모든 장소에 대해 수행하면 합성곱 연산의 출력이 완성됩니다. CNN에도 bias가 존재하는데, 이 bias는 필터를 적용한 후의 데이터에 더해집니다. <그림7, 8>은 합성곱 연산에 대한 그림 설명입니다.



<그림 8> 합성곱 연산 (세부)

② 패딩

합성곱 연산을 수행하기 전에 입력 데이터 주변을 특정 값으로 채워주기도 합니다. 이 과정을 '패딩'이라고 하는데, 주로 출력 크기를 조정할 목적으로 사용됩니다. 합성곱 연산을 수행할 때마다 출력 크기가 작아지기 때문에 어느 시점에서는 출력 크기가 1이 되는 상황이 발생합니다. 이렇게 되면 합성곱 연산을 더 이상 수행할 수가 없기 때문에 패딩을 사용해 입력 데이터의 공간적 크기를 고정한 채로 다음 계층으로 데이터를 전달할 수 있도록 합니다.



<그림 9> 패딩

③ 스트라이드

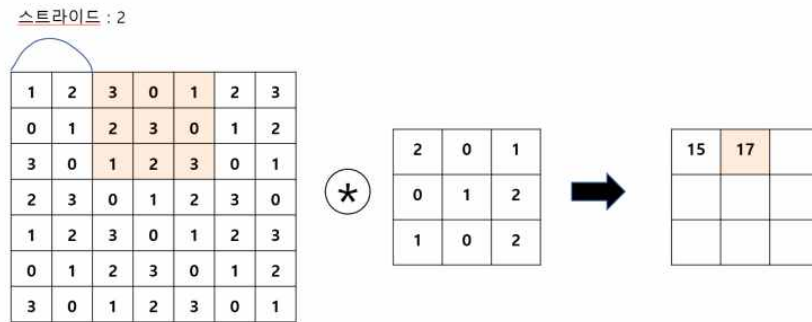
스트라이드(stride)는 필터를 적용시키는 위치의 간격입니다. 스트라이드를 2로 설정하면 적용하는 윈도우가 두 칸씩 이동합니다. 스트라이드가 커지면 패딩을 충분히 키우더라도 출력 크기가 작아지게 됩니다. 스트라이드는 그림 11과 같이 동작합니다. 입력 크기를 (H, W), 필터 크기를 (FH, FW), 출력 크기를 (OH, OW), 패딩을 P, 스트라이드를 S라고 하면 OH와 OW는 <그림 10>과 같이 나타낼 수 있습니다.

$$OH = \frac{H + 2P - FH}{S} + 1$$

$$OW = \frac{W + 2P - FW}{S} + 1$$

<그림 10> 스트라이드 수식

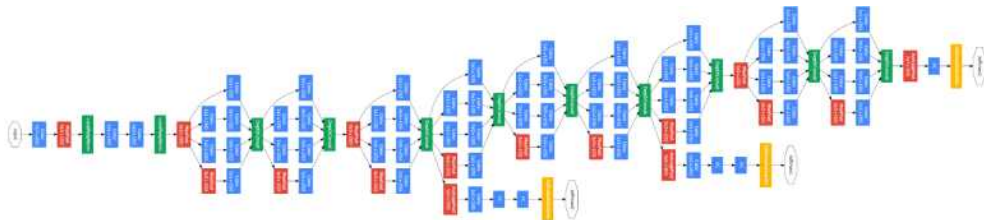
다.



<그림 11> 스트라이드

2. GoogleNet 알고리즘

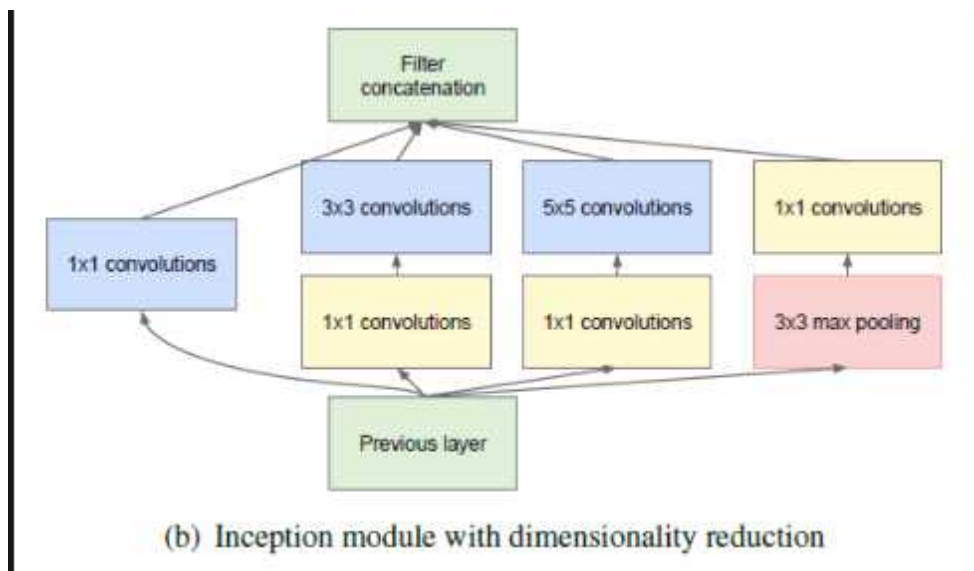
GoogleNet은 2014년 ImageNet 이미지 인식 대회(ILSVRC)에서 VGG19를 이기고 우승을 차지한 알고리즘입니다. 총 22개의 layer로 구성되어 있습니다. <그림 12> 의 파란색 블록 층수를 세어보면 22개인 것을 알 수 있습니다.



<그림 12> GoogleNet Layer 구조

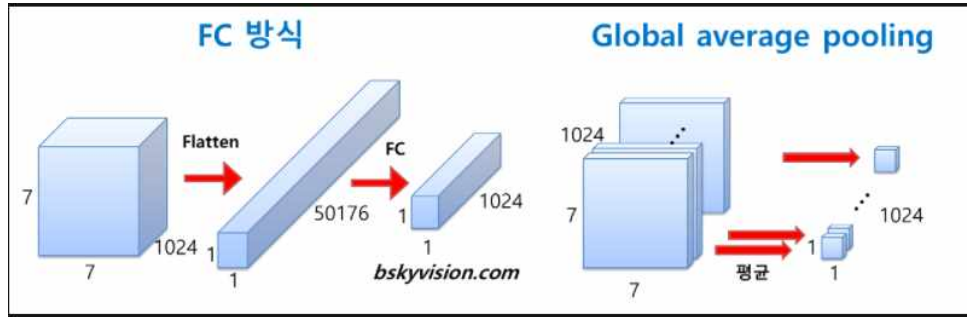
GoogleNet은 1x1 사이즈의 필터로 convolution을 해주는데, 이는 특성맵의 개수를 줄이려는 목적으로 사용됩니다. 특성맵의 개수가 줄어들면 그만큼 연산량이 줄어들기 때문에 네트워크를 더욱 깊게 만들 수 있습니다.

또한 GoogleNet에는 인셉션 구조가 존재합니다. GoogleNet에 사용된 모듈은 1x1 convolution이 포함된 <그림 13>과 같은 모델입니다.



<그림 13> GoogleNet 모듈

인셉션 모듈을 사용해 이전 층에서 생성된 특성 맵을 1x1 convolution, 3x3 convolution, 5x5 convolution하고 3x3 최대 풀링해준 결과로 얻은 특성맵들을 모두 함께 쌓아줍니다. 이러한 과정을 통해 더 다양한 종류의 특성을 얻을 수 있습니다.

