

# Hyper parameter, Fine tuning을 통한 CNN 구조의 차량 분류 모델 연구

Juan Park, Jeongkyu Lee, Georyang Park, Jiyoong Bang, Gunhee Han

2ND MINI PROJECT  
December 5, 2022

## Abstract

**이미지 전처리 (Image Preprocessing)** 딥러닝을 활용한 차량 이미지 분류 모델 제작에서 이미지 데이터의 수집, 전처리는 학습 효율의 증대를 위해 필요하다. Selenium을 활용한 이미지 데이터 크롤링을 통해 데이터를 수집해본 결과 모델별 이미지의 크기가 상이하였고 학습에 적합한 정제된 이미지의 수량의 한계점을 파악하였다. 추가 데이터 확보를 위해 각 차량 브랜드별 공식 홈페이지에서 정제된 이미지를 크롤링하였고, Keras의 Image Data Generator를 활용해 다양한 각도로 데이터를 증강시켰다.

**전이학습 (Transfer Learning)** 전이학습이란 특정 분야의 문제를 해결하기 위해 학습된 정보를 다른 문제 해결을 위해 적용하는 것을 의미한다. 특히 딥러닝(Deep Learning)에서는 이미지 분류(image classification)에서 나타나는 성능평가 향상 문제 해결을 위해 자주 사용된다. 다양한 전이학습 모델 중 ResNet50, DenseNet121, MobileNet 모델을 선별하였고, 그 중 가장 높은 정확도(accuracy)를 나타낸 모델을 선택하였다.

**하이퍼 파라미터 튜닝 (Hyper Parameter Tuning)** 파라미터(parameter)는 모델 학습을 통해 값을 결정하는데 이 중 사용자가 값을 직접 조정하여 학습률 개선에 활용되는 방법이 하이퍼파라미터 튜닝이다. 구체적으로 최적의 하이퍼파라미터 값을 찾기 위해 이미지 데이터의 Input 크기, Convolution layer, Fully Connected layer의 구성 및 내부 수치, optimizer의 종류 및 내부 수치 등을 조율하여 학습률을 개선하였다.

**파인 튜닝 (Fine Tuning)** 파인 튜닝은 선행 학습된 모델 기반으로 시작된다. 전이학습 시, 기존모델의 가중치(weights)를 그대로 사용하지 않고 프로젝트의 목적이나 Input 이미지 데이터에 맞게 추가로 학습시킨다. 이 때 개신된 새로운 가중치를 사용하는 방법이다. 다양한 전이학습 모델을 테스트 하였고 성능이 가장 뛰어난 모델을 선택하였다. 전이학습 모델들은 1000개의 카테고리를 학습한 모델이지만 그대로 적용할 경우 다양한 제한적 상황에 놓일 수 있음을 판단하였다. 모델의 층(layer)을 Convolution layer 기준으로 나누어 freezing하는 layer의 개수에 차이를 두었다. 각각 accuracy를 측정하였고, 이중 가장 정확도가 높고 과대적합률을 낮추는 방향으로 Fine tuning을 진행하였다.

# 1 Introduction

## 1.1 서론

인공지능(AI) 분야에서 컴퓨터 비전(Computer Vision)은 인간의 시각 메커니즘과 유사하다. 인간의 시각적 학습은 거리, 움직임, 문제여부 판단에 따라 사물을 직접 보고, 구분하고 판단하는 과정을 거친다. 이와 유사하게 컴퓨터 비전에서 AI를 기반으로 이미지 또는 다양한 시각적 자료에서 의미 있는 정보를 수집하여 살펴보고 관찰한 뒤 스스로 판단까지 할 수 있다. 하지만 이미지 분류는 컴퓨터에게 쉽지 않은 문제다. 예를 들어 인간은 사물을 관찰할 때 직접 시각적으로 보고 실체가 있음을 인식하고 지각하지만 컴퓨터는 0-255 사이의 숫자로 표현되기에 좀 더 구체적으로 학습할 수 있는 조건이 필요하다. 이미지 분류는 많은 데이터가 필요하며 차이점을 정확히 구분하여 인식 할 때 까지 반복적인 학습을 해야 한다. 이를 위해 딥러닝(Deep Learning)과 컨볼루션 신경망(Convolutional Neural Network) 기술이 사용된다.

## 1.2 연구 과제 목적

딥러닝을 활용한 이미지 분류 모델은 흔하게 존재한다. 다만, 유사한 이미지에 대한 분류 모델은 데이터 셋마다 처리 방식을 다르게 해야만 원하는 목표에 도달 할 수 있다. 유사한 이미지 사이의 특성(feature)을 추출하여 올바르게 분류해야 하기 때문에 적용 가능한 reference를 찾는데 한계가 있다. 그렇다면 유사 이미지 데이터를 사용해 실용적인 모델을 설계할 수 있을지 의문이 들었고 직접 도전해보기로 하였다. 단순히 높은 평가 지수만을 추구하는 것만 아니라 다양한 시도를 통해 직접 이미지 데이터 전처리, 하이퍼파라미터 조정, 파인튜닝 등을 경험하고 이해하는데 목적을 두었다. 실생활에서 직접 와닿을 수 있는 다양한 주제를 논의 해 본 결과 이미지 데이터를 활용한 자동차 모델 분류를 선택하였다. 자동차 이미지 데이터셋을 크롤링(crawling)을 통해 직접 수집 및 설계하고, 이미지 사이즈, 품질향상, 데이터 개수 증폭을 통해 품질 좋은 데이터셋을 확보할 것이다. 수집한 데이터셋을 기반으로 차량분류에 대한 모델을 설계한다. 모델을 설계하는 과정에서 다양한 참고자료(reference) 및 선행 연구 모델 및 전이학습 모델까지 찾아보고, 더 나아가 직접 하이퍼 파라미터 및 파인튜닝을 통해 적절한 수치를 조정할 것이다. 이를 통해 발생 가능한 문제의 원인을 파악할 수 있는 기회라고 파악된다. 추가적으로 kaggle의 자동차 분류(cars classification) 데이터셋을 활용해 제작한 CNN모델의 성능을 검증하기로 계획을 수립했다. 최종적으로 차량분류 모델을 제작한 뒤, 다양한 차량 데이터셋에 범용적으로 활용 가능한 모델로 발전시킬 것이다.

## 1.3 선행 연구

### 1.3.1 ImageDataGenerator

이미지 데이터 분류모델 제작에서 데이터가 충분히 방대하면 효율적으로 학습할 수 있는 이점이 있다. 하지만 데이터 수집에 많은 시간과 비용이 할애되기에 데이터의 양을 증대 시킬 수 있는 효과적인 대안으로 ImageDataGenerator를 사용한다. ImageDataGenerator는 keras에서 제공하는 패키지로 원본 이미지 데이터의 수를 증강시킬 수 있다. CNN을 통한 학습에서 과적합(Overfitting)을 극복하고 성능을 높이기 위해 학습 데이터의 다양성을 늘리는 작업이 필요하다. 이 때, ImageDataGenerator를 통해 원본 이미지를 다양한 방식으로 조금 변형하여 학습시킬 수 있다. 다양한 파라미터들을 사전에 설정할 수 있다. 예를 들어, rotation\_range, width\_shift\_range, height\_shift\_range, horizontal\_flip 으로 원본 이미지를 수평 방향으로 반전시키고, 회전시키고, 상하좌우로 이동시키는 효과를 보여준다.

### 1.3.2 CNN (Convolutional Neural Network)

CNN(Convolutional Neural Network) 또는 합성곱신경망은 딥러닝에서 이미지 분류 모델 설계에 가장 많이 사용되고 있다. CNN은 이미지 데이터를 레이블이 지정된 픽셀 단위로 분해 한 뒤 딥러닝 모델이 시각적으로 확인 가능한 형태로 만들어 준다. 이 후 적용 가능한 수학적 함수들을 연산 한 뒤 예측 까지 가능하다. 하지만 예측을 하기 전까지 반복학습을 통해 정확성(accuracy)을 높이고 확인해야 한다. 충분한 학습이 진행되면, 인간이 시각적으로 이미지를 인식하는 것처럼 AI도 인식 할 수 있다.

