SUB_PJT_01 (AI)

서울 1반 6팀 이민아

- **사전학습**
  - 사전학습1 인공지능
  - 사전학습2 회귀 및 경사하강법
  - 사전학습3 신경망
  - 사전학습4 파이썬 라이브러리

사전학습3 신경망

1. 신경망 (Neural Network)

(1) 정의

> 신경망(Neural Network)는 사람의 두뇌 모양을 흉내내서 만든 모델

- 수많은 노드들과 각 노드들간의 **가중치**로 이루어져 있습니다
- 학습 데이터를 이용해 학습하면서 그 결과값에 따라 각 노드들간의 **가중치를 조금 변경**하며 학습

(2) 델타 규칙

- **오차**가 크면 **가중치를 많이** 조절하고, **오차**가 작으면 **가중치를 적게** 조절
- Adaline, Widrow-Hoff 규칙이라고도 하며, 주어진 정보에 따라 단층 신경망의 가중치를 체계적으로 바꾸어주는 규칙
- 정답을 한 번에 바로 찾는게 아니라 **반복적인 학습 과정을 통해 정답**을 찾아가는 방식

(3) epoch

- 학습 데이터를 한번씩 모두 학습시킨 **횟수**

(4) 오차 역전파 (Backpropagation)

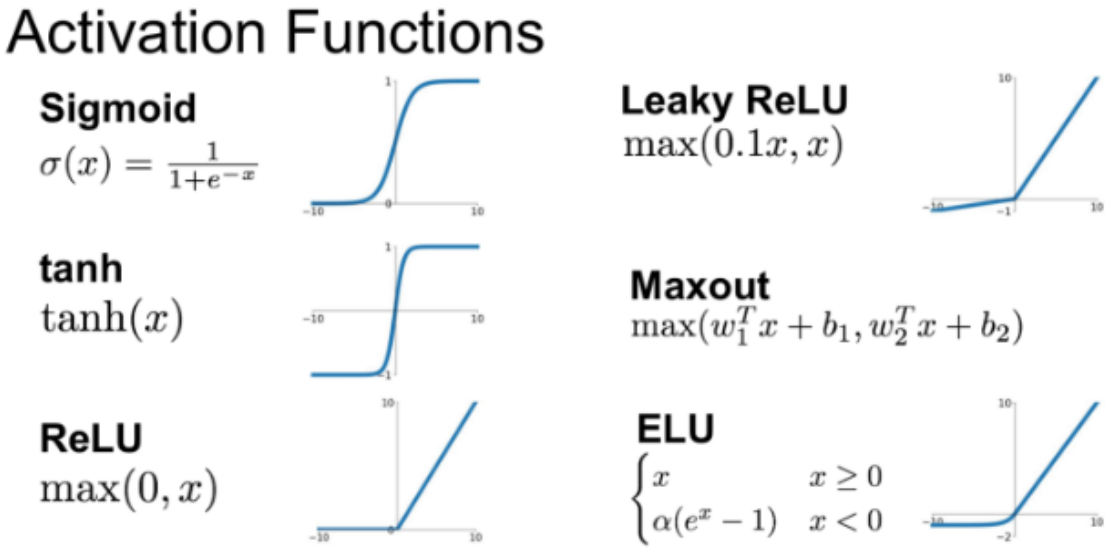
- 델타 학습법만으로는 신경망의 **모든 노드**들을 학습시킬 수 **없다**
- **은닉(Hidden) 계층**은 오차의 정의조차 되어 있지 않고, 정답 또한 정해져 있어서 학습 불가능
- 각 노드를 가중치에 대해 미분으로 계산하여 가중치를 업데이트 하는것은 무리가 있기 때문에
- **출력 노드만 가중치**에 대해 미분하고 그 값을 **이전 노드(역방향)**에 전달하여 재사용
- 출력층에서부터 시작해 **거꾸로 추적**해가며 오차에 따라 가중치를 조절하는 방법
- 순전파 이후에 출력값이 오류일 경우 출력층에서 입력층 방향으로(역방향) 가중치가 **더 이상 업데이트되지 않을때까지 반복**

2. 활성화 함수 (Activation Function)

