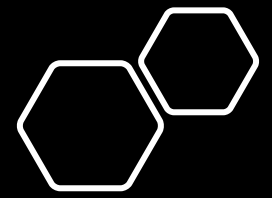
AIMS | African Institute for Mathematical Sciences RWANDA

AMMI Review sessions

Deep Learning Labs (3)

Autograd in Pytorch



13 Feb 2020

Autograd

- Autograd is now a core torch package for automatic differentiation

- In the forward phase, the autograd tape will remember all the operations it executed, and in the backward phase, it will replay the operations.

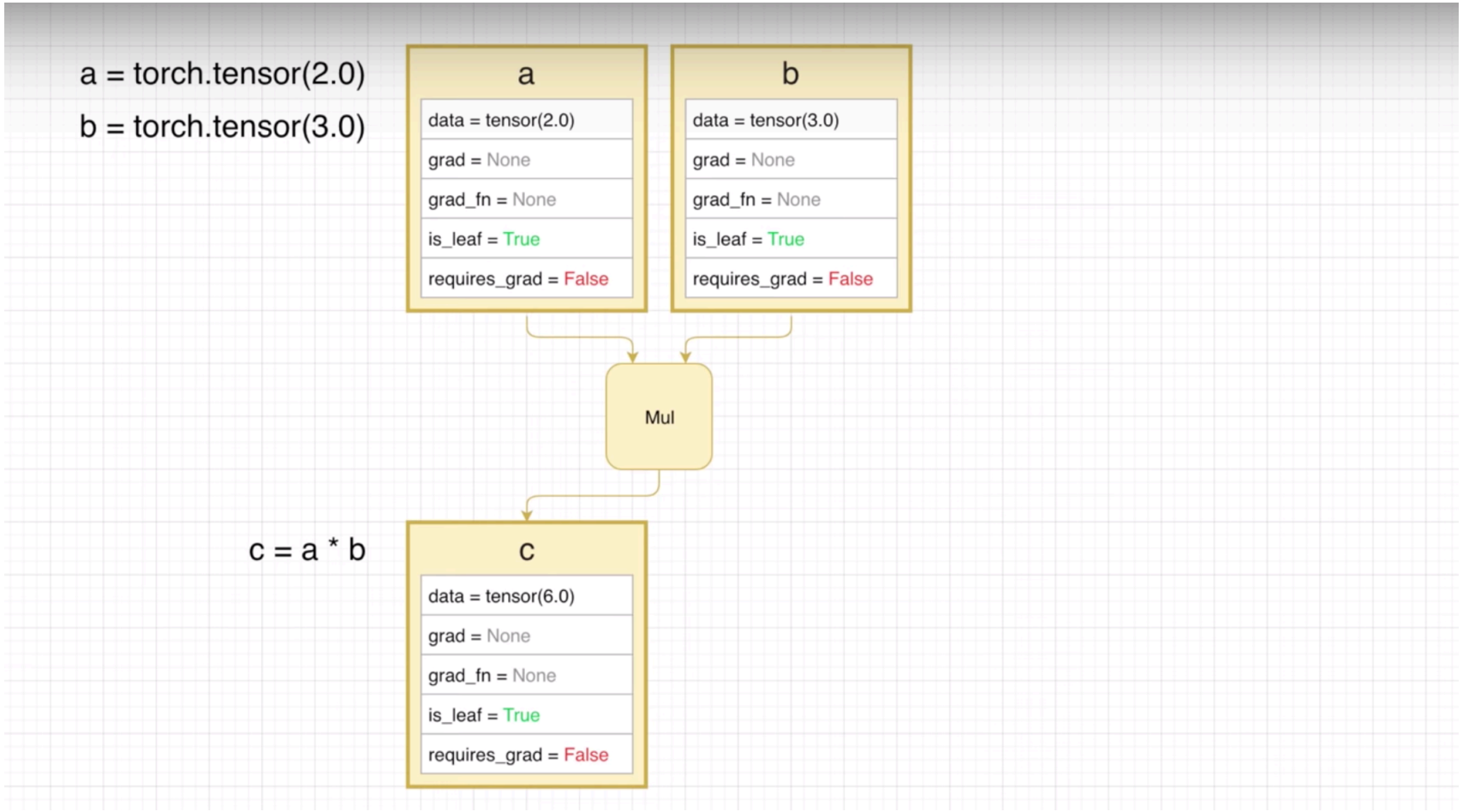
Tensors that track history

- if any input Tensor of an operation has requires_grad=True, the computation will be tracked. After computing the backward pass, a gradient w.r.t. this tensor is accumulated into .grad attribute.

- Each variable has a .grad_fn attribute that references a function that has created a function (except for Tensors created by the user - these have None as .grad_fn).

- If you want to compute the derivatives, you can call .backward() on a Tensor.

Tensors that track history



How Pytorch Backward() function works ?

The Gradient of scalar Loss function

- When we use gradient descent as learning algorithm of our model we need to compute the gradient of the loss w.r.t the model parameters.

- The loss term is usually a scalar value obtained by defining loss function (criterion) between the model prediction and and the true label — in a supervised learning problem setting — and usually we call loss.item() to get single python number out of the loss tensor.

The Gradient of scalar Loss function

- When we start propagating the gradients backward, we start by computing the derivative of this scalar loss (*L*) w.r.t to the direct previous hidden layer (*h*) which's a vector (group of weights)

- What would be the gradient in this case ?

- Simply it's a Jacobian vector (nx1) that contains the derivative of the loss with respect to all the hidden layer variables (h_1, h_2, h_3,… , h_n)

$$\text{for a function } l : \mathbf{R}^n \to \mathbf{R}$$

$$\mathbf{J_l} = \begin{bmatrix} \frac{\partial l}{\partial h_1} & \frac{\partial l}{\partial h_2} & \frac{\partial l}{\partial h_3} & \cdots & \frac{\partial l}{\partial h_n} \end{bmatrix}$$

The Gradient of vector loss function

- let say now we want to compute the gradient of a some loss vector (l) w.r.t to a hidden layer vector then we need to compute the full Jacobian

$$\text{for a function } l : \mathbf{R}^n \to \mathbf{R}^m$$

$$\mathbf{J_l} = \begin{bmatrix} \frac{\partial l_1}{\partial h_1} & \frac{\partial l_2}{\partial h_1} & \cdots & \frac{\partial l_m}{\partial h_1} \\ \frac{\partial l_1}{\partial h_2} & \frac{\partial l_2}{\partial h_2} & \cdots & \frac{\partial l_m}{\partial h_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial l_1}{\partial h_n} & \frac{\partial l_2}{\partial h_n} & \cdots & \frac{\partial l_m}{\partial h_n} \end{bmatrix}$$

- By looking into our gradient descent step

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} l(\theta)$$

- it’s obvious that we’re actually interested in the gradient vector rather than the full Jacobian

The Gradient of vector loss function

- The gradient thus can be obtained by summing the gradients of all variables in the loss vector w.r.t each variable in the hidden layer and would have the following formula:

$$\frac{l(\theta)}{dh} = \left[\frac{\partial l}{\partial h_1} \quad \frac{\partial l}{\partial h_2} \quad \cdots \quad \frac{\partial l}{\partial h_n}\right]^T$$

where

$$\frac{\partial l}{\partial h_i} = \frac{\partial l_1}{\partial h_i} + \frac{\partial l_2}{\partial h_i} + \cdots + \frac{\partial l_m}{\partial h_i}$$

The Jacobian-vector product

- We can easily show that we can obtain the gradient by multiplying the full Jacobian Matrix by a vector of ones as follows:

$$\frac{dl(\theta)}{dh} = \begin{bmatrix} \frac{\partial l_1}{\partial h_1} & \frac{\partial l_2}{\partial h_1} & \cdots & \frac{\partial l_m}{\partial h_1} \\ \frac{\partial l_1}{\partial h_2} & \frac{\partial l_2}{\partial h_2} & \cdots & \frac{\partial l_m}{\partial h_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial l_1}{\partial h_n} & \frac{\partial l_2}{\partial h_n} & \cdots & \frac{\partial l_m}{\partial h_n} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

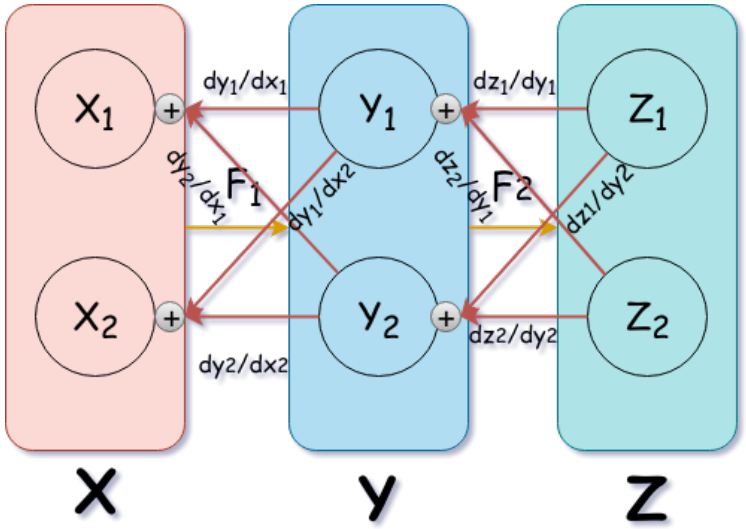
$$\ldots\ldots\ldots\ldots\ldots\ldots = \begin{bmatrix} \frac{\partial l_1}{\partial h_1} + \frac{\partial l_2}{\partial h_1} + \cdots + \frac{\partial l_m}{\partial h_1} \\ \frac{\partial l_1}{\partial h_2} + \frac{\partial l_2}{\partial h_2} + \cdots + \frac{\partial l_m}{\partial h_2} \\ \vdots \\ \frac{\partial l_1}{\partial h_n} + \frac{\partial l_2}{\partial h_n} + \cdots + \frac{\partial l_m}{\partial h_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial l}{\partial h_1} \\ \frac{\partial l}{\partial h_2} \\ \vdots \\ \frac{\partial l}{\partial h_n} \end{bmatrix}$$

- This ones vector is the argument that we pass to the Backward() function to compute the gradient, and this expression is called the Jacobian-vector product!

