PYTORCH INTRODUCTION

서울대 전기정보공학부
명예교수 성원용

Many slide from

David Völgyes

Jingfeng

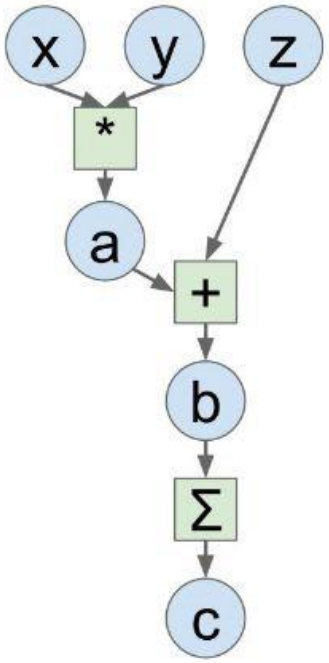
OUTLINE

- GPU and Deep learning frameworks PyTorch
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - CreatingData loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models

1. DEEP LEARNING FRAMEWORK

- Speed – deep learning requires a lot of computation
  - Leveraging the power of GPGPU (General Purpose Graphics Processing Unit)
- Deep learning model training heavily uses backpropagation for weight updates, which is very complex for programming. Autograd solves this problem.
- Easy of use and reuse

Computation Graph



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)

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

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 numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    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
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a =x * y
b =a + z
c=torch.sum(b)

c.backward()
```

GETTING STARTED WITH PYTORCH

Installation

Via Anaconda/Miniconda:

```
conda install pytorch
```

Via pip:

```
pip3 install torch
```

DEEP NEURAL NETWORK MODEL

- Matrix-vector or matrix-matrix operations
- The dimension is around 1K (1024) -> millions of trainable weights

$$z_k^{[L]} = \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]}$$

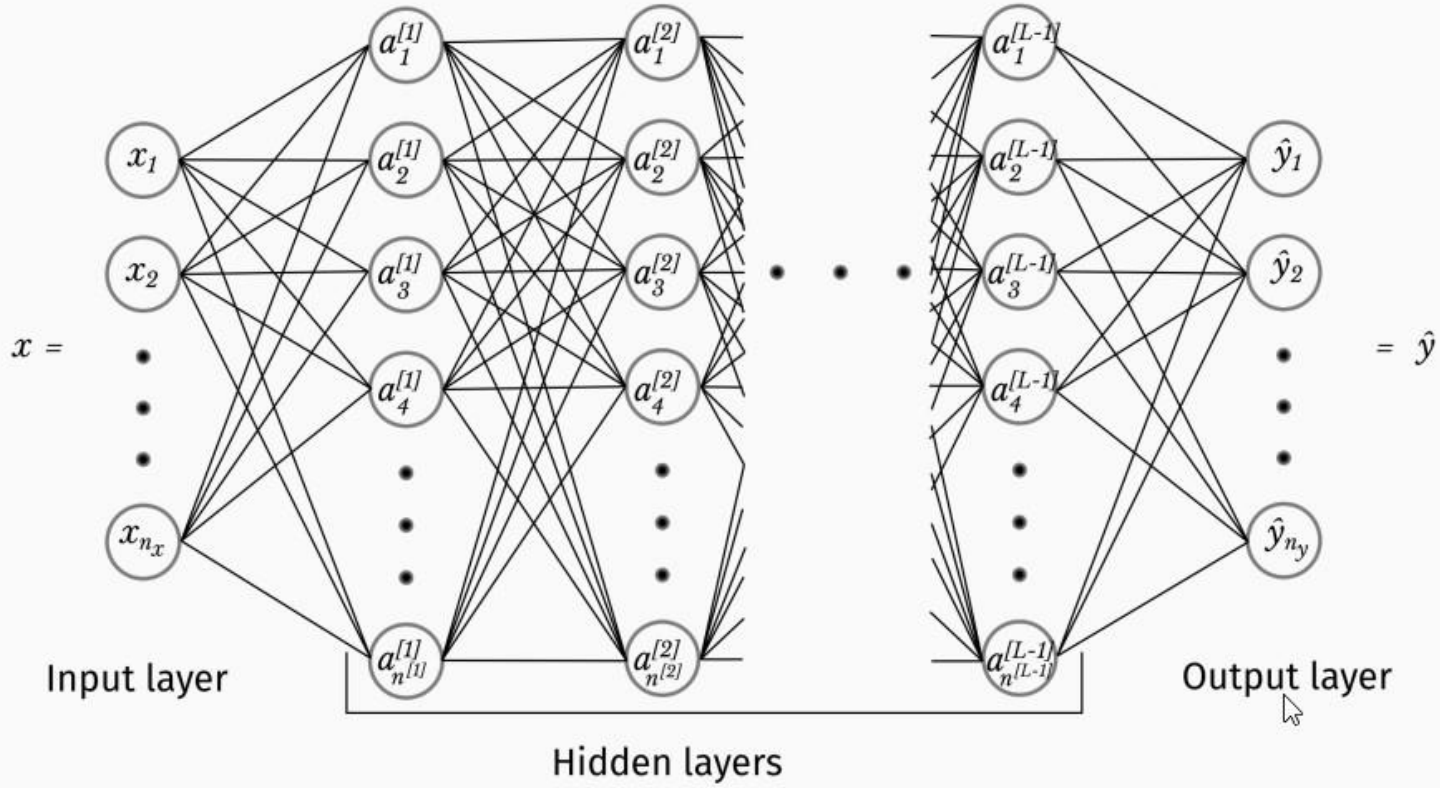
$$a_k^{[L]} = s(z_k^{[L]})$$

$$= \hat{y}_k$$

for

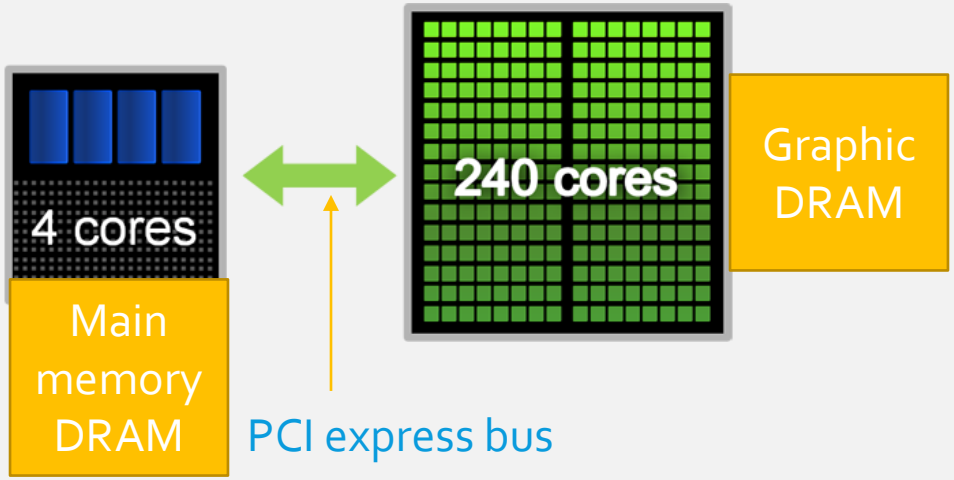
$$k = 1, \ldots, n_y,$$

$$= 1, \ldots, n^{[L]}.$$



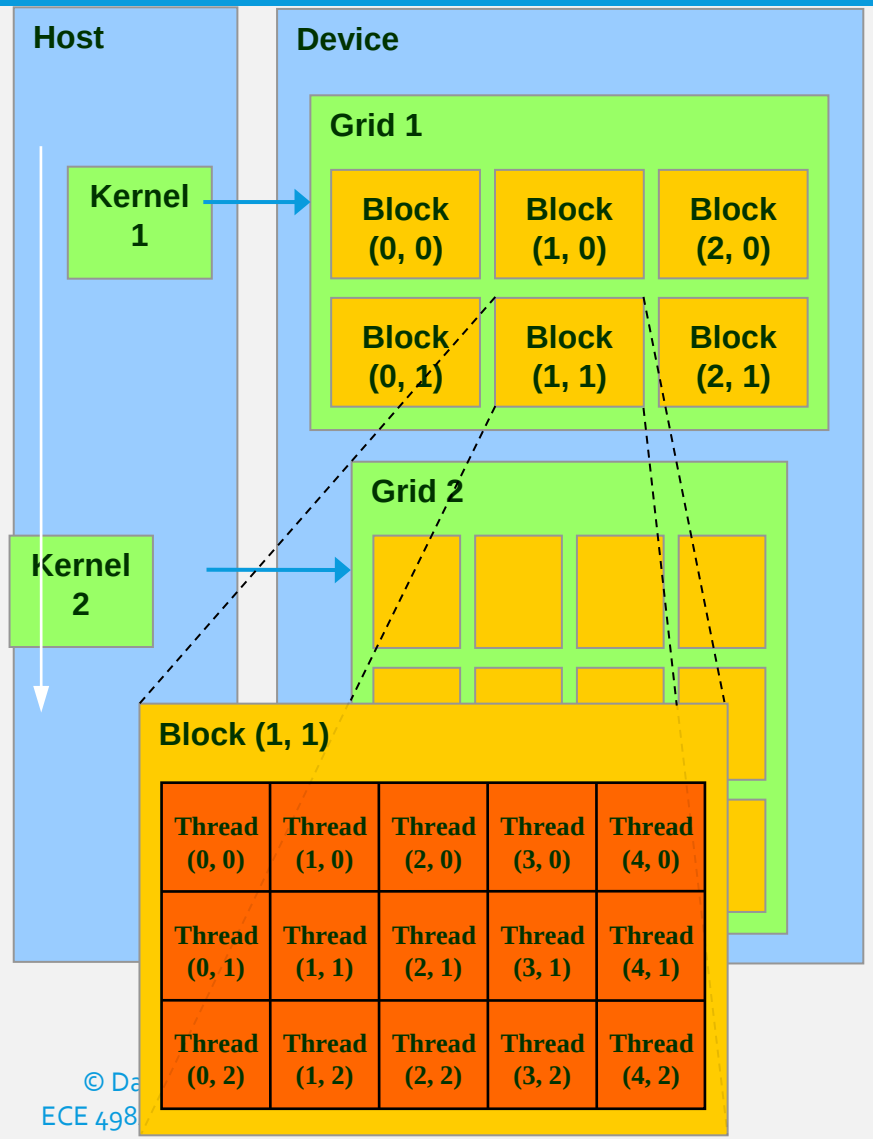
GPU AND CPU

- Typically GPU and CPU coexist in a heterogeneous setting (GPU is called a device, device 0, 1, …)
- Control oriented part runs on CPU, and more arithmetic intensive parts run on GPU (fine-grained parallelism)
- NVIDIA's GPU architecture is called CUDA (Compute Unified Device Architecture) architecture, accompanied by CUDA programming model, and CUDA C language



CUDA PROGRAMMING MODEL: A HIGHLY MULTITHREADED COPROCESSOR

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- GPU runs a kernel at a time. GPU is not intended t o run a complex program
  - Copy data from CPU to GPU
  - Compute on GPU
  - Copy data back from GPU to CPU
- By default, execution on host doesn't wait for kernel to finish



MAJOR DEEP LEARNING FRAMEWORKS

서울대학교 SEOUL NATIONAL UNIVERSITY

- pytorch (developed by Facebook) – good for model development (easy to learn)
- Tensorflow/Keras (developed by Google) - good for embedded implementations through TensorFlow Lite
- Caffe (developed by Facebook)
- All opensource

INSTALLING PYTORCH


