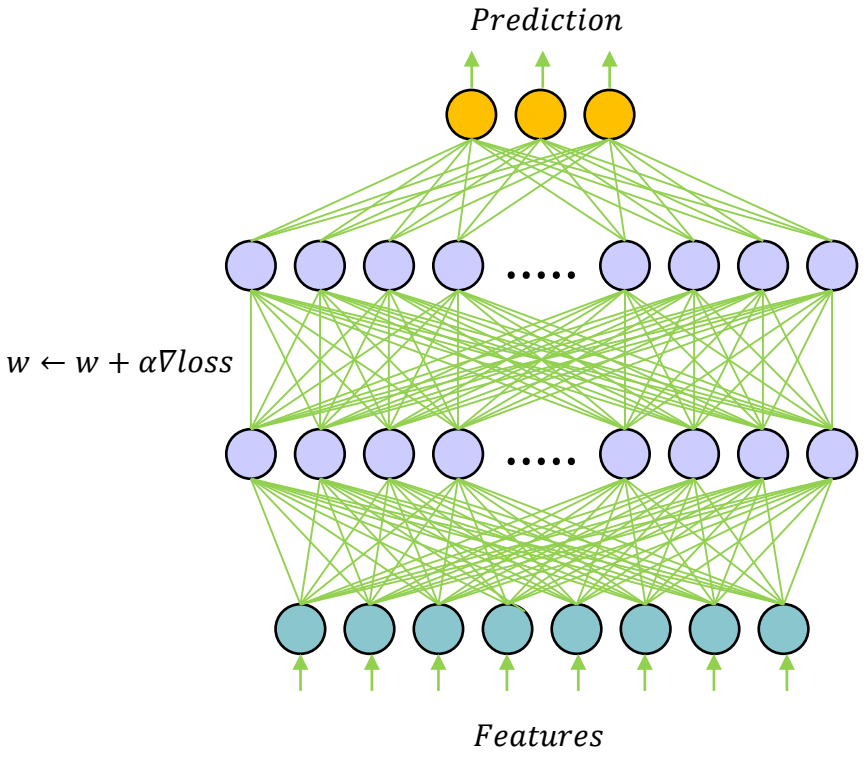


딥러닝 (Deep Learning)

(Tensorflow 2.0, Keras 예제 파일 포함)

강사 : 조성현

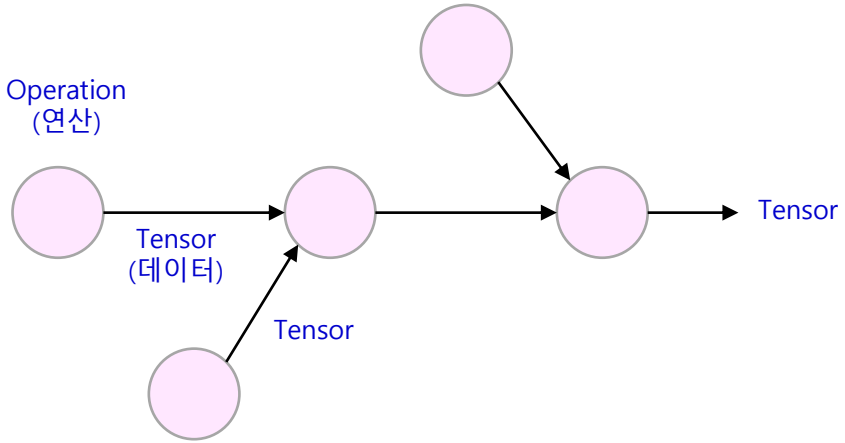
목 차

목 차

목 차

1. TensorFlow와 Keras

1-1. TensorFlow 역사 및 특징
1-2. TensorFlow 2.0 설치 (CPU, GPU)
1-3. 참고 사항 : NVIDIA GPU architecture
1-4. 참고 사항 : CPU와 GPU 메모리
1-5. 참고 사항 : GPU 동작 모습
1-6. Graph와 TensorFlow
1-7. TensorFlow 활용 연습 : 2차 방정식의 계수 추정
1-8. Stochastic Gradient Descent
1-9. Optimizers : Momentum, NAG, Adagrad, RMSProp, Adadelta, Adam
1-10. 2차 방정식의 계수 추정 연습 : Optimizer 사용
1-11. 2차 방정식의 계수 추정 연습 : minimize() 함수 사용
1-12. Keras (Sequential model, Functional API)
1-13. Stochastic, Batch, Mini-batch update
1-14. 2차 방정식의 계수 추정 연습 : Keras - Sequential model
1-15. 2차 방정식의 계수 추정 연습 : Keras - Functional API

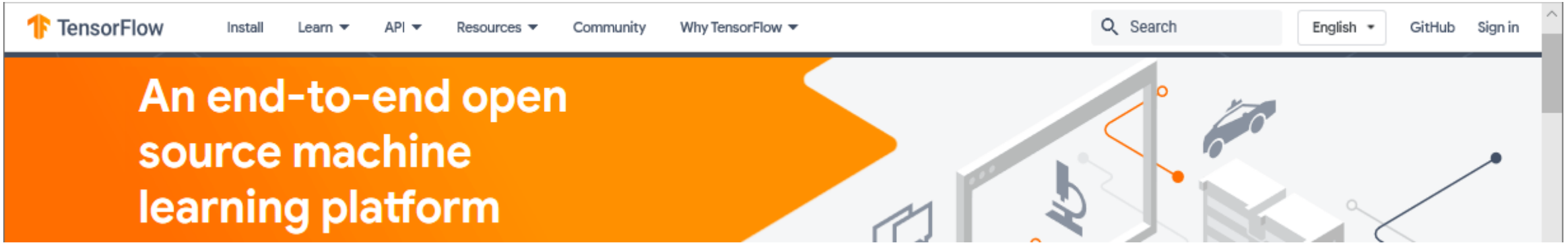


TensorFlow 역사

- Very-large-scale deep neural networks를 개발하기 위해 2011년 Google Brain 프로젝트가 시작되었고, 1세대 시스템으로 DistBelief가 개발됐다.
- 2015년에는 DistBelief의 경험을 토대로 2세대 머신러닝 시스템인 TensorFlow™가 개발됐고 오픈 소스로 공개됐다.
- Google의 많은 고객들과 구글 팀들의 연구와 제품 개발을 위해 DistBelief에서 TensorFlow로 이전했다.
- 2019년 9월에는 TensorFlow 2.0이 발표됐다. 기존의 TensorFlow (1.x)는 사용하기가 복잡하다는 지적이 있었고, 이에 따라 API들이 우후죽순으로 만들어 졌다. TensorFlow 2.0은 "쉬운 적용 (easy-to-use flatform)"이라는 요구사항을 만족시키기 위해 개발됐다.

TensorFlow 특징

- TensorFlow는 데이터 흐름 그래프 (Dataflow graph)를 사용하는 수치 연산용 오픈 소스 소프트웨어 라이브러리이다.
- 그래프의 Node는 수학적 연산 (Operation)을 나타내고, 그래프의 Edge는 노드 간에 전달되는 다차원 데이터 (Tensor)를 나타낸다.
- 유연한 아키텍처를 채택하여 단일 API를 통해 데스크톱, 서버 또는 휴대기기에 장착된 하나 이상의 CPU 또는 GPU에 연산을 배포할 수 있다.
- TensorFlow는 최초에 Google의 기계지능 연구 조직에 속한 Google Brain팀에서 근무하는 연구자 및 엔지니어에 의해 기계 학습 및 심층신경망 연구용으로 개발되었지만, 다른 분야에도 광범위하게 적용할 수 있는 범용성을 갖춘 시스템이다.
- TensorFlow 공식 웹 사이트 : https://www.tensorFlow.org



TensorFlow 2.0 설치

- Tensorflow는 아래와 같이 CPU 버전 혹은 GPU 버전을 선택적으로 설치할 수 있다. GPU 버전은 NVIDIA의 GPU가 장착된 경우에만 유효하다.

CPU 버전

- 아나콘다 (Anaconda) 파이썬 3.7버전 설치
  - 설치 사이트 : https://www.anaconda.com/
- 아나콘다 프롬프트에서 아래 명령으로 tensorflow 설치

  > conda install tensorflow 혹은

  > pip install tensorflow
- Spyder (IDE)에서 아래 명령으로 tensorflow 버전 확인

```
import tensorflow as tf
tf.__version__
'2.1.0'
```

- 딥러닝은 큰 배열의 곱셈 연산을 많이 사용하고, GPU는 많은 core를 가지고 있어 병렬 처리로 배열을 처리할 수 있으므로, tensorflow-gpu 버전을 사용하면 속도가 빨라진다 (10 ~ 20배).
- 비주얼 스튜디오 + CUDA Toolkit은 향후 CUDA C 프로그래밍을 이용한 병렬처리에도 활용할 수 있다.

