

# Training Deep Neural Nets [2]

- Transfer Learning, Faster Optimizers, Learning Rate Scheduling, Dropout

Samkeun Kim <skim@hknu.ac.kr>

<http://cyber.hknu.ac.kr/>

# Image Classification with Keras

딥 러닝 문제에 접근하는 한 가지 방법은 데이터 세트를 확보하고, 이를 훈련하기 위한 코드를 작성하고, 해당 모델을 훈련시키는데 많은 시간과 에너지를 소비한 다음, 모델을 예측에 사용하는 것이다.

그러나 우리 모두가 위의 처리과정을 반복할 필요는 없다:

⇒ 위의 복잡한 과정 대신에 사전 훈련된 모델을 사용하면 된다!!

현재 연구 커뮤니티에서 사용 가능한 많은 표준 모델을 학습시켜서 공개적으로 게시해 놓고 있다.

먼저 2015년 ILSVRC(스탠포드 대학에서 개최하는 객체 탐지 및 이미지 분류 알고리즘 경시대회)에서 우승한 ResNet-50이라는 유명한 모델 중 하나를 재사용해보자.

# Predicting an Image's Category

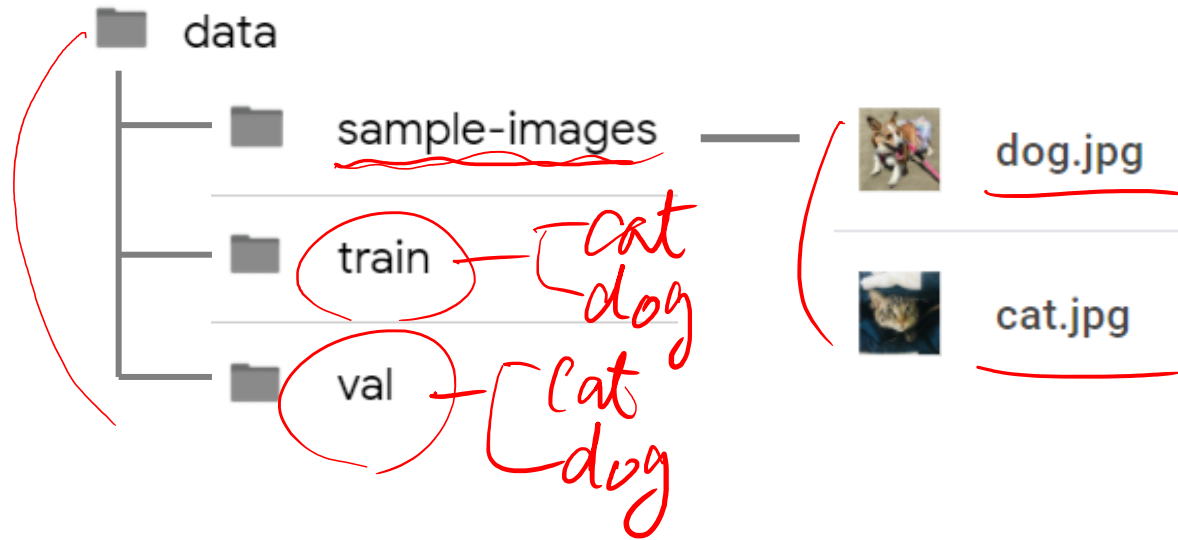
간단한 이미지 분류 파이프라인을 아래처럼 구성할 수 있다:

1. 이미지를 로드한다.
2. 224 x 224 픽셀과 같은 미리 정의된 크기로 크기를 조정한다.
3. 픽셀의 값을 [0, 1] 또는 [-1, 1], 즉 정규화 범위로 조정한다.
4. 사전 훈련된 모델을 선택한다. *ResNet-50*
5. 이미지에 대해 사전 훈련된 모델을 실행하여 카테고리 예측 및 해당 확률 목록을 가져온다.
6. 확률이 가장 높은 몇 가지 카테고리를 표시한다.



# Download dogs-cats dataset:

[https://drive.google.com/file/d/1bv4a\\_mJo8X2JuzVeehv0nJl4Gcgw78\\_P/view?usp=sharing](https://drive.google.com/file/d/1bv4a_mJo8X2JuzVeehv0nJl4Gcgw78_P/view?usp=sharing)



*Kaggle Dogs vs. Cats Redux: Kernels Edition:*  
<https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data>

# Predicting an Image's Category

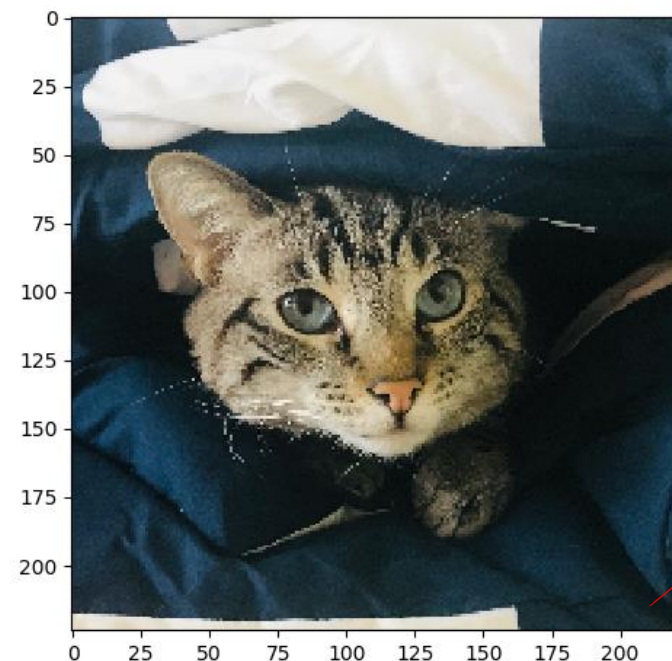
모듈 importing:

```
import tensorflow as tf
from tf.keras.applications.resnet50 import preprocess_input, decode_predictions
from tf.keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
```

이미지 load & display:

```
img_path = "data/sample-images/cat.jpg"
img = image.load_img(img_path, target_size=(224, 224))
plt.imshow(img)
plt.show()
```

Plot showing the contents  
of the input file



Keras에 이미지를 입력하기 전에 표준 형식으로 변환해야:

- **사전 훈련된 모델**은 입력이 특정 크기(224 x 224 픽셀)일 것으로 예상하기 때문

대부분의 딥러닝 모델은 이미지들의 배치(batch)를 입력으로 기대:

- 하지만 이미지가 하나만 있으면 어떻게 할까? 물론 하나의 이미지를 배치 파일로 생성! 즉, 본질적으로 하나의 객체로 구성된 배열을 만들

이것을 보는 또 다른 방법은 차원 수를 3개(이미지의 RGB 3개 채널을 나타냄)에서 4개(배열 자체의 길이에 대한 추가)로 확장하는 것:

- 각각 3개의 채널(RGB)을 포함하는 224 x 224 픽셀 크기의 64개 이미지 배치의 경우 해당 배치를 나타내는 객체의 모양은 64 x 224 x 224 x 3
- 아래 코드에서 224 x 224 x 3 이미지를 하나만 사용하는 경우 차원을 3에서 4로 **확장**하여 해당 이미지의 배치를 만들. 이 새로 생성된 배치의 모양은 1 x 224 x 224 x 3:

```
img_array = image.img_to_array(img)
img_batch = np.expand_dims(img_array, axis=0) # Increase the number of dimensions
```



머신 러닝에서 모델은 일관된 범위 내의 데이터를 제공할 때 가장 잘 수행된다.

- 일반적인 범위 —  $[0, 1]$  또는  $[-1, 1]$
- 이미지 픽셀 값이 0에서 255 사이인 경우 입력 이미지에 대해 Keras의 `preprocess_input` 함수를 실행하면 각 픽셀이 표준 범위로 정규화됨
- 정규화 또는 특징 확장은 딥 러닝에 적합하도록 이미지를 사전 처리하는 핵심 단계 중 하나임

이제 모델 등장!! ResNet-50이라는 CNN(Convolutional Neural Network) 사용:

- 첫 번째 질문 — "모델은 어디에서 찾을 수 있나?"
- Keras에서 단일 함수 호출로 우리에게 다양한 사전 훈련된 모델 제공
- `ResNet50()` 함수 호출 => Keras에서 모델 다운로드!!

```
model = tf.keras.applications.resnet50.ResNet50()
```



## Predicting an Image's Category

함수 전체 코드:

```
def classify(img_path):  
    img = image.load_img(img_path, target_size=(224, 224))  
    model = tf.keras.applications.resnet50.ResNet50()  
    img_array = image.img_to_array(img)  
    img_batch = np.expand_dims(img_array, axis=0)  
    img_preprocessed = preprocess_input(img_batch)  
    prediction = model.predict(img_preprocessed)  
    print(decode_predictions(prediction, top=3)[0])
```

```
classify("data/sample-images/cat.jpg")
```

```
[('n02123045', 'tabby', 0.50009364),  
 ('n02124075', 'Egyptian_cat', 0.21690978),  
 ('n02123159', 'tiger_cat', 0.2061722)]
```



