

그럼 입력으로 0을 넣었을 때 출력으로 1을 얻는 뉴런은 어떻게 만들 수 있을까요? 똑같은 방법으로 계산하면 될까요?

예제 3.13 x=0일 때 y=1을 얻는 뉴런의 학습

```
[IN]
x = 0
y = 1
w = tf.random.normal([1], 0, 1)

for i in range(1000):
    output = sigmoid(x * w)
    error = y - output
    w = w + x * 0.1 * error

    if i % 100 == 99:
        print(i, error, output)
```

[OUT]

```
99 0.5 0.5
199 0.5 0.5
299 0.5 0.5
399 0.5 0.5
499 0.5 0.5
599 0.5 0.5
699 0.5 0.5
799 0.5 0.5
899 0.5 0.5
999 0.5 0.5
```

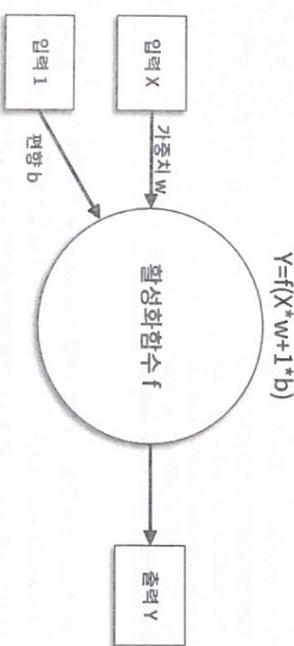


그림 3.12 편향이 더해진 뉴런의 출력 계산식

그럼 실제 코드로 확인해보겠습니다.

예제 3.14 x=0일 때 y=1을 얻는 뉴런의 학습에 편향을 더함

```
[IN]
x = 0
y = 1
w = tf.random.normal([1], 0, 1)
b = tf.random.normal([1], 0, 1)

for i in range(1000):
    output = sigmoid(x * w + 1 * b)
    error = y - output
    w = w + x * 0.1 * error
    b = b + 1 * 0.1 * error

    if i % 100 == 99:
        print(i, error, output)
```

error가 변하지 않고, 출력도 0.5에서 변하지 않습니다. 왜 이런 일이 발생하는 것일까요? 우리가 입력으로 넣은 수는 0입니다. 정사 하강법의 업데이트식은 $w = w + x \times \alpha \times error$ 입니다. $x=0$ 이기 때문에 w 에 더해지는 값은 없습니다. 결국 1,000번의 실행 동안 w 값은 변하지 않습니다.

그럼 어떻게 해야 할까요? 이런 경우를 방지하기 위해 편향(bias)이라는 것을 뉴런에 넣어줍니다. 편향이라는 말처럼 입력으로는 늘 한쪽으로 치우친 고정된 값(예: 1)을 받아서 입력으로 0을 받았을 때 뉴런이 아무것도 배우지 못하는 상황을 방지합니다. 편향의 입력으로는 보편적으로 쓰이는 1을 넣겠습니다.

편향은 w 처럼 난수로 초기화되며 뉴런에 더해져서 출력을 계산하게 됩니다. 수식에서는 관용적으로 bias의 앞 글자인 b 를 씁니다.

[OUT]

```
99 0. 0.069447938207268275 0. 9305296179273173  
199 0. 0.04186835066567673 0. 9581316499363333  
299 0. 0.02360703814277979 0. 9701929612572202  
399 0. 0.023694765971415576 0. 9769852340285844  
499 0. 0.018831817065279144 0. 9811681829347709  
599 0. 0.015888868015282664 0. 9841111319847173  
699 0. 0.012895678439407569 0. 9879043215605925  
799 0. 0.010803125392571333 0. 98919683746074287  
899 0. 0.00975994891937663 0. 9902409051088624
```

편향을 나타내는 b 가彬 번째 줄에 추가됐고, `w처럼 tf.random.normal`로 초기화합니다. 실제 출력인 `output` 값을 계산할 때 `sigmoid(x * w + 1 * b)`라는 수식을 계산해서 각 입력에 기중치와 편향을 곱해서 더해준 뒤 시그모이드 함수를 취합니다. 기대 출력과 실제 출력의 차이인 `error`로 `w`와 `b`를 각각 업데이트해서 뉴런을 학습시킵니다.

프로그램을 실행한 결과, `error`는 0에 가까워지고, `output`은 기대 출력인 1에 가까워집니다. 이로써 학습이 잘 되는 것을 확인할 수 있습니다.

3.3.3 첫 번째 신경망 네트워크: AND

어제 좀 더 의미 있는 일을 하는 신경망 네트워크를 만들어보겠습니다. 이 네트워크의 구성요소는 앞에 서본 것과 같은 하나의 뉴런입니다. 여기서는 AND 연산을 수행하는 뉴런을 만들겠습니다. AND 연산은 여러 개의 입력을 받을 수 있지만, 여기서는 2개의 입력만으로 계산하겠습니다. AND 연산의 진리표는 다음과 같습니다.

표 3.1 2개의 입력을 받을 때 AND 연산의 진리표

입력 1	입력 2	AND 연산
참	참	참
참	거짓	거짓
거짓	참	거짓
거짓	거짓	거짓

[IN]

```
print(int(True))  
print(int(False))
```

[OUT]

1
0

정수로 변환하는 `int` 함수를 사용하면 `True`와 `False`가 각각 1과 0이 되는 것을 알 수 있습니다. 사실 프로그래밍 언어에서는 관습적으로 거짓을 0으로 표시하고, 참을 0이 아닌 다른 값으로 표시해 있습니다.¹³ `if 1:이라는 명령이 if True:와 동일하게 동작하는 것과 같은 이유입니다. 여기서도 관습에 따라 True를 1, False를 0으로 표시하겠습니다.`

이처럼 참, 거짓을 정수로 치환했을 때의 진리표는 다음과 같습니다.

