지능형비전 최종 프로젝트 보고서

펭귄 종 분류 모델 성능 비교 분석

과목: 지능형 비전
담당교수: 박천수 교수님
학과: 컴퓨터교육과
팀원: 2018310077 유도현
2018310073 구재우
제출일: 2023. 06. 04

1. 주제 소개

수업시간에 배운 다양한 딥러닝 알고리즘을 활용하여 펭귄의 종을 분류하는 모델을 만들려고 했다. 펭귄의 종은 아델리 펭귄, 턱끈펭귄, 황제펭귄, 갈라파고스 펭귄, 젠투펭귄, 쇠푸른펭귄, 마카로니 펭귄, 마젤란 펭귄, 바위 뛰기 펭귄 9 종류로 선정했다. 펭귄의 종을 이렇게 설정한 이유는 펭귄의 대표적인 종들이기도 하며, 펭귄의 종을 하위 '속'으로 분류했을 때 각 '속'들에서 대표적인 종들이기 때문이다. 또한 각 9 종류들은 서로 간의 특징이 명확히 구분되는 편이어서 모델링 작업을 하는 데 더 수월할 것이라고 생각했다.

2. 데이터셋 구축

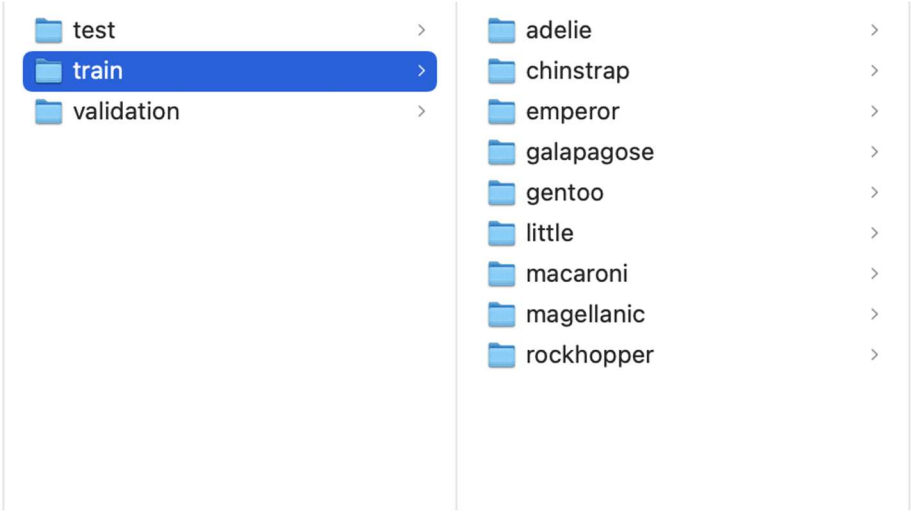
구글 이미지 크롤링을 통해 9 종류의 펭귄 데이터를 모음. 파이썬 selenium 라이브러리를 사용했다. 키워드를 입력 받아 해당 키워드를 chrome 이미지에서 검색하여 얻을 수 있는 모든 이미지들을 로컬 폴더에 저장하고, 동일한 url 의 이미지들은 중복 저장되지 않게 했다.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time
from urllib.request import (urlopen, urlparse, urlunparse, urlretrieve)
import urllib.request
import os
import pandas as pd


def selenium_scroll_option():
  SCROLL_PAUSE_SEC = 3

  # 스크롤 높이 가져옴
  last_height = driver.execute_script("return document.body.scrollHeight")

  while True:
    # 끝까지 스크롤 다운
    driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")

    # 3초 대기
    time.sleep(SCROLL_PAUSE_SEC)

    # 스크롤 다운 후 스크롤 높이 다시 가져옴
    new_height = driver.execute_script("return document.body.scrollHeight")

    if new_height == last_height:
        break
    last_height = new_height
```

로컬 폴더에 저장된 이미지들을 일일이 살펴보며 모델링을 하는 데 적합한 이미지들을 선별했다. 선별 기준은 다음과 같다.

1. 한 마리의 펭귄만 나온 사진
2. 워터마크가 최대한 없는 사진
3. 펭귄이 지나치게 작거나 크게 나오지 않은 사진
4. 화질이 뭉개지지 않은 사진

이와 같은 선별 기준을 가지고 9 종류의 펭귄 이미지들을 train, validation, test 세 종류로 나눴다.



세 종류로 나눈 이유는 모델을 훈련시킬 때 사용되는 데이터와 검증을 할 때 사용되는 데이터를 구분하기 위함이다. 두 데이터가 겹치지 않아야 모델이 학습을 제대로 할 수 있다.

이후에는 다양한 데이터들을 모델에 시험해보기 위해 데이터 전처리를 4 가지 종류로 해보았다.

Ⅰ. 데이터 resize 128*128

데이터를 원본의 크기로 처리하게 되면 훈련을 시키는 데 있어 통일성이 떨어지게 된다. 또한 데이터의 크기가 지나치게 크다면 연산량이 많아지기 때문에 모델을 훈련할 때 시간이 너무 오래 걸리게 된다. 이러한 이유로 데이터의 크기를 128*128 로 통일하여 모델의 계산 비용을 줄이고, 특징 추출을 강화하는데 도움을 주려고 했다.

Ⅱ. 데이터 resize 후, blur 처리

이미지에 존재하는 잡음의 영향을 제거하고, 거친 느낌의 이미지를 부드럽게 처리하기 위해 blur 처리를 해 보았다. OpenCV 의 Gaussian filter 를 활용하여 전처리를 했다.

Ⅲ. 데이터 resize 후, filter 처리

이미지의 중앙 픽셀을 강조하는 filter 를 적용하여 전처리를 했다.

