

▼ 데이터 불러오기 (학습 준비 단계)

```
import tensorflow as tf
tf.config.list_logical_devices()

[LogicalDevice(name='/device:CPU:0', device_type='CPU')]
```

▼ 이미지 불러오기

```
import os
import warnings
warnings.filterwarnings(action='ignore')

os.environ["CUDA_VISIBLE_DEVICES"]="0" # GPU 할당
```

```
from google.colab import drive
drive.mount('/content/gdrive')
```

Mounted at /content/gdrive

```
File "<ipython-input-16-eb9e7b163a8e>", line 7
    path = '/content/gdrive/MyDrive/project/data/train'
```

SyntaxError: EOL while scanning string literal

SEARCH STACK OVERFLOW

```
#create training dataset
from glob import glob
import numpy as np
import PIL
from PIL import Image

path = '/content/gdrive/MyDrive/project/data/train'

training_images = []
training_labels = []

for filename in glob(path+"*"):
    for img in glob(filename + "/*.JPEG"):
        an_img = PIL.Image.open(img) #read img
        img_array = np.array(an_img) #img to array
        training_images.append(img_array) #append array to training_images
        label = filename.split('WWW')[-1] #get label
        training_labels.append(label) #append label

training_images = np.array(training_images)
training_labels = np.array(training_labels)

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
training_labels= le.fit_transform(training_labels)
training_labels = training_labels.reshape(-1,1)

print(training_images.shape)
print(training_labels.shape)

(200,)
(200, 1)

#create training dataset
from glob import glob
import numpy as np
import PIL
from PIL import Image

path = '/content/gdrive/MyDrive/project/data/test'
```

파일 경로를 [/data/train](#)에서 불러옵니다. (조금 시간이 걸릴 수 있습니다)

여기서 label은 image가 저장된 폴더의 이름으로 설정되며, 총 50000개의 train image가 존재하는 것을 알 수 있습니다.

파일 경로를 [/data/test](#)에서 불러옵니다. (조금 시간이 걸릴 수 있습니다)

여기서 label은 image가 저장된 폴더의 이름으로 설정되며, 총 10000개의 train image가 존재하는 것을 알 수 있습니다.

▼ 이미지 확인하기

```
import matplotlib.pyplot as plt
%matplotlib inline

for i in range(20):
    plt.subplot(4, 5, i + 1)
    plt.imshow(training_images[i*5])
    print(training_labels[i*5], end=" ",)
plt.show()
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```



10종류의 label과 image가 나타나게 된다면 데이터 불러오기 성공입니다



더블클릭 또는 Enter 키를 눌러 수정



```
label_num = [0]
```

```
label_name = label_num
```

```
for i in range(1):
```

```
    print(label_num[i], label_name[i])
```

```
    0 0
```

▼ 데이터 전처리

▼ data augmentation을 통한 데이터 생성

data augmentation은 data set의 개수를 늘려서 overfitting을 해결하기 위해 등장한 개념입니다.

생성된 샘플은 가능한 진짜에 가까워야 하기 때문에 data의 특성에 따라서 augmentation을 조절해주어 모델이 잘 학습할 수 있도록 해야합니다.

keras에서 제공하는 ImageDataGenerator 함수를 사용하여 데이터를 생성해 보겠습니다.

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