Jacobian-vector product in backpropagation

Let's take the following example:

- assume we have the following transformation functions f1 and f2 and x, y, z three vectors each of which is of 2 dimensions



- To use the chain rule to calculate $\frac{dz}{dx}$ we calculate $\frac{\mathrm{dz}}{dx} = \frac{dz}{dy} * \frac{dy}{dx}$

- Since z is a leaf node, we can directly compute the gradient $\frac{dz}{dy} = J_z \ @ \ \vec{1}$ using the Jacobian-vector product where J_z is the full Jacobian and $\vec{1}$ is vector of ones.

Jacobian-vector product in backpropagation

- Now if we want to compute $\frac{dz}{dx}$ we need to compute the full Jacobian J_y w.r.t x but **instead of multiplying it by ones vector we multiply it by the gradient vector** $\frac{dz}{dy}$ as follows

$$\frac{dz}{dx} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \\ \vdots \\ \frac{\partial z}{\partial y_m} \end{bmatrix}$$

$$\ldots\ldots\ldots = \begin{bmatrix} \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_1} + \cdots + \frac{\partial z}{\partial y_m}\frac{\partial y_m}{\partial x_1} \\ \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_2} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_2} + \cdots + \frac{\partial z}{\partial y_m}\frac{\partial y_m}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_n} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_n} + \cdots + \frac{\partial z}{\partial y_m}\frac{\partial y_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_n} \end{bmatrix}$$

Jacobian-vector product in backpropagation

- As we propagate gradients backward keeping the full Jacobian Matrix is not memory friendly process specially if we are training a giant model where one full Jacobian Matrix could be of size bigger than100K parameters

- Instead **we only need to keep the most recent gradient which requires much less memory footprint.**

Why Pytorch uses Jacobian-vector product ?

- This way we can we can just keep multiplying the gradients vectors of descendent nodes in our computational graph with full — Jacobians of their ancestors backward to the leaf nodes.

- We can interpret the ones vector that we provide to the backward() function as **an external gradients from which the chain rule starts the multiplication process.**

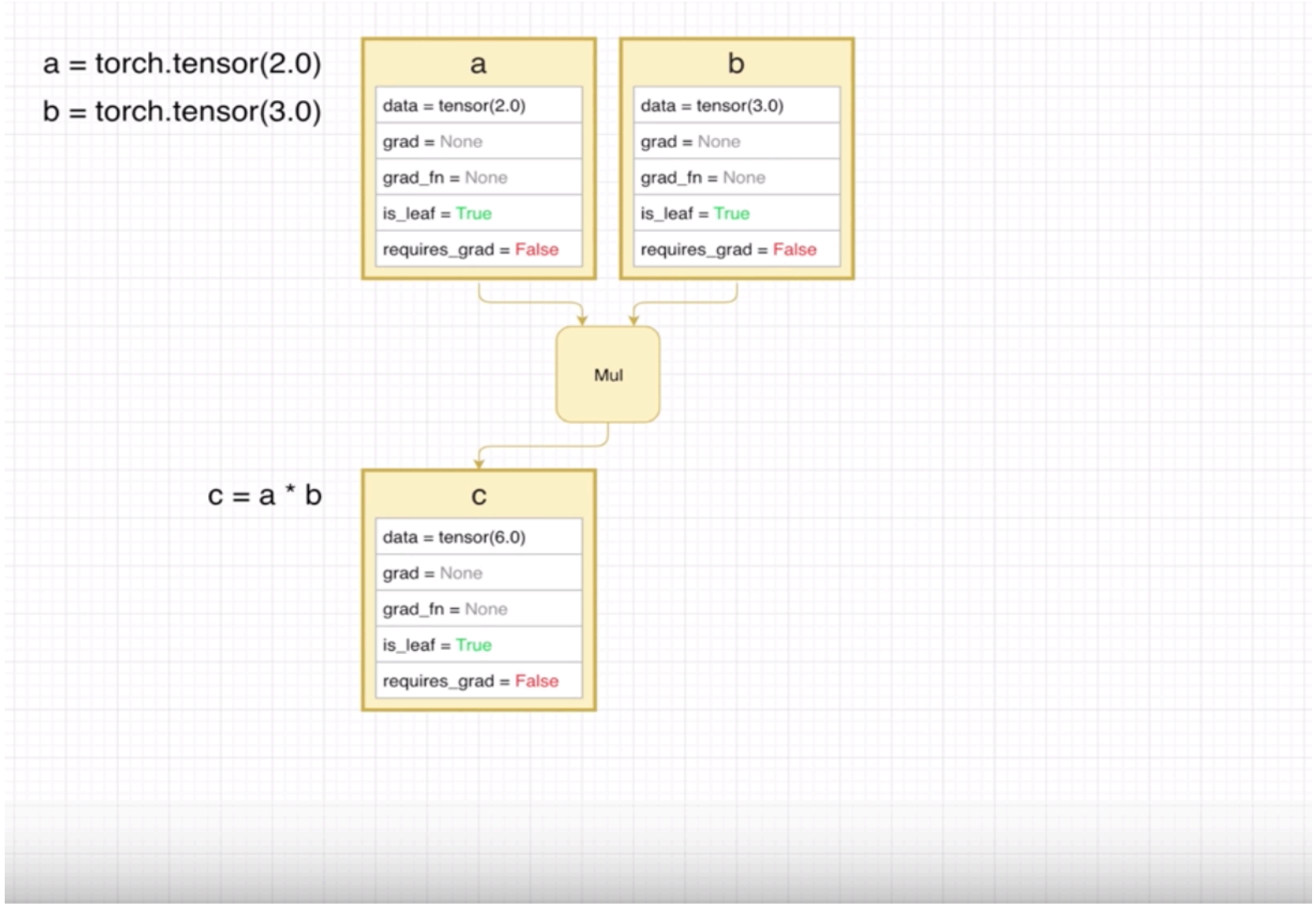
What if we submitted numbers other than ones in the external gradient vector (Backwad() argument) ?

- The values in the external gradient vector can serve like weights or **importances to each loss component**

