


# Dive into DeepLearning(Preliminaries )

☼ 상태	Done
👤 작성자	 Antimony02
🕒 최종 편집 일시	@2023년 11월 20일 오후 1:00
☰ 태그	사전 조사

## 2. Preliminaries

Preliminaries는 전체적인 내용을 간단하게 접해볼 수 있는 챕터이다. 각 소분류에서는 다음과 같은 내용을 간단하게 수행한다. (i) 데이터 저장 및 조작기술, (ii) 전처리, (iii) 고차원 데이터에 대한 연산, (iv) 미적분학, (v) 미적분 계산 자동화, (vi) 확률, (vii) 문법에 대한 문서

### 2-1. Data Manipulation

데이터를 조작할 때 사용하는 라이브러리는 PYTORCH, MXNET, JAX, TENSORFLOW 등이 있고 이후로 tensorflow를 사용해 진행을 했다.

데이터 조작법

```
import tensorflow as tf

x = tf.range(12, dtype=tf.float32) # 0~11까지의 32bit float타입
tf.size(x) # 텐서 x의 크기
x.shape # 텐서 형태(ex. [1, 2] == 2, [[1, 2, ], [3, 4], [5, 6]])
tf.zeros((2, 3, 4)) # 2개의 행, 3개의 열, 4개의 행렬 을 0으로 채운다
tf.ones((2, 3, 4)) # ... 1로 채운다.
tf.random.normal(shape=[3, 4]) # ...랜덤한 값으로 채운다.
tf.constant([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]]) # 해당
x[-1], x[1:3] # 마지막 행 선택, 두 번째 및 세번째 행을 선택

X_var = tf.Variable(x) # 텐서 x의 값을 X_var로 복사
```

```

X_var[1, 2].assign(9) # 두번째 행의 세번째 열의 값을 9로 변경

X_var = tf.Variable(x)
X_var[:2, :].assign(tf.ones(X_var[:2, :].shape, dtype=tf.float32))
# X_var에서 첫번째부터 두번째 행까지, 모든 열 위치의 값이 1인 텐서를 만들
# 이 텐서를 12와 곱한 텐서를 X_var[:2, :]위치의 값들과 변경한다.

tf.exp(x) # 텐서 x를 e^x에 넣어 지수표기법으로 된 데이터의 텐서를 생성

x = tf.constant([1.0, 2, 4, 8])
y = tf.constant([2.0, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
...

각위치의 값들을 해당 연산을 한다.
...

X = tf.reshape(tf.range(12, dtype=tf.float32), (3, 2))
Y = tf.constant([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
tf.concat([X, Y], axis=0, tf.concat([X, Y], axis=1)
...

각 X, Y들을 연결한다. 이때 axis=0이면 X 오른쪽에 Y를 연결해 행의 수를 유지
axis=1이면 X 아래에 Y를 연결시켜 열의 수를 유지한다.
...

X == Y # 각 X, Y의 각 위치의 데이터들이 같은지 비교
tf.reduce_sum(x) # x의 모든 데이터들을 더한다.

a = tf.reshape(tf.range(3), (3, 1)) # 0~2의 값을 3x1의 배열에 저장
b = tf.reshape(tf.range(2), (1, 2)) # 0~1의 값을 1x2의 배열에 저장

```

어떠한 작업을 할 때 새 메모리가 할당될 수 있다. 예를 들어 다음코드와 같다.

```

before = id(Y)
Y = Y + X
id(Y) == before # False

```

코드를 보면 기존의 Y의 주소값을 before에 저장하고 연산을 한 뒤 Y의 주소값을 before와 비교했을 때 False가 나오는것을 봐서  $Y=Y+X$ 를하는 과정에서 새 메모리 주소에 할당을 한다는 것을 알 수 있다.

```
Z = tf.Variable(tf.zeros_like(Y))
print('id(Z):', id(Z)) # id(Z): 139652041257360
Z.assign(X + Y)
print('id(Z):', id(Z)) # id(Z): 139652041257360
```

assign을 통해서 Z의 데이터를 바꾸는 과정에서 id값을 확인해본 결과 동일한 주소값을 출력하는데 이는 텐서에 대한 과도한 할당을 방지하고자 메모리 사용량을 줄인것이다.

