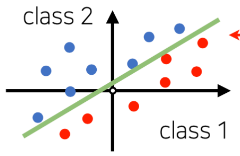
신경망

- **소프트맥스(softmax)**
  - 모델의 출력을 확률로 해석할 수 있게 변환해 주는 연산
  - **분류 문제**를 풀 때 선형모델과 소프트맥스 함수를 결합하여 예측



$$\mathrm{softmax}(\mathbf{o}) = \mathrm{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

softmax 함수를 통해 $\mathbb{R}^p$ 에 있는 벡터를 확률벡터로 변환할 수 있습니다
(예: `[1,2,0]` → `[0.24,0.67,0.09]`)

```
def softmax(vec):
  denumerator = np.exp(vec - np.max(vex, axis = 1, keepdims = True))
  # 각 출력 벡터의 성분값에 지수함수 씌운 것
  # softmax 함수가 지수 함수를 사용해서 너무 큰 벡터 들어오면 overflow 발생 => 방지 위해 np.max이
이용, 원래 softmax 그대로 이용 가능
  numerator = np.sum(denumerator, axis = -1, keepdims = True)
  # 각 출력 벡터의 성분값에 지수함수 씌운 것을 전부 다 더함
  val = denumerator / numerator
  return val

vec = np.array([[~~~]])
softmax(vec) #벡터 => 확률벡터
```

  - **추론 할 때**: **원-핫(one-hot) 벡터**로 최대값을 가진 주소만 1로 출력하는 연산을 사용해 softmax 사용 X

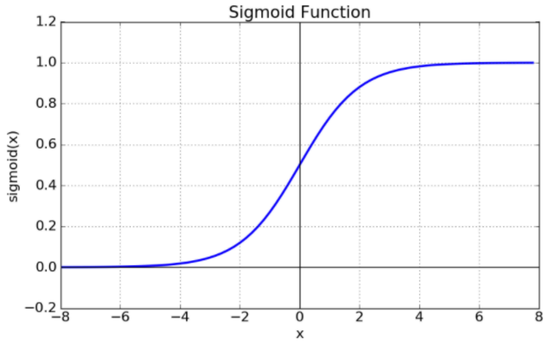
    => one_hot(softmax(o)) 이 아니라 one_hot(o)

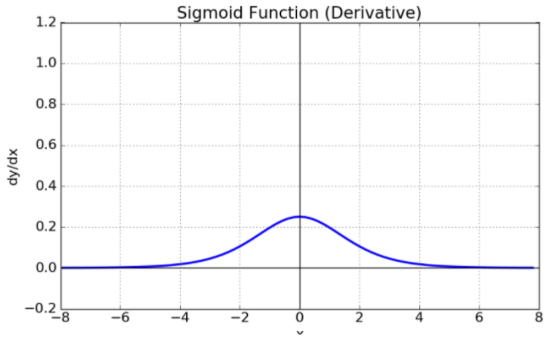
```
def one_hot(val, dim):
  return [np.eye(dim)[_] for _ in val]

def one_hot_encoding(vec):
  vec_dim = vec.shape[1]
  vec_argmax = np.argmax(vec, axis = -1)
  return one_hot(vec_argmax, vec_dim)
```

- **활성함수**
  - $\mathbb{R}$ 위에 정의된 **비선형(nonlinear) 함수**, 딥러닝에서 매우 중요한 개념
  - **활성함수를 쓰지 않으면 딥러닝은 선형모형과 차이 X**
  - 시그모이드(sigmod) 함수나 tanh 함수 => **전통적**으로 많이 쓰던 활성함수
  - ReLU 함수 => **딥러닝에서 많이 씀**
  - **시그모이드 함수(Sigmoid)**

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$





- Logistic 함수라고 불리기도 한다.
- 선형인 멀티퍼셉트론에서 비선형 값을 얻기 위해 사용 시작
- 함수값 (0, 1)로 제한
- 중간 값: $\frac{1}{2}$
- 매우 큰 값을 가지면 함수값: 거의 1
- 매우 작은 값을 가지면 함수값: 거의 0
- 신경망 초기에 많이 사용

  but, 단점 발생, 최근 자주 사용 X

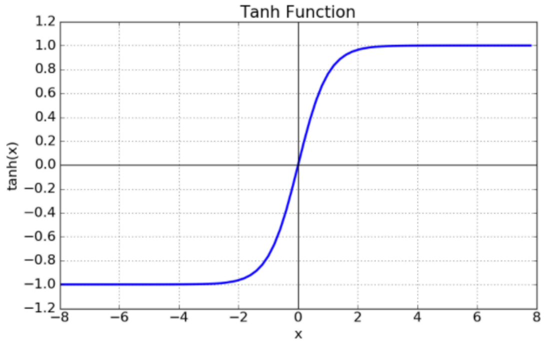
  - Gradient Vanishing 현상 발생
  - 함수값 중심이 0이 아니다 => 학습 느려질 수 이음
  - exp 함수 사용 시, 비용 큼

