PYTORCH AUTOGRAD

AUTOMATIC DIFFERENTIATION

Gene Olafsen

REFERENCE MATERIAL

- https://www.residentmar.io/2019/07/07/python-mixins.html
- https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytorchs-magic-2686cd94ec95

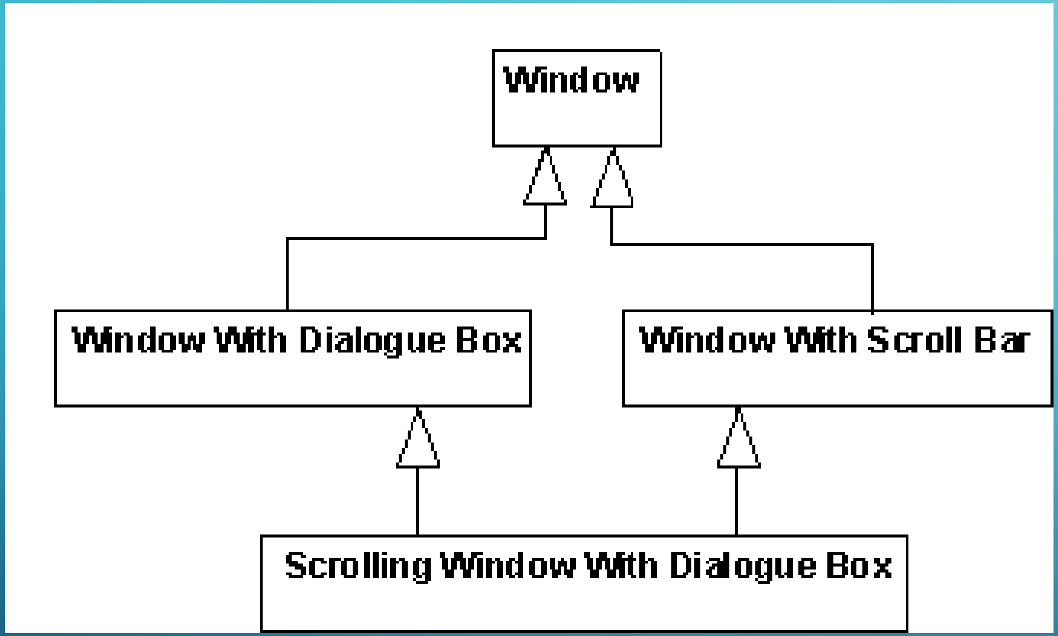
DEFINED

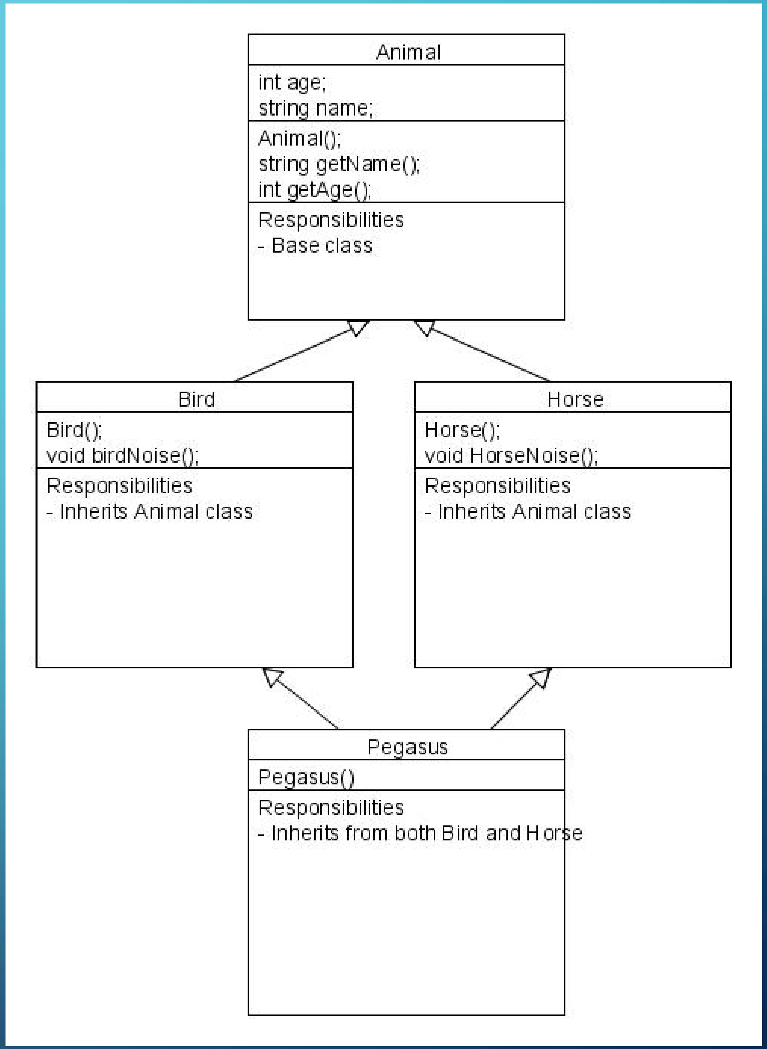
- "The `autograd` package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different." --Pytorch Documentation

