Automatic Differentiation and

Zeming Lin

Facebook AI Research

Automatic Differentiation

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Derivatives

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Derivatives

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Finite Differences

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Derivatives

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Finite Differences

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

- T(f) is how long it takes to execute f
- d is the number of inputs to f

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Derivatives

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Finite Differences

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

- T(f) is how long it takes to execute f
- d is the number of inputs to f
- Complexity?

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Derivatives

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Finite Differences

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

- T(f) is how long it takes to execute f
- d is the number of inputs to f
- Complexity? 2*d*T(f)

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Finite Differences

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

- No assumptions on f!

Automatic Differentiation

Given an arbitrary function, can we efficiently find its derivative?

Finite Differences

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

- No assumptions on f!

Estimates for x^2

x	Finite difference error estimate
0	1e-4
99	1e-4
9999	-3e-5
999999	-1.66
99999999	20002

Automatic Differentiation

Derivative Rules:

$$\frac{d}{dx}(ax+b) = a$$

$$\frac{d}{dx}x^n = nx^n - 1$$

$$\frac{d}{dx}sin(x) = cos(x)$$

$$\frac{d}{dx}e^x = e^x$$

$$\frac{d}{dx}ln(x) = x^-1$$

Automatic Differentiation

Derivative Rules:

$$\frac{d}{dx}(ax+b) = a$$

$$\frac{d}{dx}x^n = nx^n - 1$$

$$\frac{d}{dx}sin(x) = cos(x)$$

$$\frac{d}{dx}e^x = e^x$$

$$\frac{d}{dx}ln(x) = x^-1$$

```
296  - name: erfc(Tensor self) -> Tensor
297    self: -2.0 / sqrt(M_PI) * exp(-(self.pow(2))) * grad
298
299  - name: erfinv(Tensor self) -> Tensor
300    self: 0.5 * sqrt(M_PI) * exp(self.erfinv().pow(2)) * grad
301
302  - name: exp(Tensor self) -> Tensor
303    self: grad * result
304
305  - name: expm1(Tensor self) -> Tensor
306    self: grad * (result + 1)
307
308  - name: expand(Tensor(a) self, int[] size, *, bool implicit=False) -> Tensor(a)
309    self: at::sum_to(grad, self.sizes())
310
311  - name: exponential_(Tensor(a!) self, float lambd=1, *, Generator? generator=None) -> Tensor(a!)
312    self: zeros_like(grad)
```

Automatic Differentiation

Derivative Rules:

$$\frac{d}{dx}(ax+b) = a$$

$$\frac{d}{dx}x^n = nx^n - 1$$

$$\frac{d}{dx}sin(x) = cos(x)$$

$$\frac{d}{dx}e^x = e^x$$

$$\frac{d}{dx}ln(x) = x^-1$$

Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

Sum Rule: $\frac{d}{dx}[f(x)+g(x)] = f'(x)+g'(x)$

Product Rule: $\frac{d}{dx}f(x)g(x) = f'(x)g(x)+f(x)g'(x)$

...

Automatic Differentiation

Derivative Rules:

$$\frac{d}{dx}(ax+b) = a$$

$$\frac{d}{dx}x^n = nx^n - 1$$

$$\frac{d}{dx}sin(x) = cos(x)$$

$$\frac{d}{dx}e^x = e^x$$

$$\frac{d}{dx}ln(x) = x^-1$$

Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

Sum Rule: $\frac{d}{dx}[f(x)+g(x)] = f'(x)+g'(x)$

Product Rule: $\frac{d}{dx}f(x)g(x) = f'(x)g(x)+f(x)g'(x)$

...

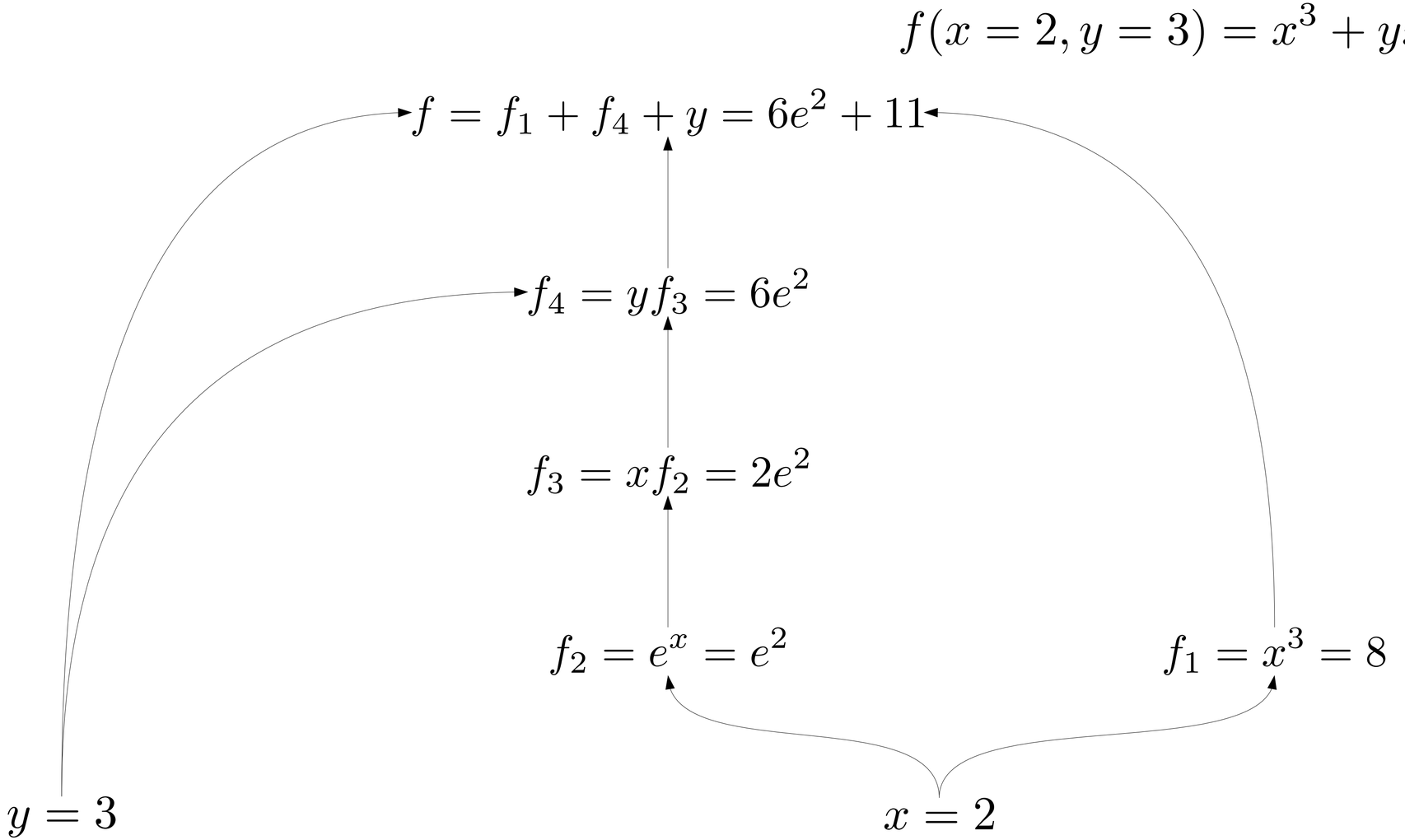
Let’s compute the derivative analytically instead!

Automatic Differentiation

Two types of AD:

- Forward Mode AD
- Reverse Mode AD

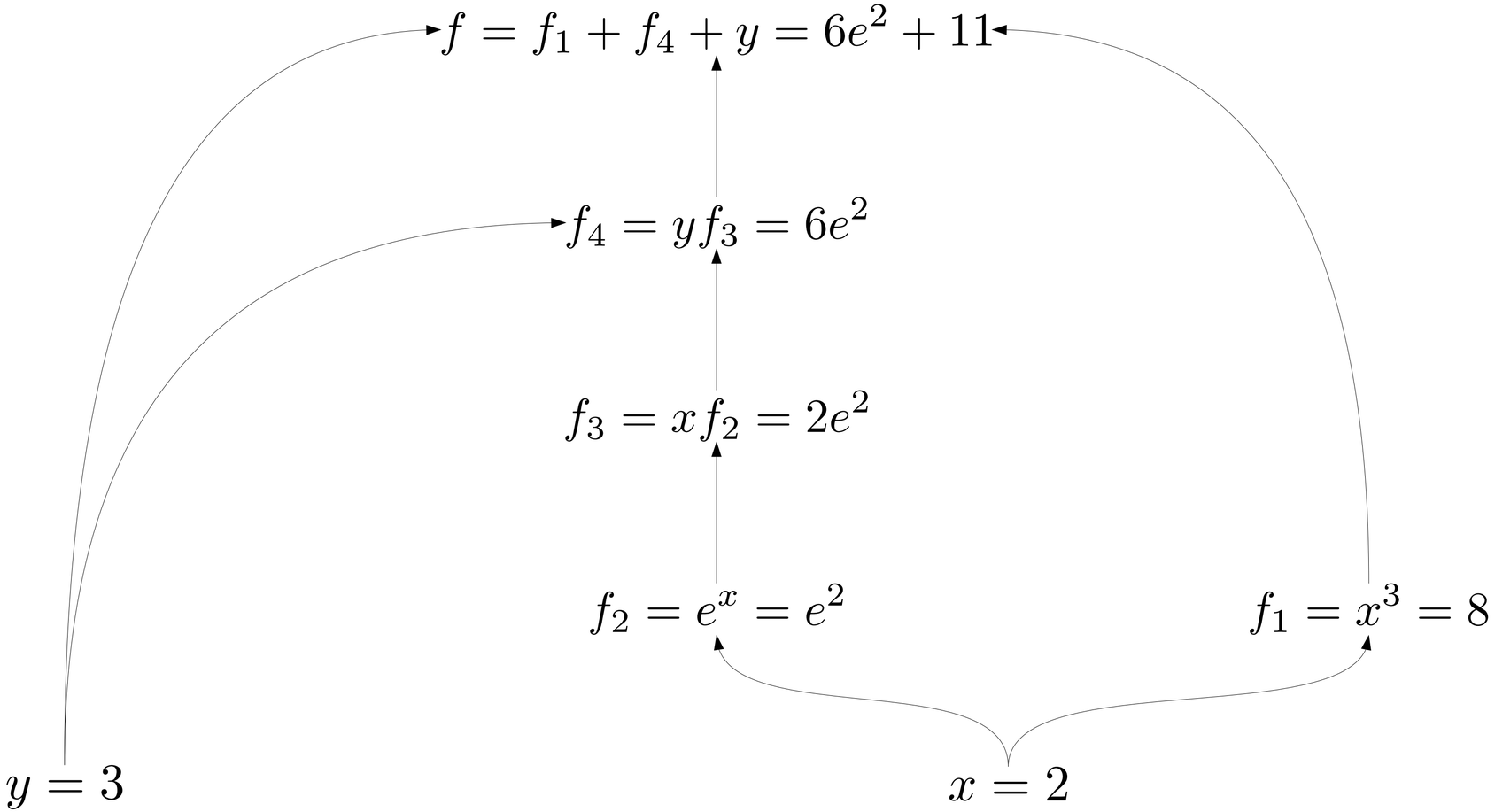
Computational Graph



Computational Graph

Forward Mode AD

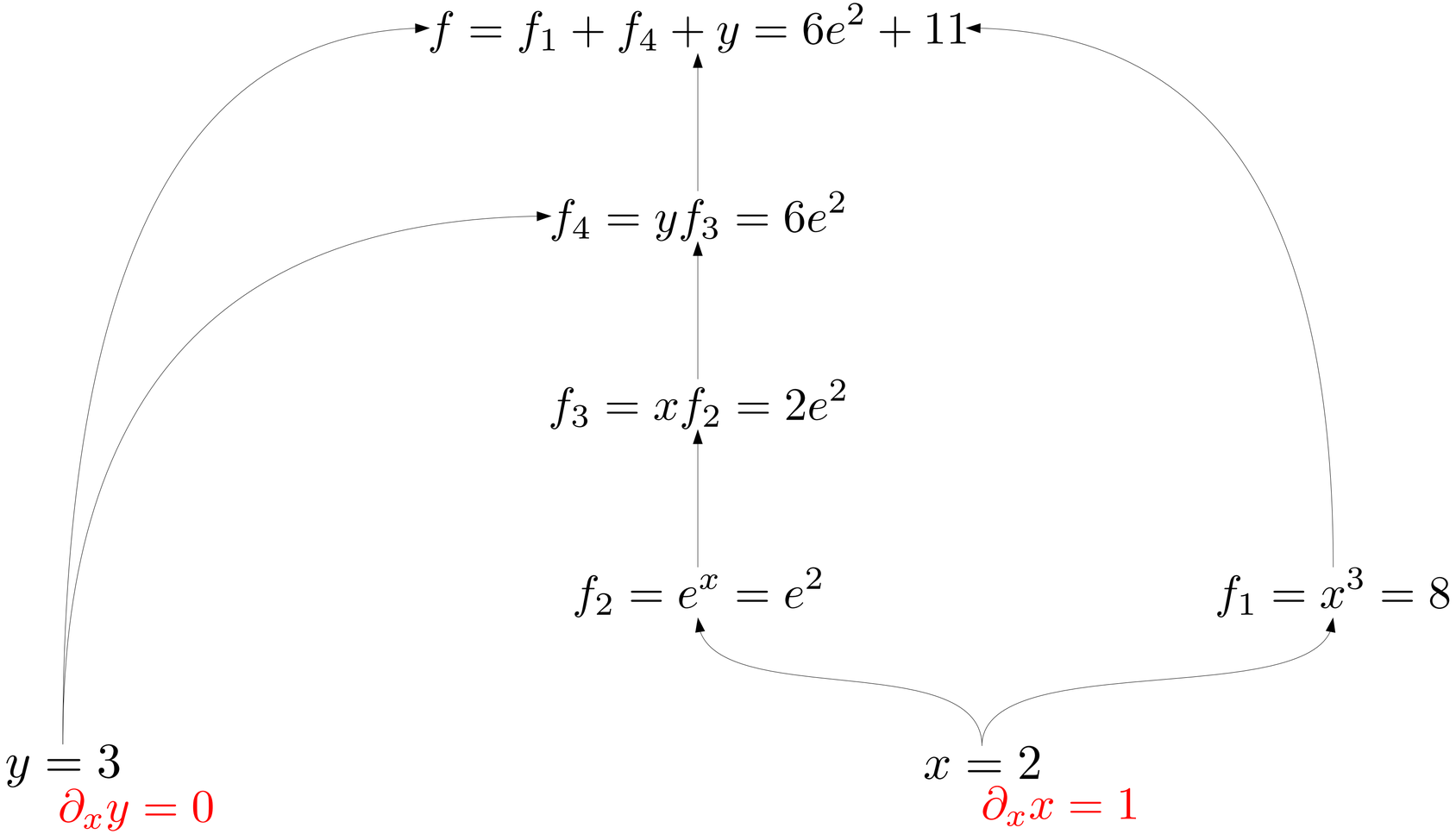
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Forward Mode AD

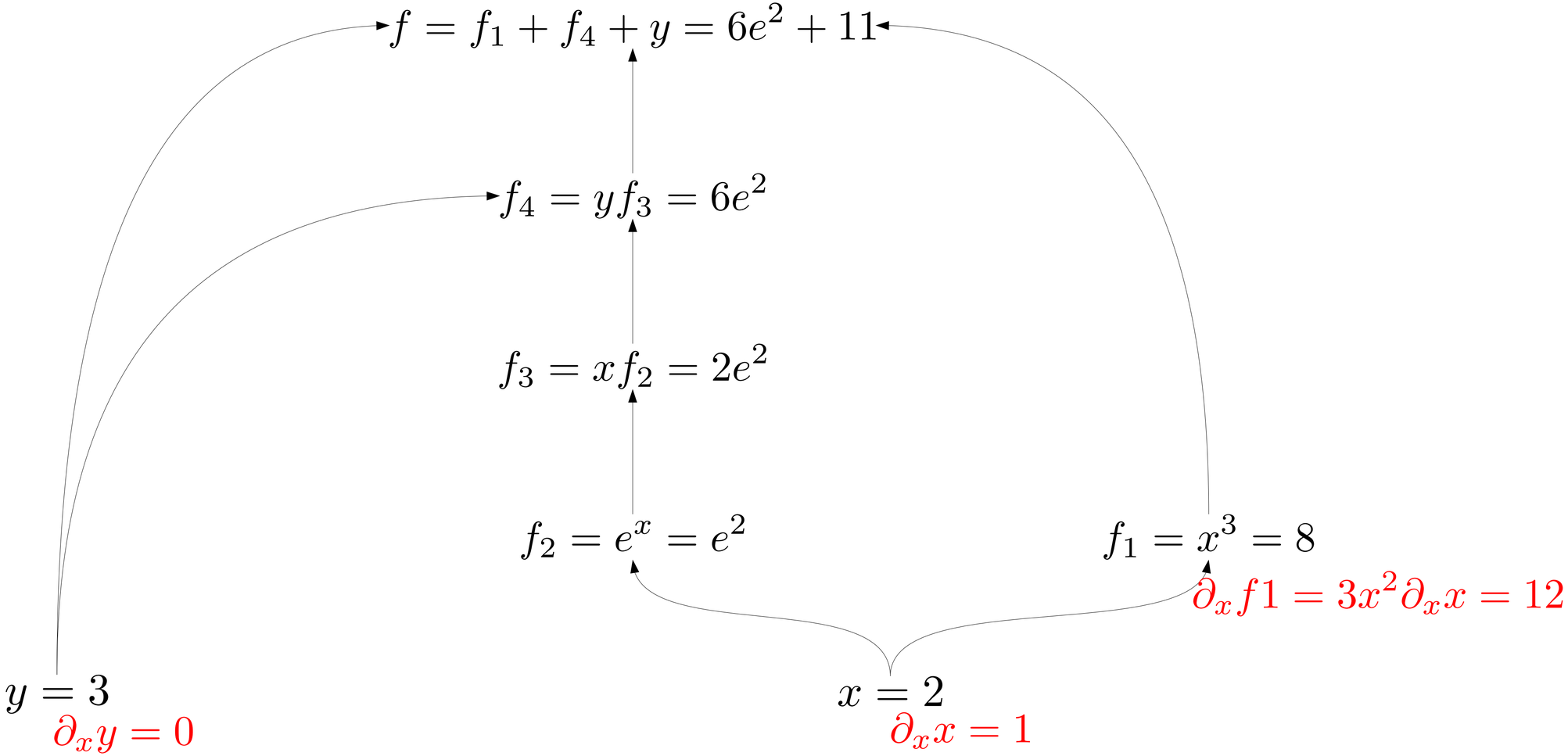
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Forward Mode AD

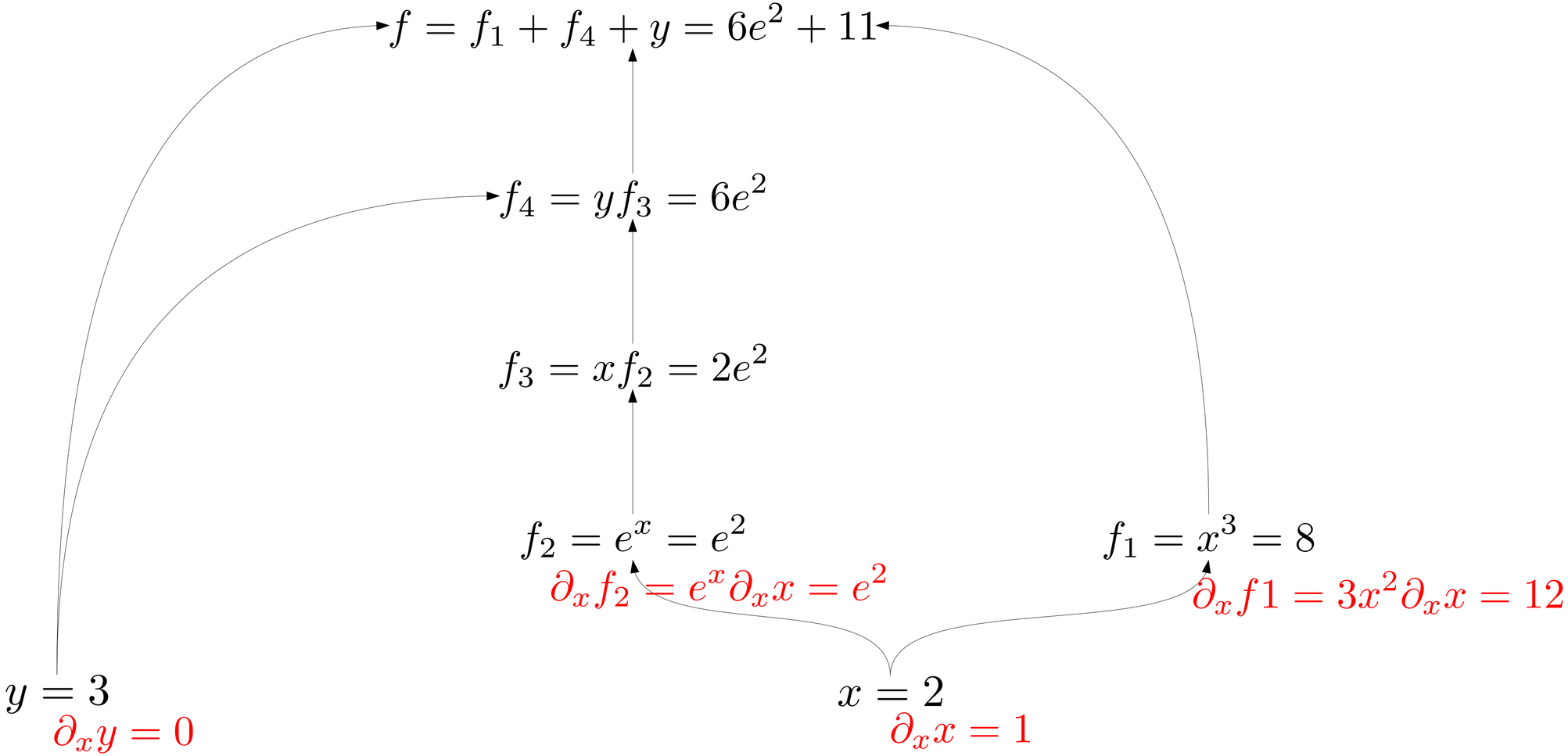
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Forward Mode AD

