CH4. 딥러닝 시작

4.1 인공 신경망의 한계와 딥러닝 출현

오늘날 인공 신경망에서 이용하는 구조(입력층, 출력층, 가중치로 구성된 구조)는 프랭크 로젠블라트(Frank Rosenblatt)가 1957년에 고안한 퍼셉트론이라는 선형 분류기입니다. 이 퍼셉트론은 오늘날 신경망(딥러닝)의 기원이 되는 알고리즘입니다.

퍼셉트론은 다수의 신호(흐름이 있는)를 입력으로 받아 하나의 신호를 출력하는데, 이 신호를 입력으로 받아 '흐른다/안 흐른다(1 또는 0)'는 정보를 앞으로 전달하는 원리로 작동합니다.

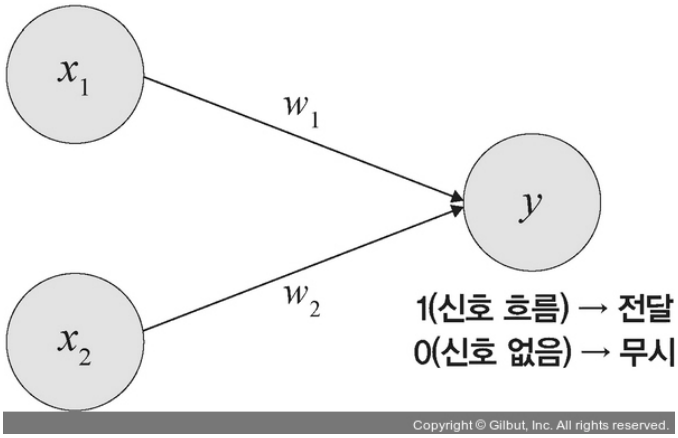


그림 4-1과 같이 입력이 두 개(x1, x2) 있다고 할 때 컴퓨터가 논리적으로 인식하는 방식을 알아보기 위해 논리 게이트로 확인해 봅시다.

AND 게이트

AND 게이트는 모든 입력이 ‘1’일 때 작동합니다. 즉, 입력 중 어떤 하나라도 ‘0’을 갖는다면 작동을 멈춤.

OR 게이트

OR 게이트는 입력에서 둘 중 하나만 ‘1’이거나 둘 다 ‘1’일 때 작동합니다. 즉, 입력 모두가 ‘0’을 갖는 경우를 제외한 나머지가 모두 ‘1’ 값을 가짐.

XOR 게이트

XOR 게이트는 배타적 논리합이라는 용어로 입력 두 개 중 한 개만 ‘1’일 때 작동하는 논리 연산.

XOR 게이트는 데이터가 비선형적으로 분리되기 때문에 제대로 된 분류가 어렵습니다. 즉, 단층 퍼셉트론에서는 AND, OR 연산에 대해서는 학습이 가능하지만, XOR에 대해서는 학습이 불가능합니다.

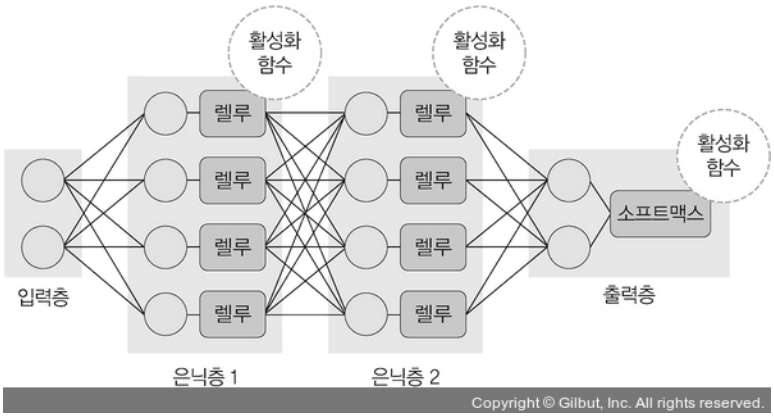
이를 극복하는 방안으로 입력층과 출력층 사이에 하나 이상의 중간층(은닉층)을 두어 비선형적으로 분리되는 데이터에 대해서도 학습이 가능하도록 다층 퍼셉트론(multi-layer perceptron)을 고안했습니다.

이때 입력층과 출력층 사이에 은닉층이 여러 개 있는 신경망을 심층 신경망(Deep Neural Network, DNN)이라고 하며, 심층 신경망을 다른 이름으로 딥러닝이라고 합니다.

4.2 딥러닝 구조

4.2.1 딥러닝 용어

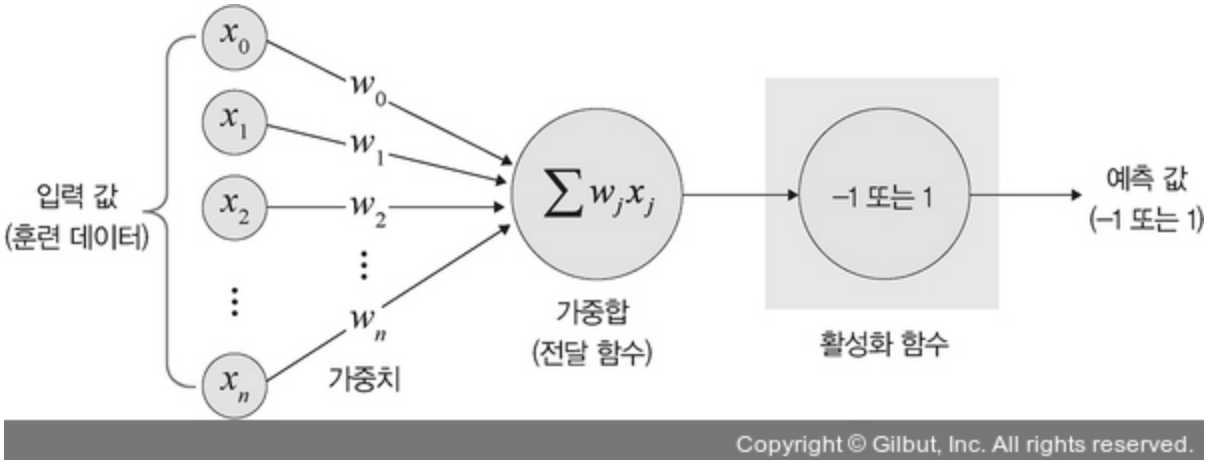
딥러닝은 다음 그림과 같이 입력층, 출력층과 두 개 이상의 은닉층으로 구성되어 있습니다. 또한, 입력 신호를 전달하기 위해 다양한 함수도 사용하고 있는데, 신경망을 이루는 구성 요소에 대해 하나씩 살펴보겠습니다.