표 3.2 2개의 정수 입력을 받을 때 AND 연산의 진리표

입력 1	입력 2	AND 연산
1	1	1
1	0	0
0	1	0
0	0	0

예외적으로 프로그래밍 언어인 Python에서는 0도 참이고, 1도 거짓이 아닙니다.

AND 연산은 입력이 모두 참 값일 때 참이 되고, 그 밖의 경우에는 모두 거짓이 됩니다. 파이썬에서 참, 거짓을 나타내는 값은 각각 `True`, `False`입니다. 이 값들은 문자열이 아니기 때문에 큰따옴표나 작은따옴표 없이 표시해야 하고, 첫 글자는 대문자로, 나머지 글자는 소문자로 써야 합니다. 그런데 앞에서 본 것처럼 딥러닝의 주요 입력값은 정수(`integer`)나 실수(`float`)입니다. 그렇다면 참, 거짓으로 어떤 수를 사용해야 할까요? 파이썬에서 `True`, `False` 값을 정수로 변환해 보면 힌트를 얻을 수 있습니다.

이제 숫자로 된 데 개의 입력과 출력(AND 연산)의 쌍이 생겼습니다. 이 데이터를 신경망에 넣어 학습시키면 AND 연산을 할 수 있는 신경망 네트워크를 만들 수 있습니다.

사실 하나의 뉴런은 여러 개의 입력을 받을 수 있기 때문에 입력이 2개여도 문제는 없습니다. 이와 달리 편향은 보통 한 개만 사용합니다.

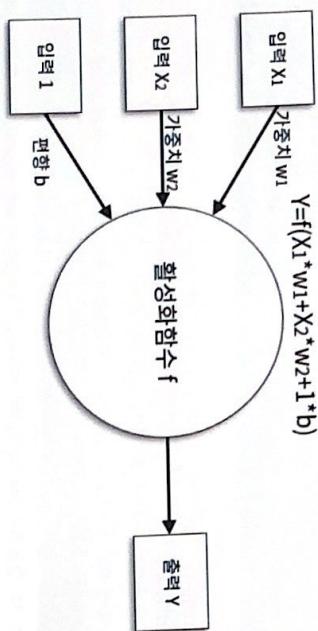


그림 3.13 입력이 2개, 편향이 1개인 뉴런의 출력 계산식

입력과 가중치를 곱한 다음에 서로 더하고(편향도 포함), 활성화함수에 넣으면 뉴런의 출력을 계산할 수 있습니다.

그럼 실제 코드로 확인해보겠습니다.

예제 3.16 첫 번째 신경망 네트워크(AND

```

import numpy as np
x = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
y = np.array([1, 0, 0, 0])
w = tf.random_normal([2, 0, 1])
b = tf.random_normal([1, 0, 1])

b_x = 1

for i in range(2000):
    error_sum = 0
    
```

```

for j in range(4):
    output = sigmoid(np.sum(x[j]*w)+b*x*b)
    error = y[j][0] - output
    w = w + x[j] * 0.1 * error
    b = b + b_x * 0.1 * error
    error_sum += error
    if i % 200 == 199:
        print(i, error_sum)
    
```

일단 첫 번째 줄에는 넘파이(numpy)라는 모듈을 np라는 이름으로 축약해서 불러오는 부분이 추가됐습니다. 넘파이는 수학과 과학 연산에 특화된 파이썬 모듈로, 딥러닝에서도 유용하게 쓰입니다. 여기서는 x와 y를 넘파이 array¹⁴로 정의했습니다. 넘파이 array는 아주 유용한 자료로 표현형으로, 파이썬의 리스트보다 적은 메모리를 차지하며 계산 속도가 빠릅니다.

여기서는 w = w + x[j] * 0.1 * error 부분의 계산을 편리하게 하기 위해 넘파이 array를 썼습니다. 바로 x[j] * 0.1 부분입니다. 참고로 파이썬에서 넘파이 array가 아닌 파이썬 기본 리스트에 숫자를 곱하면 다음과 같은 결과를 얻습니다.

¹⁴ 지금의 정체된 명칭은 tensorflow입니다. tensorflow는 파이썬의 리스트와 비슷한 구조인 numpy를 미리 지정하고 사용합니다. numpy는 파이썬의 리스트보다 메모리를 크게 차지하고 파이썬의 전부 함수를 동일하게 제공할 수 있습니다. 파이썬의 리스트와 numpy의 차이는 이전에 살펴보았던 numpy의 차이입니다.

예제 3.17 파이썬에서 리스트에 정수를 곱하기

```
[IN] print([1,2,3]*2)
print([1,2,3]*0)
print([1,2,3]*1)

[OUT]
[1, 2, 3, 1, 2, 3]
```

```
[IN] []
[OUT]
[]
```

리스트에 정수를 곱하면 양수일 경우 숫자만큼 리스트의 원소를 반복하고, 0 이하일 경우 빈 리스트를 반환합니다. 그리고 0.01 같은 실수를 곱하면 다음과 같은 에러가 발생합니다.

예제 3.18 파이썬에서 리스트에 실수를 곱하기

```
[IN] print([1,2,3]*0.01)

[OUT]
Traceback (most recent call last)
<ipython-input-52-3f950372c6b> in <module>()
      1 print([1,2,3]*0.01)

TypeError: can't multiply sequence by non-int of type 'float'
```

리스트에 정수가 아닌 수는 곱할 수 없다는 에러 메시지가 표시됩니다.

그러면 넘파이 array에 실수를 곱하면 어떨까요?