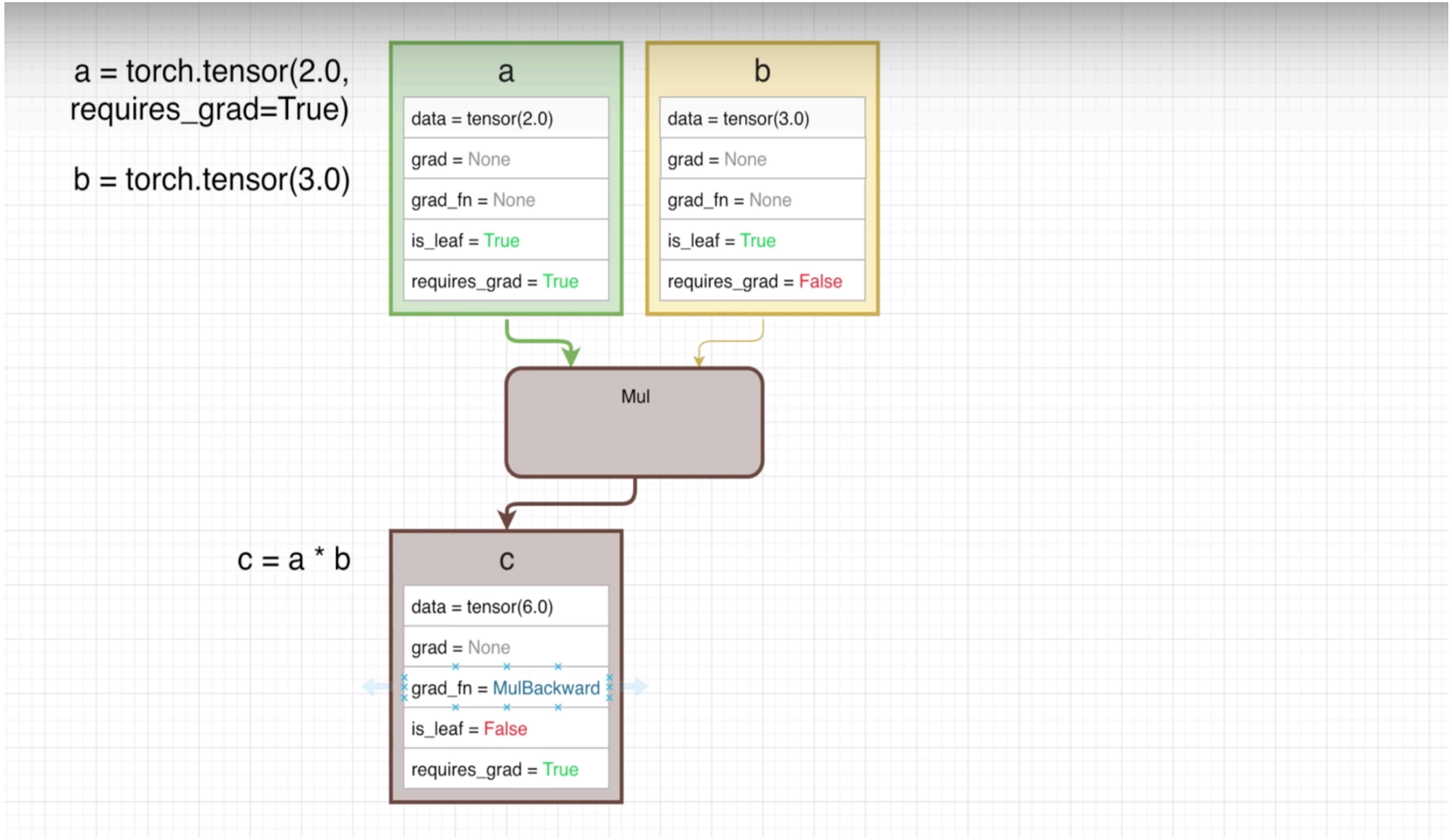
- For example if we submit the vector [0.2 0.8] as parameter in the previous example, we get this

$$\frac{dz}{dy} = \begin{bmatrix} 0.2\frac{\partial z}{\partial y1} & 0.8\frac{\partial z}{\partial y2} \end{bmatrix}^T$$

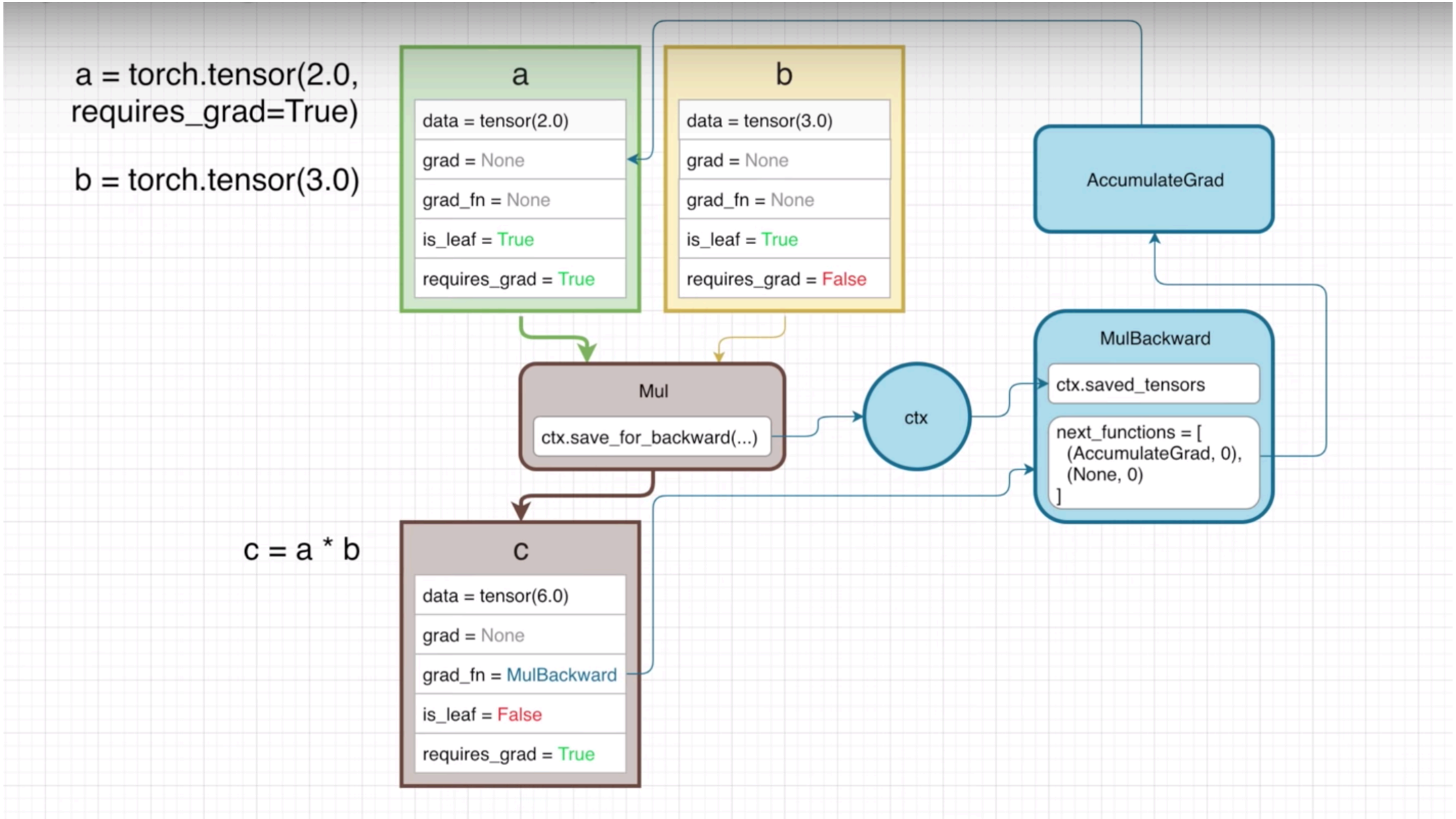
How Computational graphs are constructed in Pytorch e.g (1)



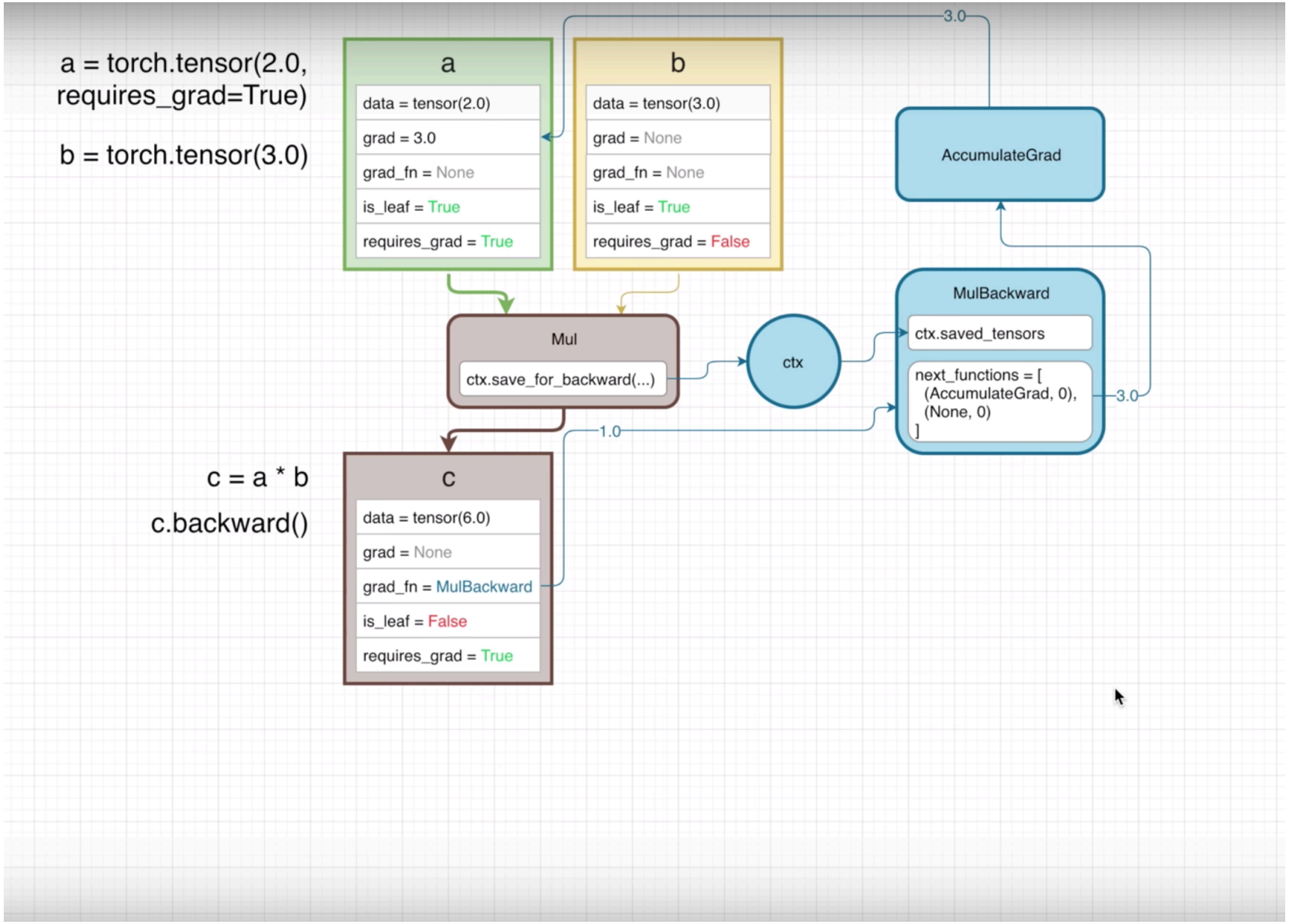
How Computational graphs are constructed in Pytorch e.g (1)



