


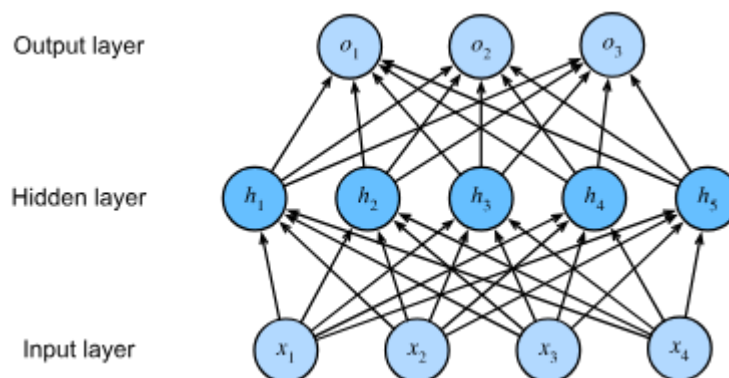
# Dive into DeepLearning(Multilayer Perceptrons)

☀ 상태	Done
👤 작성자	 Antimony02
🕒 최종 편집 일시	@2023년 11월 20일 오후 1:00
☰ 태그	사전 조사

```
%matplotlib inline
import tensorflow as tf
import tensorflow.keras as tfk
```

## 5.1 Multilayer Perceptrons

선형모델에서는 일반적으로 하나의 은닉층으로 이미지를 분류할 때 한 픽셀의 강도를 평가하며 결과를 낸다 하지만 선형모델의 한계로 인해 개별 픽셀의 밝기에 대한 평가를 하는 것은 고양이와 개를 구별하는것에는 실패할것이다. 이런 비선형적인 문제를 해결하기 위해 히든레이어 2개이상을 통합하면서 선형모델의 한계를 극복했다.



위 MLP에 대한 계산식은 다음과 같다

위 방법의 문제점은 은닉층을  $W = W_2W_1$ 과  $b = W_2b_1 + b_2$ 를 사용해 단일층 퍼셉트론식으로 재구성할 수 있기때문에, 다층이 아닌 단일층 퍼셉트론이라는 점이다.

$$h = xW_1 + b_1, o = hW_2 + b_2$$

$$o = W_2h + b_2 = W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2) = Wx + b$$

이를 해결하기 위해 모든 층에  $\max(x, 0)$ 와 같은 비선형 함수 시그마를 추가하는 것이다. 이렇게 하면 층들을 합치는것이 불가능해지는데 이것은 은닉층을 여러개 쌓는것이 가능하다는 것이다.

$$h = \sigma(W_1x + b_1)$$

$$o = W_2h + b_2$$

$$\hat{y} = softmax(o)$$

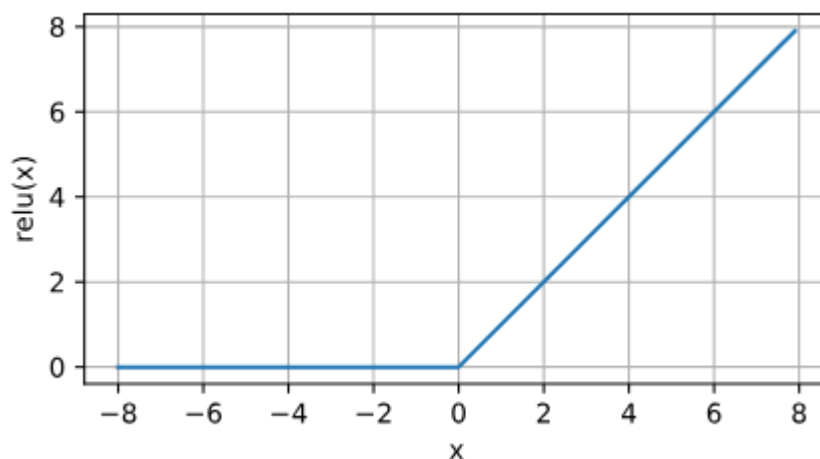
활성화 함수는 가중치 합을 계산하고 여기에 편<sub>b</sub>항을 추가하여 뉴런을 활성화할지 여부를 결정한다. 기본적인 활성화함수 몇가지를 살펴보겠다.