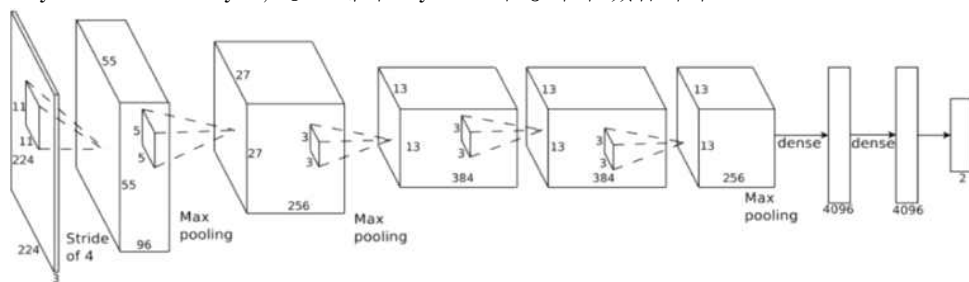


<그림 14> GoogleNet Global average pooling 방식

후반부에는 fully connected 방식 대신에 global average pooling이라는 방식을 사용합니다. Global average pooling은 전 층에서 산출된 특성맵들을 각각 평균낸 후 연결해 1차원 벡터를 만들어주는 방식입니다. 1차원 벡터를 만들어야 이미지 분류를 위한 최종 softmax layer를 연결해줄 수 있기 때문에 이러한 방식을 사용하며 가중치의 개수를 줄여준다는 장점을 갖습니다.

3. AlexNet

AlexNet은 2012년 ImageNet 이미지 인식 대회(ILSVRC)에서 우승을 차지한 CNN 구조입니다. <그림 15>와 같이 5개의 convolution layer와 3개의 fully-connected layer, 총 8개의 layer로 구성되어 있습니다.



<그림 15> AlexNet layer 구조

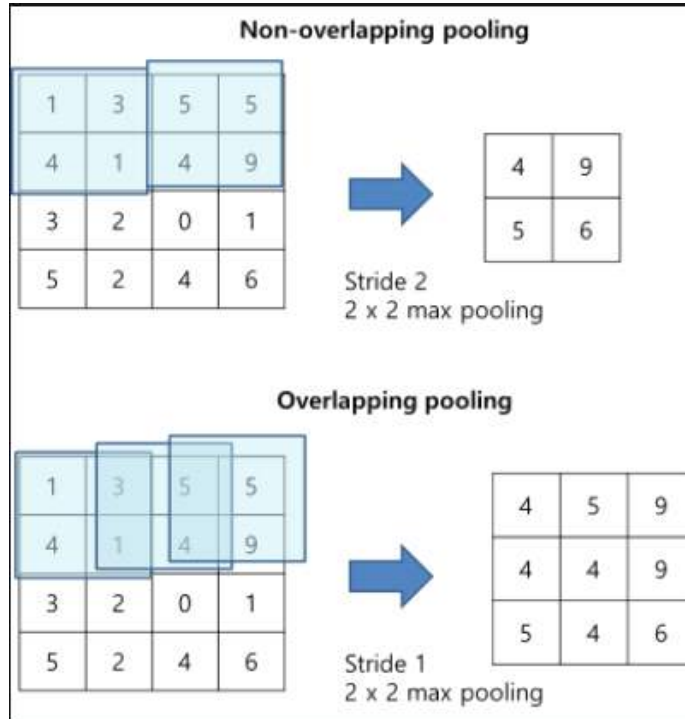
두 번째, 네 번째, 다섯 번째 layer는 전 단계의 같은 채널 특성맵들과만 연결이 되어 있는 반면 세 번째 convolution layer는 전 단계 두 채널의 특성맵들과 모두 연결되어 있습니다. 다음은 AlexNet에 사용된 여러 기법들에 대한 설명입니다.

① ReLU, dropout

AlexNet에는 ReLU 함수가 사용되었습니다. AlexNet 이전의 LeNet-5에서는 활성화 함수로 Tanh를 사용했는데, ReLU를 사용하게 되면서 동일한 정확도를 유지하면서도 학습 시간을 6배 가까이 단축시킬 수 있게 되었습니다. 또한 overfitting을 방지하기 위해 dropout 기법을 사용합니다.

② Overlapping 최대 풀링

Overlapping 풀링을 사용하면 풀링 커널이 중복되어 지나가게 됩니다. Overlapping 풀링은 ImageNet의 top-1과 top-5의 에러율을 줄이는데 효과가 있습니다.



<그림 16> Overlapping 최대 풀링

③ LRN

신경생물학에는 lateral inhibition이라는 현상이 존재하는데 이는 활성화된 뉴런이 주변 이웃 뉴런들을 억누르는 현상을 말합니다. 이 lateral inhibition 현상을 모델링한 것이 LRN입니다. LRN은 강하게 활성화된 뉴런의 주변 이웃들에 대해서 normalization을 실행합니다. 주변에 비해 비교적 강하게 활성화되어있는 뉴런이 있다면, 그 뉴런의 반응은 더욱 돋보이게 됩니다. 반면에 강하게 활성화된 뉴런의 주변 뉴런들도 모두 강하게 활성화되어 있다면, local response normalization 이후에 모두 값이 작아지게 됩니다.

④ Data augmentation

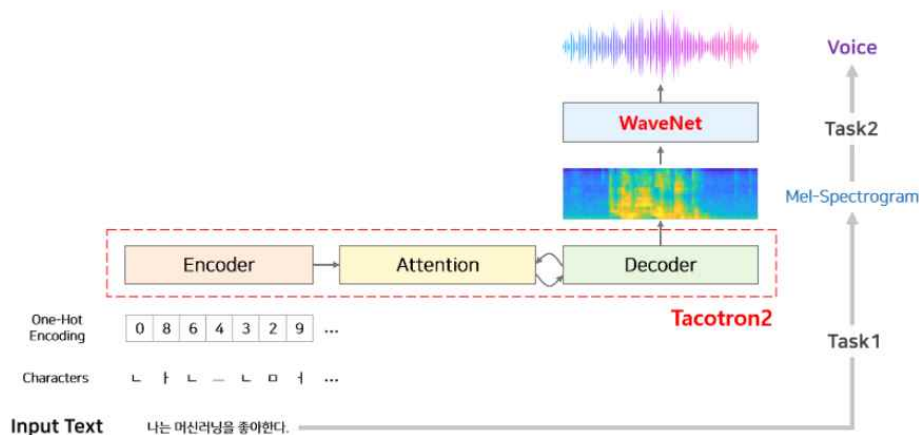
Overfitting을 막기 위한 방법으로 데이터의 양을 늘리는 기법으로 Data augmentation을 이용해 이미지를 좌우 반전, 회전, resizing 시키거나 밝기 조절 등을 이용해서 여러 장의 비슷한 이미지를 만들어 내는 기법입니다.

4. K-means Clustering

K-means clustering은 가장 간단하면서도 인기있는 unsupervised learning 기법 중 하나입니다. 클러스터의 개수에 해당하는 k 값을 hyperparameter로 받은 후 k개의 centroid를 random하게 initialize하고, 데이터들을 가장 가까운 centroid에 할당시킵니다. 클러스터링이 끝나면 클러스터 내에서 분류된 instance들의 평균을 계산해 중심점(centroid)를 재설정하고, 재설정된 centroid를 중심으로 다시 클러스터링을 하는 과정을 반복합니다. 이 때의 각 클러스터들은 이름이 아닌 단순한 번호로 구분합니다. 앞서서도 언급했듯이 매우 단순하면서도 강력한 알고리즘이지만 클러스터들 간에 크기나 밀도 차이가 많이 날 경우 제대로 clustering이 되지 않을 수도 있다는 단점이 존재합니다. 또한 비지도학습이기 때문에 결과값이 항상 optimal하다고 보장할 수 없으며 초기에 random하게 설정되는 centroid들의 위치가 전체 모델 성능에 매우 큰 영향을 줄 수 있다는 특징도 가지고 있습니다. 이러한 K-means clustering의 단점을 보완하기 위한 기법으로는 초기 centroid들을 가장 먼 거리에 위치하도록 선택하고, 학습을 여러번 수행하고, elbow point를 찾거나 silhouette diagram을 그려보는 방법 등이 존재합니다.

