**Coding팀과제 4**

목차

1. 가위바위보 데이터셋 이미지 인식
2. 실험 결과

팀원

20140269 고혁훈

20160768 김홍엽

20160886 이규복

20180623 조성은



* **과제의 목적**

1. CNN모델을 MNIST 외에 컬러 이미지(가위바위보 dataset)에 적용해본다.

2. 기본 CNN 모델을 base로 CNN 레이어의 개수가 미치는 영향을 확인하다.

3. 기본 CNN 모델을 base로 정규화의 효과성을 검증한다.

4. 기본 CNN 모델을 base로 regularize의 효과성을 검증한다.

5. 기본 CNN 모델을 base로 RELU의 효과성을 검증해본다.

6. 기본 CNN 모델 대신에 다른 형태의 CNN을 실험해 본다.

* **배운 점**

**첫 번째로**, CNN layers를 계속 증가시킨다면 모델 성능의 marginal difference는 diminishing하기 때문에 CNN layers의 디자인이 model의 data에 적합하다는 가정 아래, 일정 개수까지는 layers의 개수가 증가할수록 모델의 성능이 높아진다는 결론을 얻을 수 있었다.

**두 번째로**, dropout을 사용해 보았을 때 overfitting 문제를 어느 정도 해결할 수 있었다. 그러나 dropout rate를 너무 크게 잡으면 오히려 성능에 악영향을 끼치는 것을 보아 적절한 값을 잘 선택해야한다. 흥미로운 점은 batch normalization을 적용했을 때 성능 개선이 이루어지지 않았다. 오버피팅 문제를 해결하는 방법으로 사용되는 batch normalization이 오히려 학습을 방해한 원인에 대해서는 추가적인 연구가 필요해 보이다. 대신 학습 수렴 속도가 월등히 빨라짐을 알 수 있었다.

**세 번째로**, CNN 모델을 구성하고 튜닝하는 과정이 상당히 까다로웠다. 파라미터나 아키텍쳐가 약간만 바뀌어도 성능이 크게 저하되었다. 그리고 학습과정에서 모델의 성능을 판단하는데 있어서는 역시 검증용 데이터(validation set)에 대한 loss나 정확도를 기반으로 판단하는 것이 좋다. 우리가 사용한 Image\_data\_generator를 활용한 data augmentation은 좋은 전처리 방법이다.

더불어, 이미지 데이터의 size나 커널의 size의 차이도 결과적으론 큰 차이가 날 수 있다는 것을 다시금 깨달았다.

**네 번째로**, cnn에서 중간 layer에서 sigmoid를 하나라도 쓰면 학습이 전혀 안된다는 것을 배웠고, depthwise-separable convolution을 도입하면 성능이 개선되는 것을 배웠다. 또한 group convolution을 적용할 때는 모든 conv에 도입하지 말고 일정 부분에만 도입해야 학습이 됨을 알 수 있었다.

* **데이터셋 다운**
* !wget --no-check-certificate \
* https://storage.googleapis.com/laurencemoroney-blog.appspot.com/rps.zip \
* -O /data/rps.zip
* !wget --no-check-certificate \
* https://storage.googleapis.com/laurencemoroney-blog.appspot.com/rps-test-set.zip \
* -O /data/rps-test-set.zip
* local\_zip = './data/rps.zip'
* zip\_ref = zipfile.ZipFile(local\_zip, 'r')
* zip\_ref.extractall('./data/')
* zip\_ref.close()
* local\_zip = './data/rps-test-set.zip'
* zip\_ref = zipfile.ZipFile(local\_zip, 'r')
* zip\_ref.extractall('./data/')
* zip\_ref.close()
* !mv "./data/rps" "./data/train"
* !mv "./data/rps-test-set" "./data/test"
* **데이터셋 구축 및 전처리**

1) import library

import tensorflow as tf

import numpy as np

import pandas as pd

import os

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

from tensorflow.keras.models import Sequential

from tensorflow.keras.preprocessing.image import load\_img

from tensorflow.keras.preprocessing.image import img\_to\_array

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.models import load\_model

from tensorflow.keras.callbacks import ModelCheckpoint

from tensorflow.keras.layers import Dense, Flatten,Conv1D, Conv2D, MaxPooling2D, Dropout, BatchNormalization, Softmax, concatenate

from tensorflow.keras.optimizers import RMSprop

from tensorflow import keras

2) 데이터 Augmentation & Load & preprocess

TRAINING\_DIR = "./data/train/"

training\_datagen = ImageDataGenerator(

      rescale = 1./255, #normalize

# augmentation 기법

        rotation\_range=40,

      width\_shift\_range=0.2,

      height\_shift\_range=0.2,

      shear\_range=0.2,

      zoom\_range=0.2,

      horizontal\_flip=True,

      fill\_mode='nearest',

      validation\_split = 0.2) # validation split ratio

# test는 augmentation 불필요

TEST\_DIR = "./data/test/"

TEST\_datagen = ImageDataGenerator(rescale = 1./255)

# train

train\_generator = training\_datagen.flow\_from\_directory(

    TRAINING\_DIR,

    target\_size=(300,300),

    class\_mode='categorical',

  batch\_size=32,

  subset = 'training'

)

# validation

validation\_generator = training\_datagen.flow\_from\_directory(

    TRAINING\_DIR,

    target\_size=(300,300),

    class\_mode='categorical',

  batch\_size=32,

  subset = 'validation'

