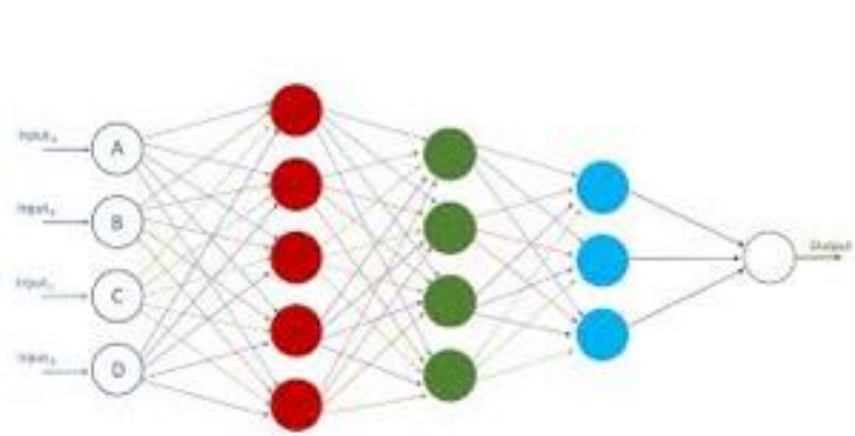
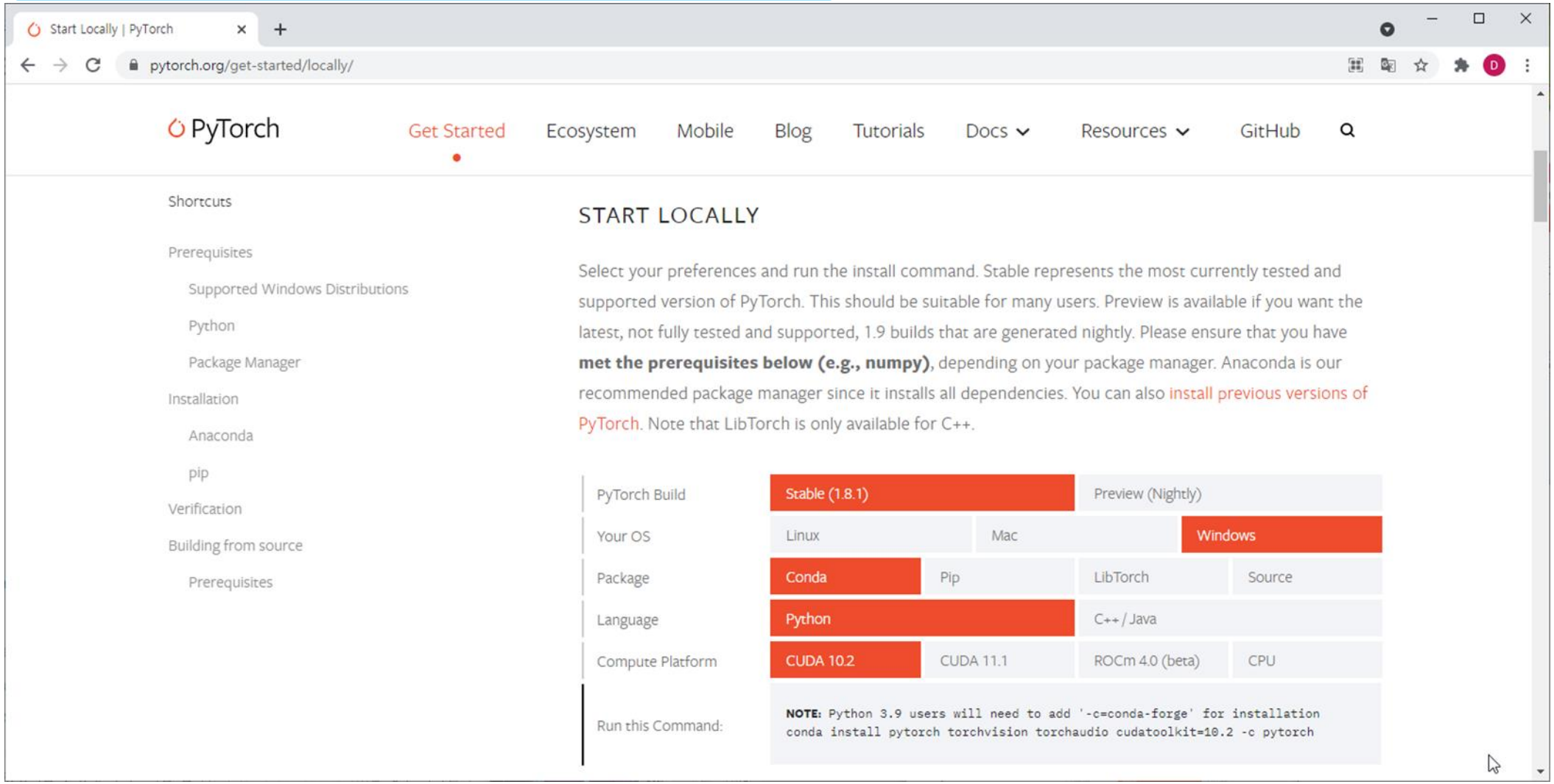