```
Mat kernel = (Mat_<float>(3,3) << 0, -1, 0,
                                  -1, 5, -1,
                                   0, -1, 0);
```

이를 통해 중앙 영역의 픽셀들을 미세하게 강조하고 이미지의 선명도를 높이고자 한다. OpenCV 의 filter2D 함수를 활용했다.

Ⅳ. 데이터 resize 후, 샤프닝

언샤프 마스크 필터를 활용하여 이미지들을 전처리했다. 블러링이 적용되어 부드러워진 영상을 활용하여 날카로운 이미지를 만들었다. alpha 값을 곱한 원본의 이미지에서 Gaussian filter 를 적용한 이미지를 빼서 전처리를 수행했다. 이를 통해 이미지의 윤곽이 뚜렷하고 선명한 느낌이 나도록 했다.

```
Mat img, blurred;
GaussianBlur(img, blurred, Size(5,5), 1);

float alpha = 1.f;
Mat dst = (1+alpha)*img - alpha*blurred;
```

3. 적용 알고리즘 및 결과

- **KNN**

가장 먼저 기본적인 머신러닝 알고리즘인 KNN 을 사용하였다. KNN 모델은 sklearn 의 KNeighborsClassifier 라이브러리를 사용하였다.

```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
```

학습에 앞서 기본적으로 앞서 TensorFlow 로 저장된 데이터셋을 Numpy 배열로 변환하고 train, validation, test 할 이미지들은 1 차원으로 재구성하였다.

```
train_images = np.concatenate([x for x, y in train_dataset])
train_labels = np.concatenate([y for x, y in train_dataset])

val_images = np.concatenate([x for x, y in val_dataset])
val_labels = np.concatenate([y for x, y in val_dataset])

test_images = np.concatenate([x for x, y in test_dataset])
test_labels = np.concatenate([y for x, y in test_dataset])

train_images = train_images.reshape(train_images.shape[0], -1)
val_images = val_images.reshape(val_images.shape[0], -1)
test_images = test_images.reshape(test_images.shape[0], -1)
```

이 후 학습을 진행하고 검증 데이터셋과 테스트 데이터셋으로 성능을 평가했다.

```
knn = KNeighborsClassifier(n_neighbors=4, weights='distance',
metric='cosine')
knn.fit(train_images, train_labels)

val_score = knn.score(val_images, val_labels)
print(f'Validation accuracy: {val_score * 100:.2f}%')

test_score = knn.score(test_images, test_labels)
print(f'Test accuracy: {test_score * 100:.2f}%')
```

KNN 의 경우 크게 기대를 하지 않았지만 생각보다 정확도가 정말 낮았다. 처음 기본적으로 k 값을 3 으로 두고 모델을 학습시켰을 때의 결과는 다음과 같았다.