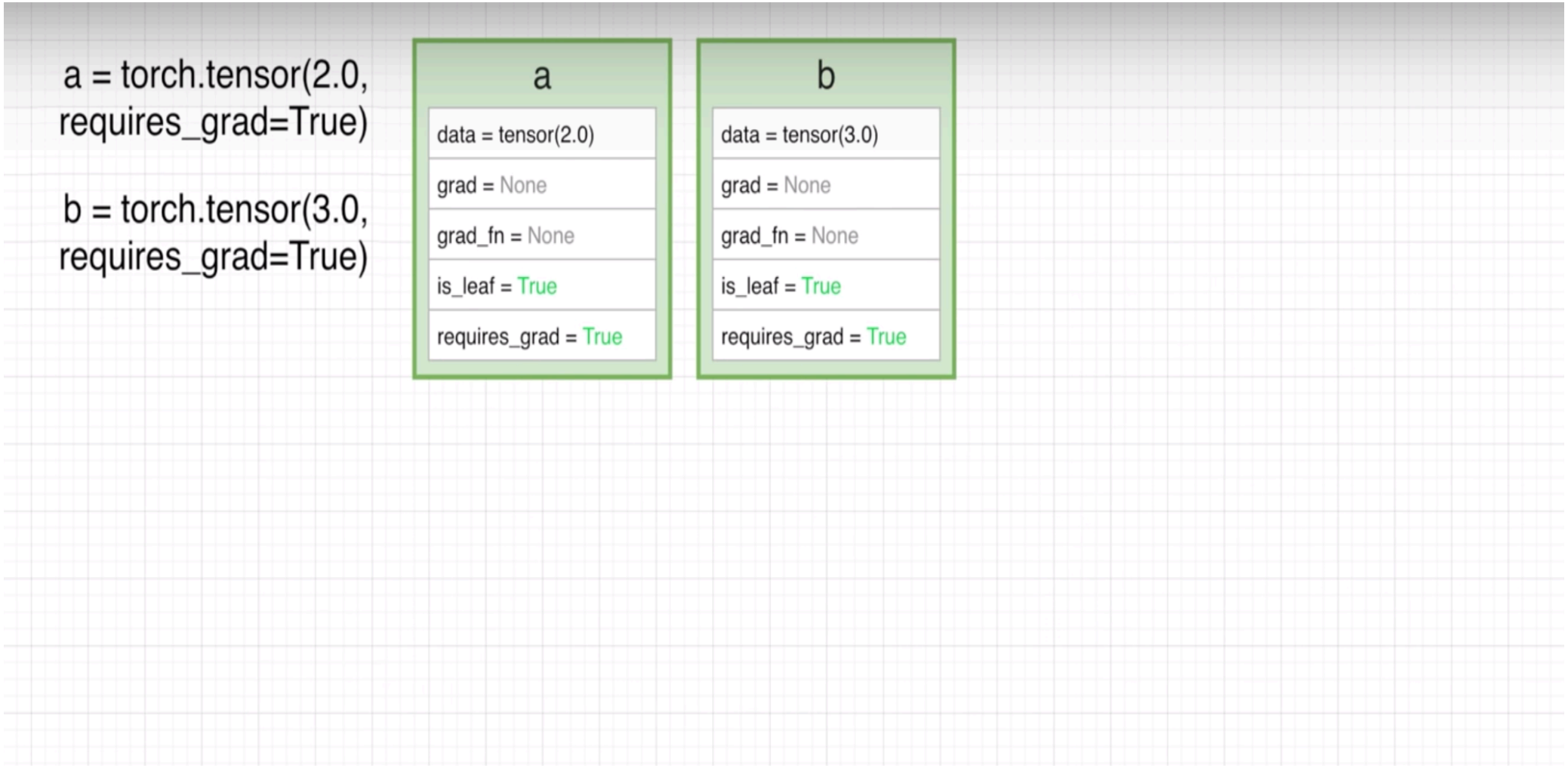
How Computational graphs are constructed in Pytorch e.g (1)



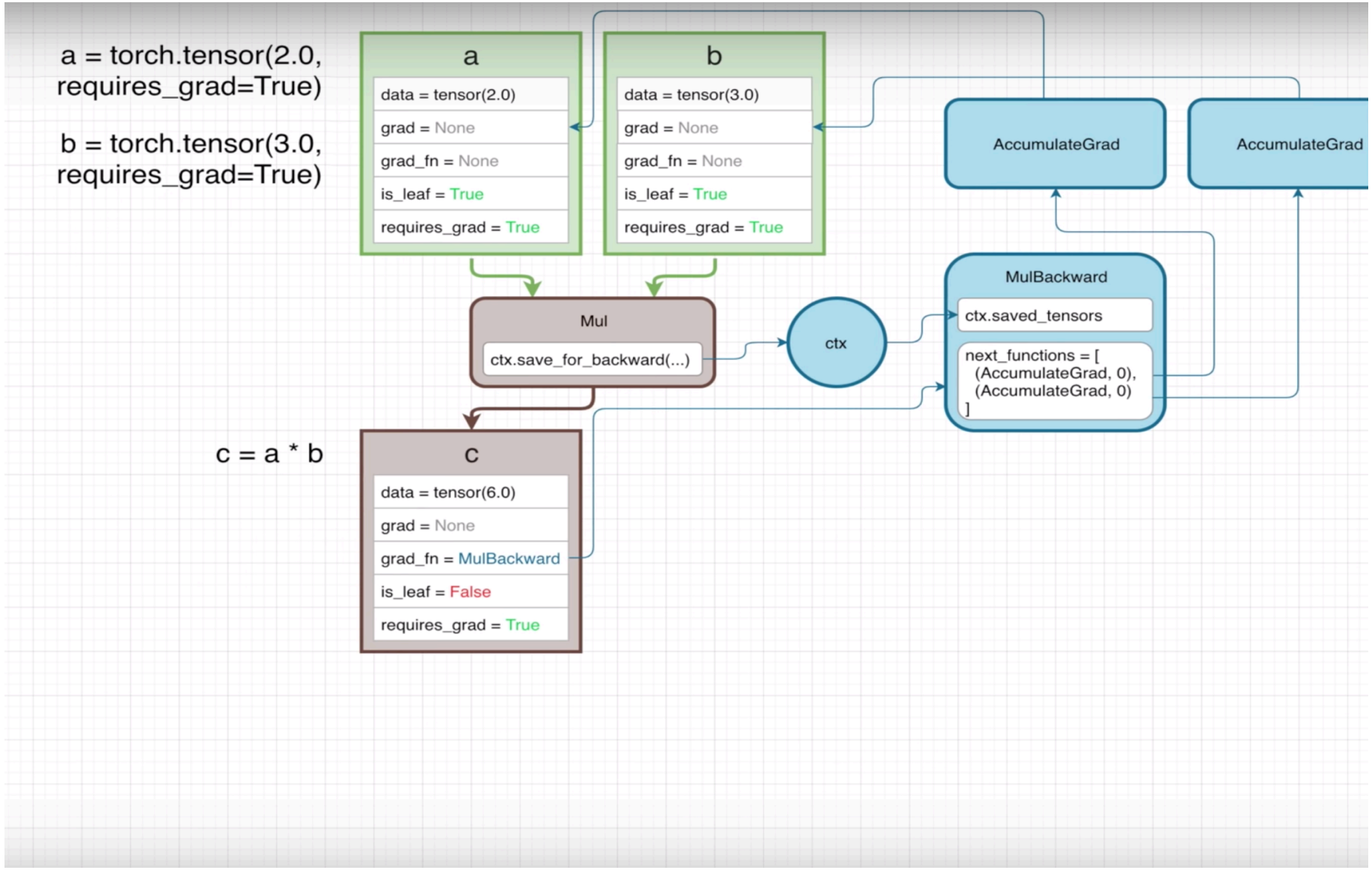
How Computational graphs are constructed in Pytorch e.g (1)



