



7장. 케라스

요약 정리

1. 순환 신경망은 **시퀀스 또는 시계열 데이터 처리에 특화**되어 있습니다.
2. Embedding층은 **수많은 단어(또는 데이터)를 표현**할 수 있습니다. 항상 모델의 첫 번째 층으로만 사용할 수 있습니다.
3. Embedding층은 (batch_size, sequence_length) 형태를 입력으로 받으며, (batch_size, sequence_length, output_dim) 형태를 출력합니다.
4. Embedding층은 **단어의 관계와 맥락을 파악할 수 없습니다**. 이를 해결하기 위해 사용되는 것이 SimpleRNN층입니다. SimpleRNN층은 순환 신경망의 가장 기본적인 형태를 나타내며, 출력값의 업데이트를 위해 **이전 상태를 사용**합니다.
5. SimpleRNN층은 (batch_size, timesteps, input_dim)의 형태를 입력으로 받으며, (batch_size, units)를 출력합니다.
6. SimpleRNN층은 **그래디언트 손실 문제를 야기**합니다. 이를 해결하기 위해 고안된 것이 LSTM입니다. LSTM은 **과거의 정보를 나르는 'Cell State'**를 가지고 있으며, 정보를 제거 또는 제공하기 위한 input_gate, forget_gate, output_gate를 보유하고 있습니다.

요약 정리

7. Conv2D층을 통해 이미지 데이터의 특징을 추출할 수 있었다면, Conv1D층을 통해 시퀀스 데이터의 특징을 추출할 수 있습니다.
8. Conv1D층은 (batch_size, timesteps, channels) 형태를 입력으로 받으며, (batch_size, timesteps, filters) 형태를 출력합니다.
9. BERT는 자연어 처리 분야에서 최고의 성능을 달성한 모델입니다. 자연어 처리뿐만 아니라 다양한 분야에서 뛰어난 성능을 보여주고 있기 때문에 충분히 관심을 가져볼 만한 방법입니다.

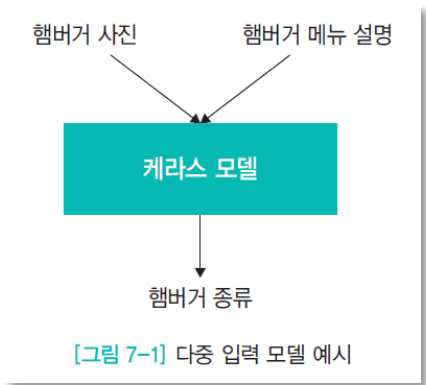
7장의 내용?

- 케라스는 모델 구성을 위해 다양한 방법을 제공하고, 문제에 적합한 모델을 구성할 수 있도록 도와줌
 - 특히 이번 장에서 배워볼 함수형 API는 텐서플로우에서 권장하고 있는 모델 구성 방법
- 성능을 향상하기 위한 다양한 방법이 존재
 - 케라스 콜백은 필수적으로 사용해야할 방법임
 - 케라스는 다양한 케라스 콜백을 제공하지만, 여기서는 가장 기본적으로 사용되는 콜백을 알아볼 것
- 이번 장에서는 다음 내용을 다뤄봅니다.
 - 케라스에서 모델을 구성하는 세 가지 방법: Sequential(), 서브클래싱, 함수형 API
 - 함수형 API 활용하기: 다중 입출력, 잔차 연결과 인셉션 모듈, 전이 학습
 - 케라스 콜백 사용하기: ModelCheckPoint, EarlyStopping, ReduceLROnPlateau, TensorBoard

케라스의 모델 구성 방법

- 케라스의 모델 구성 방법

- Sequential()
- 서브클래싱(Subclassing)
- 함수형 API(Functional API)



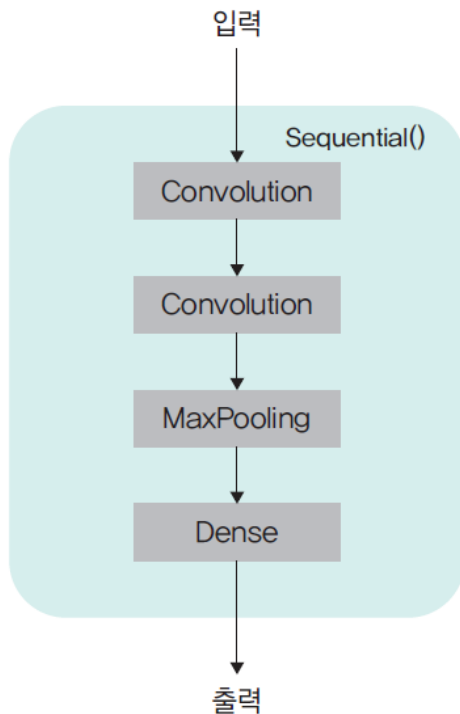
- 지금까지 우리가 사용해왔던 Sequential() 방법은 매우 쉽게 사용할 수 있지만,

복잡한 모델을 구성하기 어려움

- 다중 입출력 문제
 - 이미지 데이터와 이에 대한 설명을 동시에 입력
- 앞으로 배울 서브클래싱, 함수형 API는 모델을 유연하게 구성할 수 있도록 도와줌

