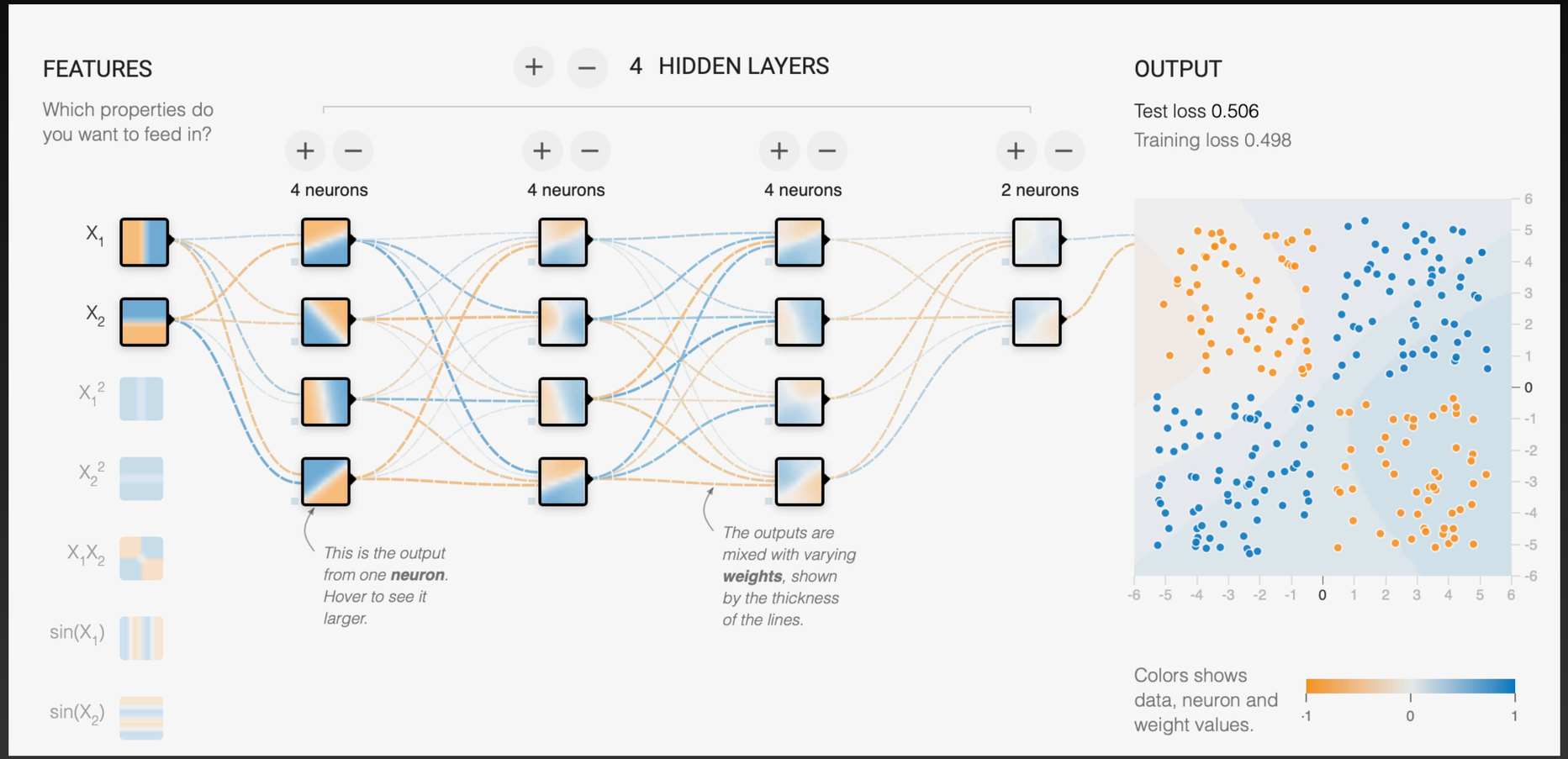
A Crash Course to PyTorch

What is PyTorch?