)

# test

test\_generator = TEST\_datagen.flow\_from\_directory(

    TEST\_DIR,

    target\_size=(300,300),

    class\_mode='categorical',

  batch\_size=32

)

3) model compile 및 학습

model.compile(optimizer='adam',loss='categorical\_crossentropy', metrics=['accuracy'])

history = model.fit(train\_generator, epochs=epoch, validation\_data = validation\_generator, verbose = 1,validation\_steps=8,callbacks=[checkpoint])

4) 시각화

history\_dic = history.history

loss = history\_dic['loss']

val\_loss = history\_dic['val\_loss']

train\_acc = history\_dic['accuracy']

val\_acc = history\_dic['val\_accuracy']

epochs = range(1, len(loss)+1)

plt.plot(epochs, loss, 'bo', label = 'Training loss')

plt.plot(epochs, val\_loss, 'b', label = 'validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

plt.plot(epochs, train\_acc, 'bo', label = 'Training acc')

plt.plot(epochs, val\_acc, 'b', label = 'validation acc')

plt.title('Training and validation acc')

plt.xlabel('Epochs')

plt.ylabel('Acc')

plt.legend()

plt.show()

* **CNN 레이어 개수 별 성능 실험**

1) 모델 생성 및 summary

- 모델 생성

#sequential\_1; 모델을 정의한다. Convoution과 MaxPooling은 총 4번 사용하였다. 컬러이미지이기 때문에 input\_shape를 (300,300,3)으로 나타내었고 input layer에서 데이터를 flatten하고 dropout을 0.5로 설정해 overfitting을 방지하려고 하였다. Dropout(n%)은 자의적으로 layer에 있는 노드의 n%를 제외시켜 neighbor node가 영향력을 행사하는 것을 막기 위함이고 이를 통해 overfitting을 방지할 수 있다. Hidden layer은 512개이고, output layer는 총 class가 3개 이기 때문에 3으로 설정, activation 은 sigmoid가 아닌 softmax로 설정하였다.

model = tf.keras.models.Sequential([

tf.keras.layers.Conv2D(64, (3,3), activation='relu', input\_shape=(300, 300, 3)),

tf.keras.layers.MaxPooling2D(2, 2),

tf.keras.layers.Conv2D(64, (3,3), activation='relu'),

tf.keras.layers.MaxPooling2D(2,2),

tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

tf.keras.layers.MaxPooling2D(2,2),

tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

tf.keras.layers.MaxPooling2D(2,2),

tf.keras.layers.Flatten(),

tf.keras.layers.Dropout(0.5),

tf.keras.layers.Dense(512, activation='relu'),

tf.keras.layers.Dense(3, activation='softmax')

])

model.summary()

model.compile(loss = 'categorical\_crossentropy', optimizer='rmsprop', metrics=['accuracy'])

#위에서 train data batch size를 126으로 주었고 총 데이터 개수가 2520개이기 때문에 steps\_per\_epoch은 20개, validation data batch size를 126으로 주었고 총 데이터 개수가 372개이기 때문에 validation\_steps는 3으로 설정하였다.

history = model.fit(train\_generator, epochs=25, steps\_per\_epoch=20, validation\_data = validation\_generator, verbose = 1, validation\_steps=3)

#모델 결과를 그래프로 표현하기 위해 matplotlib을 불러들였다.

import matplotlib.pyplot as plt

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'r', label='Training accuracy')

plt.plot(epochs, val\_acc, 'b', label='Validation accuracy')

plt.title('Training and validation accuracy')

plt.legend(loc=0)

plt.figure()

plt.show()

#sequential\_2; Convolution과 MaxPooling 조합 개수에 따른 모델의 성능을 평가해 보고자 이번에는 Convolution과 MaxPooling의 조합을 두 개만 사용하였다. Compile과 model.fit코드, 결과 출력 코드는 위와 동일하다.

model = tf.keras.models.Sequential([

tf.keras.layers.Conv2D(64, (3,3), activation='relu', input\_shape=(300, 300, 3)),

tf.keras.layers.MaxPooling2D(2, 2),

tf.keras.layers.Conv2D(64, (3,3), activation='relu'),

tf.keras.layers.MaxPooling2D(2,2),

tf.keras.layers.Flatten(),

tf.keras.layers.Dropout(0.5),

tf.keras.layers.Dense(512, activation='relu'),

tf.keras.layers.Dense(3, activation='softmax')

])

model.summary()

model.compile(loss = 'categorical\_crossentropy', optimizer='rmsprop', metrics=['accuracy'])

history = model.fit(train\_generator, epochs=25, steps\_per\_epoch=20, validation\_data = validation\_generator, verbose = 1, validation\_steps=3)

import matplotlib.pyplot as plt

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'r', label='Training accuracy')

plt.plot(epochs, val\_acc, 'b', label='Validation accuracy')

plt.title('Training and validation accuracy')

plt.legend(loc=0)

plt.figure()

plt.show()

- 모델 summary

* Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d (Conv2D) (None, 298, 298, 64) 1792

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d (MaxPooling2D) (None, 149, 149, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_1 (Conv2D) (None, 147, 147, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 73, 73, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 71, 71, 128) 73856

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2 (None, 35, 35, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 33, 33, 128) 147584