예제 3.19 넘파이 array에 정수 및 실수 곱하기

```
[IN] import numpy as np
print(np.array([1,2,3])*2)
print(np.array([1,2,3])*0)
print(np.array([1,2,3])*1)

[OUT]
[2, 4, 6]
[0, 0]
[-1, -2, -3]
```

```
[IN] print(np.array([1,2,3])*1)

[OUT]
[1, 2, 3]
```

넘파이 array에 실수를 곱하면 array의 각 원소에 대해 자동으로 실수를 곱하는 연산이 이뤄집니다. 이를 각 원소에 대한(element-wise) 연산이라고 합니다.

결과값을 자세히 보면 `sum()`가 없는 것도 확인할 수 있습니다. 각 원소에 대한 연산으로 나온 결과값은 파일의 리스트가 아닌 넘파이 array로 자동 변환되기 때문입니다. 넘파이 array는 출력할 때 슬표를 표시하지 않습니다.

그럼 다시 예제 3.16에서 가중치를 업데이트하는 $w = w + x[j] * 0.1 * \text{error}$ 에서 $x[j]$ 는 j의 값이 0에서 3으로 변함에 따라 [1, 1], [1, 0], [0, 1]이 됩니다. 이처럼 여러 개의 수에 실수를 직접 곱해서 한번에 결괏값을 얻기 위해 넘파이 array를 사용하는 것입니다.

다음 부분은 지금까지 배운 예제와 같습니다. 입력의 수가 4개로 많아졌기 때문에 네트워크가 수렴하는 데는 더 많은 연산이 필요합니다. 예제 3.16의 출력에서는 `error`의 합인 `error.sum()`이 점점 줄어드는 것을 확인할 수 있습니다.

그럼 이렇게 학습시킨 네트워크가 정상적으로 작동하는지 평가해보겠습니다. 네트워크에 x의 각 값을 넣을 때, 실제 출력이 기대 출력인 y 값에 얼마나 가깝게 되는지 예제 3.20에서 확인할 수 있습니다.

예제 3.20 AND 네트워크의 평가

```
[IN] for i in range(4):
    print('x:', x[i], 'y:', y[i], 'Output:', sigmoid(np.sum(x[i]*w)+b))

[OUT]
x: [1 1] y: [1] Output: 0.9651505299125843
x: [1 0] y: [0] Output: 0.02469764353276747
x: [0 1] y: [0] Output: 0.02477248103015434
x: [0 0] y: [0] Output: 2.3225808850192006e-05
```

실제 출력이 기대 출력에 가깝게 나오는 것을 확인할 수 있습니다. 마지막 output의 2.322580850

192006e-05는 과학적 표기법으로, 실수를 기수와 지수로 표현하는 방법입니다. e 앞에 오는 수는 가

수가 되고 e 뒤에 오는 수는 지수가 됩니다. 2.322580850192006e-05는 0.00002322580850192006

과 같은 수입니다.

이제 AND 연산의 각 진리표에 대해 기대 출력과 가까운 값을 출력하는 네트워크를 학습시켰습니다. 비록 뉴런 하나로 구성된 네트워크이지만 의미 있는 한 걸음입니다. 그럼 다른 네트워크도 학습시켜보겠습니다.

3.3.4 두 번째 신경망 네트워크: OR

OR은 AND와 매우 비슷한 진리표를 가지고 있습니다. AND 연산의 입력이 모두 참일 때만 결과값이 참이었던 것과 다르게 OR 연산은 입력 중 하나만 참이어도 결과값이 참이 되고, 모두 거짓일 때만 거짓이 됩니다.

표 3.3 2개의 입력을 받을 때 OR 연산의 진리표

입력 1	입력 2	OR 연산
참	참	참
참	거짓	참
거짓	참	참
거짓	거짓	거짓

표 3.4 2개의 정수 입력을 받을 때 OR 연산의 진리표

입력 1	입력 2	OR 연산
1	1	1
1	0	1
0	1	1
0	0	0

OR 네트워크를 평가하는 코드와 결과는 다음과 같습니다.

예제 3.21 두 번째 신경망 네트워크: OR

```
[IN]
import numpy as np
x = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
y = np.array([[1], [1], [1], [0]])
w = tf.random.normal([2, 0, 1])
b = tf.random.normal([1, 0, 1])
b_x = 1
```

```
for i in range(2000):
    error_sum = 0
```

```
    for j in range(4):
        output = sigmoid(np.sum(x[j]*w)+b_x*b)
```

```
        error = y[j][0] - output
```

```
        w = w + x[j] * 0.1 * error
```

```
        b = b + b_x * 0.1 * error
```

```
        error_sum += error
```

```
if i % 200 == 199:
    print(i, error_sum)
```

예제 3.22 OR 네트워크의 평가

```
[IN]
for i in range(4):
    print('X:', x[i], 'Y:', y[i], 'Output:', sigmoid(np.sum(x[i]*w)+b))
```

OR 연산을 계산하는 네트워크를 생성하는 코드도 예제 3.16의 AND 네트워크와 매우 비슷합니다. 달라진 것은 y 부분의 기대 출력뿐입니다.

[OUT]

```
x: [1, 1] y: [1] output: 0.999997331473692
x: [1, 0] y: [1] output: 0.9899476313428319
x: [0, 1] y: [1] output: 0.9899158941652392
x: [0, 0] y: [0] output: 0.025148692483856868
```

학습 수(for i in range(2000))의 2000)를 바꾸어보면 학습 수가 커질수록 실제 출력이 기대 출력에 가까워지는 것을 확인할 수 있습니다. 이렇게 OR 네트워크도 학습시켰으니 이제 인공지능의 거울을 볼려웠던 XOR 네트워크에 도전할 차례입니다.