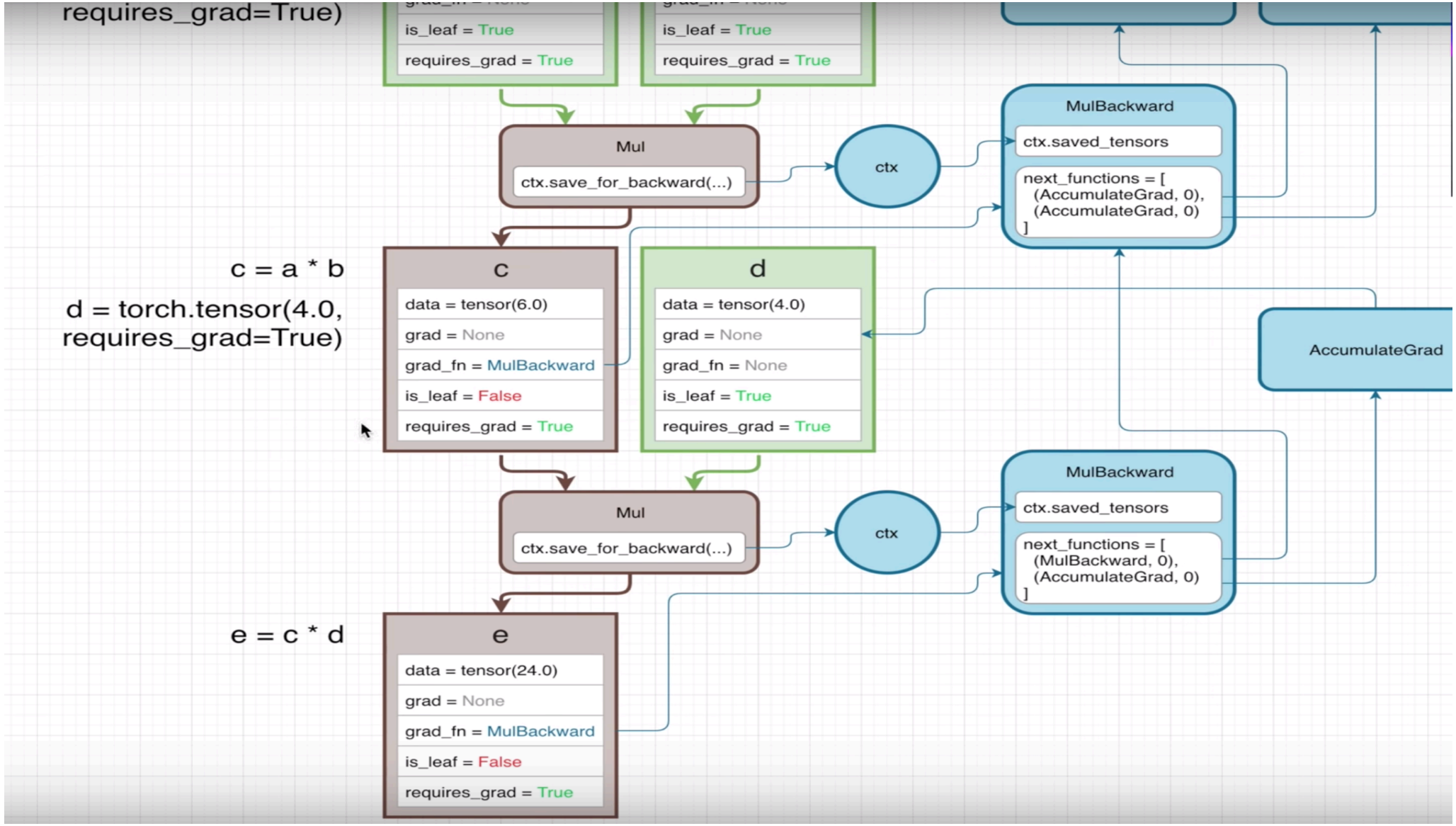
How Computational graphs are constructed in Pytorch e.g (2)



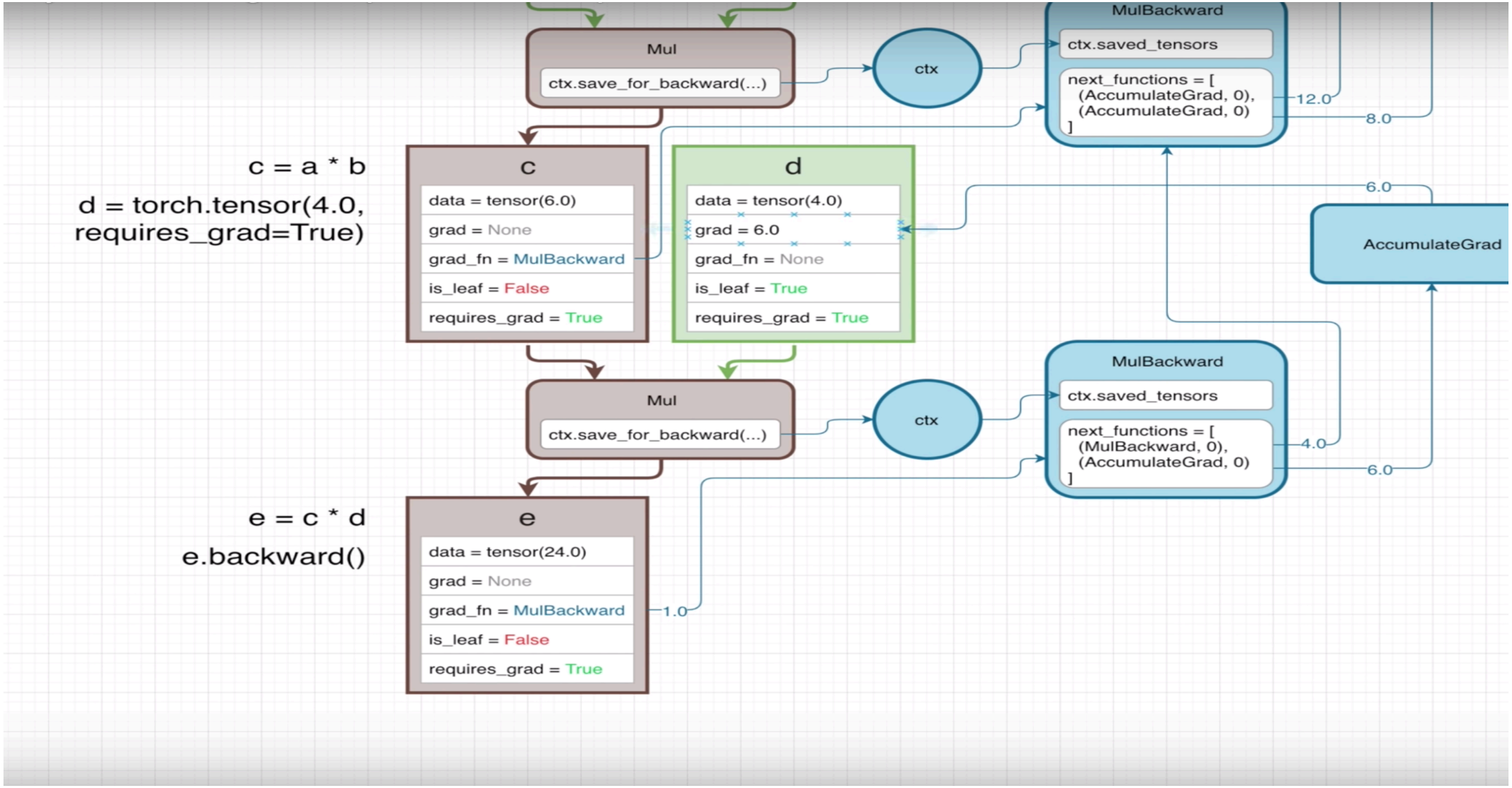
How Computational graphs are constructed in Pytorch e.g (2)



