Ch11_심층 신경망 훈련하기

▼ ref

심층신경망

수백 개의 뉴런으로 구성된 **10개 이상의 층**을 사용하는 **신경망**

ex. 고해상도 이미지에서 수백 종류의 물체 감지

- 심층 신경망을 훈련할 때 발생하는 문제
  - **심층 신경망의 아래쪽으로 갈수록 그레이디언트가 점점 더 작아지는 그레디언트 소실 또는 커지는 그레디언트 폭주 현상**
  - 대규모 신경망을 위한 훈련 데이터가 충분하지 않거나 레이블을 만드는 작업에 비용이 너무 많이 들 수 있다.
  - 훈련이 극단적으로 느려질 수 있다.
  - 수백만 개의 파라미터를 가진 모델은 훈련 세트에 **과대 적합**될 위험이 매우 크다.

▼ ***11.1 그레이디언트 소실((vanishing gradient)과 폭주 문제***

역전파 알고리즘은 **출력층에서 입력층으로 오차 그레이디언트를 전파**하면서 진행된다. 알고리즘이 신경망의 모든 **파라미터에 대한 오차 함수의 그레이디언트를 계산**하면, 경사 하강법 단계에서 이 **그레이디언트를 사용하여 각 파라미터를 수정**한다.

그레이디언트 소실

알고리즘이 하위층으로 진행될수록 **그레이디언트가 점점 작아지는 경우**를 말한다. 경사 하강법이 하위층의 연결 가중치를 **변경되지 않은 채로 두면 훈련이 좋은 솔루션으로 수렴하지 않게** 된다.

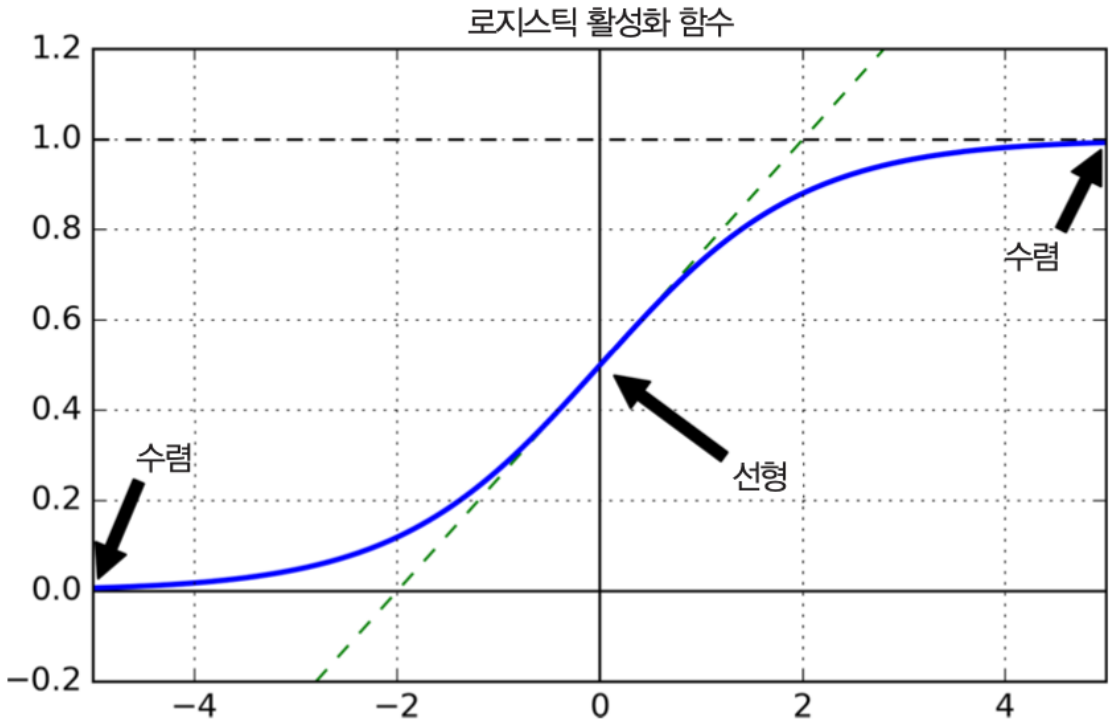
그레이디언트 폭주

심층 신경망의 아래쪽으로 갈수록 **그레이디언트가 점점 더 커지는 현상**이다. 이로 인해 여러 층이 비정상적으로 **큰 가중치로 갱신**되면 **알고리즘은 발산**하게 된다. 주로 **순환 신경망**에서 나타난다.

두 현상 모두 하위층을 훈련하기 매우 어렵게 만든다.

- **원인**

로지스틱 활성화 함수는 **입력이 커지면 활성화 함수가 아닌 0이나 1로 수렴**해서 **기울기가 0**에 매우 가까워 진다. 그래서 **역전파가 될 때 사실상 신경망으로 전파할 그레이디언트가 거의 없고**, 조금 있는 그레이디언트는 최상위층에서부터 역전파가 진행되면서 점차 약해져서 **실제로 아래쪽 층에는 아무것도 도달하지 않게 된다.**



▼ 11.1.1 글로럿과 He 초기화

- **불안정한 그레이디언트를 완화하는 방법**

예측을 할 때는 **정방향**으로, **그레이디언트를 역전파할 때는 역방향**으로 양방향 신호가 적절하게 흘러야 한다. 신호가 죽거나 폭주, 소멸하지 않아야 한다.

신호가 흐르기 위해서는

(1) **각 층의 출력에 대한 분산이 입력에 대한 분산과 같아야 한다**

(2) **역방향에서 층을 통과하기 전과 후의 그레이디언트 분산이 동일**해야 한다.

이때, **fan-in과 fan-out이 같지 않으면** 위의 2가지 내용을 **보장할 수 없기 때문**에 층에 사용되는 활성화 함수의 종류에 따라 세 가지 **방식 중 하나**를 선택하여 **초기화** 한다.

- **fan-in**: 층의 **입력 연결** 개수

- **fan-out**: 층의 **출력 연결** 개수

초기화 전략	활성화 함수	σ^2 (정규분포)
글로럿	활성화 함수 없음, 하이퍼볼릭 탄젠트, 로지스틱, 소프트맥스	$1 / fan_{\text{avg}}$
He	ReLU 함수와 그 변종들	$2 / fan_{\text{in}}$
르쿤	SELU	$1 / fan_{\text{in}}$

1)글로럿(Glorot) 초기화

평균이 0이고 분산이 $\sigma^2 = \frac{1}{fan_{avg}}$인 정규분포
또는 $r = \sqrt{\frac{3}{fan_{avg}}}$일 때 $-r$과 $+r$ 사이의 균등분포

- 훈련 속도를 높일 수 있다.

- S자 형태의 활성화 함수(시그모이드, 하이퍼볼릭탄젠트 등)와 함께 사용할 경우 좋은 성능을 보이지만, ReLU와 함께 사용할 때는 좋지 않다.

2) 르쿤(LeCun) 초기화 :

평균이 0이고 분산이 $\sigma^2 = \frac{1}{fan_{in}}$인 정규분포
또는 $r = \sqrt{\frac{3}{fan_{in}}}$일 때 $-r$과 $+r$ 사이의 균등분포

※ $fan_{in} = fan_{out}$이면 글로럿 초기화와 동일

- SELU 활성화 함수에 대한 초기화 방법

3) 헤(He) 초기화 :

ReLU 활성화 함수를 사용하는 초기화 방법이다.
평균이 0이고 분산이 $\sigma^2 = \frac{2}{fan_{in}}$인 정규분포

- 케라스는 기본적으로 균등분포의 글로럿 초기화를 사용한다.

다음과 같이 층을 만들 때 `kernel_initializer="he_uniform"` 나 `kernel_initializer="he_normal"`로 바꾸어 He 초기화를 사용할 수 있다.

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

`Variance Scaling` : fan_{in}대신 fan_{out} 기반의 균등 분포 He 초기화를 사용할 때 사용

```
he_avg_init = VarianceScaling(scale=2., mode='fan_avg',
                              distribution='uniform')
Dense(10, activation='sigmoid', kernel_initializer='he_avg_init')
```

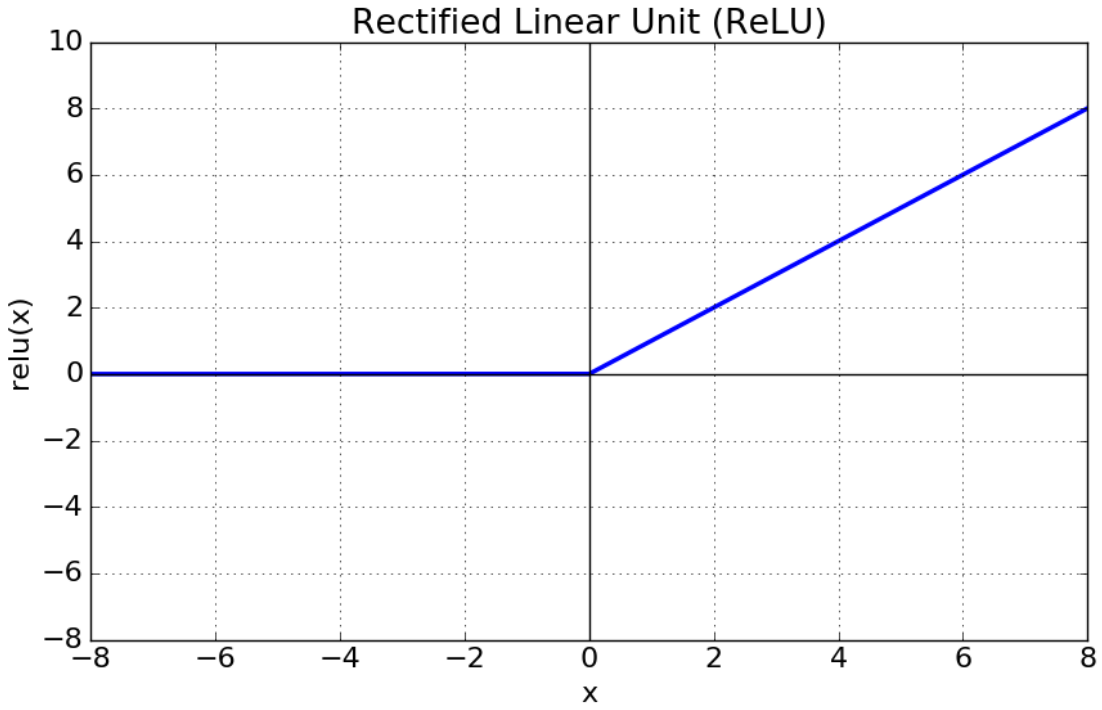
▼ 11.1.2 수렴하지 않는 활성화 함수

심층 신경망에서는 시그모이드 활성화 함수 아닌 **다른 활성화 함수**가 훨씬 더 잘 작동한다.

