# Pythonic Code

* 파이썬 스타일의 코딩 기법
  + 파이썬 특유의 문법을 최대한 활용한, 효율적 코드 표현
  + 대표적인 기법들
    - split & join
    - list comprehension
    - enumerate & zip
    - lambda & map & reduce
    - generator
    - asterisk
  + Why?
    - 다른 코드에 대한 이해도
    - 효율: for loop append 보다 list가 빠르다거나, 코드가 짧아짐
    - 간지…!
* split & join
  + string type의 값을 ‘기준값’으로 나눠서 list 형태로 변환

|  |
| --- |
| >>>items ='zero one two three'.split()# 빈칸을기준으로문자열나누기  >>>print(items)  ['zero','one','two','three']  >>>example ='python,java,javascript'# ","을기준으로문자열나누기  >>>example.split(",")  ['python',‘java','javascript']  >>>a,b,c =example.split(",")# 리스트에있는각값을a,b,c변수로unpacking  >>>example =‘teamlab.technology.io'  >>>subdomain,domain,tld=example.split('.')# "."을기준으로문자열나누기→ Unpacking |
|  |

* + join: string으로 구성된 list를 합쳐 하나의 string으로 반환

|  |
| --- |
| >>>colors =['red','blue','green','yellow']  >>>result =''.join(colors)  >>>result  'redbluegreenyellow'  >>>result =' '.join(colors)# 연결시빈칸1칸으로연결  >>>result  'red blue green yellow'  >>>result =', '.join(colors)# 연결시", "으로연결  >>>result  'red, blue, green, yellow'  >>>result ='-'.join(colors)# 연결시"-"으로연결  >>>result  'red-blue-green-yellow' |

* List Comprehension
  + 기존 list를 사용하여 간단히 다른 list를 만드는 기법
  + 파이썬의 대표적인 기법 중 하나. 일반적인 for loop보다 속도도 빠르다

|  |
| --- |
| >>>result =[i for i in range(10)]  >>>result  [0,1,2,3,4,5,6,7,8,9]  >>>result =[i for i in range(10) if I % 2 == 0]  >>>result  [0,2,4,6,8] |

|  |
| --- |
| >>>word\_1 ="Hello"  >>>word\_2 ="World"  >>>result =[i+j for I in word\_1 for j in word\_2]  # Nested For loop  >>>result  ['HW','Ho','Hr','Hl','Hd','eW','eo','er',  'el','ed','lW','lo','lr','ll', 'ld', 'lW',  'lo', 'lr', 'll', 'ld', 'oW', 'oo', 'or', 'ol', 'od'] |

* + filter: 조건에 맞는 값만 생성됨

|  |
| --- |
| >>>case\_1 =["A","B","C"]  >>>case\_2 =["D","E","A"]  >>>result =[i+jforiincase\_1 forj incase\_2]  >>>result  ['AD','AE','AA','BD','BE','BA','CD','CE','CA']  >>>result =[i+j for I in case\_1 for j in case\_2 if not(i==j)]  # Filter: i랑j과같다면List에추가하지않음  # [i+j if not(i==j) else I for I in case\_1 for j in case\_2]  >>>result  ['AD','AE','BD','BE','BA','CD','CE','CA']  >>>result.sort()  >>>result  ['AD','AE','BA','BD','BE','CA','CD','CE'] |

|  |
| --- |
| >>>words ='The quick brown fox jumps over the lazy dog'.split()  # 문장을빈칸기준으로나눠list로변환  >>>print(words)  ['The','quick','brown','fox','jumps','over','the','lazy','dog']  >>>  >>>stuff =[[w.upper(),w.lower(),len(w)] for w in words]  # list의각elemente들을대문자, 소문자, 길이로변환하여two dimensional list로변환  For I in stuff:  Print(i)  ### 각 요소들 출력 |

|  |
| --- |
| Case\_1 = [“A”, “B”, “C”]  Case\_2 = [“D”, “E”, “A”]  result =[ [i+j for I in case\_1] for j incase\_2]  # 뒤의 for문 (j)가 먼저 작동하고 (2중 for문의 상단 for문이 됨)  # 앞 쪽의 for문은 그 후에 작동한다.  Result  [[‘AD’, ‘BD’, ‘CD’], [‘AE’, ‘BE’, ‘CE’], [‘AA’, ‘BA’, ‘CA’]] |

* enumerate
  + list의 요소를 추출할 때, 번호를 붙여서 추출함

|  |
| --- |
| For I, v in enumerate([‘tic’, ‘tac’, ‘toe’]):  Print(I, v)  0 tic  1 tac  2 toe |

|  |
| --- |
| >>>mylist=['a','b','c','d']  >>>list(enumerate(mylist))# list의있는index와값을unpacking하여list로저장  [(0,'a'),(1,'b'),(2,'c'),(3,'d')]  >>>{i:jfori,jinenumerate('Artificial intelligence (AI), is intelligence demonstrated by machines, unlike the natural intelligence displayed by humans and animals.'.split())}  # 문장을list로만들고list의index와값을unpacking하여dict로저장  {0:'Artificial',1:'intelligence',2:'(AI),',3:'is',4:'intelligence',5:'demonstrated',6:'by',7:'machines,',8:'unlike',9:'the',10:'natural',11:'intelligence',12:'displayed',13:'by',14:'humans',15:'and',16:'animals.'} |

