PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH
PYTORCH PYTORCH PYTORCH PYTORCH PYTORCH

PYTORCH

Automatic Differentiation in PyTorch

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, Adam Lerer, ...

Operator Overloading - intro

Basic idea: overload operators / use custom wrapper types

Every type an operation is performed, perform it **and record it in a "tape"** (for reverse mode AD).

Does this code support AD?

```
###########################
x = np.ones((100, 100))
y = np.matmul(x, x.T)
```

Operator Overloading - intro

Basic idea: overload operators / use custom wrapper types

Every type an operation is performed, perform it **and record it in a "tape"** (for reverse mode AD).

Does this code support AD?

```
import numpy as np
x = np.ones((100, 100))
y = np.matmul(x, x.T)
```

Operator Overloading - intro

Basic idea: overload operators / use custom wrapper types

Every type an operation is performed, perform it **and record it in a "tape"** (for reverse mode AD).

Does this code support AD?

```
import autograd.numpy as np
x = np.ones((100, 100))
y = np.matmul(x, x.T)
```

Operator Overloading - pros and cons

✅ Programs are expressed in the host language
✅ Arbitrary control flow allowed and handled correctly
✅ Can be built to mimic existing interfaces
✅ Less to learn. Smaller mental overhead
✅ Debugging is easier
❌ Optimization is much harder
❌ Need to use the host language interpreter
❌ AD data structures get as large as the number of operators used

Why?

- All the benefits of OO-based AD
- **A reverse-mode AD implementation with near-zero overhead.**
- **Effective memory management.**
- **In-place support.**
- Extensibility

A simple example

```
import torch
from torch.autograd import Variable

B, F = 1000, 10
X = Variable(torch.randn(B, F))
Y = Variable((X * torch.randn(1, F)).sum(1) + torch.randn(B))
W = Variable(torch.randn(F, F), requires_grad=True)

lr = 1e-3
for i in range(100):
    dW = autograd.grad(torch.matmul(W, X).sub(Y).pow(2).mean(), W)
    W.data -= lr * dW.data
```

A simple example

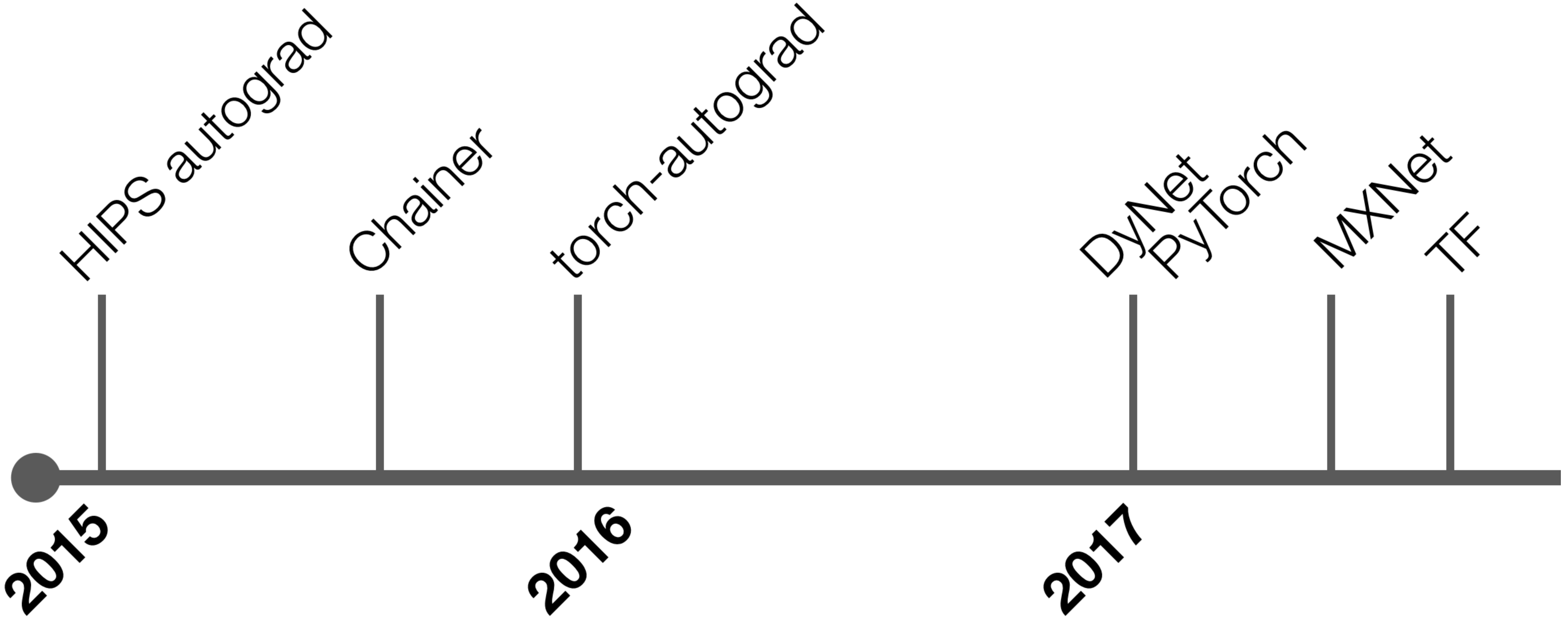
```python
import torch
from torch.autograd import Variable

B, F = 1000, 10
X = Variable(torch.randn(B, F))
Y = Variable((X * torch.randn(1, F)).sum(1) + torch.randn(B))
W = Variable(torch.randn(F, F), requires_grad=True)

lr = 1e-3
for i in range(100):
    W.grad.zero_()
    loss = torch.matmul(W, X).sub(Y).pow(2).mean()
    loss.backward()
    W.data -= lr * W.grad.data
```

Minimizing the overhead

+

Memory management

Operator Overloading revolution



Efficiency

Machine Learning/Deep Learning frameworks mostly relied on symbolic graphs.

Efficiency

Machine Learning/Deep Learning frameworks mostly relied on symbolic graphs.

All other approaches thought to be as slow and impractical.

Efficiency

Machine Learning/Deep Learning frameworks mostly relied on symbolic graphs.

All other approaches thought to be as slow and impractical.
(But were they really?)