```
>>> import numpy as np
>>> import torch
>>> import sys
>>> import matplotlib
>>> print(f'Python  version: {sys.version}')
Python  version: 3.8.1 | packaged by conda-forge | (default, Jan 29 2020, 14:55:04) [GCC 7.3.0]

>>> print(f'Numpy   version: {np.version.version}')
Numpy   version: 1.17.5

>>> print(f'PyTorch version: {torch.version.__version__}')
PyTorch version: 1.4.0

>>> print(f'Matplotlib version: {matplotlib.__version__}')
Matplotlib version: 3.1.2

>>> print(f'GPU present: {torch.cuda.is_available()}')
GPU present: False
```

Package	Description
torch	The top-level PyTorch package and tensor library.
torch.nn	A subpackage that contains modules and extensible classes for building neural networks.
torch.autograd	A subpackage that supports all the differentiable Tensor operations in PyTorch.
torch.nn.functional	A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations.
torch.optim	A subpackage that contains standard optimization operations like SGD and Adam.
torch.utils	A subpackage that contains utility classes like data sets and data loaders that make data preprocessing easier.
torchvision	A package that provides access to popular datasets, model architectures, and image transformations for computer vision.

2.TENSORS

- PyTorch ’s tensors are very similar to NumPy’s ndarrays but they have a device, 'cpu', 'cuda', or 'cuda:X’ and also grad attribute supporting trainable parameters.
- Tensors are similar to NumPy’s ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

- Common operations for creation and manipulation of these Tensors are similar to those for ndarrays in NumPy. (rand, ones, zeros, indexing, slicing, reshape, transpose, cross product, matrix product, element-wise multiplication)

LOADING DATA, DEVICES AND CUDA

서울대학교 SEOUL NATIONAL UNIVERSITY

Numpy arrays to PyTorch tensors

- `torch.from_numpy(x_train)`
- `Returns a cpu tensor!`

PyTorch tensor to numpy

- `t.numpy()`

Using GPU acceleration

- `t.to()`
- Sends to whatever device (cuda or cpu)

Fallback to cpu if gpu is unavailable:

- `torch.cuda.is_available()`

Check cpu/gpu tensor OR numpy array ?

- `type(t)` or `t.type()` returns
  - numpy.ndarray
  - torch.Tensor
    - CPU - torch.cpu.FloatTensor
    - GPU - torch.cuda.FloatTensor

PYTORCH TENSOR

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- They might require gradients (used for backpropagation based training)

```
>>> t = torch.tensor([1, 2, 3], device='cpu',
...                  requires_grad=False, dtype=torch.float32)
>>> print(t.dtype) torch.float32
>>> print(t.device) cpu
>>> print(t.requires_grad)
False
>>> t2 = t.to(torch.device('cuda'))
>>> t3 = t.cuda()   # or you can use shorthand
>>> t4 = t.cpu()
```

Performs Tensor dtype and/or device conversion. A torch.dtype and torch.device are inferred from the arguments of self.to(*args, **kwargs).

This package adds support for CUDA tensor types, that implement the same function as CPU tensors, but they utilize GPUs for computation.

PYTORCH DATA-TYPES

서울대학교 SEOUL NATIONAL UNIVERSITY

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	torch.float32 or torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64-bit floating point	torch.float64 or torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
16-bit floating point	torch.float16 or torch.half	torch.HalfTensor	torch.cuda.HalfTensor
8-bit integer (unsigned)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8-bit integer (signed)	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16-bit integer (signed)	torch.int16 or torch.short	torch.ShortTensor	torch.cuda.ShortTensor
32-bit integer (signed)	torch.int32 or torch.int	torch.IntTensor	torch.cuda.IntTensor
64-bit integer (signed)	torch.int64 or torch.long	torch.LongTensor	torch.cuda.LongTensor
Boolean	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

See: https://pytorch.org/docs/stable/tensors.html

CREATING ARRAYS/TENSOR

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- eye: creating diagonal matrix / tensor
- zeros: creating tensor filled with zeros
- ones: creating tensor filled with ones
- linspace: creating linearly increasing values
- arange: linearly increasing integers

```
>>> torch.eye(3,          dtype=torch.double)
tensor([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]],      dtype=torch.float64)
>>> torch.arange(6)
tensor([0, 1, 2, 3, 4, 5])
```

```
>>> torch.eye(2)
tensor([[1., 0.],
        [0., 1.]])
>>> torch.zeros(2,2)
tensor([[0., 0.],
        [0., 0.]])
>>> torch.ones(2,2)
tensor([[1., 1.],
        [1., 1.]])

>>> torch.rand(2,2)  tens
or([[0.6849, 0.1091],
        [0.4953, 0.8975]])

>>> torch.empty(2,2) # NEVER USE IT! Creates uninitialized tensor.
tensor([[-2.2112e-16,  3.0693e-41],
        [-3.0981e-16,  3.0693e-41]])

>>> torch.arange(6)  tensor
([0, 1, 2, 3, 4, 5])
```

- Remarks:
  - arrays / tensors must be on the same device.
  - only detached arrays can be converted to numpy (see later)
  - if data types are not the same, casting might be needed (v1.1 or older)
  - E.g. adding an integer and a float tensor together.
- `# tuples, lists, arrays, etc. can be converted automatically:`
- `>>> t2 = torch.tensor(...)`

```
>>> import imageio
>>> img = imageio.imread('example.png')  #  reading data from disk
>>> t = torch.from_numpy(a)              #  input from numpy array
>>> out = model(t)                       #  processing
>>> result = out.numpy()                 #  converting back to numpy
```

detach()와 clone()은 기존 Tensor를 복사하는 방법 중 하나입니다.

detach() : 기존 Tensor에서 gradient 전파가 안되는 텐서 생성
단 storage를 공유하기에 detach로 생성한 Tensor가 변경되면 원본 Tensor도 똑같이 변합니다.
clone() : 기존 Tensor와 내용을 복사한 텐서 생성

COPYING/CREATING NEW TENSORS

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- Copy data
  - torch.Tensor()
  - torch.tensor()
  - torch.clone()
  - type casting
- Share Data
  - torch.as_tensor()
  - torch.from_numpy()
  - torch.view()
  - torch.reshape()

torch.Tensor

int 입력시 float으로 변환
torch 데이터 입력시 입력 받은 데이터의 메모리 공간을 사용
list, numpy 데이터 입력 시 입력 받은 데이터를 복사하여 새롭게 torch.Tensor를 만든 후 사용

torch.tensor

int 입력시 int 그대로
입력 받은 데이터를 새로운 메모리 공간으로 복사 후 사용

torch.clone … Returns a copy of input . … This function is differentiable, so gradients will flow back from the result of this operation to input

```
a = a.type(torch.float64)
print(a.dtype) # torch.float64
```

Tensor class reference

CLASS `torch.Tensor`

There are a few main ways to create a tensor, depending on your use case.

- To create a tensor with pre-existing data, use `torch.tensor()`.
- To create a tensor with specific size, use `torch.*` tensor creation ops (see Creation Ops).
- To create a tensor with the same size (and similar types) as another tensor, use `torch.*_like` tensor creation ops (see Creation Ops).
- To create a tensor with similar type but different size as another tensor, use `tensor.new_*` creation ops.

torch.tensor

int 입력시 int 그대로 입력
입력받은 데이터를 새로운 메모리 공간에 복사해 Tensor 객체 생성 (call by value)

torch.Tensor
int 입력시 float으로 변환
데이터 입력 시(Tensor 객체로) 입력 받은 메모리 공간을 그대로 사용 (call by reference)
데이터 입력 시(list나 numpy 로) 입력 받은 값을 복사하여 Tensor 객체 생성(call by value)

MEMORY: SHARING VS COPYING

```
>>> a = np.arange(6)                    #  [0,1,2,3,4,5]
>>> t = torch.from_numpy(a)
>>> t[2] = 11
>>> t
tensor([ 0,  1, 11,  3,  4,  5])
>>> a
array([ 0,  1, 11,  3,  4,  5])  # Changed the underl numpy array to
                                    ying                    o!