입력층, 은닉층, 출력층은 표의 정의를 참고하면 되고, 나머지 용어는 하나씩 좀 더 자세히 살펴보겠습니다.

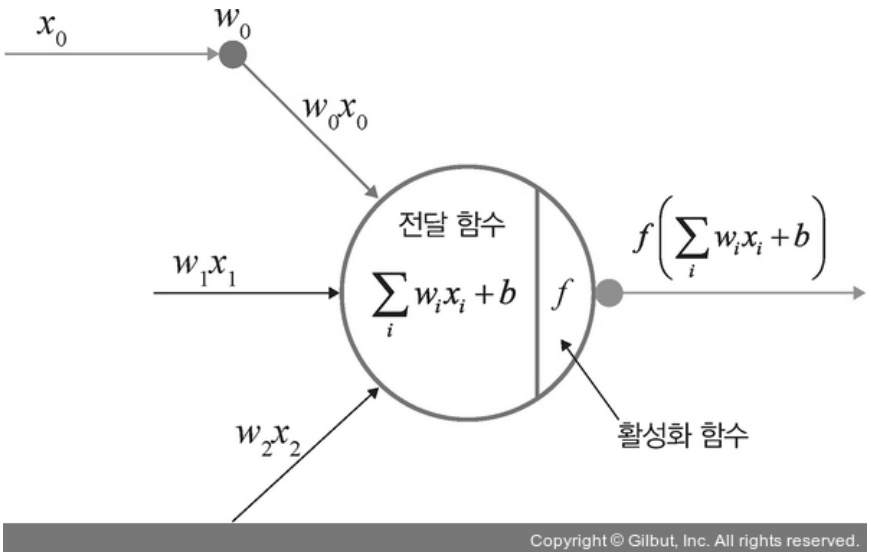
가중치

가중치는 입력 값이 연산 결과에 미치는 영향력을 조절하는 요소입니다. 예를 들어 다음 그림에서 w1 값이 0 혹은 0과 가까운 0.001이라면, x1이 아무리 큰 값이라도 x1×w1 값은 0이거나 0에 가까운 값이 됩니다. 이와 같이 입력 값의 연산 결과를 조정하는 역할을 하는 것이 가중치입니다.



가중합 또는 전달 함수

가중합은 전달 함수라고도 합니다. 각 노드에서 들어오는 신호에 가중치를 곱해서 다음 노드로 전달되는데, 이 값들을 모두 더한 합계를 가중합이라고 합니다. 또한, 노드의 가중합이 계산되면 이 가중합을 활성화 함수로 보내기 때문에 전달 함수(transfer function)라고도 합니다.



가중합을 구하는 공식은 다음과 같습니다.

$$\sum_i w_i\, x_i + b$$

(w: 가중치, b: 바이어스)

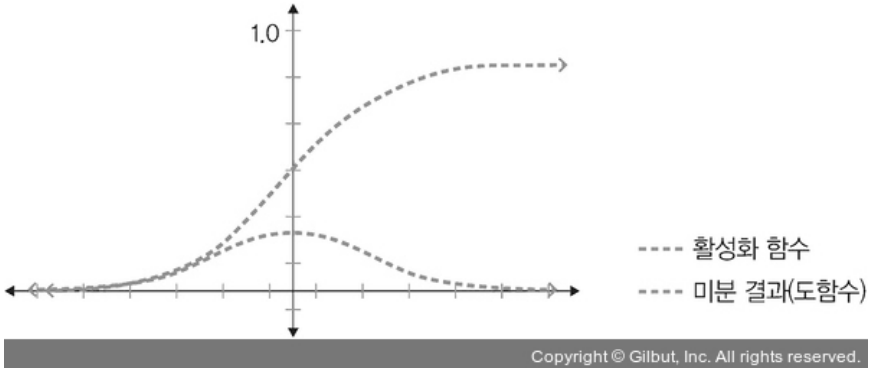
활성화 함수

다음으로 함수들에 대해 알아보겠습니다. 먼저 활성화 함수는 전달 함수에서 전달받은 값을 출력할 때 일정 기준에 따라 출력 값을 변화시키는 비선형 함수입니다. 활성화 함수로는 시그모이드(sigmoid), 하이퍼볼릭 탄젠트(hyperbolic tangent), 렐루(ReLU) 함수 등이 있습니다.

시그모이드 함수

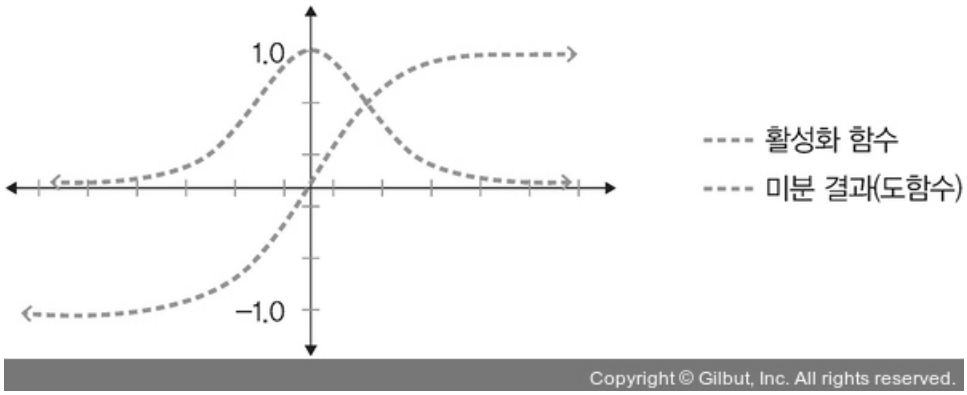
시그모이드 함수는 선형 함수의 결과를 0~1 사이에서 비선형 형태로 변형해 줍니다. 주로 로지스틱 회귀와 같은 분류 문제를 확률적으로 표현하는 데 사용됩니다. 과거에는 인기가 많았으나, 딥러닝 모델의 깊이가 깊어지면 기울기가 사라지는 '기울기 소멸 문제(vanishing gradient problem)'가 발생하여 딥러닝 모델에서는 잘 사용하지 않습니다.