(1) 정의

> 출력값을 활성화를 일으키게 할 것이냐를 결정하고, 그 값을 부여하는 함수

- **XOR 문제**를 다층 퍼셉트론으로 해결했지만, 은닉층만 늘린다고 선형분류기를 비선형분류기로 바꿀 수는 없음
- 선형 시스템의 경우 망이 아무리 깊어지더라도(은닉층의 수가 많아진다는 것), 1층의 은닉층으로 구현
- **선형인 멀티퍼셉트론**에서 **비선형 값을 얻기 위해** 사용하기 시작
- 이에 대한 해결책인 활성화 함수는 입력값에 대한 **출력값이 선형이 아니므로** 선형분류기를 **비선형 분류기**로 만들 수 있음 (binary step, Sigmoid, ReLU, TanH 등)



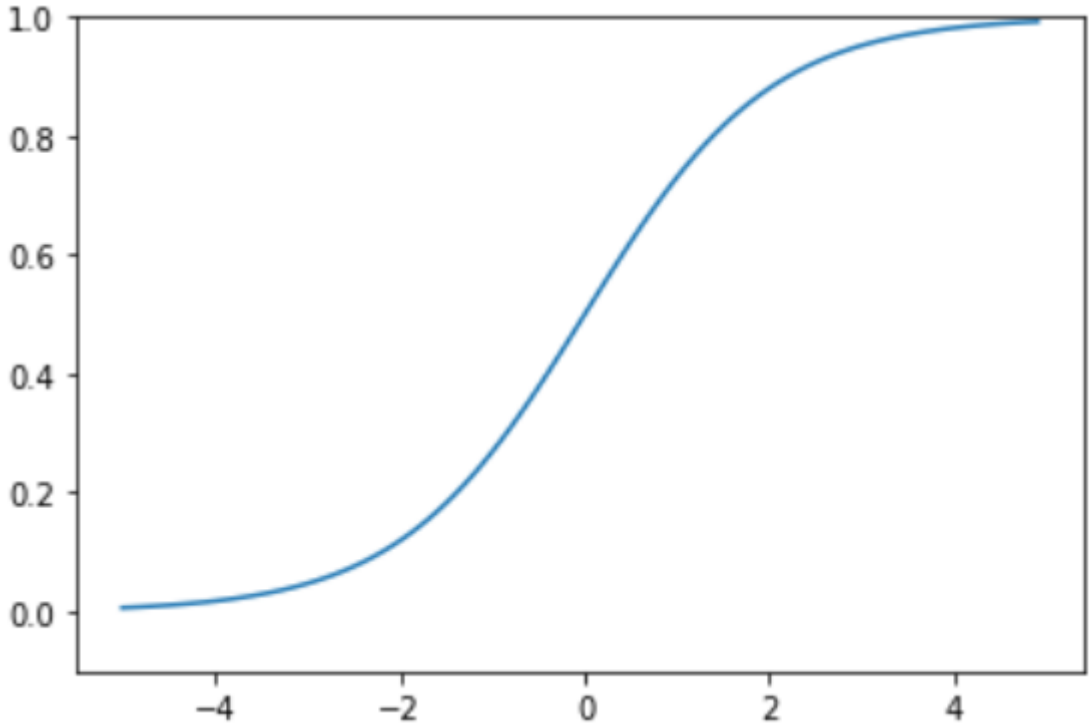
(2) 경사각 소실 (Vanishing Gradient)

- 신경망의 계층을 깊게 할수록 성능이 더 떨어지는 원인 3개
  - **경사각 손실 (Vanishing Gradient)**
  - **과적합(Overfitting)**
  - 많은 계산량
- 활성화 함수(Activation Function)으로 많이 사용하는 **시그모이드(Sigmoid) 함수**는 최대 기울기가 `0.3`을 넘지 않습니다.
- 곱하면 곱할수록 0에 가까워지고 결국 0이 되는 현상이 발생해서 **기울기가 사라집니다**
- 대안으로 시그모이드 함수대신 **ReLU** 함수를 사용해서 해결할 수 있습니다

(3) 시그모이드 함수 (Sigmoid)