>>> b = a.copy()
>>> p = t.clone()
>>> t[0] = 7                          # a,t change, b, p remain intact.
```

INDEXING – NUMPY INDEXING WORKS


```
>>> t    = torch.arange(12).reshape(3,4)
tensor([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11]])

>>> t[1,1:3]
tensor([5, 6])
>>> t[:,:]          = 0 # fill everything with 0, a.k.a. t.fill_(0)
tensor([[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]])
```

```
x.size()                          #* return tuple-like object of dimensions, old codes
x.shape                           # return tuple-like object of dimensions, numpy style
x.ndim                            # number of dimensions, also known as .dim()
x.view(a,b,...)                   #* reshapes x into size (a,b,...)
x.view(-1,a)                      #* reshapes x into size (b,a) for some b
x.reshape(a,b,...)                # equivalent with .view()
x.transpose(a,b)                  # swaps dimensions a and b
x.permute(*dims)                  # permutes dimensions; missing in numpy
x.unsqueeze(dim)                  # tensor with added axis; missing in numpy
x.unsqueeze(dim=2)                # (a,b,c) tensor -> (a,b,1,c) tensor; missing in numpy

torch.cat(tensor_seq, dim=0)      # concatenates tensors along dim
# For instance:
```

transpose() vs permute()
permute()와 transpose()는 유사한 방식으로 작동한다.
transpose()는 딱 두 개의 차원을 맞교환할 수 있다. 그러나
permute()는 모든 차원들을 맞교환할 수 있다. 예를 들면,

```
x = torch.rand(16, 32, 3)
y = x.tranpose(0, 2)  # [3, 32, 16]
z = x.permute(2, 1, 0)  # [3, 32, 16]
```

torch.view has existed for a long time. It will return a tensor with the new shape. The returned tensor will share the underling data with the original tensor. See the documentation here.

On the other hand, it seems that torch.reshape has been introduced recently in version 0.4. According to the document, this method will

Returns a tensor with the same data and number of elements as input, but with the specified shape. When possible, the returned tensor will be a view of input. Otherwise, it will be a copy. Contiguous inputs and inputs with compatible strides can be reshaped without copying, but you should not depend on the copying vs. viewing behavior.

It means that torch.reshape may return a copy or a view of the original tensor. You can not count on that to return a view or a copy. According to the developer:

if you need a copy use clone() if you need the same storage use view(). The semantics of reshape() are that it may or may not share the storage and you don't know beforehand.

Another difference is that reshape() can operate on both contiguous and non-contiguous tensor while view() can only operate on contiguous tensor. Also see here about the meaning of contiguous.

```
>>>t = torch.arange(6)
tensor([0, 1, 2, 3, 4, 5])
>>> t.reshape(2,3) # same as t.view(2,3 or t.view(2,-1)
tensor([[0, 1, 2],
[3, 4, 5]])
>>> t.reshape(2,3).unsqueeze(1)
tensor([[[0, 1, 2]],
[[3, 4, 5]]])
>>> t.reshape(2,3).unsqueeze(1).shape
torch.Size([2, 1, 3])
```

TORCH.SQUEEZE

torch.squeeze(input, dim=None, *, out=None) → Tensor

Returns a tensor with all the dimensions of input of size 1 removed.

For example, if input is of shape: (A \times 1 \times B \times C \times 1 \times D)(A×1×B×C×1×D) then the out tensor will be of shape: (A \times B \times C \times D)(A×B×C×D).

When dim is given, a squeeze operation is done only in the given dimension. If input is of shape: (A \times 1 \times B)(A×1×B), squeeze(input, 0) leaves the tensor unchanged, but squeeze(input, 1) will squeeze the tensor to the shape (A \times B)(A×B).

```
>>> x = torch.zeros(2, 1, 2, 1, 2)
>>> x.size()
torch.Size([2, 1, 2, 1, 2])
>>> y = torch.squeeze(x)
>>> y.size()
torch.Size([2, 2, 2])
```

```
>>> x = torch.tensor([1, 2, 3, 4])
>>> torch.unsqueeze(x, 0)
tensor([[ 1,  2,  3,  4]])
>>> torch.unsqueeze(x, 1)
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4]])
```

TORCH 사칙연산 '+', ADD

```
import numpy as np
import torch
x= torch.Tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
print(x)
x1 = torch.randn_like(x, dtype=torch.float)
print(x1)
print(x.size())
y = x + x1
print(y)
torch.add(x,x1)
result = torch.empty(2, 3)
print(y.add_(x))    #In_place addition
print(result)
```

torch.randn_like … Returns a tensor with the same size as input that is filled with random numbers from a normal distribution with mean 0 and variance 1.

torch.empty … Returns a tensor filled with uninitialized data. The shape is defined by the variable argument size

```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
tensor([[ 0.1117,  0.6287, -0.8823],
        [-0.5409, -0.6151,  1.5826]])
torch.Size([2, 3])
tensor([[1.1117, 2.6287, 2.1177],
        [3.4591, 4.3849, 7.5826]])
tensor([[ 2.1117,  4.6287,  5.1177],
        [ 7.4591,  9.3849, 13.5826]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

```
import torch

x = torch.zeros([3, 5])
print(x)
y = torch.ones([3, 5])
print(y)
c = torch.add(x, y) + 10
print(c)
d = x+y +10
print(d)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
tensor([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]])
tensor([[11., 11., 11., 11., 11.],
        [11., 11., 11., 11., 11.],
        [11., 11., 11., 11., 11.]])
tensor([[11., 11., 11., 11., 11.],
        [11., 11., 11., 11., 11.],
        [11., 11., 11., 11., 11.]])
```

```
import numpy
import torch

x = numpy.zeros([3, 5])
print(x)
y = numpy.ones([3, 5])
print(y)
c = numpy.add(x, y) + 10
print(c)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
[[11. 11. 11. 11. 11.]
 [11. 11. 11. 11. 11.]
 [11. 11. 11. 11. 11.]]
```

```
# Vector + scalar
m1 = torch.FloatTensor([[1, 2]])
m2 = torch.FloatTensor([3]) # [3] -> [3, 3]
print(m1 + m2)
```

```
tensor([[4., 5.]])
```

```
# 2 x 1 Vector + 1 x 2 Vector
m1 = torch.FloatTensor([[1, 2]])
m2 = torch.FloatTensor([[3], [4]])
print(m1 + m2)
```

```
tensor([4., 5.],
       [5., 6.]])
```

COMMON TENSOR OPERATIONS

- Common tensor operations:
  - reshape
  - max/min
  - shape/size
  - etc
- Arithmetic operations
  - Abs / round / sqrt / pow /etc
  - torch.tensor’s support broadcasting
  - In-place operations

```
t = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
print(t.dim())  # rank. 즉, 차원
print(t.size()) # shape
print(t.shape)
```

```
2
torch.Size([2, 3])
torch.Size([2, 3])
```

SUM, MEAN, MAX - DIMENSION을 줄 수 있다.

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
print(t.mean())
print(t.mean(dim=0))
print(t.mean(dim=1))
print("")
print(t.sum())
print(t.sum(dim=1))
print("")
print(t.max())
print(t.max(dim=1))
```

(dim은 pyTorch에서의 axis와 같은 개념이다.)

```
tensor([[1., 2.],
        [3., 4.]])
tensor(2.5000)
tensor([2., 3.])
tensor([1.5000, 3.5000])

tensor(10.)
tensor([3., 7.])

tensor(4.)
torch.return_types.max(
values=tensor([2., 4.]),
indices=tensor([1, 1]))
```

MULTIPLICATION – ELEMENT WISE

```
b = torch.ones(2, 3)
print(b)
print(x*b)
print(torch.mul(x, b))
print(x.mul(b))
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[1., 2., 3.],
        [4., 5., 6.]])
tensor([[1., 2., 3.],
        [4., 5., 6.]])
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

VIEW

#View - tensor의 크기(size), 모양 (shape) 변경

```
y = x.view(6)
z = x.view(2, -1)
print(y)
print(z)
print(z.size())
```

```
tensor([1., 2., 3., 4., 5., 6.])
tensor([[1., 2., 3.],
        [4., 5., 6.]])
torch.Size([2, 3])
```

Tensor.view(*shape) → Tensor
Returns a new tensor with the same data as the self tensor but of a different shape.

The returned tensor shares the same data and must have the same number of elements, but may have a different size.