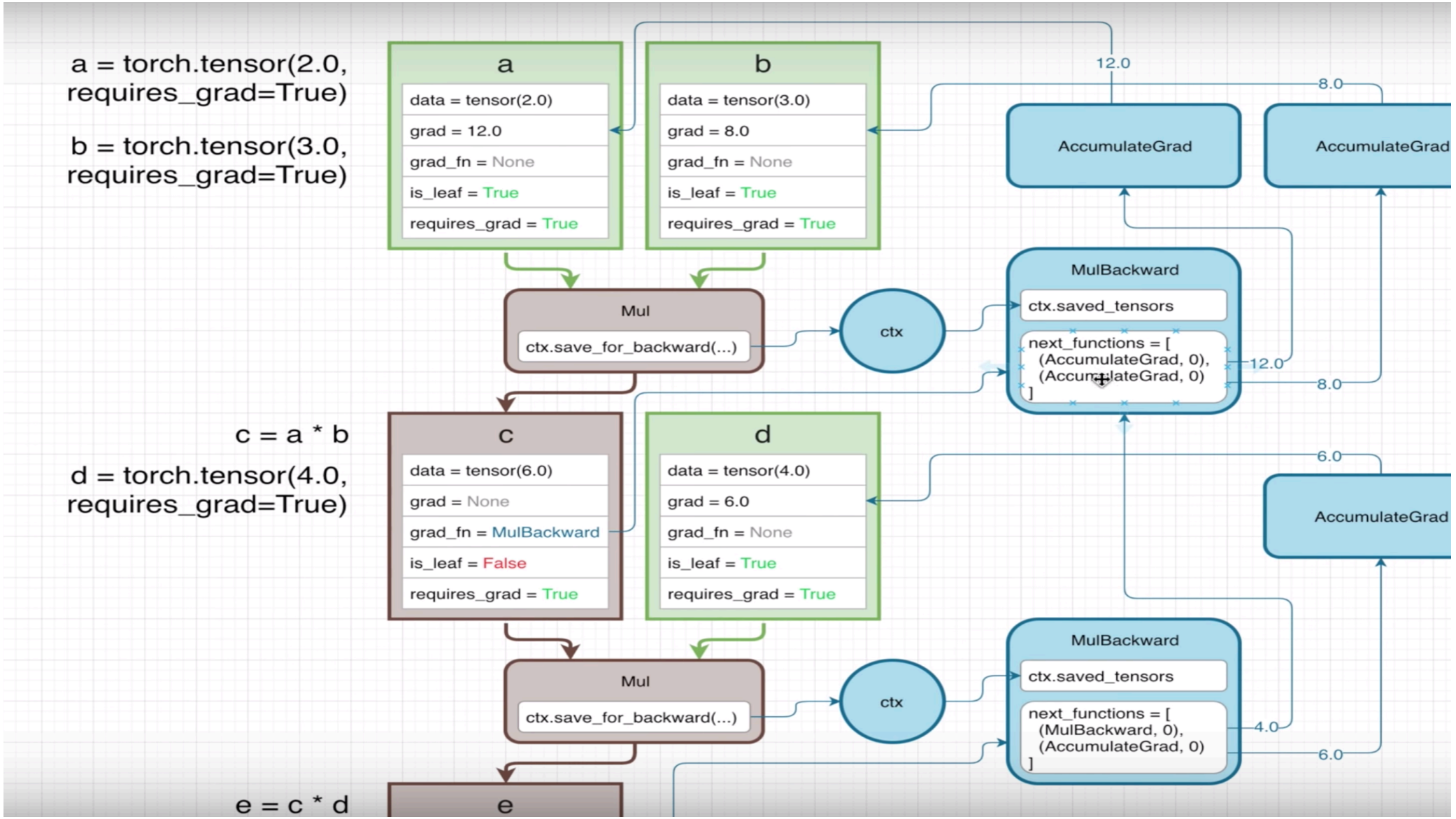
How Computational graphs are constructed in Pytorch e.g (2)



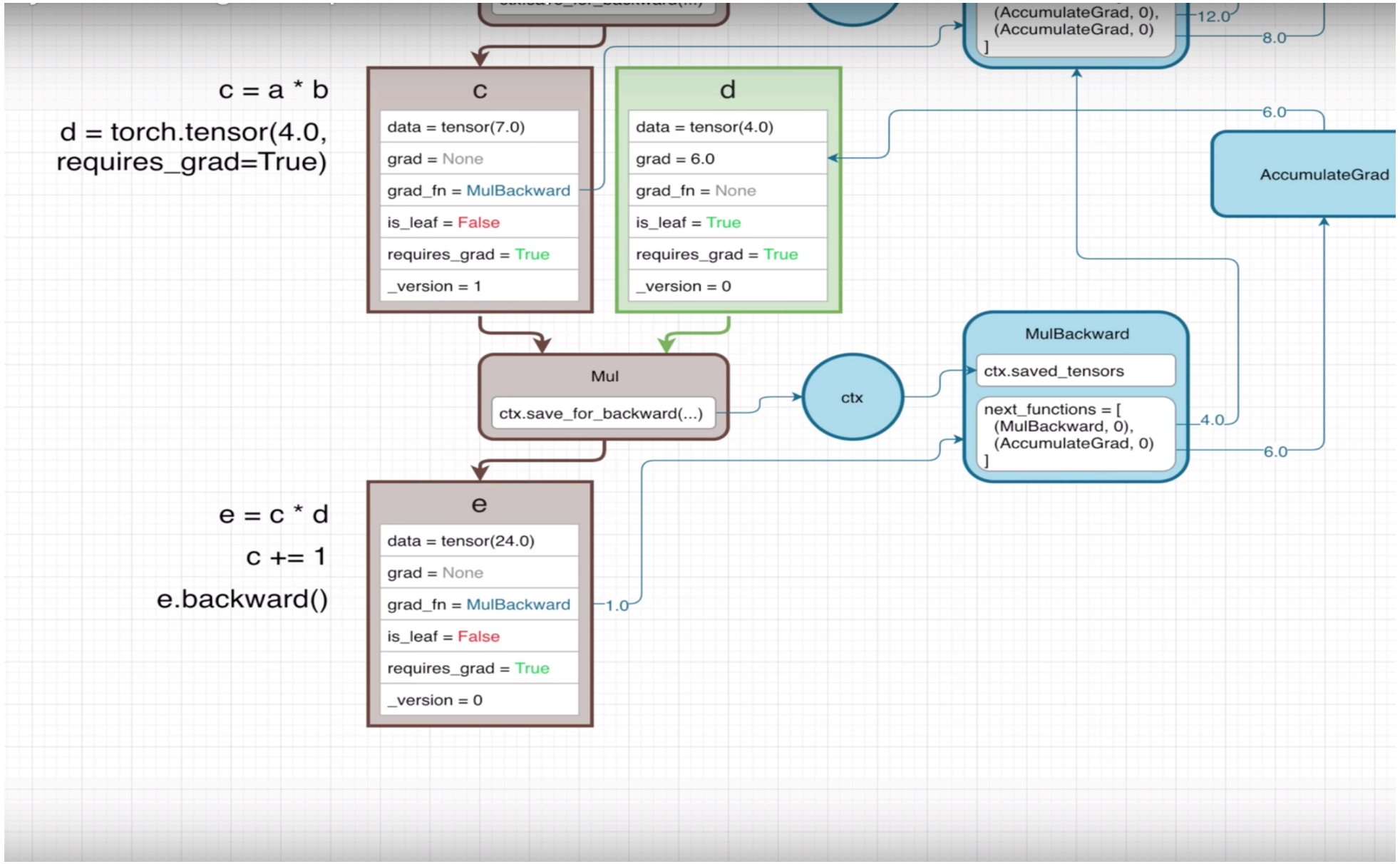
How Computational graphs are constructed in Pytorch e.g (2)



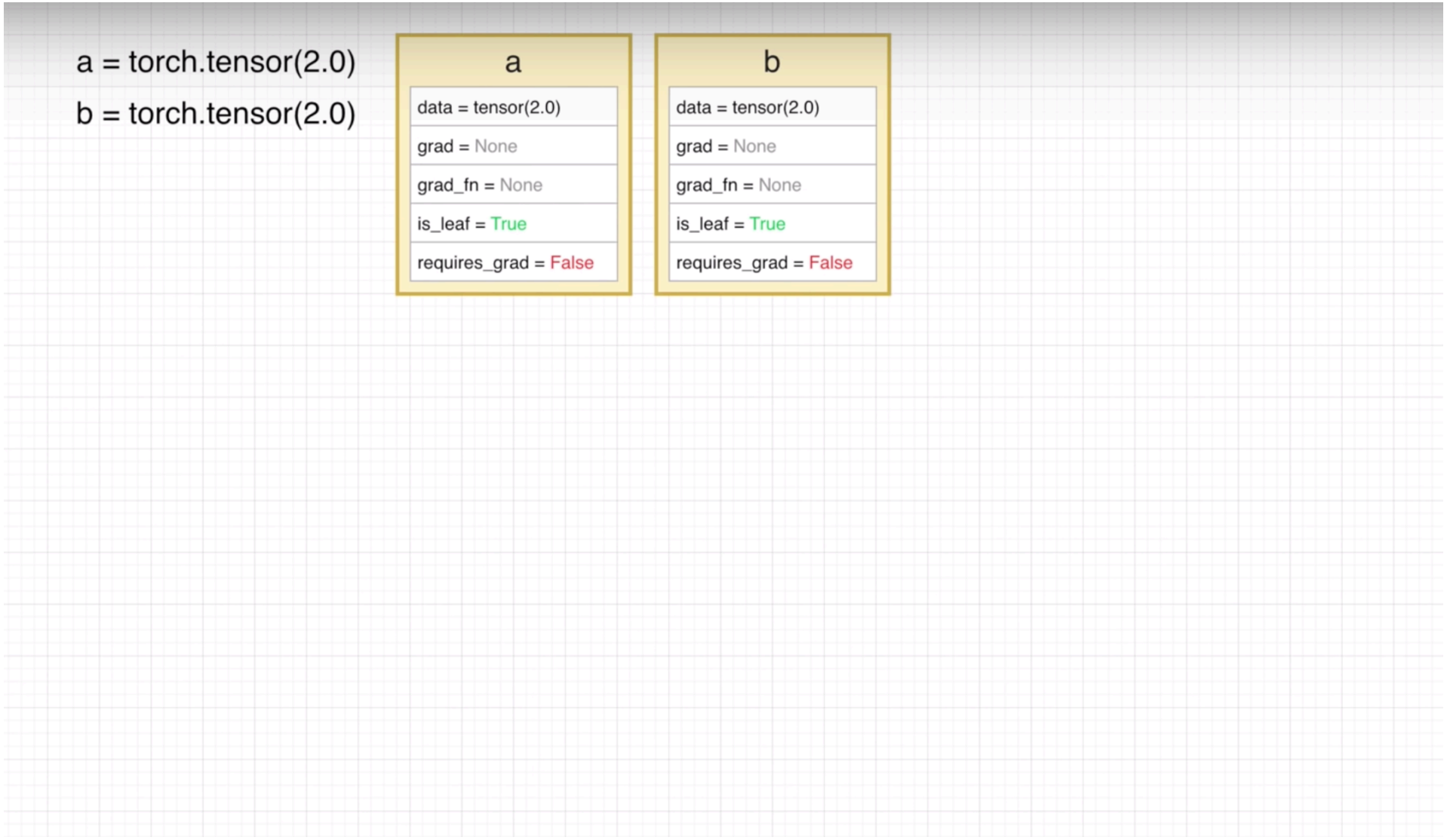
How Computational graphs are constructed in Pytorch e.g (2)