- Logistic 함수
- x 값이 작아질수록 0에 수렴하고, 커질수록 1에 수렴
- **경사각 소실 (Vanishing Gradient)**

  input값이 일정이상 올라가면 **미분값이 거의 0에 수렴**하게된다. 이는 |x||x|값이 커질 수록 Gradient Backpropagation시 미분값이 소실될 가능성이 크다

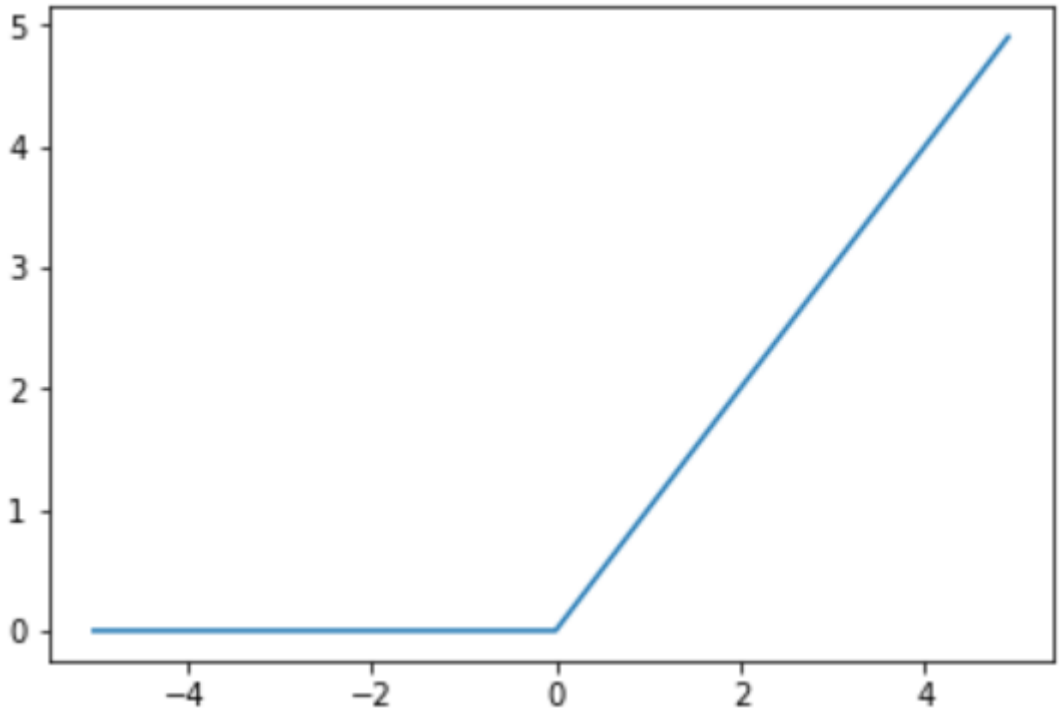



```
def sigmoid(x):
    return 1/(1+np.exp(-x))

x = np.arange(-5,5,0.1)
# -5 ~ 5 범위에서 0.1 간격으로 값을 출력
y = sigmoid(x)
plt.plot(x,y)
plt.ylim(-0.1,1.0)
plt.show()
```

(4) ReLU 함수 (Rectified Linear Unit)

- 최근 가장 많이 사용되는 활성화 함수
- sigmoid, tanh 함수와 비교시 학습이 훨씬 빨라진다
- 연산 비용이 크지않고, 구현이 매우 간단
- 입력이 특정값을 넘으면 입력이 그대로 출력되고, 0을 넘지 않을시 0을 반환하는 함수
- y = x (when x≥0), 0(when x<0)