ReLU 함수는 특정 양수값에 수렴하지 않고, 계산이 빠르다는 큰 장점이 있지만 **dying ReLU의 문제**가 있다.

dying ReLU 문제

훈련하는 동안 **일부 뉴런이 0 이외의 값을 출력하지 않는 문제**이다. 뉴런의 가중치가 바뀌면서 훈련 세트에 있는 모든 샘플에 대해 입력의 가중치 합이 음수가 되면 ReLU 함수의 그레이디언트가 0이 되므로 경사 하강법이 더는 작동하지 않는 문제가 발생한다.



LeakyReLU 함수

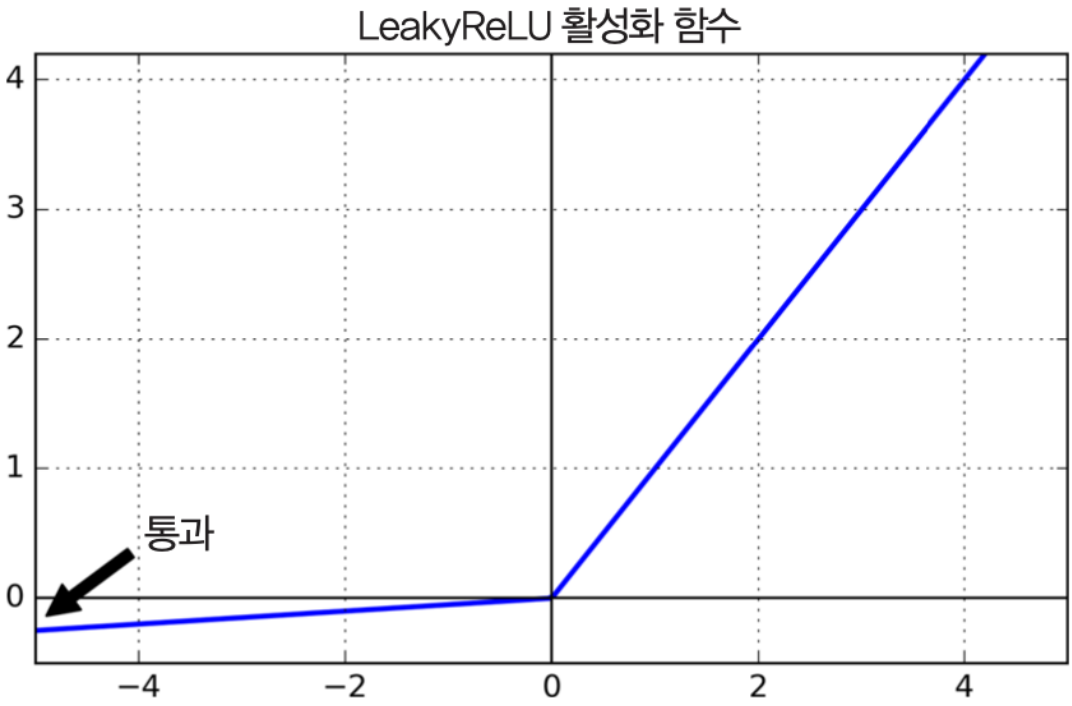
dying ReLU 문제를 해결하기 위해 사용하는 함수

→ $LeakyReLU_{\alpha}(z) = max(\alpha z, z)$

하이퍼파라미터 α가 이 함수의 **새는 정도**를 결정한다.

새는 정도란, $z < 0$ **일 때 이 함수의 기울기**이며, 일반적으로 0.01로 설정한다. 이 값을 **작게 하는 것**이 성능이 더 좋게 나타난다.

- LeakyReLU가 ReLU보다 항상 좋은 성능 발휘 (default = α=0.1)
- α=0.2로 할 때 default = α=0.1보다 좀 더 성능이 좋아짐



- **RReLU (randomized leaky ReLU) 함수**

- 훈련하는 동안 주어진 범위에서 α를 무작위로 선택하고 테스트시에는 평균을 사용하는 방법이다. 과대적합을 줄이는 규제역할도 수행햔다.

- **PReLU (parametric leaky ReLU) 함수**

- α가 훈련하는 동안 학습되는 것으로, 하이퍼파라미터가 아니라 역전파에 의해 변경되는 파라미터다.

- 대규모 이미지 데이터셋에서 ReLU보다 성능이 좋다.

- 소규모 데이터셋에서는 과대적합 위험성 존재

- 앞서 언급된 ReLU 변종들보다 훈련 시간 줄어든다.

- **ELU 활성화 함수**

식 11-2 ELU 활성화 함수

$$\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z)-1) & z < 0\text{일 때} \\ z & z \geq 0\text{일 때} \end{cases}$$

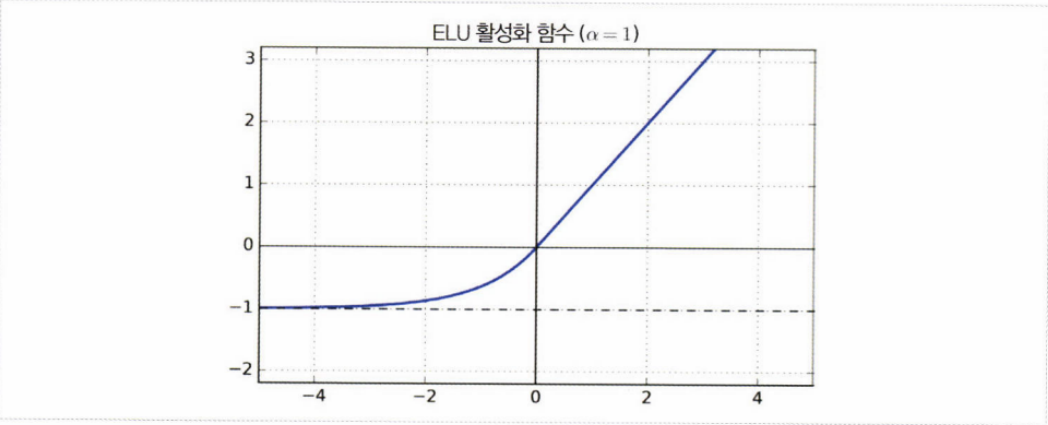


그림 11-3 ELU 활성화 함수

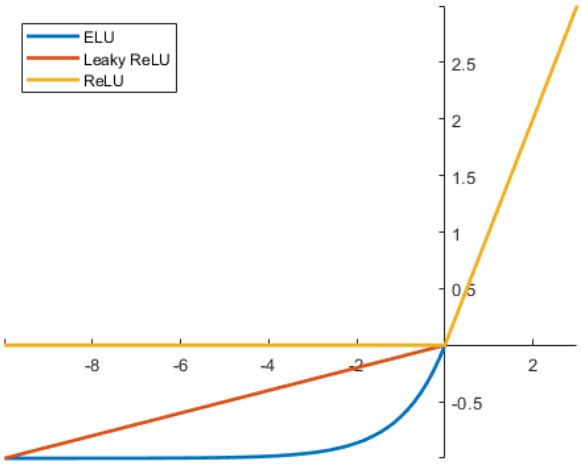
하이퍼파라미터 α는 z가 큰 음수값일 때 ELU가 수렴할 값을 정의한다.

- α : z가 큰 음숫값일 때 ELU가 수렴할 값을 정의하며, 보통 1로 설정한다.

- α=1이면 이 함수는 z=0에서 급격히 변동하지 않으므로 z=0을 포함해 모든 구간에서 매끄러워 경사 하강법의 속도를 높인다.

- z<0 일 때 활성화 함수의 평균 출력이 0에 더 가까워져 그레이디언트 소실 문제를 완화해준다. 또한 죽은 뉴런을 만들지 않는다.

ELU 활성화 함수는 수렴 속도가 빠르지만, 지수 함수를 사용하므로 ReLU나 그 변종들보다 계산이 느리다.



- **SELU (Scaled ELU) 함수**

스케일이 조정된 ELU 활성화 함수의 변종이다. 완전 연결 층만 쌓아서 신경망을 만들고 모든 은닉층이 SELU 활성화 함수를 사용하여 네트워크가 **자기 정규화(self-normalized)** 된다. 훈련하는 동안 각 층의 출력이 **평균 0과 표준편차 1을 유지**하는 경향이 있어 그**래디언트 소실과 폭주 문제를 막아준다**.

다른 활성화 함수보다 뛰어난 성능을 종종 보이지만 **자기 정규화가 일어나기 위한 몇 가지 조건**이 존재한다.

1) 입력 특성이 반드시 표준화(평균 0, 표준편차 1)되어야 한다.

2) 모든 은닉층의 가중치는 르쿤 정규분포 초기화로 초기화되어야 한다. (케라스에서는 `kernel_initializer='lecun_normal'`로 설정)

3) 네트워크는 일렬로 쌓은 층으로 구성되어야 한다.

4) 모든 층이 완전 연결층이어야 한다.