Sequential()로 구성하기

- 매우 익숙한 방법
- 여러 개의 입력과 출력으로 구성할 수 없으며, 층의 구조를 유연하게 만들지 못함



```
05 # Sequential()을 통한 모델 구성
06 model = Sequential()
07 model.add(Conv2D(32, (3, 3), activation = 'relu', input_shape = (28, 28, 1)))
08 model.add(Conv2D(32, (3, 3), activation = 'relu'))
09 model.add(MaxPooling2D(strides = 2))
10 model.add(GlobalAveragePooling2D())
11 model.add(Dense(1, activation = 'sigmoid'))
```

[그림 7-2] Sequential() 방법

서브클래싱으로 구성하기

- 모델 커스터마이징하기에 최적화된 방법

- 케라스 **Model** 클래스를 서브클래싱하여 모델을 구성 📖 프로젝트 내 모델 설계시 이 방법을 주로 활용
- Model 클래스가 보유하고 있는 기능 그대로 사용할 수 있음
 - fit(), evaluate(), model.layer 등
- 모델을 일일이 구성해야 하므로 다른 방법보다 시간이 더 오래걸림
- 이 방법보다 함수형 API를 더 자주 사용하지만, 모델 구현 코드에서 서브클래싱 방법을 사용한 코드를 빈번하게 찾아볼 수 있으므로 알아두는 것이 좋음

```
04 class MyModel(Model):
05     # 사용할 층을 정의합니다.
06     def __init__(self):
07         super(MyModel, self).__init__()
08
09         self.first_conv = Conv2D(32, (3, 3), activation = 'relu')
10         self.second_conv = Conv2D(32, (3, 3), activation = 'relu')
11         self.maxpool = MaxPooling2D(strides = 2)
12
13         self.gap = GlobalAveragePooling2D()
14         self.dense = Dense(1, activation = 'sigmoid')
15
16     # 입력 -> 출력의 흐름을 구성합니다.
17     def call(self, inputs):
18         x = self.first_conv(inputs)
19         x = self.second_conv(x)
20         x = self.maxpool(x)
21
22         x = self.gap(x)
23         x = self.dense(x)
24
25     return x
```

__init__ → 사용할 층을 정의

call → 모델 구성
inputs 인자 또는 반환값을 통해 다중입
출력 모델 구성

함수형 API로 구성하기

- 앞의 두 가지 방법을 배우면서도 함수형 API를 계속 언급해왔음
 - 그만큼 권장하고 쉬움
 - 모델을 복잡하고, 유연하게 구성할 수 있으며, 다중입출력을 다룰 수 있음

```
06 # 함수형 API는 Input층을 통해 입력값의 형태를 정의해주어야 합니다.
07 inputs = Input(shape = (224, 224, 3))
08 x = Conv2D(32, (3, 3), activation = 'relu')(inputs)
09 x = Conv2D(32, (3, 3), activation = 'relu')(x)
10 x = MaxPooling2D(strides = 2)(x)
11 x = GlobalAveragePooling2D()(x)
12 x = Dense(1, activation = 'sigmoid')(x)
13
14 # 위에서 정의한 층을 포함하고 있는 모델을 생성합니다.
15 model = Model(inputs = inputs, outputs = x)
```

- Input 층을 통해 반드시 입력값 형태를 입력해주어야 함
- Conv2D(32, ~)(inputs) 표현으로 이전 층과 연결 가능
- Model 클래스를 활용하여 모델을 생성
 - inputs, outputs 인자에 입출력값을 전달

함수형 API 사용하기

- 제공되는 코드를 통해 MNIST 데이터셋을 학습시켜보세요!
 - 함수형 API를 사용하는 경우 Input층을 통해 입력값 형태를 정의하고, Model 클래스를 통해 모델의 입력값과 출력값을 지정해주어야 함!

```
05 # 함수형 API는 Input()을 통해 입력값의 형태를 정의해주어야 합니다.
06 inputs = Input(shape = (28, 28, 1))
07 x = Conv2D(32, (3, 3), activation = 'relu')(inputs)
08 x = Conv2D(32, (3, 3), activation = 'relu')(x)
09 x = MaxPooling2D(strides = 2)(x)
10 x = GlobalAveragePooling2D(x)
11 x = Dense(10, activation = 'softmax')(x)
12
13 # 위에서 정의한 층을 포함하고 있는 모델
14 model = Model(inputs = inputs, outputs = x)
```

```
01 from tensorflow.keras.datasets import mnist
02
03 # 텐서플로우 저장소에서 데이터를 다운로드합니다.
04 (x_train, y_train), (x_test, y_test) = mnist.load_data(path='mnist.npz')
```

