Chap06.

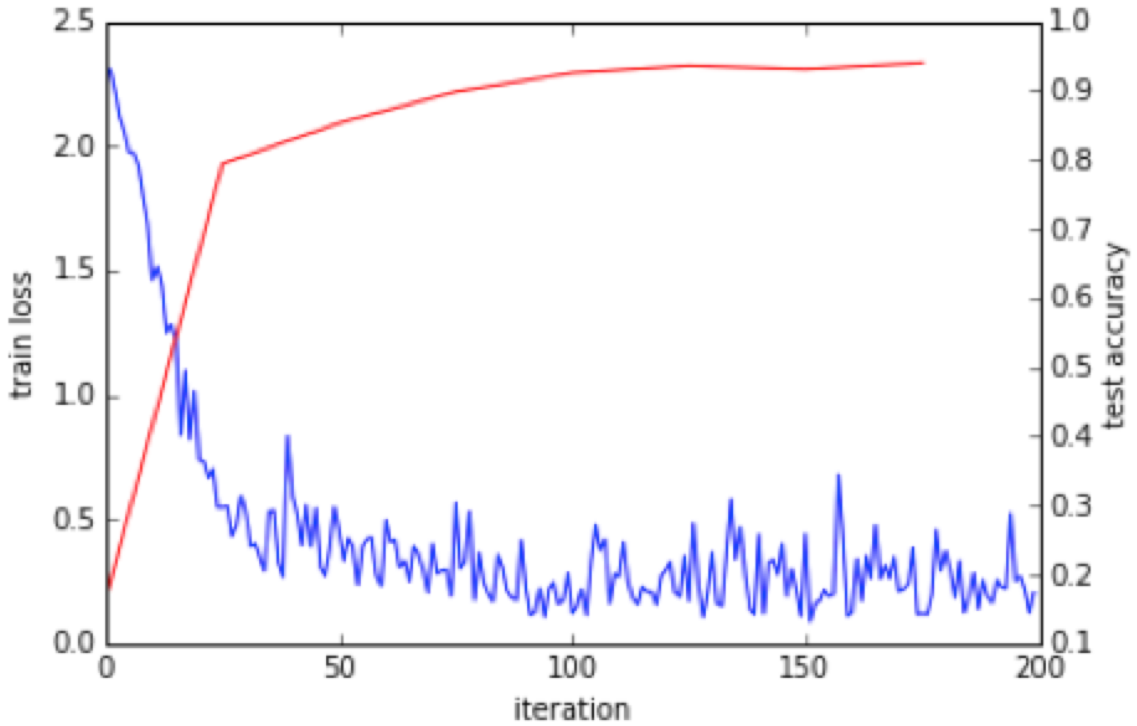
학습 관련 기술들

최종현

Overview

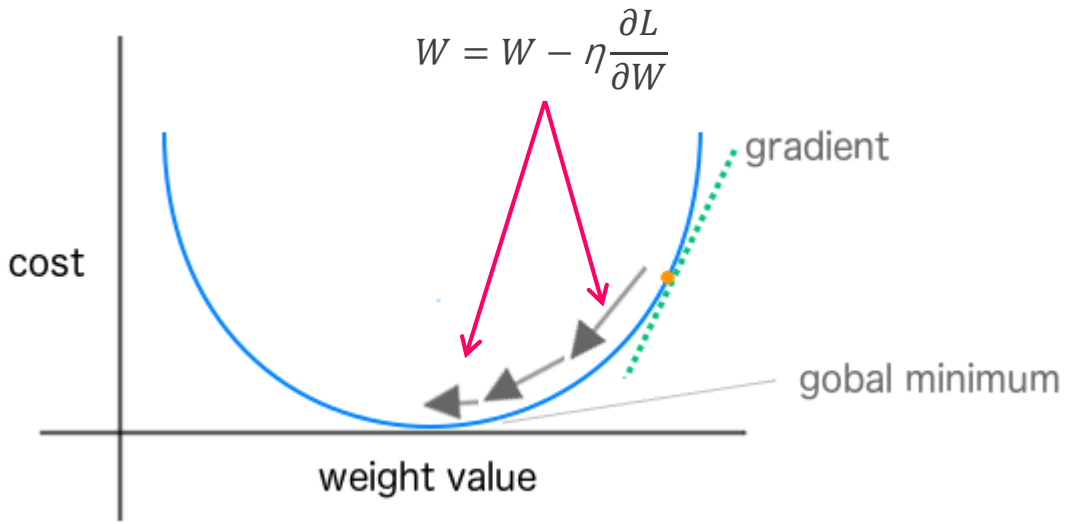
- ✓ 신경망 학습의 핵심 개념들을 공부
- ✓ 가중치 파라미터(w)의 최적값을 탐색하는 최적화 방법(Optimization), 가중치 초기값, 하이퍼 파라미터 설정 방법
- ✓ Overfitting(과적합)의 대응책인 가중치 감소(Weight decay, Regularization), 드롭아웃(DropOut) 등

딥러닝 학습의 효율과 정확도를 높일 수 있다!



6.1 매개변수 갱신

- ✓ 신경망 학습의 목적은 **손실함수의 값을 최대한 낮추는** 파라미터(w, b)를 찾는 것
- ✓ 즉, 최적의 매개변수(파라미터)를 찾는 문제 → *최적화(optimization)*
- ✓ But, 신경망에서는 수식을 풀어 한번에 최적의 값을 찾을 수 있는 방법은 없음...
  → 층이 깊어질 수록 파라미터가 엄청나게 많아지므로
- ✓ 앞에서는 **확률적 경사 하강법**(SGD)를 이용해서 가중치 파라미터의 값을 여러 번 갱신하여 최적의 값을 구함



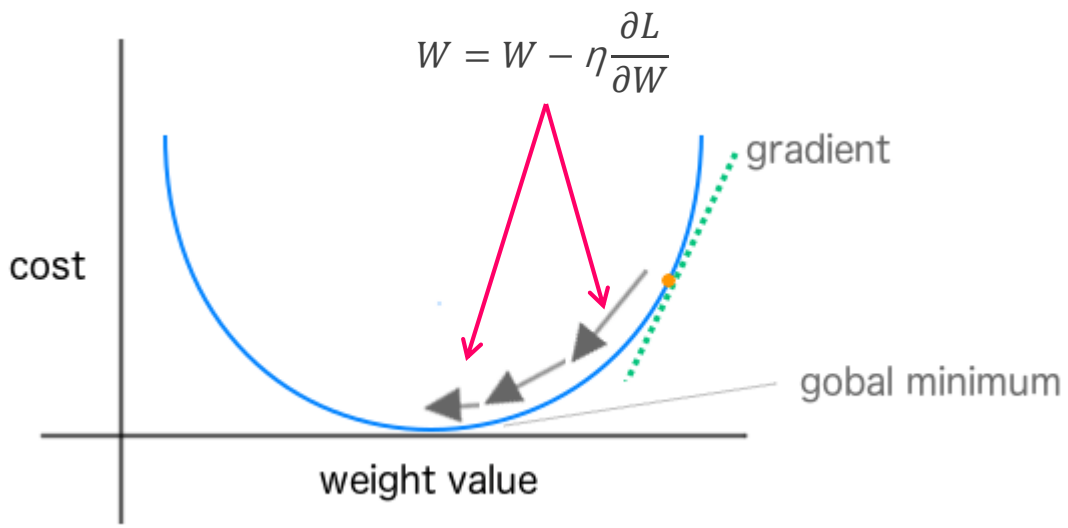
6.1 매개변수 갱신 - cont'd

6.1.1 SGD(확률적 경사 하강법)

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

W 에 대한
손실함수(L)의 기울기

```
class SGD:

    """확률적 경사 하강법(Stochastic Gradient Descent)"""

    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]
```