- ReLU함수

구현의 단순성과 다양한 예측 작업에 대한 우수한 성능으로 인해 가장널리 사용된다.

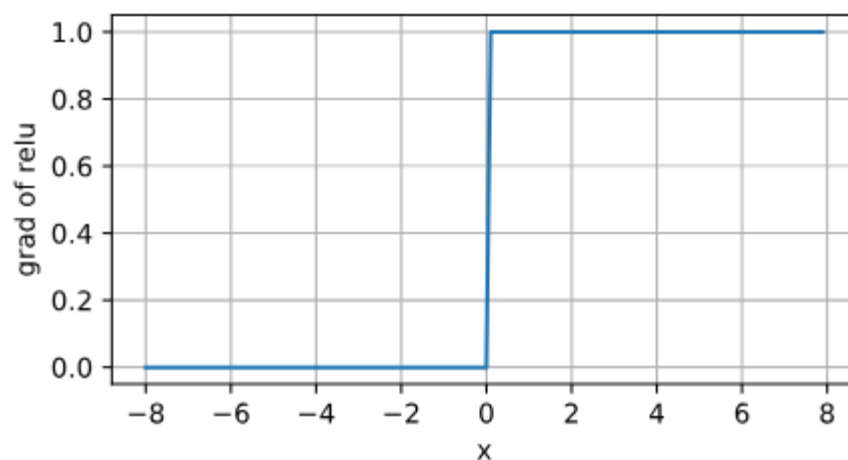
$$ReLU(x) = \max(x, 0)$$

```
x = tf.Variable(tf.range(-8.0, 8.0, 0.1), dtype=tf.float32)
y = tf.nn.relu(x)
d2l.plot(x.numpy(), y.numpy(), 'x', 'relu(x)', figsize=(5, 2.5))
```



입력이 음수이면 ReLU함수의 도함수는 0이고, 입력이 양수이면 ReLU함수의 도함수는 1이다. 입력이 0에서는 첨점이므로 미분이 불가능하다. 이럴때는 좌미분계수를 기본값으로 사용한다.

```
with tf.GradientTape() as t:  
    y = tf.nn.relu(x)  
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'grad of relu',  
         figsize=(5, 2.5))
```