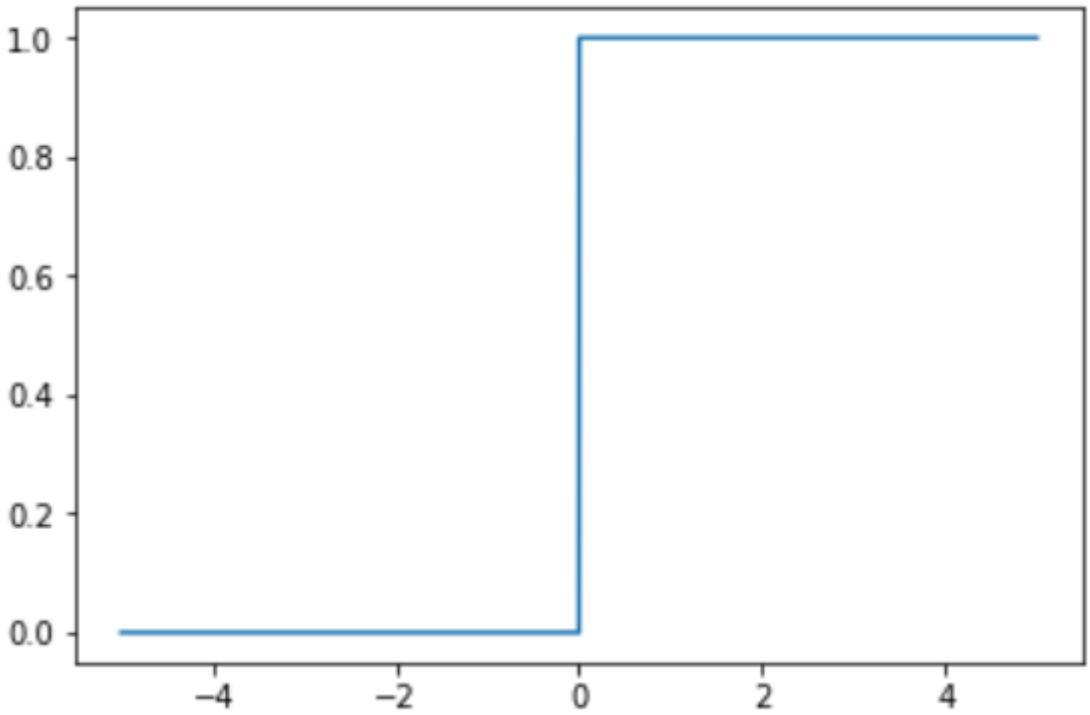

```
def ReLU(x):
    return np.maximum(0,x)
    # numpy 객체 np에 있는 maximum 메소드
    # np.maximum(0,x)는 0과 x 두 인자 가운데 큰것을 반환하는 메소드 함수

x = np.arange(-5, 5, 0.1)
y = ReLU(x)

plt.plot(x,ReLU(x))
plt.show()
```

(5) Step Function

- 계단모양 함수로, 특정값 이하는 0, 이상은 1로 출력하도록 만들어진 함수
- y = 1 (when x≥0), 0 (when x<0)




```
def step_function(x):
    return np.array(x>0.0, dtype = np.int)
    # 1번째 인자인 x>0.0은 x>0.0일 경우 True, 아닐경우 False를 반환
    # 2번째 인자인 dtype은 Data 타입을 어떤것으로 둘 것이냐 라는 함수인데 ,0과 1로 이루어
진 함수이므로, int형을 선택

x = np.arange(-5, 5, 0.001)
# -5 ~ 5 범위에서 0.001 간격으로 값을 출력하고, 0을 기준으로 0과 1의 출력을 구분지어
y = step_function(x)
plt.plot(x,y)
plt.show()
```

3. 손실 함수 (Loss Function)

(1) 정의

- **예측값과 정답** 사이의 **오차**를 정의하는 함수
- 머신러닝의 목적은 오차의 합을 **최소화**하여 **최적의 결과 값**을 도출하는 것 (잘 학습된 모델)
- **비용 함수 (cost function)** 또는 목적 함수 (object function)
- 신경망을 학습할 때 정확도를 지표로 삼아서는 안된다. **정확도**를 지표로 하면 매개변수의 미분이 대부분의 장소에서 0이 되기 때문에 정확도는 매개변수의 미소한 변화에는 거의 반응을 보이지 않고, 반응이 있더라도 그 값이 불연속적으로 갑자기 변화한다. 이는 계단함수를 활성화 함수로 사용하지 않는 이유와도 같다
- 신경망 학습에서는 **최적의 매개변수 (가중치와 편향) 값**을 탐색할 때 **손실 함수**의 값을 가능한 작게 하는 매개변수 값을 찾는다. 이 때 매개변수의 미분을 계산하고, 그 미분 값을 단서로 매개변수의 값을 서서히 갱신하는과정을 반복