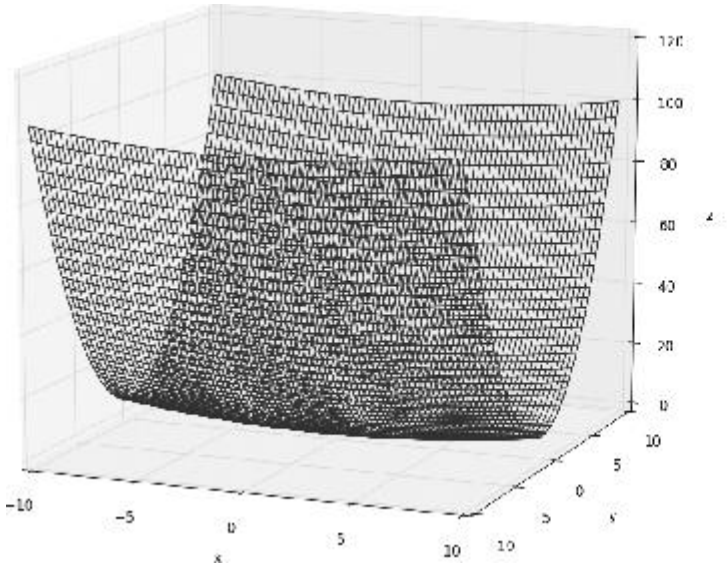


6.1 매개변수 갱신 - cont'd

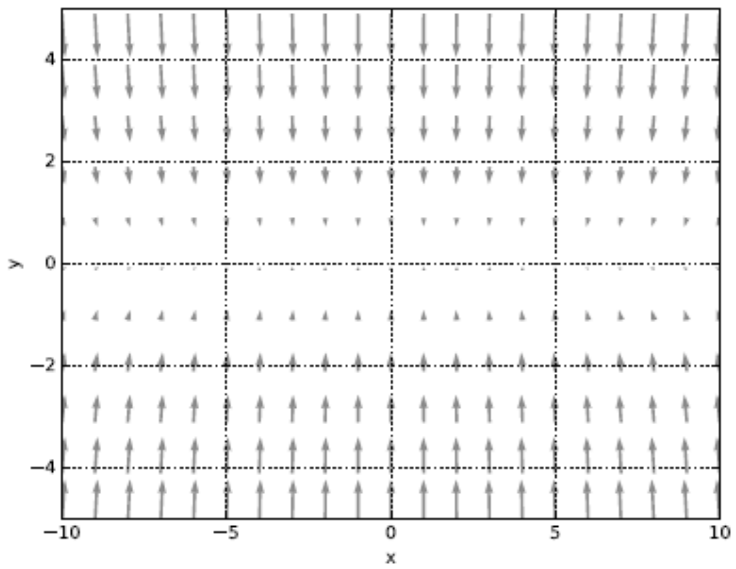
6.1.2 SGD의 단점

✓ 비등방성(anisotropy) 함수 즉, 방향에 따라 기울기가 달라지는 함수에서는 탐색 경로가 비효율적임

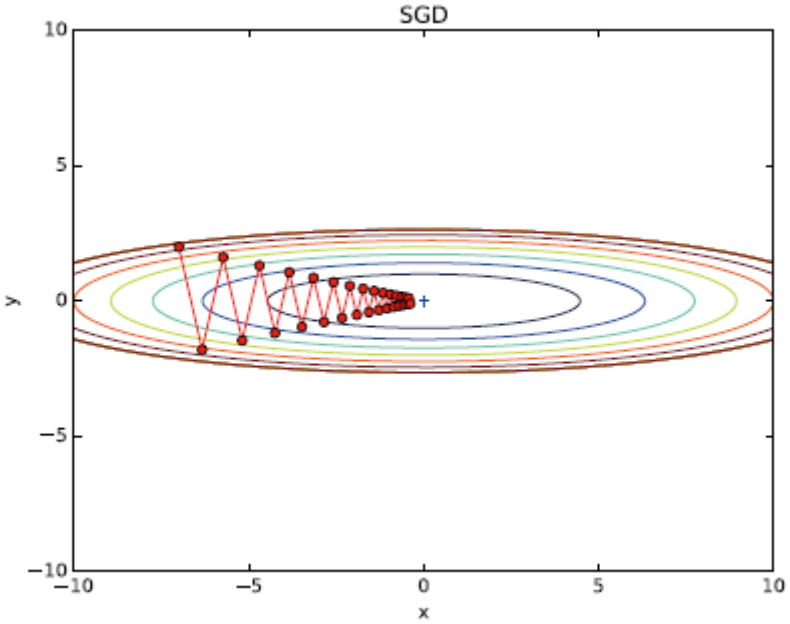
$$f(x, y) = \frac{1}{20}x^2 + y^2$$



그래프



기울기



갱신 경로

6.1 매개변수 갱신 - cont'd

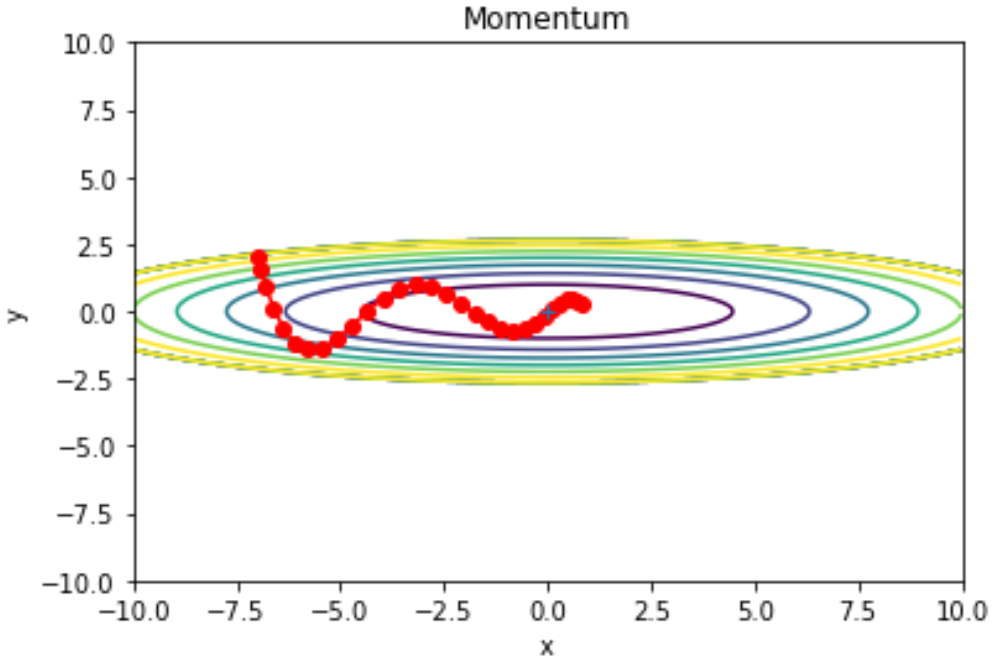
6.1.3 모멘텀(Momentum)

- ✓ 모멘텀은 '운동량'을 뜻하는 단어로, 일종의 '관성'을 주는 방법
- ✓ 속도 $\boldsymbol{v}$ 라는 개념을 추가하여 기울기 방향으로 힘을 받아 물체가 가속
- ✓ 기울기(gradient)를 통해 이동하는 방향과는 별개로, 과거에 이동했던 방식을 기억하면서 그 방향으로 일정 정도를 추가적으로 이동하는 방식

속도 모멘텀($\alpha = 0.9$)

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

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{v}$$



6.1.3 모멘텀(Momentum)

✓ 모멘텀 소스코드

```
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

초기 $v = 0$

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

$$W \leftarrow W + v$$

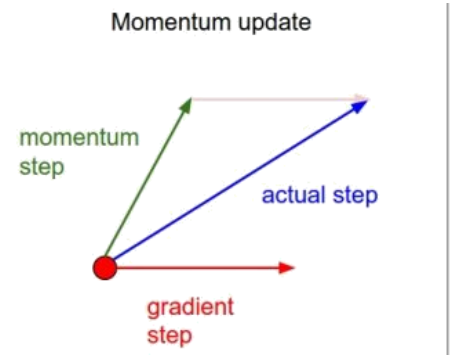
6.1 매개변수 갱신 - cont'd

6.1.4 Nesterov (Accelerated Gradient)

- ✓ 모멘텀(Momentum)을 향상 시킨 방법
- ✓ Nesterov는 $\boldsymbol{v}$를 계산할 때 momentum step($\boldsymbol{\alpha v}$)을 먼저 고려하여, 먼저 이동했다고 생각한 후 그 자리에서 기울기(gradient, $\frac{\partial L}{\partial W}$)를 구해서gradient step을 이동함

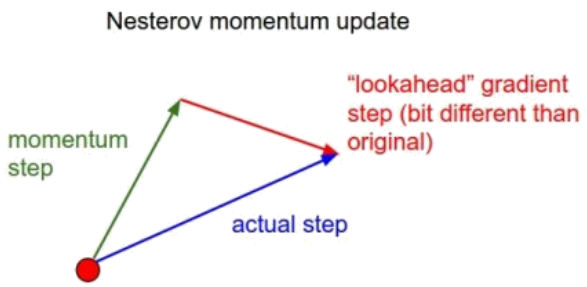
$$\boldsymbol{v \leftarrow \alpha v - \eta \frac{\partial L}{\partial W}}$$

$$\boldsymbol{W \leftarrow W + \alpha^2 v - (1+\alpha) \cdot \eta \frac{\partial L}{\partial W}}$$



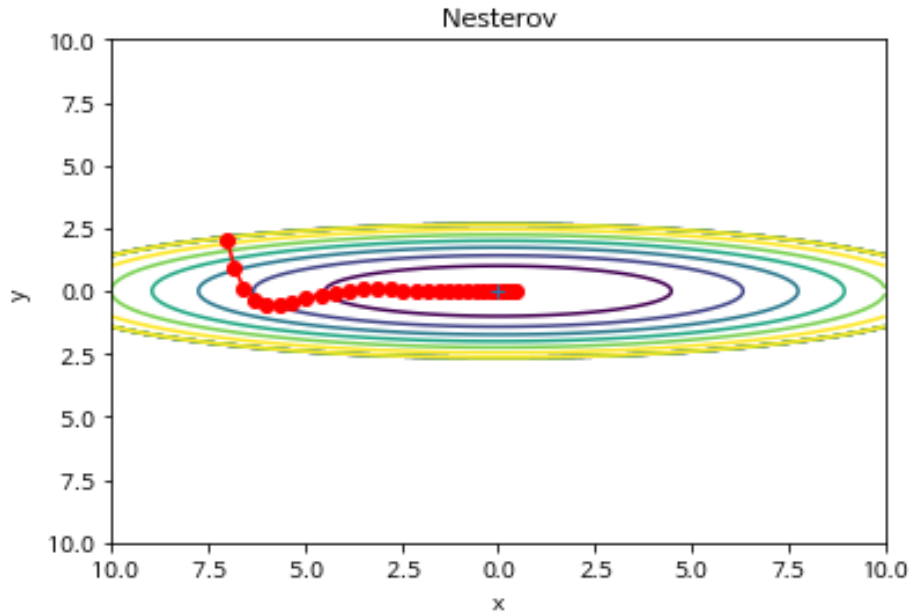
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$



$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$



6.1.4 Nesterov (Accelerated Gradient)

✓ Nesterov 소스코드

```
class Nesterov:

    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] *= self.momentum
            self.v[key] -= self.lr * grads[key]
            params[key] += self.momentum * self.momentum * self.v[key]
            params[key] -= (1 + self.momentum) * self.lr * grads[key]
```

초기 $v = 0$

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

$$W \leftarrow W + \alpha^2 v - (1 + \alpha) \cdot \eta \frac{\partial L}{\partial W}$$

6.1 매개변수 갱신 - cont'd

6.1.5 AdaGrad(Adaptive Gradient)

- ✓ 신경망 학습에서는 learning rate(η) 값이 중요 → *너무 작으면 오래 걸리고, 너무 크면 발산*
- ✓ AdaGrad는 처음에는 크게 학습하다가 조금씩 작게 학습하는 방식으로 learning rate를 줄여나감

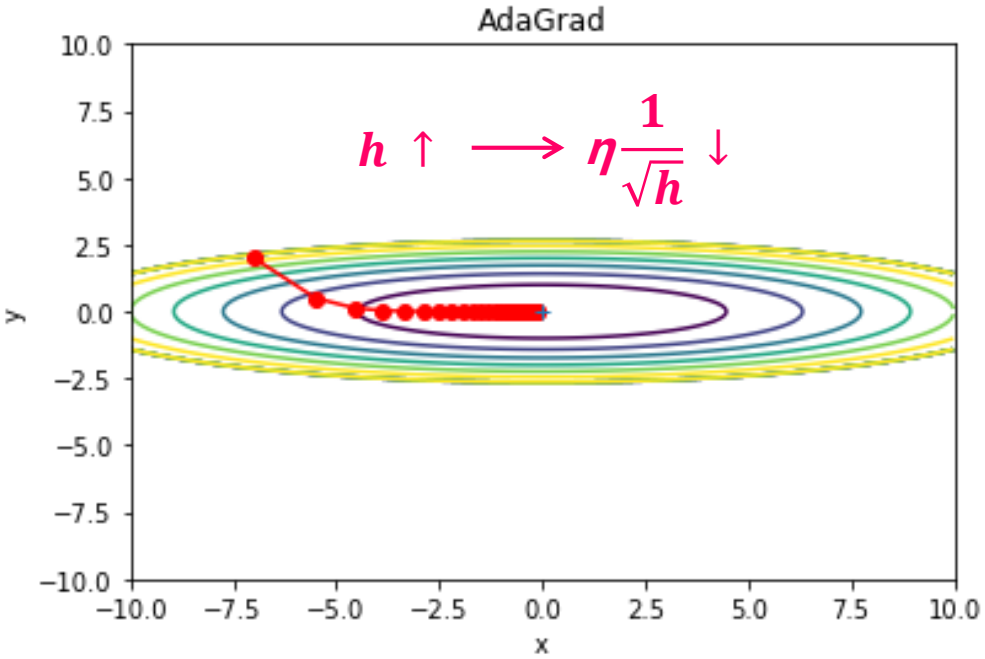
  → *learning rate decay*