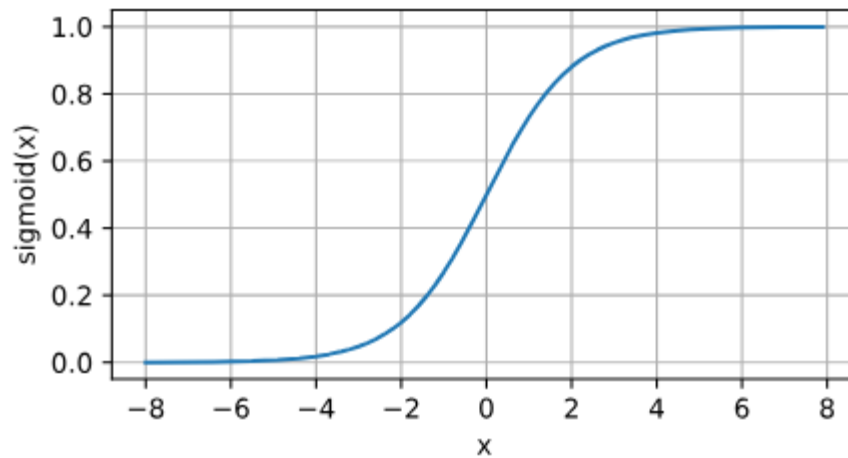


- 시그모이드 함수

시그모이드 함수에서는 x의 범위는  $(-\infty, \infty)$ 이고 y의 범위는  $(0, 1)$ 이다

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

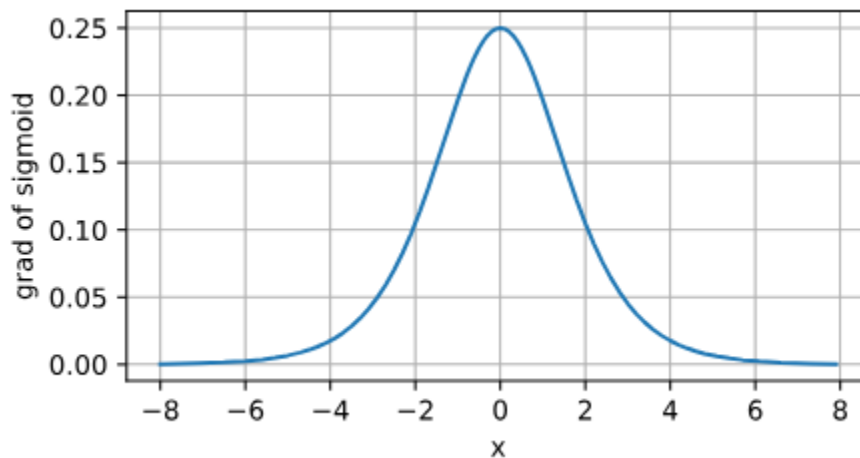
```
y = tf.nn.sigmoid(x)  
d2l.plot(x.numpy(), y.numpy(), 'x', 'sigmoid(x)', figsize=(5,
```



시그모이드 함수를 미분하면 다음과 같다.

$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

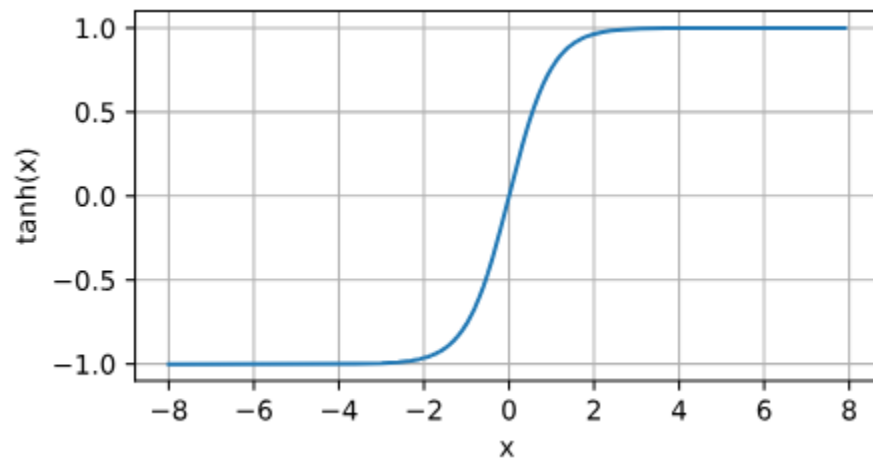
```
with tf.GradientTape() as t:
    y = tf.nn.sigmoid(x)
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'grad of sigmoid')
figsize=(5, 2.5))
```



- $\tanh(x)$

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

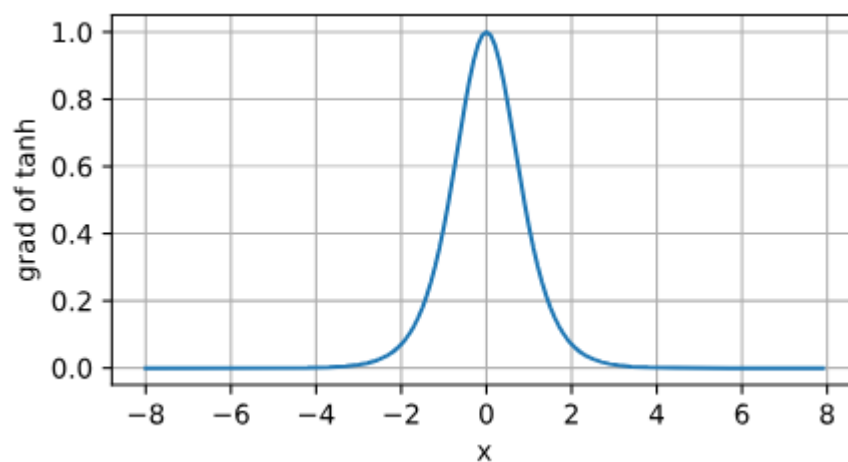
하이퍼탄젠트함수는 시그모이드와 비슷한 모양이지만 하이퍼 탄젠트는 원점을 중심으로 대칭이다.



$\tanh$ 함수의 미분은 다음과 같다.

