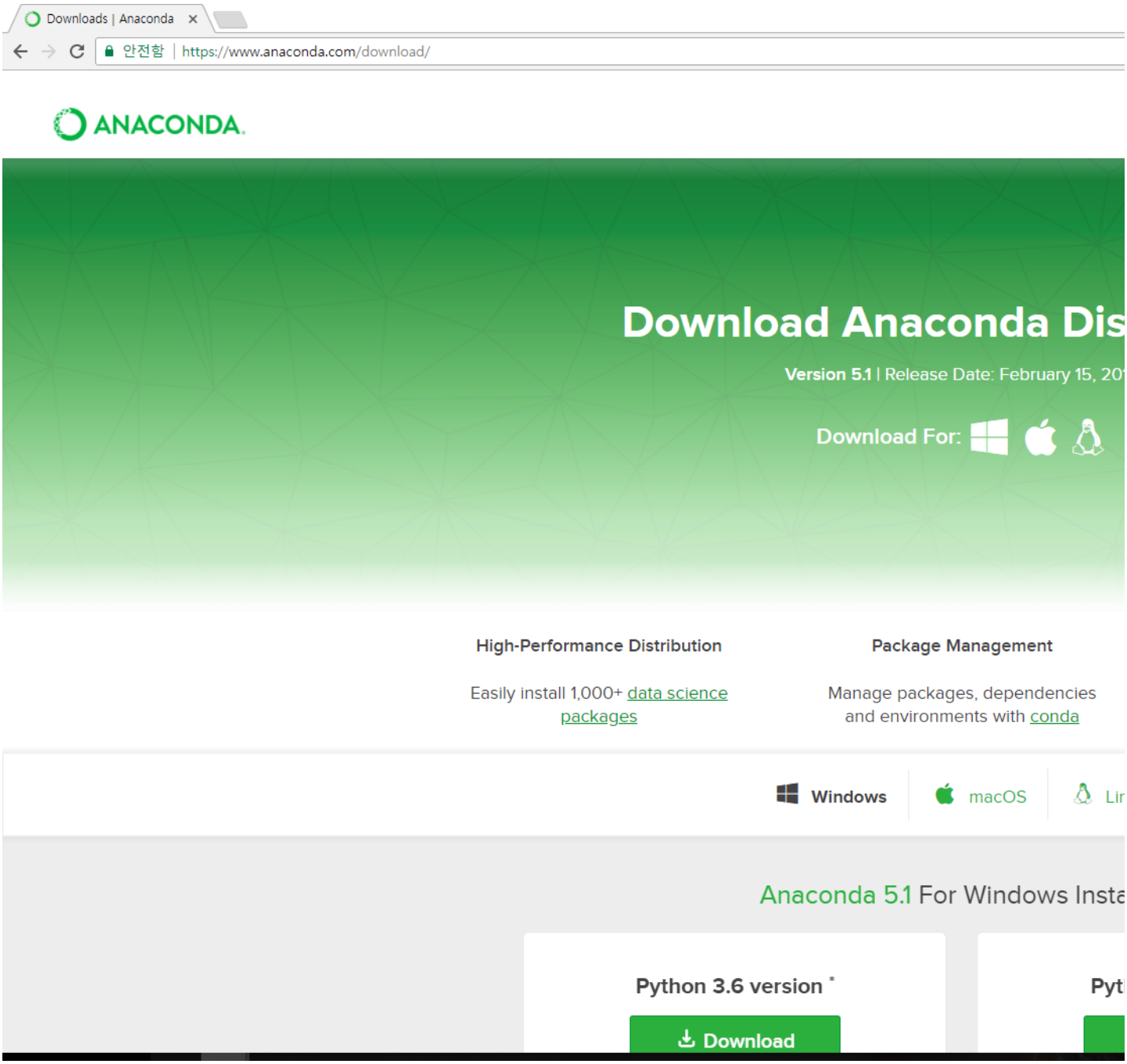
6. 인공신경망

6.1. Anaconda

인공신경망 실습을 위해서는 Python이 필요하다. Python은 공식 홈페이지에서도 무료로 다운 받을 수 있지만, 데이터 분석에는 많은 추가 패키지가 필요하다. 패키지들 중에는 윈도에서 설치가 까다로운 것들도 있다. 이런 문제를 피하기 위해 아나콘다를 설치한다. 아나콘다는 Python과 여러 패키지들을 쉽게 설치/관리할 수 있는 배포판이다.

