03

텐서플로 2.0 시작하기

이제 텐서플로를 설치했으니 직접 코드를 타이핑해볼 시간입니다. 이 책의 예제는 모두 구글 코랩으로 만들어졌기 때문에 인터넷이 접속되는 환경이라면 어디서든 아무것도 설치하지 않고도 예제를 실행할 수 있습니다. 하지만 예제 코드를 완전히 이해하려면 처음부터 끝까지 최소한 한번은 쭉 따라서 타이핑 해보는 것을 권장합니다.

이번 장을 비롯해서 이후 장에서 코드를 설명할 때는 통일성을 위해 구글 코랩에서 실행하는 것을 기준으로 하겠습니다.

3.1 Hello World

텐서플로는 C++와 파이썬 언어를 기반으로 합니다. 다른 프로그래밍 언어도 사용할 수 있지만 일반적으로는 파이썬이 가장 많이 쓰이고 구글 코랩도 파이썬을 기본적으로 지원하기 때문에 이 책에서는 파이썬에서 실행되는 예제를 공부할 것입니다.

Hello World는 화면에 "Hello, World!"를 출력하는 프로그램입니다. 많은 프로그래밍 책에서 처음 소개하는 예제로, 원하는 내용을 화면에 출력할 수 있는지 테스트하는 간단한 연습용 프로그램입니다.

예제 3.1 Hello World 프로그램

[IN]

```
# 3.1 Hello World 프로그램
print("Hello, World!")
```

[OUT]

```
Hello, World!
```

위의 코드를 확인하기 위해서는 먼저 구글 코랩에 접속해야 합니다. 이 페이지[1]로 들어가면 해당 코드를 확인할 수 있습니다. 또한 깃허브 저장소[2]에서도 확인할 수 있습니다.

구글 코랩에 접속한 뒤 위의 코드를 실행하기 위해서는 먼저 가장 첫 번째 셀[3]을 클릭해서 커서를 위치시킵니다.

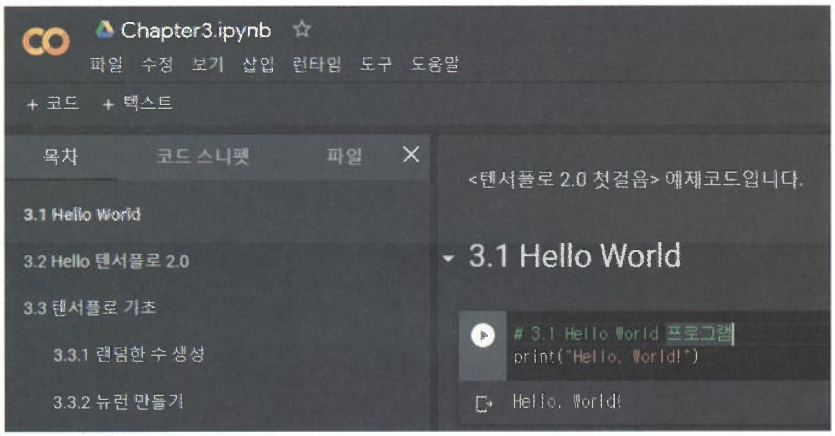


그림 3.1 구글 코랩에서의 실행 화면

그다음 셀의 좌측에 생긴 플레이 버튼을 누르거나, 단축키로 Ctrl + Enter를 누르면 셀이 실행됩니다.

그림 3.2 구글 코랩에서 첫 번째 셀을 실행한 모습. 계산 중임을 나타내기 위한 애니메이션이 플레이 버튼 주위에 표시됩니다.

1 http://bit.ly/2XEvZwI

2 http://bit.ly/2EVMxUG

3 코드나 마크다운 텍스트를 분리해서 저장할 수 있는 코랩의 기본 단위입니다. 코드의 경우 출력 결과가 함께 저장됩니다.

파이썬을 처음 접하는 분들을 위해 위 코드에 대해 간단히 설명하자면 첫 줄의 #는 다음에 나오는 내용이 주석으로 처리됨을 의미합니다. 즉, 어떤 내용을 적더라도 # 다음의 내용은 실행되지 않습니다. #을 지우면 아래와 같이 에러가 발생할 것입니다.

예제 3.2 Hello World 프로그램에서 첫 줄의 #를 지웠을 때 에러가 나는 모습

[IN]

```
3.1 Hello World
print("Hello, World!")
```

[OUT]

```
File "<ipython-input-2-8c12e2a2b3f1>", line 1
    3.1 Hello World
          ^
SyntaxError: invalid syntax
```

주석은 프로그램의 일부로 실행되지는 않지만 코드를 읽는 사람이 알아두면 좋은 유용한 정보를 적어놓을 때 필요합니다. 여기서는 이 예제의 이름이 '3.1 Hello World'라는 것을 알려주기 위해 주석을 사용했습니다. 이 책의 예제 코드에는 독자 여러분의 이해를 돕기 위한 주석이 많이 포함될 것입니다.

코드 셀 밑의 부분은 이 명령의 내용을 실행했을 때의 출력 결과를 나타냅니다. 이 책에서는 출력 결과를 생략하는 경우도 있겠지만 가급적 이해를 돕기 위해 출력 결과를 코드 본문과 함께 표시하겠습니다.

명령을 실행하는 방법은 명령을 입력한 셀에 커서가 위치한 상태에서 단축키 Ctrl + Enter를 누르거나, 구글 코랩의 상단 메뉴에서 [런타임] → [초점이 맞춰진 셀 실행]을 차례로 선택하면 됩니다.

예제 3.2의 두 번째 줄에 있는 **print**는 print 뒤에 나오는 괄호 안의 내용을 화면에 출력하는 함수입니다. 여기서는 "Hello, World!"를 화면에 출력하고 있습니다. 큰따옴표("")는 문자열 데이터를 감싸는 기호로서 대부분의 프로그래밍 언어에서 사용되는 기호입니다. 파이썬에서는 작은따옴표('')도 문자열 데이터를 감싸는 용도로 쓸 수 있습니다. 따라서 예제 3.3도 위와 동일하게 동작합니다.

예제 3.3 Hello World 프로그램의 문자열에 사용된 큰따옴표(")를 작은따옴표(')로 바꿈

[IN]

```
print('Hello, World!')
```

[OUT]

```
Hello, World!
```

프로그래밍 언어나 파이썬 언어를 처음 접하는 분들, 특히 앞에서 나온 '주석', '함수', '문자열', 'print' 등의 용어나 개념이 익숙하지 않으신 분들은 참고할 만한 인터넷 자료나 강의, 책이 많으므로 이 책을 보시기 전에 먼저 참고하시는 것이 좋습니다.

개인적으로 많은 도움이 됐던 파이썬 온라인 강의는 코세라(Coursera)의 《모두를 위한 파이썬(Python for Everyboby)》[4], 한글로 된 교재는 위키독스(wikidocs)의 《점프 투 파이썬》[5] 등이 있습니다. 파이썬 기본 문법은 텐서플로를 학습할 때 반드시 알아둬야 할 지식입니다. 더욱이 텐서플로 2.0으로 바뀌면서 기존 버전의 텐서플로에서 만들어서 쓰던 연산자나 `tf.session()`의 사용을 자제하고 데코레이터 같은 파이썬 문법을 사용하고 있기 때문에 파이썬 문법의 기초를 다져 놓으면 텐서플로 2.0을 학습하는 데도 도움이 될 것입니다. 또한 이러한 기초를 다지는 것은 프로그래밍 공부에서 중요한 일입니다. 처음에는 먼 길을 돌아가는 것 같지만 기초 개념을 잘 이해하고 그것을 바탕으로 확장된 개념을 이해하게 된다면 노력에 대한 보답을 충분히 받을 수 있을 것입니다.

3.2 Hello 텐서플로 2.0

이제 텐서플로에서도 Hello World 같은 예제를 만들어보려고 합니다. `print` 명령어가 화면에 원하는 내용을 출력하는 것이라면 텐서플로를 사용할 때도 가장 먼저 궁금해해야 하고 꼭 알아야 하기 때문에 화면에 출력해봐야 하는 정보가 있습니다. 바로 **버전 정보**입니다.