PyTorch

박경규

파이토치(PyTorch) 설치

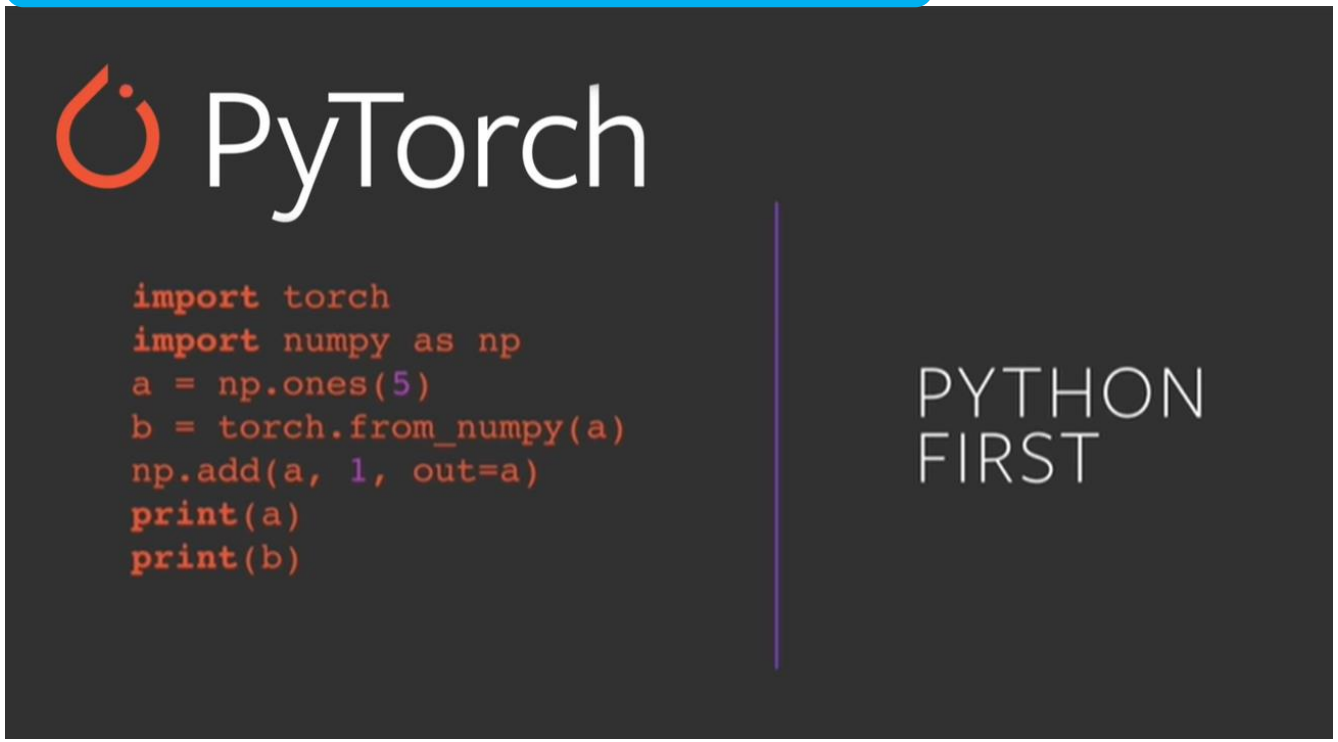
https://pytorch.org/get-started/locally/



파이토치(PyTorch)

딥러닝 프로그램을 쉽게 구현할 수 있도록 다양한 기능을 제공해주는 프레임워크입니다.