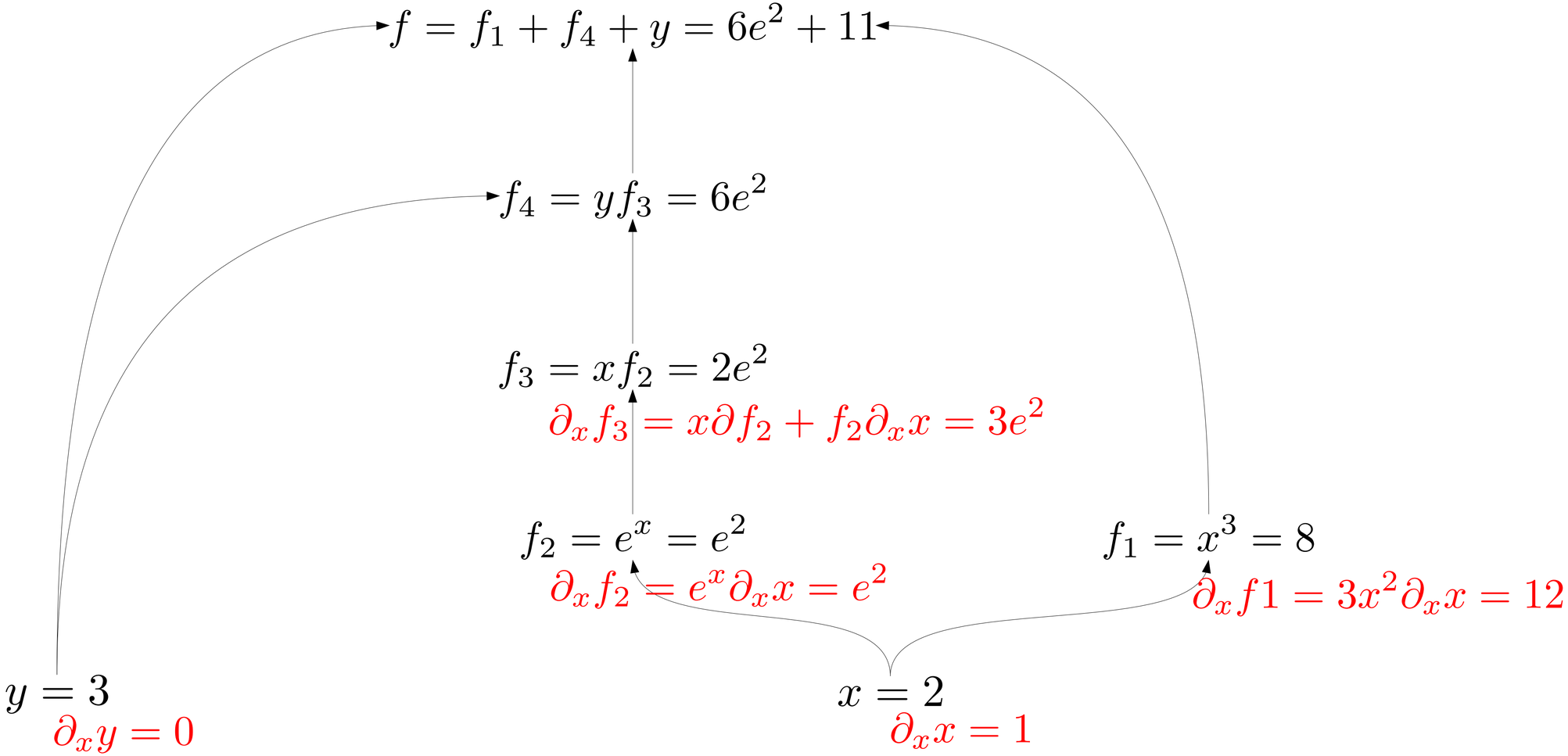
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Forward Mode AD

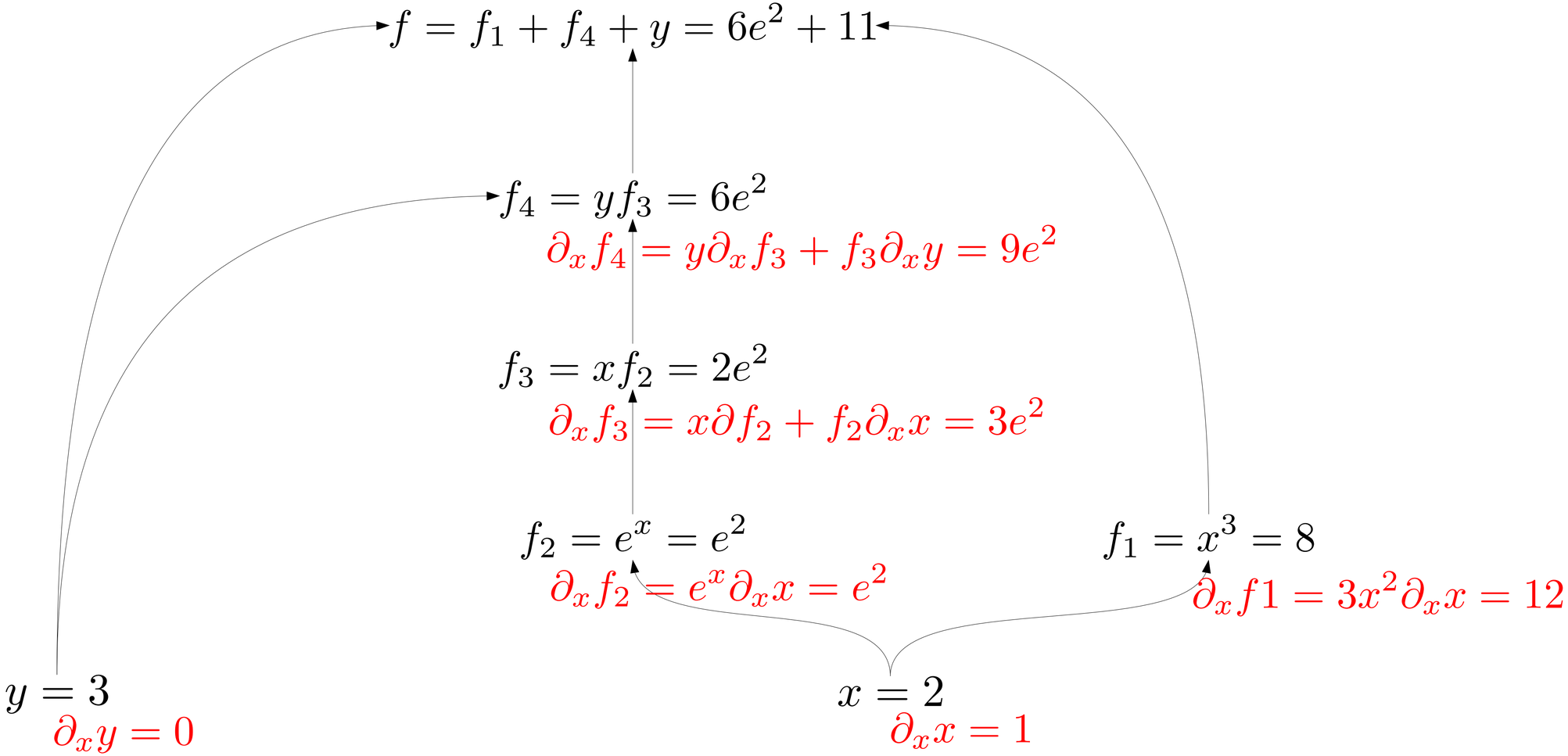
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Forward Mode AD

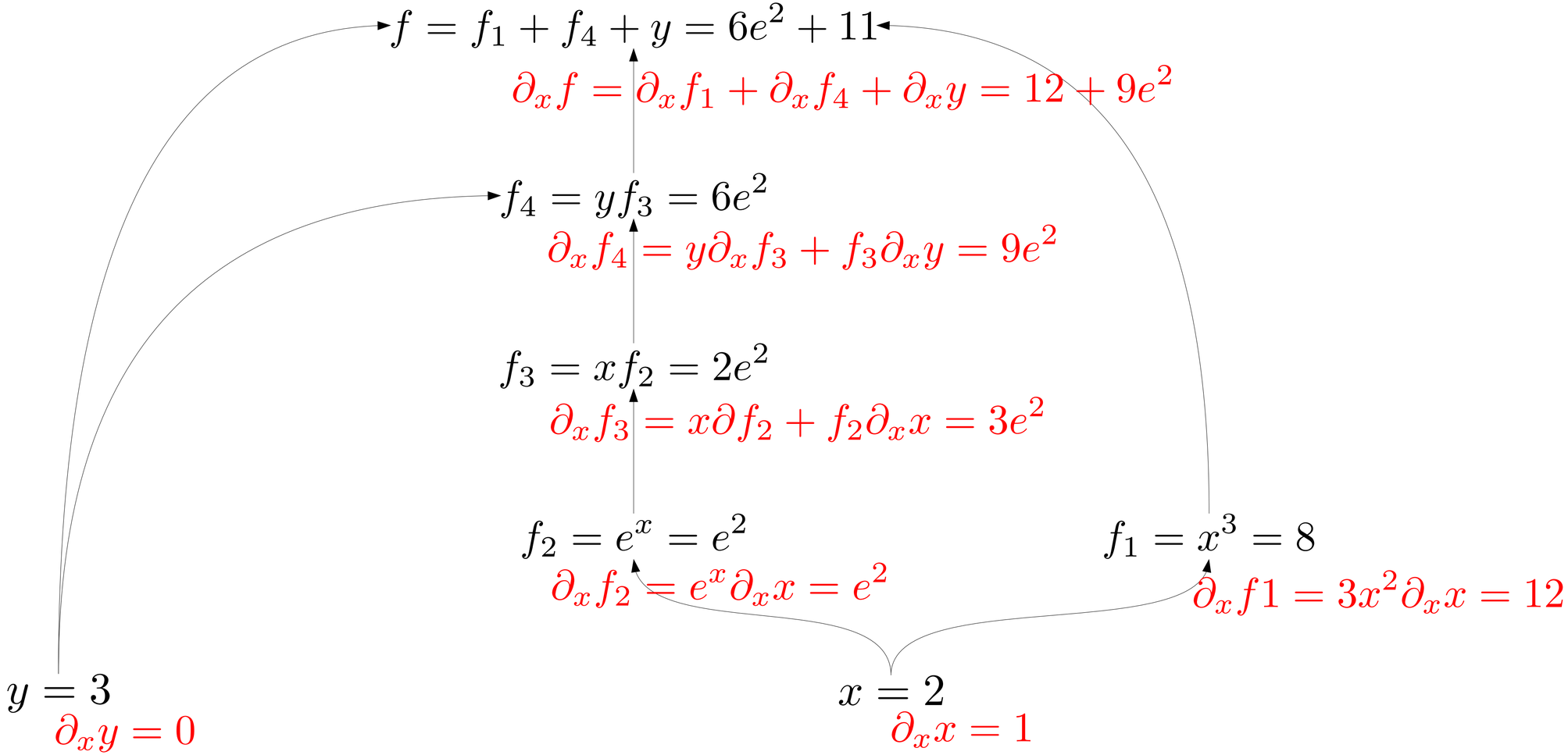
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Forward Mode AD

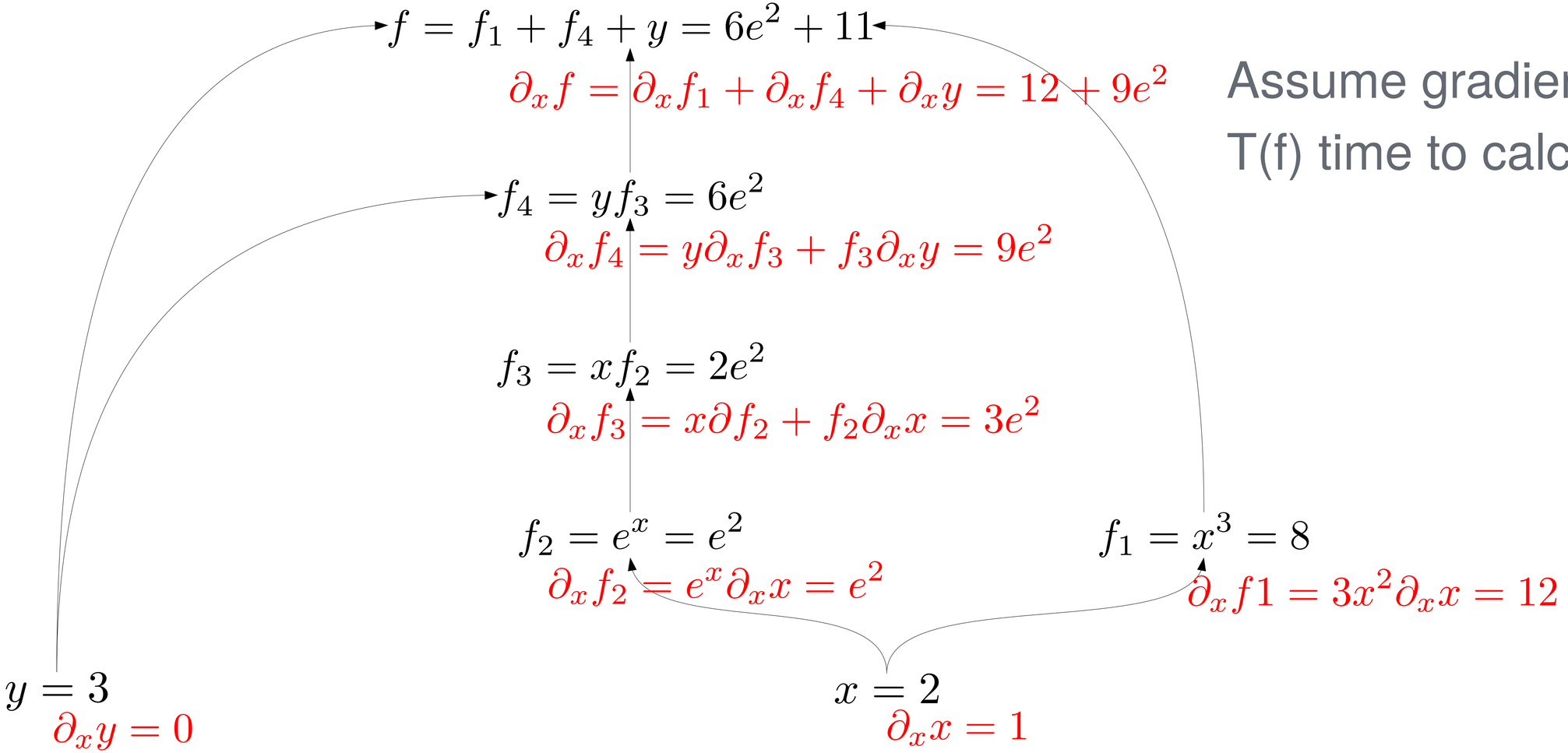
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Forward Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$

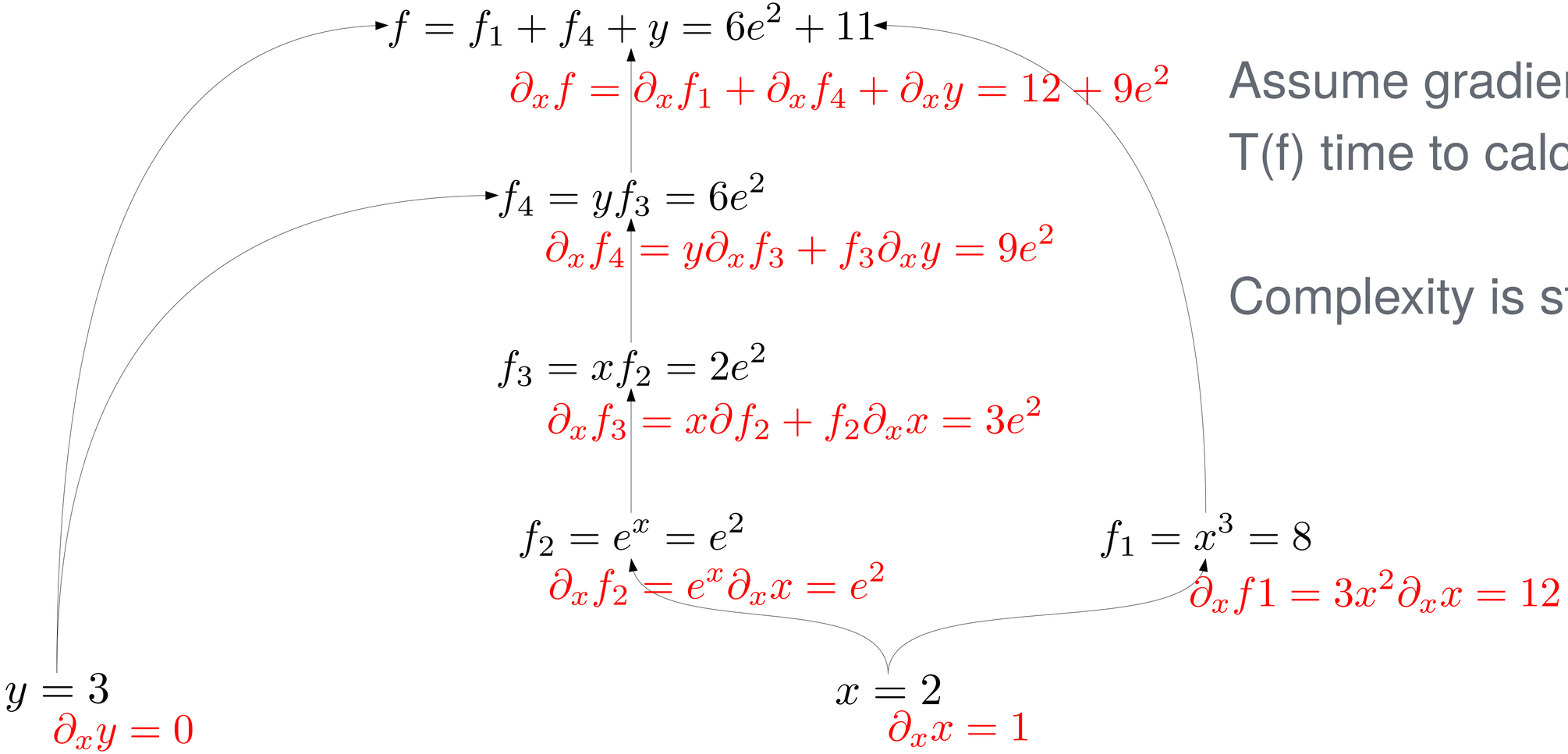


Assume gradients also take T(f) time to calculate

Computational Graph

Forward Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$



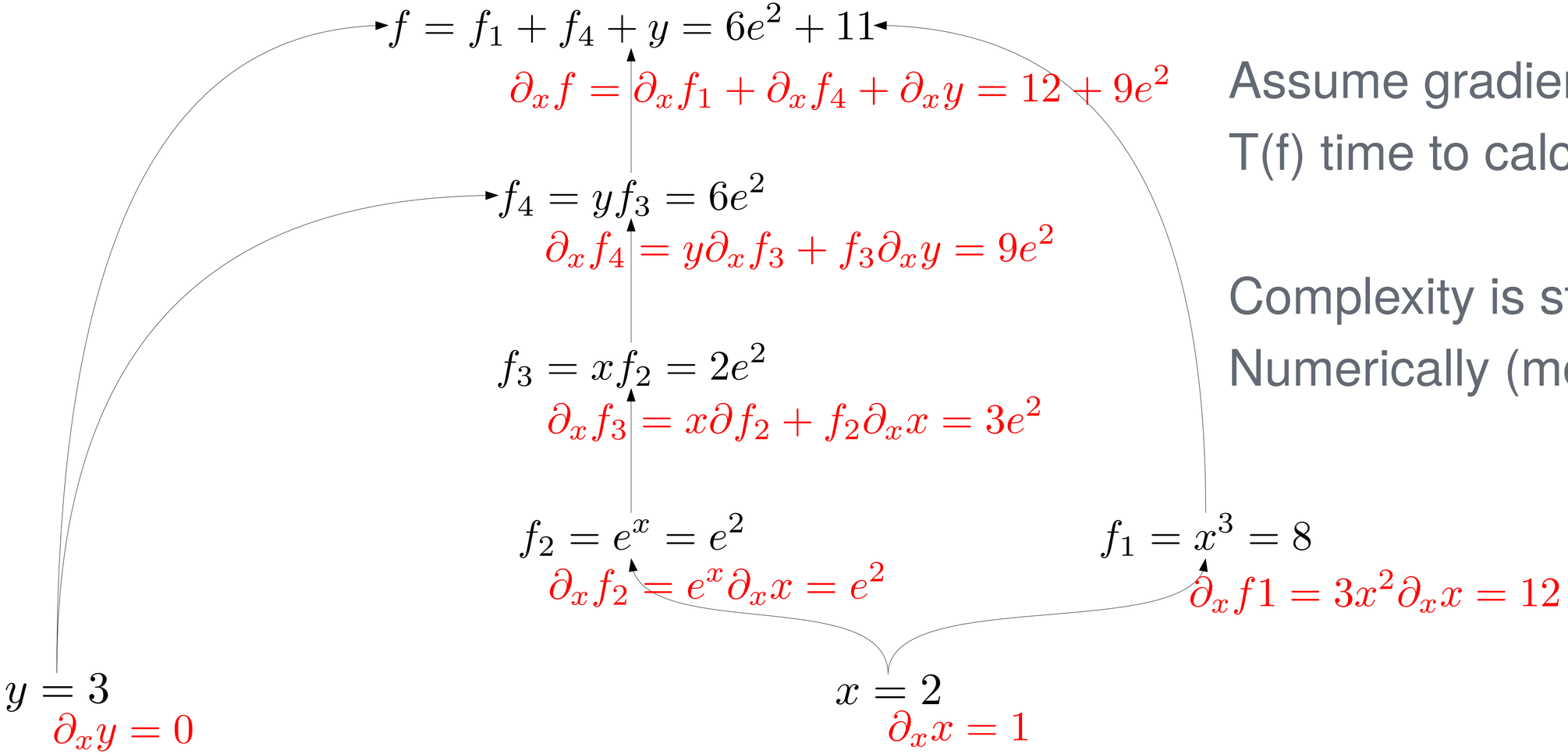
Assume gradients also take T(f) time to calculate

