Pusan National University

Computer Science and Engineering

Technical Report 2023-10

라즈베리 파이를 이용한 스마트폰 부품 판별 시스템

201501199 윤태훈

201724519 이강윤

지도교수 **백윤주**

목 차

1. 서론

1.1. 연구 배경

임베디드 시스템과 AI 기술의 상호작용은 최근 기술 발전의 한 축으로 주목받고 있다. AI 기술의 발전으로 객체 탐지와 같은 고급 기능이 이제는 작고 효율적인 임베디드 시스템에서도 실현 가능해졌다.

이러한 발전은 부품의 판정과 같은 분야에서도 접목 시킬 수 있다. 예를 들어, 부품의 분류 과정에서 AI 기반의 객체 탐지 기술을 활용하면, 이전에 수작업으로 이루어지던 작업을 자동화하고 높은 정확도로 원하는 결과를 얻을 수 있다.

최근에는 스마트폰이 지속적으로 개발됨에 따라서 내부를 구성하는 부품 역시 지속적으로 변화되고 있다. 이로인해 기존의 딥러닝으로는 새로운 센서나 부품이 추가 될 경우 이를 판독하기 위해 다량의 데이터 셋이 추가적으로 필요하다.

이러한 문제를 해결하기 위해 여러 기법을 동원하여 한 장 혹은 몇 장의 극소량의 이미지 데이터 만으로도 새로 추가되는 부품을 판독해냄으로 기존의 딥러닝 방식의 개선해보고자 이번 졸업 과제로 선정하게 되었다.

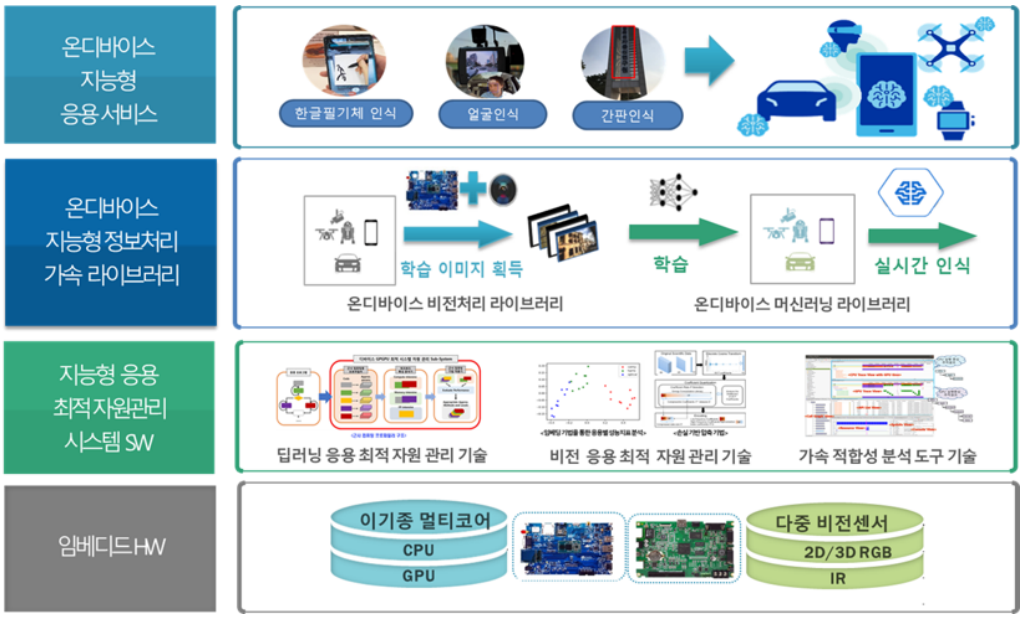

그림 1 임베디드 기반의 딥러닝 사용 분야

1.2. 기존 문제점

기존의 CNN 방식은 각 부품별로 최소 200장 이상의 이미지 데이터를 필요로 한다. 따라서 새로운 클라스가 추가할 경우, 이 추가하려는 클라스 역시 200장 이상의 데이터를 학습해야한다. 또한 데이터 수집 뿐만 아니라 라벨링과 같은 데이터 가공 역시 필요하며, 이러한 과정은 막대한 시간과 비용이 들게 된다. 그리고 모델이 복잡하며 많은 매개 변수를 가지고 있기 때문에 충분한 양의 훈련 데이터가 필요하며, 데이터가 부족한 경우 과적합 문제가 발생할 수 있다.

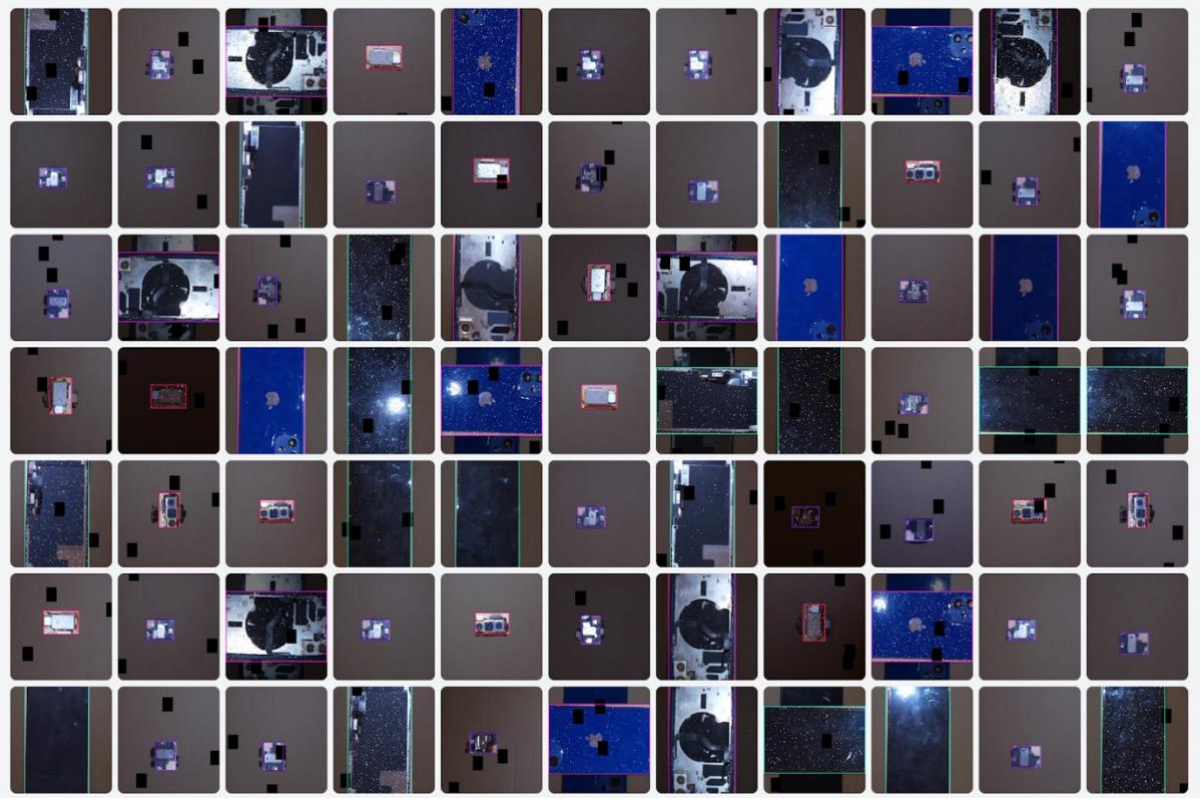

그림 2 CNN 모델 제작을 위한 전처리 예시

1.3. 연구 목표

기존의 CNN방식은 학습데이터를 수집하는데에 있어 대량의 데이터가 필요하다. 이번 과제에서는 적은 양의 학습 데이터만으로 기존의 방식과 정확도가 유사한 성능을 내는 스마트폰 부품을 판별하는 모델을 개발하여 라즈베리파이에서 작동시키는 것이 일차적인 목표이다.