GPU 버전

- 비주얼 스튜디오 community 버전 설치 (무료)
  - 설치 사이트 : https://visualstudio.microsoft.com/ko/vs/community/
- CUDA Toolkit 최신 버전 설치
  - 설치 사이트 : https://developer.nvidia.com/cuda-downloads
- cuDNN 최신 버전 설치
  - 설치 사이트 : https://developer.nvidia.com/cudnn
  - 다운로드 → 압축해제 후 bin, include, lib 폴더에 있는 파일을 각각 CUDA Toolkit이 설치된 폴더의 bin, include, lib 폴더로 복사
- 아나콘다 (Anaconda) 파이썬 3.7버전 설치
  - 설치 사이트 : https://www.anaconda.com/
- 아나콘다 프롬프트에서 아래 명령으로 tensorflow 설치

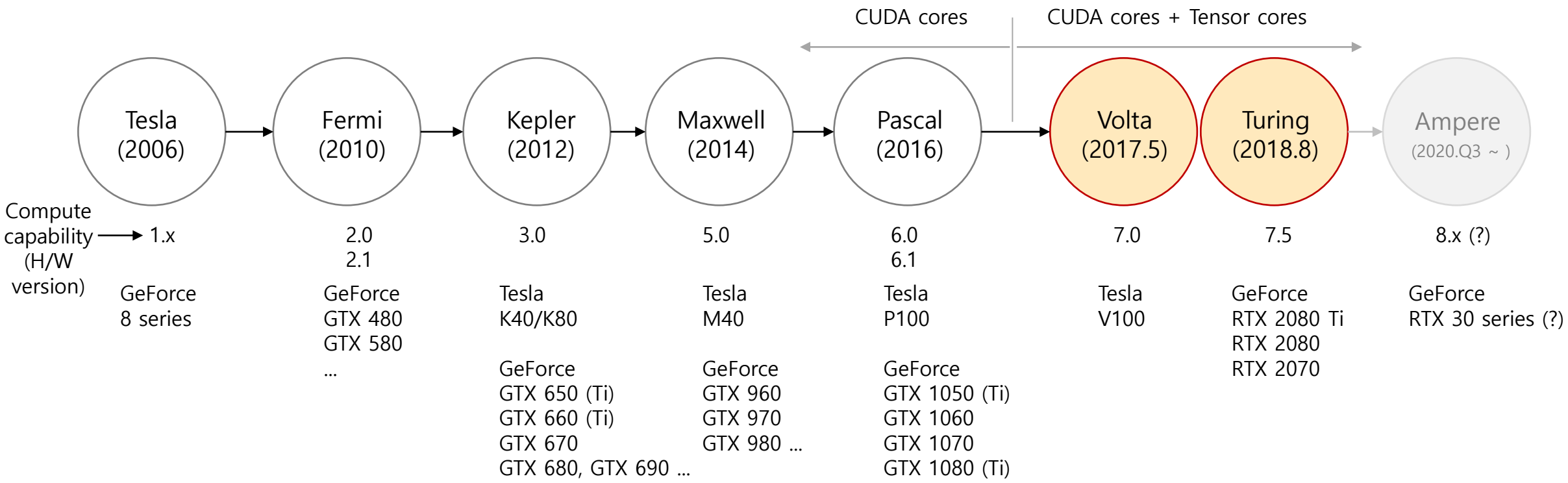
  > conda install tensorflow-gpu 혹은

  > pip install tensorflow-gpu
- Spyder (IDE)에서 아래 명령으로 tensorflow 버전 확인

```
import tensorflow as tf
tf.__version__
'2.1.0'
```

참고 사항 : NVIDIA GPU architecture

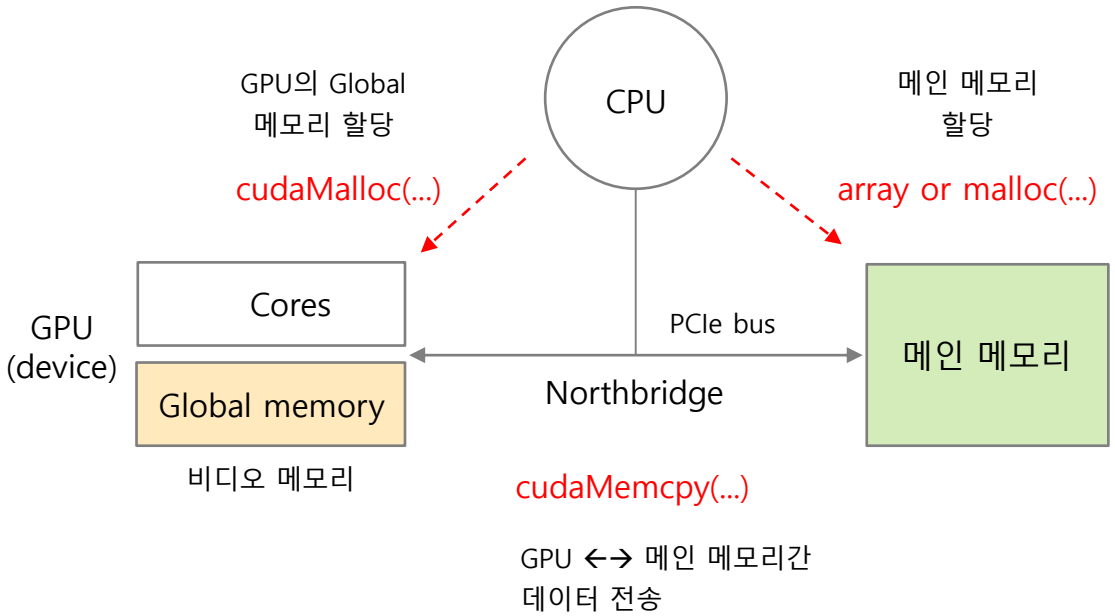
- NVIDIA의 GPU는 2006년 Tesla architecture부터 2019년 현재 Volta/Turing architecture까지 진화해 왔다.
- 그래픽 처리 이외의 일반 목적의 연산 장치 : GP-GPU (General purpose GPU)
- GPU는 그래픽 처리 기능을 넘어 병렬 처리를 사용한 고성능 컴퓨팅 기능으로 (HPC : High performance computing) 자리 잡고 있다.
- 2017년 Volta architecture부터는 인공지능을 위한 텐서 코어 (Tensor core)를 장착하고 있다. (비교 : Google의 TPU, Tensor Processing Unit)
- 참고 : 텐서 코어는 행렬 연산 (4 x 4 정도로 작은)에 특화된 코어다.



- Tesla : for HPC, High performance computing, (expensive), Quadro : for workstation (expensive), GeForce : for gaming (not expensive)

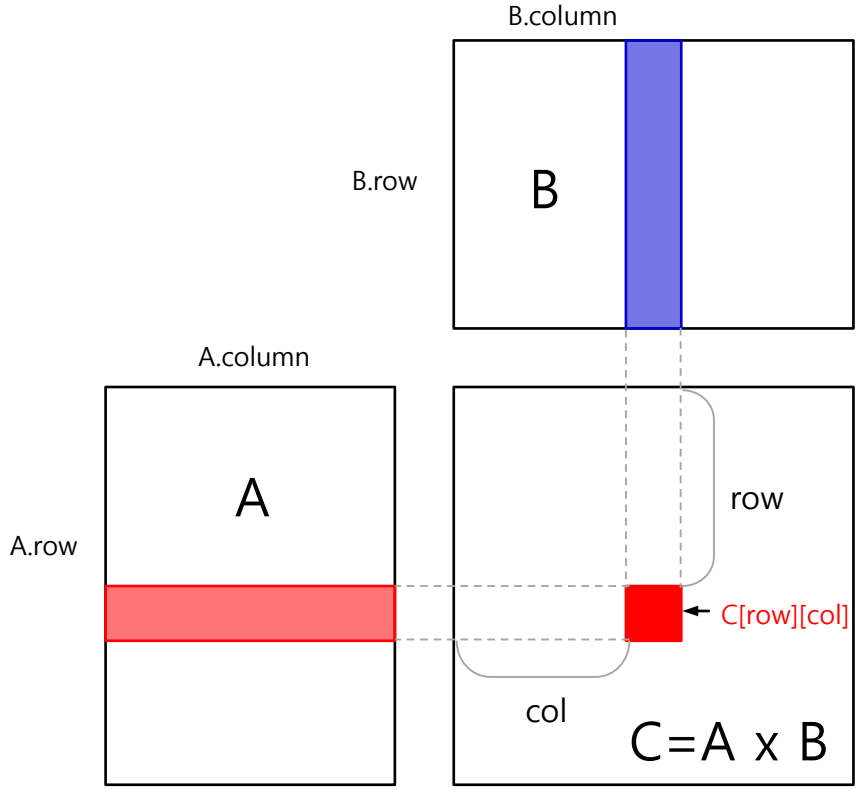
참고 사항 : CPU와 GPU 메모리

- CPU는 메인 메모리와 GPU 디바이스의 global 메모리에 데이터 영역을 할당할 수 있다. ← malloc( )과 cudaMalloc( ) 함수.
- CPU는 메인 메모리에 저장된 데이터를 GPU 메모리로 보낼 수 있고, 반대로도 보낼 수 있다. ← cudaMemcpy( ) 함수.
- CPU는 할당된 메모리 영역을 해제할 수 있다. ← free( )와 cudaFree( ) 함수
- CPU는 각 메모리 영역을 특정 값으로 리셋할 수도 있다. ← memset( )과 cudaMemSet( ) 함수.

