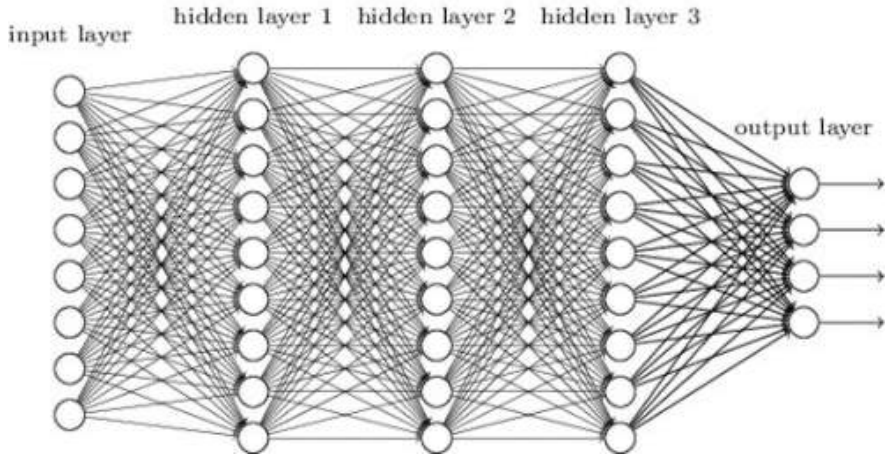
Deep Neural Network

Deep Neural Network



- DNN 학습 (BP 알고리즘) 의 문제점
  - 학습 오차 문제 (Error)
  - 학습 속도 문제 (Time)
  - 과적합 문제 (Overfitting)

  ➢ 1990~2000년대 적용 안됨

Weight Update

- 신경망 학습이란?
  - Loss function 을 최소화하는 weight 를 찾는 문제 (optimization problem)

- 신경망 학습 방법
  - weight 초기값을 임의로 설정하고, weight 를 반복적으로 update (improvement)
  - Weight 변화량은 loss function 의 기울기 크기에 비례 (gradient descent)

1) Stochastic Gradient Descent (SGD)
2) AdaGrad (Adaptive Gradient)
3) Adam

Stochastic Gradient Descent

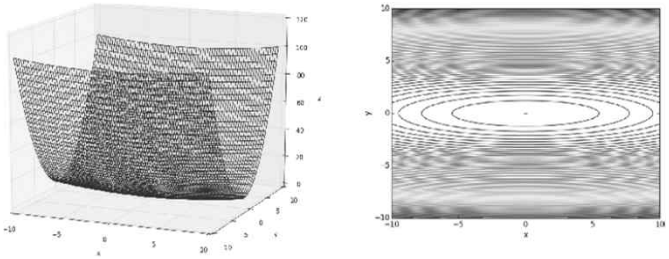
- Algorithm

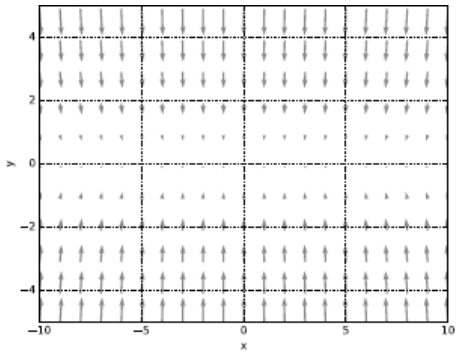
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial L}{\partial \boldsymbol{W}}$$

η: learning rate

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

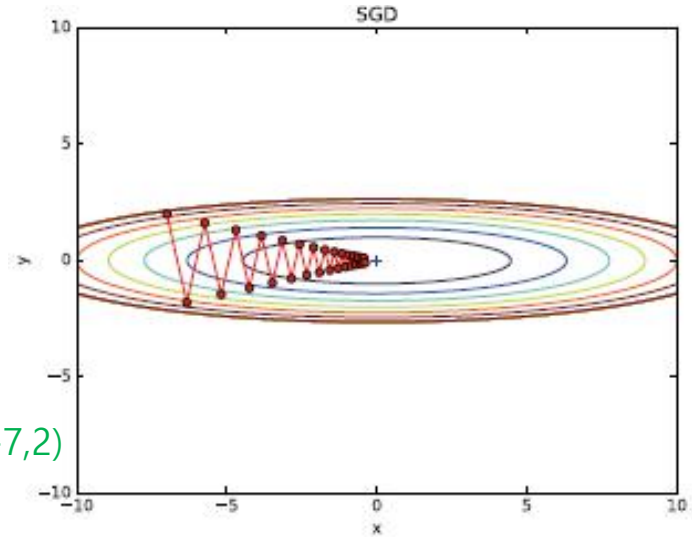
(예) $f(x,y) = \frac{1}{20}x^2 + y^2$





기울기(gradient)

Y축 방향은 가파르고 x축 방향은 완만
=> 지그재그 이동



초기값 (-7,2)

단점: 비등방성 (anisotropy) 함수에 대한 탐색경로가 비효율적임

Momentum

- Algorithm

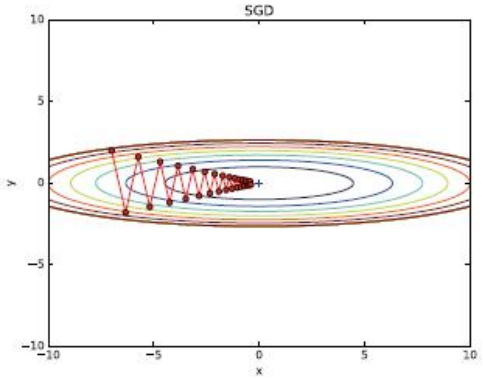
$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\frac{\partial L}{\partial \mathbf{W}}$$
$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