텐서플로는 깃허브에 공개된 오픈소스 라이브러리로서 약 75,000개의 포크(fork)가 만들어지고, 2,100여 명의 코드 기여자가 있으며, 거의 매 주마다 마이너 버전이 업데이트됩니다. 라이브러리의 버전 정보가 다르면 잘 돌아가야 하는 코드에서 에러가 발생할 수도 있고 책을 사 놓고 예제 코드를 원활하게 실행하지 못할 수도 있습니다. 다행히 이 책에서는 구글 코랩을 사용하기 때문에 그럴 가능성은 적습니다.

2019년 11월 현재, 구글 코랩에서 텐서플로를 그냥 불러오면 1.15.0 버전을 불러오지만, 간단한 명령으로 1.15.0 버전 대신 2.0 버전을 선택할 수 있습니다.

4 https://www.coursera.org/specializations/python

5 https://wikidocs.net/book/1

예제 3.4 텐서플로 2.0 버전 선택

[IN]

```
try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
except Exception:
    pass
import tensorflow as tf
```

[OUT]

```
TensorFlow 2.x selected.
```

앞에 퍼센트 기호(%)가 붙은 명령은 매직 커맨드라고 합니다. 이것은 구글 코랩의 원조격인 인터랙티브 파이썬(IPython)부터 지원하는 명령으로, 여러 가지 기능을 미리 정의해놓아서 편리하게 사용할 수 있게 해주는 역할을 합니다. %tensorflow_version은 이름 그대로 텐서플로의 버전을 선택하는 기능으로, 구글 코랩에서만 지원됩니다. 이 뒤에 2.x를 붙여서 텐서플로 2.0 버전을 선택합니다.

예제 3.4를 실행하면 텐서플로를 메모리로 불러올 수 있습니다. 이 과정을 임포트(import)라고 합니다. 그리고 이렇게 불러올 수 있는 미리 저장된 프로그램의 집합을 파이썬에서는 모듈(module)이라고 합니다.

예제 3.5 텐서플로 불러오기, 버전 확인

[IN]

```
import tensorflow as tf
print(tf.__version__)
```

[OUT]

```
2.0.0
```

import tensorflow 다음에 오는 as는 긴 모듈 이름을 줄이는 명령어입니다. tensorflow를 매번 입력하기에는 이름이 길기 때문에 as 뒤에 오는 tf로 줄여서 사용할 수 있게 됩니다.

다음에 오는 print는 앞에서 살펴본 것처럼 괄호 안에 지정한 내용을 화면에 출력하는 함수입니다. tf.__version__에는 텐서플로의 버전 정보가 들어있습니다. 여기서 주의할 점은 언더바(_)가 처음과 끝에 2개

씩, 총 4개가 붙는다는 점입니다. 출력 결과를 토대로 2.0.0 버전이 설치된 것을 확인할 수 있습니다. 여기서 실수하기 쉬운 점 중 하나는 tensorflow를 tf로 줄였기 때문에 원래 이름인 tensorflow를 사용해서 print(tensorflow.__version__)로 실행해보면 에러가 발생한다는 것입니다.

예제 3.6 텐서플로 불러오기, 버전 확인 에러

[IN]

```
import tensorflow as tf
print(tensorflow.__version__)
```

[OUT]

```
NameError                                 Traceback (most recent call last)
<ipython-input-5-35b726895828> in <module>()
      1 import tensorflow as tf
-----> 2 print(tensorflow.__version__)

NameError: name 'tensorflow' is not defined
```

이렇게 해서 텐서플로의 버전을 확인하는 부분까지 살펴봤습니다. 이제는 텐서플로에서 할 수 있는 기초적인 연산을 배우고, 간단한 신경망 레이어와 네트워크를 만들어 보겠습니다.

3.3 텐서플로 기초

인공지능은 현재 과학계에서 가장 각광받는 분야지만 늘 지금처럼 좋은 시절만 있던 것은 아니었습니다. '인공지능의 겨울(AI winter)'이라고 불리는 시기가 두 차례가 있었는데, 그중 첫 번째는 신경망을 구성하는 뉴런의 원조격인 퍼셉트론(perceptron)의 한계를 지적한 같은 이름의 책, 《퍼셉트론》의 발간이 어느 정도 영향을 끼쳤습니다. 퍼셉트론의 한계를 지적하는 데 사용됐던 AND, OR, XOR 연산을 할 수 있는 신경망 네트워크를 함께 만들어보며 이를 구체적으로 알아보겠습니다.

3.3.1 난수 생성

먼저 난수(Random Number)를 만들어보겠습니다. 랜덤은 신경망에서 꼭 필요한 기능입니다.

신경망을 쉽게 정의해보면 많은 숫자로 구성된 행렬이라고 할 수 있습니다. 이 행렬에 어떤 입력을 넣으면 출력을 얻게 되고, 잘 작동할 경우 원하는 출력에 점점 가까워지게 됩니다.

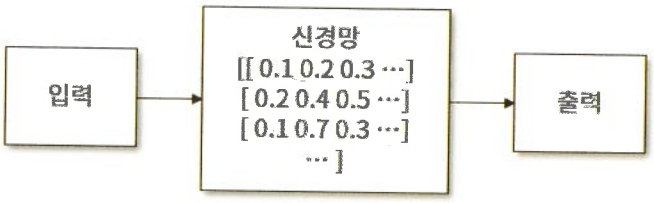


그림 3.3 많은 숫자로 구성된 행렬인 신경망

그런데 여기서 행렬을 구성하는 숫자는 어떻게 구할 수 있을까요? 처음에는 이 숫자들을 랜덤한 값으로 지정해 줄 수밖에 없습니다. 처음에 신경망의 초깃값을 지정해주는 것을 초기화(Initialization)라고 하며, 이에 관련해서도 많은 논문들이 나와 있습니다. 현재 가장 많이 쓰이는 방법은 Xavier 초기화(Xavier Initialization)와 He 초기화(He Initialization)인데, 이 방법들은 랜덤하지만 어느 정도 규칙성이 있는 범위 내에서 난수를 지정합니다. 여기서는 이 방법들보다 단순하게 난수를 지정해보겠습니다.

텐서플로에서 난수를 생성하는 방법은 아래와 같습니다.

예제 3.7 난수 얻기(균일 분포)

[IN]

```
rand = tf.random.uniform([1],0,1)
print(rand)
```

[OUT]

```
tf.Tensor([0.4458922], shape=(1,), dtype=float32)
```

`tf.random.uniform` 함수를 불러오면 균일 분포(uniform distribution)의 난수를 얻을 수 있습니다. 균일 분포란 최솟값과 최댓값 사이의 모든 수가 나올 확률이 동일한 분포에서 수를 뽑는다는 뜻입니다. 처음에 나오는 [1]은 결괏값의 Shape을 의미합니다. Shape이란 행렬을 구성하는 행, 열 등 차원의 수를 나타내는 값입니다.

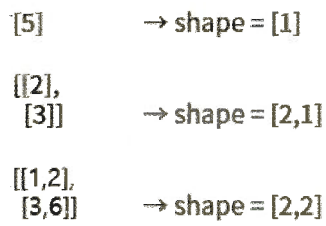


그림 3.4 행렬 Shape의 예

Shape을 확인하려면 print 명령어를 써도 되지만 간단한 데이터의 경우에는 직접 셀 수도 있습니다. 세는 방법은 바깥쪽부터 안쪽으로 원소의 개수를 세는 것입니다. 앞의 [[2],[3]]의 경우 가장 바깥쪽은 원소가 [2], [3]으로 2개이고, 안쪽으로 들어가면 2, 3이 각각 하나씩 있기 때문에 shape=[2,1]이 됩니다.[6]

tf.random.uniform의 두 번째와 세 번째 인수인 0, 1은 각각 최솟값과 최댓값을 의미합니다. 즉, 여기서는 최솟값 0과 최댓값 1 사이에서 모든 수가 나올 확률이 동일한 분포에서 난수 하나를 뽑는 것입니다. 여기서는 tf.Tensor 바로 다음에 나오는 대괄호([]) 안의 0.4458922라는 값을 얻었습니다.

