주요 CNN 알고리즘 구현

최석재 *lingua@naver.com*

LeNet

LeNet *LeNet-5*

- Yann LeCun et al.(1998)에 의해 개발된 LeNet은
- 널리 알려진 CNN 아키텍처로는 최초라고 할 수 있다

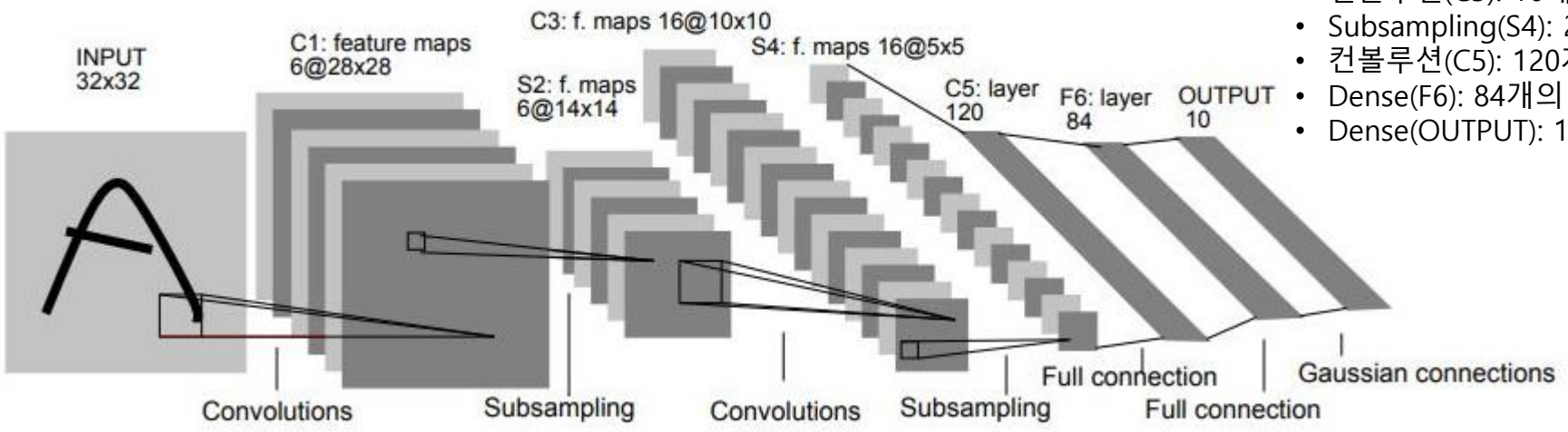
- 1988년부터 많은 연구를 거쳐 LeNet-5가 발표되었으며,
- 알려진 LeNet은 실제로는 LeNet-5를 지칭한다

- LeNet은 손글씨로 쓴 우편번호 인식에 주로 사용되었으며 실용적인 것으로 인식되어
- 이후 딥러닝 및 컴퓨터 비전 분야에서 중요한 초석을 마련한 것으로 평가된다

Architecture

- Convolution Layer, Subsampling(Pooling) Layer, 완전연결계층과의 연결 등을 사용했으며,
- 활성화함수로는 Sigmoid 활성화 함수, 서브샘플링은 Average Pooling을 사용했다

- LeNet의 아키텍처는 지금의 관점에서 보면 간단하고 작은 편
- CPU에서 실행하는 것도 큰 어려움이 없다



- 입력: 32x32 이미지 입력
- 컨볼루션(C1): 6개의 5x5 필터를 사용하여 28x28 특징 맵 생성
- Subsampling(S2): 2x2 풀링 레이어를 사용하여 14x14로 특징 맵 축소
- 컨볼루션(C3): 16개의 5x5 필터를 사용하여 10x10 특징 맵 생성
- Subsampling(S4): 2x2 풀링 레이어를 사용하여 5x5로 특징 맵 축소
- 컨볼루션(C5): 120개의 5x5 필터를 사용하여 1x1 특징 맵 생성
- Dense(F6): 84개의 뉴런을 사용하여 완전 연결 레이어 생성
- Dense(OUTPUT): 10개의 뉴런을 사용하여 출력 레이어 생성

MNIST 데이터 다운로드

- from keras.datasets import mnist

- (train_images, train_labels), (test_images, test_labels) = mnist.load_data()

- print(train_images.shape) # (60000, 28, 28)
- print(test_images.shape) # (10000, 28, 28)

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28)
(10000, 28, 28)
```

※ 현재 데이터는 28x28 사이즈이다

차원 변경

- 내장된 MNIST 데이터를 사용하면서
- 얀 르쿤이 설계한 LeNet의 구조를 그대로 따라가기 위해 데이터의 구조를 변경한다

- 먼저 32x32 사이즈로 변환하기 위하여 height와 width 부분에 두 개씩 패딩을 추가한다
- 다음으로 데이터를 4차원으로 확장한다

- import numpy as np

- train_images = np.pad(train_images, ((0, 0), (2, 2), (2, 2))).reshape((60000, 32, 32, 1))
- test_images = np.pad(test_images, ((0, 0), (2, 2), (2, 2))).reshape((10000, 32, 32, 1))

정규화

- 데이터를 0~1 사이로 정규화하기 위하여 255로 나눈다

- train_images = train_images.astype('float32')/255
- test_images = test_images.astype('float32')/255

- print(train_images.shape)
- print(test_images.shape)

```
(60000, 32, 32, 1)
(10000, 32, 32, 1)
```

종속변수 범주화

- 종속변수는 0~9 사이의 값으로 되어 있으므로
- 결과값을 10 분류로 도출하기 위하여 훈련 및 테스트 데이터의 종속변수를 범주화한다

- from tensorflow.keras.utils import to_categorical
- train_labels = to_categorical(train_labels)
- test_labels = to_categorical(test_labels)

- import numpy as np
- import sys
- np.set_printoptions(threshold=sys.maxsize)
- print(train_labels[:5])

```
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

모델 설계

- LeNet 아키텍처를 따라 모델을 설계해본다

- from keras import models
- from keras import layers

- model = models.Sequential()

C1. 합성곱 계층

- model.add(layers.Conv2D(filters=6, kernel_size=(5, 5), activation='relu', input_shape=(32, 32, 1), padding='valid'))

당시에는 패딩을 사용하지 않아 합성곱 계층을 지날 때마다 맵의 크기가 줄어들었음
padding='valid' (기본값)

S2. 평균 풀링

- model.add(layers.AveragePooling2D(pool_size=(2, 2), strides=2))

모델 설계

C3. 합성곱 계층

- model.add(layers.Conv2D(filters=16, kernel_size=(5, 5), activation='relu', padding='valid'))

S4. 평균 풀링

- model.add(layers.AveragePooling2D(pool_size=(2, 2), strides=2)) # 결과는 5x5 Feature Map

C5. 120개의 5x5 필터를 사용하여 완전연결계층으로 연결되는 합성곱 계층. 결과는 1x1

- model.add(layers.Conv2D(filters=120, kernel_size=(5, 5), activation='relu', padding='valid'))

multi dimension(3D)을 한 개 차원으로 축소하는 Flatten

- model.add(layers.Flatten())

LeNet는 Flatten을 명시적으로 말하지 않았다
1x1 결과를 통하여 사실상 Flatten과 같은 효과를 갖도록 하였다
Flatten 하지 않으려면, 대신 아래와 같이 reshape만 해주어도 된다
model.add(layers.Reshape((120,)))

모델 설계

F6. 완전연결층

- model.add(layers.Dense(units=84, activation='relu'))

분류를 위해 소프트맥스 활성화 함수를 갖는 완전연결층

- model.add(layers.Dense(units=10, activation='softmax'))

모델 요약

- model.summary()

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 average_pooling2d (Average  (None, 14, 14, 6)         0
 Pooling2D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_1 (Avera  (None, 5, 5, 16)          0
 gePooling2D)

 conv2d_2 (Conv2D)           (None, 1, 1, 120)         48120

 flatten (Flatten)           (None, 120)               0

 dense (Dense)               (None, 84)                10164

 dense_1 (Dense)             (None, 10)                850

=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

모델 컴파일 및 훈련

- model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc'])