* zip
  + 두개의 list를 병렬적으로 추출할 때

|  |
| --- |
| >>>alist=['a1','a2','a3']  >>>blist=['b1','b2','b3']  >>>for a,b in zip(alist,blist): # 병렬적으로값을추출  ...print(a,b)  ...  a1 b1  a2 b2  a3 b3  >>>a,b,c=zip((1,2,3),(10,20,30),(100,200,300))#각tuple의같은index 끼리묶음  (1,10,100)(2,20,200)(3,30,300)  >>>[sum(x) for x in zip((1,2,3),(10,20,30),(100,200,300))]  # 각Tuple 같은index를묶어합을list로변환  [111,222,333] |

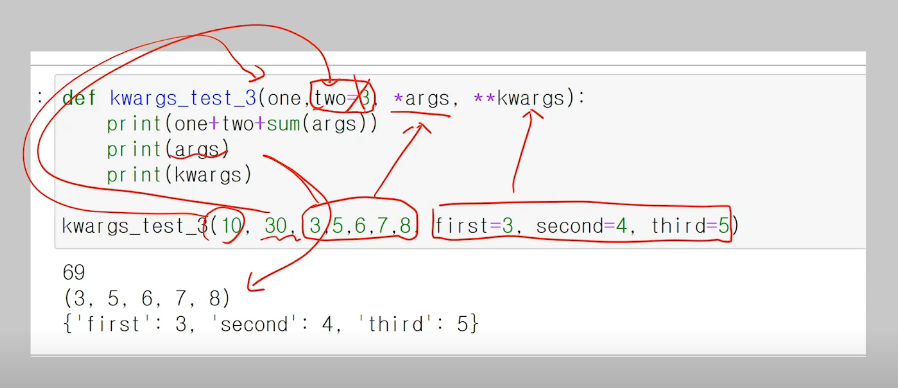
* + enumerate와 같이 사용하면 index도 같이 표현할 수 있다.
* lambda
  + 함수 이름 없이 함수처럼 쓸 수 있는 ‘익명함수’
  + f = lambda x, y: x+y ## f(x,y), return x+y 와 같은 역할
  + 단, PEP 8부터 람다의 사용을 권장하지 않도록 하고 있음
    - 문법이 어렵고, 테스트가 어렵다.
    - 문서화 docstring 지원을 못 함
    - 복잡한 함수의 경우 코드 해석이 어려움
    - 이름이 존재하지 않는 함수라는 것이 문제
    - 하지만 재사용이 필요하지 않고, 간단한 로직이 필요한 경우에는 여전히 많이 씀
* map

|  |
| --- |
| ex =[1,2,3,4,5]  print(list(map(lambda x: x+x, ex)))  print((map(lambda x: x+x, ex)))  f = lambda x: x\*\*2  print(map(f, ex))  ## 1  for I in map(f, ex):  print(i)  ## 2  [f(val) for val in ex] ## 1과 같은 결과임 |

* + 그러나, map을 쓰느니 그냥 리스트 컴프리헨션을 쓰는게 더 편하지 않느냐는 문제 때문에 잘 쓰지 않음.
* reduce
  + map과 달리 list에 똑 같은 함수를 적용해서 통합
* iterable object
  + sequence형 자료에서 데이터를 순서대로 추출하는 object
  + 내부적 구현으로 \_\_iter\_\_와 \_\_next\_\_가 사용됨
    - iter() 함수와 next() 함수로 iterable 객체를 사용할 수 있음

|  |
| --- |
| cities = ["Seoul", "Busan", "Jeju"]  iter\_obj= iter(cities)  print(next(iter\_obj))  print(next(iter\_obj))  print(next(iter\_obj))  next(iter\_obj) |

* + next: 다음 요소의 메모리 주소를 보여줌
* generator
  + iter 객체를 특수 형태로 사용해주는 함수
  + 요소가 사용되는 시점의 값을 메모리에 반환
    - :yield를 사용해 하나의 요소만 반환함
  + generator comprehension
    - list 컴프리헨션과 유사. [] 대신 ()를 사용
  + why?
    - 일반적인 반복은 generator에 비해 훨씬 큰 메모리를 사용한다
    - 따라서 list 타입의 데이터를 반환하는 함수는 generator로 만들면 좋다. 읽기 쉽다는 장점, 중간 과정에서 loop가 중단될 수 있을 때!
    - 큰 데이터를 처리할 때에도 고려하면 좋다.
    - 특히 파일 데이터를 처리하는 경우에도 generator를 쓰자



# Object-Oriented Programming: Class

* 변수 명명 규칙
  + 변수, 함수명: snake\_case
  + class이름 : CamelCase
* attribute 추가는 \_\_init\_\_, self를 사용함
  + \_\_init\_\_ 은 객체 초기화 예약함수
  + \_\_가 붙은 것들은 특수한 예약함수 들이다.
    - \_\_main\_\_, \_\_add\_\_, \_\_str\_\_, \_\_eq\_\_, …
* method 추가할 때에는 반드시 파라미터에 self를 추가해야 class 함수로 인정됨
  + self?
    - 생성된 인스턴스 자신을 의미함
    - 밖에서는 인스턴스의 이름을 사용하면 되지만, 클래스 코드 내부에서 자신을 지칭할 때 self를 사용함

|  |
| --- |
| Class SoccerPlayer(object):  Def \_\_init\_\_(self, name, position, back\_number):  Self.name = name  Self.position = position  Self.back\_number = back\_number  Def change\_back\_number(self, new\_number):  Print(“등 번호 변경함 From %d to %d” % (self.back\_number, new\_number))  Self.back\_number = new\_number  Jinhy = SoccerPlayer(“jinhy”, ‘MF’, 10)  Jinhy.change\_back\_number(5) |

