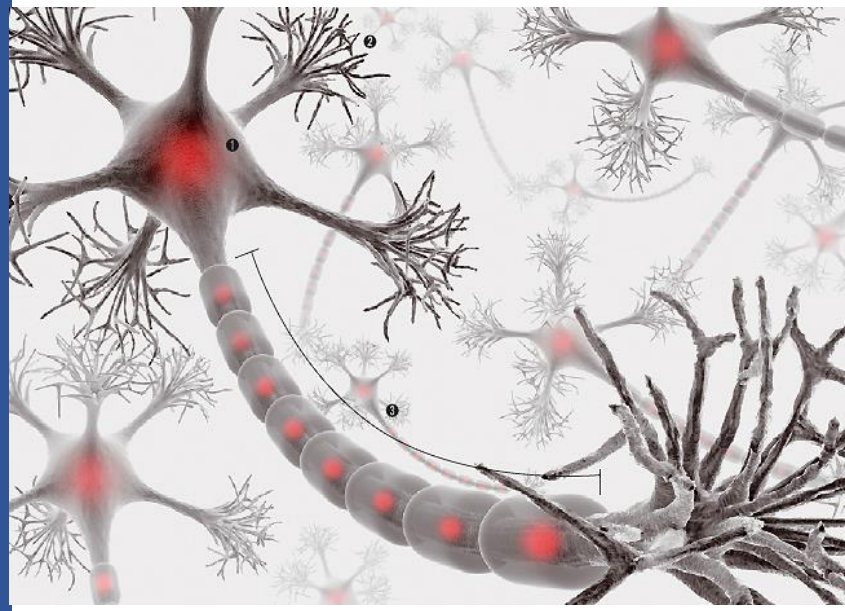
Tensorflow 2.0

학습 목표

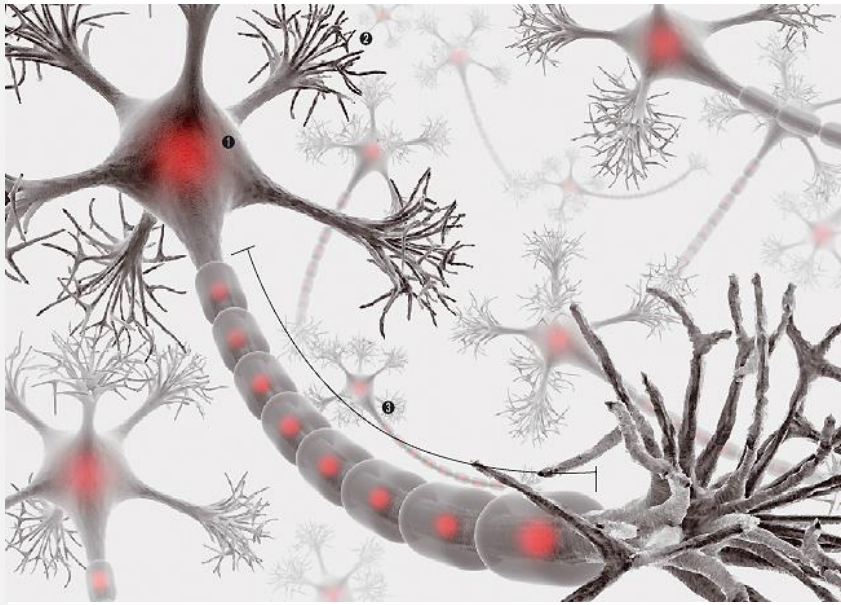
- 딥러닝 프레임워크의 역할과 Tensorflow의 사용법을 이해한다.
- .