Shape은 1인 것을 확인할 수 있습니다. 파이썬에서는 변하지 않는 값에 대해 대괄호 대신 소괄호(())로 묶는 튜플(tuple) 자료형을 사용합니다. 이때 원소의 개수가 한 개인 경우에도 앞에서처럼 (1,)로 원소 뒤에 쉼표를 써줘서 이 값이 튜플이라는 것을 나타냅니다. 이 값은 입력할 때의 [1]과 동일한 값입니다.

마지막의 dtype은 자료형(data type)을 의미합니다. float32는 32비트의 부동소수점 수[7]라는 뜻입니다. 텐서플로 2.0에는 16비트 부동소수점 수인 float16과 64비트 부동소수점 수인 float64도 존재합니다. 특별한 설정이 없다면 앞에서 확인할 수 있듯이 텐서플로는 float32 값을 tf.random.uniform의 출력으로 반환합니다.

Shape을 바꿔서 여러 개의 난수를 얻을 수도 있습니다. Shape을 [4]로 바꾸면 다음과 같은 결과를 얻습니다.

6 만약 [[2], [3,1]]처럼 특정 단계에서 원소의 개수가 다를 경우에는 shape을 특정할 수 없습니다.

7 부동소수점이라는 말은 소수점이 떠다닌다는 말로, 제한된 비트 안에서 다양한 수를 표현하기 위해 소수점이 움직이는 것입니다. 비트 수가 커질수록 표현할 수 있는 숫자의 범위는 넓어지지만 메모리를 많이 차지하고 계산 속도가 느린 단점도 있습니다. 부동소수점 수는 제한된 비트에서 연속적인 실수를 표현해야 하기 때문에 정확한 수치가 아닌 근사치를 갖습니다.

예제 3.8 난수 여러 개 얻기(균일 분포)

[IN]

```
rand = tf.random.uniform([4],0,1)
print(rand)
```

[OUT]

```
tf.Tensor([0.4000026  0.02589869 0.97795093 0.37491727], shape=(4,), dtype=float32)
```

균일 분포 외에 난수를 얻는 방법 중 중요한 분포로는 정규(normal) 분포가 있습니다. 정규 분포는 가운데가 높고 양극단으로 갈수록 낮아져서 종 모양을 그리는 분포입니다. 정규 분포의 난수를 구하기 위해서는 `tf.random.uniform`을 `tf.random.normal`로 바꾸기만 하면 됩니다.

예제 3.9 난수 얻기(정규 분포)

[IN]

```
rand = tf.random.normal([4],0,1)
print(rand)
```

[OUT]

```
tf.Tensor([ 0.846745   1.7888565  0.8378075 -0.34830892], shape=(4,), dtype=float32)
```

그런데 여기서는 1 이상의 값도 나오고 음수도 나오고 있습니다. `tf.random.normal`의 두 번째와 세 번째 인수는 앞에 나온 `tf.random.uniform`과 다릅니다. 여기서 두 번째의 0은 정규 분포의 평균(mean), 세 번째의 1은 정규 분포의 표준편차(standard deviation)를 의미합니다. 평균이 0이고 표준편차가 1일 때의 정규 분포를 표준정규분포(standard normal distribution)라고 합니다. 두 번째, 세 번째 인수가 0, 1로 같을 때 균일 분포와 정규 분포의 차이를 그래프로 나타내면 다음 페이지의 그림 3.5와 같습니다.

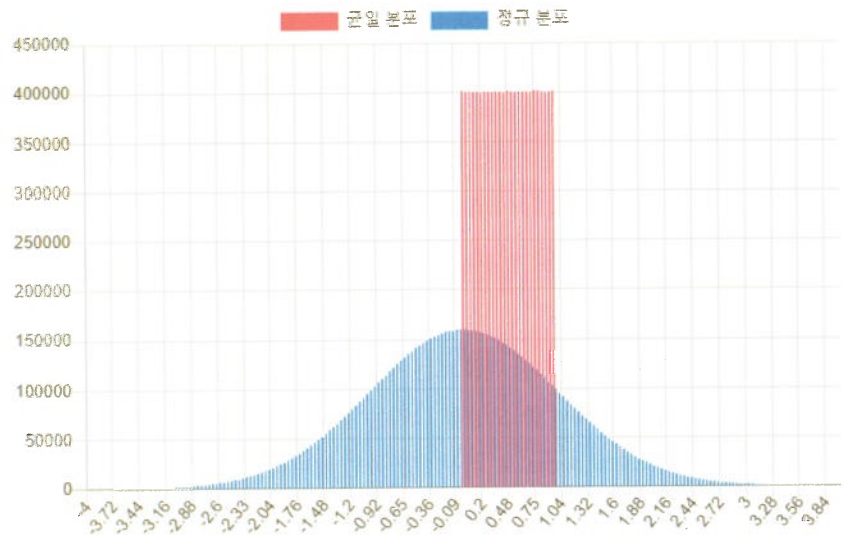


그림 3.5 균일 분포와 정규 분포의 차이

균일 분포와 정규 분포에 대해 천만 개의 샘플을 각각 구해서 히스토그램으로 나타내면 균일 분포는 최솟값과 최댓값 사이에서 일정한 확률의 난수가 나타나게 되지만 정규 분포는 주어진 평균과 표준편차를 만족하는 종형 곡선(bell curve)을 그리게 됩니다. 정규 분포는 학생들의 시험 성적이나 키, 농작물의 무게 등 실생활에서 자주 볼 수 있는 분포 형태입니다. 앞에서 소개한 Xavier 초기화나 He 초기화는 균일 분포와 정규 분포 중 하나를 택해서 신경망의 초깃값을 초기화합니다.

3.3.2 뉴런 만들기

이제 신경망의 가장 기본적인 구성요소인 뉴런을 만들어보겠습니다. 이전에는 뉴런을 퍼셉트론이라고도 불렀으며, 입력을 받아서 계산 후 출력을 반환하는 단순한 구조입니다(그림 3.6).

그림 3.6 추상화한 뉴런의 구조

사실 신경망은 뉴런이 여러 개 모여 레이어(layer)를 구성한 후, 이 레이어가 다시 모여 구성된 형태입니다. 참고로 뉴런과 레이어를 우리말로 각각 '신경 세포'와 '층'이라고 할 수도 있겠지만 이 책에서는 의미상의 혼란을 막기 위해 뉴런, 레이어라고 표기하겠습니다.

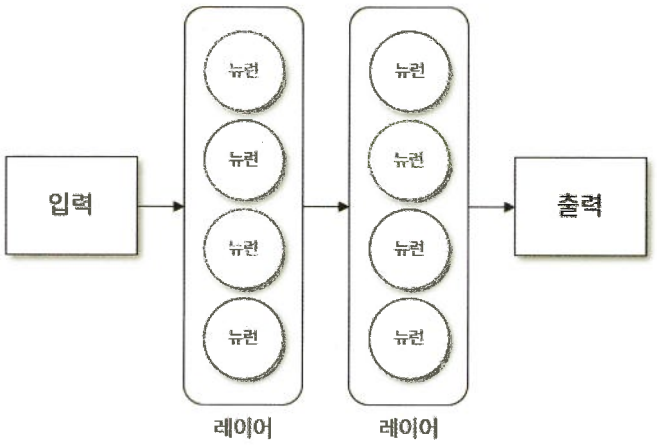


그림 3.7 뉴런과 레이어로 구체화한 신경망의 구조

뉴런은 입력, 가중치, 활성화함수, 출력으로 구성됩니다.

그림 3.8 뉴런의 구성요소

입력, 가중치, 출력은 보통 정수(integer)나 앞에서 살펴본 `float`이 많이 쓰입니다. 활성화함수는 뉴런의 출력값을 정하는 함수입니다. 가장 간단한 형태의 뉴런은 입력에 가중치를 곱한 뒤(행렬의 곱셈입니다) 활성화함수를 취하면 출력을 얻을 수 있습니다.