# Inheritance

* 상속
  + 부모 클래스로부터 attribute와 method를 물려받은 자식 클래스를 생성하는 것
  + 왜 쓰는가?
    - Polymorphism (다형성)
      * 같은 이름의 메소드 내부 로직을 다르게 작성하여 다양한 상황에서 쓰이도록 할 수 있음
    - 일부 특성만 다른 경우 상속받아 코드를 재활용
    - Visibility (가시성)
      * 객체의 정보를 적절히 숨겨야 하는 경우 사용.
      * 누구나 접근 가능한 객체가 있는 반면, 제한된 접근을 허용해야 하는 경우가 있음.
      * \_\_val : 과 같은 형식으로 private 변수로 설정할 수 있음
    - 캡슐화
      * 클래스 설계 시 클래스 간 간섭과 정보공유를 최소화한다
      * 심판 클래스가 축구선수 클래스의 가족 정보를 알 필요는 없듯 필요 없는 정보에는 접근 할 필요가 없음
      * 소스의 보호, 안정성 등등

# decorator

* @property: 숨겨진 변수를 반환할 수 있도록 해줌
  + 함수를 변수명처럼 쓰는 효과
  + 일반적으로 그대로 전달하기 보다는 copy를 해서 전달함 (변경 방지)
* first-class object (1급 객체)
  + 일등함수 또는 일급 객체
  + 변수나 데이터 구조에 할당이 가능한 객체
  + 파라미터로 전달이 가능하고, 리턴 값으로도 사용할 수 있는 객체!
  + 파이썬의 모든 함수는 일급 함수임

|  |
| --- |
| Def square(x):  Return x\*x  F = square # 함수를 변수처럼 사용할 수 있다  F(5) # 25 |

* inner function
  + 함수 내에 존재하는 함수
  + closures: inner function을 return 값으로 반환하는 경우를 말함

|  |
| --- |
| Def print\_msg(msg):  Def printer():  Print(msg)  Return printer  Another = print\_msg(“Hello Python”)  Another() |

* + 클로져 예제

|  |
| --- |
| Def tag\_func(tag, text):  Text = text  Tag = tag    Def inner\_func():  Return ‘<{0}> {1} <{0}>’.format(tag, text)  Return inner\_func  H1 = tag\_func(‘title’, ‘this is python’)  P1 = tag\_func(‘p’, ‘data’) |

* + 데코레이터와 결합하여 복잡한 클로져 함수를 간단히 사용할 수 있음

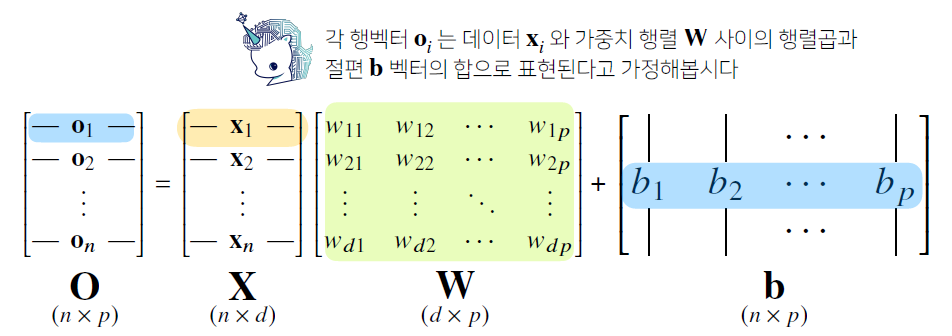
|  |
| --- |
| Def star(func):  Def inner(\*args, \*\*kwargs):  Print(“\*”, \* 30)  Func(\*args, \*\*kwargs)  Print(“\*” \* 30)  Return inner  @star  Def printer(msg):  Print(msg)  Printer(“hello”)  ‘’’  \*\*\*\*\*….  Hello  \*\*\*…..  ‘’’ |

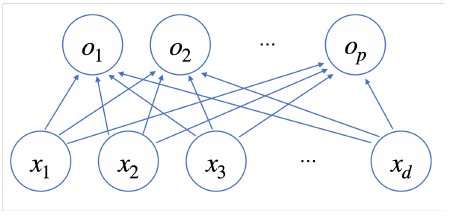
* decorator 응용: 파라미터 입력

|  |
| --- |
| Def generate\_power(exponent):  Def wrapper(f): ## raise\_two가 f로 들어감  Def inner(\*args):  result = f(\*args)  return exponent\*\*result  return inner  return wrapper  @generate\_power(2)  Def raise\_two(n):  return n\*\*2  print(raise\_two(7))  ## 562949953421312  ## 2^49임 |

# AI Math #5 – 딥러닝 학습방법

* 신경망
  + 딥러닝의 데이터는 비선형모델인 신경망을 통해 해석한다.

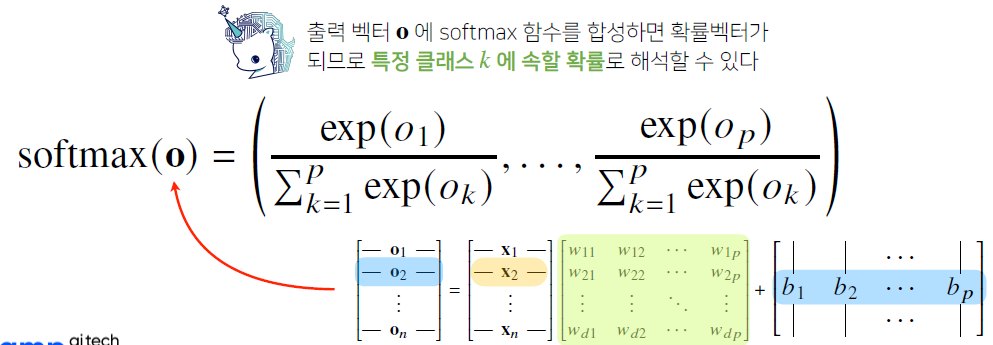


* + 데이터가 바뀌면 결과값도 바뀐다. 출력 벡터의 차원은 d에서 p로 바뀐다
    - (n x d) x (d x p) = (n x p)
    - d개 변수로 p개의 선형모델을 만들어서 p개의 잠재변수를 설명하는 모델이 된다.