6.1.1. 아나콘다 다운로드

아나콘다를 설치하려면 구글에서 "anaconda download"를 검색하거나 https://www.anaconda.com/download 에 직접 접속한다.

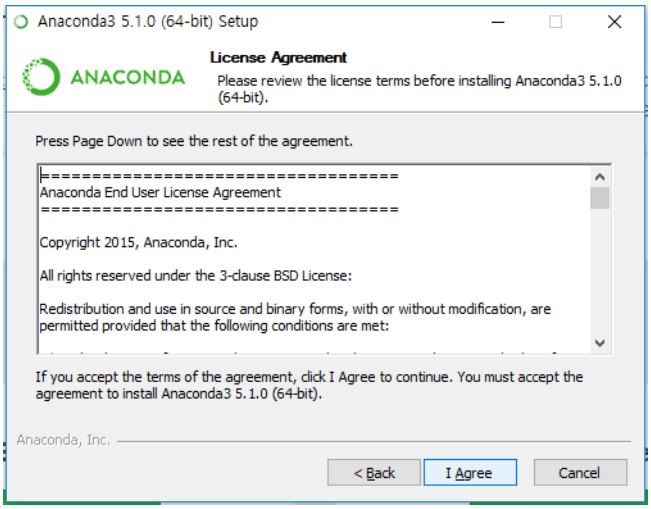


운영체제(Windows, macOS, Linux)를 선택 후 Python 버전을 선택한다. Python 버전은 최신 버전인 **3.6**을 선택한다. 설치 파일을 받고, 실행을 시키면 설치가 시작된다.

6.1.2. 설치 파일 실행

Next 를 클릭하고,

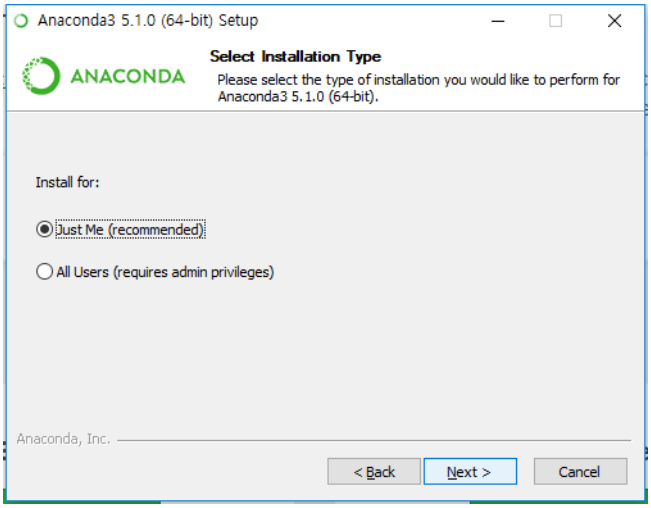
6.1.3. 라이센스 동의



I Agree 를 클릭한다.