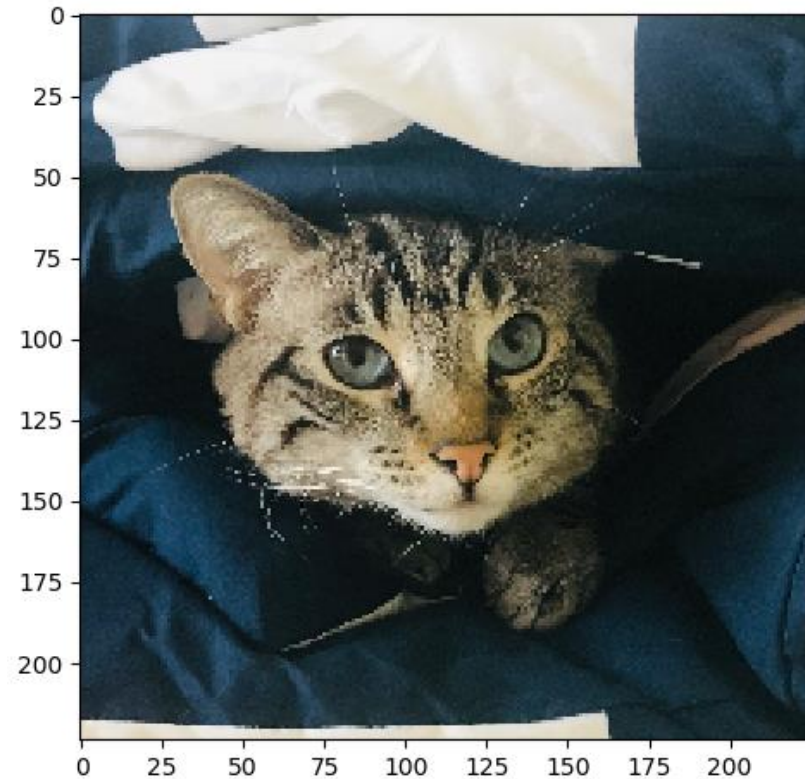


# 실습과제 19-1

제 19강 실습과제 19-1 Transfer Learning - Predicting an Image's Category.pdf

Predicting an Image's Category:

```
$ conda install tensorflow-datasets
```



# Investigating the Model

모델에서 예측을 얻었다. 그러나 그러한 예측을 이끌어 낸 요인은 무엇일까?

관련 질문:

- 모델이 학습된 데이터 세트는 무엇인가?
- 사용할 수 있는 다른 모델이 있는가? 얼마나 좋은가? 어디서 구할 수 있는가?
- 내 모델이 예측한 내용을 예측하는 이유는 무엇인가?

# ImageNet Dataset

ResNet-50이 학습된 **ImageNet** 데이터 세트를 조사해 보자.

이름에서 알 수 있듯이 ImageNet은 네트워크로 구성된 이미지 데이터 세트이다.

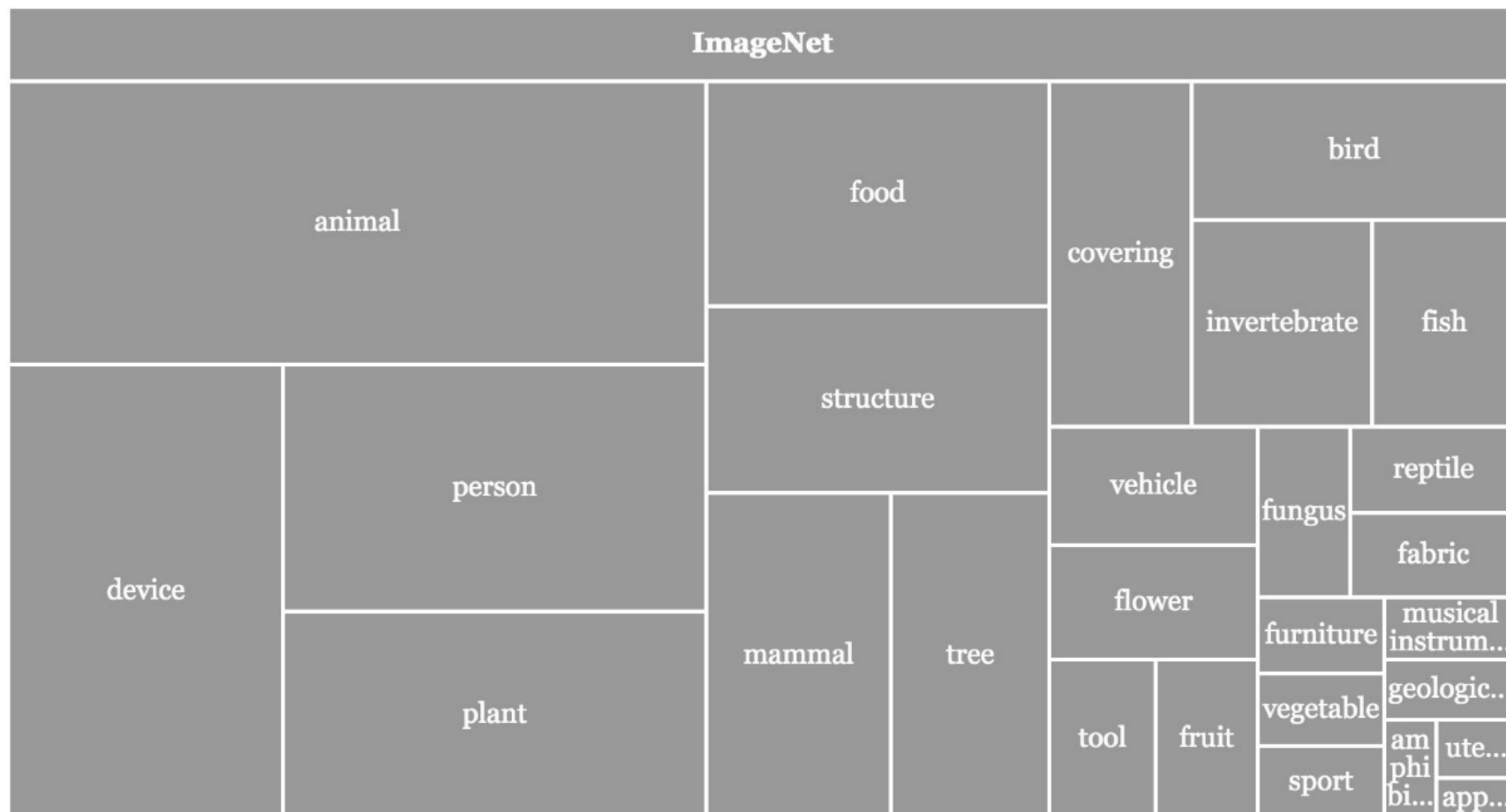
즉, 부모 노드가 해당 부모 내에서 가능한 모든 다양한 종류의 이미지 모음을 포함하도록 계층적 방식(예 : WordNet 계층 구조)으로 배열되어 있다.

예를 들어, "동물"부모 노드 내에는 물고기, 새, 포유류, 무척추 동물 등이 있다.

<http://image-net.org/download>

The most highly-used subset of ImageNet is the [ImageNet Large Scale Visual Recognition Challenge \(ILSVRC\)](#) 2012-2017 image classification and localization dataset. This dataset spans 1000 object classes and contains 1,281,167 training images, 50,000 validation images and 100,000 test images. This subset is available on [Kaggle](#).

ImageNet 데이터 세트에 포함된 다양한 상위 수준 항목을 이해하는데 도움을 주기 위한 트리 다이어그램



ImageNet 1000 class idx to human readable labels:  
<https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>

ImageNet

14,197,122 images, 21841 synsets indexed

[Home](#) [Download](#) [Challenges](#) [About](#)

Not logged in. [Login](#) | [Signup](#)

## ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

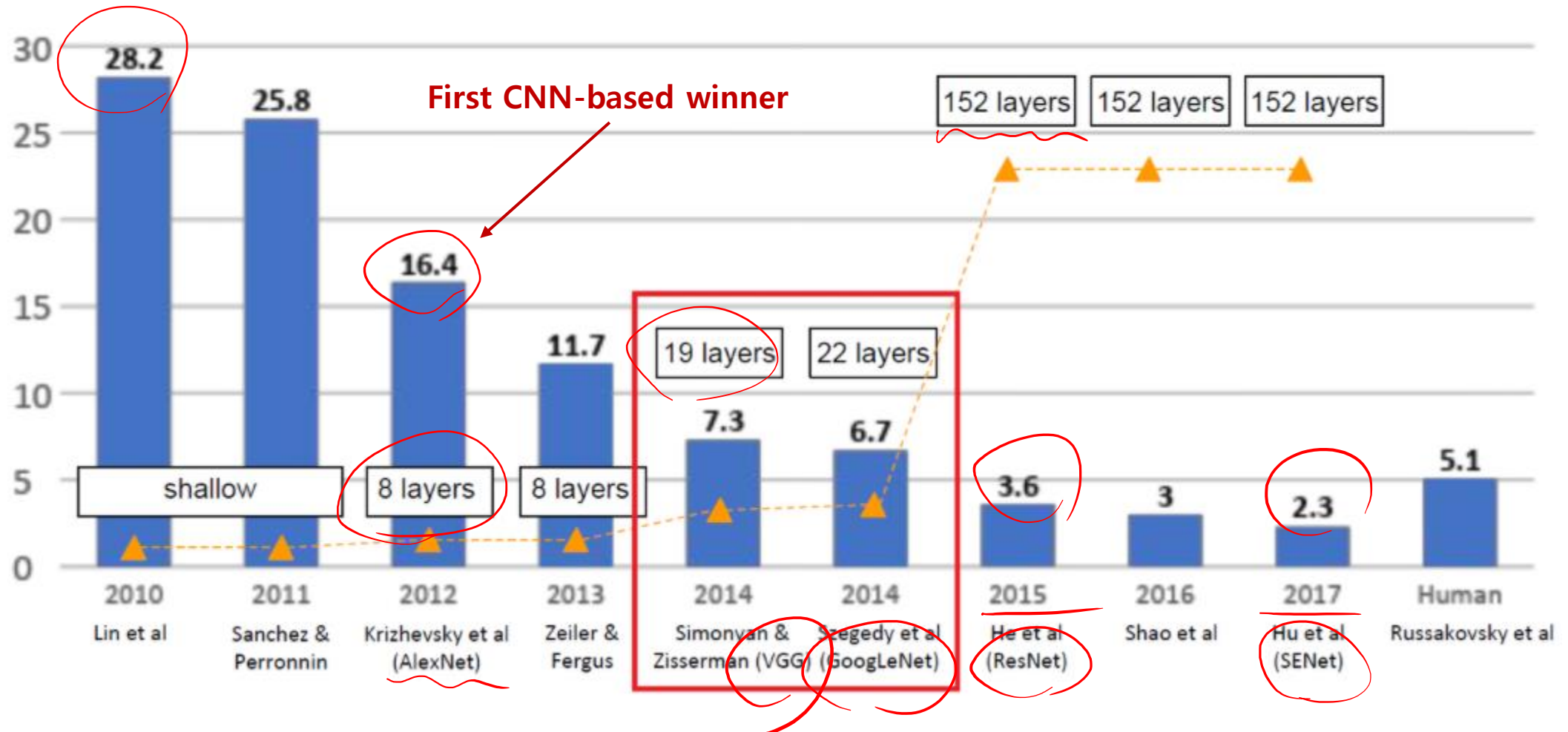
### Competition

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale. One high level motivation is to allow researchers to compare progress in detection across a wider variety of objects -- taking advantage of the quite expensive labeling effort. Another motivation is to measure the progress of computer vision for large scale image indexing for retrieval and annotation.

For details about each challenge please refer to the corresponding page.

- [ILSVRC 2017](#)
- [ILSVRC 2016](#)
- [ILSVRC 2015](#)
- [ILSVRC 2014](#)
- [ILSVRC 2013](#)
- [ILSVRC 2012](#)
- [ILSVRC 2011](#)
- [ILSVRC 2010](#)

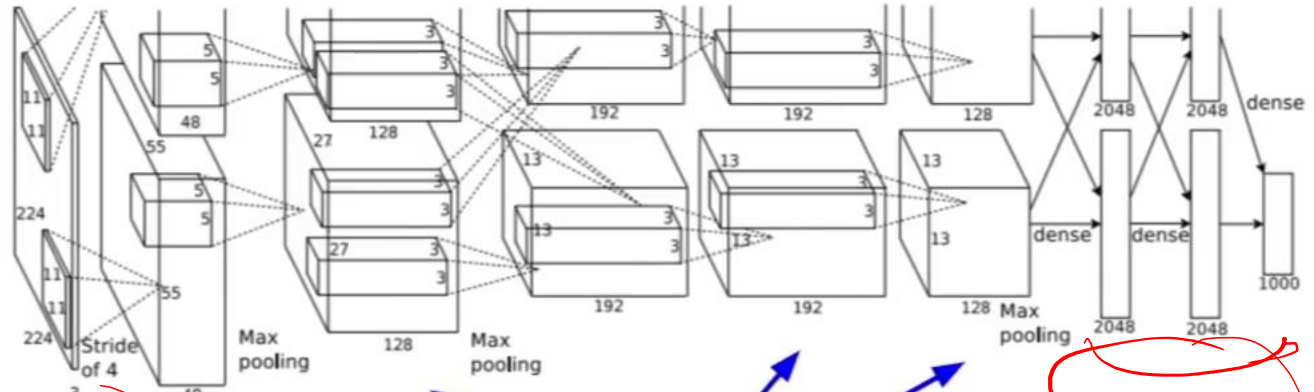
# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners





# Case Study: AlexNet

[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

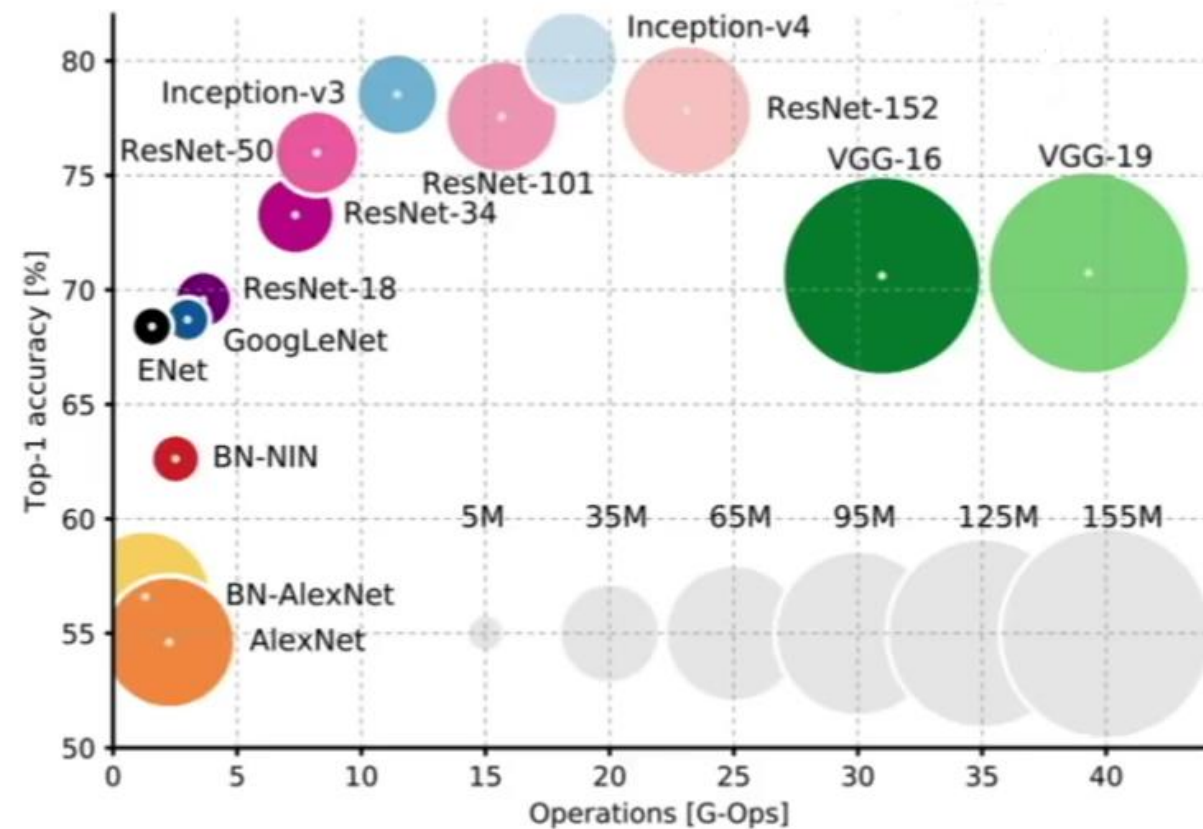
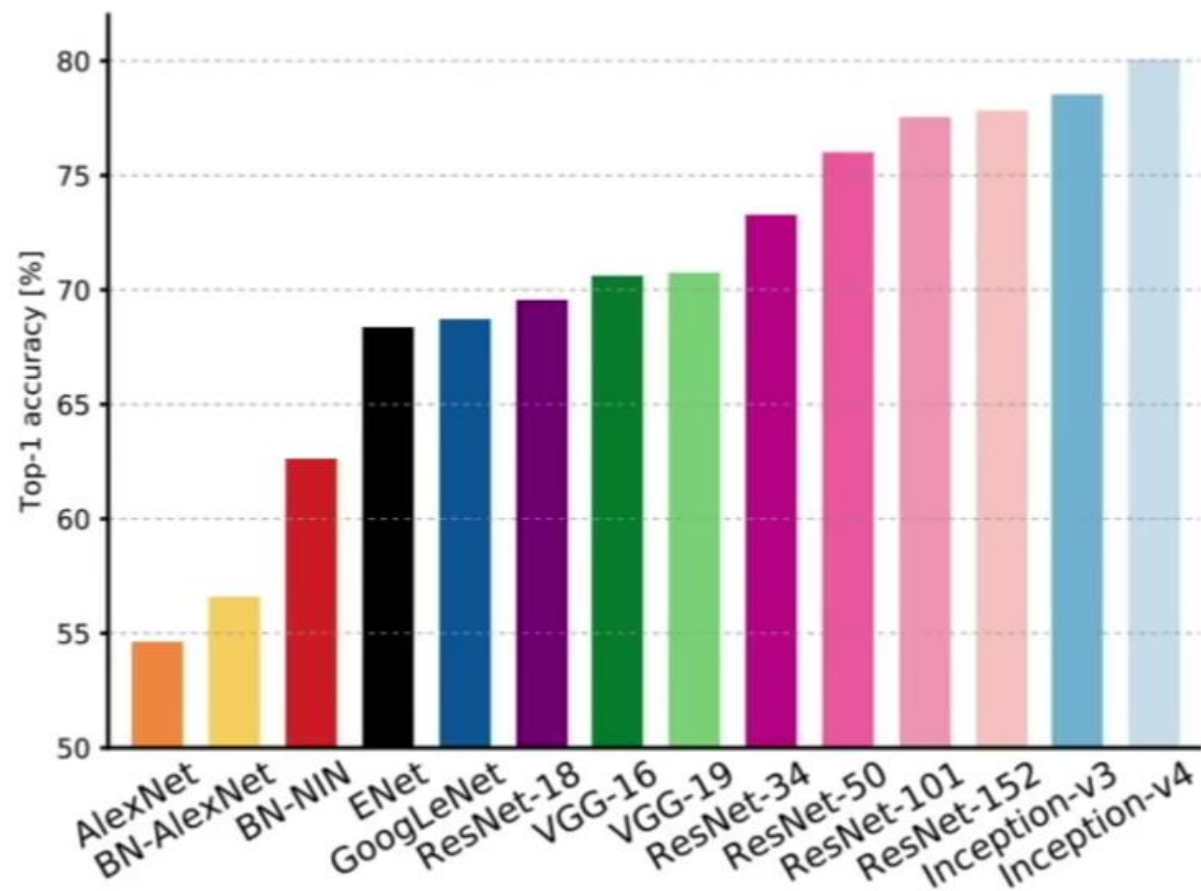
[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)

