In [1]:
```
%matplotlib inline
```

파이토치(PyTorch) 기본 익히기 || 빠른 시작 || **텐서(Tensor)** || Dataset과 Dataloader || 변형 (Transform) || 신경망 모델 구성하기 || Autograd || 최적화(Optimization) || 모델 저장하고 불러오기

텐서(Tensor)

텐서(tensor)는 배열(array)이나 행렬(matrix)과 매우 유사한 특수한 자료구조입니다. PyTorch에서는 텐서를 사용하여 모델의 입력(input)과 출력(output), 그리고 모델의 매개변수들을 부호화(encode)합니다.

텐서는 GPU나 다른 하드웨어 가속기에서 실행할 수 있다는 점만 제외하면 NumPy 의 ndarray와 유사합니다. 실제로 텐서와 NumPy 배열(array)은 종종 동일한 내부(underly) 메모리를 공유할 수 있어 데이터를 복수할 필요가 없습니다. (`bridge-to-np-label` 참고) 텐서는 또한 (Autograd_ 장에서 살펴볼) 자동 미분(automatic differentiation)에 최적화되어 있습니다. ndarray에 익숙하다면 Tensor API를 바로 사용할 수 있을 것입니다. 아니라면, 아래 내용을 함께 보시죠!

In [2]:
```
import torch
import numpy as np
```

텐서(tensor) 초기화

텐서는 여러가지 방법으로 초기화할 수 있습니다. 다음 예를 살펴보세요:

데이터로부터 직접(directly) 생성하기

데이터로부터 직접 텐서를 생성할 수 있습니다. 데이터의 자료형(data type)은 자동으로 유추합니다.

In [19]:
```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

#  type(data), type(x_data)
x_data # 이곳 변형
```

Out[19]:
```
tensor([[1, 2],
        [3, 4]])
```

NumPy 배열로부터 생성하기

텐서는 NumPy 배열로 생성할 수 있습니다. (그 반대도 가능합니다 - `bridge-to-np-label` 참고)

In [20]:
```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
type(np_array), type(x_np), type(x_np.numpy())
```

Out[20]:
```
(numpy.ndarray, torch.Tensor, numpy.ndarray)
```

다른 텐서로부터 생성하기:

명시적으로 재정의(override)하지 않는다면, 인자로 주어진 텐서의 속성(모양(shape), 자료형(datatype))을 유지합니다.

In [22]:
```
x_ones = torch.ones_like(x_data) # x_data의 속성을 유지합니다.
print(f"Ones Tensor: \n {x_ones} \n");


x_rand = torch.rand_like(x_data, dtype=torch.float) # x_data의 속성을 덮어씁니
print(f"Random Tensor: \n {x_rand} \n");
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.3491, 0.3389],
        [0.4118, 0.4157]])
```

In [6]:
```
np.ones(x_data.shape), np.ones_like(x_data)
```

Out[6]:
```
(array([[1., 1.],
        [1., 1.]]),
 array([[1, 1],
        [1, 1]]))
```

무작위(random) 또는 상수(constant) 값을 사용하기:

`shape` 은 텐서의 차원(dimension)을 나타내는 튜플(tuple)로, 아래 함수들에서는 출력 텐서의 차원을 결정합니다.

In [30]:
```
torch.manual_seed(1); # 이곳 변형, 나중에 시드넘버를 맞춰야 할수도 있으므로

shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
rand_tensor.shape, ones_tensor.shape, zeros_tensor.shape
```

```
Random Tensor:
 tensor([[0.7576, 0.2793, 0.4031],
        [0.7347, 0.0293, 0.7999]])

Ones Tensor:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])

Zeros Tensor:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

Out[30]:
```
(torch.Size([2, 3]), torch.Size([2, 3]), torch.Size([2, 3]))
```

텐서의 속성(Attribute)

텐서의 속성은 텐서의 모양(shape), 자료형(datatype) 및 어느 장치에 저장되는지를 나타냅니다.

In [28]:
```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
print(tensor)
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
tensor([[0.8750, 0.5059, 0.2366, 0.7570],
        [0.2346, 0.6471, 0.3556, 0.4452],
        [0.0193, 0.2616, 0.7713, 0.3785]])
```

텐서 연산(Operation)

전치(transposing), 인덱싱(indexing), 슬라이싱(slicing), 수학 계산, 선형 대수, 임의 샘플링(random sampling) 등, 100가지 이상의 텐서 연산들을 여기_ 에서 확인할 수 있습니다.

각 연산들은 (일반적으로 CPU보다 빠른) GPU에서 실행할 수 있습니다. Colab을 사용한다면, Edit > Notebook Settings 에서 GPU를 할당할 수 있습니다.

기본적으로 텐서는 CPU에 생성됩니다. `.to` 메소드를 사용하면 (GPU의 가용성(availability)을 확인한 뒤) GPU로 텐서를 명시적으로 이동할 수 있습니다. 장치들 간에 큰 텐서들을 복사하는 것은 시간과 메모리 측면에서 비용이 많이든다는 것을 기억하세요!

In [9]:
```
torch.cuda.is_available()
```

Out[9]:
```
True
```

In [32]:
```
# GPU가 존재하면 텐서를 이동합니다
if torch.cuda.is_available():
    tensor = tensor.to("cuda")
```

In [11]:
```
tensor.device
```

Out[11]:
```
device(type='cuda', index=0)
```

목록에서 몇몇 연산들을 시도해보세요. NumPy API에 익숙하다면 Tensor API를 사용하는 것은 식은 죽 먹기라는 것을 알게 되실 겁니다.

NumPy식의 표준 인덱싱과 슬라이싱:

In [39]:
```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
```