SQUEEZE, UNSQUEEZE

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

```
#Squeeze 차원을 축소
t1 = torch.rand(1, 2, 3)
print(t1.shape)
t2 = t1.squeeze()
print(t2, t2.shape)
```

torch.Size([1, 2, 3])
tensor([[0.4694, 0.4921, 0.6644],
[0.9148, 0.9279, 0.3639]]) torch.Size([2, 3])

torch.squeeze (Python function, in torch.squeeze)
torch.Tensor.squeeze (Python method, in torch.Tensor.squeeze)
torch.squeeze
torch.squeeze torch.squeeze(input, dim=None, *, out=None) → Tensor Returns a tensor with all the dimensions of input of size 1 removed. For example, if input is of shape: (A×1×B×C×1×D)(A \times 1 \times B...
torch.Tensor.squeeze
torch.Tensor.squeeze Tensor.squeeze(dim=None) → Tensor
See torch.squeeze()...

Returns a tensor with all the dimensions of input of size 1 removed.

For example, if input is of shape: (A \times 1 \times B \times C \times 1 \times D)(A×1×B×C×1×D) then the out tensor will be of shape: (A \times B \times C \times D)(A×B×C×D)

The returned tensor shares the storage with the input tensor, so changing the contents of one will change the contents of the other.

```
>>> x = torch.tensor([1, 2, 3, 4])
>>> torch.unsqueeze(x, 0)
tensor([[ 1,  2,  3,  4]])
>>> torch.unsqueeze(x, 1)
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4]])
```

torch.unsqueeze(input, dim) → Tensor
Returns a new tensor with a dimension of size one inserted at the specified position.

The returned tensor shares the same underlying data with this tensor.

A dim value within the range [-input.dim() - 1, input.dim() + 1) can be used. Negative dim will correspond to unsqueeze() applied at dim = dim + input.dim() + 1.

STACK/CAT

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

```
#stack  (Numpy와 비슷)
y1 = torch.FloatTensor([1, 2])
y2 = torch.FloatTensor([3, 4])
y3 = torch.FloatTensor([5, 6])
y4 = torch.stack([y1, y2, y3])
print(y4)
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
```

Stack: Concatenates a sequence of tensors along a new dimension.

```
#cat - tensor 를 결합하는 method (concatenate)
#Numpy의 stack과 유사하지만 쌓을 dim 이 존재해야 함. 해당 차원을 늘린 후 결합
y5 = torch.cat((y1, y2, y3), dim=0)
print(y5)
print(y5.size())
```

```
tensor([1., 2., 3., 4., 5., 6.])
torch.Size([6])
```

torch.cat(tensors, dim=0, *, out=None) → Tensor
Concatenates the given sequence of seq tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty.

CHUNK

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

#Chunk - tensor를 나눌 때 사용. 갯수를 지정

```
tensor = torch.rand(3, 6)
t1, t2, t3 = torch.chunk(tensor,3, dim=1)
print(tensor)
print(t1)
print(t3)
```

```
tensor([[0.9330, 0.4597, 0.3355, 0.1319, 0.7919, 0.2339],
        [0.7367, 0.2848, 0.0598, 0.7489, 0.4316, 0.5963],
        [0.9323, 0.3707, 0.4433, 0.0286, 0.3752, 0.8199]])
tensor([[0.9330, 0.4597],
        [0.7367, 0.2848],
        [0.9323, 0.3707]])
tensor([[0.7919, 0.2339],
        [0.4316, 0.5963],
        [0.3752, 0.8199]])
```

SPLIT

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

```
#Split - chunk와 비슷하지만 하나의 tensor당 크기를 줌
tensor = torch.rand(3, 6)
t1, t2 = torch.split(tensor,3, dim=1)
print(tensor)
print(t1)
print(t2)
```

```
tensor([[0.4040, 0.6738, 0.3201, 0.2434, 0.1075, 0.5485],
        [0.2889, 0.0171, 0.0201, 0.1304, 0.4529, 0.2064],
        [0.4438, 0.4910, 0.7083, 0.7237, 0.2939, 0.9327]])
tensor([[0.4040, 0.6738, 0.3201],
        [0.2889, 0.0171, 0.0201],
        [0.4438, 0.4910, 0.7083]])
tensor([[0.2434, 0.1075, 0.5485],
        [0.1304, 0.4529, 0.2064],
        [0.7237, 0.2939, 0.9327]])
```

torch.split(tensor, split_size_or_sections, dim=0)[SOURCE]
Splits the tensor into chunks. Each chunk is a view of the original tensor.

If split_size_or_sections is an integer type, then tensor will be split into equally sized chunks (if possible). Last chunk will be smaller if the tensor size along the given dimension dim is not divisible by split_size.

TORCH와 NUMPY간의 데이터변환


```
#Torch <-> Numpy
# Torch tensor를 Numpy array로 변환가능  numpy(), from_numpy()
#Tensor가 CPU상에 있다면 둘이 메모리를 공유(하나 변하면 다른 것도)
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
x_num = x.numpy()
print(x_num)
# x_num.add(1) <- not supported
print(x.add_(1))
print(x.numpy())
```

```
[[1 2 3]
 [4 5 6]]
tensor([[2, 3, 4],
        [5, 6, 7]])
[[2 3 4]
 [5 6 7]]
```

```
# Numpy data into tensor
import numpy as np
a = np.ones(6)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)    # 하나를 바꾸어도 다른 것 바뀐다.
```

```
[2. 2. 2. 2. 2. 2.]
tensor([2., 2., 2., 2., 2., 2.], dtype=torch.float64)
```

CUDA 와 DATA 옮기기

#.to method를 이용해서 CUDA 로 데이터를 옮긴다.

```
x = torch.tensor([[1, 2], [3, 4]])
print(x)
device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
y = torch.ones_like(x, device = device)
z = x.to(device)
print(y)
print(z)
```

```
tensor([[1, 2],
        [3, 4]])
tensor([[1, 1],
        [1, 1]])
tensor([[1, 2],
        [3, 4]])
```

```
# defining a PyTorch Tensor
tensor = torch.tensor([-12, -23, 0.0, 32,
                       1.32, 201, 5.02])
print("Tensor:\n", tensor)

# sorting the tensor in ascending order
print("Sorting tensor in ascending order:")
values, indices = torch.sort(tensor)

# printing values of sorted tensor
print("Sorted values:\n", values)

# printing indices of sorted value
print("Indices:\n", indices)

# sorting the tensor in  descending order
print("Sorting tensor in  descending order:")
values, indices = torch.sort(tensor, descending=True)

# printing values of sorted tensor
print("Sorted values:\n", values)

# printing indices of sorted value
print("Indices:\n", indices)
```

torch.sort(input, dim=- 1, descending=False, stable=False, *, out=None)

A namedtuple of (values, indices) is returned, where the *values* are the sorted values and *indices* are the indices of the elements in the original *input* tensor.

```
Tensor:
 tensor([-12.0000, -23.0000,   0.0000,  32.0000,   1.3200,
201.0000,   5.0200])
Sorting tensor in ascending order:
Sorted values:
 tensor([-23.0000, -12.0000,   0.0000,   1.3200,   5.0200,
32.0000, 201.0000])
Indices:
 tensor([1, 0, 2, 4, 6, 3, 5])
Sorting tensor in  descending order:
Sorted values:
 tensor([201.0000,  32.0000,   5.0200,   1.3200,   0.0000, -
12.0000, -23.0000])
Indices:
 tensor([5, 3, 6, 4, 2, 0, 1])
```

```
import numpy
import time
import random
import array
n_d = 1024*1024
A = np.random.rand(n_d)
C = A.tolist()

t1 = time.time()
for i in range(len(C)):
   C[i] = 9.0/5.0*C[i] + 32.0

t2 = time.time()
print('t2-t1=', 1000*(t2-t1))

t3 = time.time()
A = A*9.0/5.0+32.0
t4 = time.time()
print('t4-t3=', 1000*(t4-t3))
```

```
t2-t1= 204.5304775238037
t4-t3= 9.982824325561523
```

```
n_d = 1024*1024
A = np.random.rand(n_d)
C = A.tolist()
# print(A)

t1 = time.time()
for i in range(len(C)):
   if C[i] > 0.5:
      C[i]= 1.0
   else:
      C[i]=  0.0
t2 = time.time()
print('t2-t1=', 1000*(t2-t1))

t3 = time.time()
A[A<0.5] = 0
A[A>= 0.5]= 1
t4 = time.time()
print('t4-t3=', 1000*(t4-t3))
```

```
t2-t1= 228.33490371704102
t4-t3= 28.885602951049805
```

Run time difference Between Python and Numpy

```
import numpy
import time
import random
import torch

print('hello tensor')
n_d = 1024*1024
A = np.random.rand(n_d)
t = torch.tensor(A, device = 'cpu', requires_grad=False, dtype =
torch.float32)

t3 = time.time()
t = t*1.8 +32.0
t4 = time.time()
print('t4-t3=', 1000*(t4-t3))

hello tensor
t4-t3= 4.986047744750977
```

```
import numpy
import time
import random
import torch
print('hello tensor')
n_d = 6
A = np.random.rand(n_d)
t = torch.tensor(A, device = 'cpu', requires_grad=False, dtype =
torch.float32)

t3 = time.time()
t = t*1.8 +32.0
t4 = time.time()
print('t4-t3=', 1000*(t4-t3))
if n_d < 10:
    print(t)  #test code
    print(A)
hello tensor
t4-t3= 1.9960403442382812
tensor([33.5366, 32.8331, 32.8240, 32.4005, 32.3297, 32.5646])
[0.85367155 0.46283438 0.45777846 0.22251492 0.18316694
0.31364171]
```

TORCH.MATMUL - 보편적인 MATRIX MULT.

