8장. 케라스 커스터마이제이션

요약 정리

1. 케라스 모델 구성 방법은 **Sequential(), 서브클래싱, 함수형 API**가 있습니다. 권장 방법은 함수형 API입니다.
2. 함수형 API를 활용하면 **다중 입출력 모델 구조를 쉽게 구성**할 수 있습니다.
3. 잔차 연결과 인셉션 모듈 구조는 많은 모델에서 사용되고 있습니다. 가장 기본적인 형태를 알아두면, 조금 변형된 형태를 이해하기에 더욱 수월할 것입니다.
4. 전이 학습의 사용을 위해 케라스는 ImageNet을 학습한 다양한 모델을 제공하고 있습니다. **tensorflow.keras.applications**에서 확인할 수 있습니다.
5. **커스텀 제네레이터와 다중 입력 모델 구조**를 활용하여 다중 레이블 문제를 다시 해결해보았습니다.
6. 네 가지 **케라스 콜백**을 사용해보았습니다.
    1. ModelCheckpoint
    2. EarlyStopping
    3. ReduceLROnPlateau
    4. TensorBoard

8장의 내용?

- 문제를 해결하기 위해서는 텐서플로우가 제공하는 다양한 함수를 사용해야 할뿐만 아니라 이를 응용하여 본인만의 함수(Customization)를 만들 필요가 있음

- 이번 장에서는 다음 내용을 다뤄봅니다.
  - 커스터마이제이션: 케라스 층, 활성화 함수, 손실 함수, 평가지표, 케라스 콜백
  - 1x1 컨볼루션
  - 초급 단계를 위해 한걸음 더

커스터마이제이션

- 베이스라인 결과를 도출하기 위해서 가장 먼저 수행하는 것은 **간단명료한 프로토타입 모델**을 만드는 것
  - 만들어진 모델이 어느 정도의 출발선상에 놓여있는지를 파악하고 다음 단계를 결정
  - 모델 성능 향상을 위해 특정 손실 함수나 케라스 층이 필요할 것 → **Customization**

- 다섯 가지 항목에 대해 기본적인 커스터마이제이션을 시도해보자
  - 케라스 층(Keras Layer)
  - 활성화 함수(Activation Function)
  - 손실 함수(Loss Function)
  - 평가지표(Metrics)
  - 케라스 콜백(Keras Callbacks)

커스터마이제이션: 케라스 층

- 케라스 층을 커스텀화하는 두 가지 방법
  - **Lambda 층**: 가중치가 학습에 사용되지 않을 경우
  - **Layer 클래스를 상속한 커스텀 케라스 층**: 가중치가 학습에 사용될 경우

- Lambda층
  - **정의한 함수를 Layer 객체로 변환**해주어 케라스 층처럼 사용할 수 있음
  - 프로그래밍 언어의 Lambda 표현 방식과 동일

```
05  # 나만의 함수를 정의합니다.
06  def custom_f(x):
07      x_mean = K.mean(x)
08      x_std = K.std(x)
09
10      return (x - x_mean) / x_std
11
12  inputs = Input(shape = (5, ))
13  # Lambda층을 통해 Layer으로 변환하여 연결합니다.
14  x = Lambda(custom_f)(inputs)
15  # 기존에 사용하던 층과 연결하여 사용할 수 있습니다.
16  # x = Dense(32, activation = 'relu')(x)
```

커스터마이제이션: 케라스 층

- 커스텀 케라스 층, 세 가지 메서드를 정의하여 사용
  - **build(input_shape)**: 학습할 가중치 정의
  - **call(x)**: 층에서 수행할 핵심 연산 정의
  - **compute_output_shape(input_shape)**: 출력 형태 정의