Efficiency

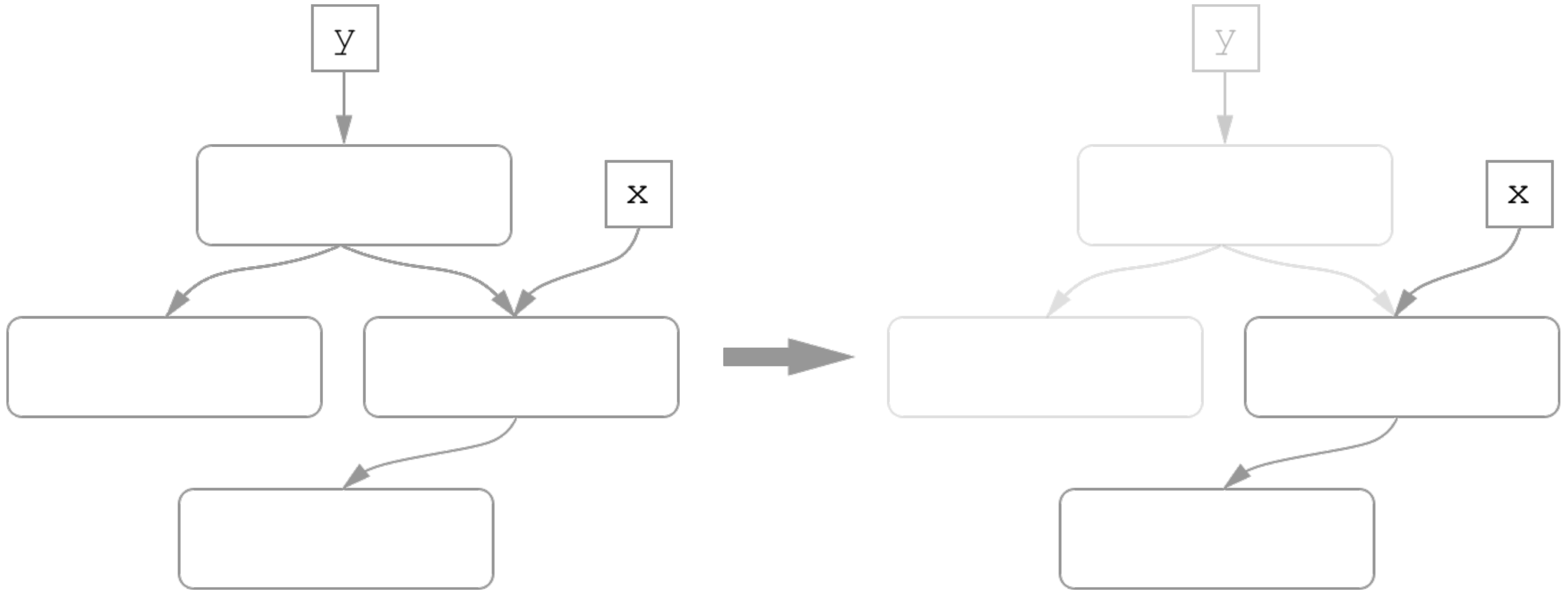
Machine Learning/Deep Learning frameworks mostly relied on symbolic graphs.

All other approaches thought to be as slow and impractical. (But were they really?)

Models in some domains require fine-grained control flow, and individual operations are performed on tiny arrays.

Lifetime of data structures

Outputs keep graph alive. Dead branches eliminated automatically thanks to reference counting.



Disabling AD

Data can be marked as "not requiring gradient", which allows to save memory and improve performance.

```
def model(x, W, b):
    return torch.matmul(W, x) + b[None, :]

x = Variable(...)
y = Variable(...)
W = Variable(..., requires_grad=True)
b = Variable(..., requires_grad=True)

(model(x, W, b) - y).pow(2).backward()
assert x.grad is None and y.grad is None
```

Efficiency-oriented syntax

Extension syntax encouraging retaining only a necessary subset of state.

```python
class Tanh(autograd.Function):
    @staticmethod
    def forward(ctx, x):
        y = x.tanh()
        ctx.save_for_backward(y)
        return y

    @staticmethod
    def backward(ctx, grad_y):
        y, = ctx.saved_variables
        return grad_y * (1 - y ** 2)
```

In-place support

Why is in-place useful?

- Enables writing more expressive code
  - Assignments are common and natural
  - Enables differentiation of a larger class of programs
- Improves memory usage
  - Potentially also increases cache hit rates

DenseNet

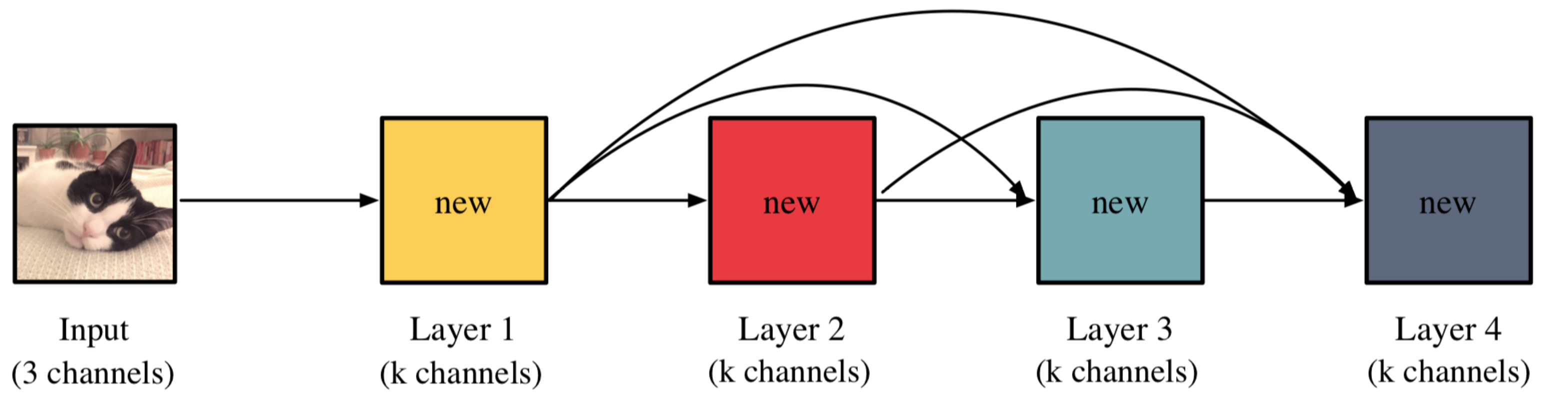


Figure 1: High-level illustration of the DenseNet architecture.

```
features = [input]
for conv, bn in zip(self.conv_layers, self.bn_layers):
  out = bn(conv(torch.cat(features, dim=1)))
  features.append(out)
return torch.cat(features)
```

$O(l^2)$ space complexity

Memory efficient DenseNet[1]

```
features = [input]
for conv, bn in zip(self.conv_layers, self.bn_layers):
  out = bn(conv(torch.cat(features, dim=1)))
  features.append(out)
return torch.cat(features)

################################################################################

features = Variable(torch.Tensor(batch_size, l * k,
                                 height, width))
features[:, :l] = input
for i, (conv, bn) in enumerate(zip(self.conv_layers, self.bn_layers)):
  out = bn(conv(features[:(i + 1) * l]))
  features[:, (i + 1) * l:(i + 2) * l] = out
return features
```

[1] Memory-Efficient Implementation of DenseNets: Geoff Pleiss et al.

Why is supporting in-place hard?

Invalidation

Consider this code:

```
y = x.tanh()
y.add_(3)
y.backward()
```

Recall that $\tanh'(x) = 1 - \tanh^2(x)$.

We have to ensure that in-place operations don't overwrite memory saved for reverse phase.