α: coefficient of momentum

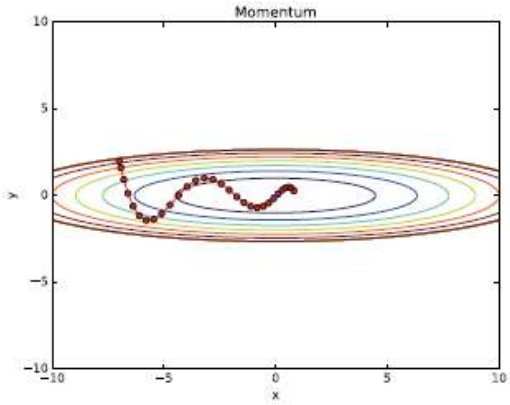
$$\nu_1 = -G_1$$
$$\nu_2 = -0.9 * G_1 - G_2$$
$$\nu_3 = -0.9 * (0.9 * G_1 - G_2) - G3 = -0.81 * (G_1) - (0.9) * G_2 - G_3$$

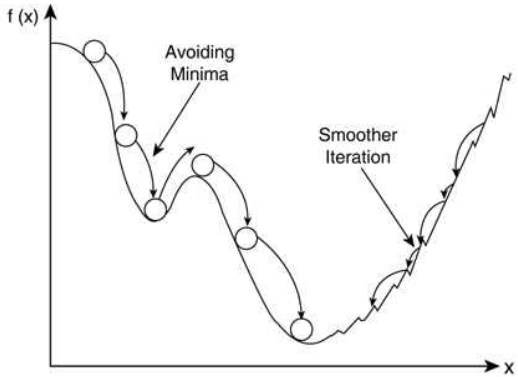
- 효과
  - 과거 업데이트 값도 현재 업데이트에 반영
  - 관성효과로 빠른 이동 가능
  - 지그재그 탐색 완화
  - Local minima 를 벗어나는 효과 기대

α= 0.0



α= 0.9





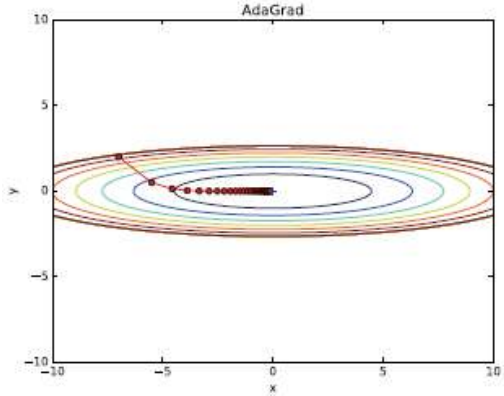
AdaGrad

- Adaptive Gradient (2011)
  - 학습을 진행하면서 learning rate (학습률) 을 점차 감소시키는 방법

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

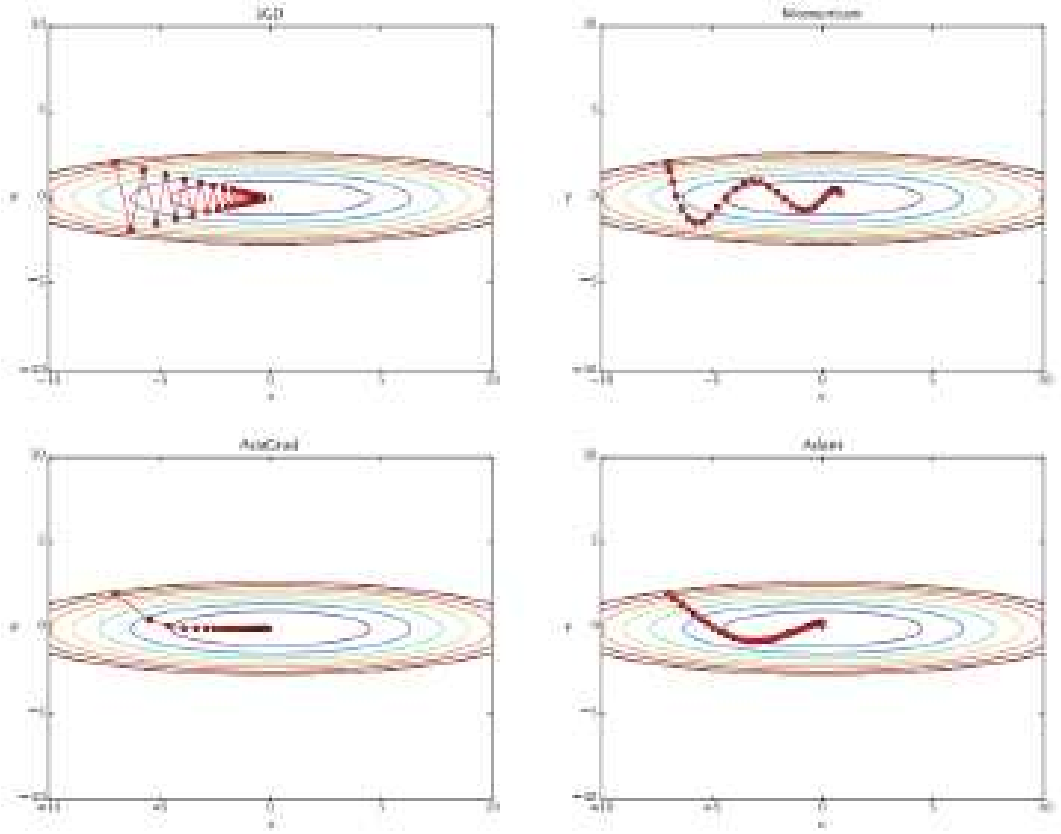
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

    - 지금까지 적게 변화한 weight 변화량 (step size) 을 크게함 => 빠른 이동
    - 지금까지 많이 변화한 weight 변화량 (step sizs) 을 작게함=> 미세 조정
  - 학습이 오래 진행되면 step size 가 너무 줄어들어 update 중단
    - RMSProp
    - AdaDelta