주요 내용

1. 하드웨어
2. 소프트웨어
3. Tensorflow
4. Tensorflow 모델 정의 및 훈련

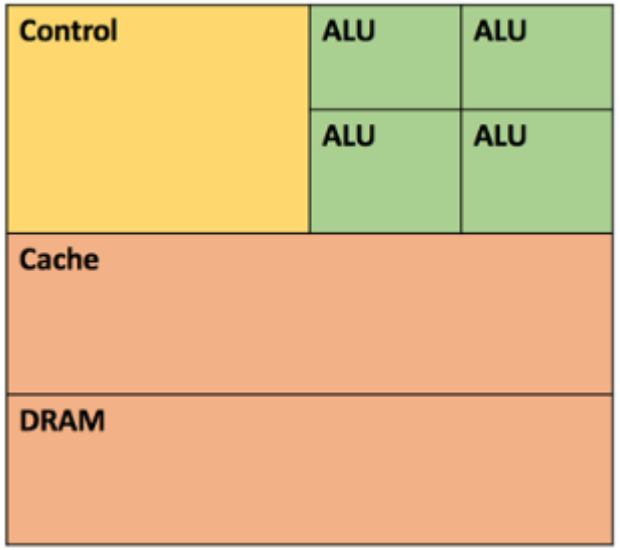


1 하드웨어



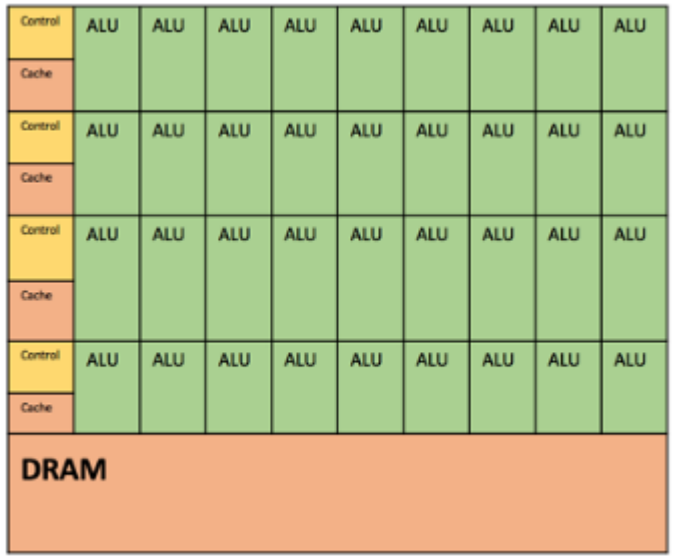
CPU vs GPU

CPU



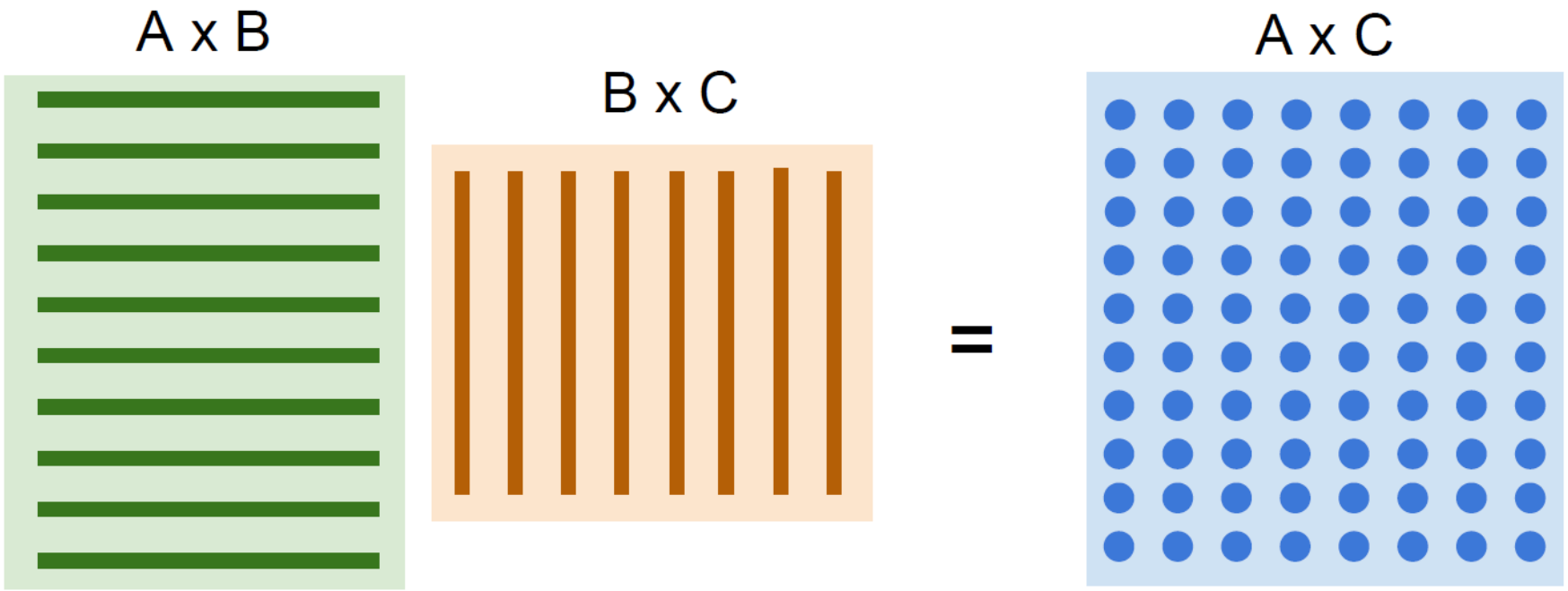
- Core 개수가 적음
- 각 Core는 매우 빠르고 범용적
- **Sequential task에 적합**

GPU



- Core 개수가 매우 많음
- 각 Core는 느리고 제한적
- **Parallel task에 적합**

Matrix Multiplication



Stanford CS231n: Convolutional Neural Networks for Visual Recognition

TPU (Tensor Processing Unit)

Deep Learning 전용 Processor

Google Cloud TPU 2.0 = 180 TFLOP!
Google Cloud TPU 3.0 = 2.0보다 8배 빨라짐

NVIDIA TITAN V =
14 TFLOP (FP32),
112 TFLOP (FP16)

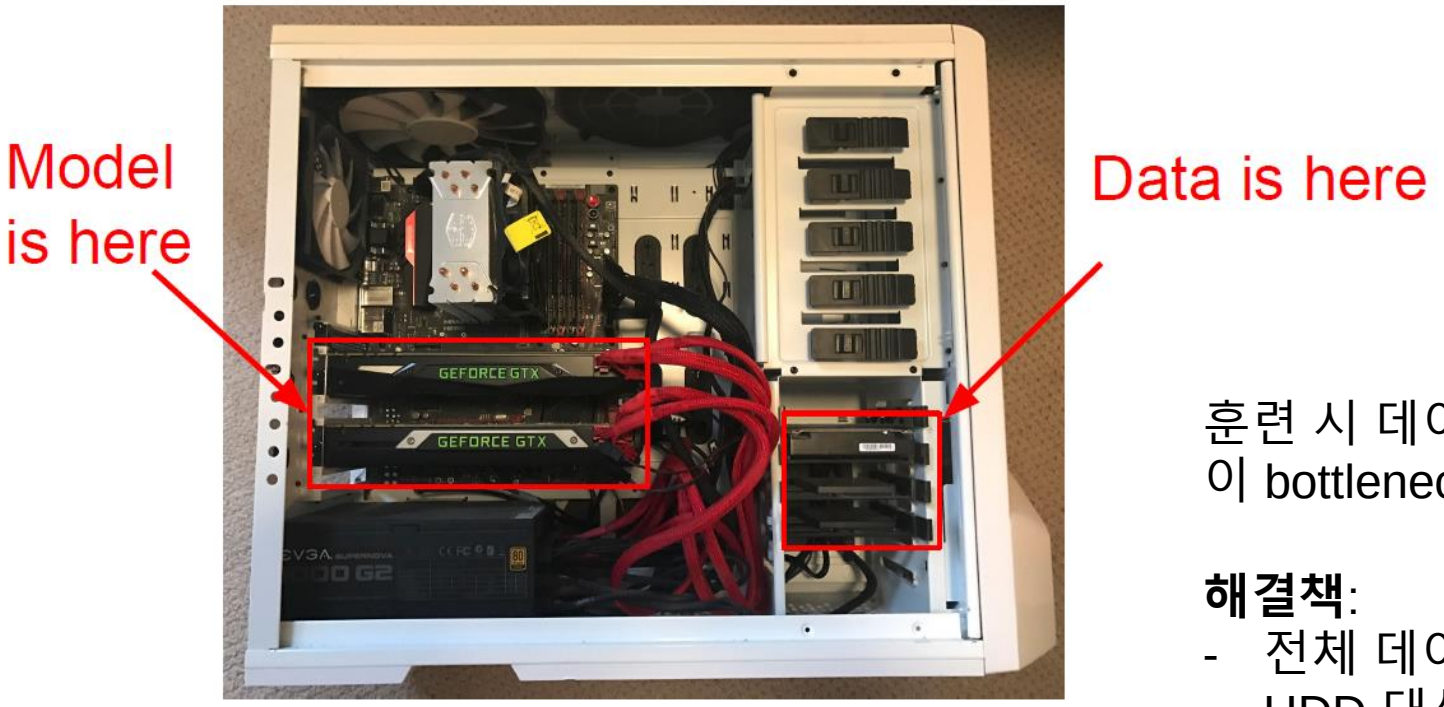
Stanford CS231n: Convolutional Neural Networks for Visual Recognition