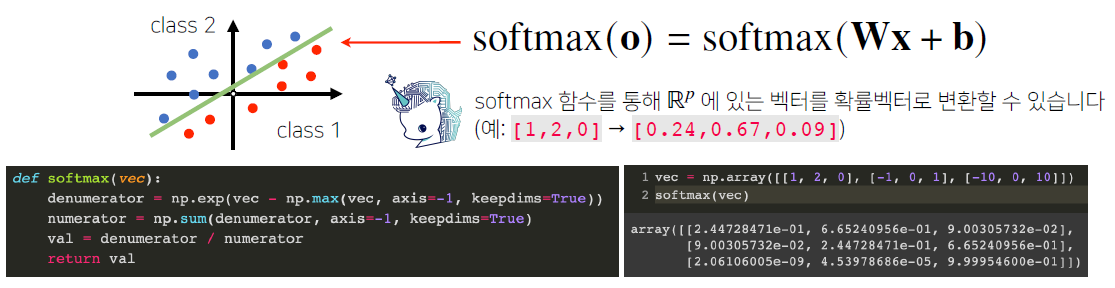
여기서 화살표에는 가중치 w가 할당된다.

각 o마다 연결되는 화살표의 개수는 d개가 된다. 즉, 전체 화살표의 개수는 (p x d)개이다.

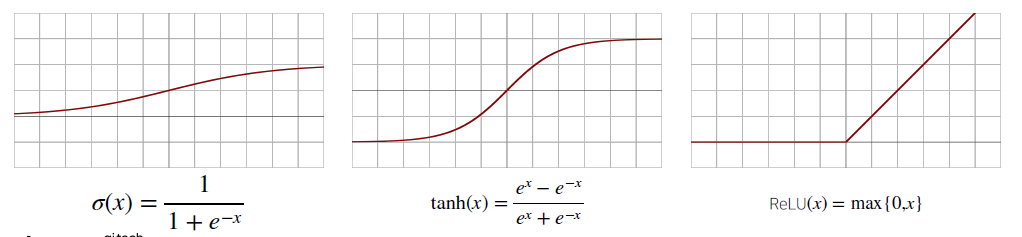
이는 곧 W 행렬과 같다



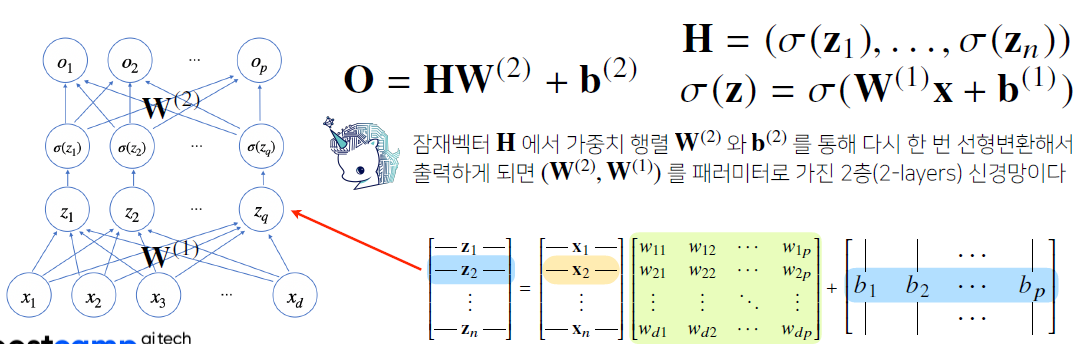
* + 소프트맥스: 모델의 출력을 확률로 해석할 수 있게 변환해주는 연산
  + 분류 문제를 풀 때 선형모델과 소프트 맥스 함수를 결합하여 예측 가능



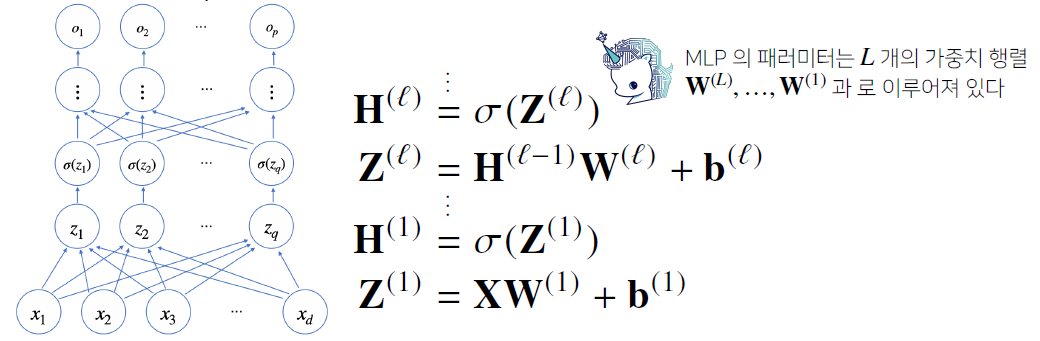
* + 특정 데이터가 어떤 클래스에 속할 지 확률로 나타낼 수 있다
    - 즉, 특정 데이터(벡터로 표현된)를 확률 벡터로 변환한다
    - np.max를 빼는 이유는 너무 큰 값이 들어와 오버-플로우를 방지하기 위함이다.
  + 추론 문제에서는 출력값에서 가장 최대값을 가진 것만 사용하는, one-hot 벡터를 사용하기 때문에 softmax를 사용하지 않는다.
    - 즉, 학습을 하는 경우에는 softmax를 사용하지만, 실제 사용시에는 사용하지 않음
  + 신경망은 선형모델과 활성함수를 합성한 함수이다.
    - 활성함수는 비선형 함수로, 잠재벡터의 각 노드에 개별적으로 적용시켜 새로운 잠재벡터를 만든다.
    - 잠재벡터를 흔히 뉴런이라고 부르게 된다.
* 활성함수
  + activation function은 비선형 함수로, 딥러닝에서 매우 중요한 개념
  + 활성함수를 쓰지 않으면 딥러닝은 선형모형과 차이가 없다!!!
  + 시그모이드 함수나 tanh 함수는 전통적으로 쓰이던 함수. 현재 딥러닝은 ReLU를 많이 쓰고 있다



