2015**** 경영학과 김주형 인공지능개론 과제2**

0. 개발 환경

본 과제에서는 window10 64비트 환경에서 Python 언어와 Pytorch 라이브러리를 사용하여 모델링을 수행하였다. 주된 연산장비는 Nvidia GTX1080으로써 8GB의 VRAM을 가지고 있었다.

1. 데이터 전처리

1.1 육안 검수

본 과제에서는 가장 먼저 모든 데이터에 대한 육안 검수를 수행하였다. 총 4천여 장의 이미지 중 흑백사진, 잘못 지정된 클래스, 손상된 이미지 파일 등을 발견하고 처리하였다.**[부록 7.1]** 흑백사진은 전체 데이터셋 중 흑백사진의 비중이 약 0.26% 정도에 불과하였다. 만약 모델이 흑백사진을 고려하도록 유도하고 싶을 경우, 데이터 어그멘테이션 과정에서 일정한 확률로 칼라 이미지를 흑백 이미지로 변환하여 학습 데이터로 추가하는 방식으로 모델링을 하는 것도 가능하겠지만, 본 과제에서는 흑백사진의 비중이 너무 적으므로 이상치로 판단하고 학습과정에서 아예 제외하였다. 잘못 지정된 클래스를 가진 이미지와 손상된 이미지 파일도 학습과정에서 모두 제외하였다.

1.2 클래스 할당

실제 CNN 모델 학습 코드를 편하게 작성하기 위해, 각 개 품종 클래스를 알파벳 순서로 0부터 24까지 할당하였다. 또한 학습과정 중 이를 one-hot vector로 변환하여 사용하였다.

2. 사용한 모델

본 과제에서 사용한 CNN 모델은 EfficientNet 모델로써, 2019년 ICML에서 공개된 “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks” 논문에서 제안된 CNN 아키텍쳐이다. 처음에는 직접 구현한 코드를 사용하였으나, GPU메모리를 비정상적으로 많이 차지하여 Onnx(Open Neural Network Exchange) 기준에 맞게 최적화되고, Imagenet 데이터셋 분류 문제에 pretrained된 모델을 다운로드 받아 사용하였다. EfficientNet 모델을 직접 구현하여 사용할 경우 GPU 메모리를 7.2GB만큼 점유하였으나, 최적화된 모델을 사용하자 GPU메모리를 6.2GB 정도만 점유하는 비약적인 차이를 보여 부득이하게 타인이 구현한 모델을 사용하였다. EfficientNet 모델은 여러 개의 버전이 존재하고 (B0 ~ B7) 더 높은 숫자의 버전을 가질수록 더 많은 파라미터와 깊은 모델 구조를 가진다. 본 과제에서는 약 920만개의 모델 파라미터를 가지는 B2 모델을 사용하였다.

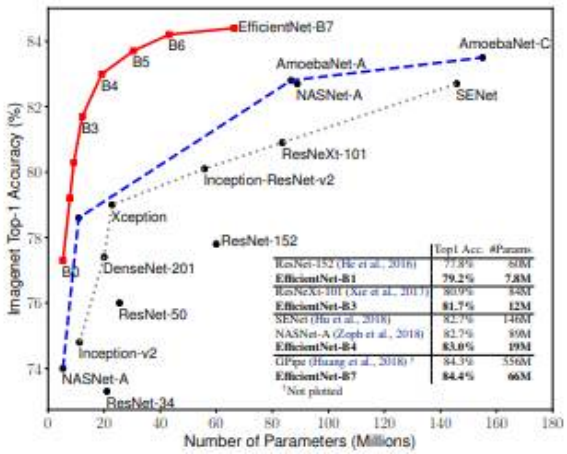


그림 1 Imagenet Classification 성능 비교
(출처: EfficientNet 논문)

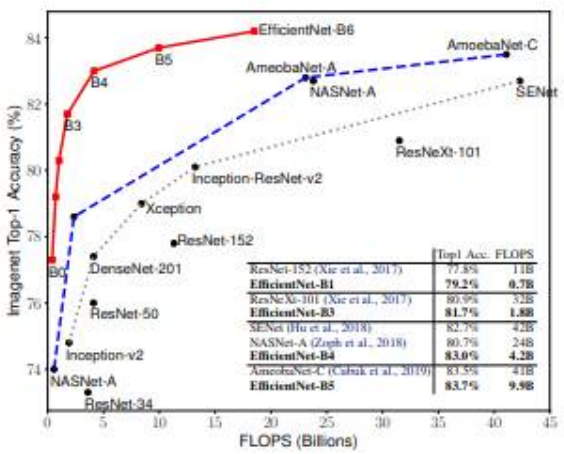


그림 2 Imagenet Classification 연산량 대비 모델 성능 비교
(출처: EfficientNet 논문)