- 간단한 버전의 Dense층 구현

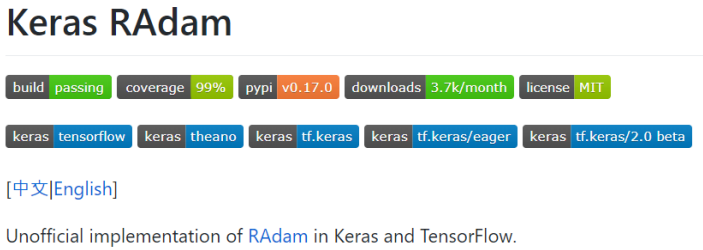
```
05 # 커스텀층을 정의합니다.
06 class CustomLayer(Layer):
07     def __init__(self, num_hidden):
08         super(CustomLayer, self).__init__()
09         self.num_hidden = num_hidden
10 
11     # 가중치를 정의합니다.
12     def build(self, input_shape):
13         self.kernels = self.add_weight('kernels',
14                                        shape = [int(input_shape[-1]), self.num_hidden])
15 
16         self.bias = self.add_weight('bias', shape = [self.num_hidden])
17 
18     # 수행할 연산을 정의합니다.
19     def call(self, x):
20         return relu(tf.matmul(x, self.kernels) + self.bias)
21 
22     # 출력값의 형태를 명시해줍니다.
23     def compute_output_shape(self, input_shape):
24         return [input_shape[0], self.num_hidden]
25 
26 # 모델을 구성합니다.
27 inputs = Input(shape = (5, ))
28 x = CustomLayer(32)(inputs)
29 model = Model(inputs = inputs, outputs = x)
```

커스터마이제이션: 활성화 함수

- 활성화 함수 커스터마이제이션은 매우 단순하며, 쉬움

- 자주 사용되어 쉽게 접근할 수 있는 활성화 및 최적화 함수
  - sigmoid, softmax, tanh, ReLU, …
  - SGD, Momentum, RMSprop, Adam, NAdam, …

- **Mish 활성화 함수와 RAdam 최적화 함수를 사용해보자**
  - 케라스에 공식적으로 포함되어 있지 않기때문에 직접 정의해서 사용해야 함



커스터마이제이션: 활성화 함수

- 방법 1: **Activation() 함수** 사용

```
# Activation 함수에 전달하는 방법입니다.
def Mish(x):
    return x * K.tanh(K.softplus(x))

inputs = Input(shape = (28, 28))
x = Flatten()(inputs)
x = Dense(50)(x)
x = Activation(Mish)(x)
x = Dense(30)(x)
x = Activation(Mish)(x)
x = Dense(10, activation = 'softmax')(x)
```

커스터마이제이션: 활성화 함수

- 방법 2: **커스텀 객체 목록** 사용
  - get_custom_objects() 함수: 케라스 전역 커스텀 객체 목록

```
05  from tensorflow.keras.utils import get_custom_objects
```

```
07  # 단순 클래스를 정의합니다.
08  class Mish(Activation):
09      def __init__(self, activation, **kwargs):
10          super(Mish, self).__init__(activation, **kwargs)
11          self.__name__ = 'Mish'
12
13  def mish(x):
14      return x * K.tanh(K.softplus(x))
15
16  # 케라스의 객체 목록에 해당 함수를 문자열로 등록합니다.
17  get_custom_objects().update({'mish': Mish(mish)})
```

```
19  # 문자열로 전달하여 사용하는 방법입니다.
20  inputs = Input(shape = (28, 28))
21  x = Flatten()(inputs)
22  x = Dense(50)(x)
23  x = Activation('mish')(x)
24  x = Dense(30)(x)
25  x = Activation('mish')(x)
26  x = Dense(10, activation = 'softmax')(x)
```

  - 커스텀 객체 범위를 지정해서 사용 가능

```
10  # with 구문을 사용한 커스텀 객체 정의 및 사용
11  with custom_object_scope({'mish':Mish}):
```

커스터마이제이션: RAdam 사용하기

- RAdam 설치
  - 깃허브 저장소 참고

```
# 설치를 진행합니다.
pip install keras-rectified-adam
```

```
from keras_radam import RAdam

model.compile(optimizer = RAdam(), loss = 'mse')
```

커스터마이제이션: 손실 함수

- 손실 함수는 모델이 어떠한 방향으로 학습해야 하는지를 결정하는 매우 중요한 요소
  - 케라스에서는 **y_true, y_pred 인자**를 잘 사용하면, 쉽게 구현 가능
  - 구현 후, compile() 함수의 loss 인자에 전달

```
01  import tensorflow as tf
02
03  # mse
04  def mse(y_true, y_pred):
05      return tf.reduce_mean(tf.math.square(y_true - y_pred), axis = -1)
06
07  # binary_crossentropy
08  def binary_crossentropy(y_true, y_pred):
09      y_pred = tf.clip_by_value(y_pred, tf.epsilon(), 1.0 - tf.epsilon())
10      total_loss = -y_true * tf.math.log(y_pred) - (1.0 - y_true)
                    * tf.math.log(1.0 - y_pred)
11
12      return total_loss
13
14  # categorical_crossentropy
15  def categorical_crossentropy(y_true, y_pred):
16      y_pred = y_pred / tf.reduce_sum(y_pred, axis = -1, keepdims = True)
17      y_pred = tf.clip_by_value(y_pred, tf.epsilon(), 1.0 - tf.epsilon())
18
19      return -tf.reduce_sum(y_true * tf.math.log(y_pred), axis = -1)
```

