발제자 이호림

6nd

딥러닝 스터디

[파이토치 기초 및 선형 회귀]

20기 분석 박진영
20기 시각화 이호림
19기 시각화 정다운

CONTENTS LIST☺
-오늘의 학습 목표

과연 이번 스터디에는 무엇을 배울까요?

출처: https://wikidocs.net/book/2788

파 이 토 치 패 키 지 의 기 본 구 성

1. torch
- 메인 네임스페이스
- 텐서 등의 다양한 수학 함수가 포함
- Numpy와 유사한 구조

2. torch.autograd
- 자동 미분을 위한 함수들 포함
- 자동 미분의 of/off를 제어하는 콘텍스트 매니저 or

자체 미분 가능 함수를 정의할 때 사용하는 기반 클래스 ‘Functhon’ 등을 포함

3. torch.nn
- 신경망을 구축하기 위한 다양한 데이터 구조나 레이어 등이 정의
- RNN, LSTM, ReLU, MSELoss

4. torch.optiom

- 확률적 경사 하강법(Stochastic Gradient Descent, SGD)을 중심으로 한 파라미터 최적화 알고리즘 구현
* 확률적 경사 하강법 - 점진적 학습의 대표적 알고리즘, 훈련 세트에서 샘플 하나씩 랜덤으로 꺼내 손실 함수의 경사를 따라 최적의 모델을 찾는 알고리즘

5. torch.utils.data

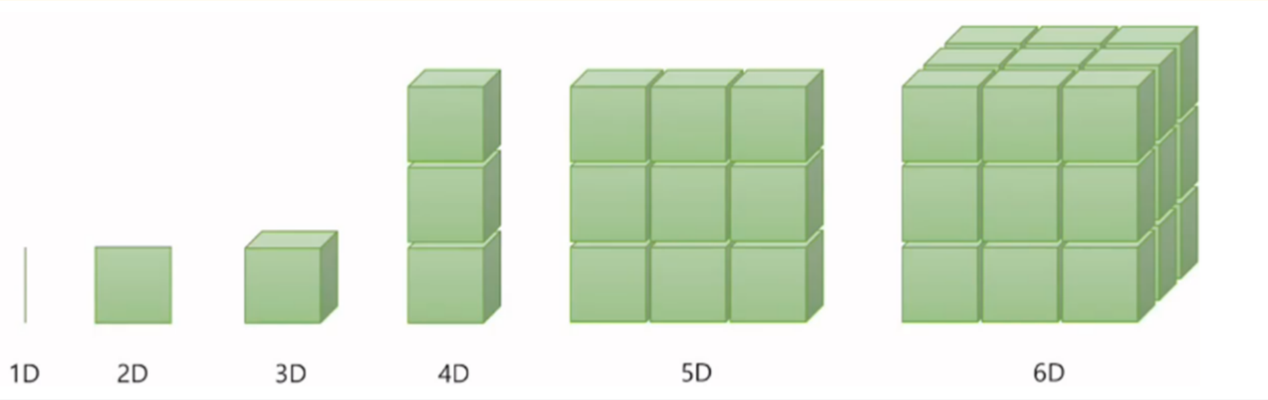
- SGD의 반복 연산 시 사용하는 미니 배치용 유틸리티 함수가 포함

6. torch.onnx

- ONNX(Open Neural Network Exchange)의 포맷으로 모델을 추출할 때 사용
* ONNX는 서로 다른 딥 러닝 프레임워크 간에 모델을 공유할 때 사용하는 포맷

1. 벡터, 행렬, 그리고 텐서(Vector, Matrix and Tensor)

1) 벡터, 행렬, 텐서 그림으로 이해하기



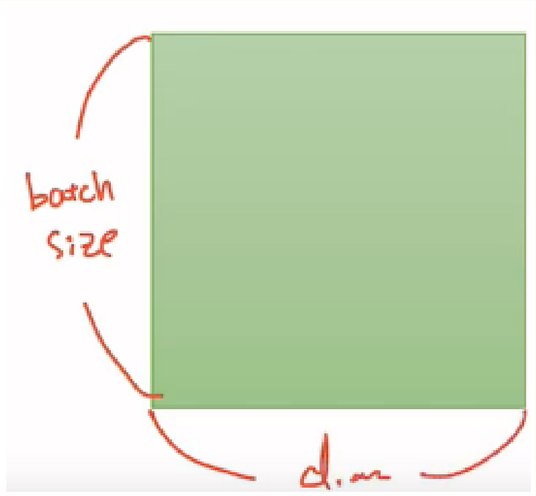
딥 러닝의 가장 기본 적인 단위는 벡터, 행렬, 텐서
벡터: 1차원으로 구성된 값
행렬(Matrix) : 2차원으로 구성된 값
텐서(Tensor): 3차원으로 구성된 값
* 벡터=1차원 텐서, 행렬= 2차원 텐서로도 표현 가능

1. 벡터, 행렬, 그리고 텐서(Vector, Matrix and Tensor)

2) PyTorch Tensor Shape Convention

- 2D Tensor(Typical Simple Setting)

|t| = (Batch size, dim)

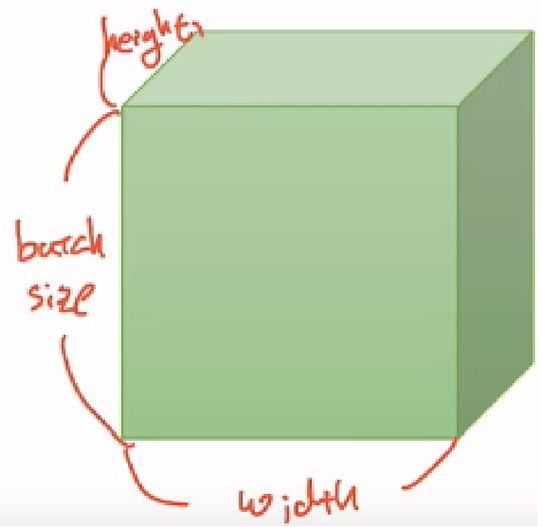


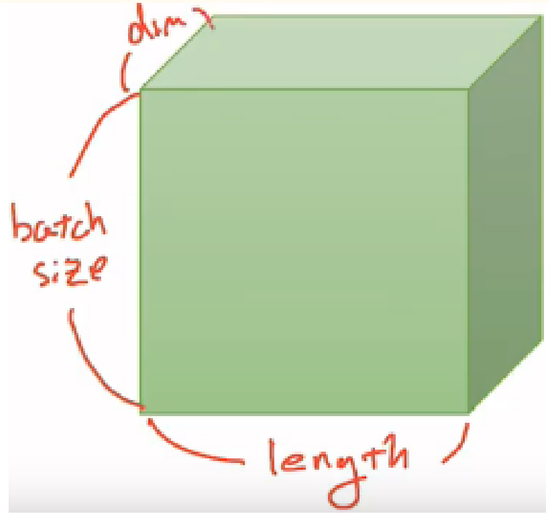
- 행의 크기가 batch size, 열의 크기가 dim

ex) 훈련데이터 하나의 크기를 256이라고 가정(벡터의 차원)
이런 데이터의 개수가 3000 -> 데이터 크기 3000 x 256
3000개에서 64개씩 처리 -> bath size 64
이는 2D 텐서의 크기 (batch size x dim) = 64 x 256

