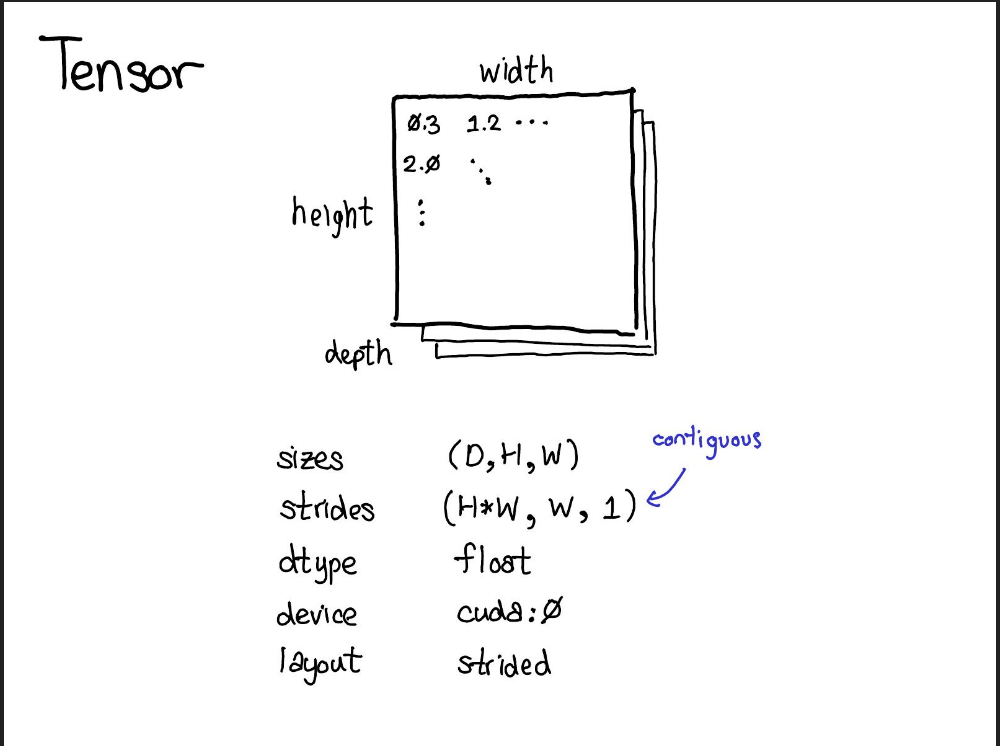
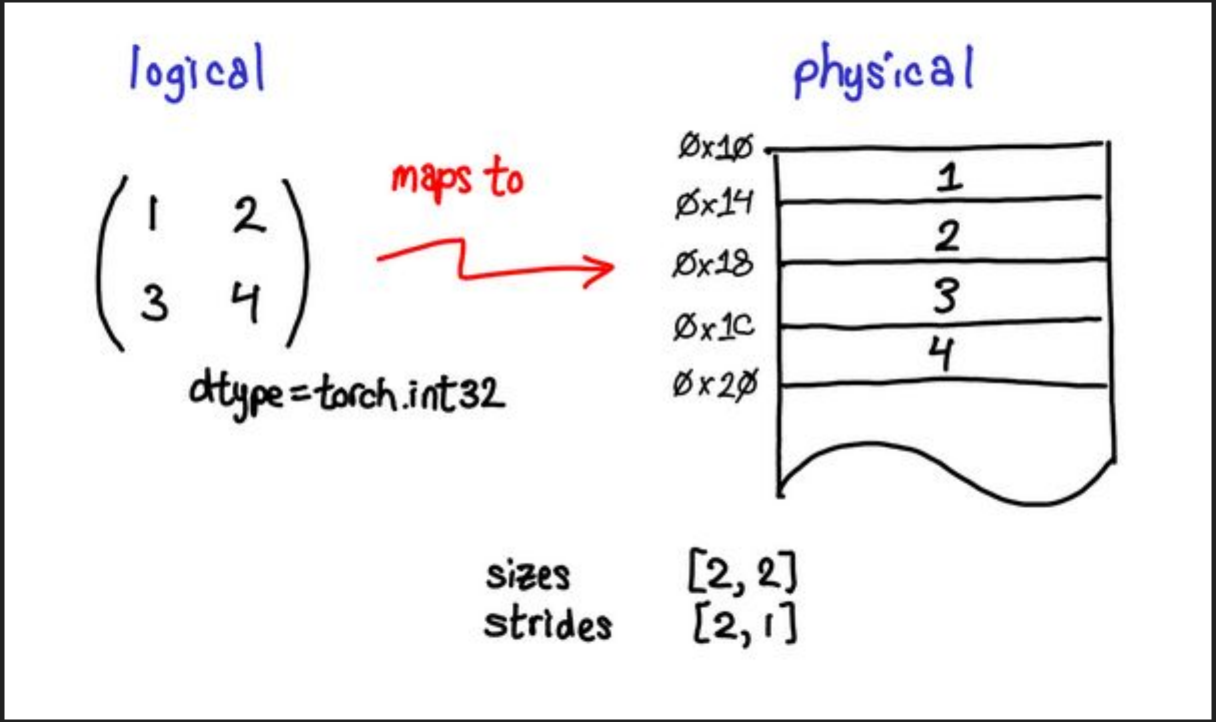
Dans les coulisse de PyTorch