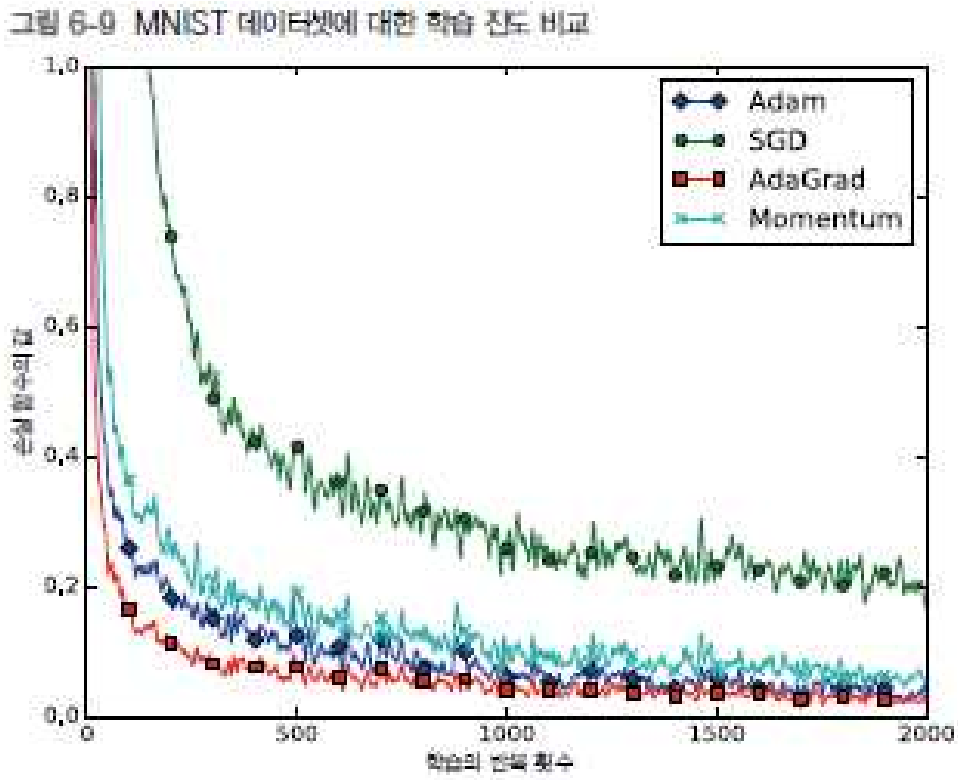
Adam

- Adaptive Momentum Estimation (2015)
  - Momentum + AdaGrad (RMSProp) 융합



Weight Update

- Optimization 방법 비교



Gradient Trouble

- Gradient Vanishing (소실)
  - gradient 값이 점점 작아짐
  - Weight update 중단 => 학습 오차가 떨어지지 않음

- Gradient Exploding (폭주)
  - gradient 값이 점점 커짐
  - 비정상적으로 큰 값으로 Weight update => 발산

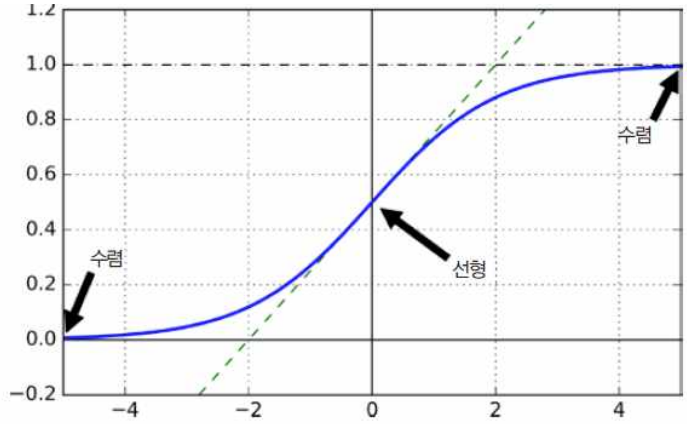
$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

$$= -\eta \frac{\partial E}{\partial y} \frac{\partial y}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

Gradient Trouble

- 원인 (X.Glorot, Y.Bengio, 2010)
  1) Activation function: sigmoid function
  2) Weight initialization: 평균=0, 분산=1 정규분포

  – 각 층에서 출력의 분산이 입력의 분산보다 큼
    → 상위 층으로 갈수록 활성화 함수값이 0 또는 1로 수렴
    → 기울기 (gradient) 가 0 에 가까워짐
    → 상위층 부터 역전파 진행 시 하위층에는 오류 전파 안됨
    → weight update stop
    → 학습오차가 줄어들지 않음



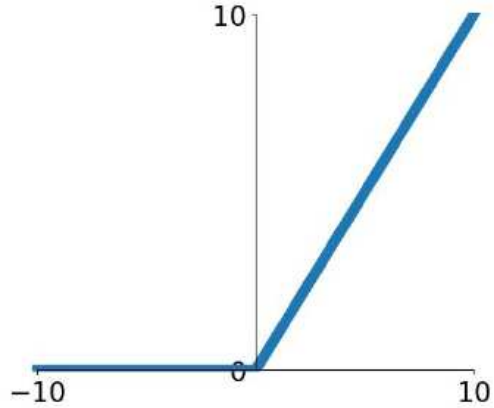
Gradient Trouble

- 해결 방법
  1) Activation Function: ReLU 및 그 변종
  2) Weight Initialization: Xavier, He 등
  3) Batch normalization

Activation Function

- ReLU
  - Rectified Linear Unit
  - F(x) = max (0, x)

  - 장점
    - Saturation 이 없다
    - Very computationally efficient
    - Converges much faster than sigmoid/tanh



Weight Initialziation

- 은닉층의 출력값 (활성화값) 분포 (histogram)

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.random.randn(1000, 100) # 1000개의 데이터
node_num = 100          # 각 은닉층의 노드(뉴런) 수
hidden_layer_size = 5   # 은닉층이 5개
activations = {}        # 이곳에 활성화 결과(활성화값)를 저장

for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    w = np.random.randn(node_num, node_num) * 1
    a = np.dot(x, w)
    z = sigmoid(a)
    activations[i] = z

# 히스토그램 그리기
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

- Input layer: 1000 nodes
- 5 hidden layers: 100 nodes / layer
- Activation function = sigmoid