Invalidation - solution

```
def tanh_forward(ctx, x):
    y = torch.tanh(x)
    ctx.save_for_backward(y)
    return y

def tanh_backward(ctx, grad_y):
    y, = ctx.saved_variables
    return grad_y * (1 - y ** 2)

################################################################################

y = x.tanh()
y.add_(3)
y.backward()
```

Invalidation - solution

```
def tanh_forward(ctx, x):
    y = torch.tanh(x)
    ctx.save_for_backward(y)
    return y

def tanh_backward(ctx, grad_y):
    y, = ctx.saved_variables
    return grad_y * (1 - y ** 2)

################################################################################

y = x.tanh()
y.add_(3)
y.backward()
```

Invalidation - solution

```
def tanh_forward(ctx, x):
    y = torch.tanh(x)                        # y._version == 0
    ctx.save_for_backward(y)
    return y

def tanh_backward(ctx, grad_y):
    y, = ctx.saved_variables
    return grad_y * (1 - y ** 2)

################################################################################

y = x.tanh()
y.add_(3)
y.backward()
```

Invalidation - solution

```
def tanh_forward(ctx, x):
    y = torch.tanh(x)                       # y._version == 0
    ctx.save_for_backward(y)                # saved_y._expected_version == 0
    return y

def tanh_backward(ctx, grad_y):
    y, = ctx.saved_variables
    return grad_y * (1 - y ** 2)

################################################################################

y = x.tanh()                                # y._version == 0
y.add_(3)
y.backward()
```

Invalidation - solution

```
def tanh_forward(ctx, x):
    y = torch.tanh(x)
    ctx.save_for_backward(y)
    return y

def tanh_backward(ctx, grad_y):
    y, = ctx.saved_variables
    return grad_y * (1 - y ** 2)

################################################################################

y = x.tanh()
y.add_(3)                                      # y._version == 1
y.backward()
```

Invalidation - solution

```python
def tanh_forward(ctx, x):
    y = torch.tanh(x)
    ctx.save_for_backward(y)
    return y

def tanh_backward(ctx, grad_y):
    y, = ctx.saved_variables                # ERROR: version mismatch
    return grad_y * (1 - y ** 2)

################################################################################

y = x.tanh()
y.add_(3)
y.backward()
```

Data versioning

- Shared among all Variables (partially) aliasing same data.
    - An overapproximation, but works well in practice.
- It would be possible to lazily clone the data, but this makes reasoning about performance harder.

Dealing with aliasing data

Aliasing data

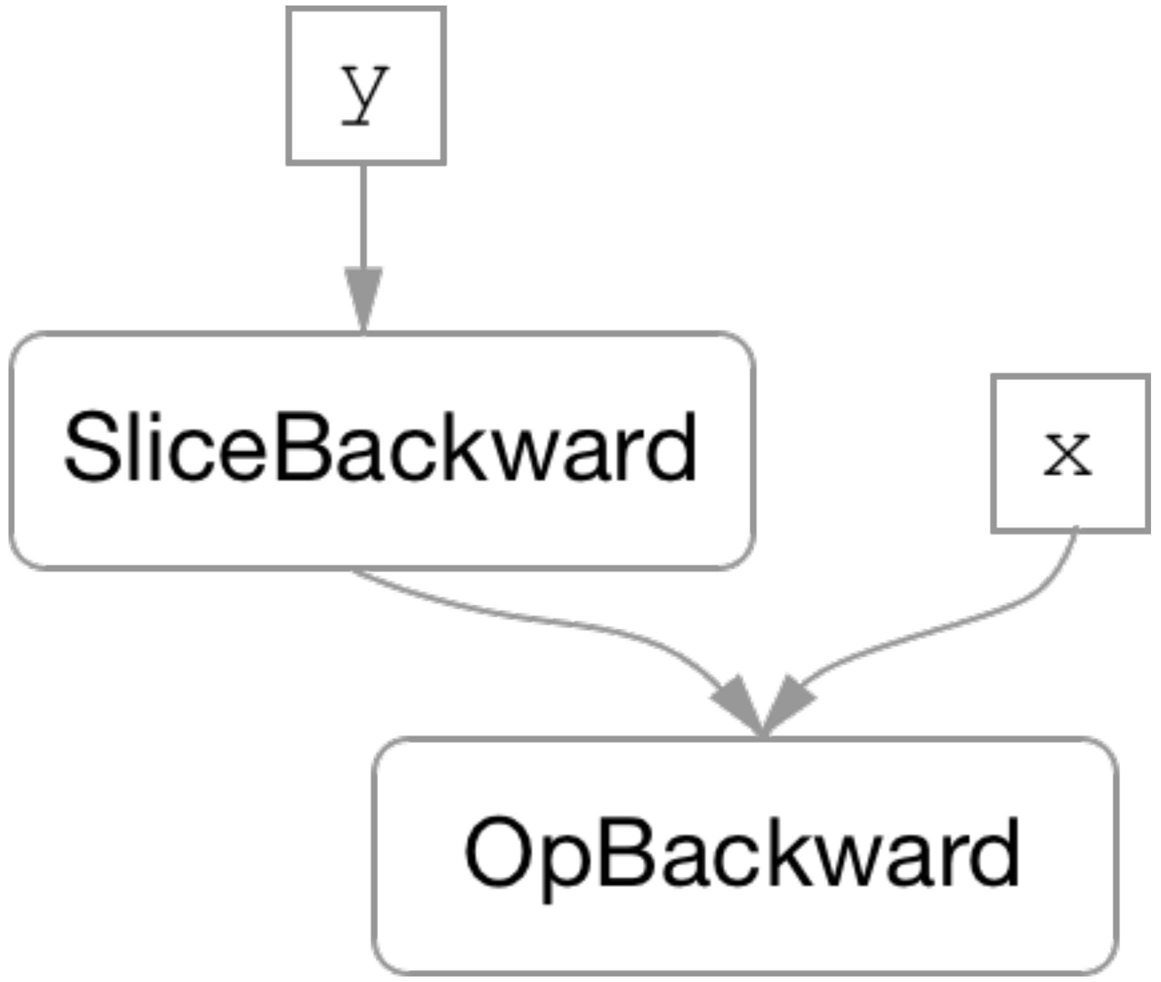
Consider this code:

```
y = x[:2]
y.mul_(3)
x.backward()
```

Aliasing data

Consider this code:

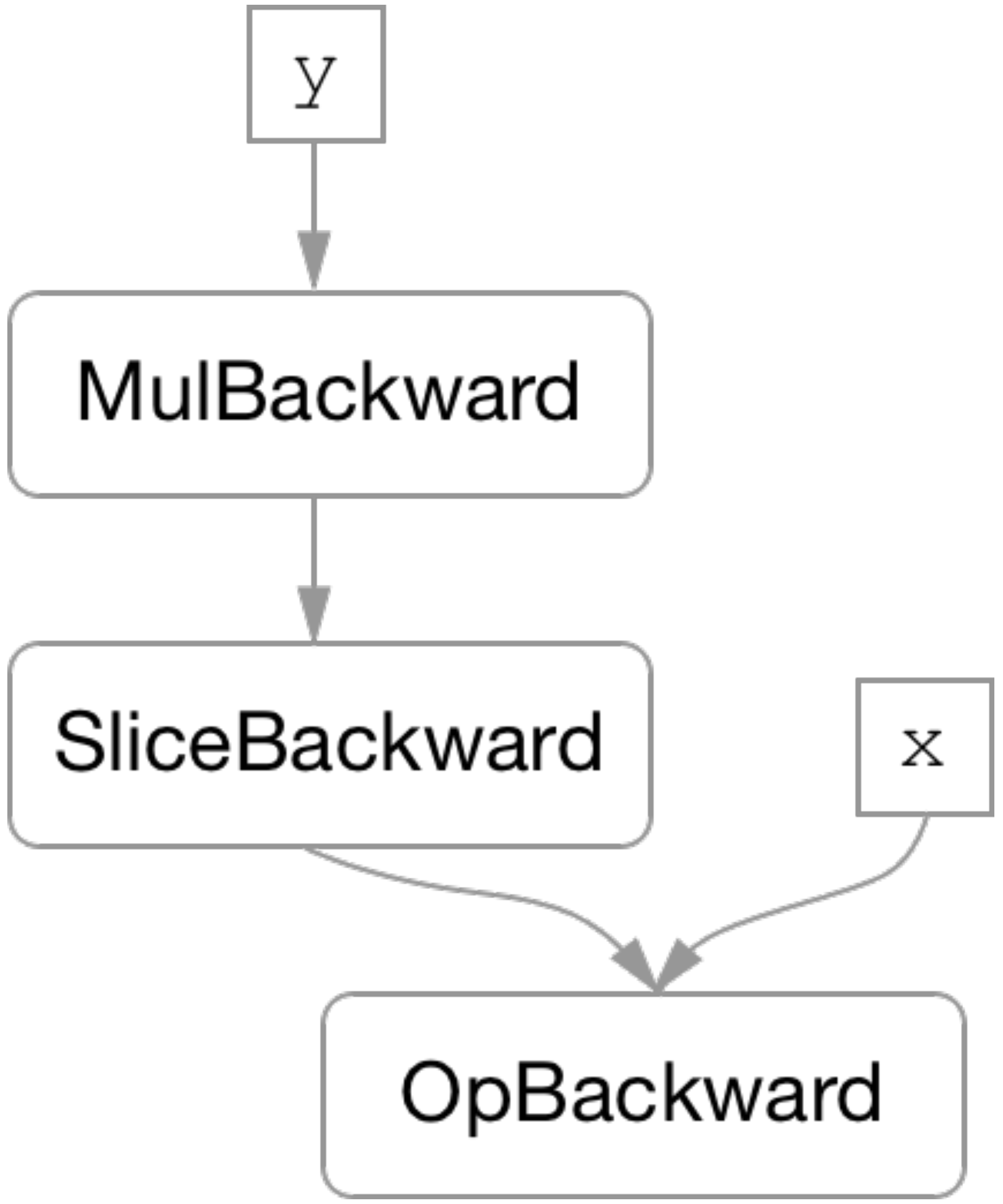
```
y = x[:2]
y.mul_(3)
x.backward()
```



