출처가 명시되지 않은 모든 자료(이미지 등)는 조성현 강사님 블로그 및 강의 자료 기반.

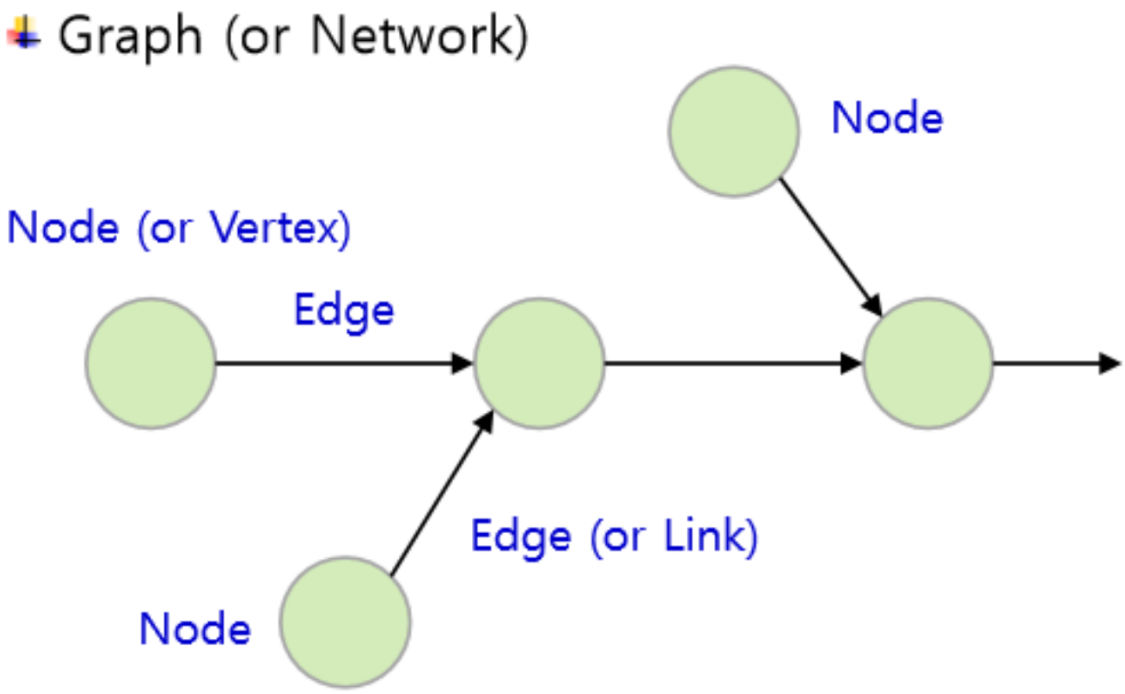
Tensorflow 2.2 ver

<< 딥러닝 - Tensorflow >>

[Tensorflow]

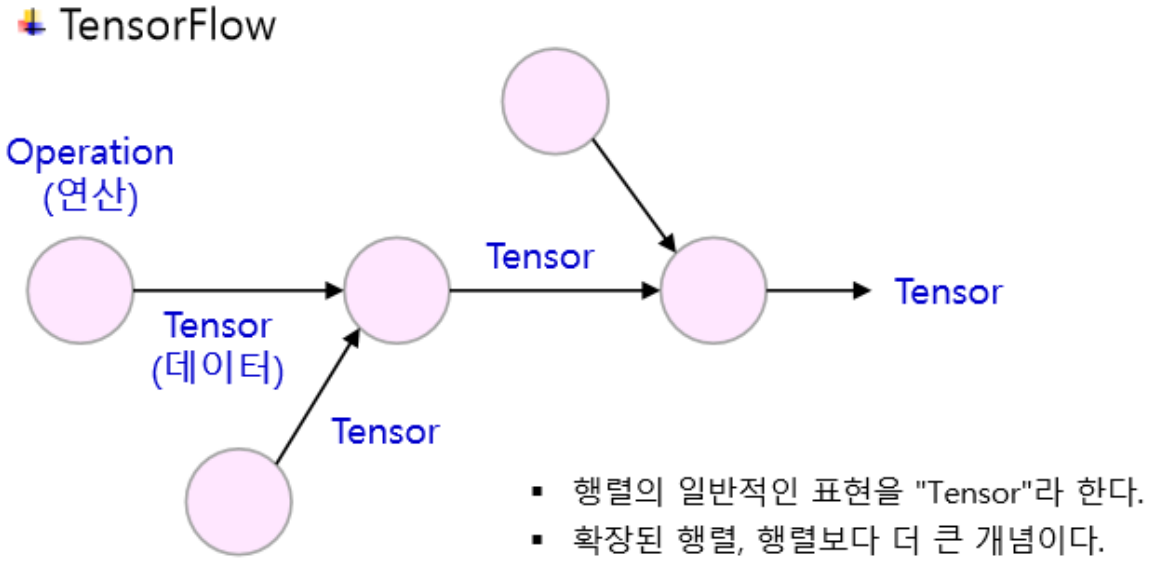
1. 개요

Tensorflow는 모든 연산을 그래프로 수행한다. 그래프의 구조를 나타내면 다음과 같다.



- 노드(Vertex) : 정점. 일반적인 그래프 자료구조에서 노드에는 데이터가 표현된다.
- 간선(Edge/Link) : 노드와 노드를 연결. 일반적인 그래프 자료구조에서 간선에는 노드와 노드 사이의 관계 정보가 포함된다.

특정 문제를 풀 때 그래프 구조가 잘 작동하는 경우가 많다. Tensorflow도 인공신경망 학습 등을 통해 문제를 풀 때 그래프 구조를 사용한다. Tensorflow에서의 그래프 구조는 다음과 같다.



Tensorflow의 그래프 구조는 일반적인 그래프 구조와는 다르다. Tensorflow 그래프 구조에서 노드에 해당하는 것은 `operation`, 간선에 해당하는 것은 `tensor`가 된다.

`operator`는 `tensor`를 받아서 연산을 수행한다. `operator`는 입력을 action을 통해 그대로 출력으로 내보내주는 `bypass` 역할을 한다. `tensor`와 `operator`를 잘 구분해야 나중에 헷갈리지 않는다. `operator` 즉, node에는 절대로 숫자가 들어가면 안 된다. Tensorflow에서 `operator`는 무조건 연산을 나타낸다.