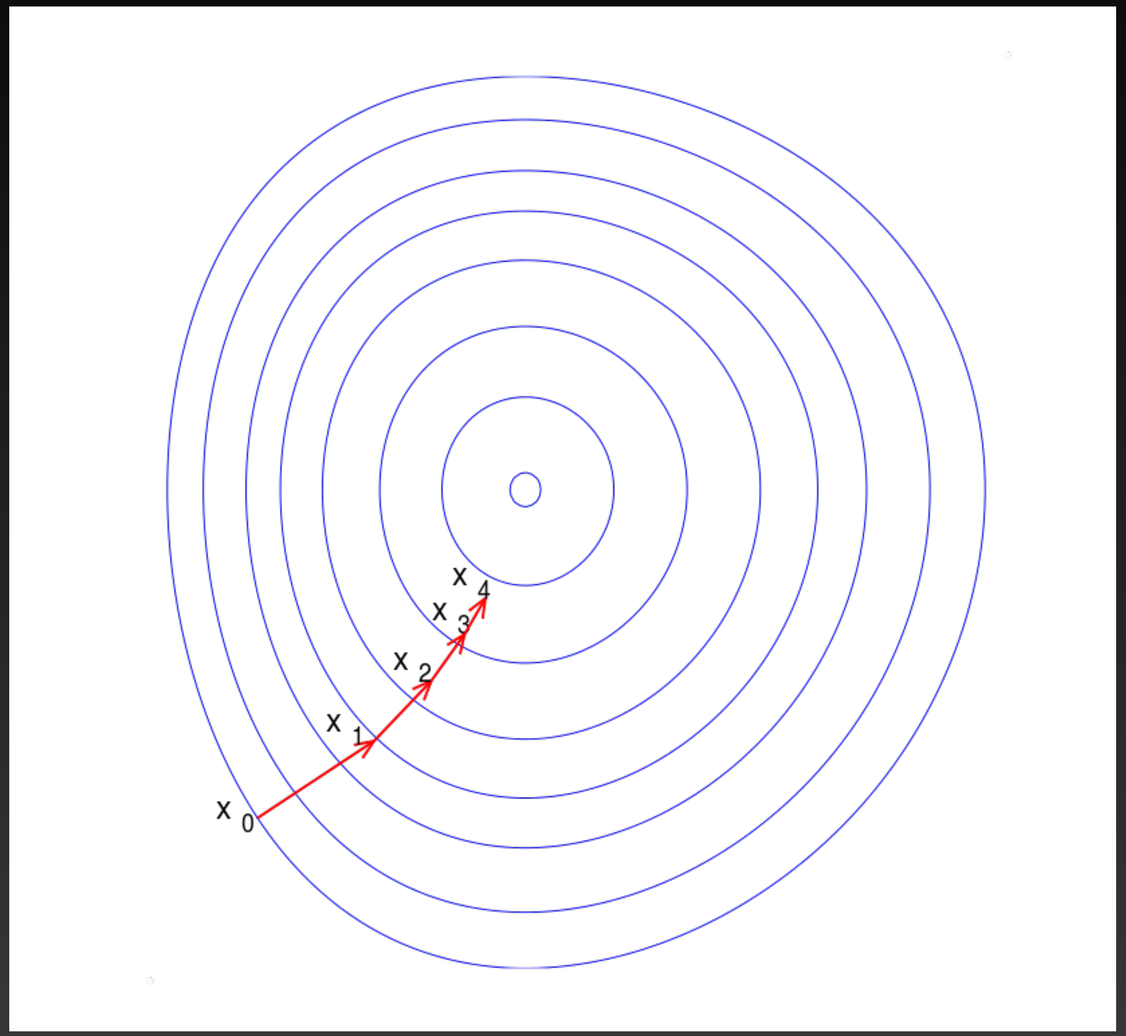
- PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR). It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.
- PyTorch provides two high-level features:
  - Tensor computing (like NumPy) with strong acceleration via *graphics processing units (GPU)*
  - Deep neural networks built on a type-based *automatic differentiation system*