CPU vs GPU vs TPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$385	~540 GFLOPs FP32
GPU (NVIDIA RTX 2080 Ti)	3584	1.6 GHz	11 GB GDDR6	$1199	~13.4 TFLOPs FP32
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	$4.50 per hour	~180 TFLOP

Stanford CS231n: Convolutional Neural Networks for Visual Recognition

CPU / GPU Communication



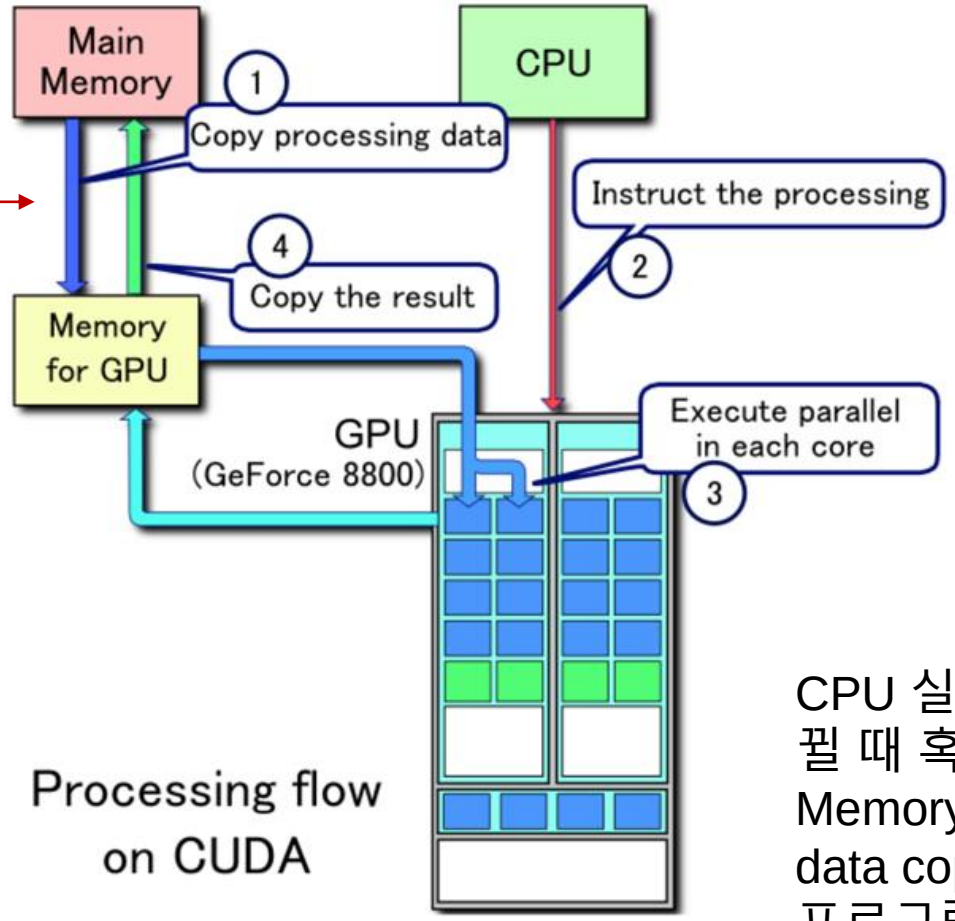
훈련 시 데이터를 읽어서 GPU로 보내는 것이 bottleneck이 될 수 있음.

해결책:
- 전체 데이터를 RAM으로 읽기
- HDD 대신 SSD 사용하기
- 데이터를 읽을 때 여러 CPU thread 사용

Stanford CS231n: Convolutional Neural Networks for Visual Recognition

CPU / GPU Communication

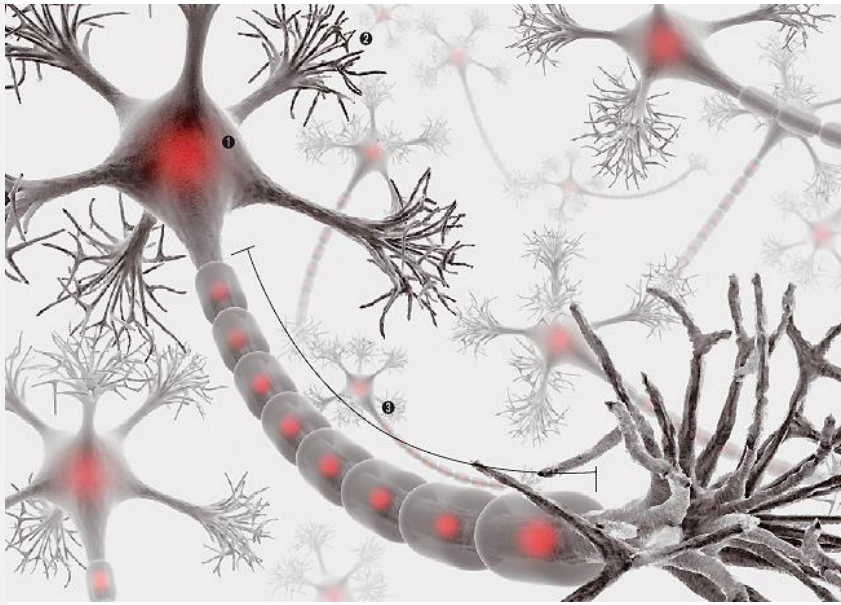
병목지점 (Bottleneck)