- history = model.fit(train_images, train_labels, epochs=10, batch_size=128, validation_data=(test_images, test_labels))

```
Epoch 1/10
469/469 [==============================] - 7s 5ms/step - loss: 0.3563 - acc: 0.8983 - val_loss: 0.1170 - val_acc: 0.9638
Epoch 2/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0980 - acc: 0.9705 - val_loss: 0.0703 - val_acc: 0.9781
Epoch 3/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0726 - acc: 0.9781 - val_loss: 0.0535 - val_acc: 0.9828
Epoch 4/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0588 - acc: 0.9818 - val_loss: 0.0479 - val_acc: 0.9842
Epoch 5/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0504 - acc: 0.9841 - val_loss: 0.0374 - val_acc: 0.9879
Epoch 6/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0416 - acc: 0.9867 - val_loss: 0.0429 - val_acc: 0.9870
Epoch 7/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0369 - acc: 0.9886 - val_loss: 0.0313 - val_acc: 0.9892
Epoch 8/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0321 - acc: 0.9898 - val_loss: 0.0301 - val_acc: 0.9902
Epoch 9/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0284 - acc: 0.9910 - val_loss: 0.0297 - val_acc: 0.9910
Epoch 10/10
469/469 [==============================] - 2s 4ms/step - loss: 0.0245 - acc: 0.9926 - val_loss: 0.0294 - val_acc: 0.9903
```

테스트 데이터 평가

- test_loss, test_acc = model.evaluate(test_images, test_labels)
- print('test_acc:', test_acc)

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0294 - acc: 0.9903
test_acc: 0.9902999997138977
```

훈련 과정 확인

- acc = history.history['acc']
- val_acc = history.history['val_acc']
- loss = history.history['loss']
- val_loss = history.history['val_loss’]

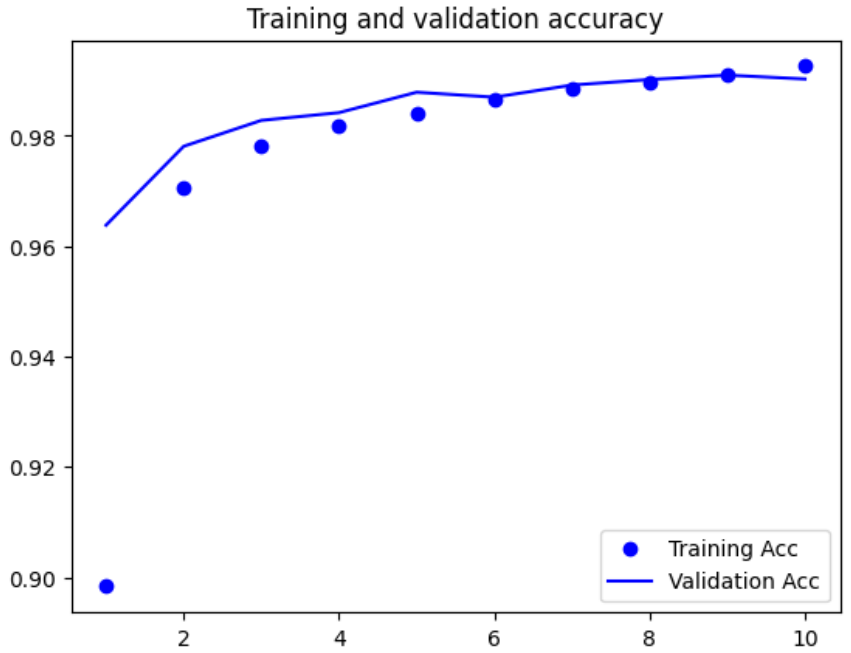
- print('Accuracy of each epoch:', acc)
- epochs = range(1, len(acc) +1)

```
Accuracy of each epoch: [0.898333311080932 6, 0.970516681671142 6, 0.978133320808410 6, 0.981783330440521 2, 0.984066665172576 9, 0.986683309078216 6, 0.988600015640258 8, 0.989783346652984 6, 0.990999966621399, 0.992616653442382 8]
```

정확도 시각화

- import matplotlib.pyplot as plt

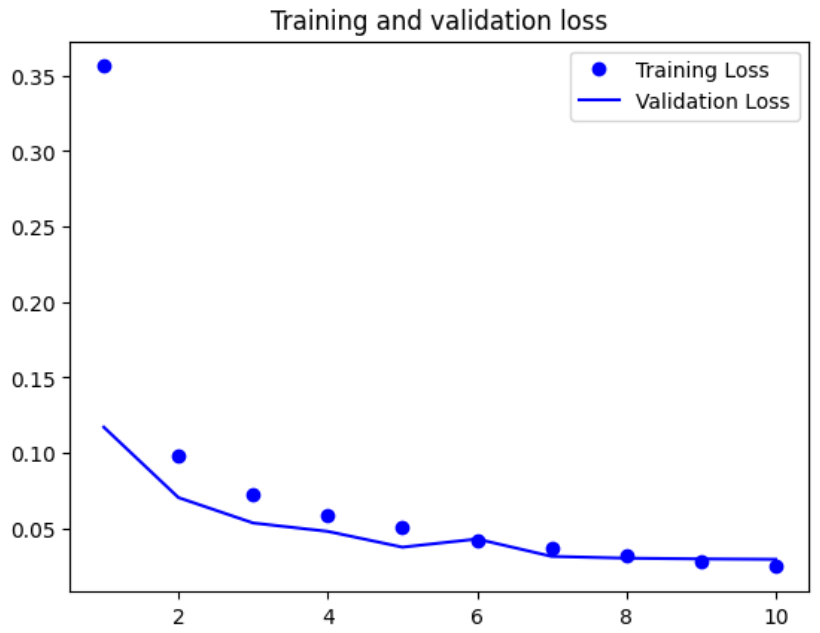
- plt.plot(epochs, acc, 'bo', label='Training Acc')
- plt.plot(epochs, val_acc, 'b', label='Validation Acc')
- plt.title('Training and validation accuracy')
- plt.legend()



손실값 시각화

- plt.figure()

- plt.plot(epochs, loss, 'bo', label='Training Loss')
- plt.plot(epochs, val_loss, 'b', label='Validation Loss')
- plt.title('Training and validation loss')
- plt.legend()



한 개 이미지 예측

- 랜덤 한 개 이미지 추출

- import numpy as np
- sample = np.random.choice(np.arange(0, len(test_images)))
- print(sample)

- print(test_labels[sample])

```
1480
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```

예측

모델을 사용해 예측

슬라이싱으로 한 개 이미지를 (1, 32, 32, 1)로 가져온다

- predictions = model.predict(test_images[sample:sample+1, :, :, :])

[sample, :, :, :]으로 가져오면 차원이 하나 줄어든 3차원으로 가져오게 된다

가장 큰 확률의 인덱스

- predicted_class = np.argmax(predictions, axis=1)[0]
- print("예측된 숫자:", predicted_class)

클래스별 확률

- print("₩n클래스별 확률:")
- for i, prob in enumerate(predictions[0]):
  print(f"Class {i}: {prob:.4f}")

```
1/1 [==============================] - 0s 166ms/step
예측된 숫자: 9

클래스별 확률:
Class 0: 0.0000
Class 1: 0.0000
Class 2: 0.0000
Class 3: 0.0000
Class 4: 0.0000
Class 5: 0.0000
Class 6: 0.0000
Class 7: 0.0000
Class 8: 0.0000
Class 9: 1.0000
```

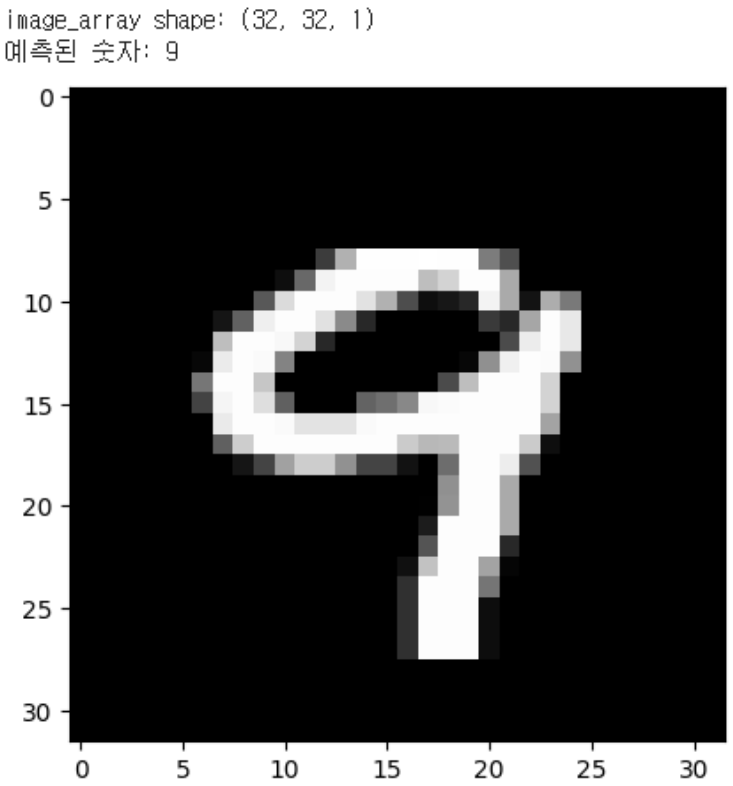
시각화

- from keras.utils import array_to_img
- import matplotlib.pyplot as plt

- image_array = test_images[sample]
- print("image_array shape:", image_array.shape)
- print("예측된 숫자:", predicted_class)