- torch.dot(a, b) --- a, b (1-d) vector 의 inner product.
  Numpy의 dot()처럼 자유롭지 않다.
- torch.mm - [n, m] x [m,p] = [n, p]
- Torch.bmm – [B, n, m]x[B, m, p] = [B, n, p]
- torch.matmul(a,b) 를 가장 자유롭게 사용한다.
  - Vector x vector
  - (2d) matrix x vector
  - (3d) Batched matrix x (1d) broadcasted vector
  - (3d) Batched matrix x (2d) broadcasted matrix
  - (3d) Batched matrix x (3d) batched matrix

BATCH PROCESSING

- GPU에서 계산할 때 GPU에 딸린 DRAM에서 data나 parameter를 읽어오는 시간이 많이 걸린다. 특히 deep neural networ은 parameter size가 엄청 큰 경우가 많다.
- Image recognition 시, image 한장 가져오고, parameter 한셋 가져와서 한 장의 인식을 반복
- Image 100장 가져와서, parameter 한 셋 가져와서, 100장의 인식을 한번에
  - Parameter를 가져오는 cost는 1/100로 준다.
  - 내부의 중간결과를 저장하는 memory size는 100배로 는다.
  - 많은 경우 상당한 speed-up을 올릴 수 있다.
  - DNN 훈련의 경우 batch size를 키우면 훈련시간이 짧아지지만, 훈련 중에는 gradient 계산을 위한 중간결과 저장이 많기 때문에 DRAM size 가 부족해진다.

BATCH PROCESSING(2)

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

Case 1:

(3d) Batched matrix x (2d) broadcasted matrix

Input image(2D, NxM)를 B장 처리하는데 model 은 하나를 공유한다.

➢(BxNxM)x(MxP) -> (BxNxP)

(3d) Batched matrix x (3d) batched matrix

- Input image(2D, NxM)를 B장 처리하는데 model 도 각자이다. 모델로 B 개이다.
  - (BxNxM)x(BxMxP) -> (BxNxP)

```
#matrix multiplication with torch.mm
t1 = torch.ones(3, 2)
t2 = torch.randn(2, 3)
t3 = torch.mm(t1, t2)
print(t1)
print(t2)
print(t3)
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
tensor([[-1.5497, -0.2812, -1.4817],
        [ 1.8720,  0.0073, -2.2898]])
tensor([[ 0.3223, -0.2739, -3.7715],
        [ 0.3223, -0.2739, -3.7715],
        [ 0.3223, -0.2739, -3.7715]])
```

```
#batch matrix multiplication with torch.bmm
# batched matrix x batched matrix
tensor1 = torch.ones(2, 3, 4)
tensor2 = torch.randn(2, 4, 5)

t = torch.bmm(tensor1, tensor2)
print(t.size(), t)
```

```
torch.Size([2, 3, 5]) tensor([[[ 1.8015, -0.3524,  1.9060,  3.4736,
0.0603],
        [ 1.8015, -0.3524,  1.9060,  3.4736,  0.0603],
        [ 1.8015, -0.3524,  1.9060,  3.4736,  0.0603]],

       [[-0.8835, -0.2584,  1.8103,  2.1970, -0.6962],
        [-0.8835, -0.2584,  1.8103,  2.1970, -0.6962],
        [-0.8835, -0.2584,  1.8103,  2.1970, -0.6962]]])
```

```
import numpy as np
import torch

# vector x vector
tensor1 = torch.ones(3)
tensor2 = torch.randn(3)
print(tensor1, tensor2)
t = torch.matmul(tensor1, tensor2)
print(t, t.size())
t1 = torch.dot(tensor1, tensor2)
print(t1, t1.size())
```

```
tensor([1., 1., 1.]) tensor([1.2402, 0.1121, 0.2668])
tensor(1.6192) torch.Size([])
tensor(1.6192) torch.Size([])
```

```
# matrix x vector
tensor1 = torch.ones(3, 4)
tensor2 = torch.randn(4)
print(tensor1, tensor2)
t = torch.matmul(tensor1, tensor2)
print(t, t.size())
```

```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]) tensor([-1.2745,  0.3913, -0.1453, -0.7014])
tensor([-1.7300, -1.7300, -1.7300]) torch.Size([3])
```

```
# batched matrix x broadcasted vector
tensor1 = torch.ones(2, 3, 4)
tensor2 = torch.tensor([1., 0., 2., -1.])
print(tensor1, tensor2)
t = torch.matmul(tensor1, tensor2)
print(t, t.size())
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]]) tensor([ 1.,  0.,  2., -1.])
tensor([[2., 2., 2.],
        [2., 2., 2.]]) torch.Size([2, 3])
```

```
# batched matrix x broadcasted matrix
tensor1 = torch.randn(4, 3, 2)
tensor2 = torch.tensor([[1.], [-1.]])  #(2.1)
dimension
t = torch.matmul(tensor1, tensor2)
print(t, t.size())
```

```
tensor([[[-0.7888],
         [ 2.0835],
         [ 0.3500]],

        [[-0.7603],
         [ 1.7824],
         [-0.4314]],

        [[ 0.7845],
         [ 0.3543],
         [-0.7257]],

        [[ 0.7852],
         [ 0.3909],
         [-0.7192]]]) torch.Size([4, 3, 1])
```

```
# batched matrix x batched matrix
tensor1 = torch.randn(2, 3, 4)
tensor2 = torch.randn(2, 4, 5)
print(tensor1, tensor2)
t = torch.matmul(tensor1, tensor2)
print(t, t.size())
```

```
tensor([[[ 1.1631, -1.1644, -0.5849,  0.1162],
         [ 1.1636,  1.7263,  0.3073,  1.0364],
         [-2.1449, -0.0396,  0.9080, -0.0368]],

        [[ 0.6111, -1.1104, -0.1103, -1.2444],
         [ 0.4317, -0.8696,  0.5541,  0.4554],
         [ 0.9930, -0.3079, -1.3046, -0.2256]]]) tensor([[[ 1.5313, -
0.3276,  2.7772, -0.4481,  0.1965],
         [ 1.5952, -0.0919,  1.1515, -0.3580, -1.6283],
         [-0.4292, -1.2339,  0.6714, -0.7756, -1.6151],
         [ 1.2285, -0.2973,  1.0524,  1.2427,  0.0254]],

        [[-0.7885, -0.3905, -1.6923,  1.9778,  0.0136],
         [-2.1463, -0.4651, -0.5532, -0.7155, -0.7816],
         [-0.2973, -0.4352,  1.3212, -0.2860, -2.0298],
         [-1.3526, -0.4848,  0.7136, -0.5906, -0.7557]]])
tensor([[[ 0.3174,  0.4132,  1.6188,  0.4937,  3.0720],
         [ 5.6770, -1.2271,  6.5166, -0.0897, -3.0522],
         [-3.7826, -0.4032, -5.4315,  0.2252, -1.8244]],

        [[ 3.6173,  0.9290, -1.4537,  2.7697,  2.0405],
         [ 0.7453, -0.2261,  0.8075,  1.0487, -0.7834],
         [ 0.5709,  0.4325, -3.3947,  2.6907,  3.0726]]]) torch.Size([2,
3, 5])
```

TORCH TENSOR SUMMARY

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- Very similar to numpy (indexing, main functions)
- Every tensor has a device, a type, and a required_grad attribute
  Conversion and/or device transfer might be needed.
- In-place operations end in underscore, e.g. .fill_()
- Some operations create new tensors, some share data. Careful with the broadcasting semantics.
- Remark: not just tensors are similar to ndarrays, but torch functions are also similar to numpy functions.

3. COMPUTATIONAL GRAPH AND AUTOGRAD

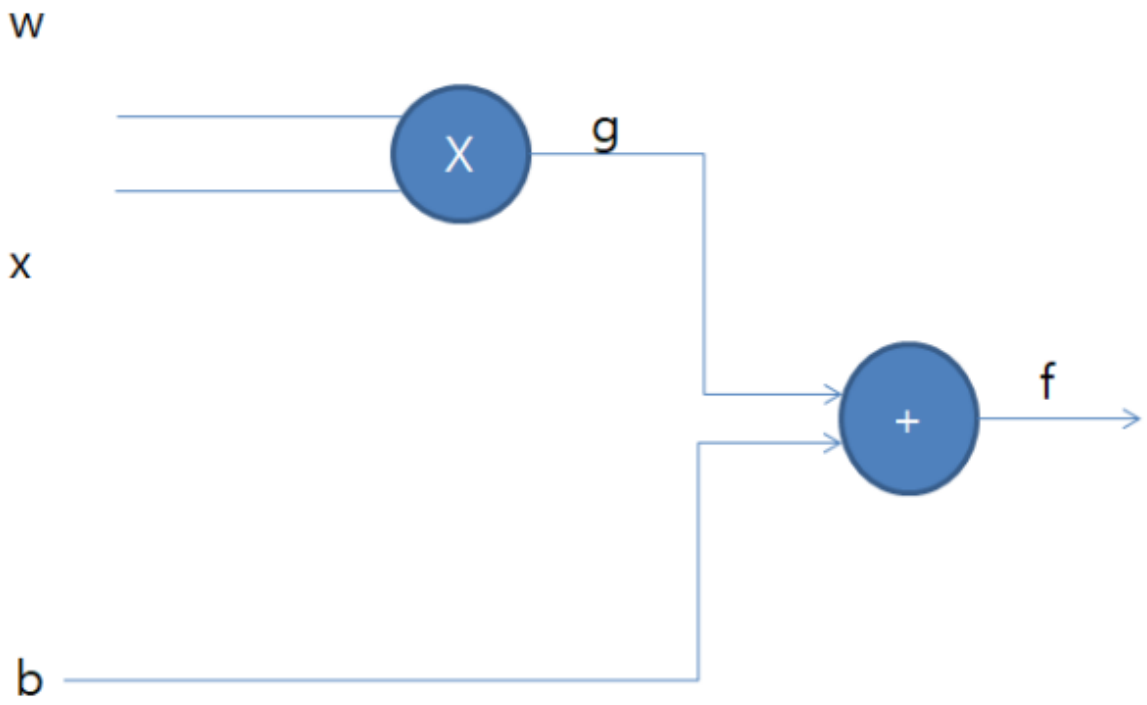
인공신경망의 훈련

- DNN은 내부에 매우 많은 parameter (weight 라 함)를 가지고 있다.
- 초기에 일정크기의 random number 로 초기화된다.
- Supervised learning 에서는 입력 data를 넣었을 때 해당 weight 로 얻은 결과가 desired value 와 다른 값인 cost 를 구한다.
- 이 cost 값이 작아지도록 weight 를 조금씩 바꾼다.
- Weight (W) 를 바꾸려면 derivate cost/derivate weight 의 값을 알아야 한다.
- 문제 – 신경망의 깊이가 깊은 것.