**CONV1, CONV2, CONV4, CONV5:**  
Connections only with feature maps  
on same GPU

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Comparing complexity...




An Analysis of Deep Neural Network Models for Practical Applications, 2017.



# Model Zoos in Keras

ImageNet 데이터셋에서 Keras 프레임워크를 사용하여 학습된 다양한 CNN 아키텍처의 모음

<https://keras.io/applications/>



| Model             | Size   | Top-1 Accuracy | Top-5 Accuracy | Parameters  | Depth |
|-------------------|--------|----------------|----------------|-------------|-------|
| Xception          | 88 MB  | 0.790          | 0.945          | 22,910,480  | 126   |
| VGG16             | 528 MB | 0.713          | 0.901          | 138,357,544 | 23    |
| VGG19             | 549 MB | 0.713          | 0.900          | 143,667,240 | 26    |
| ResNet50          | 98 MB  | 0.749          | 0.921          | 25,636,712  | -     |
| ResNet101         | 171 MB | 0.764          | 0.928          | 44,707,176  | -     |
| ResNet152         | 232 MB | 0.766          | 0.931          | 60,419,944  | -     |
| ResNet50V2        | 98 MB  | 0.760          | 0.930          | 25,613,800  | -     |
| ResNet101V2       | 171 MB | 0.772          | 0.938          | 44,675,560  | -     |
| ResNet152V2       | 232 MB | 0.780          | 0.942          | 60,380,648  | -     |
| InceptionV3       | 92 MB  | 0.779          | 0.937          | 23,851,784  | 159   |
| InceptionResNetV2 | 215 MB | 0.803          | 0.953          | 55,873,736  | 572   |
| MobileNet         | 16 MB  | 0.704          | 0.895          | 4,253,864   | 88    |
| MobileNetV2       | 14 MB  | 0.713          | 0.901          | 3,538,984   | 88    |
| DenseNet121       | 33 MB  | 0.750          | 0.923          | 8,062,504   | 121   |
| DenseNet169       | 57 MB  | 0.762          | 0.932          | 14,307,880  | 169   |
| DenseNet201       | 80 MB  | 0.773          | 0.936          | 20,242,984  | 201   |
| NASNetMobile      | 23 MB  | 0.744          | 0.919          | 5,326,716   | -     |
| NASNetLarge       | 343 MB | 0.825          | 0.960          | 88,949,818  | -     |

# Transfer Learning with Keras

Transfer learning in real life



Already Play Piano?  
Fine-tune Skills  
Effort = 3 days

Learning Melodica From Scratch?  
Effort = 3 months



딥 러닝의 세계에 적용 => called "Transfer Learning"

- 사전 학습된 모델을 활용함으로써 프로젝트를 상대적으로 빠르게 시작 가능

# Adapting Pretrained Models to New Tasks

딥러닝이 붐이 일게 된 이유:

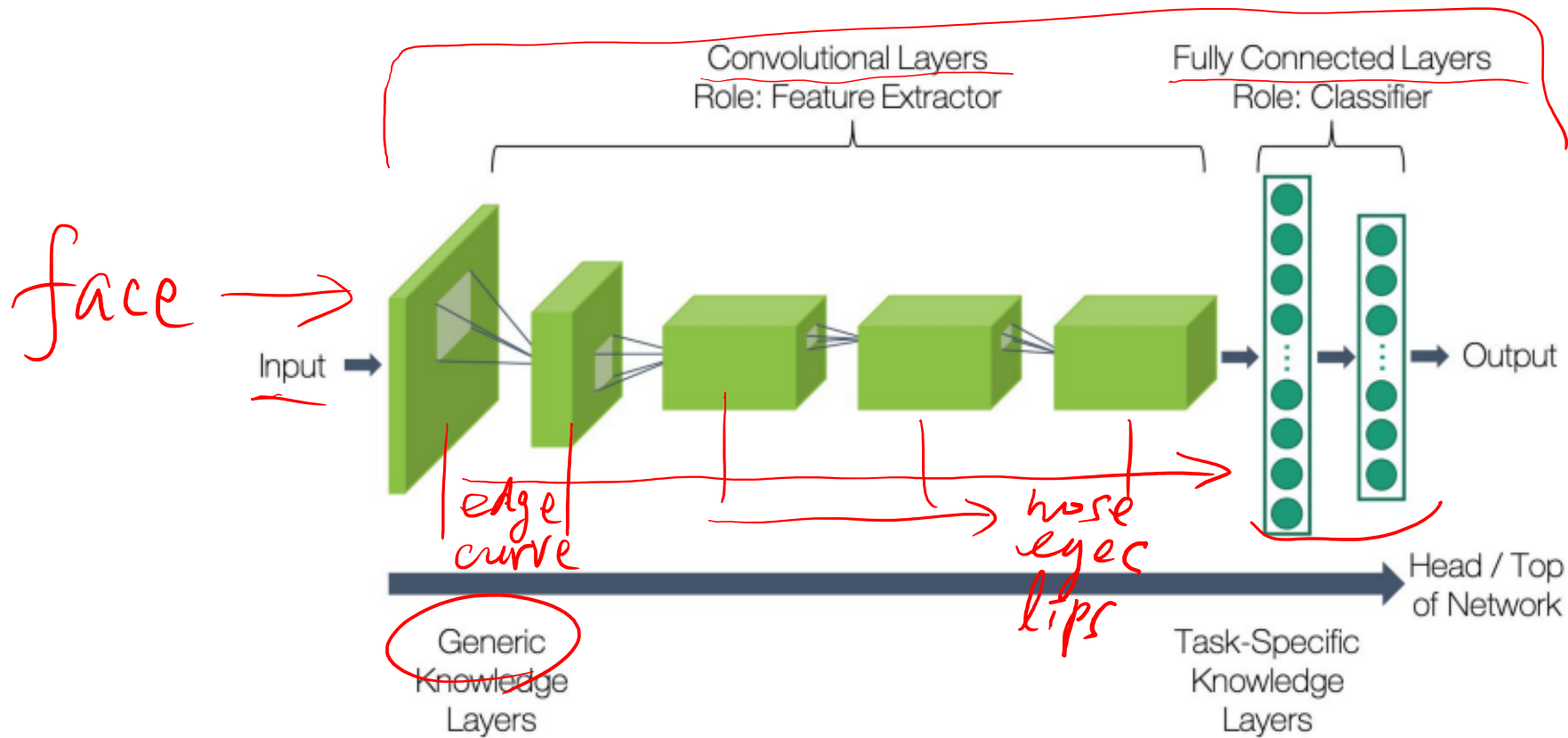
- ImageNet과 같이 더 크고 품질이 우수한 데이터 세트 이용 가능
- 더 나은 계산 가능; 즉, 더 빠르고 저렴한 GPU
- 더 나은 알고리즘 (모델 아키텍처, 최적화 프로그램 및 학습 절차)
- 학습시키는데 몇 개월이 걸릴 수 있는 **모델 재사용 가능**