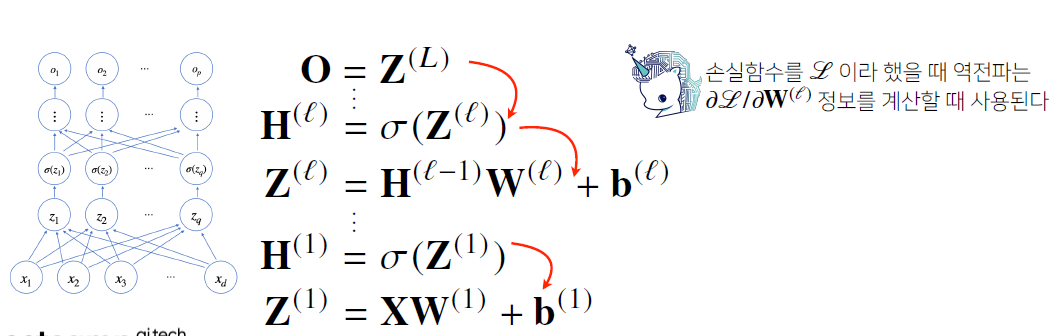
* 신경망과 수식



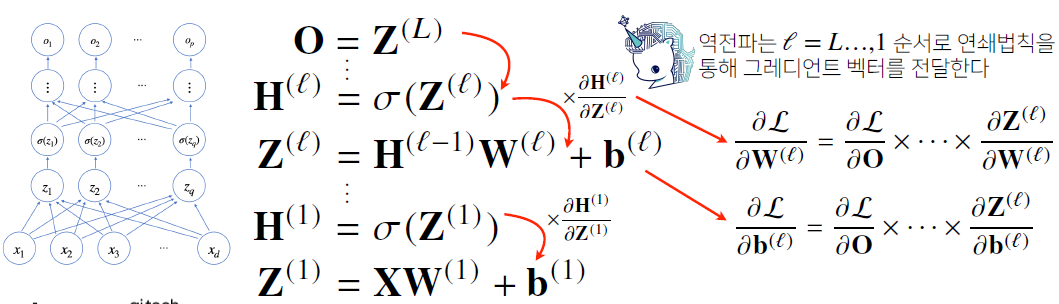
* + x는 입력으로 받는 데이터, z를 받아서 중간층을 구성하게 된다.
    - 이때, 중간층을 구현하는 과정에서 반드시 활성함수를 사용해야 한다.
    - 각 데이터간 (벡터간) 연결 시 가중치 행렬도 적용된다.
  + 층이 반복되면 다층 퍼셉트론이라고 한다. (MLP)
  + 활성 함수는 개별 층마다 개별적으로 적용된다는 점



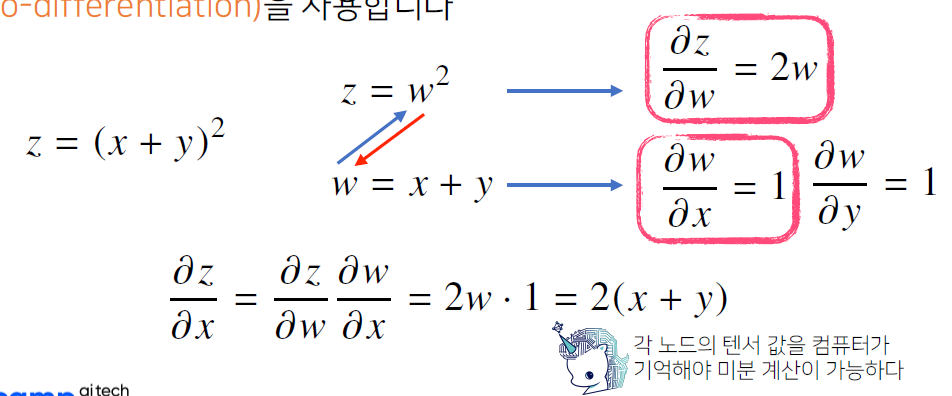
* + 1, … L까지 순차적인 신경망 계산을 ‘순전파’(Forward propagation)라고 부른다.
* 다층 퍼셉트론을 쓰는 이유?
  + 이론적으로는 2층 신경망으로도 임의의 연속함수를 근사할 수 있음
  + 그러나 층이 깊을수록 목적함수를 근사하는데 필요한 뉴런(노드)의 숫자가 훨씬 빨리 줄어들어 좀 더 효율적으로 학습이 가능
    - 층이 얇으면 필요한 뉴런의 숫자가 기하급수적으로 늘어나 넓은 신경망이 되어야 함
* 딥러닝 학습원리: 역전파
  + Back-propagation 알고리즘을 이용해 각 층에 사용된 파라미터 {W(ℓ), b(ℓ)}L(ℓ=1) 를 학습한다
    - 학습한다는 의미는 수정한다는 의미
    - 각 층의 파라미터의 gradient vector는 윗층부터 역순으로 계산한다