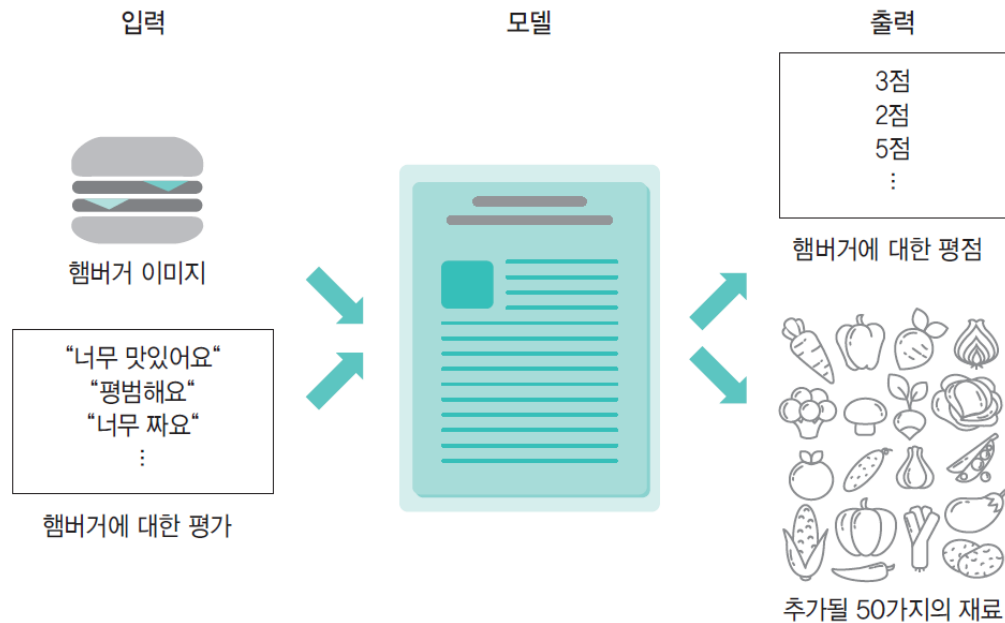
다중 입출력 사용해보기(실습)

- 다중 입출력 예제
 - 입력한 이미지(동영상)가 어떤 의미를 나타내는지 물어보는 질문과 이미지(동영상)에 대한 설명을 결과로 도출하는 문제
 - 입력: 이미지(동영상), 질문
 - 출력: 이미지(동영상)에 대한 설명
- 다중 입출력 예제를 직접 만들어보고, 학습시켜보자!
 - 성능보다 문제 해결에 집중!

다중 입출력 사용해보기

- 내용 요약

- 메뉴 보안을 도와줄 자동화 기계 도입
- (입력) 각 햄버거 종류에 대한 평가와 이미지
- (출력) 햄버거 평점과 어떤 재료가 추가되기를 원하는지에 대한 분석 결과
- 여러 햄버거의 평점과 추가를 원하는 재료에 대한 정보는 이미 수집해두었다.



[그림 7-4] 다중 입출력 예시

다중 입출력 사용해보기

- **NumPy 라이브러리**를 활용하여 데이터를 만들자

- 이미지와 텍스트 형태로 된 평가
- 평점과 추가를 원하는 50가지 재료

```
03 # 햄버거 이미지
04 hamburger_img = np.random.random((1000, 28, 28, 1))
05 # 햄버거에 대한 평가
06 customer_form = np.random.randint(10000, size = (1000, 100))
07
08 # 햄버거에 대한 평점
09 hamburger_rate = np.round(np.random.random((1000,)) * 5, 1)
10 # 햄버거에 추가되어질 50가지의 재료
11 update_for_hamburger = np.random.randint(50, size = (1000,))
```

- 1,000개 데이터 생성, MNIST 데이터셋 숫자 이미지와 동일한 형태
 - 햄버거에 대한 평가는 10,00개 단어를 사용하여 각각의 단어가 정수 형태로 변환된 상태
 - 평점 범위:[0~5], 추가될 재료의 후보는 총 50가지
-