5. Tacotron

타코트론은 Attention 기반 Seq-to-Seq TTS 모델구조로 문장과 음성 쌍으로 이루어진 데이터만으로도 별도의 작업 없이 학습이 가능한 End-to-End 모델입니다. 타코트론 모델은 텍스트를 받아 음성을 합성합니다. 따라서 최종 input과 output은 텍스트와 음성입니다. 하지만 텍스트로부터 바로 음성을 생성하는 것은 어렵기 때문에 TTS를 두 단계로 나누어서 처리합니다. 첫 번째는 텍스트로부터 Mel-Spectrogram을 생성하는 단계이고 두 번째는 Mel-spectrogram으로부터 음성을 합성하는 단계입니다. 첫 번째 단계는 Sequence to Sequence 딥러닝 구조의 타코트론2 모델이 담당하고 있고 두 번째 단계는 Vocoder로 불리며 WaveNet 모델을 변형해서 사용합니다.



<그림 17> 타코트론 구조

2. 모델 구현

1. 데이터셋

mmlab에서 제공하는 deep fashion 데이터셋을 사용하여 모델을 학습시켰습니다. Anorak, Blazer 등 총 45개의 label로 이루어져 있으며 <그림 18>은 Pytorch의 ImageFolder가 자동으로 label에 할당해준 번호입니다. <그림 19>는 각 label에 존재하는 데이터의 개수를 의미합니다.

dataset.class_to_idx	
{'Anorak': 0,	386
'Blazer': 1,	146
'Blouse': 2,	160
'Bomber': 3,	7497
'Button-Down': 4,	24562
'Caftan': 5,	309
'Capris': 6,	330
'Cardigan': 7,	54
'Chinos': 8,	77
'Coat': 9,	13311
'Coverup': 10,	527
'Culottes': 11,	2120
'Cutoffs': 12,	17
'Flannel': 13,	486
'Gauchos': 14,	1669
'Halter': 15,	324
'Henley': 16,	49
'Hoodie': 17,	17
'Jacket': 18,	716
'Jeans': 19,	4048
'Jeggings': 20,	10467
'Jersey': 21,	7076
'Jodhpurs': 22,	594
'Joggers': 23,	748
'Jumpsuit': 24,	45
'Kaftan': 25,	4416
'Kimono': 26,	6153
'Leggings': 27,	126
'Onesie': 28,	2294
'Parka': 29,	5013
'Peacoat': 30,	70
'Poncho': 31,	676
'Robe': 32,	97
'Romper': 33,	791
'Sarong': 34,	150
'Shorts': 35,	7408
'Skirt': 36,	32
'Sweater': 37,	19666
'Sweatpants': 38,	14773
'Sweatshorts': 39,	13123
'Tank': 40,	3048
'Tee': 41,	1106
'Top': 42,	15429
'Trunks': 43,	36887
'Turtleneck': 44}	total-----
	217071

<그림 19>

각 lable에

존재하는 데이터
개수

<그림 18> Label 이름과 번호

모델을 학습시키기 전에 데이터의 크기를 227*227로 resize 시키고 가운데 부분은 size만큼 자르는 CenterCrop 기법을 사용했습니다. 그리고 ToTensor() 함수를 통해 이미지를 tensor로 변경한 후 Normalize 정규화를 진행했습니다.

```
1 p = transforms.Compose([
2     transforms.Resize([227,227]),
3     transforms.CenterCrop(224),
4     transforms.ToTensor(),
5     transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
6 ])
```

<그림 20> 데이터 resizing과 정규화

또한 <그림 18>에서 확인할 수 있듯이 45개 Label 중 데이터가 아주 적은 label들이 존재하기 때문에 training data와 test data를 random한 방식으로 나눌 경우 train에 포함이 되지 않는 데이터가 발생할 가능성이 있어 sklearn을 이용해 stratify split를 진행했습니다.

```
1 from sklearn.model_selection import train_test_split
2 from torchvision.datasets import ImageFolder
3
4 dataset = ImageFolder(root='./ML_data', transform=p)
5 train_indices, val_indices = train_test_split(list(range(len(dataset.targets))), test_size=0.2, stratify=dataset.targets)
6 train_dataset = torch.utils.data.Subset(dataset, train_indices)
7 val_dataset = torch.utils.data.Subset(dataset, val_indices)
```

<그림 21> Training, Test 데이터셋 나누기

2. 데이터 전처리

Pytorch의 ImageFolder를 사용하기 위해서는 <그림 22>와 같은 계층적인 폴더구조를 가져야합니다. 즉, 이미지들이 자신의 label 이름으로 된 폴더 안에 들어가 있는 구조가 되어야 하고 실제 저희가 사용한 데이터셋도 그러한 구조를 갖습니다.

```
dataset/
  0/
    0.jpg
    1.jpg
    ...
  1/
    0.jpg
    1.jpg
    ...
```

<그림 22> ImageNet 계층 구조

mini batch 학습을 진행하기 위해 <그림 22>와 같이 train_data_loader와 test_data_loader를 생성했으며 batch_size는 256으로 설정했습니다.

```
1 train_data_loader = torch.utils.data.DataLoader(train_dataset, batch_size=256, shuffle=True, num_workers=2, drop_last=True)
2 test_data_loader = torch.utils.data.DataLoader(val_dataset, batch_size=256, shuffle=True, num_workers=2, drop_last=True)
```

<그림 23> Mini batch learning 사전준비

3. 모델 학습

① AlexNet

```
1 import torchvision.models as models
2 model = models.alexnet(pretrained=True)

1 model
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

<그림 24> Pretrained AlexNet Model

첫 번째로 학습시킨 모델은 AlexNet입니다. Pretrained 된 모델을 load 해 사용했으며 <그림 25>와 같이 전체적인 모델의 파라미터들은 freeze시키고 가장 마지막 layer만 unfreezing 시켜 모델을 학습시켰습니다.

```
1 model.classifier[6] = nn.Linear(in_features=4096, out_features=45, bias=True)

1 for param in model.parameters():
2     param.requires_grad = False

1 for name, param in model.named_parameters():
2     # print(name)
3     if name in ['classifier.6.weight', 'classifier.6.bias']:
4         param.requires_grad = True
```

<그림 25> 모델 freeze, unfreeze 설정

<그림 25>의 첫 번째 cell에서 모델의 output을 45개 뉴런으로 바꿔주었고 다음 cell에서 param.requires_grad = False을 이용하여 모든 layer들의 weight들을 freeze시킨 다음 마지막 layer만 다시 unfreeze 해주었습니다

모델의 목적이 이미지 분류이기 때문에 cross entry loss를 사용했고 optimizer로는 Adam을 사용했습니다.

```
1 import torch.optim as optim
2 import torch.nn as nn
3
4 criterion = nn.CrossEntropyLoss()
5 optimizer = optim.Adam(model.parameters(), lr=0.001)
```

<그림 26> cross entry와 optimizer 설정

다음은 모델을 학습시킨 부분입니다.