Aliasing data

Consider this code:

```
y = x[:2]
y.mul_(3)
x.backward()
```

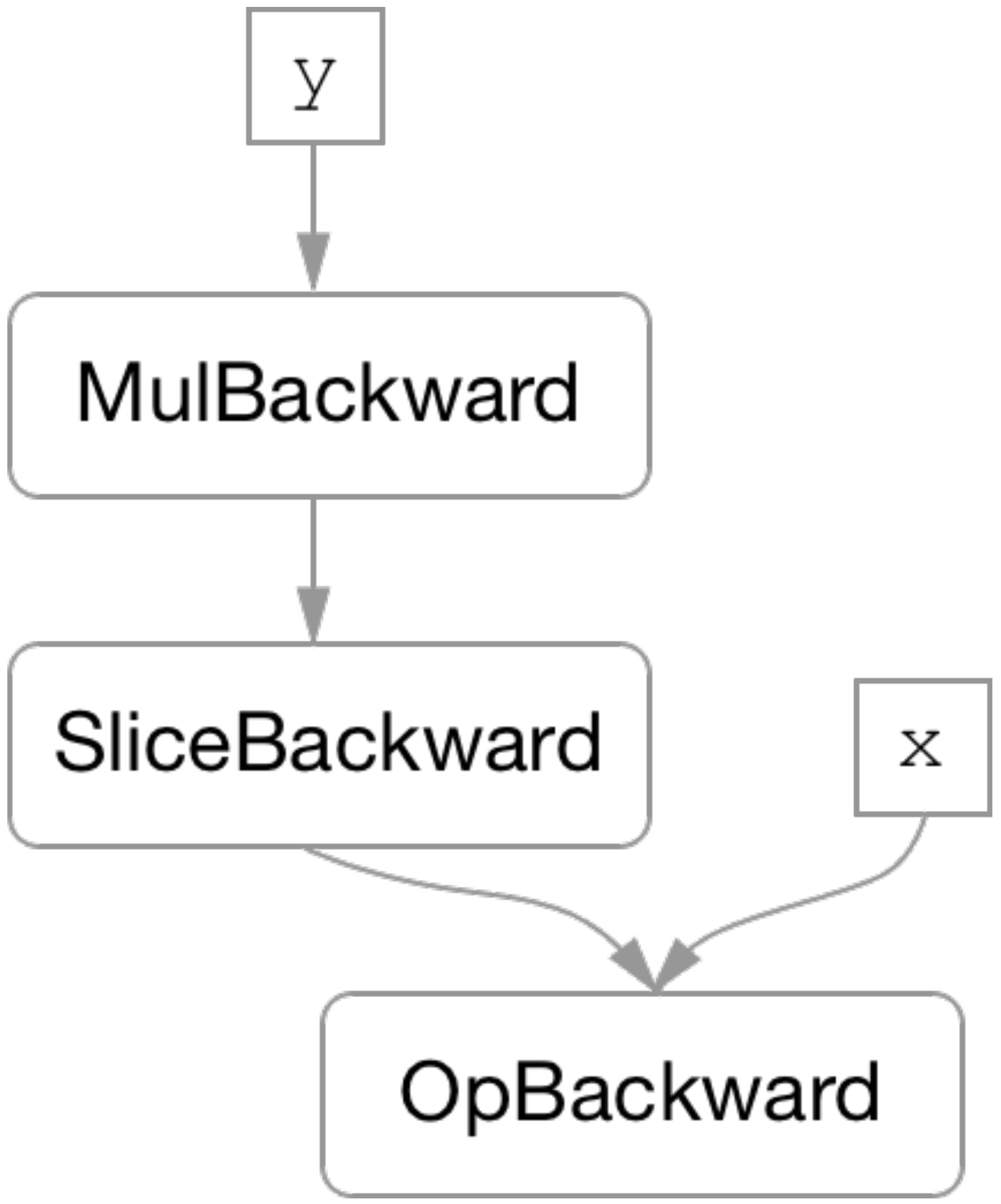


Aliasing data

Consider this code:

```
y = x[:2]
y.mul_(3)
x.backward()
```

x doesn't have the derivative of `mul()` in its trace!



$$\begin{tikzcd} y \arrow[d] & \\ \text{MulBackward} \arrow[d] & \\ \text{SliceBackward} \arrow[dr] & x \arrow[d] \\ & \text{OpBackward} \end{tikzcd}$$