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



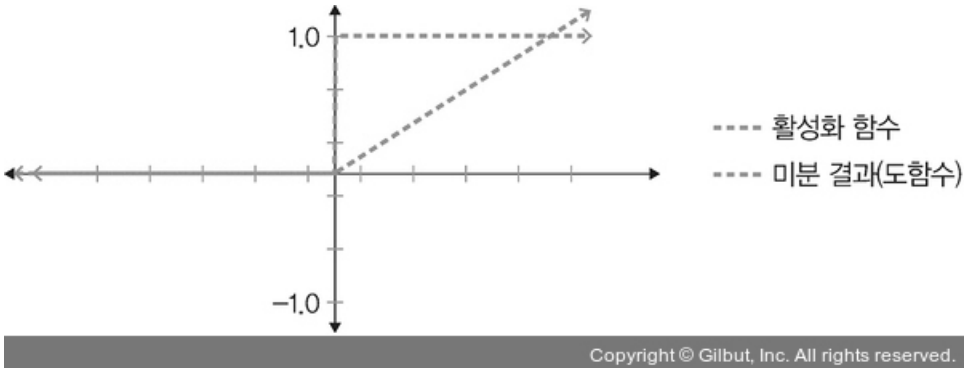
하이퍼볼릭 탄젠트 함수

하이퍼볼릭 탄젠트 함수는 선형 함수의 결과를 -1~1 사이에서 비선형 형태로 변형해 줍니다. 시그모이드에서 결괏값의 평균이 0이 아닌 양수로 편향된 문제를 해결하는 데 사용했지만, 기울기 소멸 문제는 여전히 발생합니다.



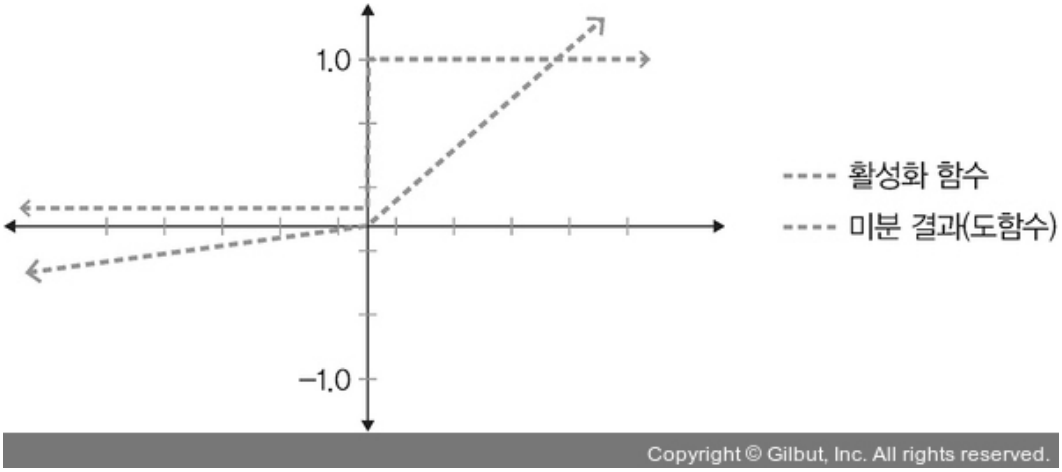
렐루 함수

최근 활발히 사용되는 렐루(ReLU) 함수는 입력(x)이 음수일 때는 0을 출력하고, 양수일 때는 x를 출력합니다. 경사 하강법(gradient descent)에 영향을 주지 않아 학습 속도가 빠르고, 기울기 소멸 문제가 발생하지 않는 장점이 있습니다. 렐루 함수는 일반적으로 은닉층에서 사용되며, 하이퍼볼릭 탄젠트 함수 대비 학습 속도가 6배 빠릅니다. 문제는 음수 값을 입력받으면 항상 0을 출력하기 때문에 학습 능력이 감소하는데, 이를 해결하려고 리키 렐루(Leaky ReLU) 함수 등을 사용합니다.



리키 렐루 함수

리키 렐루(Leaky ReLU) 함수는 입력 값이 음수이면 0이 아닌 0.001처럼 매우 작은 수를 반환합니다. 이렇게 하면 입력 값이 수렴하는 구간이 제거되어 렐루 함수를 사용할 때 생기는 문제를 해결할 수 있습니다.



소프트맥스 함수

소프트맥스(softmax) 함수는 입력 값을 0~1 사이에 출력되도록 정규화하여 출력 값들의 총합이 항상 1이 되도록 합니다. 소프트맥스 함수는 보통 딥러닝에서 출력 노드의 활성화 함수로 많이 사용됩니다. 수식으로 표현하면 다음과 같습니다.

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

exp(x)(앞의 식에서는 exp(ak)와 exp(ai)를 의미)는 지수 함수(exponential function)입니다. n은 출력층의 뉴런 개수, yk는 그중 k번째 출력을 의미합니다. 즉, 이 수식처럼 소프트맥스 함수의 분자는 입력 신호 ak의 지수 함수, 분모는 모든 입력 신호의 지수 함수 합으로 구성됩니다.

```
class Net(torch.nn.Module):
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden) ------ 은닉층
        self.relu = torch.nn.ReLu(inplace=True)
        self.out = torch.nn.Linear(n_hidden, n_output) ------ 출력층
        self.softmax = torch.nn.Softmax(dim=n_output)
    def forward(self, x):
        x = self.hidden(x)
        x = self.relu(x) ------ 은닉층을 위한 렐루 활성화 함수
        x = self.out(x)
        x = self.softmax(x) ------ 출력층을 위한 소프트맥스 활성화 함수
        return x
```

손실 함수

경사 하강법은 학습률(n, learning rate)과 손실 함수의 순간 기울기를 이용하여 가중치를 업데이트하는 방법입니다. 즉, 미분의 기울기를 이용하여 오차를 비교하고 최소화하는 방향으로 이동시키는 방법이라고 할 수 있습니다. 이때 오차를 구하는 방법이 손실 함수입니다.

즉, 손실 함수는 학습을 통해 얻은 데이터의 추정치가 실제 데이터와 얼마나 차이가 나는지 평가하는 지표라고 할 수 있습니다. 이 값이 클수록 많이 틀렸다는 의미이고, 이 값이 '0'에 가까우면 완벽하게 추정할 수 있다는 의미입니다. 대표적인 손실 함수로는 평균 제곱 오차(Mean Squared Error, MSE)와 크로스 엔트로피 오차(Cross Entropy Error, CEE)가 있습니다.

평균 제곱 오차

실제 값과 예측 값의 차이(error)를 제곱하여 평균을 낸 것이 평균 제곱 오차(MSE)입니다. 실제 값과 예측 값의 차이가 클수록 평균 제곱 오차의 값도 커진다는 것은 반대로 생각하면 이 값이 작을수록 예측력이 좋다는 것을 의미합니다. 평균 제곱 오차는 회귀에서 손실 함수로 주로 사용됩니다.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$\hat{y}_i$: 신경망의 출력(신경망이 추정한 값)
y_i : 정답 레이블
i : 데이터의 차원 개수


```
import torch

loss_fn = torch.nn.MSELoss(reduction='sum')
y_pred = model(x)
loss = loss_fn(y_pred, y)
```

크로스 엔트로피 오차

크로스 엔트로피 오차(CEE)는 분류(classification) 문제에서 원-핫 인코딩(one-hot encoding) 했을 때만 사용할 수 있는 오차 계산법입니다.