1. 벡터, 행렬, 그리고 텐서(Vector, Matrix and Tensor)

2) PyTorch Tensor Shape Convention





*NLP 분야의 3D 텐서 예제로 이해하기

- 3D Tensor(Typical Computer Vision)- 비전 분야에서의 3차원 텐서

|t| = (Batch size, width, height)

- 3D Tensor(Typical Natural Language Processing)- NLP 분야에서의 3차원 텐서

|t| = (Batch size, length, dim)

텐 서 조 작 하 기 (T e n s o r M a n i p u l a t i o n) 1

벡터, 행렬, 텐서 | 넘파이 훑어보기 | 파이토치 텐서 선언 | 행렬 곱셈 | 다른 오퍼레이션들

```
[[나는 사과를 좋아해], [나는 바나나를 좋아해], [나는 사과를 싫어해], [나는 바나나를 싫어해]]
```

- 4개의 문장으로 구성된 전체 훈련 데이터

```
[['나는', '사과를', '좋아해'], ['나는', '바나나를', '좋아해'], ['나는', '사과를', '싫어해'], ['나는', '바나나를', '싫어
해']]
```

- 4 x 3의 크기를 가지는 2D 텐서로 변환

```
'나는' = [0.1, 0.2, 0.9]
'사과를' = [0.3, 0.5, 0.1]
'바나나를' = [0.3, 0.5, 0.2]
'좋아해' = [0.7, 0.6, 0.5]
'싫어해' = [0.5, 0.6, 0.7]
```

```
[[[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.7, 0.6, 0.5]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.7, 0.6, 0.5]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.5, 0.6, 0.7]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.5, 0.6, 0.7]]]
```

- 3차원의 벡터로 변환 후 훈련 데이터 재구성
-> 4 × 3 × 3의 크기를 가지는 3D 텐서

```
첫번째 배치 #1
[[[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.7, 0.6, 0.5]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.7, 0.6, 0.5]]]

두번째 배치 #2
[[[0.1, 0.2, 0.9], [0.3, 0.5, 0.1], [0.5, 0.6, 0.7]],
 [[0.1, 0.2, 0.9], [0.3, 0.5, 0.2], [0.5, 0.6, 0.7]]]
```

- batch size를 2로 설정
- 배치의 텐서 크기는 (2 × 3 × 3) ->
(batch size, 문장 길이, 단어 벡터의 차원)
의 크기

2. 넘파이로 텐서 만들기(벡터와 행렬 만들기)

```
import numpy as np
```

- Numpy로 텐서 만들기: [숫자, 숫자, 숫자] 형식에 np.array()로 감싸줌

1) 1D with Numpy

```
t = np.array([0., 1., 2., 3., 4., 5., 6.])
# 파이썬으로 설정이된 list를 경유해서 np.array로 1차원 array로 변환함.
print(t)
```

```
[0. 1. 2. 3. 4. 5. 6.]
```

- Numpy로 1차원 텐서인 벡터 생성

```
print('Rank of t: ', t.ndim)
print('Shape of t: ', t.shape)
```

```
Rank of t:  1
Shape of t:  (7,)
```

- 1차원 텐서인 벡터의 차원과 크기를 출력
- .ndim은 몇 차원인지 출력
- shape를 표현할 때는, (컴마) 또는 (곱하기)사용
ex) 2행 3열의 2D 텐서-> (2,3) 또는 (2 x 3)

2. 넘파이로 텐서 만들기(벡터와 행렬 만들기)

1-1) Numpy 기초 이해하기

```
print('t[0] t[1] t[-1] = ', t[0], t[1], t[-1]) # 인덱스를 통한 원소 접근
```

```
t[0] t[1] t[-1] =  0.0 1.0 6.0
```

- 인덱스는 0부터 시작
- -1번 인덱스는 맨 뒤에서부터 시작

```
print('t[2:5] t[4:-1]  = ', t[2:5], t[4:-1]) # [시작 번호 : 끝 번호]로 범위 지정을 통해 가져온다.
```

```
t[2:5] t[4:-1]  =  [2. 3. 4.] [4. 5.]
```

- 슬라이싱(Slicing) 범위 지정으로 원소 출력
- [시작 번호: 끝 번호]
- 끝 번호에 해당하는 것은 미포함

2) 2D with Numpy

```
t = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.], [10., 11., 12.]])
print(t)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]
 [10. 11. 12.]]
```

- Numpy로 2차원 행렬 생성

```
print('Rank  of t: ', t.ndim)
print('Shape of t: ', t.shape)
```

```
Rank  of t:  2
Shape of t:  (4, 3)
```

- 2차원의 4행 3열 행렬

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

```
import torch
```

- 우선, torch를 임포트(Numpy와 매우 유사, 하지만 더 나음)

1) 1D with PyTorch

```
t = torch.FloatTensor([0., 1., 2., 3., 4., 5., 6.])
print(t)
```

- 파이토치로 1차원 텐서 벡터 생성

```
print(t.dim())  # rank. 즉, 차원
print(t.shape)  # shape
print(t.size()) # shape
```

```
1
torch.Size([7])
torch.Size([7])
```

- 1차원 텐서 및 원소는 7개

```
print(t[0], t[1], t[-1])  # 인덱스로 접근
print(t[2:5], t[4:-1])    # 슬라이싱
print(t[:2], t[3:])       # 슬라이싱
```

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

- Numpy의 슬라이싱과 동일

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

2) 2D with PyTorch

```
t = torch.FloatTensor([[1., 2., 3.],
                       [4., 5., 6.],
                       [7., 8., 9.],
                       [10., 11., 12.]
                      ])
print(t)
```

```
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.],
        [10., 11., 12.]])
```

- 파이토치로 2차원 텐서 행렬 생성

```
print(t.dim())  # rank. 즉, 차원
print(t.size()) # shape
```

```
2
torch.Size([4, 3])
```

- 현재 텐서의 차원은 2차원 및 (4,3)의 크기

```
print(t[:, 1]) # 첫번째 차원을 전체 선택한 상황에서 두번째 차원의 첫번째 것만 가져온다.
print(t[:, 1].size()) # ↑ 위의 경우의 크기
```

```
tensor([ 2.,  5.,  8., 11.])
torch.Size([4])
```

- 첫번째 차원을 전체 선택, 두번째 차원의 1번 인덱스 값만 추출 & 크기는 4

```
print(t[:, :-1]) # 첫번째 차원을 전체 선택한 상황에서 두번째 차원에서는 맨 마지막에서 첫번째를 제외하고 다 가져온다.
```

```
tensor([[ 1.,  2.],
        [ 4.,  5.],
        [ 7.,  8.],
        [10., 11.]])
```

- 첫번째 차원을 전체 선택, 두번째 차원의 마지막 인덱스를 제외하고 추출

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

3) 브로드캐스팅(Brodadcasting)

- 행렬의 덧셈과 뺄셈 및 곱셈

- 덧셈과 뺄셈에서는 두 행렬의 크기가 동일
- 곱셈에서는 A의 마지막 차원과 B의 첫번째 차원이 일치

```
m1 = torch.FloatTensor([[3, 3]])
m2 = torch.FloatTensor([[2, 2]])
print(m1 + m2)
```