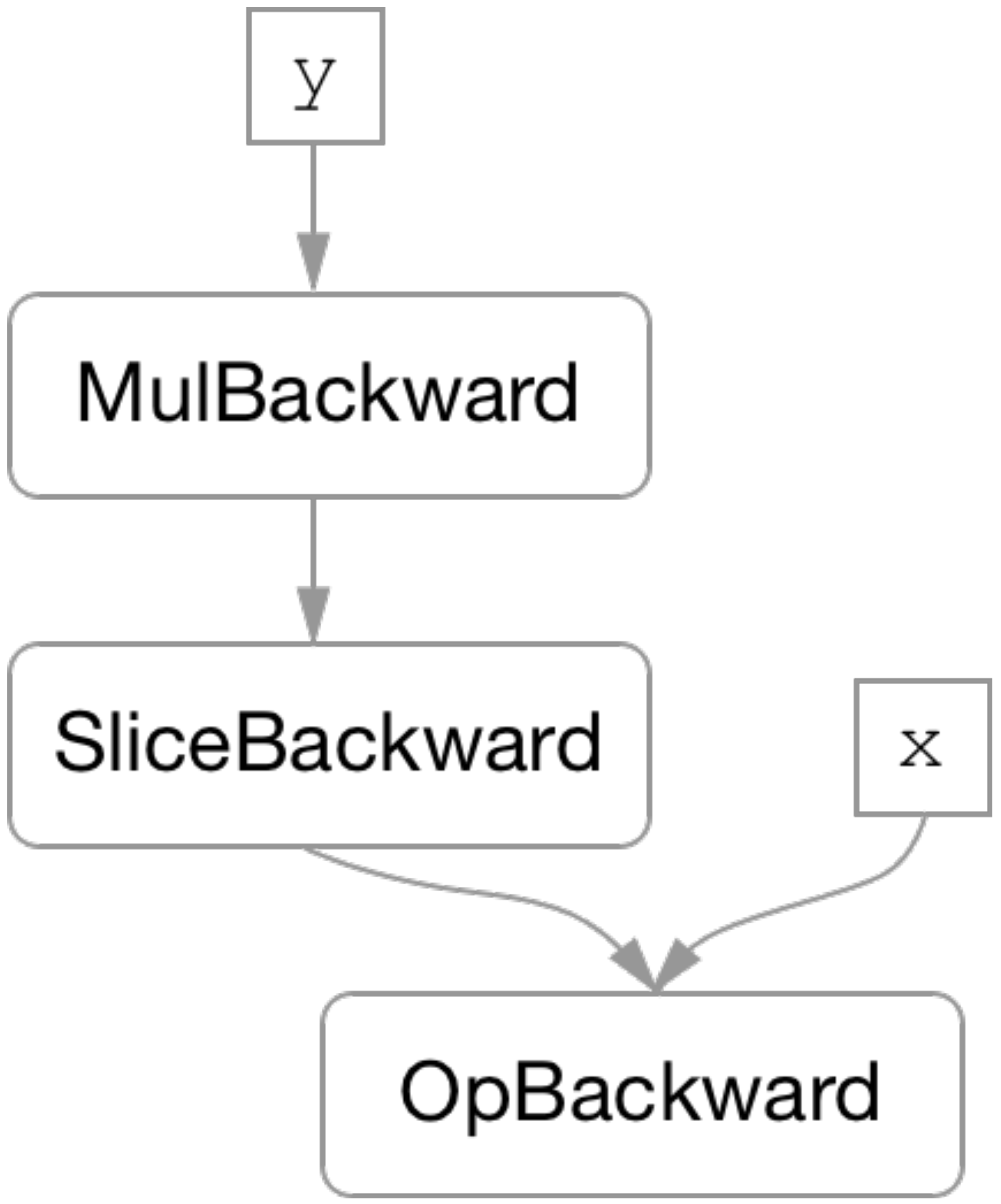
Aliasing data

Consider this code:

```
y = x[:2]
y.mul_(3)
x.backward()
```

NB: this also works the other way around:

```
y = x[:2]
x.mul_(3)
y.backward()
```



Problems

Arrays aliasing the same data share part of their trace, but have their own parts as well.

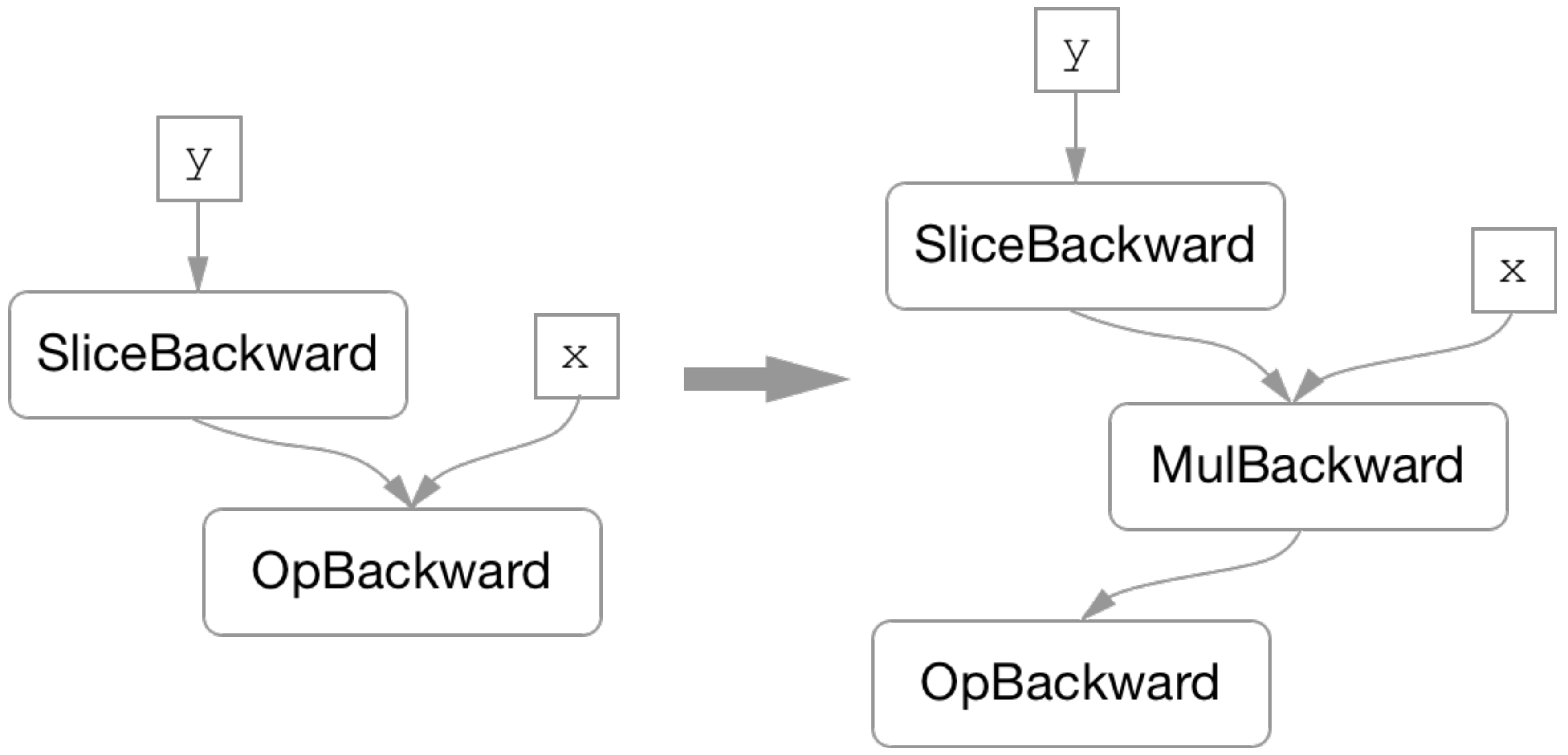
Problems

Arrays aliasing the same data share part of their trace, but have their own parts as well.

Different cases need to be handled differently (2 examples from the previous slide).

Observations

We need a mechanism to "rebase" traces onto different parts of the graph.