커스터마이제이션: 평가지표

- compile() 함수의 metrics 인자에 전달

```
01  from tensorflow.keras import backend as K
02
03  # 커스텀 평가지표를 정의합니다.
04  def recall_metric(y_true, y_pred):
05      true_pos = K.sum(K.round(K.clip(y_true * y_pred, 0.0, 1.0)))
06      pred_pos = K.sum(K.round(K.clip(y_true, 0.0, 1.0)))
07      recall = true_pos / (pred_pos + K.epsilon())
08
09      return recall
10
11  def precision_metric(y_true, y_pred):
12      true_pos = K.sum(K.round(K.clip(y_true * y_pred, 0.0, 1.0)))
13      pred_pos = K.sum(K.round(K.clip(y_pred, 0.0, 1.0)))
14      precision = true_pos / (pred_pos + K.epsilon())
15
16      return precision
17
18  def f1_metric(y_true, y_pred):
19      recall = recall_metric(y_true, y_pred)
20      precision = precision_metric(y_true, y_pred)
22      return 2 * ((precision * recall) / (precision + recall + K.epsilon()))
```

커스터마이제이션: 케라스 콜백

- 케라스 콜백은 모델 학습 과정을 모니터링하기 위해 사용되는 매우 유용한 기능
  - **기능 수행 기점을 지정하고, 지정한 기점에서 정의한 콜백을 수행**

```
04  class CustomLearningLateCallback(Callback):
05      def __init__(self):
06          pass
07
08      # 0.1배 만큼 학습률을 감소시킵니다.
09      def down_lr(self, current_lr):
10          return current_lr * 0.1
```

```
31              if((epoch == 4) or (epoch == 7) or (epoch == 9)):
32                  current_lr = self.down_lr(current_lr)
33                  # 감소된 학습률을 현재 모델 옵티마이저의 학습률로 설정합니다.
34                  K.set_value(self.model.optimizer.lr, current_lr)
35                  print('\nEpoch %03d: learning rate change! %s.' %
                    (epoch + 1, current_lr.numpy()))
```

```
Epoch 008: learning rate change! 1.0000001e-05.
Epoch 8/10
42000/42000 [==============================] - 9s 223us/sample - loss: 0.8564 -
acc: 0.7587 - val_loss: 0.8623 - val_acc: 0.7549
Epoch 9/10
42000/42000 [==============================] - 9s 215us/sample - loss: 0.8548 -
acc: 0.7591 - val_loss: 0.8607 - val_acc: 0.7558
Epoch 010: learning rate change! 1.0000001e-06.
Epoch 10/10
42000/42000 [==============================] - 8s 198us/sample - loss: 0.8536 -
acc: 0.7603 - val_loss: 0.8604 - val_acc: 0.7552
```

1x1 컨볼루션

- **“Network in Network”, “Fully Convolutional Networks for Semantic Segmentation”**
  - 후자 논문 인용 수: 15,000회 이상

- **FCN(Fully Convolutional Networks)**
  - 객체 탐지 및 분할 방법에서 활발하게 사용
  - 1x1 컨볼루션

Network In Network

Min Lin[1,2], Qiang Chen[2], Shuicheng Yan[2]
[1]Graduate School for Integrative Sciences and Engineering
[2]Department of Electronic & Computer Engineering
National University of Singapore, Singapore
{linmin,chenqiang,eleyans}@nus.edu.sg