```python
class AdaGrad:

    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr*grads[key]/(np.sqrt(self.h[key])+1e-7)
```

초기 $h = 0$

$$h \leftarrow h + \frac{\partial L}{\partial W} * \frac{\partial L}{\partial W}$$

$$W \leftarrow W - \eta * \frac{1}{\sqrt{h + \varepsilon}} * \frac{\partial L}{\partial W}$$



6.1 매개변수 갱신 - cont'd

6.1.6 RMSProp(Root Mean Square Propagation)

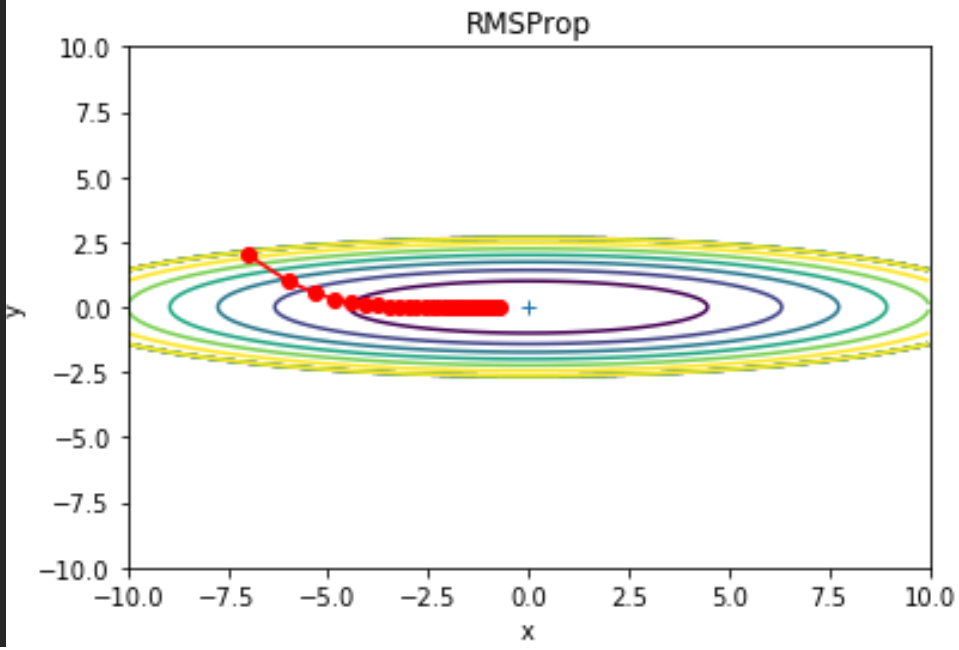
- ✓ 딥러닝의 대가 제프리 힌튼(Geoffrey Hinton)이 제안한 방법
- ✓ AdaGrad의 단점인 학습속도가 급진적으로 단조 감소하는 방법을 보완함 → *오래 될 수록 거의 학습이 되지 않음*
- ✓ AdaGrad에서 단순히 기울기(gradient, $\frac{\partial L}{\partial W}$)를 제곱하는 것이 아닌 이동평균(moving average)를 사용

```
class RMSprop:
    def __init__(self, lr=0.01, decay_rate = 0.95):
        self.lr = lr
        self.decay_rate = decay_rate
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] *= self.decay_rate
            self.h[key] += (1 - self.decay_rate) * grads[key] * grads[key]
            params[key] -= self.lr*grads[key]/(np.sqrt(self.h[key])+1e-7)
```

초기 $h = 0$

$$h \leftarrow \gamma h + (1-\gamma)\frac{\partial L}{\partial W} * \frac{\partial L}{\partial W}$$

$$W \leftarrow W - \eta * \frac{1}{\sqrt{h+\varepsilon}} * \frac{\partial L}{\partial W}$$



6.1 매개변수 갱신 - cont'd

6.1.7 AdaDelta(Adaptive Delta) - 참고

- ✓ AdaDelta는 RMSProp과 유사하게 AdaGrad의 단점을 보완하기 위해 제안된 방법
- ✓ RMSProp과 동일하게 $\boldsymbol{h}$ 를 구할 때 이동평균(MA)를 이용해 계산

```
class AdaDelta:

    def __init__(self, lr=1.0, decay_rate=0.95):
        self.lr = lr
        self.decay_rate = decay_rate
        self.h = None
        self.s = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        elif self.s is None:
            self.s = {}
            for key, val in params.items():
                self.s[key] = np.zeros_like(val)

        for key in params.keys():
            dx = np.sqrt((self.s[key]+1e-7) / (self.h[key]+1e-7)) * grads[key]
            self.h[key] = self.decay_rate*self.h[key] + (1 - self.decay_rate) * grads[key] * grads[key]
            self.s[key] = self.decay_rate*self.s[key] + (1 - self.decay_rate) * dx * dx
            params[key] -= dx
```

현재 대부분의 딥러닝 프레임워크에서는 Learning rate를 조절할 수 있게 추가되어 있음

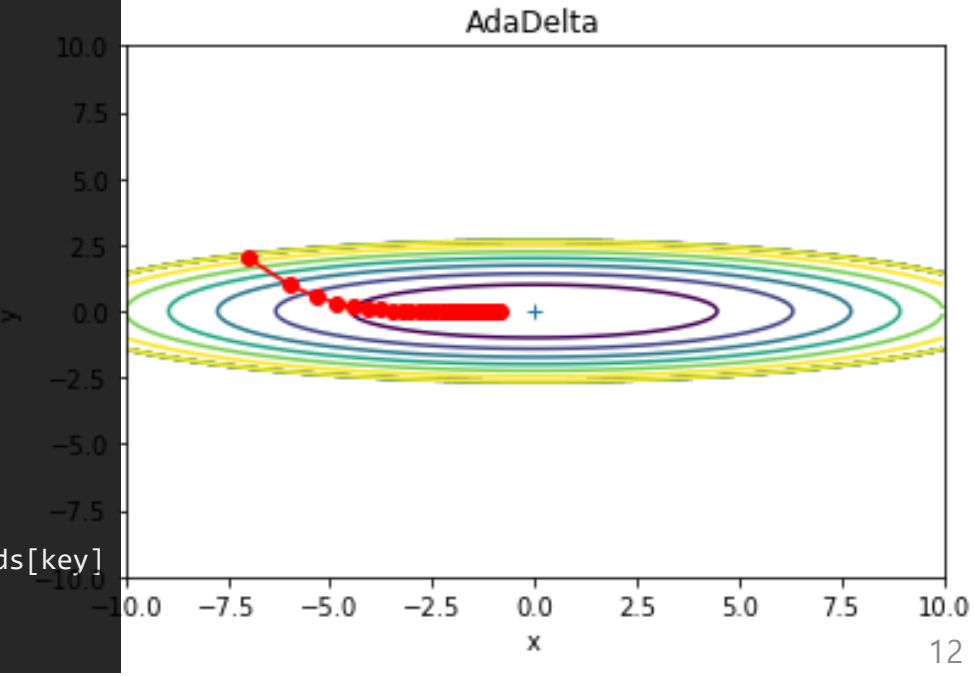
$$\boldsymbol{h} \leftarrow \boldsymbol{\gamma h} + (\boldsymbol{1}-\boldsymbol{\gamma})\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial W}} * \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial W}}$$

$$\Delta \boldsymbol{w} \leftarrow \frac{\sqrt{\boldsymbol{s}+\boldsymbol{\varepsilon}}}{\sqrt{\boldsymbol{h}+\boldsymbol{\varepsilon}}} * \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial W}}$$

Learning rate 역할

$$\boldsymbol{s} \leftarrow \boldsymbol{\gamma s} + (\boldsymbol{1}-\boldsymbol{\gamma})\Delta \boldsymbol{w}^{\boldsymbol{2}}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \Delta \boldsymbol{w}$$



6.1 매개변수 갱신 - cont'd

6.1.8 Adam(Adaptive Moment Estimation) - 참고

- ✓ Adam은 RMSProp과 Momentum을 결합한 방법
- ✓ Adam에서는 $\boldsymbol{m}$과 $\boldsymbol{v}$가 초기값은 0이기 때문에 학습 초반부에서는 0에 가깝게 편향(bias) 되어있을 것이라고 판단하여 unbiased하게 작업을 해줌

※ $\boldsymbol{\beta_1}$ 과 $\boldsymbol{\beta_2}$는 모멘텀용 계수로써,

※ $\boldsymbol{\beta_1}$은 0.9 $\boldsymbol{\beta_2}$는 0.999로 설정

$$\boldsymbol{m \leftarrow m + (1-\beta_1)\left(\frac{\partial L}{\partial W} - m\right)}$$

$$\boldsymbol{= \beta_1 m + (1-\beta_1)\frac{\partial L}{\partial W}} \longrightarrow \textit{Momentum 과 유사}$$

$$\boldsymbol{v \leftarrow v + (1-\beta_2)\left(\frac{\partial L}{\partial W} * \frac{\partial L}{\partial W} - v\right)}$$

$$\boldsymbol{= \beta_2 v + (1-\beta_2)\frac{\partial L}{\partial W} * \frac{\partial L}{\partial W}} \longrightarrow \textit{RMSProp 과 유사}$$

$$\boldsymbol{W \leftarrow W - \eta * \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} * \frac{m}{\sqrt{v+\varepsilon}}}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

6.1 매개변수 갱신 - cont'd

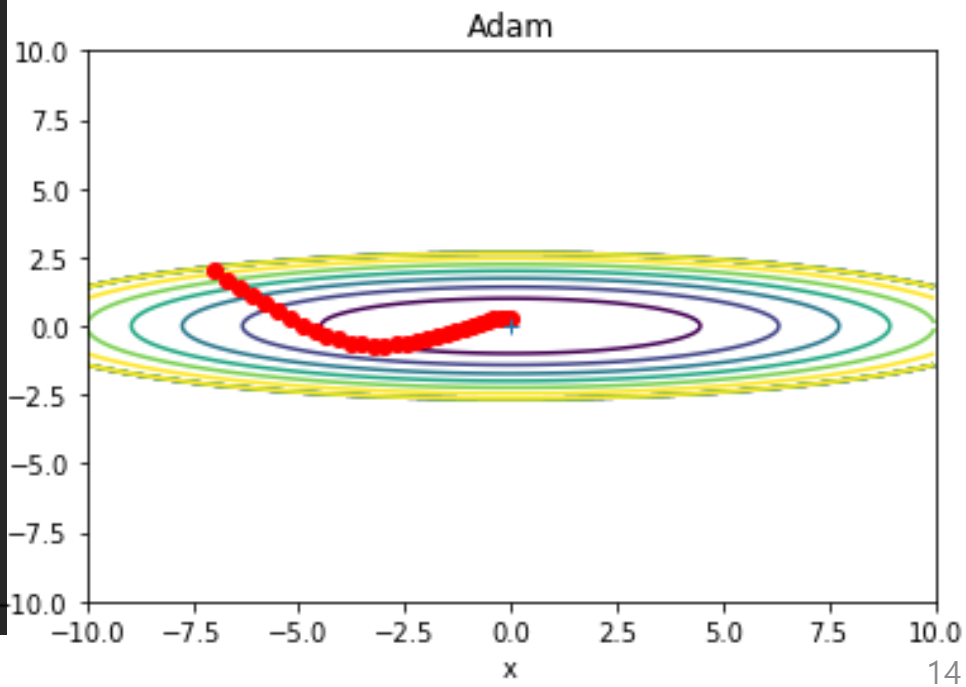
6.1.8 Adam(Adaptive Moment Estimation) - 참고