구조적으로 살펴보면, CNN은 Input data를 1차원으로 변환하지 않아도 이미지의 공간에 대한 정보를 그대로 유지한 채 학습이 가능한 인공 신경망이다. 이미지의 특징을 추출하는 Convolution Layer와 클래스를 분류하는 Fully Connected Layer로 구성되었다. 하나의 Convolution Layer에는 Filter를 통해 합성곱 연산을 수행하고, 이미지의 특징을 추출하게 된다. 이 때, Filter의 이동량을 Stride라고 정의하고, Filter를 통해 합성곱이 수행된다. Stride 값만큼 이동하여 다시 Filter를 적용하는 방식으로 출력 이미지를

생성한다. 다만, Filter를 적용하는 과정에서 출력 이미지의 크기가 입력 이미지의 크기보다 작아지고, 이미지의 가장자리에 대한 픽셀 정보가 유실되는데 이를 방지하기 위해서 Padding이라는 가장자리 픽셀을 추가하여 출력 이미지의 크기를 유지시킨다. 출력 이미지의 크기를 유지하면서 Filter를 적용하고 최종 출력 이미지를 Flatten Layer를 통해 1차원으로 만든다.

다음으로, Fully Connected Layer로 입력하면 가중치(Weight)에 대한 연산량이 매우 커지고 이미지에 대한 특정 feature를 강조할 수 없을 것이다. 이를 Pooling Layer를 통해 해결할 수 있다. 2x2 크기로 Max-pooling 해서 2x2 범위 중 가장 큰 값을 대표값으로 선택하는 방식이다. Convolution Layer와 Pooling Layer를 통해 이미지에 대한 특징 추출이 마무리 된 후에는 Flatten Layer를 통해 1차원으로 만들어주고 Fully Connected Layer를 거치면서 classification이 수행된다. 마지막 출력층에서는 softmax activation을 통해 multi-label에 대한 classification을 마무리한다.

### 1.3.3 전이학습 (Transfer Learning)

이미지 분류에 사용한 딥러닝 모델을 다른 데이터셋이나, 또는 같은 데이터셋이라도 다른 목적을 가지고 수행하는 이미지 분류에 적용하는 것을 전이학습이라고 한다. 즉, 기존에 학습된(pre trained) 모델을 기반으로 새로운 이미지 데이터를 학습할 수 있음을 의미한다. 딥러닝의 층(layer)이 깊어질수록 이미지의 구체적인 특성(feature)을 학습하지만, 초기 층에서는 다양한 이미지의 보편적인 특성에 대해 학습한다. 데이터셋의 종류가 완전히 다르지 않다면 전이학습을 통해 빠르고 정확하게 학습할 수 있게 된다. 이를 통해 적은 데이터를 가지고 높은 성능지표를 보여 줄 수 있는 좋은 전략적 방법임을 알 수 있다. 또한, 전이학습 모델이 새롭게 학습할 데이터와 유사한 도메인을 가지고 있다면 데이터가 적은 경우에도 좋은 정확도 및 손실율을 나타낼 수 있다. 전이학습에 사용될 모델은 일반적으로 많이 쓰이는 모델 중 성능과 학습 시간을 고려하여 ResNet50, DenseNet121, MobileNet으로 선정하였다.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
				3×3 max pool, stride 2		
conv2_x	56×56	$\left[ \begin{array}{c} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 2$	$\left[ \begin{array}{c} 3 \times 3, 64 \\ 3 \times 3, 64 \end{array} \right] \times 3$	$\left[ \begin{array}{c} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$	$\left[ \begin{array}{c} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$	$\left[ \begin{array}{c} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{array} \right] \times 3$
conv3_x	28×28	$\left[ \begin{array}{c} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 2$	$\left[ \begin{array}{c} 3 \times 3, 128 \\ 3 \times 3, 128 \end{array} \right] \times 4$	$\left[ \begin{array}{c} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$	$\left[ \begin{array}{c} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 4$	$\left[ \begin{array}{c} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{array} \right] \times 8$
conv4_x	14×14	$\left[ \begin{array}{c} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 2$	$\left[ \begin{array}{c} 3 \times 3, 256 \\ 3 \times 3, 256 \end{array} \right] \times 6$	$\left[ \begin{array}{c} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 6$	$\left[ \begin{array}{c} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 23$	$\left[ \begin{array}{c} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{array} \right] \times 36$
conv5_x	7×7	$\left[ \begin{array}{c} 3 \times 3, 512 \\ 3 \times 3, 512 \end{array} \right] \times 2$	$\left[ \begin{array}{c} 3 \times 3, 512 \\ 3 \times 3, 512 \end{array} \right] \times 3$	$\left[ \begin{array}{c} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$	$\left[ \begin{array}{c} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$	$\left[ \begin{array}{c} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{array} \right] \times 3$
	1×1			average pool, 1000-d fc, softmax		
FLOPs		$1.8 \times 10^9$	$3.6 \times 10^9$	$3.8 \times 10^9$	$7.6 \times 10^9$	$11.3 \times 10^9$

Figure 1: ResNet의 구조

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112		7 × 7 conv, stride 2		
Pooling	56 × 56		3 × 3 max pool, stride 2		
Dense Block (1)	56 × 56	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 6$
Transition Layer (1)	56 × 56		1 × 1 conv		
	28 × 28		2 × 2 average pool, stride 2		
Dense Block (2)	28 × 28	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 12$
Transition Layer (2)	28 × 28		1 × 1 conv		
	14 × 14		2 × 2 average pool, stride 2		
Dense Block (3)	14 × 14	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 24$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 32$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 48$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 64$
Transition Layer (3)	14 × 14		1 × 1 conv		
	7 × 7		2 × 2 average pool, stride 2		
Dense Block (4)	7 × 7	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 16$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 32$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 32$	$\left[ \begin{array}{c} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{array} \right] \times 48$
Classification Layer	1 × 1		7 × 7 global average pool		1000D fully-connected, softmax

Figure 2: DenseNet의 구조

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32 \text{ dw}$	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64 \text{ dw}$	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128 \text{ dw}$	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128 \text{ dw}$	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256 \text{ dw}$	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256 \text{ dw}$	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5 \times$ Conv dw / s1	$3 \times 3 \times 512 \text{ dw}$	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512 \text{ dw}$	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024 \text{ dw}$	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool $7 \times 7$	$7 \times 7 \times 1024$
FC / s1	1024 $\times$ 1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

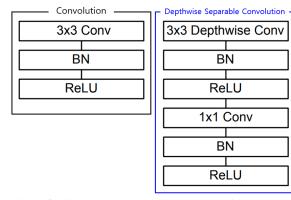


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

Figure 3: MobileNet의 구조

Figure 1부터 3까지 각 딥러닝 전이학습 모델들의 구조이다. MobileNet의 구조(Figure 3)를 살펴보면  $3 \times 3$  Depthwise Conv,  $1 \times 1$  Conv가 layer에 존재하는 것을 확인 할 수 있다. 일반적인 Convolution 연산을 통해서 만들어지는 Feature Map 한 개에는 Kernel Size \* Kernel Size \* Input Channel만큼의 parameter(parameter)가 필요하다. 연산에 필요한 parameter의 개수는 연산 속도를 결정하고, 1개의 Convolution layer가 아닌 여러 개의 Convolution layer를 통과하면서 Feature Map에 대한 연산이 쌓이게 된다. parameter의 개수는 딥러닝 모델의 학습 속도에 관한 성능 결정에 영향을 미칠 수 있다. 따라서 parameter의 개수를 줄이는 것이 딥러닝 모델 개발에 중요한 부분을 차지한다.