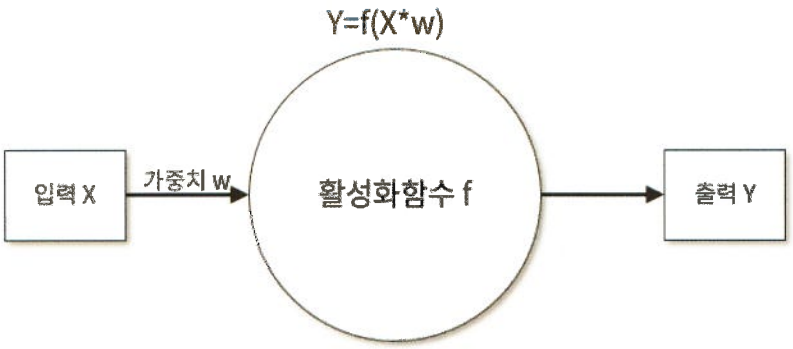


그림 3.9 뉴런의 출력 계산식

뉴런에서 학습할 때 변하는 것은 가중치입니다. 가중치는 처음에는 초기화를 통해 랜덤한 값을 넣고, 학습 과정에서 점차 일정한 값으로 수렴합니다. 학습이 잘 된다는 것은 좋은 가중치를 얻어서 원하는 출력에 점점 가까운 값을 얻는 것이라고 할 수 있습니다.

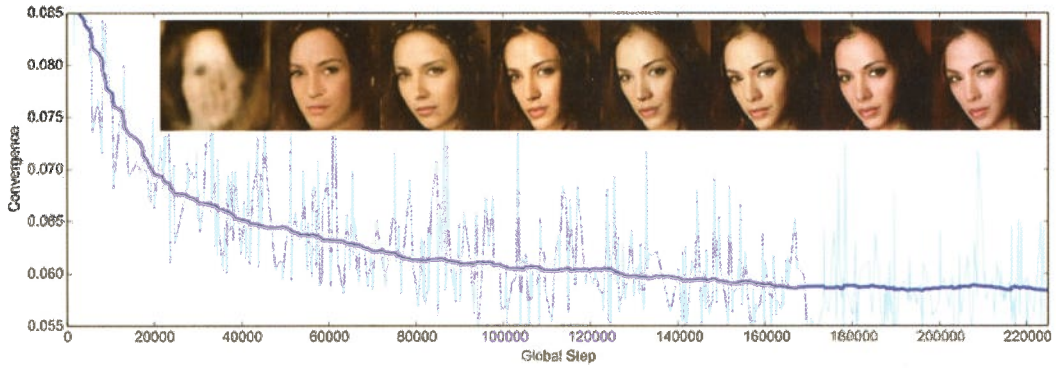


그림 3.10 딥러닝 학습을 이용한 가상 인물의 생성 사례[8]

활성화함수로는 시그모이드(sigmoid), ReLU 등을 주로 쓰게 됩니다. 시그모이드는 S자 형태의 곡선이라는 뜻입니다. ReLU는 Rectified Linear Unit의 약자로, 정류된(rectified) 선형 함수(linear unit)라는 뜻입니다. 딥러닝에서 선형 함수는 $y=x$라는 식으로 정의할 수 있는 입력과 출력이 동일한 함수를 의미합니다. 이 함수를 정류해서 음수 값을 0으로 만든 것이 ReLU입니다. 그래프로 나타내면 다음과 같습니다.

8 출처: BEGAN 깃허브 저장소(https://github.com/artcg/BEGAN)

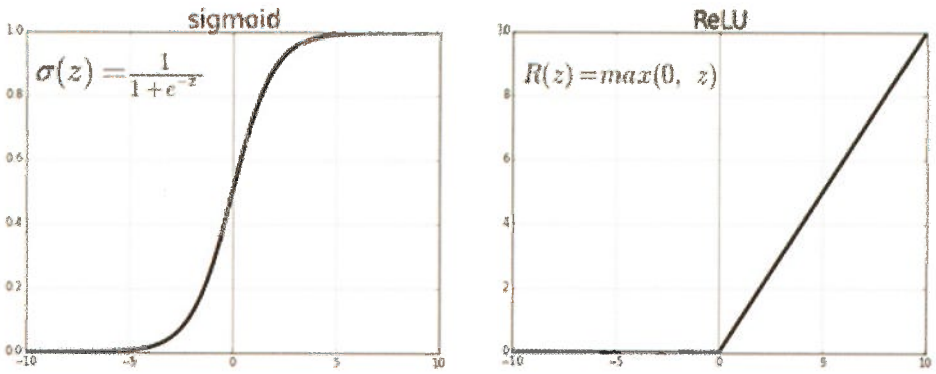


그림 3.11 시그모이드와 ReLU 함수의 그래프

신경망 초창기에는 시그모이드가 주로 쓰였지만 은닉층[9]을 다수 사용하는 딥러닝 시대가 되면서 ReLU가 더 많이 쓰이고 있습니다. 딥러닝에서 오류를 역전파(backpropagate)할 때 시그모이드 함수가 값을 점점 작아지게 하는 문제를 토론토 대학교의 비노드 네어(Vinod Nair)와 제프리 힌튼(Geoffrey Hinton) 교수가 지적하며, ReLU를 대안으로 제시한 2010년 논문[10]에 이에 대한 자세한 내용이 나와 있습니다. 앞의 그래프에서도 시그모이드는 출력값을 0~1 사이로만 제한하게 되지만 ReLU는 양수를 그대로 반환하기 때문에 값의 왜곡이 적어집니다.

여기서는 시그모이드 함수를 활성화함수로 사용해보겠습니다. 출력을 0~1 사이로 제한해서 뒤에서 살펴볼 AND, OR, XOR 연산을 다루는 데 적합하기 때문입니다.

먼저 시그모이드 함수를 파이썬으로 구현해 보겠습니다.

예제 3.10 시그모이드 함수

```
import math
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
```

9 신경망에서 입력층과 출력층을 제외한 나머지 부분을 은닉층이라고 합니다. 은닉층은 가중치와 활성화함수로 구성되며, 입력에 대한 알맞은 출력을 학습하는 신경망의 핵심 구성요소입니다. 정확한 기준은 없지만 보통 은닉층을 3개 이상 사용하면 딥러닝으로 간주합니다.

10 V. Nair and G. E. Hinton, Rectified linear units improve restricted boltzmann machines, In Proc. 27th International Conference on Machine Learning, 2010.

첫 줄에서는 앞에서 tensorflow 모듈을 불러왔던 것처럼 math 모듈을 불러옵니다. math 모듈은 상수 e의 제곱을 계산하기 위한 math.exp()를 사용하는 데 필요합니다.

두 번째 줄에서 정의한 sigmoid() 함수는 x를 입력으로 받고 return 명령을 통해 출력값을 반환합니다. 여기서 반환하는 1/(1+math.exp(-x))가 x에 시그모이드 함수를 취한 값이 됩니다.

그럼 이제 뉴런의 입력과 출력을 정의해 보겠습니다. 입력이 1일 때 기대출력[11]이 0이 되는 뉴런을 만든다면 어떨까요?

예제 3.11 뉴런의 입력과 출력 정의

[IN]

```
x = 1
y = 0
w = tf.random.normal([1],0,1)
output = sigmoid(x * w)
print(output)
```

[OUT]

```
0.43007410854723105
```

입력인 x에는 1을 넣고 가중치는 w로 정규 분포의 랜덤한 값을 넣습니다. 실제출력인 output은 sigmoid() 함수에 입력과 출력을 곱한 값을 넣어서 계산합니다.

실제출력으로 나온 0.43007과 기대출력인 0의 차이인 $0 - 0.43007 = -0.43007$을 에러(error)라고 합니다. 뉴런의 학습은 이 에러가 0에 가까워지게 해서 출력으로 기댓값에 가까운 값을 얻는 것입니다.

여기서 뉴런이란 결국 w 값입니다. 이제 이 w를 변화시켜야 하는데, 어떤 알고리즘을 사용해야 할까요? 경사 하강법(Gradient Descent)이라는 방법을 사용하면 되는데, 이것은 w에 입력과 학습률(α)과 에러를 곱한 값을 더해주는 것입니다.[12] 학습률은 w를 업데이트하는 정도로, 큰 값으로 설정하면 학습이 빨리 되지만 과도한 학습으로 적정한 수치를 벗어날 우려가 있고, 너무 작은 값으로 설정하면 학습 속도가 너무 느려질 수 있습니다. 여기서는 α=0.1로 설정하겠습니다.