```
Validation accuracy: 11.44%
```

```
Test accuracy: 12.64%
```

9 개의 종을 분류하는 것이기 때문에 랜덤으로 분류를 한다고 해도 100/9 인 약 11%의 정확도가 나오는데 해당 결과는 이와 큰 차이가 있지 않았다.

따라서 정확도를 올리기 위해 다음과 같은 작업을 진행하였다.
이미지의 경우 기본이미지, blur 이미지, filtered 이미지, sharpen 이미지를 번갈아 가며 사용하였고, K 값을 1~11 까지로 조정해 보았다. batch 크기의 경우 KNN 의 성능에 직접적으로 영향을 미치지 않지만 32, 64 를 번갈아가며 두었다. 가중치의 경우 모두 동등한 uniform 방식과 거리에 따라 가중치를 두는 distance 방식을 각각 사용해보았고, 거리 측정 방법은 euclidean, manhattan, cosine 방식을 모두 사용하였다. 그러나 아래 보이는 결과와 같이 유의미한 변화는 없었다. 9% ~ 14%사이에서만 정확도가 왔다갔다 했으며 같은 모델을 돌려도 해당 범위내에서 결과가 나왔다. 아래 결과는 학습시킨 모델 중 몇개의 정확도를 측정한 것이다.

(기본이미지 사용, K 값:4, Batch 크기 64, 가중치: distance, 거리측정방법: cosine)
```
Validation accuracy: 12.06%
Test accuracy: 12.46%
Execution time: 3.957089424133301 seconds
```

(blur 이미지 사용, K 값:4, Batch 크기 64, 가중치:distance, 거리측정방법:cosine)
```
Validation accuracy: 11.38%
Test accuracy: 11.38%
Execution time: 4.649633169174194 seconds
```

(filtered 이미지 사용, K 값:4, Batch 크기 64, 가중치:distance, 거리측정방법:cosine)
```
Validation accuracy: 8.38%
Test accuracy: 11.08%
Execution time: 4.777194023132324 seconds
```

(filtered 이미지 사용, K 값:7, Batch 크기 64, 가중치:distance, 거리측정방법:cosine)
```
Validation accuracy: 10.48%
Test accuracy: 11.08%
Execution time: 4.688286304473877 seconds
```

(filtered 이미지 사용, K 값:7, Batch 크기 32, 가중치:uniform, 거리측정방법:euclidean)
```
Validation accuracy: 12.57%
Test accuracy: 10.48%
Execution time: 7.4572227001190186 seconds
```

(filtered 이미지 사용, K 값:1, Batch 크기 32, 가중치:uniform, 거리측정방법:manhattan)
```
Validation accuracy: 8.08%
Test accuracy: 8.38%
Execution time: 16.595858573913574 seconds
```

(filtered 이미지 사용, K 값:8, Batch 크기 32, 가중치:distance, 거리측정방법:manhattan)

```
Validation accuracy: 11.98%
Test accuracy: 12.28%
Execution time: 16.36469030380249 seconds
```

이 외에도 K 값과 가중치, 거리측정 방법 등을 바꾸어가며 실행해보았지만 비슷한 결과가 나왔다. 실행시간의 경우는 구글 colab 에서 진행한 것으로 colab 서버 상황에 영향을 많이 받기 때문에 유의미하게 분석하기는 어렵다고 판단하였다. 이로 KNN 은 이미지 분석에는 적합하지 않다는 것이 확인되었다.

- **SVM**

머신 러닝 알고리즘 중 하나인 서포트 벡터 머신(Support Vector Machine)을 활용하여 펭귄 이미지들을 학습시켰다. SVM 은 두 개의 클래스로 구성된 데이터를 가장 여유 있게 분리하는 초평면을 찾는 머신 러닝 알고리즘이다. SVM 알고리즘은 지도 학습의 일종이며, 분류와 회귀에 사용될 수 있기 때문에 펭귄을 분류하는 데도 적합한 알고리즘일 것이라고 생각했다.

먼저 데이터 폴더 내의 각 종에 해당하는 폴더를 순회하며 이미지 데이터와 레이블을 수집했다.

```
# 이미지 데이터와 레이블을 저장할 리스트
image_data = []
label_data = []

# 데이터 폴더 내의 각 폴더를 순회하며 이미지 데이터와 레이블 수집
for class_label, class_name in enumerate(os.listdir(data_dir)):
    class_dir = os.path.join(data_dir, class_name)
    if class_name == '.DS_Store':
        continue
    for image_name in os.listdir(class_dir):
        image_path = os.path.join(class_dir, image_name)
        if image_name == '.DS_Store':
            continue
        image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        image = cv2.resize(image, image_size)
        image_flattened = image.reshape(-1)

        image_data.append(image_flattened)
        label_data.append(class_label)
```

그 후에 이미지 데이터와 레이블을 np.arrary 로 바꿔 데이터를 저장했다.

```
image_data = np.array(image_data)
label_data = np.array(label_data)

# 데이터 저장
np.save('train_data.npy', image_data)
np.save('train_label.npy', label_data)
```

SVM 모델의 정확도가 얼마나 될지 시험해 볼 test 데이터 또한 위 방법과 마찬가지로 np.array 로 바꿔 데이터를 저장했다.

이렇게 저장한 데이터들을 SVM 모델에 넣어 시험해 본 결과이다.

