강사 윤영주
yj.youn1103@gmail.com

다섯째마당
딥러닝 활용하기

19장 상에 없는 얼굴 GAN, 오토인코더

1 가짜 제조 공장, 생성자

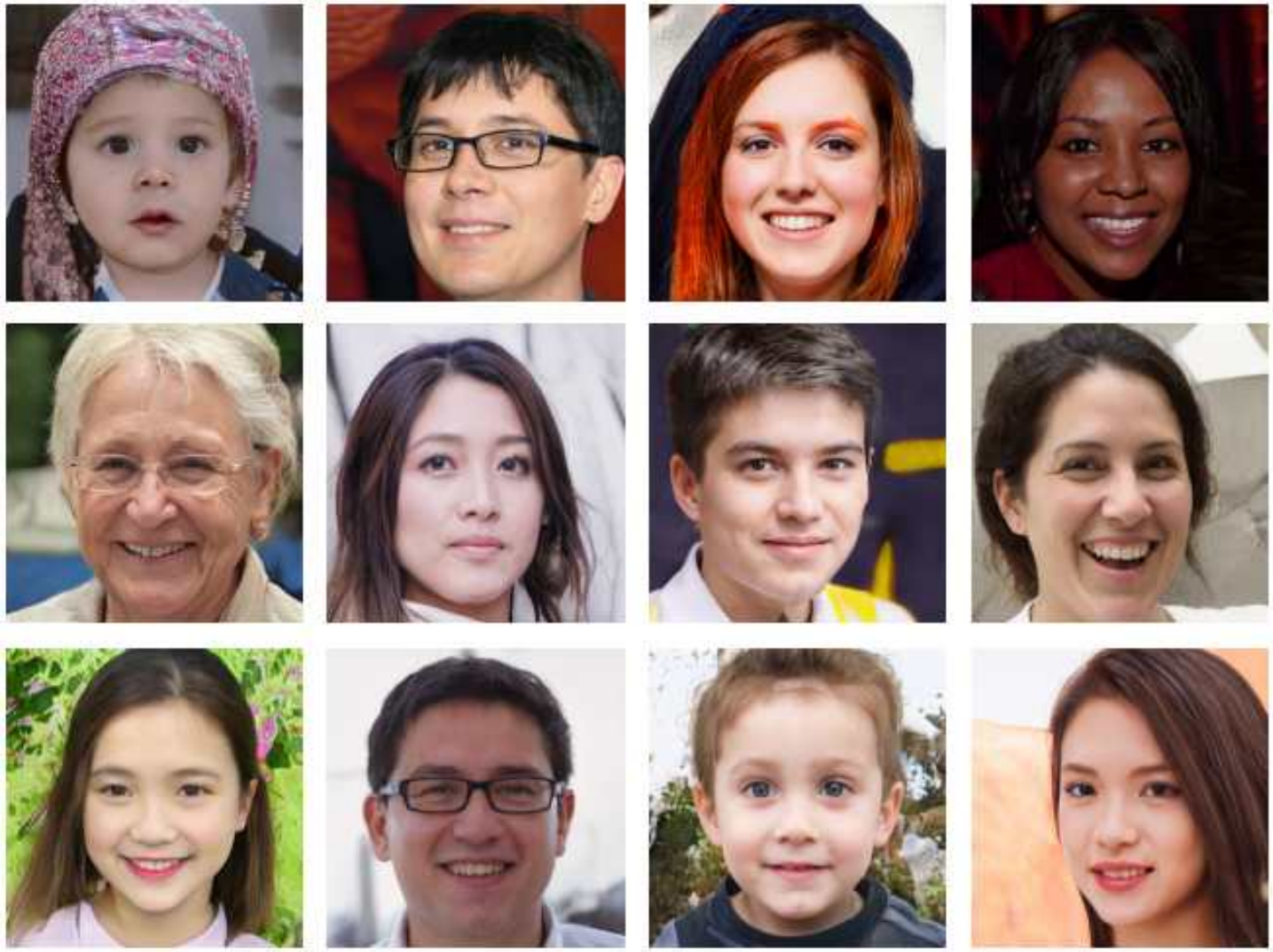
2 진위를 가려내는 장치, 판별자

3 적대적 신경망 실행하기

4 이미지의 특징을 추출하는 오토인코더

세상에 없는 얼굴 GAN, 오토인코더

- 세상에 없는 얼굴 GAN, 오토인코더



세상에 없는 얼굴 GAN, 오토인코더

- **세상에 없는 얼굴 GAN, 오토인코더**
  - 위 사진을 볼까?
  - 세계 각국에서 살아가는 인종과 성별이 다른 여러 사람들의 사진
  - 아무도 위 사진 속 사람들을 직접 만난 적은 없음
  - 세상에 존재하는 이들이 아니기 때문임
  - 모두 딥러닝이 만들어 낸 가상의 얼굴

세상에 없는 얼굴 GAN, 오토인코더

- **세상에 없는 얼굴 GAN, 오토인코더**
  - **생성적 적대 신경망**(Generative Adversarial Networks), 줄여서 '**GAN**(간)'이라고 부르는 알고리즘을 이용해 만든 것
  - GAN은 딥러닝의 원리를 활용해 가상 이미지를 생성하는 알고리즘
  - 예를 들어 얼굴을 만든다면, 이미지 픽셀들이 어떻게 조합되어야 우리가 생각하는 '얼굴'의 형상이 되는지 딥러닝 알고리즘이 예측한 결과가 앞에 나온 이미지

세상에 없는 얼굴 GAN, 오토인코더

- **세상에 없는 얼굴 GAN, 오토인코더**
  - GAN이라는 이름에는 적대적(adversarial, 서로 대립 관계에 있는)이란 단어가 들어 있는데, 이것은 GAN 알고리즘의 성격을 잘 말해 줌
  - 진짜 같은 가짜를 만들기 위해 GAN 알고리즘 내부에서는 '적대적' 경합을 진행하기 때문임
  - 이 적대적 경합을 쉽게 설명하기 위해 GAN의 아이디어를 처음으로 제안한 이안 굿펠로(Ian Goodfellow)는 그의 논문에서 위조지폐범과 경찰의 예를 들었음
  - 진짜 지폐와 똑같은 위조지폐를 만들기 위해 애쓰는 위조지폐범과 이를 가려내기 위해 노력하는 경찰 사이의 경합이 결국 더 정교한 위조지폐를 만들어 낸다는 것
  - 한쪽은 가짜를 만들고, 한쪽은 진짜와 비교하는 경합 과정을 이용하는 것이 바로 GAN의 원리

세상에 없는 얼굴 GAN, 오토인코더

▼ 그림 19-1 | 생성자는 판별자가 구별할 수 없을 만큼 정교한 가짜를 만드는 것이 목표!