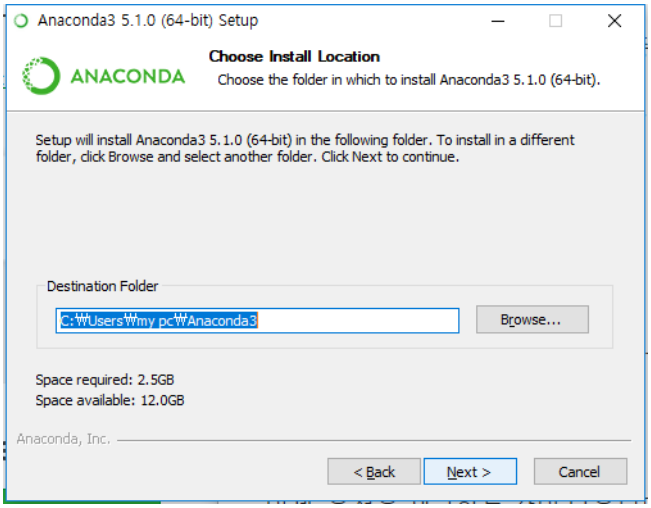
이후로는 윈도를 기준으로 설명한다. 맥이나 리눅스에서는 옵션 선택 없이 Next만 누르면 된다.

6.1.4. 설치 유형



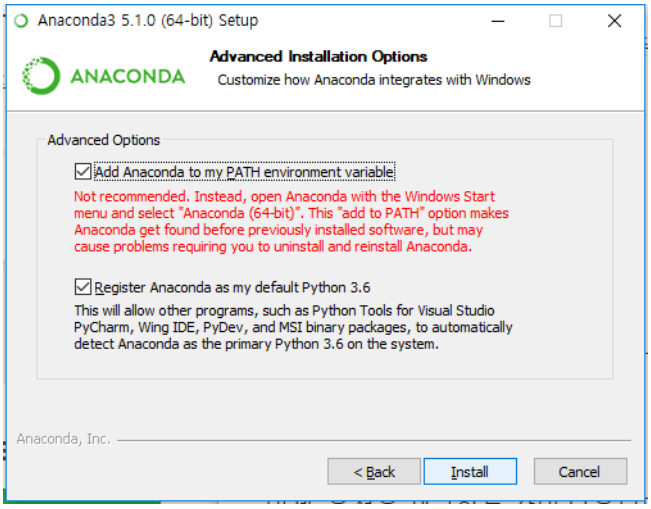
Just me를 선택하고 **Next** 를 클릭한다.

6.1.5. 설치 위치



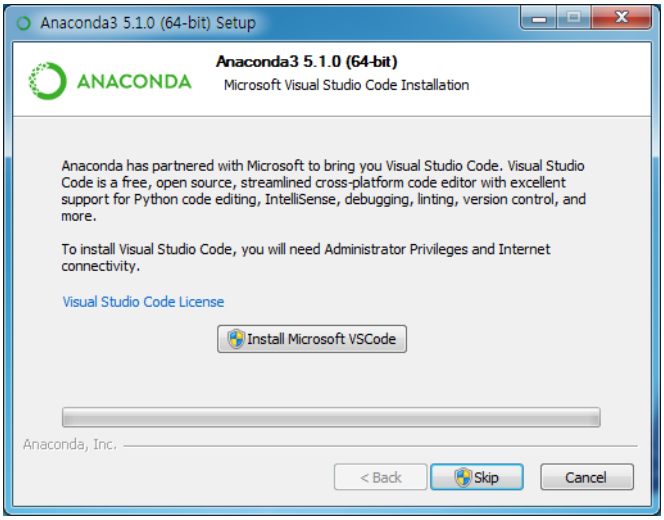
다음은 아나콘다 저장 경로를 선택한다. 경로를 바꾸지말고 **Next** 를 클릭한다.

6.1.6. 고급 설치 옵션



고급 설치 옵션을 선택한다. 아나콘다의 설치 경로를 PATH 환경 변수에 추가를 해주고 기본 Python으로 설정하는 옵션이다. **둘 다 체크**를 해주고 **Install** 을 클릭한다.

6.1.7. VS Code 설치 여부



설치가 완료되면 VS Code 설치 여부를 묻는데, 이 때 **Skip** 을 클릭하면 된다.

6.2. 인공신경망

6.2.1. 퍼셉트론

인공신경망(artificial neural network)는 생물의 신경망을 흉내낸 모형이다. 신경망은 신경 세포, 뉴런(neuron)의 네트워크다. 뉴런의 기본 작동방식은 이렇다.