상수 텐서를 생성하고 데이터 형태를 살펴 보자.

```
import tensorflow as tf

a = tf.constant(2)
>>> <tf.Tensor: shape(), dtype=int32, numpy=2>

c = a.numpy()
```

Tensorflow에서 2라는 상수는 shape이 없는 `scalar` 형태의 자료 구조이다. 자료형은 32bit의 `integer`이고, numpy로 생성한 데이터로는 2와 같다.

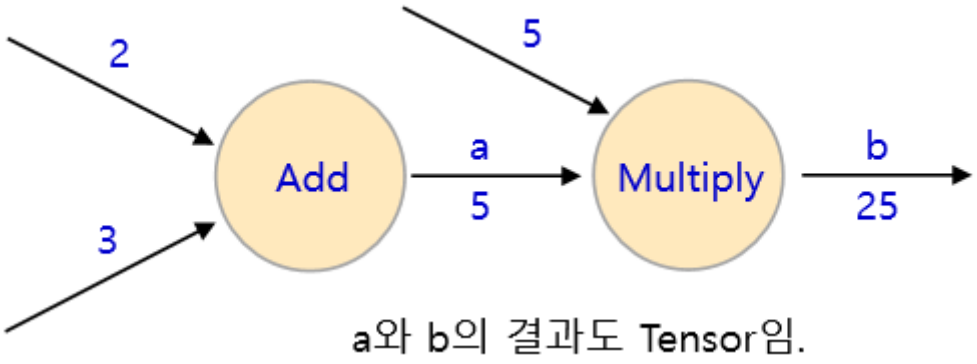
`tensor`를 `NumPy` 형태의 자료 구조로 보고 싶다면, `.numpy()` 메소드를 사용하면 된다.

이제 간단한 Tensorflow 연산을 코드 예시로 보자.

```
import tensorflow as tf

a = tf.add(2, 3)
b = tf.multyply(5, a)
```

- `tf.add` : 더하기 연산을 수행하는 `operation`이다.
- `a` : 더하기 연산의 결과로 도출된 `tensor`이다.
- `tf.multiply` : 곱하기 연산을 수행하는 `operation`이다.
- `b` : 5와 a라는 텐서를 곱하는 연산의 결과로 도출된 `tensor`이다.

위의 코드를 그래프로 나타내면 다음과 같다.



그 이름에서 알 수 있듯, Tensorflow는 `tensor`를 흘려 보내면서(`flow`) 데이터를 처리한다. `tensor`와 `operation`에 대해 잘 알아두자.

참고

Tensorflow 1.X 버전에서는 데이터를 받아들이기 위해 `tf.placeholder`라는 텐서를 별도로 지정해야 했다. 각 변수도 `tf.Variable`로 할당해야 했다. 이후 그래프를 생성한 후 `tf.Session` 객체를 통해 session을 열어서 그 안에서 그래프를 실행해야 했다.

이제 2.X 버전에서는 `NumPy`와 더불어 일반 파이썬 코드 짜듯 짜면 된다. 훨씬 쉬워졌다!

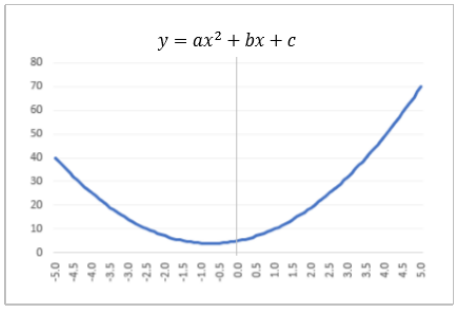
2. Gradient Descent

2차방정식 계수 추정

관측된 데이터 `y`를 `a*x^2 + b*x + c`라는 모형을 활용해서 설명하고자 할 때, 적절한 계수 `a`, `b`, `c`를 구한다고 하자.

관측 데이터

x	y
-5.00	40.00
-4.90	38.32
-4.80	36.68
-4.70	35.08
-4.60	33.52
-4.50	32.00
-4.40	30.52
-4.30	29.08
-4.20	27.68
-4.10	26.32
-4.00	25.00
-3.90	23.72
-3.80	22.48
-3.70	21.28



$error(e) = y - (ax^2 + bx + c) \longrightarrow loss\ (L) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^2}$

목표 : $\min_{a,b,c}(L)$ ← loss를 최소로 만드는 a, b, c를 찾는 것

$\frac{\partial L}{\partial a} = 0 \quad \frac{\partial L}{\partial b} = 0 \quad \frac{\partial L}{\partial c} = 0$ ← 수학적 풀이

$a \leftarrow a - \alpha \frac{\partial}{\partial a} loss$

$b \leftarrow b - \alpha \frac{\partial}{\partial b} loss$

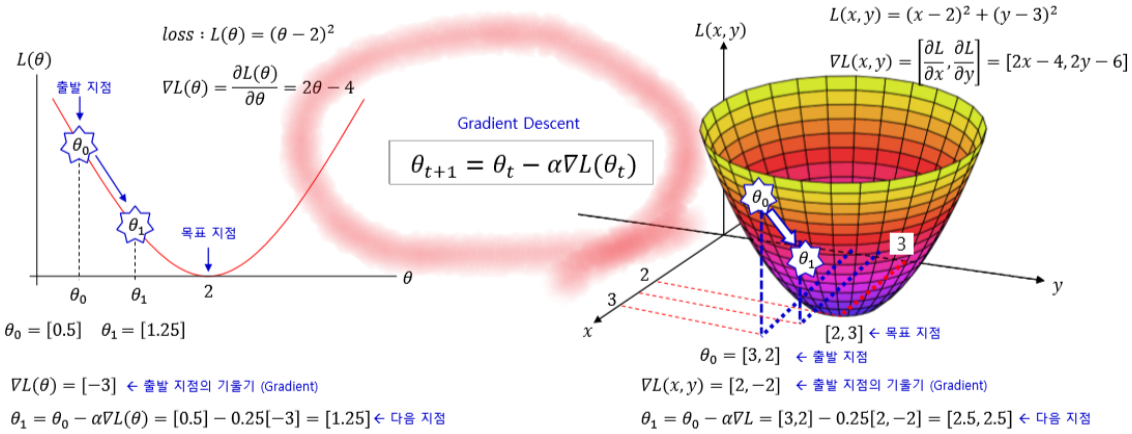
$c \leftarrow c - \alpha \frac{\partial}{\partial c} loss$

- 컴퓨터를 이용한 수치적 풀이
- loss (L) 함수의 최서점을 찾아가는 가장 기본적인 방법론은 Gradient Descent 이다.