세상에 없는 얼굴 GAN, 오토인코더

- **세상에 없는 얼굴 GAN, 오토인코더**
  - 가짜를 만들어 내는 파트를 '생성자(Generator)', 진위를 가려내는 파트를 '판별자(Discriminator)'라고 함
  - 이러한 기본 구조 위에 여러 아이디어를 더한 GAN의 변형 알고리즘들이 지금도 계속해서 발표되고 있음
  - 페이스북의 AI 연구 팀이 만들어 발표한 DCGAN(Deep Convolutional GAN)도 그중 하나
  - DCGAN은 우리가 앞서 배운 컨볼루션 신경망을 GAN에 적용한 알고리즘인데, 지금의 GAN이 되게끔 해 주었다고 해도 과언이 아닐 만큼 불안정하던 초기의 GAN을 크게 보완해 주었음
  - 지금부터 이 DCGAN을 직접 만들고 그 원리를 공부해 보자

1 가짜 제조 공장, 생성자

1 가짜 제조 공장, 생성자

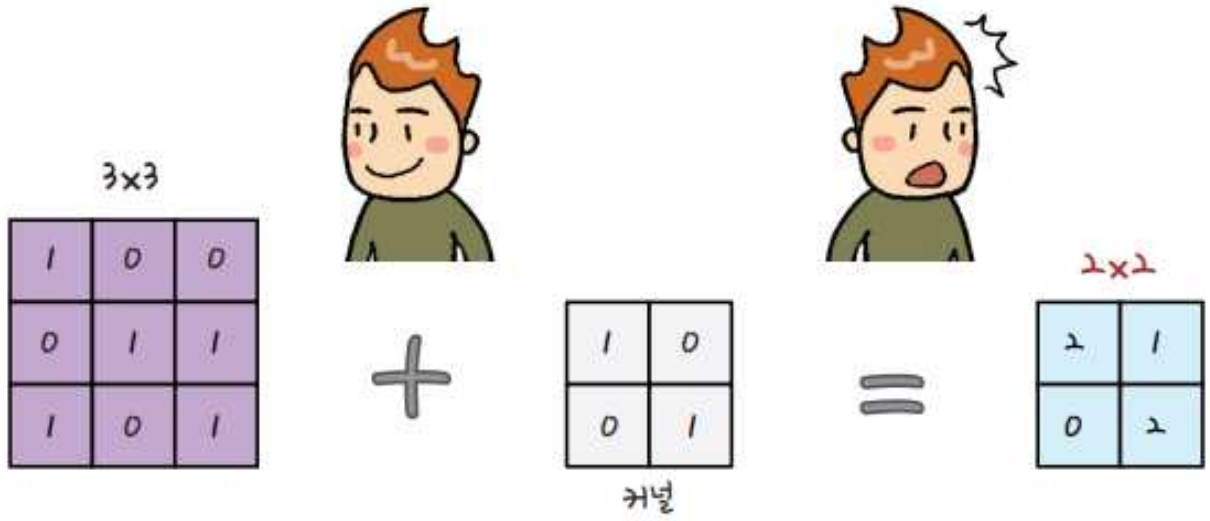
- **가짜 제조 공장, 생성자**
  - **생성자**(generator)는 가상의 이미지를 만들어 내는 공장
  - 처음에는 랜덤한 픽셀 값으로 채워진 가짜 이미지로 시작해서 판별자의 판별 결과에 따라 지속적으로 업데이트하며 점차 원하는 이미지를 만들어 감
  - DCGAN은 생성자가 가짜 이미지를 만들 때 컨볼루션 신경망(CNN)을 이용한다고 했음
  - 우리는 컨볼루션 신경망을 이미 배웠는데 DCGAN에서 사용되는 컨볼루션 신경망은 앞서 나온 것과 조금 차이가 있음
  - 먼저 옵티마이저(optimizer)를 사용하는 최적화 과정이나 컴파일하는 과정이 없다는 것
  - 판별과 학습이 이곳 생성자에서 일어나는 것이 아니기 때문임

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 일부 매개변수를 삭제하는 풀링(pooling) 과정이 없는 대신 앞 장에서 배운 패딩(padding) 과정이 포함
  - 빈 곳을 채워서 같은 크기로 맞추는 패딩 과정이 여기서 다시 등장하는 이유는 입력 크기와 출력 크기를 똑같이 맞추기 위해서임
  - 커널을 이동하며 컨볼루션 층을 만들 때 이미지의 크기가 처음보다 줄어든다는 것을 떠올려 보면 패딩 과정이 왜 필요한지 알 수 있음

1 가짜 제조 공장, 생성자

▼ 그림 19-2 | 생성자의 결과물이 실제 데이터와 같은 크기여야 하므로 차원 축소 문제를
해결해야 한다



1 가짜 제조 공장, 생성자

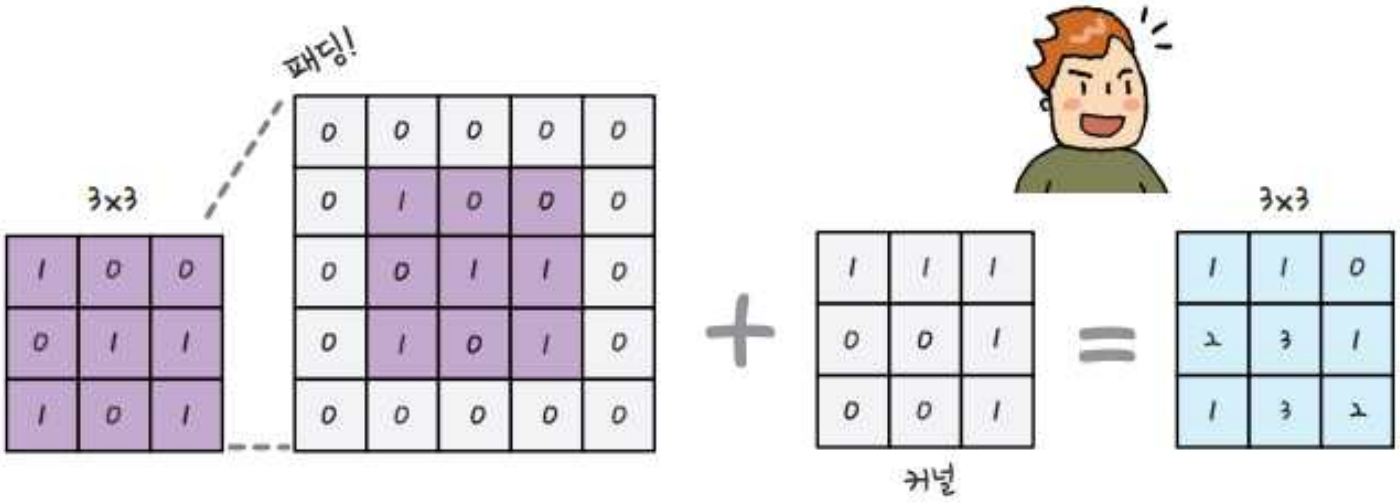
- **가짜 제조 공장, 생성자**
  - 패딩 과정을 통해 생성자가 만들어 내는 이미지의 크기를 조절해야 하는 이유는 판별자가 비교할 '진짜'와 똑같은 크기가 되어야 하기 때문임
  - 예를 들어 MNIST 손글씨 인식을 사용한다면, 모든 손글씨 사진이 가로가 28픽셀이고 세로가 28픽셀의 크기로 되어 있으므로 생성자에서 만들어질 이미지 역시 28×28의 크기가 되어야 함

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 케라스의 패딩 함수는 이러한 문제를 쉽게 처리할 수 있도록 도와줌
  - padding='same'이라는 설정을 통해 입력과 출력의 크기가 다를 경우 자동으로 크기를 확장해 주고, 확장된 공간에 0을 채워 넣을 수 있음(그림 19-3 참조)

1 가짜 제조 공장, 생성자

▼ 그림 19-3 | 패딩을 이용하면 커널을 거친 후에도 차원의 변동이 없게 만들 수 있다!