Abstract

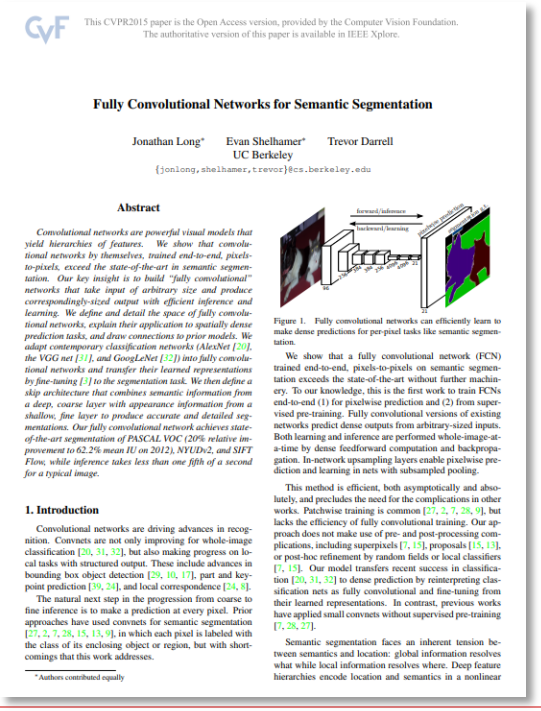
We propose a novel deep network structure called “Network In Network”(NIN) to enhance model discriminability for local patches within the receptive field. The conventional convolutional layer uses linear filters followed by a nonlinear activation function to scan the input. Instead, we build micro neural networks with more complex structures to abstract the data within the receptive field. We instantiate the micro neural network with a multilayer perceptron, which is a potent function approximator. The feature maps are obtained by sliding the micro networks over the input in a similar manner as CNN; they are then fed into the next layer. Deep NIN can be implemented by stacking mutiple of the above described structure. With enhanced local modeling via the micro network, we are able to utilize global average pooling over feature maps in the classification layer, which is easier to interpret and less prone to overfitting than traditional fully connected layers. We demonstrated the state-of-the-art classification performances with NIN on CIFAR-10 and CIFAR-100, and reasonable performances on SVHN and MNIST datasets.

1 Introduction

Convolutional neural networks (CNNs) [1] consist of alternating convolutional layers and pooling layers. Convolution layers take inner product of the linear filter and the underlying receptive field followed by a nonlinear activation function at every local portion of the input. The resulting outputs are called feature maps.

The convolution filter in CNN is a generalized linear model (GLM) for the underlying data patch, and we argue that the level of abstraction is low with GLM. By abstraction we mean that the feature is invariant to the variants of the same concept [2]. Replacing the GLM with a more potent nonlinear function approximator can enhance the abstraction ability of the local model. GLM can achieve a good extent of abstraction when the samples of the latent concepts are linearly separable, i.e. the variants of the concepts all live on one side of the separation plane defined by the GLM. Thus conventional CNN implicitly makes the assumption that the latent concepts are linearly separable. However, the data for the same concept often live on a nonlinear manifold, therefore the representations that capture these concepts are generally highly nonlinear function of the input. In NIN, the GLM is replaced with a “micro network” structure which is a general nonlinear function approximator. In this work, we choose multilayer perceptron [3] as the instantiation of the micro network, which is a universal function approximator and a neural network trainable by back-propagation.

The resulting structure which we call an mlpconv layer is compared with CNN in Figure 1. Both the linear convolutional layer and the mlpconv layer map the local receptive field to an output feature vector. The mlpconv maps the input local patch to the output feature vector with a multilayer perceptron (MLP) consisting of multiple fully connected layers with nonlinear activation functions. The MLP is shared among all local receptive fields. The feature maps are obtained by sliding the MLP



Fully Convolutional Networks for Semantic Segmentation

Jonathan Long* Evan Shelhamer* Trevor Darrell
UC Berkeley
{jonlong,shelhamer,trevor}@cs.berkeley.edu

Abstract

Convolutional networks are powerful visual models that yield hierarchies of features. We show that convolutional networks by themselves, trained end-to-end, pixels-to-pixels, exceed the state-of-the-art in semantic segmentation. Our key insight is to build “fully convolutional” networks that take input of arbitrary size and produce correspondingly-sized output with efficient inference and learning. We define and detail the space of fully convolutional networks, explain their application to spatially dense prediction tasks, and draw connections to prior models. We adapt contemporary classification networks (AlexNet [20], the VGG net [31], and GoogLeNet [32]) into fully convolutional networks and transfer their learned representations by fine-tuning [3] to the segmentation task. We then define a skip architecture that combines semantic information from a deep, coarse layer with appearance information from a shallow, fine layer to produce accurate and detailed segmentations. Our fully convolutional network achieves state-of-the-art segmentation of PASCAL VOC (20% relative improvement to 62.2% mean IU on 2012), NYUDv2, and SIFT Flow, while inference takes less than one fifth of a second for a typical image.

1. Introduction

Convolutional networks are driving advances in recognition. Convnets are not only improving for whole-image classification [20, 31, 32], but also making progress on local tasks with structured output. These include advances in bounding box object detection [29, 10, 17], part and keypoint prediction [39, 24], and local correspondence [24, 8].