수학으로는 error를 다음과 같이 정의하고, error, 즉 loss function을 최소로 만드는 값을 미분을 통해 찾으면 된다.

그러나 컴퓨터는 다른 방식으로 접근한다. 수치적으로 랜덤한 값을 잡아 주고, `a`, `b`, `c`를 업데이트하며 loss function의 최저점을 찾아 가는 방법을 사용한다. **Gradient Descent** 방법이라고 한다.

loss function의 최저점을 찾아가는 과정을 자세히 살펴 보자. loss function이 error를 제곱한 2차 함수 형태이므로, loss function을 2차 함수 형태의 그래프로 나타내 보자.



위의 예시와는 다르게, 그냥 아무 예시로나 잡은 loss function이다.

목표는 loss function이 최소가 되는 지점인 theta = 2이다. 이 지점에서는 loss function의 **기울기(gradient)가 0**이 된다.

처음 theta는 임의의 값으로 시작한다. 예컨대, 0.5에서 시작한다. 그 때의 gradient는 -3이 된다. 찾아야 하는 목표 gradient값인 0보다 작기 때문에, theta를 오른쪽으로 움직인다.

이 때, theta를 얼마나 움직일지를 결정하는 변수가 **학습률**이다.

이런 방식으로 theta를 업데이트한다. 그것을 식으로 정리한 것이 위의 그림에서 가운데 동그라미 친 부분이다. 이전 step의 theta가 있고, 그 때 loss function의 gradient를 계산한다. 그리고 gradient에 학습률인 alpha 만큼을 곱해 이후 step의 theta 위치를 결정한다.

이차방정식의 계수를 결정하는 위의 문제에서는 각 계수 a, b, c에 대해 모두 이런 방식을 따르며 계수를 결정하면 된다.

Stochastic Gradient Descent

$$Error(e) = \{y - (ax^2 + bx + c)\}$$
$$Loss = \Sigma {e_i}^2$$
$$MSE = \frac{1}{n}\Sigma {e_i}^2$$

loss의 정확한 값에 관심이 있는 게 아니니까, `1/n`을 곱하거나 `1/2`를 곱해서 MSE의 값을 측정하는 것 자체는 중요한 게 아니다. **결국 loss가 커지는지 작아지는지에만 관심이 있다.**

여기서 중요한 것은 loss function을 정의할 때, error의 제곱합을 활용한다는 것이다. 더 중요한 것은, **error에 제곱을 취한다**는 것이다. 왜 `y-y_hat`도 아니고, `abs(y-y_hat)`도 아니고, 하필 제곱의 합일까?

위의 Gradient Descent 개념과 연관이 있다.

일차함수로 정의한다면 + 에러 값과 - 에러 값의 부호 차이로 인해 에러 합이 상쇄되어 버린다.

한편, 만약 loss function을 error의 절댓값으로 정의한다면, 그 그래프 특징상 gradient 값의 변화폭이 일정하게 된다. 심지어 우리가 찾아야 할 목표 지점은 미분이 되지 않는 지점이다. 어디서 멈춰야 할지 판단되지 않는다.

그래서 **제곱합을 사용**한다.

실습 1. 2차 방정식 계수 추정

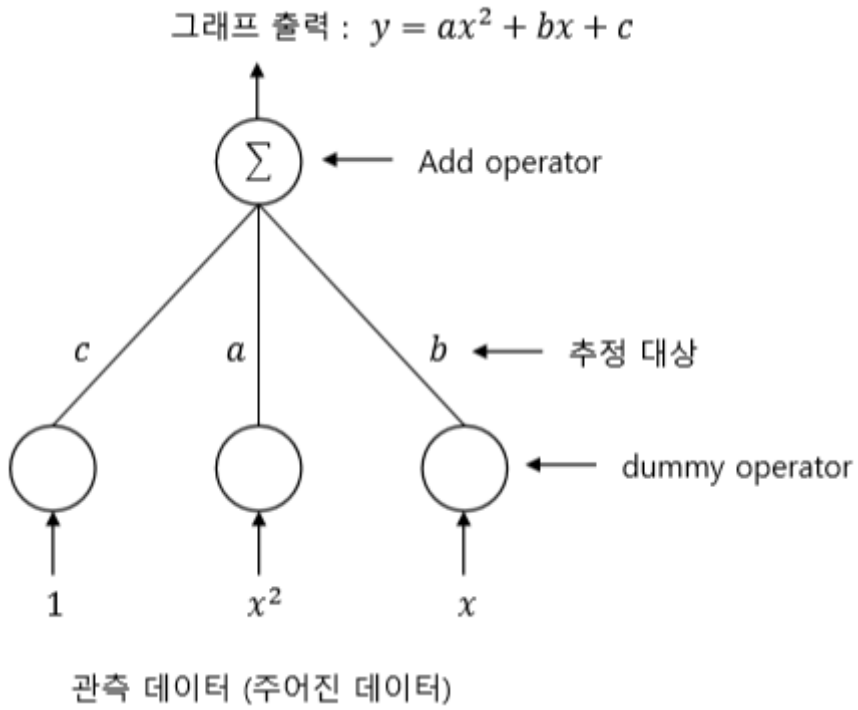
찾아야 할 핵심 파라미터 `w1 = 2`, `w2=3`, `b=5`가 나와야 한다.

```
# y = 2*x^2 + 3*x + 5, 데이터 생성
x = np.array(np.arange(-5, 5, 0.1))
y = 2*x*x + 3*x +5

# w1, w2, b를 찾기 위해 학습을 진행한다.
lr = 0.01 # learning_rate

# 계수 설정 : 초기값
w1 = tf.Variable(1.0)
w2 = tf.Variable(1.0)
b = tf.Variable(1.0)

# 학습
histLoss = []
for epoch in range(1000): # 학습횟수 100
    with tf.GradientTape() as tape:
        loss = tf.sqrt(
            tf.reduce_mean(
                tf.square(w1*x*x + w2*x + b - y)  # RMSE
            )
        )

        # loss에 대한 각 variable의 미분값
        dw1, dw2, db = tape.gradient(loss, [w1, w2, b])

    # variable update
    w1.assign_sub(lr*dw1) # a = a - lr*da
    w2.assign_sub(lr*dw2) # b = b - lr*b.numpy()와 동일
    b.assign_sub(lr*db)


    histLoss.append(loss) # loss 기록
    if epoch % 10 == 0:
        print("epoch = %d, loss = %.4f" %(epoch, histLoss[-1]))

# 추정 결과
print(' ===== 추정 결과 ======')
print('       w1 = %.2f' % w1.numpy()) # numpy 형식으로 확인
print('       w2 = %.2f' % w2.numpy())
```

```
print('       b = %.2f' % b.numpy())
print(' final loss = %.4f' % loss.numpy())
```