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_3 (MaxPooling2 (None, 16, 16, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 32768) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 32768) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 512) 16777728

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 3) 1539

=================================================================

Total params: 17,039,427

Trainable params: 17,039,427

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Model: "sequential\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_6 (Conv2D) (None, 298, 298, 64) 1792

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_6 (MaxPooling2 (None, 149, 149, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_7 (Conv2D) (None, 147, 147, 64) 36928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_7 (MaxPooling2 (None, 73, 73, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_2 (Flatten) (None, 341056) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 341056) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_4 (Dense) (None, 512) 174621184

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_5 (Dense) (None, 3) 1539

=================================================================

Total params: 174,661,443

Trainable params: 174,661,443

Non-trainable params: 0

2) epoch결과 일부

* **convolution, maxpooling 4개 사용**

Epoch 15/25

20/20 [==============================] - 50s 3s/step - loss: 0.1937 - accuracy: 0.9329 - val\_loss: 0.0536 - val\_accuracy: 0.9812

Epoch 16/25

20/20 [==============================] - 50s 3s/step - loss: 0.2151 - accuracy: 0.9190 - val\_loss: 0.1124 - val\_accuracy: 0.9946

Epoch 17/25

20/20 [==============================] - 50s 2s/step - loss: 0.1913 - accuracy: 0.9258 - val\_loss: 0.0388 - val\_accuracy: 1.0000

Epoch 18/25

20/20 [==============================] - 51s 3s/step - loss: 0.1880 - accuracy: 0.9345 - val\_loss: 0.0676 - val\_accuracy: 0.9704

Epoch 19/25

20/20 [==============================] - 51s 3s/step - loss: 0.2537 - accuracy: 0.9067 - val\_loss: 0.2325 - val\_accuracy: 0.9489

Epoch 20/25

20/20 [==============================] - 51s 3s/step - loss: 0.1242 - accuracy: 0.9567 - val\_loss: 0.0565 - val\_accuracy: 0.9731

Epoch 21/25

20/20 [==============================] - 52s 3s/step - loss: 0.1175 - accuracy: 0.9591 - val\_loss: 0.0341 - val\_accuracy: 0.9866

Epoch 22/25

20/20 [==============================] - 51s 3s/step - loss: 0.1469 - accuracy: 0.9480 - val\_loss: 0.0788 - val\_accuracy: 0.9624

Epoch 23/25

20/20 [==============================] - 52s 3s/step - loss: 0.1172 - accuracy: 0.9583 - val\_loss: 0.1867 - val\_accuracy: 0.9194

Epoch 24/25

20/20 [==============================] - 52s 3s/step - loss: 0.0904 - accuracy: 0.9702 - val\_loss: 0.2306 - val\_accuracy: 0.9167

Epoch 25/25

20/20 [==============================] - 51s 3s/step - loss: 0.0675 - accuracy: 0.9750 - val\_loss: 0.2006 - val\_accuracy: 0.9140

* **convolution, maxpooling 2개 사용**

Epoch 15/25

20/20 [==============================] - 58s 3s/step - loss: 0.2329 - accuracy: 0.9190 - val\_loss: 0.2387 - val\_accuracy: 0.8952

Epoch 16/25

20/20 [==============================] - 58s 3s/step - loss: 0.4377 - accuracy: 0.8540 - val\_loss: 0.1561 - val\_accuracy: 0.9489

Epoch 17/25

20/20 [==============================] - 58s 3s/step - loss: 0.9017 - accuracy: 0.8468 - val\_loss: 0.2138 - val\_accuracy: 0.9328

Epoch 18/25

20/20 [==============================] - 58s 3s/step - loss: 0.1698 - accuracy: 0.9504 - val\_loss: 0.1483 - val\_accuracy: 0.9435

Epoch 19/25

20/20 [==============================] - 58s 3s/step - loss: 0.3351 - accuracy: 0.8929 - val\_loss: 0.2070 - val\_accuracy: 0.9355

Epoch 20/25

20/20 [==============================] - 58s 3s/step - loss: 0.2361 - accuracy: 0.9111 - val\_loss: 0.3596 - val\_accuracy: 0.8522

Epoch 21/25

20/20 [==============================] - 58s 3s/step - loss: 0.8322 - accuracy: 0.8563 - val\_loss: 0.2682 - val\_accuracy: 0.8871

Epoch 22/25

20/20 [==============================] - 58s 3s/step - loss: 0.1966 - accuracy: 0.9448 - val\_loss: 0.2698 - val\_accuracy: 0.8925

Epoch 23/25

20/20 [==============================] - 59s 3s/step - loss: 1.2317 - accuracy: 0.8556 - val\_loss: 0.2628 - val\_accuracy: 0.9005

Epoch 24/25

20/20 [==============================] - 58s 3s/step - loss: 0.1612 - accuracy: 0.9528 - val\_loss: 0.1754 - val\_accuracy: 0.9167

Epoch 25/25

20/20 [==============================] - 58s 3s/step - loss: 1.4986 - accuracy: 0.8274 - val\_loss: 0.2984 - val\_accuracy: 0.8925

3) fit 결과 도표(x축은 epoch)