```
1 loss_list = []
2 model.train()
3 for epoch in range(3):
4     running_loss = 0.0
5     for i, data in enumerate(train_data_loader, 0):
6
7         inputs, labels = data[0].to(device), data[1].to(device)
8
9         # zero the parameter gradients
10        optimizer.zero_grad()
11
12        # forward + backward + optimize
13        output = model(inputs)
14        loss = criterion(output, labels)
15        loss.backward()
16        optimizer.step()
17
18        print(loss.item())
19        loss_list.append(loss.item())
20    torch.save(model, f'./model_unfreeze_three_layers_{epoch}.pt')
21    print(f'----- {epoch} finished!!! -----')
22 print('Finished Training of AlexNet')
```

<그림 27> AlexNet 모델 학습

앞에서 만들었던 train_data_loader를 불러와 input과 label을 생성한 후 loss를 계산합니다. optimizer.zero_grad()는 역전파 단계전에 optimizer 객체를 사용해 모델의 학습 가능 가중치인 갱신할 변수들에 대한 모든 변화도(gradient)를 0으로 만듭니다. 이렇게 하는 이유는 기본적으로 backward()를 호출할 때마다 변화도가 버퍼(buffer)에 덮어쓰지 않고 누적되기 때문입니다.

② GoogleNet

Alexnet과 마찬가지로 pretrained된 모델을 load해 <그림 29>와 같이 freeze와 unfreeze를 진행했습니다.

```
(Inception5b): Inception(
  (branch1): BasicConv2d(
    (conv): Conv2d(832, 384, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (branch2): Sequential(
    (0): BasicConv2d(
      (conv): Conv2d(832, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(192, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicConv2d(
      (conv): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(384, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (branch3): Sequential(
    (0): BasicConv2d(
      (conv): Conv2d(832, 48, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicConv2d(
      (conv): Conv2d(48, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (branch4): Sequential(
    (0): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, cell_mode=True)
    (1): BasicConv2d(
      (conv): Conv2d(832, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
)
(aux1): None
(aux2): None
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(dropout): Dropout(p=0.2, inplace=False)
(fc): Linear(in_features=1024, out_features=1000, bias=True)
```

<그림 28> GoogleNet Model load

첫 번째 cell은 모델 output 부분을 45개 뉴런으로 바꿔준 부분이고 다음 cell에서 param.requires_grad = False을 이용하여 우선 모든 layer들의 weight들을 freeze시킨 다음 마지막 layer만 다시 unfreeze 해주었습니다. 이 밖의 모델 학습 부분은 AlexNet과 완전히 동일합니다.

```
1 model.fc = nn.Linear(in_features=1024, out_features=45, bias=True)

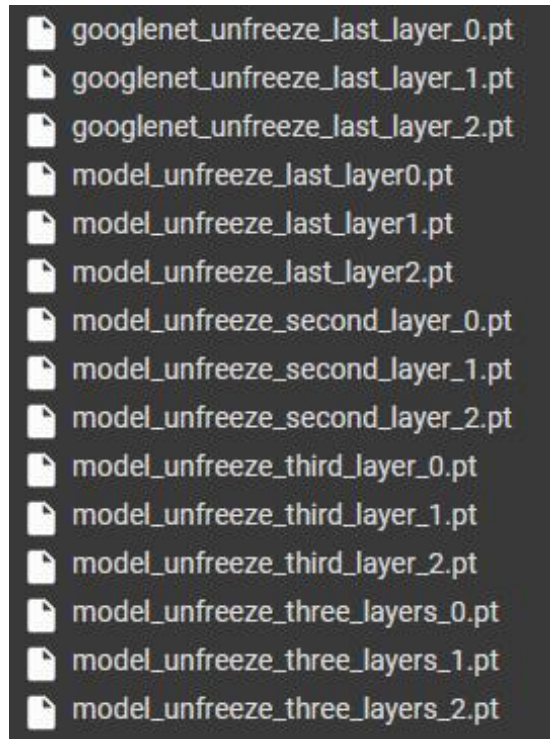
1 for param in model.parameters():
2     param.requires_grad = False