일반적으로 분류 문제에서는 데이터의 출력을 0과 1로 구분하기 위해 시그모이드 함수를 사용하는데, 시그모이드 함수에 포함된 자연 상수 e 때문에 평균 제곱 오차를 적용하면 매끄럽지 못한 그래프(울퉁불퉁한 그래프)가 출력됩니다. 따라서 크로스 엔트로피 손실 함수를 사용하는데, 이 손실 함수를 적용할 경우 경사 하강법 과정에서 학습이 지역 최소점에서 멈출 수 있습니다. 이것을 방지하고자 자연 상수 e에 반대되는 자연 로그를 모델의 출력 값에 취합니다.

$$CrossEntropy = -\sum_{i=1}^{n} y_i \log \hat{y}_i$$

$\hat{y}_i$: 신경망의 출력(신경망이 추정한 값)

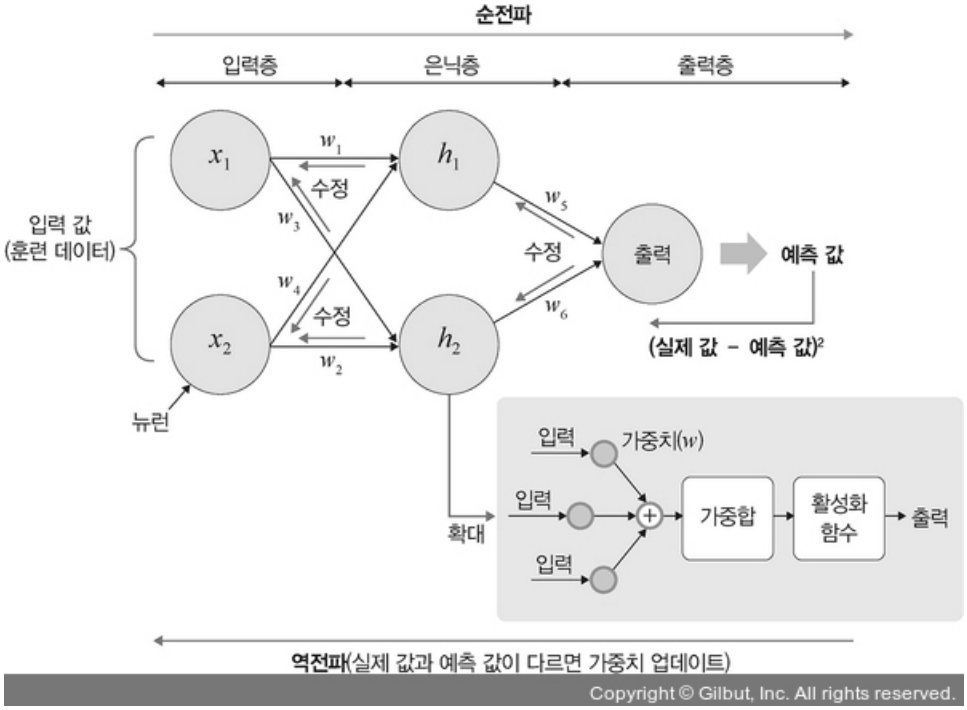
y_i : 정답 레이블

i : 데이터의 차원 개수


```
loss = nn.CrossEntropyLoss()
input = torch.randn(5, 6, requires_grad=True) ------ torch.randn은 평균이 0이고 표준편차가 1인
 가우시안 정규분포를 이용하여 숫자를 생성
target = torch.empty(3, dtype=torch.long).random_(5) ------ torch.empty는 dtype torch.floa
t32의 랜덤한 값으로 채워진 텐서를 반환
output = loss(input, target)
output.backward()
```

4.2.2 딥러닝 학습

딥러닝 학습은 크게 순전파와 역전파라는 두 단계로 진행됩니다.



첫 번째 단계인 **순전파**(feedforward)는 네트워크에 훈련 데이터가 들어올 때 발생하며, 데이터를 기반으로 예측 값을 계산하기 위해 전체 신경망을 교차해 지나갑니다. 즉, 모든 뉴런이 이전 층의 뉴런에서 수신한 정보에 변환(가중합 및 활성화 함수)을 적용하여 다음 층(은닉층)의 뉴런으로 전송하는 방식입니다. 네트워크를 통해 입력 데이터를 전달하며, 데이터가 모든 층을 통과하고 모든 뉴런이 계산을 완료하면 그 예측 값은 최종 층(출력층)에 도달하게 됩니다.

그다음 손실 함수로 네트워크의 예측 값과 실제 값의 차이(손실, 오차)를 추정합니다. 이때 손실 함수 비용은 '0'이 이상적입니다. 따라서 손실 함수 비용이 0에 가깝도록 하기 위해 모델이 훈련을 반복하면서 가중치를 조정합니다. 손실(오차)이 계산되면 그 정보는 역으로 전파(출력층 → 은닉층 → 입력층)되기 때문에 **역전파**(backpropagation)라고 합니다. 출력층에서 시작된 손실 비용은 은닉층의 모든 뉴런으로 전파되지만, 은닉층의 뉴런은 각 뉴런이 원래 출력에 기여한 상대적 기여도에 따라(즉, 가중치에 따라) 값이 달라집니다. 좀 더 수학적으로 표현하면 예측 값과 실제 값 차이를 각 뉴런의 가중치로 미분한 후 기존 가중치 값에서 뺍니다. 이 과정을 출력층 → 은닉층 → 입력층 순서로 모든 뉴런에 대해 진행하여 계산된 각 뉴런 결과를 또다시 순전파의 가중치 값으로 사용합니다.