```
tf.random.set_seed(42)
```

```
image_generator = ImageDataGenerator(
    rotation_range=30,
    brightness_range = [0.8, 1.0],
    zoom_range=0.3,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    vertical_flip=False
)
```

각각의 변수들은 다음과 같은 의미를 가집니다.

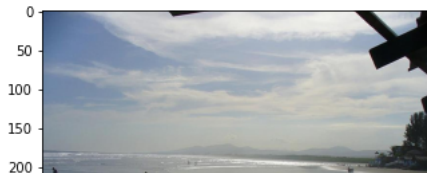
- rotation_range : 이미지를 랜덤한 각도로 돌리는 정도
- brightness_range : 이미지의 밝기를 랜덤하게 다르게 주는 정도
- zoom_range : 사진을 확대하는 정도
- width_shift_range : 사진을 왼쪽 오른쪽으로 움직이는 정도
- height_shift_range : 사진을 위 아래로 움직이는 정도
- horizontal_flip : y축을 기준으로 반전 (오른쪽 왼쪽 뒤집기)
- vertical_flip : x축을 기준으로 반전 (위 아래 뒤집기)

여기서 저는 vertical_flip를 비활성화 했습니다. 이유는 위에 설명했듯이 생성된 샘플은 가능한 진짜에 가까워야 하기 때문입니다. 예를 들어 우리가 자동차라는 데이터를 위아래로 뒤집을 경우, 바퀴가 위로간 뒤집힌 자동차의 이미지가 생성됩니다. 이런 이미지를 학습하게 되면 변별력이 증가할 수는 있지만 일반적인 자동차의 판단에 좋지않은 영향을 끼치게 될 것 같아서 제외하였습니다.

다만 위아래로 뒤집더라도 원본적인 형태는 바뀌지 않기 때문에 하는것이 좋을 것 같다는 의견도 있습니다. 이는 추후에 실험해봐야할 것 같습니다.

▼ data augmentation 살펴보기

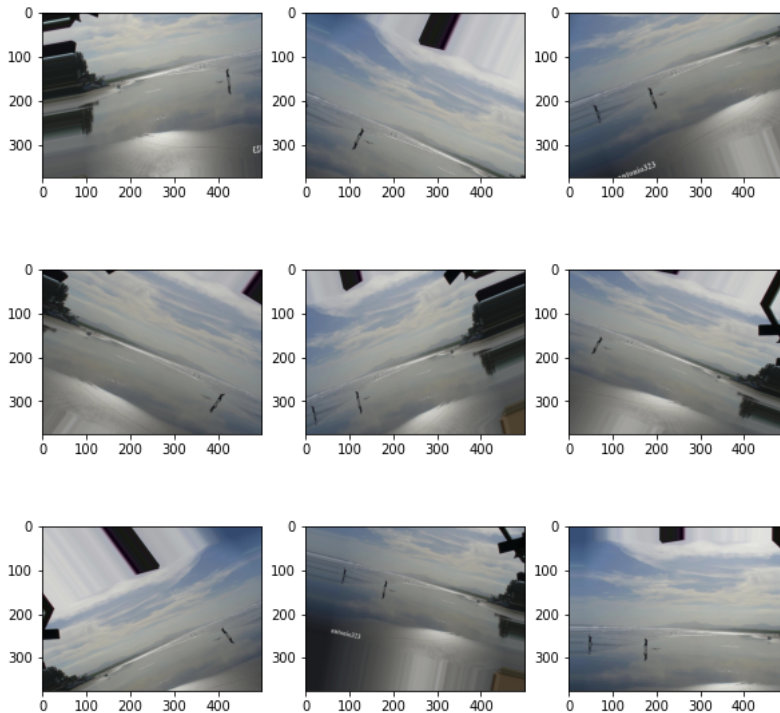
```
sample_image = training_images[1]
plt.imshow(sample_image.astype('uint8'))
sample_image = np.expand_dims(sample_image, 0)
```



```
sample_image_it = image_generator.flow(sample_image, batch_size=1, seed = 42)
```

```
---
```

```
fig = plt.figure(figsize = (10,10))
for i in range(9):
    plt.subplot(3, 3, 1+i)
    batch = sample_image_it.next()
    image = batch[0].astype('uint8')
    plt.imshow(image)
plt.show()
```



▼ 잘 작동하는 것을 확인했으니 이제는 우리가 사용할 데이터에 augmentation을 적용해 봅시다.

