생활폐기물 및 재활용품 이미지 분류
: **ResNet-34**을 이용한 전이학습을 중심으로

TEAM 7

I. 개요

전 세계적으로 재활용품 분류 체계가 제대로 정립되지 않아 자원의 효율적인 재활용이 어려운 상황이다. 재활용율이 상대적으로 높은 우리나라도 아직까지 많은 부분을 수작업으로 해결하고 있다. 예를 들어, 작업자들은 재활용품 별로 분류된 봉지를 일일이 풀고, 재질과 상태에 따라 잘못 분류된 폐기물을 직접 손으로 골라낸다. 이 과정에서 작업자들은 악취, 미세먼지, 유해물질 등에 지속적으로 노출되기 때문에 신체적 건강을 위협받고 있다. 또한 폐기물 처리 시설은 작업 환경이 열악한 경우가 많아 작업자들의 정신적 스트레스도 큰 문제로 대두되고 있다.

위와 같은 사회 문제를 고려하여 팀은 누구나 사용할 수 있는 재활용품 분류 모델을 구현하여 작업자들이 겪는 신체적, 정신적 부담을 완화하고, 스마트 자원 관리의 기초 기술의 발판이 되는 모델을 구현하고자 한다. 해당 모델을 통해 수작업으로 이루어지던 영역의 자동화를 이루어 작업시간을 단축시키고 아울러 전세계적으로 심각한 쓰레기 재활용 문제에 기여할 수 있기를 기대한다.

II. 데이터셋

1) 데이터 선별

본 프로젝트에서는 AiHub에서 제공하는 데이터를 활용했다. 아래 그림1을 보면 사이트에서 제공하는 원본 데이터셋의 용량이 도합 약 3400GB를 육박하는 것을 확인할 수 있다. 본 연구에서는 기술적인 제약으로 인해 제공되는 데이터셋을 모두 활용하지 못하고 연구 목표에 맞추어 일부 데이터만을 선별하여 사용했다.

데이터 내용	품목	데이터 형식	데이터 수량	데이터 크기
개별 재활용품 이미지	대분류 9종 (세부분류 15종)	jpg 이미지 파일	706,101건	약 2,700GB
재활용품 선별영상 추출 이미지	대분류 7종 (세부분류 13종)	jpg 이미지 파일	299,764건	약 170GB
라벨링 데이터	대분류 9종 (세부분류 15종)	json 라벨링 파일	1,005,865건	약 6GB

<그림1: 원본 데이터셋>

원본 데이터는 데이터 수집 방식에 따라 3개의 카테고리로 분류된다. 개별 이미지 데이터, 재활용품 선별영상 추출 이미지, 라벨링 데이터가 있는데 재활용품 선별영상 추출 이미지 데이터만을 활용하였다. 왜냐하면 개별 이미지 데이터를 활용한 분류 모델은 선행된 연구가 많고 라벨링 데이터는 메타데이터로 이미지 분류 태스크를 수행하기에는 적합하지 않다고 판단했다. 또한 프로젝트 목표는 실제 산업에서 컨베이어 벨트 위에서 더미로 존재하는 재활용품을 분류하고자 하는 것이기 때문에 재활용품 선별영상 추출 이미지 데이터가 가장 적합하다고 판단했다.

해당 데이터셋의 이미지는 총 13개의 카테고리로 분류된다. 아래 그림2에서 확인할 수 있듯이 해당 클래스는 철캔, 알루미늄캔, 종이, 무색단일, 유색단일, PE, PP, PS, 스티로폼, 비닐, 갈색 유리병, 녹색 유리병, 투명 유리병으로 PE, PP, PS와 같은 경우 모두 플라스틱에 해당하지만 세세한 분류가 이루어졌음을 확인할 수 있다. 하지만 본 프로젝트에서는 재활용품 가이드에 맞는 분류를 하고자 하기 때문에 소분류 보다는 데이터셋을 7개의 클래스로 분류한 중분류 기준을 사용하고자 한다. 이는 학습 및 추론 과정에서의 효율성을 높이고, 재활용 분류 시스템의 실질적인 활용 가능성을 극대화하기 위함이다.