첫째, 뉴런은 다른 뉴런들로부터 신호를 받는다.

둘째, 뉴런 사이의 연결 강도에 따라 신호는 크고 작게 변한다.

셋째, 받아들인 신호의 총량이 역치(threshold)를 넘어서면 뉴런은 활성화되어 다른 뉴런들로 신호를 전달한다.

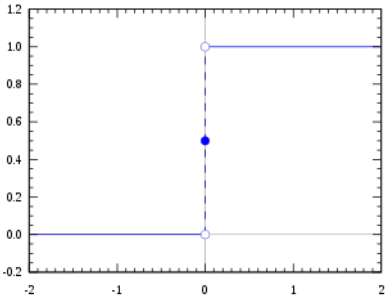
이러한 뉴런의 특성을 따른 최초의 모형은 1957년에 발표된 퍼셉트론(perceptron)이다. 뉴런의 작동방식에서 첫째와 둘째를 수식으로 나타내면 선형모형과 똑같다.

$$y = wx + b$$

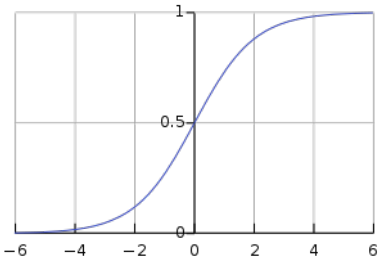
고전적인 퍼셉트론은 $y > 0$이면 활성화되서 1의 신호를 내보내고, 이외의 경우에는 0의 신호를 내보냈다.

6.2.2. 활성화 함수

다른 말로 표현하면 고전적 퍼셉트론의 활성화 함수는 계단 함수(step function)이다.



이런 함수는 기울기가 모든 점에서 0이고, 0에서는 불연속이기 때문에 경사하강법을 쓸 수가 없다. 그래서 매끄럽게 변하는 로지스틱 함수를 활성화 함수로 흔히 쓴다.



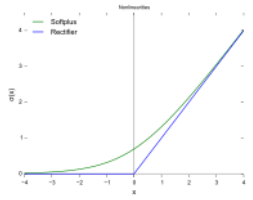
퍼셉트론에 로지스틱 함수를 활성화 함수로 적용하면 로지스틱 회귀분석과 완전히 동일한 형태가 된다. 로지스틱 함수는 출력값이 0에서 1사이의 범위를 갖는다.

신경망의 맥락에서는 로지스틱 함수는 시그모이드(sigmoid) 함수라는 이름으로 더 많이 부른다. 시그모이드는 그리스어로 "S자 모양의"라는 뜻이다.

이와 비슷하게 쌍곡 탄젠트 함수를 사용하기도 한다. 쌍곡 탄젠트는 출력값이 -1에서 1사이의 범위를 갖는다.

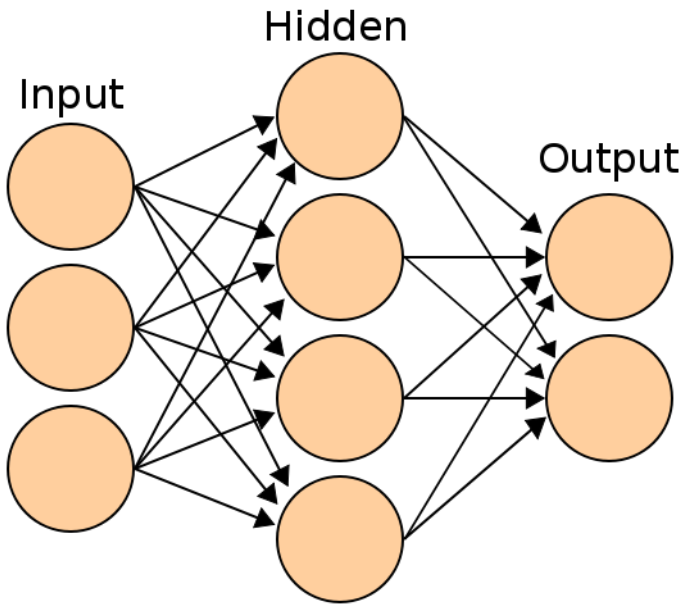
쌍곡탄젠트는 시그모이드에 비해 몇 가지 장점이 있다. 첫째, 출력값이 +, -로 나오기 때문에 편향되지 않는다. 둘째, 0 주변에서 기울기가 더 급해서 사라지는 경사 문제가 적다.