4번째 이유 => 대중이 딥 러닝을 널리 채택하는 가장 큰 이유 중의 하나

- Transfer Learning 덕택 => 몇 분 안에 기존 모델을 현재 업무에 맞게 수정 가능

# A Shallow Dive into Convolutional Neural Networks

“모델” — 컴퓨터 비전을 위한 딥 러닝에서는 일반적으로 CNN이라는 특수한 유형의 신경망

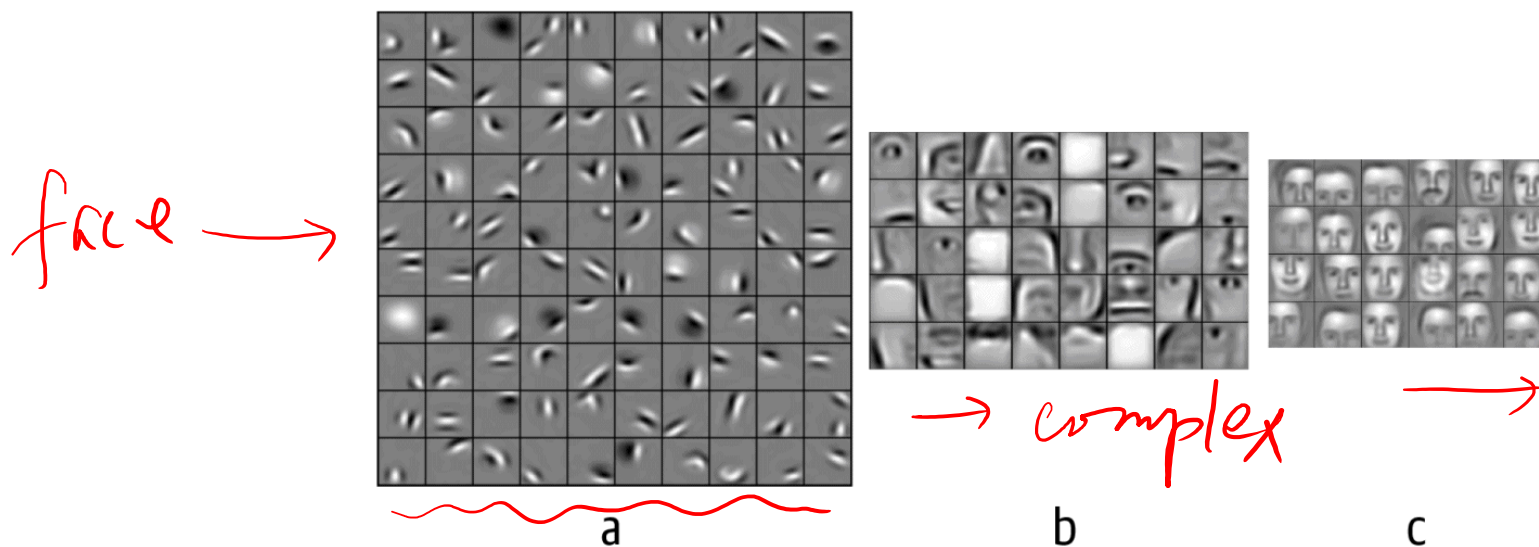


*A high-level overview of a CNN*

머신 러닝에서 — 데이터를 식별 가능한 특징 집합으로 변환한 다음 분류 알고리즘을 추가하여 분류해야

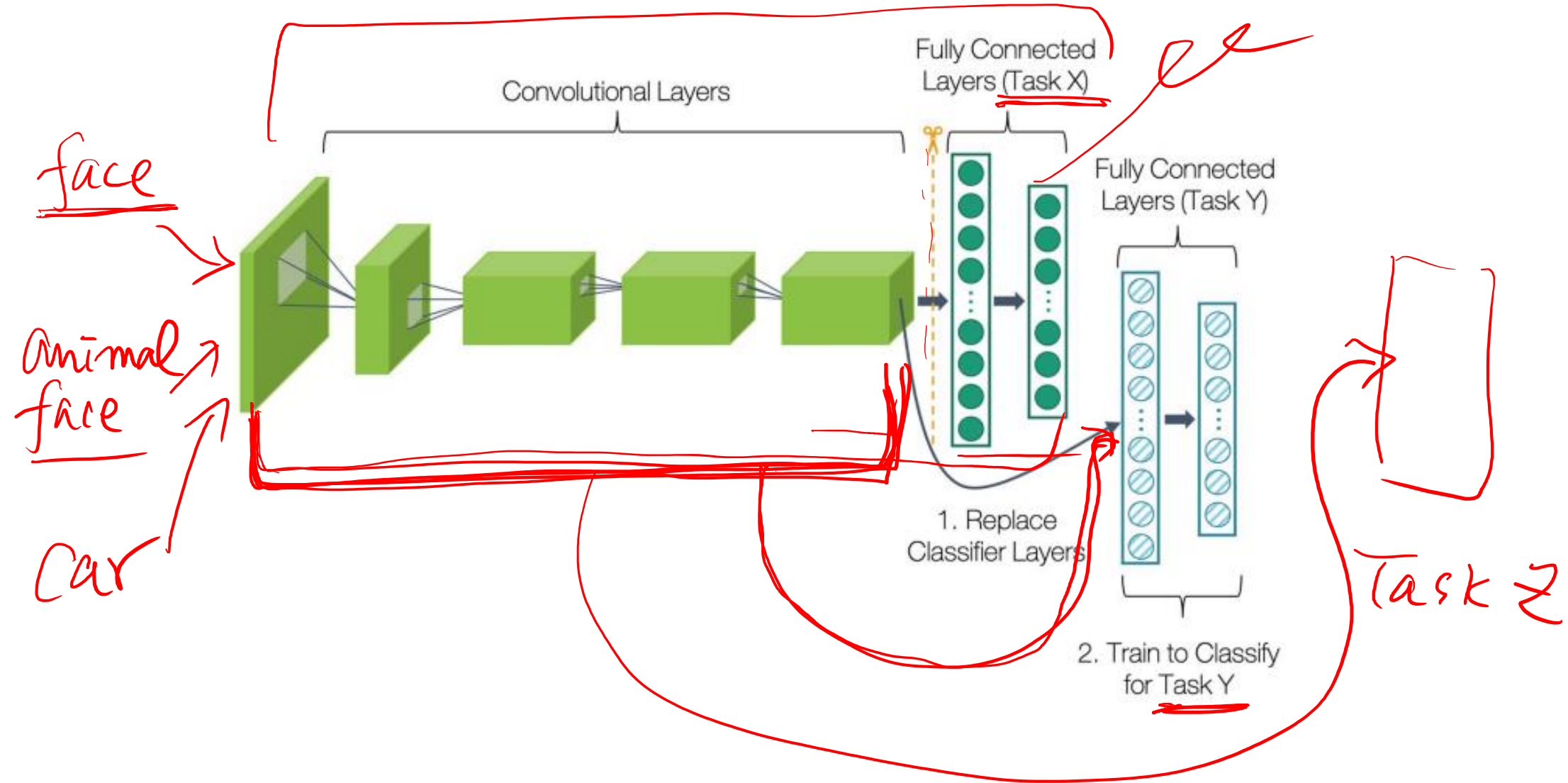
CNN — 마찬가지로!! 컨볼루션 레이어와 FC(완전 연결) 레이어의 두 부분으로 구성

- 컨볼루션 레이어의 역할 — 이미지의 많은 수의 픽셀을 가져와 훨씬 더 작은 표현, 즉 특징으로 변환하는 것 => 특징 추출기 역할
- FC 레이어 — 이러한 특성을 확률로 변환. 즉, 실제로 히든 레이어가 있는 신경망 => 분류기 역할