1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 패딩 외에도 알아야 할 것들이 몇 가지 더 있음
  - DCGAN의 제안자들은 학습에 꼭 필요한 옵션들을 제시했는데, 그중 하나가 **배치 정규화**(Batch Normalization)라는 과정
  - 배치 정규화란 입력 데이터의 평균이 0, 분산이 1이 되도록 재배치하는 것인데, 다음 층으로 입력될 값을 일정하게 재배치하는 역할을 함
  - 이 과정을 통해 층의 개수가 늘어나도 안정적인 학습을 진행할 수 있음
  - 케라스는 이를 쉽게 적용할 수 있게끔 BatchNormalization() 함수를 제공

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 생성자의 활성화 함수로는 ReLU( ) 함수를 쓰고 판별자로 넘겨주기 직전에는 tanh()함수를 쓰고 있음
  - Tanh( ) 함수를 쓰면 출력되는 값을 -1~1 사이로 맞출 수 있음
  - 판별자에 입력될 MNIST 손글씨의 픽셀 범위도 -1~1로 맞추면 판별 조건이 모두 갖추어짐

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 지금까지 설명한 내용을 코드로 정리하면 다음과 같음

```
generator = Sequential() # 모델 이름을 generator로 정하고 Sequential() 함수를 호출
generator.add(Dense(128*7*7, input_dim=100, activation=LeakyReLU(0.2))) ····· ❶
generator.add(BatchNormalization()) ····· ❷
generator.add(Reshape((7, 7, 128))) ····· ❸
generator.add(UpSampling2D()) ····· ❹
generator.add(Conv2D(64, kernel_size=5, padding='same')) ····· ❺
generator.add(BatchNormalization()) ····· ❻
generator.add(Activation(LeakyReLU(0.2))) ····· ❼
generator.add(UpSampling2D()) ····· ❽
generator.add(Conv2D(1, kernel_size=5, padding='same', activation='tanh'))
····· ❾
```

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 먼저 ❶ 부터 차례로 확인해 보자

```
generator.add(Dense(128*7*7, input_dim=100, activation=LeakyReLU(0.2))) ····· ❶
```

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 여기서 128은 임의로 정한 노드의 수
  - 128이 아니어도 충분한 노드를 마련해 주면 됨
  - input_dim=100은 100차원 크기의 랜덤 벡터를 준비해 집어넣으라는 의미
  - 꼭 100이 아니어도 좋음
  - 적절한 숫자를 임의로 넣어 주면 됨
  - 여기서 주의할 부분은 7*7
  - 이는 이미지의 최초 크기를 의미
  - MNIST 손글씨 이미지의 크기는 28×28인데, 왜 7×7 크기의 이미지를 넣어 줄까?

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - ❹ 와 ❽ 을 보면 답이 있음
  - UpSampling2D() 함수를 사용
  - UpSampling2D()함수는 이미지의 가로세로 크기를 두 배씩 늘려 줌
  - 7×7이 ❹ 레이어를 지나며 그 크기가 14×14가 되고, ❽레이어를 지나며 28×28이 되는 것
  - 이렇게 작은 크기의 이미지를 점점 늘려 가면서 컨볼루션 층(❺, ❾)을 지나치게 하는 것이 DCGAN의 특징

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 케라스는 Upsampling과 Conv2D를 합쳐 놓은 Conv2DTranspose() 함수도 제공
  - ❹ , ❺ 와 ❽, ❾를 각각 없애고 이 함수 하나만 써도 되는데, 여기서는 이해를 돕기 위해 두 개로 나누어 쓰고 설명

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - ❸ 은 컨볼루션 레이어가 받아들일 수 있는 형태로 바꾸어 주는 코드
  - Conv2D() 함수의 input_shape 부분에 들어갈 형태로 정해 줌

```
generator.add(Reshape((7, 7, 128))) ····· ❸
```

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - ❹ ,❺ 그리고 ❽,❾ 는 두 배씩 업(up)샘플링을 한 후 컨볼루션 과정을 처리
  - 커널 크기를 5로 해서 5×5 크기의 커널을 썼음
  - 바로 앞서 설명했듯이 padding='same' 조건 때문에 모자라는 부분은 자동으로

```
generator.add(UpSampling2D()) ····❹
generator.add(Conv2D(64, kernel_size=5, padding='same')) ····❺
```

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - ❶ 과❼ 에서 활성화 함수로 LeakyReLU(리키렐루)를 썼음
  - GAN에서는 기존에 사용하던 ReLU() 함수를 쓸 경우 학습이 불안정해지는 경우가 많아, ReLU()를 조금 변형한 Leaky ReLU() 함수를 씀

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - LeakyReLU() 함수는 ReLU() 함수에서 x 값이 음수이면 무조건 0이 되어 뉴런들이 일찍 소실되는 단점을 보완하기 위해, 0 이하에서도 작은 값을 갖게 만드는 활성화 함수
  - 케라스 함수를 이용해 LeakyReLU(0.2) 형태로 설정하면 0보다 작을 경우 0.2를 곱하라는 의미

```
generator.add(Dense(128*7*7, input_dim=100, activation=LeakyReLU(0.2))) ····❶
generator.add(Activation(LeakyReLU(0.2))) ····❼
```

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - ❷ ❻ 에서는 데이터의 배치를 정규 분포로 만드는 배치 정규화가 진행

```
generator.add(BatchNormalization()) ····❷
generator.add(BatchNormalization()) ····❻
```

1 가짜 제조 공장, 생성자

- **가짜 제조 공장, 생성자**
  - 끝으로⑨ 에서 한 번 더 컨볼루션 과정을 거친 후 판별자로 값을 넘길 준비를 마침
  - 앞서 이야기한 대로 활성화 함수는 tanh() 함수를 썼음

```
generator.add(Conv2D(1, kernel_size=5, padding='same', activation='tanh'))
····⑨
```

2 진위를 가려내는 장치, 판별자

2 진위를 가려내는 장치, 판별자

- **진위를 가려내는 장치, 판별자**
  - 이제 생성자에서 넘어온 이미지가 가짜인지 진짜인지를 판별해 주는 장치인 **판별자**(discriminator)를 만들 차례
  - 이 부분은 컨볼루션 신경망의 구조를 그대로 가지고 와서 만들면 됨
  - 컨볼루션 신경망이란 원래 무언가를(예를 들어 개와 고양이 사진을) 구별하는 데 최적화된 알고리즘이기 때문에 그 목적 그대로 사용하면 되는 것

2 진위를 가려내는 장치, 판별자

- **진위를 가려내는 장치, 판별자**
  - 진짜(1) 아니면 가짜(0), 둘 중 하나를 결정하는 문제이므로 컴파일 부분은 14장에서 사용된 이진 로스 함수(binary_crossentropy)와 최적화 함수(adam)를 그대로 쓰겠음
  - 16장에서 배웠던 드롭아웃(Dropout(0.3))도 다시 사용하고, 앞 절에서 다룬 배치 정규화와 패딩도 그대로 넣어 줌