https://tutorials.pytorch.kr/



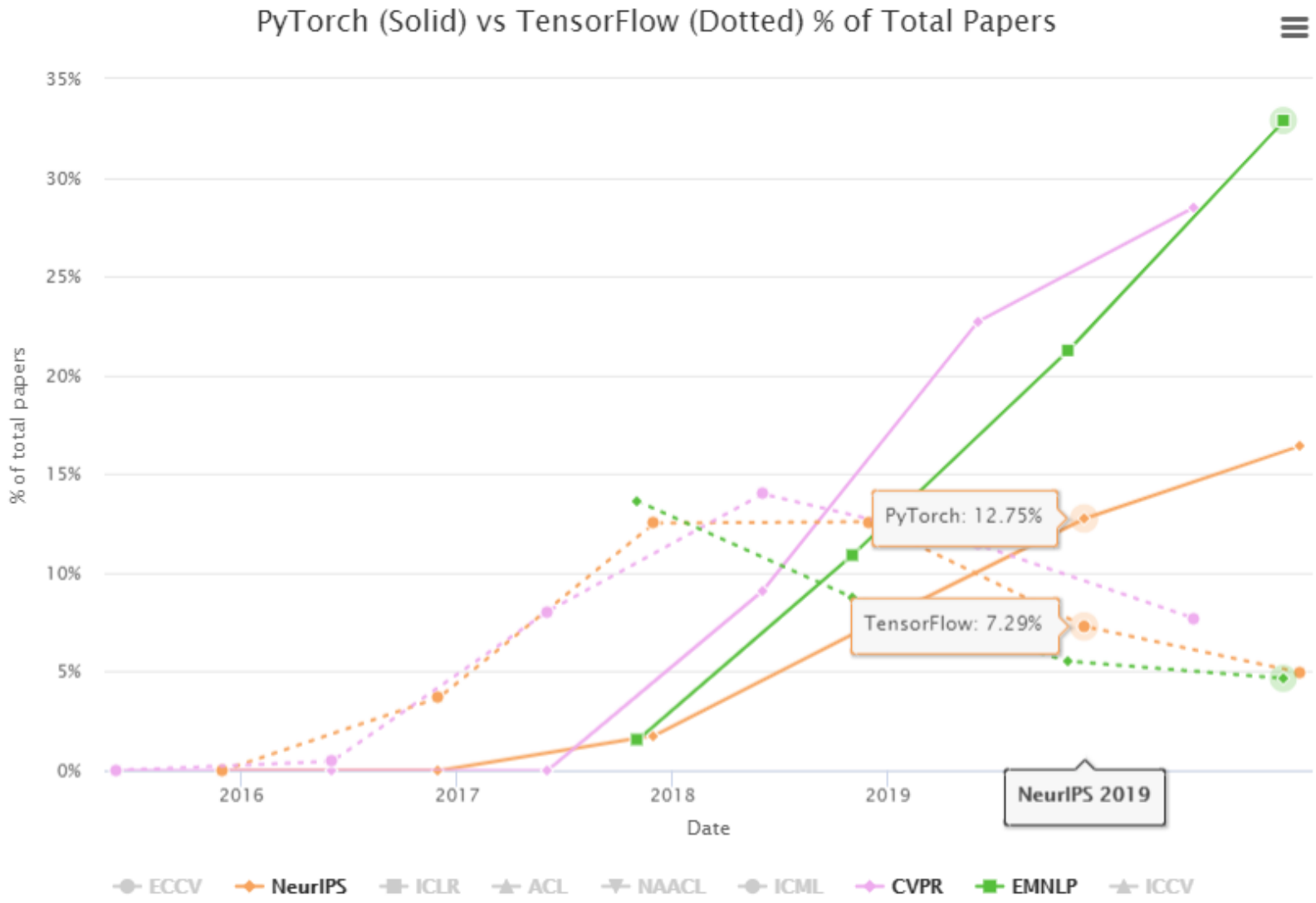
Pythonic

Great API

Dynamic Graph

PyTorch로 시작하는 딥러닝 입문 : https://wikidocs.net/book/2788

파이토치(PyTorch)



출처 : http://horace.io/pytorch-vs-tensorflow/

파이토치(PyTorch) 패키지

- torch : 메인 네임스페이스. 텐서 등의 다양한 수학 함수를 제공
  https://pytorch.org/docs/stable/torch.html

- torch.autograd : 자동 미분을 위한 함수들을 포함
  https://pytorch.org/docs/stable/autograd.html

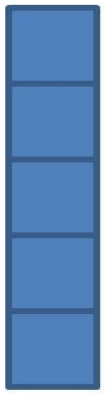
- torch.nn : 신경망을 구축하기 위한 다양한 데이터 구조나 레이어 등을 정의
  - 레이어 : CNN, RNN, LSTM 등
  - 활성화 함수: ReLU, Sigmoid 등
  - 손실 함수: MSELoss, CrossEntropyLoss 등
  https://pytorch.org/docs/stable/nn.html

- torch.optim : 파라미터 최적화 알고리즘, SGD, Adam 등
  https://pytorch.org/docs/stable/optim.html

- torch.utils.data : 미니 배치용 유틸리티 함수 포함
  https://pytorch.org/docs/stable/data.html

텐서(Tensor)

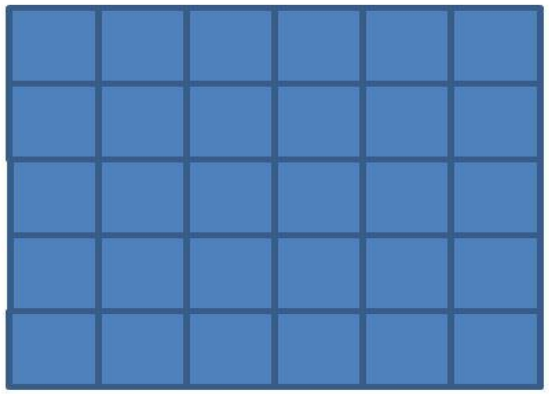
Scalar



(one dimensional tensor)

(vector of dimension [5])

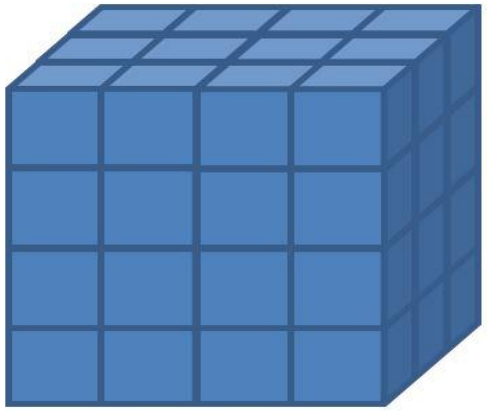
Vector



(two dimensional tensor)