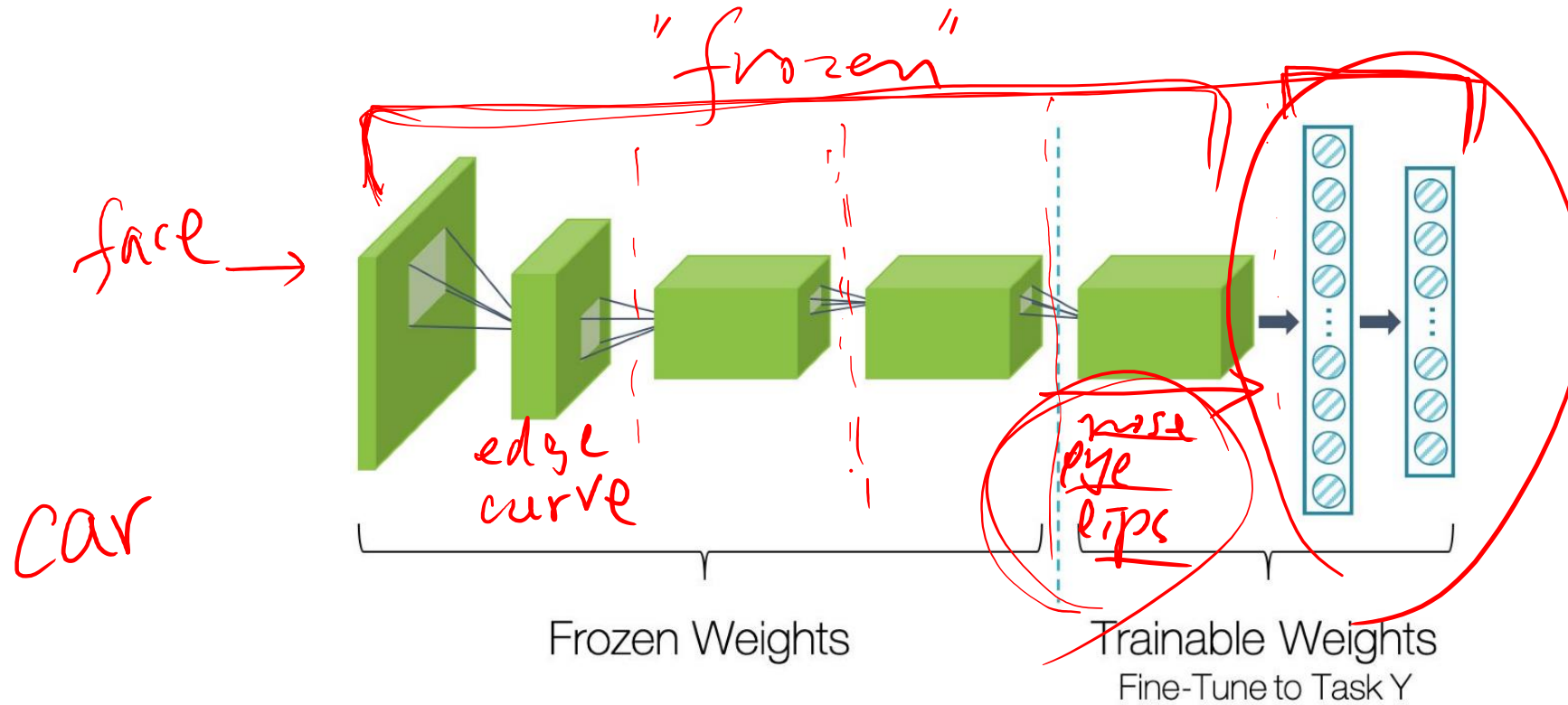


(a) Lower-level activations, followed by (b) midlevel activations and (c) upper-layer activations (image source: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, Lee et al., ICML 2009)

# Transfer Learning



# Fine Tuning



# Example — Transfer Learning with Keras

Fashion MNIST dataset에 transfer learning을 적용한 예:

- Fashion MNIST dataset이 8개의 클래스만 포함한다고 가정 (sandal과 shirt 클래스를 제외한 모든 클래스)
- 누군가 해당 데이터셋에 대해 Keras 모델을 구축해서 학습시켜 놓았다. (>90% accuracy)
- 이것을 **모델 A**라 하자.

이제 다른 작업을 다뤄보자:

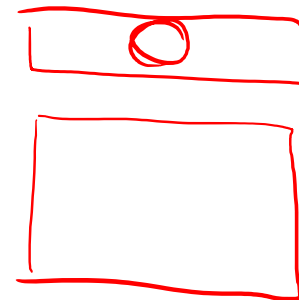
- sandal과 shirt 이미지들에 대해 binary classifier를 학습시키는 일  
(positive=shirt, negative=sandal)
- 현재 확보된 데이터셋은 매우 작다 (200 labeled images)
- 새 모델(**모델 B**)을 모델 A와 동일한 아키텍처로 학습시켜 보면 97.2%의 성능을 얻을 수 있다
- 모델 B의 성능을 더 높여 보자...



1. 모델 A 로딩 => 모델 A의 레이어에 기반한 새 모델 생성

⇒ 출력 레이어를 제외한 모든 레이어 재사용

```
model_A = keras.models.load_model("my_model_A.h5")  
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])  
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```



2. model\_A와 model\_B\_on\_A : 일부 레이어 공유

⇒ model\_B\_on\_A를 학습시킬 때 model\_A에 영향을 줄 수 있으므로 복제

```
model_A_clone = keras.models.clone_model(model_A)  
model_A_clone.set_weights(model_A.get_weights())
```

### 3. Task B에 대해 model\_B\_on\_A를 학습시킬 수 있음

- ⇒ 새 출력 레이어가 랜덤하게 초기화되었기 때문에 첫 몇 epoch 중에는 large error가 출력될 것임
- ⇒ Large error는 재사용된 가중치를 파괴시킴
- ⇒ 해결책: 처음 몇 epoch 동안에는 재사용된 레이어를 잠그고 새 레이어에게 얼마동안 합리적 학습을 할 수 있도록 기회를 줌
- ⇒ 이를 위해 모든 레이어의 trainable 속성을 False로 설정하고 모델 컴파일:

```
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = False  
  
model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",  
                     metrics=["accuracy"])
```

※ 레이어를 잠그거나 잠금 해제한 후에는 모델을 항상 다시 컴파일해야 한다!

4. 몇 epoch 동안 모델 학습 => 재사용된 레이어 잠금 해제 => Task B에 대해 재사용된 레이어 학습 계속  
⇒ 재사용된 레이어를 잠금 해제한 후에는 학습률을 줄여서 재사용된 가중치에 피해가 가지 않도록 해야 함

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,  
                           validation_data=(X_valid_B, y_valid_B))  
  
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = True  
  
optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-2  
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,  
                     metrics=["accuracy"])  
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,  
                           validation_data=(X_valid_B, y_valid_B))
```

그래서 최종 결과는?

99.25%

이 모델의 테스트 정확도는 99.25%이다. 즉, 전이 학습이 오류율을 2.8 %에서 거의 0.7 %로 줄였다!  
4분의 1 수준이다!!

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

```
(100 - 97.05) / (100 - 99.25)
```

3.933333333333337

여러분도 똑같은 결과가 나오나요? 아마도 아닐 것이다!!

사실 위의 결과는 최상의 결과가 나올때까지 데이터를 쥐어짖기 때문에 나온 것이다.

랜덤 시드 등의 하이퍼파라미터를 바꾸면 더 나쁜 결과가 나올 수도 있다...

## 실습과제 19-2

제 19강 실습과제 19-2 Transfer Learning - Fashion MNIST dataset.pdf

```
$ pip install pillow  
$ pip install tf-explain
```

# Building a Custom Classifier in Keras with Transfer Learning

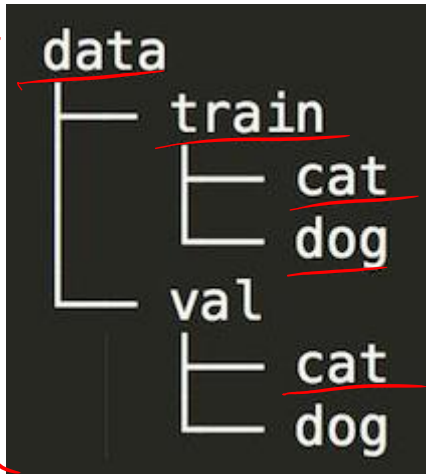
Classifier 구축 단계:

1. Organize the data.
2. Build the data pipeline.
3. Augment the data.
4. Define the model.
5. Train and test.

# Organize the Data

Organize the data:

Download labeled images of cats and dogs and then divide the images into training and validation folders.



train.zip 파일을 풀어서

← 각 dog, cat 디렉토리에 앞부분 250개의 이미지 저장

← 각 dog, cat 디렉토리에 251번째부터 250개의 이미지 저장

=> P.4 참조

# Build the Data Pipeline

패키지 import:

```
import tensorflow as tf
from tf.keras.preprocessing.image import ImageDataGenerator
from tf.keras.models import Model
from tf.keras.layers import Input, Flatten, Dense, Dropout,
GlobalAveragePooling2D
from tf.keras.applications.mobilenet import MobileNet, preprocess_input
import math
```

ResNet50

Configuration:

```
TRAIN_DATA_DIR = 'data/train/'
VALIDATION_DATA_DIR = 'data/val/'
TRAIN_SAMPLES = 500
VALIDATION_SAMPLES = 500
NUM_CLASSES = 2
IMG_WIDTH, IMG_HEIGHT = 224, 224
BATCH_SIZE = 64
```

```
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2
```

cat/dog

14M

16M



## Build the Data Pipeline

```
train_datagen = ImageDataGenerator(preprocessing_function=preprocess_input,  
                                   rotation_range=20,  
                                   width_shift_range=0.2,  
                                   height_shift_range=0.2,  
                                   zoom_range=0.2)  
  
val_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)  
  
train_generator = train_datagen.flow_from_directory(  
    TRAIN_DATA_DIR,  
    target_size=(IMG_WIDTH, IMG_HEIGHT),  
    batch_size=BATCH_SIZE,  
    shuffle=True,  
    seed=12345,  
    class_mode='categorical')  
  
validation_generator = val_datagen.flow_from_directory(  
    VALIDATION_DATA_DIR,  
    target_size=(IMG_WIDTH, IMG_HEIGHT),  
    batch_size=BATCH_SIZE,  
    shuffle=False,  
    class_mode='categorical')
```