CPU 실행에서 GPU 실행으로 바뀔 때 혹은 그 반대의 경우 Main Memory와 GPU Memory 사이의 data copy가 bottleneck이 되므로 프로그램 설계를 잘해야 함

https://www.datascience.com/blog/cpu-gpu-machine-learning

2 소프트웨어



딥러닝 프레임워크

Caffe
(UC Berkeley) → Caffe2
(Facebook)

Paddle
(Baidu)

Chainer

Torch
(NYU / Facebook) → PyTorch
(Facebook)

개발하기 편한 프레임워크

MXNet
(Amazon)
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

Theano
(U Montreal) → TensorFlow
(Google)

가장 보편적인 프레임워크

Deeplearning4j

Stanford CS231n: Convolutional Neural Networks for Visual Recognition

딥러닝 프레임워크를 왜 사용해야 하는가?

Quick

새로운 아이디어를 신속하게 개발하고 테스트 할 수 있다

Automatic

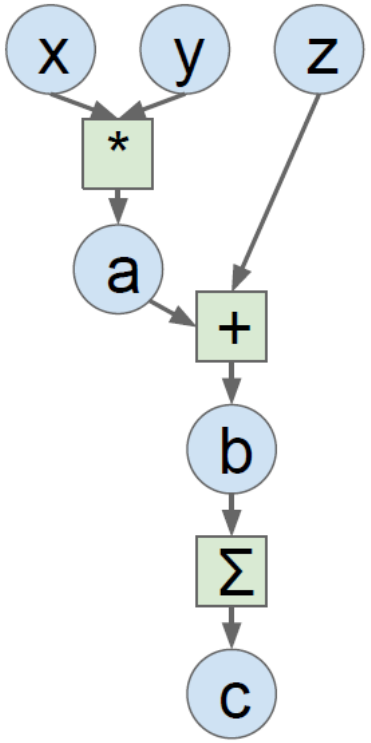
복잡한 Gradient 계산을 자동으로 해준다.

Efficient

GPU를 효율적 활용할 수 있다. (cuDNN, cuBLAS, etc Wrapper)

딥러닝 구현 시 NumPy의 한계

```
import numpy as np
np.random.seed(0)

N, D = 3, 4
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

# Forward Pass
a = x * y
b = a + z
c = np.sum(b)

# Gradient 계산
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



장점:
- Clean API
- 수치를 다루는 코드를 쉽게 작성 할 수 있음

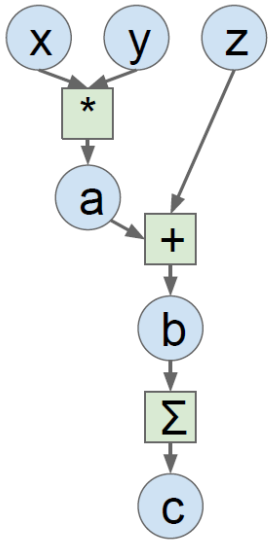
단점:
- Gradient를 직접 계산해야 함
- GPU에서 실행할 수 없음

참고 Gradient 계산

$$c = \sum_i \sum_j b_{i,j}$$

$$b = a + z$$

$$a = x * y$$



Local Gradient

$$\frac{\partial c}{\partial b_{i,j}} = 1$$

$$\frac{\partial c}{\partial b} = \begin{bmatrix} \frac{\partial c}{\partial b_{1,1}} & \frac{\partial c}{\partial b_{1,2}} & \cdots & \frac{\partial c}{\partial b_{1,D}} \\ \frac{\partial c}{\partial b_{2,1}} & \frac{\partial c}{\partial b_{2,2}} & \cdots & \frac{\partial c}{\partial b_{2,D}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial c}{\partial b_{N,1}} & \frac{\partial c}{\partial b_{N,2}} & \cdots & \frac{\partial c}{\partial b_{N,D}} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$

$$\frac{\partial b}{\partial a} = 1_{[N,D]} \qquad \frac{\partial b}{\partial z} = 1_{[N,D]}$$

$$\frac{\partial a}{\partial x} = y \qquad \frac{\partial a}{\partial y} = x$$

Global Gradient

$$\frac{\partial c}{\partial b} = 1_{[N,D]}$$

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial a} = 1_{[N,D]}$$

$$\frac{\partial c}{\partial z} = \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial z} = 1_{[N,D]}$$

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial a} \cdot \frac{\partial a}{\partial x} = y$$

$$\frac{\partial c}{\partial y} = \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial a} \cdot \frac{\partial a}{\partial y} = x$$

딥러닝 프레임워크 Gradient 자동 계산

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

# Forward Pass
a = x * y
b = a + z
c = np.sum(b)

# Gradient 계산
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

TensorFlow

```
import tensorflow as tf

N, D = 3, 4
x = tf.Variable(tf.random.normal((N, D)))
y = tf.Variable(tf.random.normal((N, D)))
z = tf.Variable(tf.random.normal((N, D)))