11 기대출력이란 뉴런이 출력하기를 기대하는 값으로 정답에 해당합니다. 실제출력은 뉴런이 실제로 출력하는 값입니다. 학습을 시작하기 전에 기대출력과 실제출력이 일치하는 경우는 거의 없습니다. 뉴런의 학습은 실제출력이 기대출력에 가까워지도록 가중치를 조정하는 과정입니다.

12 여기서 경사는 손실 곡선의 기울기를 뜻합니다. 경사 하강법은 손실 곡선을 미분한 다음, 그 값을 이용해서 가중치가 손실이 가장 낮아지는 지점에 도달하도록 반복적으로 계산합니다. 더 자세한 내용은 구글 머신러닝 단기집중과정의 《손실 줄이기: 경사하강법》에서 찾아볼 수 있습니다. http://bit.ly/2YpPJ41

$$w = w + x \times \alpha \times error$$

경사 하강법이 효과를 발휘하는지 코드로 확인해보겠습니다.

예제 3.12 경사 하강법을 이용한 뉴런의 학습

[IN]

```
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 -0.07284613375703874 0.07284613375703874
199 -0.02218122340039822 0.02218122340039822
299 -0.011637067688397104 0.011637067688397104
399 -0.007558882907100903 0.007558882907100903
499 -0.005479490244093222 0.005479490244093222
599 -0.004243807125866066 0.004243807125866066
699 -0.003434699043384763 0.003434699043384763
799 -0.0028683053134466396 0.0028683053134466396
899 -0.002451979253421888 0.002451979253421888
999 -0.0021343669481190732 0.0021343669481190732
```

for는 반복문을 나타내는 명령어입니다. i는 반복문에서 사용할 변수이고, range(1000)은 [0, 1, 2, 3, …, 998, 999] 형태의 리스트를 자동으로 생성하는 명령어입니다. 정리하면 이 반복문은 i=0부터 i=999까지 1,000번 동안 for 문 안의 내용을 반복하게 됩니다.

출력을 구하는 부분은 위와 동일하고 error를 구하는 식은 기대출력인 y에서 실제출력인 output을 뺍니다. 이렇게 구한 error를 경사 하강법의 업데이트 식에 넣어 w 값을 다시 계산합니다. if 문은 프로그램을 제어하는 명령어로, if 다음에 나오는 부분이 참(True) 값일 때 블록 안의 내용을 실행하는 것입니다. %는 나머지를 구하는 연산자입니다. i=99, i=199, … i=999일 때 if 문 안의 블록이 실행되어 i, error, output 값을 출력합니다. error 값은 0에 점점 가까워지고, output도 기대출력인 0에 가까워지는 것을 확인할 수 있습니다.

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
```

error가 변하지 않고, 출력도 0.5에서 변하지 않습니다. 왜 이런 일이 발생하는 것일까요? 우리가 입력으로 넣은 수는 0입니다. 경사 하강법의 업데이트 식은 $w = w + x \times \alpha \times error$입니다. x=0이기 때문에 w에 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 w처럼 난수로 초기화되며 뉴런에 더해져서 출력을 계산하게 됩니다. 수식에서는 관용적으로 bias의 앞 글자인 b를 씁니다.

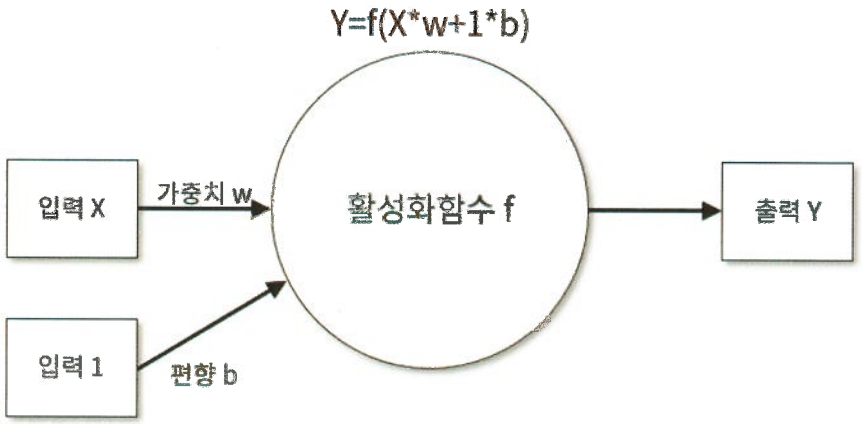


그림 3.12 편향이 더해진 뉴런의 출력 계산식

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)
```

[OUT]

```
99 0.06947038207268275 0.9305296179273173
199 0.04186835006367673 0.9581316499363233
299 0.02980703874277979 0.9701929612572202
399 0.023094765971415576 0.9769052340285844
499 0.018831817065279144 0.9811681829347209
599 0.015888868015282664 0.9841111319847173
699 0.013736879882487618 0.9862631201175124
799 0.012095678439407509 0.9879043215605925
899 0.010803125392571333 0.9891968746074287
999 0.009759094891937603 0.9902409051080624
```

편향을 나타내는 b가 네 번째 줄에 추가됐고, w처럼 `tf.random.normal`로 초기화합니다. 실제출력인 `output` 값을 계산할 때 `sigmoid(x * w + 1 * b)`라는 수식을 계산해서 각 입력에 가중치와 편향을 곱해서 더해준 뒤 시그모이드 함수를 취합니다. 기대출력과 실제출력의 차이인 `error`로 w와 b를 각각 업데이트해서 뉴런을 학습시킵니다.

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

3.3.3 첫 번째 신경망 네트워크: AND

이제 좀 더 의미 있는 일을 하는 신경망 네트워크를 만들어보겠습니다. 이 네트워크의 구성요소는 앞에서 본 것과 같은 하나의 뉴런입니다. 여기서는 AND 연산을 수행하는 뉴런을 만들겠습니다. AND 연산은 여러 개의 입력을 받을 수 있지만, 여기서는 2개의 입력만으로 제한하겠습니다. 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

AND 연산은 입력이 모두 참 값일 때 참이 되고, 그 밖의 경우에는 모두 거짓이 됩니다. 파이썬에서 참, 거짓을 나타내는 값은 각각 True, False입니다. 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.

예제 3.15 True, False의 정숫값 확인

[IN]

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

[OUT]

```
1
0
```

정수로 변환하는 int 함수를 사용하면 True와 False가 각각 1과 0이 되는 것을 알 수 있습니다. 사실 프로그래밍 언어에서는 관습적으로 거짓을 0으로 표시하고, 참을 0이 아닌 다른 값으로 표시해 왔습니다.[13] **if 1:**이라는 명령이 **if True:**와 동일하게 동작하는 것과 같은 이치입니다. 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

13 예외적으로 프로그래밍 언어인 Ruby에서는 0도 참이고, false와 nil이 거짓입니다.

이제 숫자로 된 네 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.