# Model Definition

모델 정의 - 사전 학습된 MobileNetV2 재사용:

MobileNetV2

```
def model_maker():  
    base_model = MobileNet(include_top=False, input_shape =  
        (IMG_WIDTH, IMG_HEIGHT, 3))  
    for layer in base_model.layers[:]:  
        layer.trainable = False # Freeze the layers  
    input = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))  
    custom_model = base_model(input)  
    custom_model = GlobalAveragePooling2D()(custom_model)  
    custom_model = Dense(64, activation='relu')(custom_model)  
    custom_model = Dropout(0.5)(custom_model)  
    predictions = Dense(NUM_CLASSES, activation='softmax')(custom_model)  
    return Model(inputs=input, outputs=predictions)
```

# Train the Model

model\_maker() 함수를 이용한 커스텀 모델 생성:

```
model = model_maker()
model.compile(loss='categorical_crossentropy',
               optimizer= tf.train.Adam(lr=0.001),
               metrics=['acc'])
num_steps = math.ceil(float(TRAIN_SAMPLES)/BATCH_SIZE)
model.fit_generator(train_generator,
                    steps_per_epoch = num_steps,
                    epochs=10,
                    validation_data = validation_generator,
                    validation_steps = num_steps)
```

S&D

# Test the Model

load\_model() 함수를 이용한 모델 로딩:

```
from tf.keras.models import load_model  
model = load_model('model.h5')
```

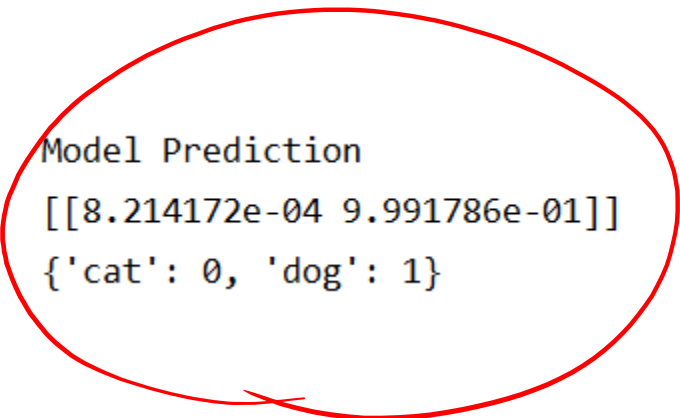
샘플 이미지 로딩 후 결과 보기:

```
img_path = 'data/sample-images/dog.jpg'  
img = image.load_img(img_path, target_size=(224, 224))  
img_array = image.img_to_array(img)  
expanded_img_array = np.expand_dims(img_array, axis=0)  
preprocessed_img = expanded_img_array / 255. # Preprocess the image  
prediction = model.predict(preprocessed_img)  
print(prediction)  
print(validation_generator.class_indices)
```

## 실습과제 19-3

제 19강 실습과제 19-3 Transfer Learning in 30 Lines with Keras - Cats Versus Dogs.pdf

★ 반드시 결과 값 및 결과로 나온 모든 그래프에 대해 자세한 분석을 하시오.



```
Model Prediction  
[[8.214172e-04 9.991786e-01]]  
{'cat': 0, 'dog': 1}
```

# Faster Optimizers

매우 큰 심층 신경망을 학습시키는 일은 고통스럽게 느낄 수 있다.

지금까지 학습 속도를 높이고 더 나은 솔루션에 도달하는 여러 가지 방법을 살펴 보았다:

- 연결 가중치에 대한 좋은 초기화 전략 적용
- 좋은 활성화 함수 사용
- 배치 정규화 사용
- 사전 훈련된 네트워크의 일부 재사용

SGD

또 다른 엄청난 속도 향상은 일반 Gradient Descent Optimizer보다 더 빠른 Optimizer를 사용하는 것이다!!

- Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, Adam 및 Nadam 최적화 등

## **Momentum optimization**

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

## **Nesterov Accelerated Gradient**

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

## **AdaGrad**

```
optimizer = keras.optimizers.Adagrad(lr=0.001)
```

## **RMSProp**

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

## **Adam Optimization**

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

## **Adamax Optimization**

```
optimizer = keras.optimizers.Adamax(lr=0.001, beta_1=0.9, beta_2=0.999)
```

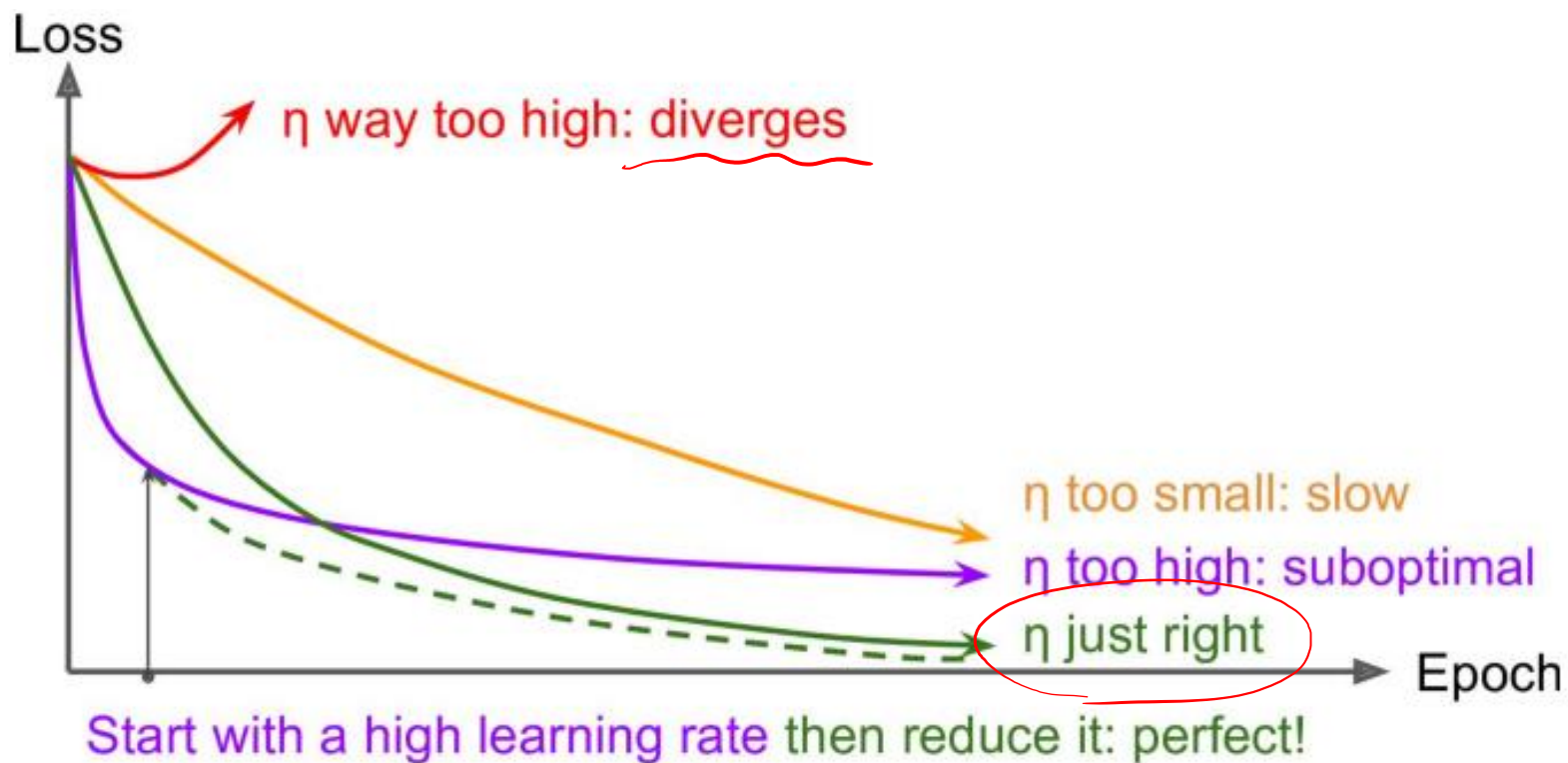
## **Nadam Optimization**

```
optimizer = keras.optimizers.Nadam(lr=0.001, beta_1=0.9, beta_2=0.999)
```



# Learning Rate Scheduling

좋은 학습률을 찾는 일은 중요!



일정한 학습률을 사용하는 것보다 더 좋은 방법이 있다!