BACKGROUND

- PyTorch Model Composition
- PyTorch Overloaded Operators

MULTIPLE INHERITANCE

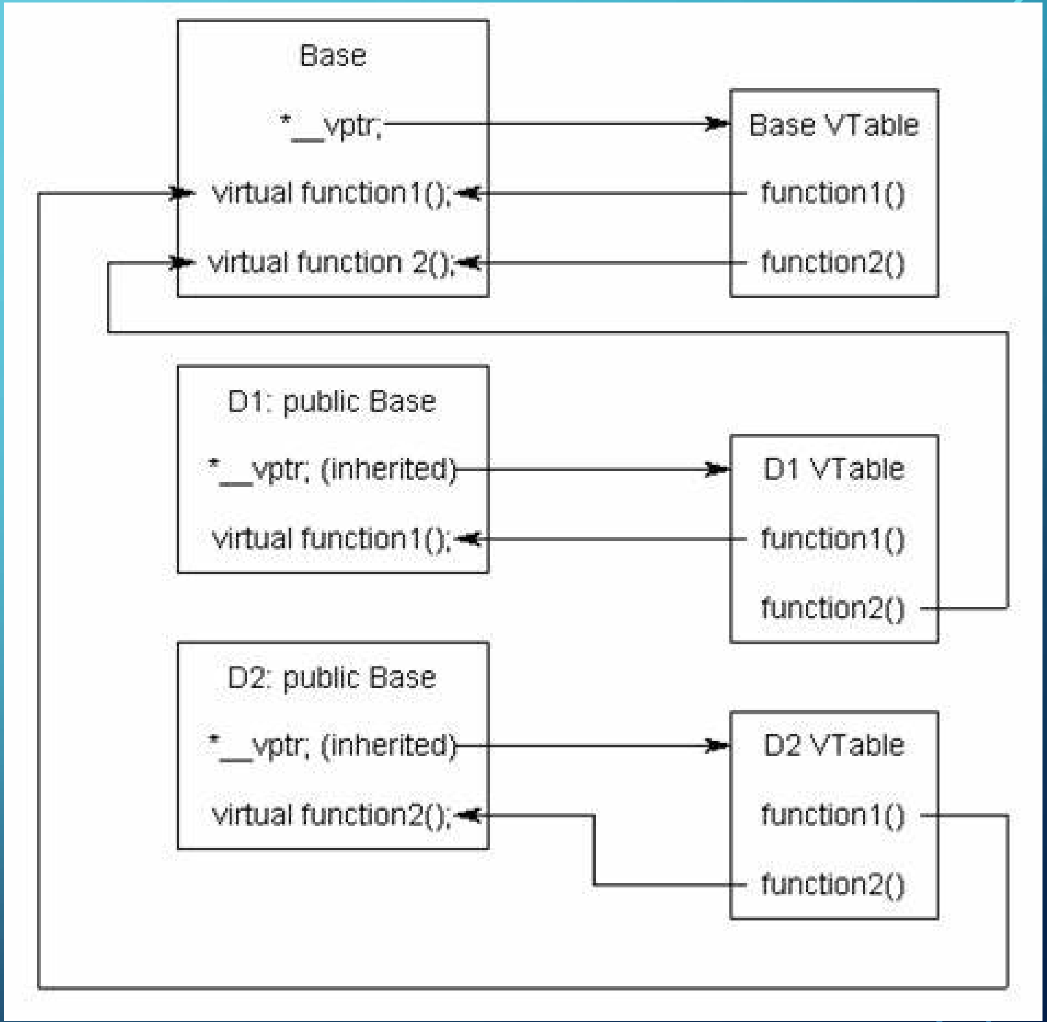




So called "Diamond of Death"