TensorFlow에서는 tf.function이 실행되기 전에 컴파일이 되고 최적화가 되는 TensorFlow 그래프 내부에서 계산을 래핑하는 데코레이터를 제공한다. 이를 통해 TensorFlow는 사용되지 않는 값을 정리하고 더 이상 필요하지 않은 이전 주소들을 재사용할 수 있다. 따라서 TensorFlow에서는 이 데코레이터를 이용해 메모리 오버헤드를 최소화 할 수 있다.

```
@tf.function
def computation(X, Y):
    Z = tf.zeros_like(Y) # 사용하지 않는 값은 제거가 된다.
    A = X + Y # 더이상 필요하지 않을 때 재사용된다.
    B = A + Y
    C = B + Y
    return C + Y

computation(X, Y)
```

현재 텐서를 NumPy 텐서로 변환하거나 그 반대로 변환하는 것은 쉽지만 메모리를 공유하지 않는다.

```
A = X.numpy()
B = tf.constant(A)
type(A), type(B)
```

```
# (numpy.ndarray, tensorflow.python.framework.ops.EagerTensor
```

```
a = tf.constant([3.5]).numpy()  
a, a.item(), float(a), int(a)  
  
# (array([3.5], dtype=float32), 3.5, 3.5, 3)
```

## 2-2. Data Preprocessing

심표로 구분된 데이터셋인 CSV파일을 읽기 위해 CSV파일을 생성하는 환경세팅

```
import os  
  
os.makedirs(os.path.join '..', 'data'), exist_ok=True)  
data_file = os.path.join '..', 'data', 'house_tiny.csv')  
with open(data_file, 'w') as f:  
    f.write(''NumRooms, RoofType, Price  
Na, NA, 127500  
2, NA, 10600  
4, Slate, 178100  
NA, NA, 140000'')
```

데이터 확인

```
import pandas as pd  
  
data = pd.read_csv(data_file)  
print(data)
```

	NumRooms	RoofType	Price
0	NaN	NaN	127500

1	2.0	NaN	106000
2	4.0	Slate	178100
3	NaN	NaN	140000

데이터를 보면 NaN이라고 되어있는 값들이 있다. 이 값들은 누락된 값들이다. 따라서 이러한 누락된 값들을 채워주는 열 기법들이 있다.

```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

	NumRooms	RoofType_Slate	RoofType_nan
0	NaN	False	True
1	2.0	False	True
2	4.0	True	False
3	NaN	False	True

출력값을 보면 RoofType열에서 어떠한 값이라도 있을 경우 RoofType\_Slate열에서 True로 출력하고, Na일 경우 RoofType\_nan에서 True를 출력하는 것을 확인할 수 있다.

따라서 이렇게 누락된 값을 이진으로 대체하는 것이다.

```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

	NumRooms	RoofType_Slate	RoofType_nan
0	3.0	False	True
1	2.0	False	True
2	4.0	True	False
3	3.0	False	True

누락된 숫자값의 경우 해당 열의 평균 값으로 채워주는 방법이 있다.

이제 inputs 의 모든 데이터들이 숫자이므로 텐서로 변환시킬 수 있다.

```
import tensorflow as tf
```

```
x = tf.constant(inputs.to_numpy(dtype=float))  
y = tf.constant(targets.to_numpy(dtype=float))  
x, y
```

```
(<tf.Tensor: shape=(4, 3), dtype=float64, numpy=  
array([[3., 0., 1.],  
       [2., 0., 1.],  
       [4., 1., 0.],  
       [3., 0., 1.]])>,  
<tf.Tensor: shape=(4, ), dtype=float64, numpy=array([127500.,
```

## 2-3. Linear Algebra

- 스칼라

```
import tensorflow as tf
```

섭씨와 화씨변환을 계산할 때 아래와 같은 식을 사용할 수 있다.

$$c = 5/9(f - 32)$$

이 식에서 c와 f는 아직 모르는 어떠한 스칼라값이다.