```
augment_size=1000

np.random.seed(42)

random_mask = np.random.randint(training_images.shape[0], size=augment_size)
training_image_aug = training_images[random_mask].copy()
training_labels_aug = training_labels[random_mask].copy()

random_mask = np.random.randint(test_images.shape[0], size=augment_size)
test_image_aug = test_images[random_mask].copy()
test_labels_aug = test_idx[random_mask].copy()

print(training_image_aug)
```

저는 15만개의 데이터를 추가로 생성하도록 하겠습니다. (뒤에서 전체다 mixup도 적용할것이기 때문입니다)

```
training_image_aug = image_generator.flow(training_image_aug, np.zeros(augment_size), batch_size=augment_size, shuffle=False, seed = 42).next()[0]
training_images = np.concatenate((training_images, training_image_aug))
training_labels = np.concatenate((training_labels, training_labels_aug))
```

```

TypeError                                Traceback (most recent call last)
TypeError: only size-1 arrays can be converted to Python scalars

```

The above exception was the direct cause of the following exception:

```

ValueError                                Traceback (most recent call last)
<ipython-input-95-d0b1247a089a> in <module>
----> 1 training_image_aug = image_generator.flow(training_image_aug, np.zeros(augment_size), batch_size=augment_size, shuffle=False, seed =
42).next()[0]
      2 training_images = np.concatenate((training_images, training_image_aug))
      3 training_labels = np.concatenate((training_labels, training_labels_aug))

```

1 frames

```

/usr/local/lib/python3.8/dist-packages/keras/preprocessing/image.py in __init__(self, x, y, image_data_generator, batch_size, shuffle,
sample_weight, seed, data_format, save_to_dir, save_prefix, save_format, subset, ignore_class_split, dtype)
    669         y = y[split_idx:]
    670

```

```

print(training_images.shape)
print(training_labels.shape)

```

```

(200,)
(200,)

```

data augmentation을 통해서 5만개였던 train data가 총 20만개로 증가한 모습입니다.

▼ mixup augmentation

data augmentation의 한 종류 (Mixup 말고도 cutmix, augmix 등 다양한 mix 기법들이 있습니다.)

```

import tensorflow as tf

sample_image = training_images[1]
sample_label = tf.one_hot(training_labels[1], 10) # one hot encoding을 진행해야 mixup을 할 수 있습니다.

sample_image2 = training_images[50]
sample_label2 = tf.one_hot(training_labels[50], 10)

fig = plt.figure(figsize = (8,8))

plt.subplot(2, 2, 1)
plt.imshow(sample_image.astype('uint8'))
print(sample_label)

plt.subplot(2, 2, 2)
plt.imshow(sample_image2.astype('uint8'))
print(sample_label2)

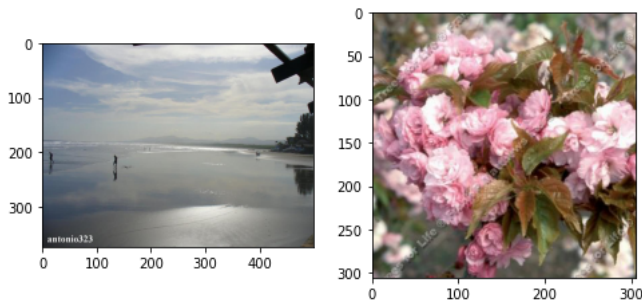
plt.show()

```

```

tf.Tensor([1. 0. 0. 0. 0. 0. 0. 0. 0. 0.], shape=(10,), dtype=float32)
tf.Tensor([1. 0. 0. 0. 0. 0. 0. 0. 0. 0.], shape=(10,), dtype=float32)

```



▼ Mixup 함수를 정의합니다

우선은 예제 함수를 그대로 가져다쓰도록 하겠습니다.

<https://keras.io/examples/vision/mixup/>

```

def sample_beta_distribution(size, concentration_0=0.2, concentration_1=0.2):
    gamma_1_sample = tf.random.gamma(shape=[size], alpha=concentration_1)
    gamma_2_sample = tf.random.gamma(shape=[size], alpha=concentration_0)
    return gamma_1_sample / (gamma_1_sample + gamma_2_sample)

```