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

m1과 m2의 크기는 둘 다 (1,2)
-> 문제없이 덧셈 연산 가능

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

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

-m1의 크기는 (1,2)이고 m2의 크기는(1,)
-m2의 크기를 (1,2)로 변경하여 연산 수행

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

```
# 브로드캐스팅 과정에서 실제로 두 텐서가 어떻게 변경되는지 보겠습니다.
[1, 2]
==> [[1, 2],
     [1, 2]]
[3]
[4]
==> [[3, 3],
     [4, 4]]
```

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

-원래는크기가 달라서 연산 불가지만 파이토치가
두 벡터의 크기를 (2,2)로 변경하여 덧셈 수행

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

4) 자주 사용되는 기능들

- 행렬 곱셈과 곰셈의 차이(Matrix Multiplication Vs. Multiplication)

```
m1 = torch.FloatTensor([[1, 2], [3, 4]])
m2 = torch.FloatTensor([[1], [2]])
print('Shape of Matrix 1: ', m1.shape) # 2 x 2
print('Shape of Matrix 2: ', m2.shape) # 2 x 1
print(m1.matmul(m2)) # 2 x 1
```

```
Shape of Matrix 1:  torch.Size([2, 2])
Shape of Matrix 2:  torch.Size([2, 1])
tensor [[ 5.],
        [11.]])
```

2 x 2 행렬과 2 x 1 행렬(벡터)의
행렬 곱셈의 결과

```
m1 = torch.FloatTensor([[1, 2], [3, 4]])
m2 = torch.FloatTensor([[1], [2]])
print('Shape of Matrix 1: ', m1.shape) # 2 x 2
print('Shape of Matrix 2: ', m2.shape) # 2 x 1
print(m1 * m2) # 2 x 2
print(m1.mul(m2))
```

```
# 브로드캐스팅 과정에서 m2 텐서가 어떻게 변경되는지 보겠습니다.
[1]
[2]
==> [[1, 1],
     [2, 2]]
```

```
Shape of Matrix 1:  torch.Size([2, 2])
Shape of Matrix 2:  torch.Size([2, 1])
tensor([[1., 2.],
        [6., 8.]])
tensor([[1., 2.],
        [6., 8.]])
```

-m1의 크기는 (2,2)이고 m2의 크기는 (2,1)
-element-wise 곱셈을 수행면 두 행령의 크기가 브로드캐스팅이 된 후에 곱셈이 수행
*element-wise: 동일한 크기의 행렬이 동일한 위치에 있는 원소끼리 곱하는 것

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

4) 자주 사용되는 기능들

- 평균(Mean)

```
t = torch.FloatTensor([1, 2])
print(t.mean())
```

```
tensor(1.5000)
```

1차원인 벡터를 선언하여 .mean()을 사용한 후 원소의 평균 구함 -> 2개의 원소의 평균

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

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

```
print(t.mean())
```

```
tensor(2.5000)
```

2차원인 행렬을 선언하여 .mean()을 사용한 후 원소의 평균 구함 -> 4개의 원소의 평균

```
print(t.mean(dim=0))
```

```
tensor([2., 3.])
```

dim=0은 첫번째 차원을 의미(행), 인자로 dim을 준다면 해당 차원을 제거한다는 의미 -> '열'만 남긴다는 의미

```
# 실제 연산 과정
t.mean(dim=0)은 입력에서 첫번째 차원을 제거한다.

[[1., 2.],
 [3., 4.]]

1과 3의 평균을 구하고, 2와 4의 평균을 구한다.
결과 ==> [2., 3.]
```

열의 차원만 보존되면서 (1,2)=(2,), 벡터가 됨

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

4) 자주 사용되는 기능들

- 평균(Mean)

```
t = torch.FloatTensor([[1, 2], [3, 4]])
```

```
print(t.mean(dim=1))
```

```
tensor([1.5000, 3.5000])
```

=

```
print(t.mean(dim=-1))
```

```
tensor([1.5000, 3.5000])
```

dim=1은 두번째 차원을 의미(열), 인자로 dim을 준다면 해당 차원을 제거한다는 의미 -> '행'만 남긴다는 의미

dim=-1는 마지막 차원을 제거한다는 의미고, 결국 열의 차원을 제거한다는 의미

```
# 실제 연산 결과는 (2 x 1)
[1. 5]
[3. 5]
```

- 열의 차원이 제거되면서 (2,2)에서 (2,1)크기가 됨
- 1과2의 평균 & 3과 4의 평균

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

4) 자주 사용되는 기능들

- 덧셈(sum)

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

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

2차원의 행렬을 선언

```
print(t.sum()) # 단순히 원소 전체의 덧셈을 수행
print(t.sum(dim=0)) # 행을 제거
print(t.sum(dim=1)) # 열을 제거
print(t.sum(dim=-1)) # 열을 제거
```

```
tensor(10.)
tensor([4., 6.])
tensor([3., 7.])
tensor([3., 7.])
```

- dim=0 행을 제거
- dim=1 열을 제거
- dim=-1 열을 제거

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

4) 자주 사용되는 기능들

- 최대(Max)와 아그맥스(ArgMax)

* 최대(Max)는 원소의 최대값을 리턴, 아그맥스(ArgMax)는 최대값을 가진 인덱스를 리턴

```
t = torch.FloatTensor([[1, 2], [3, 4]])
print(t)
```

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

(2,2) 행렬을 선언

```
print(t.max()) # Returns one value: max
```

```
tensor(4.)
```

원소 중 최대값인 4를 리턴

```
print(t.max(dim=0)) Returns two values: max and argmax
```

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

```
# [1, 1]가 무슨 의미인지 봅시다. 기존 행렬을 다시 상기해봅시다.
[[1, 2],
 [3, 4]]
첫번째 열에서 0번 인덱스는 1, 1번 인덱스는 3입니다.
두번째 열에서 0번 인덱스는 2, 1번 인덱스는 4입니다.
다시 말해 3과 4의 인덱스는 [1, 1]입니다.
```

- dim=0(행의 차원 제거), (1,2) 텐서에서 최대값 리턴 ->[3,4]
- max에서 dim 인자를 주면 argmax도 함께 리턴

3. 파이토치 텐서 선언하기(PyTorch Tensor Allocation)

4) 자주 사용되는 기능들

- 최대(Max)와 아그맥스(ArgMax)

* 최대(Max)는 원소의 최대값을 리턴, 아그맥스(ArgMax)는 최대값을 가진 인덱스를 리턴

```
print('Max: ', t.max(dim=0)[0])
print('Argmax: ', t.max(dim=0)[1])
```

```
Max:  tensor([3., 4.])
Argmax:  tensor([1, 1])
```

- max와 argmax를 함께 리턴 받는 것이 아니라 따로 리턴받고 싶으면 인덱스 사용
- 0번 인덱스는 max, 1번 인덱스는 argmax

```
print(t.max(dim=1))
print(t.max(dim=-1))
```