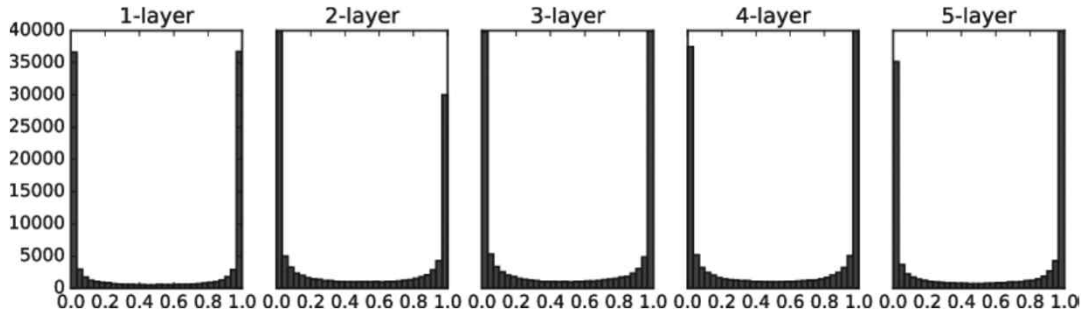
- Weight 초기값: 평균 0 인 정규분포
  분산 = 1 or 0.01

```
# w = np.random.randn(node_num, node_num) * 1
w = np.random.randn(node_num, node_num) * 0.01
```

Weight Initialziation

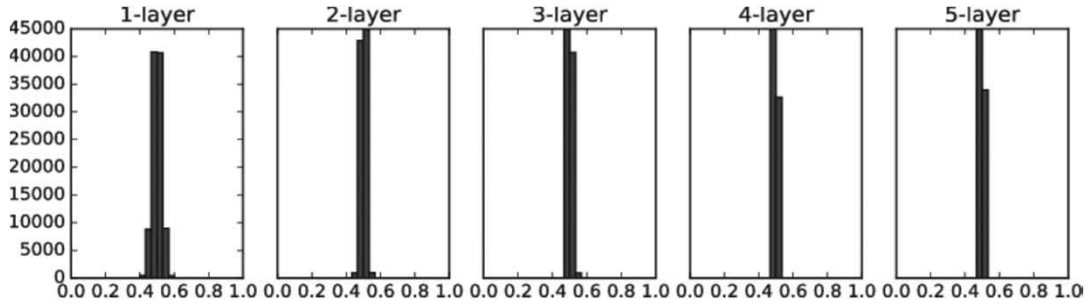
- Weight 를 정규분포로 초기화할 때의 은닉층의 출력값 분포 (histogram)

표준편차=1



0 또는 1 에 집중
→ 기울기 소실

표준편차=0.01

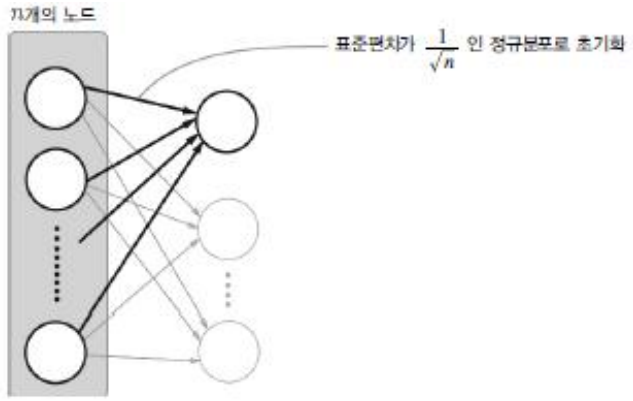


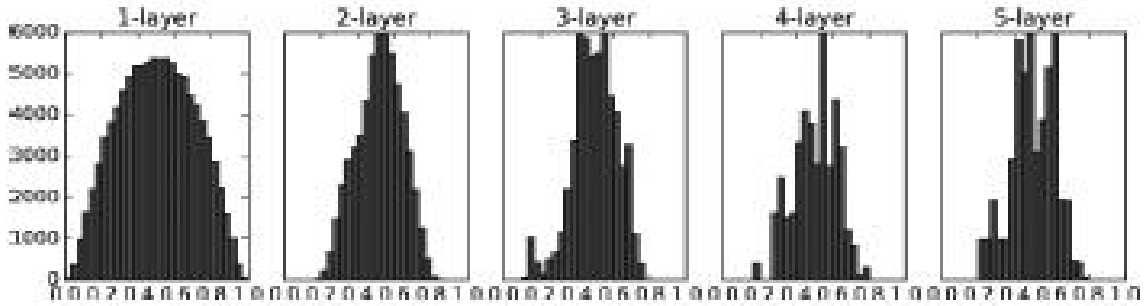
0.5 에 집중
→ 모든 neuron 이 유사값 출력
→ 표현력 제한

Weight Initialziation

- Xavier (Glorot) 초기화 (2010)
  - Activation 함수: Sigmoid, tanh function
  - $\sigma = \frac{1}{\sqrt{n}}$, $n = fan_{in}$ = 앞층의 노드 수

```
node_num = 100 # 앞 층의 노드 수
w = np.random.randn(node_num, node_num) / np.sqrt(node_num)
```



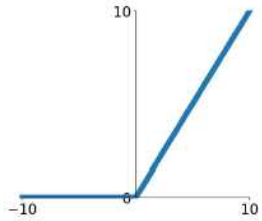


Weight Initialziation

- He 초기화 (2015)
  - Activation function : ReLU 및 변종 들
  - $\sigma = \frac{2}{\sqrt{n}}$, $n = fan_{in}$ = 앞층의 노드 수