```
Memory Usage: 48 bytes
Support Vectors:
[[214. 214. 214. ... 211. 211. 211.]
 [192. 192. 192. ... 183. 183. 184.]
 [232. 232. 233. ... 241. 240. 240.]
 ...
 [ 22.  22.  22. ...   5.  12.  18.]
 [243. 243. 242. ... 152. 149. 153.]
 [136. 137. 136. ...  83.  84.  86.]]
Coefficients:
[[ 1.74766497e-05  7.63814150e-06 -1.24526306e-05 ...  4.70269212e-06
   1.42486889e-05  1.33064756e-05]
 [-1.00779073e-05 -6.62143235e-06 -3.50696258e-06 ...  1.74721644e-05
   2.37420980e-05  2.24366857e-05]
 [ 3.89557612e-05  3.46156434e-05  2.67924843e-05 ...  1.03818178e-04
   9.01864501e-05  7.61971420e-05]
 ...
 [ 1.28285586e-05  1.55338605e-05  7.56032470e-06 ... -8.04254095e-05
  -9.84920388e-05 -1.08329841e-04]
 [ 1.45781762e-05  9.25830996e-06  8.79611835e-06 ...  1.38169710e-05
   2.62242345e-05  2.81780209e-05]
 [-3.35825358e-05 -3.40103515e-05 -1.71253058e-05 ...  1.53225080e-05
   3.99376469e-05  6.50817662e-05]]
Intercept:
[-1.86110189 -3.35355438 -2.79452274 -2.5462762  -0.0988689  -0.30576661
 -2.25747141 -2.21452515 -2.45888578 -1.9699883  -1.12188576  1.46774576
  1.832143   -0.50170797 -1.03720807  0.60923513  0.7642625   2.90714471
  3.30209241  1.37839331  0.99876664  0.17565331  2.66361681  2.8834883
  1.27036842  0.61040087  2.04111578  2.0164021   0.7951216   0.44376816
  0.0923827  -1.8832633  -2.5884886  -2.03858102 -1.97753444 -1.17535909]
Accuracy: 0.11077844311377245
```

메모리는 48byte 정도를 사용했고, 정확도는 11%에 불과했다. 다른 정보들은 SVM 모델의 서포트 벡터들, 계수들, 절편(intercept)에 해당하는 정보이다. SVM 모델의 정확도는 그냥 랜덤으로 분류했을 때의 수치인 11%와 거의 비슷했다.

- **CNN**

딥러닝 학습 알고리즘 중 하나인 cnn 을 활용하여 펭귄 이미지들을 학습시켰다. 프레임워크로는 tensorflow 를 활용했으며, 구글 colab 에서 작업했다. 앞서 blur, filter, sharpen 을 통해 전처리한 이미지들을 활용하여 cnn 모델을 학습했다. 처음에는 아무 전처리도 하지 않은 기본 128*128 크기의 이미지들을 활용하여 진행해 보았다.

이미지의 수가 많이 부족하기 때문에 데이터 증강을 진행했다. 데이터 증강은 이미지를 변형시켜 데이터셋을 인위적으로 확장시키는 기법이다. 회전, 이동, 확대/축소, 반전 등의 변환을 적용하여 다양한 형태의 이미지를 생성한다. 이를 통해 모델이 다양한 상황에서 학습할 수 있게 하는 방법이다. Tensorflow 의 'ImageDataGenerator'를 활용하여 수행했다.

```
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator

#Augmentation 적용
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range=25,
                                   width_shift_range=0.05,
                                   height_shift_range=0.05,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   vertical_flip=True,
                                   fill_mode='nearest'
                                   )

validation_datagen=ImageDataGenerator(rescale=1./255)
test_datagen=ImageDataGenerator(rescale=1./255)


train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=16,
                                                    color_mode='rgb',
                                                    class_mode='sparse',
                                                    target_size=(128,128)
                                                    )

validation_generator = validation_datagen.flow_from_directory(validation_dir,
                                                              batch_size=4,
                                                              color_mode='rgb',
                                                              class_mode='sparse',
                                                              target_size=(128,128)
                                                              )

test_generator = test_datagen.flow_from_directory(test_dir,
                                                  batch_size=4,
                                                  color_mode='rgb',
                                                  class_mode='sparse',
                                                  target_size=(128,128)
                                                  )
```

Train 데이터셋과 Validation 데이터셋, Test 데이터셋을 구분하여 generator 를 만들었다. Train 데이터셋에만 데이터 증강을 진행했다. 각 이미지마다 horizontal flip, vertical flip 이 적용되게 만들었다. 또한 픽셀의 값이 0 에서 1 사이의 값으로 나오도록 모든 픽셀 값들을 255 로 나눴다.

모델의 구성은 다음과 같이 했다.

```
#모델 구성

import tensorflow as tf

num_classes = len(penguin_classes)

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(128,128,3)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(num_classes)
])
#model.summary()

#모델 컴파일
from tensorflow.keras.optimizers import Adam

model.compile(optimizer=Adam(learning_rate=0.001),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy']
              )
#모델 훈련
history = model.fit_generator(train_generator,
                              validation_data=validation_generator,
                              steps_per_epoch=4,
                              epochs=300,
                              validation_steps=4,
                              verbose=2
                              )
```

결과는 다음과 같다.