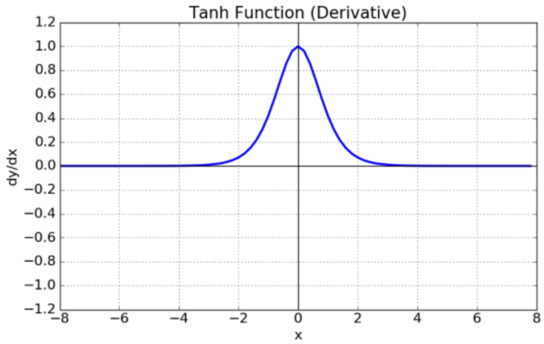
- **Tanh 함수(Hyperbolic tangent)**

$$tanh(x) = 2\sigma(2x) - 1$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$tanh'(x) = 1 - tanh^2(x)$$





- 쌍곡선 함수 중 하나

  쌍곡선 함수: 삼각함수와 유사한 성질을 가지고, 표준 쌍곡선을 매개변수로 표시할 때 나오는 함수

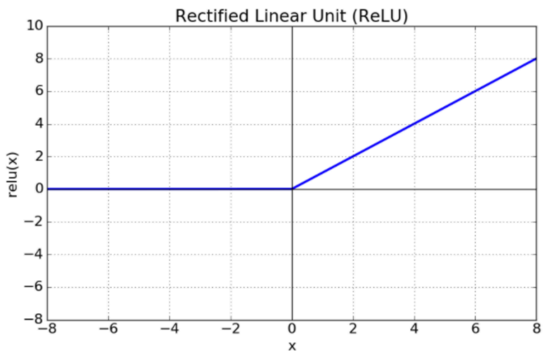
- sigmoid 함수를 transformation해서 얻을 수 있음

  -1 vertical shift & 1/2 horizontal squeeze & 2 vertical stretch

- 함수의 중심값을 0으로 옮겨 sigmoid의 최적화 과정이 느려지는 문제 해결
- but, 미분함수에 대해 일정값 이상 커질 시, 미분값 소실되는 gradient vanishing 문제 여전히 남아있음

ReLU 함수(Recified Linear Unit)

$$f(x) = max(0, x)$$



- 최근 가장 많이 사용하는 활성화 함수
- $x > 0$이면 기울기가 1인 직선
- $x < 0$이면 함수값이 0
- sigmoid, tanh 함수와 비교했을 때, 학습 훨씬 빠름
- 연산 비용 크지 않음
- 구현 매우 간단
- $x < 0$인 값들에 대해 기울기 0 => 뉴런이 죽을 수 있는 단점 존재 (Dying ReLU)

Leakly ReLU

$f(x) = max(0.01x, x)$

        - Dying ReLU 현상 해결 위해 나온 함수
        - 매우 간단한 형태
        - 0.01 대신 다른 매우 작은 값 사용 가능
        - 음수의 x값의 미분값이 0이 되지 않는다는 점을 제외하고 ReLU와 같은 특성 가짐
    - **PReLU**

      $f(x) = max(\alpha x, x)$

        - Leakly ReLU와 거의 유사, but 새로운 파라미터 α를 추가하여 $x < 0$에서 기울기 학습 가능
    - **Exponential LInear Unit(ELU)**

      $f(x) = x \ \ if \ \ x > 0$

      $f(x) = \alpha(e^x - 1) \ \ if \ \ x \leq 0$

        - 비교적 최근에 나온 함수 (2015)
        - ReLU의 모든 장점 포함
        - Dying ReLU 문제 해결
        - 출력값 거의 zero-centered 가까움
        - 일반적인 ReLU와 달리 exp함수를 계산하는 비용 발생
    - **Maxout 함수**

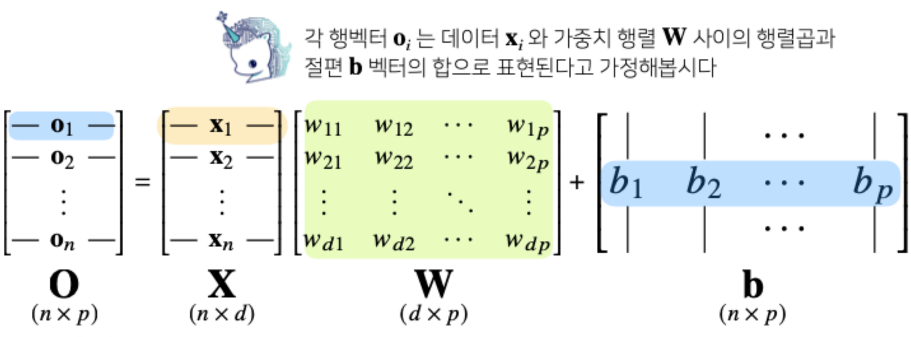
      $f(x) = max(w_1^t x + b_1, w_2^t x + b_2)$