```python
class Adam:

    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None

    def update(self, params, grads):
        if self.m is None:
            self.m, self.v = {}, {}
            for key, val in params.items():
                self.m[key] = np.zeros_like(val)
                self.v[key] = np.zeros_like(val)

        self.iter += 1
        lr_t  = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

        for key in params.keys():
            self.m[key] += (1 - self.beta1) * (grads[key] - self.m[key])
            self.v[key] += (1 - self.beta2) * (grads[key]**2 - self.v[key])

            params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)
```

$$m \leftarrow m + (1-\beta_1)\left(\frac{\partial L}{\partial W} - m\right)$$

$$v \leftarrow v + (1-\beta_2)\left(\frac{\partial L}{\partial W} * \frac{\partial L}{\partial W} - v\right)$$

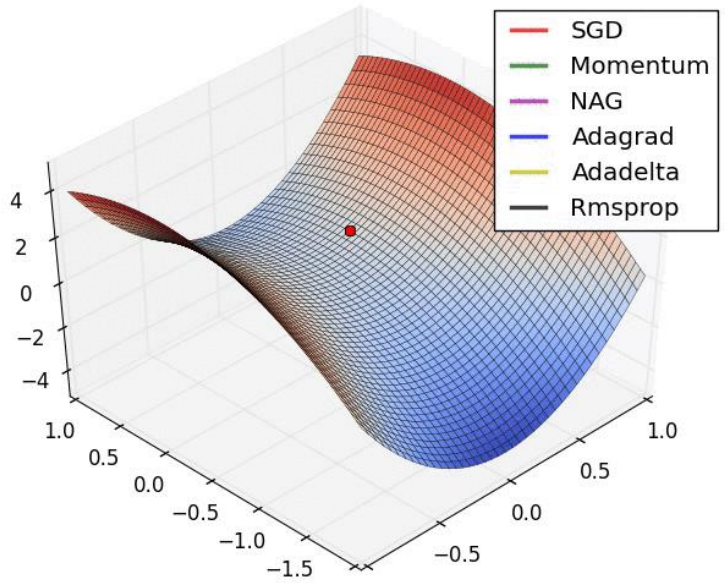
$$W \leftarrow W - \eta * \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} * \frac{m}{\sqrt{v+\varepsilon}}$$

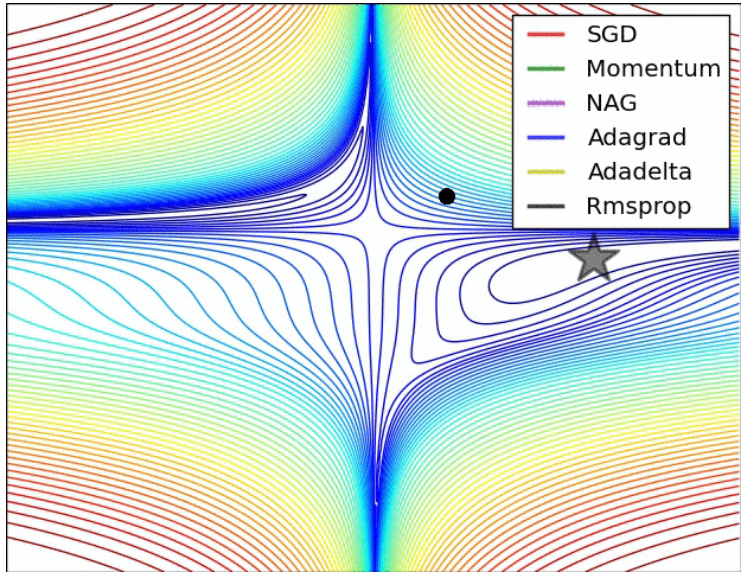


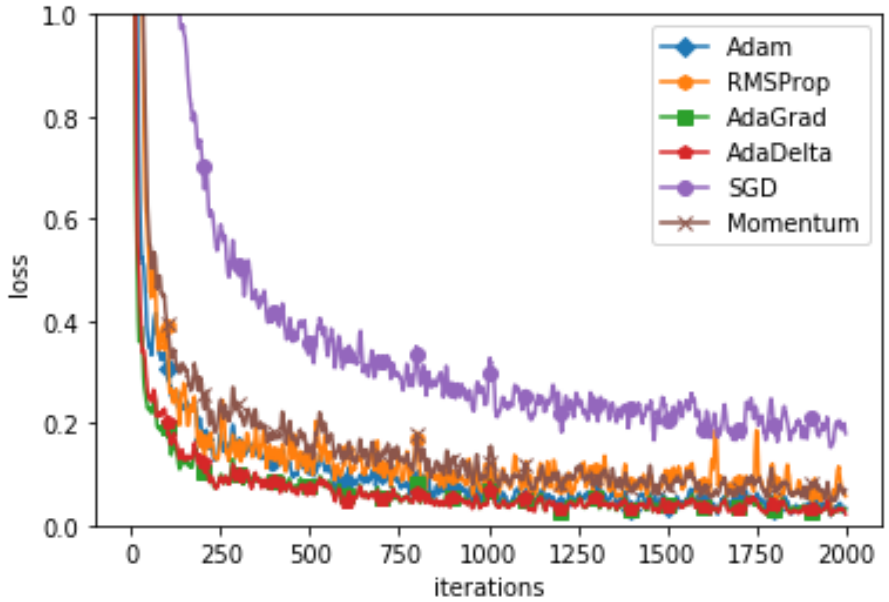
6.1 매개변수 갱신 - cont'd

6.1.9 Optimizer 비교 (feat. MNIST) - 참고

- ✓ 어떤 Optimizer가 좋다고 딱 골라 말할 수는 없음
- ✓ 상황에 따라 적절한 Optimizer를 선택 해야함
- ✓ But, 요즘 추세는 Adam을 많이 사용하는 듯함







6.2 가중치의 초기값

- ✓ 신경망 학습에서 특히 중요한 것이 가중치의 초기값을 성정하는 작업임
- ✓ 초기값을 무엇으로 설정하느냐에 따라 신경망 학습의 성패를 가르는 일도 있음