최근에는 ReLU라는 활성화 함수도 많이 사용된다. ReLU는 $\max(0, x)$ 형태의 함수로 입력값이 0보다 크면 그대로 출력값으로 내보내고 0보다 작으면 0을 내보내는 함수다.



6.2.3. 다층 퍼셉트론

아래 그림과 같이 퍼셉트론을 여러 겹으로 덧붙인 것이 다층 퍼셉트론(Multi-Layer Perceptron: MLP)이다. 앙상블에서 스택킹과도 비슷한 구조가 된다.



(출처: https://en.wikipedia.org/wiki/Artificial_neural_network)

MLP에서는 입력층(input layer)과 출력층(output layer) 사이에 여러 겹의 은닉층이 들어가게 된다. 입력층의 신호가 은닉층으로, 은닉층의 신호가 출력층으로 앞으로 앞으로 전달되는 방식이기 때문에 앞먹임 신경망(feedforward network)라고도 한다.

식으로 나타내보자면 아래와 같이 쓸 수 있다.

$$h = a(W_1 x + b_1) y = a(W_2 h + b_2)$$

$a(w \cdot + b)$를 간단히 f라고 쓴다면 다음과 같다.

$$h = f_1(x) y = f_2(x)$$

더 간단히 쓰자면 이렇게도 쓸 수 있다.

$$y = f_2(f_1(x))$$

6.2.4. 역전파 알고리즘

MLP 또는 앞먹임 신경망은 선형 모형을 스택킹해서 다시 선형 모형으로 최종 예측을 하는 것과 비슷한 구조를 가지고 있다. 그러나 스택킹의 경우 하위 모형들이 직접 종속 변수를 예측하고, 그 예측치를 바탕으로 다시 예측을 한다. 신경망에서 은닉층은 종속변수를 직접 예측하지 않는다.

MLP에서는 경사하강법으로 학습을 한다. MLP의 파라미터는 각 층과 층 사이의 가중치 W와 각 층의 편향 b이다.

경사하강법은 예측 오차를 E라고 하면 다음과 같은 방식으로 파라미터 w를 업데이트 해나가는 방식이다.

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\partial E}{\partial w}$$

W_2의 한 원소 w_2를 업데이트 하려면 $\partial E / \partial w_2$를 알아야한다. 이는 미분의 연쇄 규칙으로 구할 수 있다.

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial w_2}$$

그러면 W_1의 한 원소 w_1은 어떻게 업데이트 할까? 역시 미분의 연쇄 규칙으로 구할 수 있다.

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial h} \frac{\partial h}{\partial w_1}$$

위에서 눈여겨 볼 부분은 w_2를 업데이트 할 때 사용했던 $\partial E / \partial y$를 w_1을 업데이트할 때도 사용한다는 것이다. 앞먹임 신경망은 예측을 할 때는 입력층에서 은닉층으로, 은닉층에서 출력층으로 앞으로 신호를 전달하지만, 오차를 바탕으로 파라미터를 추정할 때는 출력층에서 은닉층으로 은닉층에서 입력층으로 경사(gradient)가 역(back)으로 전파(propagate)된다고 해서 이를 역전파 알고리즘이라고 한다.

정리하면 MLP에서 경사하강법으로 학습을 할 때, 역전파 알고리즘으로 경사를 구한다.

6.2.5. 보편적 근사 정리

충분히 큰 은닉층을 가진 인공 신경망은 파라미터만 잘 조정하면 우리가 흔히 다루는 거의 모든 형태의 모형과 비슷하게 만들 수 있다. 이를 **보편적 근사 정리**(universal approximation theorem)라고 한다.