스칼라란 방향이 없고 오직 크기만 있는 값인데 5/9, 32모두 스칼라값이다.

텐서에 스칼라 값 하나를 넣고 아래와 같이 연산을 수행 할 수 있다.

```
x = tf.constant(3.0)  
y = tf.constant(2.0)
```

```
x + y, x * y, x / y, x ** y
```

```
(<tf.Tensor: shape=(), dtype=float32, numpy=5.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=6.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=1.5>,
 <tf.Tensor: shape=(), dtype=float32, numpy=9.0>)
```

---

- 벡터

현재는 벡터를 스칼라가 여러개 있는 배열이라고 생각할 수 있다. 벡터는 다음과 같이 텐서로 구현이 되며 메모리에 따라 임의의 길이를 가질 수 있다.

```
x = tf.range(3)
x
```

```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([0, 1, 2], d
```

---

- 행렬

행렬은 행과 열로 이루어져있는 것을 말한다. 예를들어서

a_00	a_01
a_10	a_11
a_20	a_21

이것은 3개의 행과 2개의 열로 이루어졌는 3x2 행렬이다.

이 행렬을 텐서로 구현을 하면 다음과같이 할 수 있다.

```
A = tf.reshape(tf.range(6), (3, 2)) # 0~5까지의 숫자를 3x2행렬의
A
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[0, 1],
       [2, 3],
       [4, 5]], dtype=int32)>
```

전치행렬은 행렬을 왼쪽에서 오른쪽아래로 이어지는 대각(주대각선)을 기준으로 위치가 뒤집힌행렬이다. 행렬 A를 전치(Transpose)시킨것은  $A^T$ 라 쓴다

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}, A^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{bmatrix}$$

코드에서는 다음과 같이 전치시킬 수 있다.

```
tf.transpose(A)
```

만약 어떤 행렬을 전치시켰을 때 전치시키기 전과 동일하다면 그 행렬은 주대각선을 기준으로 대칭인 정사각행렬이다.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 2 \\ 3 & 2 & 1 \end{bmatrix}, A^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 2 \\ 3 & 2 & 1 \end{bmatrix}$$

```
A = tf.constant([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == tf.transpose(A)
```

```
<tf.Tensor: shape=(3, 3), dtype=bool, numpy=
array([[True, True, True],
       [True, True, True],
       [True, True, True]])>
```



- 텐서

텐서는 다차원의 배열을 말한다. 여기서 [1, 2, 3] 이런 배열이 있는데 이것은 3차원 벡터로 1차원의 배열 형태로 표현이 된다, 하지만 텐서는 여러개의 배열형태로 표현이 되는데 이러한 부분에서 차이가 있다.

텐서는 스칼라, 벡터, 행렬 등을 포함하는 포괄적인 개념이다. 따랏 모든 벡터는 텐서이지만 모든 텐서가 벡터는 아니다.

텐서도 또한 기본 적인 연산이 가능하다.

```
A = tf.reshape(tf.range(6, dtype=tf.float32), (2, 3))
B = A# No cloning of A to B by allocating new memoryA, A + B
```

```
(<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0., 1., 2.],
       [3., 4., 5.]], dtype=float32)>,
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 0.,  2.,  4.],
       [ 6.,  8., 10.]], dtype=float32)>)
```

두 행렬의 요소별 곱을 Handamard곱이라 하는데 이또한 할 수 있다.

```
A * B
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 0.,  1.,  4.],
       [ 9., 16., 25.]], dtype=float32)>
```

텐서와 스칼라를 더하거나 곱하면 원래의 텐서와 모양이 같은 결과가 생성이 된다.

```
a = 2
X = tf.reshape(tf.range(24), (2, 3, 4))
```

```
a + X, (a * X).shape
```

```
(<tf.Tensor: shape=(2, 3, 4), dtype=int32, numpy=
array([[[ 2,  3,  4,  5],
        [ 6,  7,  8,  9],
        [10, 11, 12, 13]],

       [[14, 15, 16, 17],
        [18, 19, 20, 21],
        [22, 23, 24, 25]]], dtype=int32)>,
TensorShape([2, 3, 4]))
```

텐서안의 요소들의 합을 계산할때 다음과 같은 명령어를 사용할 수 있다.

```
x = tf.range(3, dtype=tf.float32)
x, tf.reduce_sum(x)
```

```
(<tf.Tensor: shape=(3,), dtype=float32, numpy=array([0., 1., 2.], dtype=float32)>,
<tf.Tensor: shape=(), dtype=float32, numpy=3.0>)
```

기본적으로 sum함수를 호출할 경우 모든 축을 따라 텐서가 줄어들고 결국 스칼라가 생성된다.