활성화 함수 쓰는 방법

- 일반적으로 SELU > ELU > LeakyReLU(그리고 변종들) > ReLU > tanh > sigmoid 순이다.
- 네트워크가 자기 정규화되지 못하는 구조라면 SELU 보단 `ELU`
- 실행 속도가 중요하다면 `LeakyReLU`(하이퍼파라미터를 더 추가하고 싶지 않다면 케라스에서 사용하는 기본값 α 사용)
- 시간과 컴퓨팅 파워가 충분하다면 교차 검증을 사용해 여러 활성화 함수를 평가
- 신경망이 과대적합되었다면 `RReLU`
- 훈련세트가 아주 크다면 `PReLU`
- ReLU가 가장 널리 사용되는 활성화 함수이므로 많은 라이브러리와 하드웨어 가속기들이 ReLU에 특화되어 최적화. 따라서 속도가 중요하다면 `ReLU`가 가장 좋은 선택

- **`LeakyReLu` 활성화 함수 사용하기**

: LeakyReLU 층을 만들고 모델에서 적용하려는 층 뒤에 추가

```
model = Sequential([
    [...]
    Dense(10, kernel_initializer="he_normal"),
    LeakyReLU(alpha=0.2)
    [...]
])
```

- PReLU 사용하려면 `LeakyReLU(alpha=0.2)` 를 `PReLU()` 로 바꾸기
- SELU 활성화 함수를 사용하려면 층을 들 때 `activation="selu"` 와 `kernel _initializer "lecun_normal"` 로 지정한다.

```
model = Sequential()
model.add(Flatten(input_shape=[28, 28]))
model.add(Dense(300, activation='selu', kernel_initializer='lecun_normal'))

for layer in range(99):
    model.add(Dense(100, activation='selu', kernel_initializer='lecun_normal'))
model.add(Dense(10, activation='softmax'))
```

- **ReLU 활성화 함수 사용**

  PReLU층을 만들고 모델에서 적용하려는 층 뒤에 추가

```
model = Sequential([
    Flatten(input_shape=[28, 28]),
    Dense(300, kernel_initializer='he_normal'),
    PReLU(),
    Dense(100, kernel_initializer='he_normal'),
    PReLU(),
    Dense(10, activation='softmax')
])
```

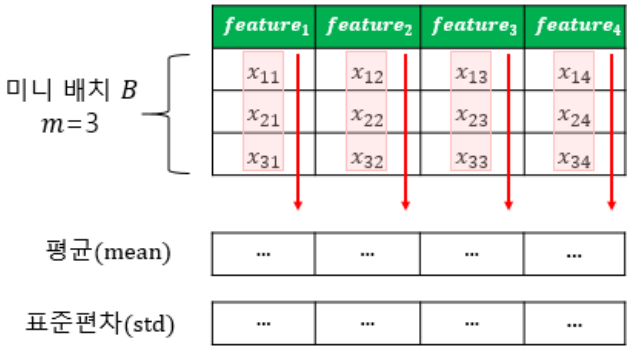
▼ 11.1.3 배치 정규화

- ELU(혹은 다른 ReLU 변종)와 함께 He 초기화를 사용하면 훈련 초기단계의 소실/폭주 문제를 해결할 수 있지만, 훈련 중 **동일 문제 재발생 방지를 보장하지 못한다.**

> **배치 정규화** (BN, batch normalization)

그레이디언트 소실과 폭주 문제를 해결하기 위해 등장한 기법으로, **활성화 함수를 통과하기 전이나 후에 모델에 연산을 하나 추가하는 것**이다. 단순하게 **입력을 원점에 맞추고 정규화**한 다음, 각 층에서 **2개의 새로운 파라미터로 결과값의 스케일을 조정하고 이동**시킨다.

- 평균: 0으로 조정
- 분산: 스케일 조정



배치 정규화를 위해서 알고리즘은 **미니배치에서 입력의 평균과 표준편차를 추정**한다.

식 11-3 배치 정규화 알고리즘

$$1.\ \boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$$

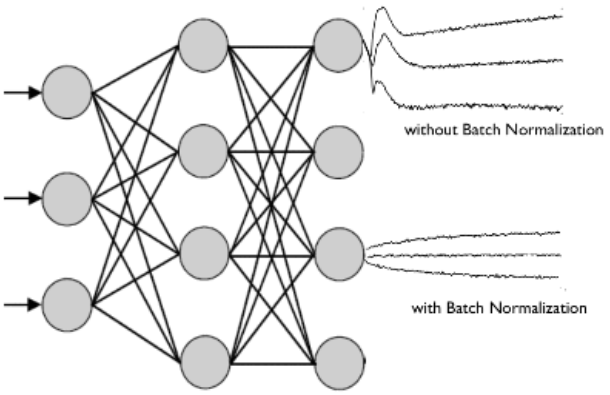
$$2.\ \boldsymbol{\sigma}_B^{\,2} = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

$$3.\ \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^{\,2} + \varepsilon}}$$

$$4.\ \mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

μ_B: 미니배치 B에 대해 평가한 입력의 평균 벡터
σ_B: 미니배치에 대한 평가한 입력의 표준편차 벡터
μ_B: 미니배치에 있는 샘플 수
$\hat{x}^{(i)}$: 평균이 0이고 정규화된 샘플 i의 입력
γ: 층의 출력 스케일 파라미터 벡터(표준편차와 비슷한 개념)
$\otimes$: 원소별 곱셈(element-wise multiplication)
β: 층의 출력 이동 파라미터 벡터(평균 이동)
ε: 분모가 0이 되는 것을 막기 위한 작은 숫자(보통 10^{-5}), 안전을 위한 항(smoothing term)이라 부름
$z^{(i)}$: 배치 정규화 연산의 출력, 즉 입력 스케일을 조정하고 이동시킨 결과

훈련하는 동안 배치 정규화는 입력을 정규화한 다음 스케일을 조정하고 이동시킨다. 하지만 테스트 시에는 샘플 하나에 대한 예측을 만들어야 하는데, 이런 경우 iid 조건을 만족하지 못하고 신뢰도가 떨어진다. 따라서 **훈련이 끝난 후 전체 훈련 세트를 신경망에 통과**시켜 **배치 정규화 층의 각 입력에 대한 평균과 표준편차를 계산**한다.



대부분 배치 정규화 구현은 층의 입력 평균과 표준편차의 이동 평균을 사용해 훈련하는 동안 최종 통계를 추정한다. 케라스의 `BatchNormalization` 층은 이를 자동으로 수행한다.

- 정리

배치 정규화 층마다 네 개의 파라미터 벡터가 학습된다.

- **γ**(출력 스케일 벡터)와 **β** (출력 이동 벡터)는 일반적인 **역전파**를 통해 학습된다.
- **μ**(최종 입력 평균 벡터)와 **σ** (최종 입력 표준편차 벡터)는 **지수 이동 평균**을 사용하여 추정된다.
- **μ**와 **σ**는 훈련하는 동안 추정되지만 훈련이 끝난 후에 사용된다.

배치 정규화는 규제와 같은 역할을 하여 다른 규제 기법의 필요성을 줄여준다. **그레이디언트 소실/폭주 문제를 감소**시켜서 **하이퍼볼릭 탄젠트 또는 로지스틱 활성화 함수 사용 가능**하고 **가중치 초기화에 덜 민감**해진다. 그러나 배치 정규화를 사용할 때 에포크마다 더 많은 시간이 걸리므로 **훈련이 오히려 느려질 수 있다.**

케라스로 배치 정규화 구현하기

은닉층의 활성화 함수 전이나 후에 `BatchNormalization` 층을 추가하면 된다.

```
model = Sequential([
    Flatten(input_shape=[28, 28]),
    BatchNormalization(),
    Dense(300, activation="elu", kernel_initializer="he_normal"),
    BatchNormalization(),
    Dense(100, activation="elu", kernel_initializer="he_normal"),
    BatchNormalization(),
    Dense(10, activation="softmax")
])

mode1.summary ()
```

```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

→ 배치 정규화 층은 입력마다 네 개의 파라미터 γ,β,μ,σ를 추가한다. (784 x 4 = 3,316개의 파라미터)

→ 마지막 두 개의 파라미터 μ,σ는 이동 평균이다. 이 파라미터는 역전파로 학습되지 않기 때문에 케라스는 `'Non-trainable'` 파라미터로 분류한다. 첫 번째 배치 정규화 층의 파라미터를 살펴보면, 두 개는 역전파로 훈련되고 두개는 훈련되지 않다.

- **활성화 함수 이전에 정규화 층 추가 방법**

은닉층에서 활성화 함수를 지정하지 않고 활성화 함수를 정규화 층 뒤에 별도의 층으로 추가한다.

또한 배치 정규화 층은 입력마다 이동 파라미터를 포함하기 때문에 이전 층에서 편향을 뺄 수 있다.
(`use_bias=False`)

```
model = Sequential([
    Flatten(input_shape=[28, 28]),
    BatchNormalization(),
    Dense(300, kernel_initializer='he_normal', use_bias=False),
```

```
    BatchNormalization(),
    Activation('elu'),
    Dense(100, kernel_initializer='he_normal', use_bias=False),
    BatchNormalization(),
    Activation('elu'),
    Dense(10, activation='softmax')
])
```

- `BatchNormalization` 클래스는 조정할 하이퍼파라미터가 적지만 가끔 momentum 매개변수를 변경해야 할 수 있다.

- `BatchNormalization` 층이 지수 이동 평균을 업데이트할 때 이 하이퍼파라미터를 사용한다.

$$\hat{\mathbf{v}} \leftarrow \hat{\mathbf{v}} \times \text{momentum} + \mathbf{v} \times (1 - \text{momentum})$$

- 새로운 값 v가 주어지면 다음 식을 사용해 이동 평균 v_hat를 업데이트한다. 적절한 모멘텀 값은 일반적으로 1에 가깝다.

- 하이퍼파라미터 axis

: 이 매개변수는 정규화할 축을 결정한다. 기본값은 1이다. 즉 다른 축을 따라 계산한 평균과 표준편차를 사용하여 마지막 축을 정규화한다. 입력 배치가 2D(즉, 배치 크기가 [샘플개수 특성 개수])이면 각 입력 특성이 배치에 있는 모든 샘플에 대해 계산한 평균과 표준편치를 기반으로 정규화된다.

- 배치 정규화 층은 훈련 도중과 훈련이 끝난 후에 수행하는 계산이 다르다. 훈련하는 동안 배치 통계를 사용하고 훈련이 끝난 후에는 **"최종"** 통계(이동 평균의 마지막 값)를 사용한다. 배치 정규화는 심층 신경망에서 매우 널리 사용하는 층이 되었고, 보통 모든 층 뒤에 배치 정규화가 있다고 가정한다.

▼ 11.1 .4 그레이디언트 클리핑

> ## 그레이디언트 클리핑 (gradient clipping)

: 그레이디언트 폭주 문제를 완화하는 방법으로 **역전파될 때 일정 임곗값을 넘어서지 못하게 그레이디언트를 잘라내는 것**이다. 이때 **임계값은 그레디언트가 가질 수 있는 최대 L2 norm을 의미**하고 하이퍼 파라미터이다. 배치 정규화를 적용하기 어려운 순환 신경망에서 유용하다.

- 케라스에서 그레이디언트 클리핑을 구현하려면 옵티마이저를 만들 때 `clipvalue`와 `clipnorm` 매개변수를 지정하면 된다.