이차 목표로 제작한 모델을 경량화 및 최적화하여, 임베디드 기기에서 구동이 가능케 한다. 추후 이 모델을 이용하여 ESP32-CAM 이나 STM32F746NG와 같은 초소형 임베디드 기기에서도 사용 가능케 하는 것을 염두하여 모델의 제작을 목표로 한다.

2. 연구 배경

2.1. Siamese Neural Network

기존의 객체 탐지 모델을 이용할 경우, 처음에 언급 했듯이, 많은 양의 데이터가 필요하고, 이를 가공해야한다. 이 문제점을 해결하기 위해, 새로운 모델을 구축해야 할 필요성을 느꼈고 SNN(Siamese Neural Network)에 대해 조사해보았다.

SNN이란, 이름에서 언급된 것처럼 Siamese Twin (샴 쌍둥이)에 나오는 샴을 의미한다. 이름에서 알 수 있듯이, 신체의 일부를 공유한다는 점이 네트워크 구조와 닮아서 붙여진 이름이다.

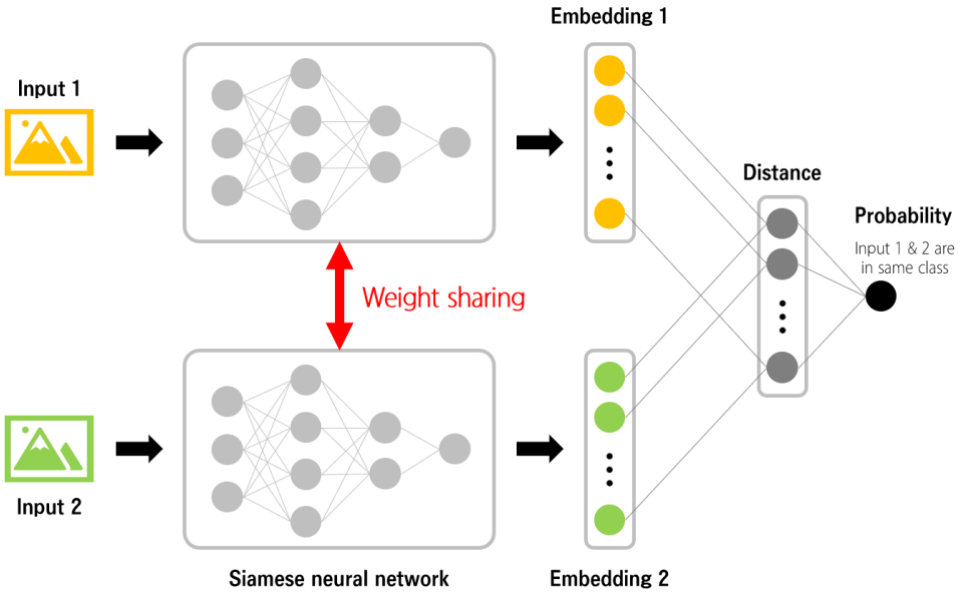

그림 3 SNN의 모델 구조

이러한 모델 구조는 weight를 공유함으로 하나의 네트워크라고 봐도 무방하다. 이 특성을 이용하면 두 input 이미지의 임베딩 과정을 거쳐, 유사도를 계산할 수 있다. 즉 두 이미지가 같은지 다른지를 판단할 수 있다는 점에서, One-Shot Learning이라는 개념이 제시되었다.

2.2. One-Shot Learning

One-Shot Learning이란 이름에서 유추할 수 있듯이 한장의 데이터 만으로 해당 데이터가 어떤 것인지 판단할 수 있도록 학습 시키는 방식이다. 기본적으로 딥러닝은 엄청난 양의 데이터를 학습시켜 모델이 해당 오브젝트가 무엇인지를 판단한다. 하지만 사람의 경우, 기존에 본 적 없는 처음보는 오브젝트라도, 해당 오브젝트가 무엇인지 판단해낼 수 있다. 다음 그림4의 위 이미지를 "총"이라는 클라스로 모델 학습을 시켰다고 가정한다.

그림4 K2 소총

기존의 CNN 모델망에서 그림 4의 아랫쪽 이미지를 어떤 객체인지 분류시켰을 경우 충분한 데이터로 학습을 시켰다면 이를 "총"으로 판별해낼 것이다. 그런데 만약 다음 그림 5와 같은 이미지가 들어왔을 경우에는 어떨까?

그림 5 나폴레옹 권총

총열 길이도, 모양도, 색도 다르기에 CNN 모델이 이를 총으로 판별하지 못 한다. 하지만 사람은 이 사진을 보고 쉽게 총이라고 판별해낼 수 있다. 이렇게 사람이 적은 데이터로도 처음 보는 사물을 유추할 수 있는 점을 모방하여, 한장의 데이터로도 이 사물을 총이라고 인식시키는 학습 방식이다. 좀 더 나아가서, 한장이 아닌 아예 학습 데이터가 없이도 사물을 유추해내도록 학습 시키면 Zero-Shot Learning이 되고, 여러장의 데이터를 이용하면 Few-Shot Learning이 된다.

부품은 대게 같은 기능을 하면 유사하게 생겼다. 만약 완전히 같은 부품이라면 정형화되어 있기에 Few-Shot Learning을 이용하면, 적은 양의 데이터로도 기존에 학습된 부품은 정확히 판별 가능하고, 기존에 학습되지 않은 부품의 경우 어떤 기능을 하는 부품인지 어느정도 유추가 가능할 것이라고 판단했다. 적은 데이터를 사용한다는 점에서 모델 크기 역시 기존의 CNN보다는 작을 것이기에 이는 제한된 임베디드 환경에서 강점을 가질 것으로 보인다.

3. 연구 내용

3.1. 데이터 수집

3.1.1. 학습 데이터 수집

먼저 부품을 촬영할 촬영대를 제작하는 것부터 제작하였다. 처음 계획은 12cm 정도의 높이에서 촬영한 이미지를 이용하려고 하였으나 촬영된 이미지가 판독을 진행하기에는 적절치 못하여 높이를 바꾸게 되었다. 여러 높이를 측정한 결과 가장 적절한 높이가 45cm임을 확인하여 그림 6과 같이 임시 제작대를 제작 후 데이터 수집을 진행하였다.

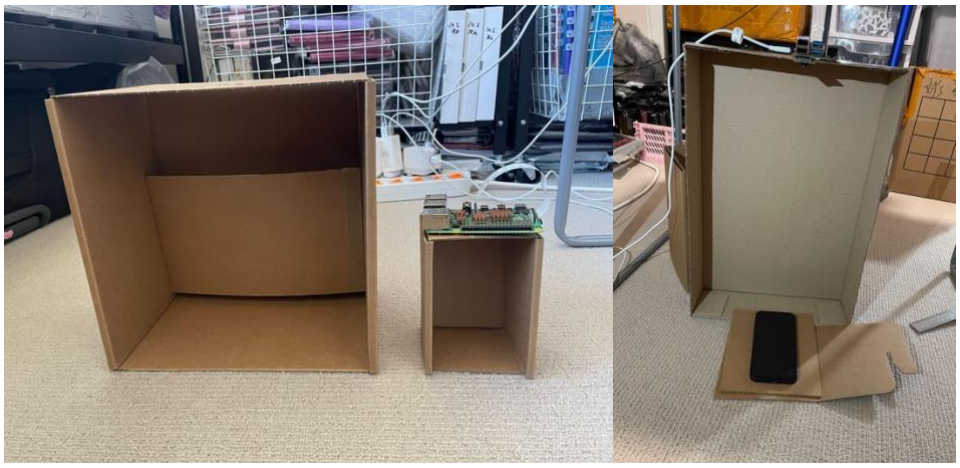

그림 6 임시 촬영대

해당 촬영대로 촬영한 사진은 그림 7과 같다. 너무 큰 부품은 높은 위치에서 촬영하는 것이 유리하지만, 작은 부품의 경우 작게 촬영되는 점에서 큰 부품을 제외할지 작은 부품을 이미지 처리로 확대할지를 생각해봐야 하였다.