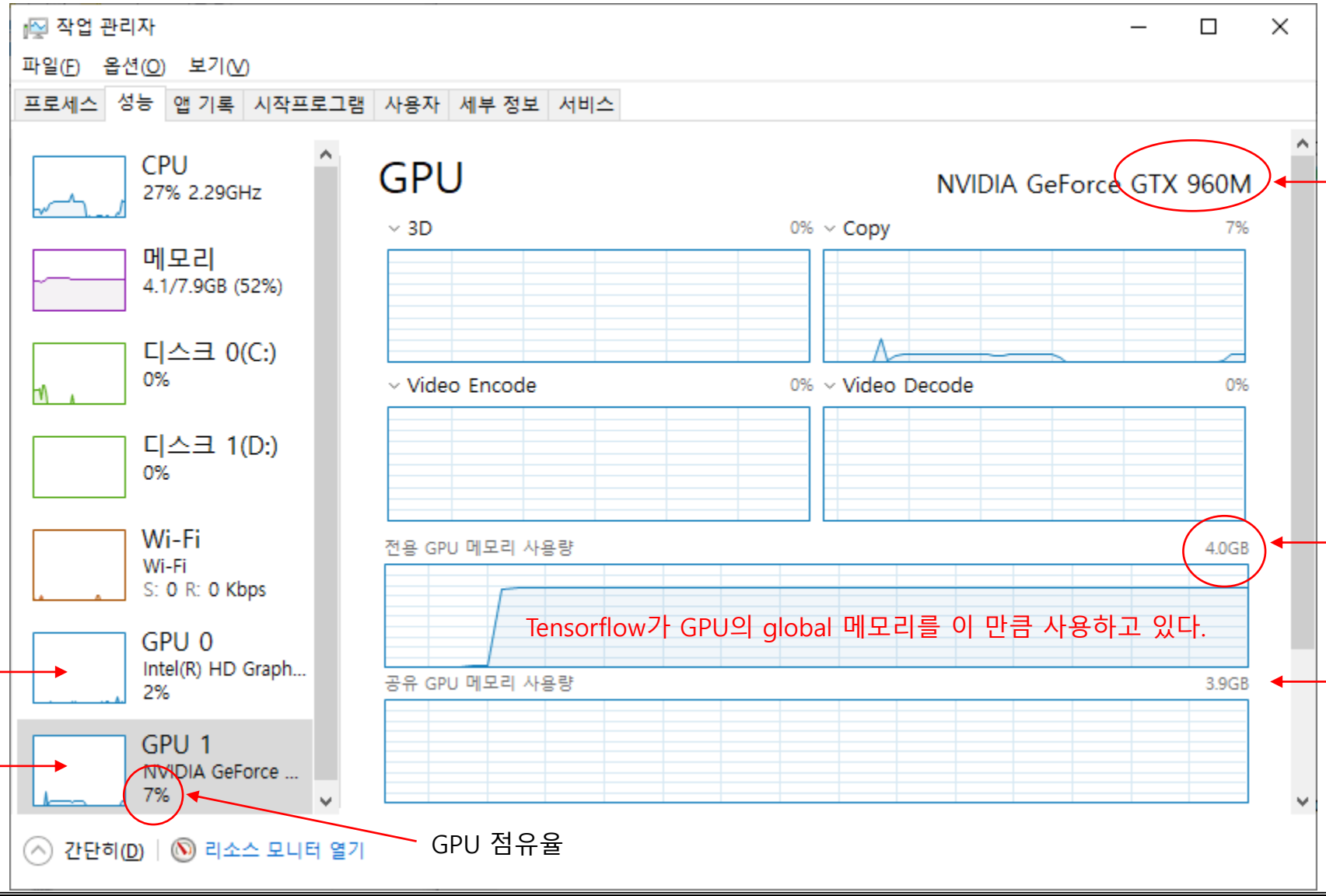


GPU의 행렬 곱셈

- 1,024 x 1,024짜리 행렬 두 개를 곱한다. 각 행렬에는 1,024 x 1,024 = 1,048,576개의 원소가 있다.
- CPU에서 순차적으로 처리하면 약 1백만 번의 for-loop가 필요하다.
- GPU에서는 백만 개의 쓰레드가 각각 1번씩만 곱셈을 수행하므로, 행렬 곱셈 속도가 대단히 빠르다.



참고 사항 : GPU에서 Tensorflow가 동작하는 모습



GPU-0은 display용으로 사용하고 있다.

GPU-1은 연산 전용으로 사용하고 있다.

Tensorflow (CUDA)에서는 이 장치를 GPU-0으로 인식한다.

GPU 점유율

GPU architecture Maxwell (2014)

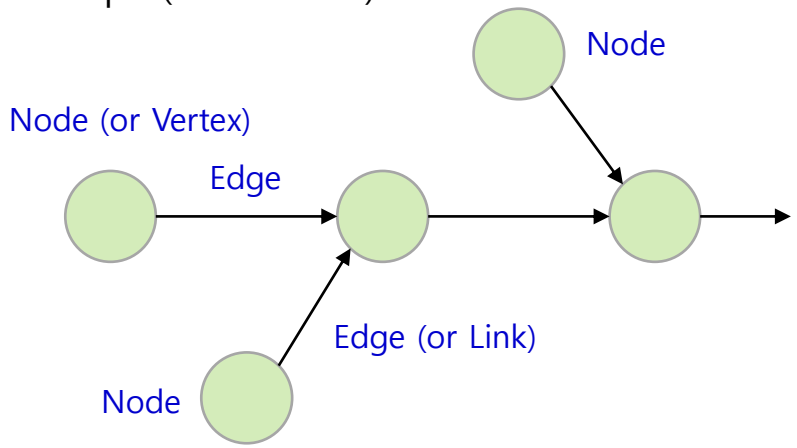
GPU global 메모리 (비디오 메모리)

GPU shared 메모리 Tensorflow는 이 영역을 사용하지 않는다.

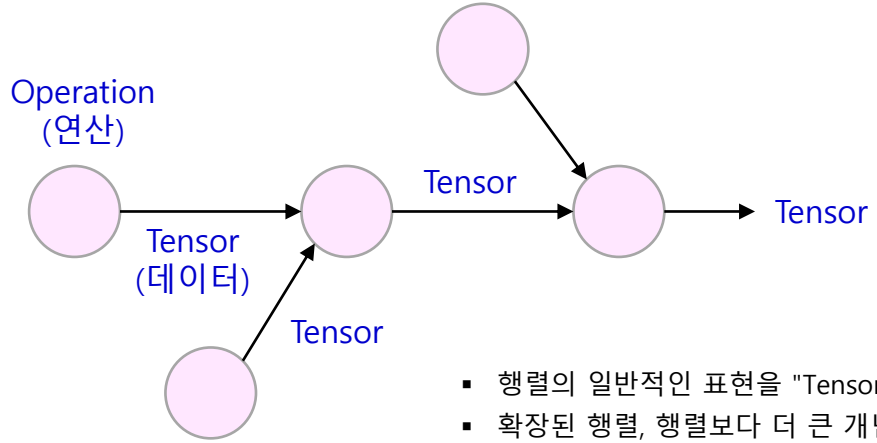
Graph와 TensorFlow

- Graph (or Network)는 Node (or Vertex)과 Edge (or Link)로 구성돼 있다. TensorFlow는 모든 연산을 그래프 구조로 표현한 후 그래프를 실행하는 방식이다.
- TensorFlow에서 Node는 연산 (Operation)을 의미하고, Edge는 데이터 (Tensor)를 의미한다. Tensor들을 그래프 상의 흐름 (flow)으로 표현하고 실행한다.
- 대부분의 기계학습 (인공 신경망 등)은 그래프로 표현되는 것들이 많으므로 TensorFlow를 이용하면 매우 유용하다. → TensorFlow 탄생 배경
- Tensorflow 1.x에서는 graph를 생성하고, 별도의 session을 생성해서 graph를 일괄적으로 실행했으나, tensorflow 2.0에서는 session의 개념은 없어지고, graph가 생성될 때 즉시 실행된다 (eager execution model)

Graph (or Network)



TensorFlow



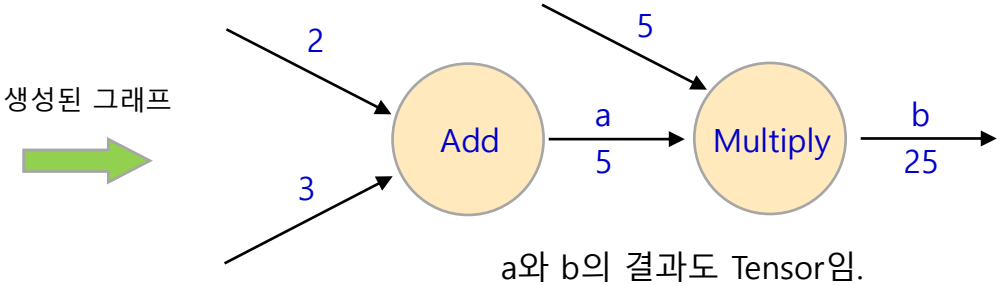
- 행렬의 일반적인 표현을 "Tensor"라 한다.
- 확장된 행렬, 행렬보다 더 큰 개념이다.

TensorFlow 예시 : 그래프 생성

```
import tensorflow as tf
a = tf.add(2, 3)
b = tf.multiply(5, a)
```

2와 3은 scalar인 tensor이고, a는 2와 3을 더하는 연산 (operation)이고, b는 5와 a를 곱하는 연산이다.