3.3.5 세 번째 신경망 네트워크: XOR

XOR은 OR과 비슷하지만 한 가지 차이가 있습니다. 바로 흔히 개의 입력이 참일 때만 결과값이 참이 됨이라는 점입니다. 간단하게 입력이 2개일 때는 입력 2개의 값이 서로 다를 때라고 생각해도 좋습니다.

표 3.5 2개의 입력을 받을 때 XOR 연산의 진리표

입력 1	입력 2	XOR 연산
참	참	거짓
참	거짓	참
거짓	참	참
거짓	거짓	거짓

표 3.6 2개의 정수 입력을 받을 때 XOR 연산의 진리표

입력 1	입력 2	XOR 연산
1	1	0
1	0	1
0	1	1
0	0	0

XOR 연산도 위에서 작성했던 AND, OR 네트워크와 똑같이 작성해보면 어떻게 될까요?

[IN]

```
import numpy as np
x = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
y = np.array([[0], [1], [1], [0]])
w = tf.random_normal([2, 0, 1])
b = tf.random_normal([1], 0, 1)
b_x = 1
```

```
for i in range(2000):
    error_sum = 0
```

```
    for j in range(4):
        output = sigmoid(np.sum(x[j]*w)+b_x*b)
        error = y[j][0] - output
```

```
w = w + x[j] * 0.1 * error
b = b + b_x * 0.1 * error
error_sum += error
```

```
if i % 200 == 199:
    print(i, error_sum)
```

[OUT]

```
199 -0.0017828529912741198
399 -7.249766076933284e-05
599 -2.94758653376703e-06
799 -1.3399160462604552e-07
999 4.653552654332316e-09
1199 3.722842145670313e-09
1399 3.722842145670313e-09
1599 3.722842145670313e-09
1799 3.722842145670313e-09
1999 3.722842145670313e-09
```

예제 3.23 세 번째 신경망 네트워크: XOR

예제 3.24 XOR 네트워크의 링가

```
[IN]
for i in range(4):
    print('X:', x[i], 'Y:', y[i], 'Output:', sigmoid(np.sum(x[i]*w)+b))

[OUT]
x: [1 1] y: [0] Output: 0.5128176286712095
x: [1 0] y: [1] Output: 0.5128176395326365
x: [0 1] y: [1] Output: 0.499999999999586774
x: [0 0] y: [0] Output: 0.50000000009313226
```

위와 output 사이에는 큰 차이가 있어 보입니다. x가 변해도 output은 0.5 근처에서 머물고 있습니다. 어제 서 이런 결과가 나오는 것일까요?

output = sigmoid(np.sum(x[j]**w+b*xb)) 공식을 구성하는 w와 b를 출력해보면 다음과 같습니다.

예제 3.25 XOR 네트워크의 w, b 값 확인

```
[IN]
print('w:', w)
print('b:', b)

[OUT]
w: tf.Tensor([ 5.1281754e-02 -7.4595806e-09], shape=(2,), dtype=float32)
# b: tf.Tensor([-0.601849], shape=(1,), dtype=float32)
```

표 3.7 XOR 네트워크의 입력과 중간 계산, 출력

X1	X2	중간 계산	출력
1	-	np.sum(x[j]*w)+b	sigmoid(np.sum(x[j]*w)+b)
1	0	0.05128175	0.5128176286893302
0	1	-3.7252903e-09	0.4999999999586774
0	0	3.7252903e-09	0.50000000009313226

중간 계산 값이 0에 가까워지기 때문에 최종 출력이 0.5에 가까워집니다. 첫 번째 입력에 따라 중간 계산값은 크게 달라지지만 출력에서는 별 차이가 없어집니다. 반면 AND 네트워크의 w와 b 값을 출력해보면 다음과 같습니다.

예제 3.26 AND 네트워크의 w, b 값 확인

```
# w: tf.Tensor([ 6.9484286 6.951607 ], shape=(2,), dtype=float32)
# b: tf.Tensor([-0.601849], shape=(1,), dtype=float32)
```

표 3.8 AND 네트워크의 입력과 중간 계산, 출력

X1	X2	중간 계산	출력
1	1	np.sum(x[j]*w)+b	sigmoid(np.sum(x[j]*w)+b)
1	0	-3.298866	0.964366548024708
0	1	-3.6534204	0.02524838724984636
0	0	-3.650242	0.025326728101507022
		-10.601849	2.48689364094058595e-05

w는 약 0.0512, -0.0000000745이고, b는 약 0.00000000373입니다. w에 순차적으로 x가 곱해지기 때문에 첫 번째 입력이 두 번째 입력보다 큰 영향을 끼치고, 편향은 두 번째 입력과 비슷하게 거의 영향이 없는 것을 알 수 있습니다.

이렇게 얻은 값에 시그모이드 함수를 취하면 최종값이 됩니다.

예제 3.27 tf.keras를 이용한 XOR 네트워크 계산

XOR 네트워크	가중치 w1 5.1281754e-02
	가중치 w2 -7.4505806e-09
	편향 b 3.7252903e-09

그림 3.14 XOR과 AND 네트워크의 가중치 그림프로

AND 네트워크	가중치 w1 6.9484285
	가중치 w2 6.951607
	편향 b -10.601849

AND 네트워크의 가중치가 하려는 작업은 XOR 네트워크에 비해 분명합니다. 두 개의 가중치가 비슷하기 때문에 입력 2개는 서로 거의 비슷한 중요도를 가집니다. 편향값은 큰 음수인데, 이것은 중간 계산값을 음수로 보내는 경향을 가집니다. 두 가중치를 모두 합쳐야 음수인데, 이것은 증간 계산값을 가지고 있는 하지만 중간 계산값은 0에 가까워지고 시그모이드 함수를 취한 값은 0.5에 가까워질 뿐입니다.