Complexity is still 2*d*T(f)

Computational Graph

Forward Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$



Assume gradients also take T(f) time to calculate

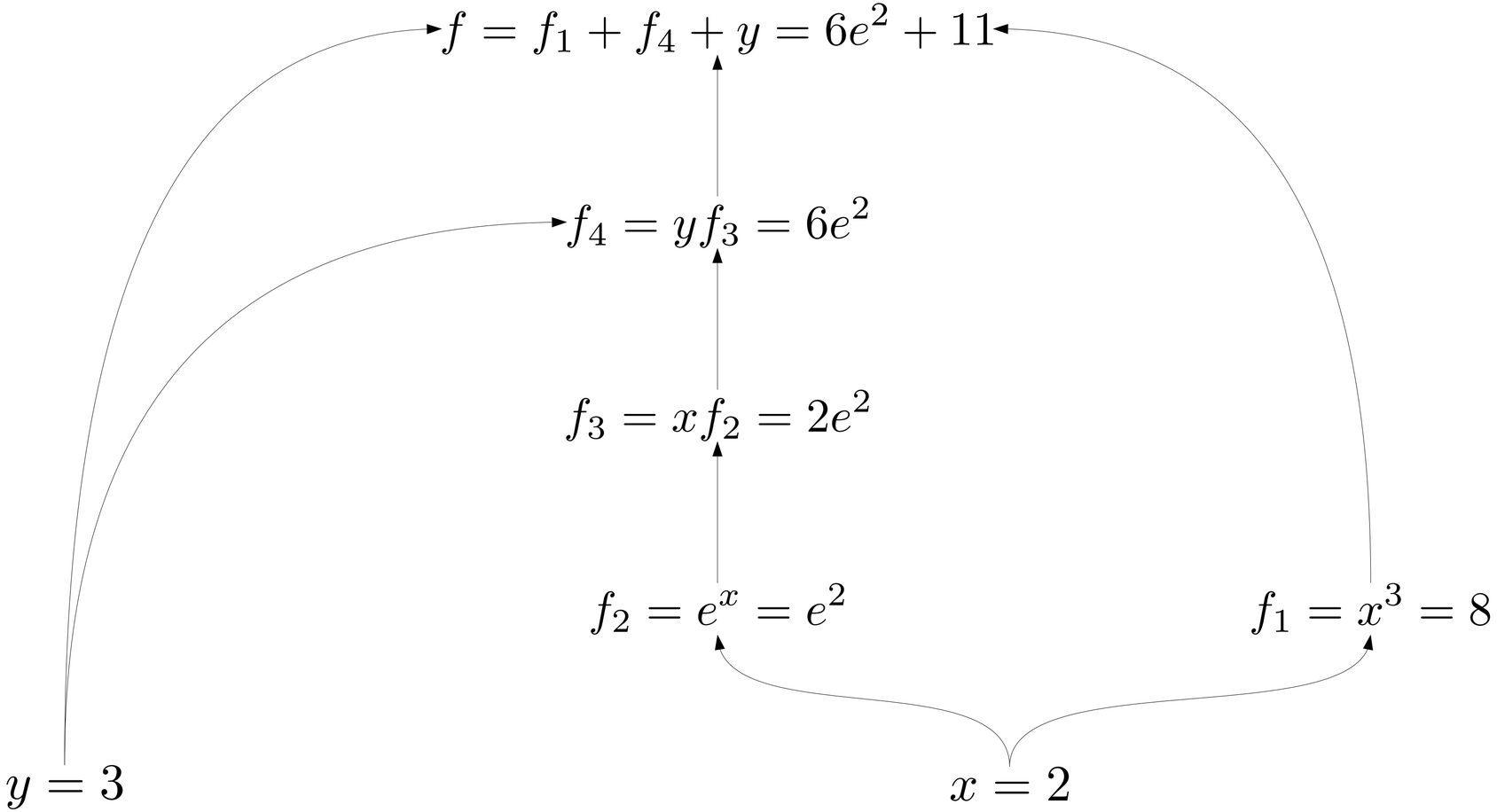
Complexity is still 2*d*T(f)

Numerically (more) stable!

Computational Graph

Reverse Mode AD

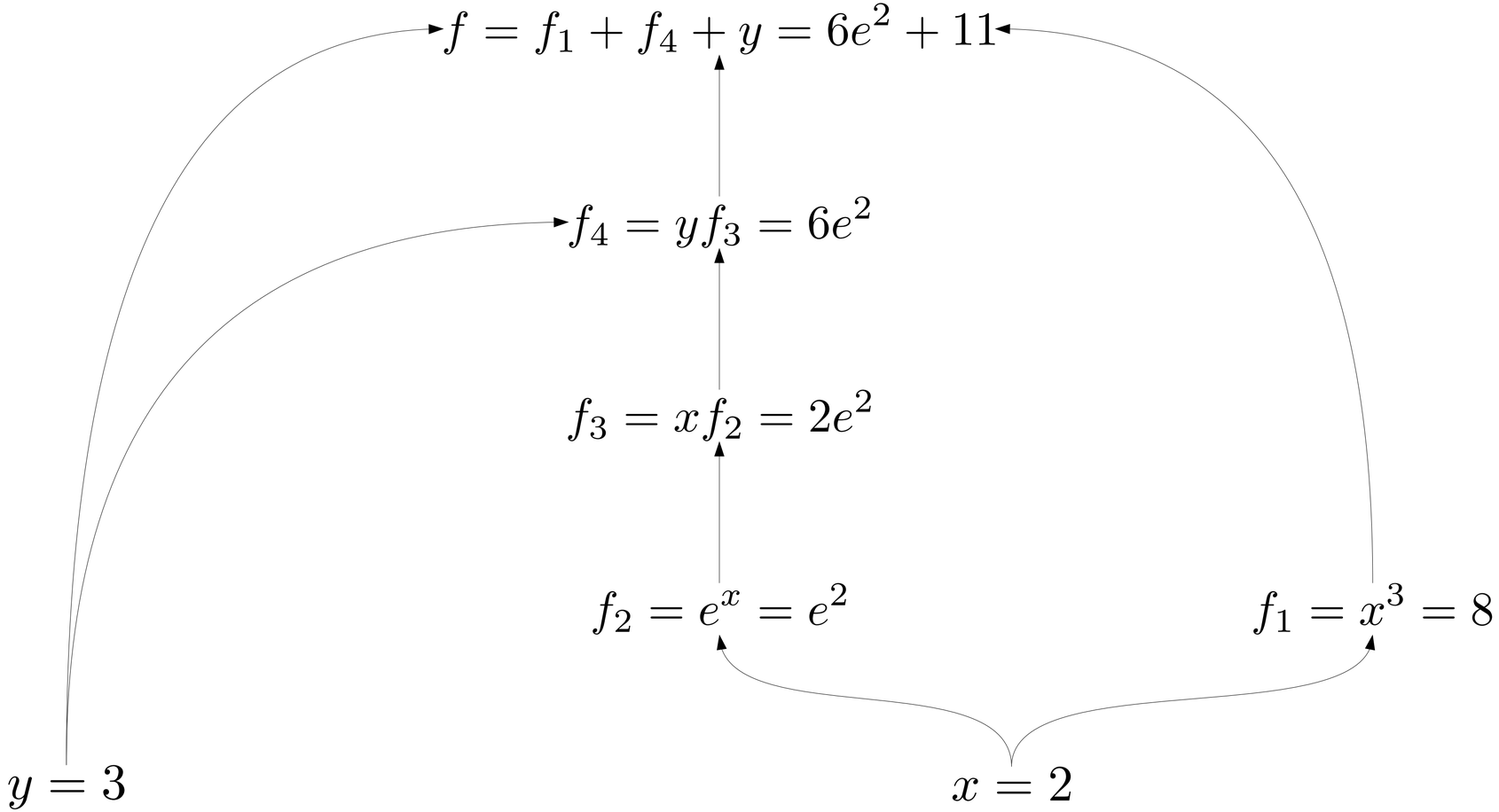
$$f(x=2, y=3) = x^3 + yxe^x + y$$

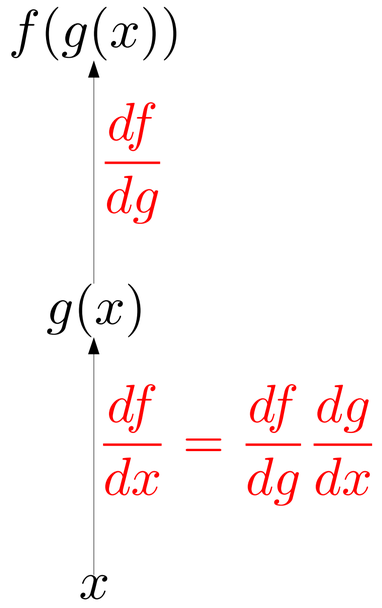


Computational Graph

Reverse Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$

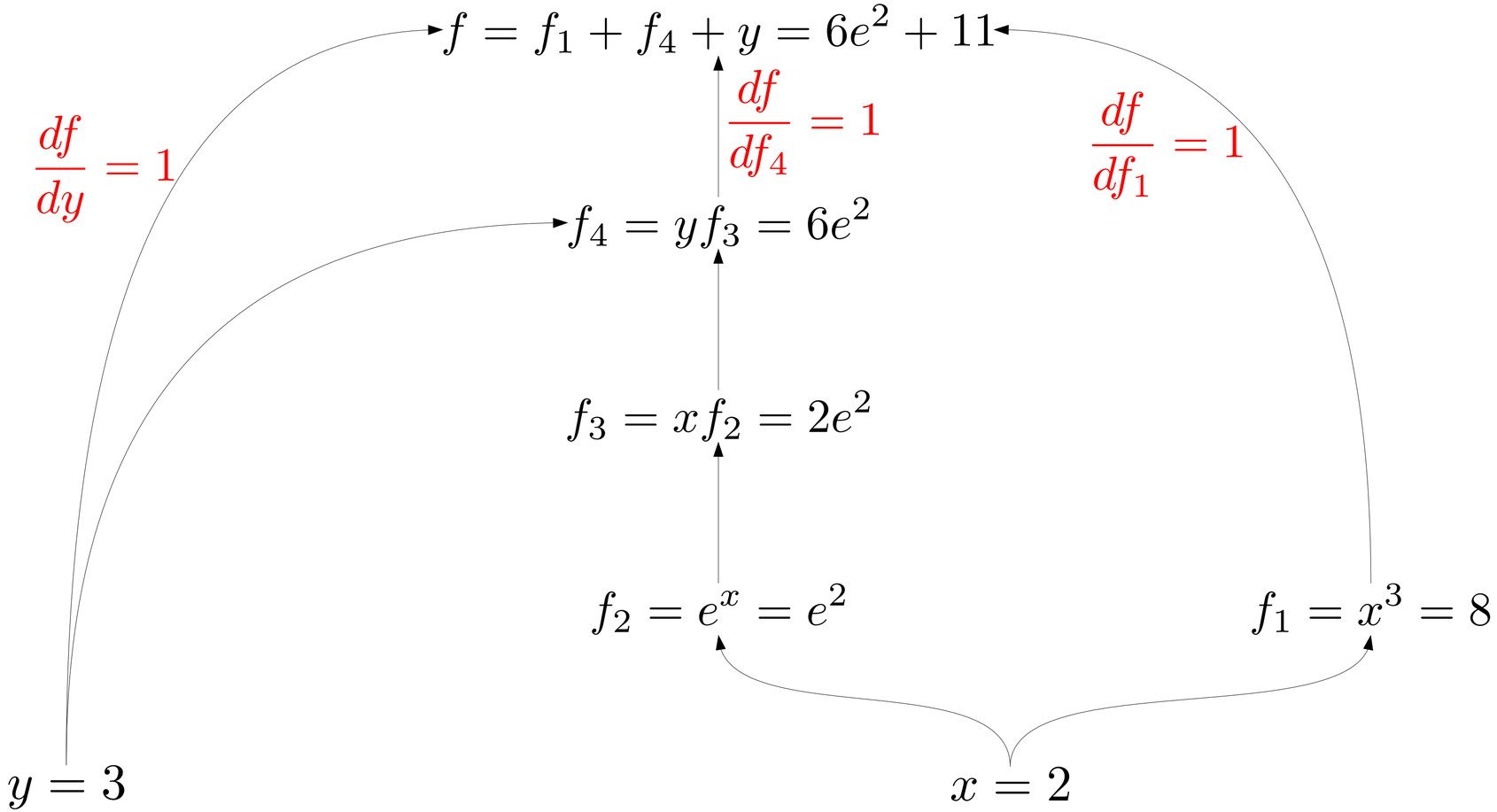


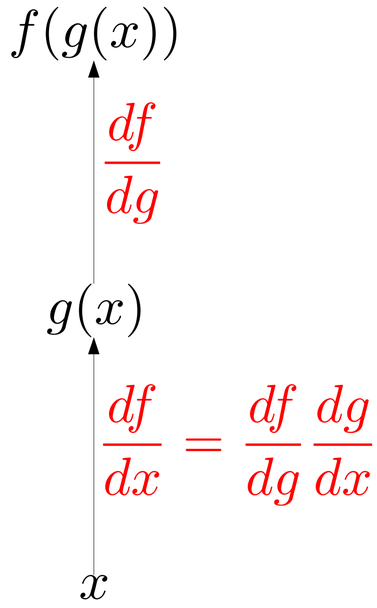


Computational Graph

Reverse Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$

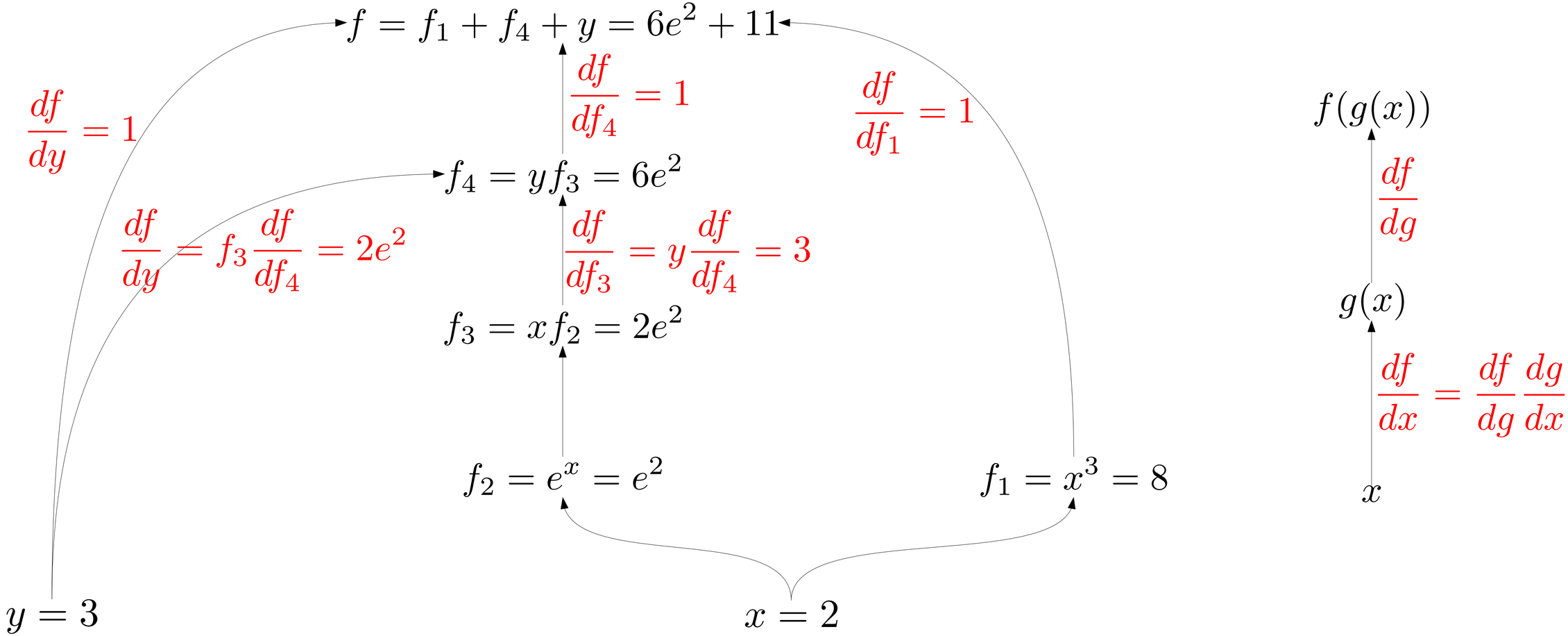




Computational Graph

Reverse Mode AD

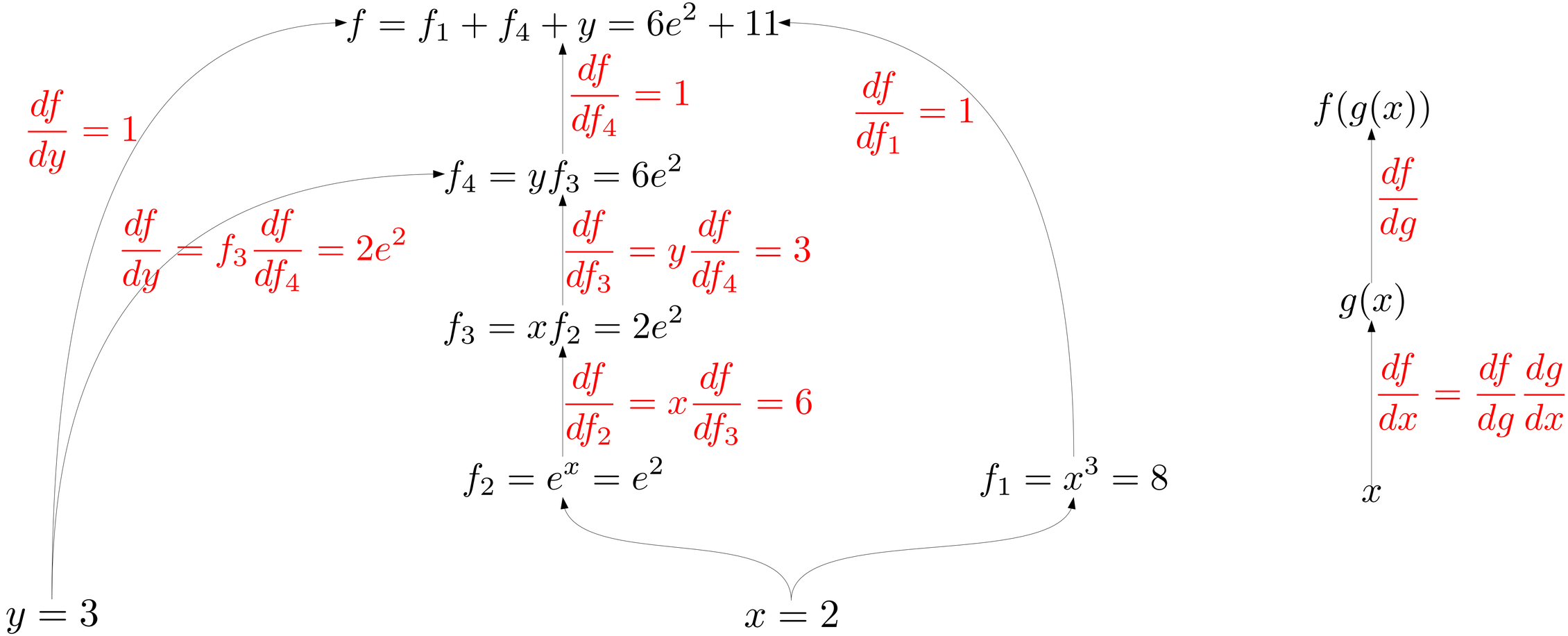
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Reverse Mode AD