파라미터의 개수를 줄이고 모델 개발 성능을 높이는 방법 중 하나가 Depthwise separable convolution이다(Figure 4). 이미지 데이터의 경우 R,G,B의 3개 채널을 가지고 있어서 Feature Map 한 개를 생성할 때 Kernel Size \* Kernel Size \* 3의 연산이 필요하다. 하지만 Depthwise separable convolution의 경우에는 R,G,B 3개 채널이 하나로 병합되지 않고 각각 Feature Map을 구성하게 되어 한 Feature Map에 필요한 연산은 Kernel Size \* Kernel Size가 된다. 즉, Depthwise separable convolution에서는 연산량이 Input Channel만큼 줄어든다.

다음으로, MobileNet의 구조에서  $1 \times 1$  Conv가 Bottleneck의 핵심이다.  $1 \times 1$  Convolution은 Pointwise convolution이다.  $1 \times 1$  Convolution에서 parameter의 개수는  $1 \times 1 \times \text{Input Channel} \times \text{Output Channel}$ 로 연산량이 작기 때문에 Feature Map을 줄일 때 사용된다.  $1 \times 1$  Conv는  $1 \times 1$ 이기 때문에 공간적인 특징을 가지고 있지 않지만 오로지 연산량을 줄이기 위해 사용된다.  $1 \times 1$  Conv를 통해 연산량을 줄이고, feature를 추출하는 역할을 하는  $3 \times 3$  Conv에서 feature를 추출한다.

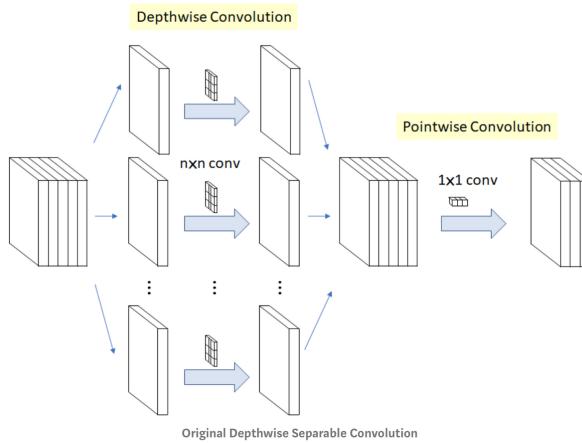


Figure 4: Depthwise &amp; Pointwise convolution

ResNet은 기울기 소실(Vanishing Gradient)과 과적합(Overfitting) 문제를 해결하기 위해 Residual block이라는 구조를 사용하였다. 가중치를 계산하는 weight layer가 있을 때, weight layer들을 통과하면서

계산된 값( $F(x)$ )으로 가중치를 업데이트 한다. 이때 weight layer를 통과하지 않은 가중치 값( $x$ )을 weight layer를 통과한 가중치 값과 더하여 이를 가중치로 업데이트하는 방식이다. 최종적으로, weight layer를 통과하지 않은 가중치 값( $x$ )도 계산에 포함되기 때문에 기존에 학습한 내용을 보존하면서 추가적으로 필요한 내용만 학습하게 된다.

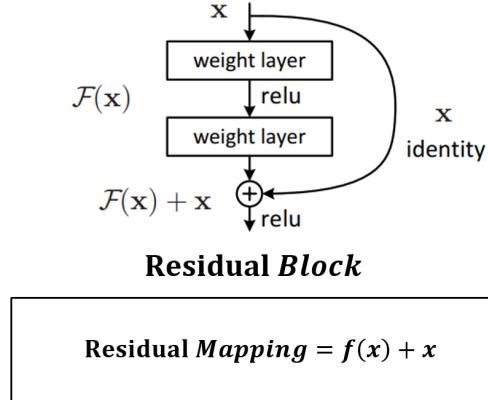


Figure 5: Residual block

DenseNet은 ResNet과 달리 이전 layer의 feature map을 그 이후의 모든 layer의 feature map에 연결한다. 이 때 채널 수가 많아지는 것을 방지하기 위해 각 layer의 채널 수를 작게 하여 parameter 수와 연산량을 줄인다. 결과적으로, feature map간의 연결(concatenation)연산을 통해 초반 layer의 feature 정보가 소실되는 문제를 해결할 수 있다. Concatenation연산을 진행 하기 위해서 feature map의 크기를 동일하게 맞춰줘야 한다. Convolutional network에서 feature map 크기를 줄여주는 pooling 연산이 꼭 필요하다. 여기서 정확한 pooling 연산을 위해 Dense Block이라는 개념이 사용된다. 여러 개의 구조로 갖춰진 Dense Block 사이에서 pooling 연산이 진행되어야 한다. 여기서 pooling 연산은 BatchNormalization, 1x1conv, 2x2 average-pooling으로 진행된다.

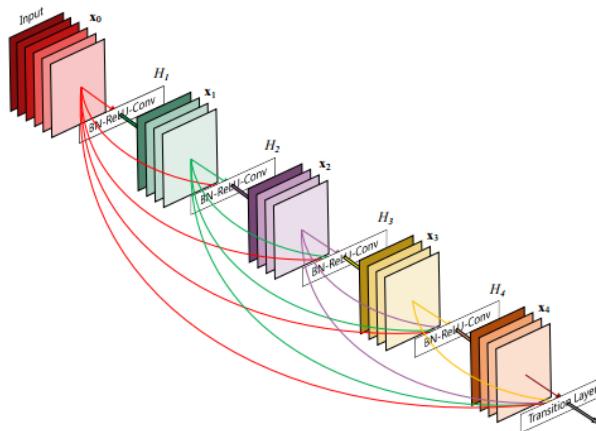


Figure 6: DenseNet의 feature map 연결

### 1.3.4 Fine Tuning

Fine tuning은 Transfer learning의 모델에서 학습한 모든 가중치에 추가로 학습하여 가중치를 조정하는 방법이다. Transfer learning으로 모델을 가져왔을 때 해당 모델의 이미 학습을 통해 얻은 가중치만

활용하기 위해 layer들을 train하지 않고 그대로 사용할 것인지, 일부 layer들은 freezing시키고 나머지 layer들을 학습가능(trainable)하게 하여 weight와 bias를 업데이트할 것인지, 전체 layer를 모두 trainable하게 하여 재학습할 것인지 조정하는 과정을 fine tuning이라고 할 수 있다. Convolution layer를 기준으로 나누어 freezing되는 layer의 개수를 조절하였고 accuracy를 비교하여 fine tuning을 진행하였다.

## 2 Method

### 2.1 데이터 수집

시중에 판매하는 5개의 국내 차량 브랜드를 기준으로 이미지 데이터를 수집했다. 기아, 현대, 르노삼성, 쌍용, 제네시스로 구성된 5개 브랜드의 차종을 Selenium을 활용해 구글 이미지에 검색하고 나오는 차량 이미지 데이터를 수집하였다. 각 모델 당 500장의 이미지를 수집하였고, Image Generator를 통해 모델 당 1000장의 데이터셋을 구성했다. 이렇게 구성된 데이터셋을 학습했을 때 전이학습의 유무와 상관없이 그 성능이 매우 좋지 않았다.

딥러닝 모델의 성능이 잘 나오지 않는 것에 대해 데이터셋에 문제가 있는지, 모델 자체에 문제가 있는지 판단하고 해결 방안을 찾아야 했다. 기존에 연구가 이미 수행되어 성능이 보장되고 데이터셋이 정제되어 있으므로 데이터셋의 라벨 별 이미지의 특징이 너무 뚜렷하지는 않은, 즉 어느 정도 유사한 이미지에 대한 분류를 할 수 있는 데이터셋을 가지고 현재 차량 이미지 데이터셋에 대해 성능이 잘 나오지 않는 딥러닝 모델에 적용시켜서 그 성능을 확인해 보았다.