3. 학습 기법

3.1 Data Augmentation

본 과제에서는 데이터셋의 부족함으로 인한 과적합, 학습 불안정 등의 문제점들을 해결하기 위해, 학습 데이터에 이미지를 무작위로 변환시켜 복제한 뒤 새로운 학습 데이터로 추가하는 데이터 어그멘테이션 기법을 적용하였다. 적용한 이미지 변환은 아래와 같다. 본 과제에서는 모든 이미지 데이터의 크기를 (256픽셀 x 256픽셀)로 통일하고자 하였다.

- RandomResizedCrop: 이미지를 원하는 사이즈의 출력의 무작위 배율(본 과제에서는 0.8~1.2)로 resize한 뒤, 그 이미지의 무작위 위치에서 최종 크기(256 x 256)만큼의 이미지를 잘라냄.
- RandomAffine: 이미지를 무작위 각도로 Affine 변환을 적용하여 회전시키고 기울임.
- 밝기 노이즈: 이미지의 밝기를 3/4 ~ 5/4 배로 무작위로 변화시킴.
- 대조 노이즈: 대조값을 3/4 ~ 5/4 배로 무작위로 변화시킴.
- Saturation 노이즈: 3/4 ~ 5/4 배로 무작위로 변화시킴.
- Hue 노이즈: Hue를 -0.1 ~ 0.1 만큼 무작위로 이동시킴.

본 과제에서 어그멘테이션은 Online 방식으로 수행하였다. 즉, 미리 이미지에 어그멘테이션을 적용해 디스크나 메모리에 저장해두고 학습과정에서 불러와서 사용하는 것이 아니라, 각 학습 배치가 수행될 때마다 CPU에서 바로 계산하여 모델에 입력하는 방식으로 진행하였다. Disk read로 인한 병목을 줄이기 위하여, 모든 데이터셋은 원본 이미지를 메모리에 올려두는 방식으로 구현하였다.

어그멘테이션을 적용한 이미지 데이터는 Normalization 또는 Standardization을 적용하지 않고 모델에 바로 입력하였다. 앞서 설명하였듯이, 본 과제에서는 Online방식의 데이터 어그멘테이션을 수행하였기 때문에 이미지 픽셀 값들의 mean, variance와 같은 통계량을 사전에 계산하더라도 오차가 생길 수 있고, 사용한 모델인 EfficientNet 모델이 첫 레이어에서 2D Batch Normalization 계층을 포함하고 있는데 Batch mean과 Batch Variance를 계산하고 배치 이미지들에서 빼는 과정과 상충하기 때문이기도 하다.

3.2 Label Smoothing

Label Smoothing은 머신러닝 모델의 학습과정을 안정시켜주고 최종 성능을 개선시키는 데에 도움이 되는 기법으로써, 모델 아키텍쳐에 구애받지 않으며 딥러닝이 아닌 머신러닝 모델들에도 때로 적용 가능한 기법이다. 일반적인 Cross Entropy Loss 는 모델이 계산한 클래스별 확률 값 중 참(실제) 클래스의 확률값만 선택해, 해당 확률 값의 log 값만큼을 역전파시키는 데 사용된다. 이는 직관적으로 해석하면, 모델이 이미지가 주어졌을 때, 실제 클래스의 확률 값이 1이 되도록 유도하는 것이다. 이것은 이상적으로는 좋은 목표이지만, 실제 데이터셋은 잘못 분류된 클래스나 주어진 태스크의 영역을 벗어나는 모호한 데이터 등 수많은 노이즈의 가능성을 가지고 있다. 따라서, 이런 경우에는 라벨스무딩을 적용하여 모델이 조금 더 달성하기 쉬운 목표를 만들어줄 수 있다.

라벨스무딩은 각 데이터별 원핫인코딩 벡터와 모델의 log 소프트맥스 출력값을 곱하던 방식에서, 원핫인코딩 벡터의 값을 0과 1이 아닌 e=0.1 정도일 때 e/[class 수 - 1]과 1-e 값으로 변환시켜주는 방식으로 이루어진다. 따라서 쉽게 구현할 수 있으며, 거의 모든 경우에서 모델 성능과 학습 안정성을 개선하는데 도움이 된다.