배열을 이미지 객체로 변환
- image = array_to_img(image_array)

정답 이미지 출력
- plt.imshow(image, cmap='gray') # 흑백 이미지로 출력
- plt.show()



AlexNet

AlexNet

- 알렉스넷은 알렉스 크리체브스키(Alex Krizhevsky)와 제프리 힌튼(Geoffrey Hinton)이
- 공동으로 설계한 CNN 아키텍처

- 2012년 ImageNet 시각 인식 챌린지에서 2위보다 Top5 에러율이 10.8% 낮은 15.3% 달성 (정확도 약 60%)
- 모델의 깊이가 좋은 결과를 얻는데 필수적이라는 점을 밝힘
  - 층마다 이미지의 다른 특징을 포착하므로 특징을 충분히 포착할 수 있도록 함
  - 층을 깊게 쌓으면 활성화 함수의 비선형 결과가 연속되어 복잡한 현상에 대응이 가능해짐
- 많은 계산량을 필요로 하여 GPU를 사용하였으며, 이후 많은 논문이 GPU를 사용하게 되었다

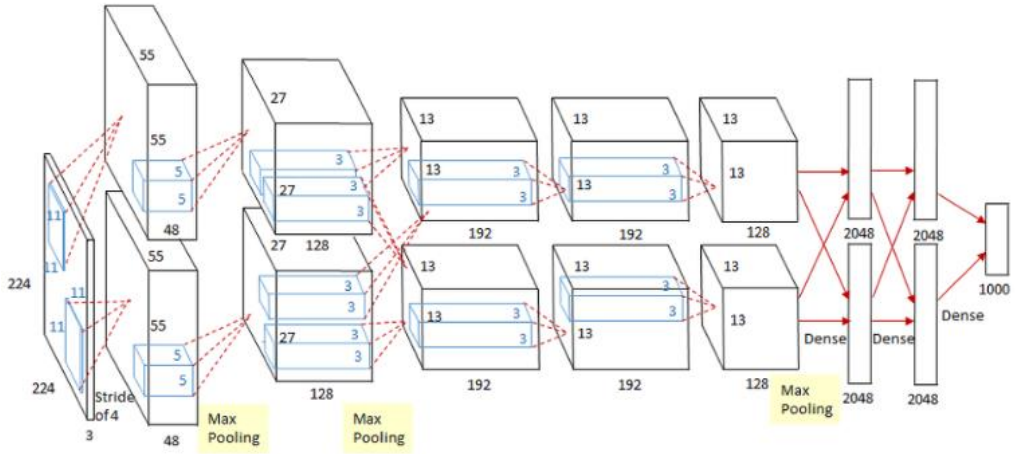
ImageNet 데이터셋은 1,000개의 클래스로 구성되며 총 백만 개가 넘는 데이터를 포함한다
약 120만 개는 학습(training)에 쓰고, 5만개는 검증(validation)에 쓴다. 학습 데이터셋 용량은 약 138GB, 검증 데이터셋 용량은 약 6GB이다.

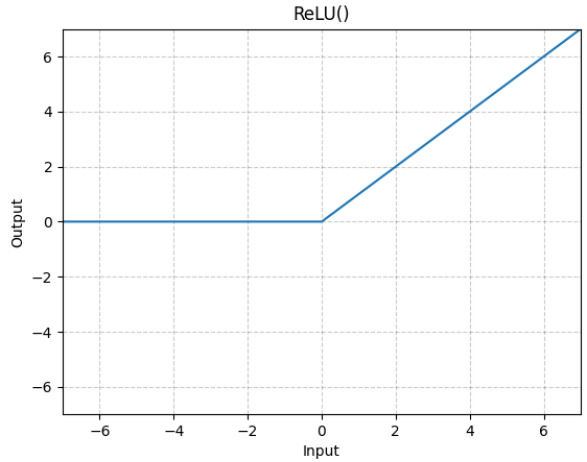
Top5 에러율: 모델이 실제 정답을 상위 5개 예측 중 하나로 정확히 포함하지 못하는 비율
분류 카테고리가 많을 경우 모델이 얼마나 근접하여 예측하였는지를 보고자 할 때 사용된다

Architecture

- 5개의 Convolution Layer 뒤에 3개의 완전연결층을 사용하였으며,
- 맥스 풀링을 본격적으로 사용하였다

- Tanh 및 Sigmoid 보다 향상된 성능을 보인 ReLU 활성화 함수도 본격적으로 사용하였으며,
- 같은 해에 제프리 힌튼에 의해 개발된 Dropout 층도 사용되었다
- AlexNet의 성공으로 이들 기법들이 이후 연구에 많은 영향을 미쳤다





CIFAR-10 데이터셋 다운로드

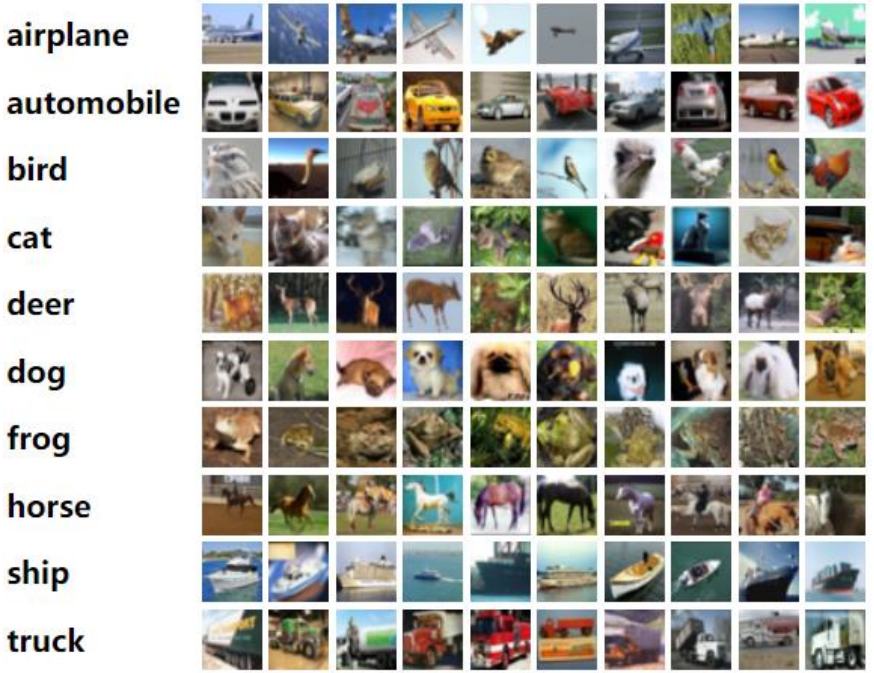
0: airplane
1: automobile
2: bird
3: cat
4: deer
5: dog
6: frog
7: horse
8: ship
9: truck

- AlexNet은 ImageNet 데이터셋으로 훈련되었지만,
- 여기서는 케라스로부터 바로 데이터셋을 다운로드하기 위하여
- 유사한 CIFAR-10 데이터셋을 사용한다

[CIFAR-10 데이터셋]

- 10개 클래스, 32x32 사이즈의 60,000개 이미지로 구성
- 50,000개는 훈련 이미지, 10,000개는 테스트 이미지

- 10개 클래스는 0~9번으로 인덱싱되어 있음



ImageNet은 아래의 사이트에서 신청해야 하며, 거절될 수 있다
https://www.image-net.org/download.php

데이터셋 다운로드

- 고용량 메모리 또는 GPU를 사용해서 진행한다

- from tensorflow.keras.datasets import cifar10
- import numpy as np

- (X_train, y_train), (X_test, y_test) = cifar10.load_data()