Observations

Eager updates would be too expensive.

```
def multiplier(i):
    ...

x = Variable(torch.randn(B, N), requires_grad=True)
for i, sub_x in enumerate(torch.unbind(x, 1)):
    sub_x.mul_(multiplier(i))
```

Observations

Eager updates would be too expensive.

```
def multiplier(i):
    ...

x = Variable(torch.randn(B, N), requires_grad=True)
for i, sub_x in enumerate(torch.unbind(x, 1)):
    sub_x.mul_(multiplier(i))
```

$O(N^2)$ "rebases"

Composing viewing operations

PyTorch uses the standard nd-array representation:
- data pointer
- data offset
- sizes for each dimension
- strides for each dimension

Composing viewing operations

PyTorch uses the standard nd-array representation:
- data pointer
- data offset
- sizes for each dimension
- strides for each dimension

If `x` is a 3-d array then:

```
addressof(x[2, 3, 4]) = x.data_ptr + x.data_offset +
                        2 * x.stride[0] +
                        3 * x.stride[1] +
                        4 * x.stride[2]
```

Composing viewing operations

PyTorch uses the standard nd-array representation:
- data pointer
- data offset
- sizes for each dimension
- strides for each dimension

Every viewing operation can be expressed in terms of a formula that transforms the metadata.

Composing viewing operations

PyTorch uses the standard nd-array representation:
- data pointer
- data offset
- sizes for each dimension
- strides for each dimension

Every viewing operation can be expressed in terms of a formula that transforms the metadata.

Composition of viewing operations can also be represented as a single transform.

Solution

We will need a concept of "base" and "view" arrays.

Every base can have arbitrarily many views, but every view has a single base.

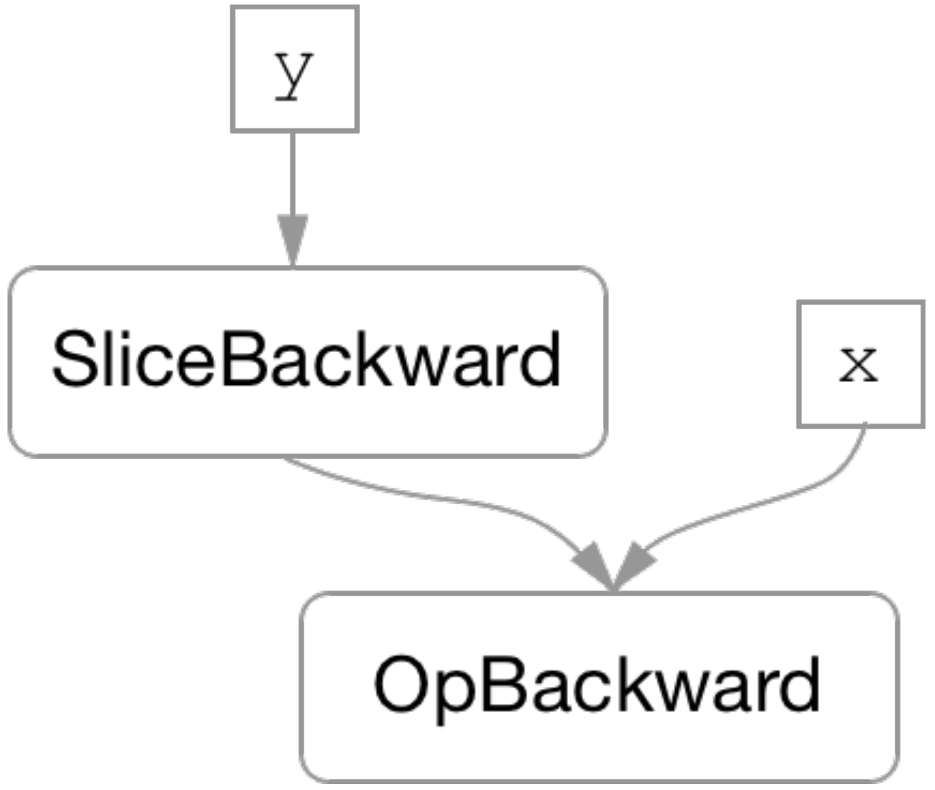
Views always share storage with their base.

In-place modifications of any of group members affect all of them.

Parts of metadata (trace pointers) need to be updated lazily.

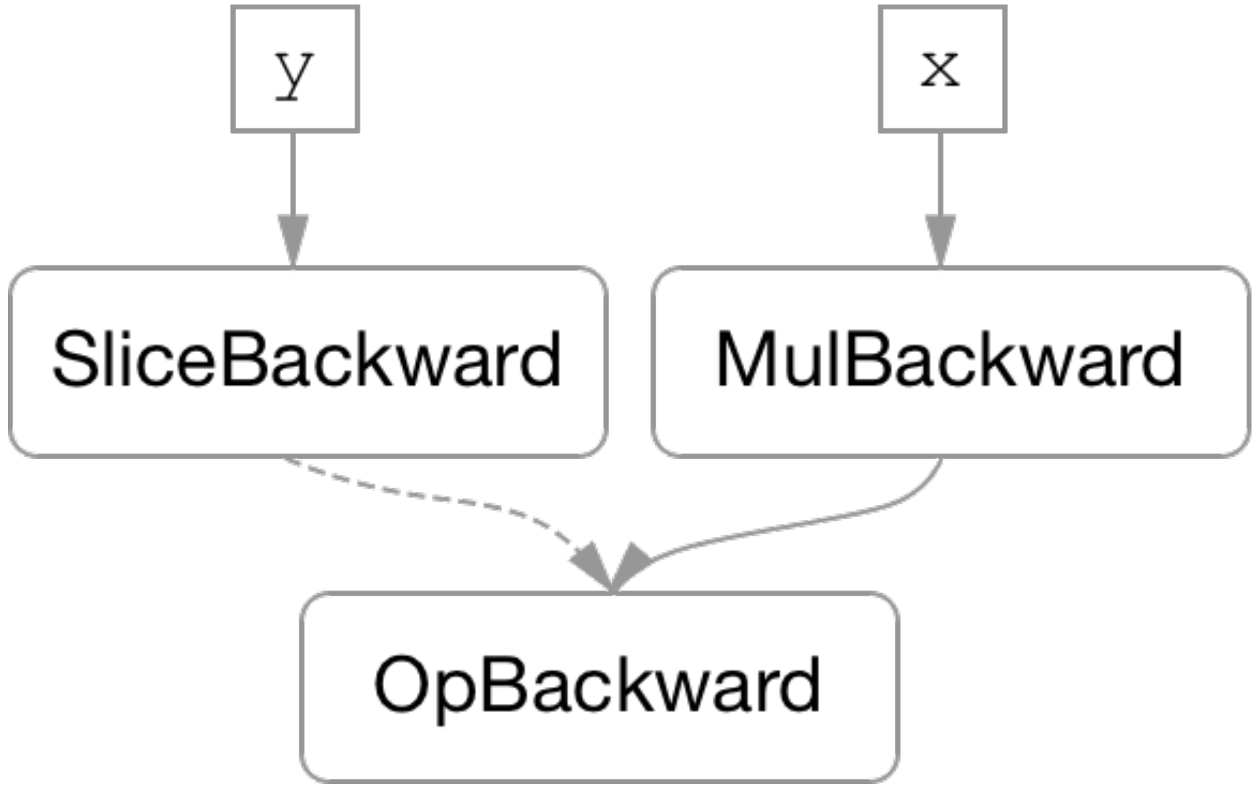
In-place update of the base

```
y = x[:2]
x.mul_(3)
z = y + 2
```