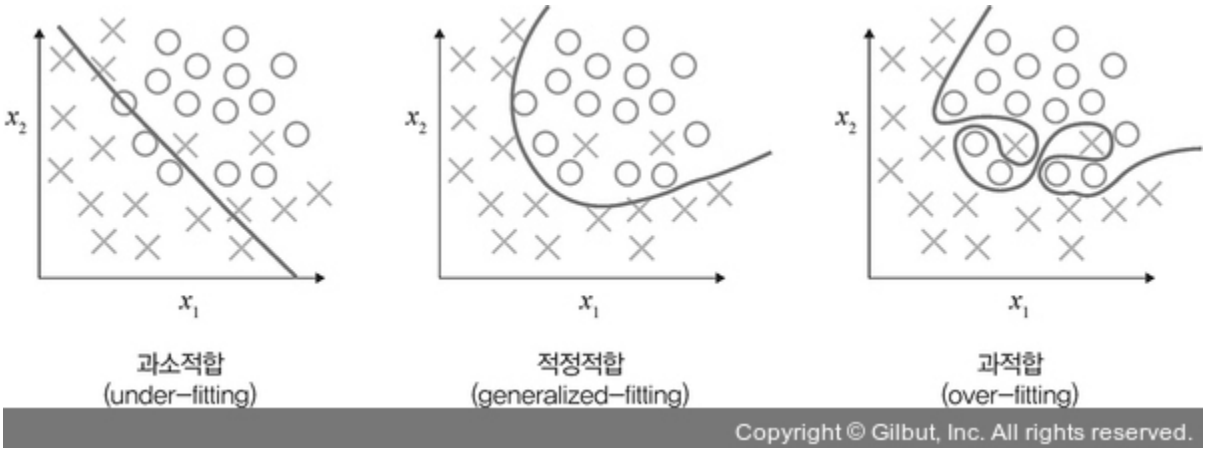
4.2.3 딥러닝의 문제점과 해결 방안

딥러닝의 핵심은 활성화 함수가 적용된 여러 은닉층을 결합하여 **비선형 영역을 표현**하는 것입니다.

하지만 은닉층이 많을수록 다음 세 가지 문제점이 생깁니다.

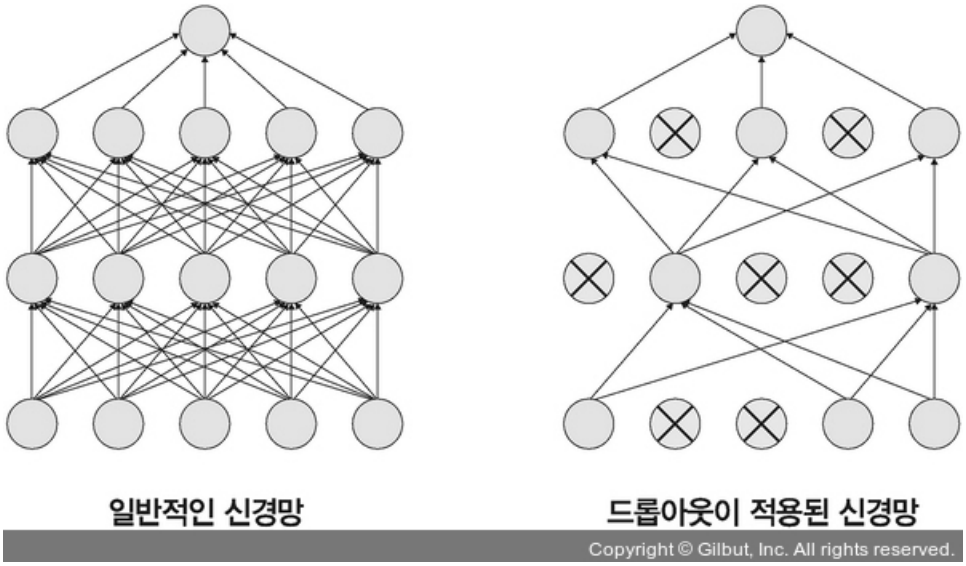
- 과적합 문제 발생 → 해결) 드롭아웃,

과적합(over-fitting)은 훈련 데이터를 과하게 학습해서 발생합니다. 일반적으로 훈련 데이터는 실제 데이터의 일부분입니다. 따라서 훈련 데이터를 과하게 학습했기 때문에 예측 값과 실제 값 차이인 오차가 감소하지만, 검증 데이터에 대해서는 오차가 증가할 수 있습니다. 이러한 관점에서 과적합은 훈련 데이터에 대해 과하게 학습하여 실제 데이터에 대한 오차가 증가하는 현상을 의미합니다.



과적합을 해결하는 방법으로 **드롭아웃(dropout)**이 있습니다.

신경망 모델이 과적합되는 것을 피하기 위한 방법으로, 학습 과정 중 임의로 일부 노드들을 학습에서 제외시킵니다.



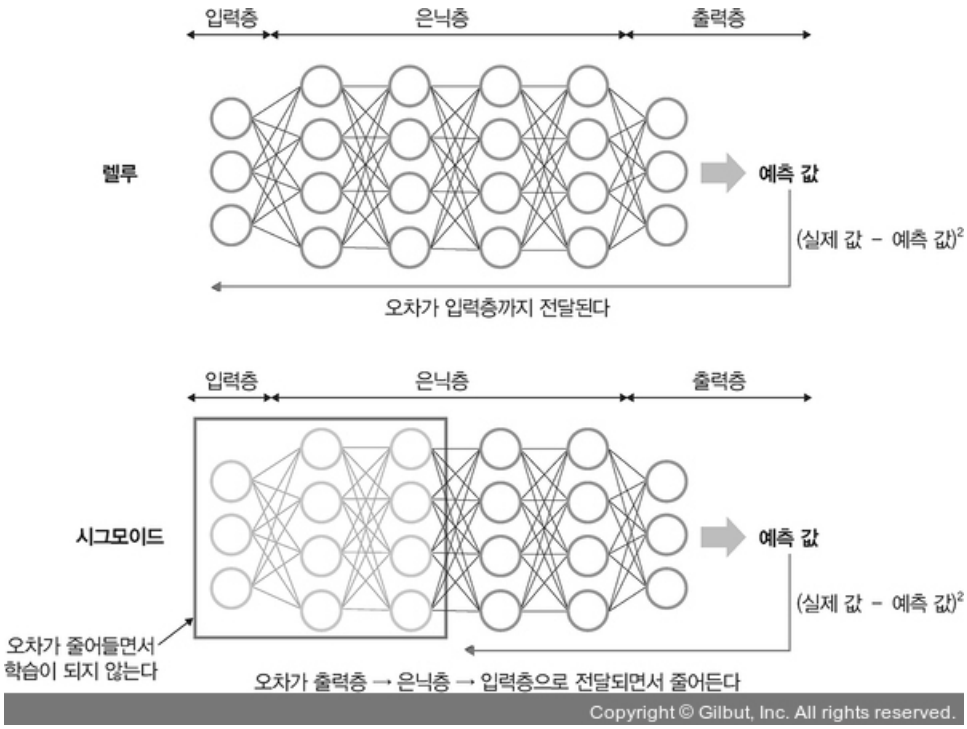

```
class DropoutModel(torch.nn.Module):
    def __init__(self):
        super(DropoutModel, self).__init__()
        self.layer1 = torch.nn.Linear(784, 1200)
        self.dropout1 = torch.nn.Dropout(0.5) ------ 50%의 노드를 무작위로 선택하여 사용하지 않겠다
는 의미
        self.layer2 = torch.nn.Linear(1200, 1200)
        self.dropout2 = torch.nn.Dropout(0.5)
        self.layer3 = torch.nn.Linear(1200, 10)

    def forward(self, x):
```

```
        x = F.relu(self.layer1(x))
        x = self.dropout1(x)
        x = F.relu(self.layer2(x))
        x = self.dropout2(x)
        return self.layer3(x)
```