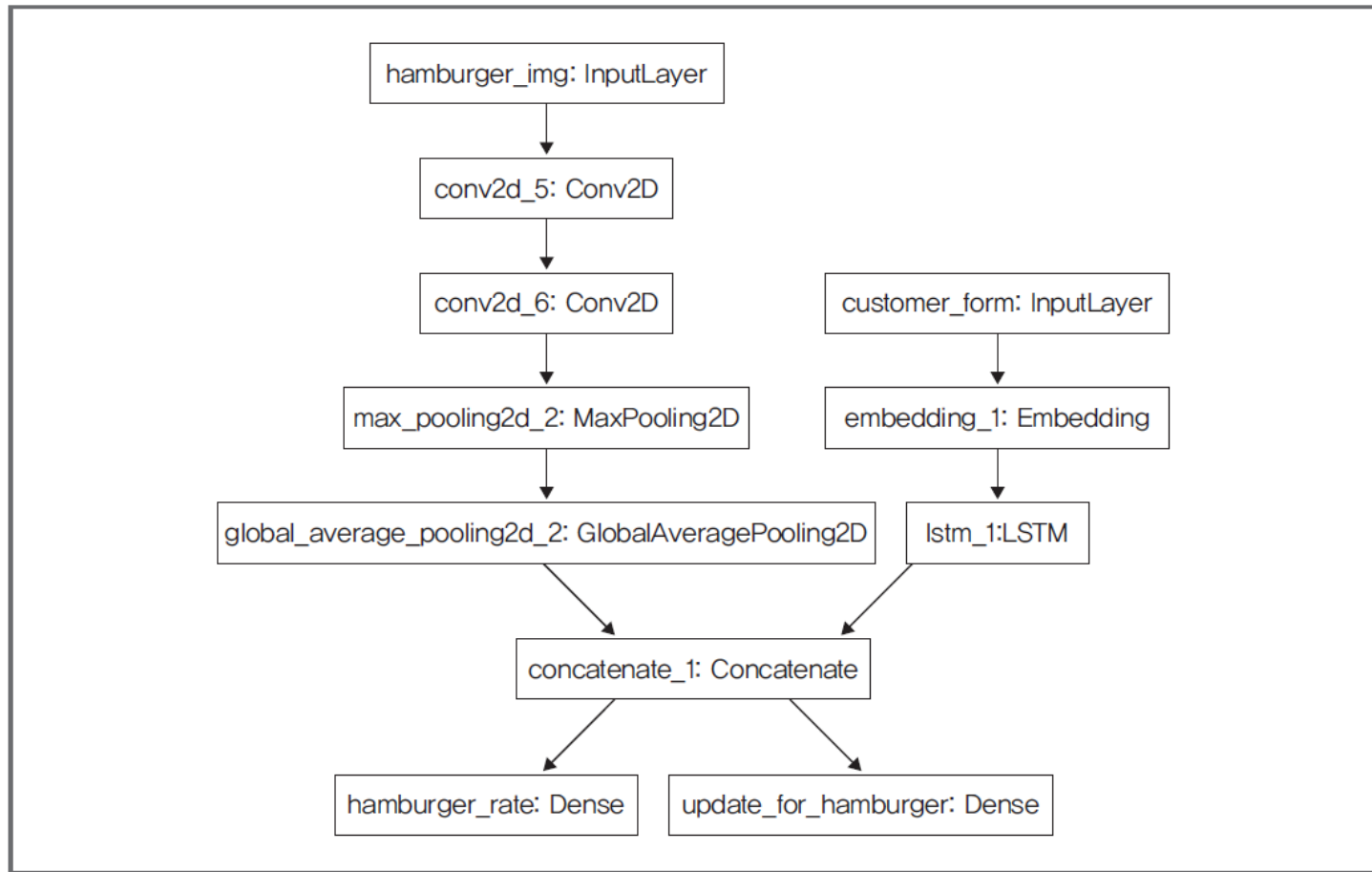
다중 입출력 사용해보기

- 모델을 구성하자

```
06 # 각 입력에 대한 형태를 명시합니다.
07 img_input = Input(shape = (28, 28, 1), name = 'hamburger_img')
08 form_input = Input(shape = (None, ), name = 'customer_form')
09
10 # 햄버거 이미지 입력
11 x_1 = Conv2D(32, (3, 3), activation = 'relu')(img_input)
12 x_1 = Conv2D(32, (3, 3), activation = 'relu')(x_1)
13 x_1 = MaxPooling2D(strides = 2)(x_1)
14 x_1 = GlobalAveragePooling2D()(x_1)
15
16 # 햄버거에 대한 평가 입력
17 x_2 = Embedding(10000, 64)(form_input)
18 x_2 = LSTM(128)(x_2)
19
20 # 출력을 만들기 위해 모든 입력을 하나의 텐서로 합칩니다.
21 x = Concatenate()([x_1, x_2])
22
23 # 햄버거 평점에 대한 출력값
24 rate_pred = Dense(1, name = 'hamburger_rate')(x)
25 # 보완될 50가지 재료에 대한 출력값
26 update_pred = Dense(50, activation = 'softmax',
27                      name = 'update_for_hamburger')(x)
28
29 # 모델을 생성합니다.
30 model = Model(inputs = [img_input, form_input],
31               outputs = [rate_pred, update_pred])
```

- 입력의 개수만큼 Input층 정의
- 다중 입출력이라면, Model 클래스의 inputs, outputs 인자에 리스트 형태로 모든 값을 전달

다중 입출력 사용해보기



다중 입출력 사용해보기

학습 과정 설정하기



[함께 해봐요] 다중 입출력 모델에서 학습 과정 설정하기 functional_api_multi_io.ipynb

```
01 # 손실 함수에 리스트 형태를 사용한 경우
02 model.compile(optimizer = 'adam',
03               loss = ['mse', 'sparse_categorical_crossentropy'],
04               metrics =
05                 {'hamburger_rate':'mse', 'update_for_hamburger':'acc'})
06 # 또는
07
08 # 손실 함수에 딕셔너리 형태를 사용한 경우
09 model.compile(optimizer = 'adam',
10               loss = {'hamburger_rate':'mse',
11                       'update_for_hamburger':'sparse_categorical_crossentropy'},
12               metrics = {'hamburger_rate':'mse', 'update_for_hamburger':'acc'})
```

학습하기



[함께 해봐요] 다중 입출력 모델 학습하기

functional_api_multi_io.ipynb

```
01 # 모델 학습에 리스트 형태를 사용한 경우
02 model.fit([hamburger_img, customer_form],
03          [hamburger_rate, update_for_hamburger],
04          epochs = 2, batch_size = 32)
05
06 # 또는
07
08 # 모델 학습에 딕셔너리 형태를 사용한 경우
09 model.fit({'hamburger_img':hamburger_img, 'customer_form':customer_form},
10          {'hamburger_rate':hamburger_rate,
11           'update_for_hamburger':update_for_hamburger},
12          epochs = 2, batch_size = 32)
```

```
Epoch 2/2
1000/1000 [=====] - 4s 4ms/sample - loss: 5.7940 -
hamburger_rate_loss: 1.9087 - update_for_hamburger_loss: 3.9014 -
hamburger_rate_mse: 1.8936 - update_for_hamburger_acc: 0.0240
```