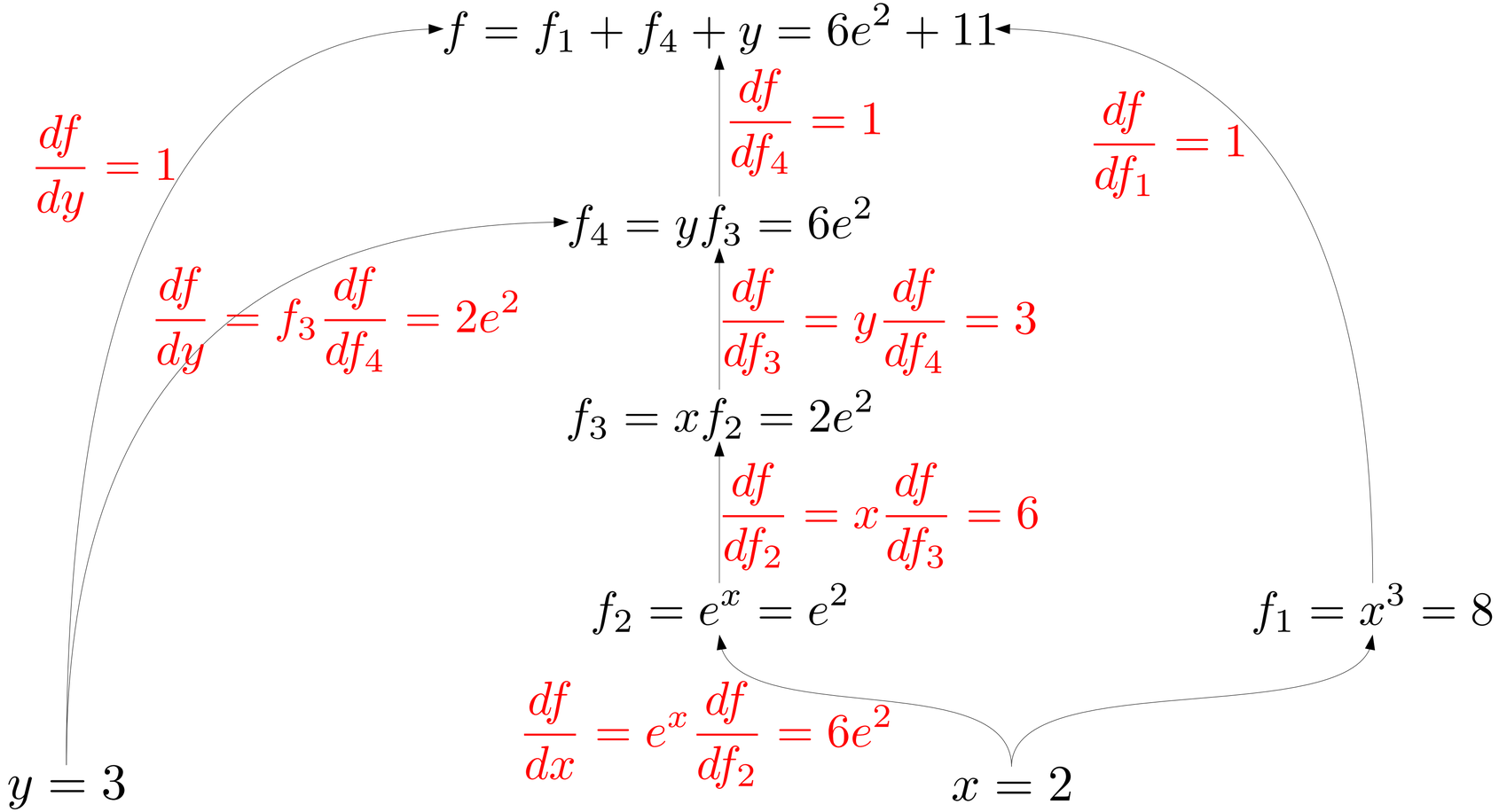
$$f(x=2, y=3) = x^3 + yxe^x + y$$

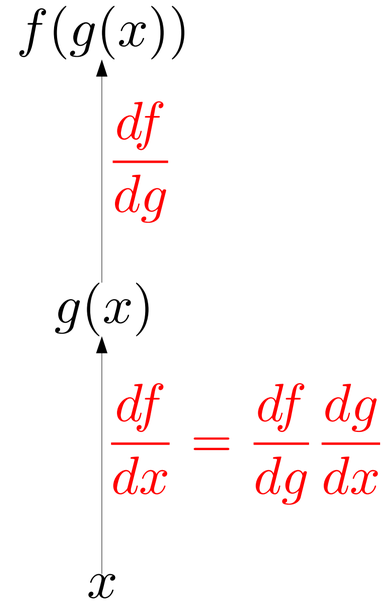


Computational Graph

Reverse Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$

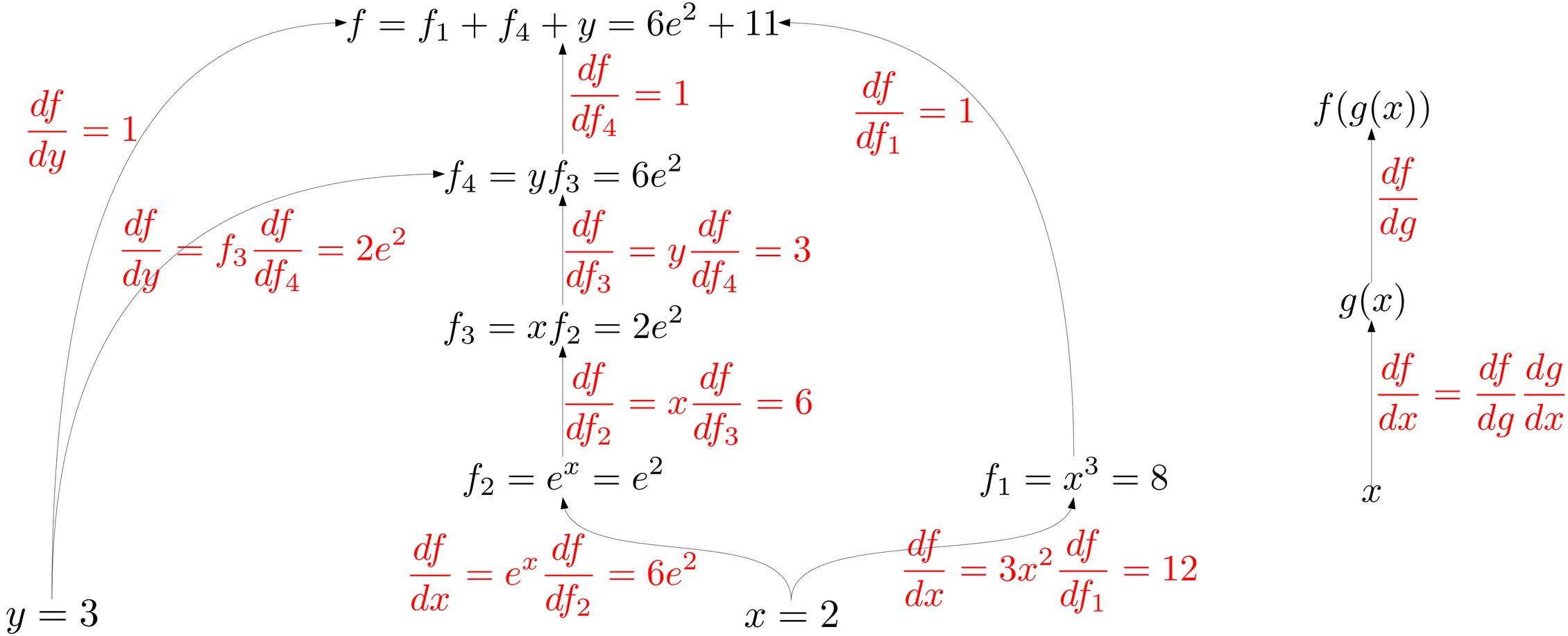




Computational Graph

Reverse Mode AD

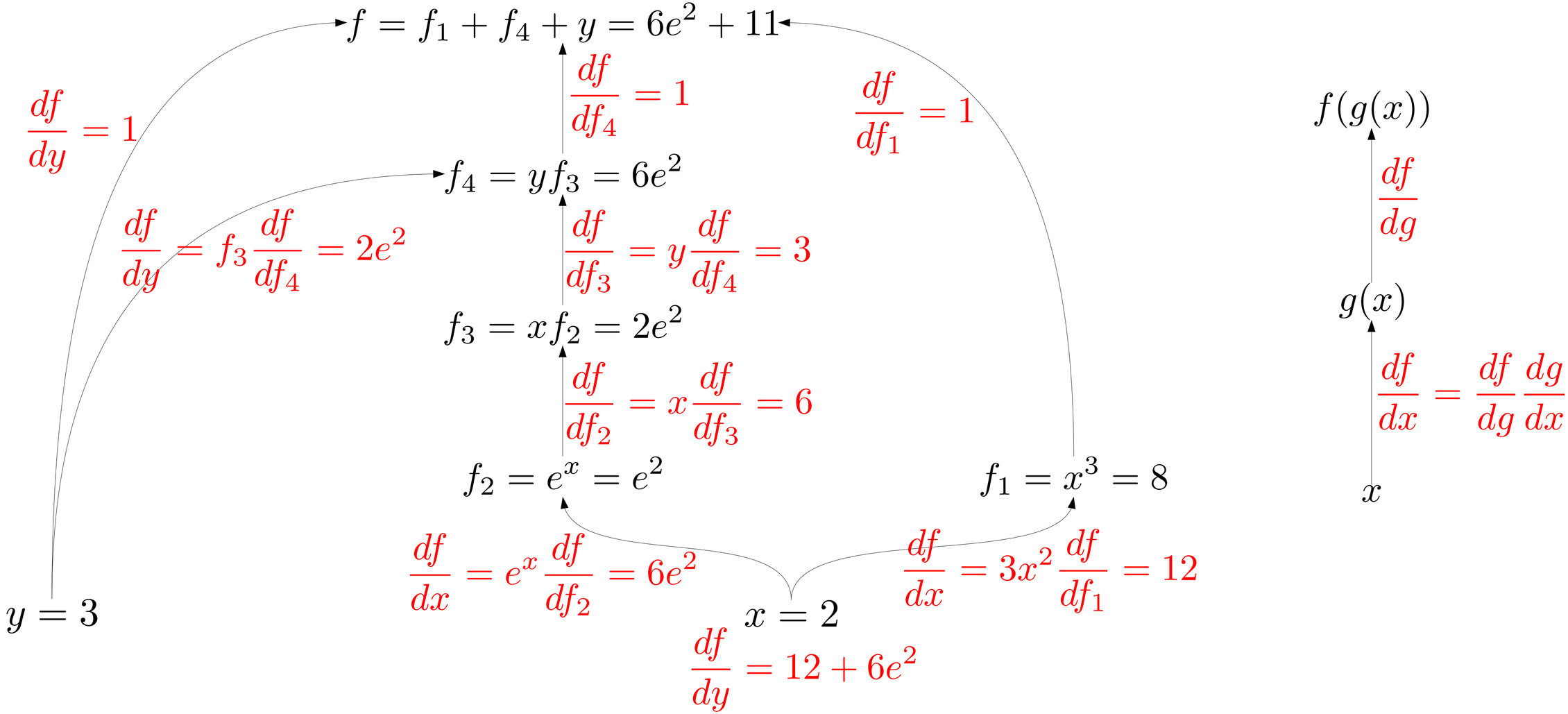
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Reverse Mode AD

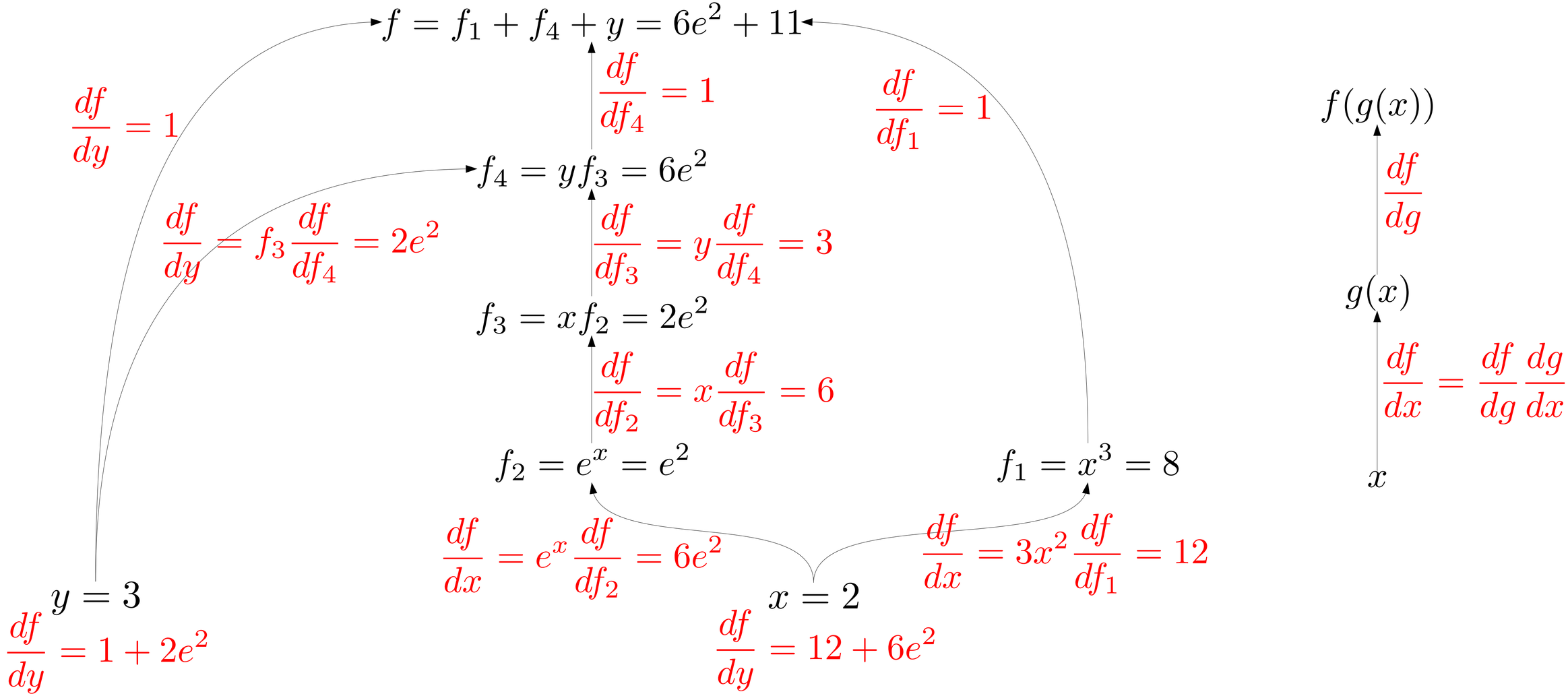
$$f(x=2, y=3) = x^3 + yxe^x + y$$



Computational Graph

Reverse Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$

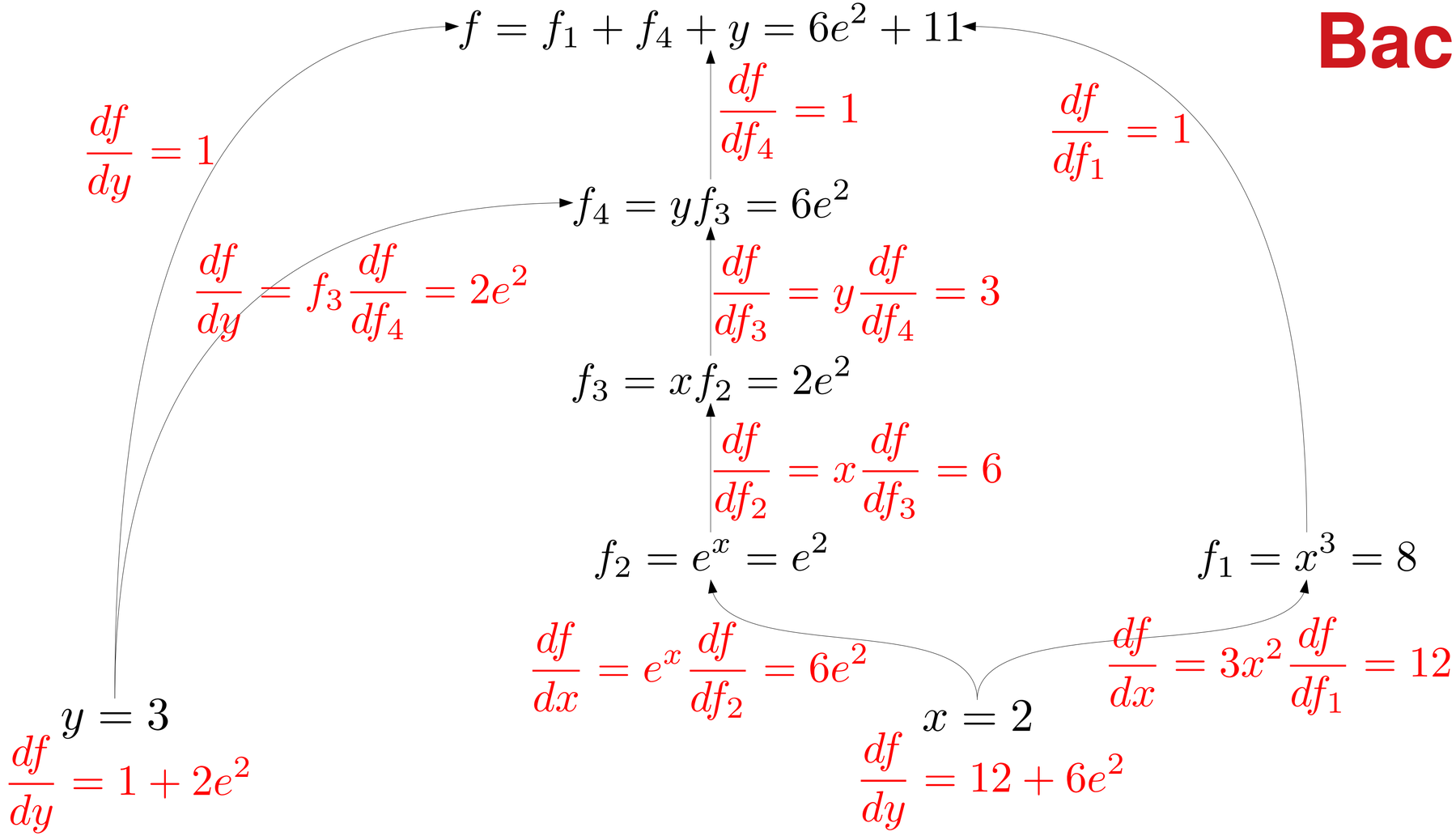


Computational Graph

Reverse Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$

Backpropagation!

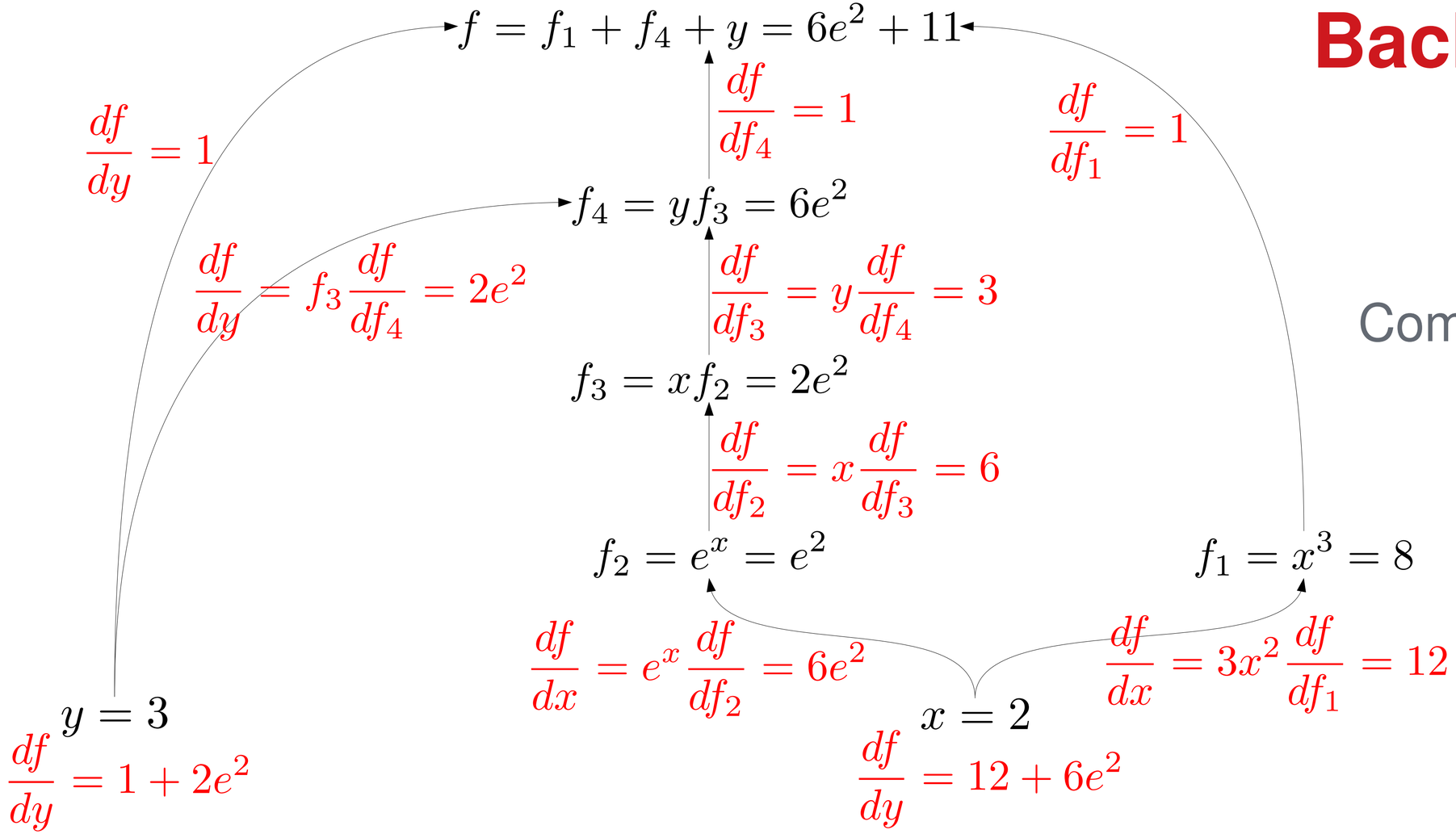


Computational Graph

Reverse Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$

Backpropagation!