midi de la bidouille

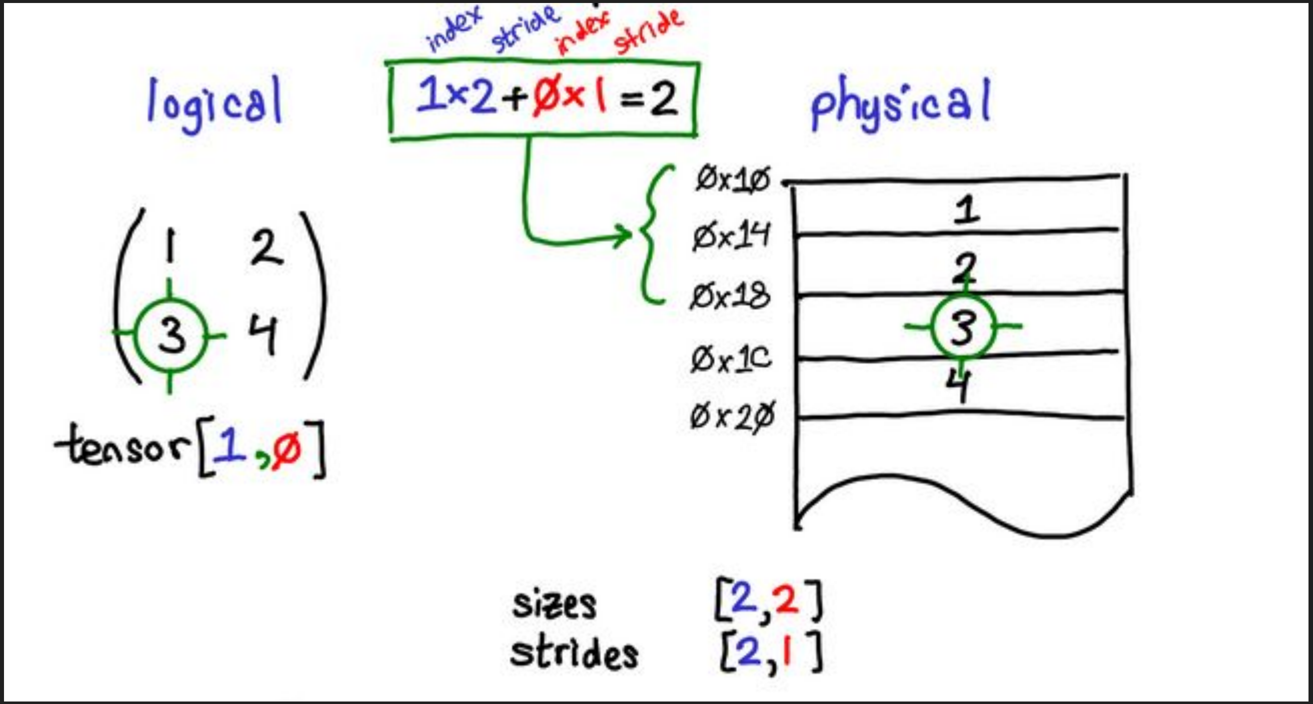
Tensor


Tensor
width
height
depth
Ø.3 1.2 ...
2.Ø
sizes (D, H, W)
strides (H*W, W, 1)
contiguous
dtype float
device cuda:Ø
layout strided