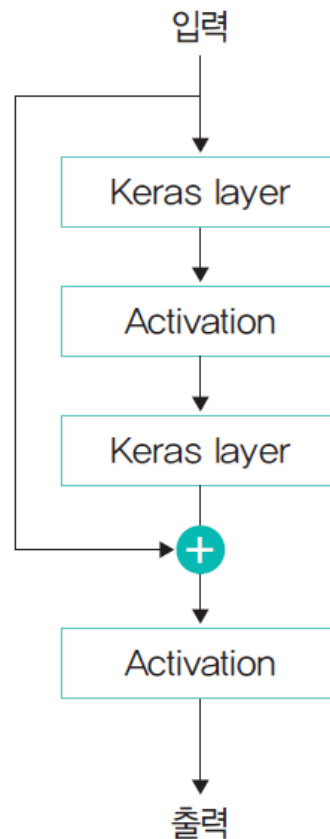
잔차 연결

- 잔차 연결(Residual block)

- short-cut 방법
- 마이크로소프트 팀에서 개발한 ResNet에서 처음 사용
- 논문 인용 수가 3만회 → 성능 향상 인정받음
- 2015년 ImageNet 대회에서 이를 활용하여 152개 층을 쌓아 1등을 차지

- 초기 신경망 모델은 높은 성능을 얻기 위해 층을 깊이 쌓는 방법을 선택

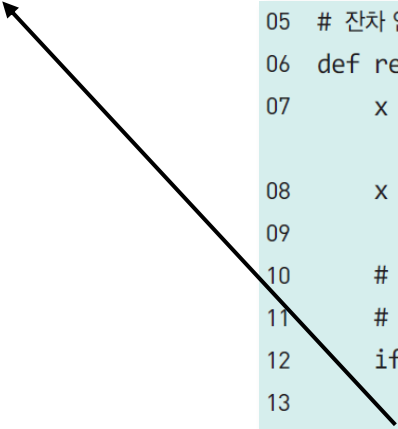
- 그래디언트 소실 및 폭발 문제 발생 → 학습되지 않음
- 잔차 연결이 이를 해결!



[그림 7-5] 잔차 연결

잔차 연결(실습)

- 제공되는 코드를 통해 잔차 연결을 구성해보세요
 - Add층**을 통해 입력값과 변환된 입력값을 더해주는 연산 추가
 - 1x1 컨볼루션**을 사용하여 입력값의 형태와 몇 개의 케라스층을 거친 형태를 동일하게 하여 연산이 가능토록 함
 - (28, 28, 1)과 (14, 14, 32)는 연산되지 않음
 - 스트라이트 2, 1x1 컨볼루션을 사용하여 (28, 28, 1) → (14, 14, 32)로 조절



```
05 # 잔차 연결을 포함한 네트워크를 구현합니다.
06 def residual_block(inputs, num_channels, use_transform = False):
07     x = Conv2D(num_channels, (3, 3), activation = 'relu',
08               padding = 'same')(inputs)
09
10     # 입력값의 형태가 변환되는 경우,
11     # 1x1 컨볼루션을 통해 형태를 조절해줍니다.
12     if use_transform:
13         x = MaxPooling2D(strides = 2)(x)
14         inputs = Conv2D(num_channels, (1, 1), strides = 2,
15                       padding = 'same')(inputs)
16
17     # 입력값과 변환된 입력값을 더해줍니다.
18     add_x = Add()(inputs, x)
19     return Activation('relu')(add_x)
```

인셉션 모듈

- 인셉션 모듈(Inception Module)은 22개의 층으로 구성된 GoogLeNet(인셉션 V1)에서 처음 사용
 - 유명한 영화 인셉션(Inception)에서 비롯된 용어 맞춤