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 4s 0us/step
```

데이터 샘플링

- 고용량 메모리를 사용해도 데이터셋이 커서 다루기 어렵기 때문에
- 10%의 데이터만 가지고 진행하기로 한다

샘플링 비율 설정 (10%)
- sampling_ratio = 0.1

훈련 데이터 샘플링
- num_samples = int(len(X_train) * sampling_ratio)
- indices = np.random.choice(len(X_train), num_samples, replace=False) # 비복원추출
- X_train_sampled = X_train[indices]
- y_train_sampled = y_train[indices]

데이터 샘플링

테스트 데이터 샘플링

- num_samples_test = int(len(X_test) * sampling_ratio)
- indices_test = np.random.choice(len(X_test), num_samples_test, replace=False)
- X_test_sampled = X_test[indices_test]
- y_test_sampled = y_test[indices_test]

- print("새로운 훈련 데이터 크기:", X_train_sampled.shape)
- print("새로운 테스트 데이터 크기:", X_test_sampled.shape)

```
새로운 훈련 데이터 크기: (5000, 32, 32, 3)
새로운 테스트 데이터 크기: (1000, 32, 32, 3)
```

이미지 리사이징

- AlexNet이 사용한 224x224에 맞추기 위하여 이미지를 그에 맞게 변환한다
- 컬러의 채널 차원은 CIFAR-10도 RGB 모두 가지고 있으므로 그대로 사용한다

- import tensorflow as tf

- X_train_resized = tf.image.resize(X_train_sampled, [224, 224])
- X_test_resized = tf.image.resize(X_test_sampled, [224, 224])

정규화

- 데이터를 0~1 사이로 정규화하기 위하여 255로 나눈다

- train_images = X_train_resized / 255.0
- test_images = X_test_resized / 255.0
- print(train_images.shape)
- print(test_images.shape)

```
(5000, 224, 224, 3)
(1000, 224, 224, 3)
```

종속변수 확인 및 변수명 변경

- 종속변수가 현재 Numpy 배열로 되어 있는 것을 확인하고,
- 변수명이 일관성을 갖추도록 한다

종속변수 확인
- print(type(y_test_sampled))
- print(y_test_sampled[:10])

변수명 변경
- train_labels = y_train_sampled
- test_labels = y_test_sampled

```
<class 'numpy.ndarray'>
[[1]
 [6]
 [4]
 [9]
 [2]
 [2]
 [0]
 [7]
 [8]
 [3]]
```

모델 설계

- from keras import models
- from keras import layers

- model = models.Sequential()

넓은 영역을 한 번에 처리하기 위하여 11x11의 큰 필터를 사용
빠른 다운샘플링을 위하여 strides를 4로 하였다

합성곱 계층 1

- model.add(layers.Conv2D(filters=96, kernel_size=(11, 11), strides=(4, 4), padding='valid', input_shape=(224, 224, 3), activation='relu'))

합성곱 계층 2

- model.add(layers.Conv2D(filters=256, kernel_size=(5, 5), strides=(1, 1), padding='valid', activation='relu'))
- model.add(layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding='valid'))

strides = 2인데, 풀링은 3x3을 하므로 겹치기(overlapping) 방식으로 풀링이 일어난다
풀링 시 정보를 공유하려는 목적이지만 효과가 분명하지 않고, 계산이 복잡하여 이후에는 잘 사용되지 않았다

모델 설계

합성곱 계층 3

- model.add(layers.Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
- model.add(layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding='valid'))

합성곱 계층 4

- model.add(layers.Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))

합성곱 계층 5

- model.add(layers.Conv2D(filters=256, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu'))
- model.add(layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding='valid'))

모델 설계

Flattening
- model.add(layers.Flatten())

완전연결층 1
- model.add(layers.Dense(4096, activation='relu'))
- model.add(layers.Dropout(0.5))

완전연결층 2
- model.add(layers.Dense(4096, activation='relu'))
- model.add(layers.Dropout(0.5))

완전연결층 3. Output Layer. AlexNet은 1,000 분류를 하였으나, CIFAR-10은 10분류이다
- model.add(layers.Dense(10, activation='softmax'))

모델 요약

- model.summary()

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 54, 54, 96)        34944

 conv2d_1 (Conv2D)           (None, 50, 50, 256)       614656

 max_pooling2d (MaxPooling2  (None, 24, 24, 256)       0
 D)

 conv2d_2 (Conv2D)           (None, 22, 22, 384)       885120

 max_pooling2d_1 (MaxPoolin  (None, 10, 10, 384)       0
 g2D)

 conv2d_3 (Conv2D)           (None, 8, 8, 384)         1327488

 conv2d_4 (Conv2D)           (None, 6, 6, 256)         884992

 max_pooling2d_2 (MaxPoolin  (None, 2, 2, 256)         0
 g2D)

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 4096)              4198400

 dropout (Dropout)           (None, 4096)              0

 dense_1 (Dense)             (None, 4096)              16781312

 dropout_1 (Dropout)         (None, 4096)              0

 dense_2 (Dense)             (None, 10)                40970

=================================================================
Total params: 24767882 (94.48 MB)
Trainable params: 24767882 (94.48 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

모델 컴파일 및 훈련

모델 컴파일

- model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['acc'])

모델 훈련

데이터와 훈련량이 충분하지 않기 때문에 성능은 좋지 않다

- history = model.fit(train_images, train_labels, epochs=10, batch_size=128, validation_data=(test_images, test_labels))

```
Epoch 1/10
40/40 [==============================] - 23s 214ms/step - loss: 2.3189 - acc: 0.1006 - val_loss: 2.2992 - val_acc: 0.0930
Epoch 2/10
40/40 [==============================] - 5s 116ms/step - loss: 2.2849 - acc: 0.1248 - val_loss: 2.3044 - val_acc: 0.0950
Epoch 3/10
40/40 [==============================] - 5s 116ms/step - loss: 2.3076 - acc: 0.0994 - val_loss: 2.3039 - val_acc: 0.0910
Epoch 4/10
40/40 [==============================] - 5s 114ms/step - loss: 2.2982 - acc: 0.1070 - val_loss: 2.2289 - val_acc: 0.0940
Epoch 5/10
40/40 [==============================] - 5s 116ms/step - loss: 2.2003 - acc: 0.1646 - val_loss: 2.6153 - val_acc: 0.0920
Epoch 6/10
40/40 [==============================] - 5s 114ms/step - loss: 2.3421 - acc: 0.0930 - val_loss: 2.3041 - val_acc: 0.0910
Epoch 7/10
40/40 [==============================] - 5s 115ms/step - loss: 2.3005 - acc: 0.1048 - val_loss: 2.2628 - val_acc: 0.1230
Epoch 8/10
40/40 [==============================] - 5s 115ms/step - loss: 2.2175 - acc: 0.1566 - val_loss: 2.1335 - val_acc: 0.1780
Epoch 9/10
40/40 [==============================] - 5s 117ms/step - loss: 2.0901 - acc: 0.1990 - val_loss: 2.1909 - val_acc: 0.1910
Epoch 10/10
40/40 [==============================] - 5s 116ms/step - loss: 2.0167 - acc: 0.2304 - val_loss: 2.2957 - val_acc: 0.2220
```

테스트 데이터 평가

데이터와 훈련량이 충분하지 않기 때문에 성능은 좋지 않다

- test_loss, test_acc = model.evaluate(test_images, test_labels)
- print('test_acc:', test_acc)