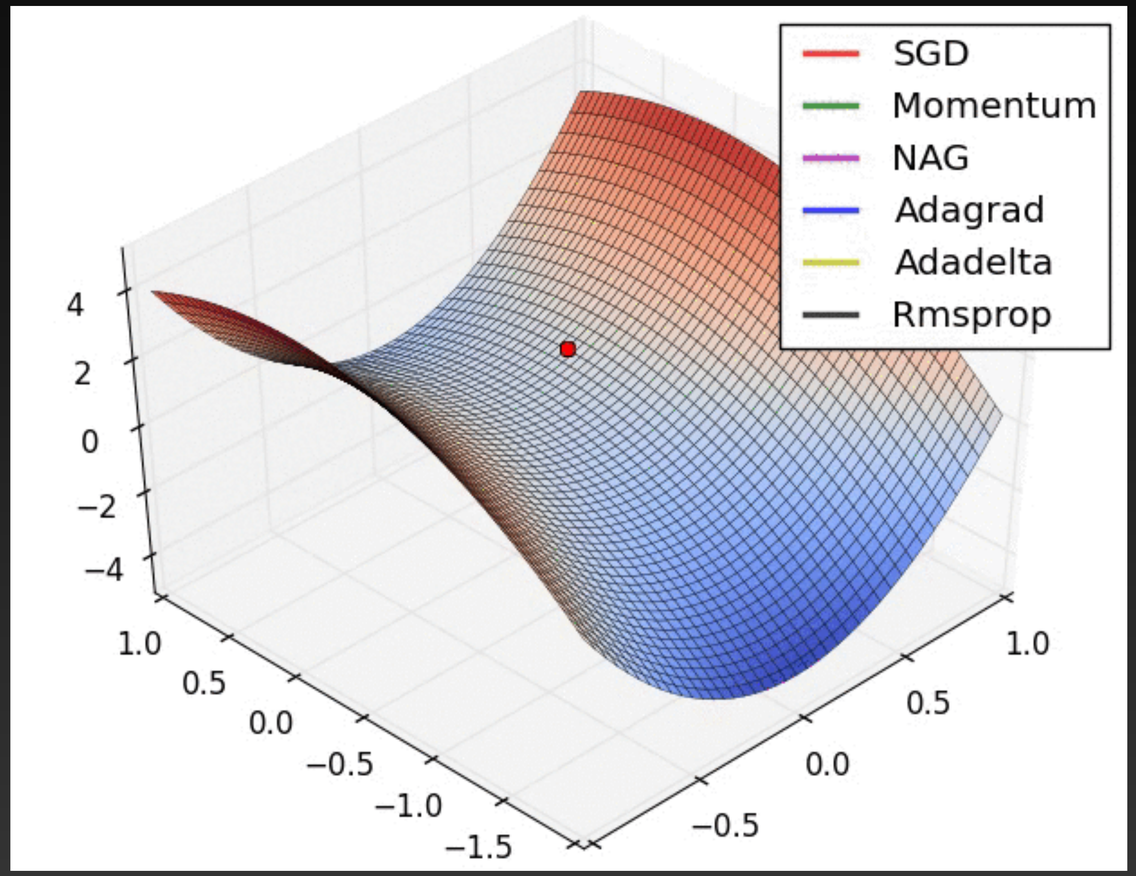
Neural Network and Gradient Descent



https://playground.tensorflow.org/

Neural Network and Gradient Descent





NumPy

- NumPy (pronounced (NUM-py)) is a library for the Python programming language, adding support for large, *multi-dimensional arrays and matrices*, along with a large collection of high-level mathematical functions to operate on these arrays.

NumPy: Installation

CONDA

If you use `conda`, you can install NumPy from the `defaults` or `conda-forge` channels:

```
# Best practice, use an environment rather than install in the base env
conda create -n my-env
conda activate my-env
# If you want to install from conda-forge
conda config --env --add channels conda-forge
# The actual install command
conda install numpy
```