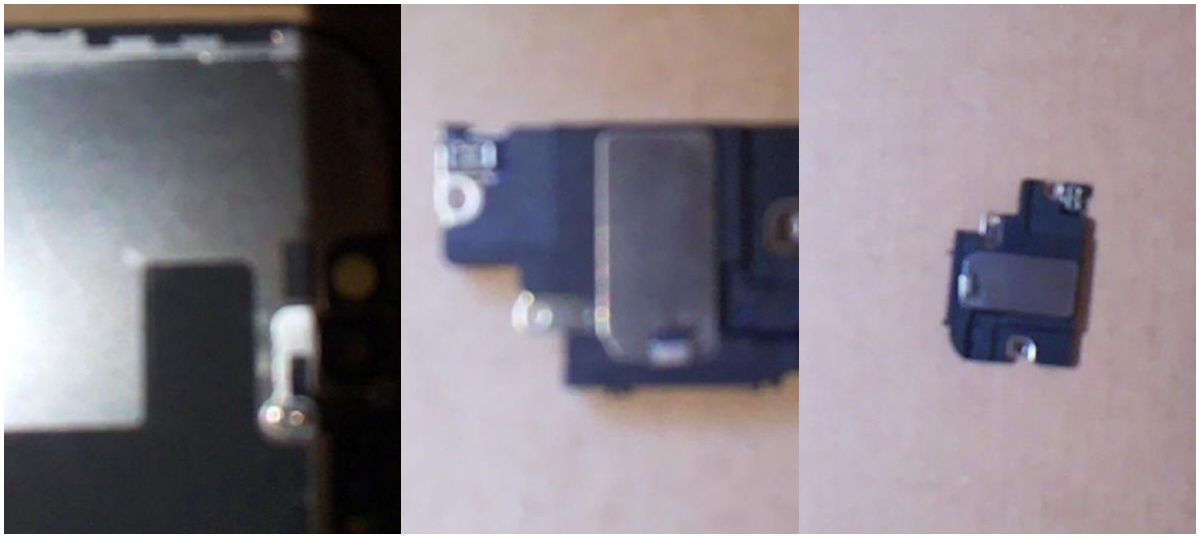

그림 7 12cm, 24cm , 45cm에서 촬영한 이미지 샘플

촬영대 제작 후, 촬영을 진행하기 위해 Pi Camera2에 맞는 코드를 작성하였고, 이를 이용하여 촬영을 진행하였다. 실제로 사용한 코드는 기술하지 않고 부가 자료로 제출할 예정이다.

3.2. 모델 제작

3.2.1. CNN 모델의 정확도 측정

처음 계획은 기존의 모델은 CNN 방식으로 제작후 추가되는 부품은 SNN 방식으로 판별을 진행하는 것으로 계획하고 진행하였다. 이를 위해 한 부품당 100장 정도의 이미지 데이터를 촬영 후 데이터 증강만을 이용하여 전처리 후 모델을 제작하였다. 당시 이용한 모델은 "ssd_efficientdet_d0_512x512_coco17_tpu-8"으로, 모델 제작의 시간을 단축하기 위해 Google Colab 환경에서 학습을 진행하였다. 그림 8에서 알 수 있듯이, 2만 스텝 정도의 학습을 진행하였고, loss는 0.058 정도로 측정되었다.

```
I0808 11:40:34.240252 132344664212096 model_lib_v2.py:705] Step 21000 per-step time 0.245s
INFO:tensorflow:{'Loss/classification_loss': 0.05873496,
 'Loss/localization_loss': 0.013632012,
 'Loss/regularization_loss': 0.044404063,
 'Loss/total_loss': 0.11677104,
 'learning_rate': 0.079239115}
I0808 11:40:34.240655 132344664212096 model_lib_v2.py:708] {'Loss/classification_loss': 0.05873496,
 'Loss/localization_loss': 0.013632012,
 'Loss/regularization_loss': 0.044404063,
 'Loss/total_loss': 0.11677104,
 'learning_rate': 0.079239115}
Traceback (most recent call last):
  File "/content/drive/MyDrive/object_detection/model_main_tf2.py", line 114, in <module>
    tf.compat.v1.app.run()
  File "/usr/local/lib/python3.10/dist-packages/tensorflow/python/platform/app.py", line 36, in run
    _run(main=main, argv=argv, flags_parser=_parse_flags_tolerate_undef)
  File "/usr/local/lib/python3.10/dist-packages/absl/app.py", line 308, in run
    _run_main(main, args)
  File "/usr/local/lib/python3.10/dist-packages/absl/app.py", line 254, in _run_main
    sys.exit(main(argv))
  File "/content/drive/MyDrive/object_detection/model_main_tf2.py", line 105, in main
    model_lib_v2.train_loop(
```

그림 8 CNN모델 학습 진행

모델을 이용하여 간단한 판별 테스트를 진행하였을 때, 그림 9의 좌측과 같이 적은 데이터라도 만족할만한 결과를 보여주었다. 또한 원샷 러닝과의 정확도 비교를 위해 1장만 학습시킨 이어스피커 파트를 판별 진행하였을 때는, 그림 9의 우측과 같이 전혀 판독해 내지 못하는 모습을 보였다.

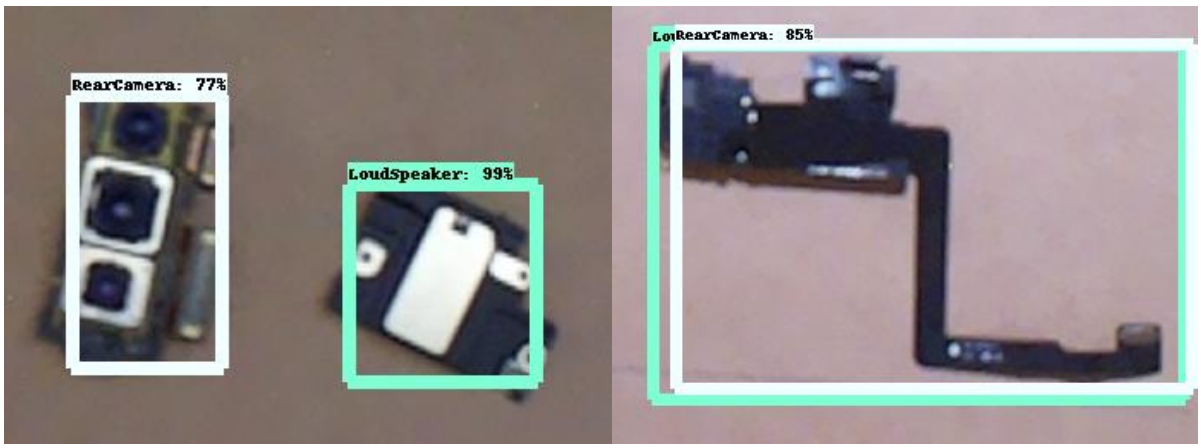

그림 9 CNN 테스트

테스트 결과, 양자화 되지 않은 모델의 크기는 약 13MB였고, 양자화를 간단히 진행하였을 때는 11MB의 크기를 가졌다. 이는 일반 임베디드 환경에서는 사용 가능하지만, 추후 TinyML을 하기에는 너무나도 큰 크기였기에, 모델 최적화 및 경량화가 필요할 것으로 판단하였다.

3.2.2. SNN 모델 테스트

SNN 모델은 잘 알려지지 않은 모델이기에, 먼저 어느 정도의 정확도를 가지는 지 파악하는 것부터 시작하였다. 기본 개념으로는 두 이미지의 유사도를 비교한다는 점을 이용하여 학습되지 않은 상태에서 두 이미지의 유사도 비교는 어떤 결과가 나오는 지 테스트 해보았다. 결과는 그림 10, 그림 11과 같았다.