```
32/32 [==============================] - 1s 14ms/step - loss: 2.2957 - acc: 0.2220
test_acc: 0.22200000286102295
```

훈련 과정 확인

- acc = history.history['acc']
- val_acc = history.history['val_acc']
- loss = history.history['loss']
- val_loss = history.history['val_loss’]

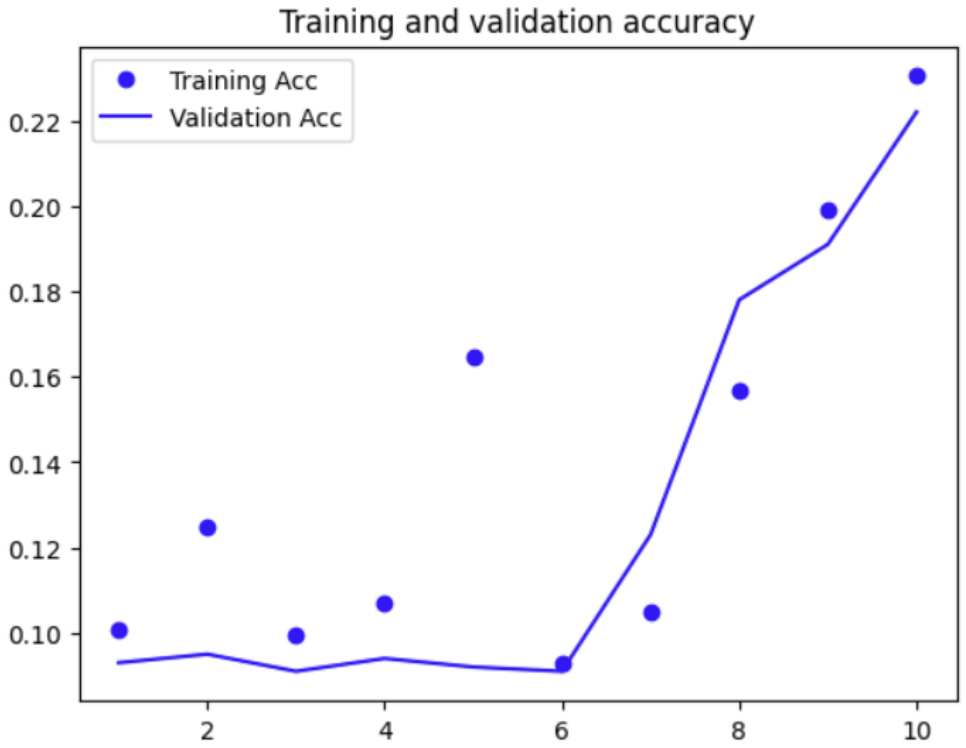
- print('Accuracy of each epoch:', acc)
- epochs = range(1, len(acc) +1)

```
Accuracy of each epoch: [0.1005999967455864, 0.12479999661445618, 0.09939999878406525, 0.10700000077486038, 0.16459999978542328, 0.09300000220537186, 0.10480000078678131, 0.15659999847412
```

정확도 시각화

- import matplotlib.pyplot as plt

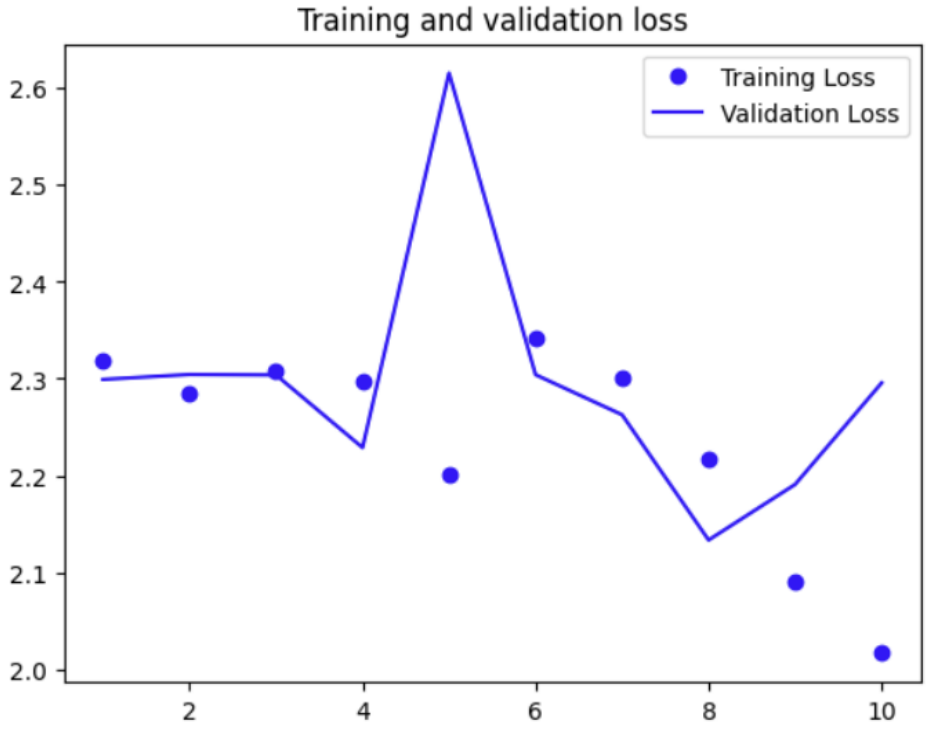
- plt.plot(epochs, acc, 'bo', label='Training Acc')
- plt.plot(epochs, val_acc, 'b', label='Validation Acc')
- plt.title('Training and validation accuracy')
- plt.legend()



손실값 시각화

- plt.figure()

- plt.plot(epochs, loss, 'bo', label='Training Loss')
- plt.plot(epochs, val_loss, 'b', label='Validation Loss')
- plt.title('Training and validation loss')
- plt.legend()



한 개 이미지 예측

랜덤 한 개 이미지 추출

- import numpy as np
- sample = np.random.choice(np.arange(0, len(test_images)))

- print(sample)
- print(test_labels[sample])

```
371
[0]
```

예측

모델을 사용해 예측
- predictions = model.predict(test_images[sample:sample+1])

가장 큰 확률의 인덱스
- predicted_class = np.argmax(predictions, axis=1)[0]
- print("예측된 숫자:", predicted_class)

클래스별 확률
- print("₩n클래스별 확률:")
- for i, prob in enumerate(predictions[0]):
  print(f"Class {i}: {prob:.4f}")

```
1/1 [==============================] - 0s 334ms/step
예측된 숫자: 0

클래스별 확률:
Class 0: 0.3208
Class 1: 0.3165
Class 2: 0.0050
Class 3: 0.0006
Class 4: 0.0036
Class 5: 0.0004
Class 6: 0.0001
Class 7: 0.0023
Class 8: 0.2976
Class 9: 0.0530
```

시각화

- 데이터셋 라벨은 문자형으로 되어 있으므로 숫자를 문자로 변환하는 코드를 추가한다

- from keras.utils import array_to_img
- import matplotlib.pyplot as plt

- image_array = test_images[sample] # 시각화를 위한 이미지 배열 준비
- print("image_array shape:", image_array.shape) # shape 확인