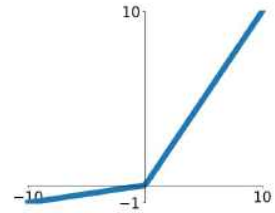
ReLU

$$\max(0, x)$$



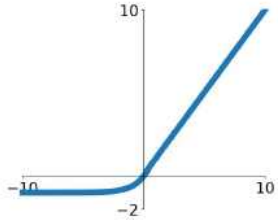
Leaky ReLU

$$\max(0.1x, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

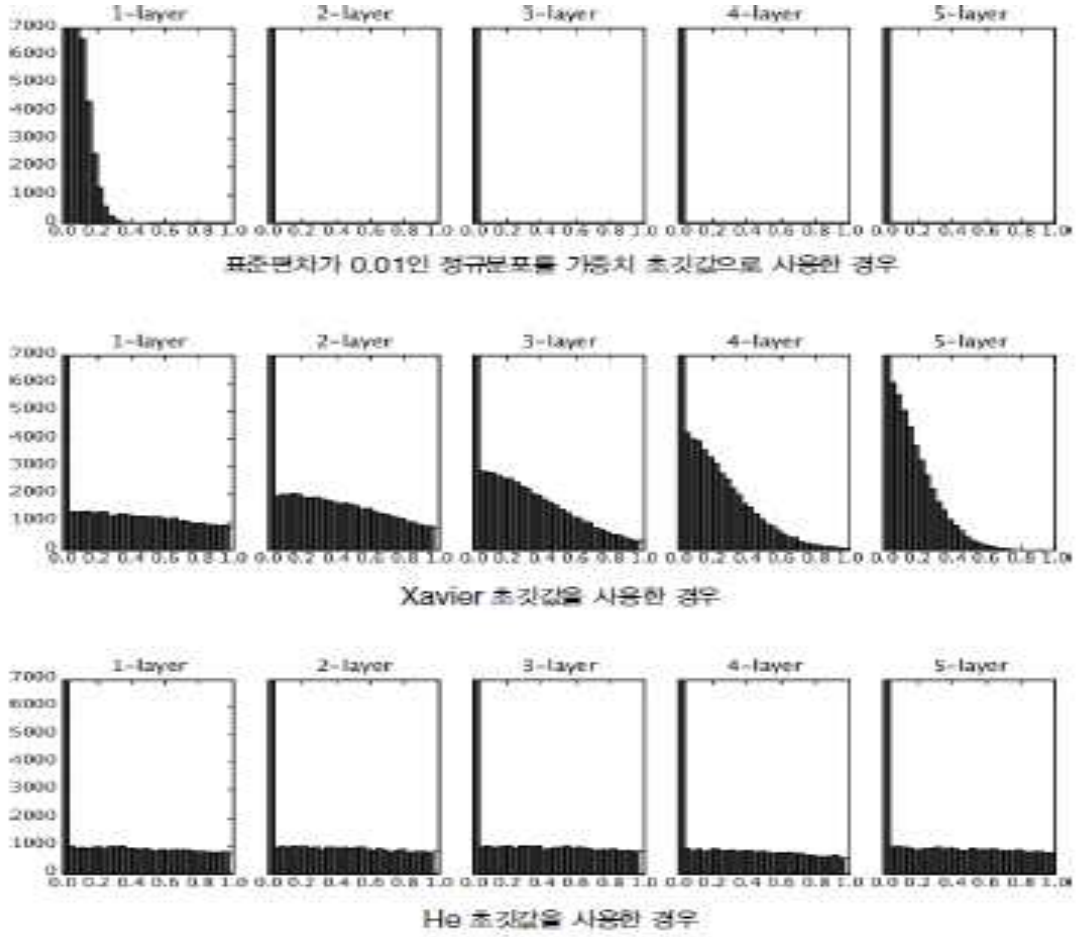


Exponential linear unit

Weight Initialziation

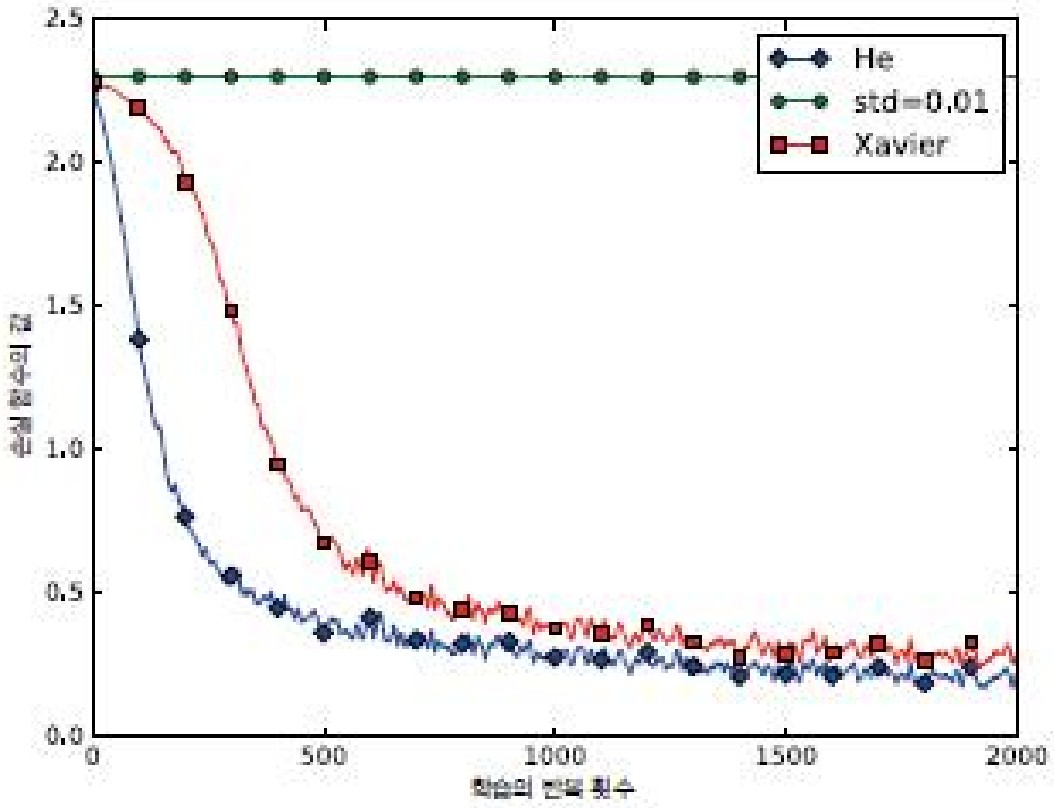
- He 초기화

ReLU 사용 시 weight 초기값에 따른 은닉층 출력값 분포



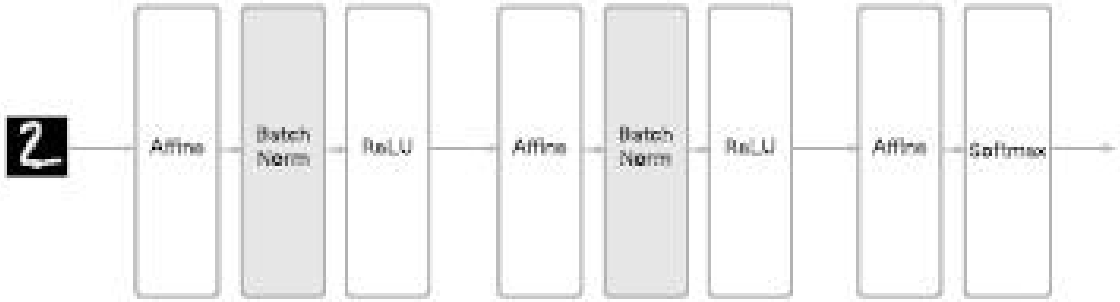
Weight Initialziation

- MNIST 데이터셋 적용
  - ReLU function + He initialization