```
def mix_up(ds_one, ds_two, batch_size = 1, alpha=0.2):
    # Unpack two datasets
    images_one, labels_one = ds_one
    images_two, labels_two = ds_two

    # Sample lambda and reshape it to do the mixup
    l = sample_beta_distribution(batch_size, alpha, alpha)
    x_l = tf.reshape(l, (batch_size, 1, 1, 1))
    y_l = tf.reshape(l, (batch_size, 1))

    # Perform mixup on both images and labels by combining a pair of images/labels
    # (one from each dataset) into one image/label
    images = images_one * x_l + images_two * (1 - x_l)
    labels = labels_one * y_l + labels_two * (1 - y_l)
    return (images, labels)
```

▼ 이후 두가지 사진을 mix한 결과를 확인합니다.

```
mix_image, mix_label = mix_up((sample_image, sample_label), (sample_image2, sample_label2), batch_size = 1, alpha = 0.5)

fig = plt.figure(figsize = (10,10))
for i in range(9):
    plt.subplot(3, 3, 1+i)
    image = mix_image[i]
    plt.imshow(image.numpy().squeeze().astype('uint8'))
    print(mix_label[i].numpy().tolist())
```

▼ 잘 작동하는 것을 확인했으니 이제는 우리가 사용할 데이터에 mixup을 적용해 봅시다.

(데이터 크기가 커서 오래 걸릴 수 있습니다)

```
import random
random.seed(42)
from sys import stdout

# training_labels = tf.one_hot(training_labels, 10) #mixup을 적용하기 위해 one-hot 기법을 적용해줍니다

mix_training_images = []
mix_training_labels = []

for i in range(3):
    random_num = random.sample(range(0,50), 50) #augmentation을 적용한 데이터를 제외하고 mix해보겠습니다
    print("WnAttempt", i)
    progress_before = 0

    for i in range(0,50,2):
        image_1 = training_images[random_num[i]]
        label_1 = training_labels[random_num[i]]

        image_2 = training_images[random_num[i+1]]
        label_2 = training_labels[random_num[i+1]]

    # mix_image, mix_label = mix_up((image_1, label_1), (image_2, label_2))

    # mix_training_images.append(mix_image[0])
    # mix_training_labels.append(mix_label[0])

    #just for ui
    progress = int(100*(i/49998))
    if progress != progress_before:
        progress_before = progress
        stdout.write("Wr ===== %d%% completed ===== " %progress)
        stdout.flush()

mix_training_images = np.array(mix_training_images)
mix_training_labels = np.array(mix_training_labels)

Attempt 0

Attempt 1

Attempt 2

print('mix_train 크기:',mix_training_images.shape)
print('mix_label 크기:',mix_training_labels.shape)

mix_train 크기: (0,)
mix_label 크기: (0.)
```

▼ Data set 나누기

```
from sklearn.model_selection import train_test_split

training_labels = np.array(training_labels)
# training_labels = training_labels.reshape(-1,10) #mixup에서 one-hot 기법을 적용했다면, shape을 바꿔줍니다.

X_train, X_valid, y_train, y_valid = train_test_split(training_images,
                                                    training_labels,
                                                    test_size=0.05,
                                                    stratify = training_labels,
                                                    random_state=42)

X_train = np.concatenate((X_train, mix_training_images)) #mixup한 75000개의 데이터를 train set에 추가해줍니다
# y_train = np.concatenate((y_train, mix_training_labels))