V-TABLE

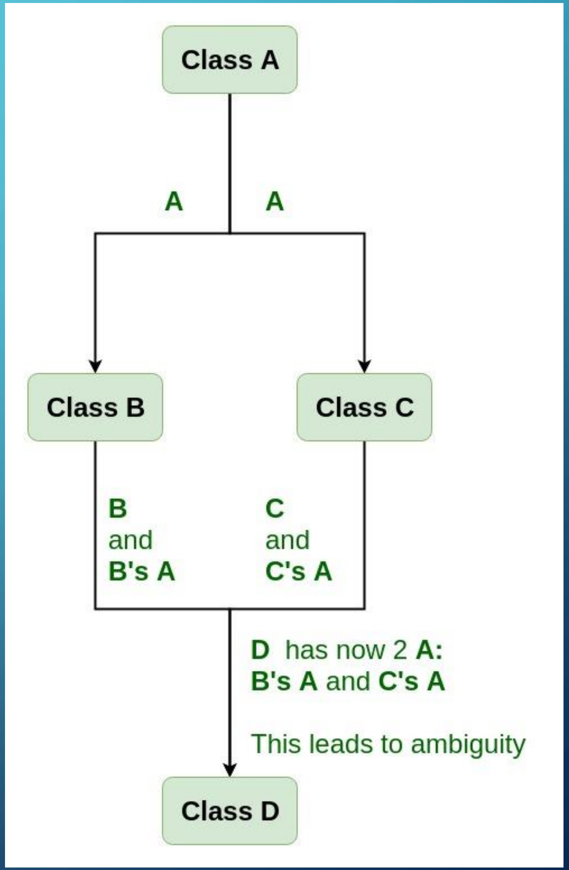
- In this diagram:
    - D1 overrides *function1* and provides its own implementation. It inherits *function2* from the base.
    - D2 inherits *function1* from the base and implements its own version of *function2*.



VIRTUAL BASE CLASS

- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

```
class B : virtual
public A
{
};

class C : public
virtual A
{
};
```



PYTHON MULTIPLE INHERITANCE

```
class SuperClassA:
    def __init__(self):
        self.a = 1

class SuperClassB:
    def __init__(self):
        self.b = 1

class SubClass(SuperClassA, SuperClassB):
    def __init__(self):
        super().__init__()

obj = SubClass()
# trivia question: which of self.a and self.b is defined?
```

PYTHON MULTIPLE INHERITANCE SOLUTION

- Python solves this problem using a scary-sounding algorithm called C3 linearization, but this is a fragile solution.
- It is very easy to change or add or rename a method on a class, or to modify an inheritance hierarchy in some way, and in doing so inadvertently change which parent's version of a function or attributed is being called on in your code.

GUIDO VAN ROSSUM

- Python's Guido van Rossum summarizes C3 superclass linearization thusly:
- Basically, the idea behind C3 is that if you write down all of the ordering rules imposed by inheritance relationships in a complex class hierarchy, the algorithm will determine a monotonic ordering of the classes that satisfies all of them. If such an ordering can not be determined, the algorithm will fail.