(matrix of dimensions [5,6])

Matrix



(three dimensional tensor)

(tensor of dimension [4,4,3])

Tensor

파이토치 텐서(Tensor)

```
import torch

#  일반적인 파이썬 변수 사용 예
x = 3.0
y = x*x + 1
print(x, y)
```

```
3.0 10.0
```

```
# 파이토치 텐서
x = torch.tensor(3.0)
print(x)
```

```
tensor(3.)
```

```
# 텐서를 이용한 간단한 연산
y = x + 1
print(y)
```

```
tensor(4.)
```

파이토치 텐서(Tensor)

```
[1]  import torch
```

Create a 5x3 uninitialized Tensor

```
[2]  x = torch.empty(5, 3)
     print(x)
```

```
tensor([[-9.5339e+15,  3.0941e-41,  3.3631e-44],
        [ 0.0000e+00,         nan,  3.0941e-41],
        [ 1.1578e+27,  1.1362e+30,  7.1547e+22],
        [ 4.5828e+30,  1.2121e+04,  7.1846e+22],
        [ 9.2198e-39,  7.0374e+22, -3.0787e+15]])
```

Create a 5x3 randomly initializedTensor:

```
[3]  x = torch.rand(5, 3)
     print(x)
```

```
tensor([[0.5344, 0.4397, 0.4300],
        [0.7659, 0.2413, 0.0428],
        [0.8061, 0.0041, 0.0440],
        [0.5186, 0.8815, 0.3750],
        [0.2990, 0.1815, 0.4794]])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

Create a 5x3 long type with all 0s Tensor

```
[4]  x = torch.zeros(5, 3, dtype=torch.long)
     print(x)
```

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

Create directly based on data

```
[5]  x = torch.tensor([5.5, 3])
     print(x)
```

```
tensor([5.5000, 3.0000])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

You can also use existingTensorTo create,
this method will reuse the input by default Tensor Some attributes,
such as data type, unless custom data type.

```
[6]  x = x.new_ones(5, 3, dtype=torch.float64)  # The returned tensor has the same torch.dtype
     print(x)

     x = torch.randn_like(x, dtype=torch.float) # Specify new data type
     print(x)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[-0.0880, -1.3758, -1.8259],
        [-0.1816, -1.0372, -0.6719],
        [ 0.4647, -0.5224,  1.0968],
        [-0.6860, -0.8599,  0.2682],
        [ 1.9959,  0.2772, -1.5417]])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

In PyTorch, the same operation may have many forms. The following uses addition as an example.

Arithmetic operation

```
[8]  y = torch.rand(5, 3)
     print(x + y)
```

```
tensor([[ 0.8418, -1.2775, -1.2605],
        [ 0.6622, -0.9492, -0.4408],
        [ 0.5281,  0.4050,  1.3145],
        [-0.6313, -0.6020,  0.3028],
        [ 2.7758,  0.3471, -0.7824]])
```

Additive form two

```
[9]  print(torch.add(x, y))
```

```
tensor([[ 0.8418, -1.2775, -1.2605],
        [ 0.6622, -0.9492, -0.4408],
        [ 0.5281,  0.4050,  1.3145],
        [-0.6313, -0.6020,  0.3028],
        [ 2.7758,  0.3471, -0.7824]])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

You can also specify the output:

```
[10]  result = torch.empty(5, 3)
      torch.add(x, y, out=result)
      print(result)
```

```
      tensor([[ 0.8418, -1.2775, -1.2605],
              [ 0.6622, -0.9492, -0.4408],
              [ 0.5281,  0.4050,  1.3145],
              [-0.6313, -0.6020,  0.3028],
              [ 2.7758,  0.3471, -0.7824]])
```

Additive form three, inplace

Note: PyTorch operation inplace version has a suffix , *E.g x.copy*(y), x.t_()

```
[11]  # adds x to y
      y.add_(x)
      print(y)
```

```
      tensor([[ 0.8418, -1.2775, -1.2605],
              [ 0.6622, -0.9492, -0.4408],
              [ 0.5281,  0.4050,  1.3145],
              [-0.6313, -0.6020,  0.3028],
              [ 2.7758,  0.3471, -0.7824]])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

index

- We can also use NumPy-like index operations to accessTensor
- The part that needs attention is:

  The indexed result shares memory with the original data, that is, if one is modified, the other will be modified accordingly.