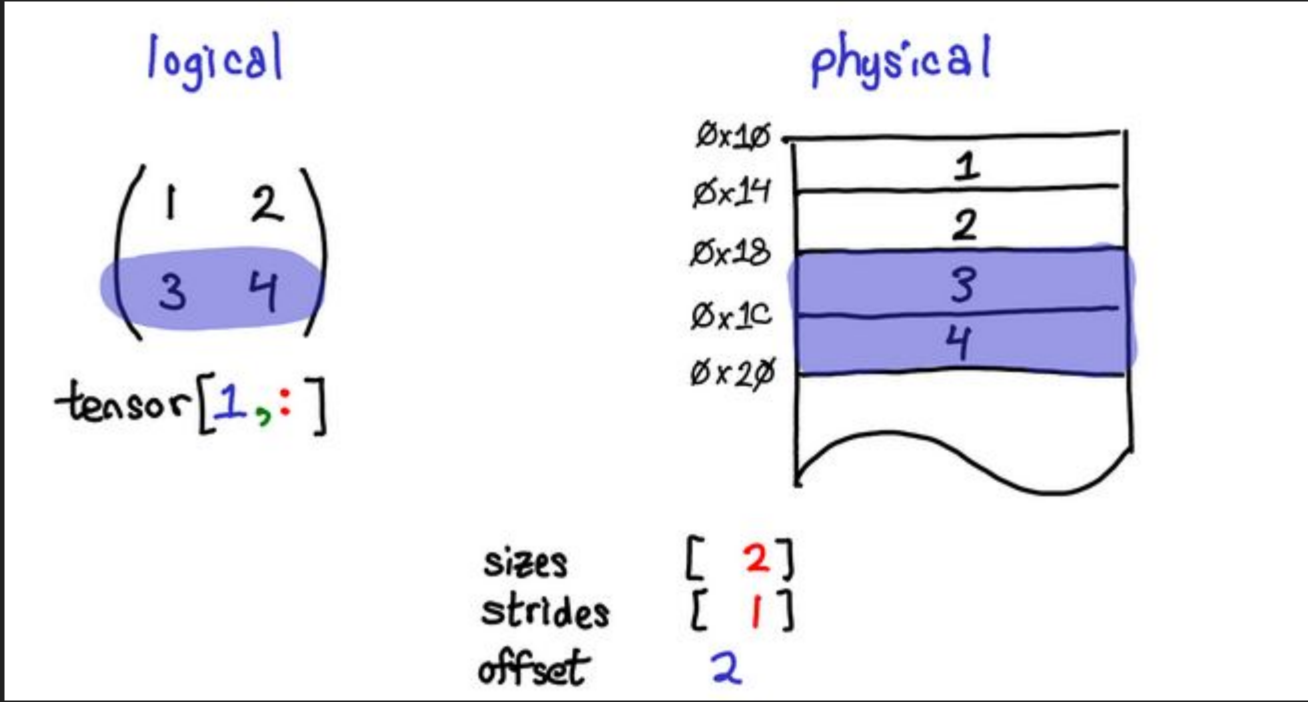
Tensor : Strided representation



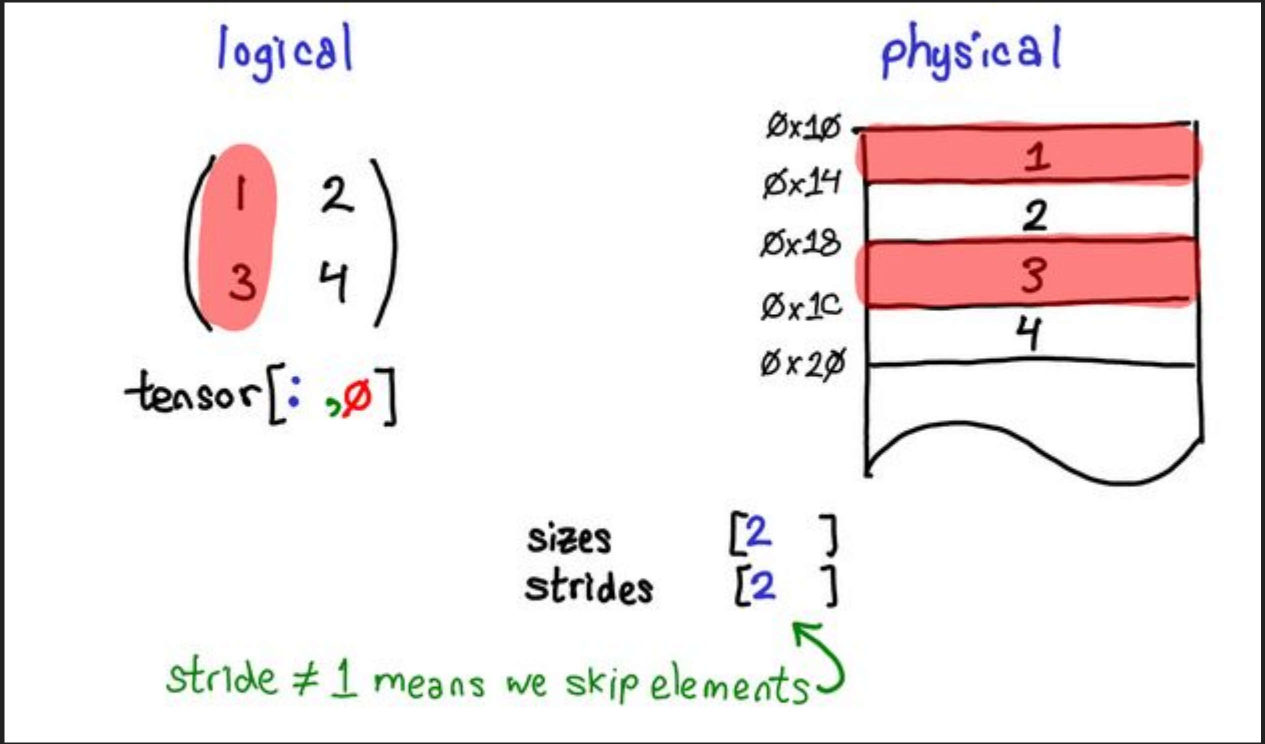
Strided representation



Stride representation



Stride representation



Tensor views

1. Supporting `View` avoids explicit data copy, thus allows us to do fast and memory efficient reshaping, slicing and element-wise operations.
2. https://pytorch.org/docs/stable/tensor_view.html