```
model.evaluate(train_generator)
```

```
80/80 [==============================] - 22s 269ms/step - loss: 1.7467 - accuracy: 0.3690
[1.7466636896133423, 0.36900079250335693]
```

+ 코드 + 텍스트

```
model.evaluate(validation_generator)
```

```
79/79 [==============================] - 4s 46ms/step - loss: 1.9813 - accuracy: 0.2803
[1.9812575578689575, 0.2802547812461853]
```

생각보다 결과가 좋게 나오지 않아서 수정해야 할 필요가 있었다. 따라서 모델 구성을 다르게 해 보았다. CNN 모델의 구조는 합성곱 레이어, 완전 연결 레이어, 드롭아웃 레이어, 배치 정규화 이렇게 4 가지의 구성으로 되어있다. 처음 모델의 결과에는 드롭아웃 레이어가 추가되지 않은 상태였기 때문에 이를 추가해보았고, 모델을 훈련할 때 한 epoch 에 사용할 step 수를 더 늘려봤다.

- Dropout 추가한 결과

```
model.evaluate(train_generator)
```

```
85/85 [==============================] - 11s 127ms/step - loss: 1.4487 - accuracy: 0.4982
[1.4486836194992065, 0.4981604218482971]
```

```
model.evaluate(validation_generator)
```

```
85/85 [==============================] - 1s 16ms/step - loss: 2.0459 - accuracy: 0.3765
[2.0459330081939697, 0.3764705955982208]
```

- Steps_per_epoch 4->8 로 증가(Dropout 은 추가하지 않음)

```
[20] model.evaluate(train_generator)
```

```
85/85 [==============================] - 11s 128ms/step - loss: 0.9445 - accuracy: 0.6689
[0.9445359706878662, 0.6688741445541382]
```

```
model.evaluate(validation_generator)
```

```
85/85 [==============================] - 1s 17ms/step - loss: 1.4069 - accuracy: 0.5235
[1.4068524837493896, 0.5235294103622437]
```

성능이 꽤 개선된 것을 볼 수 있다. 특히 steps_per_epoch 을 증가시켰을 때 성능이 많이 개선됐다. 이것을 바탕으로 전처리한 이미지들을 학습시켰을 때 성능이 어떻게 나올지 실행해봤다.

- blur 처리 한 데이터셋을 훈련한 결과

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_15 (Conv2D)          (None, 126, 126, 16)      448

 max_pooling2d_15 (MaxPoolin  (None, 63, 63, 16)       0
 g2D)

 conv2d_16 (Conv2D)          (None, 61, 61, 32)        4640

 max_pooling2d_16 (MaxPoolin  (None, 30, 30, 32)       0
 g2D)

 conv2d_17 (Conv2D)          (None, 28, 28, 64)        18496

 max_pooling2d_17 (MaxPoolin  (None, 14, 14, 64)       0
 g2D)

 flatten_5 (Flatten)         (None, 12544)             0

 dense_9 (Dense)             (None, 512)               6423040

 dense_10 (Dense)            (None, 9)                 4617

=================================================================
Total params: 6,451,241
Trainable params: 6,451,241
Non-trainable params: 0
_________________________________________________________________
```

```
[25] model.evaluate(train_generator)

     83/83 [==============================] - 8s 99ms/step - loss: 0.9058 - accuracy: 0.6694
     [0.9058096408843994, 0.6694402694702148]
```

```
[21] model.evaluate(validation_generator)

     85/85 [==============================] - 1s 17ms/step - loss: 1.4069 - accuracy: 0.5235
     [1.4068524837493896, 0.5235294103622437]
```

이전과 비교했을 때 성능이 아주 미세하게 높아진 것을 알 수 있었다. 하지만 유의미한 차이는 아니다.

- filter 처리한 데이터셋을 훈련한 결과

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_18 (Conv2D)          (None, 126, 126, 16)      448

 max_pooling2d_18 (MaxPoolin  (None, 63, 63, 16)       0
 g2D)

 conv2d_19 (Conv2D)          (None, 61, 61, 32)        4640

 max_pooling2d_19 (MaxPoolin  (None, 30, 30, 32)       0
 g2D)

 conv2d_20 (Conv2D)          (None, 28, 28, 64)        18496

 max_pooling2d_20 (MaxPoolin  (None, 14, 14, 64)       0
 g2D)

 flatten_6 (Flatten)         (None, 12544)             0

 dense_11 (Dense)            (None, 512)               6423040

 dense_12 (Dense)            (None, 9)                 4617

=================================================================
Total params: 6,451,241
Trainable params: 6,451,241
Non-trainable params: 0
```

```
[31] model.evaluate(train_generator)

     83/83 [==============================] - 10s 119ms/step - loss: 1.0145 - accuracy: 0.6419
     [1.0144851207733154, 0.6419379115104675]
```

```
[32] model.evaluate(validation_generator)