```
[12]  y = x[0, :]
      y += 1
      print(y)
      print(x[0, :]) # The source tensor has also been changed
```

```
tensor([ 0.9120, -0.3758, -0.8259])
tensor([ 0.9120, -0.3758, -0.8259])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

Change shape

Use view() to changeTensorshape:

```
[13]  y = x.view(15)
      z = x.view(-1, 5)  # -1 The dimension referred to can be derived from the values of other dimensions
      print(x.size(), y.size(), z.size())
```

```
torch.Size([5, 3]) torch.Size([15]) torch.Size([3, 5])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

note view() Returned new Tensor With source Tensor

Although there may be different size, But shared data Yes,

that is, if you change one of them, the other will change accordingly.

(As the name suggests, view only changes the viewing angle of this tensor,

the internal data has not changed)

```
[14]  x += 1
      print(x)
      print(y) # Also added 1
```

```
tensor([[ 1.9120,  0.6242,  0.1741],
        [ 0.8184, -0.0372,  0.3281],
        [ 1.4647,  0.4776,  2.0968],
        [ 0.3140,  0.1401,  1.2682],
        [ 2.9959,  1.2772, -0.5417]])
tensor([ 1.9120,  0.6242,  0.1741,  0.8184, -0.0372,  0.3281,  1.4647,  0.4776,
         2.0968,  0.3140,  0.1401,  1.2682,  2.9959,  1.2772, -0.5417])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

So what if we want to return a truly new copy (ie without sharing data memory)?

Pytorch also provides are shape()

You can change the shape, but this function does not guarantee that it will return a copy, so it is not recommended.

Recommended first clone Create a copy and use it again view

```
[15]  x_cp = x.clone().view(15)
      x -= 1
      print(x)
      print(x_cp)
```

```
      tensor([[ 0.9120, -0.3758, -0.8259],
              [-0.1816, -1.0372, -0.6719],
              [ 0.4647, -0.5224,  1.0968],
              [-0.6860, -0.8599,  0.2682],
              [ 1.9959,  0.2772, -1.5417]])
      tensor([ 1.9120,  0.6242,  0.1741,  0.8184, -0.0372,  0.3281,  1.4647,  0.4776,
               2.0968,  0.3140,  0.1401,  1.2682,  2.9959,  1.2772, -0.5417])
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

Another commonly used function is item(),

It can convert a scalar Tensor Converted into a Python number:

```
[16]  x = torch.randn(1)
      print(x)
      print(x.item())
```

```
tensor([0.0920])
0.09201678633689880
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

Broadcast mechanism

Earlier we saw how to treat two identical shapes Tensor Do element wise operations.
When two different shapes Tensor When calculating by element,
the broadcasting mechanism may be triggered:
first copy the elements appropriately to make these two Tensor Operate by element after the same shape.

```
[17]  x = torch.arange(1, 3).view(1, 2)
      print(x)
      y = torch.arange(1, 4).view(3, 1)
      print(y)
      print(x + y)
```

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

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

Operational memory overhead

As mentioned earlier, index operations will not open up new memory,

but like y = x + y

This kind of calculation will newly open the memory, and then y Point to new memory.

To demonstrate this, we can use Python's own id Function:

If the IDs of the two instances are the same, then their corresponding memory addresses are the same;

otherwise, they are different.

```
[18]  x = torch.tensor([1, 2])
      y = torch.tensor([3, 4])
      id_before = id(y)
      y = y + x
      print(id(y) == id_before) # False
```

```
False
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

If you want to assign the result to the original y

We can use the index introduced earlier to replace the memory.

In the following example, we put x + y The result passed [:] Write in y The corresponding memory.

```
[19]  x = torch.tensor([1, 2])
      y = torch.tensor([3, 4])
      id_before = id(y)
      y[:] = y + x
      print(id(y) == id_before) # True
```

```
      True
```

We can also use the operator full name function in out Parameter or addition operator += (That is add_())

To achieve the above effects, for example torch.add(x, y, out=y) with y += x (y.add_(x))

```
[20]  x = torch.tensor([1, 2])
      y = torch.tensor([3, 4])
      id_before = id(y)
      torch.add(x, y, out=y) # y += x, y.add_(x)
      print(id(y) == id_before) # True
```

```
      True
```

https://www.programmersought.com/article/40345471852/

파이토치 텐서(Tensor)

Tensor on GPU

How to use to() can Tensor Move between CPU and GPU (requires hardware support).

```
[21]  # The following code will only be executed on the PyTorch GPU version
      if torch.cuda.is_available():
          device = torch.device("cuda")          # GPU
          y = torch.ones_like(x, device=device)  # Create a Tensor on the GPU directly
          x = x.to(device)                       # Equivalent to .to("cuda")
          z = x + y
          print(z)
          print(z.to("cpu", torch.double))       # to()You can also change the data type at the same time
```

pytorch_tensor.ipynb

https://www.programmersought.com/article/40345471852/

자동기울기 계산(AUTOGRAD)

autograd 패키지는 Tensor의 모든 연산에 대해 자동 미분을 제공합니다.