AI Hub의 딥러닝 모델 성능 분류 데이터셋이 언급한 데이터셋의 성격에 부합하다고 생각되어 해당 데이터셋으로 딥러닝 모델을 검증해보았다. 기존에 수행된 연구에서 98.7%의 accuracy로 우수한 성능을 보였고, 이를 우리 딥러닝 모델에 적용시켜보았을 때, 딥러닝 모델은 성능에 적합하게 따로 전처리나 하이퍼 파라미터 조정, Fine Tuning을 수행하지 않은 상태에서 training data에 대해 약 98%, validation data에 대해 약 77%의 성능을 보였다. 물론 과대적합이 발생하였지만, 데이터셋과 딥러닝 모델에 대한 수정 없이 어느 정도 성능을 보인다는 점에서 딥러닝 모델 자체의 문제는 없다고 판단하였다.

기존의 차량 이미지 데이터셋보다 품질이 더 뛰어난 데이터셋을 갖춰야 하기 때문에 크롤링을 통한 차량 이미지가 아닌 정제된 형태의 이미지를 추가적으로 수집했다. 각 브랜드의 홈페이지에 모델링된 차량 이미지가 있었고, 해당 차량 이미지를 회전하여 다른 각도에서의 이미지를 수집할 수 있었다.



Figure 7: 모델링 차량 예시

단, 모델링된 차량 이미지만 사용하게 되면 모델링 되지 않은 차량 이미지에 대해서는 학습하지 못할 수 있다. 한 차량 모델에 대해 모델링된 차량 이미지 120장, 크롤링을 통한 차량 이미지 180장으로 총 300장으로 구성하였다. 이 때, 크롤링을 통한 차량 이미지는 기존의 500장에서 뚜렷하게 차량이 보이는 이미지로 선별하여 구성하였고, 모델링 차량 이미지, 크롤링 차량 이미지 모두 Image Generator를 통해 데이터를 증폭시켜서 데이터 개수를 충족시켰다.

### 2.2 딥러닝 모델 선정

우선, 전이학습 없이 제작한 CNN 모델 중 accuracy가 높은 구조를 기준으로 전이학습을 수행하기로 하였다. 전이학습을 하지 않은 Base CNN 모델에 대한 evaluate 결과값과 그래프는 다음과 같다.

```

model.evaluate(train_ds), model.evaluate(val_ds)

124/124 [=====] - 4s 32ms/step - loss: 1.0920 - accuracy: 0.6773
31/31 [=====] - 1s 32ms/step - loss: 1.2432 - accuracy: 0.6444
([1.092036247253418, 0.6772727370262146],
 [1.2432498931884766, 0.644444655637207])

```

Figure 8: Base CNN 모델의 evaluate 결과

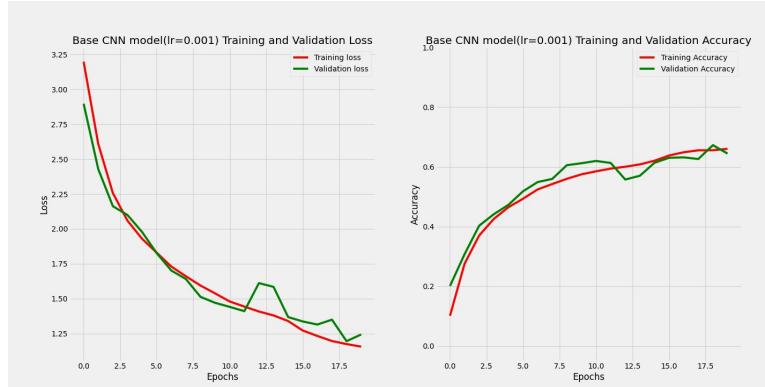


Figure 9: Base CNN 모델의 loss, accuracy 그래프

아래의 Base CNN 모델의 구조를 토대로 세 가지의 전이학습 모델(MobileNet, ResNet, DenseNet)을 제작하여 default learning rate 값인 0.001로 accuracy를 측정하였다.

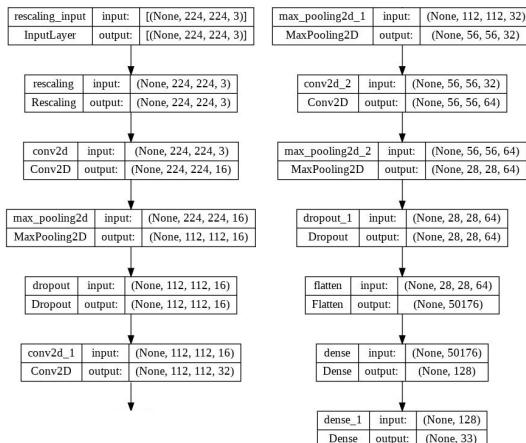


Figure 10: Base CNN 모델의 구조

각 모델의 evaluate 결과와 loss, accuracy에 대한 그래프는 다음과 같다.

```

1 model3.evaluate(ds_train), model3.evaluate(ds_valid)

117/117 [=====] - 8s 69ms/step - loss: 0.0177 - accuracy: 0.9952
39/39 [=====] - 3s 69ms/step - loss: 0.2043 - accuracy: 0.9511
([0.01770494319498539, 0.9951515197753906],
 [0.20431873202323914, 0.9511111378669739])

```

Figure 11: MobileNet 모델의 evaluate 결과

```

model.evaluate(train_ds), model.evaluate(val_ds)

124/124 [=====] - 4s 30ms/step - loss: 0.7434 - accuracy: 0.7759
31/31 [=====] - 1s 30ms/step - loss: 0.9323 - accuracy: 0.7646
([0.7433649301528931, 0.7758838534355164],
 [0.9323174953460693, 0.7646464705467224])

```

Figure 12: ResNet 모델의 evaluate 결과

```

124/124 [=====] - 22s 173ms/step - loss: 0.5990 - accuracy: 0.8140
31/31 [=====] - 5s 176ms/step - loss: 0.4967 - accuracy: 0.8263
([0.5989903211593628, 0.8140151500701904],
 [0.4967050552368164, 0.8262626528739929])

```

Figure 13: DenseNet 모델의 evaluate 결과

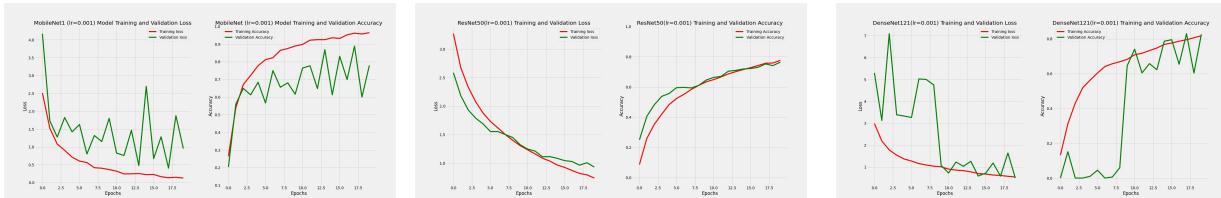


Figure 14: MobileNet, ResNet, DenseNet 모델의 loss, accuracy 그래프

전이학습 시에 default learning rate 값인 0.001을 사용했을 때, MobileNet을 제외하고는 성능이 좋지 않았다. Learning rate는 신경망에서 parameter 값을 얼마나 재조정할 것인지를 결정해주는 수치인데, 전이학습한 모델이 imageNet 데이터셋을 학습할 때 조정한 parameter 값을 재조정하는 과정에서 성능이 떨어졌다고 판단하였다. 차량 이미지 데이터셋의 크기가 작기 때문에 learning rate 값을 줄여서 크기가 큰 imageNet 데이터셋에 대한 parameter 값을 조금 더 반영하게 만들었다. 다음은 learning rate를 0.0001로 줄인 각 전이학습 모델의 evaluate 결과와 loss, accuracy에 대한 그래프이다.

```

model.evaluate(train_ds), model.evaluate(val_ds)

124/124 [=====] - 9s 72ms/step - loss: 0.0556 - accuracy: 0.9835
31/31 [=====] - 2s 72ms/step - loss: 0.1554 - accuracy: 0.9606
([0.05561702698469162, 0.9834595918655396],
 [0.15537835657596588, 0.960606038570404])