PIP

If you use `pip`, you can install NumPy with:

```
pip install numpy
```

NumPy: Beginning

An example

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class 'numpy.ndarray'>
```

NumPy: Beginning

Array Creation

There are several ways to create arrays.

For example, you can create an array from a regular Python list or tuple using the `array` function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

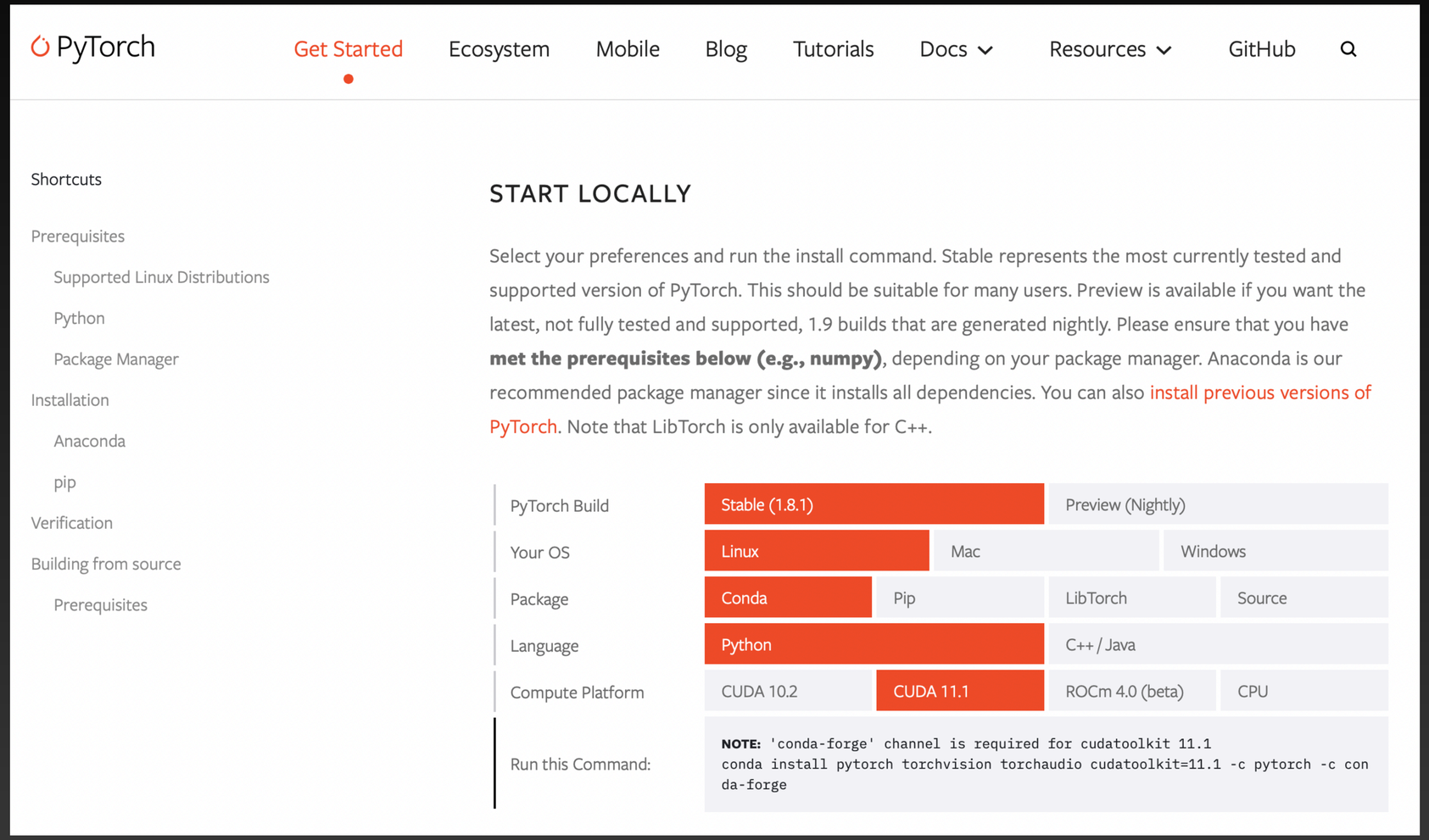
A frequent error consists in calling `array` with multiple arguments, rather than providing a single sequence as an argument.

```
>>> a = np.array(1,2,3,4)    # WRONG
Traceback (most recent call last):
  ...
TypeError: array() takes from 1 to 2 positional arguments but 4 were given
>>> a = np.array([1,2,3,4])  # RIGHT
```

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be specifie
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                                 # uninitialized
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],  # may vary
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

PyTorch: Installation



PyTorch: Beginning

```
import torch
import numpy as np
```

Initializing a Tensor

Tensors can be initialized in various ways. Take a look at the following examples:

Directly from data

Tensors can be created directly from data. The data type is automatically inferred.

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

From a NumPy array

Tensors can be created from NumPy arrays (and vice versa - see Bridge with NumPy).

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

PyTorch: Beginning

```
import torch
import numpy as np
```

Initializing a Tensor

Tensors can be initialized in various ways. Take a look at the following examples:

Directly from data

Tensors can be created directly from data. The data type is automatically inferred.

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

From a NumPy array

Tensors can be created from NumPy arrays (and vice versa - see Bridge with NumPy).

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

Attributes of a Tensor

Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
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}")
```

Out:

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

PyTorch: Beginning

```
cuda = torch.device('cuda')     # Default CUDA device
cuda0 = torch.device('cuda:0')
cuda2 = torch.device('cuda:2')  # GPU 2 (these are 0-indexed)

x = torch.tensor([1., 2.], device=cuda0)
# x.device is device(type='cuda', index=0)
y = torch.tensor([1., 2.]).cuda()
# y.device is device(type='cuda', index=0)

with torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.tensor([1., 2.], device=cuda)

    # transfers a tensor from CPU to GPU 1
    b = torch.tensor([1., 2.]).cuda()
    # a.device and b.device are device(type='cuda', index=1)

    # You can also use ``Tensor.to`` to transfer a tensor:
    b2 = torch.tensor([1., 2.]).to(device=cuda)
    # b.device and b2.device are device(type='cuda', index=1)

    c = a + b
    # c.device is device(type='cuda', index=1)

    z = x + y
    # z.device is device(type='cuda', index=0)

    # even within a context, you can specify the device
    # (or give a GPU index to the .cuda call)
    d = torch.randn(2, device=cuda2)
    e = torch.randn(2).to(cuda2)
    f = torch.randn(2).cuda(cuda2)
    # d.device, e.device, and f.device are all device(type='cuda', index=2)
```

PyTorch: Beginning

torch.device

CLASS `torch.device`

A `torch.device` is an object representing the device on which a `torch.Tensor` is or will be allocated.