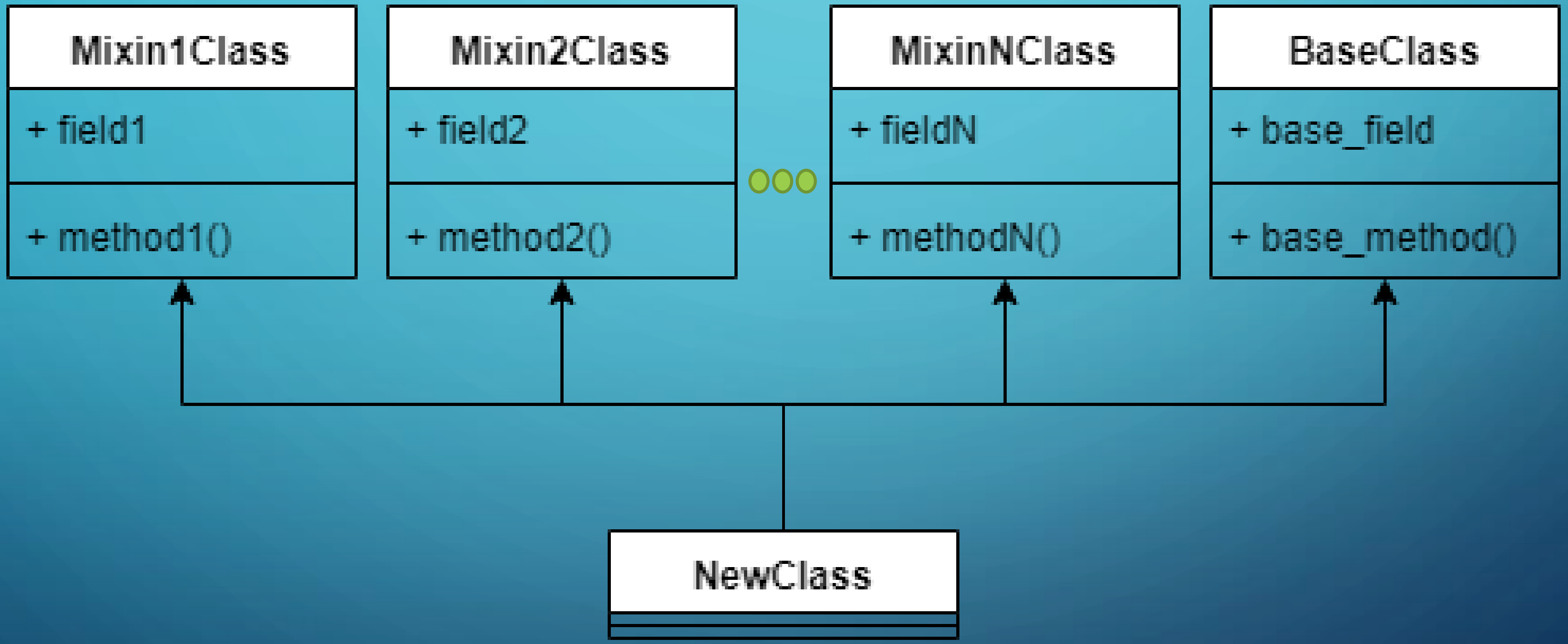
AVOIDING MULTIPLE INHERITANCE

- Single inheritance can be used to avoid the pitfalls of multiple inheritance. The downside is that you may end up with a deep inheritance chain or many different class heirarchies where each class relies on a subset of library features.

MIXINS

- **Mixins** are an alternative class design pattern that avoids both single-inheritance class fragmentation and multiple-inheritance diamond dependencies.
- A mixin is a class that defines and implements a single, well-defined feature. Subclasses that inherit from the mixin inherit this feature—and nothing else.
- By convention, mixins in python end in "Mixin"

MIXIN SINGLE RESPONSIBILITY



MIXIN ADVANTAGE

- **Mixins are a safe form of multiple inheritance.**
  - They enforce a new constraint on your classes: that all functionality relating to a specific feature must live in the appropriate mixin. Thus methods thus can't be defined in more than one place, and thus can't fall prey to diamond inheritance problems.
- **Mixins are more legible than single inheritance classes.**
  - Flat "single-level" inheritance (courtesy of multiple inheritance!) and clear division of labor on a feature-to-feature basis makes it obvious which parent class is responsible for which object properties. In fact, mixins make it so obvious which features an object supports that oftentimes you can read it right off of the class signature:

MIXIN CODE EXAMPLE

```
•    # supports no optional parameters
•     class SimpleDiagram(Plot)
•
•     # supports 'hue', 'legend', and 'clip' optional parameters
•     class LegendDiagram(Plot, HueMixin, LegendMixin, ClipMixin)
•
•     # supports 'hue', 'scale', and 'legend' optional parameters
•     class CartogramDiagram(Plot, HueMixin, ScaleMixin, LegendMixin)
```

OVERLOADED OPERATORS

- Just like NumPy, PyTorch overloads a number of python operators to make PyTorch code shorter and more readable.
- What
  - Operator Overloading means giving extended meaning beyond their predefined operational meaning.