X_test = test_images
```

train set과, validation set으로 데이터를 나눠줍니다. 여기서 저는 **stratify**를 추가하여 각각의 **label**이 골고루 분포되도록 설정하였습니다.

stratify를 추가할 경우, train set과 validation set은 데이터를 나눠주기 이전처럼 각 label마다 10%의 이상적인 데이터 비율을 유지할 수 있습니다.

만약 stratify를 추가하지 않을 경우, 어떤 label은 18%로 많이 차지하고, 어떤 label은 2%로 적게 차지하여 샘플링 편향이 발생할 수 있습니다.

우리가 지금 사용하는 데이터를 예시로 들어 쉽게 설명하자면, 비행기에 대한 데이터가 90%, 새에 대한 데이터가 10%의 양으로 들어있다면, 새에 대한 학습이 비행기보다 많이 이루어지지 않아서 새를 비행기로 착각하는 현상이 더욱 심하게 발생할 수 있습니다.

추가로 **validation data**의 크기는 항상 20%처럼 %로 확보해야 하는 것이 아닙니다. 충분히 **validation**에 사용하기에 많은 양의 데이터를 확보했다고 생각하면 나머지 **data**는 **test data**쪽으로 몰아주는것이 좋습니다.

<https://brunch.co.kr/@coolmindory/31>

따라서 기존에 데이터가 50000개일 때에도 10000개의 validation set으로 잘 동작했기 때문에 전체 데이터의 0.05, 5%인 10000개 정도면 통계적으로 여러가지 분포를 충분히 포함할 수 있을 것이라고 생각하여 나머지 265000개의 데이터를 Train에 사용함으로써 모델이 최대한 많은 데이터를 학습하고, 파라미터를 최적화하도록 했습니다.

```
print('X_train 크기:', X_train.shape)
print('y_train 크기:', y_train.shape)
print('X_valid 크기:', X_valid.shape)
print('y_valid 크기:', y_valid.shape)
print('X_test 크기:', X_test.shape)
```

```
X_train 크기: (190,)
y_train 크기: (190, 10)
X_valid 크기: (10,)
y_valid 크기: (10, 10)
X_test 크기: (25,)
```

```
X_train = X_train / 255.0
X_valid = X_valid / 255.0
X_test = X_test / 255.0
```

```
print(X_train)
print(y_train)
print(X_valid)
print(y_valid)
print(X_test)
```

[0, 255] 사이의 값을 갖는 경우 255로 나눠 [0, 1] 사이의 값으로 바꿀 수 있습니다.

[0, 1]의 값으로 변환하면 나중에 Sigmoid 함수의 출력과 비교할 수 있습니다.

입력의 범위(dynamic range)가 너무 넓은 것은 훈련이 안 되는 원인으로 작용할 수 있기 때문에 꼭 0~1 사이로 맞춰줍니다.

▼ 모델 생성하기

모델 생성은 매우 중요한 부분입니다. 다양한 경진대회에서 승리한 유명한 모델을 구현해볼 수도 있고, 최신 논문에 구현된 기법들을 가져와 적용해볼 수도 있습니다.

현재는 resnet, efficientnet 등 유명한 모델을 구현하여 사용하지 않고 단순한 커스텀 CNN 모델을 구현해보았습니다.

모델의 깊이와 성능은 어느정도 양의 상관관계에 있기 때문에 과적합과 gradient vanishing을 견제하면서 커스텀 모델을 깊게 짜보려고 합니다.

추후 커스텀 모델의 성능이 어느정도 진척도가 나지 않았을때, 논문에 나온 모델들 혹은 모듈을 구현해보며 단순 CNN 커스텀 모델과 성능 차이를 비교해보려고 합니다.

이처럼 모델의 층을 더 쌓아보고, 수정해보면서 데이터에 적합한 모델을 찾아나가는 과정이 필요합니다.

```
import tensorflow as tf

model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(64, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(64, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer

    tf.keras.layers.MaxPooling2D(2, 2, padding='SAME'), #pooling layer

    tf.keras.layers.Conv2D(128, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(128, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(128, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer

    tf.keras.layers.MaxPooling2D(2, 2, padding='SAME'), #pooling layer

    tf.keras.layers.Conv2D(256, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(256, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(256, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(256, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer

    tf.keras.layers.MaxPooling2D(2, 2, padding='SAME'), #pooling layer

    tf.keras.layers.Conv2D(512, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(512, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(512, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer
    tf.keras.layers.Conv2D(512, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3)), #cnn layer
    tf.keras.layers.BatchNormalization(), #batch norm layer