이것이 바로 첫 번째 인공지능의 계율을 불러온 것으로 잘 알려진 XOR 문제(XOR Problem)입니다. 하버의 괴팅트론으로는 간단한 XOR 연산자도 만들 수 없다는 것을 동명의 책인 『괴팅트론(Perceptron)』에서 마린 민스키(Marvin Minsky)와 시모어 페퍼트(Seymour Papert)가 증명해냈습니다.

그럼 해결책은 무엇일까요? 바로 여러 개의 괴팅트론을 사용하는 것입니다. 사실 『괴팅트론』에서도 여러 개의 괴팅트론을 사용하면 XOR 문제를 포함한 어떤 불린 함수(Boolean function, 정수를 넣었을 때 0 또는 1이 출력되는 함수)를 만들 수 있다는 사실을 언급하고 있습니다.

여기서는 세 개의 괴팅트론과 뉴런을 사용해 보겠습니다. 코드가 복잡해지는 것을 막기 위해 1.2.4절에서 살펴보았던 tf.keras를 사용하겠습니다. 네트워크를 만드는 코드는 다음과 같습니다.

```
[IN]
import numpy as np
x = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
y = np.array([[0], [1], [1], [0]])

model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=2, activation='sigmoid', input_shape=(2,)),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.1), loss='mse')

model.summary()
```

```
[OUT]
Model: "sequential"
Layer (type)                 Output Shape            Param #
dense (Dense)                (None, 2)               6
dense_1 (Dense)              (None, 1)               3
Total params: 9
Trainable params: 9
Non-trainable params: 0
```

먼저 model에 대해 알아보겠습니다. tf.keras에는 딥러닝 계산을 간편하게 하기 위한 추상적인 클래스인 model이 있습니다. 쉽게 말해서 딥러닝 계산을 위한 여러 함수와 변수의 묶음이라고 생각하면 됩니다. model은 tf.keras에서 딥러닝을 계산하는 가장 핵심적인 단위인 만큼 앞으로 이 책의 전반에 걸쳐 꾸준히 나올 것입니다.

model에서 가장 많이 쓰이는 구조가 tf.keras.Sequential 구조입니다. 이것은 영어의 뜻 그대로 순차적(sequential)으로 뉴런과 뉴런이 학습된 단위인 레이어를 일직선으로 배치한 것입니다. 이 책에서는 외국어 표기법 그대로 시퀀셜 네트워크, 시퀀셜 모델로 부르겠습니다.

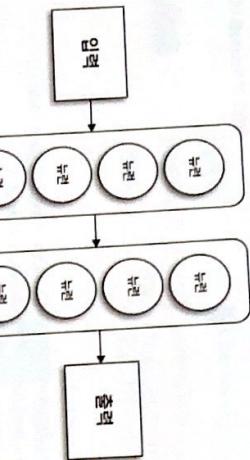


그림 3.15 tf.keras.Sequential의 일자선 구조

시퀀셜 모델의 인수로는 레이어가 차례대로 정의된 리스트를 전달합니다.

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=2, activation='sigmoid', input_shape=(2,)),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])
```

`tf.keras.layers.Dense`는 `model`에서 사용하는 레이어를 정의하는 명령입니다. `Dense`는 가장 기본적인 레이어로서, 레이어의 입력과 출력 사이에 있는 모든 뉴런이 서로 연결되는 레이어입니다.

`tf.keras.layers.Dense` 안의 `units`는 레이어를 구성하는 뉴런의 수를 정의합니다. 뉴런이 많을수록 일반적으로 레이어의 성능은 좋아지지만 계산량 또한 많아지고 메모리도 많이 차지하게 됩니다. `activation`은 우리가 계속 봐온 활성화함수입니다. 여기서는 두 레이어에 모두 `sigmoid`를 썼습니다.

`input_shape`은 시퀀셜 모델의 첫 번째 레이어에서만 정의하는데, 입력의 차원 수가 어떻게 되는지를 정의합니다. 여기서는 각 데이터가 $[1, 1]$, $[1, 0]$ 처럼 2개의 입력을 받는 1차원 `array`이기 때문에 1차원의 원소의 개수인 2를 명시해서 (2) 라고 정의했습니다. 3.3.1절의 "난수 생성"에서도 비슷한 내용을 다뤘습니다.

이렇게 정의된 2-레이어 XOR 네트워크의 구조를 그림으로 나타내면 다음과 같습니다.

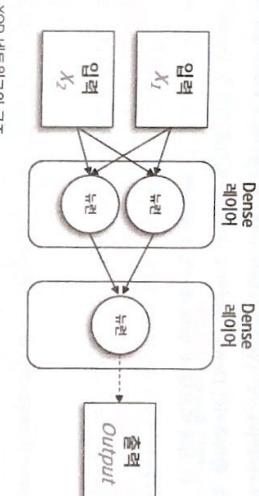


그림 3.16 2-레이어 XOR 네트워크의 구조

네트워크의 구조에서 실선으로 그려진 화살표는 가중치를 나타냅니다. 입력에서 첫 번째 레이어로 향하는 화살표가 4개, 첫 번째 레이어에서 두 번째 레이어로 향하는 화살표가 2개인 것을 확인할 수 있습니다. 두 번째 레이어의 결과에 활성화함수를 취한 결과가 바로 출력이 되기 때문에 마지막 화살표는 가중치로 치지 않습니다.

그런데 [OUT]에 표시된 콘솔 출력 결과에서 `Param #`를 보면, 첫 번째 레이어에는 6개, 두 번째 레이어에는 3개의 파라미터가 있다고 나옵니다. 이것은 바로 각 레이어가 기본적으로 편향을 포함하고 있기 때문입니다. 편향을 포함한 네트워크의 구조를 다시 그려면 다음과 같습니다.

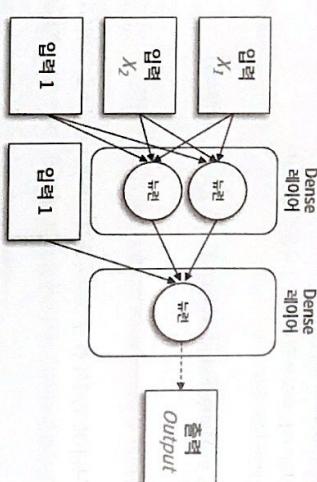


그림 3.17 편향을 포함한 2-레이어 XOR 네트워크의 구조

보통 Dense 레이어의 파라미터 수는 (입력층 뉴런의 수+1) × (출력층 뉴런의 수)의 식으로 구할 수 있습니다. 여기서 입력층, 출력층이란 Dense 레이어에 들어오는 입력을 입력층, Dense 레이어의 뉴런을 출력층이라고 합니다. 이 식에 따르면 첫 번째 레이어의 파라미터 수는 $(2+1) \times 2 = 6$ 이고, 두 번째 레이어의 파라미터 수는 $(2+1) \times 1 = 3$ 으로 [OUT]에서 출력되는 결과와 동일합니다.

다시 코드 설명으로 돌아와서, 다음 부분은 model이 실제로 동작할 수 있도록 준비하는 명령입니다.