* **convolution, maxpooling 4개 사용**

Chart, line chart

Description automatically generated

* **convolution, maxpooling 2개 사용**

Chart

Description automatically generated

4) 결과 및 insight 요약

Convolution과 MaxPooling의 조합을 4개로 했을 경우, 최종 결과 값이 accuracy=0.9750, val\_accuracy=0.9140으로 overfitting이 일어나지 않고 비교적 좋은 결과가 나왔음을 알 수 있었다. 반면, 그 조합이 2개인 경우, 최종 결과 값은 accuracy=0.8274 , val\_accuracy=0.8925로 4개인 경우와 비교했을 때, 모델의 성능이 더 낮게 관측되었다. 또한, 그래프에서도 볼 수 있듯이, 4개의 조합을 이용한 경우 training과 validation data가 우상향의 형상을 그리며 epoch이 높아질 수록 accuracy가 높아지는 것을 확인할 수 있었다. 이와 달리, 2개의 조합을 이용한 경우, training과 validation data 모두 fluctuation이 심하고 최종 epoch이 끝난 이후에도 두 데이터의 accuracy가 수렴하는 형상이 아님을 확인하였다.

이를 통해, CNN이 ‘특정하고 다른 것과 구별이 가게 해주는 디테일들에 집중하기 위해 이미지의 특징을 좁혀가는 과정에 기여한다’는 특징이 옳음을 확인할 수 있었다. 따라서, CNN layers를 계속 증가시킨다면 모델 성능의 marginal difference는 diminishing하기 때문에 CNN layers를 몇 개로 정할 것인지는 모델 학습 시간과 비용 등을 고려해 선택 해야겠지만 CNN layers의 디자인이 model의 data에 적합하다는 가정 아래, 일정 개수까지는 layers의 개수가 증가할수록 모델의 성능이 높아진다는 결론을 얻을 수 있었다.

* **Weight decay 실험**

1) 모델 생성 및 summary

1-1. model1 : 기본 모델(정규화 x)

model1 = Sequential([

    tf.keras.layers.Conv2D(64, (3,3), activation='relu', input\_shape=(300, 300, 3)),

    tf.keras.layers.MaxPooling2D(2, 2),

    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),

    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dropout(0.5),

    tf.keras.layers.Dense(512, activation='relu'),

    tf.keras.layers.Dense(3, activation='softmax')

])

1-2. model2 : l1 regularizer추가

model1과 아키텍쳐는 동일하며, regularizer를 다음과 같이 파라미터로 추가했다.

tf.keras.layers.Conv2D(64, (3,3), activation='relu', input\_shape=(300, 300, 3),

                           kernel\_regularizer=regularizers.l1(l1=1e-5))

tf.keras.layers.Dense(512, activation='relu',

                          bias\_regularizer=regularizers.l1(l1=1e-5))

1-3. model3 : l2 regularizer 추가

model1과 아키텍쳐는 동일하며, regularizer를 다음과 같이 파라미터로 추가했다.

tf.keras.layers.Conv2D(64, (3,3), activation='relu', input\_shape=(300, 300, 3),

                           kernel\_regularizer=regularizers.l2(l2=0.0001))

tf.keras.layers.Dense(512, activation='relu',

                          bias\_regularizer=regularizers.l2(l2=0.0001))

1-4. model4 : l1\_l2 regularizer 추가

l1, l2 regularizer 모두 다음과 같이 파라미터로 추가했다.

tf.keras.layers.Conv2D(64, (3,3), activation='relu', input\_shape=(300, 300, 3),

                           kernel\_regularizer=regularizers.l1\_l2(l1=1e-5, l2=1e-4))

tf.keras.layers.Dense(512, activation='relu',

                          bias\_regularizer=regularizers.l1\_l2(l1=1e-5, l2=1e-4))

2) epoch결과 일부

Epoch 7/15

30/30 [==============================] - 37s 1s/step - loss: 0.6451 - accuracy: 0.7437 - val\_loss: 1.0381 - val\_accuracy: 0.5820

Epoch 8/15

30/30 [==============================] - 34s 1s/step - loss: 0.6106 - accuracy: 0.7604 - val\_loss: 1.4631 - val\_accuracy: 0.3945

Epoch 9/15

30/30 [==============================] - 34s 1s/step - loss: 0.5264 - accuracy: 0.8146 - val\_loss: 0.9029 - val\_accuracy: 0.6680

Epoch 10/15

30/30 [==============================] - 34s 1s/step - loss: 0.3852 - accuracy: 0.8740 - val\_loss: 0.6909 - val\_accuracy: 0.7617

Epoch 11/15

30/30 [==============================] - 34s 1s/step - loss: 0.3961 - accuracy: 0.8656 - val\_loss: 0.6125 - val\_accuracy: 0.7852

Epoch 12/15

30/30 [==============================] - 34s 1s/step - loss: 0.3554 - accuracy: 0.8729 - val\_loss: 0.6856 - val\_accuracy: 0.8203

Epoch 13/15

30/30 [==============================] - 34s 1s/step - loss: 0.3062 - accuracy: 0.9167 - val\_loss: 0.6046 - val\_accuracy: 0.8008

Epoch 14/15

30/30 [==============================] - 33s 1s/step - loss: 0.4305 - accuracy: 0.8708 - val\_loss: 0.8244 - val\_accuracy: 0.6758

Epoch 15/15

30/30 [==============================] - 33s 1s/step - loss: 0.3219 - accuracy: 0.8958 - val\_loss: 1.0379 - val\_accuracy: 0.6172