    tf.keras.layers.MaxPooling2D(2, 2, padding='SAME'), #pooling layer

    tf.keras.layers.GlobalAveragePooling2D(), #pooling layer

    tf.keras.layers.Dense(256, activation = 'relu'), #fully connected layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(64, activation = 'relu'), #fully connected layer
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation = 'softmax') # ouput layer
])
```

batch normalization을 사용하면 regularization 효과가 있기 때문에 dropout 효과는 맨 마지막에만 사용하였으며, conv 연산 이후에 항상 추가하여 gradient의 vanishing 혹은 exploding을 방지하고 local minima에 빠지지 않도록 하였습니다.

Flatten 대신 **GlobalAveragePooling2D**를 사용하여 feature map을 벡터화시켜주고 fc layer 중간에 dropout을 꺼서 regularization을 통해 과적합을 방지하였습니다. (resnet에서부터 global average pooling을 사용했다고 하네요)

▼ 초보분들을 위한 keras layer들의 설명 및 사용법

▼ tf.keras.layers.Conv2D(512, (3,3), activation='relu', padding='SAME',input_shape=(32, 32, 3))

우리가 CNN이라고 부르는 Convolution 레이어를 생성해줍니다.

주요 인수들의 의미는 아래 글에 자세히 나와있습니다. https://tykimos.github.io/2017/01/27/CNN_Layer_Talk/

tf.keras.layers.MaxPooling2D(2, 2, padding='SAME')

pooling방식에는 크게 Max와 Average가 있습니다. 보통 Max가 최대값이라는 개념을 활용하여 이미지의 특성을 잘 잡아내기 때문에 성능이 좋아 많이 쓰입니다.

여기서 2, 2의 값이 의미하는 것은 2 by 2 matrix로 2 by 2 pixel을 1pixel로 크기와 특성을 뽑아 축소하겠다는 말입니다.

지금 우리가 사용하는 image는 32 by 32 의 이미지인데, 이를 2 by 2로 pooling 할 경우 16 by 16으로 줄어드게 됩니다.

이렇게 데이터를 줄이게 되면 학습 시간을 단축하고, 오버피팅을 완화할 수 있다고 합니다.

여기서는 이미지 사이즈가 그렇게 크지 않기 때문에 2 by 2 이상으로 pooling하는것은 데이터 손실로 이어질 가능성이 높다고 생각하여 모두 2 by 2로 진행하였습니다.

`tf.keras.layers.GlobalAveragePooling2D()`

global average pooling은 flatten과 같은 역할을 수행하지만, flatten에 비해 장점이 많고, 성능이 좋습니다.

global average pooling은 차원이 $h * w * d$ 인 텐서를 $1 * 1 * d$ 로 줄입니다. 즉, $h * w$ 값의 평균을 취하여 단일 숫자로 줄이는 역할을 합니다.

예를들어 4 by 4 by 3 (뒤에 3는 사진에서 RGB 3가지의 값을 쓰기 때문에 생기는 차원)에 global average pooling을 적용하면 1 by 1 by 3으로 줄어들게 되는 것입니다.

따라서 global average pooling에서는 feature를 1차원 벡터로 만들기 위한 목적으로 pooling을 합니다.

`tf.keras.layers.BatchNormalization()`

batch normalization을 진행합니다. 특별하게 추가해야 되는 인수는 없습니다.

batch normalization이 dropout의 기능처럼 regularization의 효과가 있기 때문에 따로 dropout를 추가로 진행하지 않았습니다.

<https://eehoeskrap.tistory.com/430>

`tf.keras.layers.Dropout(0.7)`

인수인 0.7은 70%의 뉴런을 제외한다는 뜻을 가지고 있습니다.

모델의 과적합을 막기 위한 방식으로 사용됩니다.