The `torch.device` contains a device type (`'cpu'` or `'cuda'`) and optional device ordinal for the device type. If the device ordinal is not present, this object will always represent the current device for the device type, even after `torch.cuda.set_device()` is called; e.g., a `torch.Tensor` constructed with device `'cuda'` is equivalent to `'cuda:X'` where X is the result of `torch.cuda.current_device()`.

A `torch.Tensor`'s device can be accessed via the `Tensor.device` property.

A `torch.device` can be constructed via a string or via a string and device ordinal

Via a string:

```
>>> torch.device('cuda:0')
device(type='cuda', index=0)

>>> torch.device('cpu')
device(type='cpu')

>>> torch.device('cuda')  # current cuda device
device(type='cuda')
```

Via a string and device ordinal:

```
>>> torch.device('cuda', 0)
device(type='cuda', index=0)

>>> torch.device('cpu', 0)
device(type='cpu', index=0)
```

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
  tensor = tensor.to('cuda')
```

NOTE

Methods which take a device will generally accept a (properly formatted) string or (legacy) integer device ordinal, i.e. the following are all equivalent:

```
>>> torch.randn((2,3), device=torch.device('cuda:1'))
>>> torch.randn((2,3), device='cuda:1')
>>> torch.randn((2,3), device=1)  # legacy
```

PyTorch: Your First Network

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt

# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-
```

```
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print("Shape of X [N, C, H, W]: ", X.shape)
    print("Shape of y: ", y.shape, y.dtype)
    break
```

```
Shape of X [N, C, H, W]:  torch.Size([64, 1, 28, 28])
Shape of y:  torch.Size([64]) torch.int64
```

PyTorch: Your First Network

```python
# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using {} device".format(device))

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

```
Using cuda device
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
    (5): ReLU()
  )
)
```

PyTorch: Your First Network

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

def test(dataloader, model):
    size = len(dataloader.dataset)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= size
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")

epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model)
print("Done!")
```

Important!

```
Epoch 1
-------------------------------
loss: 2.310126  [    0/60000]
loss: 2.300893  [ 6400/60000]
loss: 2.295896  [12800/60000]
loss: 2.287010  [19200/60000]
loss: 2.273310  [25600/60000]
loss: 2.288515  [32000/60000]
loss: 2.263278  [38400/60000]
loss: 2.270398  [44800/60000]
loss: 2.265662  [51200/60000]
loss: 2.249428  [57600/60000]
Test Error:
 Accuracy: 35.3%, Avg loss: 0.035290

Epoch 2
-------------------------------
loss: 2.273786  [    0/60000]
loss: 2.253283  [ 6400/60000]
```

- For a more complete code structure, please refer to *https://github.com/junyanz/CycleGAN*

PyTorch: Your First Network

Saving Models

A common way to save a model is to serialize the internal state dictionary (containing the model parameters).

```
torch.save(model.state_dict(), "model.pth")
print("Saved PyTorch Model State to model.pth")
```

Out:

```
Saved PyTorch Model State to model.pth
```

Loading Models

The process for loading a model includes re-creating the model structure and loading the state dictionary into it.

```
model = NeuralNetwork()
model.load_state_dict(torch.load("model.pth"))
```

Advanced: Indexing

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

Use x = x.reshape(2, 5) in PyTorch

```
>>> x[0]
array([0, 1, 2, 3, 4])
```

```
>>> x[0][2]
2
```

So note that `x[0,2] = x[0][2]` though the second case is more inefficient as a new temporary array is created after the first index that is subsequently indexed by 2.

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13],
       [21, 24, 27]])
```

Note that slices of arrays do not copy the internal array data but only produce new views of the original data. This is different from list or tuple slicing and an explicit `copy()` is recommended if the original data is not required anymore.

Indexing in NumPy and PyTorch are similar

Advanced: Indexing

The ellipsis syntax maybe used to indicate selecting in full any remaining unspecified dimensions. For example:

```
>>> z = np.arange(81).reshape(3,3,3,3)
>>> z[1,...,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

This is equivalent to:

```
>>> z[1,:,:,2]
array([[29, 32, 35],
       [38, 41, 44],
       [47, 50, 53]])
```

```
>>> y.shape
(5, 7)
>>> y[:,np.newaxis,:].shape      # y[:, None. :] works for both PyTorch and NumPy
(5, 1, 7)
```

Note that there are no new elements in the array, just that the dimensionality is increased. This can be handy to combine two arrays in a way that otherwise would require explicitly reshaping operations. For example:

Advanced: Indexing

Indexing array with array

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
```

```
>>> x[np.array([[1,1],[2,3]])]
array([[9, 9],
       [8, 7]])
```

Generally speaking, what is returned when index arrays are used is an array with the same shape as the index array, but with the type and values of the array being indexed. As an example, we can use a multidimensional index array instead:

Advanced: Indexing

Indexing multi-dimensional arrays