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

- dim=1로 인자를 주었을 때, dim=-1을 인자로 주었을 때
- 둘 다 열의 차원을 제거한 후 최대값 출력
- 함께 리턴된 argmax로 인해 첫번째 열에서 2의 인덱스 1, 두번째 열에서 4의 인덱스 1

4. 뷰(View)-원소의 수를 유지하면서 텐서의 크기 변경

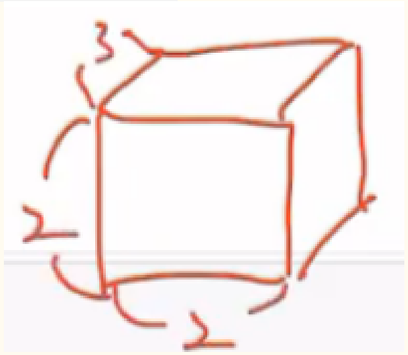
- Numpy에서의 Reshape 역할
- 텐서의 크기(Shape)를 변경

```
t = np.array([[[0, 1, 2],
               [3, 4, 5]],
              [[6, 7, 8],
               [9, 10, 11]]])
ft = torch.FloatTensor(t)

print(ft.shape)
```

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

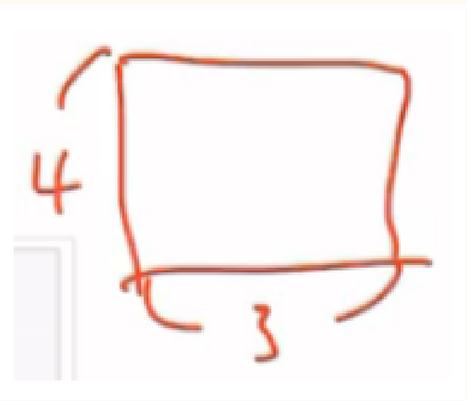
위 텐서의 크기는 (2,2,3)



4-1) 3차원 텐서에서 2차원 텐서로 변경

```
print(ft.view([-1, 3])) # ft라는 텐서를 (?, 3)의 크기로 변경
print(ft.view([-1, 3]).shape)
```

```
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.],
        [ 9., 10., 11.]])
torch.Size([4, 3])
```



- view([-1,3])에서 -1은 사용자가 잘 모르겠으니 파이토치에 맡기겠다는 의미, 3은 차원의 길이를 3을 가지도록 하라는 의미
-> 현재 3차원의 텐서을 2차원 텐서로 변경하되 (?,3)의 크기로 변경
-> 결과적으로 (4,3)의 크기를 가짐
- (2,2,3) -> (2 x 2, 3) -> (4,3)

- view는 기본적으로 변경 전과 변경 후의 텐서 안의 원소의 개수가 유지
- view의 사이즈가 -1로 설정되면 다른 차원으로부터 해당 값 유추

4-2) 3차원 텐서의 크기 변경

```
print(ft.view([-1, 1, 3]))
print(ft.view([-1, 1, 3]).shape)
```

```
tensor([[[ 0.,  1.,  2.]],

        [[ 3.,  4.,  5.]],

        [[ 6.,  7.,  8.]],

        [[ 9., 10., 11.]]])
torch.Size([4, 1, 3])
```

(2 x 2 x 3)의 텐서를 (? x 1 x 3) 텐서로 변경
-> 크기는 변경하더라도 원소의 수는 유지
-> 12= ? x 3 -> ?=4

5. 스퀴즈(Squeeze)-1인 차원을 제거

- 차원이 1인 경우에는 해당 차원 제거

```
ft = torch.FloatTensor([[0], [1], [2]])
print(ft)
print(ft.shape)
```

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

(3 x 1)의 크기를 가진 임의의 텐서 선정

```
print(ft.squeeze())
print(ft.squeeze().shape)
```

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

두번째 차원이 1이므로 (3,)의 크기를 가지는 텐서로 변경
-> 1차원 벡터로 변경

6. 언스퀴즈(Unsqueeze)-특정 위치에 1인 차원을 추가

- 스퀴즈와 정반대
- 특정 위치에 1인 차원을 추가

```
ft = torch.Tensor([0, 1, 2])
print(ft.shape)
```

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

임의로 (3,)의 크기를 가진 1인 차원 텐서 생성
-> 차원이 1개인 1차원 벡터

```
print(ft.unsqueeze(0)) # 인덱스가 0부터 시작하므로 0은 첫번째 차원을 의미한다.
print(ft.unsqueeze(0).shape)
```

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

첫번째 차원의 인덱스를 의미하는 숫자 0을 인자로 넣기
-> 첫번째 차원에 1인 차원이 추가
-> (3,)의 크기의 1차원 벡터가 (1,3)의 2차원 텐서로 변경

```
print(ft.view(1, -1))
print(ft.view(1, -1).shape)
```

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

2차원으로 바꾸고 싶으면서 첫번째 차원은 1로
-> view에서 (1, -1)을 인자로 사용

```
print(ft.unsqueeze(1))
print(ft.unsqueeze(1).shape)
```

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

두번째 차원의 인덱스를 의미하는 숫자 1을 인자로 넣기
-> 두번째 차원에 1인 차원이 추가
-> (3,)의 크기의 1차원 벡터가 (3,1)의 2차원 텐서로 변경

텐서 조작하기(Tensor Manipulation)2

7. 타입 캐스팅(Type Casting)

- 텐서의 자료형을 변환

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

```
lt = torch.LongTensor([1, 2, 3, 4])
print(lt)
```

long 타입의 lt라는 텐서 선언

```
print(lt.float())
```

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

텐서에다가 .float()를 붙이면 float형으로 타입이 변경

8. 연결하기(concatenate)

- 두 텐서를 연결

```
x = torch.FloatTensor([[1, 2], [3, 4]])
y = torch.FloatTensor([[5, 6], [7, 8]])
```

(2 x 2) 크기의 텐서 두 개 생성

```
print(torch.cat([x, y], dim=0))
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.],
        [7., 8.]])
```

- torch.cat를 통해 두 텐서 연결
- dim=0을 통해 첫번째 차원을 늚
-> (2 x 2) 텐서가 (4 x 2)로 변경
* dim=1은 (2 x 4)

텐서 조작하기(Tensor Manipulation)2

9. 스택킹(Stacking)

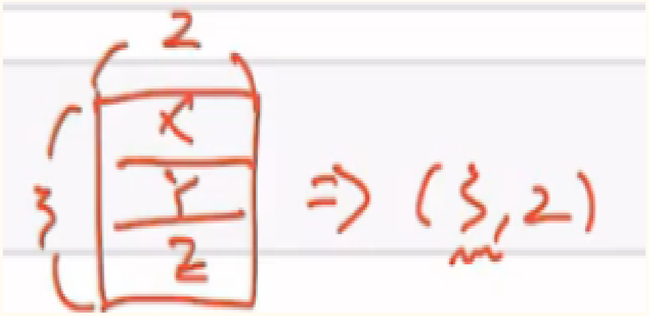
- 두 텐서를 연결
- 많은 연산을 포함할 때

```
x = torch.FloatTensor([1, 4])
y = torch.FloatTensor([2, 5])
z = torch.FloatTensor([3, 6])
```