OPERATORS

- Combined operators execute overloads as well.
  - x += y
  - x **= 2
- **Note:** Python doesn’t allow overloading *and*, *or*, and *not* keywords.

```
z = -x  # z = torch.neg(x)
z = x + y  # z = torch.add(x, y)
z = x - y
z = x * y  # z = torch.mul(x, y)
z = x / y  # z = torch.div(x, y)
z = x // y
z = x % y
z = x ** y  # z = torch.pow(x, y)
z = x @ y  # z = torch.matmul(x, y)
z = x > y
z = x >= y
z = x < y
z = x <= y
z = abs(x)  # z = torch.abs(x)
z = x & y
z = x | y
z = x ^ y  # z = torch.logical_xor(x, y)
z = ~x  # z = torch.logical_not(x)
z = x == y  # z = torch.eq(x, y)
z = x != y  # z = torch.ne(x, y)
```

```
# Python code illustrating how
# to overload the + operator

class MyCombine:
    def __init__(self, a):
        self.a = a

    # combine two objects
    def __add__(self, o):
        return self.a + o.a

obj1 = MyCombine(1)
obj2 = MyCombine(2)
obj3 = MyCombine("Hello")
obj4 = MyCombine("World")

print(obj1 + obj2)
print(obj3 + obj4)
```

```
3
HelloWorld
```

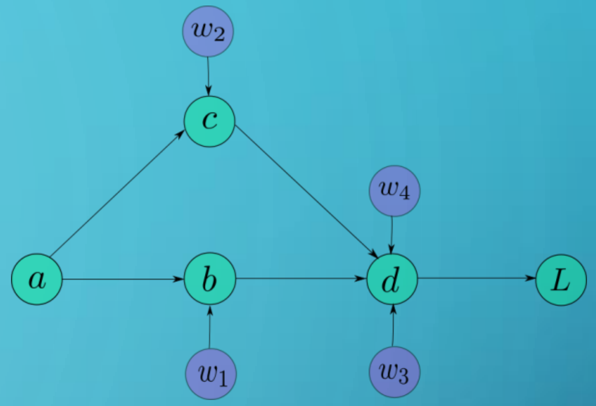
PYTORCH TENSORS

- N-dimensional array
- Support additional enhancements
  - Can be loaded or the GPU for faster computations.
  - On setting `.requires_grad = True` they start forming a backward graph that tracks every operation applied on them to calculate the gradients via a dynamic computation graph (DCG)
- **Note:** PyTorch only allows gradients to be calculated for floating point tensors.

DEPRECATED PYTORCH VARIABLE CLASS

- In earlier versions of PyTorch, the *torch.autograd.Variable* class was used to create tensors that support gradient calculations and operation tracking.
- As of PyTorch v0.4.0 the *Variable* class has been deprecated.
- *torch.Tensor* now contains the functionality of *torch.autograd.Variable*

AUTOGRAD CLASS



- An engine to calculate derivatives (Jacobian-vector product).
- Autograd records a graph of all the operations performed on gradient enabled tensors and creates an acyclic graph.
  - The leaves of this graph are input tensors
  - The roots of this graph are output tensors.
  - Gradients are calculated by tracing the graph from the root to the leaf and multiplying every gradient in the way using the chain rule.

NEURAL NETWORK TRAINING REFRESHER

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

TRAINING ITERATION

- The Dynamic Computation Graph (DCG) is built from scratch in every iteration providing maximum flexibility to gradient calculation.
- Consider the following code. The forward multiplication operation is performed by a Pytorch overload named *Mul*. PyTorch's backward operation named *MulBackward* is dynamically integrated in the backward graph for computing the gradient.