```
model.compile(optimizer=tf.keras.optimizers.SGD(1=0.1), loss='mse')
```

최적화 함수(optimizer)는 딥러닝의 학습식을 정의하는 부분입니다. 원래는 미분과 복잡한 수학을 써야 하지만, tf.keras에서는 이렇게 미리 정의된 최적화 함수를 불러오는 것으로 바로 사용할 수 있습니다. SGD는 확률적 경사 하강법(Stochastic Gradient Descent)의 약자이며, 여기서 경사 하강법은 앞에서 살펴본 대로 가중치를 업데이트할 때 미분을 통해 기울기를 구한 다음 기울기가 낮은 쪽으로 업데이트 하겠다는 뜻입니다. 확률적(Stochastic)은 전체를 한번에 계산하지 않고 확률적으로 일부 샘플을 구해서 조금씩 나눠서 계산하겠다는 뜻입니다. 자세한 내용이 궁금하신 분들은 구글 머신러닝 단기진증과정의 “손실 줄이기”¹⁵ 부분을 참고하면 좋습니다.

손실(loss)은 앞에서 살펴본 error와 비슷한 개념입니다. 딥러닝은 보통 이 손실을 줄이는 방향으로 학습 합니다. mse는 평균 제곱 오차(Mean Squared Error)의 약자로, 기대출력에서 실제출력을 뺀 뒤에 제곱한 값을 평균하는 것입니다. 수식으로 나타내면 다음과 같습니다.

$$\text{Mean Squared Error} = \frac{1}{n} \sum_{k=1}^n (y_k - \text{output}_k)^2$$

앞의 예제들에서 썼던 예리 식 $\text{error} = y - \text{output}$ 과 비슷한 기능을 합니다.

그다음에 나오는 부분은 [OUT]에서 볼 수 있듯이 현재 네트워크의 구조를 알아보기 쉽게 출력하는 기능입니다. 이 명령을 실행했을 때 예리 메시지가 표시된다면 뭔가 문제가 있는 것입니다.

```
model.summary()
```

그럼 이제 네트워크를 실제로 학습시켜볼 차례입니다.

예제 3.28 tf.keras를 이용한 2-레이어 XOR 네트워크 학습

```
[IN] history = model.fit(x, y, epochs=2000, batch_size=1)
```

[OUT]

```
Epoch 1/2000  
4/4 [=====] - 0s 17ms/sample - loss: 0.2718  
Epoch 2/2000  
4/4 [=====] - 0s 2ms/sample - loss: 0.2697  
Epoch 3/2000  
4/4 [=====] - 0s 2ms/sample - loss: 0.2697  
...  
Epoch 1999/2000  
4/4 [=====] - 0s 2ms/sample - loss: 0.0022  
Epoch 2000/2000  
4/4 [=====] - 0s 2ms/sample - loss: 0.0022
```

model.fit() 함수는 앞의 예제에서 for 문을 실행한 것처럼 예포크(epochs)¹⁶에 지정된 횟수만큼 학습합니다. batch_size는 한번에 학습시키는 데이터의 수인데, 여기서는 1로 지정해서 입력을 넣었을 때 정확한 값을 출력하는지 알아보려고 합니다. 첫 부분의 x, y는 각각 입력과 기대출력을 나타냅니다. 학습이 끝나면 네트워크를 평가해볼 수 있습니다. 여기서도 간단한 함수 하나만 사용하면 됩니다.

예제 3.29 tf.keras를 이용한 2-레이어 XOR 네트워크 평가

[IN]