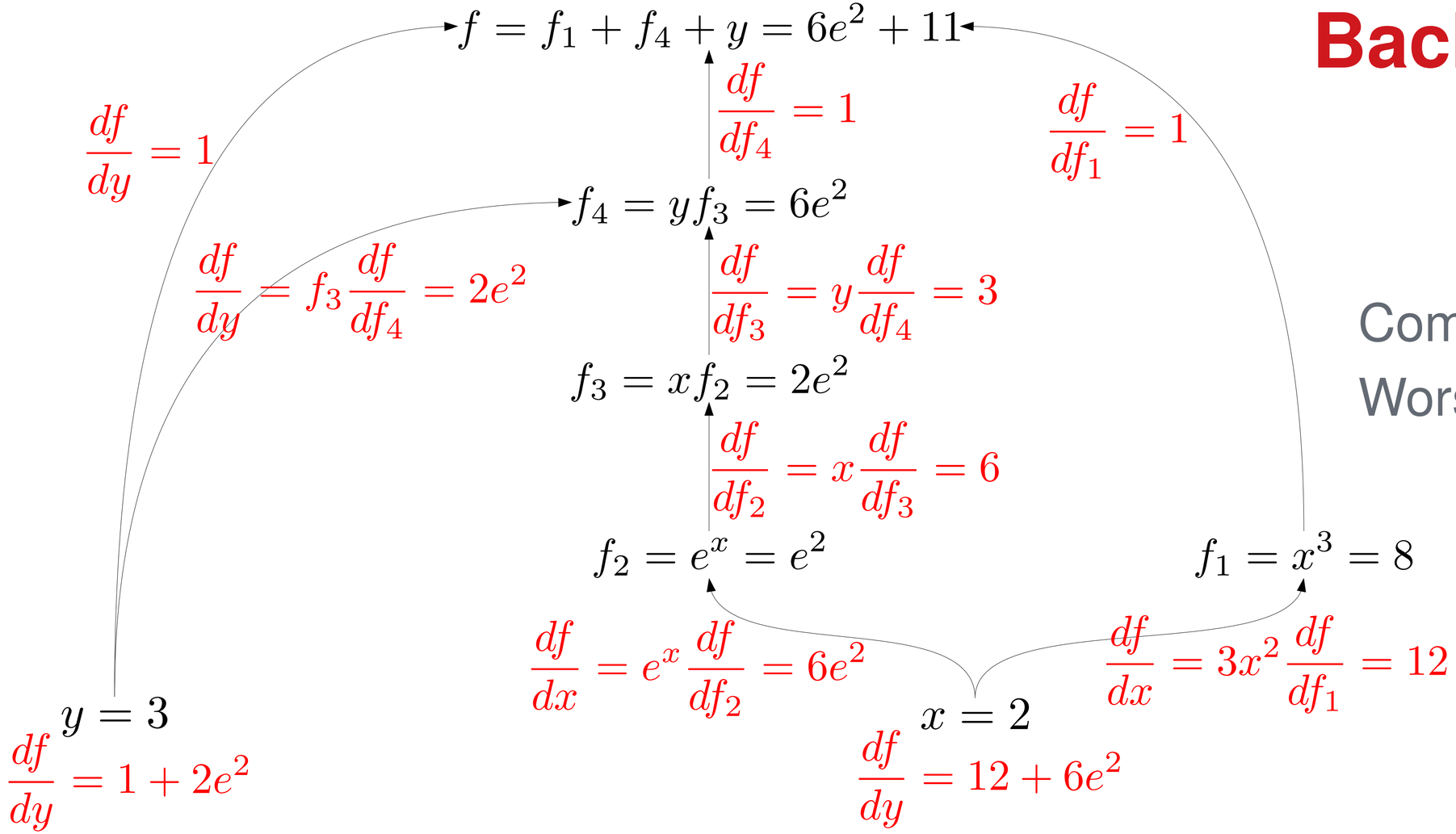


Complexity is now 2*T(f) !!

Computational Graph

Reverse Mode AD

$$f(x=2, y=3) = x^3 + yxe^x + y$$



Backpropagation!

Complexity is now 2*T(f) !!

Worse space complexity

Automatic Differentiation

Core difference:

- Forward mode AD
    - Gradients flows forwards with the nodes
    - Efficient derivative of many output functions, w.r.t one input
- Reverse mode AD
    - Gradient flows backward on the edges
    - Efficient derivative of one output function w.r.t. many inputs

Adam Paszke, Sam Gross, Soumith Chintala, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, **Zeming Lin**, Natalia Gimelshein, Alban Desmaison, Andreas Kopf, Edward Yang, Zach Devito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Dmytro Dzughalov, Yangqing Jia, Orion Richardson, James Reed & Team

Static Computational Graph

TensorFlow, Torch7, Theano etc

- Write your computational graph
- Run inputs through computational graph

Static Computational Graph

TensorFlow, Torch7, Theano etc

- Write your computational graph
- Run inputs through computational graph

```
>>> import theano
>>> import theano.tensor as T
>>> x = T.dmatrix('x')
>>> s = 1 / (1 + T.exp(-x))
>>> logistic = theano.function([x], s)
>>> logistic([[0, 1], [-1, -2]])
array([[ 0.5        ,  0.73105858],
       [ 0.26894142,  0.11920292]])
```

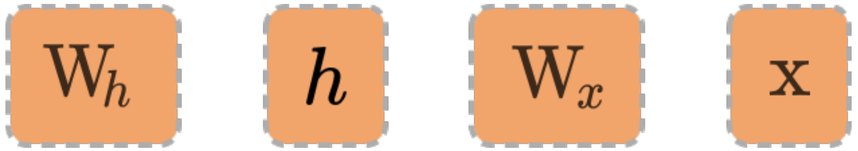
Tape Based Computational Graph

HIPS Autograd, Chainer, TF2.0, PyTorch etc

- Variables are graph nodes

- Implement mathematical functions on variables that create a dynamic computational graph

PyTorch Autograd

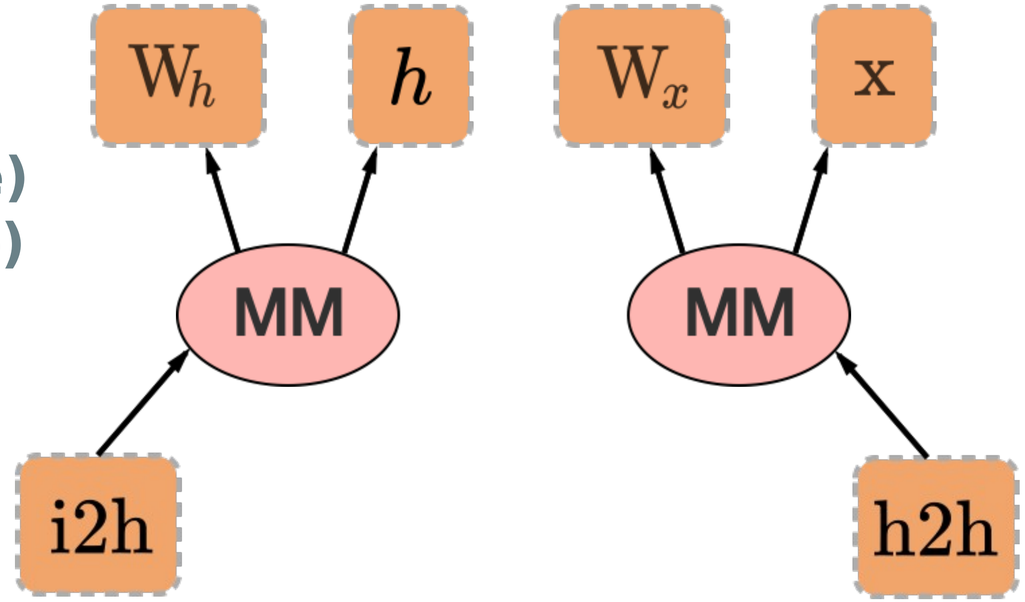



```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

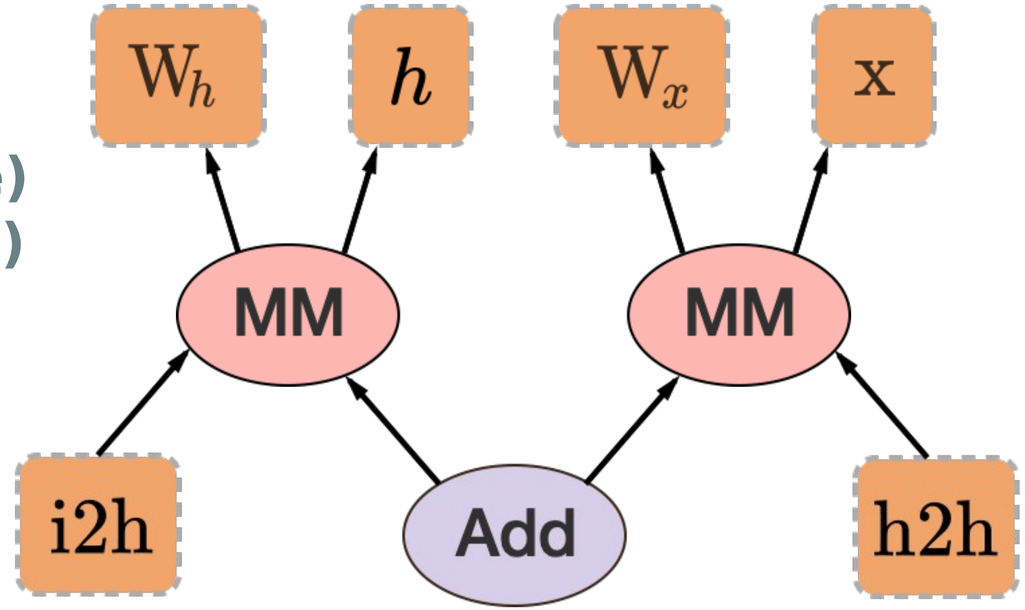
```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
```



PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

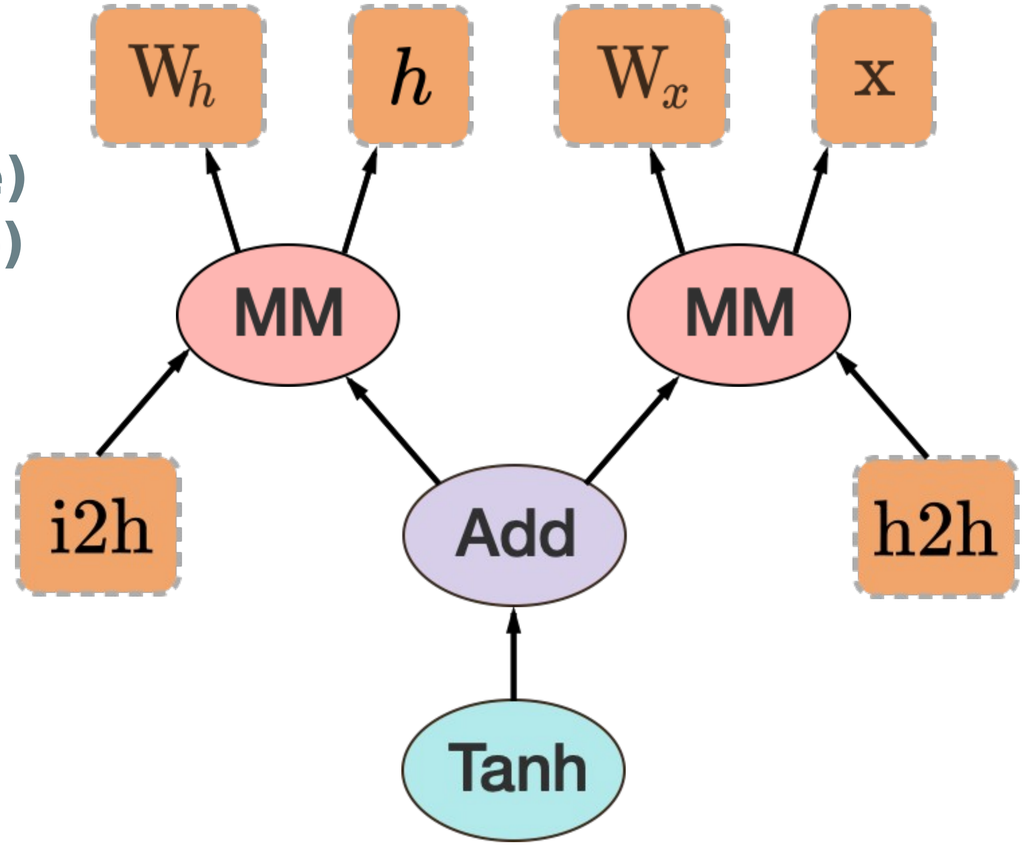
```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
sum = i2h + h2h
```



PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
sum = i2h + h2h
next_h = sum.tanh()
```

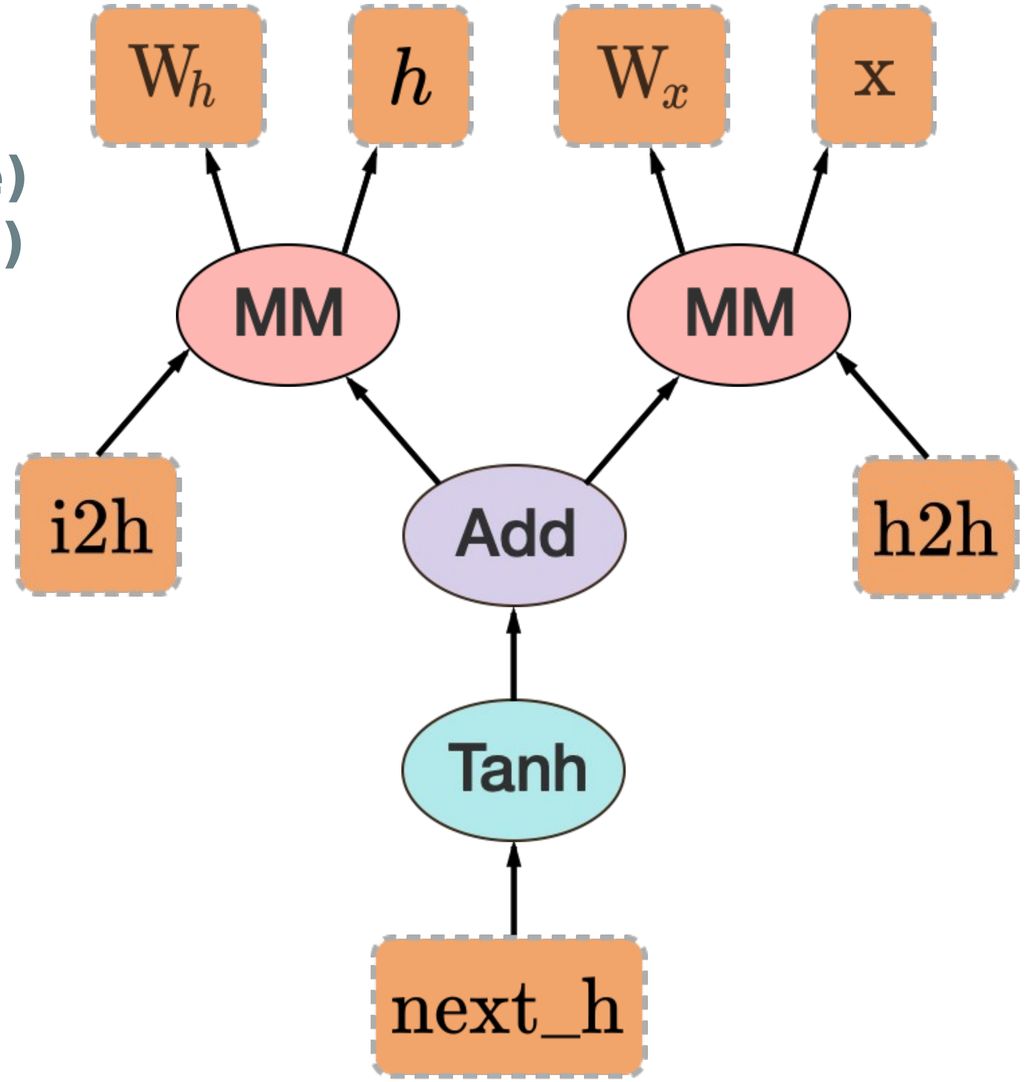


PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
sum = i2h + h2h
next_h = sum.tanh()
```

```
next_h.backward(torch.ones(1, 20))
```

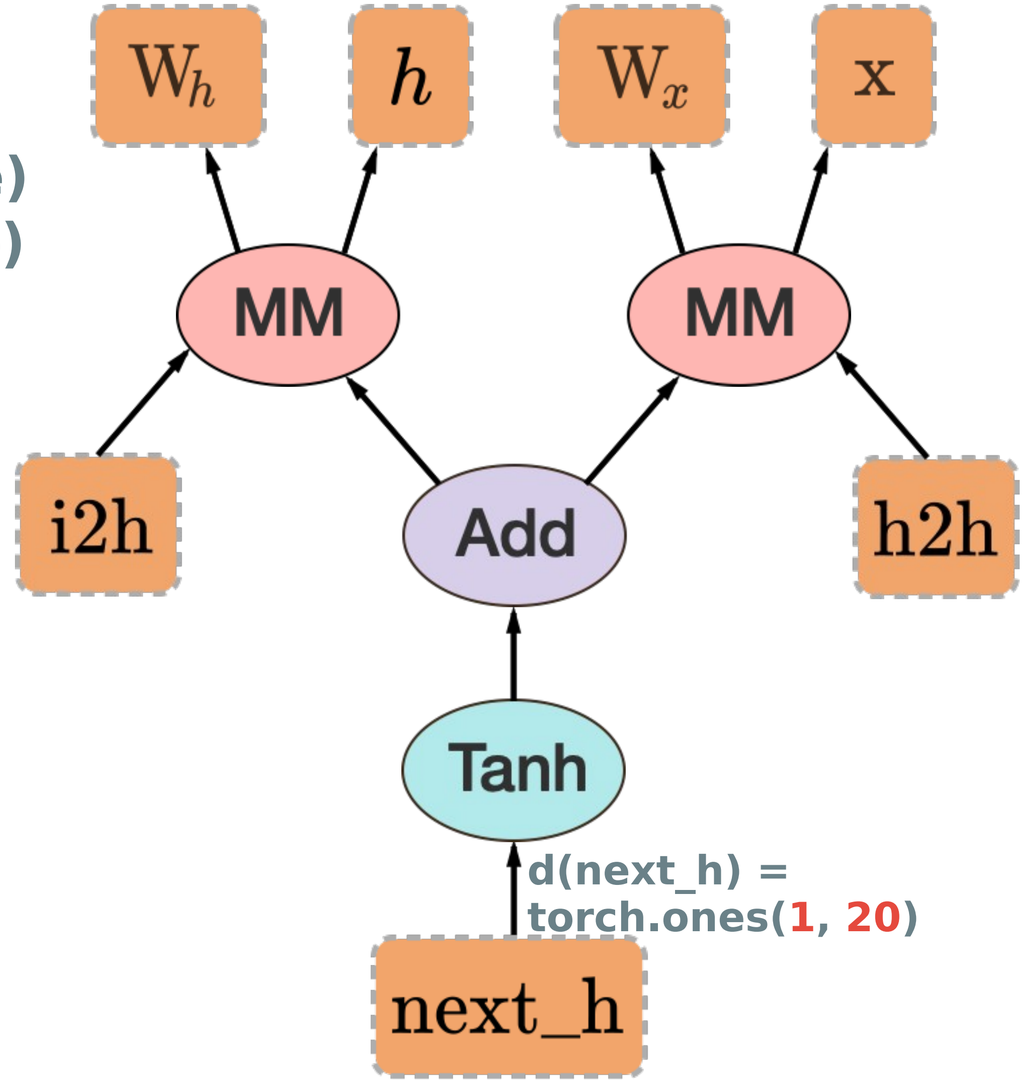


PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

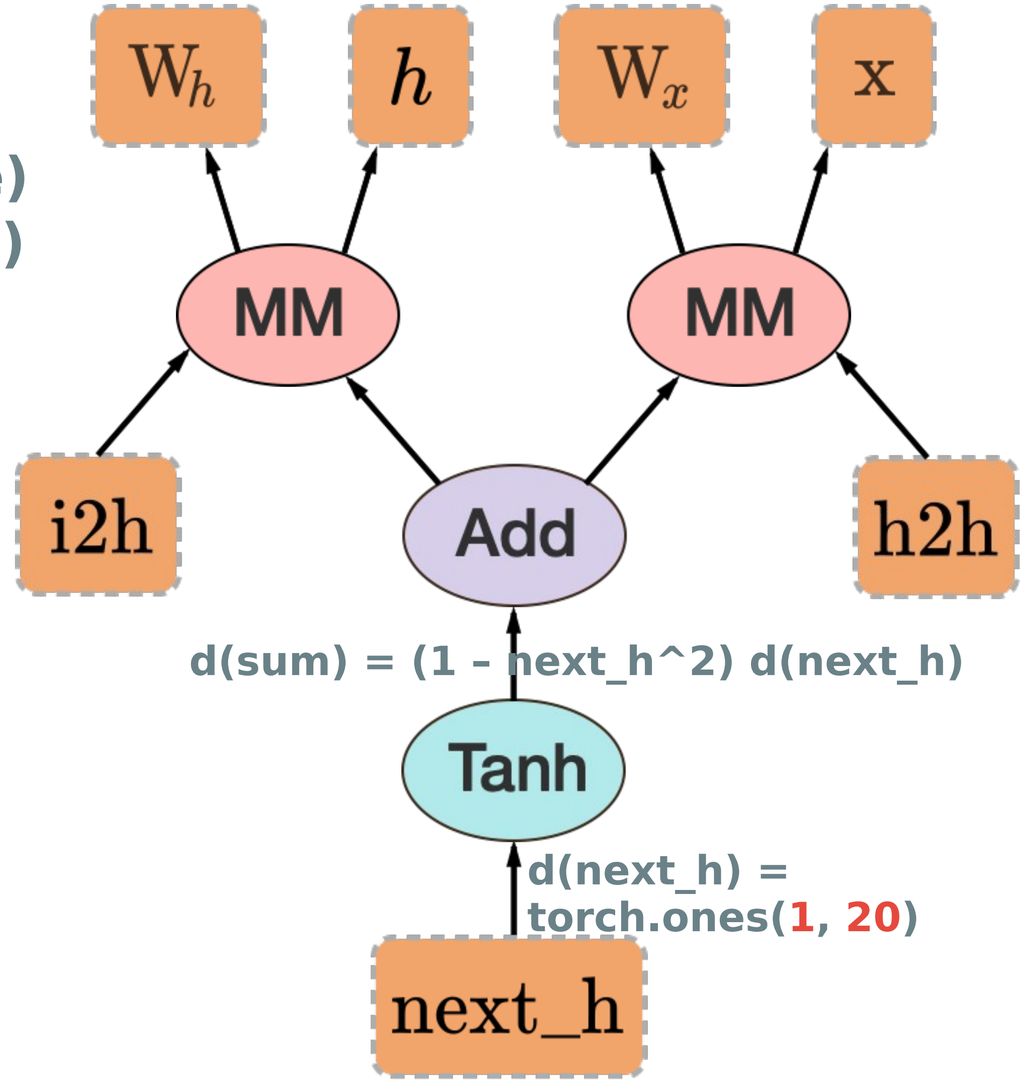
```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
sum = i2h + h2h
next_h = sum.tanh()
```