The natural next step in the progression from coarse to fine inference is to make a prediction at every pixel. Prior approaches have used convnets for semantic segmentation [27, 2, 7, 28, 15, 13, 9], in which each pixel is labeled with the class of its enclosing object or region, but with shortcomings that this work addresses.

*Authors contributed equally

Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

We show that a fully convolutional network (FCN) trained end-to-end, pixels-to-pixels on semantic segmentation exceeds the state-of-the-art without further machinery. To our knowledge, this is the first work to train FCNs end-to-end (1) for pixelwise prediction and (2) from supervised pre-training. Fully convolutional versions of existing networks predict dense outputs from arbitrary-sized inputs. Both learning and inference are performed whole-image-at-a-time by dense feedforward computation and backpropagation. In-network upsampling layers enable pixelwise prediction and learning in nets with subsampled pooling.

This method is efficient, both asymptotically and absolutely, and precludes the need for the complications in other works. Patchwise training is common [27, 2, 7, 28, 9], but lacks the efficiency of fully convolutional training. Our approach does not make use of pre- and post-processing complications, including superpixels [7, 15], proposals [15, 13], or post-hoc refinement by random fields or local classifiers [7, 15]. Our model transfers recent success in classification [20, 31, 32] to dense prediction by reinterpreting classification nets as fully convolutional and fine-tuning from their learned representations. In contrast, previous works have applied small convnets without supervised pre-training [7, 28, 27].

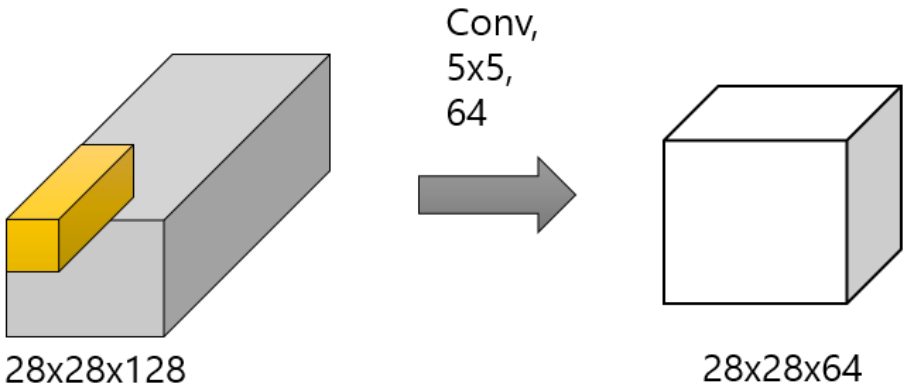
Semantic segmentation faces an inherent tension between semantics and location: global information resolves what while local information resolves where. Deep feature hierarchies encode location and semantics in a nonlinear

1x1 컨볼루션

- 1x1 컨볼루션의 장점
  - 모델 파라미터 감소
  - 비선형성 증가
  - 채널 수 조절

- ResNet, Inception 계열 모델을 직접 구성해서 1x1 컨볼루션이 어떻게 사용되는지 느껴보세요
- **다음 페이지를 통해 컨볼루션에서 사용되는 각 숫자가 무엇을 의미하는지 생각해보세요!**

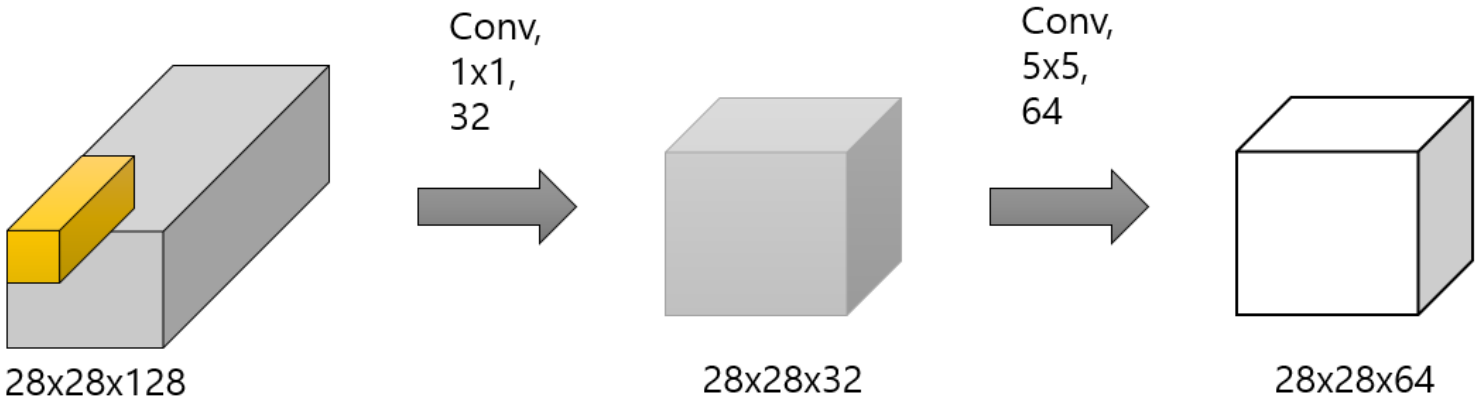
1x1 컨볼루션(실습)