(2) 교차 엔트로피 오차(Cross Entropy Error, CEE)

- log는 밑이 e인 자연로그 이다. yk는 신경망의 출력, tk는 정답레이블

$$E = -\sum_{k} t_k log y_k$$

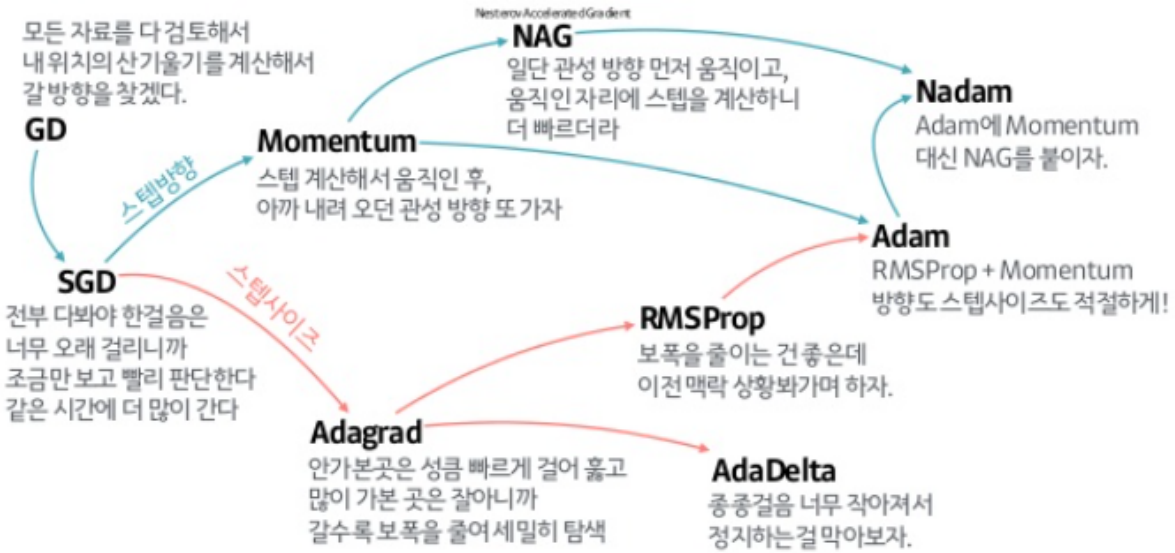
(3) 평균 제곱 오차 (Mean Squared Error, MSE)

- 오차(예측값과 목표 값의 차이)를 제곱하여 더한 다음 그 수로 나눈 값

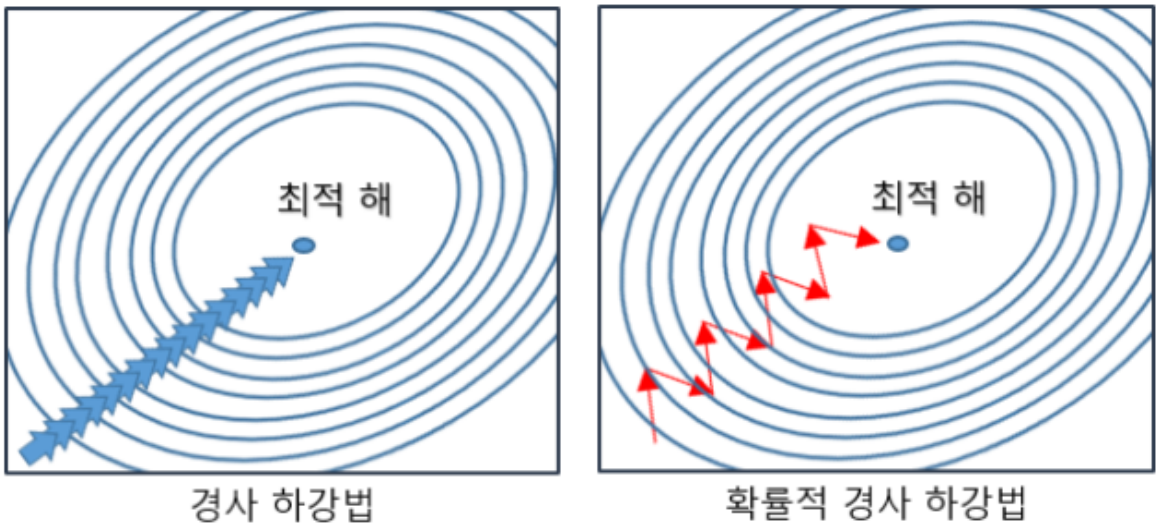
$$MSE = \frac{1}{n}\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2$$