Batch Normalization

- Batch Normalization (S.Loffe & C.Szegedy, 2015)
  - Gradient vanishing/Exploding 방지 목적
  - Activation 함수 통과 전 또는 후 데이터 '정규화' 과정 추가



  - Mini-batch 단위로 정규화: 평균 = 0, 분산 = 1

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$

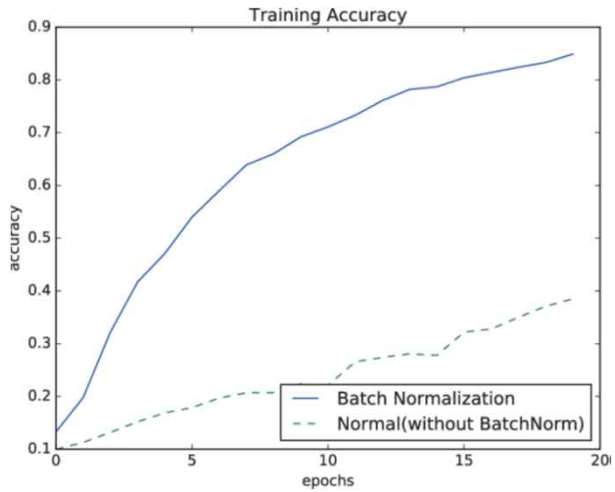
$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

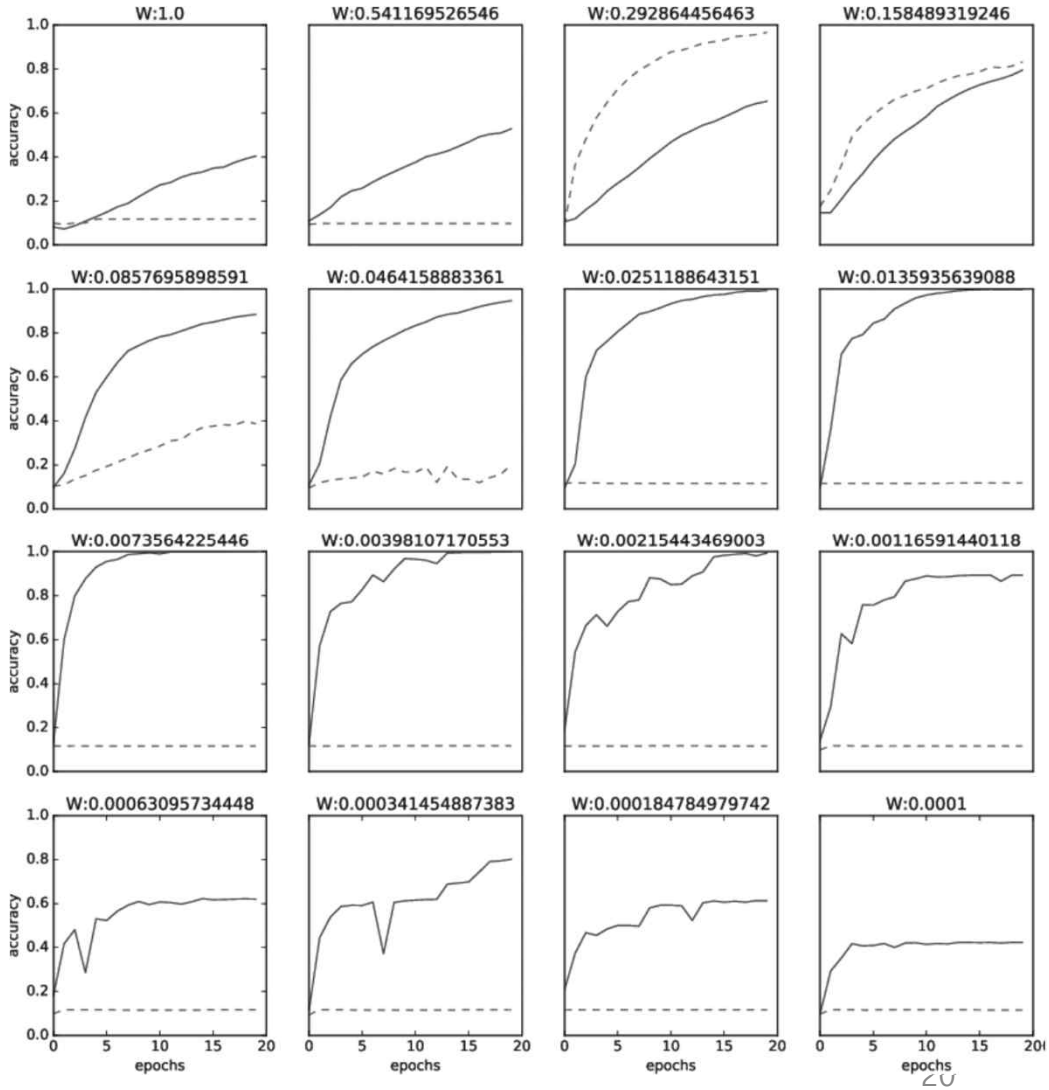
$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