```
image_array shape: (224, 224, 3)
```

시각화

- class_names = {
    0: "airplane",
    1: "automobile",
    2: "bird",
    3: "cat",
    4: "deer",
    5: "dog",
    6: "frog",
    7: "horse",
    8: "ship",
    9: "truck"
  }
- print(class_names[predicted_class]) # airplane

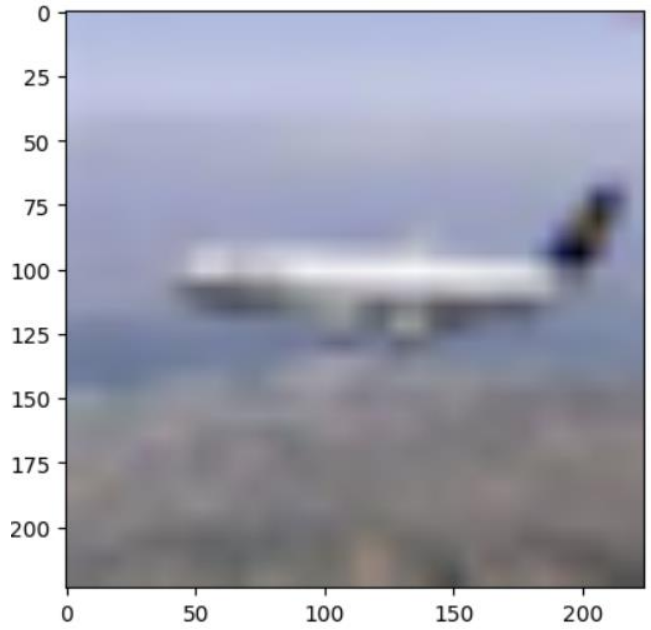
시각화

배열을 이미지 객체로 변환

- image = array_to_img(image_array)

정답 이미지 출력

- plt.imshow(image)
- plt.show()



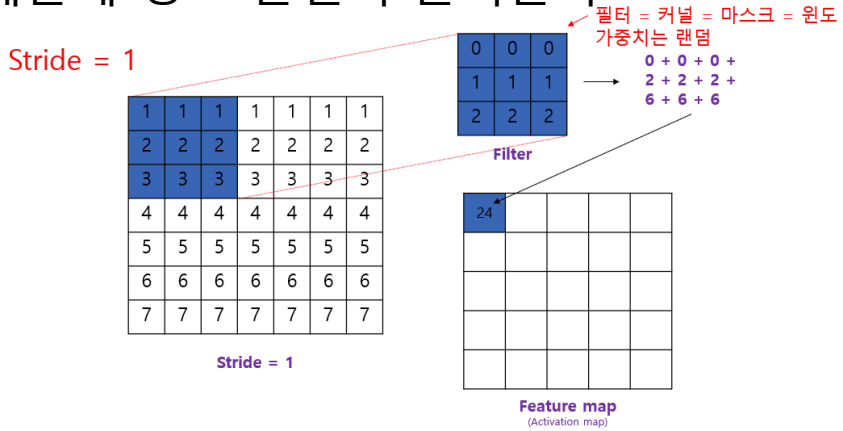
VGG19

VGG19

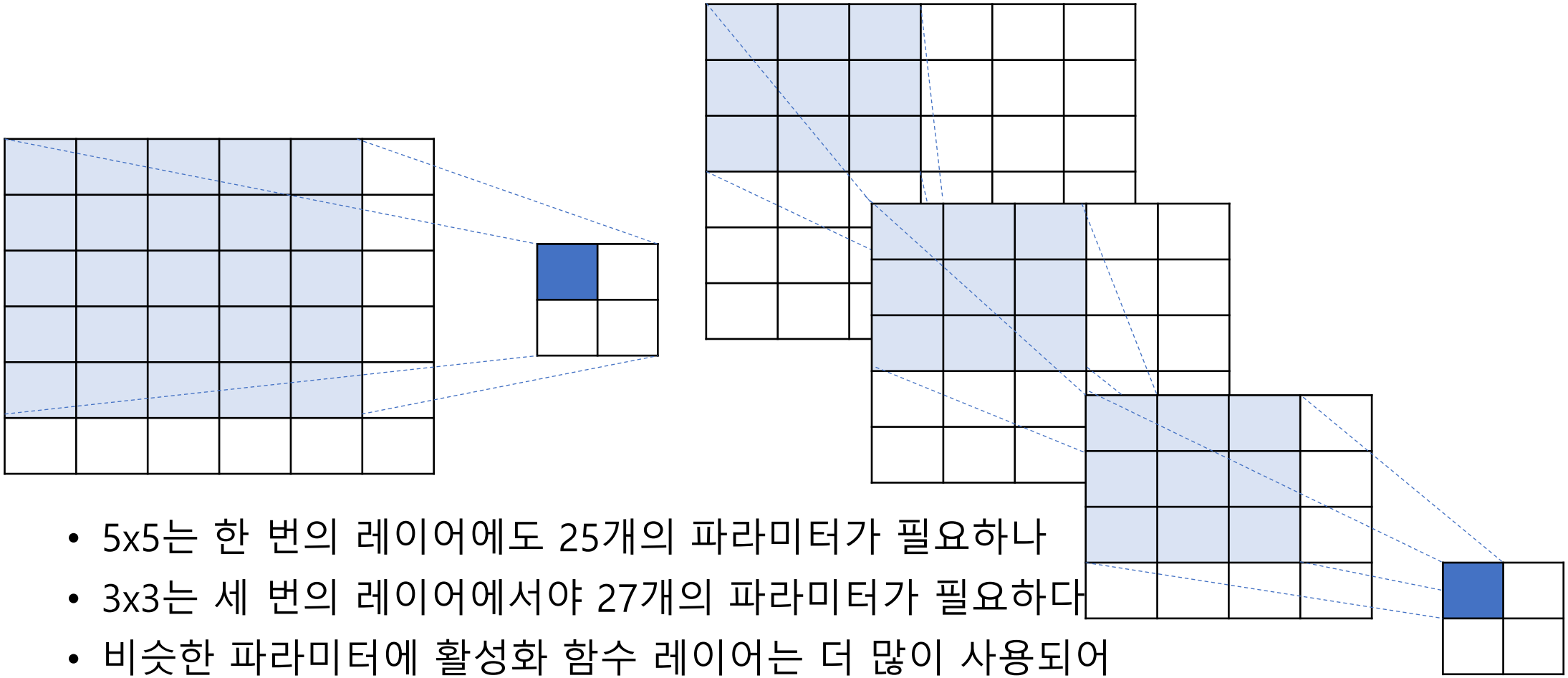
- VGG19는 옥스퍼드 대학교의 Visual Geometry Group의
- Karen Simonyan과 Andrew Zisserman이 개발한 아키텍처로 VGG16과 함께 2014년 발표되었다
- AlexNet에서의 5개 Convolution Layer를 16개로 크게 늘리고, 3개의 완전연결층을 더했다(총 19층)
- 이미지넷의 정확도도 AlexNet 60%에서 71%로 크게 향상 되었다

- 특징은 3x3의 작고 동일한 커널을 사용하는 합성곱 층을 여러 층 배치하는 데 있다
- 작은 커널은 필요한 매개변수의 수가 줄어들기 때문에(3x3=9개) 과적합 가능성이 줄고,
- 연산 시간이 줄기 때문에 깊은 층을 쌓기도 유리하다
- 작은 커널은 좁은 범위의 입력 픽셀을 하나의 출력값으로 압축하기 때문에 정보 손실이 줄어든다

- VGG19는 깊고 작은 커널의 효용을 드러낸 아키텍처이다



작은 필터의 사용

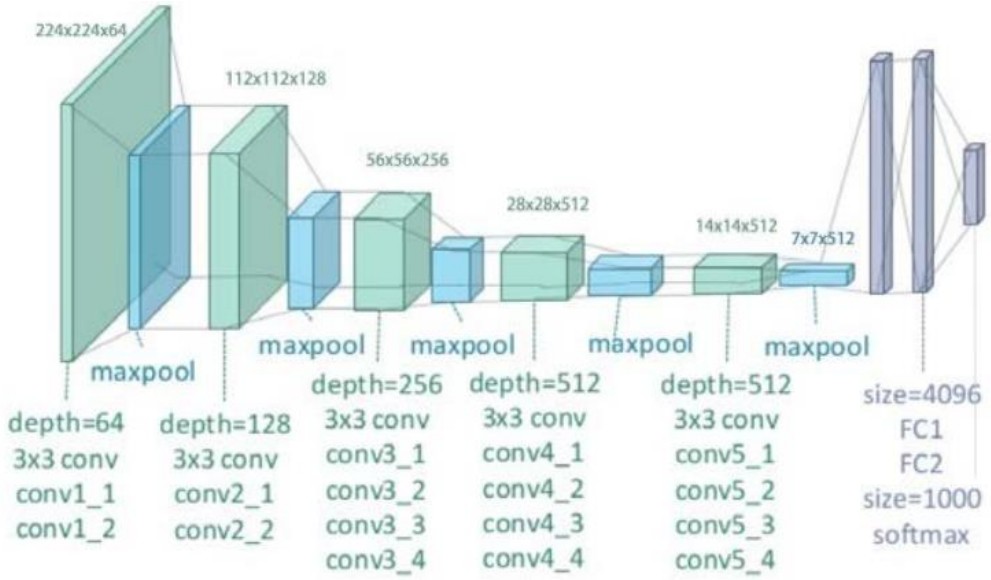


- 5x5는 한 번의 레이어에도 25개의 파라미터가 필요하나
- 3x3는 세 번의 레이어에서야 27개의 파라미터가 필요하다
- 비슷한 파라미터에 활성화 함수 레이어는 더 많이 사용되어
- 모델의 표현력이 높아진다

Architecture

- 전체 아키텍처를 단계별로 구분하고,
- 각 단계에는 3x3의 작은 여러 층의 합성곱 층을 배치하되,
- padding='same'을 사용해 구현해 해상도를 유지한다
- 그리고 각 블록 다음에는 맥스 풀링을 배치하여 특징 맵의 크기를 점차 줄여나간다

- 논문에서는 L2 정규화와 Dropout도 사용하였기에
- 이를 포함하여 구현해본다



오른쪽 그림에는 L2 정규화와 Dropout이 포함되어 있지 않다
보통은 이 둘을 포함하지 않고 구현한다

데이터셋 다운로드

- 고용량 메모리 또는 GPU를 사용해서 진행한다

- from tensorflow.keras.datasets import cifar10
- import numpy as np

- (X_train, y_train), (X_test, y_test) = cifar10.load_data()

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 4s 0us/step
```