$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

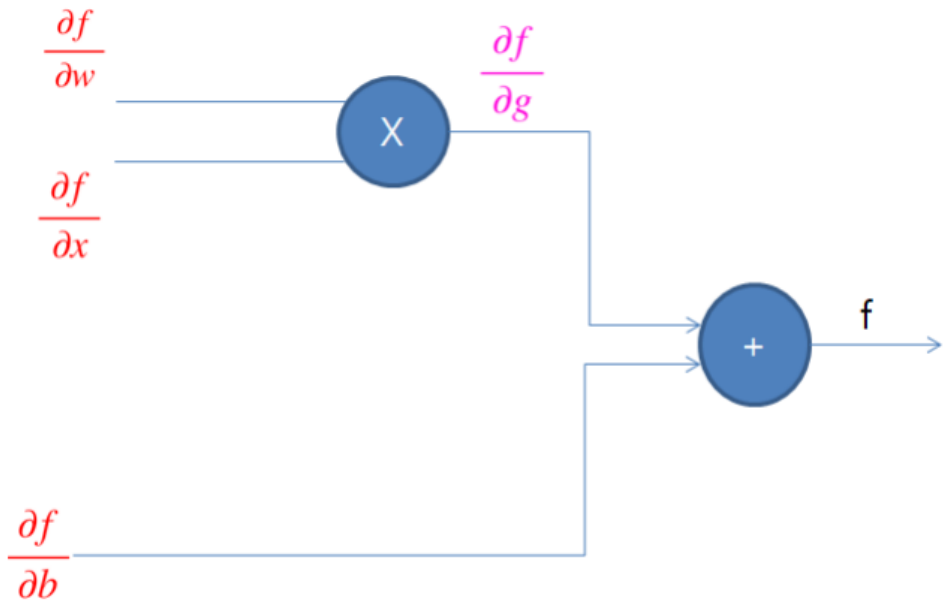
음과 같은 신경망(뉴럴네트워크, Neural Network)을 생각해보자.

$$f = wx + b, \qquad g = wx, \qquad f = g + b$$



wx+b 라는 함수가 있을 때, 위 그림처럼 신경망을 그려볼 수 있다. 이 때, g=wx라는 또 다른 함수 만들어보면 f=g+b라는 복합함수로 생각해볼 수 있다.

신경망을 학습시키기 위해서는 각 w, x, b가 결과값인 f에 미치는 영향도를 알아야 한다. 즉, f라는 과값을 각 입력 값인 w, x, b 값으로 미분한 값을 알아야 한다.



다시 말해서 위와 같은 신경망에서 결과값 f가 있을 때, 입력값 w, x, b에 대한 편미분 값을 알아야 한다. 이 때, 위에서 봤던 Chain Rule을 이용해서 w, x로 편미분한 값을 얻어올 수 있다.

위 식에서 각각 입력에 대한 편미분은 다음과 같이 구할 수 있다.

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} = 1 \times w = w$$

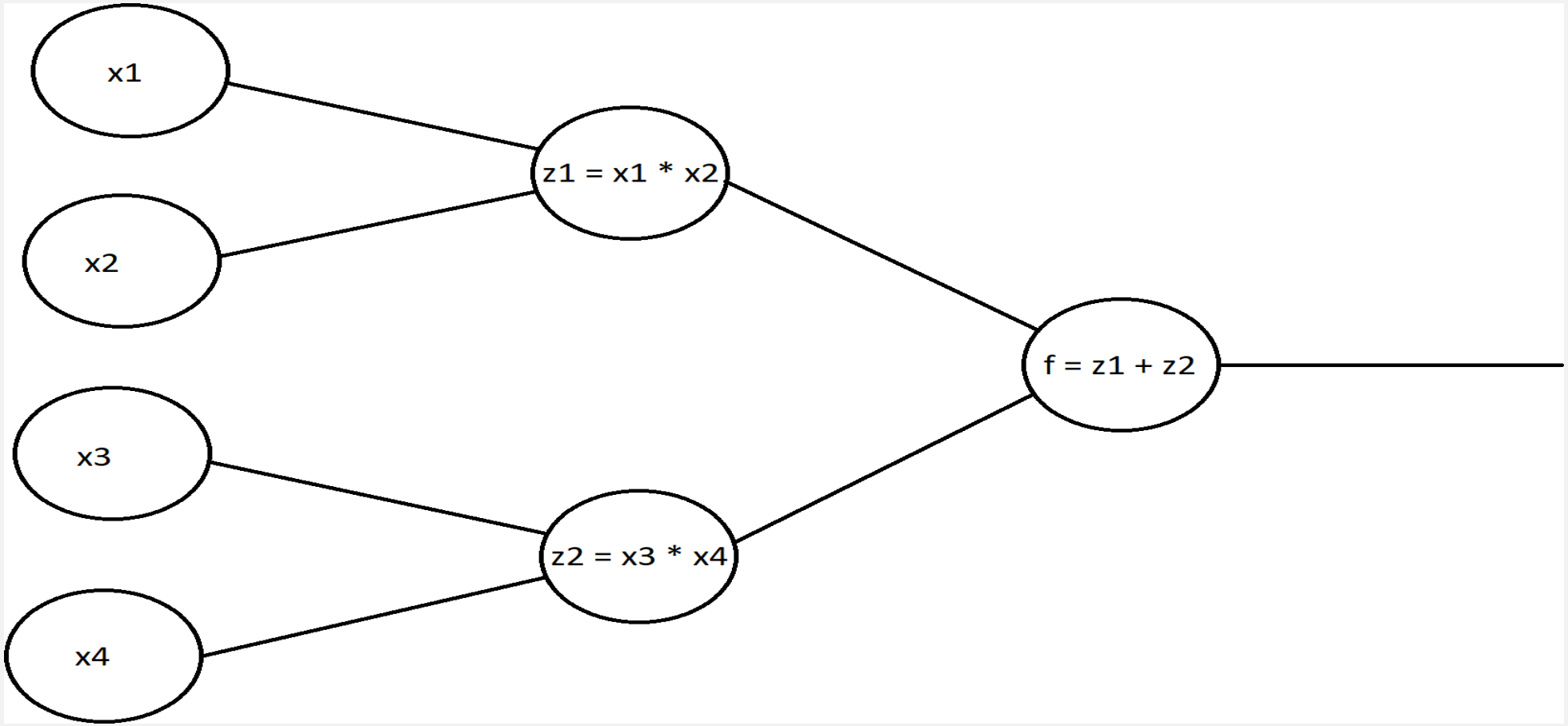
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} = 1 \times x = x$$