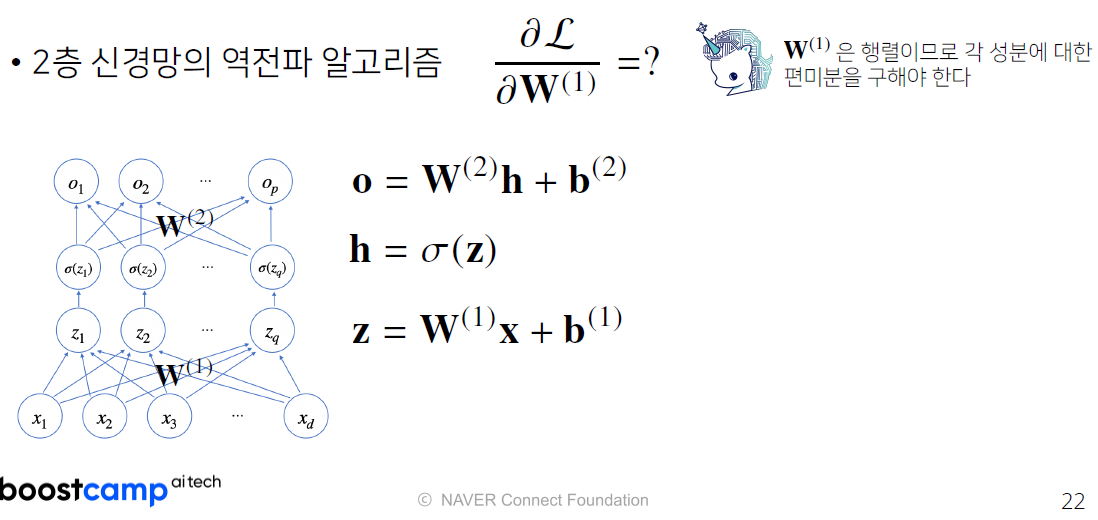
* + 빨간색 화살표: 각 층에서 계산된 gradient들을 밑의 층으로 전달함.

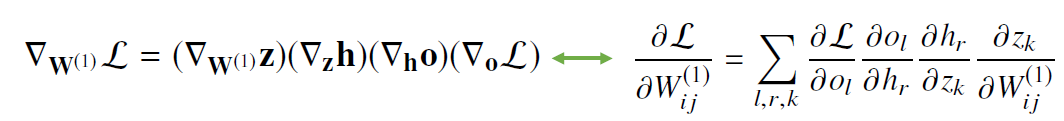


* + 연쇄법칙: 합성함수의 미분법(??)
* 역전파 알고리즘 원리
  + 합성함수 미분법인 연쇄법칙 기반 자동 미분을 사용함

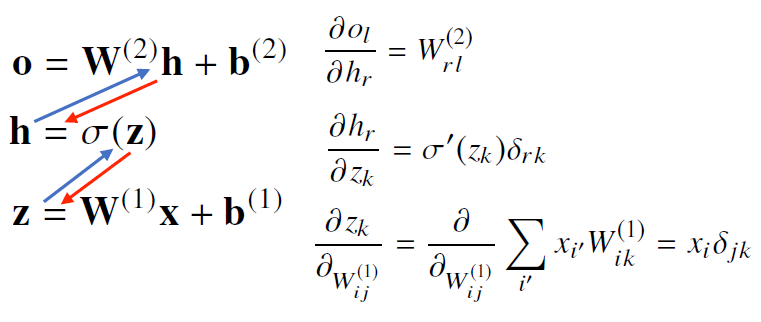


* + 각 뉴런에 해당하는 값 (텐서라고 부름)은 컴퓨터의 메모리에 저장해야 역전파가 가능하다.
  + 위 식에서 z를 x로 미분했을 때 나오는 식을 보자. 같이 사용된 x와 y값을 알아야 계산할 수 있다. 즉, 텐서의 값을 컴퓨터에 저장해야 한다. 즉, 메모리 사용량이 순전파보다 좀 더 많아진다
* 2층 신경망 역전파 예제

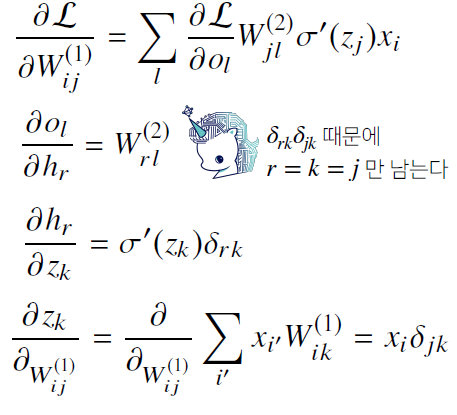




* + 연쇄 미분이다.
    - W1에 대한 손실함수 미분은…
    - 손실함수를 출력 o에 대해 미분하고, 다시 출력 h로 미분하고, 히든 z로 미분하고, 마지막으로 W1으로 미분하게 된다.