# Forward Pass
with tf.GradientTape() as tape:
    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

# Gradient 계산
grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
```

딥러닝 프레임워크 GPU 실행

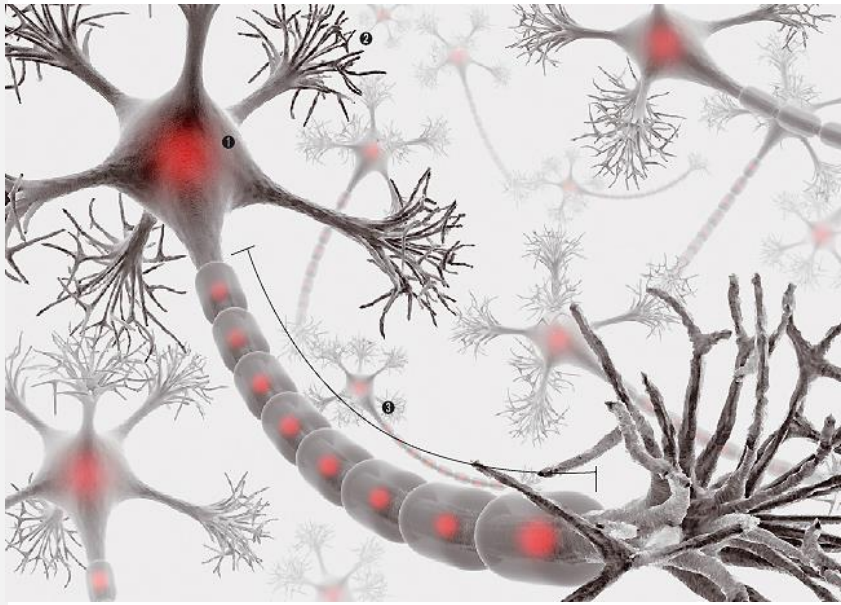
CPU 사용

```
import tensorflow as tf

N, D = 3, 4
with tf.device("CPU:0"):
    x = tf.Variable(tf.random.normal((N, D)))
    y = tf.Variable(tf.random.normal((N, D)))
    z = tf.Variable(tf.random.normal((N, D)))

    # Forward Pass
    with tf.GradientTape() as tape:
        a = x * y
        b = a + z
        c = tf.reduce_sum(b)

    # Gradient 계산
    grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
```

GPU 사용

```
import tensorflow as tf

N, D = 3, 4
with tf.device("GPU:0"):
    x = tf.Variable(tf.random.normal((N, D)))
    y = tf.Variable(tf.random.normal((N, D)))
    z = tf.Variable(tf.random.normal((N, D)))