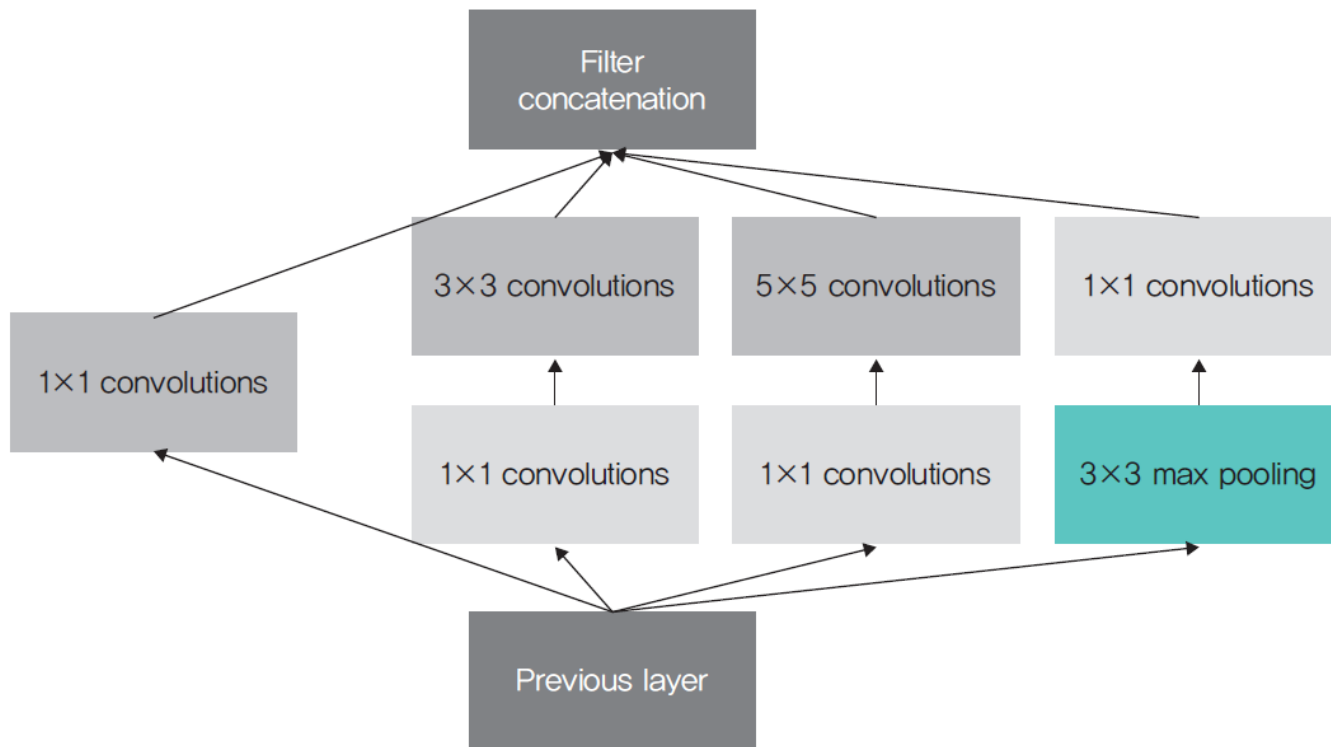
C. Szegedy et al, "Going Deeper with Convolutions" (CVPR 2015)



Figure 3: GoogLeNet network with all the bells and whistles.

인셉션 모듈

- 여러 가지 필터 크기를 사용하여 데이터의 다양한 특징을 담아냄
- 1x1 컨볼루션을 사용하여 차원 감소와 효과적인 다운샘플링을 통해 전체적인 자원 감소의 목적



[그림 7-6] 인셉션 V1 모듈

인셉션 모듈(실습)

- 제공되는 코드를 통해 인셉션 모듈을 구성해보세요
 - **Concatenate**층을 사용 → 채널 단위로 병합
 - 인셉션 계열은 구글 팀의 성능 및 자원 사용에 관한 다양한 고민이 담겨있음
 - 단계별로 각 버전에 어떤 방법이 사용되었는지 살펴본다면 매우 흥미로울 것!

```
05 def inception_module(x):
06     x_1 = Conv2D(32, (1, 1), activation = 'relu')(x)
07
08     x_2 = Conv2D(48, (1, 1), activation = 'relu')(x)
09     x_2 = Conv2D(64, (3, 3), activation = 'relu', padding = 'same')(x_2)
10
11     x_3 = Conv2D(16, (1, 1), activation = 'relu')(x)
12     x_3 = Conv2D(16, (5, 5), activation = 'relu', padding = 'same')(x_3)
13
14     x_4 = MaxPooling2D(pool_size = (3, 3), strides = 1, padding = 'same')(x)
15     x_4 = Conv2D(32, (1, 1), activation = 'relu')(x_4)
16
17     output = Concatenate()([x_1, x_2, x_3, x_4])
18
19     return output
```

텐서플로우 허브(실습)