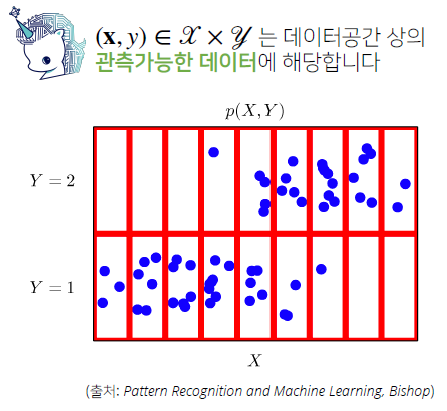


* + 연쇄적인 미분인 것!
  + 이렇게 최종적으로 계산된 값을 gradient로 사용하여 경사하강법에 쓴다

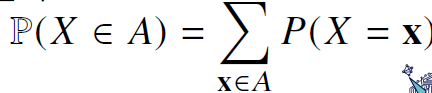


# AI Math #6. 확률론

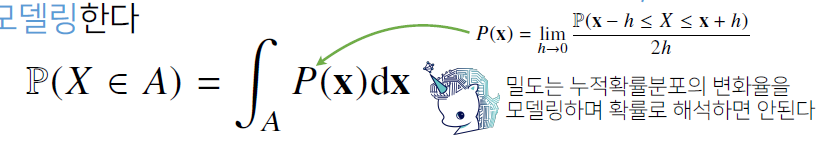
* 딥러닝과 확률론
  + 확률 기반의 기계학습 이론이기 때문
  + 기계 학습에서 쓰이는 손실함수들의 작동 원리는 데이터 공간을 통계적으로 해석하여 유도함
  + 회귀 분석에서 손실함수로 사용되는 L2 노름은 예측오차의 분산을 가장 최소화하는 방향으로 학습하도록 유도함
  + 분류 문제에서 사용되는 교차 엔트로피는 모델 예측의 불확실성을 최소화하는 방향으로 학습을 유도함
  + 즉, 분산과 불확실성을 최소화하기 위해 통계적 방법을 쓰는 것임
* 확률분포는 데이터의 초상화
  + 데이터 공간을 X x Y 라 표기하고, D는 데이터 공간에서 데이터를 추출하는 분포.
  + 데이터는 확률변수로 (X, y) ~ D 라고 표기



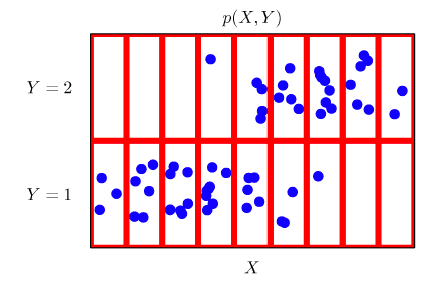
* 이산확률변수 vs 연속확률변수
  + 확률변수는 확률분포 D에 따라 이산형과 연속형으로 나뉨.
    - 데이터공간 X x Y에 의해 결정되는 것이 아니고, D에 결정됨!!
  + 이산 확률 변수는 확률변수가 가질 수 있는 경우의 수를 모두 고려하여 확률을 더해서 모델링한다.



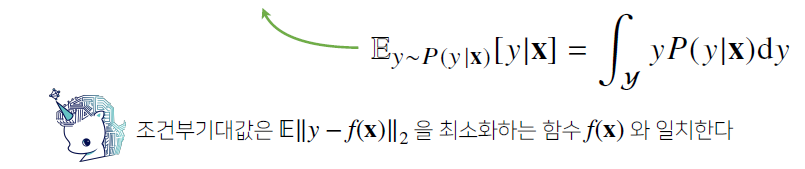
* + - 이를 확률 질량 함수라고 함
  + 연속형 확률 변수는 데이터 공간에 정의된 확률변수의 밀도(density) 위에서 적분을 통해 모델링한다.



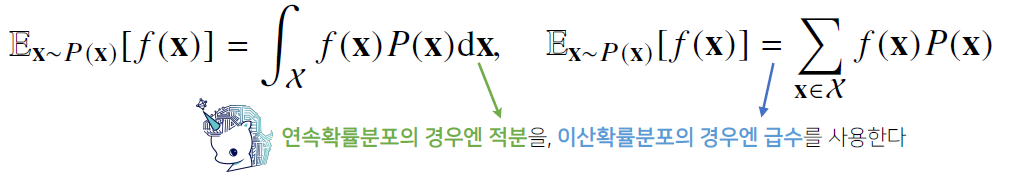
* + - 밀도는 확률이 아니라 변화율!!!

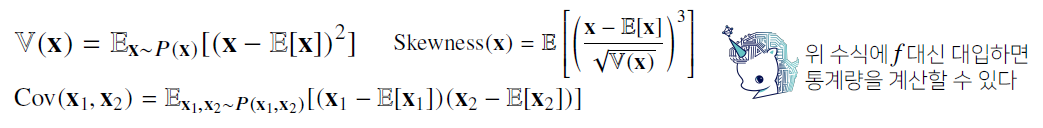


* + 전체 데이터, X와 Y가 주어졌을 때, 이 분포를 결합분포 P(X, Y)라고 한다. (joint distribution)
    - 빨간색 칸에서 파란색 점의 분포가 곧 결합분포이다. 이렇게 하면 마치 이산분포처럼 다룰 수 있다. 즉, 이산형으로 다룰 수 있음
* 조건부 확률
  + 조건부 확률 P(y|X) 는 입력변수 X에 대해 정답이 y일 확률
    - 연속확률분포의 경우 P(y|X)는 확률이 아니고 밀도로 해석함
  + 로지스틱 회귀에서 사용한 선형 모델과 소프트 맥스 함수의 결합은 데이터에서 추출된 패턴을 기반으로 확률을 해석하는데 사용
  + 분류 문제에서 softmax(Wfi + b)는 데이터 x로부터 추출된 특징패턴 Fi(x)와 가중치행렬 W를 통해 조건부 확률을 계산함.
    - P(y| fi(x)) 라고 쓰기도 함
  + 회귀문제는 연속확률 변수를 다루기 때문에 밀도함수를 해석해야 한다.
    - 이 경우 조건부 기대값 E[y|x]를 추정한다