6.2.1 초기값을 0으로 하면?

- ✓ 초기값을 0 또는 동일한 값으로 설정하게 되면 오차역전파시 모든 가중치의 값이 똑같이 갱신이 됨

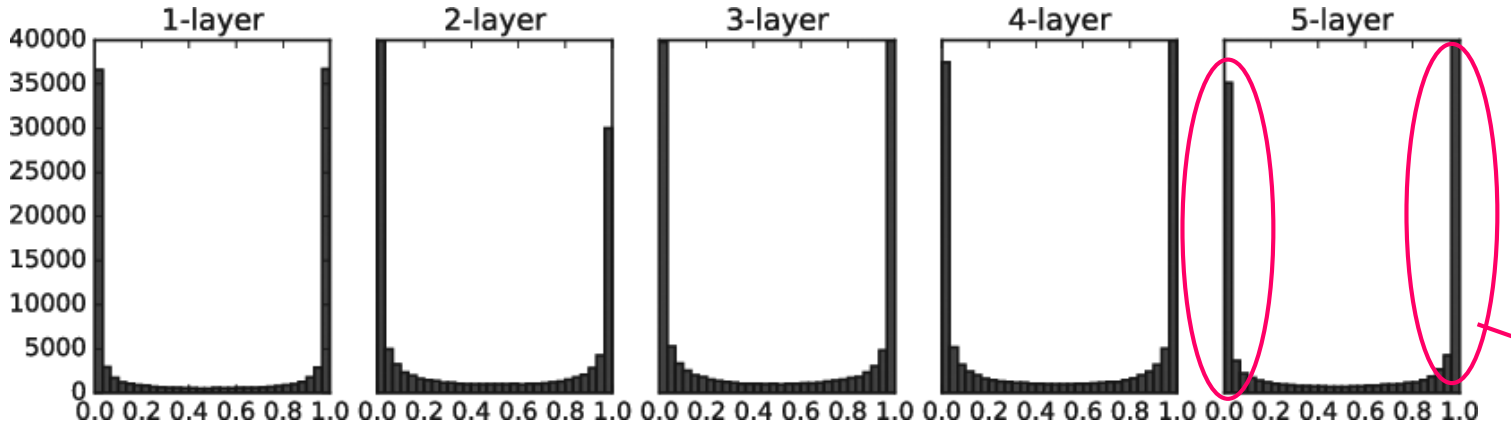
  → *갱신을 해도 여전히 같기 때문에 학습의 의미가 없음*
- ✓ 따라서, 이러한 문제를 해결하기 위해서는 초기값을 동일하지 않도록 **랜덤하게** 설정해야 함
- ✓ 특히, 가중치의 값이 클 경우 Overfitting이 일어날 확률이 높기 때문에 가중치를 가능한 작게 만들어 줘야함

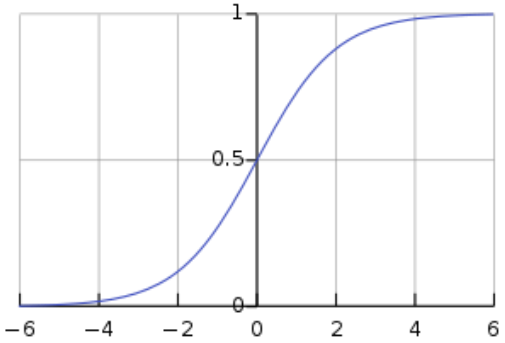
  → *weight decay*

6.2 가중치의 초기값

6.2.2 은닉층의 활성화 함수의 출력값 분포

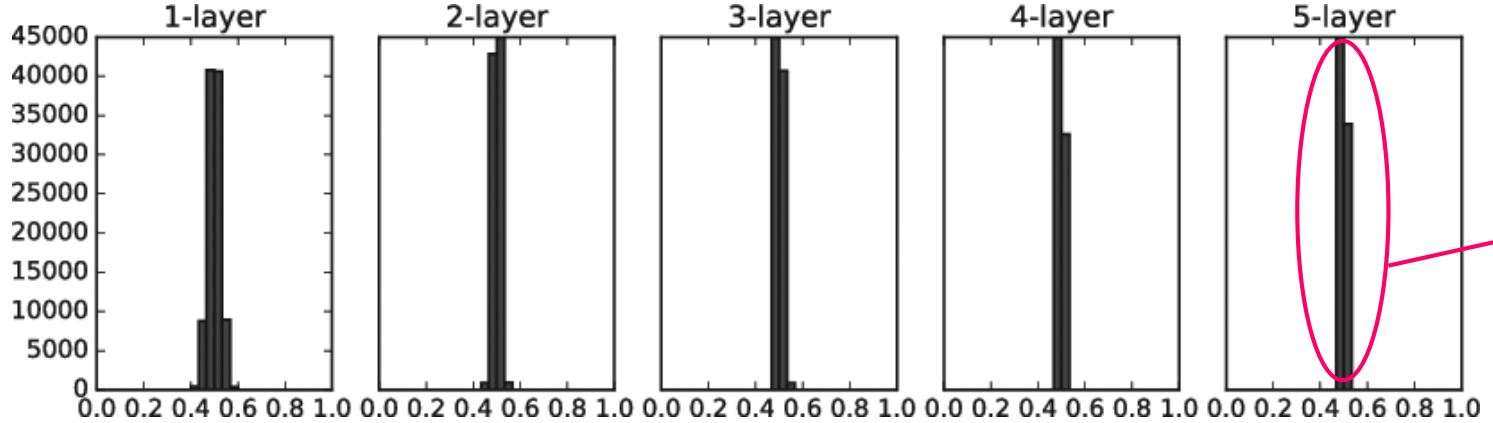
✓ 가중치를 표준편차가 1인 정규분포로 초기화할 때의 각 층의 활성화 값 분포





기울기 소실
(gradient vanishing)

✓ 가중치를 표준편차가 0.01인 정규분포로 초기화할 때의 각 층의 활성화 값 분포

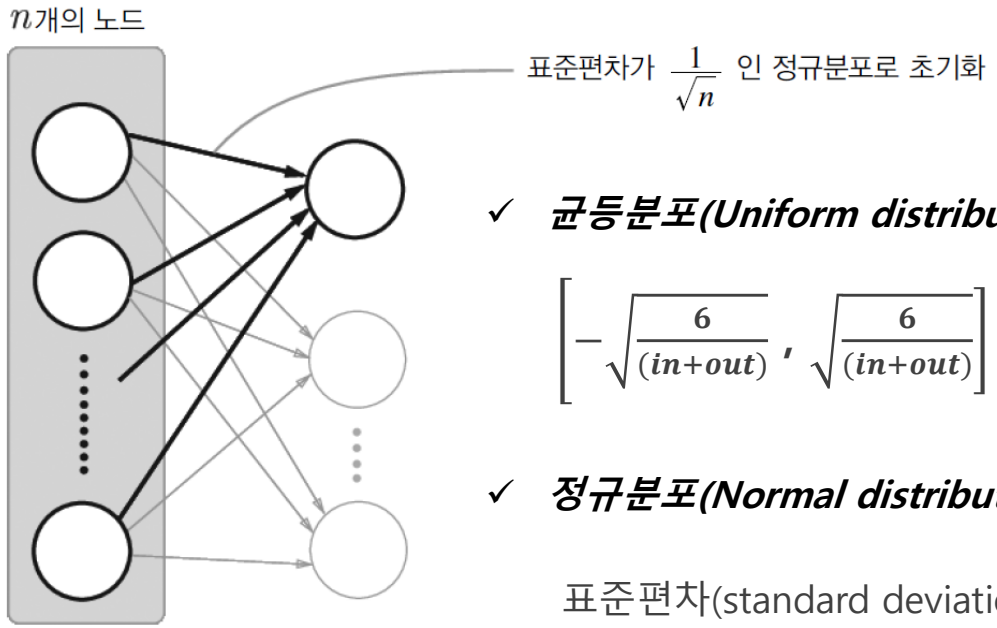


가중치가 거의 같은
값이므로 의미 없음

6.2 가중치의 초기값

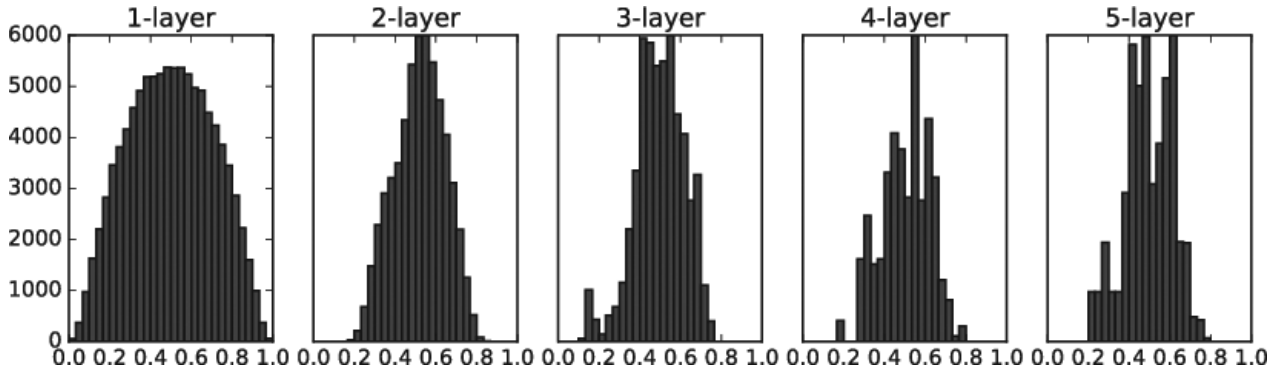
6.2.3 Xavier 초기값 - 참고

- ✓ Sigmoid 계열의 함수에 적합한 가중치 초기값 설정 방법 → *sigmoid, tanh 등*
- ✓ RBM(Restricted Boltzmann Machine)과 같은 복잡한 방법이 아닌 간단한 방법으로 가중치 초기값 설정
- ✓ **Xavier 초기값**: 초기값의 표준편차가 $\frac{1}{\sqrt{n}}$이 되도록 설정



- ✓ ***균등분포(Uniform distribution)***

$$\left[-\sqrt{\frac{6}{(in+out)}}, \sqrt{\frac{6}{(in+out)}}\right]$$



- ✓ ***정규분포(Normal distribution)***

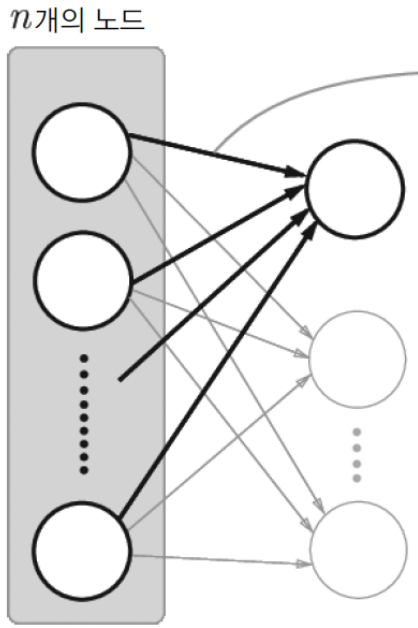
표준편차(standard deviation) $= \sqrt{\frac{2}{(in+out)}}$

$$N\left(0, \sqrt{\frac{2}{(in+out)}}^2\right)$$

6.2 가중치의 초기값

6.2.4 He 초기값 - 참고

- ✓ ReLU 에 특화된 가중치 초기값 설정 방법
- ✓ 카이밍 히(Kaiming He)의 이름을 따 **He 초기값**이라고 함
- ✓ **He 초기값**: 초기값의 표준편차가 $\frac{2}{\sqrt{n}}$이 되도록 설정



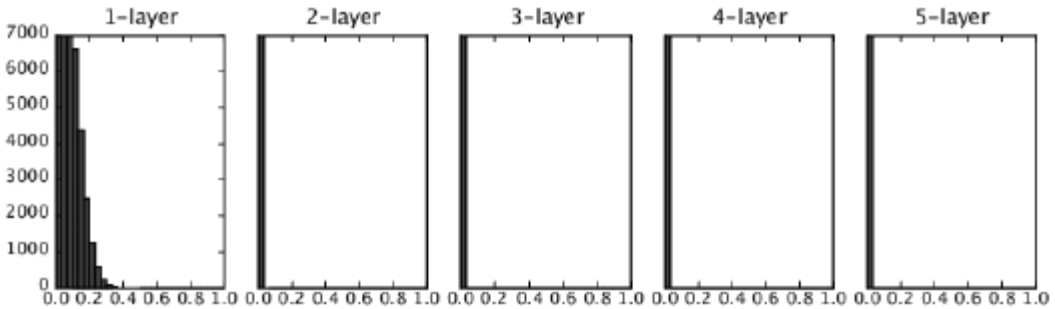