데이터 샘플링

- 고용량 메모리를 사용해도 데이터셋이 커서 다루기 어렵기 때문에
- 10%의 데이터만 가지고 진행하기로 한다

샘플링 비율 설정 (10%)
- sampling_ratio = 0.1

훈련 데이터 샘플링
- num_samples = int(len(X_train) * sampling_ratio)
- indices = np.random.choice(len(X_train), num_samples, replace=False) # 비복원추출
- X_train_sampled = X_train[indices]
- y_train_sampled = y_train[indices]

데이터 샘플링

테스트 데이터 샘플링

- num_samples_test = int(len(X_test) * sampling_ratio)
- indices_test = np.random.choice(len(X_test), num_samples_test, replace=False)
- X_test_sampled = X_test[indices_test]
- y_test_sampled = y_test[indices_test]

- print("새로운 훈련 데이터 크기:", X_train_sampled.shape)
- print("새로운 테스트 데이터 크기:", X_test_sampled.shape)

```
새로운 훈련 데이터 크기: (5000, 32, 32, 3)
새로운 테스트 데이터 크기: (1000, 32, 32, 3)
```

이미지 리사이징

- VGG19가 사용한 224x224에 맞추기 위하여 이미지를 그에 맞게 변환한다
- 컬러의 채널 차원은 CIFAR-10도 RGB 모두 가지고 있기 때문에 그대로 사용한다

- import tensorflow as tf

- X_train_resized = tf.image.resize(X_train_sampled, [224, 224])
- X_test_resized = tf.image.resize(X_test_sampled, [224, 224])

정규화

- 데이터를 0~1 사이로 정규화하기 위하여 255로 나눈다

- train_images = X_train_resized / 255.0
- test_images = X_test_resized / 255.0

- print(train_images.shape)
- print(test_images.shape)

```
(5000, 224, 224, 3)
(1000, 224, 224, 3)
```

종속변수 확인 및 변수명 변경

- 종속변수가 현재 Numpy 배열로 되어 있는 것을 확인하고,
- 변수명이 일관성을 갖추도록 한다

종속변수 확인
- print(type(y_test_sampled))
- print(y_test_sampled[:10])

변수명 변경
- train_labels = y_train_sampled
- test_labels = y_test_sampled

```
<class 'numpy.ndarray'>
[[1]
 [6]
 [4]
 [9]
 [2]
 [2]
 [0]
 [7]
 [8]
 [3]]
```

모델 설계

- from keras import models
- from keras import layers
- model = models.Sequential()

Block 1

- model.add(layers.Conv2D(64, (3, 3), padding='same',
  input_shape=(224, 224, 3),
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(64, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

모델 설계

Block 2

- model.add(layers.Conv2D(128, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(128, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

모델 설계

Block 3

- model.add(layers.Conv2D(256, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(256, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(256, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(256, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

모델 설계

Block 4

- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

모델 설계

Block 5

- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.Conv2D(512, (3, 3), padding='same',
  kernel_regularizer=l2(weight_decay),
  activation='relu'))
- model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))

모델 설계

완전연결층

- model.add(layers.Flatten())
- model.add(layers.Dense(4096, activation='relu'))
- model.add(layers.Dropout(0.5)) # Dropout
- model.add(layers.Dense(4096, activation='relu'))
- model.add(layers.Dropout(0.5)) # Dropout
- model.add(layers.Dense(10, activation='softmax')) # 본래 1000 분류이나, CIFAR-10의 10분류로 수정

모델 설계

모델 요약

- model.summary()

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 224, 224, 64)      1792

 conv2d_1 (Conv2D)           (None, 224, 224, 64)      36928

 max_pooling2d (MaxPooling2
 D)                          (None, 112, 112, 64)      0

 conv2d_2 (Conv2D)           (None, 112, 112, 128)     73856

 conv2d_3 (Conv2D)           (None, 112, 112, 128)     147584

 max_pooling2d_1 (MaxPoolin
 g2D)                        (None, 56, 56, 128)       0

 conv2d_4 (Conv2D)           (None, 56, 56, 256)       295168

 conv2d_5 (Conv2D)           (None, 56, 56, 256)       590080

 conv2d_6 (Conv2D)           (None, 56, 56, 256)       590080

 conv2d_7 (Conv2D)           (None, 56, 56, 256)       590080

 max_pooling2d_2 (MaxPoolin
 g2D)                        (None, 28, 28, 256)       0

 conv2d_8 (Conv2D)           (None, 28, 28, 512)       1180160

 conv2d_9 (Conv2D)           (None, 28, 28, 512)       2359808

 conv2d_10 (Conv2D)          (None, 28, 28, 512)       2359808

 conv2d_11 (Conv2D)          (None, 28, 28, 512)       2359808

 max_pooling2d_3 (MaxPoolin
 g2D)                        (None, 14, 14, 512)       0

 conv2d_12 (Conv2D)          (None, 14, 14, 512)       2359808

 conv2d_13 (Conv2D)          (None, 14, 14, 512)       2359808

 conv2d_14 (Conv2D)          (None, 14, 14, 512)       2359808

 conv2d_15 (Conv2D)          (None, 14, 14, 512)       2359808

 max_pooling2d_4 (MaxPoolin
 g2D)                        (None, 7, 7, 512)         0

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 4096)              102764544

 dropout (Dropout)           (None, 4096)              0

 dense_1 (Dense)             (None, 4096)              16781312

 dropout_1 (Dropout)         (None, 4096)              0

 dense_2 (Dense)             (None, 10)                40970

=================================================================
Total params: 139611210 (532.57 MB)
Trainable params: 139611210 (532.57 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

모델 컴파일 및 훈련

모델 컴파일

- model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['acc'])

모델 훈련

데이터와 훈련량이 충분하지 않기 때문에 성능은 좋지 않다

- history = model.fit(train_images, train_labels, epochs=10, batch_size=128, validation_data=(test_images, test_labels))

```
Epoch 1/10
40/40 [==============================] - 115s 2s/step - loss: 2.9079 - acc: 0.1006 - val_loss: 2.3353 - val_acc: 0.0870
Epoch 2/10
40/40 [==============================] - 44s 1s/step - loss: 2.3126 - acc: 0.1036 - val_loss: 2.3053 - val_acc: 0.0870
Epoch 3/10
40/40 [==============================] - 44s 1s/step - loss: 2.3033 - acc: 0.1098 - val_loss: 2.3038 - val_acc: 0.0870
Epoch 4/10
40/40 [==============================] - 44s 1s/step - loss: 2.3033 - acc: 0.1022 - val_loss: 2.3034 - val_acc: 0.0870
Epoch 5/10
40/40 [==============================] - 43s 1s/step - loss: 2.3029 - acc: 0.1098 - val_loss: 2.3036 - val_acc: 0.0870
Epoch 6/10
40/40 [==============================] - 43s 1s/step - loss: 2.3026 - acc: 0.1088 - val_loss: 2.3039 - val_acc: 0.0870
Epoch 7/10
40/40 [==============================] - 43s 1s/step - loss: 2.3027 - acc: 0.1098 - val_loss: 2.3038 - val_acc: 0.0870
Epoch 8/10
40/40 [==============================] - 43s 1s/step - loss: 2.3025 - acc: 0.1098 - val_loss: 2.3045 - val_acc: 0.0870
Epoch 9/10
40/40 [==============================] - 43s 1s/step - loss: 2.3023 - acc: 0.1098 - val_loss: 2.3049 - val_acc: 0.0870
Epoch 10/10
40/40 [==============================] - 43s 1s/step - loss: 2.3025 - acc: 0.1098 - val_loss: 2.3045 - val_acc: 0.0870
```

테스트 데이터 평가

데이터와 훈련량이 충분하지 않기 때문에 성능은 좋지 않다

- test_loss, test_acc = model.evaluate(test_images, test_labels)
- print('test_acc:', test_acc)

```
32/32 [==============================] - 8s 105ms/step - loss: 2.3045 - acc: 0.0870
test_acc: 0.08699999749660492
```

훈련 과정 확인

- acc = history.history['acc']
- val_acc = history.history['val_acc']
- loss = history.history['loss']
- val_loss = history.history['val_loss’]