- **정답**에 가까울수록 **MSE 값이 작고**, 오답에 가까울수록 큰 값을 가짐
- 오차가 크면 클수록 MSE는 더 큰 값을 가지므로 **학습에 유리함**

4. 최적화 함수 (optimization)



(1) 확률적 경사 하강법 Stochastic Gradient Descent (SGD)

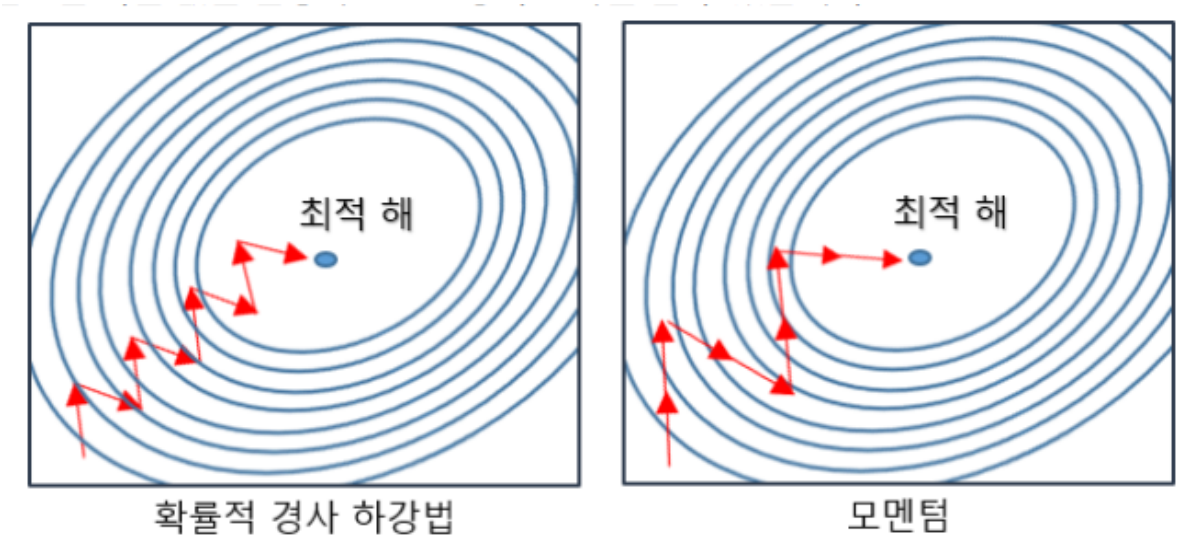


경사 하강법 확률적 경사 하강법

- 파라미터를 업데이트할 때 무작위로 샘플링된 학습 데이터를 하나씩만 사용하여 추출된 각각 **한개의 데이터**에 대해서 cost function(손실함수)의 Gradient 계산 후 업데이트 하는 알고리즘
- 확률적이기에 불안정하여 아래와 같이 **요동**치는 것을 볼 수 있으며 **자주 업데이트한다**
- 성능 개선 정도를 빠르게 확인
- 최소 cost에 수렴했는지의 판단이 상대적으로 어려움
- **Shooting**이 발생하므**로 Local Optimal 빠질 리스크가 다소 적어진다**
- 훈련 데이터 한개씩 처리하므로, GPU 성능에 대해서 **전부 활용이 불가하다**
- 때로는 **Global Optimal 찾지 못할 수도 있다**
- **속도는 개선**되었으나 **정확도가 낮다**

```
weight[i] += - learning_rate * gradient
```

(2) 모멘텀(Momentum)