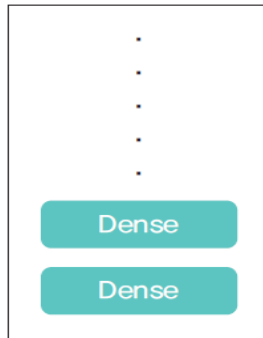
- 제공되는 코드를 통해 텐서플로우 허브를 실습해보세요

무슨 옷과 무슨 색

- 다중 입력 모델과 커스텀 제네레이터를 사용해보자

4장에서 구성한 모델

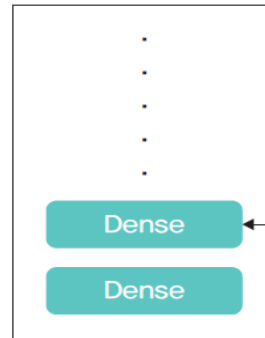
이미지 데이터



- 옷의 종류
- 옷의 색깔

이번 절에서 구성할 모델

이미지 데이터



- 옷의 종류
- 옷의 색깔

[그림 7-7] 모델 구조 확인하기

무슨 옷과 무슨 색

<pre>05 class DataGenerator(tf.keras.utils.Sequence):</pre>	<ul style="list-style-type: none">제네레이터 기본 기능을 사용하기 위해 Sequence 클래스 상속
<pre>06 def __init__(self, df, batch_size = 32, target_size = (112, 112), 07 shuffle = True): 08 self.len_df = len(df) 09 self.batch_size = batch_size 10 self.target_size = target_size 11 self.shuffle = shuffle 12 self.class_col = ['black', 'blue', 'brown', 'green', 'red', 'white', 13 'dress', 'shirt', 'pants', 'shorts', 'shoes'] 14 15 # 제네레이터를 통해 이미지를 불러옵니다. 16 self.generator = ImageDataGenerator(rescale = 1./255)</pre>	<p>__init__ 함수</p> <ul style="list-style-type: none">이미지 제네레이터, 필요 인자 정의
<pre>37 # ([이미지 데이터, 색 정보], 레이블)을 반환합니다. 38 # 이미지는 미리 정의한 제네레이터를 통해, 39 # 색 정보는 __data_generation 메서드를 활용합니다. 40 def __getitem__(self, index): 41 indexes = 42 self.indexes[index * self.batch_size : (index + 1) * self.batch_size] 43 colors = self.__data_generation(indexes) 44 images, labels = self.df_generator.__getitem__(index) 45 46 # return multi-input and output 47 return [images, colors], labels</pre>	<p>__getitem__ 함수</p> <ul style="list-style-type: none">데이터 반환이미지, 색, 레이블([images, colors], labels)를 반환
<pre>48 # 데이터를 생성합니다. 49 def __data_generation(self, indexes): 50 colors = np.array([self.colors_df[k] for k in indexes]) 51 52 return colors</pre>	<p>__data_generation 함수</p> <ul style="list-style-type: none">색 데이터 생성

무슨 옷과 무슨 색(실습)

- 제공되는 코드를 통해 옷과 색을 구별해보세요
 - 다중 입출력, 커스텀 제네레이터 실습

케라스 콜백

- **모델의 학습 방향, 저장 시점, 학습 정지 시점 등에 관한 상황을 모니터링하기 위해 주로 사용**
 - fit() 함수를 통해 반환되는 History 객체를 활용해서 학습 과정을 그려보았음
 - 케라스 콜백 중 하나인 History 콜백이 모든 케라스 모델에 자동으로 적용되어 있기 때문
- **대표적으로 사용되는 4가지 케라스 콜백을 알아보자**
 - ModelCheckpoint
 - EarlyStopping
 - ReduceLROnPlateau
 - TensorBoard

ModelCheckpoint

- 지정한 평가지표를 기준으로 가장 뛰어난 성능을 보여주는 모델을 저장할 때 사용

```
ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False,  
                save_weights_only=False, mode='auto')
```

- 사용되는 인자에 대한 해석은 책 내용을 참고

```
05 # 콜백을 정의합니다.  
06 callbacks = [ModelCheckpoint(filepath = filepath, monitor = 'val_loss',  
                                verbose = 1, save_best_only = True)]  
07  
08 # callbacks 인자를 통해 정의한 콜백을 전달합니다.  
09 model.fit(x_train, y_train,  
10         batch_size = 32,  
11         validation_data = (x_val, y_val),  
12         epochs = 10,  
13         callbacks = callbacks)
```

```
Epoch 00009: val_loss improved from 0.41602 to 0.38909, saving model to .  
/best_model.hdf5
```

val_loss가 가장 낮은 모델을 저장!

EarlyStopping

- 모델 학습 시에 지정된 기간 동안 모니터링하는 평가지표에서 성능 향상이 일어나지 않은 경우 학습을 중단

```
EarlyStopping(monitor='val_loss', patience=0, verbose=0, mode='auto')
```

- 사용되는 인자에 대한 해석은 책 내용을 참고
- 어느 정도 학습을 진행해 본 뒤에 사용하는 것을 추천
 - 30번, 50번 또는 그 이상의 에폭이 진행된 후에 모델의 성능이 향상될 수 있기 때문
 - 그러나 사용 방법에 대한 정답은 없음. 여러 횟수의 에폭으로 학습을 진행해보고, 학습 동향을 파악한 뒤 적절한 patience 인자 값을 결정하여 사용!

```
03 # 콜백을 정의합니다.
04 callbacks = [EarlyStopping(monitor = 'val_loss', patience = 3, verbose = 1)]
05
06 # callbacks 인자를 통해 정의한 콜백을 전달합니다.
07 model.fit(x_train, y_train,
08           batch_size = 32,
09           validation_data = (x_val, y_val),
10           epochs = 30,
11           callbacks = callbacks)
```

```
Epoch 25/30
42000/42000 [=====] - 8s 191us/sample - loss: 0.2565 -
acc: 0.9230 - val_loss: 0.2600 - val_acc: 0.9216
Epoch 00025: early stopping
```

학습을 중단!

ReduceLROnPlateau

- EarlyStopping 콜백과 같이 patience 인자를 지정하여, 지정된 기간 동안 평가지표에서 성능 향상이 일어나지 않으면 학습률을 조정

```
ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=10, verbose=0,  
                  min_lr=0)
```

- 사용되는 인자에 대한 해석은 책 내용을 참고
- 일반적으로 factor는 0.1 또는 0.2, min_lr은 1e-6 또는 1e-7을 사용하여 살펴봄

```
03 # 콜백을 정의합니다.  
04 callbacks = [ReduceLROnPlateau(monitor = 'val_loss', patience = 3, factor = 0.2,  
05                               verbose = 1, min_lr = 1e-5)]  
06  
07 # callbacks 인자를 통해 정의한 콜백을 전달합니다.  
08 model.fit(x_train, y_train,  
09           batch_size = 32,  
10           validation_data = (x_val, y_val),  
11           epochs = 50,  
12           callbacks = callbacks)
```

```
Epoch 00049: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.  
42000/42000 [=====] - 8s 189us/sample - loss: 0.1854 -  
acc: 0.9443 - val_loss: 0.2004 - val_acc: 0.9404
```

학습률 감소!