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$$

```
with tf.GradientTape() as t:
    y = tf.nn.tanh(x)
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'grad of tanh',
         figsize=(5, 2.5))
```



## 5.2 Implementation of Multilayer Perceptrons

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

이러한 다층 퍼셉트론을 처음부터 구현해보겠다.

하나의 은닉층을 갖는 MLP를 만드는데 은닉층이 256개의 유닛을 갖도록 하겠다. 물론 레이어 수와 너비는 모두 조정이 가능하지만 일반적으로 2의 거듭제곱으로 나눌 수 있는 레이어 너비를 선택한다. 이는 메모리가 할당되고 처리되는 계산을 효율적으로 하기위해서이다.

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, ):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = tf.Variable(
            tf.random.normal((num_inputs, num_hiddens)) * sig
        self.b1 = tf.Variable(tf.zeros(num_hiddens))
        self.W2 = tf.Variable(
            tf.random.normal((num_hiddens, num_outputs)) * sig
        self.b2 = tf.Variable(tf.zeros(num_outputs))
```

위 코드는 tf.Variable모델 매개변수를 정의하는데 사용한다.

어떻게 작동하는지 확인하기 위해 내장함수호출대친 ReLU활성화를 직접 구현해 사용하겠다.

```
def relu(x):
    return tf.math.maximum(x, 0)
```

2차원으로 구현하기 위해 reshape를이용해 평면 벡터로 변환한다.

```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
```

```

X = tf.reshape(X, (-1, self.num_inputs))
H = relu(tf.matmul(X, self.W1) + self.b1)
return tf.matmul(H, self.W2) + self.b2

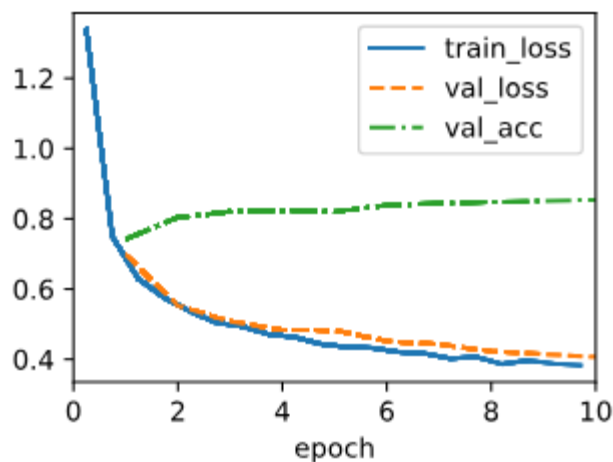
```

이 MLP의 훈련 루프는 소프트맥스 회귀와 정확히 동일 하다. 모델, 데이터 및 트레이너를 정의한 다음 fit모델과 데이터에 대한 메서드를 호출한다.

```

model = MLPScratch(num_inputs=784, num_outputs=10, num_hidden:
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)

```



API를 사용하면 위 코드보다 MLP를 더욱 간결하게 구현할 수 있다.

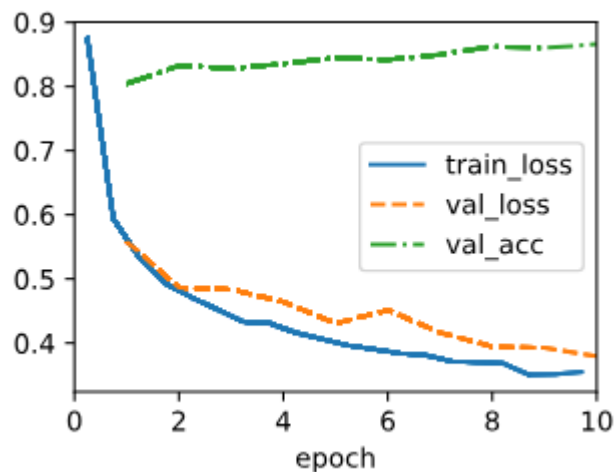
```

classMLP(d2l.Classifier):
def __init__(self, num_outputs, num_hiddens, lr):
    super().__init__()
    self.save_hyperparameters()
    self.net = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(num_hiddens, activation='re:
        tf.keras.layers.Dense(num_outputs)])

```

훈련루프는 동일하다.