3.3 최적화 알고리즘

본 과제의 모델을 학습시키기 위해, Adam 최적화 알고리즘이 아닌 SGD 알고리즘과 momentum 메커니즘을 적용하여 모델을 최적화 하였다. Learning Rate는 별도의 Schedule을 고안하여 적용했고, momentum은 사람들이 많이 사용하는 0.9의 계수를 사용하였다.

3.4 Hyperparameter Tuning

본 과제에서는 작성자의 노력의 부족으로, 하이퍼파라미터 튜닝을 심도 있게 수행하지 못했다. Bayesian Optimization, Neural Architecture Search 등 다양한 하이퍼파라미터 튜닝 방법이 존재하겠지만, 본 과제에서는 각 하이퍼파라미터마다 매뉴얼하게 값을 바꿔가며 조정하였다. 예를 들어 Learning Rate 값 같은 경우는 10의 승 단위로 바꿔가면서 성능을 비교한다거나 하는 방식으로 조금씩 조정하였다.

3.5 Learning Rate Scheduling

Learning Rate는 모델을 학습시키는 데에 매우 중요한 하이퍼파라미터이다. 최근 연구에 따르면, 일정한 주기를 가지고 Learning Rate 값을 규칙적으로 변화시키는 Cyclic learning rate 기법이 우수한 성능 향상 효과로 많은 주목을 받고 있다. 본 과제에서도 Cyclic learning rate 기법을 적용하였다.

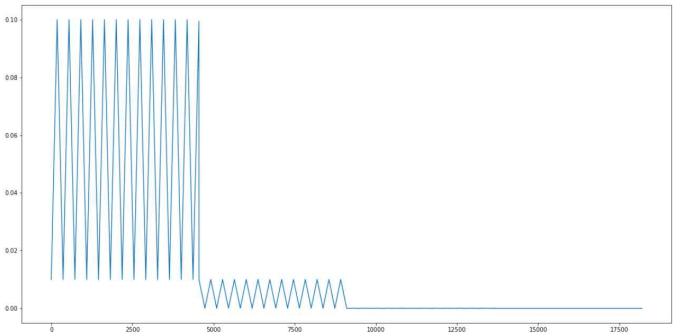


그림 3 본 과제에서 step 수에 따른 Learning Rate 값 변화

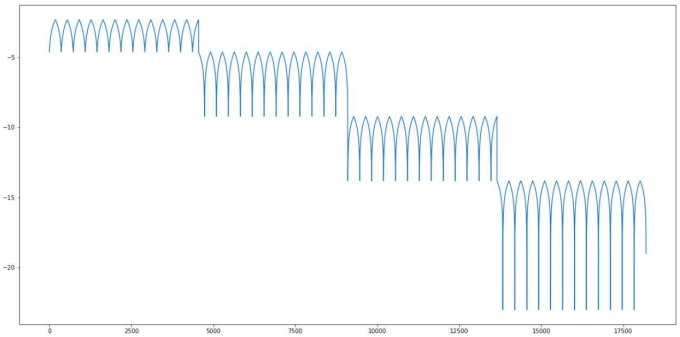


그림 4 Learning Rate의 log값 변화

3.6 Stochastic Weight Averaging

일반적으로 딥러닝 모델을 최적화하는 과정을 완료하면 Test 데이터셋의 local minimum의 근처 어딘가에 파라미터의 조합이 위치하게 된다. 그 상태에서 아무리 최적화를 더 진행하더라도, 정확한 local minimum의 위치를 도달하는 것은 불가능하고, local minimum이 있는 방향을 가로지르며 주변을 영역을 이동하게 된다. 하지만 이때, 주변 영역을 이동하는 동안의 위치를 각각 기록하여, 평균을 내면 실제 local minimum과 굉장히 가까운 위치를 알 수 있다. 이것이 Stochastic Weight Averaging의 motivation이다.

학습이 완료된 딥러닝 모델을 가지고, Train dataset에 대해 Batch Gradient(minibatch gradient가 아닌)를 계산하고 고정된 Learning Rate 값을 가지는 SGD를 일정한 횟수만큼 수행한 후, 각 Step마다 파라미터 값을 저장해 놓고 나중에 각 파라미터의 평균값을 구하여 최종 파라미터 값으로 사용하는 방식으로 구현하였다.