생성된 그래프



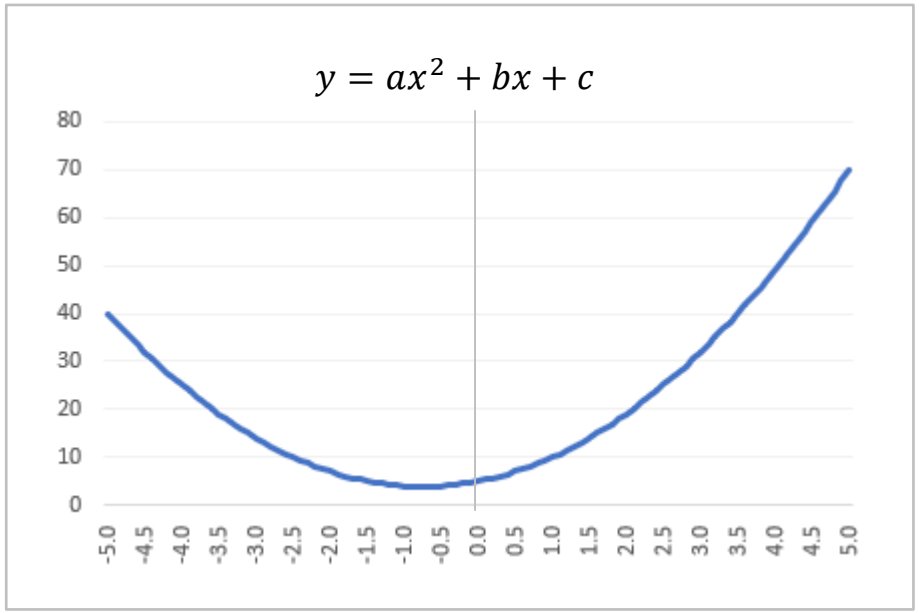
a와 b의 결과도 Tensor임.

Tensorflow 활용 연습 : 2차 방정식의 계수 추정

- 아래와 같은 데이터가 관측되었고, 이 데이터가 모형 $y = ax^2 + bx + c$를 따른다고할 때, 적절한 a, b, c는 무엇인가?
- 관측 데이터와 모형 간의 error를 측정하고, 적절한 loss 함수를 정의한다 (ex : root mean square error).
- Loss 함수가 최소가 되는 a, b, c를 추정한다. 이 문제는 수학으로도 충분히 구할 수 있으나, 여기서는 tensorflow를 이용하여 추정하기로 한다.
- 추정 순서 : 문제를 그래프로 표현한다 → loss 함수를 정의한다 → Gradient descent 방법으로 loss가 최소가 되는 파라메터 (a, b, c)를 추정한다.

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

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

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

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

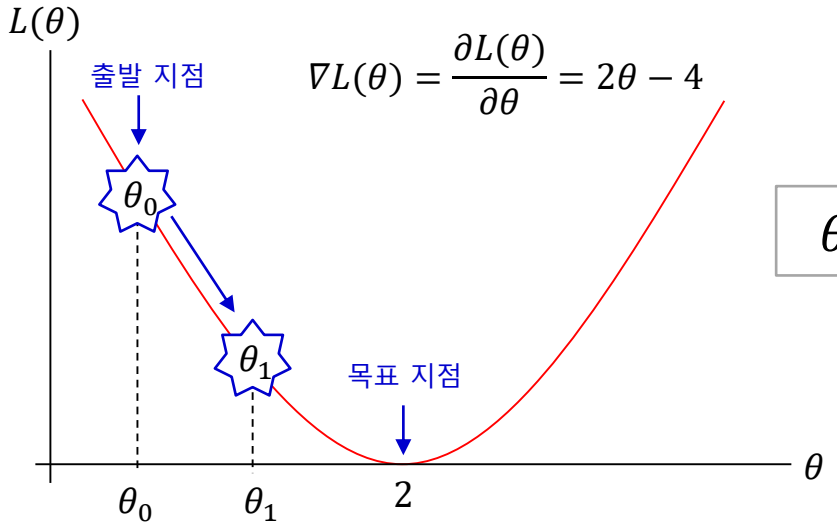
$$c \leftarrow c - \alpha \frac{\partial}{\partial c} loss$$

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

Tensorflow 활용 연습 : 2차 방정식의 계수 추정 - Stochastic Gradient Descent (SGD) : 경사 하강법

- 목적 함수인 $loss\ (L)$의 최소 지점 (목표 지점)을 찾기 위해 출발 지점 (θ_0)에서 부터 경사를 타고 내려 간다. (출발 지점 = 초깃값)
- 경사는 목적 함수의 기울기 (Gradient)로 판단할 수 있다. 경사가 급하면 많이 이동하고, 경사가 완만하면 조금씩 이동한다.
- 다음 지점은 $\theta_{t+1} = \theta_t - \alpha\nabla L(\theta_t)$식으로 계산하고, 이 과정을 반복하면 목표 지점에 도달할 수 있다. (α : Step size, Learning rate)
- 반복을 진행할 때 step size (α)를 점차 줄여나갈 수도 있다 (Decay, Annealing). ex : 지수적 감소, 10번 반복할 때마다 얼마씩 등.

$$loss : L(\theta) = (\theta - 2)^2$$



$$\nabla L(\theta) = \frac{\partial L(\theta)}{\partial \theta} = 2\theta - 4$$

Gradient Descent

$$\theta_{t+1} = \theta_t - \alpha\nabla L(\theta_t)$$

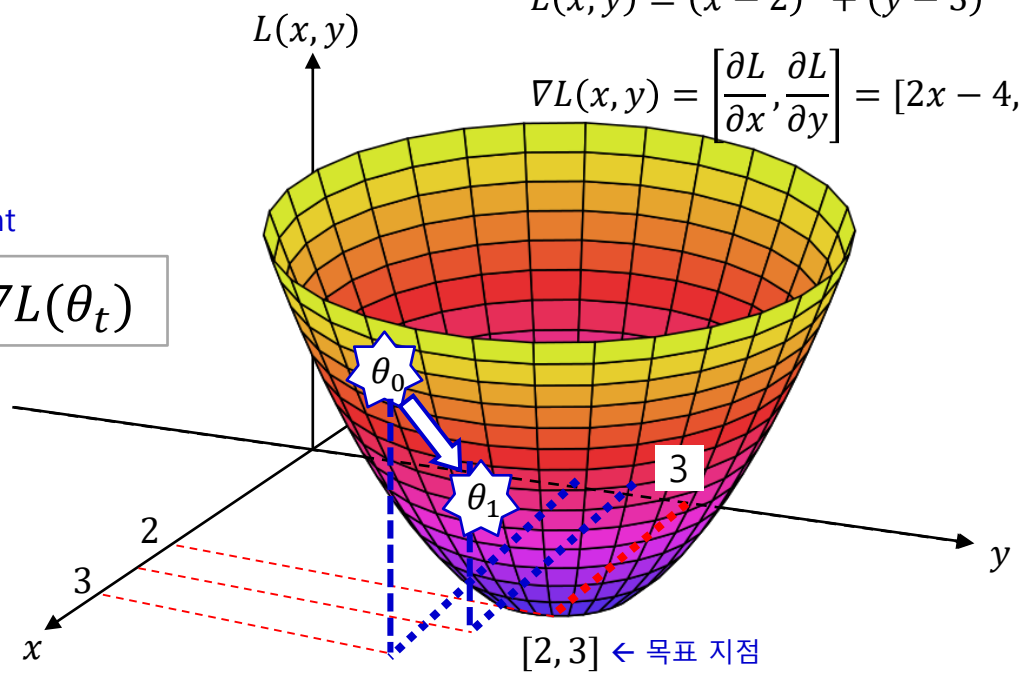
$\theta_0 = [0.5] \quad \theta_1 = [1.25]$

$\nabla L(\theta) = [-3]$ ← 출발 지점의 기울기 (Gradient)

$\theta_1 = \theta_0 - \alpha\nabla L(\theta) = [0.5] - 0.25[-3] = [1.25]$ ← 다음 지점

$$L(x, y) = (x - 2)^2 + (y - 3)^2$$

$$\nabla L(x, y) = \left[\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}\right] = [2x - 4, 2y - 6]$$



$[2, 3]$ ← 목표 지점

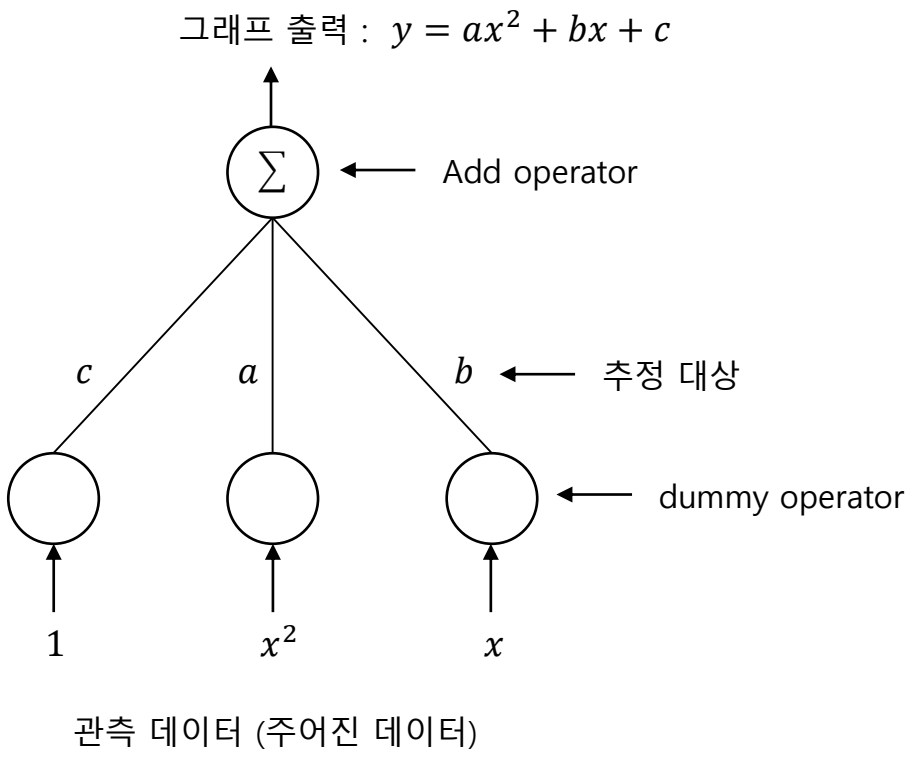
$\theta_0 = [3, 2]$ ← 출발 지점

$\nabla L(x, y) = [2, -2]$ ← 출발 지점의 기울기 (Gradient)