```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



## 5.3 Forward Propagation, Backword Propagation, and Computational Graphs

순전파는 신경망에 대한 중간변수를 입력 계층에서 출력 계층까지 순서대로 계산하고 저장하는 것을 의미한다. 하나의 은닉층이 있는 신경망 메커니즘을 단계별로 알아보자면 다음과 같다. 입력은  $d$ 차원의 실수 공간

$$x \in \mathbb{R}^d$$

으로부터 선택되고, 편향항목은 생략하겠다. 중간 변수는 다음과 같이 정의 된다.

$$z = W^{(1)}x$$

$$W^{(1)} \in \mathbb{R}^{h \times d}$$



위는 은닉층의 가중치 파라미터이다. 중간변수  $z$ 를 활성화함수  $\phi$ 에 입력해서 벡터길이가  $h$ 인 은닉층변수를 얻는다.

$$h = \phi(z)$$

은닉 변수  $h$ 도 중간 변수이다. 출력층의 가중치  $W^{(2)}$ 만을 사용한다고 가정하면, 벡터의 길이가  $q$ 인 출력층의 변수를 다음과 같이 계산할 수 있다.

$$o = W^{(2)}h$$

손실함수를  $l$ 이라고 하고, 샘플 레이블을  $y$ 라 하면 하나의 데이터 샘플에 대한 손실값을 계산할 수 있다.

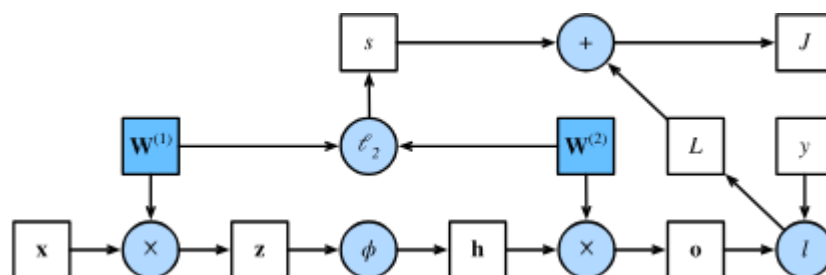
$$L = l(o, y)$$

L2놈 정규화의 정의에 따라 하이퍼파라미터  $\lambda$ 가 주어졌을 때 정규화항목은 다음과 같다.

$$s = \frac{\lambda}{2} (\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2)$$

여기서 행렬의 Frobenius표준은 다음과 같다. L2행렬을 벡터로 평탄화한 후 표준을 적용한다. 마지막으로 모델의 정규화된 손실은 다음과 같다.

$$J = L + s$$



## • 역전파

역전파는 신경망 매개변수의 기울기를 계산하는 방법을 의미한다.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \quad (5.3.8)$$

다음으로 출력 레이어의 변수에 대한 목적 함수의 기울기를 계산합니다. 체인 규칙에 따르면:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod} \left( \frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}} \right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \quad (5.3.9)$$

다음으로 두 매개변수에 대한 정규화 항의 기울기를 계산합니다.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \quad (5.3.10)$$

이제 그래디언트를 계산할 수 있습니다.  $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$  출력 레이어에 가장 가까운 모델 매개변수 중 하나입니다. 체인 규칙을 사용하면 다음이 생성됩니다.

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}} \right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \quad (5.3.11)$$

에 대한 그래디언트를 얻으려면  $\mathbf{W}^{(1)}$  출력 레이어를 따라 숨겨진 레이어까지 역전파를 계속해야 합니다. 은닉층 출력에 대한 그래디언트  $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$ 에 의해 주어진다

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}} \right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}. \quad (5.3.12)$$

활성화 함수부터  $\phi$  요소별로 적용하여 기울기를 계산합니다.  $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$  중간변수의  $\mathbf{z}$  요소별 곱셈 연산자를 사용해야 합니다.  $\odot$ :

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}). \quad (5.3.13)$$

마지막으로 그래디언트를 얻을 수 있습니다.  $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  입력 레이어에 가장 가까운 모델 매개변수 중 하나입니다. 체인 규칙에 따르면, 우리는 다음을 얻습니다.

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}} \right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \quad (5.3.14)$$

신경망을 훈련할 때 순전파와 역전파는 서로 의존한다. 특히 순전파는 계산 그래프를 종속성 방향으로 탐색하고 해당 경로의 모든 변수를 계산한다. 그런다음 그래프의 계산 순서가 반전되는 역전파가 사용된다.