     84/84 [==============================] - 1s 13ms/step - loss: 2.4191 - accuracy: 0.4581
     [2.4190666675567627, 0.458083838224411]
```

전처리 하기 전의 결과와 비교했을 때 성능이 오히려 떨어진 것을 알 수 있었다. 또한 train generator 의 정확도 수치와 validation generator 의 정확도 수치가 전보다 많이 차이나는 것을 확인할 수 있다. 이는 과적합의 징후라고 할 수 있다. 과적합은 모델이 학습 데이터의 노이즈나 이상치를 과도하게 학습할 때 생기는 현상이다. 이를 막기 위해 Dropout 을 하거나 데이터셋을 늘리는 방법이 있다. blur 처리한 것과 비교했을 때도 성능이 떨어졌다. filter2D 를 활용한 것보다 Gaussian filter 처리한 것이 정확도가 조금 더 높은 것을 알 수 있었다.

- Sharpen 데이터셋을 훈련한 결과

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_21 (Conv2D)          (None, 126, 126, 16)      448

 max_pooling2d_21 (MaxPoolin  (None, 63, 63, 16)       0
 g2D)

 conv2d_22 (Conv2D)          (None, 61, 61, 32)        4640

 max_pooling2d_22 (MaxPoolin  (None, 30, 30, 32)       0
 g2D)

 conv2d_23 (Conv2D)          (None, 28, 28, 64)        18496

 max_pooling2d_23 (MaxPoolin  (None, 14, 14, 64)       0
 g2D)

 flatten_7 (Flatten)         (None, 12544)             0

 dense_13 (Dense)            (None, 512)               6423040

 dense_14 (Dense)            (None, 9)                 4617

=================================================================
Total params: 6,451,241
Trainable params: 6,451,241
Non-trainable params: 0
_________________________________________________________________
```

```
model.evaluate(train_generator)
```

```
83/83 [==============================] - 10s 114ms/step - loss: 0.8086 - accuracy: 0.7222
[0.8085974454879761, 0.722180187702179]
```

```
model.evaluate(validation_generator)
```

```
84/84 [==============================] - 1s 12ms/step - loss: 1.9522 - accuracy: 0.5240
[1.952231764793396, 0.523952066898346]
```

blur 처리, filter 처리와 비교했을 때 성능이 크게 개선된 것을 확인할 수 있었다. 하지만 validation generator 의 정확도는 모든 경우에서 비슷한 성능을 보였다. 또한 sharpening 처리한 데이터셋도 train generator 의 정확도 수치와 validation generator 의 정확도 수치가 20% 가까이 차이나는 것을 확인할 수 있다. 이것은 과적합의 징후라고 할 수 있다.

결과적으로 데이터셋을 각각 blur 처리, filter 처리, sharpening 처리한 것과 원본 데이터셋을 포함하여 비교해봤을 때, sharpening 처리한 것이 가장 성능이 좋은 것을 확인할 수 있었다. 이를 통해 펭귄 이미지의 엣지 부분을 강조했을 때 정확도가 가장 높게 나온 것을 알 수 있었다. 하지만 시험한 모든 데이터셋은 validation generator 를 활용하여 정확도를 측정한 결과가 비슷하게 나온 것도 알 수 있었다. 이를 통해 모델이 새로운 데이터를 시험했을 때 정확도가 학습할 때의 정확도보다 떨어지는 것을 알 수 있었다. 마지막으로 알 수 있던 점은 train generator 와 validation generator 의 성능 차이가 많이 난 것이었다.

- RESNET50

ResNet 50 의 경우 tensorflow 의 ResNet50 모델을 가져와서 학습시켰다. batch 크기 32, epochs 수 100 으로 두고 실행시킨 경우 결과는 다음과 같다.

```
Layer (type)                 Output Shape              Param #
=================================================================
resnet50 (Functional)        (None, 2048)              23587712

flatten_1 (Flatten)          (None, 2048)              0

dense_1 (Dense)              (None, 9)                 18441

=================================================================
Total params: 23,606,153
Trainable params: 23,553,033
Non-trainable params: 53,120
_________________________________________________________________
11/11 - 1s - loss: 0.7850 - accuracy: 0.8018 - 1s/epoch -
118ms/step
Validation accuracy: 80.18%
11/11 - 1s - loss: 0.7880 - accuracy: 0.8108 - 1s/epoch -
112ms/step
Test accuracy: 81.08%
```

같은 조건에서의 성능이 ResNet50 보다 아래 ResNet101 이 좋았기 때문에 ResNet50 으로의 학습은 더 이상 진행하지 않았다. ResNet 에 대한 코드와 성능을 올리기 위한 구조 변화는 아래 ResNet101 을 참고하면 된다.