이미지 종류	중분류	세부분류	파일 포맷	수량
선별영상 추출 이미지	01. 금속캔	001. 철캔	.jpg / .json	30,129
		002. 알류미늄캔	.jpg / .json	21,171
	02. 종이	001. 종이	.jpg / .json	21,176
	03. 페트병	001. 무색단일	.jpg / .json	39,538
		002. 유색단일	.jpg / .json	33,170
	04. 플라스틱	001. PE	.jpg / .json	25,273
		002. PP	.jpg / .json	39,088
		003. PS	.jpg / .json	39,009
	05. 스티로폼	001. 스티로폼	.jpg / .json	15,039
	06. 비닐	001. 비닐	.jpg / .json	9,015
	07. 유리병	001. 갈색	.jpg / .json	9,052
		002. 녹색	.jpg / .json	9,036
		003. 투명	.jpg / .json	9,068

<그림2: 재활용품 선별영상 추출 이미지 데이터셋>

그러나, 재활용품 선별영상 추출 이미지 데이터 역시 그 양이 방대하여 모두 활용할 수는 없기 클래스 별로 일부 데이터를 추출하여 활용했다. 클래스 간의 데이터 수가 다른 것을 고려하여 원본 데이터의 각 클래스 별 데이터 수의 비율을 고려하여 약 900장에서 1300장의 데이터를 랜덤하게 샘플링하여 사용했다.

2) 데이터 전처리

선별한 데이터셋의 이미지를 확인해보니 모델의 학습을 방해하는 이상치 데이터들이 많아 이를 정제한 데이터셋을 만들어 모델을 학습시켰다. 아래 그림3에 모이는 예시들과 같이 정상 데이터는 컨베이어벨트 위에서 촬영된 재활용품의 사진으로, 재활용품 선별 환경을 정확히 반영하는 데이터를 말한다. 정상데이터는 이미지 안의 모든 재활용품을

하나의 클래스로 분류할 수 있어야 하며, 컨베이어 벨트 위에 존재하고(너무 측면에 존재하지 않아야 한다.), 라벨링이 정상적으로 잘 이루어진 데이터를 뜻한다.

<그림3: 정상 데이터>

반면에 이상치 데이터는 크게 세 가지 유형으로 분류된다. 첫째는, 그림3의 첫 번째 예시처럼 서로 다른 클래스의 재활용품이 섞여 있는 이미지로 이미지 안의 모든 재활용품이 하나의 클래스로 분류되기 어려운 데이터이다. 둘째는, 아래 그림의 가운데 예시처럼 재활용품이 컨베이어벨트에서 벗어나거나 제대로 촬영이 되지 않은 이미지이다. 셋째, 그림의 마지막 예시처럼 데이터 라벨링이 잘못 된 경우로 그림3의 마지막 사진은 실제 클래스는 종이이지만 비닐로 잘못 라벨링 된 경우이다. 팀이 향후 사용하고자 하는 모델은 ResNet 중심이기 때문에 이러한 이상치 데이터는 모델의 성능을 떨어뜨릴 수 있다고 판단하여 제거한 후 학습했다.

<그림 4: 이상치 데이터>

데이터 전처리를 마친 후 train set, validation set, test set을 70%, 15%, 15%로 나누어 학습했다.

III. 모델링 및 학습 결과

컴퓨터 리소스의 문제로 데이터의 양과 학습 시간을 고려하여 기존의 이미지 분류 모델을 활용한 전이학습을 진행하고자 했다.

VGG 16 , VGG 19, Resnet-18, Resnet-34, Resnet-50 등 분류에서 좋은 성능을 보이는 여러 모델로 학습을 진행했다. 그 중 ResNet-34보다 무겁고 복잡한 모델의 경우 학습시간이 너무 길어져 본 프로젝트의 모델로 채택하기 적합하지 않았다. 따라서 이 중 가장 결과가 좋은 ResNet-34를 채택하여 미세조정을 통해 모델을 향상시키고자 했다. 이때 ResNet-34 여기 1000개의 클래스를 분류하는 모델이기 때문에 팀의 데이터셋을 학습시키기에 위해 충분한 모델이라고 판단했다.