1 for name, param in model.named_parameters():
2     # print(name)
3     if name in ['fc.weight', 'fc.bias']:
4         param.requires_grad = True
```

<그림 29> GoogleNet freezing 설정

6. 모델 학습 결과

학습 결과를 매 epoch마다 저장했고 3 epoch 씩 5개 모델을 돌렸습니다. 'model_' 로 시작하는 파일은 AlexNet의 학습 데이터이고 'googlenet_'으로 시작하는 파일은 GoogleNet의 학습 데이터입니다. GoogleNet은 모델의 성능이 좋지 않아 3개 모델만 생성했습니다.



<그림 30> AlexNet, GoogleNet 모델 학습

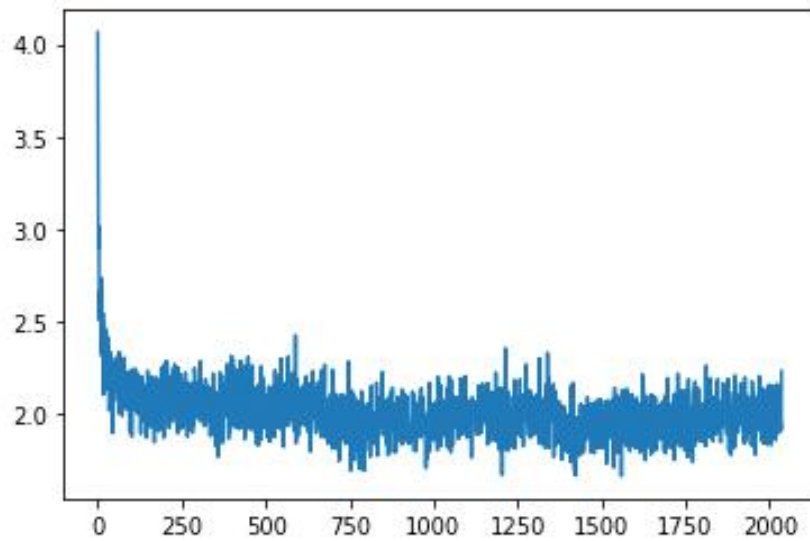
<그림 30>의 파일들 중 'unfreeze_last_layer'로 끝나는 파일은 마지막 layer를 제외한 모든 layer를 freeze시킨 모델들입니다. 마지막에 붙어있는 '_0' 등과 같은 번호는 epoch 번호를 의미합니다.

'unfreeze_second_layer'로 끝나는 파일은 마지막에서 두 번째 layer를 제외한 모든 layer를 freeze시킨 모델입니다. 이 때 마지막 layer는 학습을 완료한 후에 freeze 시켰습니다.

'unfreeze_third_layer'로 끝나는 파일은 마지막에서 세 번째 layer만 제외하고 모든 layer를 freeze했을 때의 모델입니다. 이 때 마지막 layer와 마지막에서 두 번째 layer는 학습을 완료한 후에 freeze 시켰습니다.

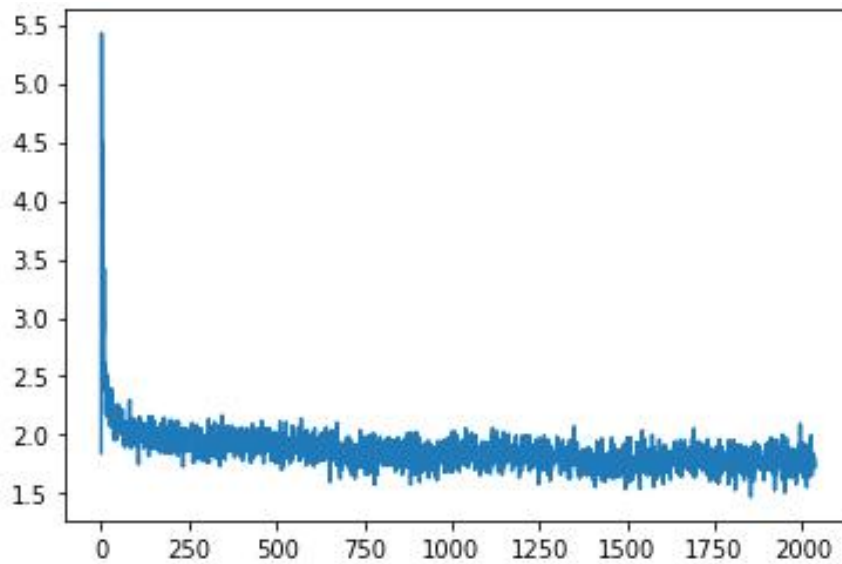
마지막으로 'unfreeze_three_layers'로 끝나는 파일은 fully connected layer모두를 한 번에 학습시킨 모델입니다.

<그림 31>은 마지막 layer만 unfreezing한 모델의 loss 그래프입니다. loss가 어느 정도 작아진 것을 확인할 수 있습니다. 또한 mini batch 방식으로 학습했기 때문에 그래프가 위아래로 요동치는 것을 볼 수 있습니다. Batch size를 256으로 잡아 약 2000번의 iteration이 발생한 것을 확인할 수 있습니다.. (전체데이터 size * 0.8(train_test_split)) / (256) = 약 678이므로 한 epoch가 678이고 3 epoch를 돌렸기 때문에 678 * 3 = 2034이라는 값을 갖습니다.



<그림 31> Unfreezing last layer only 모델의 loss

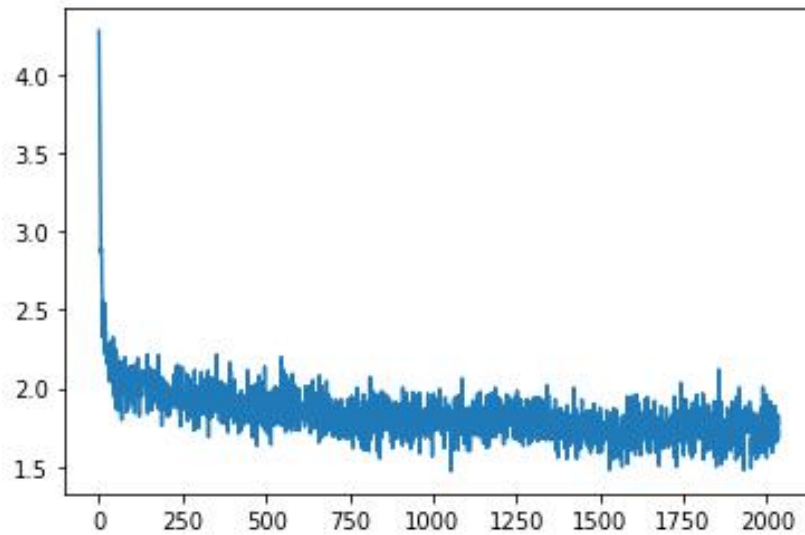
<그림 32>는 마지막에서 두 번째 layer만 unfreezing 한 상태로 학습시킨 모델의 loss 그래프입니다.



<그림 32> Unfreezing second last layer only 모델의 loss

loss가 <그림 31>보다 안정적으로 변한 것을 확인할 수 있는데, 이는 마지막 layer를 학습시킨 후에 마지막에서 두 번째 layer만 unfreezing했기 때문인 것으로 추정됩니다. 또한 loss도 어느 정도 작아진 것을 확인할 수 있었습니다.

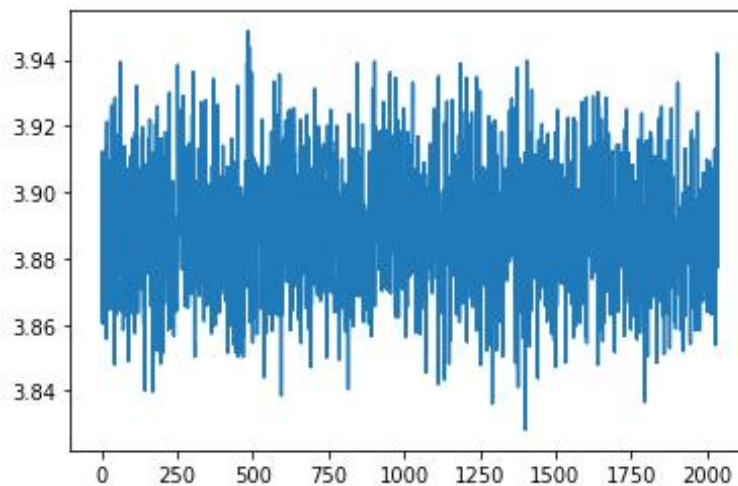
마지막으로 <그림 >은 마지막에서 세 번째 layer를 unfreezing 모델의 loss 그래프입니다.



<그림 33> Unfreezing third last layer only 모델의 loss

loss_only_unfreeze_second의 loss와 매우 비슷했지만 loss가 조금 더 감소하는 경향이 있음을 확인했습니다.

<그림 34>는 GoogleNet의 loss 그래프입니다. GoogleNet은 loss가 전혀 떨어지지 않았는데, 이는 이미지 데이터가 간단하고 모델 자체가 복잡하기 때문이라고 추측하였습니다.



<그림 34> GoogleNet loss 그래프

7. 모델 테스트

<그림 35>는 모델 성능 테스트 코드입니다.

```
1 def test(epoch, model, data_loader, criterion, device):
2     model.eval()
3     with torch.no_grad():
4         total = 0
5         correct = 0
6         cnt = 0
7         for i, (imgs, labels) in enumerate(data_loader):
8             imgs, labels = imgs.to(device), labels.to(device)
9             outputs = model(imgs)
10            total += imgs.size(0)
11            _, argmax = torch.max(outputs, 1)
12            correct += (labels == argmax).sum().item()
13            cnt += 1
14
15     print('Accuracy: {:.2f}%'.format(correct / total * 100))
```

<그림 35> 이미지 분류 모델 테스트 코드

```
alexnet_model_unfreeze_last_layer0.pt
Accuracy: 42.81%
-----
alexnet_model_unfreeze_last_layer1.pt
Accuracy: 43.47%
-----
alexnet_model_unfreeze_last_layer2.pt
Accuracy: 42.64%
-----
alexnet_model_unfreeze_second_layer_0.pt
Accuracy: 45.86%
-----
alexnet_model_unfreeze_second_layer_1.pt
Accuracy: 47.50%
-----
alexnet_model_unfreeze_second_layer_2.pt
Accuracy: 47.93%
-----
alexnet_model_unfreeze_third_layer_0.pt
Accuracy: 47.34%
-----
alexnet_model_unfreeze_third_layer_1.pt
Accuracy: 47.56%
-----
alexnet_model_unfreeze_third_layer_2.pt
Accuracy: 49.91%
-----
googlenet_unfreeze_last_layer_0.pt
Accuracy: 1.39%
-----
googlenet_unfreeze_last_layer_1.pt
Accuracy: 1.39%
-----
googlenet_unfreeze_last_layer_2.pt
Accuracy: 1.36%
-----
alexnet_model_unfreeze_three_layers_0.pt
Accuracy: 47.43%
-----
alexnet_model_unfreeze_three_layers_1.pt
Accuracy: 49.00%
-----
alexnet_model_unfreeze_three_layers_2.pt
Accuracy: 50.54%
```

<그림 36> 이미지 분류 모델 테스트 결과

<그림 36>은 모델을 테스트한 결과입니다. 우선 GoogleNet모델의 경우 성능이 1% 대로 매우 좋지 않음을 확인할 수 있습니다. AlexNet 모델은 40% ~ 50% 정도의 정확도가 나왔습니다. 이는 만족스럽지 않은 정확도인데, 데이터의 label이 45로 많은 편이고 데이터가 너무 적은(17개, 32개, 45개, 49개, 90개 등) label이 존재하기 때문에 stratify split을 진행하게 되면 데이터의 양이 너무 적어져 학습 제대로 되지 않았다고 추측하였습니다. AlexNet의 fully connected layer를 학습시킨 결과가 50.54%로 모든 모델들 중 가장 좋았기 때문에 프로젝트에는 해당 모델을 사용했습니다.

① 모델 테스트 1

학습시킨 AlexNet 모델에 <그림 37>과 같은 한 장의 Image를 넣어 결과를 테스트해봤습니다. <그림 37>의 이미지 분류는 'Coat'입니다.



<그림 37> 테스트 이미지 1

<그림 38>의 결과를 보면 Coat 이미지를 넣었지만 Blazer로 분류가 된 것을 알 수 있습니다. 그런데 Blazer로 분류된 이미지들 중에는 <그림 39>와 같이 코드 이미지와 굉장히 비슷해보이는 이미지들이 존재합니다.