dropout 방식은 training때만 사용되며 test 단계에서는 동작하지 않습니다.

`tf.keras.layers.Dense(10, 'softmax')`

input image가 Flatten()등의 layer를 거치면 직렬화가 됩니다(1차원).

Dense layer는 Fully connected layer라고도 불리는데, 1차원 배열의 형태로 평탄화된 행렬을 통해 이미지를 분류하는데 사용되는 계층이라고 합니다.

Relu 함수를 이용하여 뉴런을 활성화하고, 마지막에는 softmax 함수를 사용하여 추론 결과를 출력합니다. 10이 의미하는 바는 10개의 label (추론 class)가 있다는 것입니다.

보통 모델의 맨 마지막에 위치해있는 경우가 많습니다.

더블클릭 또는 Enter 키를 눌러 수정

one_hot encoding을 이용했기 때문에 loss를 categorical_crossentropy로 바꿔줘야 합니다.

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

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 64)	1792
batch_normalization (Batch Normalization)	(None, 32, 32, 64)	256
conv2d_1 (Conv2D)	(None, 32, 32, 64)	36928
batch_normalization_1 (Batch Normalization)	(None, 32, 32, 64)	256
conv2d_2 (Conv2D)	(None, 32, 32, 64)	36928
batch_normalization_2 (Batch Normalization)	(None, 32, 32, 64)	256
max_pooling2d (MaxPooling2D)	(None, 16, 16, 64)	0
conv2d_3 (Conv2D)	(None, 16, 16, 128)	73856
batch_normalization_3 (Batch Normalization)	(None, 16, 16, 128)	512
conv2d_4 (Conv2D)	(None, 16, 16, 128)	147584
batch_normalization_4 (Batch Normalization)	(None, 16, 16, 128)	512
conv2d_5 (Conv2D)	(None, 16, 16, 128)	147584
batch_normalization_5 (Batch Normalization)	(None, 16, 16, 128)	512

max_pooling2d_1 (MaxPooling2)	(None, 8, 8, 128)	0
conv2d_6 (Conv2D)	(None, 8, 8, 256)	295168
batch_normalization_6 (Batch Normalization)	(None, 8, 8, 256)	1024
conv2d_7 (Conv2D)	(None, 8, 8, 256)	590080
batch_normalization_7 (Batch Normalization)	(None, 8, 8, 256)	1024
conv2d_8 (Conv2D)	(None, 8, 8, 256)	590080
batch_normalization_8 (Batch Normalization)	(None, 8, 8, 256)	1024
conv2d_9 (Conv2D)	(None, 8, 8, 256)	590080
batch_normalization_9 (Batch Normalization)	(None, 8, 8, 256)	1024
max_pooling2d_2 (MaxPooling2)	(None, 4, 4, 256)	0
conv2d_10 (Conv2D)	(None, 4, 4, 512)	1180160
batch_normalization_10 (Batch Normalization)	(None, 4, 4, 512)	2048
conv2d_11 (Conv2D)	(None, 4, 4, 512)	2359808
batch_normalization_11 (Batch Normalization)	(None, 4, 4, 512)	2048

total param이 더욱 많을수록, 학습 시간이 오래 걸리며 복잡한 모델을 의미합니다.

▼ 모델 학습하기

batch size와 epoch을 조절하여 적절한 학습을 합니다.

여기서 batch_size를 16, 32 등 2의 n제곱으로 만드는 이유는 컴퓨터의 연산을 고려하기 때문입니다.