- loss : RMSE 사용.
- `tf.Variable()` : 변수의 초기값의 shape 및 data type을 결정한다. 이후 assign을 통해 값이 변화하게 된다.
- `tf.GradientTape()` : 앞의 인자가 피미분(?)함수, 뒤의 인자가 미분할 변수가 된다. 미분값을 구하는 함수라고 생각하자. 위의 코드에서는 loss에 대해 `w1`, `w2`, `b`에 대해 미분한 값을 찾아 기록한다.
- `tf.assign_sub` : `tf.Variable`의 메소드. 데이터에서 값을 빼 준다. variable update 과정에서, learning rate에 gradient 미분값을 곱한것 만큼을 감소시키는 역할을 한다.

위의 코드를 그래프로 표현하면 다음과 같다.

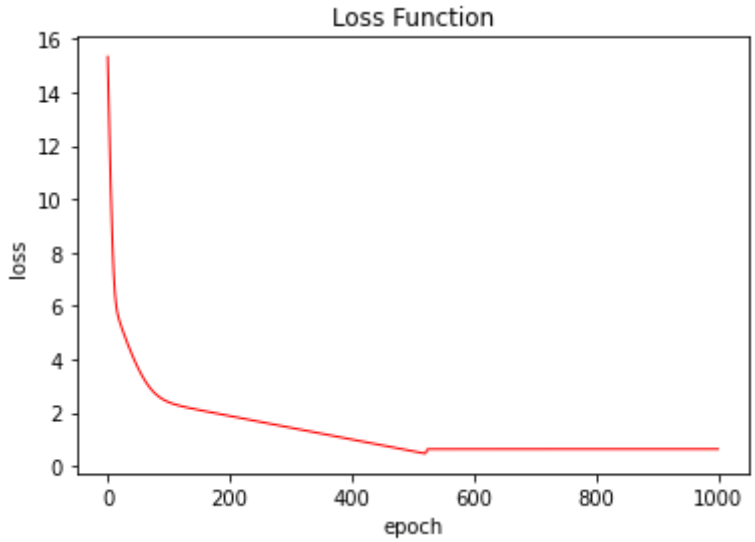


> *참고*
>
> operator들이 전부 다 뉴런. 뉴런으로 입력되는 데이터를 수상돌기, 뉴런에서 출력되는 데이터를 축색돌기, 다음 뉴런으로 데이터를 넘길 때 어느 정도의 비중을 통과시킬지 결정하는 파라미터를 시냅스라고 표현한다.
>
> Tensorflow 그래프를 위와 같이 표현하면 인공신경망 개념. 딥러닝의 기본이 된다.

- 결과
  - epoch이 증가하며 loss 값이 점차 감소한다.
  - 추정 결과는 기존에 의도한 계수 값과 비슷하게 나온다.

```
epoch = 0, loss = 15.3396
epoch = 10, loss = 7.1791
epoch = 20, loss = 5.3826
epoch = 30, loss = 4.7400
```

```
epoch = 40, loss = 4.1643
...
epoch = 960, loss = 0.6283
epoch = 970, loss = 0.6283
epoch = 980, loss = 0.6283
epoch = 990, loss = 0.6283

 ===== 추정 결과 ======
       w1 = 1.95
       w2 = 3.00
       b = 4.97
 final loss = 0.6283
```



- early stopping 조건을 custom해 보자.
  - 처음 강사님이 제시한 조건: 모든 gradient 미분 값이 일정 정도 이하로 작아지면 멈추도록 하자.
  - 그런데 문제가 있다. `da.numpy()`를 관찰해 보면, 어느 순간 커지는 부분이 있다. 지금 축이 3개고 loss function이 4차원에 그려지게 된다. 따라서 `dw1`, `dw2`, `db`까지 모두 다 보면, 컴퓨터의 계산은 무조건 loss가 최소화되는 방향으로 가니까, `dw2`가 작아질 때 `dw1`은 커질 수도 있다. 3개를 다 봐야 하는데, 4차원 공간이라서 머릿속으로는 그릴 수가 없다.
  - `or` 조건을 써서 바꾸도록 하자.
  - 내가 푼 방법: loss 값이 4번 연속 감소하지 않으면 멈추도록 했는데, 이건 안 되나?

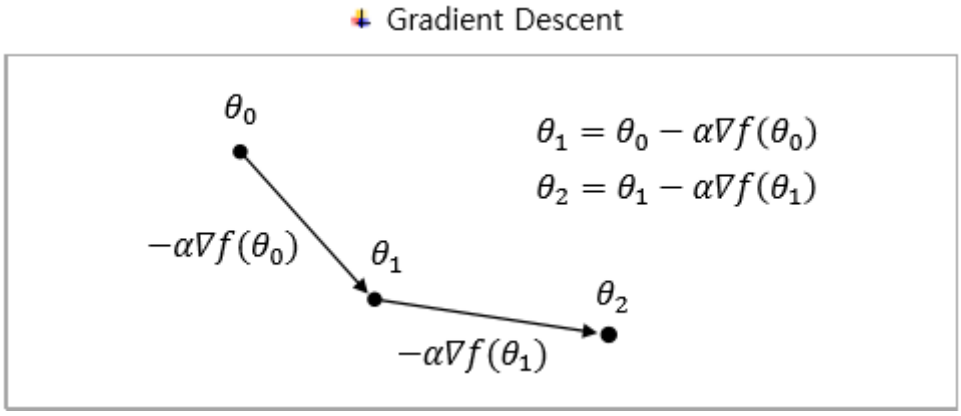
3. Optimizer

학습을 위해 loss function을 정의하고, loss를 줄여야 함을 알았다. 그런데 loss를 줄이는 방법이 위와 같이 단순히 `Gradient Descent`만 있는 것은 아니다. 가급적이면 짧은 경로로 빠르게 loss를 줄이기 위한 방법론이 여러 가지가 연구되어 왔다.

Gradient Descent가 기본이 되는 optimizer이기 때문에, Gradient Descent Optimizer에서 어떤 것을 수정하는지에 따라 그 방향이 바뀌어 왔다..