- 기울기 소멸 문제 발생 → 해결) 렐루 활성화 함수 사용

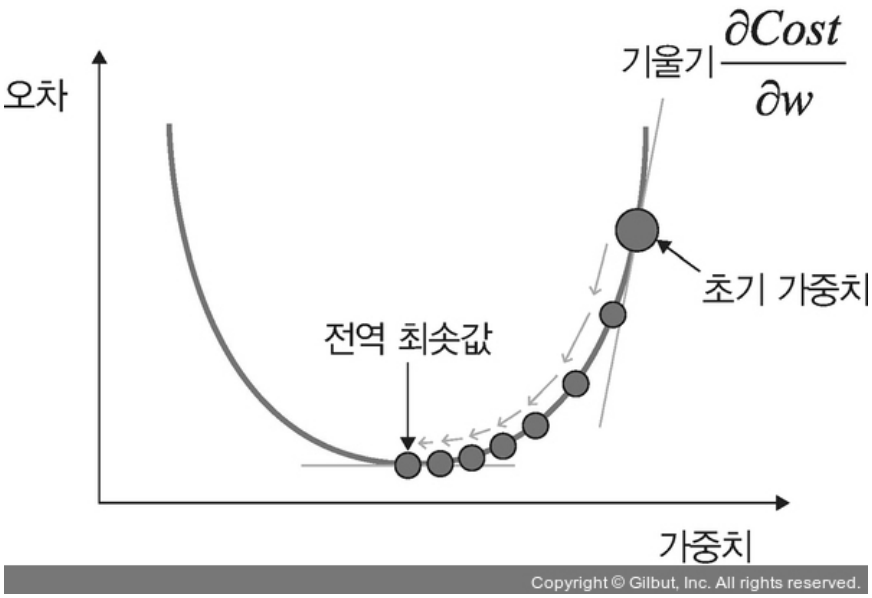
기울기 소멸 문제는 은닉층이 많은 신경망에서 주로 발생하는데, 출력층에서 은닉층으로 전달되는 오차가 크게 줄어들어 학습이 되지 않는 현상입니다. 즉, 기울기가 소멸되기 때문에 학습되는 양이 '0'에 가까워져 학습이 더디게 진행되다 오차를 더 줄이지 못하고 그 상태로 수렴하는 현상입니다.

기울기 소멸 문제는 시그모이드나 하이퍼볼릭 탄젠트 대신 렐루 활성화 함수를 사용하면 해결할 수 있습니다.



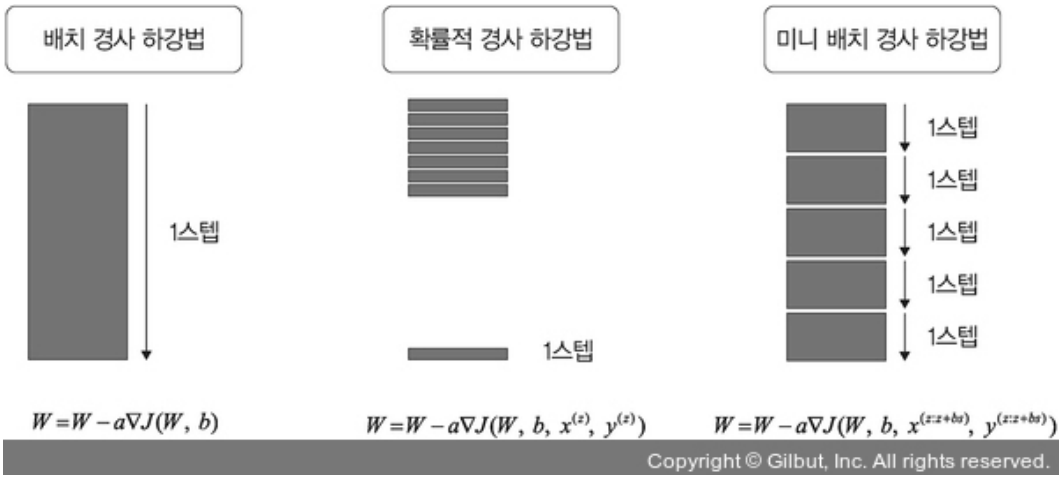
- 성능이 나빠지는 문제 발생 → **미니 배치 경사 하강법**

경사 하강법은 손실 함수의 비용이 최소가 되는 지점을 찾을 때까지 기울기가 낮은 쪽으로 계속 이동시키는 과정을 반복하는데, 이때 성능이 나빠지는 문제가 발생합니다.

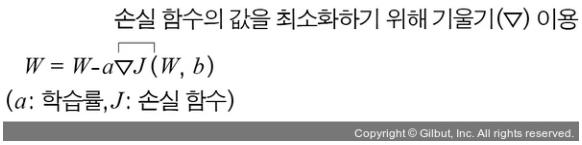


▲ **그림 4-17** 경사 하강법

이러한 문제점을 개선하고자 확률적 경사 하강법과 미니 배치 경사 하강법을 사용합니다.

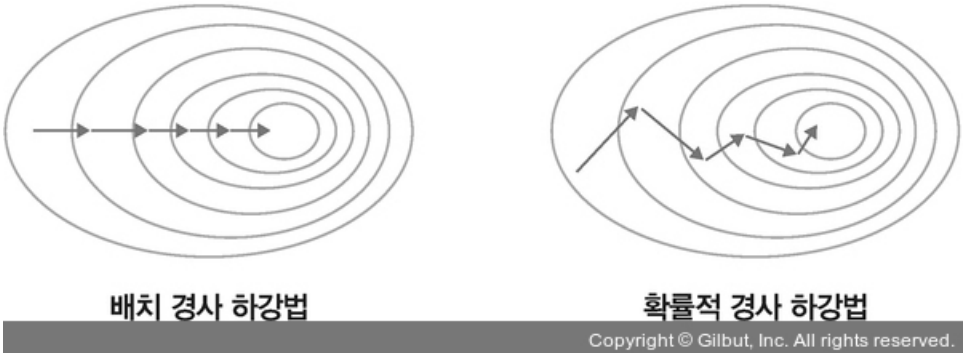


배치 경사 하강법(Batch Gradient Descent, BGD)은 전체 데이터셋에 대한 오류를 구한 후 기울기를 한 번만 계산하여 모델의 파라미터를 업데이트하는 방법입니다. 즉, 전체 훈련 데이터셋(total training dataset)에 대해 가중치를 편미분하는 방법입니다.