```
x = torch.tensor(3.0, requires_grad=True)
print(x)
```

```
tensor(3., requires_grad=True)
```

```
y = (x-1)*(x-2)*(x-3)
print(y)
```

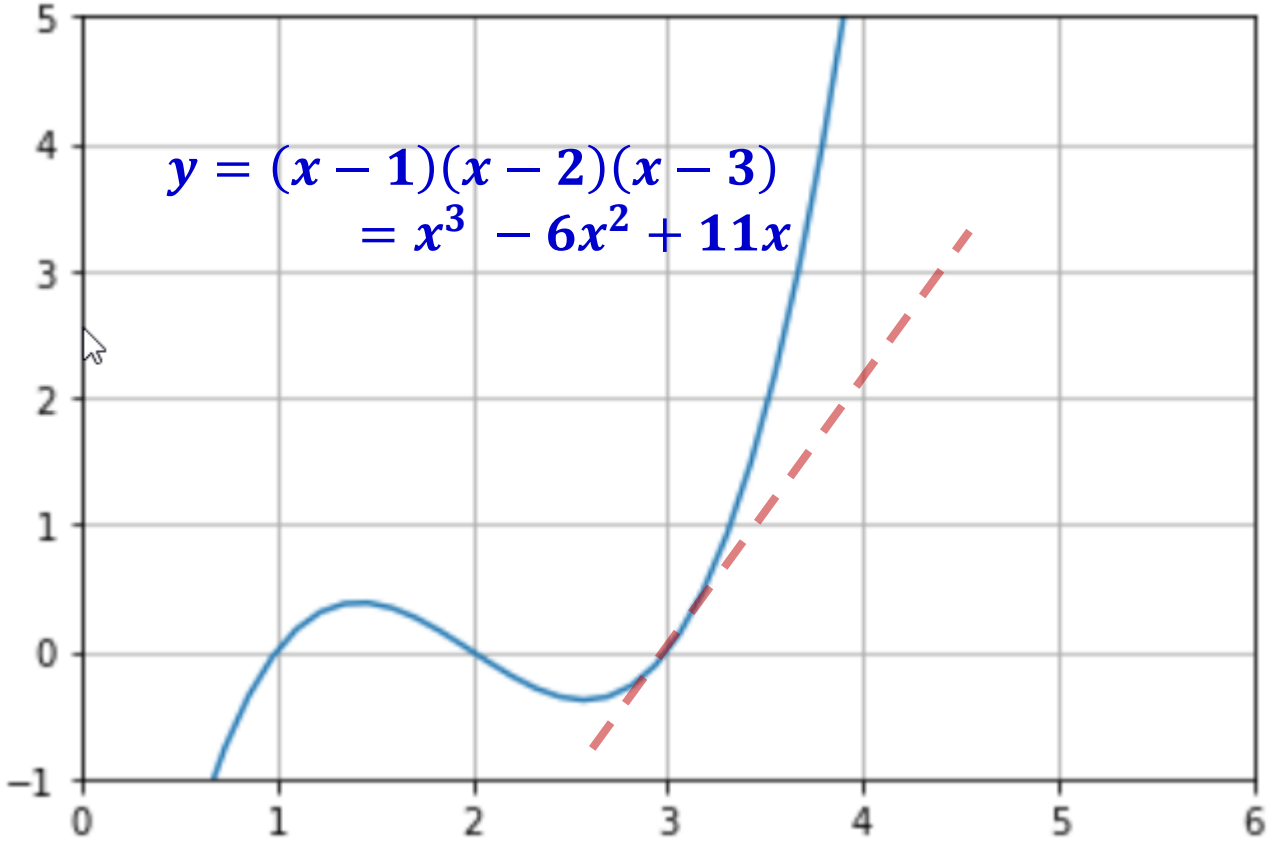
```
tensor(0., grad_fn=<MulBackward0>)
```

```
# 기울기 계산
y.backward()

# x = 3.0일 때의 기울기
print(x.grad)
```

```
tensor(2.)
```

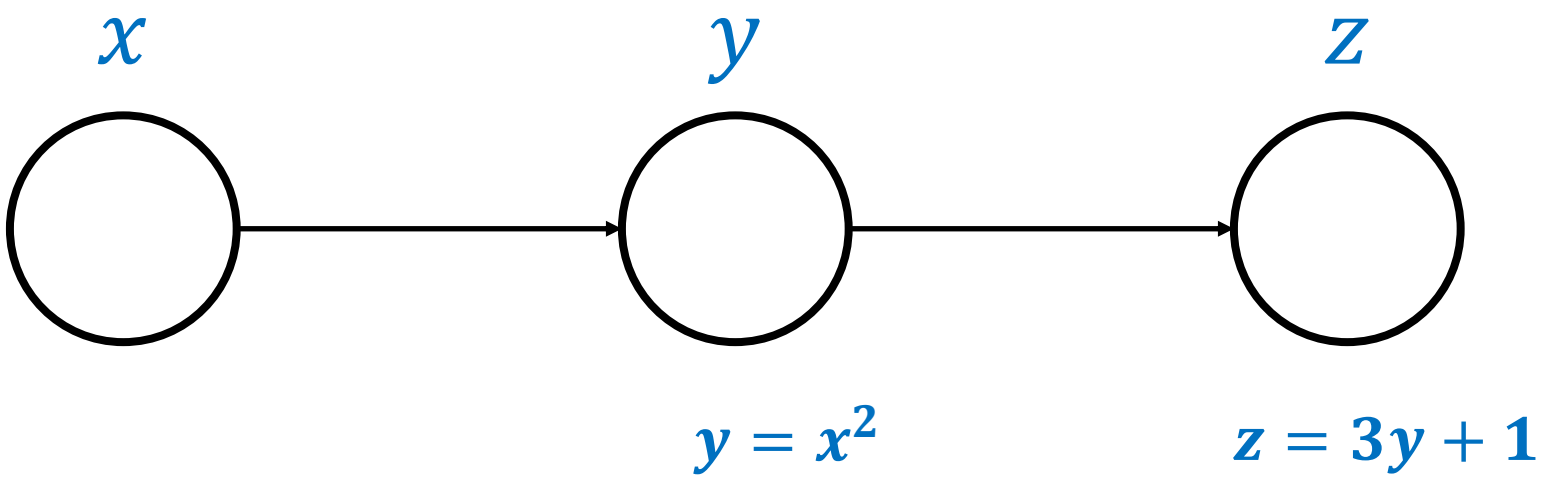
자동기울기 계산(AUTOGRAD)

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 6)
y = (x-1)*(x-2)*(x-3)
plt.plot(x, y)
plt.xlim(0, 6)
plt.ylim(-1, 5)
plt.grid(True)
plt.show()
```