TensorBoard

- 텐서보드는 학습 과정을 편리하게 모니터링할 수 있도록 텐서플로우에서 제공하고 있는 도구
 - 여러 가지 지표를 그래프로 시각화해주어 모델을 쉽게 분석할 수 있도록 도와줌

```
TensorBoard(log_dir='./logs', histogram_freq=0, batch_size=32,  
            write_graph=True, write_images=True)
```

- 사용되는 인자에 대한 해석은 책 내용을 참고

```
05 # 콜백을 정의합니다.  
06 callbacks = [TensorBoard(log_dir = logdir, histogram_freq = 1,  
07                          write_graph = True, write_images = True)]  
08  
09 # callbacks 인자를 통해 정의한 콜백을 전달합니다.  
10 model.fit(x_train, y_train,  
11          batch_size = 32,  
12          validation_data = (x_val, y_val),  
13          epochs = 30,  
14          callbacks = callbacks)
```

TensorBoard

01 tensorboard --logdir ./logs

```
library cudart64_100.dll  
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all  
TensorBoard 2.0.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

'http://localhost:6006' 주소로 접속

[그림 7-8] 텐서보드 서버 실행

학습 및 검증 데이터의 평가지표 및 손실

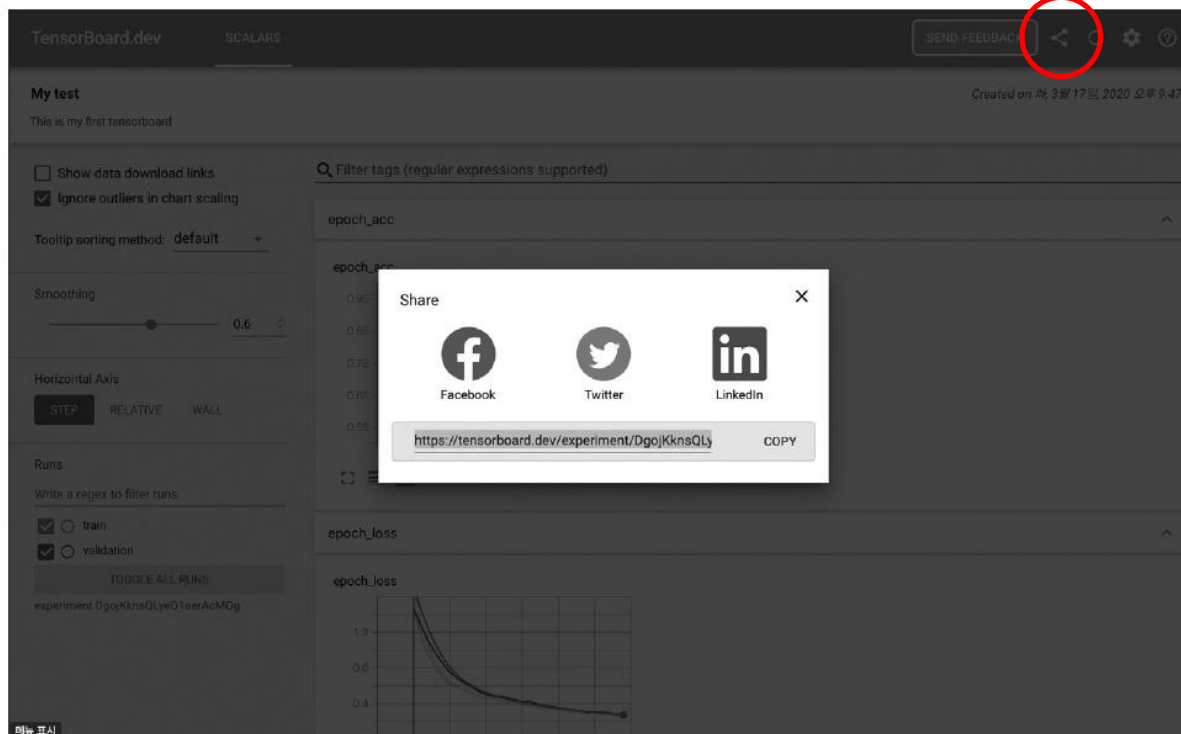
지정된 층의 파라미터(weight) 분포

활성화층에서의 데이터 형태

TensorBoard Dev.

- 텐서보드를 공유할 수 있는 기능
 - 텐서보드는 실험결과를 효율적으로 시각화하여 상대방에게 제공할 수 있음

```
01 tensorboard dev upload --logdir ./logs/ --name "My test" -  
-description "This is my first tensorboard"
```



[그림 7-12] TensorBoard Dev.를 통한 실험 결과 공유

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.applications에서 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