```
1 img = image.open('/content/drive/Shareddrives/ML_project/ML_data/Coat/Coat_1.jpg')
2 img = p(img)
3 img = img.to(device)
4 img = img.unsqueeze(0)
5 model = torch.load('/content/drive/MyDrive/Colab Notebooks/model_unfreeze_three_layers_2.pt')
6 model = model.to(device)
7
8 model.eval()
9 with torch.no_grad():
10     result = model(img)
11
12 dataset.classes[torch.argmax(result,1).item()]

'Blazer'
```

<그림 38> 테스트 이미지 분류 결과



<그림 39> Blazer Image

실제로 <그림 40>을 보면 모델이 최종적으로 예측한 분류는 Blazer이지만 Coat역시 두 번째로 높은 점수를 가지는 것을 확인할 수 있습니다. Blazer는 1번 (2.8412)이고 Coat는 9번 (2.1693)입니다. 따라서 이 경우, 모델이 분류를 완전히 잘못된 것은 아니라고 판단하였습니다.

```
1 result
tensor([[ -2.8199,  2.8412,  0.7742, -2.6334, -4.9682, -6.5868, -4.2117,  1.0055,
          -0.5113,  2.1693, -8.5969, -2.0406, -4.8272, -2.1140, -2.4399, -5.8876,
          -2.8140, -1.1900,  2.1135,  0.8867, -2.8514, -1.6841, -4.6279, -0.8906,
           1.3120, -9.6252, -2.5809,  0.0918, -3.0807, -0.6386,  1.0313, -3.4927,
          -4.1043, -1.0838, -9.8231, -0.1761, -0.5760,  0.8558, -0.4257, -1.9693,
          -1.5436,  0.3700, -0.4834, -6.7606, -3.0478]], device='cuda:0')
```


<그림 40> Test image 분류 결과

② 모델 테스트 2

두 번째로 shirt 이미지를 테스트 해봤습니다.

```
1 img = Image.open('/content/drive/SharedDrives/ML_project/ML_data/Skirt/Skirt_1.jpg')
2 img = p(img)
3 img = img.to(device)
4 img = img.unsqueeze(0)
5 model = torch.load('./model_unfreeze_three_layers_2.pt')
6 model = model.to(device)
7
8 model.eval()
9 with torch.no_grad():
10     result = model(img)
11
12 dataset.classes[torch.argmax(result,1).item()]
'Skirt'

1 Image.open('./ML_data/Skirt/Skirt_1.jpg')
```



<그림 41> Skirt image 테스트 결과

skirt 이미지는 잘 예측이 된 것을 확인할 수 있습니다.

정확한 테스트를 위해 다른 모델들 중 가장 높은 정확도를 가지는 마지막에서 두 번째 layer만 unfreezing 한 AlexNet 모델을 사용해 한번 더 예측을 진행하였는데 <그림 42>와 같이 잘못된 예측 결과가 나왔습니다.

```

1 img = Image.open('./ML_data/Skirt/Skirt_1.jpg')
2 img = p(img)
3 img = img.to(device)
4 img = img.unsqueeze(0)
5 model = torch.load('./model_unfreeze_third_layer_2.pt')
6 model = model.to(device)
7
8 model.eval()
9 with torch.no_grad():
10     result = model(img)
11
12 dataset.classes[torch.argmax(result,1).item()]


```

'Tank'

```

1 Image.open('./ML_data/Tank/Tank_3.jpg')

```



<그림 42> 두 번째 모델 사용 예측 결과

같은 Skirt 이미지를 넣었지만 서로 다른 결과가 나온 것은 <그림 41>의 분류된 사진과 비교했을 때 <그림 42>의 사진이 비슷한 탑 상의를 입고 있기 때문이라고 판단했습니다.

3. 프로젝트 실습

1. 사진 배경 제거

다음은 실제 촬영한 사진을 분류하기 위해 옷을 제외한 배경을 삭제하고 옷의 색을 탐지하기 위한 과정입니다. 코드 구현을 Google Colab에서 하였기 때문에 <그림 43>상에서는 사진의 경로가 Google Drive로 지정되어있습니다. 배경을 지우는데 사용할 numpy package와 cv2 package를 설치합니다. 기타 패키지는 Colab 사용과 K-means Clustering을 수행하기 위함입니다.

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import cv2
5 from sklearn.cluster import KMeans
6
7 from IPython.display import Image
8
9 import os
10 from google.colab import drive
11 drive.mount('/content/gdrive', force_remount=True)

```

<그림 43> 배경 제거 코드 1

Package들을 import 한 후 이미지를 불러옵니다. 이미지의 차원을 낮추기 위해 gray scale로 변환하고 이미지의 테두리를 따기 위한 mask의 임계값을 조절합니다.

```

[37] 1 # 이미지 불러오기
2 img=cv2.imread('/content/gdrive/My Drive/철자켓.jpg')
3 #img=cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
4
5 # 변환 gray
6 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
7
8 # 임계값 조절
9 mask = cv2.threshold(gray, 250, 255, cv2.THRESH_BINARY)[1]
10
11 # mask
12 mask = 255 - mask

```

<그림 44> 배경 제거 코드 2

```

[38] 1 # anti-alias the mask
2 # blur alpha channel
3 mask = cv2.GaussianBlur(mask, (0,0), sigmaX=2, sigmaY=2, borderType = cv2.BORDER_DEFAULT)
4
5 # linear stretch so that 127.5 goes to 0, but 255 stays 255
6 mask = (2*(mask.astype(np.float32))-255.0).clip(0,255).astype(np.uint8)

[39] 1 # put mask into alpha channel
2 result = img.copy()
3 result = cv2.cvtColor(result, cv2.COLOR_BGR2BGRA)
4 result[:, :, 3] = mask

```

<그림 45> 배경 제거 코드 3

배경 제거를 완료한 후에는 경로를 지정해 이미지를 저장할 수 있습니다. 배경 제거 과정을 거친 이미지를 바로 모델에 입력으로 넣을 경우 따로 저장할 필요가 없지만 여기서는 배경이 잘 제거되었는지 확인하기 위해 사진을 저장하고 확인해보았습니다.

```
1 # 저장
2 cv2.imwrite('/content/gdrive/My Drive/translated_청자켓.png', result)

True
```

<그림 46> 배경 제거 이미지 저장 코드

<그림 47>은 <그림 46>에서 지정한 경로에 저장된 배경이 제거 된 이미지입니다.



<그림 47> 배경 제거된 이미지



<그림 48> 배경 제거 전 이미지

이 이미지의 원본은 <그림 48>입니다. 위에서 구현한 배경 제거 알고리즘의 경우 <그림 48>같이 배경이 단색이고 흰색에 가까울수록 잘 동작하는 특징을 보였습니다. 배경을 제거하기 위해 이미지를 gray scaling 하기 때문에 그럴 것이라고 예상하였습니다.