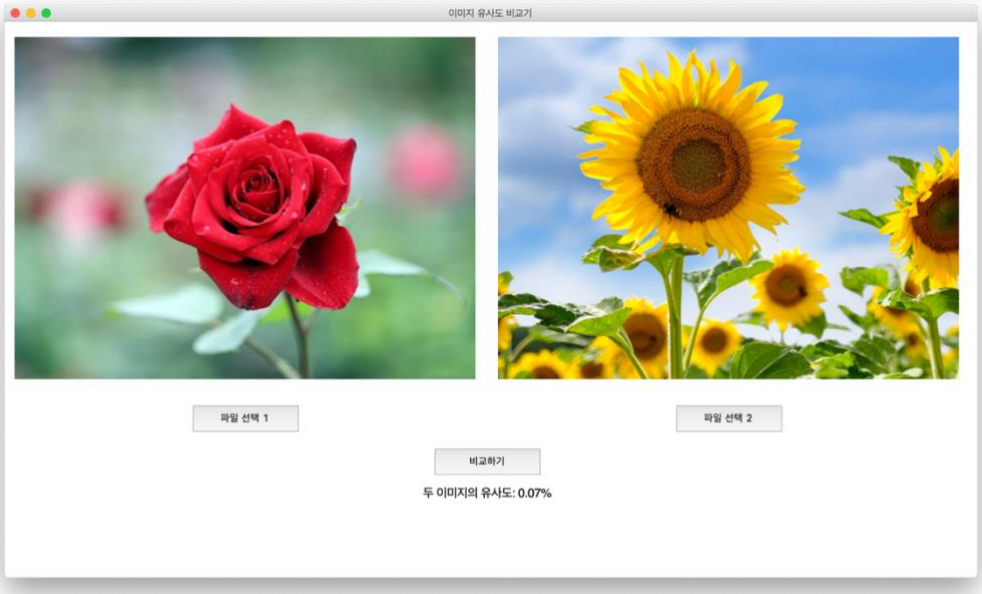

그림 10 SNN 테스트 (정상)

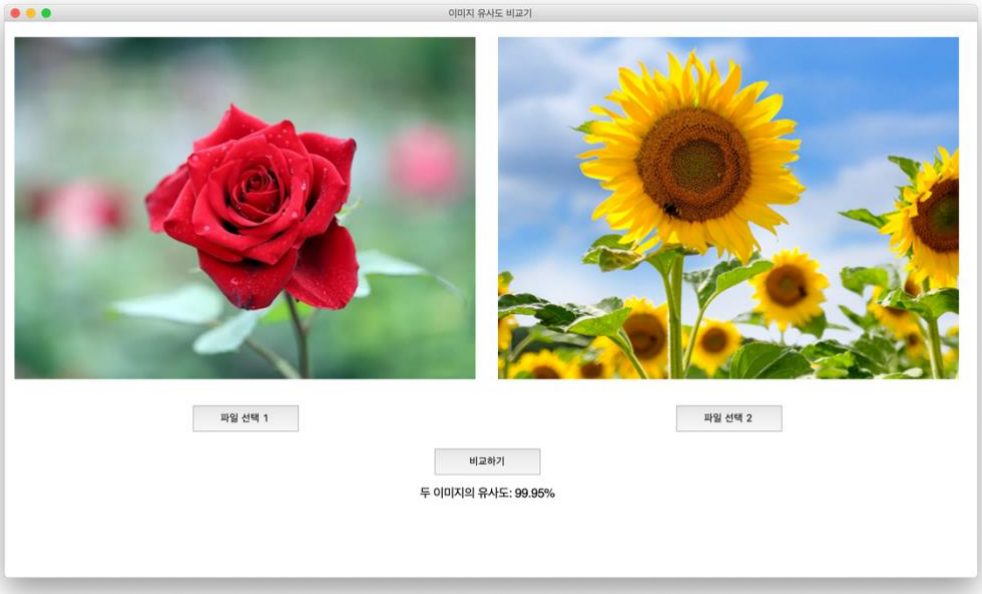

그림 11 SNN 테스트 (비정상)

학습된 모델 없이, 단순히 두 이미지를 비교할 경우, 올바른 유사도가 때로는 맞게 나오지만, 때로는 올바르지 못한 값이 나왔다. 이를 결과를 바탕으로, 모델을 만들지 않고 판별을 진행할 때, 동시에 클라스를 비교할 경우 올바르지 못한 값이 나올 것으로 판단하였고, 이는 미리 SNN 모델로 Few-Shot Learing을 진행해야 한다는 점을 시사하였다.

따라서 SNN 모델을 이용하여 판별을 진행시 이용할 사전 학습 모델을 제작을 위해 코드 제작이 필요하였다. 이렇게 제작된 모델을 CNN 모델과 동시에 이용하는 방법을 연구해보았으나, 좋은 결과를 얻지 못하였다. 먼저 두 개의 모델을 이용하면 필연적으로 모델 크기가 커지는 문제점이 있었고, 또한 이 모델을 잘 접목 시킬 방법을 찾기가 어려웠다. 그래서 모델을 하나만 이용한다고 가정하면, 차라리 SNN 모델만을 이용하는 것이 하드웨어 자원을 이용하는데 있어 유리하다고 판단하여 모델을 수정하게 되었다.

3.3. SNN 모델 제작

3.3.1. 모델 제작 및 테스트

앞에서 테스트한 SNN 모델의 기본 구성은 그림 12와 같다. 모델 구성은 동일하지만, SNN 테스트와는 달리 유사도를 저장해야 하는 점에서 코드를 변형하여, 모델 학습 코드를 제작하였다. 처음 코드로 모델을 제작할 때, 약 3시간 정도의 시간이 걸렸기에 정상 작동하는 코드로 판단하였으나, 추후 다른 머신에서 학습 진행시, 코드의 문제점을 발견하였다.

그림 12에서 확인할 수 있듯이 파이썬이 약 30GB의 메모리를 사용하였다. 이는 한번에 모든 이미지 쌍을 생성하여 메모리에 올려서 판별을 진행하였기에, 메모리 사용량이 과하여 가상 메모리로 스왑하여 사용하는 상황이었다. 이 때문에 한번에 메모리에 모든 이미지 쌍을 올리는 것이 아닌, 학습 때마다 제네레이터를 이용하여 작은 단위의 이미지 쌍을 형성하여 사용이 끝난 데이터는 반환하는 방식으로 코드 수정이 필요하였다.

문제는 이렇게 제네레이터를 이용하는 방식이 한 번에 모든 쌍을 메모리에 탑재하여 학습시킨 모델과 정확도에 차이가 없는지 보장할 수 없었다. 이를 확인하기 위해 일단 같은 양의 데이터를 준비하여, 두 방식으로 모델 학습을 진행 후 결과가 어느정도 차이가 나는지를 확인하는 방법으로 제네레이터를 이용한 방식의 정확도를 확인하였다.