```
EPOCH = 32
BATCH_SIZE = 128

earlystopping = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', # 모니터 기준 설정 (val loss)
                                                  patience=5,          # 5 Epoch동안 개선되지 않는다면 종료
                                                  )

reduceLR = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_accuracy',
    factor=0.5,
    patience=3,
)

X_train = X_train.astype(np.float32)
y_train = y_train.astype(np.float32)
```

```
-----
TypeError                                Traceback (most recent call last)
TypeError: only size-1 arrays can be converted to Python scalars
```

The above exception was the direct cause of the following exception:

```
ValueError                                Traceback (most recent call last)
<ipython-input-123-fc5239b6e7e1> in <module>
----> 1 X_train = X_train.astype(np.float32)
      2 y_train = y_train.astype(np.float32)

ValueError: setting an array element with a sequence.
```

SEARCH STACK OVERFLOW

```
data = model.fit(X_train,
                y_train,
                validation_data=(X_valid, y_valid),
                epochs=EPOCH,
                batch_size=BATCH_SIZE,
                callbacks=[reduceLR, earlystopping],)
```

```

ValueError                                Traceback (most recent call last)
<ipython-input-125-cfa6dab0ad4e> in <module>
----> 1 data = model.fit(X_train,
      2                 y_train,
      3                 validation_data=(X_valid, y_valid),
      4                 epochs=EPOCH,
      5                 batch_size=BATCH_SIZE,

1 frames
/usr/local/lib/python3.8/dist-packages/tensorflow/python/framework/constant_op.py in convert_to_eager_tensor(value, ctx, dtype)
    100 dtype = dtypes.as_dtype(dtype).as_datatype_enum
    101 ctx.ensure_initialized()
--> 102 return ops.EagerTensor(value, ctx.device_name, dtype)
    103
    104

```

▼ 학습된 그래프를 보고 overfitting, underfitting의 여부를 확인합니다.

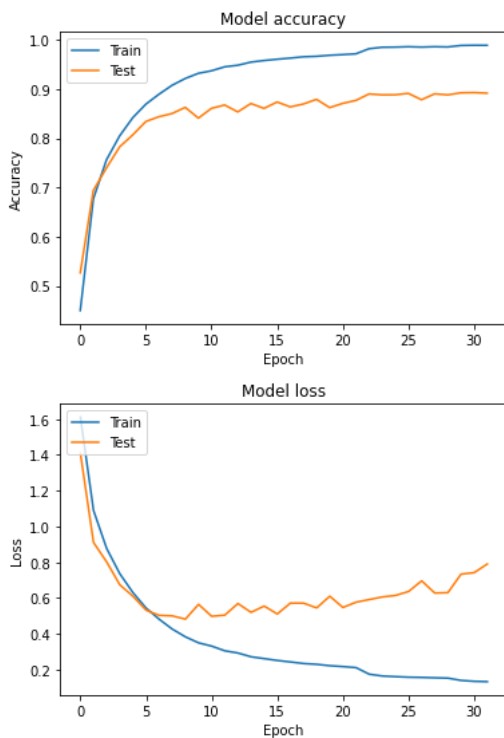
```

import matplotlib.pyplot as plot

plot.plot(data.history['accuracy'])
plot.plot(data.history['val_accuracy'])
plot.title('Model accuracy')
plot.ylabel('Accuracy')
plot.xlabel('Epoch')
plot.legend(['Train', 'Test'], loc='upper left')
plot.show()

plot.plot(data.history['loss'])
plot.plot(data.history['val_loss'])
plot.title('Model loss')
plot.ylabel('Loss')
plot.xlabel('Epoch')
plot.legend(['Train', 'Test'], loc='upper left')
plot.show()

```



▼ 제출하기

학습된 모델을 이용하여 제출합니다

```

pred_proba = model.predict(X_test)

pred_class = []

for i in pred_proba:
    pred = np.argmax(i)
    pred_class.append(pred)

```

```
pred_class = le.inverse_transform(pred_class)
pred_class[0:5]

array(['horse', 'airplane', 'airplane', 'horse', 'airplane'], dtype='<U10')

import pandas as pd

sample_submission = pd.read_csv("./data/sample_submission.csv")

sample_submission.target = pred_class
sample_submission.to_csv("submit_15.csv", index=False)

sample_submission.head(10)
```

	id	target
0	0000.jpg	horse
1	0001.jpg	airplane
2	0002.jpg	airplane
3	0003.jpg	horse
4	0004.jpg	airplane
5	0005.jpg	deer
6	0006.jpg	bird
7	0007.jpg	deer
8	0008.jpg	airplane
9	0009.jpg	deer