```
model.predict(x)
```

```
[OUT]  
array([ 0.05724718,  
       [ 0.9555997 ],  
       [ 0.9552847 ],  
       [ 0.03977039 ]], dtype=float32)
```

15 예리(Foolish)는 훈련 단계를 빼고 훈련시키는 뜻입니다. 예리 그림자면 예리라는 데 예고제 훈련시키는 훈련 데이터의 수입니다.

`model1.predict()` 함수에 입력을 넣으면 네트워크의 출력 결과를 알 수 있습니다. 첫 번째와 세 번째 값은 0에 가깝고, 두 번째와 세 번째 값은 1에 가깝게 나오았습니다. 예제 3.24와 비교하면 XOR 네트워크를 잘 계산하고 있습니다.

어떻게 이런 결과가 나오는 걸까요? `model1`을 구성하고 있는 파라미터, 즉 가중치와 편향을 출력해보겠습니다.

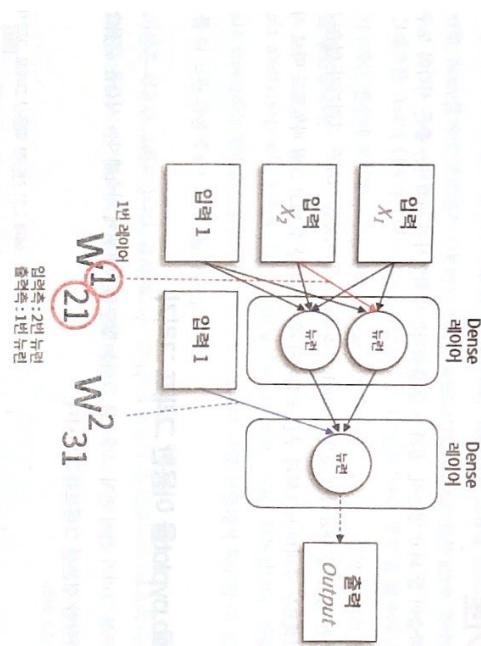
```
[In] for weight in model1.weights:  
    print(weight)
```

```
[Out]  
<tf.Variable 'dense_14/kernel:0' shape=(2, 2) dtype=float32, numpy=  
array([[-4.0317945, -6.0787964],  
       [-4.0165396, -5.9774931]], dtype=float32)>  
<tf.Variable 'dense_14/bias:0' shape=(2,) dtype=float32, numpy=array([15.927255 ,  2.2947845],  
dtype=float32)>  
<tf.Variable 'dense_15/kernel:0' shape=(2, 1) dtype=float32, numpy=  
array([[ 7.936609],  
      [-8.1995941]], dtype=float32)>  
<tf.Variable 'dense_15/bias:0' shape=(1,) dtype=float32, numpy=array([-3.656685],  
dtype=float32)>
```

가중치 `model1.weights[0]`에 저장되어 있습니다. 입력과 레이어 또는 레이어 사이의 뉴런을 연결할 때 사용되는 가중치는 `kernel`이고, 편향과 연결된 가중치는 `bias`로 표시됩니다.
보통 네트워크의 가중치 숫자가 많기 때문에 구분을 위해 편의상 가중치에 침자를 붙여서 표시합니다.

레이어의 순서대로 위침자를 붙이고, 아래 침자는 각 뉴런의 순서에 맞게 차례로 붙입니다.
그림 3.19 2-레이어 XOR 네트워크의 가중치 그래프화

앞에서 본 1개의 뉴런, 1개의 레이어를 사용한 XOR, AND와는 다르게 뉴런 개수가 3개, 레이어 개수가 2개로 늘자 이 가중치들이 무슨 일을 하는지 한눈에 잘 들이오지 않습니다. 뉴런과 레이어가 많아지면 이 문제는 더욱 커집니다. 가중치 시각화보다 네트워크의 학습 상황을 더 잘 파악할 수 있는 방법이 필요합니다.



예제 3.30 2-레이어 XOR 네트워크의 가중치와 편향 확인

3.4 시각화 기초

딥러닝 네트워크의 학습이 잘 되고 있는지, 결과가 잘 출력되는지 확인하기 위해서는 좋은 시각화 도구가 필요합니다.

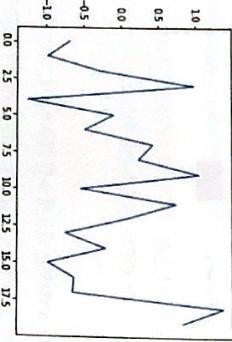
파이썬에서 시각화하는 방법 중 대표적인 것으로 matplotlib.pyplot을 이용한 그레프 그리기가 있습니다. 이렇게 그린 그레프는 주피터 노트북과 구글 코랩에서 즉시 확인할 수 있고 그림 파일로도 따로 저장할 수 있습니다.

3.4.1 matplotlib.pyplot을 이용한 그레프 그리기

딥러닝 데이터 그레프를 그리기 전에 먼저 그레프를 그리는 함수의 사용법에 익숙해지는 시간을 갖겠습니다. 다음 예제는 간단한 깨은선 그레프를 그립니다.

예제 3.31 간단한 깨은선 그레프 그리기

```
[IN]  
import matplotlib.pyplot as plt  
x = range(20)  
y = tf.random.normal([20], 0, 1)  
plt.plot(x, y)  
plt.show()
```



matplotlib.pyplot을 사용하기 위해서는 tensorflow, numpy처럼 모듈을 임포트해야 합니다. tensorflow를 tf로, numpy를 np로 축약해서 사용하는 것처럼 matplotlib.pyplot은 보통 plt로 축약합니다.

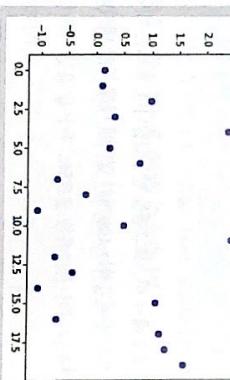
그레프를 그리기 위해서는 데이터가 필요합니다. 3.1절의 “난수 생성”을 활용해 랜덤 데이터를 생성해서 y라는 변수에 저장했습니다. x에는 range(20)으로 [0, 1, 2, ..., 19]의 20개의 정수로 구성된 리스트를 넣었습니다.

plt.plot(x, y)는 x축, y축에 각각 x, y를 넣어서 그레프를 그리는 부분입니다. 이렇게 그려진 그레프는 plt.show()라는 함수를 호출하면 주피터 노트북이나 구글 코랩에서 확인할 수 있습니다. 보통 그레프를 다 그린 다음에 plt.show()를 호출합니다.

기본은 깨은선 그레프입니다. 표시 형식은 얼마든지 조절할 수 있습니다. 점으로 바꾸려면 plt.plot(x, y, 'bo')라고 입력하면 됩니다.

예제 3.32 간단한 점 그레프 그리기