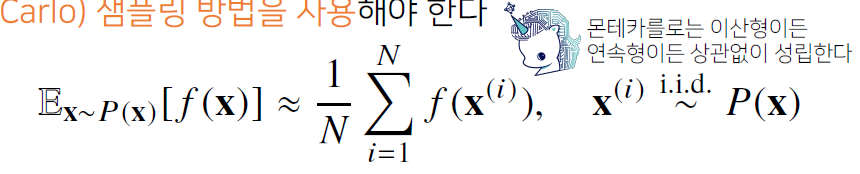


* + - 조건부 기대값은 L2 노름을 최소화하는 함수와 일치한다는 증명이 되어있음. 그래서 회귀문제에서는 조건부 기대값을 사용한다.
* 기대값
  + 확률분포가 주어졌을 때, 데이터 분석을 위한 여러가지 수단(범함수)가 있음
  + 기대값은 데이터를 대표하는 통계량이며 동시에 확률분포를 통해 다른 범함수를 계산하는데 쓰일 수 있음
  + 기대값을 이용해 분산, 첨도, 공분산 등 여러 통계량을 계산할 수 있음

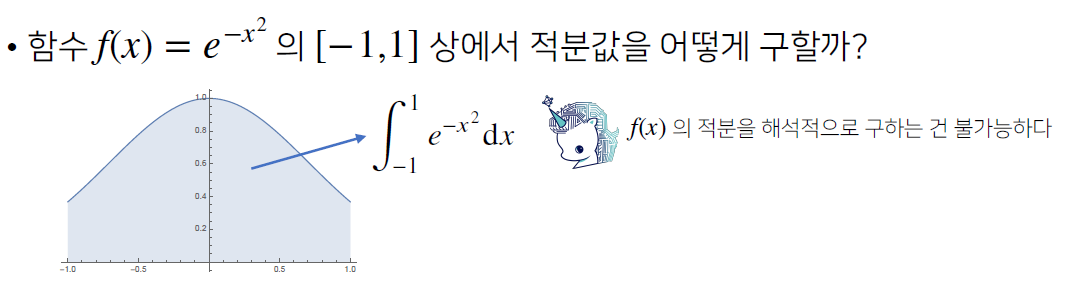




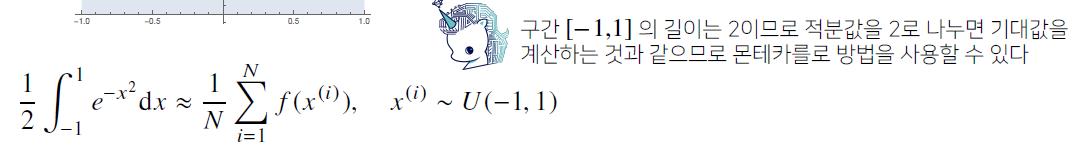
* + 중간 정리
    - 연속확률변수냐 이산확률변수냐에 따라 적분을 취하거나, SUM을 취해준다는 차이가있다.
    - 연속확률변수에서는 밀도함수를 곱하고
    - 이산확률변수에서는 질량함수를 곱한다
* 조건부확률 +
  + 딥러닝은 다층신경망을 사용해 데이터로부터 특징패턴 Fi를 추출한다
    - 특징 패턴을 학습하기 위해 어떤 손실함수를 사용할 지는 기계학습 문제와 모델에 의해 결정
* 몬테카를로 샘플링
  + 기계학습의 많은 문제들은 확률분포를 명시적으로 모를 때가 대부분이다.
  + 즉, 확률분포를 모를 때, 데이터를 이용해 기대값을 계산하는 방법이다.



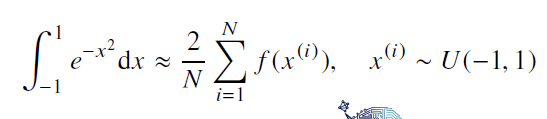
* + - 타겟 함수 f에 샘플링한 데이터 X를 대입한다.
    - 이 f(X)들의 산술평균이 기대값에 근사하게 된다.
    - 단, 독립추출로 샘플링을 진행해야 한다. (대수의 법칙에 의한 수렴성 보장이 됨)
* 몬테카를로 예제: 적분 계산



* + 부정적분 같은 해석적 분석으로는 구할 수 없음… 매우 어려움
  + 몬테카를로 방법을 통하면 근사치를 쉽게 구할 수 있음



* + -1 부터 1 사이의 균등 분포이다. 따라서 길이인 2로 나눠주게 됨
  + 이렇게 구한 값에 2를 곱해주면 원래 적분값의 근사치를 얻을 수 있다!





* + 1.49387 +- 0.0039 범위에 있다
* 결론
  + 데이터의 초상화로서 확률 분포
  + 확률 변수의 종류에 따라 기대값 계산 방식이 다르다
    - 연속형: 적분
    - 이산형: 급수
  + 확률분포를 모르는 경우에도 ‘몬테카를로 샘플링’ 기법으로 기대값 계산 가능
* 퀴즈 #5 분산
  + 1, 3, 5, 7
  + 평균: 4
  + 편차: -3, -1, 1, 3
  + 편차제곱합: 9 + 1 + 1 + 9 = 20
  + 분산 = 편차제곱 평균 = 20 / 4 = 5