## 5.4 Numerical Stability and Initialization

```
%matplotlib inline
import tensorflow as tf
from d2l import tensorflow as d2l
```

이번 절에서는 파라미터의 초기화와, 활성화 함수선택에 대한 내용에 대해 설명한다. 만약 이러한 이슈들을 중요하게 생각하지 않을 경우 그래디언트 소실 또는 폭발이 발생할 수 있다.

- 그래디언트 소실과 폭발

입력이  $x$ , 출력이  $o$ 이고  $d$ 층을 갖는 딥 네트워크를 예로들자면 각 층은 다음을 만족한다.

$$h^{t+1} = f_t(h^t) \text{이고, 따라서 } o = f_d \circ \dots \circ f_1(x)$$

모든 활성화들과 입력들이 벡터인 경우,  $t$ 번째 층의 함수  $f_t$ 와 관련된 파라미터  $W_t$ 의 임의의 세트에 대한  $o$ 의 그래디언트는 다음과 같이 표현된다.

$$\partial_{W_t} o = \underbrace{\partial_{h^{d-1}} h^d}_{:-M_d} \cdot \dots \cdot \underbrace{\partial_{h^t} h^{t+1}}_{:-M_t} \underbrace{\partial_{h^t} h^t}_{:-V_t}$$

$d-1$ 부터  $d$ 까지의 행렬  $M_d \dots M_t$ 와 그래디언트 벡터  $v_t$ 의 곱이다. 너무 많은 확률을 곱할 때 산술적인 언더플로우를 경험할 때와 비슷한 상황이 발생하는데 이것을 로그공간으로 전환시켜 완화할 수 있다.

어떤 행렬을 곱하다보면 그 값이 아주작아지거나 아주커질 수도 있다. 이것은 수치적인 문제 뿐만이 아닌 최적화 알고리즘이 수렴되지 않을 수 있다는 것을 말한다. 즉 아주 큰 그래디언트가 되거나 너무 조금씩 업데이트가 된다. 따라서 파라미터가 너무 커져버리거나 소실될 가능성이 생긴다.

예시로 하나의 행렬을 선택한 후 100개의 가우시안 랜덤 행렬을 선택해서 모두 곱하겠다.

```
%matplotlib inline
import mxnet as mx
from mxnet import nd, autograd
from matplotlib import pyplot as plt

M = nd.random.normal(shape=(4,4))
print('A single matrix', M)
for i in range(100):
    M = nd.dot(M, nd.random.normal(shape=(4,4)))

print('After multiplying 100 matrices', M)
```

```
A single matrix
[[ 2.2122064  0.7740038  1.0434405  1.1839255 ]
 [ 1.8917114 -1.2347414 -1.771029  -0.45138445]
 [ 0.57938355 -1.856082  -1.9768796 -0.20801921]
 [ 0.2444218  -0.03716067 -0.48774993 -0.02261727]]
<NDArray 4x4 @cpu(0)>
After multiplying 100 matrices
[[ 3.1575275e+20 -5.0052276e+19  2.0565092e+21 -2.3741922e+20]
 [-4.6332600e+20  7.3445046e+19 -3.0176513e+21  3.4838066e+20]
```

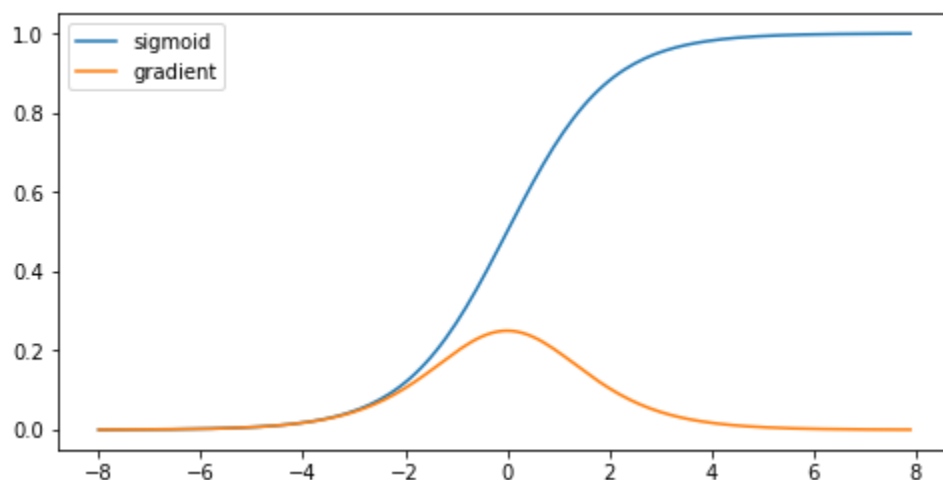
```
[ -5.8487235e+20  9.2711797e+19 -3.8092853e+21  4.3977330e+20
 [-6.2947415e+19  9.9783660e+18 -4.0997977e+20  4.7331174e+19
 <NDArray 4x4 @cpu(0)>
```