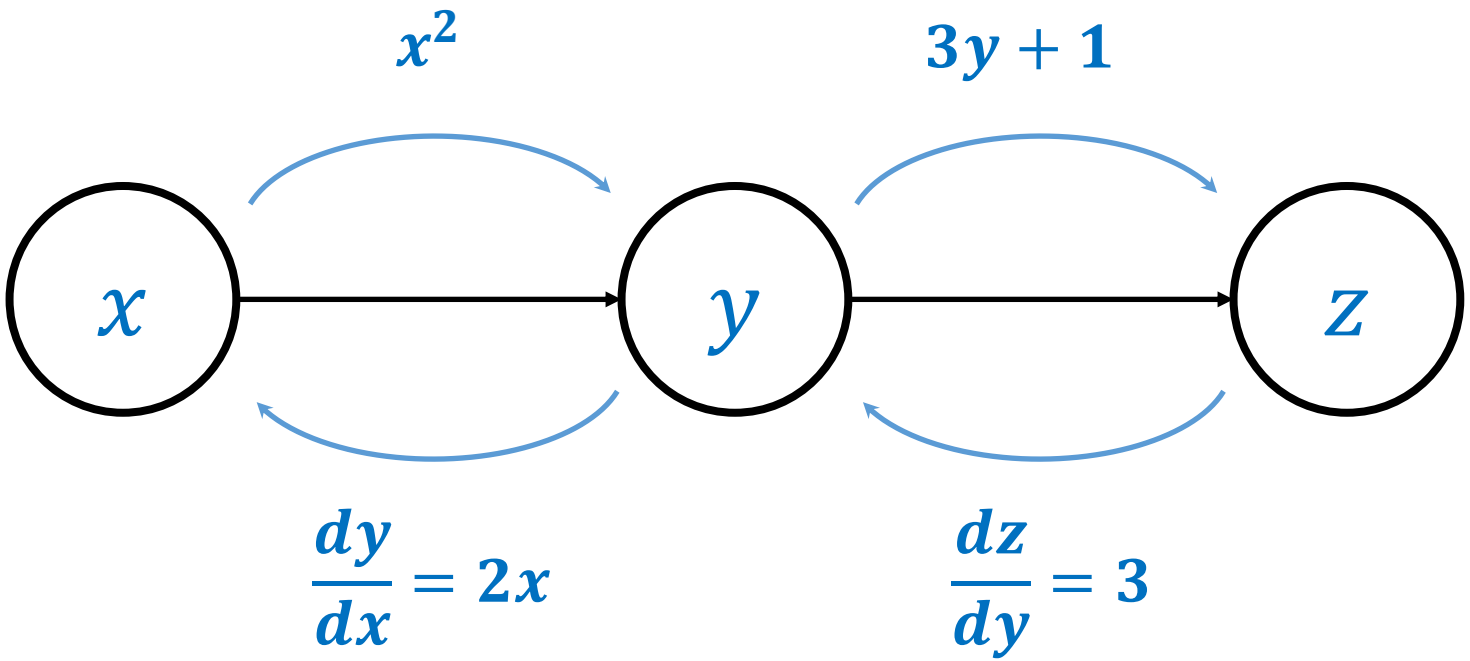
$$\frac{dy}{dx} = 3x^2 - 12x + 11$$

계산그래프(Computational Graph)



$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = 3 \cdot 2x = 6x$$

계산그래프(Computational Graph)



계산그래프(Computational Graph)

```
# x, y, z에 관련된 간단한 그래프 설정
x = torch.tensor(2.3, requires_grad=True)
y = x*x
z = 3*y + 1

# 기울기 계산
z.backward()

# 기울기 dz/dx는 x.grad에 저장됩니다.
print(x.grad)
```

```
tensor(13.8000)
```

데이터셋(Dataset), 데이터로더(DataLoader)

- 파이토치에서는 데이터셋을 좀 더 쉽게 다룰 수 있도록 유용한 도구로서

  torch.utils.data.Dataset과 torch.utils.data.DataLoader를 제공합니다.

- 이를 사용하면 미니 배치 학습, 데이터 셔플(shuffle), 병렬 처리까지

  간단히 수행할 수 있습니다.

- 기본적인 사용 방법은 Dataset을 정의하고, 이를 DataLoader에 전달하는 것입니다.