$\theta_1 = \theta_0 - \alpha\nabla L = [3,2] - 0.25[2, -2] = [2.5, 2.5]$ ← 다음 지점

Tensorflow 활용 연습 : 2차 방정식의 계수 추정 - 그래프 표현 및 tensorflow 코드

- 주어진 문제를 아래 왼쪽과 같이 그래프로 표현한다.
- 그래프를 아래 오른쪽의 tensorflow로 표현한다. loss 함수를 정의하고 추정할 파라메터 (a, b, c)에 대한 loss의 미분값을 계산한다.
- 계산된 미분값을 이용하여 gradient descent 방법으로 각 파라메터를 업데이트한다.
- 이 과정을 반복하면 loss 값이 점차 줄어든다. loss 값이 더 이상 줄어들지 않을 때까지 충분히 반복한다. 이 때 학습률인 lr 값은 적절한 크기의 상수이다.




```
import tensorflow as tf

a = tf.Variable(1.0)
b = tf.Variable(1.0)
c = tf.Variable(1.0)

for epoch in range(100):   # 반복
    with tf.GradientTape() as tape:
        # loss 함수 : root mean squared error
        loss = tf.sqrt(
               tf.reduce_mean(
               tf.square(a * x * x + b * x + c - y)))

        # loss에 대한 각 variable들의 미분값을 계산한다.
        da, db, dc = tape.gradient(loss, [a, b, c])

    # variable들을 업데이트한다 (Gradient descent)
    a.assign_sub(lr * da)     # a <- a - lr * da의 의미
    b.assign_sub(lr * db)     # b.assign(b - lr * b.numpy())와 동일함
    c.assign_sub(lr * dc)
```

(실습 파일 : 1-1.이차방정식(tensorflow_1).py)

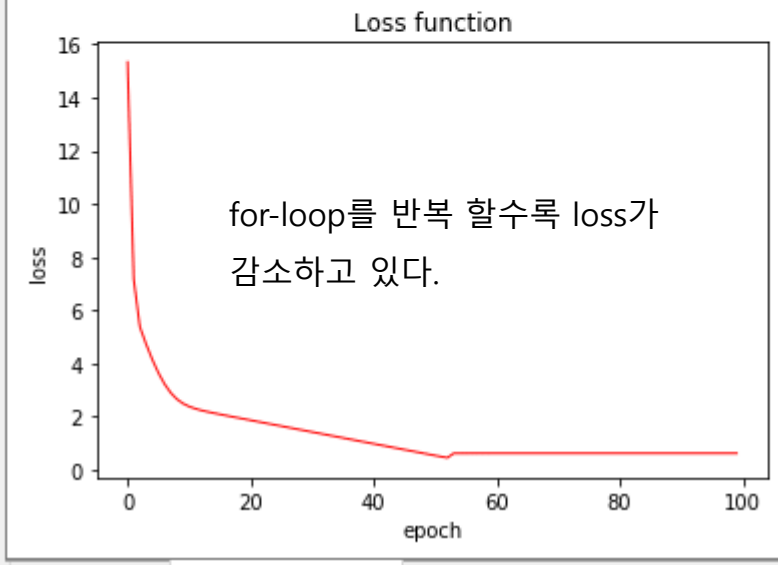
TensorFlow 동작 예시 : 이차방정식 계수 추정

- 주어진 데이터 세트 (x, y)로 이차방정식의 계수를 추정한다. Variable의 사용법, 그리고 미분값 계산에 의한 gradient descent 방법을 연습한다.

```python
# Tensorflow 버전 : 직접 미분한 방법을 사용한 예시
# x, y 데이터 세트가 있을 때, 이차 방정식 y = w1x^2 + w2x + b를 만족하는
# parameter w1, w2, b를 추정한다.
# ------------------------------------------------------------------
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# y = 2x^2 + 3x + 5 일 때 x, y 데이터 집합을 생성한다
x = np.array(np.arange(-5, 5, 0.1))
y = 2 * x * x + 3 * x + 5

# x, y 만족하는 w1, w2, b를 찾는다.
# y = w1x^2 + w2x + b -->  w1 = 2, w2 = 3, b = 5가 나와야 한다.
lr = 0.01   # learning rate

# 그래프를 생성한다.
w1 = tf.Variable(1.0)
w2 = tf.Variable(1.0)
b = tf.Variable(1.0)

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

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

    # variable들을 업데이트한다 (Gradient descent)
    w1.assign_sub(lr * dw1)     # w1 <- w1 - lr * dw1의 의미
    w2.assign_sub(lr * dw2)     # w2.assign(w2 - lr * dw2.numpy())와 동일함
```

```
epoch = 920, loss = 0.6283
epoch = 930, loss = 0.6283
epoch = 940, loss = 0.6283
epoch = 950, loss = 0.6283
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
```

추정된 파라메터 3개 : 2, 3, 5가 정답임

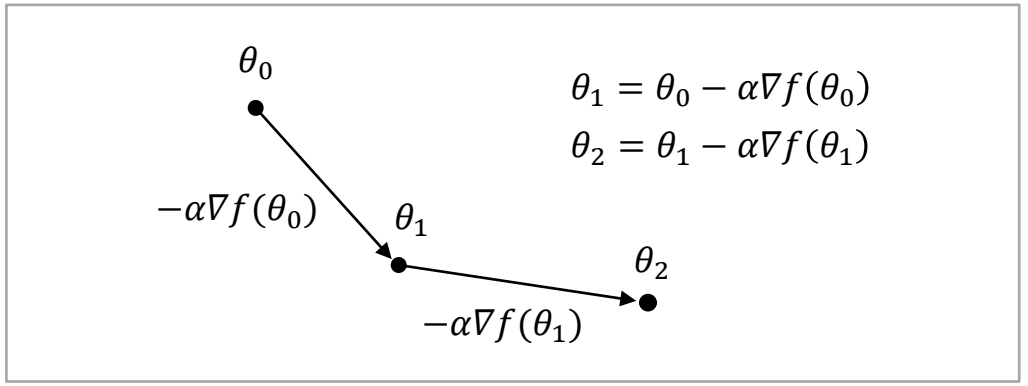


History log | IPython console

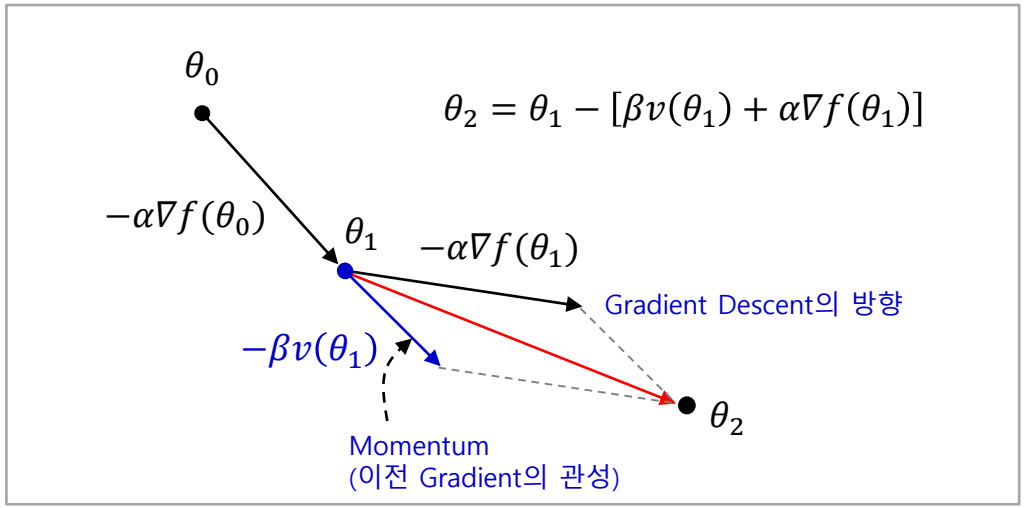
Optimizer : Momentum

- Momentum 방식은 Gradient Descent를 보완한 것으로 이전에 이동했던 방향과 동일한 방향으로 관성 (Momentum)을 추가한 방식이다.
- Gradient Descent 벡터 $[-\alpha\nabla f(\theta_1)]$ 와 Momentum 벡터 $[-\beta v(\theta_1)]$ 가 합해지는 곳으로 이동한다.
- β는 1보다 작은 값을 지정하여 $(0 < \beta < 1)$ 먼 과거의 관성은 점차 약화 시키고 최근 관성에 큰 가중치를 부여함. 최근 관성을 중시한다. (지수이동평균)
- Gradient Descent는 Gradient = 0 일 때 더 이상 이동하지 못하는 단점이 발생하는데 (Local minimize), Momentum 방식은 Gradient = 0 이어도 이전 방향의 관성 방향으로 움직일 수 있다. ← Local min. 지점에 빠지는 것을 일부 완화시킬 수 있다.
- Gradient Descent보다 빠르게 목표지점에 도달한다. (수렴 속도가 빠름)