4. 학습과정

4.1 학습 횟수

본 과제에서는 모델 학습 단계를 총 4단계로 구성하여, 각 단계가 끝나면 Learning Rate 값의 범위를 크게 감소시켰다. 각 단계마다 동일하게 50 epoch씩 학습을 진행하였고, Learning Rate 값의 범위는 아래와 같았다.

- Phase 1: 1e-1 ~ 1e-2
- Phase 2: 1e-2 ~ 1e-4
- Phase 3: 1e-4 ~ 1e-6
- Phase 4: 1e-6 ~ 1e-10

4.2 초기 학습 과정

초기 학습과정을 모니터링 하고자 train loss와 validation loss를 출력하였다. 단, train loss는 label smoothing을 적용하여 계산한 것이고, validation loss는 일반적인 Cross Entropy를 계산한 것이므로 둘을 정량적으로 비교해서는 안 된다는 점에 유의해야할 것이다. 학습 초기에는 큰 값의 learning rate를 사용하여 성능이 불안정하였으나, 시간이 지날수록 점점 안정되어 작은 loss 값으로 수렴하기 시작하였다.

epoch : 0, train loss = 1.399054170, valid loss = 0.471706003, acc = 0.890, time: 91.930320 sec
epoch : 1, train loss = 1.200555444, valid loss = 1.710415602, acc = 0.557, time: 80.668284 sec
epoch : 2, train loss = 1.381217122, valid loss = 0.498225719, acc = 0.881, time: 81.504284 sec
epoch : 3, train loss = 0.896900237, valid loss = 0.340960443, acc = 0.924, time: 84.739947 sec
epoch : 4, train loss = 0.784339249, valid loss = 0.421292663, acc = 0.898, time: 82.826652 sec
epoch : 5, train loss = 0.787814379, valid loss = 0.455258876, acc = 0.890, time: 83.073292 sec
epoch : 6, train loss = 0.857626319, valid loss = 0.347748637, acc = 0.924, time: 84.141215 sec

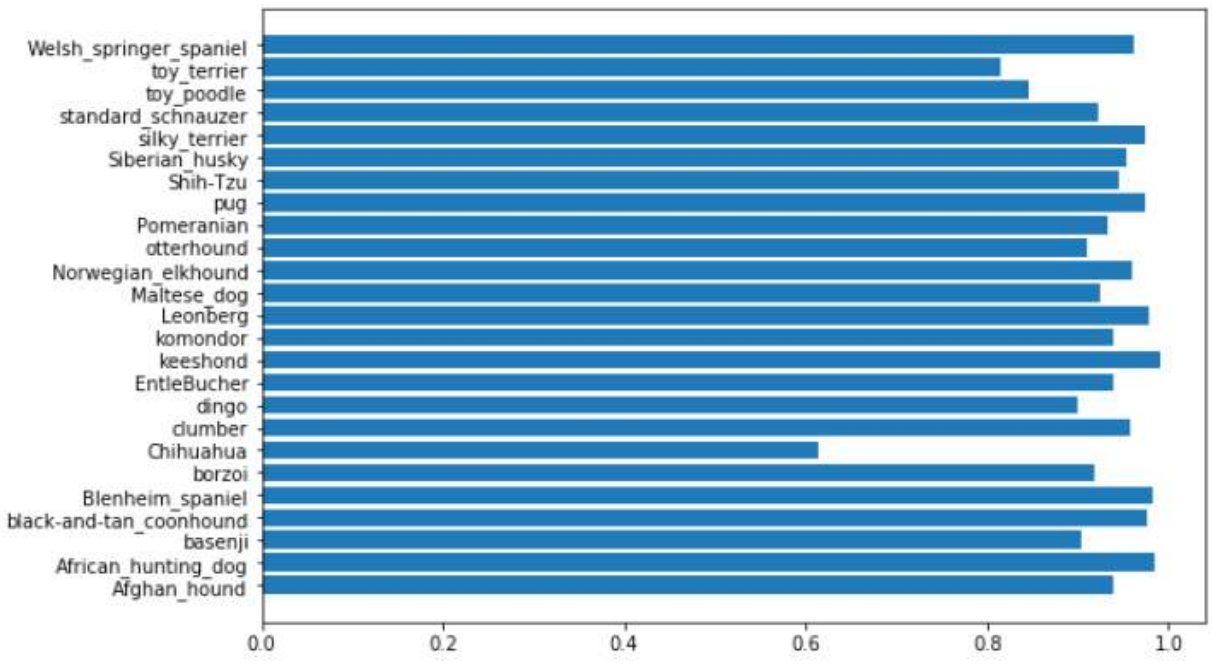
EfficientNet을 Pretrained 모델을 사용하여서 그런지, 학습 초기부터 쉽게 높은 validation 성능을 보였다.