각 숫자가 무엇을 의미하는지 해석할 수 있나요?

그렇다면 제공되는 코드를 통해 컨볼루션 층만 사용하여 모델을 구성하세요!

#params = 28x28x64x5x5x128 = 160M



#params = 28x28x32x128x1x1
= 4.8M

#params = 28x28x64x5x5x32
= 40M

#total = 44.8M

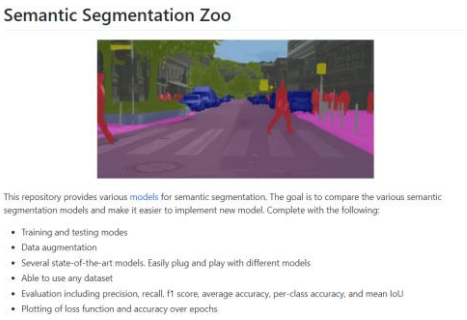
다양한 문제를 위한 딥러닝

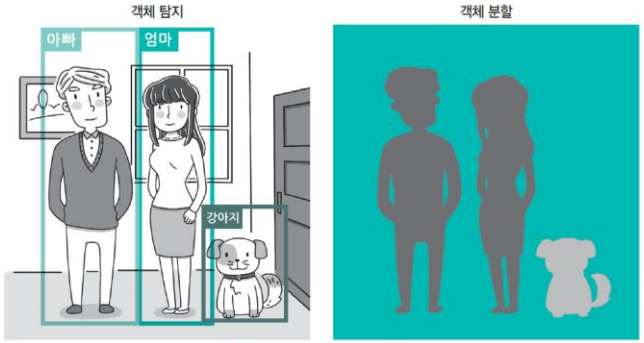
- 딥러닝은 기존의 전통적인 알고리즘으로 해결하기 어려웠던 문제를 해결할 수 있음
  - 객체 탐지(Object Detection), 객체 분할(Semantic Segmentation)
  - 딥페이크(DeepFake)
  - 질의응답(Question&Answering)
  - 이상 현상 탐지(Anomaly Detection), 슈퍼 레졸루션(Super Resolution), 스타일 트랜스퍼(Style Transfer) 등

- **객체 탐지(Object Detection)**
  - 이미지 내에서 객체의 위치를 추적하고, 이를 분류
  - TensorFlow Object Detection API
    - 상당히 복잡하고, 사용하기 어려움 ☞ 기초 모형은 Keras Core를 활용하면 굉장히 쉬움(코드 몇줄)
  - 케라스 코리아 오픈소스팀에서 케라스로 구현한 객체 탐지 라이브

다양한 문제를 위한 딥러닝

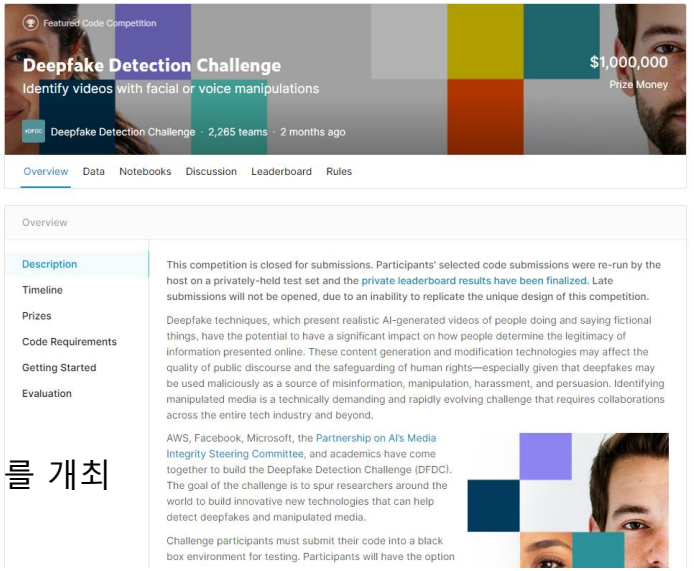
- **객체 분할(Semantic Segmentation)**
  - 이미지 픽셀 단위, 인스턴스 단위 등으로 분류하는 방법
  - 세포 선, 제품 및 도로 균열과 같은 구체적인 특징 파악에 유용
  - 자율 주행뿐만 아니라 의료 분야에서도 활발하게 사용