결과로 우리가 선택한 스케일링으로 인해 행렬의 곱이 너무 커지게 된다.

반대로 그래디언트 소실의 예시로 시그모이드 활성화 함수를 비선형 활성화 함수로 사용했을 때 이다.

```
x = nd.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = x.sigmoid()
y.backward()

plt.figure(figsize=(8, 4))
plt.plot(x.asnumpy(), y.asnumpy())
plt.plot(x.asnumpy(), x.grad.asnumpy())
plt.legend(['sigmoid', 'gradient'])
plt.show()
```



위 그림처럼 그래디언트는 아주 큰수나 아주 작은 수에서 소멸한다. 체인룰로 인해 활성화들이 범위에 들어가지 않으면 소멸될 수 있다는 것을 의미한다.

따라서 이것을 해결하기 위해 ReLU함수가 활성화로 설계할 때 기본선택이 됐다.

두 개의 은닉 유닛  $h_1, h_2$ 를 갖는 한개의 은닉층을 가지고 있는 딥네트워킹에서 첫번째 층의 가중치  $W_1$ 를 뒤집고, 두번째 층의 결과도 뒤집을 경우, 동일한 함수를 얻게 된다.

- 기본 초기화

이전에 `net.initialize(init.Normal(sigma=0.01))` 을 이용해서 가중치의 초기값으로 정규분포에서 임의의 수를 선택하는 방법을 사용했다. `net.initialize()` 를 호출하는 경우 MXNet은 기본 랜덤 초기화 방법을 적용한다.

- Xavier초기화

어떤 층의 은닉 유닛  $h_i$ 에 적용된 활성화의 범주 분포를 살펴보자면 다음과 같이 계산된다.

$$h_i = \sum_{j=1}^{n_m} W_{ij} x_j$$

가중치  $W_{ij}$ 들은 같은 분포에서 서로 독립적으로 선택된다. 이 분포는 평균이 0이고 분산이  $\sigma^2$ 라고 가정하고 층의 입력  $x_j$ 를 제어할 수 있는 방법이 없지만, 그 값들의 평균이 0이고 분산이  $\gamma^2$ 이고,  $W$ 과는 독립적이라는 가정을 하자면  $h_i$  평균과 분산을 다음과 같이 계산할 수 있다.

$$\begin{aligned} \mathbf{E}[h_i] &= \sum_{j=1}^{n_{in}} \mathbf{E}[W_{ij} x_j] = 0 \\ \mathbf{E}[h_i^2] &= \sum_{j=1}^{n_{in}} \mathbf{E}[W_{ij}^2 x_j^2] \\ &= \sum_{j=1}^{n_{in}} \mathbf{E}[W_{ij}^2] \mathbf{E}[x_j^2] \\ &= n_{in} \sigma^2 \gamma^2 \end{aligned}$$

$n_{in} * \sigma^2 = 1$ 을 적용하면 분산을 고정시킬 수 있다.  $n_{out} \sigma^2 = 1$ 이 아닐 경우에는 그래디언트 분산이 너무 커질 수 있다. 즉 두 조건을 동시에 만족시킬 수 없지만 다음 조건은 쉽

게 만족시킬 수 있다

$$\frac{1}{2}(n_{in} + n_{out})\sigma^2 = 1 \text{ 또는 동일하게 } \sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

이것이 Xavier 초기화의 기본이 되는 논리이다. ▮