2 진위를 가려내는 장치, 판별자

- **진위를 가려내는 장치, 판별자**
  - 주의할 점은 이 판별자는 가짜인지 진짜인지를 판별만 해 줄 뿐, 자기 자신이 학습을 해서는 안 된다는 것
  - 판별자가 얻은 가중치는 판별자 자신이 학습하는 데 쓰이는 것이 아니라 생성자로 넘겨주어 생성자가 업데이트된 이미지를 만들도록 해야 함
  - 판별자를 만들 때는 가중치를 저장하는 학습 기능을 꺼 주어야 함

2 진위를 가려내는 장치, 판별자

- **진위를 가려내는 장치, 판별자**
  - 모든 과정을 코드로 정리해 보면 다음과 같음

```
# 모델 이름을 discriminator로 정하고 Sequential() 함수를 호출합니다.
discriminator = Sequential()
discriminator.add(Conv2D(64, kernel_size=5, strides=2, input_shape=
(28,28,1), padding="same")) ····❶
discriminator.add(Activation(LeakyReLU(0.2))) ····❷
discriminator.add(Dropout(0.3)) ····❸
discriminator.add(Conv2D(128, kernel_size=5, strides=2, padding="same")) ····❹
discriminator.add(Activation(LeakyReLU(0.2))) ····❺
discriminator.add(Dropout(0.3)) ····❻
discriminator.add(Flatten()) ····❼
discriminator.add(Dense(1, activation='sigmoid')) ····❽
discriminator.compile(loss='binary_crossentropy', optimizer='adam') ····❾
discriminator.trainable = False ····❿
```

2 진위를 가려내는 장치, 판별자

- **진위를 가려내는 장치, 판별자**
  - 먼저❶ ❹ 를 살펴보면 노드의 수는 각각 64개, 128개로 정했고, 커널 크기는 5로 설정해 5×5 커널이 사용된다는 것을 알 수 있음

```
discriminator.add(Conv2D(64, kernel_size=5, strides=2, input_shape=
(28,28,1), padding="same")) ····❶
discriminator.add(Conv2D(128, kernel_size=5, strides=2, padding="same")) ····❹
```

2 진위를 가려내는 장치, 판별자

- **진위를 가려내는 장치, 판별자**
  - 여기에 strides 옵션이 처음 등장
  - strides는 커널 윈도를 몇 칸씩 이동시킬지 정하는 옵션
  - 특별한 설정이 없으면 커널 윈도는 한 칸씩 움직임
  - strides=2라고 설정했다는 것은 커널 윈도를 두 칸씩 움직이라는 뜻

2 진위를 가려내는 장치, 판별자

- **진위를 가려내는 장치, 판별자**
  - strides를 써서 커널 윈도를 여러 칸 움직이게 하는 이유는 무엇일까?
  - 가로세로 크기가 더 줄어들어 새로운 특징을 뽑아 주는 효과가 생기기 때문임
  - 드롭아웃이나 풀링처럼 새로운 필터를 적용한 효과가 생기는 것
  - 생성자에서는 출력 수를 28로 맞추어야 하기 때문에 오히려 업샘플링을 통해 가로세로의 수를 늘려 주었지만 판별자는 진짜와 가짜만 구분하면 되기 때문에 그럴 필요가 없음
  - strides나 드롭아웃(❸,❻) 등 차원을 줄여 주는 기능을 적극적으로 사용하면서 컨볼루션 신경망 본래의 목적을 달성하면 됨

2 진위를 가려내는 장치, 판별자

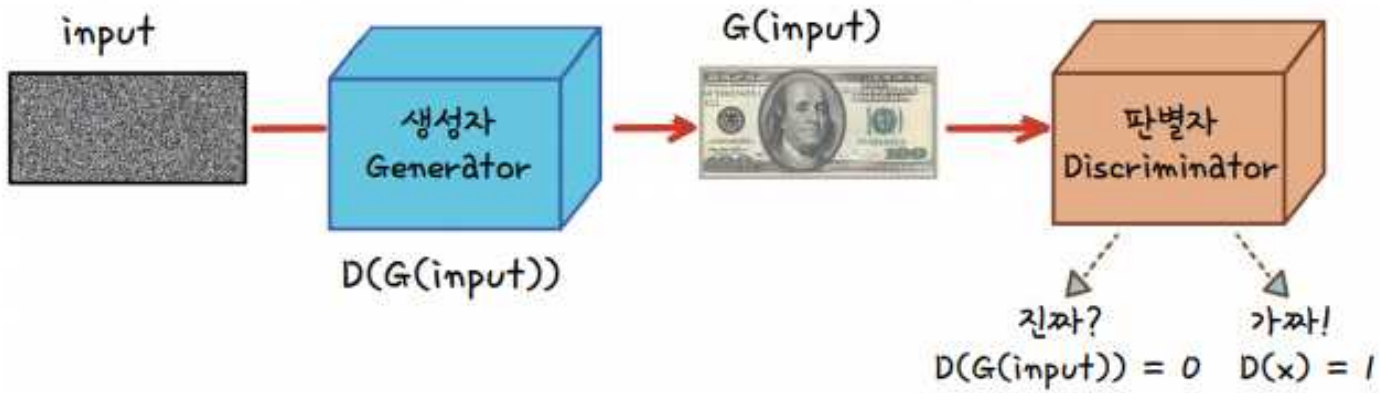
- **진위를 가려내는 장치, 판별자**
  - ❷,❺ 는 활성화 함수로 LeakyReLU() 함수를 사용한 것을 보여 줌
  - ❼,❽ 은 가로×세로의 2차원으로 진행된 과정을 1차원으로 바꾸어 주는 Flatten() 함수와 마지막 활성화 함수로 sigmoid() 함수를 사용하는 과정
  - 판별의 결과가 진짜(1) 혹은 가짜(0), 둘 중에 하나가 되어야 하므로 sigmoid() 함수를 썼음
  - ❾ 에서는 이제 이진 로스 함수(binary_crossentropy)와 최적화 함수(adam)를 써서 판별자에 필요한 준비를 마무리
  - ❿ 에서는 앞서 설명한 대로 판별이 끝나고 나면 판별자 자신이 학습되지 않게끔 학습 기능을 꺼 줌
  - Discriminator.trainable = False라는 옵션으로 이를 설정할 수 있음

3 적대적 신경망 실행하기

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 이제 생성자와 판별자를 연결시키고 학습을 진행하며 기타 여러 가지 옵션을 설정하는 순서
  - 생성자와 판별자를 연결시킨다는 것은 생성자에서 나온 출력을 판별자에 넣어서 진위 여부를 판별하게 만든다는 의미
  - 282쪽에 나온 그림에 생성자를 G(), 판별자를 D(), 실제 데이터를 x, 입력 값을 input이라고 하여 다시 나타내면 그림 19-4와 같음

▼ 그림 19-4 | 생성자 모델이 G(), 판별자 모델이 D(), 실제 데이터가 x, 입력 값이 input일 때 GAN의 기본 구조



3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 생성자 G()에 입력 값 input을 넣은 결과는 G(input)
  - 이것을 판별자 D()에 넣은 결과는 D(G(input))이 됨
  - 생성자는 D(G(input))이 참(1)이라고 주장하지만, 판별자는 실제 데이터인 x로 만든 D(x)만이 참이라고 여김
  - 그러다 학습이 진행될수록 생성자가 만든 G(input)이 실제와 너무나 가까워져서 이것으로 만든 D(G(input))과 실제 데이터로 만든 D(x)를 잘 구별하지 못하게 됨
  - 너무나 유사해진 D(G(input))과 D(x)를 판별자가 더는 구별하지 못하게 되어 정확도가 0.5에 가까워질 때, 비로소 생성자는 자신의 역할을 다하게 되어 학습은 종료