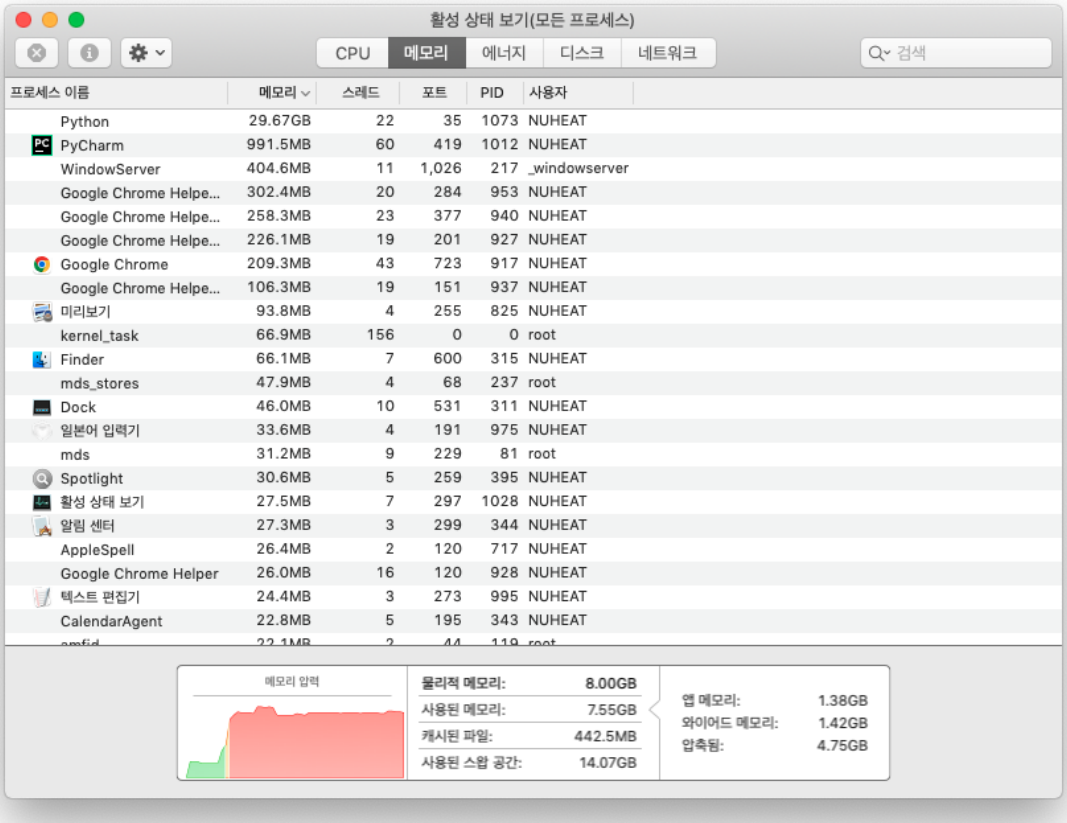

그림 12 메모리 사용량

표 1과 같이 동일한 학습 데이터로 학습을 진행할 경우 정확도는 거의 유사하였고, epoch를 조절하였을 때 더 많은 epoch를 진행한다고해서 정확도가 더 좋게 나오지는 않았다. 이는 과적합 문제를 일으키는 것으로 판단하였고, 모델 제작시 높은 정확도의 모델을 제작하려면 적당한 batch 크기와 epoch를 주어야 함을 확인하였다. 이를 고려하여 실제 모델 제작시 파라미터를 바꾸어가며 테스트 해야 할 것이다.

	Directly (4PCS)	Generator (4PCS)	Generator (40PCS) - epoch 10	Generator (40PCS) - epoch 100
Class Accuracy for Housing	77.50%	82.50%	97.50%	97.50%
Class Accuracy for Dock	45.00%	42.50%	85.00%	0.00%
Class Accuracy for Vibrator	5.00%	10.00%	0.00%	0.00%
Overall Accuracy	42.50%	45.00%	60.83%	32.50%

표 1 정확도 테스트 결과

3.3.2. 모델 최적화

모델 최적화를 위한 방법으로는 Network Quantisation, Network Compiling, Efficient Architecture Design, Network Pruning, Knowledge Distillation, Matrix / Tensor Decomposition 등의 방법이 있다. 그 중 가장 대중적인 방법으로 Network Quantisation, 모델 양자화가 있다.

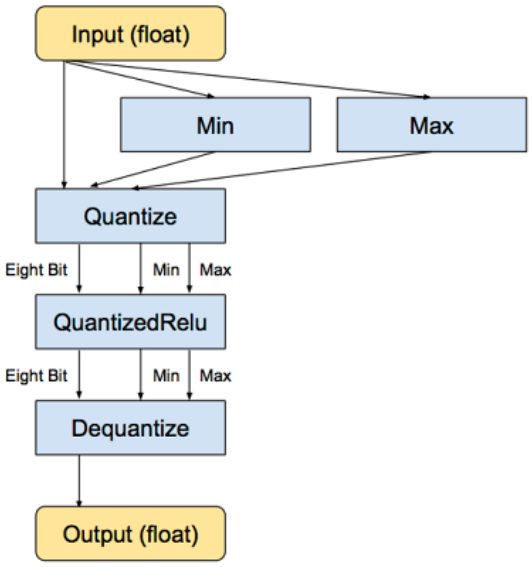

그림 13 양자화를 통한 크기 축소 모델

양자화는 weight값의 사용 비트를 줄여서 모델 크기를 줄인다. 보통은 float32 데이터 타입으로 네트워크 연산 과정이 표현 되는데, 이를 그보다 더 작은 크기의 데이터 타입으로 변환하여 연산 후, 다시 float32 형태로 변환한다. 이러한 과정을 거치면 보통 1/4 정도로 모델의 크기가 감소한다.

양자화에 사용 가능한 데이터 타입이 uint8, int8, float16 등의 방식이 있는데, 데이터 타입이 모델 크기에 영향을 미친다. 정수 양자화를 진행할 경우 더 적은 모델사이즈를 제공해주지만, float16 방식에 비해 정확도가 떨어지고, 대표 데이터셋을 제작해주어야 하는 단점이 있다. 이를 확인하여 모델 사이즈가 얼마나 줄어드는 지 테스트 해본 결과 그림 14에서 알수 있듯이 float16 방식의 경우 1/8 정도로 크기가 줄어들었다., int8의 경우 제대로 양자화가 되지 않았기에 확인을 할 수 없었지만, float16 보다 더 줄어들 것으로 추측된다.

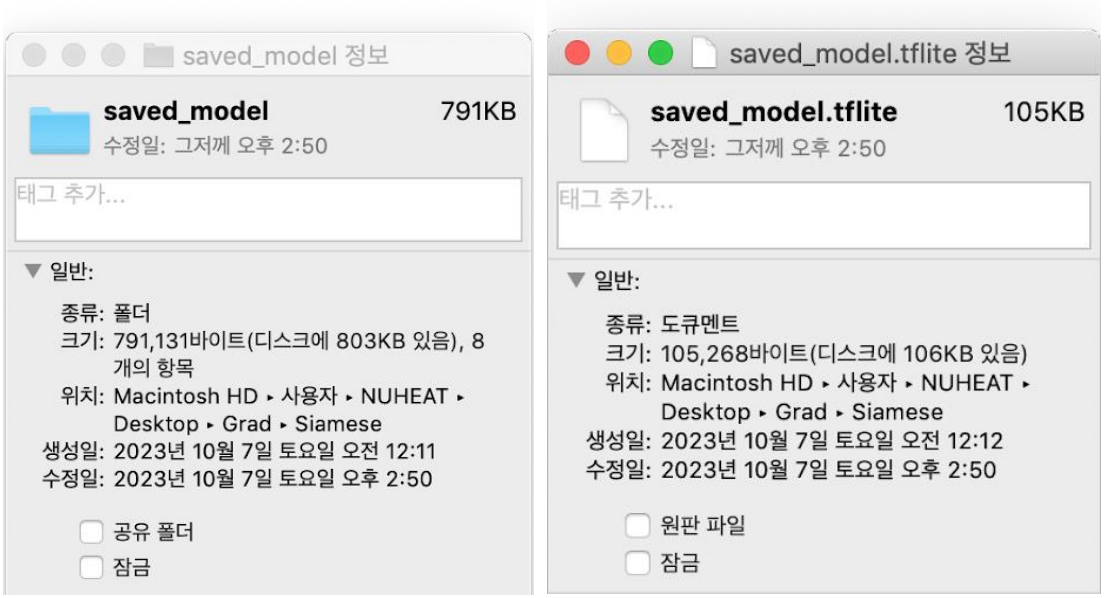

그림 14 양자화 진행 결과

크기의 최적화가 진행되었으면 다음으로 모델의 메모리 사용량을 최적화 한다. 데스크톱 환경과는 달리, MCU 모드에 업로드해서 어느 정도의 메모리를 사용하는지 매번 측정하는 것은 현실적으로 어렵기에, STM에서 제공되는 STM CUBE MX의 CUBE AI를 이용하여 메모리 사용량을 체크해보는 방법을 이용하였다.