In-place update of the base

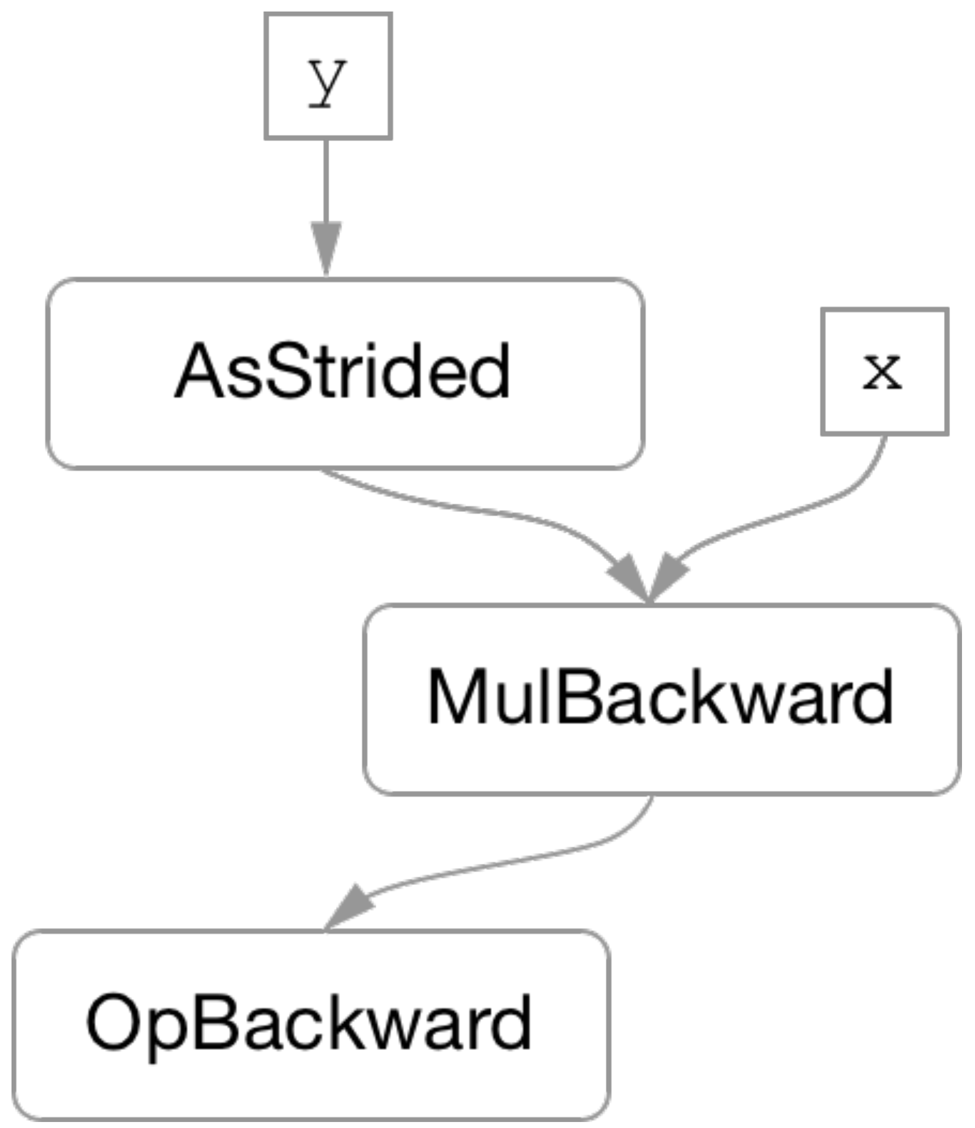
```
y = x[:2]
x.mul_(3)
z = y + 2
```



In-place update of the base

```
y = x[:2]
x.mul_(3)
z = y + 2
```

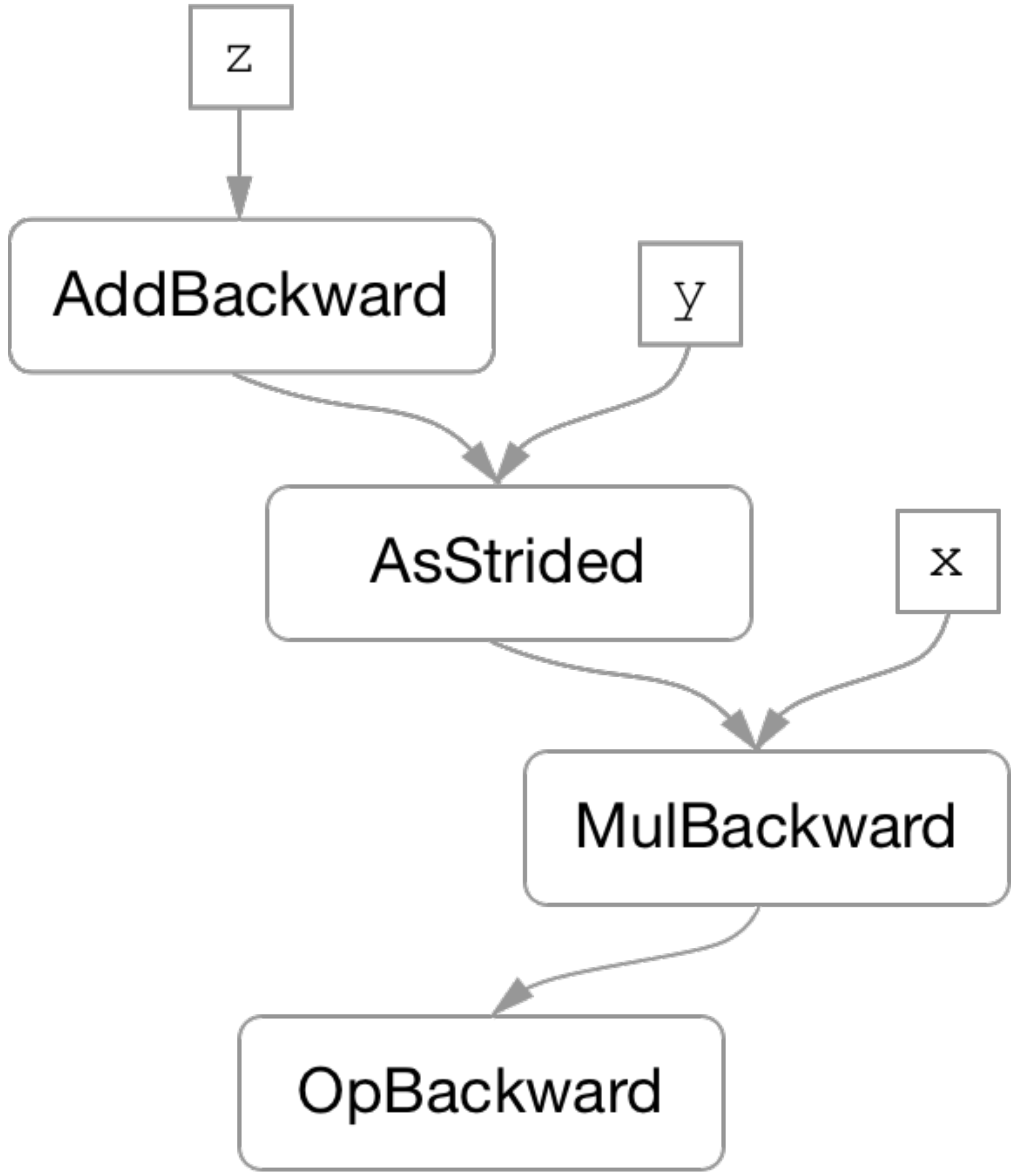
Use the version counter to check if trace pointer is stale.