γ: scaling, β: shift

Batch Normalization

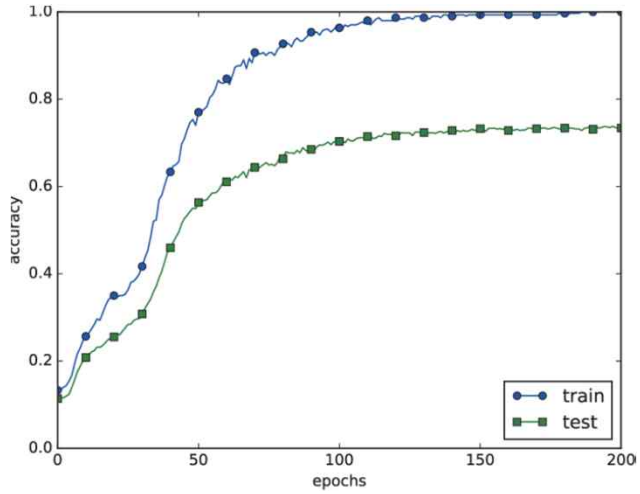
- 효과
  - 학습속도 향상
  - Weight 초기값 영향이 적음





Overfitting

- Overfitting (과적합)
  - 신경망이 훈련 데이터에 지나치게 적응. 그 외의 데이터에는 대응하지 못하는 상태
  - 원인
    - 매개변수가 많고 표현력이 높은 모델
    - 훈련데이터가 적음
  - 예
    - 7 layer network
    - 100 neurons/layer, ReLU
    - MNIST 데이터셋 (60,000개)
      - training: 300개
  - 해결방안
    - Weight decay
    - Dropout



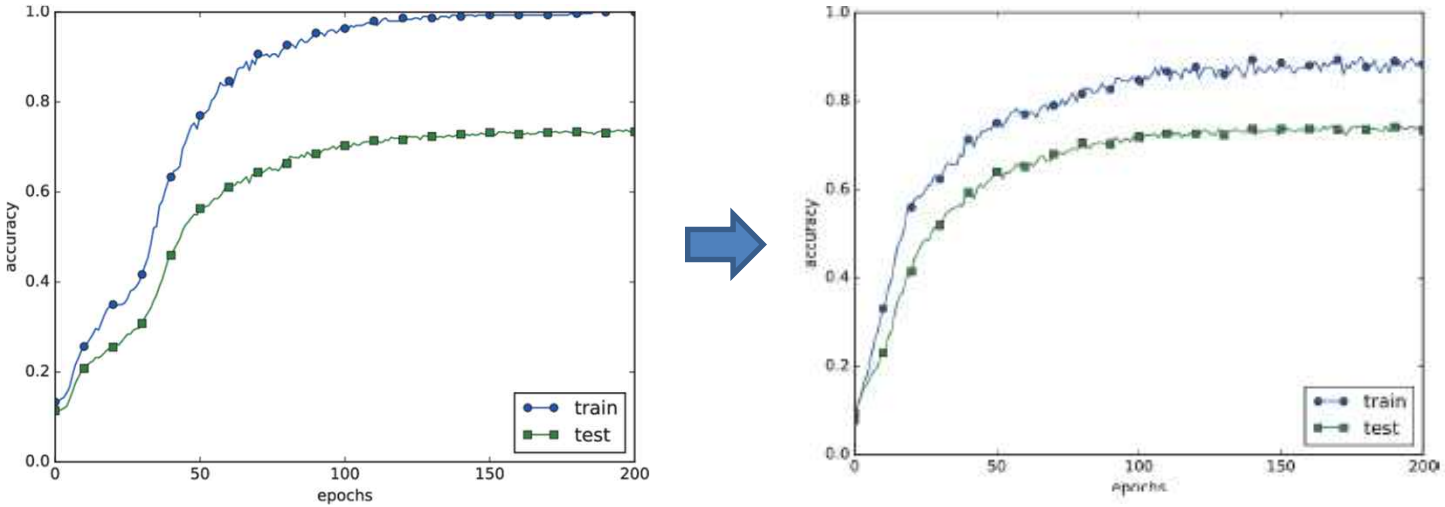
Weight Decay

- Weight Decay (가중치 감소)
  - Weight 값이 커지는 것을 억제 => overfitting 억제 효과
  - Loss function 에 weight norm 추가

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2 + \frac{1}{2}\lambda W^2$$

$$W = (w_1, w_2, \ldots, w_n)$$

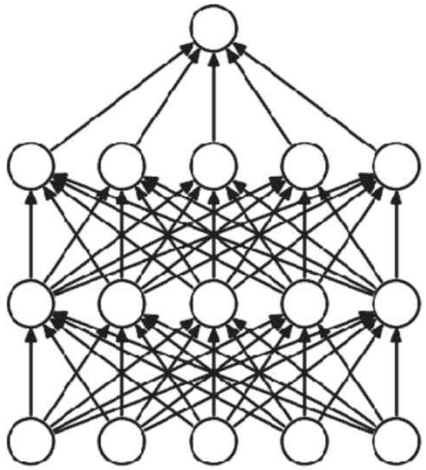
L2 norm : $\sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$
L1 norm : $|w_1| + |w_2| + \cdots + |w_n|$
L∞ norm : max $(|w_1|, |w_2|, \ldots, |w_n|)$

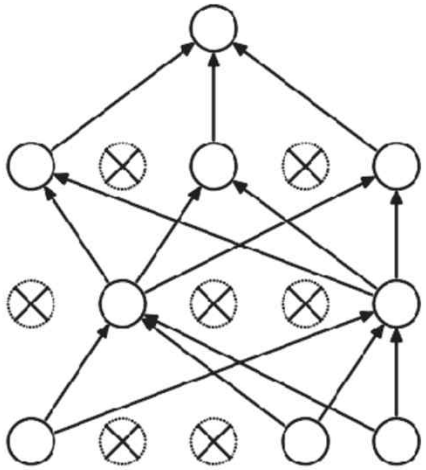


Dropout

- Dropout
  - Training 시
    - Dropout 비율 (dropout_ratio) 에 해당하는 neuron 은 학습에서 삭제
  - Test 시
    - 각 neuron 의 출력에 training 시 삭제 안 한 비율을 곱하여 출력