기존에 처음 만든 모델의 경우 320*240 크기의 RGB 3채널 이미지를 이용하여 tflite 모델을 제작하였다. 그림 15에서 확인한 압축된 모델의 크기는 약 1.19MB로 크기 자체는 MCU에서 사용 가능한 크기였지만, 메모리 사용량이 12.11MB로 측정되었다. 추가적으로 이미지 전처리나 카메라 모듈 작동시 추가 메모리가 필요할 것으로 보이기에, 입력 이미지 크기를 줄여서 메모리 사용량을 줄이는 방법을 시도하였다.

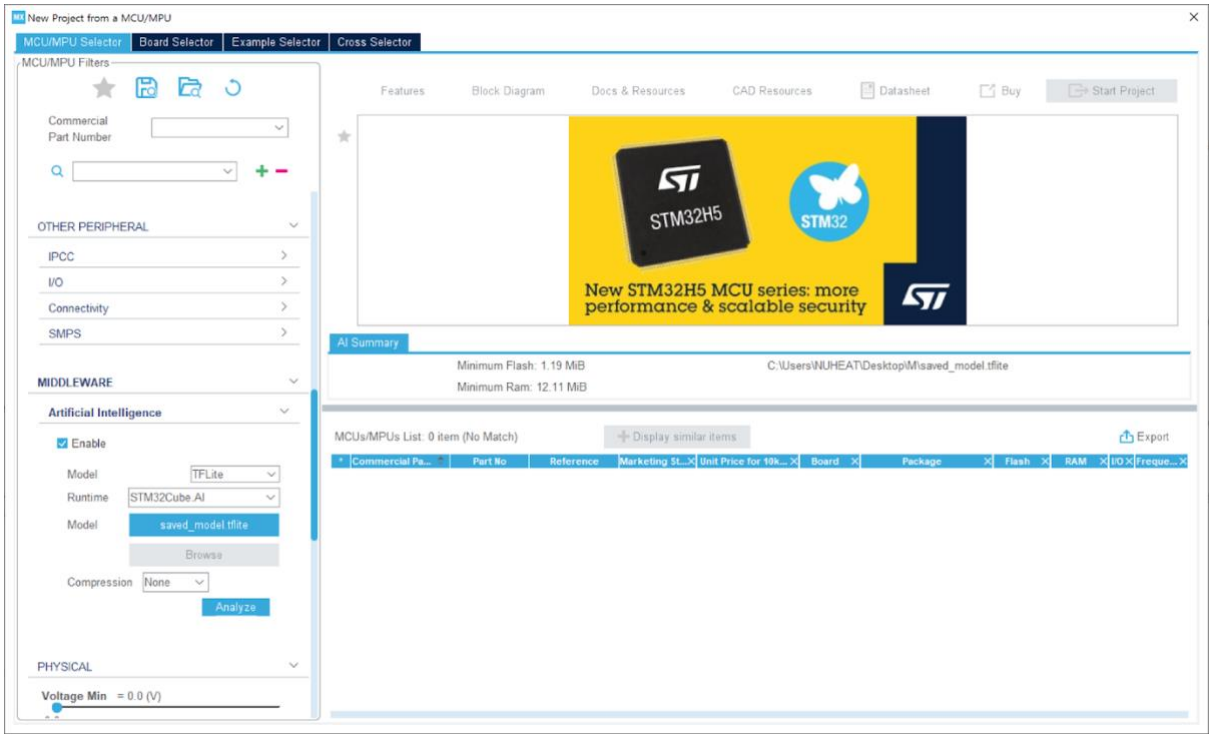

그림 15 컬러 모델 리소스 사용량

학습 이미지와 입력 이미지 크기를 96*96에 흑백 모델로 교체하여 이 모델을 바탕으로 다시 리소스 사용량을 측정해본 결과 그림 16과 같이, 모델 크기는 210KB에 메모리 사용량은 1.27MB가 나오는 것을 확인하였다. IDE의 문제로 양자화 된 모델은 사용할 수 없었기에, 양자화 되지 않은 모델을 기준으로 메모리 사용량이 측정되었다. 이는 양자화 된 모델을 사용시 더 적은 메모리와 크기를 가질 것으로 판단된다. 따라서 TinyML을 가능케 할 만큼의 메모리 최적화도 충분히 가능할 것으로 보인다.

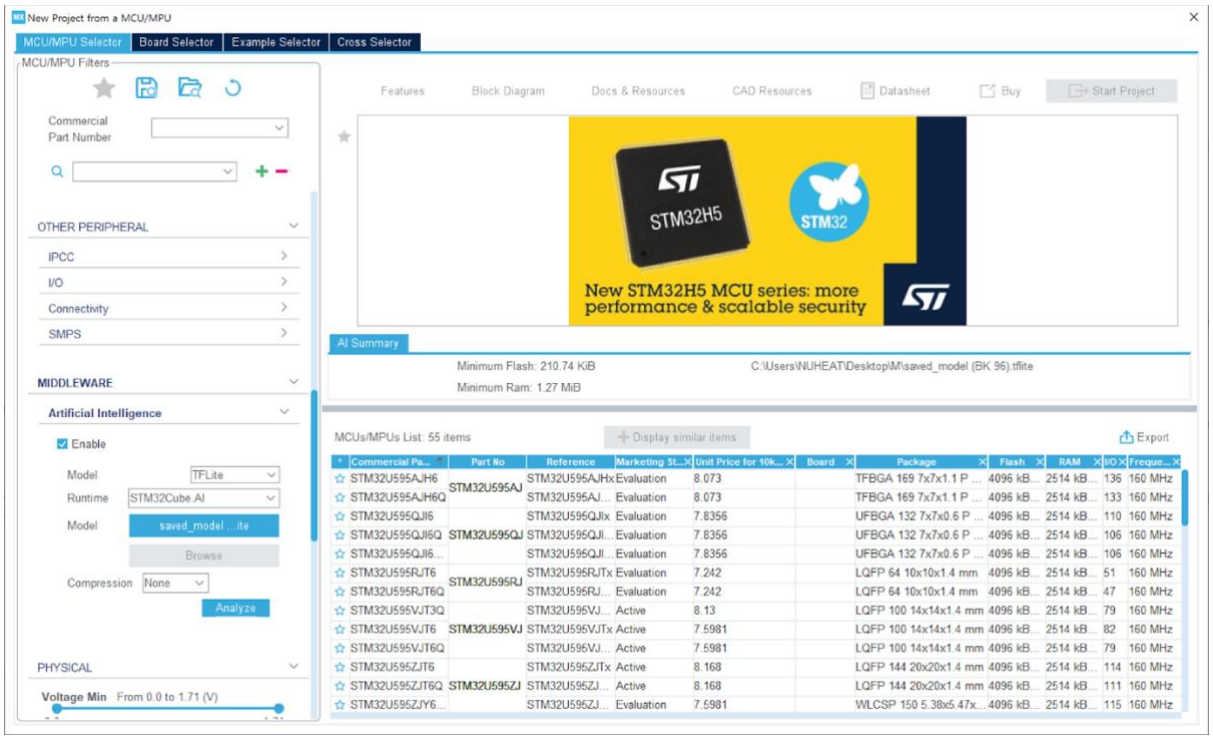

그림 16 흑백 모델 리소스 사용량

4. 연구 결과 분석 및 평가

샴 네트워크에 대한 연구가 그렇게 많지 않아서 테스트 후 결과를 확인하기가 쉽지 않다. 여러 방식으로 모델을 제작하여도 어느 경우에는 올바른 결과가 나오지만 어떤 경우에는 결과가 제대로 나오지 않는 어려움이 있었다. 이를 확인하기 위해, 매번 모델을 새로 만들어야 했다.