3.1. Gradient Descent

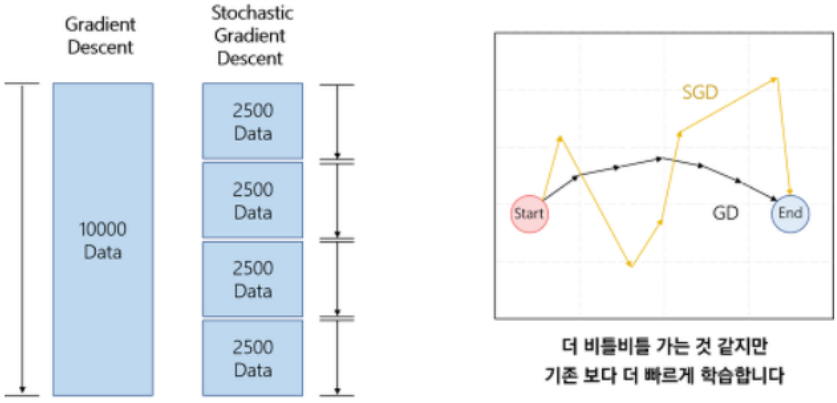
가장 기본적인 이동 방법이다. Loss function의 직전 gradient를 반영하여 gradient를 줄여 나가는 방식으로 이동한다.



Stochastic Gradient Descent

> 참고 : https://gomguard.tistory.com/187

Gradient Descent가 전체 데이터를 고려해서 한 번에 가중치를 수정하기 때문에, 배치를 통해 나누어서 학습을 진행하자는 아이디어에서 나왔다.



> Stochastic 방법을 사용해서 기존 방법보다 빨라지기는 했지만 만족할만한 수준은 아니었습니다. 또한 Cost 의 최소점이 아닌 극소점을 찾은 뒤 더 이상 학습이 되지 않는 현상이 발생하기 시작했습니다.

이후

Gradient Descent 식에서 **Gradient**를 수정하는 방식과 **Learning Rate**인 alpha를 조절하는 adaptive 방식으로 나누어 옵티마이저 연구가 진행되어 왔다.

큰 그림을 먼저 잡고 들어가자면,

- Gradient를 수정한 관성 위주의 `Momentum`, `NAG`,
- Learning Rate를 수정한 `Adagrad`, `RMSprop`, `AdaDelta`,

- 두 장점을 합한 `Adam`, `Nadam`

으로 발전되어 왔다.

해당 강의에서는 `Adam Optimizer`까지 살펴보도록 한다. ~~수식에 깊이 매몰되지 말자!~~

$$w^{+} = w - \eta * \frac{\partial E}{\partial w}$$

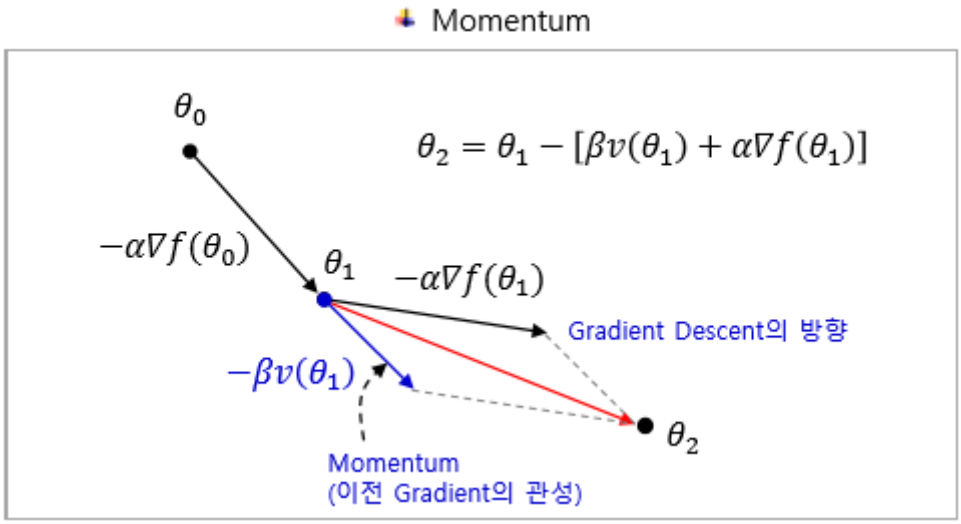
learning rate :
한번에 얼마나 학습할지

gradient :
어떤 방향으로 학습할지

출처: https://gomguard.tistory.com/187

3.2. Momentum

Gradient Descent 방향으로 바로 이동하지 않는다. 이전 Gradient의 관성 방향으로 진행할 때 가야 할 지점과 Gradient 방향으로 진행할 때 가야 할 지점을 섞어서(?) 이동한다.



momentum 방향으로 이동할 때에 얼마나 이동시킬지를 나타내는 파라미터 beta가 추가된다.

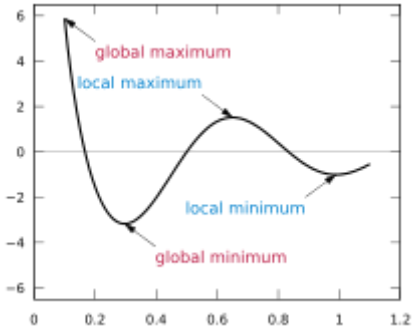
$$v_t = \beta v_{t-1} + \alpha \nabla f(\theta_{t-1})$$
$$\theta_{t+1} = \theta_t - v_t$$

$$v_t = \beta\left(\beta v_{t-2} + \alpha \nabla f(\theta_{t-2})\right) + \alpha \nabla f(\theta_{t-1})$$
$$= \alpha \nabla f(\theta_{t-1}) + \beta \alpha \nabla f(\theta_{t-2}) + \beta^2 \alpha \nabla f(\theta_{t-3}) + \cdots$$

관성 항 : 과거로 갈수록 가중치가 지수적으로 감소함.

beta가 0과 1사이이므로, 진행할 수록 과거의 가중치가 지수적으로 감소해 나간다.

local minimum 문제에 빠지는 것을 완화한다.