```
next_h.backward(torch.ones(1, 20))
```



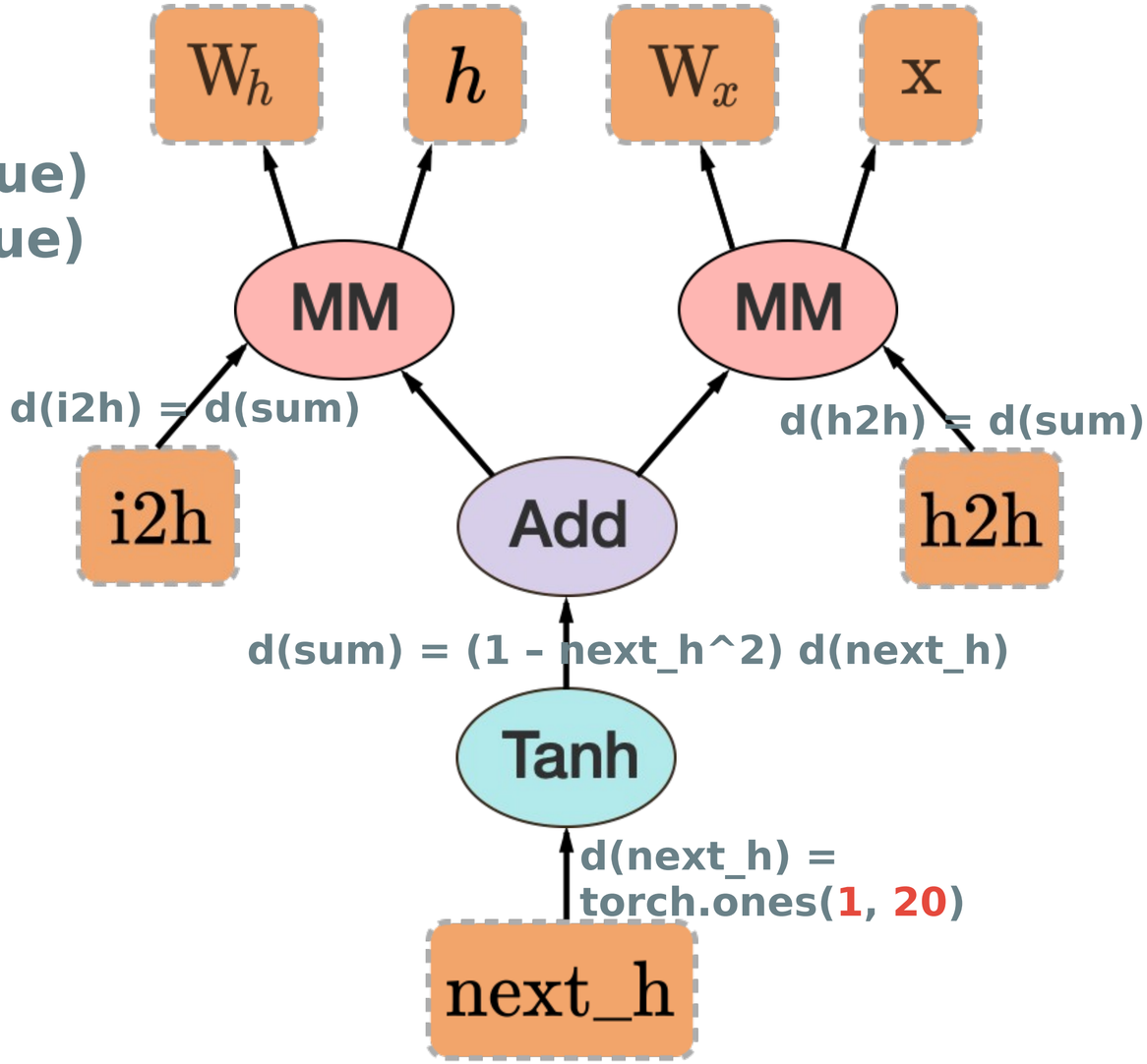
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
sum = i2h + h2h
next_h = sum.tanh()

next_h.backward(torch.ones(1, 20))
```



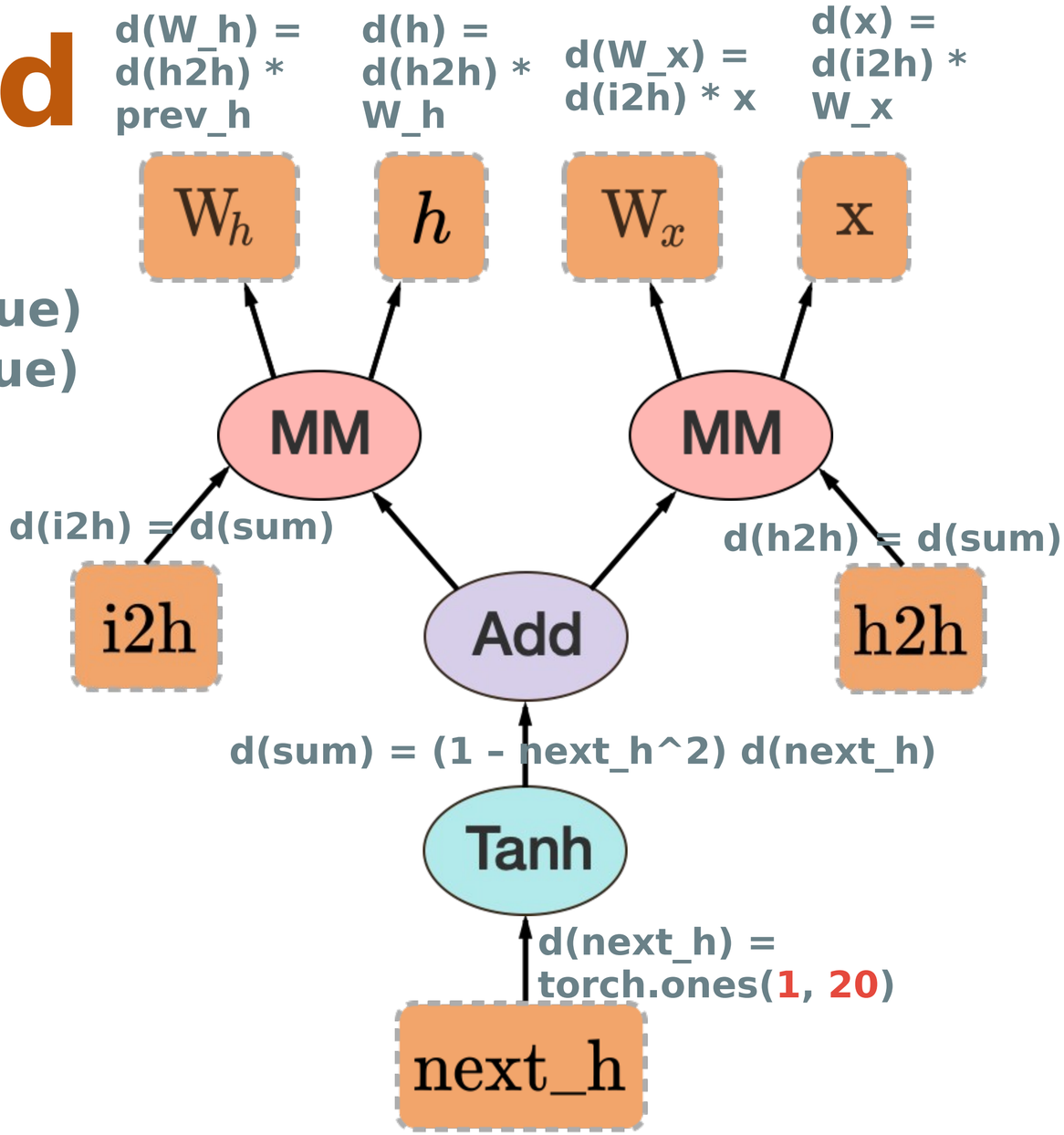
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
sum = i2h + h2h
next_h = sum.tanh()

next_h.backward(torch.ones(1, 20))
```



PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
sum = i2h + h2h
next_h = sum.tanh()

next_h.backward(torch.ones(1, 20))
```



PyTorch: Module Abstraction

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

PyTorch: Module Abstraction

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

PyTorch: Module Abstraction

```
1   class Net(nn.Module):
2       def __init__(self):
3           super(Net, self).__init__()
4           self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5           self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6           self.conv2_drop = nn.Dropout2d()
7           self.fc1 = nn.Linear(320, 50)
8           self.fc2 = nn.Linear(50, 10)
9
10      def forward(self, x):
11          x = F.relu(F.max_pool2d(self.conv1(x), 2))
12          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13          x = x.view(-1, 320)
14          x = F.relu(self.fc1(x))
15          x = F.dropout(x, training=self.training)
16          x = self.fc2(x)
17          return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(10, 20))
21  output = model(input)
```

PyTorch: Module Abstraction

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

- nn.Linear, Conv2d are Modules

PyTorch: Module Abstraction

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

- nn.Linear, Conv2d are Modules
- Modules form a tree

PyTorch: Module Abstraction

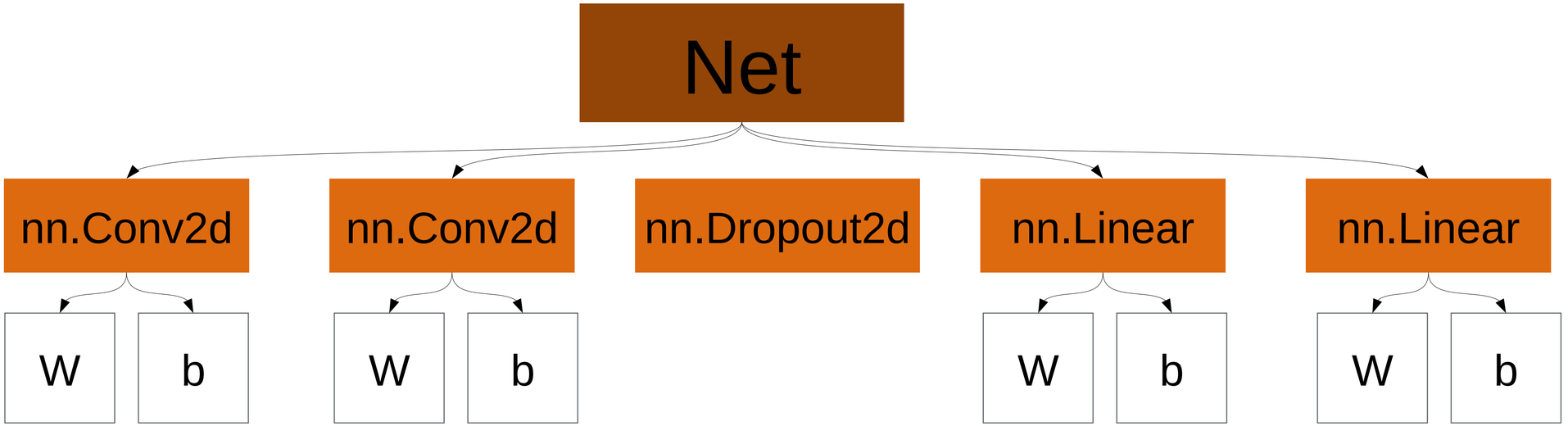
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

- nn.Linear, Conv2d are Modules
- Modules form a tree
- Composability!

PyTorch: Module Abstraction

```
1  class Net(nn.Module):
2      def __init__(self):
3          super(Net, self).__init__()
4          self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5          self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6          self.conv2_drop = nn.Dropout2d()
7          self.fc1 = nn.Linear(320, 50)
8          self.fc2 = nn.Linear(50, 10)
9
10     def
11
12
13
14
15
16
17
18
19 model = Net()
20 input = Variable(torch.randn(10, 20))
21 output = model(input)
```

- nn.Linear, Conv2d are Modules
- Modules form a tree
- Composability!



Optimization package

SGD, Adagrad, RMSProp, LBFGS, etc.

```
net = Net()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = F.cross_entropy(output, target)
    loss.backward()
    optimizer.step()
```

PyTorch: Memory Efficiency

- Some functions do not depend on inputs to compute gradients
- e.g. $\frac{d}{dx} e^{f(x)} = e^{f(x)} f'(x)$, tape needs to store $e^{f(x)}$
- $\frac{d}{dx}[f(x) + g(x)]$, tape does not need to store $f(x)$
- Reshaping operations, unary ops, scalar ops, etc... do not have to store inputs

- Most of the time, only parameters require gradients!
- Keep a requires_grad variable around

PyTorch: Memory Efficiency

- Significant memory savings in computer vision
- RNNs are still inefficient :(

PyTorch: Memory Efficiency

- Significant memory savings in computer vision
- RNNs are still inefficient :(

Training memory

The lower, the better

Caffe2 **PyTorch**

type	Detectron (P100)	mmdetection (V100)	maskrcnn_benchmark (V100)
Faster R-CNN R-50 C4	6.3	-	5.8
Faster R-CNN R-50 FPN	7.2	4.9	4.4
Faster R-CNN R-101 FPN	8.9	-	7.1
Faster R-CNN X-101-32x8d FPN	7.0	-	7.6
Mask R-CNN R-50 C4	6.6	-	5.8
Mask R-CNN R-50 FPN	8.6	5.9	5.2
Mask R-CNN R-101 FPN	10.2	-	7.9
Mask R-CNN X-101-32x8d FPN	7.7	-	7.8

PyTorch: Memory Efficiency

- Significant memory savings in computer vision
- RNNs are still inefficient :(

Training memory

The lower, the better

Caffe2 **PyTorch**

type	Detectron (P100)	mmdetection (V100)	maskrcnn_benchmark (V100)
Faster R-CNN R-50 C4	6.3	-	5.8
Faster R-CNN R-50 FPN	7.2	4.9	4.4
Faster R-CNN R-101 FPN	8.9	-	7.1
Faster R-CNN X-101-32x8d FPN	7.0	-	7.6
Mask R-CNN R-50 C4	6.6	-	5.8
Mask R-CNN R-50 FPN	8.6	5.9	5.2
Mask R-CNN R-101 FPN	10.2	-	7.9
Mask R-CNN X-101-32x8d FPN	7.7	-	7.8

~15% savings!

Debugging

-
-
-

Debugging

- PyTorch is a Python extension
-
-

Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger
-

Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger
- Use the most popular debugger:

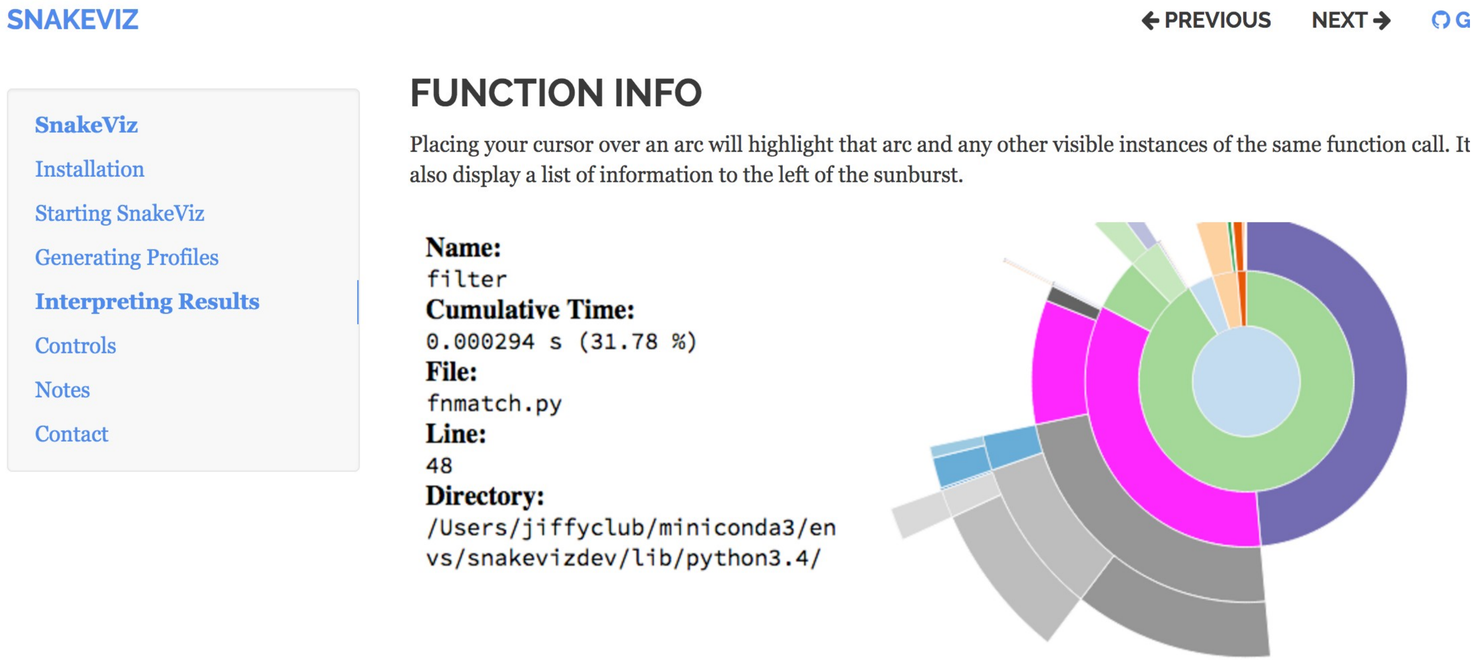
Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger
- Use the most popular debugger:

print(foo)