- 처음에는 큰 값으로 시작했다가 점차 작은 값으로 줄여 나가는 방법
- 일정한 값을 가지고 학습시킬 때보다 더 빨리 수렴

학습 중에 학습률을 줄이기 위한 다양한 전략:

- Power scheduling : 학습률을  $t$ :  $\eta(t) = \eta_0 / (1 + \frac{t}{s})^c$ 로 설정
- Exponential scheduling : 학습률을  $\eta(t) = \eta_0 0.1^{t/s}$ 로 설정
- Piecewise constant scheduling : 처음 몇 epoch 동안에는 일정한 학습률 (예:  $\eta_0 = 0.1$ ) 사용, 나머지 많은 epoch 동안 더 작은 학습률 (예:  $\eta_1 = 0.001$ ) 사용
- Performance scheduling :  $N$  스텝마다 validation error를 측정해서 error가 더 이상 떨어지지 않을 때 학습률을  $\lambda$  factor 만큼 줄임
- 1cycle scheduling : 학습률  $\eta_0$ 로 시작해서 학습 중간 부근까지  $\eta_1$ 으로 선형적으로 증가시킴. 남은 학습의 반까지는 다시  $\eta_0$ 까지 선형적으로 줄임. 마지막 몇 epoch은 학습률을 크게 줄임

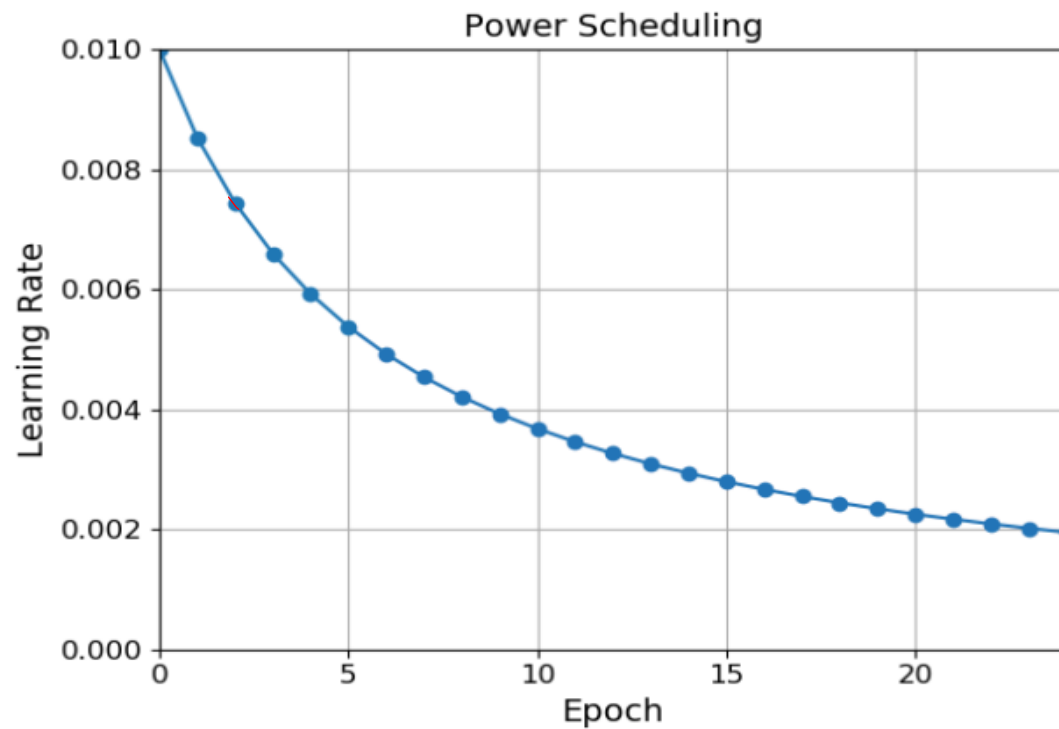
# Learning Rate Scheduling

Keras에서 Power scheduling( $t$ :  $\eta(t) = \eta_0 / (1 + \frac{t}{s})^c$ )을 구현하는 가장 쉬운 방법:

- Optimizer를 생성할 때 단순히 decay 하이퍼파라미터를 설정하면 됨

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

decay : s(step)의 역함수, Keras : c = 1이라고 가정



# Avoiding Overfitting Through Regularization

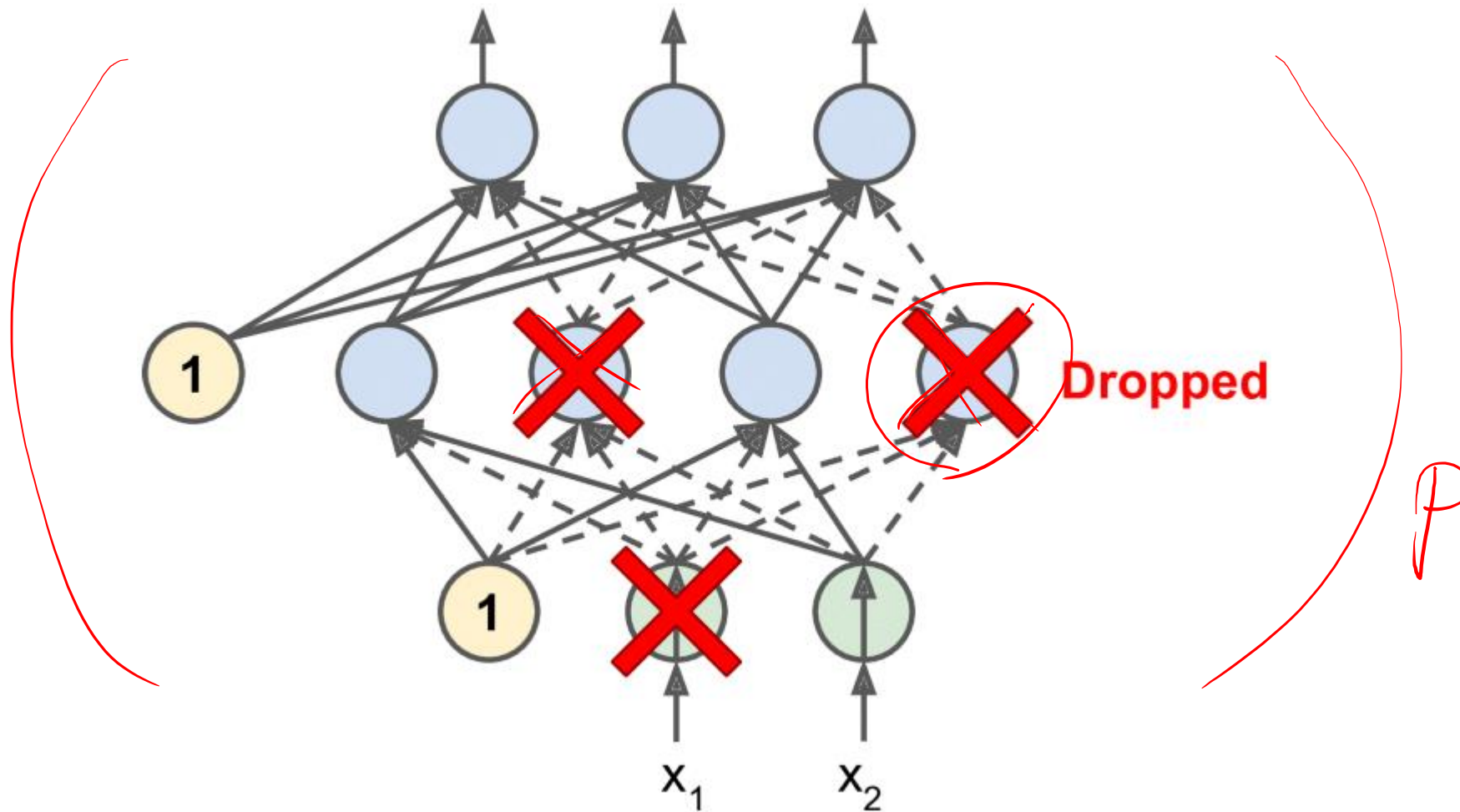
Regularization 기법들:

- Early stopping
- Batch Normalization
- $\ell_1$  and  $\ell_2$  regularization
- **dropout**
- max-norm regularization.

# Dropout

## Dropout

- DNN을 위한 가장 인기있는 regularization 기법 중의 하나
- Geoffrey Hinton 교수가 제안 (2012)
- 알고리즘:
  - ✓ 모든 학습 단계에서 출력 뉴런을 제외한 모든 뉴런들이  $p$ 의 확률로 누락될 기회를 가짐
  - ✓ 선정된 뉴런은 해당 학습 단계에서 완전히 무시됨
  - ✓ 다음 단계에서는 다시 활성화될 수 있음
  - ✓ Hyperparameter  $p$  (dropout rate) : 전형적으로 10% - 50% 사이로 설정



드롭 아웃 정규화를 사용하면 각 학습 반복에서 하나 이상의 레이어(출력 레이어를 제외한)의 모든 뉴런의 임의의 하위 집합이 "삭제"된다. 이 뉴런은 이번 반복에서 0을 출력한다(점선 화살표로 표시)

Keras를 이용한 dropout 구현:

- `keras.layers.Dropout` 이용
- Dropout rate = 0.2
- 그래도 모델이 overfitting 증상을 보이면 dropout rate를 증가시킴
- Underfitting 증상을 보이면 dropout rate를 감소시킴

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