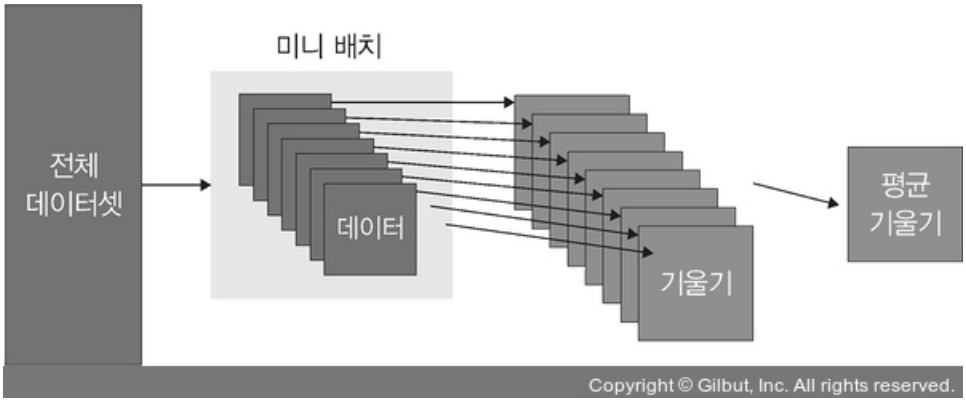


배치 경사 하강법은 한 스텝에 모든 훈련 데이터셋을 사용하므로 학습이 오래 걸리는 단점이 있습니다. 배치 경사 하강법의 학습이 오래 걸리는 단점을 개선한 방법이 확률적 경사 하강법입니다.

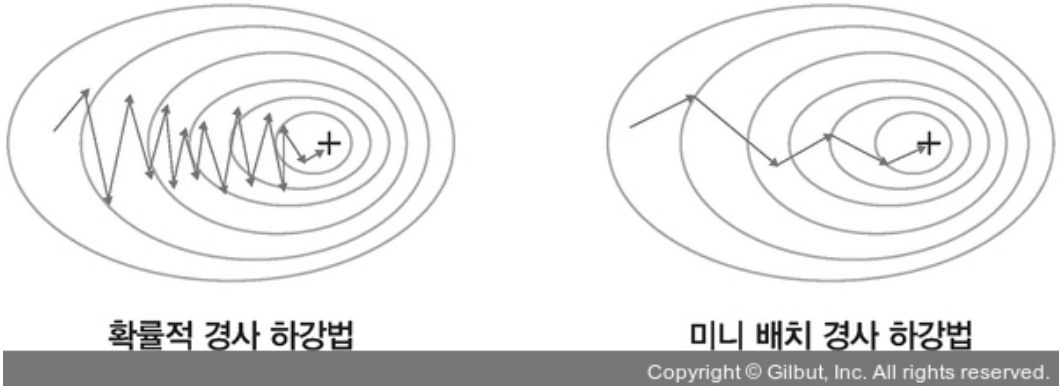
확률적 경사 하강법(Stochastic Gradient Descent, SGD)은 임의로 선택한 데이터에 대해 기울기를 계산하는 방법으로 적은 데이터를 사용하므로 빠른 계산이 가능합니다. 다음 그림의 오른쪽과 같이 파라미터 변경 폭이 불안정하고, 때로는 배치 경사 하강법보다 정확도가 낮을 수 있지만 속도가 빠르다는 장점이 있습니다.



미니 배치 경사 하강법(mini-batch gradient descent)은 전체 데이터셋을 미니 배치(mini-batch) 여러 개로 나누고, 미니 배치 한 개마다 기울기를 구한 후 그것의 평균 기울기를 이용하여 모델을 업데이트해서 학습하는 방법입니다.



미니 배치 경사 하강법은 전체 데이터를 계산하는 것보다 빠르며, 확률적 경사 하강법보다 안정적이라는 장점이 있기 때문에 실제로 가장 많이 사용합니다. 다음 그림의 오른쪽과 같이 파라미터 변경 폭이 확률적 경사 하강법에 비해 안정적이면서 속도도 빠릅니다.

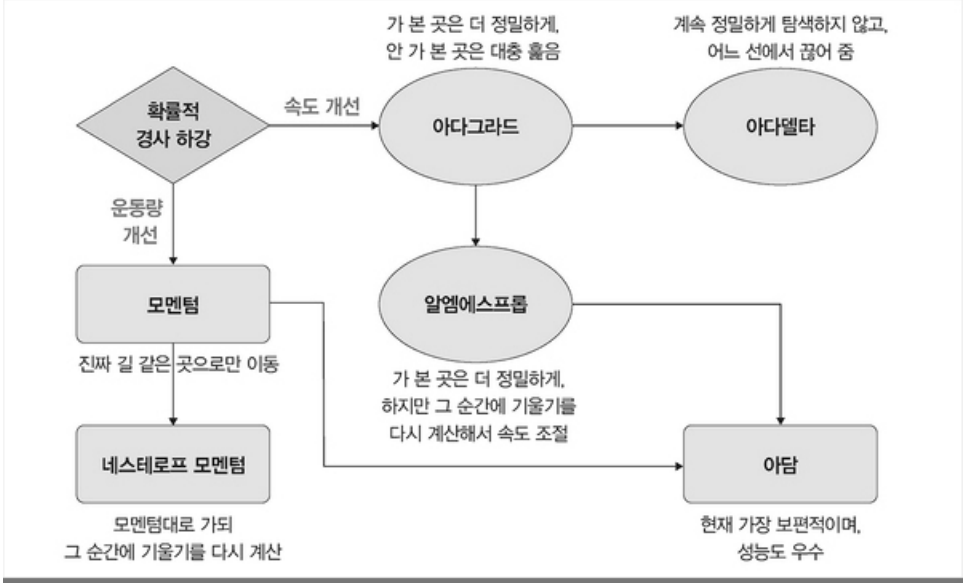



```
class CustomDataset(Dataset):
    def __init__(self):
        self.x_data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
        self.y_data = [[12], [18], [11]]
        def __len__(self):
            return len(self.x_data)
        def __getitem__(self, idx):
            x = torch.FloatTensor(self.x_data[idx])
            y = torch.FloatTensor(self.y_data[idx])
            return x, y
dataset = CustomDataset()
dataloader = DataLoader(
    dataset, ------ 데이터셋
    batch_size=2, ------ 미니 배치 크기로 2의 제곱수를 사용하겠다는 의미입니다.
    shuffle=True, ------ 데이터를 불러올 때마다 랜덤으로 섞어서 가져옵니다.
)
```

💡 NOTE)

- **옵티마이저**

확률적 경사 하강법의 파라미터 변경 폭이 불안정한 문제를 해결하기 위해 학습 속도와 운동량을 조정하는 옵티마이저(optimizer)를 적용해 볼 수 있습니다.