Resnet-34는 ImageNet의 데이터셋을 학습한 모델로 이미지 분류에 우수한 성능을 보인다고 알려져 모델의 아키텍쳐와 파라미터를 최대한 활용하고자 앞부분은 변형하지 않았다. 다만 해당 모델은 1000개의 클래스를 분류하는 모델이므로 팀의 데이터셋에 맞추어 말단부에 층을 추가해 7개의 클래스를 분류하는 형태로 모델을 변경했다.

```
resnet = models.resnet34(pretrained=True)

# Freezing backbone parameters
if freeze_backbone:
    for param in resnet.parameters():
        param.requires_grad = False
self.freeze_backbone = freeze_backbone

# Retain all layers except the classifier
self.backbone = nn.Sequential(*list(resnet.children())[:-2])  # Exclude fc and avgpool layers
```

<그림5: ResNet-34 freeze>

아래 그림5는 모델의 출력 노드의 개수, 에포크 수, 학습률과 ResNet-34 모델 freezing 등 학습에 필요한 하이퍼파라미터의 기본 세팅을 나타낸다. 이때 학습률은10의 -4승 부터 10의 -3승까지 바꾸어가며 여러 번 학습했다.

```
# Hyperparameters
num_classes = 7
num_epochs = 100
learning_rate = 1e-3
freeze_backbone = False
```