Gradient Descent



Momentum



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

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

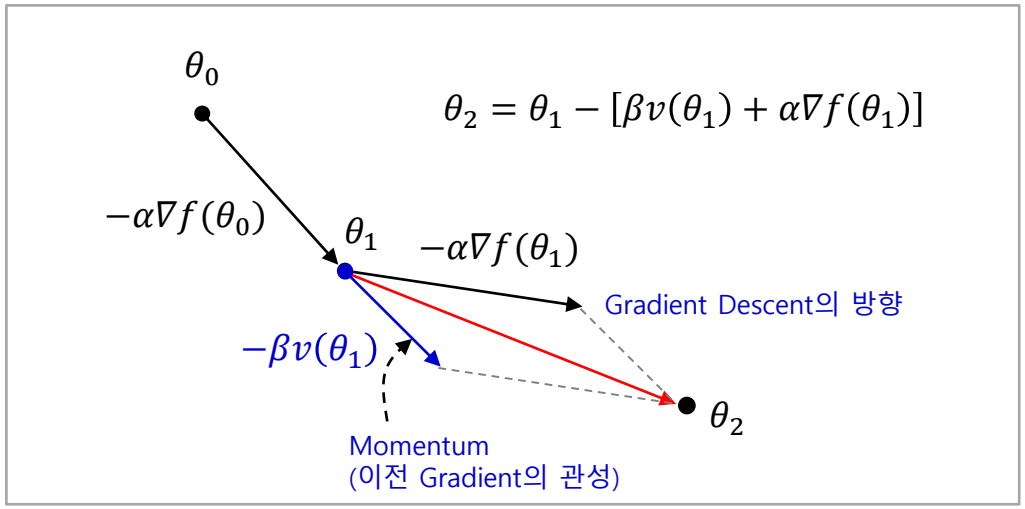
관성 항 : 과거로 갈수록 가중치가 지수적으로 감소함.

Optimizer : Nesterov Accelerated Gradient (NAG)

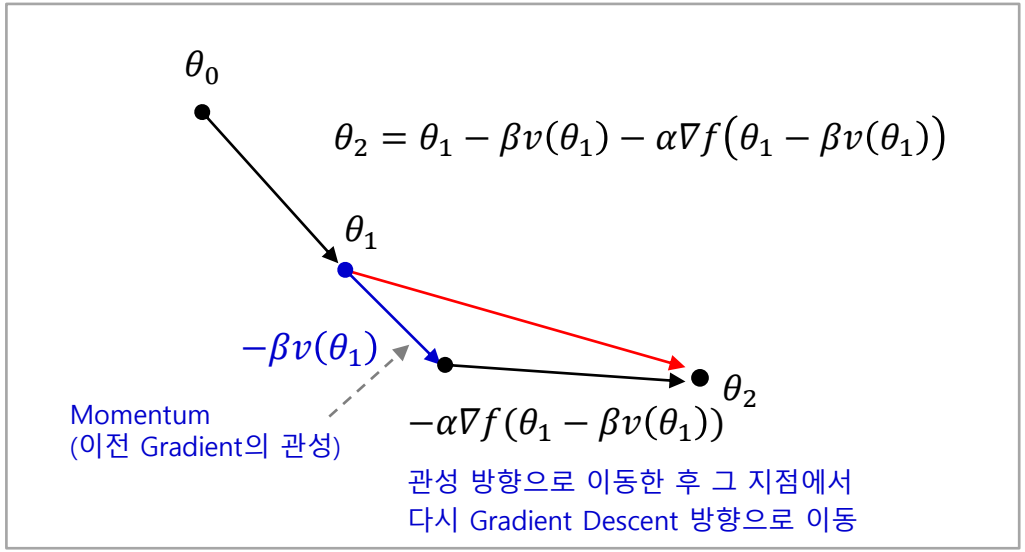
- Momentum 방식은 관성 방향과, Gradient 방향의 벡터 합인 방향으로 이동하지만, NAG 방식은 관성 방향으로 먼저 이동한 후 그 지점의 Gradient 방향으로 이동한다.
- 미분 (Gradient) 지점이 Momentum 방식과 다르다.
- Momentum 방식은 관성 때문에 최적점을 지나칠 수 있으나, NAG 방식은 관성 방향으로 이동한 후 다시 그 지점에서 어디로 갈지 판단하기 때문에 최적점 부근에서 멈출 가능성이 높아진다. (관성과 반대 방향일 수도 있음.)
- 최적점이 멀리 있을 때에는 Momentum 방식의 효과로 빨리 최적점 부근으로 이동하고, 최적점 부근에서는 NAG 방식의 효과로 빨리 멈출 수 있다.
- Momentum 방식의 제동 장치로 생각할 수도 있다.

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

Momentum



Nesterov Accelerated Gradient



Optimizer : Adaptive Gradient (Adagrad)

- SGD 방식은 모든 θ에 대해 동일한 step size (α)를 적용한다.
- α를 decay 시킬 때도 모든 θ에 대해 동일하게 적용한다.
- Adagrad 방식은 여러 θ에 대해 서로 다른 α를 적용한다.
- Update가 많이 적용된 θ에는 작은 α를 적용하고, update가 조금 적용된 θ에는 큰 α를 적용한다. 특정 방향으로 많이 이동했으면 다음 부터는 조금씩 이동하고, 특정 방향으로 조금 이동했으면 다음 부터는 많이 이동한다는 의미이다. 많이 이동한 방향은 목표 지점에 가까이 있을 가능성이 있으므로 조금씩 (세밀하게) 이동하고, 조금 이동한 방향은 목표 지점에서 멀리 있을 수 있으므로 많이 이동하겠다는 취지이다.
- 반복이 진행될수록 α가 자연히 decay 된다는 장점이 있으나 반복을 진행할수록 G_t가 지수적으로 증가하기 때문에 α가 과도하게 작아지는 단점이 있다.
- 제곱과 루트 연산은 행렬 연산이 아닌 element-wise 연산이다 (element 마다 제곱을 하거나 루트를 계산함)
- ϵ은 분모가 0이 되는 것을 방지하기 위해 작은 값을 지정한다.
- Ex : $G_0 = 0, \epsilon = 10^{-6}$

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

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

Root Mean Square Propagation (RMSprop)

- Adagrad에서 step size (α)가 과도하게 작아지는 것을 보완한 것이다.
- G_t를 계산할 때 최근에 움직인 크기와 과거에 움직여 왔던 크기에 가중 평균을 적용한다. (지수이동평균)
- ρ를 작게 적용할수록 최근에 움직인 크기를 중시하고, ρ를 크게 적용할수록 먼 과거에 움직였던 크기를 중시한다.

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

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

Optimizer : Adaptive Delta (Adadelta)

- 2012년 Zeiler에 의해 제안된 것으로, RMSprop과 유사하게 Adagrad의 단점인 step size (α)가 과도하게 작아지는 것을 보완한 알고리즘이다.
- Step size (α)를 인위적으로 지정하지 않고 자동 조절되도록 한 것과 파라메터 (θ) 업데이트 식의 단위 (unit)를 맞춘 것이 특징이다.
- [참고 자료] : 2012, Matthew D. Zeiler, "Adadelta : An adaptive learning rate method"

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

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

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

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

$Require: Decay\ rate\ \rho, constant\ \epsilon$

$Initialize\ parameter\ \theta_1$

$Initialize\ G_0 = 0, s_0 = 0$

$For\ t = 1,2,3, \ldots:$

$Compute\ gradient: \nabla f(\theta_t)$

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

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

$Accumulate\ delta: s_t = \rho s_{t-1} + (1-\rho)\Delta_t{}^2$

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

$End\ For$

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

Optimizer : Adaptive Moment Estimation (Adam)

- 2014년 Kingma와 Ba에 의해 제안된 것으로 RMSprop (Adaptive α) 과 Momentum 방식을 결합한 알고리즘이다.
- Moment (m_t)와 Adaptive α (v_t)이 초깃값 0 에서 출발할 때 초반에는 (t 가 작을 때) Bias가 크므로 이를 보정해 준 것이 특징이다.
- 반복 초반에는 (t 가 작을 때) 보정이 많이 되지만, 반복이 진행될수록 (t 가 커질수록) 보정의 정도를 줄여 나간다. (초기 bias를 크게 보정하겠다는 취지임).
- [참고 자료] : 2014, Diederik P. Kingma and Jimmy Lei Ba, "Adam : A method for stochastic optimization"

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

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

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

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

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

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)

(실습 파일 : 1-2.이차방정식(tensorflow_2).py)

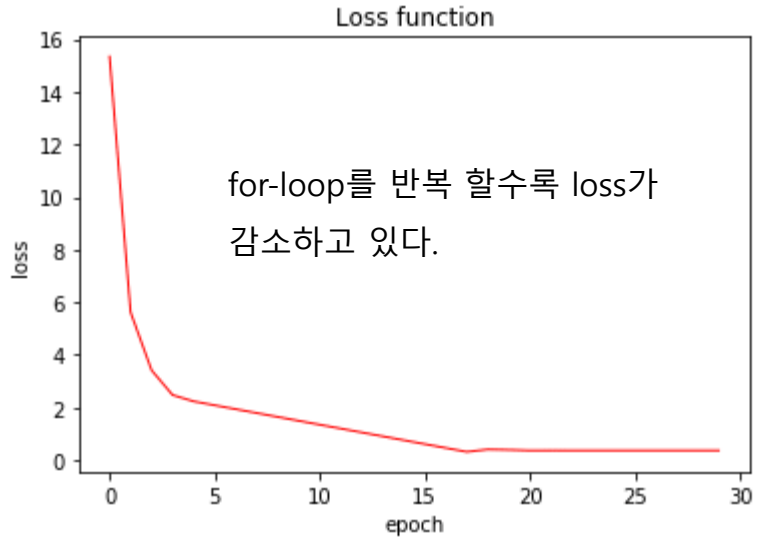
TensorFlow 동작 예시 : 이차방정식 계수 추정 - SGD + Momentum 방법 적용