```
optimizer = tensorflow.keras.SGD(clipvalue=1.0) # 손실의 모든 편미분 값을 -1.0에서 1.0으로 잘라낸다.
model.compile(loss='mse', optimizer=optimizer)
```

이 옵티마이저는 그레이디언트 벡터의 모든 원소를 과 [-1.0 ~ 1.0] 사이로 클리핑한다. 즉 훈련되는 각 파라미터에 대한 손실의 모든 편미분 값을 -1.0에서 1.0으로 잘라낸다. 임계값은 이 하이퍼파라미터로 튜닝할 수 있다.

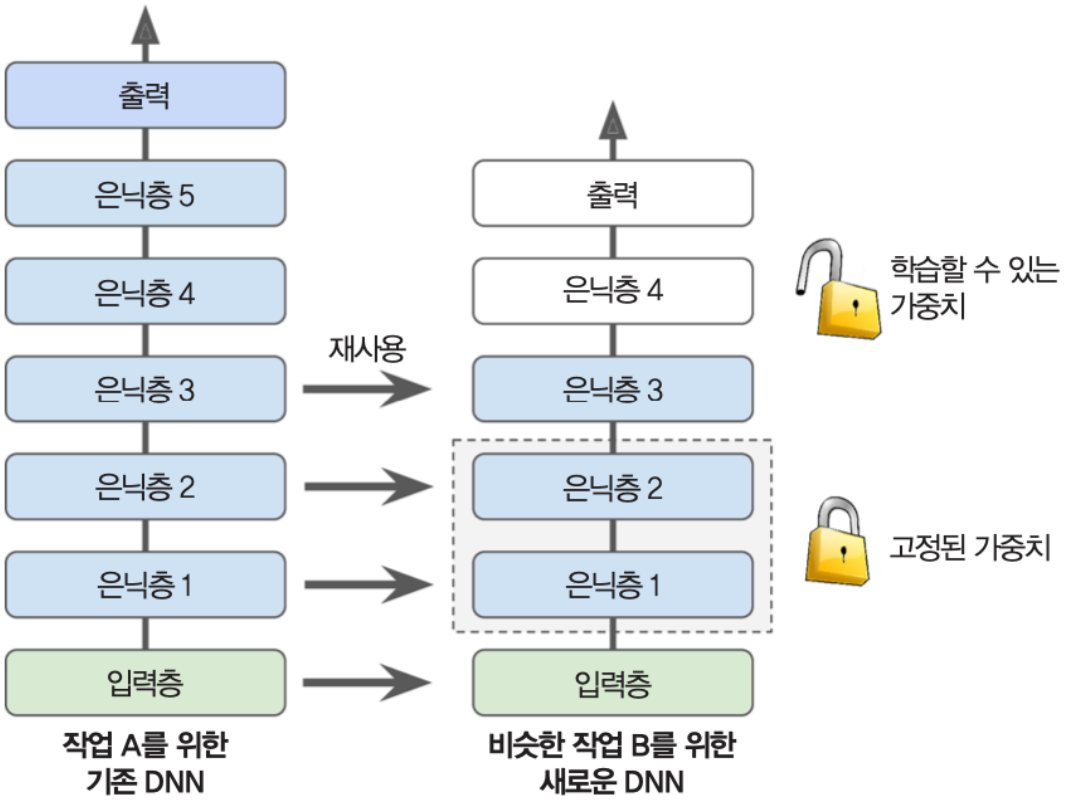
이 기능은 그레이디언트 벡터의 방향을 바꿀 수 있다.

ex. 본 그레이디언트 벡터가 [0.9, 100.0]라면 대부분 2번째 축 방향을 향한다. 이를 클리핑하면 [0.9, 1.0]이 되고 거의 두 축 사이 대각선 방향을 향한다.

▼ 11.2 사전 훈련된 층 재사용하기

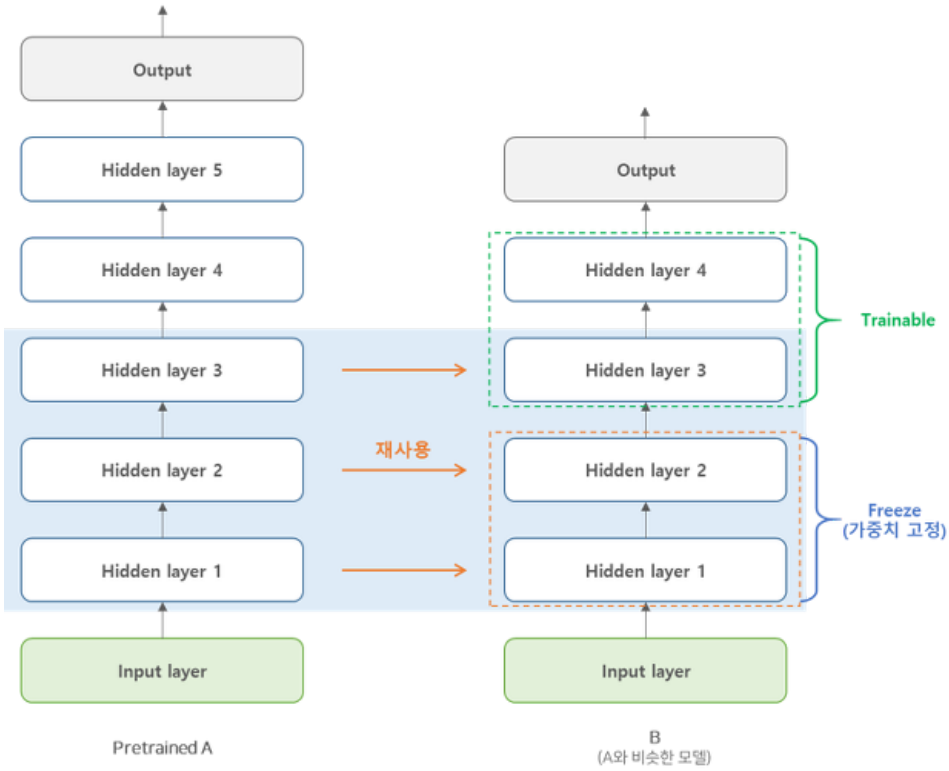
> **전이 학습** (Transfer Learning)

특정 분야에서 학습된 신경망의 일부 능력을 유사하거나 전혀 새로운 분야에서 사용되는 신경망의 학습에 이용하는 것을 의미한다. 이 방법은 훈련 속도를 크게 높일 뿐만 아니라 필요한 훈련 데이터도 크게 줄여 준다.



전이 학습 시 **원본 모델의 출력층은 새로운 작업에서 가장 유용하지 않은 층**이므로 바꿔주어야 하는데, 이때 **비슷한 원본 모델의 하위 은닉층**이 훨씬 유용하다. 그리고 원본 모델의 상위 은닉층은 하위 은닉층보다 덜 유용하다. 따라서 **재사용할 층의 개수를 잘 선정**해야 한다.

- **과정**



(1) 재사용하는 층을 모두 동결하고 훈련 및 성능 평가한다.

동결한다는 의미는 경사 하강법으로 가중치가 바뀌지 않도록 훈련되지 않는 가중치로 만드는 것이다.

(2) 맨 위에 있는 한 두 개 은닉층의 동결을 해제하고 역전파를 통해 가중치를 조정한다.

(3) 성능이 좋아지지 않거나 훈련 데이터가 적을 경우 상위 은닉층 제거 후 남은 은닉층 동결하고 다시 훈련한다.

(4) 훈련 데이터가 많을 경우 은닉층을 제거하기보다는 다른 것으로 바꾸거나 더 많은 은닉층을 추가한다.

(5) 위 과정을 반복 한다.

▼ 11.2.1 케라스를 사용한 전이 학습

8개 클래스만 담겨 있는 패션 MNIST 데이터셋이 있다고 가정하자. 모델 A는 8개 클래스 모두를 분류하는 것이고, 모델 B는 샌들과 셔츠만 분류하는 이진 분류기다. 모델 B는 모델 A와 매우 비슷하여 전이 학습을 진행해보려고 한다.

- 전이학습을 이용한 모델(model_B_on_A) 훈련

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

model_A와 model_B_on_A는 일부 층을 공유하므로 model_B_on_A를 훈련할 때 model_A도 영향을 받는다. 이를 원치 않는다면 층을 재사용하기 전에 model_A를 클론해야 한다.

```
from tensorflow.keras.models import clone_model

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

`clone_model()` 메서드로 모델 A의 구조를 복제한 후 가중치를 복사한다. (`clone_model()` 메서드는 가중치를 복제하지 않는다.)

이제 작업 B를 위해 model_B_on_A을 훈련할 수 있다.

새로운 출력층이 랜덤하게 초기화되어 있으므로 큰 오차를 만들 것이다. 이를 피하는 한 가지 방법은 **처음 몇 번의 에포크 동안 재사용된 층을 동결하고 새로운 층에게 적절한 가중치를 학습할 시간을 주는 것이다.**

훈련이이를 위해 모든 층의 `trainable` 속성을 False로 지정하고 모델을 컴파일한다.

```
# 층을 동결하거나 동결을 해제한 후 반드시 모델을 컴파일해야 한다.
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                     metrics=["accuracy"])
```

이제 몇 번의 에포크 동안 모델을 훈련할 수 있다. 그다음 재사용된 층의 동결을 해제하고 작업 B에 맞게 재사용된 층을 세밀하게 튜닝하기 위해 훈련을 계속한다.일반적으로 재사용된 층의 동결을 해제한 후에 학습률을 낮추는 것이 좋다.

다음의 코드는 재사용된 가중치가 망가지는 것을 막아준다.

```
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 = SGD(lr=1e-4) # 기본 학습률은 le-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))
```

얕은 신경망 모델에서는 전이학습 성능이 좋지 않다.

- 전이 학습은 조금 더 일반적인 특성을 감지하는 경향이 있는 심층 합성곱 신경망에서 잘 동작한다.