- RESNET101

ResNet101 의 경우 마찬가지로 tensorflow 의 ResNet101 모델을 가져와서 학습시켰다.
이미지 분류에 효과적이기 때문에 좋은 성능을 기대했다.

```
from tensorflow.keras.applications import ResNet101
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
```

ResNet 의 경우 224x224 로 이미지를 받기 때문에 해당 코드에 앞서 이미지를 Resize 하고 배치크기를 설정하여 데이터셋을 구성하였다.

```
img_height = 224  # 이미지 높이 설정
img_width = 224  # 이미지 너비 설정
batch_size = 32  # 배치 사이즈 설정

train_dataset = tf.keras.preprocessing.image_dataset_from_directory(
 train_dir,
 image_size=(img_height, img_width),
 batch_size=batch_size)

val_dataset = tf.keras.preprocessing.image_dataset_from_directory(
 validation_dir,
 image_size=(img_height, img_width),
 batch_size=batch_size)

test_dataset = tf.keras.preprocessing.image_dataset_from_directory(
 test_dir,
 image_size=(img_height, img_width),
 batch_size=batch_size)
```

이후 ResNet101 에 맞게 이미지를 전처리하고 클래스의 수를 설정하였다.

```
def format_image(image, label):
   image = tf.cast(image, tf.float32)  # 이미지를 float32 타입으로 변환
   image = (image/127.5) - 1  # 이미지의 픽셀 값을 [-1, 1] 범위로 조정
   return image, label  # 전처리된 이미지와 라벨을 반환
num_classes = 9

train_dataset = train_dataset.map(format_image)
val_dataset = val_dataset.map(format_image)
test_dataset = test_dataset.map(format_image)
```

다음으로 모델 구조를 정의하였다. Flatten 레이어로 이미지를 1 차원 벡터로 평탄화 하였고, Dense 레이어를 통해 최종적으로 각 클래스에 대해 계산할 수 있도록 하였다.

```
model = Sequential([
 ResNet101(include_top=False, weights='imagenet', pooling='avg',
input_shape=(img_height, img_width, 3)),
 Flatten(),
 Dense(num_classes, activation='softmax'), ])
```

다음으로 모델을 컴파일 하고 학습을 진행했다.

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

history = model.fit(train_dataset, validation_data=val_dataset,
epochs=100)
```

학습 완료 후에는 모델의 정보를 출력하고 validation 데이터셋과 테스트 데이터셋으로 모델을 평가했다.

```
model.save('resnet101_model')
model.summary()

val_loss, val_acc = model.evaluate(val_dataset, verbose=2)
print(f'Validation accuracy: {val_acc * 100:.2f}%')

test_loss, test_acc = model.evaluate(test_dataset, verbose=2)
print(f'Test accuracy: {test_acc * 100:.2f}%')
```

Blur 처리된 이미지를 Batch 사이즈 32 로 100 의 Epochs 수로 학습시킨 결과는 다음과 같았다.

```
      Layer (type)                 Output Shape              Param #
=================================================================
 resnet101 (Functional)      (None, 2048)              42658176

 flatten_7 (Flatten)         (None, 2048)              0

 dense_7 (Dense)             (None, 9)                 18441

=================================================================
Total params: 42,676,617
Trainable params: 42,571,273
Non-trainable params: 105,344
_________________________________________________________________
11/11 - 0s - loss: 0.7900 - accuracy: 0.8234 - 441ms/epoch - 40ms/step
Validation accuracy: 82.34%
11/11 - 0s - loss: 0.8877 - accuracy: 0.7934 - 441ms/epoch - 40ms/step
Test accuracy: 79.34%
Model size: 9.582812309265137 MB
```

우선 parms 수를 보면 Resnet101 이 Resnet50 보다 2 배 가까이 크므로 모델 복잡도도 2 배가까이 큰 것을 볼 수 있다. 정확도의 경우 80%가까이 나오며 앞서 KNN, CNN, SVM 에 비해 상당히 높음을 확인할 수 있다. 모델이 매우 복잡하므로 학습에 걸리는 시간도 앞선 알고리즘보다 확실히 오래 걸렸다. 100 epoch 를 돌리는데 20~30 분가량 시간이 필요했다.

모델의 성능을 높이기 위해 먼저 이미지를 기본 이미지, blur 이미지, filtered 이미지, sharpen 이미지 이렇게 4 개를 사용해보았다. 그리고 batch size 를 16, 32, 64 로 변화해가며 학습시켜보았고, epoch 수는 100, 200, 300, 500 으로 학습을 진행해보았다. 그 결과는 다음과 같다.

(filtered 이미지, Batch Size : 64, Epochs : 300)

```
6/6 - 1s - loss: 0.8120 - accuracy: 0.8293 - 501ms/epoch - 83ms/step
Validation accuracy: 82.93%
6/6 - 10s - loss: 0.7694 - accuracy: 0.8263 - 10s/epoch - 2s/step
Test accuracy: 82.63%
```

(sharpen 이미지, Batch Size : 64, Epochs : 100)