멋진 정리지만 2가지 함정이 숨어 있다. 하나는 '잘 조정하면'이다. 즉, 어떤 파라미터가 존재한다는 것이지 그걸 반드시 찾을 수 있다는 것은 아니다. 경사하강법이든 어떤 학습 알고리즘든 항상 국소최적에 빠지거나 과적합될 수 있다.

또 다른 함정은 '충분히 큰'(sufficiently large)이다. 수학에서 이 말은 어떤 명제가 어떤 크기 N에서 참이면 $N+1$이든 $N+2$이든 N보다 더 큰 크기에서도 참이라는 말이다. 예를 들면 충분히 큰 n에 대해 다음 식이 성립한다.

$$\frac{1}{n} < 0.001$$

보편적 근사 정리를 다시 풀어서 써보면 은닉층의 크기가 N인 신경망을 어떤 모형과 비슷하게 만들 수 있으면, 그 신경망보다 은닉층이 더 큰 신경망도 그 모형과 비슷하게 만들 수 있다. 실제로는 N은 다룰 수 없을 정도로 매우 클 수도 있다.

6.2.6. 심층 신경망 또는 딥러닝

경험적으로 작은 은닉층을 여러 층으로 겹쳐 쌓으면 큰 은닉층 하나와 비슷한 성능을 보인다. 이를 심층 신경망(deep neural network) 또는 딥러닝이라고 한다. 은닉층이 몇 개부터 심층인가에 대해서는 명확한 기준이 있는 것은 아니다.

은닉층을 많이 넣으면 예측력이 좋아진다는 것은 1980년대부터 알려져 있었다. 그렇지만 2000년대 초반까지도 딥러닝은 현실화되지 못했다.

시그모이드 함수의 기울기는 최대가 .25, 쌍곡탄젠트도 1이다. 위에서 봤듯이 신경망에 경사하강법을 적용하면 뒤쪽의 기울기가 여러 번 곱해지게 된다. 1보다 작은 수를 계속 곱하면 점점 작아지므로 앞쪽의 레이어에는 경사가 거의 0이 된다. 이를 사라지는 경사(vanishing gradient)라고 한다.

경사하강법은 말그대로 경사를 따라 내려가는 식으로 파라미터를 개선하는 방법이다. 그런데 경사가 사라지면 파라미터 개선이 안되고, 학습도 안된다. 은닉층을 많이 넣으면 성능이 좋아져야 하지만 실제로는 학습이 안되서 성능이 좋아지지 못하는 것이다.

2000년 초반에 이 문제는 한 가지 해결책이 제시되었다. 그것은 사전 훈련(pretraining)이라는 방식으로, 매 층을 따로 학습시켜서 쌓아 올리는 방법이었다. 그러나 현재는 사전 훈련 없이 한 번에 학습시키는 방식을 쓰고 있다.

6.2.7. 딥러닝이 가능한 이유

현재 딥러닝이 활성화된 이유는 바꿔말하면 사라지는 경사가 어느 정도 해결되었기 때문이다. 대표적인 것이 ReLU다. ReLU는 0보다 큰 범위에서는 기울기가 항상 1이기 때문에 사라지는 경사 문제가 적다.

그러나 가장 큰 이유는 데이터가 많아지고 컴퓨터가 빨라졌기 때문이다. 딥러닝에서 경사가 완전히 사라지는 것은 아니기 때문에 학습이 느려질 뿐 멈추지는 않는다. 현재는 데이터도 많고 컴퓨터도 빠르기 때문에 많은 데이터로 오래 학습을 하면 결국에는 충분한 성능이 나올 때까지 학습을 시킬 수 있다.

바꿔말하면 데이터 양이나 컴퓨터 성능이 충분치 않다면 딥러닝은 충분한 효과를 보여주지 못한다.