5. 정량평가

Test dataset에 대해서 계산된 분류 정확도는 다음과 같다. Test dataset에 대한 성능을 계산하기 위해, 각각의 데이터별로 RandomResizedCrop만 적용하여 생성된 10개의 이미지에 대한 정확도를 평균내어 최종 결과로 기록하였다. 각 데이터를 10번씩 추론해서 계산한 Test 정확도는 **0.931** 또는 **93.1%**를 기록하였다. (1번씩 추출해서 계산한 결과도 Total Accuracy 값은 비슷했지만 각 클래스별 Accuracy 값은 데이터 수가 작아 아래 결과와 꽤 차이가 있었다.)

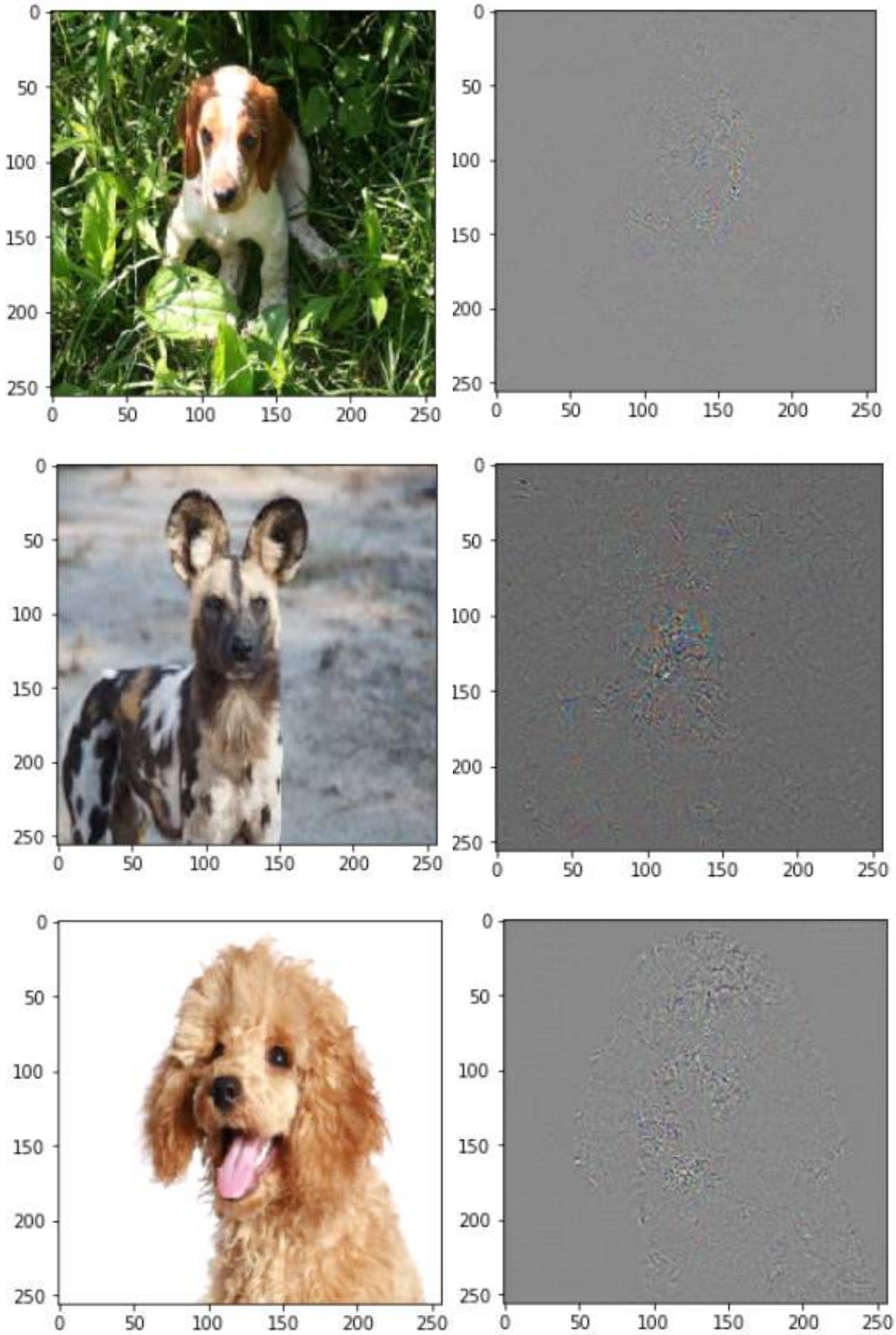
```
Total Accuracy: 0.931
Accuracy of Afghan_hound(n=36): 0.928
Accuracy of African_hunting_dog(n=32): 0.981
Accuracy of basenji(n=34): 0.921
Accuracy of black-and-tan_coonhound(n=33): 0.985
Accuracy of Blenheim_spaniel(n=36): 0.994
Accuracy of borzoi(n=26): 0.935
Accuracy of Chihuahua(n=30): 0.630
Accuracy of clumber(n=28): 0.964
Accuracy of dingo(n=30): 0.883
Accuracy of EntleBucher(n=40): 0.935
Accuracy of keeshond(n=32): 0.997
Accuracy of komondor(n=28): 0.943
Accuracy of Leonberg(n=33): 0.970
Accuracy of Maltese_dog(n=45): 0.911
Accuracy of Norwegian_elkhound(n=39): 0.949
Accuracy of otterhound(n=30): 0.930
Accuracy of Pomeranian(n=39): 0.949
Accuracy of pug(n=32): 0.969
Accuracy of Shih-Tzu(n=39): 0.938
Accuracy of Siberian_husky(n=36): 0.947
Accuracy of silky_terrier(n=38): 0.979
Accuracy of standard_schnauzer(n=32): 0.931
Accuracy of toy_poodle(n=26): 0.896
Accuracy of toy_terrier(n=34): 0.809
Accuracy of Welsh_springer_spaniel(n=31): 0.952
```