3) fit 결과 도표(x축은 epoch)

화이트보드이(가) 표시된 사진

자동 생성된 설명

(Test data set 성능)

12/12 [==============================] - 60s 5s/step - loss: 0.2867 - accuracy: 0.8710

12/12 [==============================] - 2s 196ms/step - loss: 0.1525 - accuracy: 0.9597

12/12 [==============================] - 2s 199ms/step - loss: 0.1131 - accuracy: 0.9435

12/12 [==============================] - 2s 198ms/step - loss: 0.1133 - accuracy: 0.9758

4) 결과 및 insight 요약

기본모델인 model1이 train set에서의 학습 효율은 매우 좋았으며, 다른 Regularizer 모델들은 비교적 좋지 않았다. 그러나 검증용 데이터에서의 loss와 accuracy를 보면 Regularizer 모델들이 보다 좋은 성능을 보였다. 이는 과적합을 방지하여 보다 예측력이 좋은 모델이 만들어졌음을 의미한다.

다만 l1 Regularizer의 경우 그 값이 1e-4이상만 되어도 학습에 크게 악영향을 끼치므로 값을 1e-5 정도로 작게 할 필요가 있으며, 최종 test dataset 성능은 l1과 l2를 모두 사용한 model4가 가장 좋았지만 위 학습과정을 미루어 보았을 때는 l1이나 l2 중 하나만(l1을 추천한다)을 사용하는 것이 바람직해 보인다.

4) 결과 및 insight 요약

**- Dropout 실험**

1) 모델 생성 및 summary

1-1. model1 : 기본 모델(Dropout X)

테이블이(가) 표시된 사진

자동 생성된 설명

1-2. model2: Dropout(0.3) 추가

텍스트이(가) 표시된 사진

자동 생성된 설명

1-3. model3: Dropout(0.5) 추가

텍스트이(가) 표시된 사진

자동 생성된 설명

2) epoch결과 일부

model1

Epoch 15/20

63/63 [==============================] - 64s 1s/step - loss: 0.0617 - accuracy: 0.9812 - val\_loss: 1.3850 - val\_accuracy: 0.7044

Epoch 16/20

63/63 [==============================] - 63s 1s/step - loss: 0.0608 - accuracy: 0.9826 - val\_loss: 1.4636 - val\_accuracy: 0.6310

Epoch 17/20

63/63 [==============================] - 63s 1s/step - loss: 0.0528 - accuracy: 0.9812 - val\_loss: 1.3433 - val\_accuracy: 0.7639

Epoch 18/20

63/63 [==============================] - 64s 1s/step - loss: 0.0494 - accuracy: 0.9841 - val\_loss: 2.0948 - val\_accuracy: 0.6905

Epoch 19/20

63/63 [==============================] - 63s 1s/step - loss: 0.0578 - accuracy: 0.9816 - val\_loss: 1.0805 - val\_accuracy: 0.7063

Epoch 20/20

63/63 [==============================] - 63s 1s/step - loss: 0.0385 - accuracy: 0.9886 - val\_loss: 1.8790 - val\_accuracy: 0.6190

model2

Epoch 15/20

63/63 [==============================] - 64s 1s/step - loss: 0.0538 - accuracy: 0.9846 - val\_loss: 0.3800 - val\_accuracy: 0.8770

Epoch 16/20

63/63 [==============================] - 65s 1s/step - loss: 0.0623 - accuracy: 0.9757 - val\_loss: 0.4713 - val\_accuracy: 0.8333

Epoch 17/20

63/63 [==============================] - 64s 1s/step - loss: 0.0621 - accuracy: 0.9767 - val\_loss: 0.5742 - val\_accuracy: 0.7837

Epoch 18/20

63/63 [==============================] - 64s 1s/step - loss: 0.0606 - accuracy: 0.9812 - val\_loss: 0.2978 - val\_accuracy: 0.8869

Epoch 19/20

63/63 [==============================] - 63s 1s/step - loss: 0.0684 - accuracy: 0.9762 - val\_loss: 0.5523 - val\_accuracy: 0.8115

Epoch 20/20

63/63 [==============================] - 63s 1s/step - loss: 0.0884 - accuracy: 0.9757 - val\_loss: 0.6781 - val\_accuracy: 0.7242

model3

Epoch 15/20

63/63 [==============================] - 63s 1s/step - loss: 0.0748 - accuracy: 0.9702 - val\_loss: 1.3827 - val\_accuracy: 0.5714

Epoch 16/20

63/63 [==============================] - 63s 1s/step - loss: 0.0880 - accuracy: 0.9717 - val\_loss: 0.9504 - val\_accuracy: 0.6845

Epoch 17/20

63/63 [==============================] - 63s 1s/step - loss: 0.0608 - accuracy: 0.9792 - val\_loss: 0.8715 - val\_accuracy: 0.7421

Epoch 18/20

63/63 [==============================] - 63s 1s/step - loss: 0.0694 - accuracy: 0.9767 - val\_loss: 0.9465 - val\_accuracy: 0.7024

Epoch 19/20

63/63 [==============================] - 63s 1s/step - loss: 0.0642 - accuracy: 0.9782 - val\_loss: 0.6987 - val\_accuracy: 0.7778