▼ 11.2.2 비지도 사전훈련

> **비지도 사전 훈련**

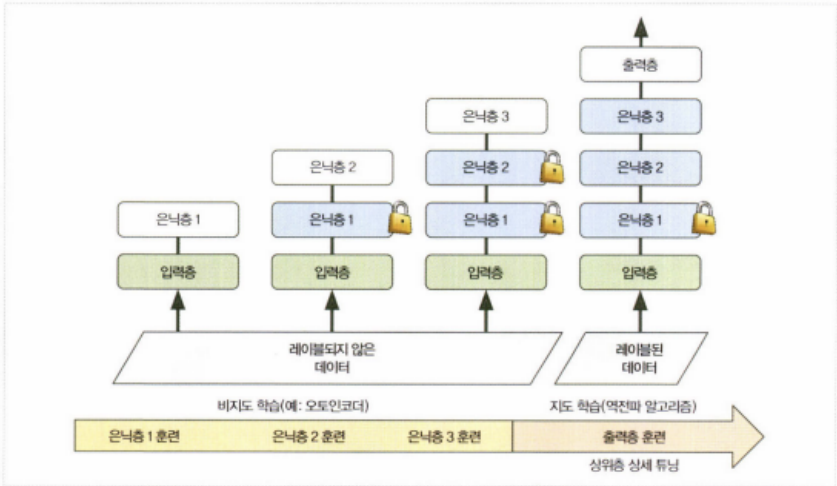


그림 11-5 비지도 훈련에서는 비지도 학습 기법으로 레이블이 없는 데이터(또는 전체 데이터)로 모델을 훈련합니다. 그 다음 지도 학습 기법을 사용하여 레이블된 데이터에서 최종 학습을 위해 세밀하게 튜닝합니다. 비지도 학습 부분은 그림처럼 한 번에 하나의 층씩 훈련하거나 바로 전체 모델을 훈련할 수도 있습니다.

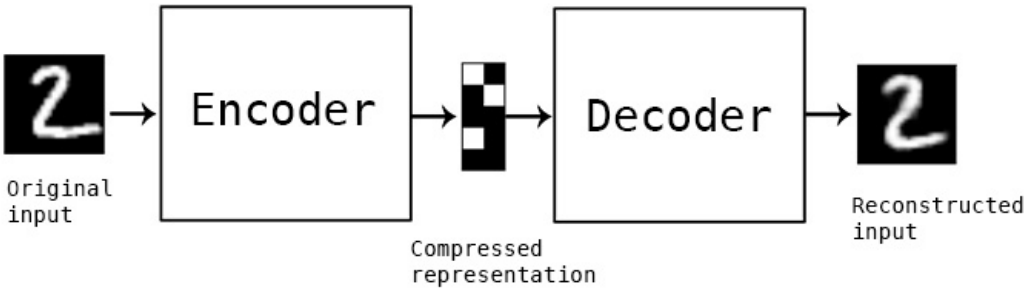
풀어야 할 문제가 복잡하고 재사용할 수 있는 비슷한 모델이 없으며 **레이블된 훈련 데이터가 적고** 그렇지 않은 데이터가 많을 때 사용한다.

레이블되지 않은 훈련 데이터를 사용하여 **오토인코더나 생성적 적대 신경망과 같은 비지도 학습 모델**을 훈련할 수 있다. 오토인코더나 GAN 판별자의 **하위층을 재사용**하고, 그 **위에 새로운 작업에 맞는 출력층을 추가**할 수 있다. **그 후 지도 학습으로 최종 네트워크를 세밀하게 튜닝**한다.

1. 비지도 학습 기법으로 레이블이 없는 데이터 또는 전체 데이터로 모델을 학습.
2. 지도 학습 기법읊 사용하여 레이블된 데이터에서 최종 학습을 위해 튜닝

- **Autoencoder**

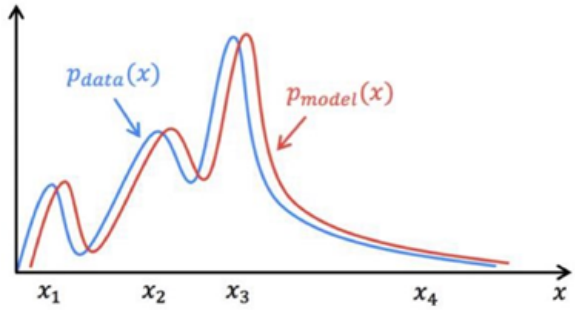
: 입**력 데이터를 작은 단위의 대푯값들만 남겨 압축시킨 후 다시 확장시켜 출력 데이터를 입력 데이터와 동일하게 하는 뉴럴 네트워크 모델이다**. 입력 데이터를 정답 데이터로 사용하기 때문에 self-supervised learning이라고도 부르고 **정답이 없기 때문에 unsupervised learning 방법**이라고도 한다. **최소한의 차원만 가지고 특징을 표현할 수 있다는 것이 포인트로 입력 데이터의 의미 있는 속성들을 추출한다.**



- **GAN**

Generator과 Discriminator이 서로 대립(Adversarial)하여 **성능을 개선**해나가는 방식이다. **가짜 데이터를 실제 데이터처럼 만들어내는 Generator**, **가짜 데이터와 실제 데이터를 판별하는 Discriminator** 두 부분으로 나눠져 있다.

ex. 색이 없는 사진에 색을 더해주거나 진짜 같은 가짜 데이터를 만들어내는 방법으로 실제 사진을 변형해서 만들어내는 방법



GAN 모델의 **목표**는 **실제 데이터의 분포에 근사하는 모델을 만드는 것**으로 **실제 데이터에 대한 확률분포(확률 밀도 함수)와 비슷하게 따라가도록 학습**한다. 그래서 파란선의 실제 데이터의 확률 분포와 빨간 선의 모델에서 나온 확률 분포의 차이가 적도록 만들어야 한다.

▼ 11.2.3 보조 작업에서 사전훈련

레이블된 훈련 데이터가 많지 않다면 레이블된 훈련 데이터를 쉽게 얻거나 생성할 수 있는 보조 작업에서 **첫 번째 신경망**을 훈련하는 것이다. 이 신경망의 하위층을 실제 작업을 위해 재사용한다. 첫 번째 신경망의 하위층은 두 번째 신경망에 재사용될 수 있는 특성 추출기를 학습하게 된다.

▼ 11.3 고속 옵티마이저

지금까지는 훈련 속도를 높이는 네 가지 방법을 살펴보았다. (연결 가중치에 좋은 초기화 전략 적용하기, 좋은 활성화 함수 사용하기, 배치 정규화 사용하기, 사전훈련된 네트워크의 일부 재사용하기)

훈련 속도를 크게 높일 수 있는 또 다른 방법으로 표준적인 경사 하강법, 옵티마이저 대신 더 빠른 옵티마이저를 사용할 수 있다.

→ 모멘텀 최적화 / 네스테로프(Nesterov) 가속 경사(NAG) / AdaGrad / RMSProp / Adam 최적화 / Nadam 최적화

▼ 11.3.1 모멘텀 최적화

표준적인 경사 하강법은 경사면을 따라 일정한 크기의 스텝으로 조금씩 내려 간다.

- 경사 하강법의 공식 : $\theta \leftarrow \theta - \eta \nabla \theta J(\theta)$
- θ : 가중치, η : 학습률, J(θ) : 가중치에 대한 비용 함수
- 이 식은 이전 그레이디언트가 얼마였는지 고려하지 않는다.

모멘텀 최적화

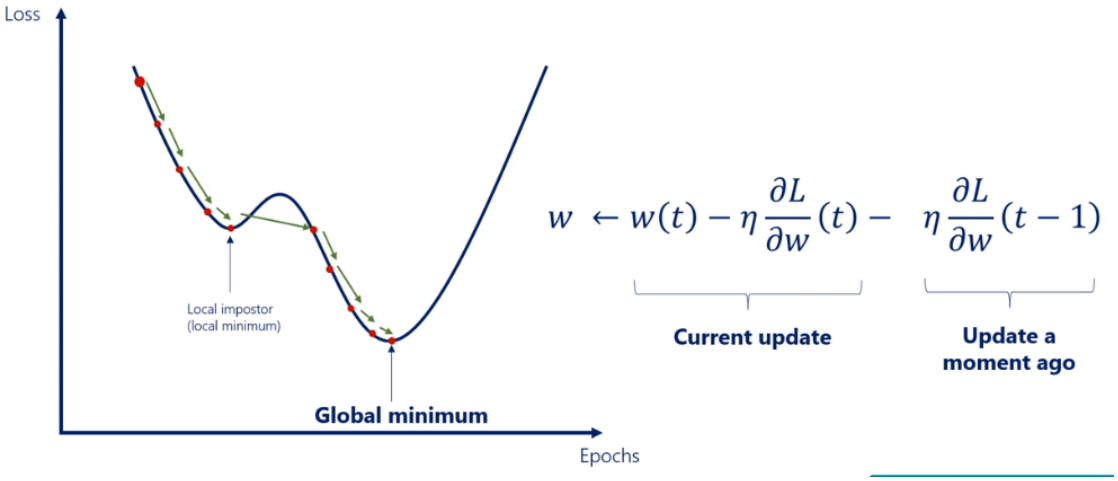
모멘텀은 **가중치 조정 과정을 추적**하면서 **변화 가속도를 조절**한다.

식 11-4 모멘텀 알고리즘[25]

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta + \mathbf{m}$

$d\theta J_{(\theta)}$: 음수
m: 음수
위에서 모멘텀 벡터 m에 더하고 이 값을 빼는 방식으로 가중치를 갱신한다는 의미가 헷갈릴 수 있는데, 부호를 생각하면 1번 식은 결국 더하는 것이고, 2번 식은 결국 빼는 것을 의미

매 반복에서 **현재 그레이디언트를 학습률 η를 곱한 후 모멘텀 벡터 *m*에 더하고 이 값을 빼는 방식으로 가중치를 갱신**한다. **그레이디언트를 속도가 아닌 가속도로 사용**하므로 모멘텀이 너무 커지는 것을 막기 위해 하이퍼파라미터 β를 추가한다. β는 0 (높은 마찰저항)과 1 (마찰저항 없음) 사이의 값으로 설정된다. (default : 0.9)



→ 모멘텀 최적화는 **골짜기를 따라 바닥(최적점)에 도달할 때까지 점점 더 빠르게** 내려간다.

→ 모멘텀 때문에 옵티마이저가 최적값에 안정되기 전까지 건너뛰었다가 다시 돌아오고, 다시 또 건너뛰는 식으로 여러 번 왔다 갔다 할 수 있다.

→ 지**역 최적점을 건너뛰도록** 하는 데도 도움이 된다.