6.3. keras 실습

`keras`는 `tensorflow` 등을 손쉽게 사용할 수 있도록 도와주는 라이브러리다. `tensorflow`, `keras`는 모두 Python으로 만들어져 있기 때문에 anaconda 설치가 먼저 필요하다. R에 있는 같은 이름의 라이브러리는 Python 쪽의 기능을 불러서 R에서 사용할 수 있게만 한 것이다.

6.3.1. keras 설치

먼저 R쪽에 `keras`를 설치한다. 이것은 실제의 `keras`가 아닌 R에서 `keras`를 불러 쓰는 기능만 제공한다.

```
install.packages('keras')
```

`keras`를 불러들인다.

```
library(keras)
```

Python 쪽의 실제 `keras`를 설치한다.

```
install_keras()
```

6.3.2. 전처리

```
red.url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'
white.url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv'

red.wine = read.csv(url(red.url), sep = ';')
white.wine = read.csv(url(white.url), sep = ';')

red.wine$color = 'red'
white.wine$color = 'white'

wine = rbind(red.wine, white.wine)

set.seed(1234)

library(caret)

Loading required package: lattice
Loading required package: ggplot2

idx = createDataPartition(wine$color, p=.8, list=F)

data.train = wine[idx, ]
data.test = wine[-idx, ]
```

6.3.3. 데이터 준비

`keras`의 기능들은 `caret`에서 지원해주지 않기 때문에 `train` 함수를 쓸 수 없다. 모형에 입력될 데이터 형식들을 수동으로 맞춰줘야 한다.

먼저 훈련용 데이터에서 입력 데이터(X)를 따로 정리한다. 아래 코드는 컬럼명이 `color`가 아닌 것만 뽑아 행렬(`matrix`) 형태로 변환한다.

```
x.train = as.matrix(data.train[names(wine) != 'color'])
```

`keras`는 문자형 자료를 받아들이지 못하기 때문에 `red`와 비교해서 참/거짓 형태로 바꾼다. 학습 단계에서는 `red`는 1, `white`는 0으로 다시 변환된다.

```
y.train = data.train$color == 'red'
```

X의 변수 수를 확인해둔다.

```
NX = ncol(x.train)
```

6.3.4. 단층 신경망

먼저 단층 신경망을 만들어보자.

```
library(keras)
```

keras는 다양한 형태의 모형을 직접 만들어 볼 수 있도록 하고 있다. 아래 모형은 로지스틱 회귀와 같다.

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 1, activation = 'sigmoid', input_shape = c(NX))
```

모형 요약을 확인한다.

```
summary(model)
```

```
_________________________________________________________________________________
Layer (type)                         Output Shape                      Param #
=================================================================================
dense_2 (Dense)                      (None, 1)                         13
=================================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
_________________________________________________________________________________
```

만들어진 모형을 학습할 수 있는 상태로 변환한다. 손실함수에는 크로스 엔트로피, 최적화 알고리즘에는 경사하강법의 일종인 ADAM, 지표로는 정확도(accuracy)를 사용하겠다.

```
model %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)
```

이제 실제로 학습을 한다. epoch는 학습 횟수를 뜻한다. 여기서는 전체 데이터를 총 30번 입력하여 학습시킨다. 신경망을 학습시킬 때는 한 번에 전체 데이터를 모두 학습시키는 것이 아니라 작은 덩어리(mini batch)로 잘라서 학습을 시킨다. batch_size는 한 번에 학습시킬 데이터의 크기를 뜻한다. 여기서는 한 번에 128개씩 데이터를 학습시킨다. vailidation_split은 검증용으로 사용할 데이터의 비율이다. 여기서는 매 에포크마다 20%의 데이터는 검증용으로 사용한다.

```
history = model %>%  fit(
  x.train, y.train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2)
```

학습 과정에서 검증 데이터로 측정된 손실함수의 값을 확인해보면 에포크마다 점점 낮아지는 것을 볼 수 있다.