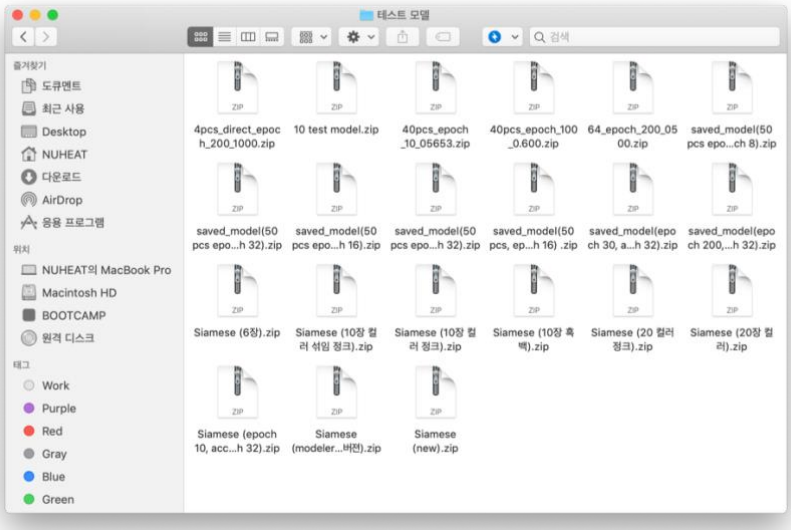

그림 17 테스트 진행한 모델

자적화를 진행하여 소형 MCU에서 구동 가능한 사이즈로 만드는 것은 성공하였으나, 이 모델을 사용하여 MCU에서 테스트를 진행시, 제대로 된 결과가 나오지 않는 문제가 있어서 이를 해결하기위해 여러 모델을 제작해보았으나, 원인을 찾을 수 없었다. 따라서 기존에 신뢰성이 보장된 모델을 최종적으로 이용하는 것으로 결정하였다.

6가지 부품을 전후로 각각 10장씩, 총 20장의 데이터만으로도 해당 부품이 어떤 부품인지를 판별하는 것이 가능하였다. CNN 방식으로 만든 모델의 경우, 그림 17과 같이, 스크린에 해당하는 부품을 하우징 파트로 오판하는 결과를 보였으나, SNN 방식으로 만든 모델로 판정을 진행할 경우, 이를 정확히 스크린으로 판별해내었다. CNN 방식이 각각 200장 정도의 이미지에 데이터 증강도 이용했다는 점에서, 더 적은 데이터로 좋은 결과를 낼 수 있음을 보이는 경우였다.

그림 18 하우징으로 오판한 예

다른 부품들에 대해서도 테스트를 진행해본 결과, 모양이 명확하게 다른 부품과는 차이를 보이는 경우에는 높은 정확도를 보였다. 그림 18과 같이 여백이 많은 경우에도 해당 부품이 스피커임을 정확하게 파악하는 점에서, Few-Shot Learning의 강점을 실감할 수 있는 부분이였다.

SNN의 모델이 이미지의 유사성을 판별하고 어느 클라스에 유사한지를 알려줄 수 있는 특징은 장점이지만, 항상 장점을 가진 것은 아니었다. 일례로 모양이 유사한 객체일 경우 SNN 모델을 사용할 경우, 해당 객체를 정확히 판별하지 못하고, 다른 클라스로 유추하는 문제점이 있었다. 그림 19에서 좌측은 독커넥터이고 우측은 이어스피커이다. 두 부품은 배치되는 방향에 따라서, "ㄱ" 형태를 가지게 될 경우, SNN 모델에서는 이 두 부품을 오판 하는 경우가 많았다. 학습 데이터를 늘리고, 학습 횟수를 조절해보았지만, 모델 특성상 유사한 경우는 구분하기 힘들다는 결론을 얻었다.

즉 유사할 경우, 처음보는 객체라도 해당 객체가 어떤 객체인지 유추할 수 있다는 강점과 학습된 객체라도 모양이 유사하다면 구분하기 쉽지 않다는 장단점을 가진 모델이다. CNN 모델과 함께 사용할 수 있다면, 더 정확도 높은 객체 탐지가 가능할 것으로 보인다.