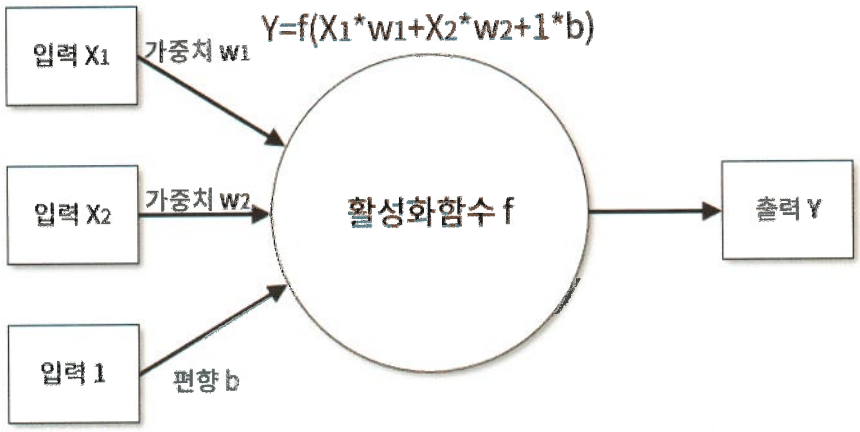


그림 3.13 입력이 2개, 편향이 1개인 뉴런의 출력 계산식

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

[IN]

```
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)
```

[OUT]

```
199 -0.10302681614107792
399 -0.06306310495729046
599 -0.04530885125417762
799 -0.03525784399909018
999 -0.028805892304839046
1199 -0.024323379694299915
1399 -0.021032284593734747
1599 -0.018516145609933733
1799 -0.01653245648377979
1999 -0.014928035388999245
```

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 array[14]로 정의했습니다. 넘파이 array는 아주 유용한 자료 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 자료형의 정확한 명칭은 ndarray입니다. ndarray는 파이썬의 리스트와 비슷하지만 크기(Shape)를 미리 지정하고 사용합니다. ndarray는 파이썬의 리스트보다 메모리를 적게 차지하고 넘파이의 전용 함수를 이용해 빠르게 계산할 수 있습니다. 파이썬의 리스트와 ndarray의 차이는 이 URL(http://bit.ly/2Df1r7C)에 쉽게 정리돼 있습니다.

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]
[]
[]
```

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]

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-52-3f950372c2b1> 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)
print(np.array([1,2,3])*0.01)
```

[OUT]

```
[2 4 6]
[0 0 0]
[-1 -2 -3]
[0.01 0.02 0.03]
```

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

결괏값을 자세히 보면 쉼표(,)가 없는 것도 확인할 수 있습니다. 각 원소에 대한 연산으로 나온 결괏값은 파이썬의 리스트가 아닌 넘파이 array로 자동 변환되기 때문입니다. 넘파이 array는 출력할 때 쉼표를 표시하지 않습니다.

그럼 다시 예제 3.16에서 가중치를 업데이트하는 `w = w + x[j] * 0.1 * error`에서 `x[j]`는 `j`의 값이 `0`에서 `3`으로 변함에 따라 `[1,1]`, `[1,0]`, `[0,1]`, `[0,0]`이 됩니다. 이처럼 여러 개의 수에 실수를 직접 곱해서 한 번에 결괏값을 얻기 위해 넘파이 array를 사용하는 것입니다.

다른 부분은 지금까지 봐온 예제와 같습니다. 입력의 수가 4배로 많아졌기 때문에 네트워크가 수렴하는 데는 더 많은 연산이 필요합니다. 예제 3.16의 출력에서는 error의 합인 error_sum이 점점 줄어드는 것을 확인할 수 있습니다.

그럼 이렇게 학습시킨 네트워크가 정상적으로 작동하는지 평가해보겠습니다. 네트워크에 x의 각 값을 넣었을 때, 실제출력이 기대출력인 y 값에 얼마나 가까운지 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: 0.02469764355276747
X: [0 1] Y: [0] Output: 0.024772481083015434
X: [0 0] Y: [0] Output: 2.322580850192006e-05
```

실제출력이 기대출력에 가깝게 나오는 것을 확인할 수 있습니다. 마지막 Output의 2.322580850192006e-05는 과학적 표기법으로, 실수를 가수와 지수로 표현하는 방법입니다. e 앞에 오는 수는 가수가 되고 e 뒤에 오는 수는 지수가 됩니다. 2.322580850192006e-05는 0.00002322580850192006과 같은 수입니다.

이제 AND 연산의 각 진리표에 대해 기대출력과 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 결괏값이 참이었던 것과 다르게 OR 연산은 입력 중 하나만 참이어도 결괏값이 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 연산을 계산하는 네트워크를 생성하는 코드도 예제 3.16의 AND 네트워크와 매우 비슷합니다. 달라진 것은 y 부분의 기대출력뿐입니다.

[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)
```

[OUT]

```
199 -0.0425037323397693
399 -0.02377204660067718
599 -0.016403460122359978
799 -0.012483777088005
999 -0.01005965678340738
1199 -0.008415806216678347
1399 -0.007229549298837526
1599 -0.006333452249524744
1799 -0.005634139262710159
1999 -0.005071500290296207
```

OR 네트워크를 평가하는 코드와 결과는 다음과 같습니다.

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))
```

[OUT]

```
X: [1 1] Y: [1] Output: 0.999997331473692
X: [1 0] Y: [1] Output: 0.9899476313420319
X: [0 1] Y: [1] Output: 0.9899158841652302
X: [0 0] Y: [0] Output: 0.025148692483856868
```

학습 수(`for i in range(2000):`의 2000)를 바꿔가며 학습해보면 학습 수가 커질수록 실제출력이 기대출력에 가까워지는 것을 확인할 수 있습니다. 이렇게 OR 네트워크도 학습시켰으니 이제 인공지능의 겨울을 불러왔던 XOR 네트워크에 도전할 차례입니다.

3.3.5 세 번째 신경망 네트워크: XOR

XOR은 OR과 비슷하지만 한 가지 차이가 있습니다. 바로 홀수 개의 입력이 참일 때만 결괏값이 참이 된다는 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 -2.947558653376703e-06
799 -1.3309160462604552e-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
```

에러 값은 점점 줄어들다가 어느 순간 변하지 않습니다. 이렇게 학습시킨 네트워크를 평가해보면 결과는 다음과 같습니다.

예제 3.24 XOR 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: [0] Output: 0.5128176286712095
X: [1 0] Y: [1] Output: 0.5128176305326305
X: [0 1] Y: [1] Output: 0.4999999990686774
X: [0 0] Y: [0] Output: 0.5000000009313226
```

Y와 Output 사이에는 큰 차이가 있어 보입니다. X가 변해도 Output은 0.5 근처에서 머물고 있습니다. 어째서 이런 결과가 나오는 것일까요?

output = sigmoid(np.sum(x[j]*w)+b_x*b) 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 -7.4505806e-09], shape=(2,), dtype=float32)
b: tf.Tensor([3.7252903e-09], shape=(1,), dtype=float32)
```

w는 약 0.0512, −0.000000007450이고, b는 약 0.000000003730입니다. 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.

표 3.7 XOR 네트워크의 입력과 중간 계산, 출력

X1	X2	중간 계산 np.sum(x[j]*w)+b	출력 sigmoid(np.sum(x[j]*w)+b)
1	1	0.05128175	0.5128176286893302
1	0	0.05128176	0.5128176305507514
0	1	-3.7252903e-09	0.4999999990686774
0	0	3.7252903e-09	0.5000000009313226