Identifying bottlenecks

- PyTorch is a Python extension
- Use your favorite Python profiler



Identifying bottlenecks

- PyTorch is a Python extension
- Use your favorite Python profiler: Line_Profiler

```
File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO + 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob == 'A':
   154     50000        66354      1.3     10.9              IntLoc = IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO = IntLoc - IntGlob
   156     50000        65494      1.3     10.8              EnumLoc = Ident1
   157     50000        68001      1.4     11.2          if EnumLoc == Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

Identifying bottlenecks

.torch.autograd.profiler

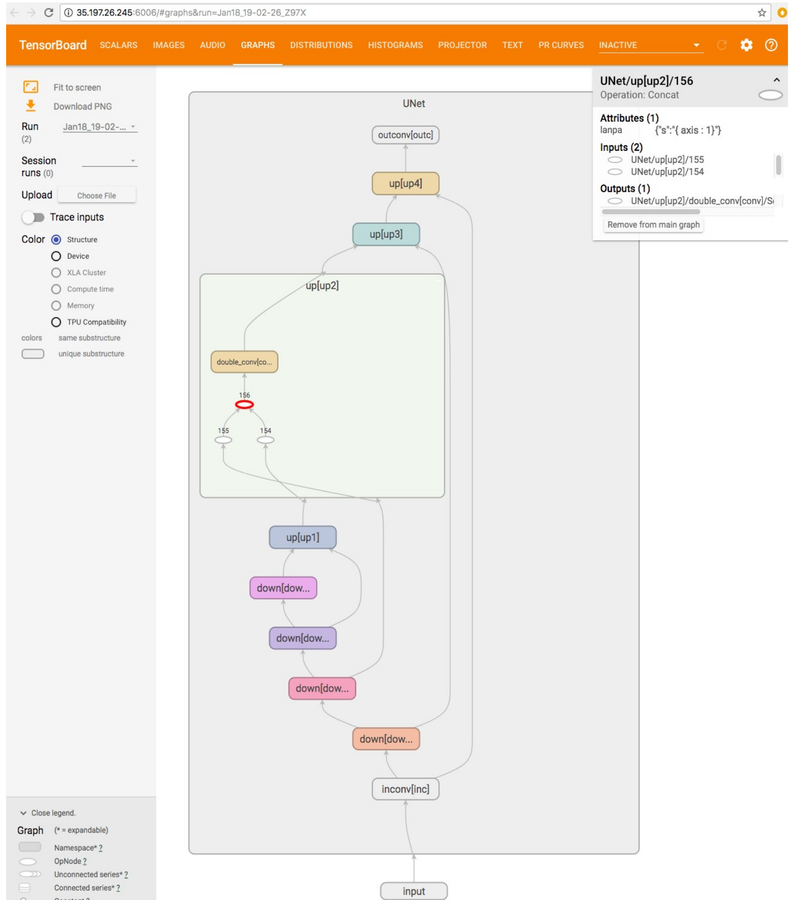
```
>>> x = torch.randn((1, 1), requires_grad=True)
>>> with torch.autograd.profiler.profile() as prof:
...     y = x ** 2
...     y.backward()
>>> # NOTE: some columns were removed for brevity
... print(prof)
-----------------------------------  --------------  --------------
Name                                        CPU time        CUDA time
-----------------------------------  --------------  --------------
PowConstant                               142.036us         0.000us
N5torch8autograd9GraphRootE                63.524us         0.000us
PowConstantBackward                       184.228us         0.000us
MulConstant                                50.288us         0.000us
PowConstant                                28.439us         0.000us
Mul                                        20.154us         0.000us
N5torch8autograd14AccumulateGradE          13.790us         0.000us
N5torch8autograd5CloneE                     4.088us         0.000us
```

Visualization

TensorBoard-PyTorch

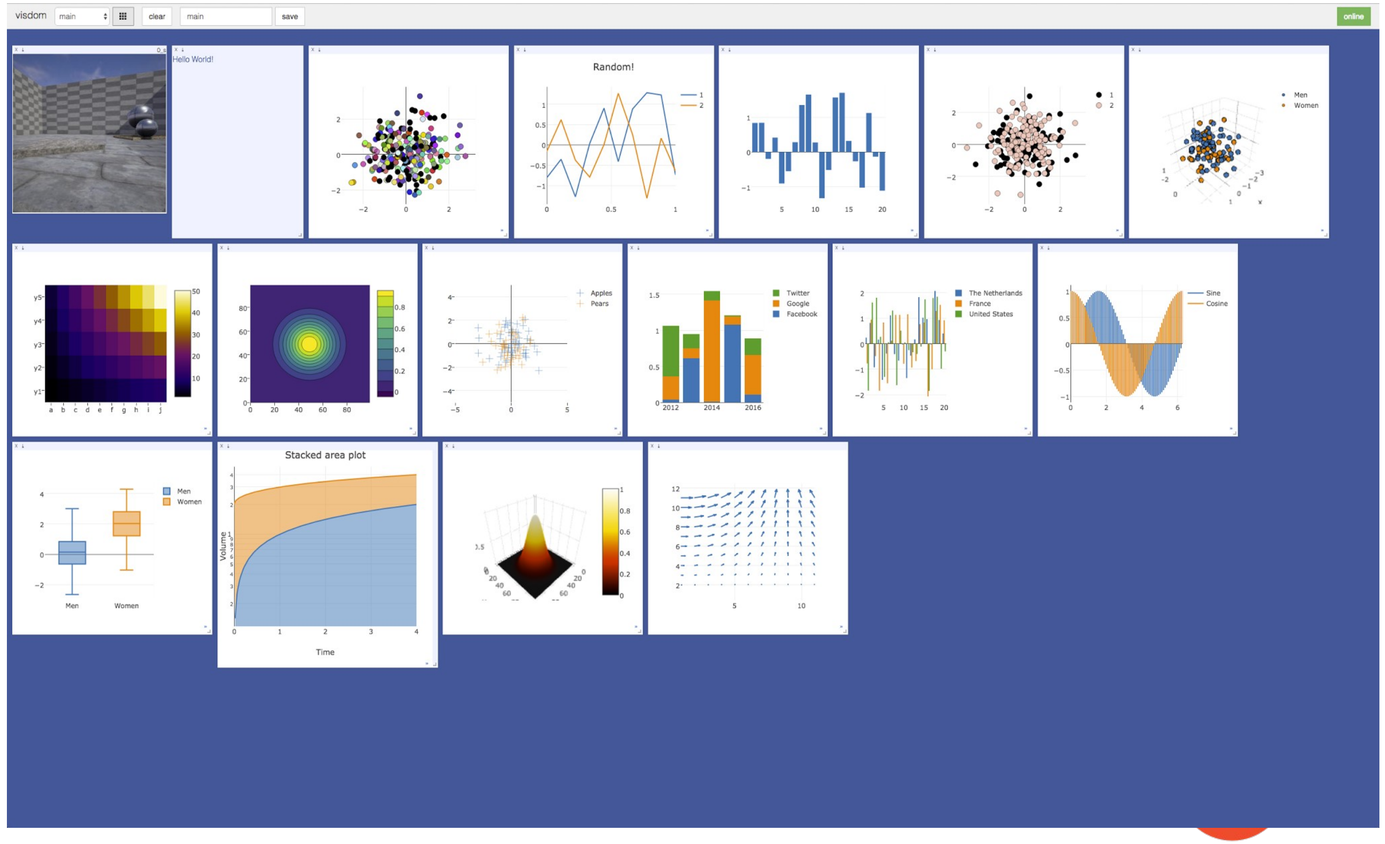
https://github.com/lanpa/tensorboard-pytorch

torch.utils.tensorboard (as of v1.1.0)



Visdom

https://github.com/facebookresearch/visdom



PyTorch

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize

PyTorch *Eager Mode*

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize

PyTorch *Eager Mode*

Models are Python programs

- Simple
- Debuggable — `print` and `pdb`
- Hackable — use any Python library

- Needs Python to run
- Difficult to optimize and parallelize

PyTorch *Script Mode*

Models are programs written in an optimizable subset of Python

- Production deployment
- No Python dependency
- Optimizable

PYTORCH JIT

Tools to transition eager code into script mode

EAGER MODE

For prototyping, training, and experiments

`@torch.jit.script`

`torch.jit.trace`

SCRIPT MODE

For use at scale in production

Transitioning a model with `torch.jit.trace`

Take an existing eager model, and provide example inputs.

The tracer runs the function, recording the tensor operations performed.

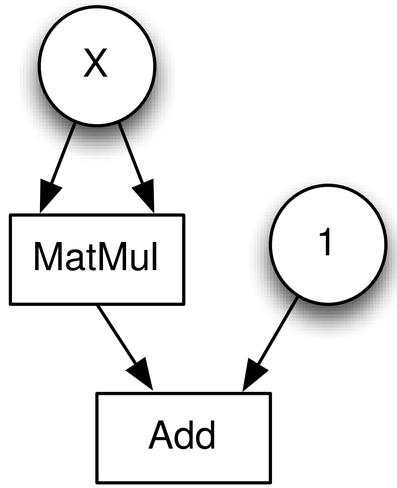
We turn the recording into a Torch Script module.

- Can reuse existing eager model code
- ⚠ Control-flow is ignored

```
import torch
import torchvision


def foo(x, y):
  return 2*x + y


# trace a model by providing example inputs
traced_foo = torch.jit.trace(foo,
  (torch.rand(3), torch.rand(3)))


traced_resnet = torch.jit.trace(
  torchvision.models.resnet18(),
  torch.rand(1, 3, 224, 224))
```

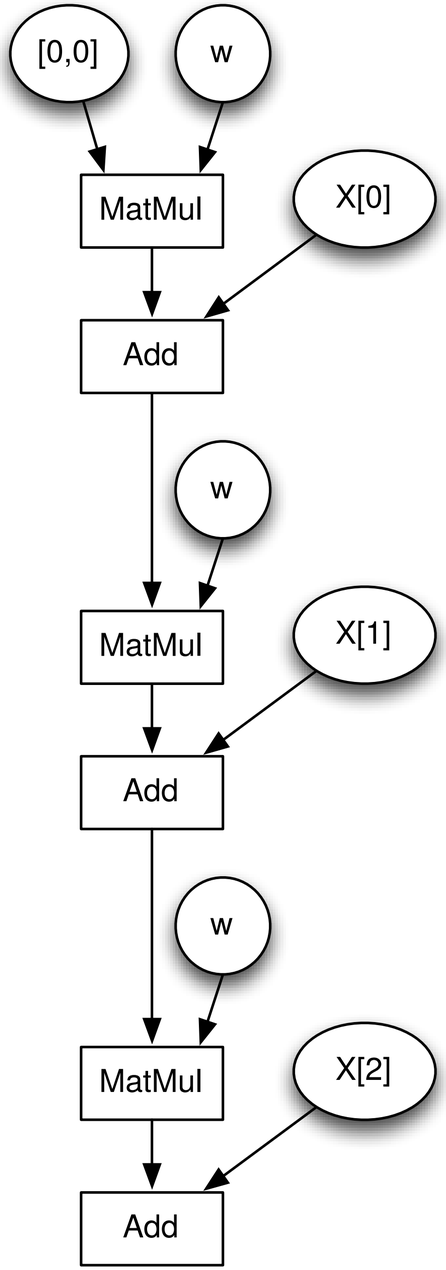
Tracing

```
def foo(x, t):
  y = x.mm(x)
  print(y) # still works!
  return y + t

x = torch.Tensor([[1,2],[3,4]])
foo(x, 1)

trace = torch.jit.trace(foo, (x, 1))
trace.save(“serialized.pt”)
```



Tracing

```
def foo(x, t):
  y = x.mm(x)
  print(y) # still works!
  return y + t

def bar(x, w):
    y = torch.zeros(1, 2)
    for t in x:
        y = foo(y, w, t)
    return y

trace = torch.jit.trace(foo, (x, 1))
trace.save(“serialized.pt”)
```



Transitioning a model with `@torch.jit.script`

Write model directly in a subset of Python, annotated with `@torch.jit.script` or `@torch.jit.script_method`

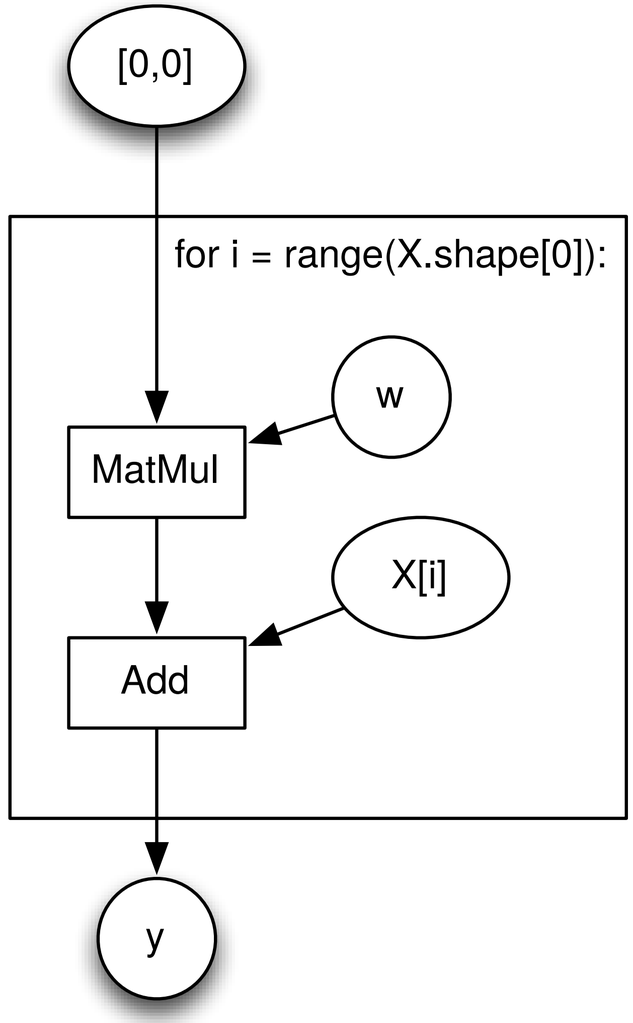
- Control-flow is preserved
- `print` statements can be used for debugging
- Remove the annotations to debug using standard Python tools.

```
class RNN(torch.jit.ScriptModule):
  def __init__(self, W_h, U_h, W_y, b_h, b_y):
    super(RNN, self).__init__()
    self.W_h = nn.Parameter(W_h)
    self.U_h = nn.Parameter(U_h)
    self.W_y = nn.Parameter(W_y)
    self.b_h = nn.Parameter(b_h)
    self.b_y = nn.Parameter(b_y)
  @torch.jit.script_method
  def forward(self, x, h):
    y = []
    for t in range(x.size(0)):
      h = torch.tanh(x[t] @ self.W_h + h @ self.U_h +
      y += [torch.tanh(h @ self.W_y + self.b_y)]
      if t % 10 == 0:
        print("stats: ", h.mean(), h.var())
    return torch.stack(y), h
```

Script

```
def foo(x, t):
  y = x.mm(x)
  print(y) # still works!
  return y + t
trace = torch.jit.trace(foo, (x, 1))
```

```
@script
def bar(x, w):
    y = torch.zeros(1, 2)
    for t in x:
        y = trace(y, w, t)
    return y
```



Under the hood of `@torch.jit.script`

```
class MyModule(torch.jit.ScriptModule):
    def __init__(self, N, M):
        super(MyModule, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(N, M))

    @torch.jit.script_method
    def forward(self, input):
        if bool(input.sum() > 0):
          output = self.weight.mv(input)
        else:
          output = self.weight + input
        return output

ms = MyModule(3, 4)
```

```
ms.weight[1,2]
```

```
tensor(0.5476, grad_fn=<SelectBackward>)
```

```
ms(torch.rand(4))
```

```
tensor([1.2406, 1.4690, 1.8646], grad_fn=<MvBackward>)
```

```
ms.graph
```

```
graph(%input : Dynamic
      %5 : Dynamic) {
  %7 : int = prim::Constant[value=1]()
  %2 : int = prim::Constant[value=0]()
  %1 : Dynamic = aten::sum(%input)
  %3 : Dynamic = aten::gt(%1, %2)
  %4 : bool = prim::TensorToBool(%3)
  %output : Dynamic = prim::If(%4)
    block0() {
      %output.1 : Dynamic = aten::mv(%5, %input)
      -> (%output.1)
    }
    block1() {
      %output.2 : Dynamic = aten::add(%5, %input, %7)
      -> (%output.2)
    }
  return (%output);
}
```

```
print(ms.graph.pretty_print())
```

```
def graph(self,
    input: Tensor,
    _0: Tensor) -> Tensor:
  if bool(aten.gt(aten.sum(input), 0)):
    output = aten.mv(_0, input)
  else:
    output = aten.add(_0, input, alpha=1)
  return output
```

```
ms.save('mymodel.pt')
```

Predictable error messages `@torch.jit.script`

PARSE TIME

```
@torch.jit.script_method
def forward(self, input):
    if bool(input.sum() > 0):
      output = self.weight.mv(input_foo)
    else:
      output = self.weight + input
    return output
```

```
RuntimeError:
undefined value inpu:
@torch.jit.script_method
def forward(self, input):
    if bool(input.sum() > 0):
      output = self.weight.mv(inpu)
                              ~~~~ <--- HERE
    else:
      output = self.weight + input
    return output
```

RUNTIME

```
ms(torch.rand(111))
```

```
RuntimeError:
size mismatch, [3 x 4], [111] at caffe2/
operation failed in interpreter:
@torch.jit.script_method
def forward(self, input):
    if bool(input.sum() > 0):
      output = self.weight.mv(input)
               ~~~~~~~~~~~~~~~ <--- HERE
    else:
      output = self.weight + input
    return output
```

Loading a model without Python

Torch Script models can be saved to a model archive, and loaded in a python-free executable using a C++ API.

Our C++ Tensor API is the same as our Python API, so you can do preprocessing and post processing before calling the model.

```
# Python: save model
traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
traced_resnet.save("serialized_resnet.pt")
```

```
// C++: load and run model
auto module = torch::jit::load("serialized_resnet.pt");
auto example = torch::rand({1, 3, 224, 224});
auto output = module->forward({example}).toTensor();
std::cout << output.slice(1, 0, 5) << '\n';
```

PyTorch C++ API

```cpp
#include <torch/torch.h>

// Define a new Module.
struct Net : torch::nn::Module {
  Net() {
    // Construct and register two Linear submodules.
    fc1 = register_module("fc1", torch::nn::Linear(784, 64));
    fc2 = register_module("fc2", torch::nn::Linear(64, 32));
    fc3 = register_module("fc3", torch::nn::Linear(32, 10));
  }

  // Implement the Net's algorithm.
  torch::Tensor forward(torch::Tensor x) {
    // Use one of many tensor manipulation functions.
    x = torch::relu(fc1->forward(x.reshape({x.size(0), 784})));
    x = torch::dropout(x, /*p=*/0.5, /*train=*/is_training());
    x = torch::relu(fc2->forward(x));
    x = torch::log_softmax(fc3->forward(x), /*dim=*/1);
    return x;
  }

  // Use one of many "standard library" modules.
  torch::nn::Linear fc1{nullptr}, fc2{nullptr}, fc3{nullptr};
};
```

```cpp
auto net = std::make_shared<Net>();

// Create a multi-threaded data loader for the MNIST dataset.
auto data_loader = torch::data::make_data_loader(
    torch::data::datasets::MNIST("./data").map(
        torch::data::transforms::Stack<>()),
    /*batch_size=*/64);

// Instantiate an SGD optimization algorithm to update our Net's parameters.
torch::optim::SGD optimizer(net->parameters(), /*lr=*/0.01);

for (size_t epoch = 1; epoch <= 10; ++epoch) {
  size_t batch_index = 0;
  // Iterate the data loader to yield batches from the dataset.
  for (auto& batch : *data_loader) {
    // Reset gradients.
    optimizer.zero_grad();
    // Execute the model on the input data.
    torch::Tensor prediction = net->forward(batch.data);
    // Compute a loss value to judge the prediction of our model.
    torch::Tensor loss = torch::nll_loss(prediction, batch.target);
    // Compute gradients of the loss w.r.t. the parameters of our model.
    loss.backward();
    // Update the parameters based on the calculated gradients.
    optimizer.step();
```

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion
- Reordering chunk + elementwise

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion
- Reordering chunk + elementwise
- Loop unrolling

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion
- Reordering chunk + elementwise
- Loop unrolling
- Batch Matrix Multiply

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion
- Reordering chunk + elementwise
- Loop unrolling
- Batch Matrix Multiply
  AB, AC → chunk(A * cat(B, C))

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion
- Reordering chunk + elementwise
- Loop unrolling
- Batch Matrix Multiply
  AB, AC → chunk(A * cat(B, C))
- “Tree” Batch Matrix Multiple

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion
- Reordering chunk + elementwise
- Loop unrolling
- Batch Matrix Multiply
  AB, AC → chunk(A * cat(B, C))
- “Tree” Batch Matrix Multiple
  AB + CD

Static Graph Optimizations

```
class LSTMCell(jit.ScriptModule):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.weight_ih = Parameter(torch.randn(4 * hidden_size, input_size))
        self.weight_hh = Parameter(torch.randn(4 * hidden_size, hidden_size))
        self.bias_ih = Parameter(torch.randn(4 * hidden_size))
        self.bias_hh = Parameter(torch.randn(4 * hidden_size))

    @jit.script_method
    def forward(self, input, state):
        # type: (Tensor, Tuple[Tensor, Tensor]) -> Tuple[Tensor, Tuple[Tensor, Tensor]]
        hx, cx = state
        gates = (torch.mm(input, self.weight_ih.t()) + self.bias_ih +
                 torch.mm(hx, self.weight_hh.t()) + self.bias_hh)
        ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

        ingate = torch.sigmoid(ingate)
        forgetgate = torch.sigmoid(forgetgate)
        cellgate = torch.tanh(cellgate)
        outgate = torch.sigmoid(outgate)

        cy = (forgetgate * cx) + (ingate * cellgate)
        hy = outgate * torch.tanh(cy)

        return hy, (hy, cy)
```

- Elementwise fusion
- Reordering chunk + elementwise
- Loop unrolling
- Batch Matrix Multiply
  AB, AC → chunk(A * cat(B, C))
- “Tree” Batch Matrix Multiple
  AB + CD
  → cat([A,C], 1) * cat([B,D], 0)

Ecosystem

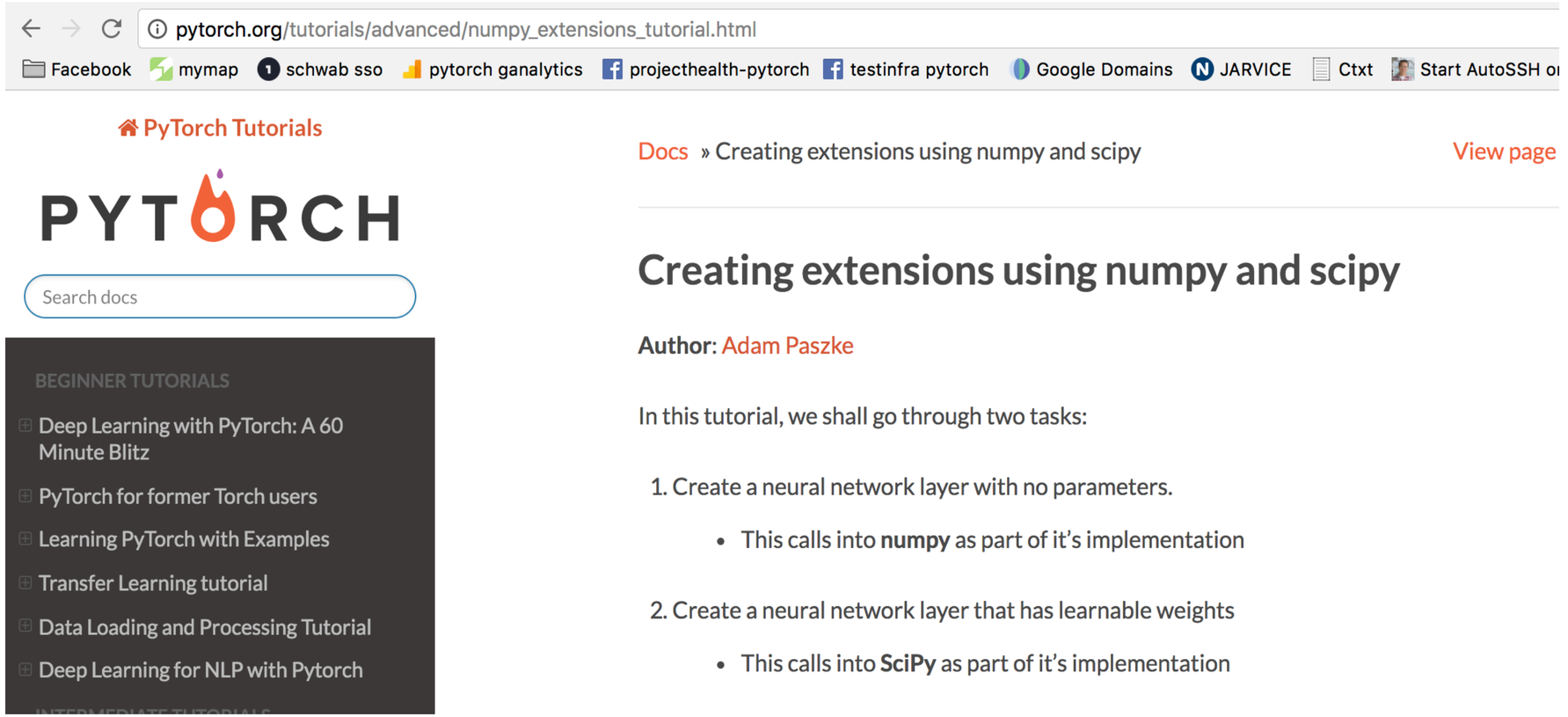
- Use the entire Python ecosystem at your will

Ecosystem

- Use the entire Python ecosystem at your will
- Including SciPy, Scikit-Learn, etc.

Ecosystem

- Use the entire Python ecosystem at your will
- Including SciPy, Scikit-Learn, etc.



Ecosystem

- A shared model-zoo:

We provide pre-trained models for the ResNet variants and AlexNet, using the PyTorch `torch.utils.model_zoo`. These can constructed by passing `pretrained=True`:

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
```