```
[IN]  
import matplotlib.pyplot as plt  
x = range(20)  
y = tf.random.normal([20], 0, 1)  
plt.plot(x, y, 'bo')  
plt.show()
```

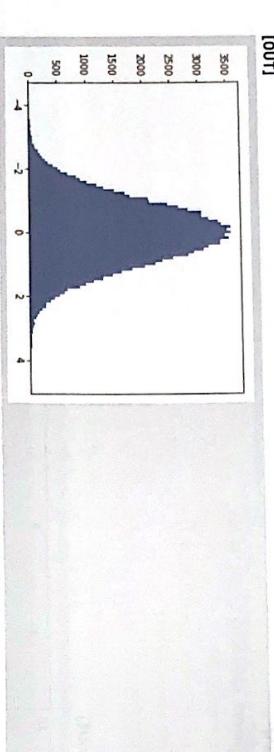


여기서 추가된 ‘b’ 부분은 파란색(blue) 점(o)을 나타냅니다. 선을 나타내고 싶으면 ‘b-’, 점선을 나타내고 싶으면 ‘b--’, 색깔을 바꾸고 싶으면 b를 다른 색을 나타내는 값(빨간색(r), 노란색(y), 초록색(g), 검은색(k) 등)으로 바꾸면 됩니다.

또 한 가지 알아두면 유용한 그래프는 히스토그램입니다. 앞에서 살펴본 정규분포가 실제로 가운데에서 빈번한 값을 가지고 가장자리에서 드물게 나타나는지를 확인하기 위해 히스토그램을 그려볼 수 있습니다.

예제 3.33 정규분포 그래프를 히스토그램으로 나타내기

```
[IN]
import matplotlib.pyplot as plt
random_normal = tf.random.normal([100000], 0, 1)
plt.hist(random_normal, bins=100)
plt.show()
```



100,000개의 난수를 생성한 다음 plt.hist() 함수를 호출해서 히스토그램을 만들었습니다. bins는 대입자를 얼마나 많은 수의 영역으로 나눌지 정의하는 것으로서 여기서는 100개의 영역에 대해 각각 데이터의 수인 도수를 계산한 뒤에 막대그래프로 그려줍니다.

그리고 풀어서 처음에는 손실이 서서히 감소하다가 어느 시점부터 급격히 감소하고, 나중에는 거의 감소하지 않는 뒤집힌 S자곡선을 그리는 모습을 확인할 수 있습니다. 이렇게 손실을 시각화하면 네트워크의 학습 현황을 한눈에 파악할 수 있습니다.

참고로 plt.plot()에 하나의 변수만 전달하면 그 변수를 y로 간주하고, x는 자동으로 range(len(y))에 해당하는 값을 넣어서 그레프를 만들어줍니다. 위에서 x를 입력하지 않았는데도 x축에 값이 출력되는 이유입니다.

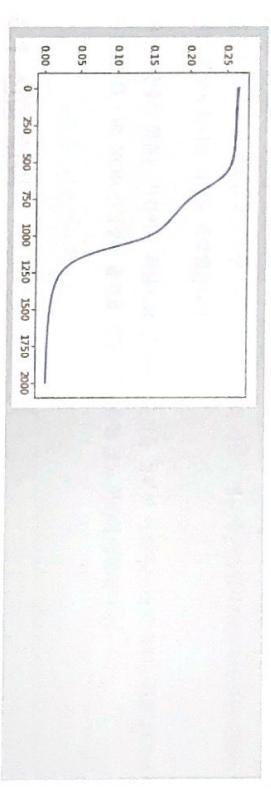
3.4.2 2-레이어 XOR 네트워크의 정보 시각화

딥러닝을 학습시킬 때 가장 많이 보게 될 그래프는 바로 학습이 잘 되고 있는지 확인하기 위한 측정치(metric) 변화량을 나타내는 선 그래프입니다.¹⁷ 여기서는 선 그래프를 이용해 손실이 어떻게 변했는지를 알아보겠습니다. 예제 3.28에서 history라는 변수에 tf.keras.history를 진행한 내용을 저장했습니다.

여기서 저장되는 정보 중 loss를 물리ически 그래프를 그릴 수 있습니다.

예제 3.34 2-레이어 XOR 네트워크의 손실 변화를 선 그래프로 표시

```
[IN]
import matplotlib.pyplot as plt
plt.plot(history.history['loss'])
```



3.5 정리

이번 장에서는 구글 코랩으로 작성된 예제를 통해 텐서플로의 기초를 배우고, 그 과정에서 파이썬의 넘파이 등 딥러닝 계산에 필요한 기초적인 라이브러리의 사용법에 대해서도 간단하게 살펴봤습니다.

또한 실제로 동작하는 AND, OR, XOR 네트워크를 사용하는 방법을 배웠고, `tf.keras`를 이용해 간편하게 2-레이어 구조의 네트워크를 만들 수 있다는 사실을 배웠습니다. 마지막으로 `matplotlib.pyplot`을 이용해 학습 결과를 확인하기 위해 시각화하는 방법도 간단히 살펴봤습니다.