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 출력해 보면 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([-10.601849], shape=(1,), dtype=float32)
```

표 3.8 AND 네트워크의 입력과 중간 계산, 출력

X1	X2	중간 계산 np.sum(x[j]*w)+b	출력 sigmoid(np.sum(x[j]*w)+b)
1	1	3.2981866	0.964366548024708
1	0	-3.6534204	0.02524838724984636
0	1	-3.650242	0.025326728701507022
0	0	-10.601849	2.4869364094058595e-05

두 네트워크의 가중치를 그래프로 나타내 보면 다음과 같습니다.

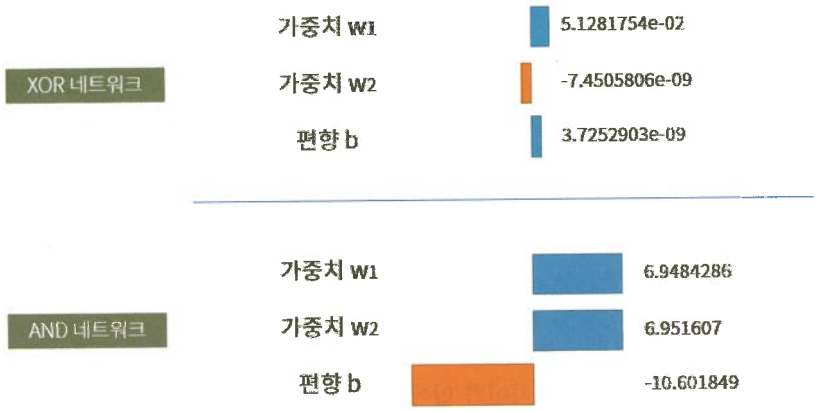


그림 3.14 XOR과 AND 네트워크의 가중치 그래프화

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 음수 편향을 이겨낼 수 있습니다.

반면 XOR 네트워크는 어떤 일을 하려는지 명확하지 않습니다. 가중치 w1이 w2에 비해 좀 더 큰 값을 가지고 있기는 하지만 중간 계산값은 0에 가까워지고 시그모이드 함수를 취한 값은 0.5에 가까워질 뿐입니다.

이것이 바로 첫 번째 인공지능의 겨울을 불러온 것으로 잘 알려진 XOR 문제(XOR Problem)입니다. 하나의 퍼셉트론으로는 간단한 XOR 연산자도 만들어낼 수 없다는 것을 동명의 책인 《퍼셉트론(Perceptron)》에서 마빈 민스키(Marvin Minsky)와 시모어 페퍼트(Seymour Papert)가 증명해냈습니다.

그럼 해결책은 무엇일까요? 바로 여러 개의 퍼셉트론을 사용하는 것입니다. 사실 《퍼셉트론》에서도 여러 개의 퍼셉트론을 사용하면 XOR 문제를 포함한 어떤 불린 함수(Boolean function, 정수를 넣었을 때 0 또는 1이 출력되는 함수)든지 풀 수 있다는 사실을 언급하고 있습니다.

여기서는 세 개의 퍼셉트론과 뉴런을 사용해 보겠습니다. 코드가 복잡해지는 것을 막기 위해 1.2.4절에서 설명한 `tf.keras`를 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 단위인만큼 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 합쳐진 단위인 레이어를 일직선으로 배치한 것입니다. 이 책에서는 외국어 표기법 그대로 시퀀셜 네트워크, 시퀀셜 모델로 부르겠습니다.

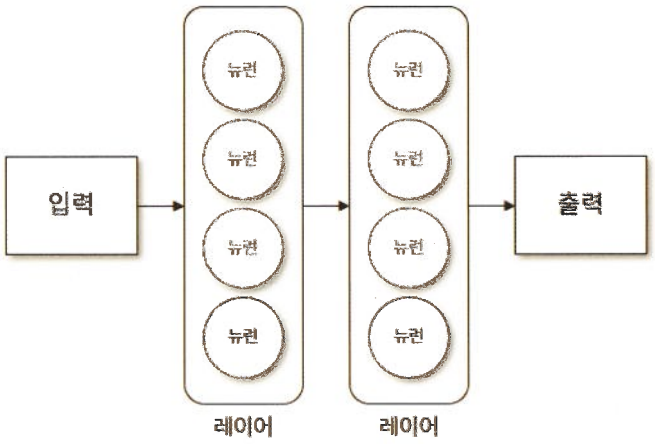


그림 3.15 tf.keras.sequential의 일직선 구조

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.Dense는 model에서 사용하는 레이어를 정의하는 명령입니다. Dense는 가장 기본적인 레이어로서, 레이어의 입력과 출력 사이에 있는 모든 뉴런이 서로 연결되는 레이어입니다.

tf.keras.layers.Dense 안의 units는 레이어를 구성하는 뉴런의 수를 정의합니다. 뉴런이 많을수록 일반적으로 레이어의 성능은 좋아지지만 계산량 또한 많아지고 메모리도 많이 차지하게 됩니다. activation은 우리가 계속 봐 온 활성화함수입니다. 여기서는 두 레이어에 모두 sigmoid를 썼습니다.

input_shape은 시퀀셜 모델의 첫 번째 레이어에서만 정의하는데, 입력의 차원 수가 어떻게 되는지를 정의합니다. 여기서는 각 데이터가 [1,1], [1,0]처럼 2개의 입력을 받는 1차원 array이기 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.

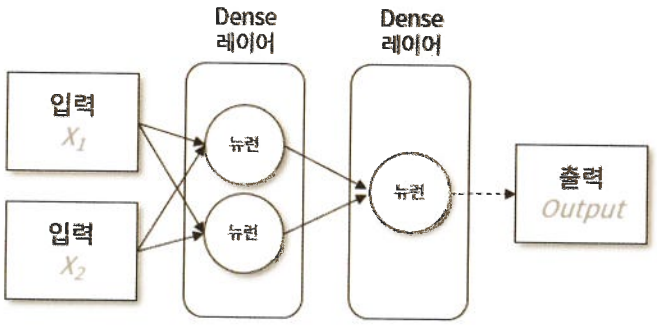


그림 3.16 2-레이어 XOR 네트워크의 구조

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 그리면 다음과 같습니다.

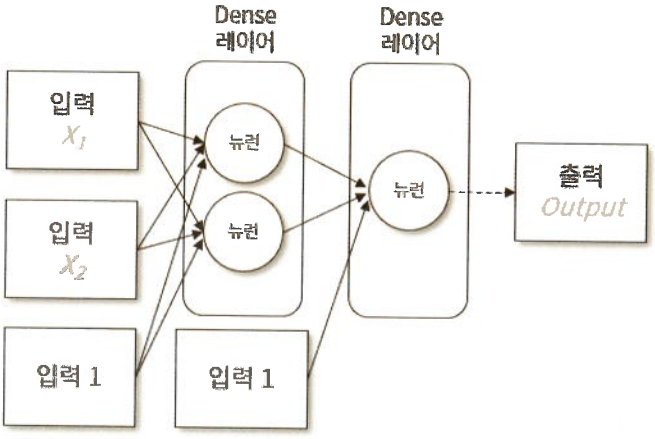


그림 3.17 편향을 포함한 2-레이어 XOR 네트워크의 구조

보통 Dense 레이어의 파라미터 수는 (입력측 뉴런의 수+1) × (출력측 뉴런의 수)의 식으로 구할 수 있습니다. 여기서 입력측, 출력측이란 Dense 레이어에 들어오는 입력을 입력측, Dense 레이어의 뉴런을 출력측이라고 합니다. 이 식에 따르면 첫 번째 레이어의 파라미터 수는 (2+1) × 2 = 6이고, 두 번째 레이어의 파라미터 수는 (2+1) × 1 = 3으로 [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(lr=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 업데이트하겠다는 뜻이고, 확률적(Stochastic)은 전체를 한번에 계산하지 않고 확률적으로 일부 샘플을 구해서 조금씩 나눠서 계산하겠다는 뜻입니다. 자세한 내용이 궁금하신 분들은 구글 머신러닝 단기집중과정의 "손실 줄이기"[15] 부분을 참고하면 좋습니다.

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

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

앞의 예제들에서 썼던 에러 식 error = y - output과 비슷한 기능을 합니다.

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

```
model.summary()
```

15 http://bit.ly/2PlcKkC

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: 0.2757
Epoch 2/2000
4/4 [==============================] - 0s 2ms/sample - loss: 0.2718
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 에포크(epochs)[16]에 지정된 횟수만큼 학습시킵니다. 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],
       [0.9555097 ],
       [0.9552847 ],
       [0.03977039]], dtype=float32)
```

16 에포크(Epoch)는 훈련 데이터를 반복 학습시키는 횟수입니다. 배치 크기(batch size)는 각 에포크에 학습시키는 훈련 데이터의 수입니다.

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

어떻게 이런 결과가 나오는 걸까요? model을 구성하고 있는 파라미터, 즉 가중치와 편향을 출력해보겠습니다.

예제 3.30 2-레이어 XOR 네트워크의 가중치와 편향 확인

[IN]

```
for weight in model.weights:
    print(weight)
```

[OUT]

```
<tf.Variable 'dense_14/kernel:0' shape=(2, 2) dtype=float32, numpy=
array([[-4.0317945, -6.0787964],
       [-4.0165396, -5.977493 ]], dtype=float32)>
<tf.Variable 'dense_14/bias:0' shape=(2,) dtype=float32, numpy=array([5.927255 , 2.2947845],
dtype=float32)>
<tf.Variable 'dense_15/kernel:0' shape=(2, 1) dtype=float32, numpy=
array([[ 7.936609],
       [-8.199594]], dtype=float32)>
<tf.Variable 'dense_15/bias:0' shape=(1,) dtype=float32, numpy=array([-3.650685],
dtype=float32)>
```

가중치 정보는 model.weights에 저장돼 있습니다. 입력과 레이어 또는 레이어 사이의 뉴런을 연결할 때 사용되는 가중치는 kernel이고, 편향과 연결된 가중치는 bias로 표시됩니다.