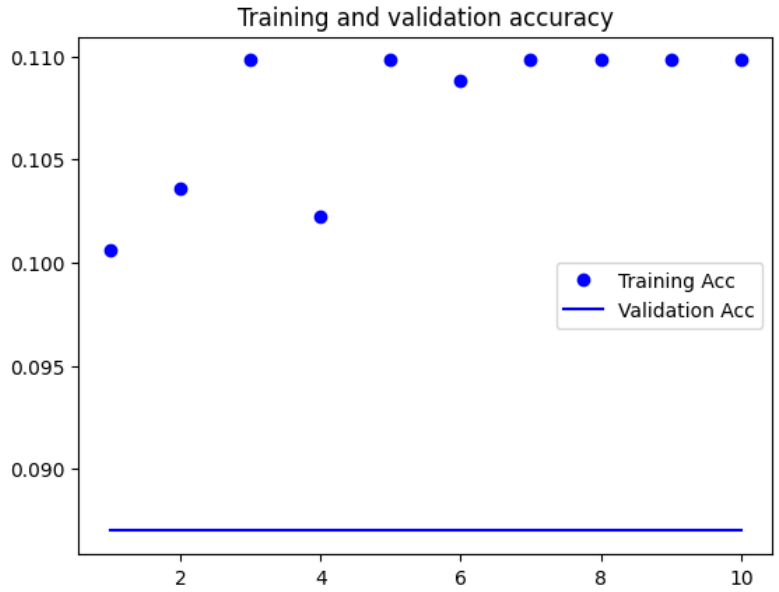
- print('Accuracy of each epoch:', acc)
- epochs = range(1, len(acc) +1)

```
Accuracy of each epoch: [0.1005999967455864, 0.10360000282526016, 0.10980000346899033, 0.10220000147819519, 0.10980000346899033, 0.1088000014424324, 0.10980000346899033, 0.10980000346899033, 0.10980000346899033, 0.10980000346899033]
```

정확도 시각화

- import matplotlib.pyplot as plt

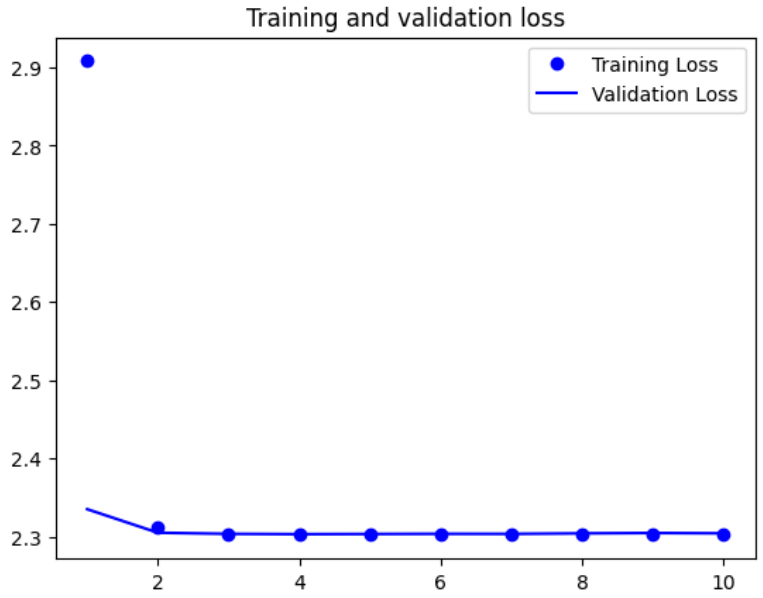
- plt.plot(epochs, acc, 'bo', label='Training Acc')
- plt.plot(epochs, val_acc, 'b', label='Validation Acc')
- plt.title('Training and validation accuracy')
- plt.legend()



손실값 시각화

- plt.figure()

- plt.plot(epochs, loss, 'bo', label='Training Loss')
- plt.plot(epochs, val_loss, 'b', label='Validation Loss')
- plt.title('Training and validation loss')
- plt.legend()



한 개 이미지 예측

랜덤 한 개 이미지 추출

- import numpy as np
- sample = np.random.choice(np.arange(0, len(test_images)))

- print(sample)
- print(test_labels[sample])

```
321
[8]
```

예측

모델을 사용해 예측
- predictions = model.predict(test_images[sample:sample+1])

가장 큰 확률의 인덱스
- predicted_class = np.argmax(predictions, axis=1)[0]
- print("예측된 숫자:", predicted_class)

클래스별 확률
- print("₩n클래스별 확률:")
- for i, prob in enumerate(predictions[0]):
    print(f"Class {i}: {prob:.4f}")

```
1/1 [==============================] - 1s 845ms/step
예측된 숫자: 6

클래스별 확률:
Class 0: 0.1007
Class 1: 0.0952
Class 2: 0.0996
Class 3: 0.0982
Class 4: 0.0994
Class 5: 0.1025
Class 6: 0.1103
Class 7: 0.0968
Class 8: 0.0992
Class 9: 0.0982
```

시각화

- 데이터셋 라벨은 문자형으로 되어 있으므로 숫자를 문자로 변환하는 코드를 추가한다

- from keras.utils import array_to_img
- import matplotlib.pyplot as plt

- image_array = test_images[sample] # 시각화를 위한 이미지 배열 준비
- print("image_array shape:", image_array.shape) # shape 확인

```
image_array shape: (224, 224, 3)
```

시각화

- class_names = {
    0: "airplane",
    1: "automobile",
    2: "bird",
    3: "cat",
    4: "deer",
    5: "dog",
    6: "frog",
    7: "horse",
    8: "ship",
    9: "truck"
  }
- print(class_names[predicted_class]) # frog

시각화

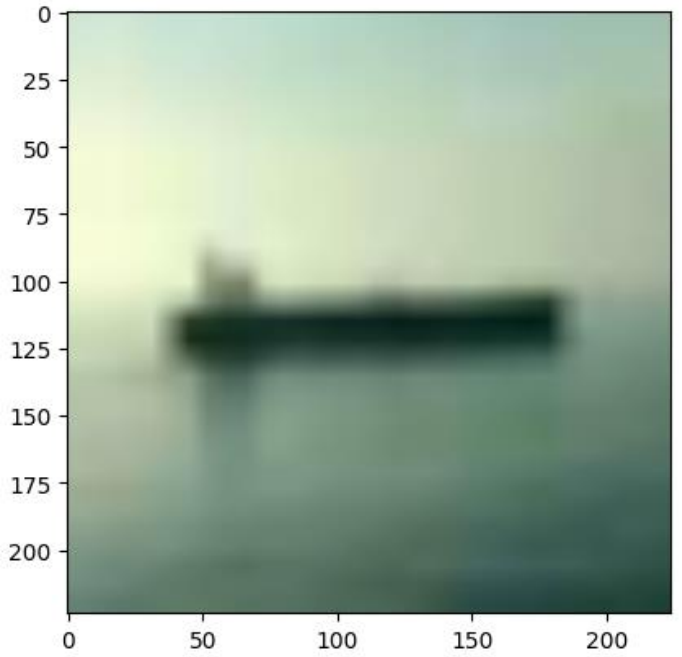
배열을 이미지 객체로 변환

- image = array_to_img(image_array)

정답 이미지 출력

- plt.imshow(image)
- plt.show()

정답은 ship이었다



케라스의 VGG19

- Keras에는 사전학습된 모델들이 있다
- VGG19도 포함되어 있으므로, 이를 로드해본다
- L2 정규화와 Dropout을 사용하지 않은 점을 제외하고는 위와 동일하다

- from tensorflow.keras.applications.vgg19 import VGG19
- vgg = VGG19(input_shape=[224, 224, 3], weights='imagenet', include_top=False)
- vgg.summary()

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
80134624/80134624 [==============================] - 0s 0us/step
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv4 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv4 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv4 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 20024384 (76.39 MB)
Trainable params: 20024384 (76.39 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Keras 내장 사전학습 모델

- 케라스가 제공하는 사전학습 모델의 예
- https://keras.io/api/applications/

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	96.4	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7
NASNetMobile	23	74.4%	91.9%	5.3M	389	27.0	6.7
NASNetLarge	343	82.5%	96.0%	88.9M	533	344.5	20.0
EfficientNetB0	29	77.1%	93.3%	5.3M	132	46.0	4.9