크기가 (2,)로 모두 동일한 3개의 벡터 생성

```
print(torch.stack([x, y, z]))
```

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



3개의 벡터가 순차적으로 쌓여 (3 x 2) 텐서가 됨

* concatenate에 비해 한 번의 커맨드로 수행 가능

10. ones_like와 zeros_like-0으로 채워진 텐서 & 1로 채워진 텐서

```
x = torch.FloatTensor([[0, 1, 2], [2, 1, 0]])
print(x)
```

```
tensor([[0., 1., 2.],
        [2., 1., 0.]])
```

(2 x 3) 텐서 생성

```
print(torch.ones_like(x)) # 입력 텐서와 크기를 동일하게 하면서 값을 1로 채우기
```

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

동일한 크기에 1으로만 값이 채워진 텐서 생성

```
print(torch.zeros_like(x)) # 입력 텐서와 크기를 동일하게 하면서 값을 0으로 채우기
```

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

동일한 크기에 0으로만 값이 채워진 텐서 생성

11. In-place Operation(덮어쓰기 연산)

```
x = torch.FloatTensor([[1, 2], [3, 4]])
```

(2 x 2) 텐서 생성 후 x에 저장

```
print(x.mul(2.)) # 곱하기 2를 수행한 결과를 출력
print(x) # 기존의 값 출력
```

```
tensor([[2., 4.],
        [6., 8.]])
tensor([[1., 2.],
        [3., 4.]])
```

- 곱하기 2가 수행된 결과, 기존의 값 그대로 출력
- 곱하기 2를 수행했지만 이를 x에다가 다시 저장하지 않았기에 x는 그대로

```
print(x.mul_(2.)) # 곱하기 2를 수행한 결과를 변수 x에 값을 저장하면서 결과를 출력
print(x) # 기존의 값 출력
```

```
tensor([[2., 4.],
        [6., 8.]])
tensor([[2., 4.],
        [6., 8.]])
```

- mul 연산 뒤에 _을 붙이면 기존의 값을 덮어쓰기
- x의 값이 덮어쓰기 되어 2곱하기 연산된 결과가 출력

nn.Module로 구현하는 선형 회귀

1. 단순 선형 회귀 구현하기

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
torch.manual_seed(1)
```

- 필요한 도구들을 임포트

```
# 데이터
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```

- 데이터를 선언
- y=2x를 가정된 상태에서 만들어진 데이터

```
# 모델을 선언 및 초기화. 단순 선형 회귀이므로 input_dim=1, output_dim=1.
model = nn.Linear(1,1)
```

- 하나의 입력 x에 대해서 하나의 출력 y를 가지므로
입력 차원과 출력 차원 모두 1을 인수로 사용

```
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[0.5153]], requires_grad=True), Parameter containing:
tensor([-0.4414], requires_grad=True)]
```

- model.parameters()함수 이용해서 가중치 W와 편향 b 확인
- 첫번째 값이 W, 두번째 값이 b에 해당
- 두 값 모두 학습의 대상이므로 requires_grad=True

nn.Module로 구현하는 선형 회귀

- 옵티마이저 정의 및 전체 훈련 데이터에 경사 하강법 실행

```
# optimizer 설정. 경사 하강법 SGD를 사용하고 learning rate를 의미하는 lr은 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
# 전체 훈련 데이터에 대해 경사 하강법을 2,000회 반복
nb_epochs = 2000
for epoch in range(nb_epochs+1):

    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.mse_loss(prediction, y_train) # <== 파이토치에서 제공하는 평균 제곱 오차 함수

    # cost로 H(x) 개선하는 부분
    # gradient를 0으로 초기화
    optimizer.zero_grad()
    # 비용 함수를 미분하여 gradient 계산
    cost.backward() # backward 연산
    # W와 b를 업데이트
    optimizer.step()

    if epoch % 100 == 0:
    # 100번마다 로그 출력
      print('Epoch {:4d}/{} Cost: {:.6f}'.format(
          epoch, nb_epochs, cost.item()
      ))
```

```
Epoch    0/2000 Cost: 13.103540
... 중략 ...
Epoch 2000/2000 Cost: 0.000000
```

학습된 결과에서 Cost의 값이 매우 작음

```
# 임의의 입력 4를 선언
new_var =  torch.FloatTensor([[4.0]])
# 입력한 값 4에 대해서 예측값 y를 리턴받아서 pred_y에 저장
pred_y = model(new_var) # forward 연산
# y = 2x 이므로 입력이 4라면 y가 8에 가까운 값이 나와야 제대로 학습이 된 것
print("훈련 후 입력이 4일 때의 예측값 :", pred_y)
```

```
훈련 후 입력이 4일 때의 예측값 : tensor([[7.9989]], grad_fn=<AddmmBackward>)
```

- W와 b의 값도 최적화가 되었는지 확인
- x에 임의의 값 4를 넣어 모델의 예측 y값 확인
- 정답 y=2x에서 y값이 8에 가까우면 최적화가 된 것으로 볼 수 있음 -> 실제 예측된 y값은 7.9989로 8에 매우 가까움

```
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[1.9994]], requires_grad=True), Parameter containing:
tensor([0.0014], requires_grad=True)]
```

-학습 후의 W와 b의 값을 출력
-W의 값이 2에 가깝고 b의 값이 0에 가까움
-> 성능이 괜찮은 모델 구현 ok!

2. 다중 선형 회귀 구현하기

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
torch.manual_seed(1)
```

- 필요한 도구들을 임포트

```
# 데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

- 3개의 x로부터 하나의 y를 예측
- 가설 수식은 H(x)=w1x1+w2x2+w3x3+b

```
# 모델을 선언 및 초기화. 다중 선형 회귀이므로 input_dim=3, output_dim=1.
model = nn.Linear(3,1)
```

- 3개의 입력 x에 대해서 하나의 출력 y를 가지므로
입력 차원은 3, 출력 차원은 1을 인수로 사용

```
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[ 0.2975, -0.2548, -0.1119]], requires_grad=True), Parameter containing:
tensor([0.2710], requires_grad=True)]
```

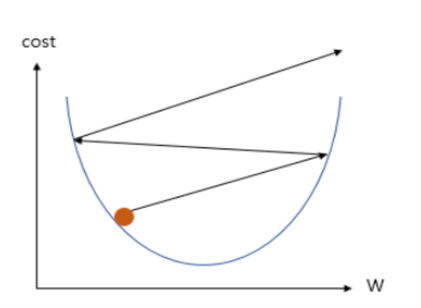
- model.parameters()함수 이용해서 가중치 W와 편향 b 확인
- 첫번째 값이 3개의 W, 두번째 값이 b에 해당
- 두 값 모두 학습의 대상이므로 requires_grad=True

nn.Module로 구현하는 선형 회귀

- 옵티마이저 정의 및 전체 훈련 데이터에 경사 하강법 실행

```
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
```