```
>>> y = np.arange(35).reshape(5,7)
```

```
>>> y[np.array([0,2,4]), np.array([0,1,2])]
array([ 0, 15, 30])
```

```
>>> y[np.array([0,2,4]), np.array([0,1])]
<type 'exceptions.ValueError'>: shape mismatch: objects cannot be
broadcast to a single shape
```

```
>>> y[np.array([0,2,4]), 1]
array([ 1, 15, 29])
```

```
>>> y[np.array([0,2,4])]
array([[ 0,  1,  2,  3,  4,  5,  6],
       [14, 15, 16, 17, 18, 19, 20],
       [28, 29, 30, 31, 32, 33, 34]])
```

In general, the shape of the resultant array will be the concatenation of the shape of the index array (or the shape that all the index arrays were broadcast to) with the shape of any unused dimensions (those not indexed) in the array being indexed.

Advanced: Indexing

Indexing with boolean or “mask” index arrays

```
>>> b = y>20
>>> y[b]
array([21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
>>> b[:,5] # use a 1-D boolean whose first dim agrees with the first dim of y
array([False, False, False,  True,  True])
>>> y[b[:,5]]
array([[21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
>>> x = np.arange(30).reshape(2,3,5)
>>> x
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14]],
       [[15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]])
>>> b = np.array([[True, True, False], [False, True, True]])
>>> x[b]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
```

In general, when the boolean array has fewer dimensions than the array being indexed, this is equivalent to y[b, …]

Advanced: Indexing

Assigning values to indexed arrays

```
>>> x = np.arange(10)
>>> x[2:7] = 1
```

```
>>> x[2:7] = np.arange(5)
```

```
>>> x = np.arange(0, 50, 10)
>>> x
array([ 0, 10, 20, 30, 40])
>>> x[np.array([1, 1, 3, 1])] += 1
>>> x
array([ 0, 11, 20, 31, 40])
```

Where people expect that the 1st location will be incremented by 3. In fact, it will only be incremented by 1. The reason is because a new array is extracted from the original (as a temporary) containing the values at 1, 1, 3, 1, then the value 1 is added to the temporary, and then the temporary is assigned back to the original array. Thus the value of the array at x[1]+1 is assigned to x[1] three times, rather than being incremented 3 times.

Advanced: Broadcasting

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```

?

Advanced: Broadcasting

- Each tensor has at least one dimension.
- When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

```
>>> x=torch.empty(5,7,3)
>>> y=torch.empty(5,7,3)
# same shapes are always broadcastable (i.e. the above rules always hold)

>>> x=torch.empty((0,))
>>> y=torch.empty(2,2)
# x and y are not broadcastable, because x does not have at least 1 dimension

# can line up trailing dimensions
>>> x=torch.empty(5,3,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are broadcastable.
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dimension doesn't exist

# but:
>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are not broadcastable, because in the 3rd trailing dimension 2 != 3
```

Advanced: Broadcasting

```
# can line up trailing dimensions to make reading easier
>>> x=torch.empty(5,1,4,1)
>>> y=torch.empty(  3,1,1)
>>> (x+y).size()
torch.Size([5, 3, 4, 1])