$$\frac{\partial f}{\partial b} = 1$$

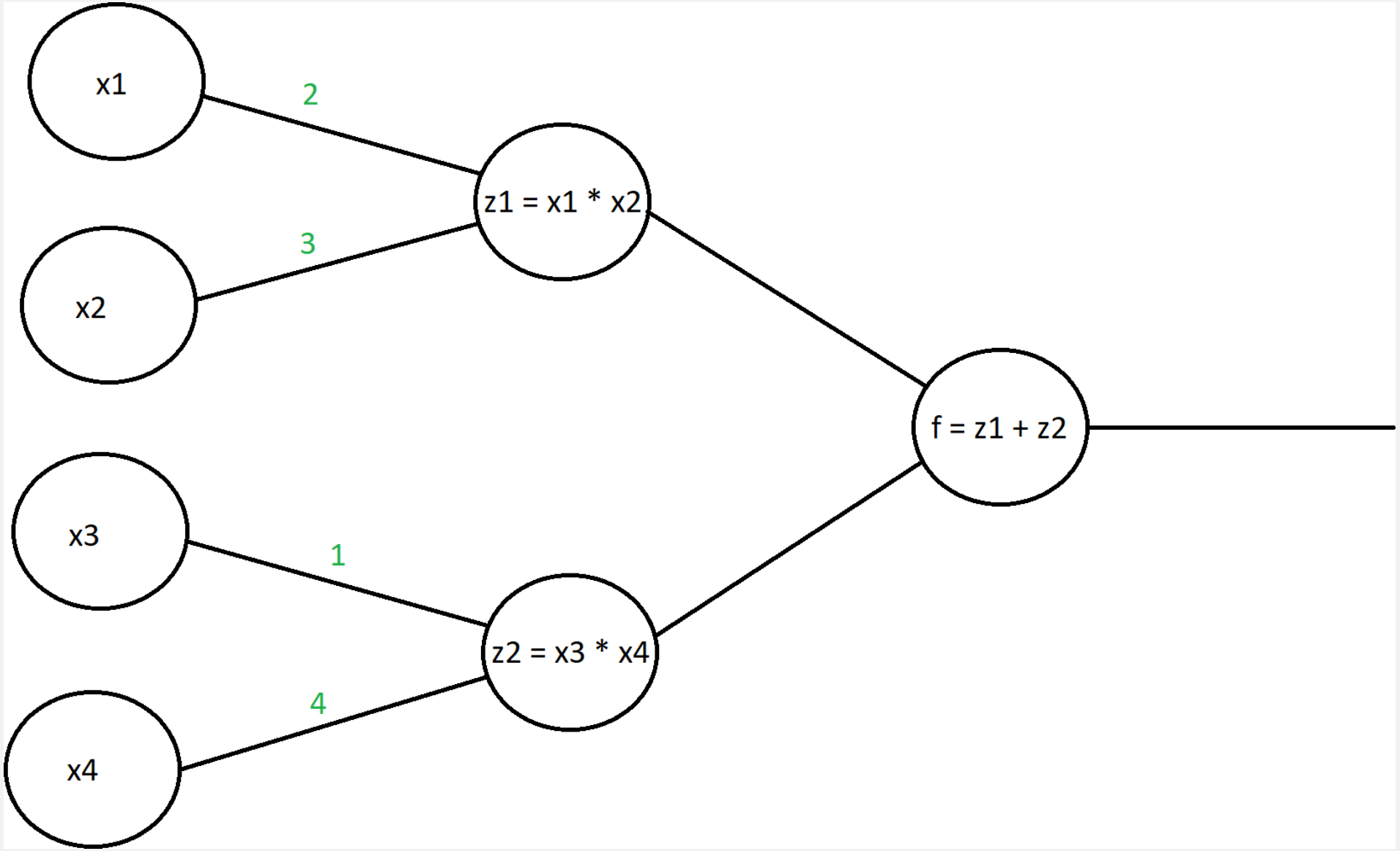
COMPUTATIONAL GRAPH

$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$



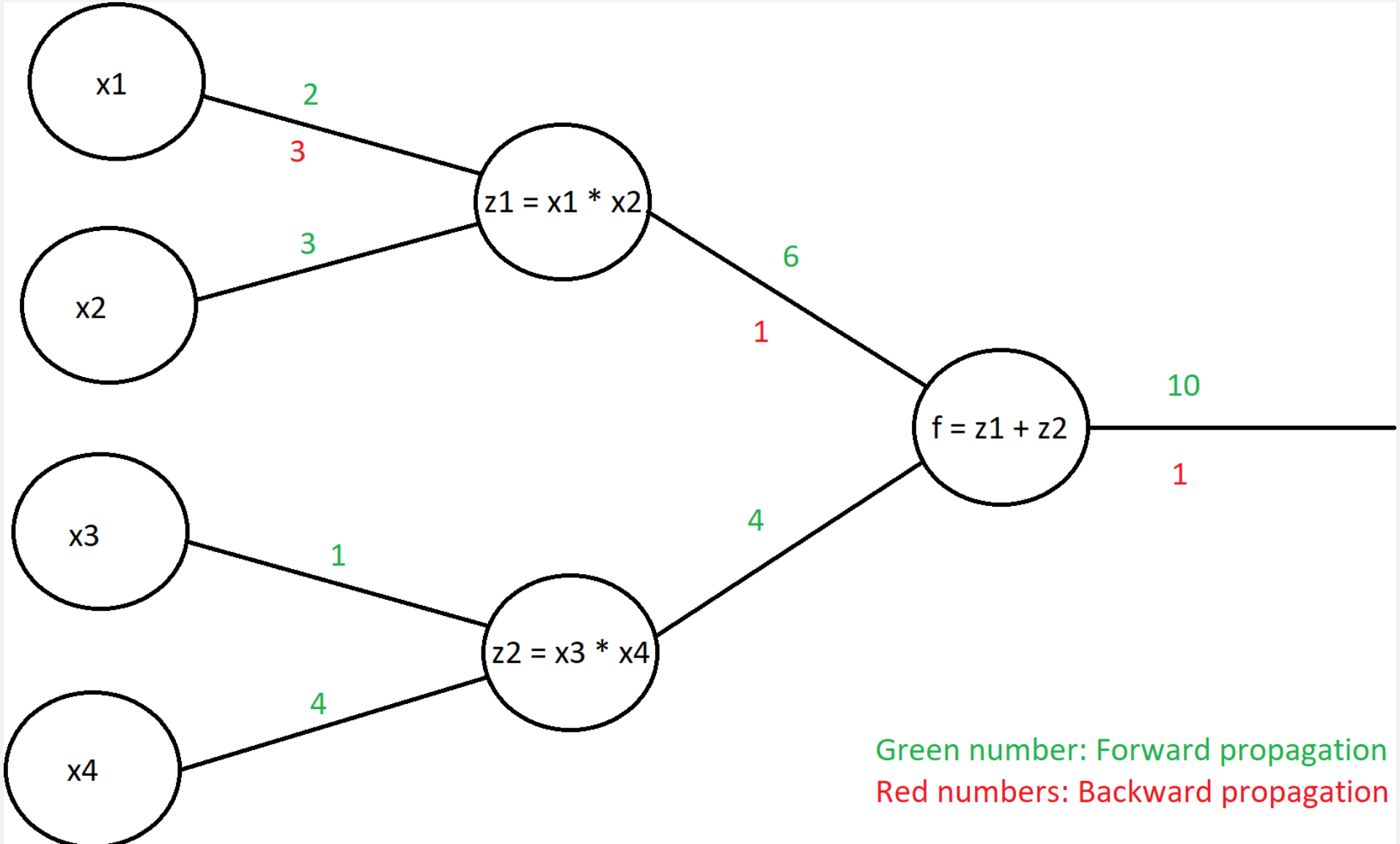
FORWARD PROPAGATION



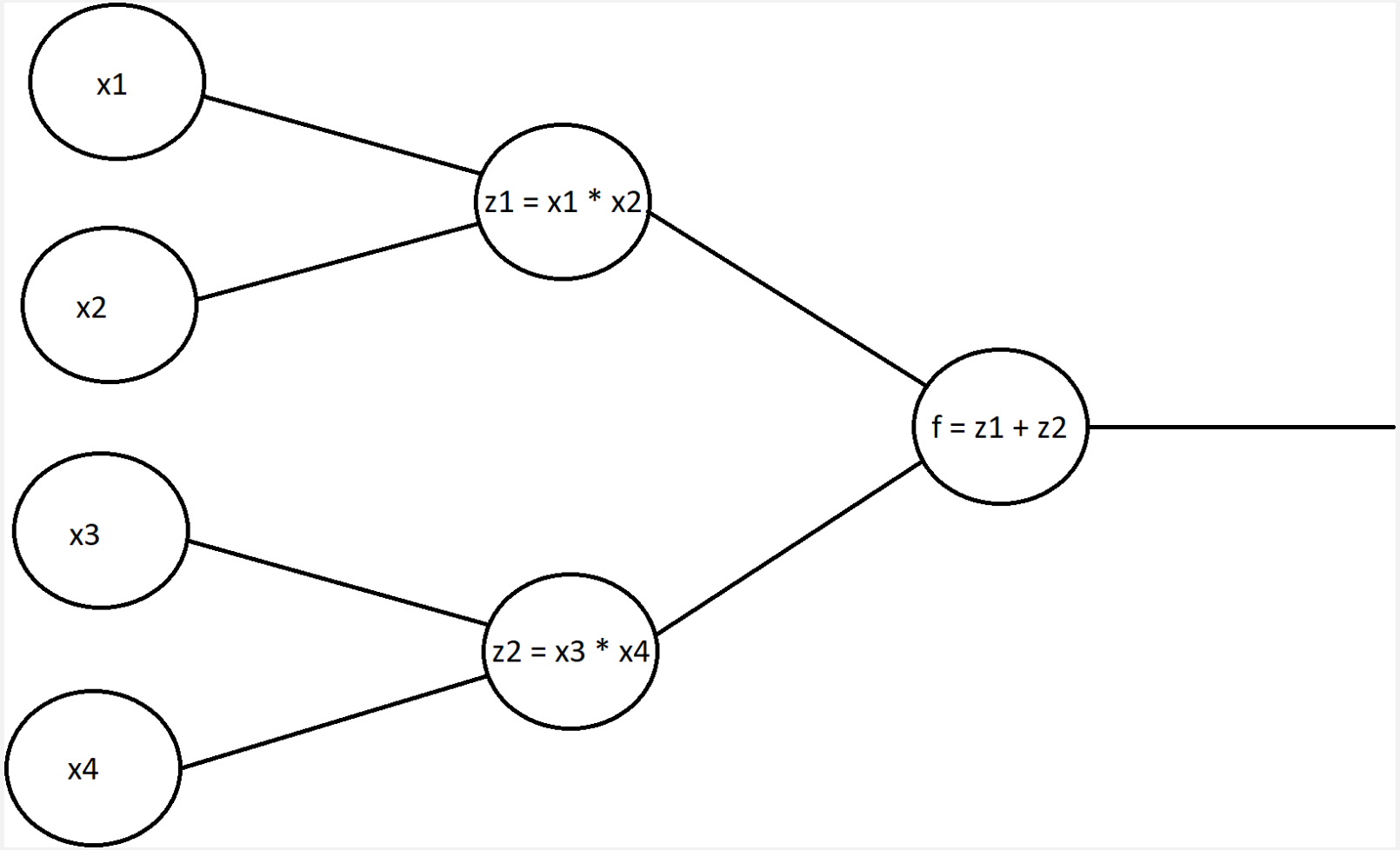
Backward propagation

What if we want to get the derivative of *f* with respect to the *x1*?