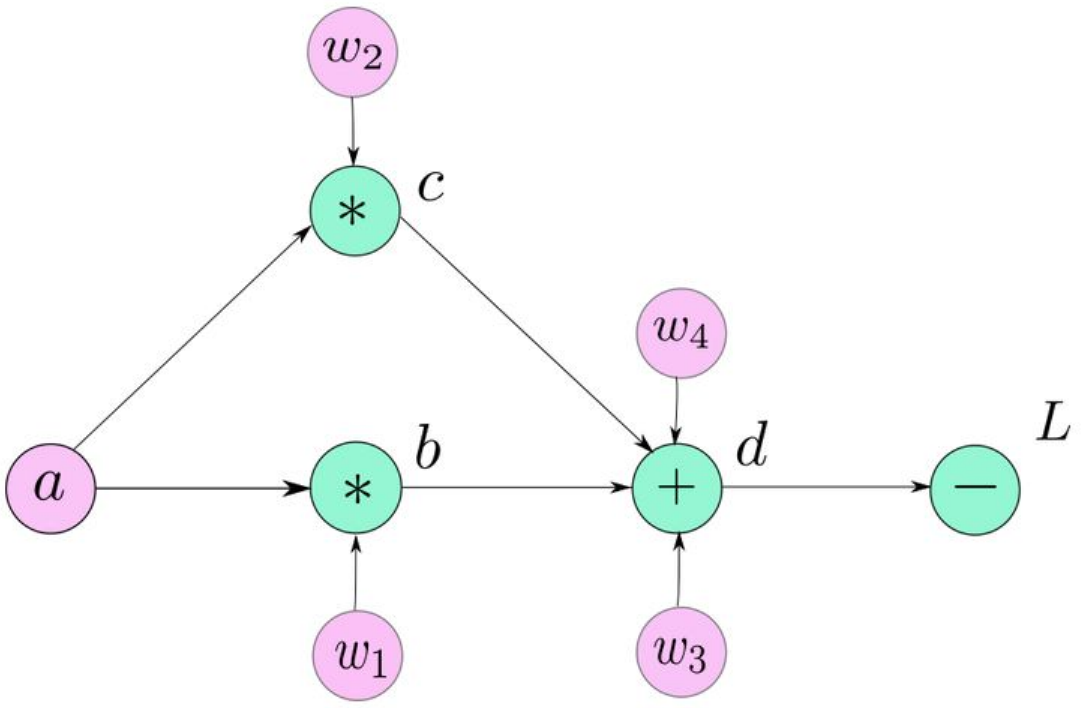
Autograd

Neural network training process

1. Define the architecture
2. Forward propagate on the architecture using input data
3. Calculate the loss
4. Backpropagate to calculate the gradient for each weight
5. Update the weights using a learning rate

```python
import torch
import torch_ort

# Model definition
class NeuralNet(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        ...

    def forward(self, x):
        ...

model = NeuralNet(input_size=784, hidden_size=500, num_classes=10
model = torch_ort.ORTModule(model)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

# Training Loop
for data, target in data_loader:
    # reset gradient buffer