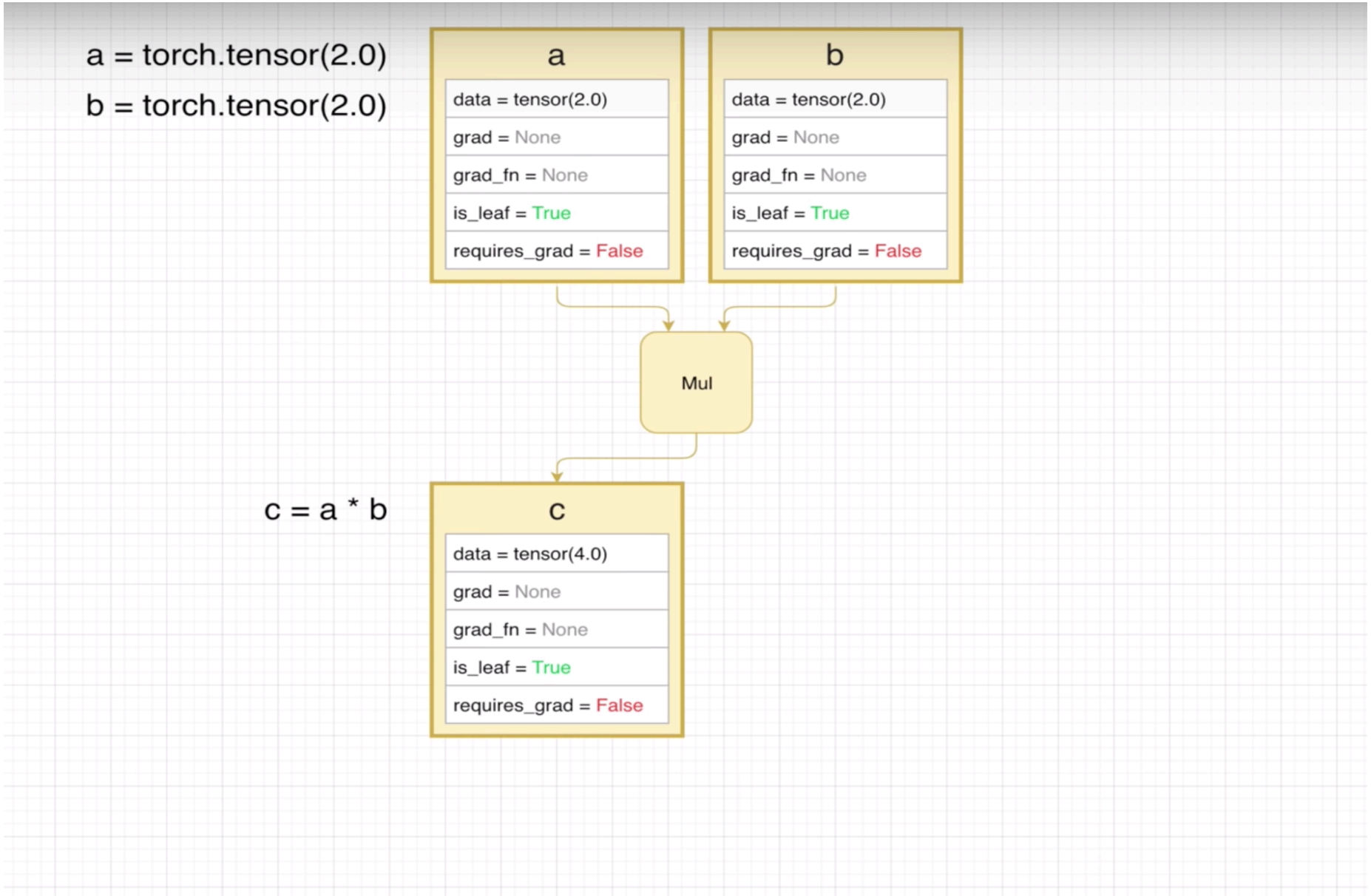
How Computational graphs are constructed in Pytorch e.g (2)



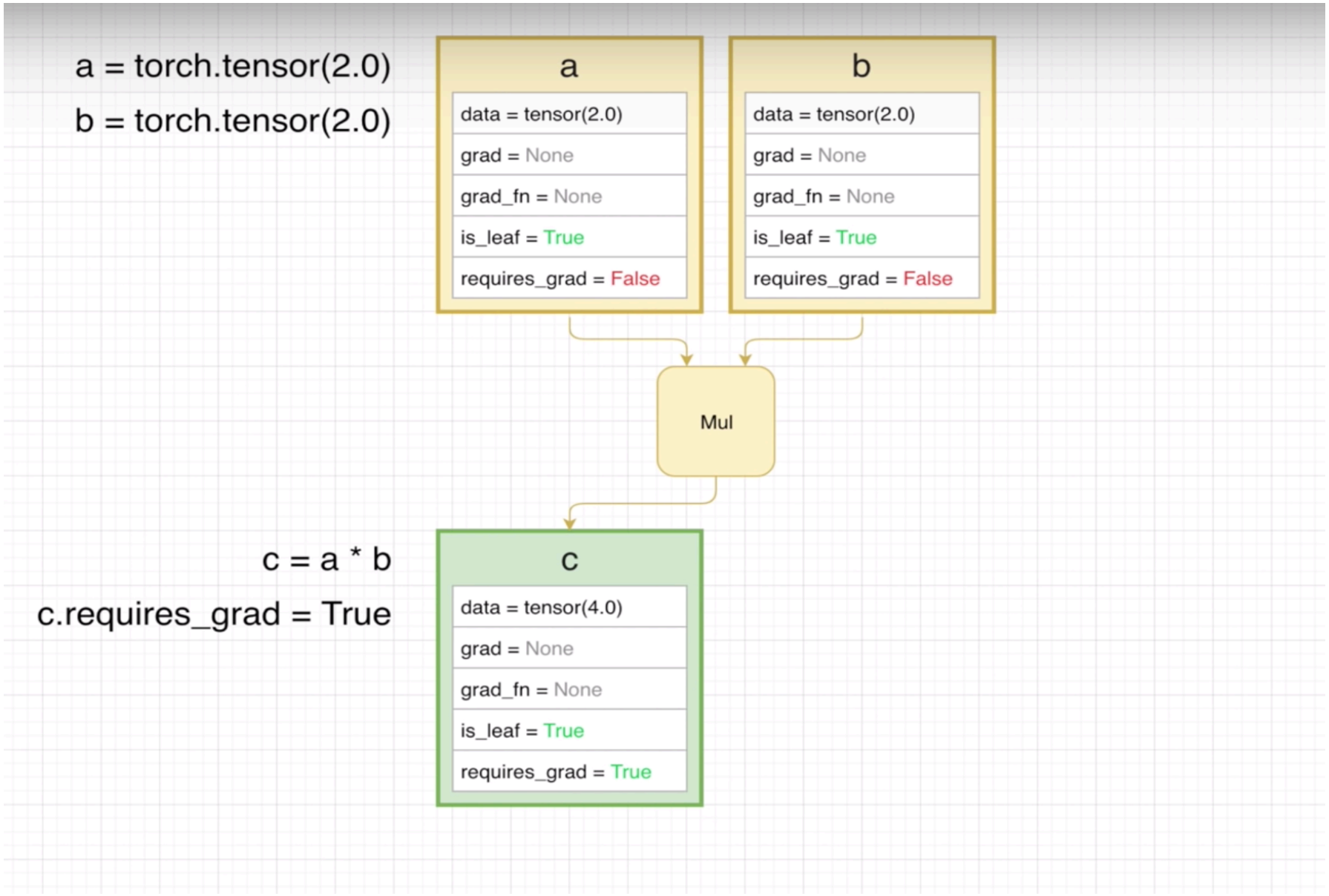
How Computational graphs are constructed in Pytorch e.g (3)