이때 줄어드는 축을 지정할 수있다.

```
A.shape, tf.reduce_sum(A, axis=0).shape
```

```
(TensorShape([2, 3]), TensorShape([3]))
```

이렇게 할 경우

```
[[1, 2, 3],
 [4, 5, 6],
```

```
[7, 8, 9]]
```

이 행렬에서 `[12 15 18]` 이렇게 요소들이 더해진다.

`axis=1`도 마찬가지로 열을 기준으로 요소들이 더해진다.

텐서요소들의 평균을 구하는 법으로는 요소들을 `reduce_sum` 를 사용해 더한 뒤

`tf.size(A).numpy()` 로 나눌 수 있다. 하지만 `reduce_mean` 을 사용하면 간단하게 평균을 구할 수 있다.

이 함수도 또한 `axis`인자를 사용해 특정 축을 따라 평균을 계산할 수 있다

```
tf.reduce_mean(A, axis=0), tf.reduce_sum(A, axis=0) / A.shape
```

```
(<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1.5, 2.5  
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1.5, 2.5
```

---

- 내적

내적은 동일한 위치에 있는 두 벡터의 요소들의 곱에대한 합이다. 이것은 일반적으로 두 벡터의 유사도를 측정하기 위해 사용한다.

$$A = \begin{bmatrix} 0 & 1 & 2 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

$$\langle A, B \rangle = 0 * 1 + 1 * 1 + 2 * 1 = 3$$

이때 결과가 양수일경우 두 벡터는 같은 방향을 가리키는 것이고, 0일경우 동일한 방향을 가리키고, 음수일경우 반대방향을 가리키는것이다.

---

- 행렬

행렬에서 곱을 할때는 행렬의 순서와 크기가 중요하다. 만약 행렬 A와 행렬 B를 곱한다고 할 때 A의 열의개수와 B의 행의개수가 같아야한다. 즉  $AB = BA$  가 무조건 성립하는것이 아니라는 것을 알 수 있다.

다음 두 행렬에서 AB연산을 수행해보겠다.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{21} & b_{31} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

우선 A는 3x2행렬이고 B는 2x3행렬이다. 따라서 A의 열의개수 2 = B의 행의 개수는 2이므로 곱연산이 가능하다. 연산을 하면 다음과같은 결과가 나온다.

$$AB = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{21} + a_{12}b_{22} & a_{11}b_{31} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{21} + a_{22}b_{22} & a_{21}b_{31} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{21} + a_{32}b_{22} & a_{31}b_{31} + a_{32}b_{23} \end{bmatrix}$$

행렬곱셈을 할때 단순 요소들끼리의 곱인 Hadamard곱과 혼동해서는 안된다.

---

## • 노름

노름은 길이를 측정하는 방법이다. 현재 우리가 사용하고 있는 길이는 유클리드 노름으로 L2 노름이라한다.

### 1. L2노름(유클리드)

$\|x\|_2$ 또는  $\|x\|$ 로표기한다.

L2노름에서는 벡터의 각 성분을 제곱하여 더한뒤 제곱근을 취한다.

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

### 2. L1노름(맨하튼거리)

$\|x\|_1$ 로표기한다.