    optimizer.zero_grad()
    # forward
    y_pred = model(data)
    loss = criterion(output, target)
    # backward
    loss.backward()
    # weight update
    optimizer.step()
```

Computation graph



Computation Graph for our very simple Neural Network

$$b = w_1 * a$$

$$c = w_2 * a$$

$$d = w_3 * b + w_4 * c$$

$$L = 10 - d$$

$$\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_4}$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_3}$$

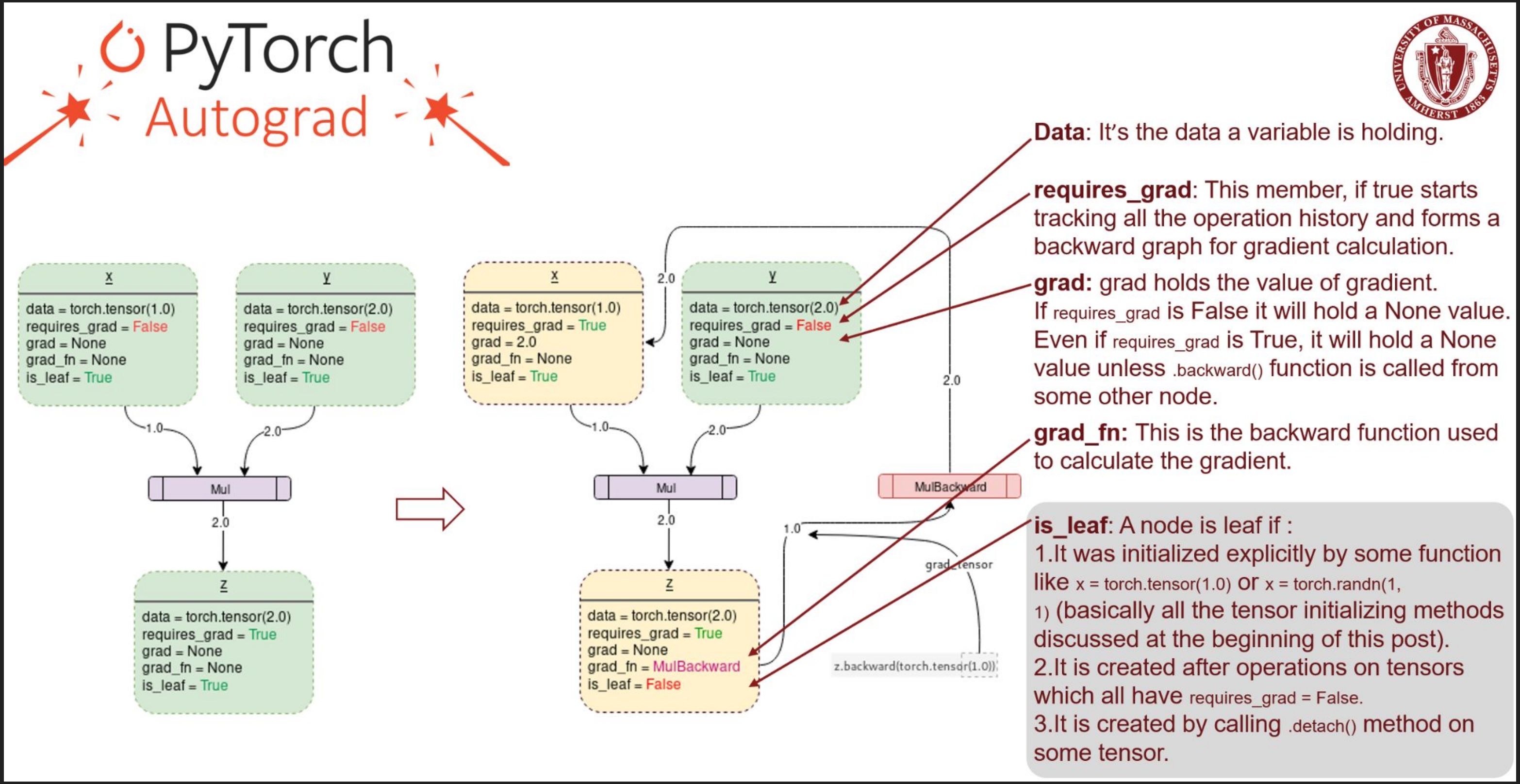
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial c} * \frac{\partial c}{\partial w_2}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial b} * \frac{\partial b}{\partial w_1}$$