- Gradient descent에 더 개선된 optimizer 기능을 추가로 적용한다. Momentum = 0.7, nestrov = False

```python
# Tensorflow 버전 : Optimizers 기능을 사용한 예시 (1) - SGD + Momentum
# x, y 데이터 세트가 있을 때, 이차 방정식 y = w1x^2 + w2x + b를 만족하는
# parameter w1, w2, b를 추정한다.
# ------------------------------------------------------------------
import tensorflow as tf
from tensorflow.keras import optimizers
import numpy as np
import matplotlib.pyplot as plt

# y = 2x^2 + 3x + 5 일 때 x, y 집합을 생성한다
x = np.array(np.arange(-5, 5, 0.1))
y = 2 * x * x + 3 * x + 5

# 그래프를 생성한다.
w1 = tf.Variable(1.0)
w2 = tf.Variable(1.0)
b = tf.Variable(1.0)
var_list = [w1, w2, b]  # variable list

# SGD optimizers 기능을 사용하고, Momentum 방법을 사용한다.
opt = optimizers.SGD(learning_rate = 0.01, momentum = 0.7, nesterov = False)

histLoss = []
for epoch in range(300):
    with tf.GradientTape() as tape:
        # root mean squared error
        loss = tf.sqrt(tf.reduce_mean(tf.square(w1 * x * x + w2 * x + b - y)))

    # loss에 대한 각 variable들의 미분값을 계산한다.
    grads = tape.gradient(loss, var_list)

    # variable들을 업데이트한다 (Gradient descent)
    opt.apply_gradients(zip(grads, var_list))
```

```
epoch = 240, loss = 0.3696
epoch = 250, loss = 0.3696
epoch = 260, loss = 0.3696
epoch = 270, loss = 0.3696
epoch = 280, loss = 0.3696
epoch = 290, loss = 0.3696

추정 결과 :
w1 = 2.03
w2 = 3.00
b = 5.00
final loss = 0.3696
```

추정된 파라메터 3개 : 2, 3, 5가 정답임



for-loop를 반복 할수록 loss가 감소하고 있다.

```
In [4]:
```

History log | IPython console

(실습 파일 : 1-3.이차방정식(tensorflow_3).py)

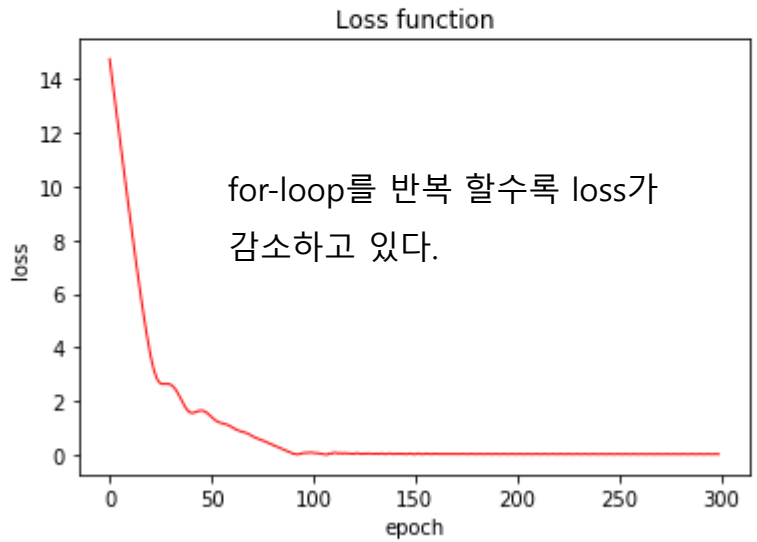
TensorFlow 동작 예시 : 이차방정식 계수 추정 - Adam optimizer

- Adam optimizer를 적용하고, minimize() 함수를 이용한다. minimize() 함수를 이용하면 직접 미분할 필요가 없어 편리하다.

```
# Tensorflow 버전 : Optimizers 기능을 사용한 예시 (2)
# - Adam optimizer를 사용하고, minimize() 함수를 이용한다.
#
# x, y 데이터 세트가 있을 때, 이차 방정식 y = w1x^2 + w2x + b를 만족하는
# parameter w1, w2, b를 추정한다.
# ----------------------------------------------------------------------
import tensorflow as tf
from tensorflow.keras import optimizers
import numpy as np
import matplotlib.pyplot as plt

# y = 2x^2 + 3x + 5 일 때 x, y 집합을 생성한다
x = np.array(np.arange(-5, 5, 0.1))
y = 2 * x * x + 3 * x + 5

# 그래프를 생성한다.
w1 = tf.Variable(1.0)
w2 = tf.Variable(1.0)
b = tf.Variable(1.0)

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

# Adam optimizers 기능을 사용한다.
opt = optimizers.Adam(learning_rate = 0.05)

histLoss = []
for epoch in range(300):
    opt.minimize(loss, var_list = [w1, w2, b])

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

```
epoch = 230, loss = 0.0411
epoch = 240, loss = 0.0334
epoch = 250, loss = 0.0402
epoch = 260, loss = 0.0328
epoch = 270, loss = 0.0394
epoch = 280, loss = 0.0323
epoch = 290, loss = 0.0388

추정 결과 :
w1 = 2.00
w2 = 3.00
b = 5.00
```

추정된 파라메터 3개 : 2, 3, 5가 정답임



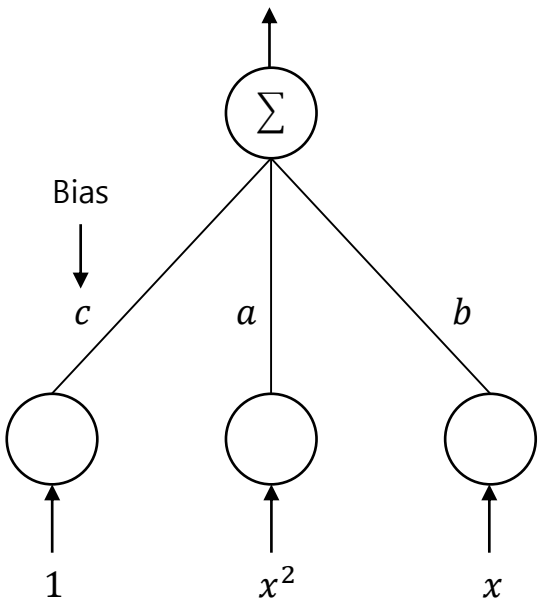

```
In [10]:
```

History log | IPython console

Keras

- Keras는 tensorflow를 쉽게 사용하기 위해 제작된 도구다. Keras 코드를 작성하면 내부적으로 tensorflow 기능을 동작 시킨다. ← 사용자 편리성
- Keras는 sequential model 방식과 functional API 방식이 있다. Functional API 방식이 더 유연성이 좋고 더욱 다양한 기능을 수행할 수 있다.
- Tensorflow만 설치하면 tensorflow에 내장된 Keras 기능을 사용할 수 있다. Keras 패키지를 별도로 설치할 수도 있다.
- Tensorflow 2.0은 사용자 친화성을 목표로 하므로 Keras 사용을 권장하고 있다. 본 교육의 이후 내용은 모두 Keras를 사용한다.

그래프 출력 : $y = ax^2 + bx + c$



관측 데이터 (주어진 데이터)

Sequential model

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential

# 그래프를 생성한다.
model = Sequential()
model.add(Dense(1, input_dim = 2))      # Bias는 자동으로 설정됨
model.compile(loss='mse', optimizer=optimizers.Adam(lr=0.05))

# 학습한다.
model.fit(dataX, y, epochs = 300)
```

Functional API

```
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# 그래프를 생성한다.
xInput = Input(batch_shape=(None, 2))   # Bias는 자동으로 설정됨
yOutput = Dense(1)(xInput)
model = Model(xInput, yOutput)
model.compile(loss='mse', optimizer=optimizers.Adam(lr=0.05))

# 학습한다.
model.fit(dataX, y, epochs = 300)
```

Keras : Stochastic, batch, mini-batch update

1) Stochastic GD update :
- 데이터마다 error를 계산하고, 그때 그때 파라메터 (a, b, c)를 업데이트하는 방식.
- error의 편차가 크고, 속도가 늦다.
- 목표 지점으로 수렴하지 않을 수도 있다.

$$error(e) = y - (ax^2 + bx + c)$$

$$loss\ (L) = \sqrt{\frac{1}{100}\sum_{i=1}^{100} e_i^2}$$

2) Batch update :
- 데이터 전체에 대한 error를 한꺼번에 계산하고 파라메터 (a, b, c)를 업데이트하는 방식.
- error의 편차가 작고 안정적이다.
- 데이터가 많은 경우 error를 계산하는 시간이 오래 걸릴 수 있고 메모리도 많이 필요하다.