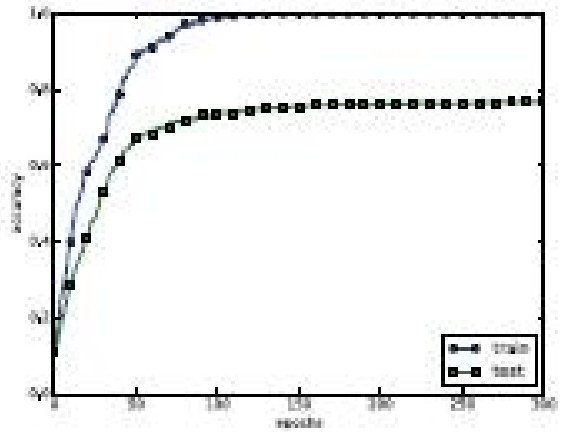
```
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
```

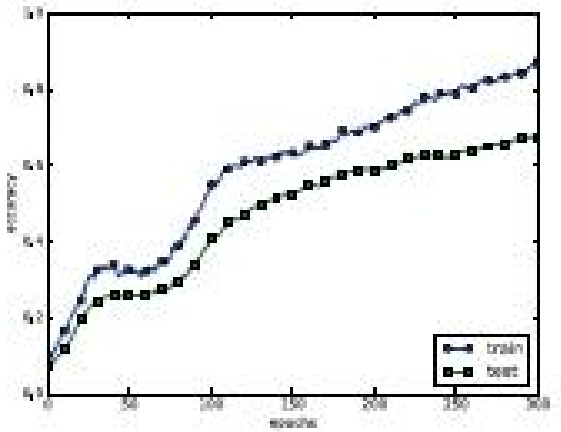




Dropout

- Dropout 의 효과
  - Ensemble learning 효과
    - 개별적으로 학습시킨 여러 개의 모델 출력 활용
  - Tabu search 효과
    - Hill-climbing (Gradient descent) 에 의한 weight 탐색 개선 => 탐색 공간 다변화





dropout_ratio=0.15

Hyperparameter Optimization

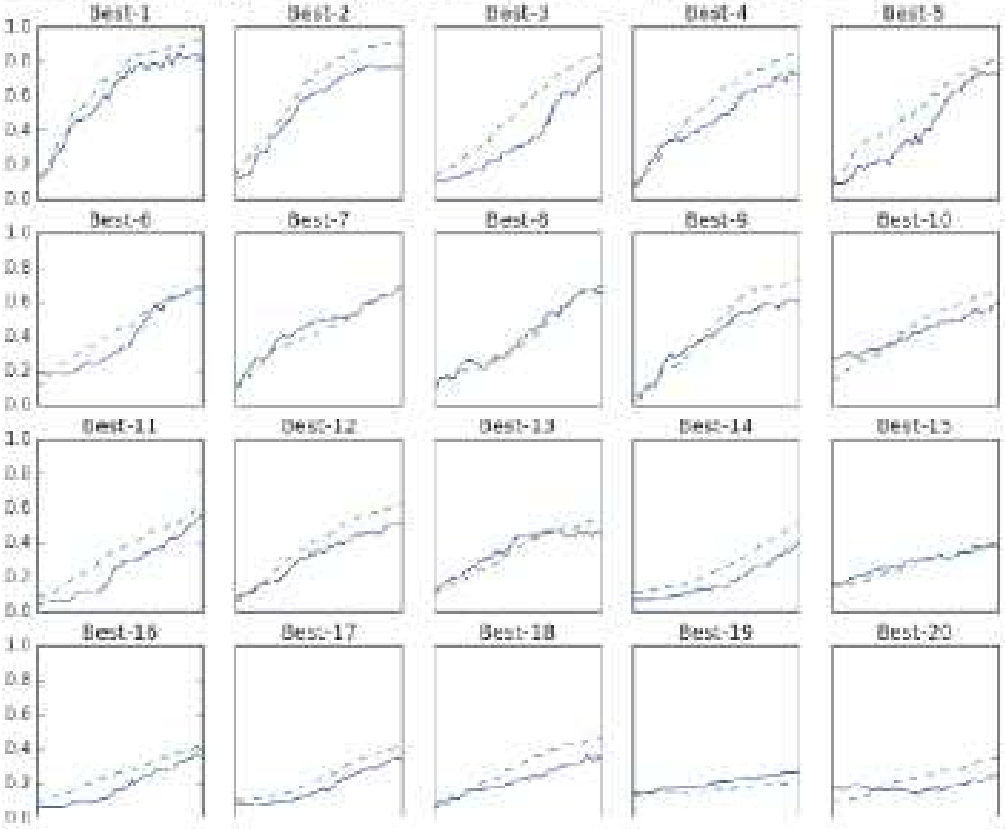
- Hyperparameter
  - Layer 수, neuron 수, batch size, …
  - Learning rate, momentum coefficient, …
  - Weight decay, dropout ratio, …

- Training data vs. validation data vs. test data
  - Training data: weight 학습용
  - Validation data: hyperparameter 검증용
  - Teat data: 성능평가용

Hyperparameter Optimization

- Example

그림 6-24 실선은 검증 데이터에 대한 정확도, 점선은 훈련 데이터에 대한 정확도




```
weight_decay = 10 ** np.random.uniform(-8, -4)
lr = 10 ** np.random.uniform(-6, -2)
```

```
Best-1 (val acc:0.83) | lr:0.0092, weight decay:3.86e-07
Best-2 (val acc:0.78) | lr:0.00956, weight decay:6.04e-07
Best-3 (val acc:0.77) | lr:0.00571, weight decay:1.27e-06
Best-4 (val acc:0.74) | lr:0.00626, weight decay:1.43e-05
Best-5 (val acc:0.73) | lr:0.0052, weight decay:8.97e-06
```