<그림6: Hyper parameter>

아래 그림6은 옵티마이저와 손실함수 세팅을 나타낸다. 옵티마이저는 딥러닝 모델에서 많이 사용하는 Adam을, 손실함수는 분류 모델을 구현할 때 많이 사용하는 cross entropy loss를 사용했다.

```python
# 모델 초기화
model = R34(num_classes=num_classes, freeze_backbone=freeze_backbone).to('cuda' if torch.cuda.is_available() else 'cpu')

# 손실 함수와 옵티마이저 설정
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
#optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)

# 스케줄러(Optional)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

# 학습 및 검증 루프
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

<그림7: Optimizer와 Loss function>

ResNet-34로 모델을 정한 후 말단부 층을 2개, 3개, 4개로 구성해 각 모델의 성능을 비교했다. 각 층에서의 노드 수는 출력층에 가까워질수록 줄어들도록 구성했다. 층을 늘릴수록 모델의 성능이 좋아질 것이라고 예측이 되지만 학습시간을 고려하여 4개의 층까지만 시도할 수 있었다. 아래 표1의 코드는 ResNet-34 의 구조를 freeze 하고 말단부에 각각 층 2개, 3개, 4개를 추가한 모델을 구현한 것이다.

모델1(말단부 층 2개)	모델2(말단부 층 3개)	모델3(말단부 층 4개)

`def forward(self, x):`
`  x = F.relu(self.top_model(x))`
`  x = nn.AdaptiveAvgPool2d((1,1))(x)`
`  x = x.view(x.shape[0], -1) # flattening`
`  x = self.bn1(x)`
`  x = F.relu(self.fc1(x))`
`  x = self.bn2(x)`
`  x = self.fc2(x)`
`  return x`
	`def forward(self, x):`
	`  # Forward through the backbone (pret`
	`  x = self.backbone(x)  # Pass through`
	`  x = nn.AdaptiveAvgPool2d((1, 1))(x)`
	`  x = x.view(x.size(0), -1)  # Flatten`
	`  # Classifier part`
	`  x = self.bn1(x)`
	`  x = F.relu(self.fc1(x))`
	`  x = self.dropout1(x)`
	`  x = self.bn2(x)`
	`  x = F.relu(self.fc2(x))`
	`  x = self.dropout2(x)`
	`  x = self.bn3(x)`
	`  x = self.fc3(x)`
	`  return x`
		`def forward(self, x):`
		`  # Forward through the backbone (pretrained ResNet-34)`
		`  x = self.backbone(x)  # Pass through ResNet-34 feature extractor`
		`  x = nn.AdaptiveAvgPool2d((1, 1))(x)  # Global average pooling`
		`  x = x.view(x.size(0), -1)  # Flatten features`
		`  # Classifier part`
		`  x = self.bn1(x)`
		`  x = F.relu(self.fc1(x))`
		`  x = self.dropout1(x)`
		`  x = self.bn2(x)`
		`  x = F.relu(self.fc2(x))  # Intermediate FC layer`
		`  x = self.dropout2(x)`
		`  x = self.bn3(x)`
		`  x = F.relu(self.fc3(x))  # Intermediate FC layer`
		`  x = self.dropout3(x)`
		`  x = self.bn4(x)`
		`  x = self.fc4(x)  # Final output layer`
		`  return x`

<표1: 모델1, 모델2, 모델3 코드 구현>

각 층의 노드 수는 16부터 4096까지 변경해보며 성능을 비교했고 활성화함수로는 relu를 사용했다. 또한 과적합을 방지하기 위해 dropout을 사용했는데 dropout의 비율은 0.3부터 0.6까지 변경해보며 학습을 진행했다. 이때 얕은 층에서는 dropout의 비율을 작게 설정해 입력값의 정보가 많이 손실되지 않도록 했다.

아래의 표2은 모델1, 모델2, 모델3에 층 별 노드 개수와 dropout rate을 다르게 설정한 것 중 가장 좋은 결과를 나타낸다. 모델3은 첫 번째 층에서 512개 노드로 설정하고 층이 깊어질수록 노드 수를 절반으로 줄여 256개, 128개의 노드를 가지도록 설정했다. 또한 dropout rate은 0.3, 0.4, 0.5로 설정한 경우이다.

모델1(말단부 층 2개)	모델2(말단부 층 3개)	모델3(말단부 층 4개)