# but not necessary:
>>> x=torch.empty(1)
>>> y=torch.empty(3,1,7)
>>> (x+y).size()
torch.Size([3, 1, 7])

>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(3,1,1)
>>> (x+y).size()
RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton
dimension 1
```

Advanced: Autograd

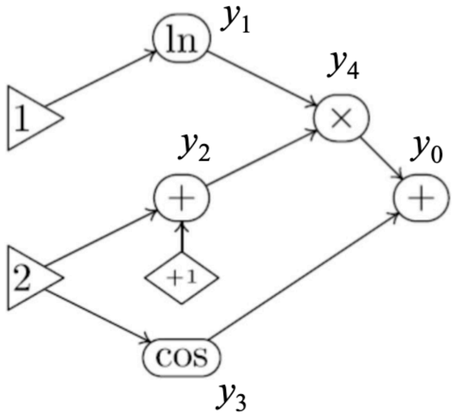
Analysis in R^n

Seppo Linnainmaa

- Automatic differentiation

$$y_o = f(x_1, x_2) = (1 + x_2) \ln x_1 + \cos x_2.$$

expression DAG



(i) $y_{0,1} = y_{3,1}\partial y_0/\partial y_3 + y_{4,1}\partial y_0/\partial y_4,$
(ii) $y_{3,1} = 0,$
(iii) $y_{4,1} = y_{1,1}\partial y_4/\partial y_1 + y_{2,1}\partial y_4/\partial y_2,$
(iv) $y_{1,1} = x_1^{-1},$
(v) $y_{2,1} = 0,$
(vi) $y_{0,2} = y_{3,2}\partial y_0/\partial y_3 + y_{4,2}\partial y_0/\partial y_4,$
(vii) $y_{3,2} = -\sin x_2,$
(viii) $y_{4,2} = y_{1,2}\partial y_4/\partial y_1 + y_{2,2}\partial y_4/\partial y_2,$
(ix) $y_{1,2} = 0,$
(x) $y_{2,2} = 1,$

Two different differentiations: $y_{i,j} = \partial y_i/\partial x_j$, with $j = 1, 2$, and $\partial y_i/\partial y_k$.

$$\partial y_0/\partial y_3 = 1, \quad \partial y_0/\partial y_4 = 1, \quad \partial y_4/\partial y_1 = y_2, \quad \partial y_4/\partial y_2 = y_1.$$

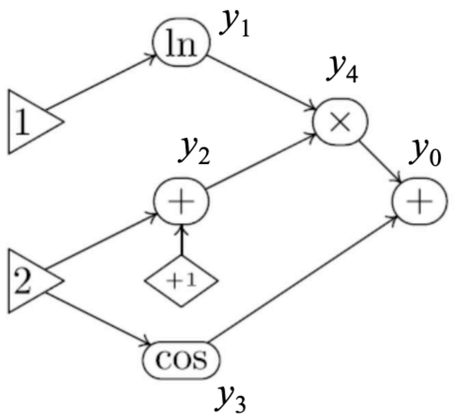
$$y_{0,1} \rightsquigarrow \{(\text{iv}), (\text{v}), (\text{ii})\}, (\text{iii}), (\text{i}), \quad y_{0,2} \rightsquigarrow \{(\text{vii}), (\text{ix}), (\text{x})\}, (\text{viii}), (\text{vi})$$

Credit: Zhouchen Lin

Advanced: Autograd

Analysis in R^n

- Automatic differentiation



$$\frac{\partial y_o}{\partial x_i} = \sum_{j \in \mathrm{pa}(o)} \frac{\partial y_o}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

output node

$$\frac{\partial y_j}{\partial x_i} = \sum_{k \in \mathrm{pa}(j)} \frac{\partial y_j}{\partial y_k} \frac{\partial y_k}{\partial x_i}.$$

intermediate node

$$\frac{\partial y_k}{\partial x_i} = \begin{cases} \dfrac{\partial y_k}{\partial x_i}, & k \in \mathrm{ch}(i), \\ 0, & \text{otherwise.} \end{cases}$$

connect-to-input node

Credit: Zhouchen Lin

Advanced: Autograd in PyTorch

Every Tensor has a flag: `requires_grad` that allows for fine grained exclusion of subgraphs from gradient computation and can increase efficiency.

`requires_grad`

If there's a single input to an operation that requires gradient, its output will also require gradient. Conversely, only if all inputs don't require gradient, the output also won't require it. Backward computation is never performed in the subgraphs, where all Tensors didn't require gradients.

```
>>> x = torch.randn(5, 5)  # requires_grad=False by default
>>> y = torch.randn(5, 5)  # requires_grad=False by default
>>> z = torch.randn((5, 5), requires_grad=True)
>>> a = x + y
>>> a.requires_grad
False
>>> b = a + z
>>> b.requires_grad
True
```

Advanced: Autograd in PyTorch

This is especially useful when you want to freeze part of your model, or you know in advance that you're not going to use gradients w.r.t. some parameters. For example if you want to finetune a pretrained CNN, it's enough to switch the `requires_grad` flags in the frozen base, and no intermediate buffers will be saved, until the computation gets to the last layer, where the affine transform will use weights that require gradient, and the output of the network will also require them.

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

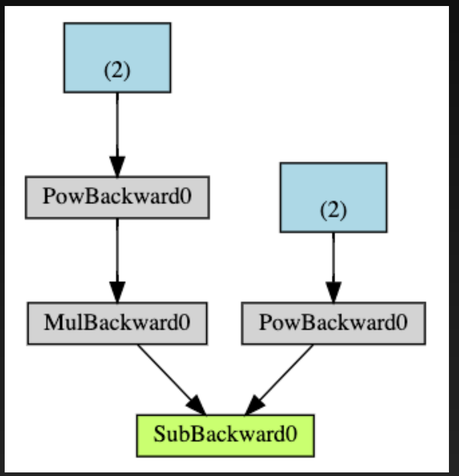
Credit: Zhouchen Lin

Advanced: Autograd in PyTorch

We create another tensor `Q` from `a` and `b`.

$$Q = 3a^3 - b^2$$