- 관성, 탄력, 가속도
- 경사 하강법과 마찬가지로 매번 기울기를 구하지만, 가중치를 수정하기전 이전 수정 방향(+,-)를 참고하여 같은 방향으로 일정한 비율만 수정
- 수정이 양(+) 방향, 음(-) 방향 순차적으로 일어나는 지그재그 현상이 줄어들고, 이전 이동 값을 고려해여 일정 비율만큼 다음 값을 결정하므로 관성의 효과
- av 값을 더해주며 a는 고정된 상수값이고(ex 0.9) v는 물체의 속도
- 해당 방향으로 진행할 수록 공이 기울기를 따라 구르듯 힘을 받는다
- 기존의 SGD를 이용할 경우 좌측의 local minima에 빠지면 gradient가 0이 되어 이동할 수가 없다
- momentum 방식의 경우 기존에 이동했던 방향에 관성이 있어 이 local minima를 빠져나오고 더 좋은 minima로 이동할 것을 기대할 수 있다

```
# 파이썬 소스 코드
v = m * v - learning_rate * gradient
weight[i] += v
```

```
# Tensorflow 소스 코드
optimize =
tf.train.MomentumOptimizer(learning_rate=0.01,momentum=0.9).minimize(loss)
```

(3) AdaGrad

- 변수의 업데이트 횟수에 따라 학습률(Learning rate)를 조절하는 옵션이 추가된 최적화 방법
- 변수란 가중치(W) 벡터의 하나의 값(w[i])
- 학습률을 정하는 효과적 기술로 학습률 감소(learning rate decay)가 있다. 이는 학습을 진행하면서 학습률을 점차 줄여나가는 방법
- 가장 간단한 방법은 전체 학습률 값을 일괄적으로 낮추는 것이지만, 이를 더 발전시킨 것이 AdaGrad이다. AdaGrad는 '각각의' 매개변수에 '맞춤형'값을 만들어준다

```
# 파이썬 소스 코드
g += gradient**2
weight[i] += - learning_rate ( gradient / (np.sqrt(g) + e)
```

```
# Tensorflow 소스 코드
optimizer = tf.train.AdagradOptimizer(learning_rate=0.01).minimize(loss)
```

(5) RMSprop(알엠에스프롭)

- 아다그라드의 G(t)의 값이 무한히 커지는 것을 방지하고자 제안된 방법
- **지수 이동평균**을 이용한 방법

```
# 파이썬 소스 코드
g = gamma * g + (1 - gamma) * gradient**2
weight[i] += -learning_rate * gradient / (np.sqrt(g) + e)
```

```
# Tensorflow 소스 코드
optimize =
tf.train.RMSPropOptimizer(learning_rate=0.01,decay=0.9,momentum=0.0,epsilon=1e-
10).minimize(cost)
```

(6) Adam

- Momentum + RMSprop
- RMSprop의 특징인 gradient의 제곱을 지수평균한 값을 사용하며 Momentum의 특징으로 gradient를 제곱하지 않은 값을 사용하여 지수평균을 구하고 수식에 활용

```
# Tensorflow 소스 코드
optimizer =
tf.train.AdamOptimizer(learning_rate=0.001,beta1=0.9,beta2=0.999,epsilon=1e-08
).minimize(loss)
```

(7) AdaDelta

- Adagrad + RMSprop + Momentum 모두를 합친 경사하강법

```
# Tensorflow 소스 코드
optimizer = tf.train.adadeltaoptimizer(learning_rate=0.001, rho=0.95,
epsilon=1e-08).minimize(loss)
```

(8) Nesterov Accelrated Gradient(NAG, 네스테로프 모멘텀)

- momentum값과 gradient값이 더해저 실제(actual)값을 만드는 기존 모멘텀과 차별
- momentum값이 적용된 지점에서 gradient값이 계산

```
# 파이썬 소스 코드
v = m * v - learning_rate * gradient(weight[i-1]+m*v)
weight[i] += v
```

```
# Tensorflow 소스 코드
optimize =
tf.train.MomentumOptimizer(learning_rate=0.01,momentum=0.9,use_nesterov=True).mi
nimize(loss)
```