학습률을 (lr) 0.01로 하지 않는 이유는 오른쪽 그림처럼 기울기가 발산하기 때문




```
nb_epochs = 2000
for epoch in range(nb_epochs+1):

    # H(x) 계산
    prediction = model(x_train)
    # model(x_train)은 model.forward(x_train)와 동일함.

    # cost 계산
    cost = F.mse_loss(prediction, y_train) # <== 파이토치에서 제공하는 평균 제곱 오차 함수

    # cost로 H(x) 개선하는 부분
    # gradient를 0으로 초기화
    optimizer.zero_grad()
    # 비용 함수를 미분하여 gradient 계산
    cost.backward()
    # W와 b를 업데이트
    optimizer.step()

    if epoch % 100 == 0:
    # 100번마다 로그 출력
      print('Epoch {:4d}/{} Cost: {:.6f}'.format(
          epoch, nb_epochs, cost.item()
      ))
```

```
Epoch    0/2000 Cost: 31667.597656
... 중략 ...
Epoch 2000/2000 Cost: 0.199777
```

학습된 결과에서 Cost의 값이 매우 작음

```
# 임의의 입력 [73, 80, 75]를 선언
new_var =  torch.FloatTensor([[73, 80, 75]])
# 입력한 값 [73, 80, 75]에 대해서 예측값 y를 리턴받아서 pred_y에 저장
pred_y = model(new_var)
print("훈련 후 입력이 73, 80, 75일 때의 예측값 :", pred_y)
```

```
훈련 후 입력이 73, 80, 75일 때의 예측값 : tensor([[151.2305]], grad_fn=<AddmmBackward>)
```

- W와 b의 값도 최적화가 되었는지 확인
- x에 임의의 값 [73, 80,75]를 넣어 모델의 예측 y값 확인
- 당시 y의 값은 152였는데, 현재 예측값이 151

-> 3개의 W와 b의 값이 최적화

```
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[0.9778, 0.4539, 0.5768]], requires_grad=True), Parameter containing:
tensor([0.2802], requires_grad=True)]
```

학습 후의 3개의 w와 b의 값 출력

클래스로 파이토치 모델 구현하기

1. 모델을 클래스로 구현기

```
# 모델을 선언 및 초기화. 단순 선형 회귀이므로 input_dim=1, output_dim=1.
model = nn.Linear(1,1)
```

앞선 단순 선형 회귀 모델은 위와 같이 구현

```
class LinearRegressionModel(nn.Module): # torch.nn.Module을 상속받는 파이썬 클래스
    def __init__(self): #
        super().__init__()
        self.linear = nn.Linear(1, 1) # 단순 선형 회귀이므로 input_dim=1, output_dim=1.

    def forward(self, x):
        return self.linear(x)
```

```
model = LinearRegressionModel()
```

이를 클래스로 구현

```
# 모델을 선언 및 초기화. 다중 선형 회귀이므로 input_dim=3, output_dim=1.
model = nn.Linear(3,1)
```

앞선 다중 선형 회귀 모델은 위와 같이 구현

```
class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1) # 다중 선형 회귀이므로 input_dim=3, output_dim=1.

    def forward(self, x):
        return self.linear(x)
```

```
model = MultivariateLinearRegressionModel()
```

이를 클래스로 구현

(*자세한 코드 설명은 녹화 영상에 설명할 예정*)

2. 단순 선형 회귀 클래스로 구현하기

*앞선 코드와 동일(모델을 클래스로 구현했다는 점만 다름)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
torch.manual_seed(1)
```

```
# 데이터
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```

```
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)
```

```
model = LinearRegressionModel()
```

```
# optimizer 설정. 경사 하강법 SGD를 사용하고 learning rate를 의미하는 lr은 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
# 전체 훈련 데이터에 대해 경사 하강법을 2,000회 반복
nb_epochs = 2000
for epoch in range(nb_epochs+1):

    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.mse_loss(prediction, y_train) # <== 파이토치에서 제공하는 평균 제곱 오차 함수

    # cost로 H(x) 개선하는 부분
    # gradient를 0으로 초기화
    optimizer.zero_grad()
    # 비용 함수를 미분하여 gradient 계산
    cost.backward() # backward 연산
    # W와 b를 업데이트
    optimizer.step()

    if epoch % 100 == 0:
    # 100번마다 로그 출력
      print('Epoch {:4d}/{} Cost: {:.6f}'.format(
          epoch, nb_epochs, cost.item()
      ))
```

3. 다중 선형 회귀 클래스로 구현하기

*앞선 코드와 동일(모델을 클래스로 구현했다는 점만 다름)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
torch.manual_seed(1)
```

```
# 데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

```
class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1) # 다중 선형 회귀이므로 input_dim=3, output_dim=1.

    def forward(self, x):
        return self.linear(x)
```

```
model = MultivariateLinearRegressionModel()
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
```

```
nb_epochs = 2000
for epoch in range(nb_epochs+1):

    # H(x) 계산
    prediction = model(x_train)
    # model(x_train)은 model.forward(x_train)와 동일함.

    # cost 계산
    cost = F.mse_loss(prediction, y_train) # <== 파이토치에서 제공하는 평균 제곱 오차 함수

    # cost로 H(x) 개선하는 부분
    # gradient를 0으로 초기화
    optimizer.zero_grad()
    # 비용 함수를 미분하여 gradient 계산
    cost.backward()
    # W와 b를 업데이트
    optimizer.step()

    if epoch % 100 == 0:
    # 100번마다 로그 출력
      print('Epoch {:4d}/{} Cost: {:.6f}'.format(
          epoch, nb_epochs, cost.item()
      ))
```