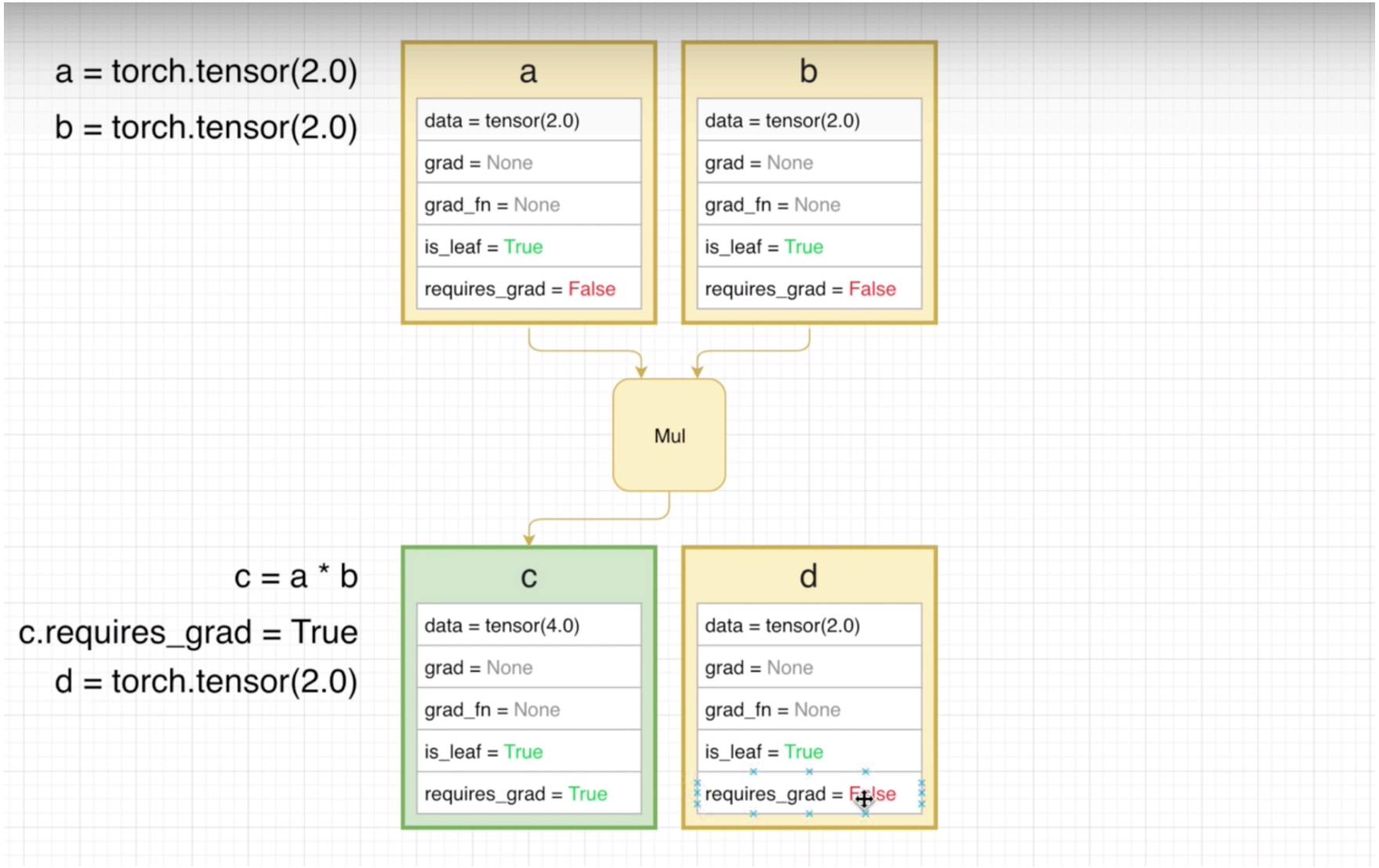
How Computational graphs are constructed in Pytorch e.g (3)



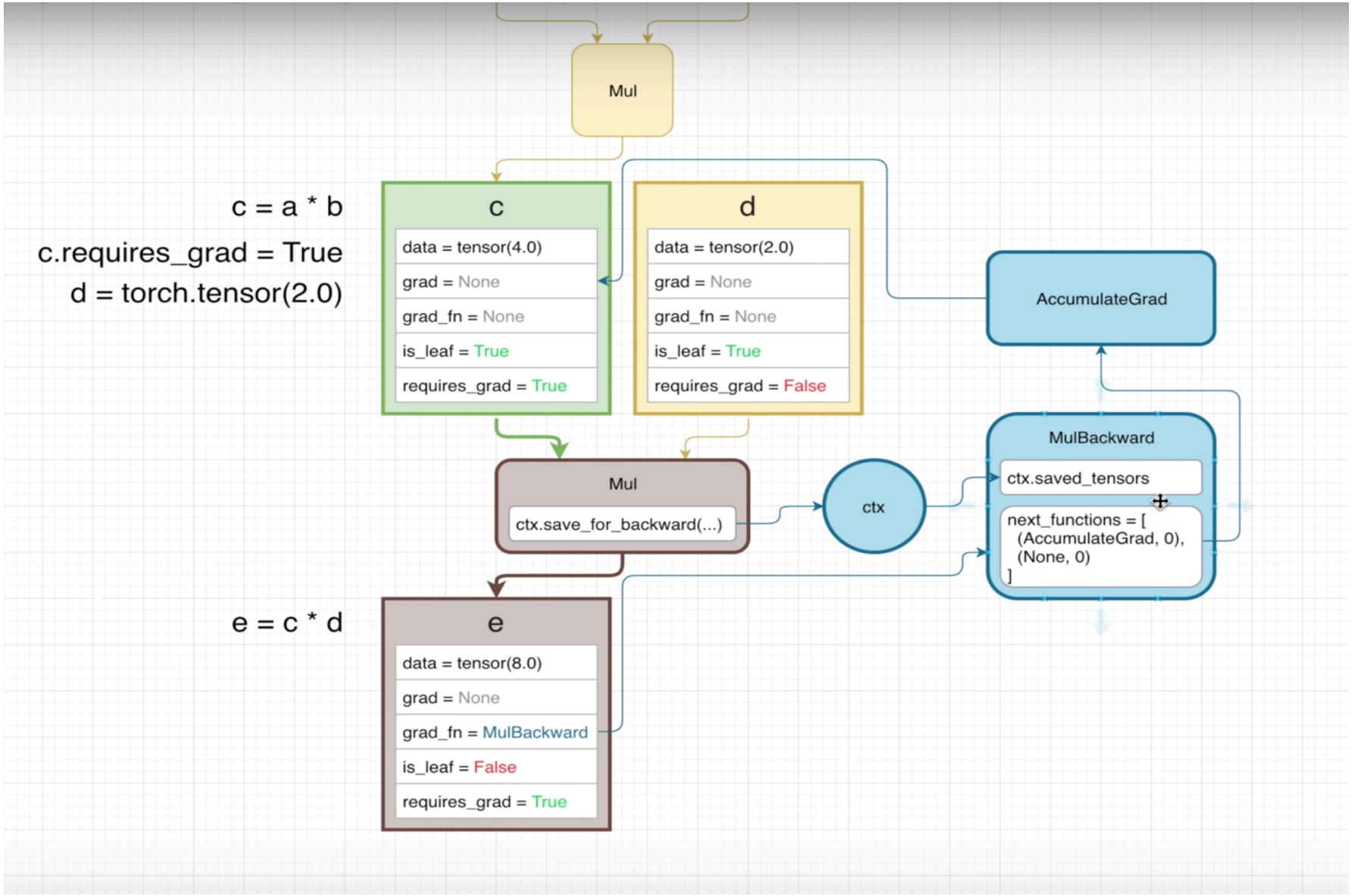
How Computational graphs are constructed in Pytorch e.g (3)



How Computational graphs are constructed in Pytorch e.g (3)



How Computational graphs are constructed in Pytorch e.g (3)



References

- Pytorch AUTOGRAD: AUTOMATIC DIFFERENTIATION
- Mustafa Alghali, how-pytorch-backward-function-works
- Elliot Waite, PyTorch Autograd Explained - In-depth Tutorial