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 이제 이것을 코드로 만들겠음

```
ginput = Input(shape=(100,)) ····❶
dis_output = discriminator(generator(ginput)) ····❷
gan = Model(ginput, dis_output) ····❸
gan.compile(loss='binary_crossentropy', optimizer='adam') ····❹
```

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - ❶ 은 랜덤한 100개의 벡터를 케라스의 Input( ) 함수에 집어넣어 생성자에 입력할 ginput을 만드는 과정
  - ❷ 는 생성자 모델 generator( )에 ❶ 에서 만든 ginput을 입력
  - 그 결과 출력되는 28×28 크기의 이미지가 그대로 판별자 모델 discriminator( )의 입력 값으로 들어감
  - 판별자는 이 입력 값을 가지고 참과 거짓을 판별하는데, 그 결과를 dis_output이라고 하겠음
  - ❸ 에서는 케라스의 Model() 함수를 이용해 ginput 값과 ❷ 에서 구한 dis_output 값을 넣어 gan이라는 이름의 새로운 모델을 만듦
  - ❹ 에서는 참과 거짓을 구분하는 이진 로스 함수(binary_crossentropy)와 최적화 함수(adam)를 사용해 ❸ 에서 만든 gan 모델을 컴파일
  - 드디어 생성자와 판별자를 연결하는 gan 모델까지 만들었음

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 이제 지금까지 모든 과정을 실행할 함수를 만들 차례
  - gan_train( ) 함수를 사용해 학습이 진행되도록 하겠음
  - 이때 변수는 epoch, batch_size 그리고 중간 과정을 저장할 때 몇 번마다 한 번씩 저장할지 정하는 saving_interval 이렇게 세 가지로 정함
  - 판별자에서 사용할 MNIST 손글씨 데이터도 불러 줌
  - 앞서 생성자 편에서 tanh() 함수를 사용한 이유는 지금 불러올 이 데이터의 픽셀 값을 -1~1 사이의 값으로 지정하기 위해서였음
  - 0~255의 값으로 되어 있는 픽셀 값을 -1~1 사이의 값으로 바꾸려면 현재의 픽셀 값에서 127.5를 뺀 후 127.5로 나누면 됨

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
# 실행 함수를 선언합니다.
def gan_train(epoch, batch_size, saving_interval): # 세 가지 변수 지정

# MNIST 데이터 불러오기
    # MNIST 데이터를 다시 불러와 이용합니다. 단, 테스트 과정은 필요 없고
    # 이미지만 사용할 것이기 때문에 X_train만 호출합니다.
    (X_train, _), (_, _) = mnist.load_data()

    # 가로 28픽셀, 세로 28픽셀이고 흑백이므로 1을 설정합니다.
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')

    # 0~255 사이 픽셀 값에서 127.5를 뺀 후 127.5로 나누면 -1~1 사이 값으로 바뀝니다.
    X_train = (X_train - 127.5) / 127.5
```

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - batch_size는 한 번에 몇 개의 실제 이미지와 몇 개의 가상 이미지를 판별자에 넣을지 결정하는 변수
  - 먼저 batch_size만큼 MNIST 손글씨 이미지를 랜덤하게 불러와 판별자에 집어넣는 과정은 다음과 같음
  - 실제 이미지를 입력했으므로 '모두 참(1)'이라는 레이블을 붙임

```
true = np.ones((batch_size, 1)) ····❶
idx = np.random.randint(0, X_train.shape[0], batch_size) ····❷
imgs = X_train[idx] ····❸
d_loss_real = discriminator.train_on_batch(imgs, true) ····❹
```

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - ❶ 에서는 '모두 참(1)'이라는 레이블 값을 가진 배열을 만듦
  - batch_size 길이만큼 만들어 ❹ 에서 사용
  - ❷ 에서는 넘파이 라이브러리의 random( ) 함수를 사용해서 실제 이미지를 랜덤하게 선택해 불러옴
  - np.random.randint(a, b, c)는 a부터 b까지 숫자 중 하나를 랜덤하게 선택해 가져오는 과정을 c번 반복하라는 의미
  - 0부터 X_train 개수 사이의 숫자를 랜덤하게 선택해 batch_size만큼 반복해서 가져오게 했음
  - ❸ 에서는 ❷ 에서 선택된 숫자에 해당하는 이미지를 불러옴

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - ❹ 에서는 판별자 모델에 train_on_batch( ) 함수를 써서 판별을 시작
  - train_on_batch(x, y) 함수는 입력 값(x)과 레이블(y)을 받아서 딱 한 번 학습을 실시해 모델을 업데이트
  - ❸ 에서 만든 이미지를 x에 넣고 ❶ 에서 만든 배열을 y에 놓아 준비를 마침

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 실제 이미지에 이어서 이번에는 생성자에서 만든 가상의 이미지를 판별자에 넣겠음
  - 가상의 이미지는 '모두 거짓(0)'이라는 레이블을 준비해 붙임
  - 학습이 반복될수록 가짜라는 레이블을 붙인 이미지들에 대한 예측 결과가 거짓으로 나올 것

```
fake = np.zeros((batch_size, 1)) ····❶
noise = np.random.normal(0, 1, (batch_size, 100)) ····❷
gen_imgs = generator.predict(noise) ····❸
d_loss_fake = discriminator.train_on_batch(gen_imgs, fake) ····❹
```

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - ❶ 에서는 '모두 거짓(0)'이라는 레이블 값을 가진 열을 batch_size 길이만큼 만듦
  - ❷ 에서는 생성자에 집어넣을 가상 이미지를 만듭니다. 정수가 아니기 때문에 np.random.normal( ) 함수를 사용
  - 조금 전과 마찬가지로 np.random.normal(a, b, c) 형태를 가지며 a부터 b까지 실수 중 c개를 랜덤으로 뽑으라는 의미
  - 여기서 c 자리에 있는(batch_size, 100)은 batch_size만큼 100열을 뽑으라는 의미
  - ❸ 에서는 ❷ 에서 만들어진 값이 생성자에 들어가고 결괏값이 gen_imgs로 저장
  - ❹ 에서는 ❸ 에서 만든 값에 ❶에서 만든 '모두 거짓(0)'이라는 레이블이 붙음
  - 이대로 판별자로 입력

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 이제 실제 이미지를 넣은 d_loss_real과 가상 이미지를 입력한 d_loss_fake가 판별자 안에서 번갈아 가며 진위를 판단하기 시작
  - 각각 계산되는 오차의 평균을 구하면 판별자의 오차 d_loss는 다음과 같이 정리

```
# d_loss_real, d_loss_fake 값을 더해 둘로 나눈 평균이 바로 판별자의 오차
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 이제 마지막 단계
  - 판별자와 생성자를 연결해서 만든 gan 모델을 이용해 생성자의 오차, g_loss를 구하면 다음과 같음
  - 역시 train_on_batch( ) 함수와 앞서 만든 gen_imgs를 사용
  - 생성자의 레이블은 무조건 참(1)이라 해놓고 판별자로 넘김
  - 이번에도 앞서 만든 true 배열로 레이블을 붙임

```
g_loss = gan.train_on_batch(noise, true)
```