```

Figure 15: MobileNet 모델의 evaluate 결과

```

model.evaluate(train_ds), model.evaluate(val_ds)

124/124 [=====] - 23s 184ms/step - loss: 0.0580 - accuracy: 0.9854
31/31 [=====] - 6s 187ms/step - loss: 0.2257 - accuracy: 0.9439
([0.05796976387500763, 0.98535352945327761],
 [0.22574779391288757, 0.9439393877983093])

```

Figure 16: ResNet 모델의 evaluate 결과

```

124/124 [=====] - 24s 190ms/step - loss: 0.0434 - accuracy: 0.9860
31/31 [=====] - 6s 190ms/step - loss: 0.1568 - accuracy: 0.9576
([0.043444275856018066, 0.9859848618507385],
 [0.15679757297039032, 0.9575757384300232])

```

Figure 17: DenseNet 모델의 evaluate 결과

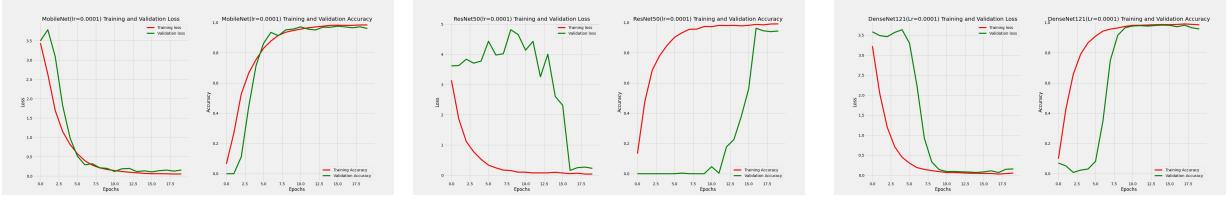


Figure 18: MobileNet, ResNet, DenseNet 모델의 loss, accuracy 그래프

다른 모델과 다르게 ResNet 모델에서는 validation data의 loss와 accuracy 값이 epoch 16부터 정상화되고, 그 수치도 다른 모델에 비해 낮게 측정되었다. 이러한 이유로 Fine tuning은 MobileNet과 DenseNet 모델에 대해 진행하였다.

### 2.3 Fine Tuning

앞선 과정에서 전체 layer에 대해 trainable하게 설정하고 전이학습을 진행했는데, fine tuning을 통해서 일부 layer는 freezing시켜서 학습되지 않고 imagenet 데이터셋을 학습할 때의 weight 값을 그대로 사용하게 만들었다. Layer를 freezing 시키는 위치의 기준은 Convolution 층을 기준으로 나누어서, 설정한 Convolution 층부터 trainable하게 fine tuning을 진행하였다. 코드는 Figure 19와 같은 방식으로 작성하였다.

DenseNet Freezing 140

```
[ ] from tensorflow.keras.applications.densenet import DenseNet121
base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

[ ] num = 1
for layer in base_model.layers:
    if num >= 140:
        layer.trainable = True
    else:
        layer.trainable = False
    num += 1
```

Figure 19: DenseNet의 140번 층에 대한 freezing 코드

#### 2.3.1 MobileNet

```
<keras.layers.convolutional.Conv2D object at 0x7eff0633fb0> 2
<keras.layers.convolutional.Conv2D object at 0x7eff0644710d> 8
<keras.layers.convolutional.Conv2D object at 0x7eff08201c70> 15
<keras.layers.convolutional.Conv2D object at 0x7eff0f619dc4d> 21
<keras.layers.convolutional.Conv2D object at 0x7eff0823bac0> 28
<keras.layers.convolutional.Conv2D object at 0x7eff062c3100> 34
<keras.layers.convolutional.Conv2D object at 0x7eff062e9c70> 41
<keras.layers.convolutional.Conv2D object at 0x7eff062c3340> 47
<keras.layers.convolutional.Conv2D object at 0x7eff08277340> 53
<keras.layers.convolutional.Conv2D object at 0x7eff0f624c580> 59
<keras.layers.convolutional.Conv2D object at 0x7eff0f624c7f0> 65
<keras.layers.convolutional.Conv2D object at 0x7eff06332220> 71
<keras.layers.convolutional.Conv2D object at 0x7eff0f6273880> 78
<keras.layers.convolutional.Conv2D object at 0x7eff0f60f91c0> 84
```

Figure 20: MobileNet의 Convolution 층과 그 위치

MobileNet은 Convolution 층은 14개로 구성되어 있다. 각 Convolution 층 이후부터 학습시키기 만들어서 accuracy를 측정했고, 결과가 뚜렷하게 차이 나는 8층, 47층, 84층에 대해 그려보았다.

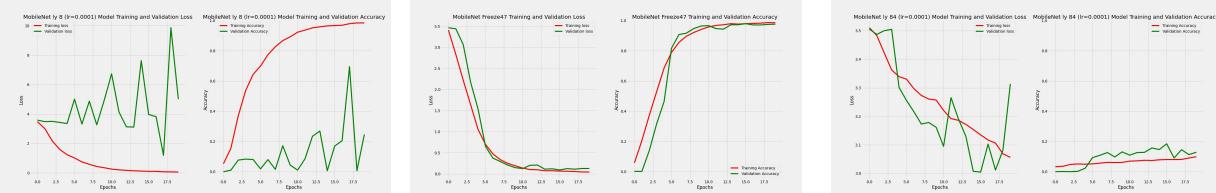


Figure 21: MobileNet 모델의 8층, 47층, 84층에 대해 fine tuning을 진행한 loss, accuracy 그래프

```
# mobilenet_freeze_47
model.evaluate(ds_train),model.evaluate(ds_valid)

124/124 [=====] - 9s 71ms/step - loss: 0.0154 - accuracy: 0.9960
31/31 [=====] - 2s 71ms/step - loss: 0.1287 - accuracy: 0.9758
([0.015416977927088737, 0.9959595799446106],
 [0.12867313623428345, 0.9757575988769531])
```