Overall Test Accuracy: 78.20% Class-wise Accuracy: Class 0: 100.00% (135/135) Class 1: 45.93% (62/135) Class 2: 42.96% (58/135) Class 3: 96.30% (130/135) Class 4: 72.59% (98/135) Class 5: 96.30% (130/135) Class 6: 93.33% (126/135)	Overall Test Accuracy: 80.74% Class-wise Accuracy: Class 0: 100.00% (135/135) Class 1: 57.78% (78/135) Class 2: 44.44% (60/135) Class 3: 97.78% (132/135) Class 4: 75.56% (102/135) Class 5: 96.30% (130/135) Class 6: 93.33% (126/135)	Overall Test Accuracy: 82.86% Class-wise Accuracy: Class 0: 100.00% (135/135) Class 1: 62.96% (85/135) Class 2: 47.41% (64/135) Class 3: 97.04% (131/135) Class 4: 81.48% (110/135) Class 5: 96.30% (130/135) Class 6: 94.81% (128/135)

<표2: 모델 성능 비교>

아래 그림8은 이 중 가장 성능이 좋은 최종 모델을 구현한 코드이다.

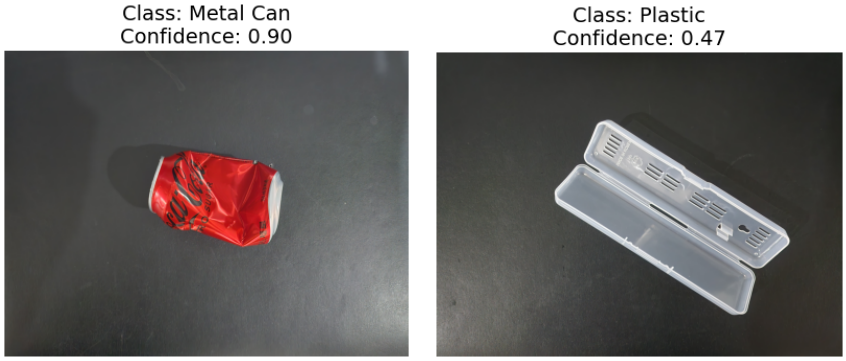
```
# Classifier part with additional layers
self.bn1 = nn.BatchNorm1d(512)
self.fc1 = nn.Linear(512, 512)
self.dropout1 = nn.Dropout(p=0.3)  # Dropout added
self.bn2 = nn.BatchNorm1d(512)
self.fc2 = nn.Linear(512, 256)  # Added intermediate FC layer
self.dropout2 = nn.Dropout(p=0.4)  # Another Dropout
self.bn3 = nn.BatchNorm1d(256)
self.fc3 = nn.Linear(256, 128)  # Added intermediate FC layer
self.dropout3 = nn.Dropout(p=0.5)  # Another Dropout
self.bn4 = nn.BatchNorm1d(128)
self.fc4 = nn.Linear(128, num_classes)  # Final classification layer
```

<그림8: 최종 모델>

IV. 결론

1) 실제 이미지의 적용

모델 성능을 실제 이미셋으로 검증해보기 위해 컨베이어 벨트 환경을 구축하여 이미지를 수집했다. 아래 그림9에서 보이는 것처럼 콜라 캔을 메탈 캔으로, 칫솔통을 플라스틱으로 잘 분류하는 것을 확인할 수 있다.



<그림9: 모델 적용 결과>

콜라캔이 해당 클래스에 속할 확률은 0.9로 매우 높았으나 칫솔통의 경우 플라스틱 클래스에 속할 확률이 0.47로 낮았다. 이는 팀이 구축한 컨베이어 벨트 환경이 실제 환경과의 유사도가 떨어지기 때문이라고 판단한다. 실제 산업 환경과 유사도가 높은 환경을 만들수록 모델의 정확성이 증가할 것으로 기대된다.

2) 모델의 개선점

GPU, 컴퓨터 내 용량 부족 등과 같은 기술적인 제약으로 성능이 우수한 ResNet-50 등과 같은 모델을 학습할 수 없었다. 또한 원본데이터(100만 장)을 전부 활용할 수 있는 메모리 환경이 아니었기 때문에 일부 데이터만 추출하여 모델을 학습했다는 아쉬움이 있다. 영상 추출 이미지 데이터와 더불어 개별 이미지 데이터를 같이 활용한다면 더 좋은 성능의 모델을 구현할 수 있을 것으로 기대된다.

제한적인 조건으로 구현한 팀의 모델은 특정 클래스만 잘 구분하는 경향을 보인다. 예를 들어, 메탈 캔, 유리병, 페트병과 같이 형태가 변형되기 어려운 클래스는 높은 정확도를 보이나 비닐, 스티로폼과 같이 형태가 변형되기 쉬운 클래스는 잘 구분하지 못하는 경향을 보인다. 이처럼 클래스 별 성능 차이에 대한 연구도 추가적으로 진행될 수 있다. 또한 YOLO 모델을 활용한다면 빠르게 움직이는 컨베이어 벨트 위의 재활용품을 효과적으로 분류하는 모델을 구축할 수 있을 것으로 기대된다.

V. 참조

- https://www.khan.co.kr/article/202402250800001
- https://www.youtube.com/watch?v=T3l2aF0z6Bo
- https://blog.lgchem.com/2023/01/26_geekble_collaboration/
- https://www.donga.com/news/It/article/all/20201231/104713889/1
- https://www.epa.gov/recycle/recycling-basics-and-benefits#:~:text=Recycling%20provides%20many%20benefits%20to,and%20minerals%20for%20new%20products

[팀 구성원 기여도**]**

2020147039 오동하 : 데이터 전처리, 모델링
2020147030 홍승은 : 데이터 전처리, 발표자료 제작
2022147047 김진웅 : 데이터 전처리, 모델링
2022121011 김윤성 : 선행연구 조사, 발표자료 제작