$L_1 = e_1^2$

$L_2 = e_2^2$

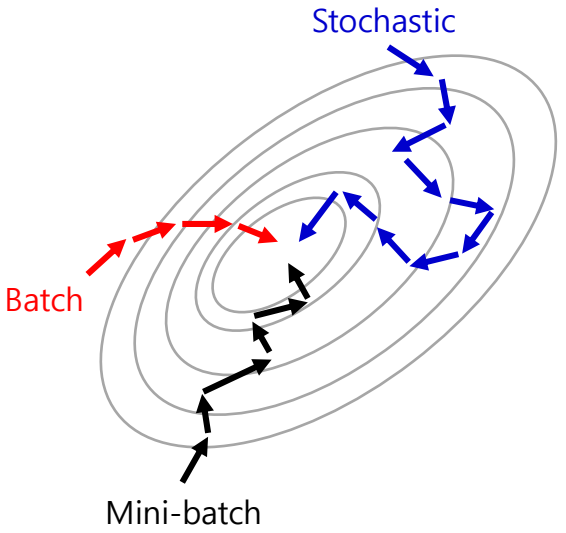
관측 데이터

n	x	y
1	-5.00	40.00
2	-4.90	38.32
3	-4.80	36.68
4	-4.70	35.08
5	-4.60	33.52
6	-4.50	32.00
7	-4.40	30.52
8	-4.30	29.08
9	-4.20	27.68
10	-4.10	26.32
11	-4.00	25.00
12	-3.90	23.72
13	-3.80	22.48
14	-3.70	21.28
15	-3.60	20.12
16	-3.50	19.00
17	-3.40	17.92
18	-3.30	16.88
19	-3.20	15.88
20	-3.10	14.92
⋮	⋮	⋮
90	3.90	47.12
91	4.00	49.00
92	4.10	50.92
93	4.20	52.88
94	4.30	54.88
95	4.40	56.92
96	4.50	59.00
97	4.60	61.12
98	4.70	63.28
99	4.80	65.48
100	4.90	67.72

$$L_1 = \sqrt{\frac{1}{10}\sum_{i=1}^{10} e_i^2}$$

$$L_2 = \sqrt{\frac{1}{10}\sum_{i=1}^{10} e_i^2}$$

$$L_{10} = \sqrt{\frac{1}{10}\sum_{i=1}^{10} e_i^2}$$



3) Mini-batch update :
- 일부 데이터로 error를 계산하고 그 때마다 파라메터 (a, b, c)를 업데이트하는 방식.
- Stochastic과 batch update의 중간 특성을 보인다.
- 가장 많이 사용된다.

(실습 파일 : 1-5.이차방정식(keras_1).py)

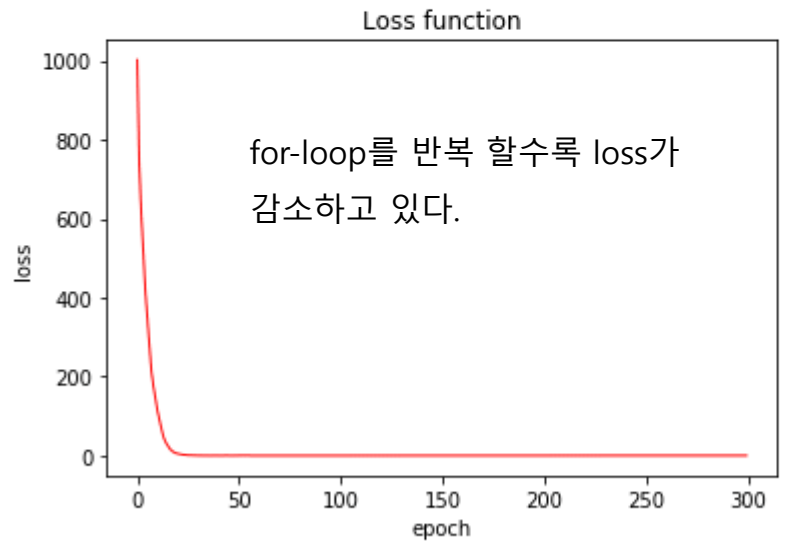
TensorFlow 동작 예시 : 이차방정식 계수 추정 - Keras (Sequential model)

- Sequential model 방식의 Keras를 사용한다. RMSprop optimizer를 적용한다.

```python
# Keras (Sequential model)를 사용한 버전
# x, y 데이터 세트가 있을 때, 이차 방정식 y = w1x^2 + w2x + b를 만족하는
# parameter w1, w2, b를 추정한다.
# ------------------------------------------------------------------
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras import optimizers
import numpy as np
import matplotlib.pyplot as plt

# y = 2x^2 + 3x + 5 일 때 x, y 집합을 생성한다
x = np.array(np.arange(-5, 5, 0.1))
y = 2 * x * x + 3 * x + 5
dataX = np.stack([x*x, x]).T

# 그래프를 생성한다.
model = Sequential()
model.add(Dense(1, input_dim = 2))
model.compile(loss='mse', optimizer=optimizers.RMSprop(lr=0.05))

# 학습한다.
h = model.fit(dataX, y, batch_size = 10, epochs = 300)

# 학습 결과를 확인한다.
parameters = model.layers[0].get_weights()
print("\n추정 결과 :")
print("w1 = %.2f" % parameters[0][0][0])
print("w2 = %.2f" % parameters[0][1][0])
print("b = %.2f" % parameters[1][0])

plt.plot(h.history['loss'], color='red', linewidth=1)
plt.title("Loss function")
plt.xlabel("epoch")
plt.ylabel("loss")
```

```
100/100 [==============================] - 0s 160us/sample - loss: 0.1681
Epoch 298/300
100/100 [==============================] - 0s 160us/sample - loss: 0.0195
Epoch 299/300
100/100 [==============================] - 0s 120us/sample - loss: 0.1216
Epoch 300/300
100/100 [==============================] - 0s 120us/sample - loss: 0.1952

추정 결과 :
w1 = 2.03
w2 = 2.99
b = 5.01
```

← 추정된 파라메터 3개




```
In [16]:
```

History log | IPython console

(실습 파일 : 1-6.이차방정식(keras_2).py)

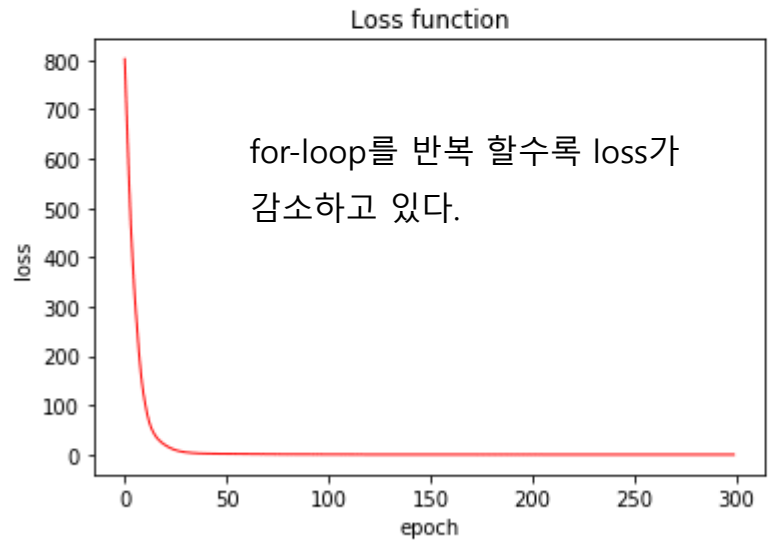
TensorFlow 동작 예시 : 이차방정식 계수 추정 - Keras (Functional API)

- Functional API 방식의 Keras를 사용한다. Adam optimizer를 적용한다.

```python
# Keras (Functional API)를 사용한 버전
# x, y 데이터 세트가 있을 때, 이차 방정식 y = w1x^2 + w2x + b를 만족하는
# parameter w1, w2, b를 추정한다.
# ------------------------------------------------------------------
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
import numpy as np
import matplotlib.pyplot as plt

# y = 2x^2 + 3x + 5 일 때 x, y 집합을 생성한다
x = np.array(np.arange(-5, 5, 0.1))
y = 2 * x * x + 3 * x + 5
dataX = np.stack([x*x, x]).T

# 그래프를 생성한다.
xInput = Input(batch_shape=(None, dataX.shape[1]))
yOutput = Dense(1)(xInput)
model = Model(xInput, yOutput)
model.compile(loss='mse', optimizer=optimizers.Adam(lr=0.05))

# 학습한다.
h = model.fit(dataX, y, batch_size = 10, epochs = 300)

# 학습 결과를 확인한다.
parameters = model.layers[1].get_weights()
print("\n추정 결과 :")
print("w1 = %.2f" % parameters[0][0][0])
print("w2 = %.2f" % parameters[0][1][0])
print("b = %.2f" % parameters[1][0])

plt.plot(h.history['loss'], color='red', linewidth=1)
plt.title("Loss function")
plt.xlabel("epoch")
```

```
100/100 [==============================] - 0s 156us/sample - loss: 5.4265e-07
Epoch 298/300
100/100 [==============================] - 0s 156us/sample - loss: 4.9373e-07
Epoch 299/300
100/100 [==============================] - 0s 0s/sample - loss: 4.4867e-07
Epoch 300/300
100/100 [==============================] - 0s 156us/sample - loss: 4.0533e-07

추정 결과 :
w1 = 2.00
w2 = 3.00
b = 5.00
```

추정된 파라메터 3개



for-loop를 반복 할수록 loss가 감소하고 있다.

```
In [17]:
```

History log | IPython console