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 학습이 진행되는 동안 생성자와 판별자의 오차가 출력되게 하겠음

```
print('epoch:%d' % i, ' d_loss:%.4f' % d_loss, ' g_loss:%.4f' % g_loss)
```

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - 이제 실행할 준비를 마쳤음
  - 앞서 배운 GAN의 모든 과정을 한곳에 모으면 다음과 같음

실습 | GAN 모델 만들기

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Input, Dense, Reshape,Flatten, Dropout
from tensorflow.keras.layers import BatchNormalization, Activation,
LeakyReLU, UpSampling2D, Conv2D
from tensorflow.keras.models import Sequential, Model

import numpy as np
import matplotlib.pyplot as plt
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
# 생성자 모델을 만듭니다.
generator = Sequential()
generator.add(Dense(128*7*7, input_dim=100, activation=LeakyReLU(0.2)))
generator.add(BatchNormalization())
generator.add(Reshape((7, 7, 128)))
generator.add(UpSampling2D())
generator.add(Conv2D(64, kernel_size=5, padding='same'))
generator.add(BatchNormalization())
generator.add(Activation(LeakyReLU(0.2)))
generator.add(UpSampling2D())
generator.add(Conv2D(1, kernel_size=5, padding='same', activation='tanh'))
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
# 판별자 모델을 만듭니다.
discriminator = Sequential()
discriminator.add(Conv2D(64, kernel_size=5, strides=2,
input_shape=(28,28,1), padding="same"))
discriminator.add(Activation(LeakyReLU(0.2)))
discriminator.add(Dropout(0.3))
discriminator.add(Conv2D(128, kernel_size=5, strides=2, padding="same"))
discriminator.add(Activation(LeakyReLU(0.2)))
discriminator.add(Dropout(0.3))
discriminator.add(Flatten())
discriminator.add(Dense(1, activation='sigmoid'))
discriminator.compile(loss='binary_crossentropy', optimizer='adam')
discriminator.trainable = False
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
# 생성자와 판별자 모델을 연결시키는 gan 모델을 만듭니다.
ginput = Input(shape=(100,))
dis_output = discriminator(generator(ginput))
gan = Model(ginput, dis_output)
gan.compile(loss='binary_crossentropy', optimizer='adam')
gan.summary()

# 신경망을 실행시키는 함수를 만듭니다.
def gan_train(epoch, batch_size, saving_interval):

# MNIST 데이터 불러오기
    # 앞서 불러온 MNIST를 다시 이용합니다. 테스트 과정은 필요 없고
    # 이미지만 사용할 것이기 때문에 X_train만 호출합니다.
    (X_train, _), (_, _) = mnist.load_data()
    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
# 127.5를 뺀 후 127.5로 나누어서 -1~1 사이의 값으로 바꿉니다.
X_train = (X_train - 127.5) / 127.5

true = np.ones((batch_size, 1))
fake = np.zeros((batch_size, 1))

for i in range(epoch):
    # 실제 데이터를 판별자에 입력하는 부분입니다.
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx]
    d_loss_real = discriminator.train_on_batch(imgs, true)
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
# 가상 이미지를 판별자에 입력하는 부분입니다.
noise = np.random.normal(0, 1, (batch_size, 100))
gen_imgs = generator.predict(noise)
d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)

# 판별자와 생성자의 오차를 계산합니다.
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
g_loss = gan.train_on_batch(noise, true)

print('epoch:%d' % i, ' d_loss:%.4f' % d_loss, ' g_loss:%.4f' % g_loss)
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
        # 중간 과정을 이미지로 저장하는 부분입니다. 정해진 인터벌만큼 학습되면
        # 그때 만든 이미지를 gan_images 폴더에 저장하라는 의미입니다.
        # 이 코드는 이 장의 주된 목표와는 관계가 없어서 소스 코드만 소개합니다.
        if i % saving_interval == 0:
        # r, c = 5, 5
        noise = np.random.normal(0, 1, (25, 100))
        gen_imgs = generator.predict(noise)