    # Forward Pass
    with tf.GradientTape() as tape:
        a = x * y
        b = a + z
        c = tf.reduce_sum(b)

    # Gradient 계산
    grad_x, grad_y, grad_z = tape.gradient(c, [x, y, z])
```

3 TensorFlow



TensorFlow

- TensorFlow 0.5 Release (2015. 11)
- **TensorFlow 2.0** Release (2019. 10)
- C++ core, Python API
- High-level API : **Keras**
- Community
  - 117,000+ GitHub stars
  - TensorFlow.org : Blogs, Documentation, DevSummit, YouTube talks

http://tensorflow.org

TensorFlow 2.0

Static Graph 방식

Define and Run

- 계산 그래프를 정의하고 난 후 명시적으로 그래프를 실행하는 방식

Dynamic Graph 방식

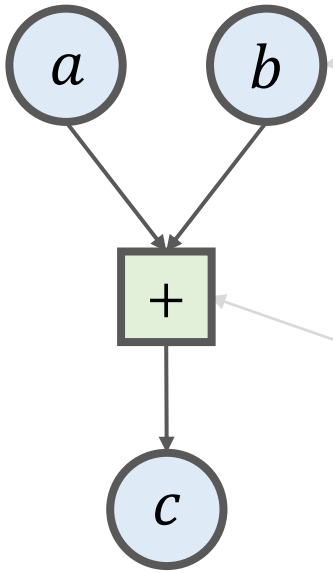
Define by Run

- 코드를 실행하면서 동시에 계산 그래프를 생성하는 방식
- PyTorch, Chainer 등에서 지원

Computational Graph

신경망에 필요한 계산 과정을 계산 그래프(Computational Graph)로 정의

계산 그래프



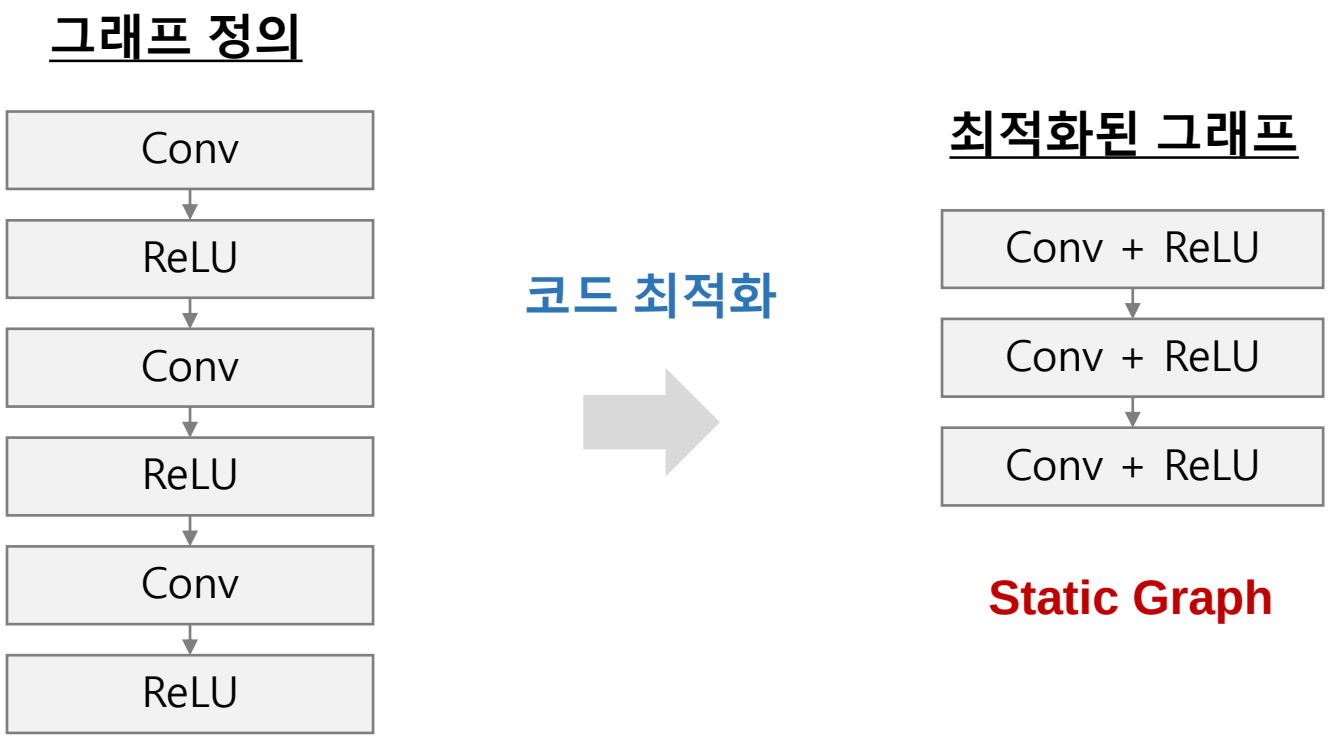
텐서 (Tensor) :
- 데이터가 저장된 다차원 배열

오퍼레이션 (Operator) :
- 다차원 배열에 대한 연산을 수행
- 산술연산, 논리연산, 행렬연산 등

예제 : c = a + b 계산

Define and Run

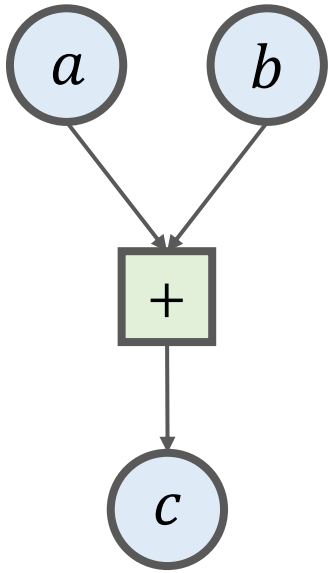
실행에 최적화된 Static Graph 생성



Define and Run

세션을 통해 계산 그래프를 실행

계산 그래프 정의