```
history$metrics$val_loss
```

```
[1] 11.9443606 10.8271304  7.1177690  3.1876421  2.4009639  2.3184363
 [7]  2.2799400  2.2462448  2.2062077  2.1673731  2.1211101  2.0778268
[13]  2.0280327  1.9772945  1.9216399  1.8615892  1.8025173  1.7391974
[19]  1.6774342  1.6089310  1.5407790  1.4728204  1.3999547  1.3311191
[25]  1.2585428  1.1850394  1.1169559  1.0493464  0.9823730  0.9175414
```

반대로 정확도는 에포크마다 점점 높아진다.

```
history$metrics$val_acc
```

```
[1] 0.2307692 0.2384615 0.2769231 0.5028846 0.6173077 0.6365385 0.6413462
 [8] 0.6413462 0.6413462 0.6403846 0.6423077 0.6394231 0.6384615 0.6326923
[15] 0.6278846 0.6326923 0.6307692 0.6317308 0.6317308 0.6336538 0.6336538
[22] 0.6317308 0.6326923 0.6336538 0.6326923 0.6442308 0.6442308 0.6528846
[29] 0.6528846 0.6673077
```

이제 테스트를 해보자.

```
x.test = as.matrix(data.test[names(wine) != 'color'])
y.test = data.test$color == 'red'

model %>% evaluate(x.test, y.test)
```

```
$loss
[1] 0.846889

$acc
[1] 0.6656394
```

6.3.5. 다층 신경망

다층 신경망을 만드는 방법은 단층 신경망을 만드는 방법과 거의 같다. 아래는 4개의 뉴론으로 된 은닉층과 1개의 뉴론으로 된 출력층을 가지는 다층 신경망이다.

```
mlp <- keras_model_sequential()
mlp %>%
  layer_dense(units = 4, activation = 'sigmoid', input_shape = c(NX)) %>%
  layer_dense(units = 1, activation = 'sigmoid')
```

모형 요약

```
summary(mlp)
```

```
_________________________________________________________________________________
Layer (type)                         Output Shape                      Param #
=================================================================================
dense_3 (Dense)                      (None, 4)                         52
_________________________________________________________________________________
dense_4 (Dense)                      (None, 1)                         5
=================================================================================
Total params: 57
Trainable params: 57
Non-trainable params: 0
_________________________________________________________________________________
```

모형 변환

```
mlp %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)
```

학습

```
history = mlp %>%  fit(
  x.train, y.train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2)
```

테스트

```
mlp %>% evaluate(x.test, y.test)
```

```
$loss
[1] 0.5571135

$acc
[1] 0.7542373
```

6.3.6. 회귀 문제의 경우

회귀 문제를 신경망으로 다룰 경우 출력층의 활성화 함수를 선형(`linear`)으로, 손실함수를 `mse`로 바꿔주면 된다.

```
reg <- keras_model_sequential()
reg %>%
  layer_dense(units = 4, activation = 'sigmoid', input_shape = c(NX - 1)) %>%
  layer_dense(units = 1, activation = 'linear')

reg %>% compile(
  loss = 'mse',
  optimizer = optimizer_adam()
)
```

학습과 평가는 크게 다르지 않다. 아래는 12열 즉, `quality`를 예측하도록 학습시킨 것이다.

```
reg %>% fit(
  x.train[,-12],
  x.train[,12],
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)

reg %>% evaluate(
    x.test[,-12],
    x.test[,12]
)

loss
0.8028317
```

6.4. reticulate

`reticulate`는 R에서 Python 기능을 불러 쓸 수 있게 해주는 라이브러리다.

다음과 같은 방식으로 쓸 수 있다.

```
library(reticulate)
os <- import("os")
os$listdir(".")
```

만약 사용하는 Python을 바꾸고 싶다면 `use_python`을 쓰면 된다.

```
library(reticulate)
use_python("/usr/local/bin/python")
```

`use_virtualenv()`와 `use_condaenv()`로 가상환경을 골라서 쓸 수도 있다.

```
library(reticulate)
yenv")
```