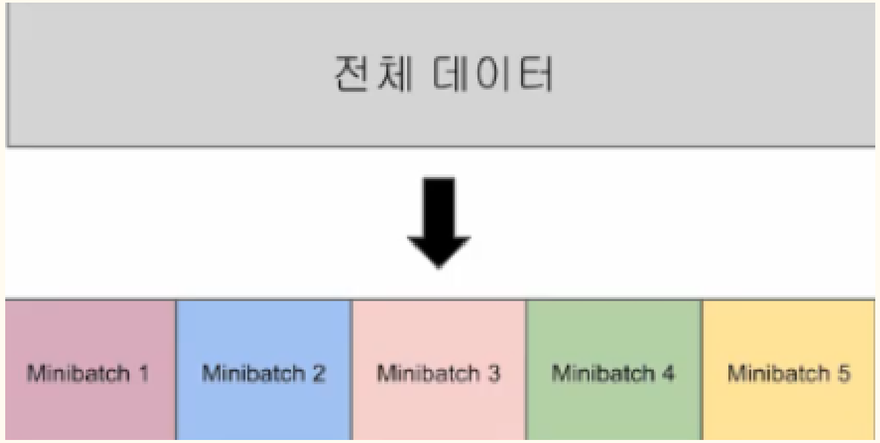
1. 미니 배치와 배치 크기(Mini Batch and Batch Size)

```
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

위 데이터의 샘플의 수는 5개
-> 경사 하강법을 수행하여 학습 가능

- 현업에 비해 굉장히 적은 양
- 수십만개 이상의 데이터일 때 경사하강법은 매우 느리고 많은 계산량 필요

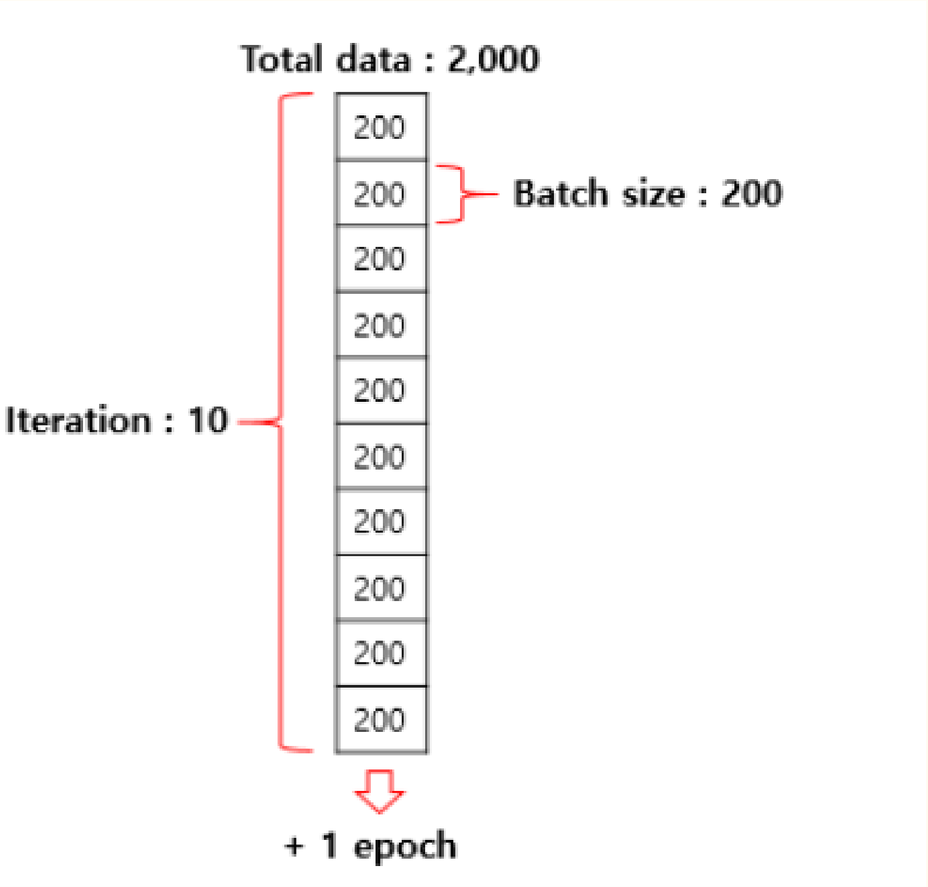
-> 전체 데이터를 더 작은 단위로 나누어 해당 단위로 학습하는 미니 배치(Mini Batch) 등장



전체 데이터를 미니 배치 단위로 나누고 미니배치 학습 후 미니 배치만큼만 가져가 미니 배치에 대한 비용(cost)을 계산하고 경사 하강법 수행

a. 배치 경사 하강법
b. 미니 배치 경사 하강법

2. 이터레이션(Iteration)



- 이터레이션은 한 번의 에포크 내에서 이루어지는 매개변수인 가중치 W와 b의 업데이트 수
- ex) 전체 데이터가 2000일 때 배치 크기를 200으로 한다면 이터레이션의 수는 총 10개 -> 한 번의 에포크 당 매개변수 업데이트가 10번

3. 데이터 로드하기(Data Load)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

파이토치의 도구들을 임포트

```
from torch.utils.data import TensorDataset # 텐서데이터셋
from torch.utils.data import DataLoader # 데이터로더
```

TensorDataset와 DataLoader 임포트

```
x_train  =  torch.FloatTensor([[73,  80,  75],
                               [93,  88,  93],
                               [89,  91,  90],
                               [96,  98,  100],
                               [73,  66,  70]])
y_train  =  torch.FloatTensor([[152],  [185],  [180],  [196],  [142]])
```

텐서 형태로 데이터 정의

- 옵티마이저 정의 및 전체 훈련 데이터에 경사 하강법 실행

```
dataset = TensorDataset(x_train, y_train)
```

```
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)
```

```
model = nn.Linear(3,1)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
```

```
nb_epochs = 20
for epoch in range(nb_epochs + 1):
  for batch_idx, samples in enumerate(dataloader):
    # print(batch_idx)
    # print(samples)
    x_train, y_train = samples
    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.mse_loss(prediction, y_train)

    # cost로 H(x) 계산
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    print('Epoch {:4d}/{} Batch {}/{} Cost: {:.6f}'.format(
        epoch, nb_epochs, batch_idx+1, len(dataloader),
        cost.item()
        ))
```

```
Epoch    0/20 Batch 1/3 Cost: 26085.919922
Epoch    0/20 Batch 2/3 Cost: 3660.022949
Epoch    0/20 Batch 3/3 Cost: 2922.390869
... 중략 ...
Epoch   20/20 Batch 1/3 Cost: 6.315856
Epoch   20/20 Batch 2/3 Cost: 13.519956
Epoch   20/20 Batch 3/3 Cost: 4.262849
```

```
# 임의의 입력 [73, 80, 75]를 선언
new_var =  torch.FloatTensor([[73, 80, 75]])
# 입력한 값 [73, 80, 75]에 대해서 예측값 y를 리턴받아서 pred_y에 저장
pred_y = model(new_var)
print("훈련 후 입력이 73, 80, 75일 때의 예측값 :", pred_y)
```

```
훈련 후 입력이 73, 80, 75일 때의 예측값 : tensor([[154.3850]], grad_fn=<AddmmBackward>)
```

1. 커스텀 데이터셋(Custom Dataset)

*Dataset을 상속받아 아래 메소드들을 오버라이드 하여 커스텀 데이터셋 생성

```
class CustomDataset(torch.utils.data.Dataset):
  def __init__(self):

  def __len__(self):

  def __getitem__(self, idx):
```

```
class CustomDataset(torch.utils.data.Dataset):
  def __init__(self):
  데이터셋의 전처리를 해주는 부분

  def __len__(self):
  데이터셋의 길이. 즉, 총 샘플의 수를 적어주는 부분

  def __getitem__(self, idx):
  데이터셋에서 특정 1개의 샘플을 가져오는 함수
```

커스텀 데이터셋을 만들 때, 필요한 기본적인 define 3개

-len(dataset)을 했을 때 데이터 셋의 크기를 리턴할 len
-dataset[i]을 했을 때 i번째 샘플을 가져오도록 하는 인덱싱을 위한 get_item

2. 커스텀 데이터셋(Custom Dataset)으로 선형 회귀 구현하기

```
import torch
import torch.nn.functional as F
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
```

```
# Dataset 상속
class CustomDataset(Dataset):
  def __init__(self):
    self.x_data = [[73, 80, 75],
                   [93, 88, 93],
                   [89, 91, 90],
                   [96, 98, 100],
                   [73, 66, 70]]
    self.y_data = [[152], [185], [180], [196], [142]]

  # 총 데이터의 개수를 리턴
  def __len__(self):
    return len(self.x_data)

  # 인덱스를 입력받아 그에 맵핑되는 입출력 데이터를 파이토치의 Tensor 형태로 리턴
  def __getitem__(self, idx):
    x = torch.FloatTensor(self.x_data[idx])
    y = torch.FloatTensor(self.y_data[idx])
    return x, y
```

```
dataset = CustomDataset()
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)
```

```
model = torch.nn.Linear(3,1)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
```