Epoch 20/20

63/63 [==============================] - 63s 1s/step - loss: 0.0696 - accuracy: 0.9772 - val\_loss: 0.9654 - val\_accuracy: 0.6627

3) fit 결과 도표(x축은 epoch)

|  |  |  |
| --- | --- | --- |
| 모델 | Loss | Accuracy |
| model1 |  |  |
| model2 |  |  |
| model3 |  |  |

4) 결과 및 insight 요약

model1, model2, model3 모두 train set에서의 학습 성능은 좋지만, validation set에서의 정확도는 모두 90퍼센트를 넘기지 못하였다. Validation loss 그래프를 보면, model2를 제외하고는 loss가 감소하지 못하고 있음을 확인할 수 있다. model2가 dropout이 적용되지 않은 model1보다 훨씬 좋은 성능을 내는 것으로 보아, dropout이 오버피팅의 문제를 해결하는데 큰 도움을 주는 것을 실험적으로 확인할 수 있었다. 하지만 dropout이 0.5인 model3은 오히려 dropout이 적용되지 않은 model1과 비슷한 성능을 보이고 있는데, 적당한 dropout 확률을 선택하는 것이 중요해 보인다.

**- Batch Normalization 실험**

1) 모델 생성 및 summary

1-1. model1 : 기본 모델

테이블이(가) 표시된 사진

자동 생성된 설명

1-2. model2: Convolution layer마다 batch normalization이 적용된 모델

**텍스트이(가) 표시된 사진

자동 생성된 설명**

2) epoch결과 일부

model1

Epoch 15/20

63/63 [==============================] - 64s 1s/step - loss: 0.0617 - accuracy: 0.9812 - val\_loss: 1.3850 - val\_accuracy: 0.7044

Epoch 16/20

63/63 [==============================] - 63s 1s/step - loss: 0.0608 - accuracy: 0.9826 - val\_loss: 1.4636 - val\_accuracy: 0.6310

Epoch 17/20

63/63 [==============================] - 63s 1s/step - loss: 0.0528 - accuracy: 0.9812 - val\_loss: 1.3433 - val\_accuracy: 0.7639

Epoch 18/20

63/63 [==============================] - 64s 1s/step - loss: 0.0494 - accuracy: 0.9841 - val\_loss: 2.0948 - val\_accuracy: 0.6905

Epoch 19/20

63/63 [==============================] - 63s 1s/step - loss: 0.0578 - accuracy: 0.9816 - val\_loss: 1.0805 - val\_accuracy: 0.7063

Epoch 20/20

63/63 [==============================] - 63s 1s/step - loss: 0.0385 - accuracy: 0.9886 - val\_loss: 1.8790 - val\_accuracy: 0.6190

model2

Epoch 15/20

63/63 [==============================] - 65s 1s/step - loss: 0.0954 - accuracy: 0.9747 - val\_loss: 2.0862 - val\_accuracy: 0.7183

Epoch 16/20

63/63 [==============================] - 66s 1s/step - loss: 0.1082 - accuracy: 0.9683 - val\_loss: 103.8188 - val\_accuracy: 0.3333

Epoch 17/20

63/63 [==============================] - 65s 1s/step - loss: 0.0770 - accuracy: 0.9772 - val\_loss: 33.8130 - val\_accuracy: 0.3690

Epoch 18/20

63/63 [==============================] - 65s 1s/step - loss: 0.0961 - accuracy: 0.9747 - val\_loss: 65.1188 - val\_accuracy: 0.3333

Epoch 19/20

63/63 [==============================] - 65s 1s/step - loss: 0.0561 - accuracy: 0.9821 - val\_loss: 1.8732 - val\_accuracy: 0.6647

Epoch 20/20

63/63 [==============================] - 64s 1s/step - loss: 0.0541 - accuracy: 0.9831 - val\_loss: 101.8535 - val\_accuracy: 0.3333

3) fit 결과 도표(x축은 epoch)

|  |  |  |
| --- | --- | --- |
| 모델 | Loss | Accuracy |
| model1 |  |  |
| model2 |  |  |

4) 결과 및 insight 요약

model1과 model2 모두 좋지 않은 성능을 보였지만, 특히 각 convolution layer마다 batch normalization이 적용된 model2의 학습이 정상적으로 이루어지지 않았다. 오버피팅 문제를 해결하는 방법으로 사용되는 batch normalization이 오히려 학습을 방해한 원인에 대해서는 추가적인 연구가 필요해 보이다. 다만, 흥미로운 점은 model2의train set에서의 학습 속도가 매우 빠르다는 점이다. 다른 모델들과 달리3 epoch만에 train set 정확도가 90퍼센트를 넘겼다.

* **활성함수,Group,Depthwise-Separable-CNN 실험**

기본 모델

model = Sequential([

    tf.keras.layers.Conv2D(32, (7,7), activation='relu', input\_shape=(300, 300, 3)),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(512, activation='relu'),

    tf.keras.layers.Dense(3, activation='softmax')

])

**\* 활성함수 실험 Relu vs sigmoid**

1) 모델 생성 및 summary

Relu와 sigmoid가 미치는 영향을 탐구하기 위해, relu를 base모델로 해서 차근차근 relu를 sigmoid로 바꿔서 실험을 해본다.