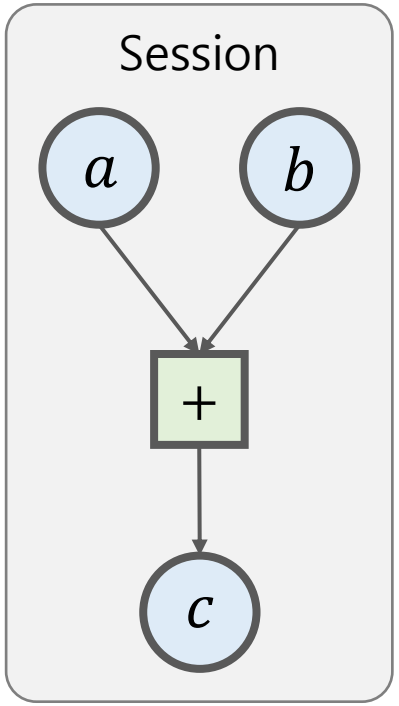
세션 생성

Session

세션 (Session) :

- 계산 그래프를 실행하는 단위
- 실행 환경을 추상화 한 개념

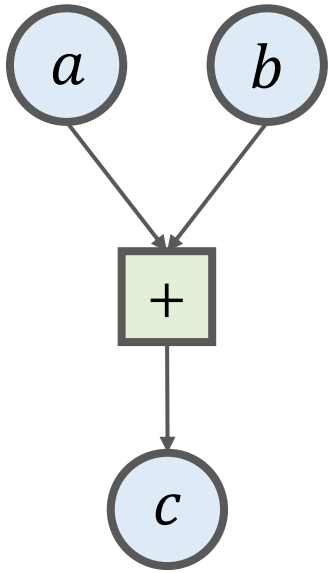
세션 실행



Define and Run

동일 계산 그래프를 여러 세션으로 동시에 실행 가능

계산 그래프 정의



세션 생성

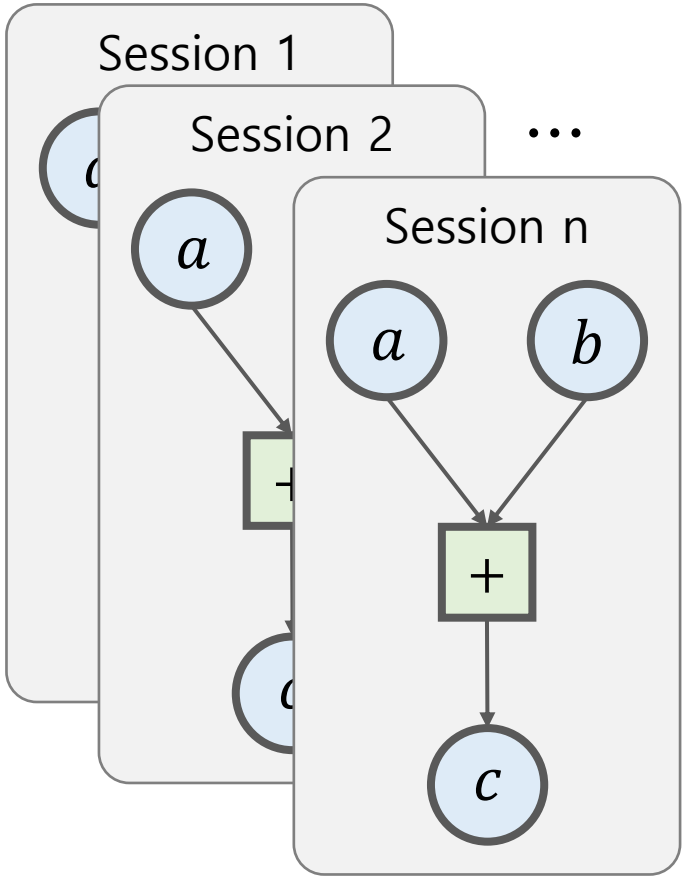
Session 1

Session 2

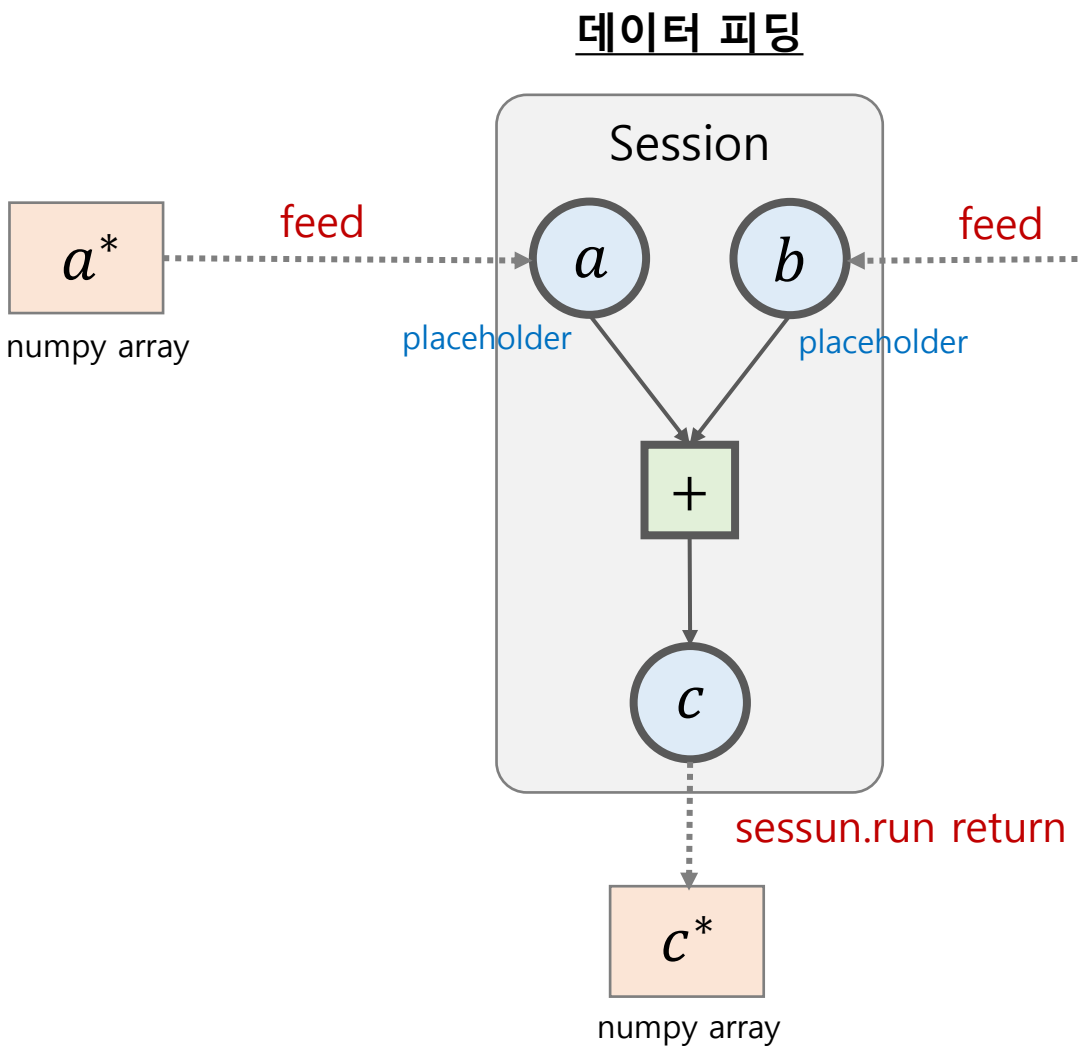
...

Session n

세션 실행



Define and Run



Tensor의 종류

PlaceHolder :
그래프 외부에서 데이터를 전달 받기위한 텐서

Variable :
그래프 내부 데이터를 저장하기 위한 텐서
초기화가 필요

Constant :
그래프 내부 상수 데이터를 갖고 있는 텐서

Define and Run

TensorFlow 1.x 코드 형태

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

계산 그래프 정의

Gradient 계산

세션 실행

장점:
- 실행 성능을 최적화 할 수 있다.
- 확장성이 좋다.

단점:
- 프로그램 방식이 익숙하지 않다.
- 디버깅이 어렵다.
- 조건에 따라 동적으로 변화하거나 반복적으로 확장되는 Dynamics Graph를 만들기 어렵다.

Static Graph 방식의 한계

Conditional

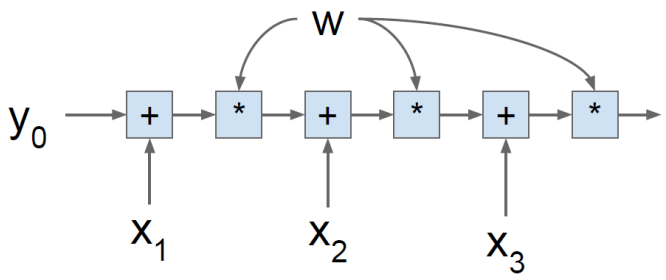
- 조건에 따라 실행되는 그래프가 변경 되어야 하는 경우

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

Loop

- 입력의 크기에 따라 그래프가 재귀적으로 확장되어야 하는 경우

$$y_t = (y_{t-1} + x_t) * w$$



Dynamic Graph 방식이 필요!

Define by Run

Define and Run에서 Define by Run 으로!