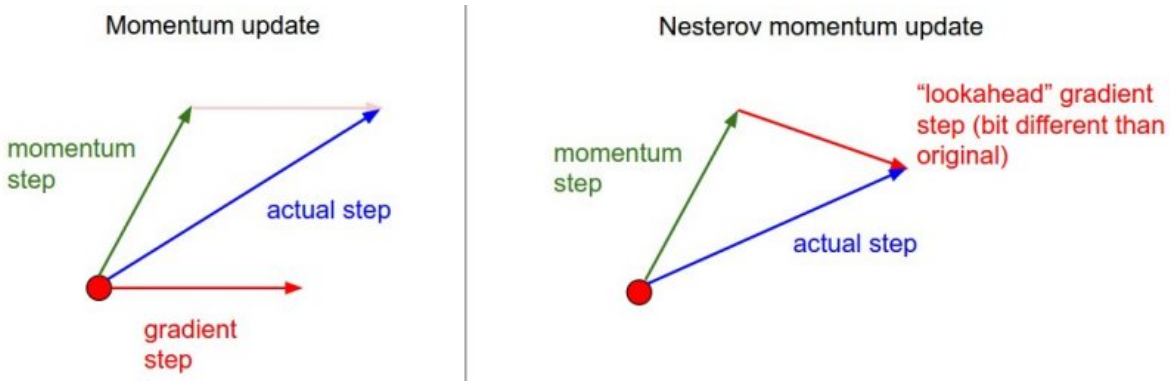


출처 : 위키피디아 local minimum

기존의 gradient 방법으로는 local mimum에 도달하면 gradient가 0이 되어 멈추지만, momentum 방법으로는 gradient 항이 0이 되더라도 앞의 관성 항이 살아있기 때문에, 이동할 여지가 남는다. local minimum 문제에 빠지는 것을 *일부* 완화할 수 있다.

또한, gradient descent보다 빠르게 목표 지점에 도달한다. 기존에 이동해 오던 방향으로 관성이 걸리고, 한편으로는 목표 지점에 가는 방향까지의 힘도 있기 때문에 *상대적으로* 빠른 이동 속도를 보이는 것이다.

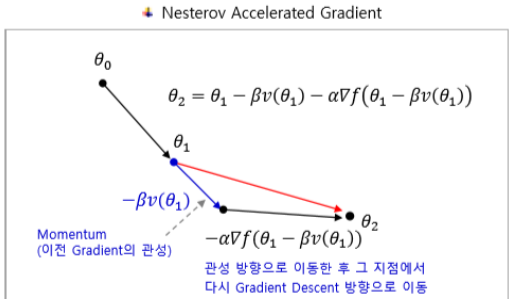
3.3. Nesterov Accelerated Gradient



출처: [http://shuuki4.github.io/deep%20learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html]

(http://shuuki4.github.io/deep learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html)

$$v_t = \beta v_{t-1} + \alpha \nabla f(\theta_{t-1} - \beta v_{t-1})$$
$$\theta_{t+1} = \theta_t - v_t$$



Momentum 방향으로 이동했다고 가정하고, 그 지점에서의 gradient descent를 취해서 이동한다. Momentum 방식이 관성 방향과 gradient descent 방향 벡터의 합으로 이동하지만, NAG의 경우 일단 관성 방향으로 이동하기 때문에, 미분해서 기울기를 계산하는 지점이 달라진다.

Momentum 방법에 비해 보다 효과적으로 이동할 수 있다. Momentum 방법의 경우, 최적점에서 멀리 떨어져 있을 때 빨리 최적점을 향해 가기는 하지만, 기존 진행 방향이 가지는 관성의 힘의 영향으로 최적점 부근에서 최적점을 지나쳐버릴 수도 있다.

이러한 단점을 해결하기 위해 Momentum 방식에서 일종의 브레이크를 추가한 것이라고 이해하자. 일단 관성 방향으로 이동한 후, 어떤 식으로 이동해야 할지 결정하는 방식이다. 적절한 시점에서 제동을 걸 수 있다. 따라서 최적점 **부근**에서 momentum보다 더 잘 작동한다.

3.4. Adagrad

학습률 alpha를 상황에 맞게 조절한다. 각각의 변수별로 이동할 step의 size를 다르게 설정하는 것이다.

구체적으로는, 업데이트가 많이 일어나는 데이터의 경우 작은 alpha를, 적게 일어나는 데이터의 경우 큰 alpha를 적용한다.

$$G_t = G_{t-1} + [\nabla f(\theta_t)]^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \varepsilon}} \nabla f(\theta_t)$$

알고 보면 상당히 직관적이다. 자주 등장했거나, 많이 변화했던 변수들의 경우 최적점에 가까이 있을 확률이 높다고 보는 것이다. 이러한 가정 하에 최적점 방향으로 많이 이동했을 경우 다음에는 그 방향으로 적게 이동하고, 적게 이동했을 경우에는 그 방향으로 많이 이동하도록 한다.

굳이 내가 학습률을 정해주지 않아도 되니 편한 알고리즘이다. 이후에 계속 바꾸지 않아도 된다.

그러나 학습을 계속 진행할 때 alpha가 너무 줄어들어 버린다는(*혹은 자연스럽게 사라져 버린다는(?)*) 단점이 있다. 위의 식에서 G에 계속 제곱한 값이 들어가기 때문에, alpha의 분모는 그만큼 빠르게 커지게(*조금 유식하게 말하면, 지수적으로 증가하게*) 된다.

결국에는 alpha가 **너무 작아져서** 거의 움직이지 않게 되어 버린다. 최적점 부근에서는 alpha가 너무 조금씩 움직일 것이다.

3.5. RMSprop

Adagrad의 문제점을 보완한다. ~~무려~~ Hinton 교수가 제안한 방법이다.

Adagrad 식에서 gradient의 제곱 값을 더해 나가며 구한 G 부분을 합이 아니라 지수 평균으로 바꿔서 대체했다.

$$G_t = \rho G_{t-1} + (1-\rho)[\nabla f(\theta_t)]^2 \rightarrow E[\{\nabla f(\theta_t)\}^2]$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \varepsilon}} \nabla f(\theta_t)$$

식을 이렇게 바꾸면, G가 무한정 커지지는 않는다. 특히 지수평균을 하기 때문에, 최근에 변화한 G_(t-1)의 변화량을 어느 정도 유지할 수 있다.

하이퍼파라미터 rho가 새로 생긴다. rho가 작을수록 직전 시점의 G에 걸리는 영향이 작아지기 때문에, **최근에 움직인 크기**를 중시한다.

3.6. Adadelta

RMSprop과 유사하게 G를 구할 때 지수평균을 활용한다. 다만, 학습률의 변화값의 제곱(~~응?~~)을 가지고 지수평균 값을 사용한다.