- ✓ ***균등분포(Uniform distribution)***

$$\left[-\sqrt{\frac{6}{in}}, \sqrt{\frac{6}{in}}\right]$$

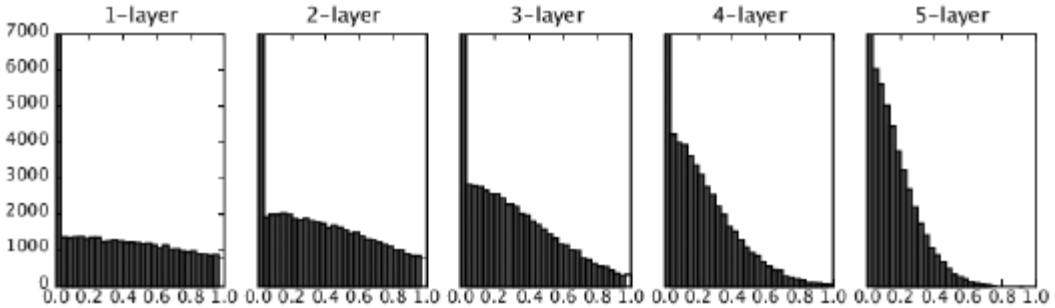
- ✓ ***정규분포(Normal distribution)***

표준편차(standard deviation) $= \sqrt{\frac{2}{in}}$

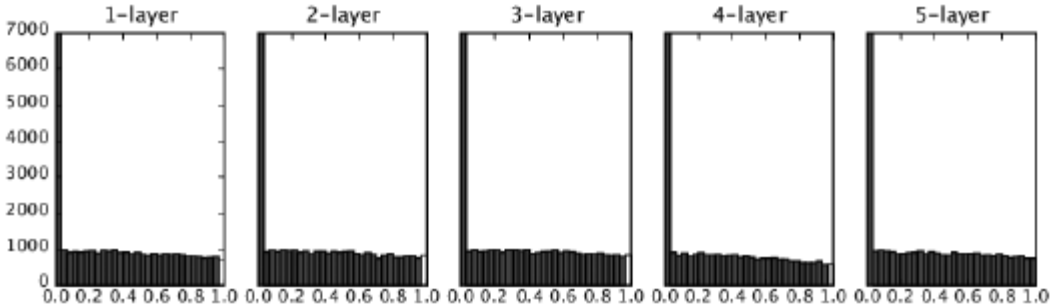
$$N(0, \sqrt{\frac{2}{in}}^{2})$$



표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우



Xavier 초깃값을 사용한 경우



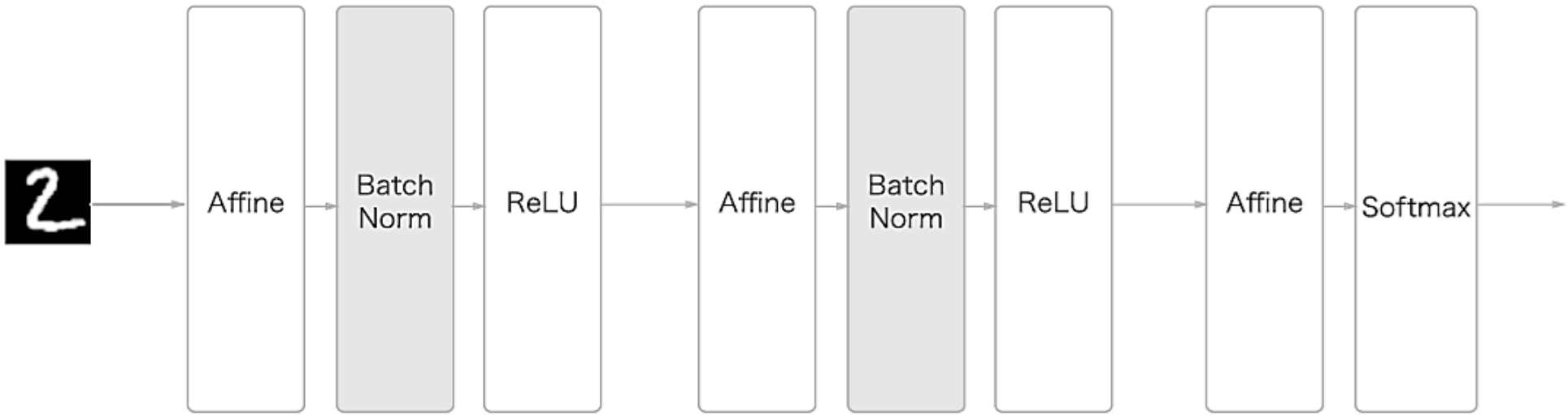
He 초깃값을 사용한 경우

6.3 배치 정규화

- ✓ 앞에서는 각 층의 가중치의 초기값을 적절히 설정하면 활성화 함수의 출력값이 적당이 퍼지면서 학습이 원활하게 수행 된다는 것을 알 수 있음
- ✓ 그렇다면, 가중치 초기값 설정에 의존하지 않고 각층이 활성화 함수의 출력값을 적당히 퍼뜨리도록 **강제** 하는 방법이 없을까? → ***배치 정규화(Batch Normalization)***

6.3.1 배치 정규화 알고리즘

- ✓ 학습을 빨리 진행할 수 있음
- ✓ 초기값에 크게 의존하지 않음
- ✓ 과적합(Overfitting)을 억제함(드롭아웃 등의 필요성 감소)
- ✓ 각 층에서의 활성화값(활성화 함수의 출력값)이 적당히 분포되도록 조정해줌



6.3 배치 정규화

6.3.1 배치 정규화 알고리즘 - 참고

- ✓ 학습 시 **미니배치**(Mini-batch)를 단위로 정규화를 해줌 → *mini-batch 단위로* $N(0,1)$ *되도록*
- ✓ 단순히 $N(0,1)$ 로 만들어 주게 되면 활성화 함수의 비선형성이(Non-linearity) 없어질 수 있음
- ✓ 따라서, 배치 정규화 시 정규화된 데이터에 고유한 **확대**(scale, γ)와 **이동**(shift, β)변환을 수행
  → $\gamma = 1, \beta = 0$ 에서 시작(원본 그대로에서 시작한다는 의미)

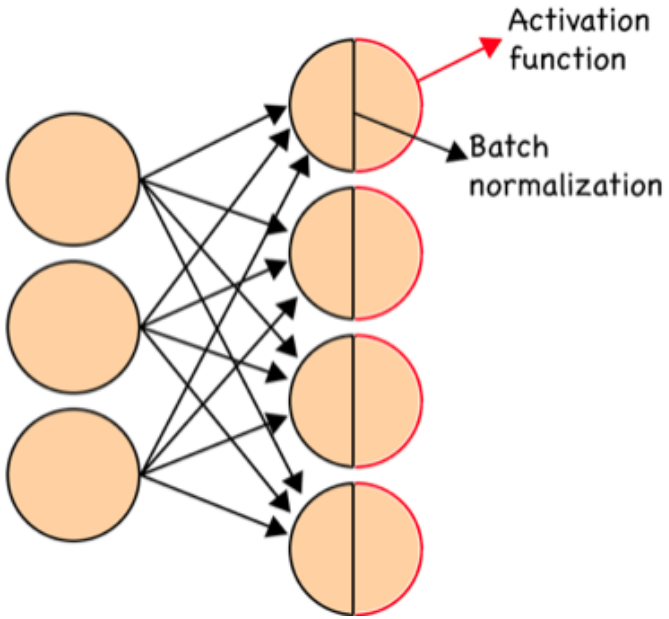
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

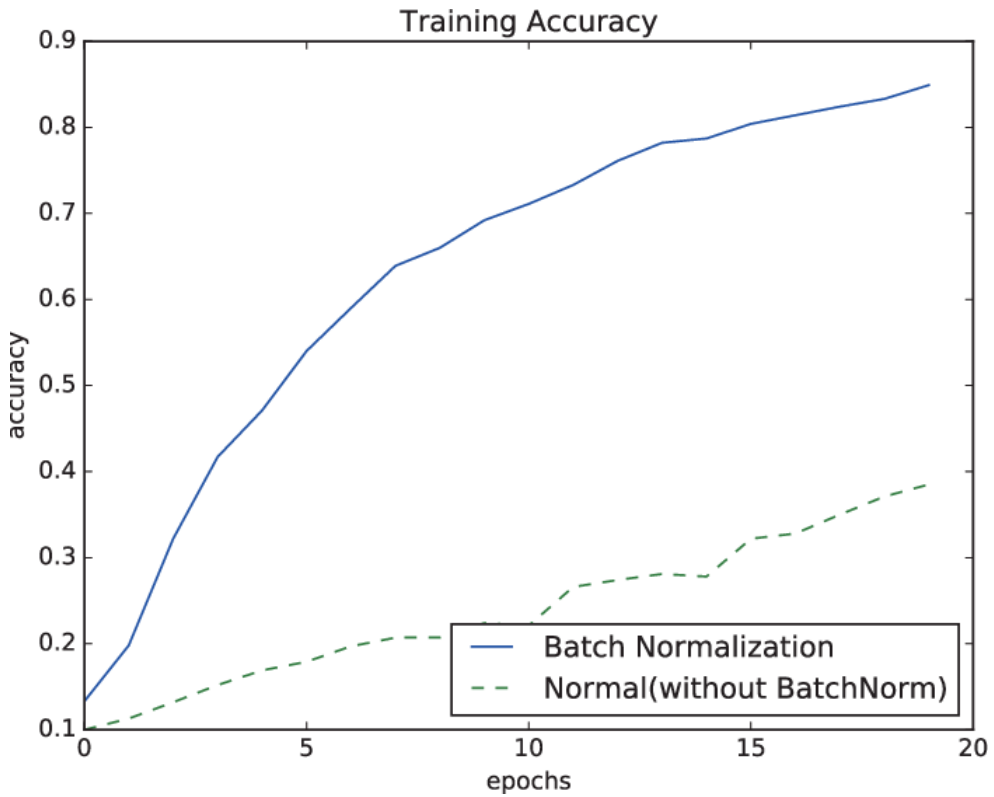


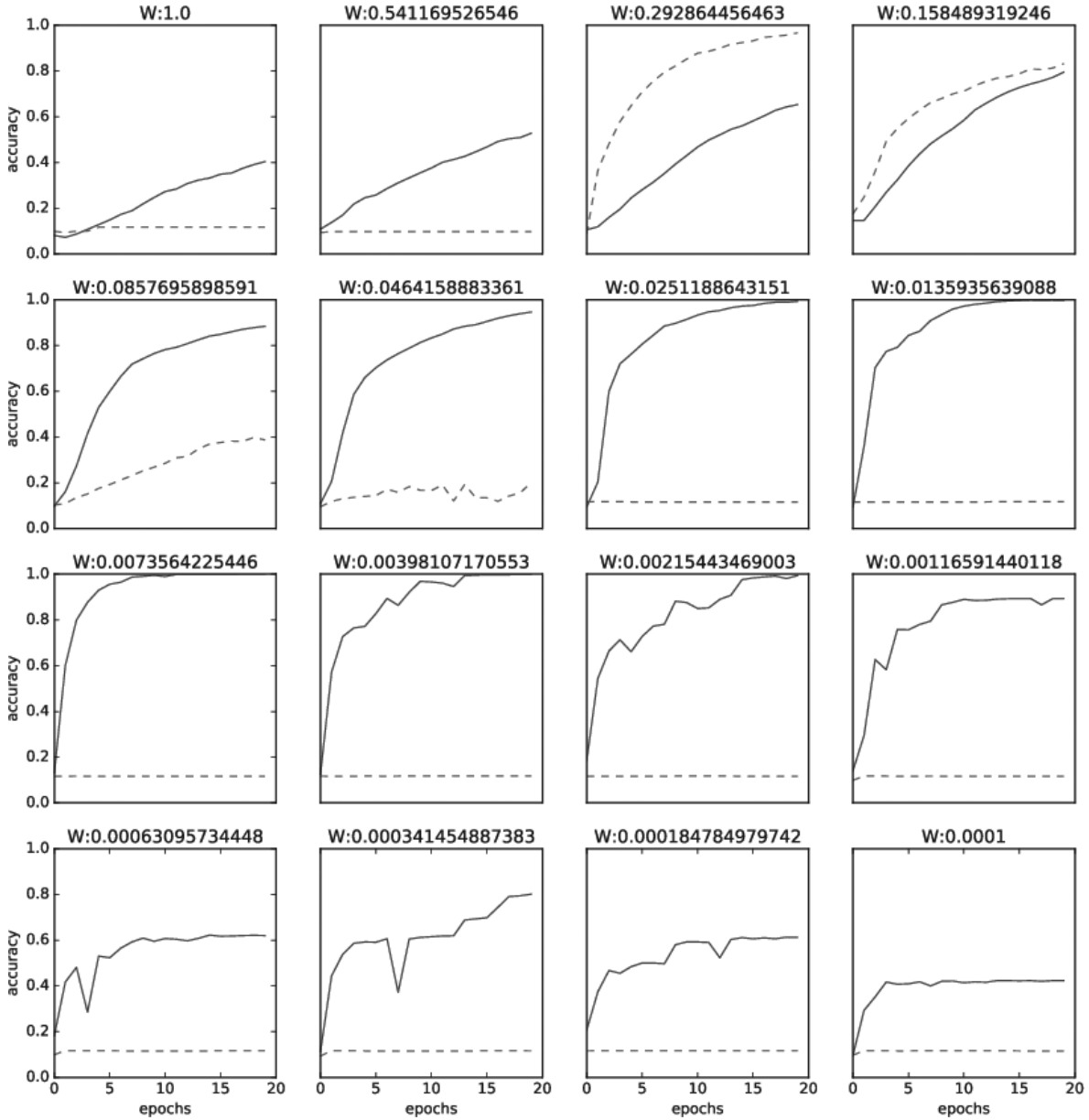
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

6.3 배치 정규화

6.3.1 배치 정규화 알고리즘 - 참고