PyTorch basics

- Tensor attributes
  - data
  - requires_grad = True
  - grad_fn
  - grad
  - is_leaf
- Autograd package : an engine to calculate derivatives (Jacobian-vector product to be more precise)

Dynamic computation graph



Gradients

the gradient of the scalar loss ***l*** with respect the vector **Y**

$$v = \left(\frac{\partial l}{\partial y_1} \quad \cdots \quad \frac{\partial l}{\partial y_m} \right)^T$$

the gradient of **f(X)** with respect to **X**

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jacobian matrix (Source: Wikipedia)

$$J \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

Gradients

But, when the output tensor is non-scalar we need to pass the external gradient vector as $\vec{v}$ and the resulting gradient is calculated Jacobian Vector Product i.e ***J@*$\vec{v}$*.T***

Here, for `F = a*b` at `a = [10.0, 10.0]` `b = [20.0, 20.0]` and `v = [1., 1.]` we get **∂F/∂a** as:

```
a = torch.tensor([10.,10.],requires_grad=True)
b = torch.tensor([20.,20.],requires_grad=True)

F = a * b

#claculate the gradients
F.backward(gradient=torch.tensor([1.,1.])) #modified

print(a.grad)
print(b.grad)
```

```
tensor([20., 20.])
tensor([10., 10.])
```

$$\begin{bmatrix} \frac{\partial f1}{\partial a1} & \frac{\partial f1}{\partial a2} \\ \frac{\partial f2}{\partial a1} & \frac{\partial f2}{\partial a2} \end{bmatrix} @ \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$=> \begin{bmatrix} b1 & 0 \\ 0 & b2 \end{bmatrix} @ \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$=> \begin{bmatrix} b1 & b2 \end{bmatrix}$$

$$=> \begin{bmatrix} 10 & 10 \end{bmatrix}$$

References

- http://blog.ezyang.com/2019/05/pytorch-internals/
- https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html?highlight=parameter
- https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/
- https://pytorch.org/blog/how-computational-graphs-are-executed-in-pytorch/
- https://www.miracleyoo.com/2019/12/11/Pytorch-Core-Code-Research/
- https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytorchs-magic-2686cd94ec95
- https://pytorch.org/docs/stable/notes/extending.html
- https://jovian.ml/attyuttam/01-tensor-operations
- https://abishekbashyall.medium.com/playing-with-backward-method-in-pytorch-bd34b58745a0
- https://www.youtube.com/watch?v=MswxJw-8PvE&pp=ygUQcHl0b3JjaCBhdXRvZ3JhZA%3D%3D