        # Rescale images 0 - 1
        gen_imgs = 0.5 * gen_imgs + 0.5

        fig, axs = plt.subplots(5, 5)
        count = 0
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

```
        for j in range(5):
            for k in range(5):
                axs[j, k].imshow(gen_imgs[count, :, :, 0], cmap='gray')
                axs[j, k].axis('off')
                count += 1
        fig.savefig("gan_images/gan_mnist_%d.png" % i)

# 2000번 반복되고(+1을 하는 것에 주의),
# 배치 크기는 32, 200번마다 결과가 저장됩니다.
gan_train(2001, 32, 200)
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

실행 결과

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 100)]             0
_________________________________________________________________
sequential_2 (Sequential)    (None, 28, 28, 1)         865281
_________________________________________________________________
sequential_3 (Sequential)    (None, 1)                 212865
=================================================================
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

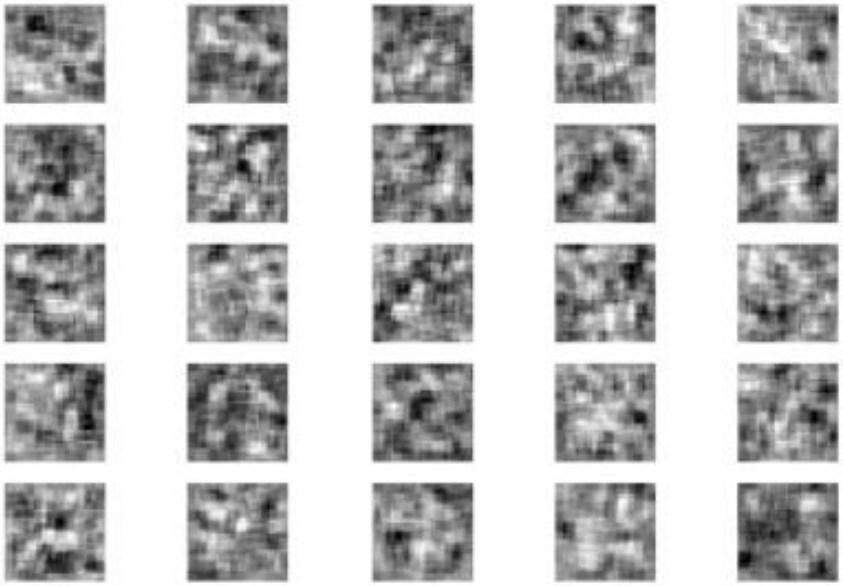
```
Total params: 1,078,146
Trainable params: 852,609
Non-trainable params: 225,537
_________________________________________________________________
epoch:0  d_loss:0.7238  g_loss:0.5319
... (중략) ...
epoch:2000  d_loss:0.4667  g_loss:2.1844
```

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

▼ 그림 19-5 | GAN 실행 결과

시작 이미지:



3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

Epoch 1000:

3 적대적 신경망 실행하기

- 적대적 신경망 실행하기

Epoch 2000:

3 적대적 신경망 실행하기

- **적대적 신경망 실행하기**
  - Epoch 2000까지 진행했을 때의 결과
  - 점점 숫자에 가까운 형태가 만들어짐
  - 생성된 이미지들은 gan_images 폴더에 저장

4 이미지의 특징을 추출하는 오토인코더

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 딥러닝을 이용해 가상의 이미지를 만드는 또 하나의 유명한 알고리즘이 있음
  - 바로 **오토인코더**(Auto-Encoder, AE)
  - 지금까지 설명한 GAN을 이해했다면 오토인코더의 핵심적인 부분은 이미 거의 이해한 셈
  - 다음 장으로 넘어가기 전에 오토인코더 의미를 알아보고 실습해 보자

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 오토인코더는 GAN과 비슷한 결과를 만들지만, 다른 성질을 지니고 있음
  - GAN이 세상에 존재하지 않는 완전한 가상의 것을 만들어 내는 반면에, 오토인코더는 입력 데이터의 특징을 효율적으로 담아낸 이미지를 만들어 냄
  - 예를 들어 GAN으로 사람의 얼굴을 만들면 진짜 같아 보여도 실제로는 존재하지 않는 완전한 가상 이미지가 만들어짐
  - 오토인코더로 사람의 얼굴을 만들 경우 초점이 좀 흐릿하고 윤곽이 불명확하지만 사람의 특징을 유추할 수 있는 것들이 모여 이미지가 만들어짐

4 이미지의 특징을 추출하는 오토인코더

▼ 그림 19-6 | GAN으로 만든 이미지(왼쪽), 오토인코더로 만든 이미지(오른쪽)

GAN으로 만든 결과

오토인코더로 만든 결과

4 이미지의 특징을 추출하는 오토인코더

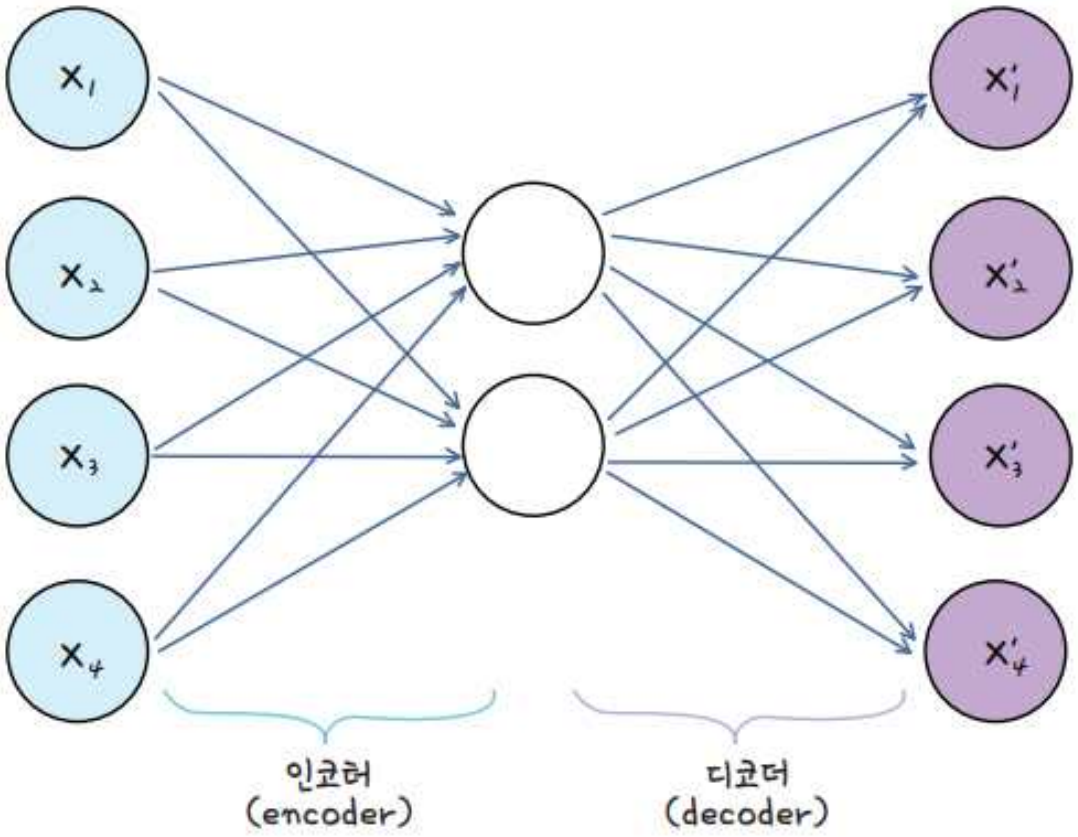
- **이미지의 특징을 추출하는 오토인코더**
  - 오토인코더는 과연 어디에 활용할 수 있을까?
  - 영상 의학 분야 등 아직 데이터 수가 충분하지 않은 분야에서 사용될 수 있음
  - 학습 데이터는 현실 세계의 정보를 담고 있어야 하므로, 세상에 존재하지 않는 가상의 것을 집어넣으면 예상치 못한 결과를 가져올 수 있음
  - 데이터의 특징을 잘 담아내는 오토인코더라면 다름
  - 부족한 학습 데이터 수를 효과적으로 늘려 주는 효과를 기대할 수 있음

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 오토인코더의 학습은 GAN의 학습보다 훨씬 쉬움
  - 이전 절에서 GAN의 원리를 이해했다면 매우 수월하게 익힐 수 있을 것
  - 오토인코더의 원리를 그림으로 표현하면 그림 19-7과 같음

4 이미지의 특징을 추출하는 오토인코더

▼ 그림 19-7 | 오토인코더의 구조. 입력 값(X)과 출력 값(X') 사이에 차원이 작은 노드가 포함됨



4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 그림 19-7을 보면 입력한 이미지와 똑같은 크기로 출력층을 만들었음
  - 입력층보다 적은 수의 노드를 가진 은닉층을 중간에 넣어서 차원을 줄여 줌
  - 이때 소실된 데이터를 복원하기 위해 학습을 시작하고, 이 과정을 통해 입력 데이터의 특징을 효율적으로 응축한 새로운 출력이 나오는 원리

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 가장 핵심이 되는 인코딩과 디코딩 과정을 코딩해 보면 다음과 같음

```
# 생성자 모델 만들기
autoencoder = Sequential()

# 인코딩 부분
autoencoder.add(Conv2D(16, kernel_size=3, padding='same', input_shape=(28,
28,1), activation='relu')) ····❶
autoencoder.add(MaxPooling2D(pool_size=2, padding='same')) ····❷
autoencoder.add(Conv2D(8, kernel_size=3, activation='relu', padding='same'))
····❸
autoencoder.add(MaxPooling2D(pool_size=2, padding='same')) ····❹
autoencoder.add(Conv2D(8, kernel_size=3, strides=2, padding='same',
activation='relu')) ····❺
```

4 이미지의 특징을 추출하는 오토인코더

- 이미지의 특징을 추출하는 오토인코더

```
# 디코딩 부분
autoencoder.add(Conv2D(8, kernel_size=3, padding='same',
activation='relu')) ····❻
autoencoder.add(UpSampling2D()) ····❼
autoencoder.add(Conv2D(8, kernel_size=3, padding='same',
activation='relu')) ····❽
autoencoder.add(UpSampling2D()) ····❾
autoencoder.add(Conv2D(16, kernel_size=3, activation='relu')) ····❿
autoencoder.add(UpSampling2D()) ····⓫
autoencoder.add(Conv2D(1, kernel_size=3, padding='same',
activation='sigmoid')) ····⓬