보통 네트워크의 가중치 숫자가 많기 때문에 구분을 위해 편의상 가중치에 첨자를 붙여서 표시합니다. 레이어의 순서대로 위첨자를 붙이고, 아래첨자는 각 뉴런의 순서에 맞게 차례로 붙입니다.

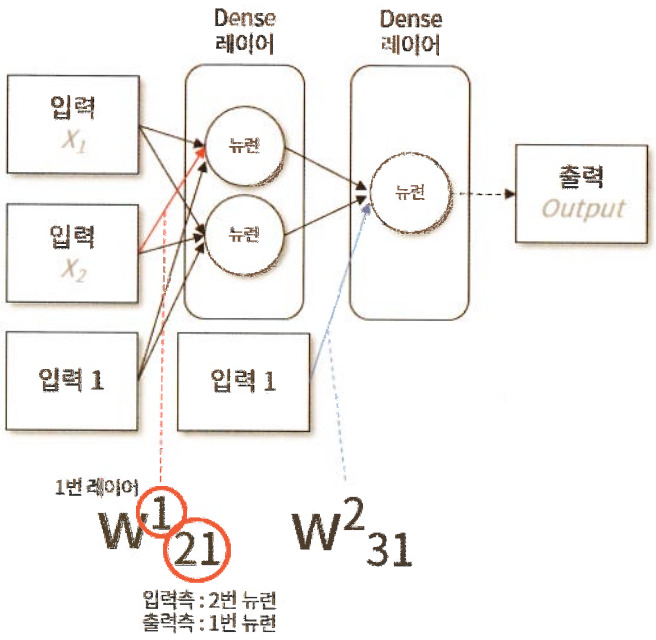


그림 3.18 가중치에 이름을 붙이는 규칙

이 규칙에 의해 이름을 붙인 가중치들을 그래프로 나타내면 다음과 같습니다.

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 들어오지 않습니다. 뉴런과 레이어가 많아지면 이 문제는 더욱 커집니다. 가중치 시각화보다 네트워크의 학습 상황을 더 잘 파악할 수 있는 방법이 필요합니다.

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 그래프 그리기가 있습니다. 이렇게 그려진 그래프는 주피터 노트북과 구글 코랩에서 즉시 확인할 수 있고 그림 파일로도 따로 저장할 수 있습니다.

3.4.1 matplotlib.pyplot을 이용한 그래프 그리기

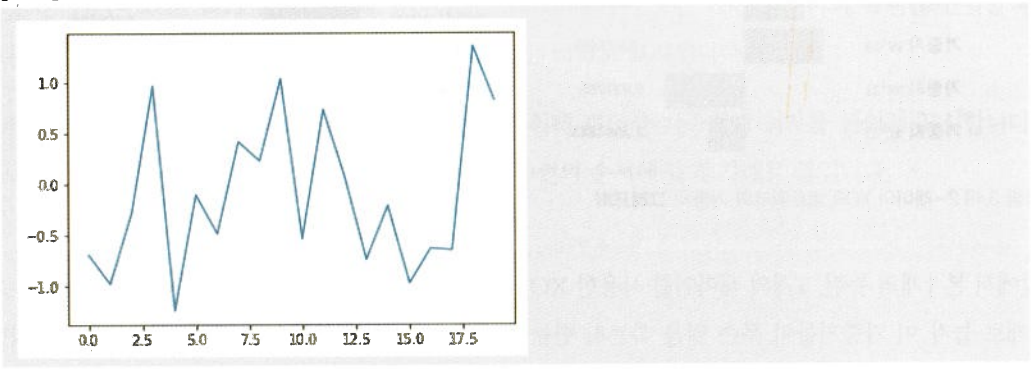
딥러닝 데이터 그래프를 그리기 전에 먼저 그래프를 그리는 함수의 사용법에 익숙해지는 시간을 갖겠습니다. 다음 예제는 간단한 꺾은선 그래프를 그립니다.

예제 3.31 간단한 꺾은선 그래프 그리기

[IN]

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

[OUT]



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

그래프를 그리기 위해서는 데이터가 필요합니다. 3.3.1절의 "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 그래프를 그리는 부분입니다. 이렇게 그려진 그래프는 plt.show()라는 함수를 호출해야만 주피터 노트북이나 구글 코랩에서 확인할 수 있습니다. 보통 그래프를 다 그린 다음에 plt.show()를 호출합니다.

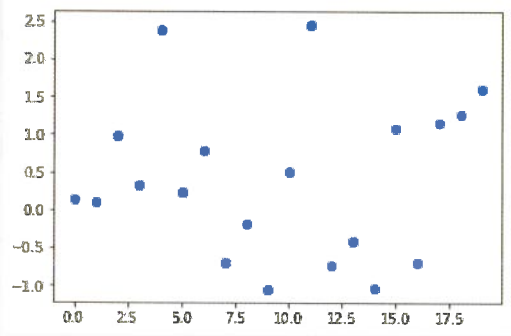
기본은 꺾은선 그래프입니다. 표시 형식은 얼마든지 조절할 수 있습니다. 점으로 바꾸려면 plt.plot(x,y) 대신 plt.plot(x,y,'bo')라고 입력하면 됩니다.

예제 3.32 간단한 점 그래프 그리기

[IN]

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

[OUT]



여기서 추가된 'bo' 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.

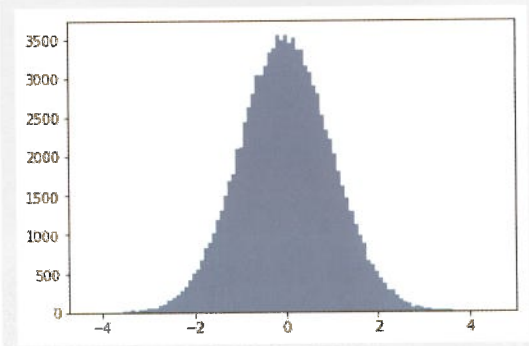
또 한 가지 알아두면 유용한 그래프는 히스토그램입니다. 앞에서 살펴본 정규분포가 실제로 가운데에서 빈번한 값을 가지고 가장자리에서 드물게 나타나는지를 확인하기 위해 히스토그램을 그려볼 수 있습니다.

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()
```

[OUT]



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

3.4.2 2-레이어 XOR 네트워크의 정보 시각화

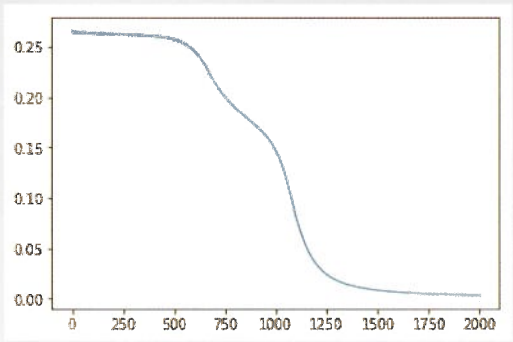
딥러닝을 학습시킬 때 가장 많이 보게 될 그래프는 바로 학습이 잘 되고 있는지 확인하기 위한 측정치(metric) 변화량을 나타내는 선 그래프입니다.[17] 여기서는 선 그래프를 이용해 손실이 어떻게 변했는지를 알아보겠습니다. 예제 3.28에서 history라는 변수에 tf.keras가 학습을 진행한 내역을 저장했습니다. 여기에 저장되는 정보 중 loss를 불러와서 그래프를 그릴 수 있습니다.

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

[IN]

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

[OUT]



그래프를 통해 처음에는 손실이 서서히 감소하다가 어느 시점부터 급격히 감소하고, 나중에는 거의 감소하지 않는 뒤집힌 S자 곡선을 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 그래프를 만들어줍니다. 위에서 x를 입력하지 않았는데도 x축에 값이 출력되는 이유입니다.

17 분류 등에서는 정확도(accuracy) 등 다른 측정치도 관찰해야 합니다.

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.