```
6/6 - 0s - loss: 1.9814 - accuracy: 0.6287 - 468ms/epoch - 78ms/step
Validation accuracy: 62.87%
6/6 - 0s - loss: 1.9718 - accuracy: 0.6647 - 466ms/epoch - 78ms/step
Test accuracy: 66.47%
Model size: 9.56942081451416 MB
```

(filtered 이미지, Batch Size : 32, Epochs : 300)

```
11/11 - 0s - loss: 0.7316 - accuracy: 0.8503 - 477ms/epoch - 43ms/step
Validation accuracy: 85.03%
11/11 - 0s - loss: 0.8440 - accuracy: 0.8144 - 458ms/epoch - 42ms/step
Test accuracy: 81.44%
```

(blur 이미지, Batch Size : 64, Epochs : 500)

```
6/6 - 1s - loss: 0.9305 - accuracy: 0.8024 - 502ms/epoch - 84ms/step
Validation accuracy: 80.24%
6/6 - 12s - loss: 1.0120 - accuracy: 0.7725 - 12s/epoch - 2s/step
Test accuracy: 77.25%
```

해당 결과를 보면 오히려 epochs 수를 500 까지 늘렸을 때 과적합으로 인해 성능이 낮게 나온 것을 확인할 수 있었다. 이는 데이터셋의 크기가 작아서 과적합이 빨리 왔다고 예상되었다.

따라서 이미지 증강을 통해 다양한 이미지를 학습할 수 있도록 하였다.
다음과 같이 증강을 하고 훈련데이터셋에 이미지 증강을 적용하고 학습할 수 있도록 하였다..

```
datagen = ImageDataGenerator(
    rotation_range=90,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    shear_range=0.2,
    horizontal_flip=True,
    rescale=1./255,
    fill_mode='nearest')

train_generator = datagen.flow_from_directory(
    train_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical')

val_datagen = ImageDataGenerator(rescale=1./255)
val_generator = val_datagen.flow_from_directory(
    validation_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical')

test_datagen = ImageDataGenerator(rescale=1./255)
```

```
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical')
```

```
train_num = train_generator.samples
val_num = val_generator.samples
test_num = test_generator.samples

history = model.fit(train_generator,
                    validation_data=val_generator,
                    steps_per_epoch=train_num // batch_size,
                    validation_steps=val_num // batch_size,
                    epochs=500)
```

그런데 오히려 성능이 떨어졌다.

(blur 이미지, Batch Size : 32, Epochs 수: 100 + 데이터증강)

```
10/10 - 1s - loss: 1.0194 - accuracy: 0.7969 - 844ms/epoch - 84ms/step
Validation accuracy: 79.69%
10/10 - 1s - loss: 1.7610 - accuracy: 0.7844 - 772ms/epoch - 77ms/step
Test accuracy: 78.44%
```

(filter 이미지, Batch Size : 32, Epochs 수: 300 + 데이터증강)

```
10/10 - 1s - loss: 1.3568 - accuracy: 0.7312 - 918ms/epoch - 92ms/step
Validation accuracy: 73.12%
10/10 - 1s - loss: 1.2988 - accuracy: 0.7594 - 874ms/epoch - 87ms/step
Test accuracy: 75.94%
```

다음으로는 학습률 스케줄링을 추가하였다.

```
def lr_schedule(epoch, lr):
    if epoch > 0 and epoch % 10 == 0:          return lr * 0.97
    return lr
callbacks = [LearningRateScheduler(lr_schedule)]
```

(Blur 이미지, Batch Size: 32, Epochs : 500 + 데이터 증강, 학습률 스케줄링 10 번마다 0.97 로)

```
10/10 - 1s - loss: 0.7941 - accuracy: 0.8594 - 820ms/epoch - 82ms/step
Validation accuracy: 85.94%
10/10 - 1s - loss: 0.8669 - accuracy: 0.8531 - 779ms/epoch - 78ms/step
Test accuracy: 85.31%
```

해당 요건으로 모델을 학습시켰을 때 85 퍼센트가 넘는 정확도로 가장 정확함을 확인할 수 있었다. 다만 데이터 증강 없이 학습률 스케줄링만 실행한 경우 오히려 결과가 더 떨어졌다.

이외에도 추가로 작업한 내용은 먼저 optimizer 를 기본 adam 에서 RMSprop, SGD 를 사용해 보았다. 기본 adam 에서 가장 좋은 성능을 보여 모든 작업을 adam 을 사용하였다. 다음으로는 Early Stopping 을 통해 성능 향상이 되지 않을 경우 훈련을 중단하는 방법을 사용했는데, 제대로 학습이 되기도 전에 종료가 되어 효과가 없었다. 또 과적합을 막기 위해 Dropout 레이어를 사용하여 무작위로 절반의 노드를 끄는 방식도 사용해 보았는데 이 또한 제대로 모델이 학습이 되지 않았다.

4. 결론