# 전체 구조 확인
autoencoder.summary() ····⓭
```

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - ❶ ~❺ 는 입력된 값의 차원을 축소시키는 인코딩 부분이고 ❻~ ⓬ 는 다시 차원을 점차 늘려 입력 값과 똑같은 크기의 출력 값을 내보내는 디코딩 부분
  - 두 부분이 하나의 Sequential( ) 함수로 쭉 이어져 오토인코더 모델을 만듦
  - 인코딩 파트에서 입력 크기를 줄이는 방법으로 맥스 풀링을 사용(❷, ❹)
  - 반대로 디코딩 부분에서는 크기를 늘리기 위해 앞에서 배운 UpSampling을 썼음(❼, ❾, ⓫)

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 여기서 놓치지 말아야 할 것은 ❶에서 입력된 28×28 크기가 층을 지나면서 어떻게 바뀌는지 파악하는 것
  - 입력된 값은 MaxPooling 층❷ ,❹ 를 지나면서 절반씩 줄어들 것이고, Upsampling 층 ❼,❾ ⓫ 을 지나면서 두 배로 늘어남
  - 이상한 점이 하나 있음
  - 어째서 MaxPooling 층은 두 번이 나오고 Upsampling 층은 세 번이 나올까?
  - 이대로라면 처음 입력된 28×28보다 더 크게 출력되는 것은 아닐까?

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 해답은 ⑩ 에 있음
  - 잘 보면 padding 옵션이 없음
  - 크기를 유지시켜 주는 패딩 과정이 없으므로 커널이 적용되면서 크기가 줄어듦
  - 이를 다시 확인하기 위해 전체 구조를 확인해 보면(⑬) 다음과 같음

실행 결과

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 8)         1160
```

4 이미지의 특징을 추출하는 오토인코더

- 이미지의 특징을 추출하는 오토인코더

```
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 7, 7, 8)         0
_________________________________________________________________
conv2d_2 (Conv2D)               (None, 4, 4, 8)         584
_________________________________________________________________
conv2d_3 (Conv2D)               (None, 4, 4, 8)         584
_________________________________________________________________
up_sampling2d (UpSampling2D)    (None, 8, 8, 8)         0
_________________________________________________________________
conv2d_4 (Conv2D)               (None, 8, 8, 8)         584
_________________________________________________________________
up_sampling2d_1 (UpSampling2D)  (None, 16, 16, 8)       0
```

4 이미지의 특징을 추출하는 오토인코더

- 이미지의 특징을 추출하는 오토인코더

```
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 16)        1168
_________________________________________________________________
up_sampling2d_2 (UpSampling2D)  (None, 28, 28, 16)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 28, 28, 1)         145
=================================================================
```

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 전체 구조에서 ⑭ 에서 ⑮로 넘어갈 때 다른 Conv2D 층과 달리 벡터 값이 줄어들었음에 주의
  - ⑮의 Conv2D 층에는 padding이 적용되지 않았고 kernel_size=3이 설정되었으므로 3×3 커널이 훑고 지나가면서 벡터의 차원을 2만큼 줄였음
  - 마지막 층의 벡터 값이 처음 입력 값과 같은 28×28 크기가 되는 것을 확인하면 모든 준비가 된 것

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 이제 이를 사용해 오토인코더를 완성하면 다음과 같음

실습 | 오토인코더 실습하기

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D,
UpSampling2D, Flatten, Reshape

import matplotlib.pyplot as plt
import numpy as np
```

4 이미지의 특징을 추출하는 오토인코더

- 이미지의 특징을 추출하는 오토인코더

```
# MNIST 데이터셋을 불러옵니다.
(X_train, _), (X_test, _) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32') / 255
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1).astype('float32') / 255

# 생성자 모델을 만듭니다.
autoencoder = Sequential()

# 인코딩 부분입니다.
autoencoder.add(Conv2D(16, kernel_size=3, padding='same', input_shape=(28,
28,1), activation='relu'))
autoencoder.add(MaxPooling2D(pool_size=2, padding='same'))
autoencoder.add(Conv2D(8, kernel_size=3, activation='relu', padding='same'))
```

4 이미지의 특징을 추출하는 오토인코더

- 이미지의 특징을 추출하는 오토인코더

```
autoencoder.add(MaxPooling2D(pool_size=2, padding='same'))
autoencoder.add(Conv2D(8, kernel_size=3, strides=2, padding='same',
activation='relu'))

# 디코딩 부분입니다.
autoencoder.add(Conv2D(8, kernel_size=3, padding='same',
activation='relu'))
autoencoder.add(UpSampling2D())
autoencoder.add(Conv2D(8, kernel_size=3, padding='same',
activation='relu'))
autoencoder.add(UpSampling2D())
autoencoder.add(Conv2D(16, kernel_size=3, activation='relu'))
autoencoder.add(UpSampling2D())
autoencoder.add(Conv2D(1, kernel_size=3, padding='same',
activation='sigmoid'))
```

4 이미지의 특징을 추출하는 오토인코더

- 이미지의 특징을 추출하는 오토인코더

```
# 컴파일 및 학습을 하는 부분입니다.
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(X_train, X_train, epochs=50, batch_size=128, validation_
data=(X_test, X_test))

# 학습된 결과를 출력하는 부분입니다.
random_test = np.random.randint(X_test.shape[0], size=5)

# 테스트할 이미지를 랜덤으로 호출합니다.
ae_imgs = autoencoder.predict(X_test) # 앞서 만든 오토인코더 모델에 넣습니다.

plt.figure(figsize=(7,2)) # 출력 이미지의 크기를 정합니다.
```

4 이미지의 특징을 추출하는 오토인코더

- 이미지의 특징을 추출하는 오토인코더

```
for i, image_idx in enumerate(random_test):
    # 랜덤으로 뽑은 이미지를 차례로 나열합니다.
    ax = plt.subplot(2, 7, i+1)
    # 테스트할 이미지를 먼저 그대로 보여 줍니다.
    plt.imshow(X_test[image_idx].reshape(28, 28))
    ax.axis('off')
    ax = plt.subplot(2, 7, 7+i+1)
    # 오토인코딩 결과를 다음 열에 입력합니다.
    plt.imshow(ae_imgs[image_idx].reshape(28, 28))
    ax.axis('off')
plt.show()
```

4 이미지의 특징을 추출하는 오토인코더

● 이미지의 특징을 추출하는 오토인코더

실행 결과

```
Epoch 1/50
469/469 [==============================] - 4s 8ms/step - loss: 0.2041 -
val_loss: 0.1329
... (중략) ...
Epoch 50/50
469/469 [==============================] - 3s 7ms/step - loss: 0.0822 -
val_loss: 0.0813
```

▼ 그림 19-8 | 원본 이미지와 오토인코더 이미지

원본:

오토인코더:

4 이미지의 특징을 추출하는 오토인코더

- **이미지의 특징을 추출하는 오토인코더**
  - 실행 결과에서 첫 번째 줄이 테스트로 사용된 원본 이미지, 두 번째 줄이 원본의 특징을 고스란히 담은 채 만들어진 오토인코더 이미지