L1노름에서는 벡터의 각성분의 절댓값을 더한다.

$$\|x\|_1 = |x_1| + |x_2| + \cdots + |x_n|$$

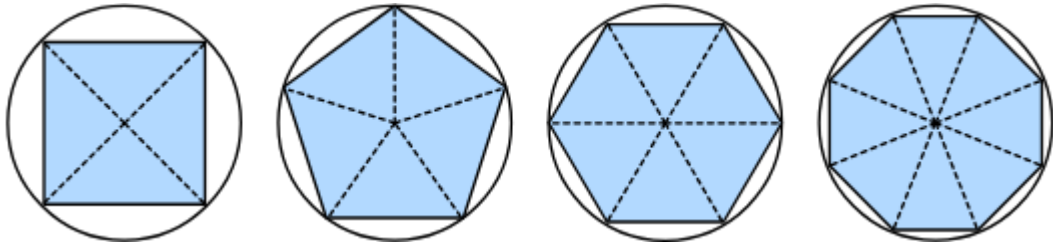
### 3.L\_Infinity

L\_Infinity에서는 벡터의 각성분의 절댓값 중 가장 큰 값을 선택한다.

$$\|x\|_{\infty} = \max(|x_1|, |x_2|, \dots, |x_n|)$$

## 2-4. Calculus

오랫동안 원의 넓이를 계산하는데에 많은 어려움이있었다. 이때 원 내부에 정점의 수가 증가하는 일련의 다각형을 만드는 아이디어를 생각해냈다.



$$n \cdot r \cdot 1/2(2\pi/n) = \pi r^2$$

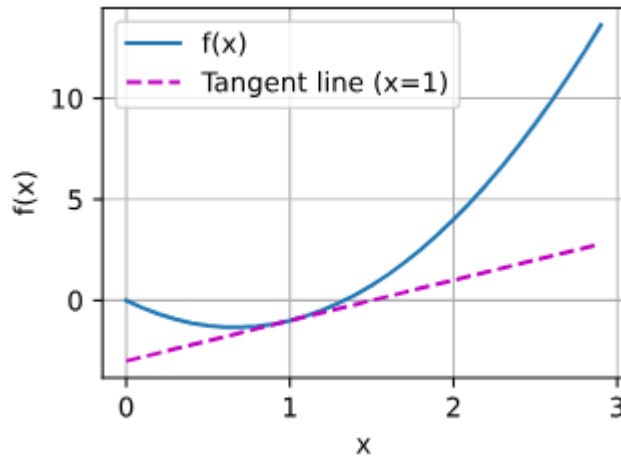
이러한 과정은 미분과 적분의 시초이다. 이러한 계산을 이용해 손실 함수를 줄이는데 사용이 된다.

## (미적분 기초 생략)

아래와 같은 명령어를 통해 함수 그래프를 만들 수 있다.

```
%matplotlib inline
import numpy as np
from matplotlib_inline import backend_inline
from d2l import tensorflow as d2l

x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2*x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent'])
```



## 2-5 Automatic 또 Differentiation

tensorflow라이브러리에서는 미분을 하는데에 도움을 줄 수 있는 기능들이 있다.

만약에  $Y=2 \cdot \langle x, x \rangle$ 를 컬럼  $x$ 에 대해 미분을 하고자한다. 처음에 변수  $x$ 를 생성하고, 초기값을 할당한다.

```
x = tf.range(4, dtype=tf.float32)
x
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([0., 1., 2., 3.], dtype=float32)>
```

$x$ 에 대한  $y$ 의 미분을 계산한 결과를 저장할 공간이 필요하다.

```
x = tf.Variable(x)
```

```
# Record all computations onto a tape
with tf.GradientTape() as t:
    y = 2 * tf.tensordot(x, x, axes=1)
y
```

이제 메소드를 호출하여  $y$ 에 대한 기울기를 계산할 수 있다.

```
x_grad = t.gradient(y, x)
x_grad
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([ 0.,  4.,  8., 12.], dtype=float32)>
```

2<x, x>의 미분값은 4x이므로 위와같이 예상과 동일하게 출력이 됐다.

```
x_grad == 4 * x
```

```
<tf.Tensor: shape=(4,), dtype=bool, numpy=array([ True,  True,  True,  True ])>
```

## 2-6 Probability and Statistics

- 확률

```
%matplotlib inline
import random
import tensorflow as tf
from tensorflow_probability import distributions as tfd
from d2l import tensorflow as d2l
```

```
num_tosses = 100
heads = sum([random.random() > 0.5 for _ in range(num_tosses)])
tails = num_tosses - heads
print("heads, tails: ", [heads, tails])
```

동전을 100번 던졌을 때 앞면이 나오는 횟수와 뒷면이 나오는 횟수를 이 코드와 같이 random함수를 이용해 시뮬레이션할 수 있다.

```
fair_probs = tf.ones(2) / 2
tfd.Multinomial(100, fair_probs).sample()
```

이렇게도 구현할 수 있다.