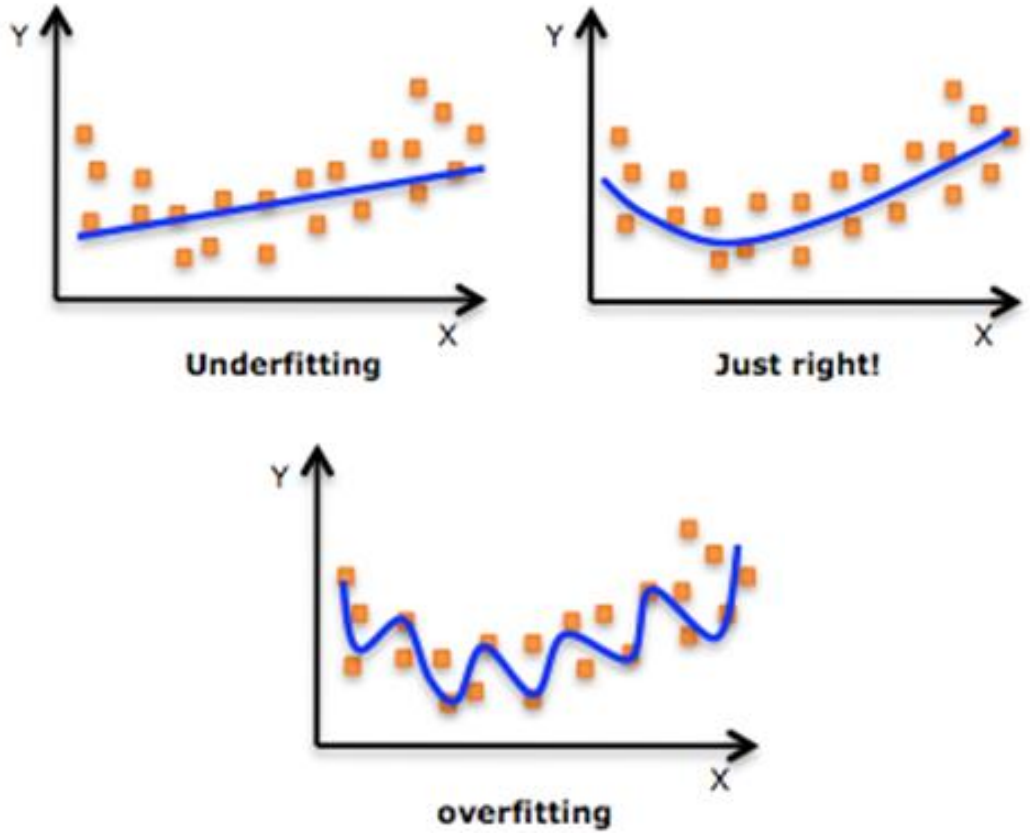


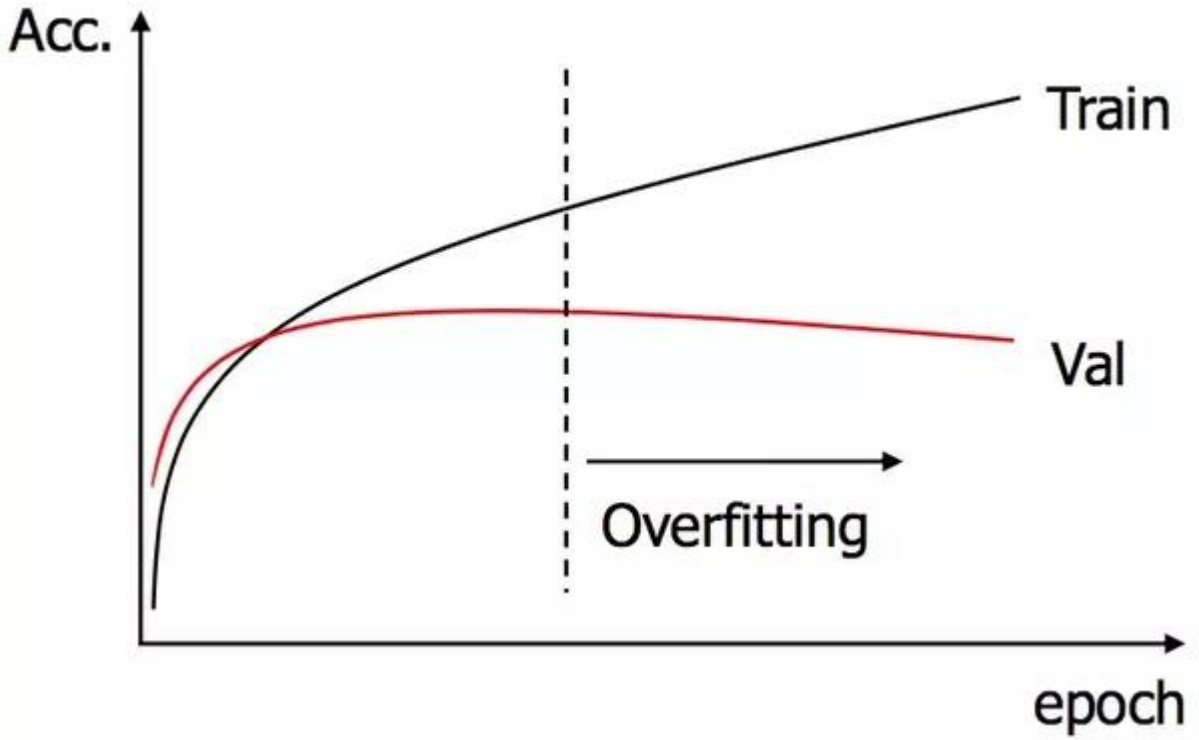


6.4 바른 학습을 위해

6.4.1 오버피팅(Overfitting) 이란?

- ✓ 훈련 데이터(training set)에만 적합(fit)되어 그 외의 데이터(test set 등)에는 제대로 대응하지 못하는 상태
- ✓ 매개 변수가 많고 표현력이 높은 모델 → *Complexity가 높은 모델*
- ✓ 훈련 데이터가 적을 경우





6.4 바른 학습을 위해

6.4.2 가중치 Regularization

- ✓ 학습 과정에서 큰 가중치에 대해서는 큰 페널티를 부과하여 오버피팅을 억제하는 방법
- ✓ L1 보다 **L2** 가 더 많이 쓰임

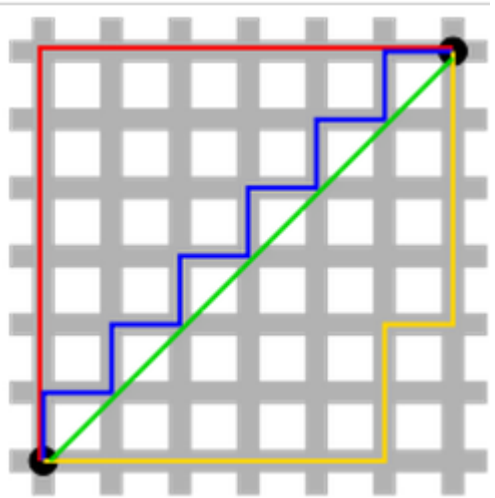
> **NOTE_** L2 법칙은 각 원소의 제곱들을 더한 것에 해당합니다. 가중치 $\mathbf{W} = (w_1, w_2, \cdots, w_n)$ 이 있다면, L2 법칙에서는 $\sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$ 으로 계산할 수 있습니다. L2 법칙 외에 L1 법칙과 L∞ 법칙도 있습니다. L1 법칙은 절댓값의 합, 즉 $|w_1| + |w_2| + \cdots + |w_n|$ 에 해당합니다. L∞ 법칙은 Max 법칙이라고도 하며, 각 원소의 절댓값 중 가장 큰 것에 해당합니다. 정규화 항으로 L2 법칙, L1 법칙, L∞ 법칙 중 어떤 것도 사용할 수 있습니다. 각자 특징이 있는데, 이 책에서는 일반적으로 자주 쓰는 L2 법칙만 구현합니다.

L1 regularization on least squares:

$$\boldsymbol{L} = \arg\min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \lambda \sum_{i=1}^{k} |w_i|$$

L2 regularization on least squares:

$$\boldsymbol{L} = \arg\min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \lambda \sum_{i=1}^{k} w_i^2$$

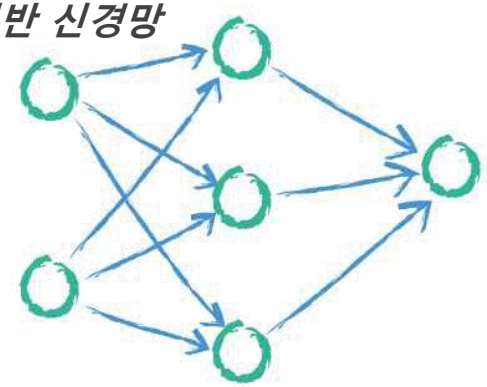


6.4 바른 학습을 위해

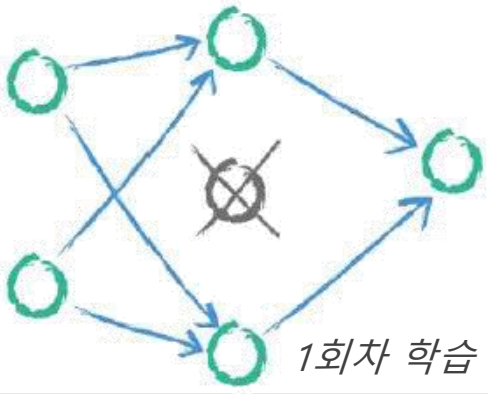
6.4.3 드롭아웃 (Dropout)

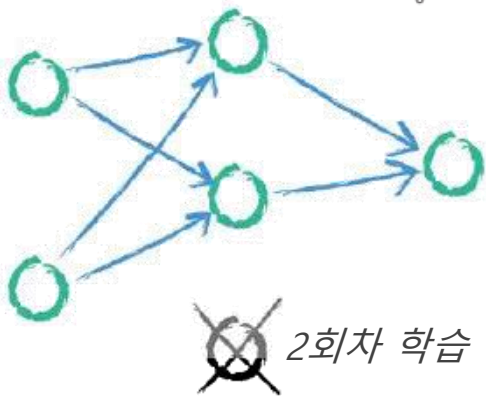
- ✓ 학습 시 전체 신경망 중 일부만 사용하도록 하는 방법
- ✓ 즉, 학습 단계마다 **일부 뉴런을 제거**(사용하지 않도록)함으로써, **일부 특징이 특정 뉴런들에 고정되는 것을 막아** 가중치의 균형을 잡도록 해줌 → *overfitting 방지*