위 결과를 확인해보면, Chihuahua 클래스의 경우 30개의 test 데이터로 이루어져 있었고(n=30), 0.630 의 성능을 기록하여, 제일 낮은 성능을 보였다. 실제로 데이터를 확인해보니, 개의 색상과 생김새의 다양성이 다른 개 품종들에 비해 확연히 큰 추세를 보였다.



6. 정성평가

학습된 모델이 입력 이미지의 어떤 부분에 반응하는지 직관적으로 분석해보기 위해, 무작위 Test 이미지와 실제 라벨과 예측값과의 Cross Entropy Loss를 모델에 Backprop시켜 계산해 보았다(계산한 Gradient는 실수로라도 모델 업데이트에 사용하거나 한 바가 없습니다.) 모델의 추론과정을 시각화하는 데에 사용되는 Grad-CAM 방법과는 차이가 있는 방법이어서 의미가 있는지는 모르겠지만, 모델의 행동양상에 대한 통찰을 얻을 수 있었다. 출력된 모델을 보면 개의 얼굴 부분에 강한 반응(큰 gradient 값)을 보였고, 따라서 모델이 개의 얼굴 생김새를 통해 품종을 분류한다는 것을 간접적으로 알 수 있었다.



7. 부록

7.1 데이터 전처리 시 학습 과정에서 제외하기로 결정한 이미지 - 흑백사진

Afghan_hound_25.jpg

black-and-tan_coonhound_20.jpg

black-and-tan_coonhound_131.jpg

Shih-Tzu_189.jpg

Norwegian_elkhound_186.jpg

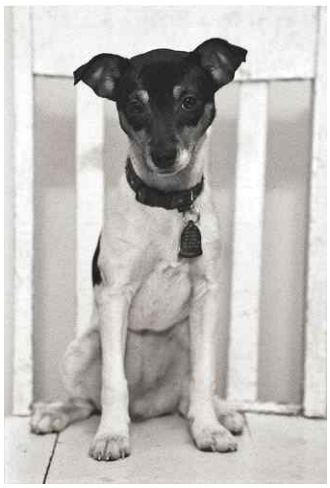
Norwegian_elkhound_177.jpg

toy_terrier_59.jpg

toy_terrier_61.jpg

toy_terrier_106.jpg

toy_terrier_116.jpg



toy_terrier_148.jpg

7.2 데이터 전처리 시 학습 과정에서 제외하기로 결정한 이미지 - 잘못 지정된 클래스

Chihuahua_81.jpg (새끼 허스키 또는 말라뮤트 종으로 추정됨)

새끼 말라뮤트 사진

(출처: https://www.notepet.co.kr/news/article/article_view/?groupCode=AB700AD710&idx=12496)

7.4 사용한 코드