```
Q = 3*a**3 - b**2
```



- NOTE

DAGs are dynamic in PyTorch An important thing to note is that the graph is recreated from scratch; after each `.backward()` call, autograd starts populating a new graph. This is exactly what allows you to use control flow statements in your model; you can change the shape, size and operations at every iteration if needed.

Advanced: Autograd in PyTorch

`backward(gradient=None, retain_graph=None, create_graph=False, inputs=None)` [SOURCE]

Computes the gradient of current tensor w.r.t. graph leaves.

The graph is differentiated using the chain rule. If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally requires specifying `gradient`. It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. `self`.

This function accumulates gradients in the leaves - you might need to zero `.grad` attributes or set them to `None` before calling it. See Default gradient layouts for details on the memory layout of accumulated gradients.

• NOTE

If you run any forward ops, create `gradient`, and/or call `backward` in a user-specified CUDA stream context, see Stream semantics of backward passes.

Parameters

- **gradient** (*Tensor or None*) – Gradient w.r.t. the tensor. If it is a tensor, it will be automatically converted to a Tensor that does not require grad unless `create_graph` is True. None values can be specified for scalar Tensors or ones that don't require grad. If a None value would be acceptable then this argument is optional.
- **retain_graph** (*bool, optional*) – If `False`, the graph used to compute the grads will be freed. Note that in nearly all cases setting this option to True is not needed and often can be worked around in a much more efficient way. Defaults to the value of `create_graph`.
- **create_graph** (*bool, optional*) – If `True`, graph of the derivative will be constructed, allowing to compute higher order derivative products. Defaults to `False`.
- **inputs** (*sequence of Tensor*) – Inputs w.r.t. which the gradient will be accumulated into `.grad`. All other Tensors will be ignored. If not provided, the gradient is accumulated into all the leaf Tensors that were used to compute the attr::tensors. All the provided inputs must be leaf Tensors.

`detach()`

Returns a new Tensor, detached from the current graph.

The result will never require gradient.

- Generally speaking, you can only calculate the gradient of a "scalar" tensor

Advanced: Autograd in PyTorch

`torch.autograd.grad(outputs, inputs, grad_outputs=None, retain_graph=None, create_graph=False, only_inputs=True, allow_unused=False)` [SOURCE]

Computes and returns the sum of gradients of outputs w.r.t. the inputs.

`grad_outputs` should be a sequence of length matching `output` containing the "vector" in Jacobian-vector product, usually the pre-computed gradients w.r.t. each of the outputs. If an output doesn't require_grad, then the gradient can be `None`).

If `only_inputs` is `True`, the function will only return a list of gradients w.r.t the specified inputs. If it's `False`, then gradient w.r.t. all remaining leaves will still be computed, and will be accumulated into their `.grad` attribute.

• NOTE

If you run any forward ops, create `grad_outputs`, and/or call `grad` in a user-specified CUDA stream context, see Stream semantics of backward passes.

Parameters

- **outputs** (*sequence of Tensor*) – outputs of the differentiated function.
- **inputs** (*sequence of Tensor*) – Inputs w.r.t. which the gradient will be returned (and not accumulated into `.grad`).
- **grad_outputs** (*sequence of Tensor*) – The "vector" in the Jacobian-vector product. Usually gradients w.r.t. each output. None values can be specified for scalar Tensors or ones that don't require grad. If a None value would be acceptable for all grad_tensors, then this argument is optional. Default: None.
- **retain_graph** (*bool, optional*) – If `False`, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to `True` is not needed and often can be worked around in a much more efficient way. Defaults to the value of `create_graph`.
- **create_graph** (*bool, optional*) – If `True`, graph of the derivative will be constructed, allowing to compute higher order derivative products. Default: `False`.
- **allow_unused** (*bool, optional*) – If `False`, specifying inputs that were not used when computing outputs (and therefore their grad is always zero) is an error. Defaults to `False`.

```python
import torch
import torch.autograd as autograd

x = torch.tensor(1., requires_grad=True)
y = 2*x**3 + 5*x**2 + 8
y.backward(retain_graph=True)
print(x.grad)

first_derivative = autograd.grad(y, x, create_graph=True)[0]
print(first_derivative)
# We now have dy/dx
second_derivative = autograd.grad(first_derivative, x)[0]
print(second_derivative)
# This computes d/dx(dy/dx) = d2y/dx2
```

Tips

- Do not use loop unless there is no workaround
- Beware of frequently transferring data between CPUs and GPUs
- Profile your code (e.g. PyCharm) and optimize the bottleneck

References

- https://en.wikipedia.org/wiki/PyTorch
- https://en.wikipedia.org/wiki/NumPy
- https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html
- https://numpy.org/doc/stable/user/basics.indexing.html
- https://pytorch.org/docs/stable/
- https://pytorch.org/docs/stable/notes/autograd.html