- 케라스에서 모멘텀 최적화를 구현

`SGD` 옵티마이저를 사용하고 `momentum` 매개변수를 지정하고 기다리면 된다.

```
optimizer = SGD(lr=0.001, momentum=0.9)
```

- 모멘텀 최적화의 한 가지 단점은 튜닝할 하이퍼파라미터가 하나 늘어난다는 것이다. 그러나 실제로 모멘텀 0.9 에서 보통 잘 작동하며 경사 하강법보다 거의 항상 더 빠르다.

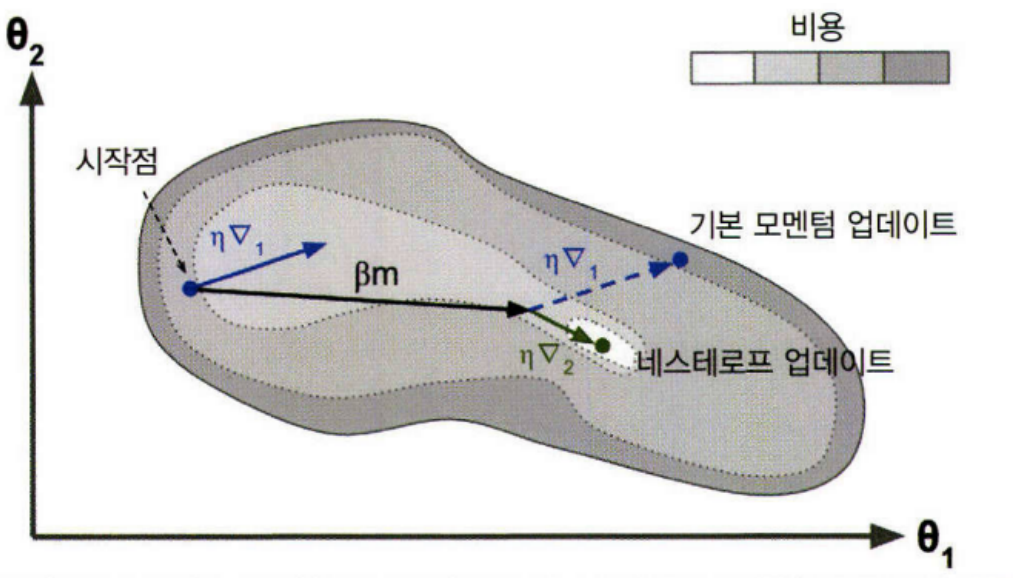
▼ 11.3.2 네스테로프 가속 경사

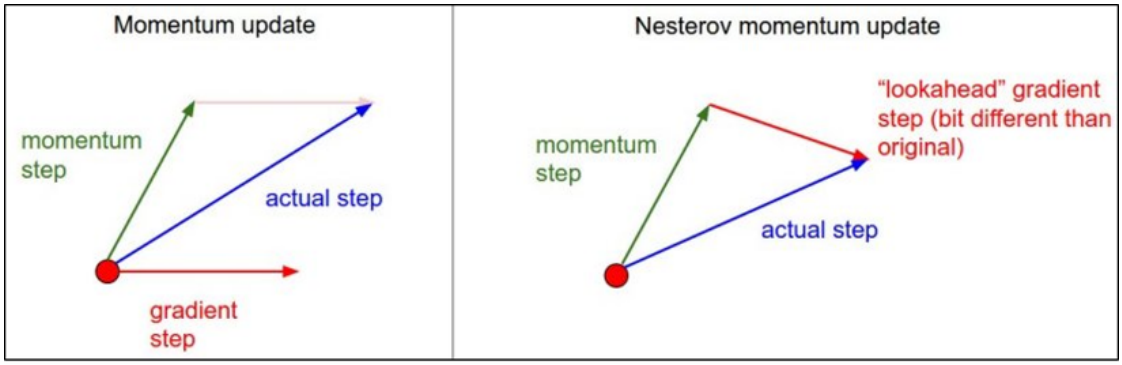
> **네스테로프 가속 경사** (NAG; Nesterov accelerated gradient)

현재 위치가 θ가 아니라 **모멘텀의 방향으로 조금 앞선 θ+βm에서 비용 함수의 그레이디언트를 계산한 것**이다. 기본 모멘텀 최적화보다 일반적으로 훈련 속도가 빠르다. 원래 위치에서의 그레이디언트를 사용하는 것보다 그 방향으로 조금 더 나아가서 측정한 그레이디언트를 사용하는 것이 더 정확하다.

식 11-5 네스테로프 가속 경사 알고리즘

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_\theta J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$





Difference between Momentum and NAG. Picture from CS231.

- `SGD` 옵티마이저를 만들 때 `use_nesterov=True` 라고 설정

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

▼ **11.3.3 AdaGrad**

AdaGrad

경사 하강법은 전역 최적점 방향으로 곧장 향하지 않고 가장 가파른 경사를 따라 빠르게 내려가기 시작해서 골짜기 아래로 느리게 이동한다.

`AdaGrad` 알고리즘은 가**장 가파른 차원을 따라 그레이디언트 벡터의 스케일을 감소**시켜 한쪽이 길쭉한 그릇 문제에서 **경사 하강법이 골짜기 아래로 느리게 이동하는 문제를 해결**한다.

식 11-6 AdaGrad 알고리즘

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$

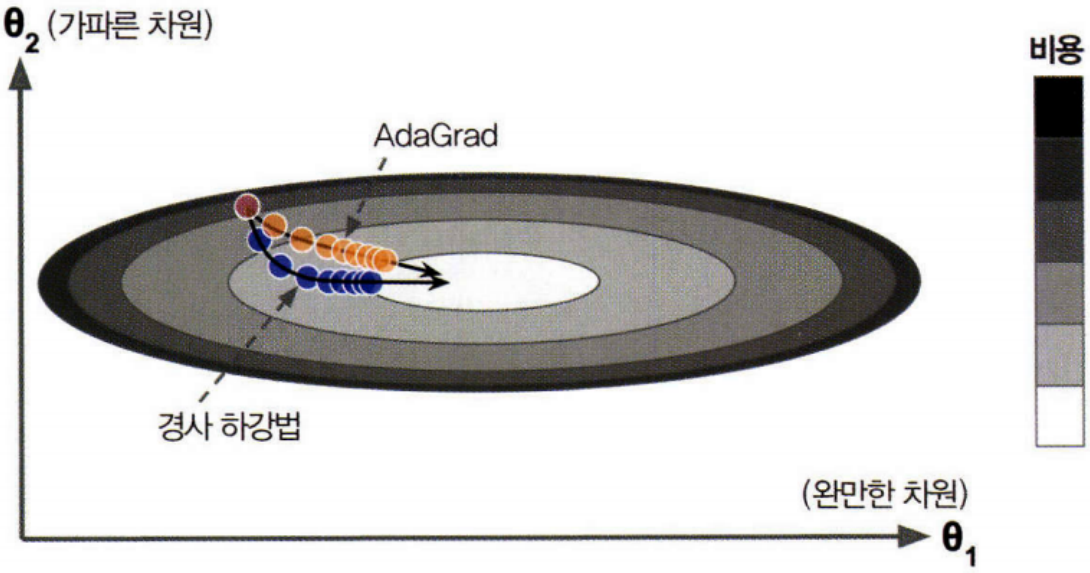
$\oslash$: 원소별 나눗셈
ϵ: 일반적으로 10^{-10}

적응적 학습률(adaptive learning rate)

이 알고리즘은 학습률을 감소시키지만 경사가 완만한 차원보다 가파른 차원에 대해 더 빠르게 감소된다.

2차방정식에 대해 잘 작동하며, **전역 최적점 방향으로 더 곧장 가도록 갱신되는 데 도움**이 된다.

하지만 **단점**으로 학습률이 너무 감소되어 전역 최적점에 도착하기 전에 알고리즘이 완전히 멈추므로 심층 신경망에는 사용하지 말아야 한다.



▼ 11.3.4 RMSProp

AdaGrad는 너무 빨리 느려져서 전역 최적점에 수렴하지 못하는 문제가 있다. RMSProp 알고리즘은 **가장 최근 반복에서 비롯된 그레이디언트만 누적함으로써이 문제를 해결**했다. 알고리즘의 첫 번째 단계에서 지수 감소를 사용한다.

식 11-7 RMSProp 알고리즘

1. $\mathbf{s} \leftarrow \beta\mathbf{s} + (1-\beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta\nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s}+\varepsilon}$

보통 감쇠율 β는 0.9로 설정한다.

```
optimizer = RMSProp(lr=0.001, rho=0.9) # rho는 베타에 해당 (default = 0.9)
```

rho 매개변수는 [식 11 -7] 의 β에 해당한다. 아주 간단한 문제를 제외하고는 이 옵티마이저가 언제나 AdaGrad때보다 훨씬 더 성능이 좋다.

▼ 11.3.5 Adam과 Nadam 최적화

> **Adam (적응적 모멘트 추정,** adaptive moment estimation)

모멘텀 최적화와 **RMSProp** 의 아이디어를 합친 것이다. **모멘텀 최적화처럼 지난 그레이디언트의 지수 감소 평균**을 따르고 **RMSProp처럼 그레이디언트 제곱의 지수 감소된 평균**을 따른다.

식 11-8 Adam 알고리즘

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1-\beta_1)\nabla_\theta J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_\theta J(\boldsymbol{\theta}) \otimes \nabla_\theta J(\boldsymbol{\theta})$
3. $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1-\beta_1^t}$
4. $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\beta_2^t}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$

- t는 1부터 시작하는 반복 횟수

- β1 : 모멘텀 감쇠 하이퍼파라미터

- β2 : 스케일 감쇠 하이퍼파라미터

- **Adam 클래스의 기본값**

: 모멘텀 감쇠 하이퍼파라미터 **β1은 보통 0.9**로 초기화하고, 스케일 감쇠 하이퍼파라미터 **β2는 0.999**로 초기화된다.

ϵ은 보통 아주 작은 숫자($10^{(-7)}$)

- **케라스에서 Adam 옵티마이저 만드는 방법**

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

Adam은 적응적 학습 률알고리즘이기 때문에 학습률 하이퍼파라미터를 튜닝할 필요가 적다. 또한 기본값 $\eta = 0.001$을 일반적으로 사용하므로 경사 하강법보다 Adam 이 사용하기 더 쉽다.

AdaMax 최적화

Adam 알고리즘 개선하기 위한 방법 중 하나이다.