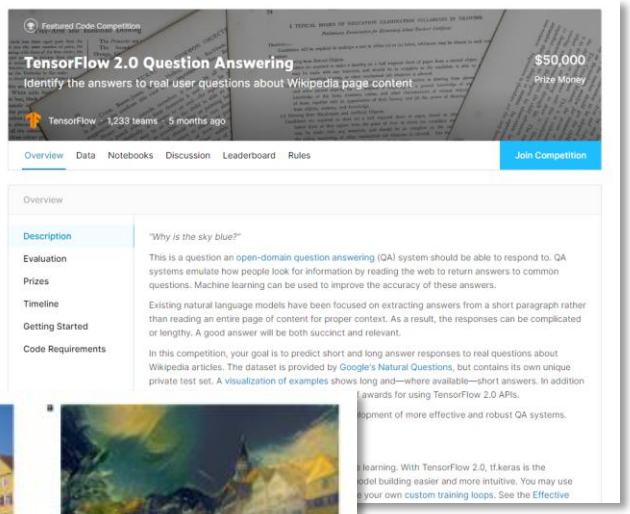
[그림 8-2] 객체 탐지와 객체 분할

- **딥페이크(DeepFake)**
  - GAN(Generative Adversarial Network) 방법 활용
  - 사람 얼굴, 행동 등을 표현한 가짜 동영상을 생성
  - 가짜 동영상, 포르노 영상 생성에 활용되는 등 악용 사례 존재
  - 많은 연구자들이 이를 심각하게 받아들이고 50만 달러를 건 대회를 개최



다양한 문제를 위한 딥러닝

- **질의응답(Question And Answering)**
  - 이미지를 보고 설명하거나, 뉴스 기사에 관한 질문에 대해 답변을 예측
  - 캐글의 TensorFlow 2.0 Question Answering에 참가하여 텐서플로우 2.x도 공부하고, 대회도 체험해보세요
  - BERT를 사용한 많은 노트북이 존재
  - 자연어 처리를 공부하고 싶다면 여길 참고해도 좋음!



- **이상탐지(Anomaly Detection)**
  - 기존 데이터와 다른 특성을 보이는 데이터를 잡아냄
  - 카드 사기, 불량품 검출 등

- **슈퍼 레졸루션(Super Resolution)**
  - 저해상도 이미지 → 고해상도 이미지

- **스타일 트랜스퍼(Style Transfer)**
  - 이미지 스타일 변환

- **이미지 색상화(Image Colorization)**
  - 흑백 또는 특정 사진에 원하는 색을 추가

그림 출처: "**Image Style Transfer Using Convolutional Neural Networks**"

코드와 논문, 논문과 코드

- 입문(초급) 수준의 책을 공부하고 나서 다음 방향을 어떻게 선택해야 할까?
  - 정답은 없지만, 다양한 방법이 존재

- 개인 프로젝트 수행
  - 문제 정의부터 제공까지의 전체적인 프로세스를 그려보자
  - **문제를 정의하는 연습은 꾸준히!**
  - **데이터 구성, 전처리하는 과정은 매우 값진 시간**
    - 빨리 모델을 구성하려는 조급한 마음을 가지지 말자

- 논문 구현
  - 딥러닝은 논문을 통해 매우 많은 정보를 알 수 있음
  - **논문을 보고 구현하는 능력은 실력을 증명하는 것과 같음 → 매우 중요!**
  - "Papers with Code", Google Scholar, arXiv 등을 활용

딥러닝, 실제 환경에서는?