- relu를 전부 sigmoid로 바꿔본다. -> activation=’sigmoid’

tf.keras.layers.Conv2D(32, (7,7), activation='sigmoid', input\_shape=(300, 300, 3)),

    tf.keras.layers.MaxPooling2D(3,3),

- relu 중 한 두개만 sigmoid로 바꿔본다. -> sigmoid 1개 or sigmoid 2개

tf.keras.layers.Conv2D(32, (7,7), activation='relu', input\_shape=(300, 300, 3)),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(64, (3,3), activation='sigmoid'),

    tf.keras.layers.MaxPooling2D(3,3),

2) epoch결과 일부

- 전부 relu

Epoch 13/15

126/126 [==============================] - 60s 477ms/step - loss: 0.0710 - accuracy: 0.9742 - val\_loss: 0.5045 - val\_accuracy: 0.8594

Epoch 14/15

126/126 [==============================] - 60s 473ms/step - loss: 0.0714 - accuracy: 0.9772 - val\_loss: 0.4253 - val\_accuracy: 0.8516

- 전부 sigmoid

Epoch 16/20

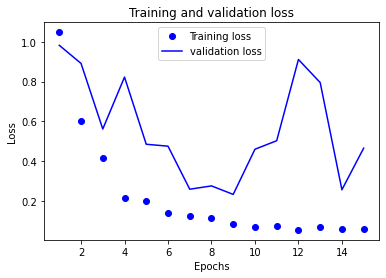
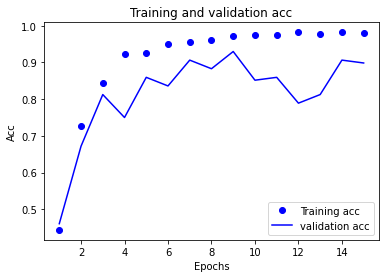
126/126 [==============================] - 54s 424ms/step - loss: 1.1116 - accuracy: 0.3279 - val\_loss: 1.0984 - val\_accuracy: 0.4297

- sigmoid 1개 or 2개만 넣기

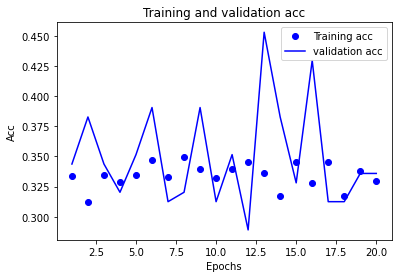
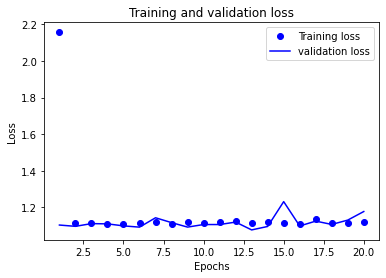
Epoch 11/15

126/126 [==============================] - 55s 434ms/step - loss: 1.0988 - accuracy: 0.3199 - val\_loss: 1.0985 - val\_accuracy: 0.3594

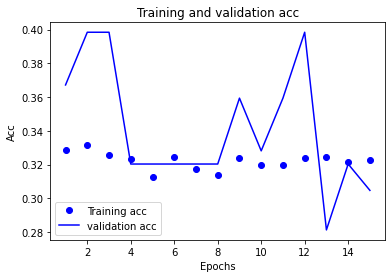
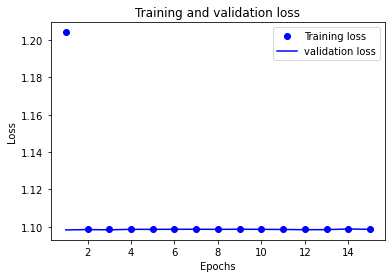
3) fit 결과 도표(x축은 epoch)



- 전부 sigmoid



- sigmoid 1개 or 2개



4) 결과 및 insight 요약

우리가 sigmoid 대신 relu를 쓰는 이유는 결과적으로 성능개선의 차원이 아니라 학습의 가능/불가능을 결정하기 때문이다. Sigmoid를 쓰면 성능이 올라가지 않고 loss또한 감소하지 않는다. Gradient vanishing문제가 심각하게 발생하는 것이다. Sigmoid가 몇 개이든 상관없이 conv2d와 함께 한 번이라도 쓰이면 학습은 진행되지 않는다. 따라서 cnn은 무조건 relu와 함께 써야할 듯 싶다.

**\* groups Conv2d 와 depthwise separable Conv2d 실험**

1) 모델 생성 및 summary

다양한 형태의 cnn이 성능을 개선하는데 유의미한 효과가 있는 지 판단하기 위해 기본 모델을 baseline으로 잡고 그룹 cnn과 depthwise separable cnn을 넣어서 성능을 비교해본다.

- 전체 group conv화

    tf.keras.layers.Conv2D(33, (7,7), activation='relu', input\_shape=(300, 300, 3),groups=3),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(66, (3,3), activation='relu',groups=3),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(132, (3,3), activation='relu',groups=3),

채널을 groups개수에 맞게 나누어져야 하므로 가장 가까운 나눠 떨어지는 수에 맞게 채널을 약간씩 수정했습니다.

- 일부분 group conv화

    tf.keras.layers.Conv2D(33, (7,7), activation='relu', input\_shape=(300, 300, 3),groups=3),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),

- depthwise separable cnn추가

    tf.keras.layers.Conv2D(32, (7,7), activation='relu', input\_shape=(300, 300, 3)),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),

    tf.keras.layers.MaxPooling2D(3,3),

    tf.keras.layers.DepthwiseConv2D((3, 3), use\_bias=False),

    tf.keras.layers.Conv2D(128, (1, 1), activation='relu'),

2) epoch결과 일부

- 전체 group conv화 : loss가 explode해버린다.