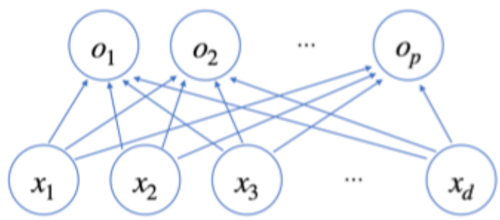
        - ReLU의 모든 장점 포함
        - Dying ReLU 문제 해결
        - 계산량 복잡
    - **가장 많이 사용되는 함수: ReLU**, 간단, 사용 쉬움, 우선적으로 사용
    - sigmoid 사용 X / tanh: 큰 성능 X

- **신경망(neural network)**

  비선형모델

  각 데이터에 대해 d개의 변수로 p개의 선형모델을 만들어 p개의 잠재변수를 설명하는 모델

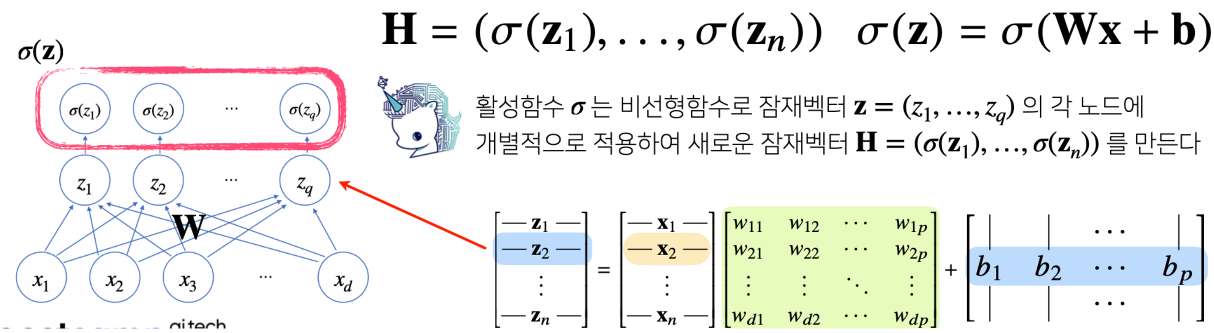




- 신경망: 선형모델과 활성함수(activation function)를 합성한 함수

- 신경망에서 출력벡터 O를 얻기 전 활성함수 σ가 적용되는 하나의 은닉층 H를 생각

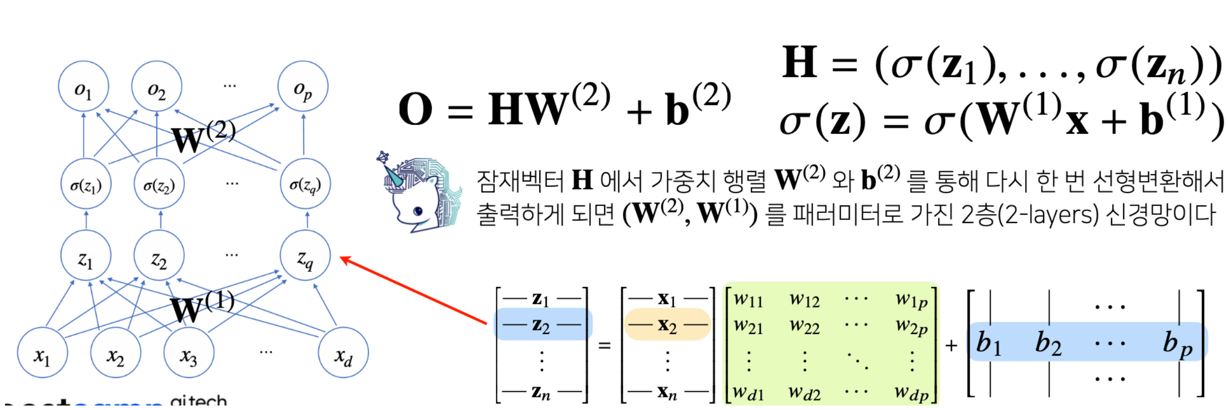
  => 각 잠재벡터 $z_i = (z_1, z_2, ..., z_p)$에 활성함수 σ를 적용하여 생성된 $H = (\sigma(z_1), ..., \sigma(z_n))$ 얻을 수 있음