```
nb_epochs = 20
for epoch in range(nb_epochs + 1):
  for batch_idx, samples in enumerate(dataloader):
    # print(batch_idx)
    # print(samples)
    x_train, y_train = samples
    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.mse_loss(prediction, y_train)

    # cost로 H(x) 계산
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    print('Epoch {:4d}/{} Batch {}/{} Cost: {:.6f}'.format(
        epoch, nb_epochs, batch_idx+1, len(dataloader),
        cost.item()
        ))
```

```
Epoch    0/20 Batch 1/3 Cost: 29410.156250
Epoch    0/20 Batch 2/3 Cost: 7150.685059
Epoch    0/20 Batch 3/3 Cost: 3482.803467
... 중략 ...
Epoch   20/20 Batch 1/3 Cost: 0.350531
Epoch   20/20 Batch 2/3 Cost: 0.653316
Epoch   20/20 Batch 3/3 Cost: 0.010318
```

```
# 임의의 입력 [73, 80, 75]를 선언
new_var =  torch.FloatTensor([[73, 80, 75]])
# 입력한 값 [73, 80, 75]에 대해서 예측값 y를 리턴받아서 pred_y에 저장
pred_y = model(new_var)
print("훈련 후 입력이 73, 80, 75일 때의 예측값 :", pred_y)
```

```
훈련 후 입력이 73, 80, 75일 때의 예측값 : tensor([[151.2319]], grad_fn=<AddmmBackward>)
```

1. 파이토치의 nn.Linear와 nn.Sigmoid로 로지스틱 회귀 구현하기

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```
torch.manual_seed(1)
```

필요한 파이토치의 도구들 임포트

```
x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]
y_data = [[0], [0], [0], [1], [1], [1]]
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

훈련 데이터를 텐서로 선언

```
model = nn.Sequential(
   nn.Linear(2, 1), # input_dim = 2, output_dim = 1
   nn.Sigmoid() # 출력은 시그모이드 함수를 거친다
)
```

로지스틱 회귀 구현 시작

```
model(x_train)
```

```
tensor([[0.4020],
        [0.4147],
        [0.6556],
        [0.5948],
        [0.6788],
        [0.8061]], grad_fn=<SigmoidBackward>)
```

```
# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=1)

nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = model(x_train)

    # cost 계산
    cost = F.binary_cross_entropy(hypothesis, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 20번마다 로그 출력
    if epoch % 10 == 0:
        prediction = hypothesis >= torch.FloatTensor([0.5]) # 예측값이 0.5를 넘으면 True로 간주
        correct_prediction = prediction.float() == y_train # 실제값과 일치하는 경우만 True로 간주
        accuracy = correct_prediction.sum().item() / len(correct_prediction) # 정확도를 계산
        print('Epoch {:4d}/{} Cost: {:.6f} Accuracy {:2.2f}%'.format( # 각 에포크마다 정확도를 출력
            epoch, nb_epochs, cost.item(), accuracy * 100,
        ))
```

```
Epoch    0/1000 Cost: 0.539713 Accuracy 83.33%
... 중략 ...
Epoch 1000/1000 Cost: 0.019843 Accuracy 100.00%
```

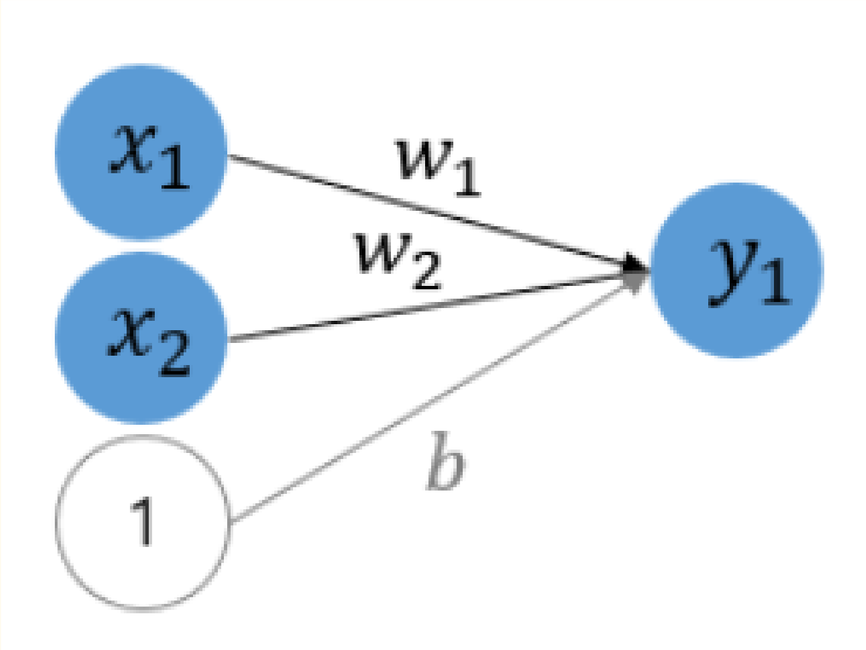
```
model(x_train)
```

```
tensor([[0.0240],
        [0.1476],
        [0.2739],
        [0.7967],
        [0.9491],
        [0.9836]], grad_fn=<SigmoidBackward>)
```

```
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[3.2534, 1.5181]], requires_grad=True), Parameter containing:
tensor([-14.4839], requires_grad=True)]
```

2. 인공 신경망으로 표현되는 로지스틱 회귀



- 검은색 화살표는 가중치
- 회색 화살표는 편향이 곱해짐
- 각 입력 x는 각 입력의 가중치 W와 곱해지고 편향 b는상수 1과 곱해지는 것으로 표현

H(x)=sigmoid(x1w1+x2w2+b)

클래스로 파이토치 모델 구현하기

1. 모델을 클래스로 구현하기

```
model = nn.Sequential(
   nn.Linear(2, 1), # input_dim = 2, output_dim = 1
   nn.Sigmoid() # 출력은 시그모이드 함수를 거친다
)
```

앞선 로지스틱 회귀 모델은 위와 같이 구현

```
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.linear(x))
```

이를 클래스로 구현

2. 로지스틱 회귀 클래스로 구현하기

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```
torch.manual_seed(1)
```

```
x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]
y_data = [[0], [0], [0], [1], [1], [1]]
x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

```
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return self.sigmoid(self.linear(x))
```

```
model = BinaryClassifier()
```

```
# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=1)

nb_epochs = 1000
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = model(x_train)

    # cost 계산
    cost = F.binary_cross_entropy(hypothesis, y_train)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 20번마다 로그 출력
    if epoch % 10 == 0:
        prediction = hypothesis >= torch.FloatTensor([0.5]) # 예측값이 0.5를 넘으면 True로 간주
        correct_prediction = prediction.float() == y_train # 실제값과 일치하는 경우만 True로 간주
        accuracy = correct_prediction.sum().item() / len(correct_prediction) # 정확도를 계산
        print('Epoch {:4d}/{} Cost: {:.6f} Accuracy {:2.2f}%'.format( # 각 에포크마다 정확도를 출력
            epoch, nb_epochs, cost.item(), accuracy * 100,
        ))
```

발제자 이호림

THANK ☺ YOU!

멋지고 똑똑한 딥러닝 보아즈들 고생했어요 :D