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$ $f(\vec{x}) = z_1 + z_2$ $\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$



- Autograd - Automatic differentiation for all operations on Tensors Static computatio
  - nal graph (TensorFlow 1.0)
  - Dynamic computational graph (PyTorch) – 매 iteration마다 바뀔 수 있다.
- The backward graph is defined by the forward run!



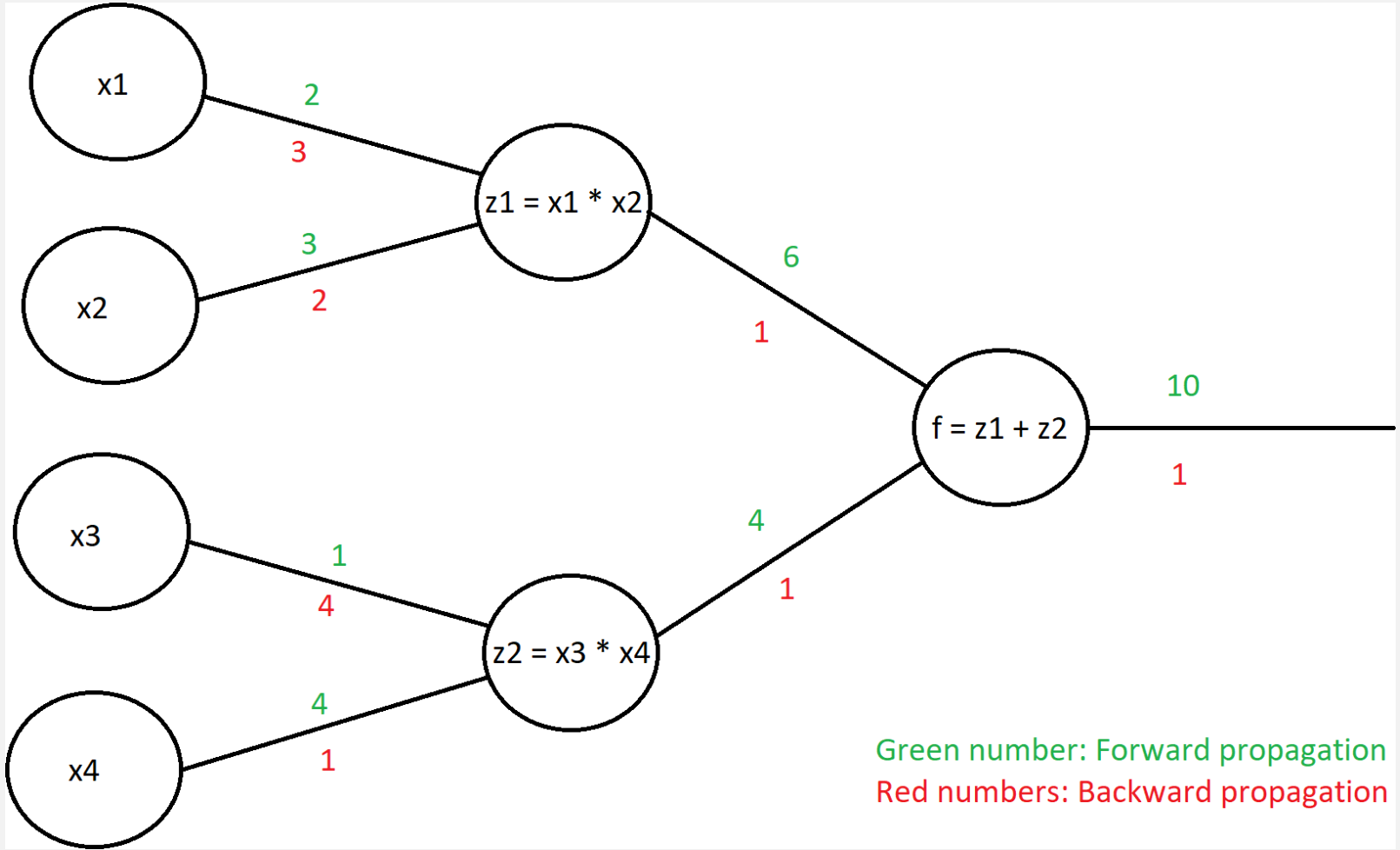
EXAMPLE OF AUTOGRAD

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

```
>>> import torch
>>> from torch import autograd
>>> x1 = torch.tensor(2, requires_grad=True,    dtype=torch.float32)
>>> x2 = torch.tensor(3, requires_grad=True,    dtype=torch.float32)
>>> x3 = torch.tensor(1, requires_grad=True,    dtype=torch.float32)
>>> x4 = torch.tensor(4, requires_grad=True,    dtype=torch.float32)
>>> # Forward propagation
>>> z1 = x1 * x2
>>> z2 = x3 * x4
>>> f = z1 + z2
```

```
>>> df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4])
>>> df_dx
(tensor(3.), tensor(2.), tensor(4.), tensor(1.))
```

```
>>> df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4])
>>> df_dx
(tensor(3.), tensor(2.), tensor(4.), tensor(1.))
```



LEAF TENSOR

A «leaf tensor» is a tensor you created directly, not as the result of an operation.

```
>>> x = torch.tensor(2)   # x is a leaf tensor
>>> y = x + 1             # y is not a leaf tensor
```

Remember the computation graph:
x1, x2, x3, x4 are the leaf tensors.

The need for specifying all tensors is inconvenient.

```
>>> import torch
>>> from torch import autograd
>>> x1 = torch.tensor(2, requires_grad=True,      dtype=torch.float32)
>>> x2 = torch.tensor(3, requires_grad=True,      dtype=torch.float32)
>>> x3 = torch.tensor(1, requires_grad=True,      dtype=torch.float32)
>>> x4 = torch.tensor(4, requires_grad=True,      dtype=torch.float32)
>>> # Forward propagation
>>> z1 = x1 * x2
>>> z2 = x3 * x4
>>> f = z1 + z2
>>> #  df_dx = grad(outputs=f, inputs = [x1,    x2, x3, x4]) # inconvenient
>>> f.backward()                                             # that is better!

>>> print(f" f's derivative w.r.t. x1 is {x.grad}")
tensor(3.)
```

Chain rule is applied back to all the leaf tensors with *requires_grad=True* attribute.

- We can locally disable/enable gradient calculation with
  - torch.no_grad()
  - torch.enable_grad()
- or using the @torch.no_grad @torch.enable_grad decorators

```
>>> x = torch.tensor([1], requires_grad=True)
>>> with torch.no_grad():
...   y = x * 2
>>> y.requires_grad
False

>>> with torch.no_grad():
...   with torch.enable_grad():
...     y = x * 2
>>> y.requires_grad
True
```

Note: Use «torch.no_grad()» during inference

- https://www.youtube.com/watch?v=MswxJw-8PvE
- https://pytorch.org/docs/stable/autograd.html

LOADING DATA, DEVICES AND CUDA

서울대학교 SEOUL NATIONAL UNIVERSITY

Numpy arrays to PyTorch tensors

- `torch.from_numpy(x_train)`
- `Returns a cpu tensor!`

PyTorch tensor to numpy

- `t.numpy()`

Using GPU acceleration

- `t.to()`
- Sends to whatever device (cuda or cpu)

Fallback to cpu if gpu is unavailable:

- `torch.cuda.is_available()`

Check cpu/gpu tensor OR numpy array ?

- `type(t)` or `t.type()` returns
  - numpy.ndarray
  - torch.Tensor
    - CPU - torch.cpu.FloatTensor
    - GPU - torch.cuda.FloatTensor

AUTOGRAD

- Automatic Differentiation Package
- Don’t need to worry about partial differentiation, chain rule etc.
  - `backward()` does that
- Gradients are accumulated for each step by default:
  - Need to zero out gradients after each update
  - `tensor.grad_zero()`

```
# Create tensors.
x = torch.tensor(1., requires_grad=True)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)

# Build a computational graph.
y = w * x + b    # y = 2 * x + 3

# Compute gradients.
y.backward()

# Print out the gradients.
print(x.grad)    # x.grad = 2
print(w.grad)    # w.grad = 1
print(b.grad)    # b.grad = 1
```

OPTIMIZER AND LOSS

Optimizer

- Adam, SGD etc.
- An optimizer takes the parameters we want to update, the learning rate we want to use along with other hy per-parameters and performs the u pdates

Loss

- Various predefined loss functions to choose from
- L1, MSE, Cross Entropy

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)


# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

print(a, b)
```

MODEL

In PyTorch, a model is represented by a regular Python class that inherits from the Module class.

- Two components
  - `__init__(self)` : it defines the parts that make up the model- in our case, two parameters, a and b
  - `forward(self, x)` : it performs the actual computation, that is, it outputs a prediction, given the inputx

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```

파이썬 클래스 상속 : super().__init__()