2. 옷 색상 Clustering

배경을 제거한 옷의 색상을 탐지하기 위해 K-means Clustering 기법을 사용하는 과정입니다. 해당 설명에서는 <그림 45>에서 'result'라는 이름으로 배경 제거를 완료한 image를 그대로 사용합니다.

OpenCV를 사용해 이미지를 읽어올 경우 RGB 형식이 아닌 BRG 형식으로 읽어오기 때문에 <그림 49>와 같이 붉은색과 푸른색이 반전된 형태의 이미지가 나타납니다. 이 이미지를 `cv2.cvtColor(image, cv2.COLOR_BRG2RGB)` 함수를 통해 색 정렬을 바꿔주면 다시 <그림 50>과 같이 정상적인 색상으로 변경된 것을 확인할 수 있습니다. 또한 현재 (300 * 900, 3) 배열을 가지고 있는 픽셀을 일렬로 정렬해주기 위해 `reshape(-1, 3)`함수를 사용합니다.

전처리가 완료된 이미지를 가지고 <그림 51>과 같이 Clustering을 시작합니다. 기본적으로 옷의 대표색은 1~2가지 색이라고 생각하였고, 배경이 완전히 제거되지 않았을 경우를 대비해 클러스터의 개수를 3개로 설정했습니다.


```
1 plt.imshow(result)
```



<그림 49> BRG 이미지

```
1 #이미지 RGB값 재정렬
```

```
2 img=cv2.cvtColor(result,cv2.COLOR_BGR2RGB)
```

```
3 plt.imshow(img)
```

```
4 img = img.reshape(-1,3)
```



<그림 50> 전처리 완료된 이미지

클러스터링 시작

```
[43] 1 kmeans=KMeans(n_clusters=3)
      2 s=kmeans.fit(img)
      3
      4 labels=kmeans.labels_
      5 print(labels)
      6 labels=list(labels)
```

```
[2 2 2 ... 2 2 2]
```

```
[44] 1 centroid=kmeans.cluster_centers_
      2 print(centroid)
```

```
[[181,21588624 198,13509532 207,51015552]
 [125,80486826 138,32018189 144,52320449]
 [244,4211327 245,67249844 244,79012516]]
```

<그림 51> K=3으로 두고 클러스터링

세 개의 centroid 값이 나온 것을 확인할 수 있습니다.

<그림 52>와 같이 각 centroid의 점유 percent도 확인할 수 있습니다.

```
1 percent=[]
2 for i in range(len(centroid)):
3     j=labels.count(i)
4     j=j/(len(labels))
5     percent.append(j)
6 print(percent)
```

```
[0.49242677824267783, 0.10415620641562064, 0.40341701534170155]
```

```
1 print(type(centroid))
2 centerList = centroid.tolist()
3 print(centerList)
4 print(percent)
5
```

```
<class 'numpy.ndarray'>
[[181,21588623551452, 198,13509532308092, 207,51015551967498], [125,
 [0.49242677824267783, 0.10415620641562064, 0.40341701534170155]]
```

<그림 52> percent 와 그에 상응하는 centroid 값

centroid 값을 percent 순으로 정렬해주었습니다.

```

1 sorted_center = []
2 sorted_percent = []
3
4 for i in range(len(centerList)):
5     idx = percent.index(max(percent))
6     sorted_center.append(centerList[idx])
7     sorted_percent.append(percent[idx])
8     del percent[idx]
9
10 percent = sorted_percent
11 print(percent)
12 print(sorted_center)
13

[0.49242677824267783, 0.40341701534170155, 0.10415620641562064]
[[181,21588623551452, 198,13509532308092, 207,51015551967498], [1

1 centroid = np.array(sorted_center)

```

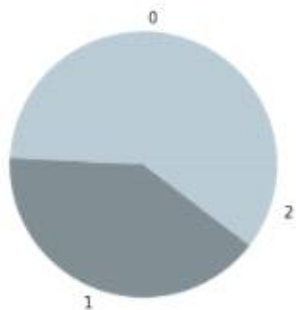
<그림 53> percent 값 기준 centroid 정렬

Clustering이 잘 된 것을 확인할 수 있습니다.

```

1 plt.pie(percent, colors=np.array(centroid/255), labels=np.arange(len(centroid)))
2 plt.show()

```



<그림 54> 클러스터링 결과

이제 Clustering 결과를 바탕으로 centroid의 값을 color name으로 표현하기 위해 webcolors package로부터 rgb_to_name() 함수를 import 해 사용하였습니다.

```

1 !pip install webcolors
2
3 from scipy.spatial import KDTree
4 from webcolors import rgb_to_name, CSS3_HEX_TO_NAMES, hex_to_rgb
5
6 def convert_rgb_to_names(rgb_tuple):
7
8     # a dictionary of all the hex and their respective names in css3
9     css3_db = CSS3_HEX_TO_NAMES
10    names = []
11    rgb_values = []
12    for color_hex, color_name in css3_db.items():
13        names.append(color_name)
14        rgb_values.append(hex_to_rgb(color_hex))
15
16    kdt_db = KDTree(rgb_values)
17    distance, index = kdt_db.query(rgb_tuple)
18    color = names[index]
19    return color
20
21 print(convert_rgb_to_names(first_color))

```

lightsteelblue

<그림 55> 클러스터링 결과

'lightsteelblue'라는 색상 명이 잘 출력되는 것을 확인할 수 있습니다.

4. 프로젝트 구현

⑤ 타코트론 모델 구현 (Text-to-Speech 서비스)

```

1 waveglow = torch.hub.load('NVIDIA/DeepLearningExamples:torchhub', 'nvidia_waveglow', model_math='fp16', map_location=torch.device('cpu'))
2 waveglow = waveglow.remove_weightnorm(waveglow)
3 #waveglow = waveglow.to(device)
4 waveglow.eval()

```

Using cache found in /root/.cache/torch/hub/NVIDIA_DeepLearningExamples_torchhub
 Downloading checkpoint from <https://git.io/nvidia-waveglow-checkpoint>
 /root/.cache/torch/hub/NVIDIA_DeepLearningExamples_torchhub/PyTorch/SpeechSynthesis/Tacotron2/waveglow/model.py:55: UserWarning: torch.qr is deprecated. The boolean parameter 'some' has been replaced with a string parameter 'mode'.
 0, R = torch.qr(A, some)
 should be replaced with
 0, R = torch.linalg.qr(A, 'reduced' if some else 'complete') (Triggered internally at ../aten/src/ATen/native/BatchLinearAlgebra.cpp:1937.)
 #W = torch.qr(torch.FloatTensor(c, c).normal_())[0]
 WaveGlow(
 (upsample): ConvTranspose1d(80, 80, kernel_size=(1024,), stride=(256,))
 (WN): ModuleList(
 (0): WN(
 (in_layers): ModuleList(
 (0): Conv1d(512, 1024, kernel_size=(3,), stride=(1,), padding=(1,))
 (1): Conv1d(512, 1024, kernel_size=(3,), stride=(1,), padding=(2,), dilation=(2,))
 (2): Conv1d(512, 1024, kernel_size=(3,), stride=(1,), padding=(4,), dilation=(4,))
 (3): Conv1d(512, 1024, kernel_size=(3,), stride=(1,), padding=(8,), dilation=(8,))
 (4): Conv1d(512, 1024, kernel_size=(3,), stride=(1,), padding=(16,), dilation=(16,))
 (5): Conv1d(512, 1024, kernel_size=(3,), stride=(1,), padding=(32,), dilation=(32,))
 (6): Conv1d(512, 1024, kernel_size=(3,), stride=(1,), padding=(64,), dilation=(64,))