식 11-8 Adam 알고리즘

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1-\beta_1)\nabla_\theta J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_\theta J(\boldsymbol{\theta}) \otimes \nabla_\theta J(\boldsymbol{\theta})$
3. $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1-\beta_1^t}$

$$4.\ \hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\beta_2^t}$$

$$5.\ \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$$

Adam의 2단계를 보면 s에 그레이디언트 제곱을 누적한다(최근 그레이디언트에 더 큰 가중치 부여). 5단계에서는 s의 제곱근으로 파라미터 업데이트의 스케일을 낮추는데, 이는 다시 말하면 시간에 따라 감쇠된 그레이디언트의 $l_2 norm$으로 파라미터 업데이트의 스케일을 낮추는 것이다.

- **AdaMax는 $l_2 norm$을 l_∞ norm으로 바꾼다.**

$$1.\ m \leftarrow \beta_1 m + (1-\beta_1)\nabla_\theta J(\theta)$$
$$2.\ s \leftarrow max(\beta_2 s, \nabla_\theta J(\theta))$$
$$3.\ \hat{m} \leftarrow \frac{m}{1-\beta_1^t}$$
$$4.\ \theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{s+\varepsilon}$$

구체적으로 **2단계에서 $s < max(\beta 2 s, d\theta J(\theta))$로 바꾸고 4단계를 삭제**한다. **5단계에서 s에 비례하여 그레이디언트 업데이트의 스케일을 낮추며** 이는 시간에 따라 감쇠된 그레이디언트의 최댓값을 사용하는 것이다.

이 때문에 **실전에서 AdaMax가 Adam보다 더 안정적이지만 일반적으로 Adam의 성능이 더 낮다**. Adam이 잘 동작하지 않는다면 시도할 수 있는 옵티마이저 중에 하나이다.

Nadam 최적화

Adam 옵티마이저에 네스테로프 기법을 더한 것이다. 일반적으로 Adam보다 빠르게 수렴해 성능이 좋다고 알려져 있다.

지금까지 논의한 모든 최적화 기법은 **1차 편미분** (야코비얀)에만 의존한다. 2차 편미분(헤시안, 야코비안의 편미분)을 기반으로 한 알고리즘들은 심층 신경망에 적용하기 어렵다.

- **옵티마이저 비교**

표 11-2 지금까지 소개한 모든 옵티마이저 비교(* = 나쁨, ** = 보통, *** = 좋음).

클래스	수렴 속도	수렴 품질
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (너무 일찍 멈춤)
RMSprop	***	** 또는 ***
Adam	***	** 또는 ***
Nadam	***	** 또는 ***
AdaMax	***	** 또는 ***

▼ 11.3.6 학습률 스케줄링

좋은 학습률을 찾는 것은 매우 중요하다.

학습률을 너무 **크게 잡으면 훈련이 실제로 발산**하고, 너무 **낮게 잡으면 최적점에 수렴하겠지만 시간이 매우 오래 걸린다**. 만약 조금 높게 잡으면 처음에는 매우 빠르게 진행하겠지만 최적점 근처에서는 요동이 심해져 수렴하지 못한다.

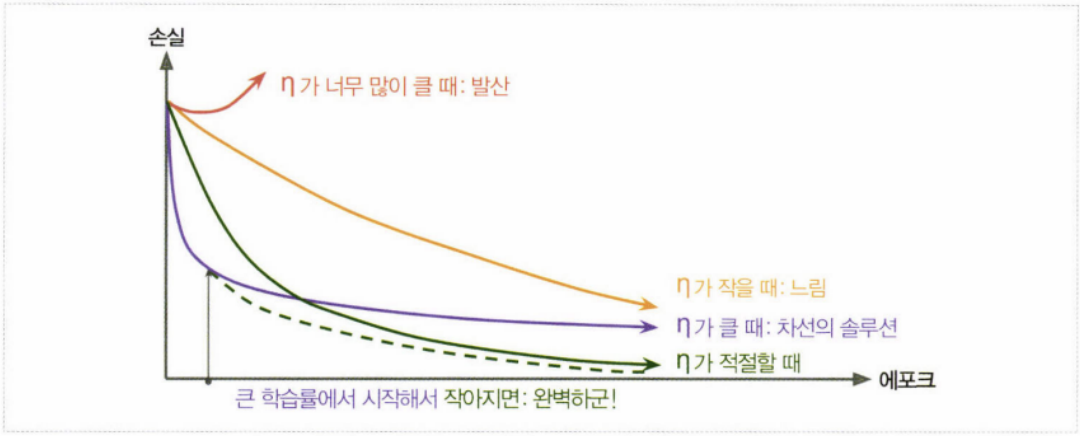


그림 11-8 여러 가지 학습률 η에 대한 학습 곡선

학습 스케줄(learning schedule)

훈련하는 동안 **학습률을 감소시키는 전략**을 말한다. **큰 학습률로 시작하고 학습 속도가 느려질 때 학습률을 낮추면** 최적의 고정 학습률보다 좋은 솔루션을 더 빨리 발견할 수 있다.

거듭제곱 기반 스케줄링 (power scheduling)

- 학습률을 반복 횟수 t에 대한 함수 $\eta(t) = \eta 0/(1 + t/s)^c$로 지정
- $t = k \cdot s$로 커지면 학습률이 $\eta_0 k + 1$로 줄어든다.
- 하이퍼파라미터
  - η0 : 초기 학습률
  - c : 거듭제곱수, 일반적으로 1로 지정
  - s : 스텝 횟수

- 옵티마이저를 만들 때 `decay` 매개변수만 지정한다.

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

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="selu", kernel_initializer="lecun_normal"),
    keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal"),
    keras.layers.Dense(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=
```

```
n_epochs = 25
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid))
```

지수 기반 스케줄링 (exponential scheduling)

- 학습률을 $\eta(t) = \eta_0 * 0.1^{(t/s)}$으로 설정.
- **학습률이 s스텝마다 10배씩 점차 줄어든다. 거듭제곱 기반 스케줄링이 학습률을 갈수록 천천히 감소시키는 반면 지수 기반 스케줄링은 s번 스텝마다 계속 10배씩 감소**한다.

```
def exponential_decay_fn(epoch): # 현재 에포크의 학습률을 받아 반환하는 함수 필요
    return 0.01 * 0.1**(epoch / 20)
```

구간별 고정 스케줄링 (piecewise constant scheduling)

- 일정 횟수의 에포크 동안 일정한 학습률을 사용하고 그 다음 또 다른 횟수의 에포크 동안 작은 학습률을 사용하는 방식이다.

- 학습률

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
```

```
    else:
        return 0.001
```

- 콜백함수 지정

```
lr_scheduler = LearningRateScheduler(piecewise_constant_fn)

history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbakcs=[lr_scheduler])
```

성능 기반 스케줄링 (performance scheduling)

매 N스텝마다 검증 오차를 측정하고 오차가 줄어들지 않으면 **λ배만큼 학습률을 감소**시킨다.

- `ReduceLROnPlateau` 콜백 클래스 활용

```
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.schedules import ExponentialDecay
from tensorflow.keras.optimizers import SGD

# 다음 콜백을 fit() 메서드에 전달하면 최상의 검증 손실이 다섯 번의 연속적인 에포크 동안 향상되지 안흥ㄹ 때마다 학습률에 0.5를 곱한다.
lr_scheduler = ReduceLROnPlateau(factor=0.5, patience=5)

s = 20 * len(X_train) // 32 # 20번 에포크에 담긴 전체 스텝 수 (배치 크기 = 32)
learning_rate = ExponentialDecay(0.01, s, 0.1)
optimizer = SGD(learning_rate=learning_rate)
```

1 사이클 스케줄링 (1cycle scheduling)

- **학습률을 훈련 과정 중에 올리거나 내리도록 조정**
  - 훈련 전반부: 낮은 학습률 η0에서 높은 학습률 η1까지 선형적으로 높힌다.
  - 훈련 후반부: 다시 선형적으로 η0까지 낮춘다.
  - 훈련 마지막 몇 번의 에포크: 학습률을 소수점 몇 째 짜리까지 선형적으로 낮춘다.

- 지수 기반 스케줄링, 성능 기반 스케줄링, 1사이클 스케줄링이 수렴 속도를 크게 높일 수 있다.
- 튜닝 쉽고, 높은 성능, 구현이 쉽기 때문에 **지수 기반 스케줄링** 선호한다.

▼ 11.4 규제를 사용해 과대 적합 피하기

심층 신경망의 높은 자유도는 네트워크를 훈련 세트에 과대 적합되기 쉽게 만들기 때문에 규제가 필요하다.

지금까지 살펴본 규제기법으로는

- **조기종료 기법** : `EarlyStopping` 콜백을 사용하여 일정 에포크 동안 성능이 향상되지 않는 경우 자동 종료시키기
- **배치정규화** : 불안정한 그레이디언트 문제해결을 위해 사용하지만 규제용으로도 활용 가능 (가중치의 변화를 조절하는 역할)

신경망에서 널리 사용되는 규제 방법으로는 l_1 및 l_2 규제, 드롭아웃(dropout), 맥스-노름(max-norm) 규제 등이 있다.

▼ **11 .4.1 l_1과 l_2 규제**

- **l_2 규제 : 신경망의 연결 가중치를 제한하기 위해 사용**
- **l_1 규제 : 많은 가중치가 0인 희소 모델을 만들기 위해 사용**

층을 선언할 때 규제 방식과 규제 강도를 지정할 수 있다.

- 케라스 층의 연결 가중치에 규제 강도 0.01을 사용하여 **l_2규제를 적용하는 방법** - `kernel_regularizer` **옵션 사용**

```
layer = Dense(100, activation='relu',
              kernel_initializer='he_normal',
              kernel_regularizer=l2(0.01))
```

$l_2()$ 함수는 훈련하는 동안 규제 손실을 계산하기 위해 각 스텝에서 호출되는 규제 객체를 반환한다. 이 손실은 최종 손실에 합산된다.