- 우리가 지금까지 보았던 MNIST, CIFAR-10, IMDB 등의 데이터셋은 처리가 매우 잘 되어있어 성능 향상을 위해 많은 시간 투자가 필요하지 않았음
  - 실제 환경에서 다루는 데이터는 매우 복잡하고, 까다로움
  - **"현업에서의 데이터", "현업에서의 실제 딥러닝 적용"과 같은 경험적인 주제를 다루는 글이 있다면, 지나치지 말고 꼭 읽어볼것**
  - **아직 딥러닝은 경험에 의해 결정되는 요소가 굉장히 많음**

- 데이터? 또 데이터!
  - **100시간 중 80시간은 데이터를 파악하는 데 투자!**
  - 정제되지 않은 데이터를 어떻게 구성해야 하고, 어떤 전처리 방법을 사용해야 하며, 어디에, 어떻게, 얼마나 효율적으로 저장해두어야 할지에 대한 고민은 계속해서 우리를 괴롭힐 것
    - 데이터는 매우, 매우, 매우 중요합니다
    - 모델이 전부가 아닙니다. 빙산의 일각입니다.
    - 우리가 흔히 말하는 모델은 데이터가 우리에게 말하려는 바를 표현해주기 위한 일종의 수단입니다.
    - 양질의 데이터는 보통의 모델도 최상의 모델로 바꿀 수 있습니다.
    - 데이터를 다루기 위한 방법을 소개하는 매우 좋은 책과 글이 많지만, 가장 좋은 방법은 직접 대회에 참가하거나 개인 프로젝트를 수행하는 것입니다.
    - 본인이 수집한 데이터가 양질의 데이터가 아니라고 실망하지 마세요! 그 과정이 실력입니다.

정통(머신러닝) 모델? 딥러닝 모델?

- 딥러닝이 많은 문제를 우수하게 해결하고 있음
  - 그렇다고 직면한 문제를 무조건 딥러닝을 통해 해결하려는 마음은 가지면 안됨
  - 실제로 캐글에서도 아직 수많은 머신러닝 모델이 사용되고 있음
  - **더 효율적인 머신러닝 모델이 있다면, 딥러닝 모델을 사용할 필요가 있는지 본인에게 물어볼 것**

- **문제를 다양한 관점에서 다양한 모델을 사용하여 해결하려는 노력이 필요(많은 노가다와 시간)**
  1. 머신러닝 모델을 탐색하고 적용한다. 데이터를 수집한다.
  2. 딥러닝 모델을 탐색하고 적용한다. 데이터를 수집한다.
  3. 데이터가 충분하지 않다면, 딥러닝 모델이 충분히 일반화되지 않을 가능성이 높으므로 머신러닝 모델을 선택해도 좋다. 데이터를 수집한다.
  4. 데이터가 충분히 모였다면, 딥러닝 모델을 다시 학습시키고 이를 제품에 적용한다. 데이터를 수집한다.

요약 정리

1. 이 장에서 다루는 내용은 **텐서플로우 공식 문서 튜토리얼**에서도 찾아볼 수 있습니다. 번갈아 가면서 공부하면, 효과는 두 배입니다.
2. 우리가 직면한 특정 문제를 해결하려면 **커스터마이제이션 방법**이 필요합니다.
3. **가중치의 학습 여부**(Y/N)에 따라 적합한 층(**Lambda층/Layer 클래스를 상속한 커스텀 케라스층**)을 사용합니다.
4. 2019년에 발표된 **최적화 함수(RAdam), 활성화 함수(Mish)**를 다뤄보았습니다.
5. 실제값(y_true), 모델이 예측한 값(y_pred)을 적절히 조정하면 쉽게 커스텀 손실 함수를 정의할 수 있습니다.
6. 1x1 컨볼루션은 **모델 파라미터 감소, 비선형성 증가, 채널 수 조절의 장점**을 제공합니다.
7. Dense층을 사용하지 않아도, 모델을 구성할 수 있습니다.
8. **딥러닝은 아직 경험적으로 결정되는 요소가 매우 많습니다.**
9. 객체 탐지, 객체 분할, 딥페이크, 질의응답, 이상 현상 탐지 등을 살펴보았습니다.
10. 이 책을 공부하고 나서의 다음 방향으로 **(1) 개인 프로젝트 (2) 논문 구현**을 추천합니다.
11. 실제 환경에서는 **데이터가 매우, 매우, 매우 중요**합니다.
12. 딥러닝이 모든 문제를 해결할 것이라는 생각은 아직 위험합니다. 때론, **기존에 사용되고 있던 정통 모델을 고려해볼 줄 알아야합니다.**