- ✓ ***주의:* 학습 시**에는 드롭아웃을 적용하고, **테스트(검증) 시**에는 신경망 전체를 사용하도록 해줘야 함

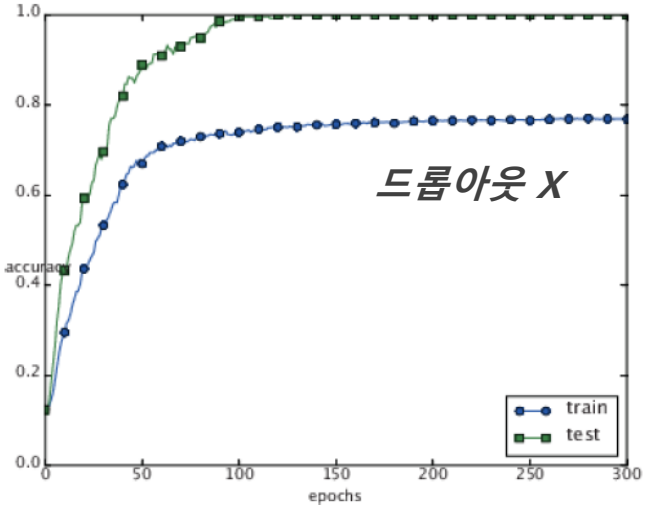
✓ *일반 신경망*

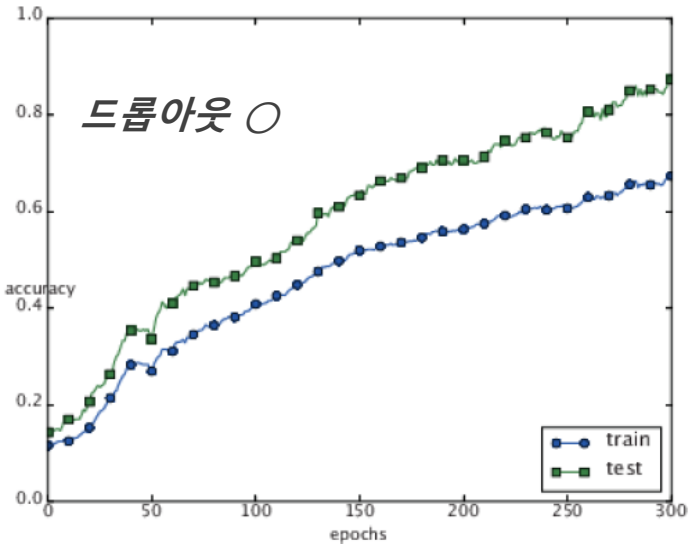


✓ *드롭아웃 적용*









6.5 적절한 하이퍼파라미터 값 찾기

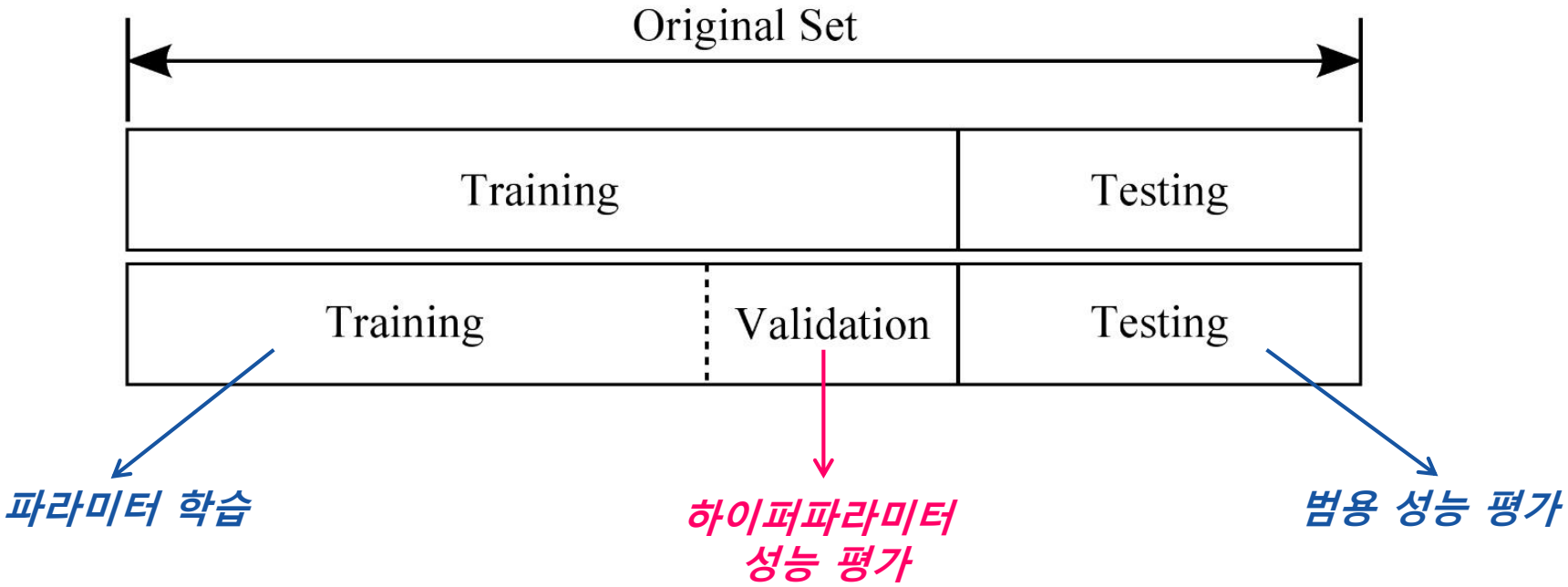
✓ 딥러닝에는 하이퍼파라미터(Hyper-parameter)가 많이 등장함 → *뉴런 수, 배치 크기, 학습률 등*

✓ 하이퍼파라미터의 값은 매우 중요하지만 그 값을 결정하기까지는 많은 시행착오를 거침

6.5.1 검증 데이터

✓ 하이퍼파라미터를 조정할 때는 하이퍼파라미터 전용 확인 데이터가 필요

✓ 이러한 데이터를 **검증 데이터(Validation data)**라고 함



6.5 적절한 하이퍼파라미터 값 찾기

6.5.2 하이퍼파라미터 최적화

- ✓ 하이퍼파라미터를 최적화할 때의 핵심은 '최적 값'이 존재하는 범위를 조금씩 줄여나가야 함
- ✓ 우선, 대략적인 범위를 설정하고 그 범위에서 무작위로 하이퍼파라미터 값을 샘플링 한 후 정확도를 평가
- ✓ '대략적'으로 0.001 ~ 1,000 사이 $(10^{-3}\sim10^{3})$ **10의 계승** 단위로 범위를 지정 → *로그 스케일(log scale)*

- **0단계**
  하이퍼파라미터 값의 범위를 설정한다.
- **1단계**
  설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출한다.
- **2단계**
  1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 검증 데이터로 정확도를 평가한다(단, 에폭은 작게 설정한다).
- **3단계**
  1단계와 2단계를 특정 횟수(100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힌다.

6.5.2 하이퍼파라미터 최적화

이번 장에서 배운 것

- 매개변수 갱신 방법에는 확률적 경사 하강법(SGD) 외에도 모멘텀, AdaGrad, Adam 등이 있다.
- 가중치 초깃값을 정하는 방법은 올바른 학습을 하는 데 매우 중요하다.
- 가중치의 초깃값으로는 'Xavier 초깃값'과 'He 초깃값'이 효과적이다.
- 배치 정규화를 이용하면 학습을 빠르게 진행할 수 있으며, 초깃값에 영향을 덜 받게 된다.
- 오버피팅을 억제하는 정규화 기술로는 가중치 감소와 드롭아웃이 있다.
- 하이퍼파라미터 값 탐색은 최적 값이 존재할 법한 범위를 점차 좁히면서 하는 것이 효과적이다.

THANK YOU