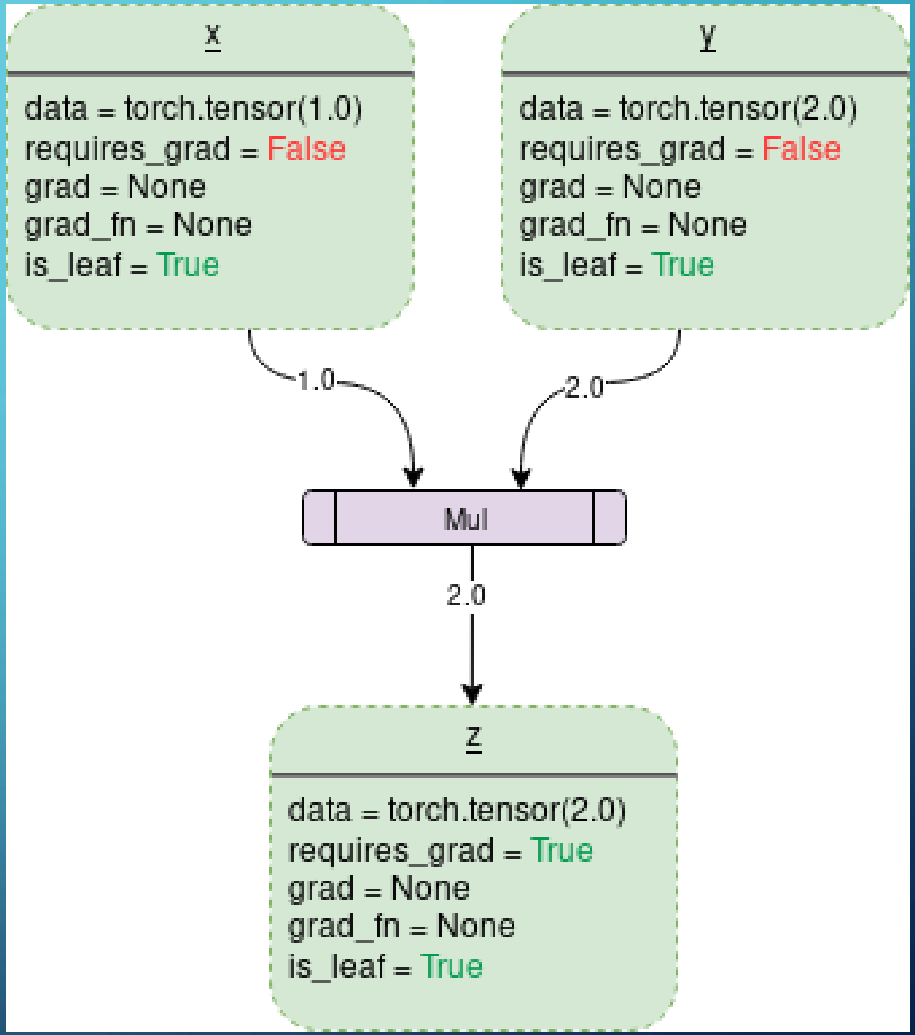
```
import torch

# Creating the graph
x = torch.tensor(1.0, requires_grad = True)
y = torch.tensor(2.0)
z = x * y
```

HENCE

- You don’t have to encode all possible paths before you launch the training — what you run, is what you differentiate.
- If your forward method had branching logic in it, only the codepath that is executed, contributes to the computation graph.

REQUIRES_GRAD = FALSE

- data: a 1x1 tensor holding values of either 1.0 or 2.0
- drad: contains the value of the gradient.
- grad_fn: identifies the 'backward' function to calculate the gradient.
- is_leaf: A node is leaf if :
  - It was initialized explicitly by some function like `x = torch.tensor(1.0)` or `x = torch.randn(1, 1)`
  - It is created after operations on tensors which all have `requires_grad = False.`
  - It is created by calling `.detach()` method on some tensor.