- [참고 자료] : 2012, Matthew D. Zeiler, "Adadelta : An adaptive learning rate method"

$$G_t = \rho G_{t-1} + (1-\rho)[\nabla f(\theta_t)]^2 \quad \rightarrow E[\{\nabla f(\theta_t)\}^2]$$

$$\Delta_t = \frac{\sqrt{s_{t-1}+\epsilon}}{\sqrt{G_t+\epsilon}} \nabla f(\theta_t)$$

$$s_t = \rho s_{t-1} + (1-\rho)\Delta_t^2 \qquad \rightarrow E[\Delta_t^2]$$

$$\theta_{t+1} = \theta_t - \Delta_t$$

Require : Decay rate ρ, constant ϵ
Initialize parameter θ_1
Initialize $G_0 = 0, s_0 = 0$
For $t = 1,2,3,\ldots$:
Compute gradient : $\nabla f(\theta_t)$
Accumulate gradient : $G_t = \rho G_{t-1} + (1-\rho)[\nabla f(\theta_t)]^2$
Compute update (Delta) : $\Delta_t = \frac{\sqrt{s_{t-1}+\epsilon}}{\sqrt{G_t+\epsilon}} \nabla f(\theta_t)$
Accumulate delta : $s_t = \rho s_{t-1} + (1-\rho){\Delta_t}^2$
Apply update : $\theta_{t+1} = \theta_t - \Delta_t$
End For

- [참고 자료]의 Algorithm 1 : Computing Adadelta update at time t (재구성)

이 식은 좀 어렵다. 일단 alpha가 과도하게 작아지는 것을 보완하는 또 다른 알고리즘이라는 것만 기억하자.

3.7. Adam : Adaptive Moment Estimation

가장 최근에 나온 것으로, 관성 및 학습률을 모두 조정하고자 한다. RMSprop과 Momentum 방식을 합쳐놓은 듯한 알고리즘이다.

$$g_t = \nabla f(\theta_{t-1})$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \leftarrow \text{Momentum}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) {g_t}^2 \quad \leftarrow \text{Adaptive } \alpha$$

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

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

Initialization bias correction terms
(초깃값에 따른 Bias를 보정함)

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\hat{v}_t} - \epsilon} \hat{m}_t$$

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0,1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t+1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\hat{m}_t \leftarrow m_t / (1-\beta_1^t)$ (Compute bias-corrected first moment estimate)
$\hat{v}_t \leftarrow v_t / (1-\beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

관성과 학습률 alpha가 초깃값 0에서 출발하면 bias가 커지므로, 이를 보정해준 것이 그 특징이다. 학습이 진행되며 반복이 더 진행되어 갈 수록, bias가 보정되는 정도가 줄어든다. 어떻게 보면, 처음에 bias를 크게 보정하겠다는 취지이다.

보정할 때의 과정이 어떻게 되는지까지는 기록하지 않는다.

> *참고*

> Tensorflow의 adam optimizer에서 alpha 값을 처음에 주면, 그건 계속 유지된다. 다만 alpha가 decay된다는 것은 위의 식에서, 동그라미 친 부분을 조정해 나가며 decay시켜나간다는 것을 의미한다.

실습 2. gradient 바꿔 가면서 적용

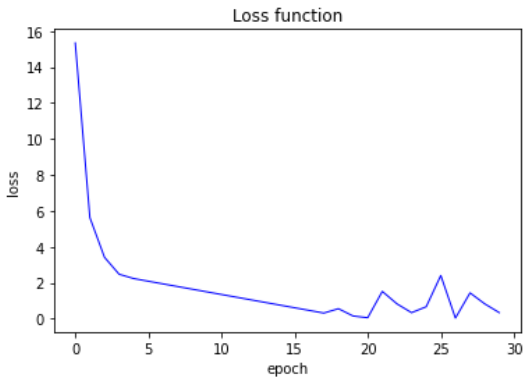
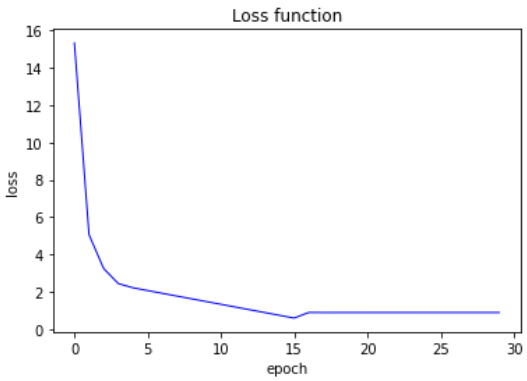
- `optimizers.SGD(momentum` : 관성 방향 계수, `nesterov` 적용하지 않음)
- `grads = tape.gradient(loss, var_list)` : 출력해 보면 w1, w2, b가 tensor 형태로 나온다.

```
[<tf.Tensor: shape=(), dtype=float32, numpy=4.311927>, <tf.Tensor: shape=(), dtype=float32, numpy=-2.3566594>, <tf.Tensor: shape=(), dtype=float32, numpy=-0.026132978>]
```

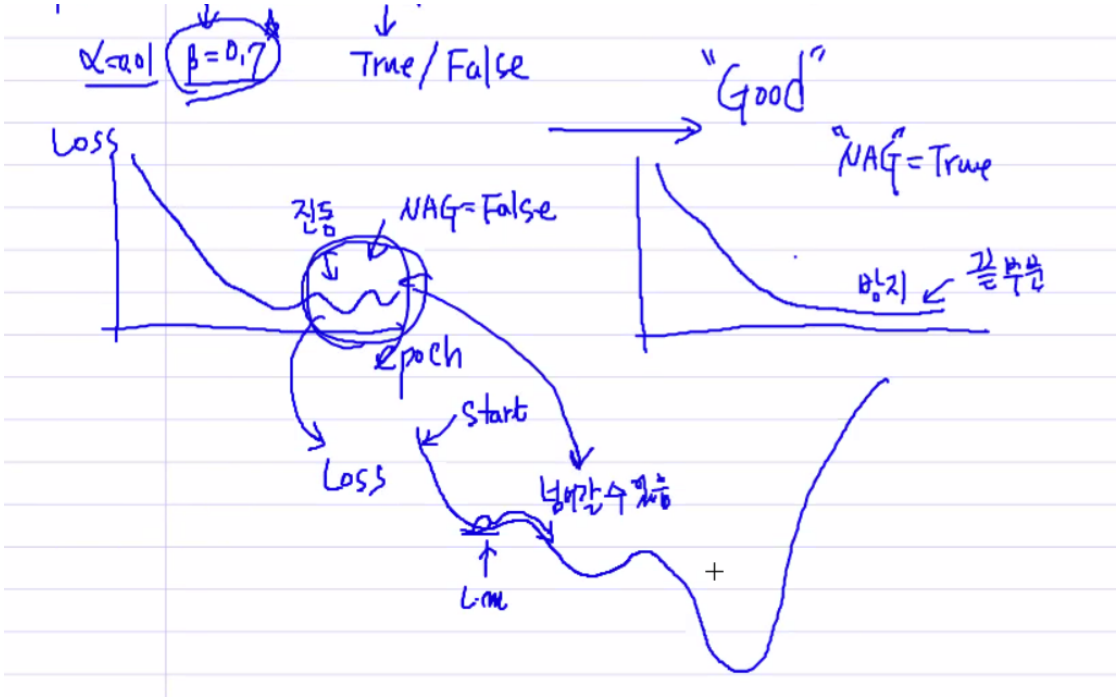
- gradient update할 때 `zip` 이용했다.

```
opt.apply_gradients(zip(grads, var_list))
```