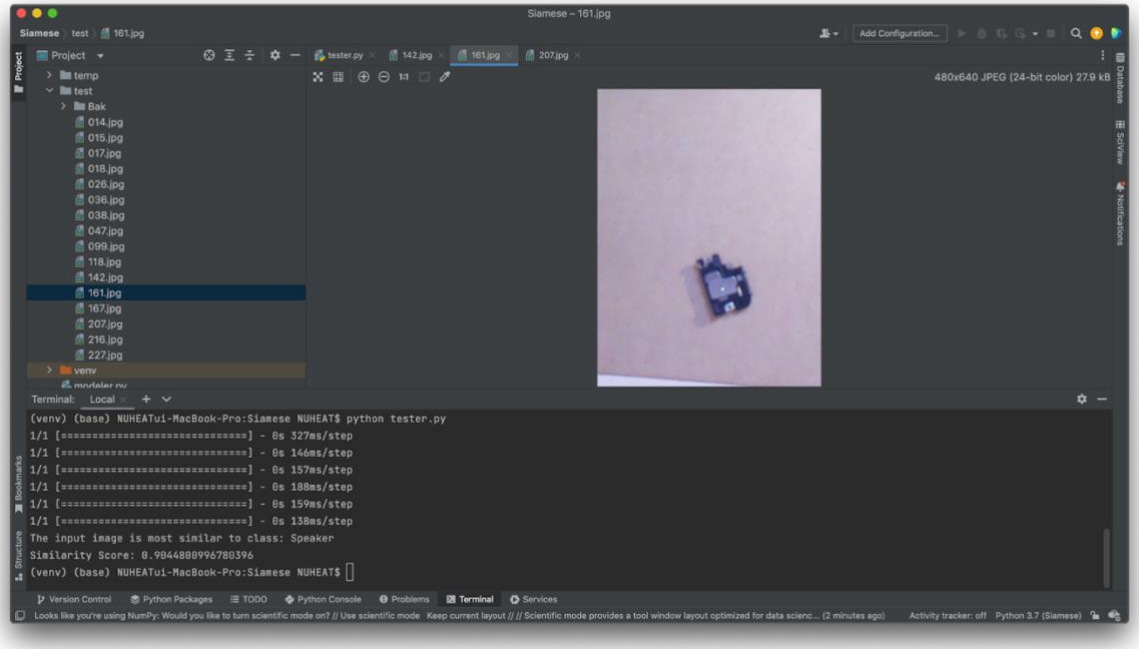

그림 19 스피커 판정 예시

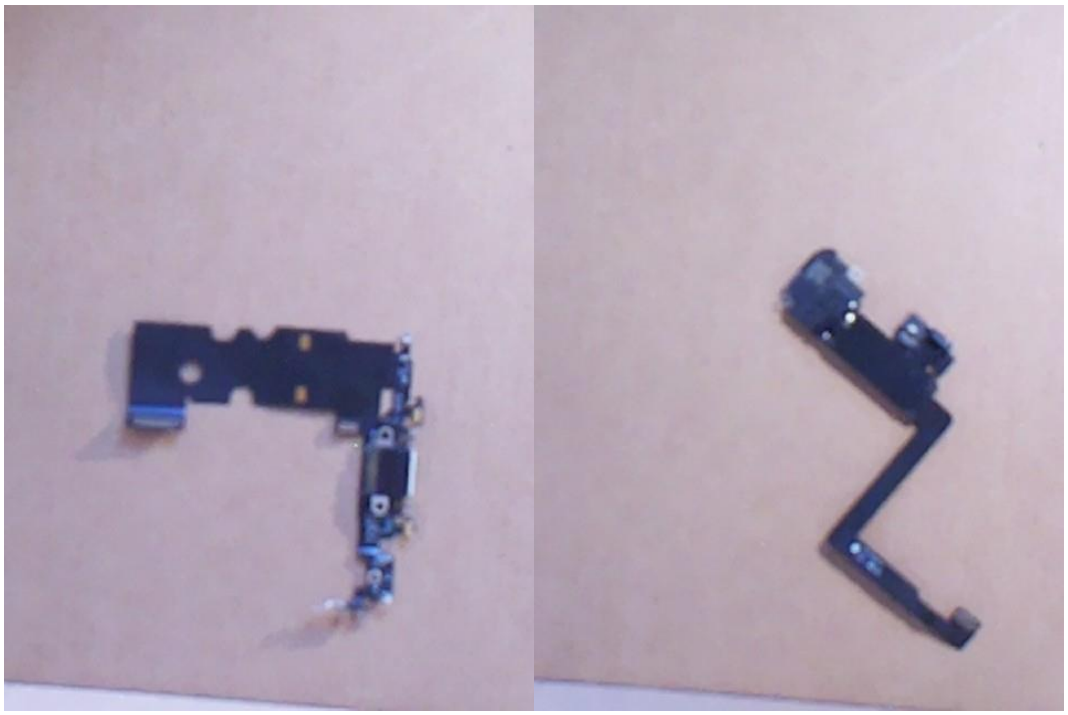

그림 20 독커넥터와 이어스피커

다음으로 라즈베리파이 4에서 해당 모델을 구동하여, 데스크톱 환경과 같은 결과를 도출해내는지 테스트해보았다. 데스크톱 환경과는 달리, 소형 MCU는 속도면에서 더 느린 결과를 보였다. 하지만 판정 결과 자체는 데스크톱 환경과 동일한 결과를 내었기에 부품 판별이라는 목적은 원만하게 달성하였다.

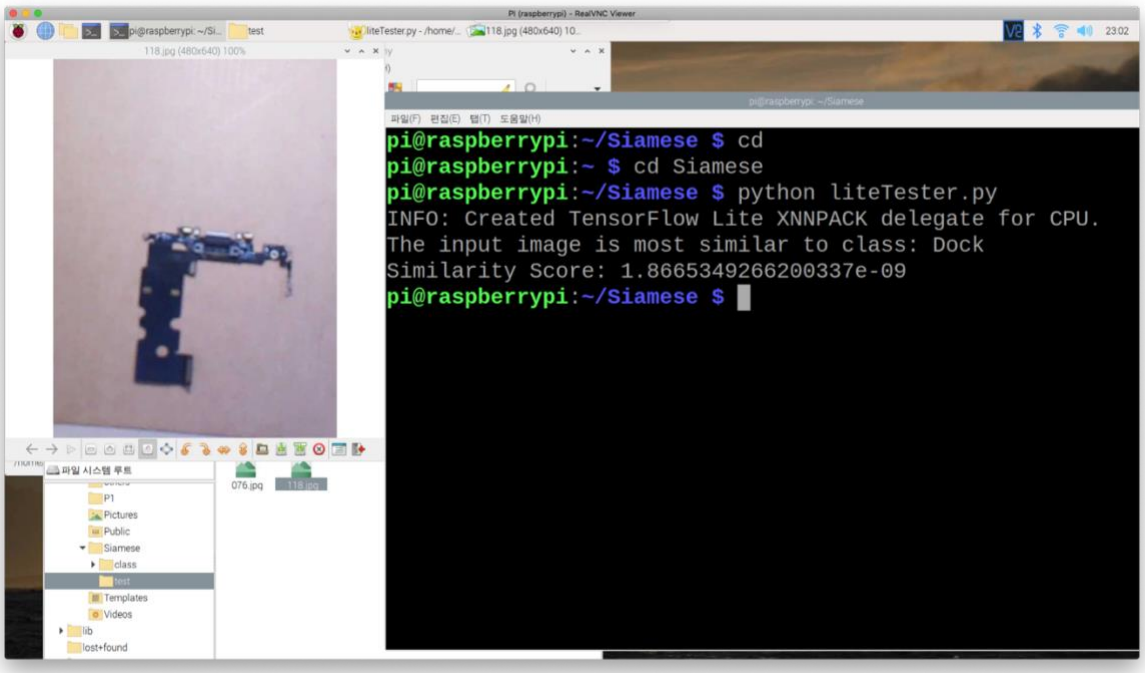

그림 21 라즈베리파이에서 진행한 모습

추가적으로 이러한 판정을 진행하는 부분은 매번 커맨드를 입력하여 구해내는 않고 간단하게 클릭하는 형태로 GUI를 구현하였다. 그림 21과 같이, 촬영한 이미지 혹은 이미 촬영된 이미지만 선택하고 판정 진행을 누르면, 하단에 해당 부품이 어떤 부품이며 어떠한 기능을 하는지 표시되도록 구현되었다. 데스크톱 환경에서의 UI와 라즈베리파이에서의 UI가 조금 상이한 부분이 있어서, 사용 환경에 맞도록 수정 및 최적화가 추가적으로 진행될 것이다.

Siamese Neural Network는 인간이 사물을 구별하는 특성을 모방하여 적은 데이터로도 학습이 가능하다는 장점이 있는 매력적인 네트워크이다. 현재에도 많은 연구가 이루어지고 있지만 아직은 데이터가 적었기에 직접 모델 설계부터 학습을 진행하여야 했기에, 만족스러운 결과를 얻지 못하였다. 모델 테스트 및 평가 역시 CNN과 같이 방대한 자료가 있는 모델과 달리, 테스트 하는 코드도 직접 구현해야했기에, 이 코드의 신뢰성 확보에 많은 시간이 소요되었다.

그림 22 판정 프로그램

5. 결론 및 향후 연구 방향

Siamese Neural Network를 이용한 Few-Shot Learning으로 객체의 판별이 가능하다는 것을 실제로 증명해보일 수 있었다. 이 과정에서 많은 연구 자료를 찾아보려고 노력하였으나, 아직은 더 많은 연구가 필요한 분야로 판단된다. 이러한 점으로 인해, 정확히 어떤 방식으로 모델을 구축하고 개선해 나갈 수 있는지 직접 시행착오를 겪으면서 찾아야 하였고, 찾아낸 개선점이 정말로 올바르게 개선되었는지 확신할 수 없었기에 더 나은 결과물을 내기가 어려웠다.

이번 연구는 기존 CNN방식이 아닌 SNN방식이라는 새로운 모델을 활용하여 더 적은 데이터로 객체 분류가 가능하게 하는 것이 주된 목표였다. 추후에는 더 작고 경량화하여 TinyML을 가능케 하는 것을 염두하고 모델 개발을 하였기에, 모델 경량화와 소형화를 중점적으로 진행하였다. SNN 방식으로 TinyML을 구현한 사례가 현재는 존재하지 않아 자료를 찾는데 어려움을 겪었다. 일단 현재 제작한 모델을 ESP32-CAM에 탑재하는 것까지만 진행해보고 촬영 및 판별까지 진행하는 것은 향후의 연구 방향이 될 것이다.

그림 23 ESP32-CAM에 모델 업로드 완료

6. 구성원 별 역할 및 개발 일정

6월					7월					8월					9월					10월		
1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3
TinyML 관련 기술 연구																						
모델 경량화 및 데이터 증강 학습																						
					데이터 수집					테스트 모델 개발												
												실 사용 모델 개발										
															모델 최적화							
																		최종 테스트				
																				최종 보고서		

이름	역할
윤태훈	테스트 모델 제작 및 데이터 수집 데이터 전처리 및 모델 학습 라즈베리 파이 코드 작성
이강윤	모델 테스트 및 최적화 보고서 작성 및 발표 자료 준비 UI 제작
공통	모델 연구 및 자료 수집 발표 및 시연 준비

7. 참고 문헌

[1] GreGort Koch, Richard Zemel, and Ruslan Salakhutdinov, "*Siamese Neural Networks for One-shot Image Recognition*"

[2] S. Chopra, R. Hadsell, Y. LeCun, "*Learning a similarity metric discriminatively, with application to face verification*"

[3] Github, Mohamed Elsayed : https://github.com/mohamed-elsayed-mohamed

[4] Github, Tyami : https://tyami.github.io/deep%20learning/Siamese-neural-networks/

[5] Facial Similarity with Siamese Networks in PyTorch : https://hackernoon.com/facial-similarity-with-siamese-networks-in-pytorch-9642aa9db2f7