Figure 22: MobileNet 모델의 47층에 대해 fine tuning을 진행한 evaluate 결과

47층 이후부터 trainable하게 만들었을 때 가장 accuracy가 높았다. Figure 23이 MobileNet의 layer 구조를 나타낸 것인데, 일정하게 layer가 반복되는 구조인 것을 확인할 수 있다. 따라서, 47층 이전을 freezing 시켰을 때 성능이 가장 좋게 나온 이유는 다음과 같다. MobileNet 모델이 imageNet의 데이터셋을 학습하였고, 추출한 이미지의 포괄적인 feature를 기반으로 차량 이미지 데이터셋을 학습하였다. 여기서 차량 이미지에 대한 feature를 추출하기에 가장 적당한 기준이 47층임을 확인할 수 있었다.

```
<keras.engine.input_layer.InputLayer object at 0x7f347017e7c0>
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f34e0aee4fd0>
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34e0c7c310> 3
<keras.layers.activation.relu.ReLU object at 0x7f34e0c7c1cd0> 4
<keras.layers.convolutional.depthwise_conv2d.DepthwiseConv2D object at 0x7f346edf7e20> 5
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34603497c0> 6
<keras.layers.activation.relu.ReLU object at 0x7f34603497f0> 7
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f3460333730> 8
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f346033730> 9
<keras.layers.activation.relu.ReLU object at 0x7f34602ea2e0> 10
<keras.layers.reshape.zero_padding2d.ZeroPadding2D object at 0x7f34602ea7c0> 11
<keras.layers.convolutional.depthwise_conv2d.DepthwiseConv2D object at 0x7f3460349940> 12
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602fb5b0> 13
<keras.layers.activation.relu.ReLU object at 0x7f34602fb7b0> 14
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f34602fb7f0> 15
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602fb130> 16
<keras.layers.activation.relu.ReLU object at 0x7f346030c2b0> 17
<keras.layers.convolutional.depthwise_conv2d.DepthwiseConv2D object at 0x7f346030c0a0> 18
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460291af0> 19
<keras.layers.activation.relu.ReLU object at 0x7f34602984c0> 20
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f3460298f70> 21
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460298e20> 22
<keras.layers.activation.relu.ReLU object at 0x7f34602ab7b0> 23
<keras.layers.reshape.zero_padding2d.ZeroPadding2D object at 0x7f34602ab7f0> 24
<keras.layers.convolutional.depthwise_conv2d.DepthwiseConv2D object at 0x7f346029e7c0> 25
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602b1730> 26
<keras.layers.activation.relu.ReLU object at 0x7f34602b9100> 27
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f34602b9550> 28
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602b9b50> 29
<keras.layers.activation.relu.ReLU object at 0x7f34602c52b0> 30
<keras.layers.convolutional.depthwise_conv2d.DepthwiseConv2D object at 0x7f34602c54c0> 31
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602b1970> 32
<keras.layers.activation.relu.ReLU object at 0x7f34602914c0> 33
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f3460291df0> 34
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34603050a0> 35
<keras.layers.activation.relu.ReLU object at 0x7f34602abe20> 36
<keras.layers.reshape.zero_padding2d.ZeroPadding2D object at 0x7f3460349250> 37
<keras.layers.convolutional.depthwise_conv2d.DepthwiseConv2D object at 0x7f34602f2430> 38
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602ce9d0> 39
<keras.layers.activation.relu.ReLU object at 0x7f34602558e0> 40
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f346025570d> 41
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460255a60> 42
<keras.layers.activation.relu.ReLU object at 0x7f34602d8b0> 43
<keras.layers.convolutional_depthwise_conv2d.DepthwiseConv2D object at 0x7f346025df0> 44
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602b5040> 45
<keras.layers.activation.relu.ReLU object at 0x7f3460269100> 46
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f34602b95b0> 47
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602693a0> 48
<keras.layers.activation.relu.ReLU object at 0x7f3460276070> 49
<keras.layers.convolutional_depthwise_conv2d.DepthwiseConv2D object at 0x7f346027610> 50
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602ce4c0> 51
<keras.layers.activation.relu.ReLU object at 0x7f34602b8520> 52
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f34602b96b0> 53
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602b86b0> 54
<keras.layers.activation.relu.ReLU object at 0x7f3460212460> 55
<keras.layers.convolutional_depthwise_conv2d.DepthwiseConv2D object at 0x7f3460212a30> 56
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460219460> 57
<keras.layers.activation.relu.ReLU object at 0x7f3460228490> 58
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f34602264c0> 59
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460226760> 60
<keras.layers.activation.relu.ReLU object at 0x7f34602354c0> 61
<keras.layers.convolutional_depthwise_conv2d.DepthwiseConv2D object at 0x7f34602354f0> 62
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460226b70> 63
<keras.layers.activation.relu.ReLU object at 0x7f3460224850> 64
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f3460224a30> 65
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460244910> 66
<keras.layers.activation.relu.ReLU object at 0x7f34601d0880> 67
<keras.layers.convolutional_depthwise_conv2d.DepthwiseConv2D object at 0x7f34601d0a90> 68
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460244e80> 69
<keras.layers.activation.relu.ReLU object at 0x7f3460244f00> 70
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f3460226b50> 71
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460226d90> 72
<keras.layers.activation.relu.ReLU object at 0x7f34602c39d0> 73
<keras.layers.reshape.zero_padding2d.ZeroPadding2D object at 0x7f34602765b0> 74
<keras.layers.convolutional_depthwise_conv2d.DepthwiseConv2D object at 0x7f346025dd0> 75
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34602b1df0> 76
<keras.layers.activation.relu.ReLU object at 0x7f34602b86d0> 77
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f3460250b20> 78
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f3460298a00> 79
<keras.layers.activation.relu.ReLU object at 0x7f34602fe610> 80
<keras.layers.convolutional_depthwise_conv2d.DepthwiseConv2D object at 0x7f346027e2e0> 81
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34601dd60> 82
<keras.layers.activation.relu.ReLU object at 0x7f346025b7b0> 83
<keras.layers.convolutional.conv2d.Conv2D object at 0x7f34601dc880> 84
<keras.layers.normalization.batch_normalization.BatchNormalization object at 0x7f34601dc9a0> 85
<keras.layers.activation.relu.ReLU object at 0x7f34601dbbb0> 86
```

Figure 23: MobileNet의 전체 layer 구조

### 2.3.2 DenseNet

DenseNet은 MobileNet과 다르게 Convolution layer의 개수가 120층으로 구성되어 있다. Convolution 층마다 fine tuning을 진행하기에 어려움이 있었다. 우선, DenseNet도 MobileNet처럼 일정한 패턴대로 layer 구조를 반복하였는지 확인해보았다.

Figure 24: DenseNet의 전체 layer 구조

MobileNet과 다르게 DenseNet에는 Average Pooling 층이 3개 존재했고, 일정한 layer 구조를 반복 하긴 하지만 Concatenate층이 비교적 큰 간격으로 사이사이 존재했다. Average Pooling 층이 accuracy에 얼마나 영향을 주는지, 우선 Average Pooling 바로 이전 층인 52층, 140층, 312층과 Average Pooling 바로 이후 층인 56층, 144층, 316층에 대해 fine tuning을 진행하였고, accuracy를 비교하였다. Figure 25, 26, 27을 통해 Average Pooling의 accuracy를 상승시킨다는 것을 확인할 수 있었다.

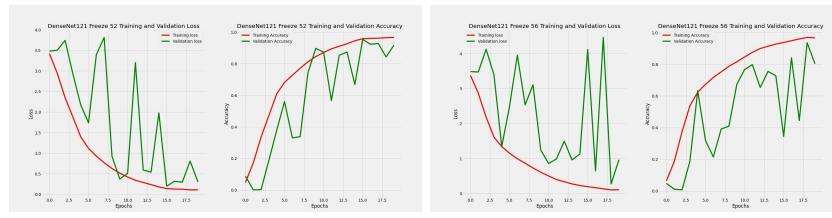


Figure 25: Average Pooling 층을 기준으로 52층과 56층에 대한 fine tuning

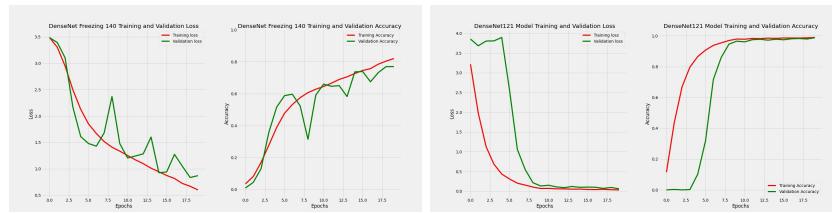


Figure 26: Average Pooling 층을 기준으로 140층과 144층에 대한 fine tuning

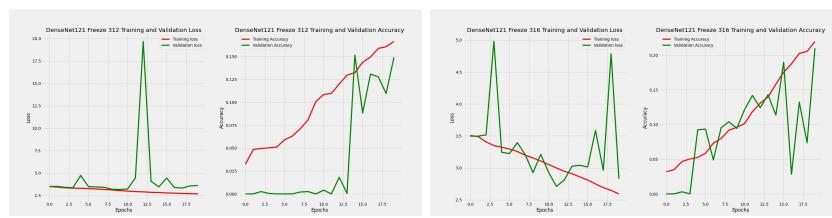


Figure 27: Average Pooling 층을 기준으로 312층과 316층에 대한 fine tuning

MobileNet과 마찬가지로 중간층에 대해 fine tuning을 진행했을 때 성능이 가장 좋았고, 이후의 fine tuning은 중간층을 기준으로 학습했다.