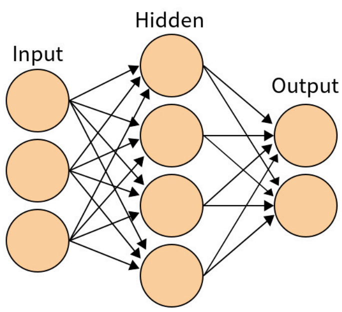


- 소프트맥스: 출력물의 모든 값을 고려해서 출력
- 활성함수: 해당 주소의 출력값만 고려, 벡터가 아닌 하나의 실수값을 input으로 받음
- 만약 선형함수로 활성함수 쌓을 시, **층을 무한정 깊게 쌓아도 선형성을 띄므로** 선형함수를 사용한 하나의 은닉층을 가지는 뉴럴넷과 깊게 쌓은 뉴럴넷 차이 없음 => **비선형 함수여야함**
- 활성함수 이용해서 선형모델을 비선형모델 변형

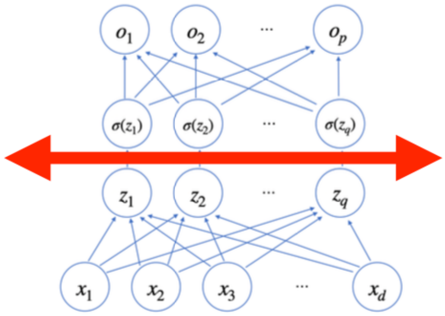
  => 변형한 백터를 **잠재벡터(히든벡터)**

  => 뉴럴, 뉴럴으로 이루어진 모델을 **신경망, 뉴럴 네트워크**라고 부름
- 출력백터 O를 얻기 위해 가중치 행렬 $W^{(2)}$ 과 $b^{(2)}$ 를 통해 선형변환하면 $(W^{(2)}, W^{(1)})$ 을 파라미터로 가지는 **2-layers Neural Net(1-hidden-layer Neural Net)** 이 만들어짐





- 다층(multi-layer) 퍼셉트론(MLP): **신경망이 여러층 합성된 함수**



  - 이론적으로는 2층으로 신경망으로도 임의의 연속함수를 근사할 수 있음 => universal approximation theorem
  - 층이 깊을수록 목적함수를 근사하는데 필요한 뉴런(노드)의 숫자가 훨씬 빨리 줄어들어 좀 더 효율적으로 학습 가능
  - 층이 얇으면 필요한 뉴련의 숫자가 기하급수적으로 늘어남 => 넓은(wide) 신경망이 되어야 함
- **순전파(forward propagation)**

  뉴럴 네트워크 모델의 입력층으로부터 출력층까지 순서대로 변수들을 계산하고 저장한 것

  => **학습과정 아닌 단순히 값들을 다음 층을 넘겨주는 과정**
- **역전파(back propagation) 알고리즘**

  신경망이 여러 층으로 쌓이게 되고, 각 층에 있는 학습 파라미터에 각각에 학습이 이루어지기 위해 이용

  뉴럴 네트워크의 파라미터들에 대한 gradient를 계산하는 방법
  - 뉴럴 네트워크의 각 층과 관련된 목적 함수의 중간 변수들과 파라미터들의 **그레디언트를 출력층에서 입력층 순, 즉 역순으로 합성함수 미분법인 연쇄법칙(chain-rule) 기반 자동미분(auto-differentiation)을 통해 계산 및 저장**

$$\begin{aligned}
\mathbf{O} &= \mathbf{Z}^{(L)} \\
&\vdots \\
\mathbf{H}^{(\ell)} &= \sigma(\mathbf{Z}^{(\ell)}) \quad \times\frac{\partial \mathbf{H}^{(\ell)}}{\partial \mathbf{Z}^{(\ell)}} \\
\mathbf{Z}^{(\ell)} &= \mathbf{H}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{b}^{(\ell)} \\
&\vdots \\
\mathbf{H}^{(1)} &= \sigma(\mathbf{Z}^{(1)}) \quad \times\frac{\partial \mathbf{H}^{(1)}}{\partial \mathbf{Z}^{(1)}} \\
\mathbf{Z}^{(1)} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}
\end{aligned}$$

역전파는 $\ell = L, \ldots, 1$ 순서로 연쇄법칙을 통해 그레디언트 벡터를 전달한다

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{O}} \times \cdots \times \frac{\partial \mathbf{Z}^{(\ell)}}{\partial \mathbf{W}^{(\ell)}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{O}} \times \cdots \times \frac{\partial \mathbf{Z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}}$$

$$z = (x+y)^2$$

$$z = w^2 \rightarrow \boxed{\frac{\partial z}{\partial w} = 2w}$$

$$w = x + y \rightarrow \boxed{\frac{\partial w}{\partial x} = 1} \quad \frac{\partial w}{\partial y} = 1$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial w}\frac{\partial w}{\partial x} = 2w \cdot 1 = 2(x+y)$$

각 노드의 텐서 값을 컴퓨터가 기억해야 미분 계산이 가능하다