```
print(f"First row's size: {tensor[0].size()}") # 이곳 수정
print(f"First row's shape: {tensor[0].shape}") # 이곳 수정

print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
print(tensor)
```

```
First row: tensor([1., 1., 1., 1.])
First row's size: torch.Size([4])
First row's shape: torch.Size([4])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

텐서 합치기 `torch.cat` 을 사용하여 주어진 차원에 따라 일련의 텐서를 연결할 수 있습니다.

`torch.cat` 과 미묘하게 다른 또 다른 텐서 결합 연산인 torch.stack_ 도 참고해보세요.

In [38]:
```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

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

In [42]:
```
print(torch.stack([tensor, tensor, tensor]).shape)# 쌓아 올리는 것 # 이곳 수정
print(torch.stack([tensor, tensor, tensor]))# 그냥 합치는 것 # 이곳 수정
```

```
torch.Size([3, 4, 4])
tensor([[[1., 0., 1., 1.],
         [1., 0., 1., 1.],
         [1., 0., 1., 1.],
         [1., 0., 1., 1.]],

        [[1., 0., 1., 1.],
         [1., 0., 1., 1.],
         [1., 0., 1., 1.],
         [1., 0., 1., 1.]],

        [[1., 0., 1., 1.],
         [1., 0., 1., 1.],
         [1., 0., 1., 1.],
         [1., 0., 1., 1.]]])
```

In [44]:
```
print(torch.vstack([tensor, tensor, tensor]).shape) # verticle하게 합치는 것
print(torch.vstack([tensor, tensor, tensor]))# 이곳 수정
```

```
torch.Size([12, 4])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

In [45]:
```
print(torch.hstack([tensor, tensor, tensor]).shape)# horizon 하게 합치는 것
print(torch.hstack([tensor, tensor, tensor]))
```

```
torch.Size([4, 12])
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

산술 연산(Arithmetic operations)

In [51]:
```
# 두 텐서 간의 행렬 곱(matrix multiplication)을 계산합니다. y1, y2, y3은 모두 같은
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
print(f"torch.matmul : \n {torch.matmul(tensor, tensor.T, out=y3)}") # 이


# 요소별 곱(element-wise product)을 계산합니다. z1, z2, z3는 모두 같은 값을 갖습니다.
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
print(f"torch.mul : \n {torch.mul(tensor, tensor, out=z3)}") # 이곳 수정
```

```
torch.matmul :
 tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])
torch.mul :
 tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

단일-요소(single-element) 텐서 텐서의 모든 값을 하나로 집계(aggregate)하여 요소가 하나인 텐서의 경우, `item()` 을 사용하여 Python 숫자 값으로 변환할 수 있습니다:

In [17]:
```
agg = tensor.sum()
print(agg)
agg_item = agg.item()
print(agg_item, type(agg_item))
```

```
tensor(12.)
12.0 <class 'float'>
```

바꿔치기(in-place) 연산 연산 결과를 피연산자(operand)에 저장하는 연산을 바꿔치기 연산이라고 부르며, `_` 접미사를 갖습니다. 예를 들어: `x.copy_(y)` 나 `x.t_()` 는 `x` 를 변경합니다.

In [18]:
```
print(f"{tensor} \n")
tensor.add_(5)
print(tensor)
```

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

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

> **Note**
>
> 바꿔치기 연산은 메모리를 일부 절약하지만, 기록(history)이 즉시 삭제되어 도함수(derivative) 계산에 문제가 발생할 수 있습니다. 따라서, 사용을 권장하지 않습니다.

NumPy 변환(Bridge)

CPU 상의 텐서와 NumPy 배열은 메모리 공간을 공유하기 때문에, 하나를 변경하면 다른 하나도 변경됩니다.

텐서를 NumPy 배열로 변환하기

In [ ]:
```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
type(t), type(n)
```

```
t: tensor([1., 1., 1., 1., 1.])
n: [1. 1. 1. 1. 1.]
```

Out[ ]:
```
(torch.Tensor, numpy.ndarray)
```

텐서의 변경 사항이 NumPy 배열에 반영됩니다.

In [ ]:
```
t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([6., 6., 6., 6., 6.])
n: [6. 6. 6. 6. 6.]
```

NumPy 배열을 텐서로 변환하기

In [ ]:
```
n = np.ones(5)
t = torch.from_numpy(n)
type(n), type(t)
```

Out[ ]:
```
(numpy.ndarray, torch.Tensor)
```

In [ ]:
```
torch.Tensor(n)
```

Out[ ]:
```
tensor([1., 1., 1., 1., 1.])
```

NumPy 배열의 변경 사항이 텐서에 반영됩니다.

In [ ]:
```
np.add(n, 1, out=n)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([6., 6., 6., 6., 6.], dtype=torch.float64)
n: [6. 6. 6. 6. 6.]
```

In [ ]:
```

```