여기서 동전을 던졌을 때 앞이나 뒤가 나오는 확률은 1/2이다. 하지만 이러한 100번의 시뮬레이션으로는 아직 표본이 적기 때문에 확률이 1/2와 근사하지 않을 수 있다. 따라서 10000번의 던지기 시뮬레이션을 통해 다시 결과를 보면

```
counts = tfd.Multinomial(10000, fair_probs).sample()  
counts / 10000
```

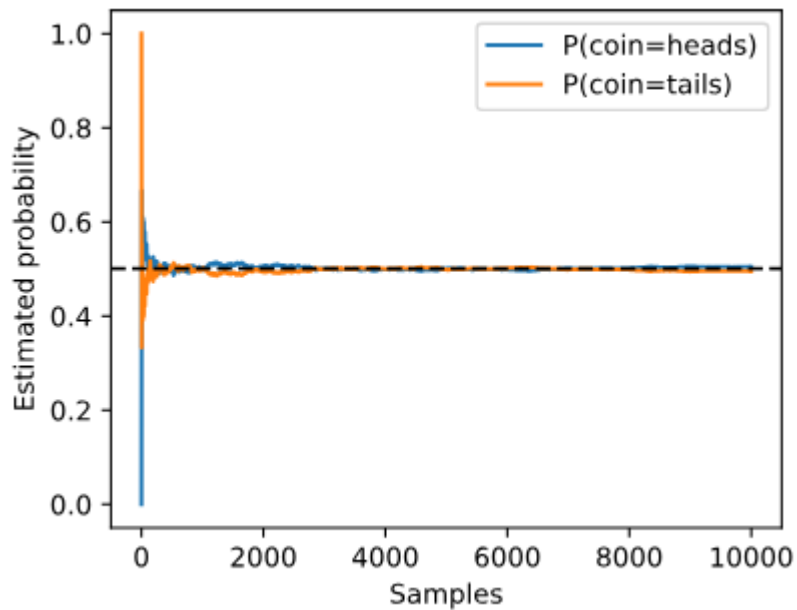
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.5019, 0.4981], dtype=float32)>
```

1/2과 근접한 확률이 나오는것을 확인할 수있다.

이처럼 표본이 많을수록 이상적인 확률에 가까워진다는 것을 예상할 수 있다. 여기서 1번 던지는것부터해서 10000번까지 증가시키면서 추정치가 어떻게 변화하는지 그래프로 표현해 보겠다.

```
counts = tfd.Multinomial(1, fair_probs).sample(10000)  
cum_counts = tf.cumsum(counts, axis=0)  
estimates = cum_counts / tf.reduce_sum(cum_counts, axis=1, keepdims=True)  
estimates = estimates.numpy()  
  
d2l.set_figsize((4.5, 3.5))  
d2l.plt.plot(estimates[:, 0], label="P(coin=heads)")  
d2l.plt.plot(estimates[:, 1], label="P(coin=tails)")  
d2l.plt.axhline(y=0.5, color='black', linestyle='dashed')  
d2l.plt.gca().set_xlabel('Samples')  
d2l.plt.gca().set_ylabel('Estimated probability')  
d2l.plt.legend();
```





## 2-7 Documentation

```
import tensorflow as tf
```

해당 라이브러리를 사용하다 어떤 클래스들이 있는지 확인하기 위해 다음과같은 명령어를 사용할 수 있다.

```
print(dir(tf.random))
```

특정함수를 사용하는 방법에 대해서 구체적인 설명을 보려면 help함수를 호출할 수 있다,

```
help(tf.ones)
```