DenseNet에서는 Concatenate층을 통해 이전의 feature map을 이후의 모든 feature map과 연결하여 layer 초반의 feature를 잃지 않고 보존한다. 이를 토대로 Average Pooling 층을 통과하고, 이후 첫 Concatenate 층을 통과한 151층에 대해 fine tuning을 진행했을 때 가장 성능이 좋을 것이라고 가설을 세웠고, 이에 대한 fine tuning을 진행하였다.

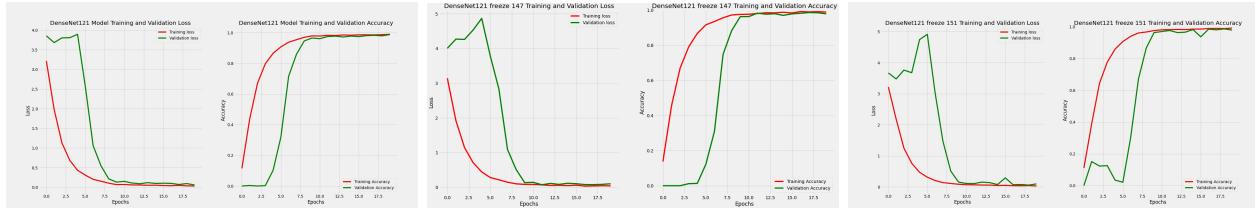


Figure 28: 144층, 147층, 151층에 대한 fine tuning

```
freezing_model.evaluate(train_ds), freezing_model.evaluate(val_ds)

124/124 [=====] - 22s 176ms/step - loss: 0.0048 - accuracy: 0.9986
31/31 [=====] - 6s 178ms/step - loss: 0.0612 - accuracy: 0.9884
([0.004802966024726629, 0.998610925674438],
 [0.061211179941892624, 0.9883838295936584])
```

Figure 29: DenseNet 모델의 144층에 대해 fine tuning을 진행한 evaluate 결과

```
freezing_model2.evaluate(train_ds), freezing_model2.evaluate(val_ds)

124/124 [=====] - 22s 179ms/step - loss: 0.0459 - accuracy: 0.9876
31/31 [=====] - 6s 176ms/step - loss: 0.1004 - accuracy: 0.9783
([0.04589194431900978, 0.9876262545585632],
 [0.10038437694311142, 0.9782828092575073])
```

Figure 30: DenseNet 모델의 147층에 대해 fine tuning을 진행한 evaluate 결과

```
freezing_model4.evaluate(train_ds), freezing_model4.evaluate(val_ds)

124/124 [=====] - 22s 178ms/step - loss: 0.0243 - accuracy: 0.9919
31/31 [=====] - 6s 182ms/step - loss: 0.0977 - accuracy: 0.9788
([0.024253303185105324, 0.991919219493866],
 [0.09773522615432739, 0.978787899017334])
```

Figure 31: DenseNet 모델의 151층에 대해 fine tuning을 진행한 evaluate 결과

예상과는 달리 concatenate층 아래의 151층보다 Average Pooling층 아래의 144층에 대해 fine tuning을 진행했을 때 가장 성능이 좋았다. 최종적으로 MobileNet에 대한 fine tuning과 비교해보면, DenseNet의 144층에 대해 fine tuning을 진행했을 때 train dataset에 대해 loss 0.0048, accuracy 0.9986, validation dataset에 대해 loss 0.0612, accuracy 0.9884로 가장 높은 성능을 나타냈다.

## 3 Result

### 3.1 최종 result

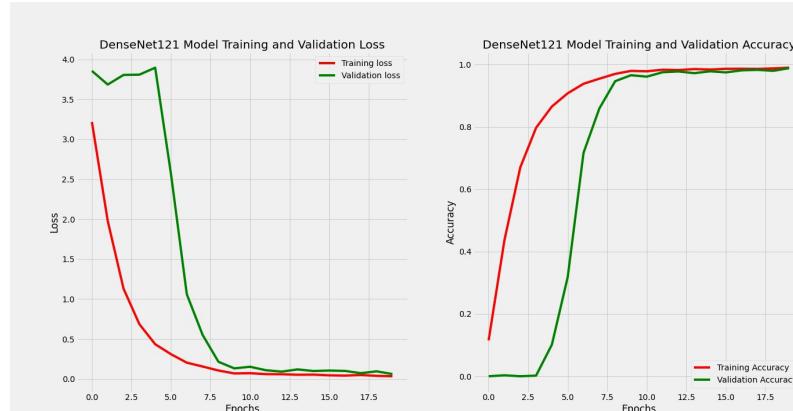


Figure 32: Final model loss, accuracy graph

```

freezing_model.evaluate(train_ds), freezing_model.evaluate(val_ds)

124/124 [=====] - 22s 176ms/step - loss: 0.0048 - accuracy: 0.9986
31/31 [=====] - 6s 178ms/step - loss: 0.0612 - accuracy: 0.9884
([0.004802966024726629, 0.9986110925674438],
[0.061211179941892624, 0.9888838295936584])

```

Figure 33: Final model evaluation

### 3.2 최종 모델 Architecture

DenseNet121을 전이학습 모델로 선택하였고, Figure 10의 Base CNN 모델을 토대로 전이학습을 진행하였다. 144번 층 이전의 layer는 freezing하는 fine tuning을 진행하였고, compile 시 learning rate 값은 0.0001로 하였다. 최종 모델의 architecture은 Figure 34와 같다.

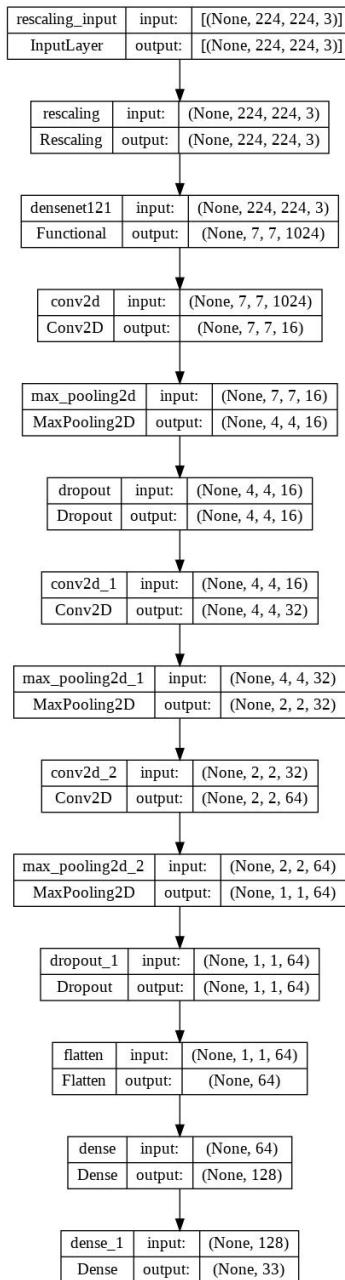


Figure 34: Final model architecture

## 4 Discussion

### 4.1 연구 보완점

#### 1. GPU & RAM 부족

연구 환경은 Google의 Colab에서 진행되었다. GPU와 RAM의 성능과 사용량에 한계가 있기 때문에 차량 이미지 데이터의 크기를 224x224보다 더 키워서 input에 넣어보거나, 차량 이미지 데이터셋의 크기를 9900개보다 크게 하여 모델 제작 시 더 많은 학습을 시켜보는 등 다양한 시도를 하지 못했다.