- l_1 규제가 필요하면 `keras.regu1arizers.l1()` **사용**

- l_1와 l_2 가 모두 필요하면 `keras.regu1arizers.l1 l2()` **사용**

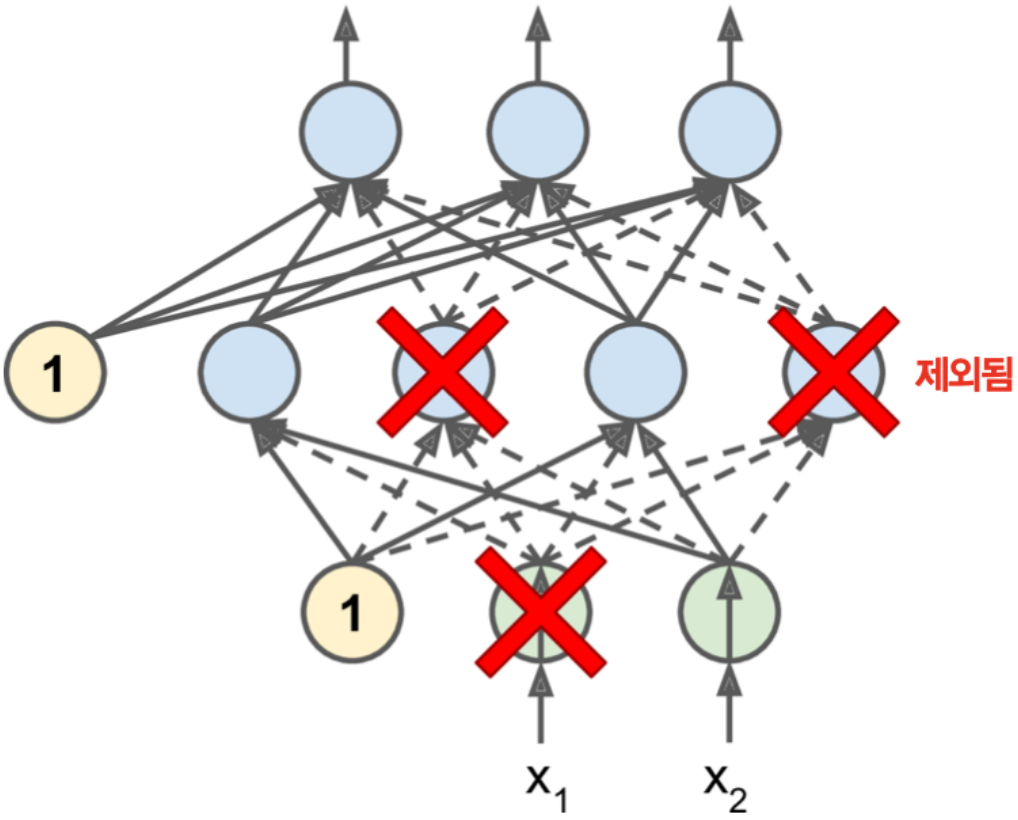
• 일반적으로 네트워크의 모든 은닉층에 동일한 활성화 함수, 동일한 초기화 전략을 사용하거나 모든 층에 동일한 규제를 적용하므로 동일한 매개변수 값을 반복하는 경우가 많다 .

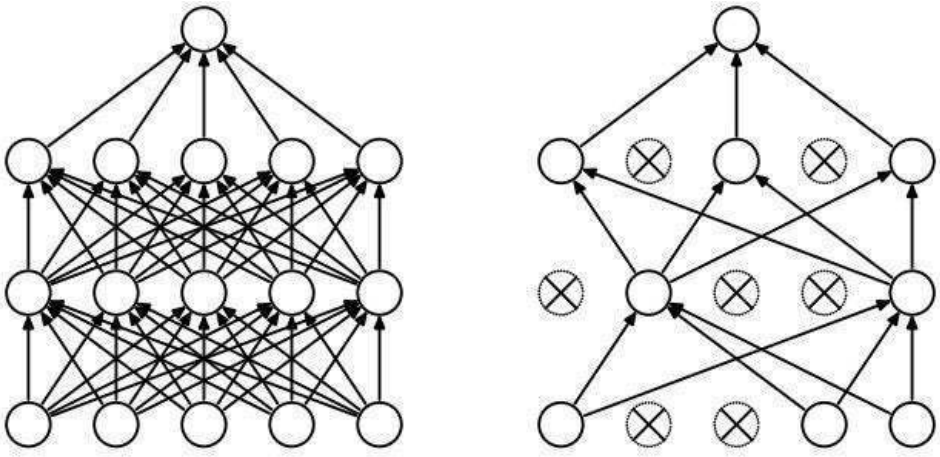
이런 경우 `functools.partial()` 함수를 사용하여 기본 매개변수 값을 사용하여 함수 호출을 감싸는 것이 좋다.

```
from functools import partial

RegularizedDense = partial(Dense,
                           activation='elu',
                           kernel_initializer='he_normal',
                           kernel_regularizer=l2(0.01))

model = Sequential([
    Flatten(input_shape=[28,28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation='softmax',
                     kernel_initializer='glorot_uniform',)
])
```

▼ 11.4.2 드롭 아웃

드롭 아웃




(a) Standard Neural Net (b) After applying dropout.

- **매 훈련 스텝**에서 각 뉴런((입력 뉴런은 포함, 출력 뉴런은 제외)은 **임시적으로 드롭아웃될 확률** p를 가진다. 즉, 이번 훈련 스텝에는 완전히 무시되지만 다음 스텝에는 활성화될 수 있다.

하이퍼파라미터 p를 **드롭아웃 비율**이라고 하고 **보통 10%와 50% 사이를 지정**한다.

- 순환 신경망: 20% - 30%

- 합성곱 신경망: 40% - 50%

드롭이웃으로 훈련된 뉴런은 이웃한 뉴런에 맞추어 적응될 수 없기 때문에 **가능한 자기 자신이 유용**해야 한다. 또 이런 뉴런들은 **몇 개의 입력 뉴런에만 지나치게 의존할 수 없다**. 모든 입력 뉴런에 주의를 기울여야 하기 때문에 입**력값의 작은 변화에 덜 민감**해진다. 결국 **더 안정적인 네트워크가되어 일반화 성능이 좋아진다.** 훈련이 끝난 후에는 뉴런에 더는 드롭아웃을 적용하지 않는다.

각 훈련 스텝에서 고유한 네트워크가 생성된다고 생각할 수 있다. 개개의 뉴런이 있을 수도 없을 수도 있기 때문에 2N개의 네트워크가 가능하다(N=드롭아웃이 가능한 뉴런 수). 이는 아주 큰 값이라 같은 네트워크가 2번 선택될 가능성은 사실 거의 없으며 대부분 가중치를 공유하지만 그럼에도 매우 다르다고 볼 수 있다. 결과적으로 만들어진 신경망은 이 모든 신경망을 평균한 앙상블로 볼 수 있다.

- 케라스에서는 `keras.layers.Dropout` **층을 사용하여 드롭아웃을 구현**한다. 이 층은 훈련하는 동안 일부 입력을 랜덤하게 버린다. (0으로 설정) . 그다음 남은 입력을 보존 확률로 나눈다. 훈련이 끝난 후에는 어떤 작업도 하지 않고 입력을 다음 층으로 그냥 전달만 한다.

- 드롭 아웃 비율 0.2를 사용한 드롭아웃 규제를 모든 Dense 층 이전에 적용

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

  - 훈련시에만 드롭아웃을 적용하기 때문에 훈련손실과 검증손실을 그대로 비교하면 안된다.
  - 훈련 후 드롭아웃을 끄고 훈련손실을 재평가해야 한다.

- 모델이 **과대적합 되었다면 드롭아웃 비율을 늘리고**, **과소적합되면 드롭아웃 비율을 낮추어**야 한다.
- **층이 클 때는 드롭아웃 비율을 늘리고 작은 층에는 드롭이웃 비율을 낮춰**야한다.
- 많은 최신의 신경망 구조는 마지막 은닉층 뒤에만 드롭아웃을 사용한다.

▼ 11.4.3 몬테 카를로 드롭아웃

> ## 몬테 카를로 드롭아웃(Monte Carlo dropout)

훈련된 드롭아웃 모델을 재훈련하거나 수정하지 않으면서 성능을 향상시키는 기법이다. 훈련된 모델의 예측기능을 이용한 결과를 스택으로 쌓은 후 평균값을 계산한다.

```
import numpy as np
```

```
y_probas = np.stack([model(X_test_scaled, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)

# 해석
# 테스트 세트 샘플 수는 10,000개. 클래스 수는 10개. 따라서 [10000, 10] 모양의 행렬 100개
# [100, 10000, 10] 모양의 행렬이 y_probas에 저장됨
```

→ *training=True* 로 지정하여 Dropout 층을 활성화하고 테스트셋에서 100번의 예측을 만들어 쌓는 구조. 최종적으로 [100, 10000, 10] 크기의 행렬을 반환한다.

→ 첫 번째 차원(axis=0)을 기준으로 평균을 내면 한 번의 예측을 수행했을 때와 같은 [10000, 10] 크기의 배열 y_proba를 얻게 된다.

- 모델이 훈련하는 동안 다르게 작동하는 층을 가지고 있다면 앞에서와 같이 훈련 모드를 강제로 설정하면 안 된다. 대신 Dropout 층을 MCDropout 클래스로 바꿔준다.

```
class MCDropout(Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

▼ 11.4.4 맥스-노름 규제

맥스-노름 규제

각각의 뉴런에 대해 입력의 연결 가중치 w가 $||w||_2 \leq r$이 되도록 제한한다.

- r : 맥스-노름 하이퍼파라미터
- $||\cdot||_2$: l_2 노름

맥스-노름 규제는 전체 손실 함수에 규제 손실 항을 추가하지 않는 대신, **훈련 스텝이 끝나고 $||w||_2$를 계산하고 필요하면 w의 스케일을 조정**한다. $(w \leftarrow w * (r/||w||_2)$.

r을 줄이면 규제의 양이 증가하여 **과대적합을 감소**시키는데 도움이 된다. 맥스-노름 규제는 (배치 정규화를 사용하지 않았을 때) **불안정한 그레이디언트 문제를 완화**하는 데 도움을 줄 수 있다.

- 케라스에서 맥스-노름 규제를 구현하려면 적절한 최댓값으로 지정한 `max_norm()` **이 반환환 객체로 은닉층의** `kernel_constraint` **매개변수를 지정**한다.

```
Dense(100, activation='elu', kernel_initializer='he_normal',
      kernel_constraint=max_norm(1.))
```

→ *max_norm()* 함수는 기본값이 0인 axis 매개변수가 있다. *axis=0* 을 사용하면 맥스-노름 규제는 각 뉴런의 가중치 벡터에 독립적으로 적용되는데, CNN에 맥스-노름을 사용하려면 *axis* 매개변수를 적절하게 지정해야 한다.