<그림 56> 음성의 주파수 특징을 분석한 Mel Spectrum 데이터

Mel Spectrum은 음성 데이터의 주요한 정보들이 효율적으로 정리되어서 담겨져 있습니다. 하지만 Mel Spectrum을 곧바로 음성으로 변환할 수 없기에 vocoder를 사용해서 멜 스펙트럼을 음성 데이터로 변환하기 위한 추가적인 작업이 필요합니다. 이를 위해서 waveglow를 사용합니다. waveglow는 Normalizing Flow 기반의 뉴럴 vocoder입니다. WaveGlow는 가역성을 지닌 변환 함수를 이용해서 음성 데이터셋으로부터 가우시안 분포와 같이 단순한 분포가 나오도록 학습을 합니다. 학습이 끝난 후 변환 함수(f)의 역함수를 이용해서 가우시안 분포의 샘플로부터 음성을 합성합니다. waveglow를 사용하기 위해서 torch.hub에서 pretrained된 모델을 불러옵니다.

```
1 utils = torch.hub.load('NVIDIA/DeepLearningExamples:torchhub', 'nvidia_tts_utils')
2 sequences, lengths = utils.prepare_input_sequence([text])

Using cache found in /root/.cache/torch/hub/NVIDIA_DeepLearningExamples_torchhub
```

<그림 57> Tacotron 형식 지정

utils를 이용해서 입력 형식을 지정합니다.
 생성된 sequences와 lengths를 to(device)를 이용해서 cpu를 사용할 수 있게 합니다.
 그 다음 체인 모델을 실행합니다.

```
1 sequences = sequences.to(device)
2 lengths = lengths.to(device)

1 with torch.no_grad():
2     mel, _, _ = tacotron2.infer(sequences, lengths)
3     audio = waveglow.infer(mel)
4 audio_numpy = audio[0].data.cpu().numpy()
5 rate = 22050
```

<그림 58> 체인 모델 실행

마지막으로 write를 이용해서 audio 파일을 만든 후 저장하고 IPython.display의 Audio를 이용해서 파일을 실행시킵니다.

```
1 from scipy.io.wavfile import write
2 write("audio.mp3", rate, audio_numpy)

1 from IPython.display import Audio
2 Audio(audio_numpy, rate=rate, autoplay=True)
```



<그림 59> 음성 파일 출력

⑥ 프로젝트 구현

실제 모델의 동작 방식은 다음과 같습니다.

1. 라즈베리 파이의 카메라로 옷 사진을 찍고 배경을 제거한다.
2. 배경을 제거한 이미지를 AlexNet 모델에 입력값으로 넣어 옷을 분류한다.
3. 배경을 제거한 이미지를 Clustering 모델에 넣어 색상 값을 찾아낸다.
4. 타코트론 모델을 통해 옷의 종류와 색상 값을 음성으로 출력한다.

4. 예측 결과

총 6장의 사진을 예측했습니다.



<그림 60> Label = Sweater

<그림 62>을 보면 sweater는 색상과 분류 모두 올바르게 예측됨을 확인할 수 있습니다. 두 번째 사진은 검은 티셔츠입니다. label을 tee로 주었습니다. <그림 64>을 보면 두 번째 사진 역시 색상과 분류 모두 올바르게 예측됨을 알 수 있습니다.



<그림 61> label = tee

```

1 img = Image.open('./Sweater.jpg')
2 img = p(img)
3 img = img.to(device)
4 img = img.unsqueeze(0)
5 model = model.to(device)
6
7 model.eval()
8 with torch.no_grad():
9     result = model(img)
10
11 prediction = data[torch.argmax(result,1).item()]
12 prediction

```

'Sweater'

```

1 Image.open('./Sweater.jpg').resize([300,300])

```



<그림 62> Sweater 예측 결과 - 맞음

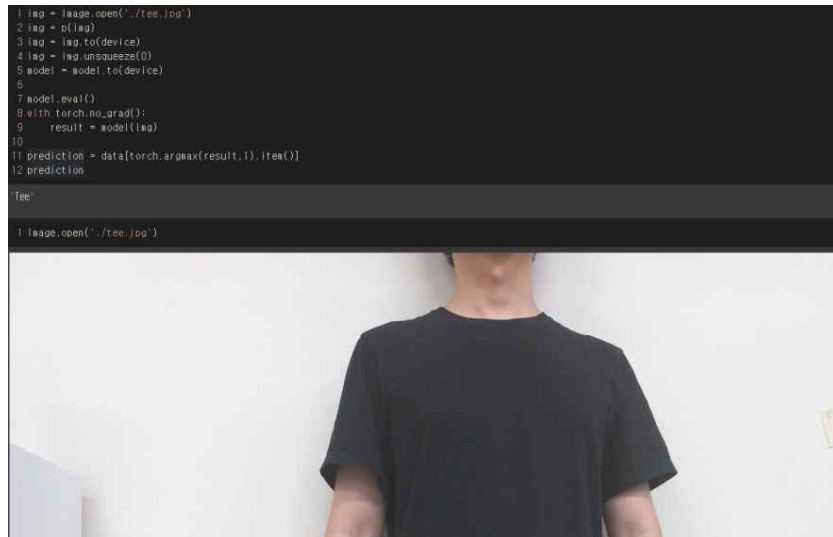
```

1 result += ' and type is '
2 result += prediction
3 result

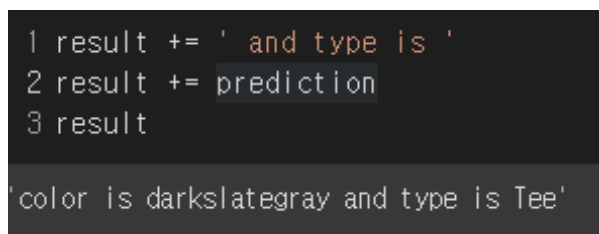
```

'color is silver and type is Sweater'

<그림 63> Color detection



<그림 64> tee 예측결과 - 맞음



<그림 65> Color detection 2

결론

1. 어려웠던 점

프로젝트를 시작할 때 구글 colab에 데이터를 올리려고 하는데 계속해서 문제가 발생해서 label끼리 사진을 합친 다음 zip파일로 올려서 압축을 해제해서 해결을 했지만 몇시간 뒤 다시 데이터를 사용하려고 할 때 데이터가 자꾸 소실되는 현상이 발생했습니다. 이 부분이 제일 어려웠습니다.

모델 자체가 파라미터의 수가 많고 데이터의 수도한 많기에 학습을 하는 데 너무 오랜 시간이 걸려 이를 위해 코랩 프로를 구입하기도 했고 모델의 크기가 200MB가 넘어가기에 github에 올릴 수 있는 용량의 파일은 100MB가 최대이기에 올릴 수 없다는 문제점도 있었습니다.

또한 Color detection을 위해 배경을 제거하는 과정에서 배경을 제거한 이미지를 사용해도 제거된 부분이 다시 흰색으로 표시되는 경우가 있어 곤란했습니다.

2. 한계

AlexNet의 모델의 정확도가 약 51%로 높은 수준이 아닙니다. 또한 어떤 레이블은 데이터가 많지만 어떤 레이블은 데이터의 수가 굉장히 적기에 stratify split을 진행하면 데이터의 수가 더 적어져서 데이터의 수가 적은 레이블에 대해서 제대로 된 예측이 이루어지지 않았습니다.

배경 제거 역시 이미지의 배경이 단색이거나, 흰색이 아닐 경우 제대로 제거가 되지 않는다는 한계점이 존재합니다.

3. 의의

정확도가 많이 높지는 않지만 절반 정도의 확률로 정확한 detection이 가능하므로 모델의 성능을 향상시키고 더 정확한 데이터셋과 좋은 성능의 배경 제거 모델을 사용한다면 실제로 사용 가능할 정도의 모델을 구현할 수 있을 것이라 생각합니다. 또한 소프트웨어를 개발해 실제 하드웨어에서 동작할 수 있도록 구현한 점에서 의미가 있다고 생각합니다.