- 속도를 조정하는 방법

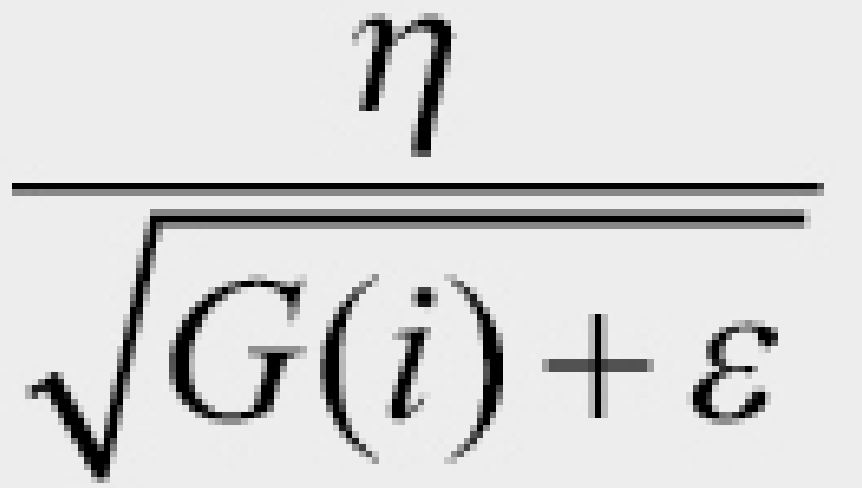
아다그라드(Adagrad, Adaptive gradient)

아다그라드는 변수(가중치)의 업데이트 횟수에 따라 학습률을 조정하는 방법입니다. 아다그라드는 많이 변화하지 않는 변수들의 학습률은 크게 하고, 많이 변화하는 변수들의 학습률은 작게 합니다. 즉, 많이 변화한 변수는 최적 값에 근접했을 것이라는 가정하에 작은 크기로 이동하면서 세밀하게 값을 조정하고, 반대로 적게 변화한 변수들은 학습률을 크게 하여 빠르게 오차 값을 줄이고자 하는 방법입니다.

$$w(i+1) = w(i) - \frac{\eta}{\sqrt{G(i)+\varepsilon}} \nabla E(w(i))$$

$$G(i) = G(i-1) + (\nabla E(w(i)))^2$$

파라미터마다 다른 학습률을 주기 위해 G 함수를 추가했습니다. 이때 G 값은 이전 G 값의 누적(기울기 크기의 누적)입니다. 기울기가 크면 G 값이 커지기 때문에



에서 학습률(η)은 작아집니다. 즉, 파라미터가 많이 학습되었으면 작은 학습률로 업데이트되고, 파라미터 학습이 덜 되었으면 개선의 여지가 많기 때문에 높은 학습률로 업데이트됩니다.

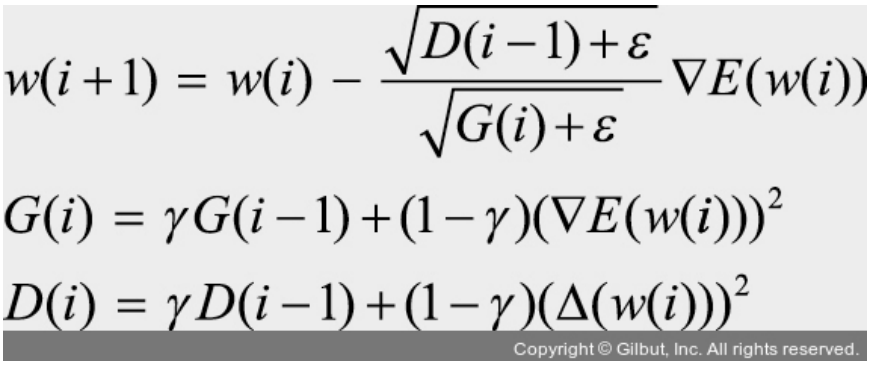
예를 들어 파이토치에서는 아다그라드를 다음과 같이 구현할 수 있습니다.

```
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01) ------ 학습률 기본값은 1e-2
```

하지만 아다그라드는 기울기가 0에 수렴하는 문제가 있어 사용하지 않으며, 대신에 알엠에스프롭을 사용합니다.

아다델타(Adadelta, Adaptive delta)

아다델타는 아다그라드에서 G 값이 커짐에 따라 학습이 멈추는 문제를 해결하기 위해 등장한 방법입니다. 아다델타는 아다그라드의 수식에서 학습률(η)을 D 함수(가중치의 변화량(Δ) 크기를 누적한 값)로 변환했기 때문에 학습률에 대한 하이퍼파라미터가 필요하지 않습니다.



예를 들어 파이토치에서는 아다델타를 다음과 같이 구현할 수 있습니다.

```
optimizer = torch.optim.Adadelta(model.parameters(), lr=1.0) ------ 학습률 기본값은 1.0
```

알엠에스프롭(RMSProp)

알엠에스프롭은 아다그라드의 G(i) 값이 무한히 커지는 것을 방지하고자 제안된 방법입니다.

$$w(i+1) = w(i) - \frac{\eta}{\sqrt{G(i)+\varepsilon}} \nabla E(w(i))$$

$$G(i) = \gamma G(i-1) + (1-\gamma)(\nabla E(w(i)))^2$$

아다그라드에서 학습이 안 되는 문제를 해결하기 위해 G 함수에서 γ(감마)만 추가되었습니다. 즉, G 값이 너무 크면 학습률이 작아져 학습이 안 될 수 있으므로 사용자가 γ 값을 이용하여 학습률 크기를 비율로 조정할 수 있도록 했습니다.

예를 들어 파이토치에서는 알엠에스프롭을 다음과 같이 구현할 수 있습니다.

```
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.01) ------ 학습률 기본값은 1e-2
```