Epoch 7/15

126/126 [==============================] - 54s 429ms/step - loss: 848.6597 - accuracy: 0.3398 - val\_loss: 158568.6094 - val\_accuracy: 0.2344

Epoch 00007: saving model to checkpoints

Epoch 8/15

126/126 [==============================] - 54s 429ms/step - loss: 125641776.0000 - accuracy: 0.3249 - val\_loss: 1.0973 - val\_accuracy: 0.3984

Epoch 00008: saving model to checkpoints

Epoch 9/15

126/126 [==============================] - 53s 424ms/step - loss: nan - accuracy: 0.3259 - val\_loss: nan - val\_accuracy: 0.2969

- 일부분 group conv화

Epoch 14/15

126/126 [==============================] - 54s 428ms/step - loss: 0.0626 - accuracy: 0.9792 - val\_loss: 1.0386 - val\_accuracy: 0.6641

Epoch 00014: saving model to checkpoints

Epoch 15/15

126/126 [==============================] - 54s 432ms/step - loss: 0.0495 - accuracy: 0.9851 - val\_loss: 0.4413 - val\_accuracy: 0.7969

- depthwise separable cnn추가

Epoch 9/15

126/126 [==============================] - 55s 436ms/step - loss: 0.1237 - accuracy: 0.9588 - val\_loss: 0.6397 - val\_accuracy: 0.7344

Epoch 00009: val\_loss did not improve from 0.47815

Epoch 10/15

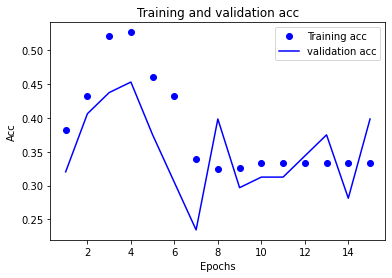
126/126 [==============================] - 55s 438ms/step - loss: 0.1163 - accuracy: 0.9578 - val\_loss: 0.2152 - val\_accuracy: 0.9219

Epoch 00010: val\_loss improved from 0.47815 to 0.21524, saving model to ./checkpoints/

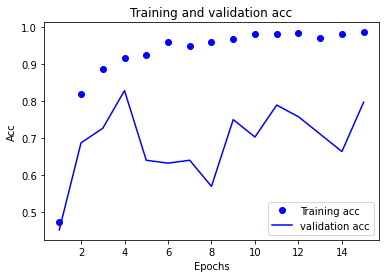
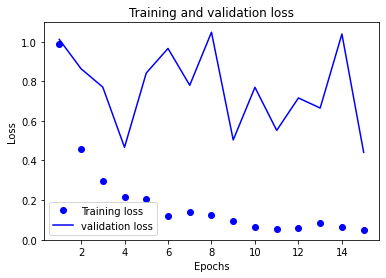
3) fit 결과 도표(x축은 epoch)

- 전체 group conv화

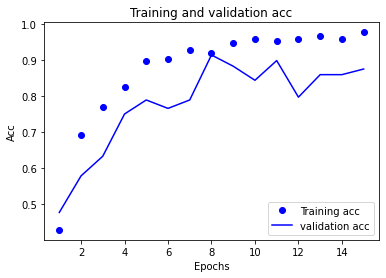
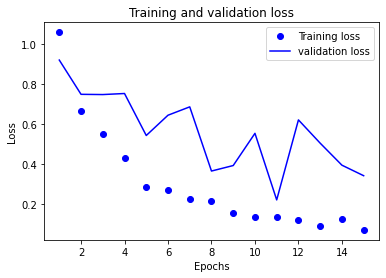
텍스트이(가) 표시된 사진

자동 생성된 설명

- 일부분 group conv화



- depthwise separable cnn추가



4) 결과 및 insight 요약

그룹 convolution은 일종의 채널 사이의 독립성을 확보하고 경량화 시키는데 사용되는 만큼 모든 부분에 group conv을 쓰면 group 채널 간의 정보가 파악이 안 되어서 학습이 전혀 안 되는 것을 알 수 있다. 특히 loss가 explode해서 NAN값으로 바뀌는 것이 인상적이었다. 모든 부분에 사용하지 않고 특정 convolution에만 적용하면 학습이 되는 것을 알 수 있으나 기존 모델보다 성능이나 수렴속도가 확실히 떨어진다. 이는 채널 간의 독립성 보장이 성능 향상에 기여하지 못함을 알 수 있다. 마지막으로 depthwise-separable convolution을 사용했을 때는 놀랍게도 유의미한 성능개선이 일어났다. Base 모델을 사용했을 때는 val\_acc가 약 0.86였지만, 새롭게 도입된 모델은 0.92까지 성능이 올라가는 것을 발견할 수 있었다. 따라서 성능 개선이 필요할 때(generalize 목적으로) depthwise-separable convolution을 도입하는 것은 나쁘지 않은 방법이다. 실제로 test\_accuracy를 측정해본 결과 [0.03930472210049629, 0.9811828136444092] 로스와 accuracy가 매우 준수함을 알 수 있다.