#### 2. 범용적인 모델 제작

제작한 모델의 범용적 모델로써 활용 가능한지 파악하기 위해 kaggle의 cars classification 데이터셋을 추가로 학습시켜보았다. 기존에 학습시킨 자동차 이미지 데이터셋은 33종의 국산차에 대한 데이터셋이고, kaggle의 cars classification 데이터셋은 20종의 외제차에 대한 데이터셋이다. 서로 다른 데이터셋이지만 유사한 차량 이미지에 대한 데이터셋이라는 공통점이 있기 때문에 33종의 국산차에 대해 학습시켜서 추출한 이미지 feature map을 통해 20종의 외제차를 분류할 수 있는지 테스트했다.

```
[ ] last_layer = model.get_layer('dense_1')
last_output = last_layer.output

x = layers.Flatten()(last_output)
x = layers.Dense(1024,activation='relu')(x)
x = layers.Dense(512,activation='relu')(x)
x = layers.Dense(256,activation='relu')(x)
x = layers.Dense(128,activation='relu')(x)
x = layers.Dense(20,activation='softmax')(x)

new_model = Model(model.input, x)

▶ for layer in new_model.layers:
    layer.trainable = True
```

Figure 35: Kaggle 데이터셋 학습을 위한 기존 모델의 출력층 변경 코드

Figure 35와 같이 기존 모델의 마지막 출력층의 output을 Flatten layer를 통해 펴주고, 이를 Fully Connected layer와 softmax 출력층을 통과시켜서 기존 33종에 대한 classification을 20종의 외제차 classification으로 만들어주었다.

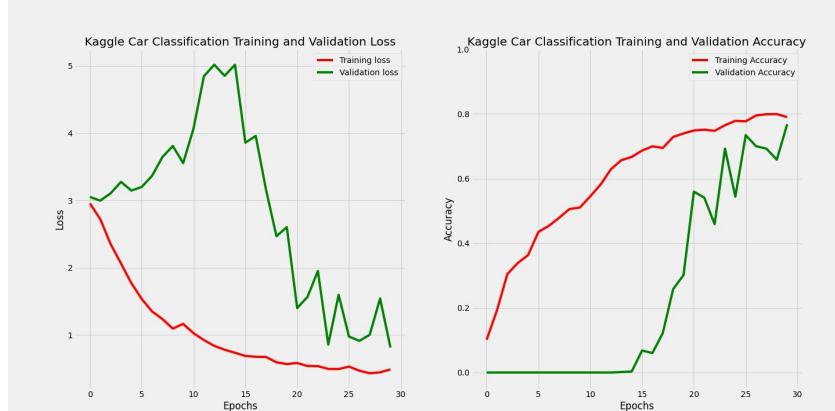


Figure 36: Kaggle 데이터셋에 대한 loss, accuracy 그래프

```

| new_model.evaluate(train_ds2), new_model.evaluate(val_ds2)

41/41 [=====] - 7s 174ms/step - loss: 0.6618 - accuracy: 0.7720
11/11 [=====] - 2s 161ms/step - loss: 0.8147 - accuracy: 0.7682
([0.661840558052063, 0.7720247507095337],
 [0.8146848082542419, 0.7681607604026794])

```

Figure 37: Kaggle 데이터셋에 대한 evaluation 결과

train dataset에 대해 약 77%, validation dataset에 대해 약 76%의 accuracy라는 결과가 나왔다. 그래프 또한 validation이 train의 그래프 형태를 따라가는 일반적인 형태의 그래프 모양이 아닌 일반적이지 않은 모양의 결과가 나왔다. 학습하지 않은 다른 차량 이미지 데이터셋에 대해서 좋은 성능을 가지려면 기존에 학습한 차량 이미지 데이터셋의 크기가 커야하는데 학습한 데이터셋의 크기가 총 9900개의 이미지로, 그 크기가 크지 않아서 범용적인 모델로 사용되기에는 무리가 있다고 판단하였다.

## 4.2 결론

국산 차량 33종 이미지를 크롤링을 하여 차량 이미지 데이터셋을 제작하였다. CNN 모델을 설계하였고, 이를 기반으로 DenseNet121 모델에 적용해 전이학습시켰다. DenseNet121 모델이 학습한 imageNet의 데이터셋에서 추출한 이미지 feature를 조금 더 반영하기 위해 0.0001의 learning rate 값을 사용하였다. Fine tuning을 통해 DenseNet121 모델의 144번째 layer부터 trainable하게 설정하여 validation 데이터셋에 대해 98.84%의 accuracy를 가지는 모델을 최종 제작하였다.

## 5 References

- [1] Chollet, François. "Xception: Deep learning with depthwise separable convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.
- [2] G. Huang, Z. Liu, L. Van Der Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 2261-2269, doi: 10.1109/CVPR.2017.243.
- [3] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
- [4] Howard, Andrew G., et al. "Mobileneets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).
- [5] Zoph, Barret & Vasudevan, Vijay & Shlens, Jonathon & Le, Quoc. (2018). Learning Transferable Architectures for Scalable Image Recognition. 8697-8710. 10.1109/CVPR.2018.00907.
- [6] Eunwan Kim, Jeonghoon Choi, Bowoo Kim, Dongjun Suh. (2022). A Study on the Development of Strawberry Growth Stage Classification Model. Proceedings of Symposium of the Korean Institute of communications and Information Sciences, (), 1002-1003.
- [7] "Bag-of-Tricks — 딥러닝 성능을 높이기 위한, 다양한 꿀팁들", scalalang2, 2020년 8월 19일 수정, 2022년 11월 28일 접속, <https://medium.com/curg/%EB%94%A5%EB%9F%AC%EB%8B%9D-%EC%84%B1%EB%8A%A5%EC%9D%84-%EB%86%92%EC%9D%B4%EA%B8%B0-%EC%9C%84%ED%95%9C%EB%8B%A4%EC%96%91%ED%95%9C-%EA%BF%80%ED%8C%81%EB%93%A4-1910c6c7094a>.
- [8] "딥러닝 모델 성능 개선하는 법 (캐글 Tip!)", FacerAin, 2022년 7월 31일 수정, 2022년 11월 29일 접속, <https://facerain.club/improve-dl-performance>.
- [9] "전이학습", 밥, 머신러닝, 코딩 말고는 를, 2020년 3월 30일 수정, 2022년 11월 28일 접속, <https://woongjun-warehouse.tistory.com/2>.
- [10] "14. 전이 학습 활용하기", Codetorial, 2020년 8월 25일 수정, 2022년 11월 28일 접속, [https://codetorial.net/tensorflow/transfer\\_learning.html](https://codetorial.net/tensorflow/transfer_learning.html).
- [11] "텐서플로2.0 대표 모델 시리즈(이미지 분류)", Loner의 학습노트, 2020년 12월 5일 수정, 2022년 11월 29일 접속, <https://wiserloner.tistory.com/1218>.

- [12] “작은 데이터셋으로 강력한 이미지 분류 모델 설계하기”, KEKOxTutorial, 2018년 10월24일 수정, 2022년 11월 30일 접속, [https://keraskorea.github.io/posts/2018-10-24little\\_data\\_powerful\\_Model](https://keraskorea.github.io/posts/2018-10-24little_data_powerful_Model).
- [13] “Transfer Learning / Fine Tuning”, 두리안의코딩나무, 2021년3월12일수정, 2022년 12월 1일 접속, <https://durian9s-coding-tree.tistory.com/9>.