- 하이퍼파라미터 바꿔 봤을 때, `nestrov = True`로 변경하니 마지막에 울퉁불퉁한 게 잡히는 효과가 있다. 아래 경우는 momentum 관성 조건을 크게 줬기 때문에, NAG를 True로 주면, 동일한 조건에서 끝에가 매끄럽게 잡히는 측면이 있다.

momentum = 0.7, nestrov = False	**momentum = 0.7, nestrov = True**
	

> *강사님의 가르침*
>
> 얼핏 보면 마지막이 오른쪽이 더 좋아 보인다. 그런데 꼭 그렇지만도 않다.

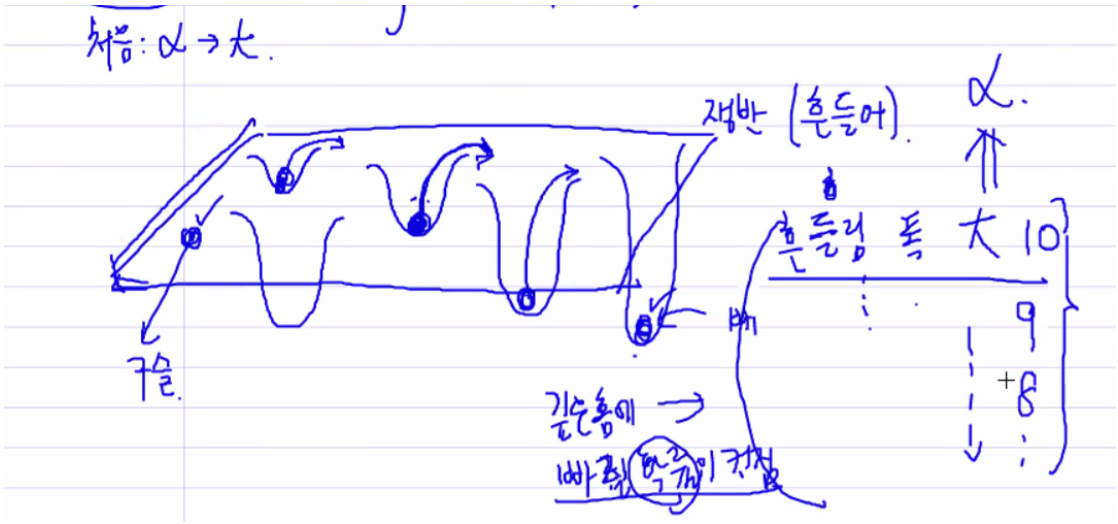


loss function이 진동하다 보면, local minimum을 훅 넘어갈 수 있다. 그런데 얕은 홈에서도 `NAG` 로 제동 장치를 넣어 버리면 넘어가지 못한다.

강사님의 가르침2

Adaptive 방식의 경우에도 local minimum을 넘어갈 수 있도록 할 수 있는 방법이 있다.

아래와 같이 구슬을 홈에 빠뜨리는 경우를 생각해 보자.



흔들림의 폭이 커질수록 구슬이 홈에 빠질 가능성이 커진다. 이게 바로 alpha를 줄여 나가는 경우다. alpha를 decay시켜서 local minumum 문제가 해결될 수도 있는 것이다.

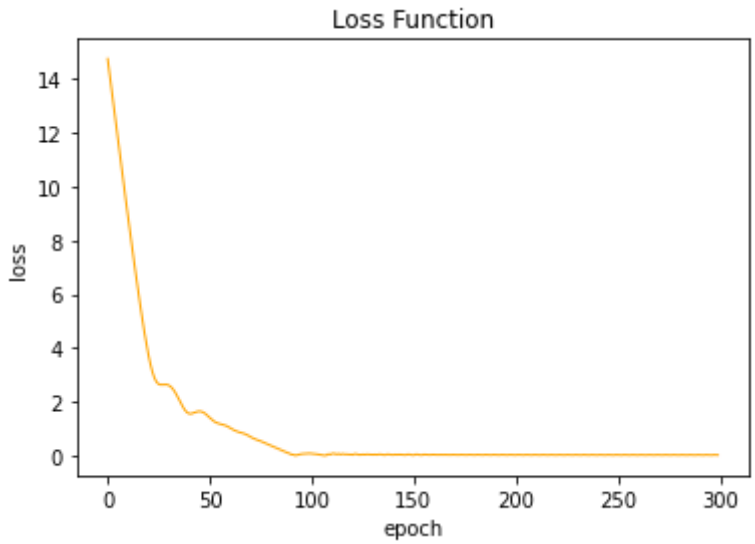
실습 3. AdamOptimizer

- 점점 더 코드가 쉬워진다!
- 강사님 코드에서 loss를 함수로 구현했다.

```
def loss():
    return tf.sqrt(tf.reduce_mean(tf.square(w1*x*x + w2*x + b - y)))

# AdamOptimizer
opt = optimizers.Adam(learning_rate=0.05)

histLoss = []
for epoch in range(300):
    opt.minimize(loss, var_list=[w1, w2, b])
    histLoss.append(loss()) # loss 함수만 호출!
```

- 결과 : 오, 완벽하게 추정 결과가!

```
최종 loss : 0.0357
```



- 원래 AdamOptimizer에 learning rate 말고도 beta 옵션이 있다.

> *참고* : Tensorflow Documentation
>
> ```
> tf.keras.optimizers.Adam(
>     learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07,
> amsgrad=False,
>     name='Adam', **kwargs
> )
> ```