파이토치(PyTorch) 모델링 절차

1. nn.Module 클래스를 상속받아 모델 아키텍처 클래스 선언
2. 모델 클래스 객체 생성
3. SGD 또는 Adam 등의 옵티마이저를 생성하고, 생성한 모델의 파라미터를 최적화 대상으로 등록
4. 데이터로더로 미니배치를 구성하여 피드포워드 계산그래프 생성
5. 손실함수에 결과값(predicted value)과 타깃값(target value)을 입력하고 손실값 계산
6. 손실에 대해 backward() 호출 → 계산그래프 텐서들의 기울기(gradient)가 채워짐
7. 3번의 옵티마이저에서 step()을 호출하여 경사하강법 1스텝 수행
8. 4번으로 돌아가 수렴조건이 만족할 때까지 반복 수행

붓꽃(Iris) 품종 분류 모델 구현

setosa

versicolor

virginica

붓꽃(Iris) 품종 분류 모델 구현

Iris 데이터셋

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

붓꽃(Iris) 품종 분류 모델 구현

데이터 로드

[1]
```
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()
X = iris['data']
y = iris['target']
names = iris['target_names']
feature_names = iris['feature_names']
```

[2]
```
df = pd.DataFrame(iris.data)
df.columns = iris.feature_names
df['label'] = iris.target
```

[3]
```
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

붓꽃(Iris) 품종 분류 모델 구현

■ 입력과 레이블(Label) 데이터 분리

```
[4]  X = df.drop('label', axis=1).to_numpy()
     Y = df['label'].to_numpy().reshape((-1,1))
```

```
[5]  X.shape
```

```
     (150, 4)
```

■ 입력 데이터 정규화

```
[7]  from sklearn.preprocessing import StandardScaler

     scaler = StandardScaler()
     X_scaled = scaler.fit_transform(X)
```

■ 훈련데이터셋과 테스트데이터셋으로 분리

```
[8]  from sklearn.model_selection import train_test_split

     X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)
```

붓꽃(Iris) 품종 분류 모델 구현

■ 데이터셋 클래스

```
[9]  import torch
     from torch import nn, optim
     import torch.nn.functional as F
     from torch.utils.data import DataLoader, Dataset

     class TensorData(Dataset):
         def __init__(self, x_data, y_data):
             self.x_data = torch.FloatTensor(x_data)
             self.y_data = torch.LongTensor(y_data)
             self.len = self.y_data.shape[0]

         def __getitem__(self, index):
             return self.x_data[index], self.y_data[index]

         def __len__(self):
             return self.len
```

붓꽃(Iris) 품종 분류 모델 구현

■ 데이터 로더

```
[10]  train_ds = TensorData(X_train, Y_train)
      trainloader = torch.utils.data.DataLoader(train_ds, batch_size=16, shuffle=True)

      test_ds = TensorData(X_test, Y_test)
      testloader = torch.utils.data.DataLoader(test_ds, batch_size=16, shuffle=False)
```

```
[11]  test_ds[0]
```

```
      (tensor([ 0.3110, -0.5924,  0.5354,  0.0009]), tensor(1))
```

붓꽃(Iris) 품종 분류 모델 구현

■ 신경망 모델 클래스

```
[12]  class Classifier(nn.Module):
          def __init__(self):
              super(Classifier, self).__init__()
              self.fc1 = nn.Linear(4, 200)
              self.fc2 = nn.Linear(200, 100)
              self.fc3 = nn.Linear(100, 3)
              self.softmax = nn.Softmax(dim=1)

          def forward(self, X):
              X = F.relu(self.fc1(X))
              X = self.fc2(X)
              X = self.fc3(X)
              X = self.softmax(X)

              return X
```

붓꽃(Iris) 품종 분류 모델 구현

■ 신경망 모델 객체 생성

```
[13]  model = Classifier()
```

■ 손실함수, 옵티마이저

```
[14]  criterion = nn.CrossEntropyLoss()
```

■ 옵티마이저를 생성하고, 생성한 모델의 파라미터를 최적화 대상으로 등록

```
[15]   optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-7)
```

붓꽃(Iris) 품종 분류 모델 구현

■ 신경망 모델 훈련(학습)

```
[16]  epochs = 10

      for epoch in range(epochs):
          print(f'### epoch {epoch+1} ################')
          for i, data in enumerate(trainloader, 0):
              input, target = data
              optimizer.zero_grad()
              pred = model(input)
              loss = criterion(pred, target)
              loss.backward()
              optimizer.step()
              print(f'{i+1} : Loss {loss}')
```

```
### epoch 1 ################
1 : Loss 1.096687078475952
2 : Loss 1.068744659423828
3 : Loss 1.017545700073242
```

붓꽃(Iris) 품종 분류 모델 구현

■ 신경망 모델 평가

```
[17] correct = 0
     with torch.no_grad():
       for i, data in enumerate(test_ds):
         label = data[1].numpy()
         output = model.forward(data[0].reshape(1,-1))
         pred = output.argmax().item()

         if label == pred:
           correct += 1

     print(f'정확도 : {correct/len(test_ds)*100:.2f}%')
```

```
정확도 : 95.56%
```

Thank you