In-place update of the base

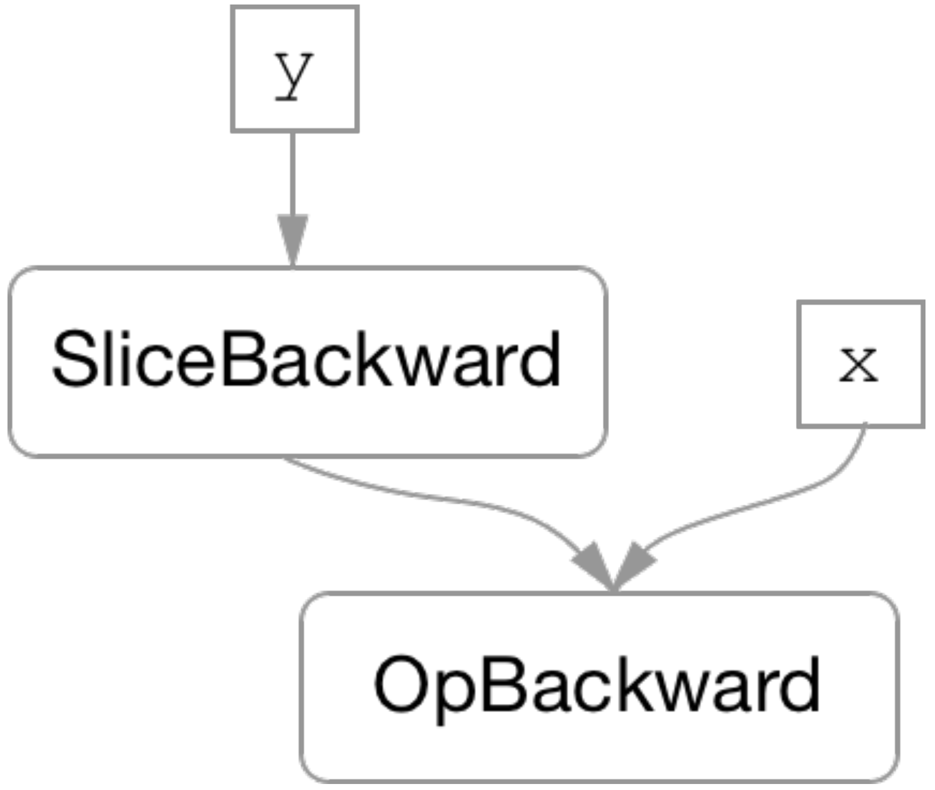
```
y = x[:2]
x.mul_(3)
z = y + 2
```

Use the version counter to check if trace pointer is stale.



In-place update of a view

```
y = x[:2]
y.mul_(3)
```

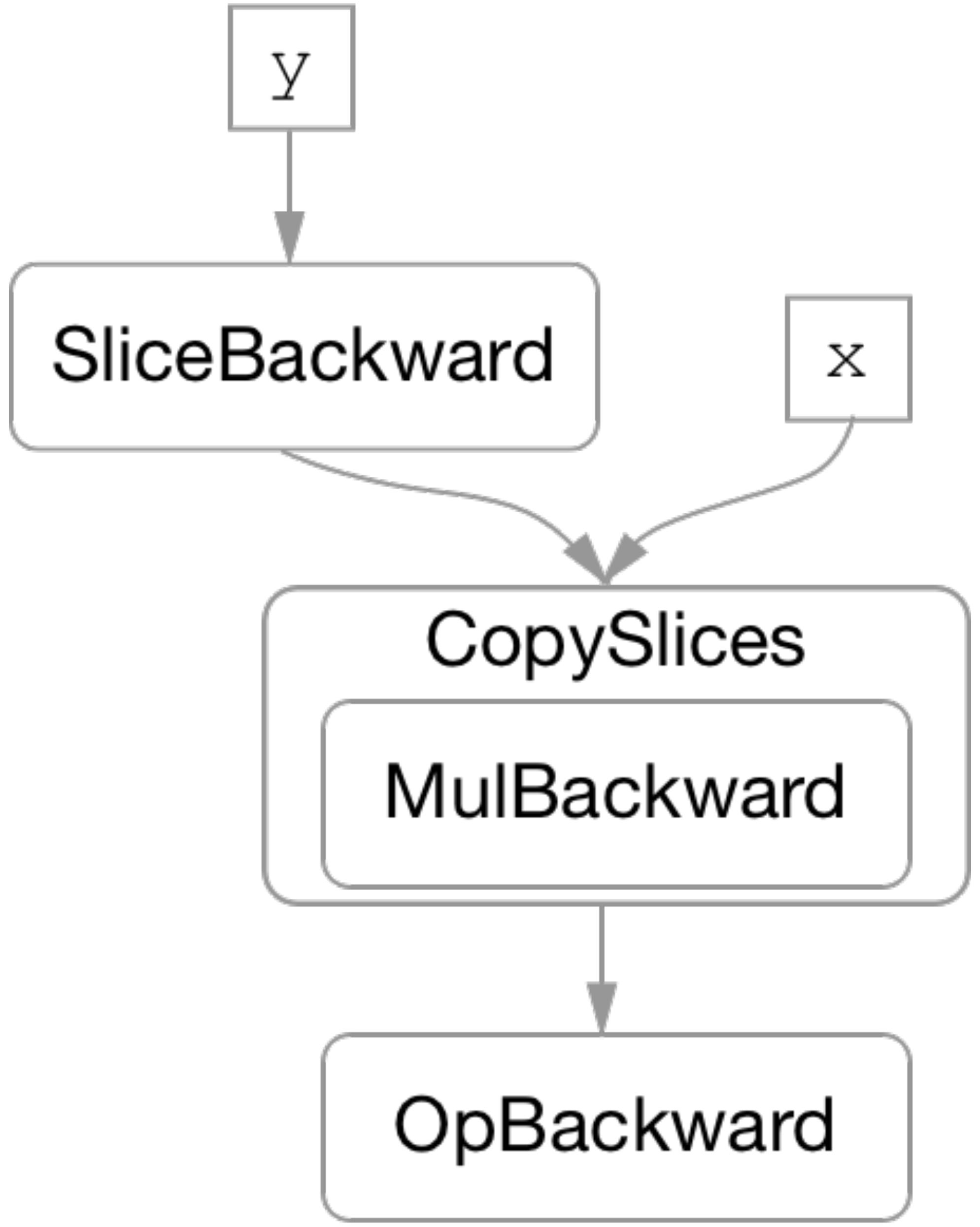


In-place update of a view

```
y = x[:2]
y.mul_(3)
```

with `CopySlices` being:

```
def copy_slices(fn, view_g, base_g, *grads):
    result = torch.Tensor(base_g.sizes,
                          base_g.strides)
    result.copy_(grads[0])

    offset_diff = view_g.offset - base.offset
    grad_slice = result.as_strided(view_g.sizes,
                                   view_g.strides)

    grad_slice.copy_(fn(grad_slice.clone()))
    grad_outputs[0] = result
    return grad_outputs
```



What AD systems for ML research need?

- Tight integration with Python
  - CPython is slow
  - Python is complicated
- Be invisible
  - Ideally an imperative interface
  - Metaprogramming in Python is very unnatural!
- Focus on memory efficiency
  - But don't assume full prior knowledge about the code!
- Simple to reason about performance

Can we build hybrid ST-OO systems for Python?

Summary

- Efficient reverse-mode AD
- In-place support
- Eager evaluation
- Pure C++ implementation
- Extensions (both in Python and C++)
- Quickly growing user base

Thank you!