```
import tensorflow as tf

a = tf.constant(5)
b = tf.constant(3)

c = a + b

with tf.session() as sess:
    print(sess.run(c))
```

symbolic

```
import tensorflow as tf

a = tf.constant(5)
b = tf.constant(3)

c = a + b

print(c)
```

concrete

TensorFlow 1.x :	8	Tensor("add_2:0", shape=(), dtype=int32)
TensorFlow 2.x :	Error	tf.Tensor(8, shape=(), dtype=int32)

Define by Run

TensorFlow 1.x

z = w * x + b 구현

```
import tensorflow as tf

## 그래프 정의
g = tf.Graph()
with g.as_default():
    x = tf.placeholder(dtype=tf.float32,
                       shape=(None), name='x')
    w = tf.Variable(2.0, name='weight')
    b = tf.Variable(0.7, name='bias')
    z = w * x + b
    init = tf.global_variables_initializer()

## 세션 생성 및 그래프 g 전달
with tf.Session(graph=g) as sess:
    ## w와 b 초기화
    sess.run(init)
    ## z 평가
    for t in [1.0, 0.6, -1.8]:
        print('x=%4.1f --> z=%4.1f'%(
            t, sess.run(z, feed_dict={x:t})))
```

TensorFlow 2.x

```
import tensorflow as tf

w = tf.Variable(2.0, name='weight')
b = tf.Variable(0.7, name='bias')

# ## z 평가
for x in [1.0, 0.6, -1.8]:
    z = w * x + b
    print('x=%4.1f --> z=%4.1f'%(x, z))
```

AutoGraph

모듈 별로 Static Graph 생성을 지원하는 방식

tf.Graph() + tf.Session() → @tf.function

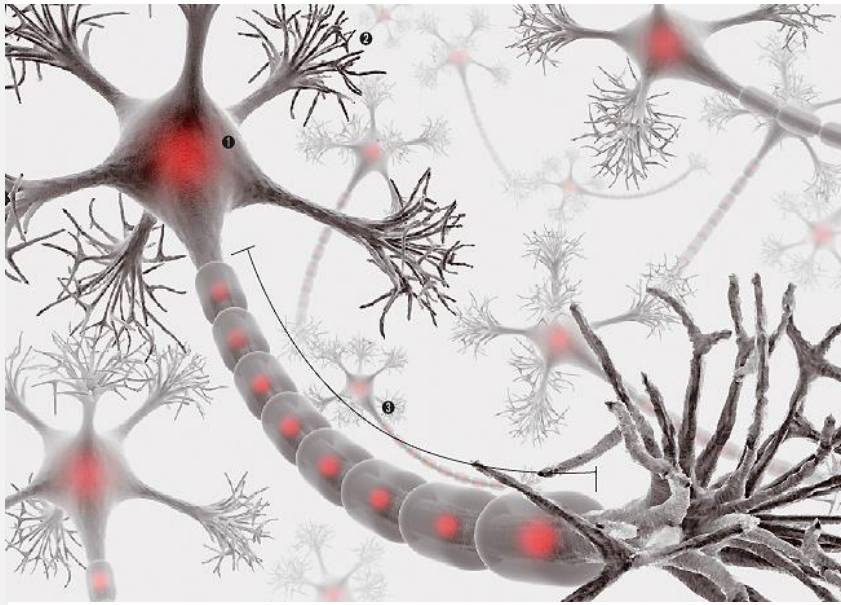
```
# TensorFlow 2.x
@tf.function
def simple_func():
    # complex computation with pure python
    ...
    return z

output = simple_func(input)
```

- for/while → tf.while_loop
- if → tf.cond

- @tf.function를 붙이면 그래프 생성해서 GPU나 TPU를 사용해서 작동
- @tf.function가 붙은 함수로부터 호출된 모든 함수들은 그래프 모드에서 동작
- 조건, 루프와 같은 제어문 사용 시 tf.cond, tf.while_loop와 같은 텐서플로 연산으로 변환

Keras in TensorFlow2.0 by 박해선님

4 Tensorflow
모델 정의 및 훈련



모델 정의 tf.Module

tf.Module : 신경망 모듈 클래스의 베이스 클래스

```
tf.Module(
    name=None
)
```

- **trainable_variables** : 훈련 변수들의 목록
- Variables : 모든 변수들의 목록
- Submodules : 멤버 Module의 목록

모듈에 포함된 tf.Variable, tf.Module, input에 적용되는 function들을 관리함

https://www.tensorflow.org/api_docs/python/tf/Module

모델 정의 tf.Module

tf.Module을 이용해서 계층 정의하기

```
class Dense(tf.Module):

  def __init__(self, in_features, out_features, name=None):

    super(Dense, self).__init__(name=name)

    # 가중치와 편향 정의
    self.w = tf.Variable(tf.random.normal([in_features, out_features]), name='w')
    self.b = tf.Variable(tf.zeros([out_features]), name='b')

  def __call__(self, x):

    y = tf.matmul(x, self.w) + self.b    # 가중 합산
    return tf.nn.relu(y)  # 활성 함수 실행
```

모델 정의 tf.Module

tf.Module을 이용해서 모델 정의하기

```
class MLP(tf.Module):

  def __init__(self, input_size, sizes, name=None): # sizes에는 각 계층의 뉴런 개수가 정의되어 있음
    super(MLP, self).__init__(name=name)

    self.layers = [] # 계층의 리스트
    with self.name_scope:
      for size in sizes:
        self.layers.append(Dense(input_size=input_size, output_size=size)) # 각 계층 정의
        input_size = size

  @tf.Module.with_name_scope
  def __call__(self, x):
    for layer in self.layers: # 각 계층을 순서대로 호출
      x = layer(x)
    return x
```

신경망 훈련 tf.GradientTape

```
@tf.function
def train_step(input, target):
  with tf.GradientTape() as tape:
    # forward Pass
    predictions = model(input)
    # compute the loss
    loss = tf.reduce_mean(
            tf.keras.losses.sparse_categorical_crossentropy(
            target, predictions, from_logits=True))
  # compute gradients
  grads = tape.gradient(loss, model.trainable_variables)
  # perform a gradient descent step
  optimizer.apply_gradients(zip(grads, model.trainable_variables))
  return loss
```

신경망 모델 실행 및 Loss 계산

Gradient 계산 (Backpropagation)

Parameter Update (최적화)

Thank you!