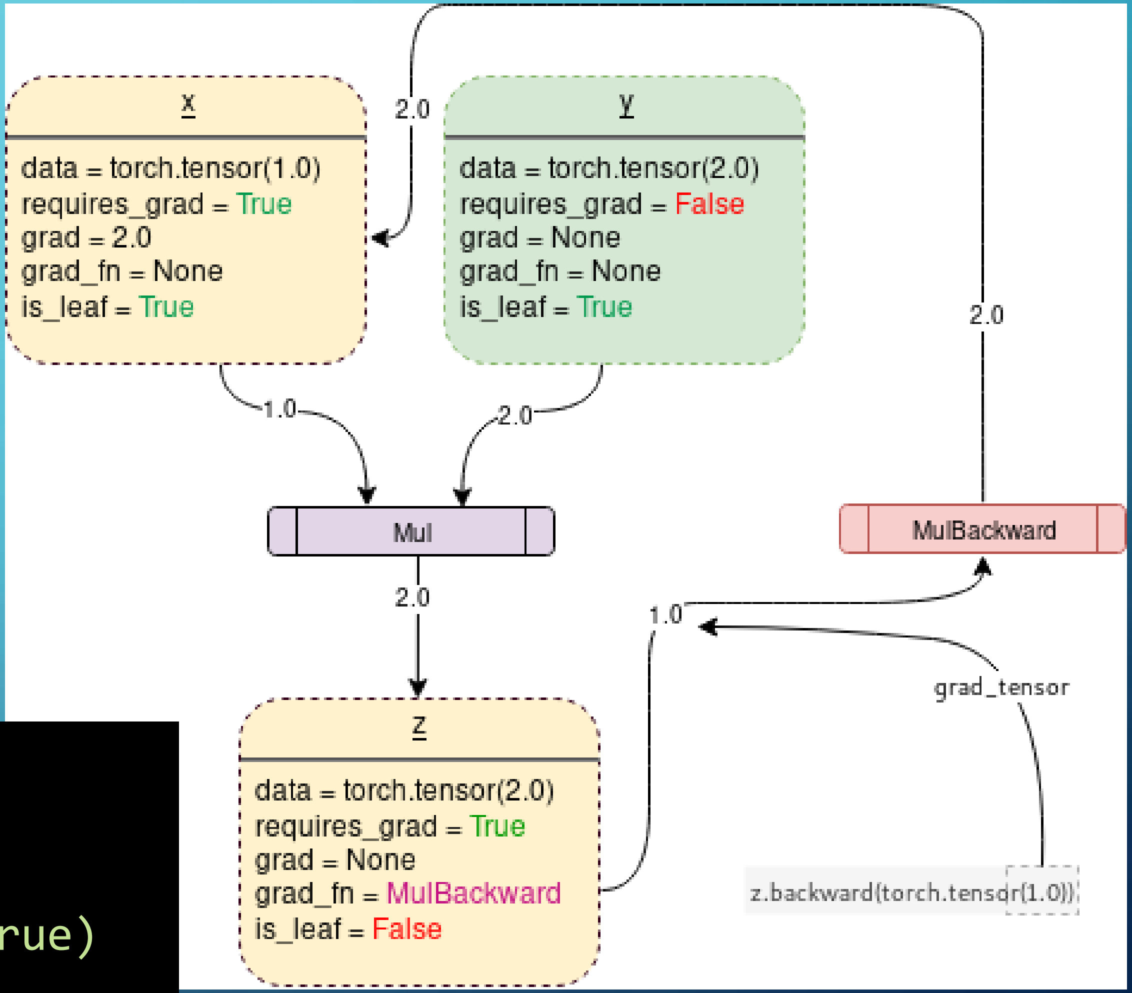
BACK'WARD' TO THE FUTURE

- Let's get ready to turn the requires_grad = True
- The backward() function is responsible for calculating the gradient.
- On calling `backward()`, gradients are populated only for the nodes which have both:
  - `requires_grad = True`
  - `is_leaf` = True.
- Gradients are of the output node from which `.backward()` is called, with respect to other leaf nodes.

REQUIRES_GRAD = TRUE

- On turning *requires_grad = True* PyTorch will start tracking the operation and store the gradient functions at each step.

```
import torch

# Creating the graph
x = torch.tensor(1.0, requires_grad = True)
y = torch.tensor(2.0)
z = x * y
```


x
data = torch.tensor(1.0)
requires_grad = True
grad = 2.0
grad_fn = None
is_leaf = True
y
data = torch.tensor(2.0)
requires_grad = False
grad = None
grad_fn = None
is_leaf = True
2.0
1.0
2.0
Mul
MulBackward
2.0
1.0
grad_tensor
z
data = torch.tensor(2.0)
requires_grad = True
grad = None
grad_fn = MulBackward
is_leaf = False
z.backward(torch.tensor(1.0))