Ecosystem

https://github.com/skorch-dev/skorch

Scikit + PyTorch

Ecosys

SK F

```python
import numpy as np
from sklearn.datasets import make_classification
from torch import nn
import torch.nn.functional as F

from skorch import NeuralNetClassifier


X, y = make_classification(1000, 20, n_informative=10, random_state=0)
X = X.astype(np.float32)
y = y.astype(np.int64)

class MyModule(nn.Module):
    def __init__(self, num_units=10, nonlin=F.relu):
        super(MyModule, self).__init__()

        self.dense0 = nn.Linear(20, num_units)
        self.nonlin = nonlin
        self.dropout = nn.Dropout(0.5)
        self.dense1 = nn.Linear(num_units, 10)
        self.output = nn.Linear(10, 2)

    def forward(self, X, **kwargs):
        X = self.nonlin(self.dense0(X))
        X = self.dropout(X)
        X = F.relu(self.dense1(X))
        X = F.softmax(self.output(X), dim=-1)
        return X


net = NeuralNetClassifier(
    MyModule,
    max_epochs=10,
    lr=0.1,
    # Shuffle training data on each epoch
    iterator_train__shuffle=True,
)

net.fit(X, y)
y_proba = net.predict_proba(X)
```

Ecosystem

SKORCH

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler


pipe = Pipeline([
    ('scale', StandardScaler()),
    ('net', net),
])

pipe.fit(X, y)
y_proba = pipe.predict_proba(X)
```

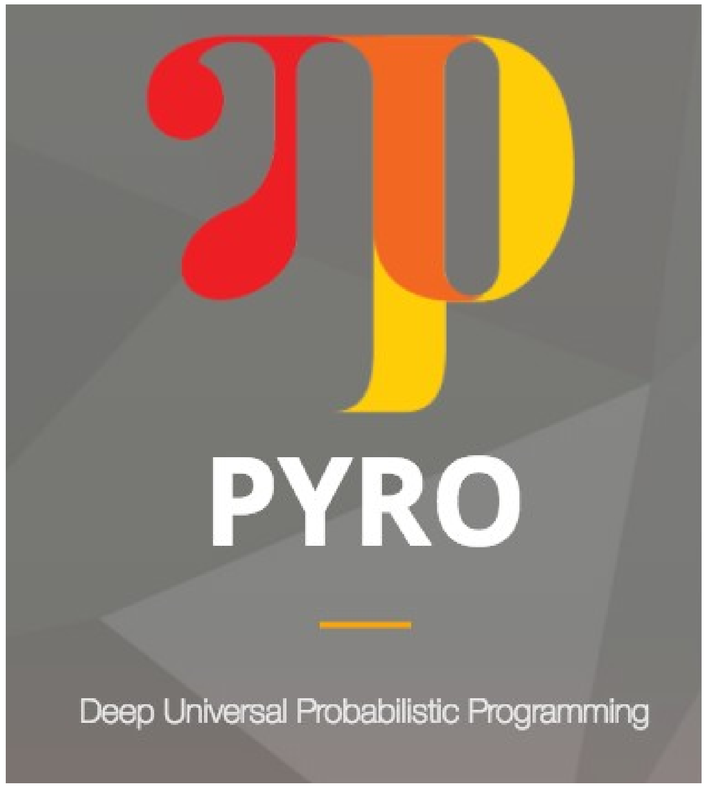
Ecosystem

SKORCH

```
from sklearn.model_selection import GridSearchCV


params = {
    'lr': [0.01, 0.02],
    'max_epochs': [10, 20],
    'module__num_units': [10, 20],
}
gs = GridSearchCV(net, params, refit=False, cv=3, scoring='accuracy')

gs.fit(X, y)
print(gs.best_score_, gs.best_params_)
```

Ecosystem

- Probabilistic Programming



PROB TORCH

github.com/probtorch/probtorch

http://pyro.ai/

Ecosystem

- Gaussian Processes

GPyTorch (Alpha Relase)

build passing

GPyTorch is a Gaussian Process library, implemented using PyTorch. It is designed for creating flexible and modular Gaussian Process models with ease, so that you don't have to be an expert to use GPs.

This package is currently under development, and is likely to change. Some things you can do right now:

- Simple GP regression (example here)
- Simple GP classification (example here)
- Multitask GP regression (example here)
- Scalable GP regression using kernel interpolation (example here)
- Scalable GP classification using kernel interpolation (example here)
- Deep kernel learning (example here)
- And (more!)

https://github.com/cornellius-gp/gpytorch

Ecosystem

- CycleGAN, pix2pix

  https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

Ecosystem

Machine Translation

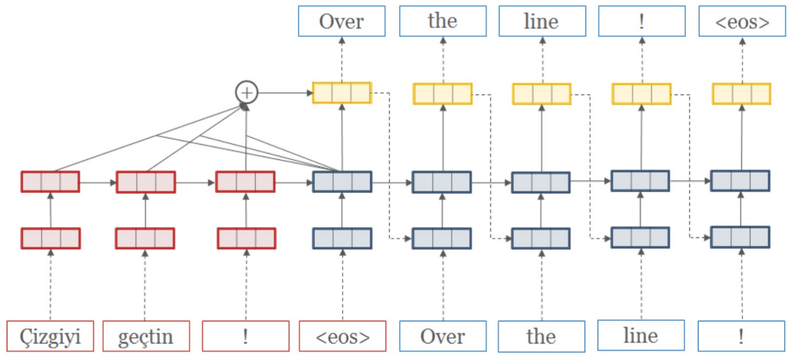
OpenNMT-py: Open-Source Neural Machine Translation

build passing

This is a Pytorch port of OpenNMT, an open-source (MIT) neural machine translation system. It is designed to be research friendly to try out new ideas in translation, summary, image-to-text, morphology, and many other domains.

Codebase is relatively stable, but PyTorch is still evolving. We currently recommend forking if you need to have stable code.

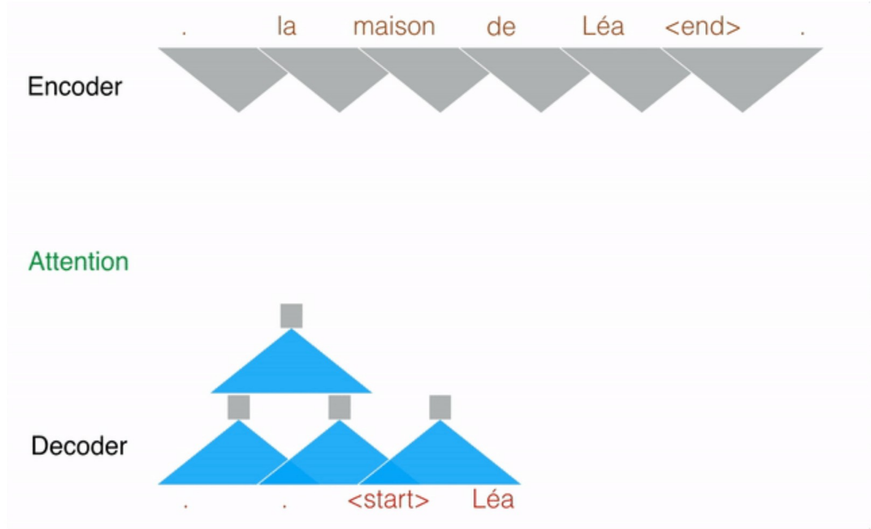
OpenNMT-py is run as a collaborative open-source project. It is maintained by Sasha Rush (Cambridge, MA), Ben Peters (Saarbrücken), and Jianyu Zhan (Shenzhen). The original code was written by Adam Lerer (NYC). We love contributions. Please consult the Issues page for any Contributions Welcome tagged post.



https://github.com/OpenNMT/OpenNMT-py

FAIR Sequence-to-Sequence Toolkit (PyTorch)

This is a PyTorch version of fairseq, a sequence-to-sequence learning toolkit from Facebook AI Research. The original authors of this reimplementation are (in no particular order) Sergey Edunov, Myle Ott, and Sam Gross. The toolkit implements the fully convolutional model described in Convolutional Sequence to Sequence Learning and features multi-GPU training on a single machine as well as fast beam search generation on both CPU and GPU. We provide pre-trained models for English to French and English to German translation.



https://github.com/facebookresearch/fairseq-py

Ecosystem

.Machine Translation

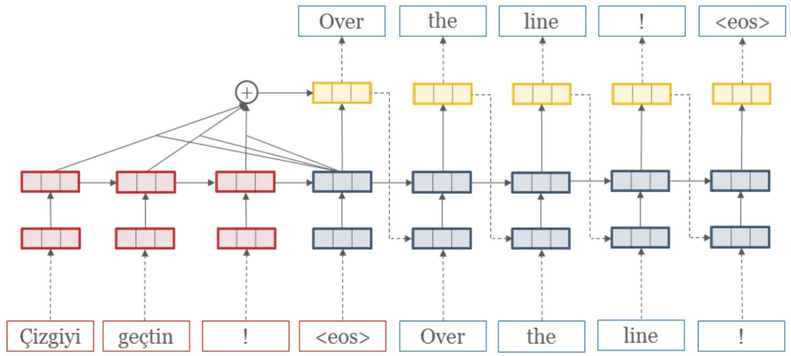
OpenNMT-py: Open-Source Neural Machine Translation

build passing

This is a Pytorch port of OpenNMT, an open-source (MIT) neural machine translation system. It is designed to be research friendly to try out new ideas in translation, summary, image-to-text, morphology, and many other domains.

Codebase is relatively stable, but PyTorch is still evolving. We currently recommend forking if you need to have stable code.

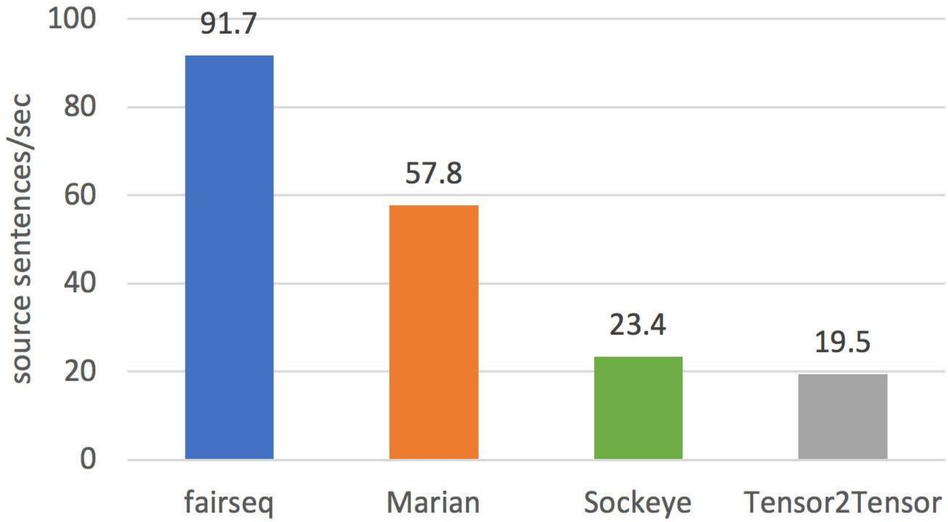
OpenNMT-py is run as a collaborative open-source project. It is maintained by Sasha Rush (Cambridge, MA), Ben Peters (Saarbrücken), and Jianyu Zhan (Shenzhen). The original code was written by Adam Lerer (NYC). We love contributions. Please consult the Issues page for any Contributions Welcome tagged post.



https://github.com/OpenNMT/OpenNMT-py

FAIR Sequence-to-Sequence Toolkit (PyTorch)

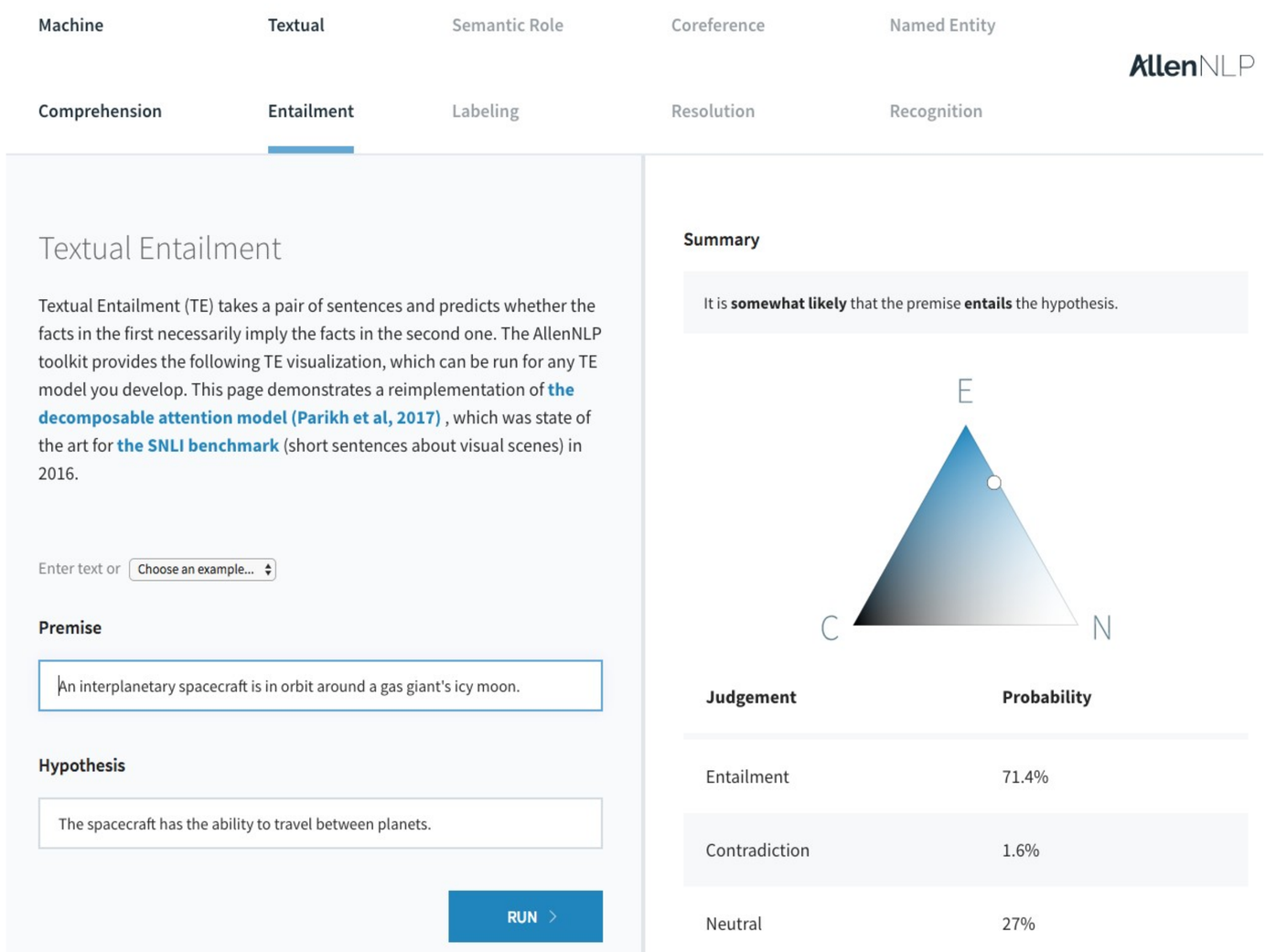
This is a PyTorch version of fairseq, a sequence-to-sequence learning toolkit from Facebook AI Research. The original authors of this reimplementation are (in no particular order) Sergey Edunov, Myle Ott, and Sam Gross. The toolkit implements the fully convolutional model described in Convolutional Sequence to Sequence Learning and features multi-GPU training on a single machine as well as fast beam search generation on both CPU and GPU. We provide pre-trained models for English to French and English to German translation.



https://github.com/facebookresearch/fairseq-py

Ecosystem

- AllenNLP http://allennlp.org/

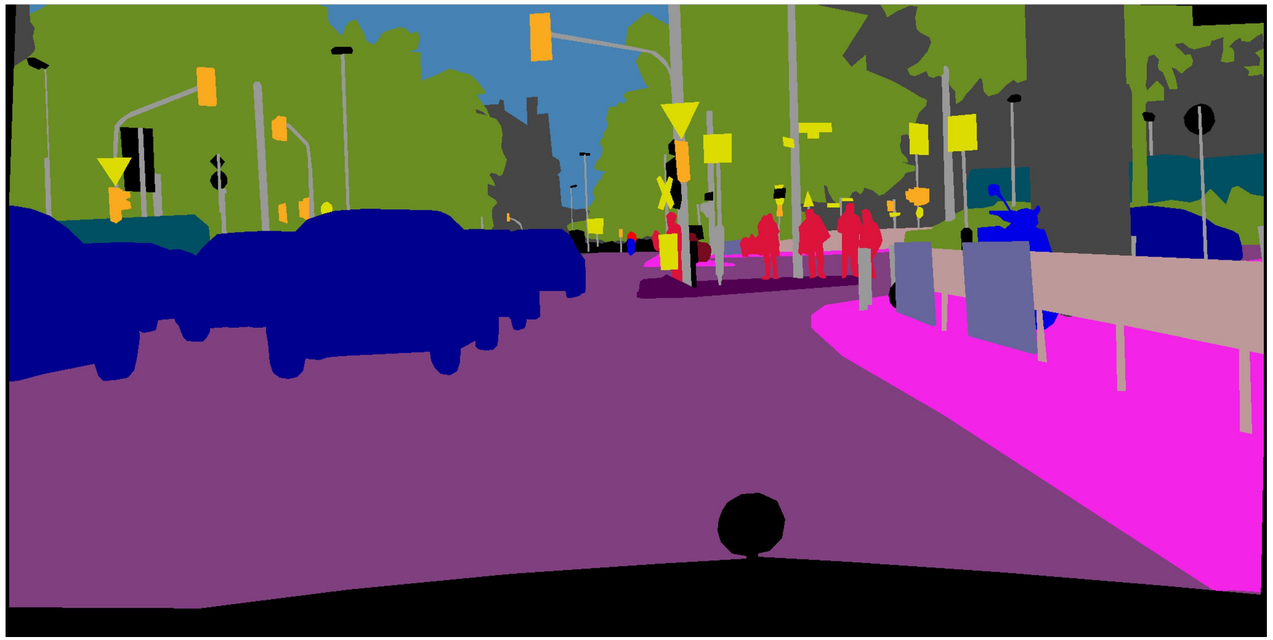


Ecosystem

- Pix2PixHD

https://github.com/NVIDIA/pix2pixHD

Input labels

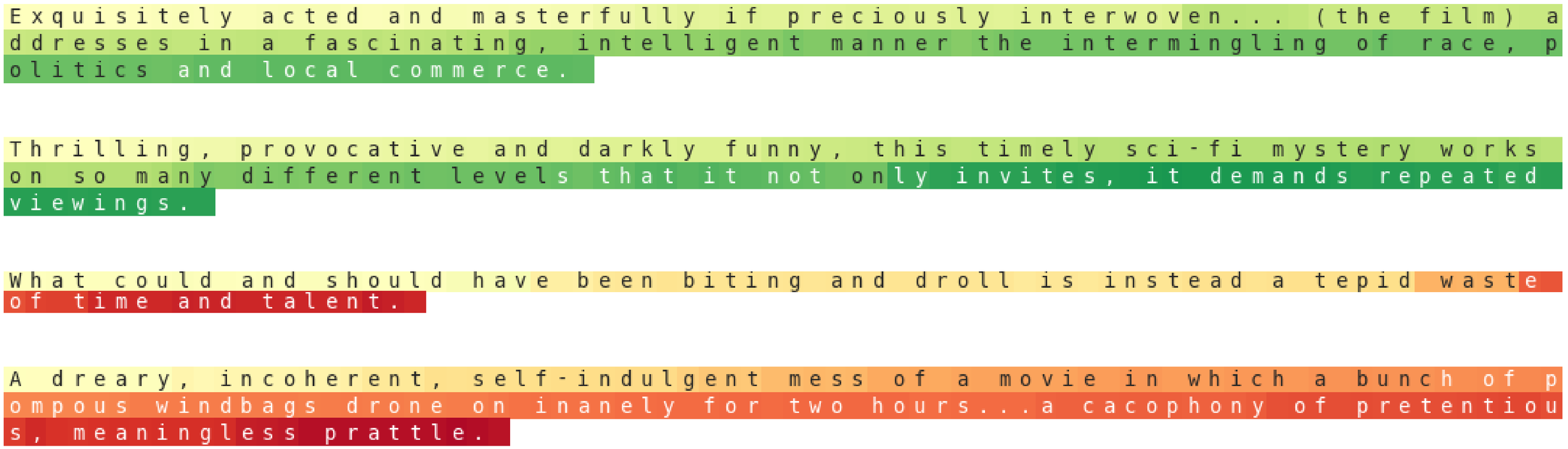


Synthesized image

Ecosystem

- Sentiment Discovery

https://github.com/NVIDIA/sentiment-discovery



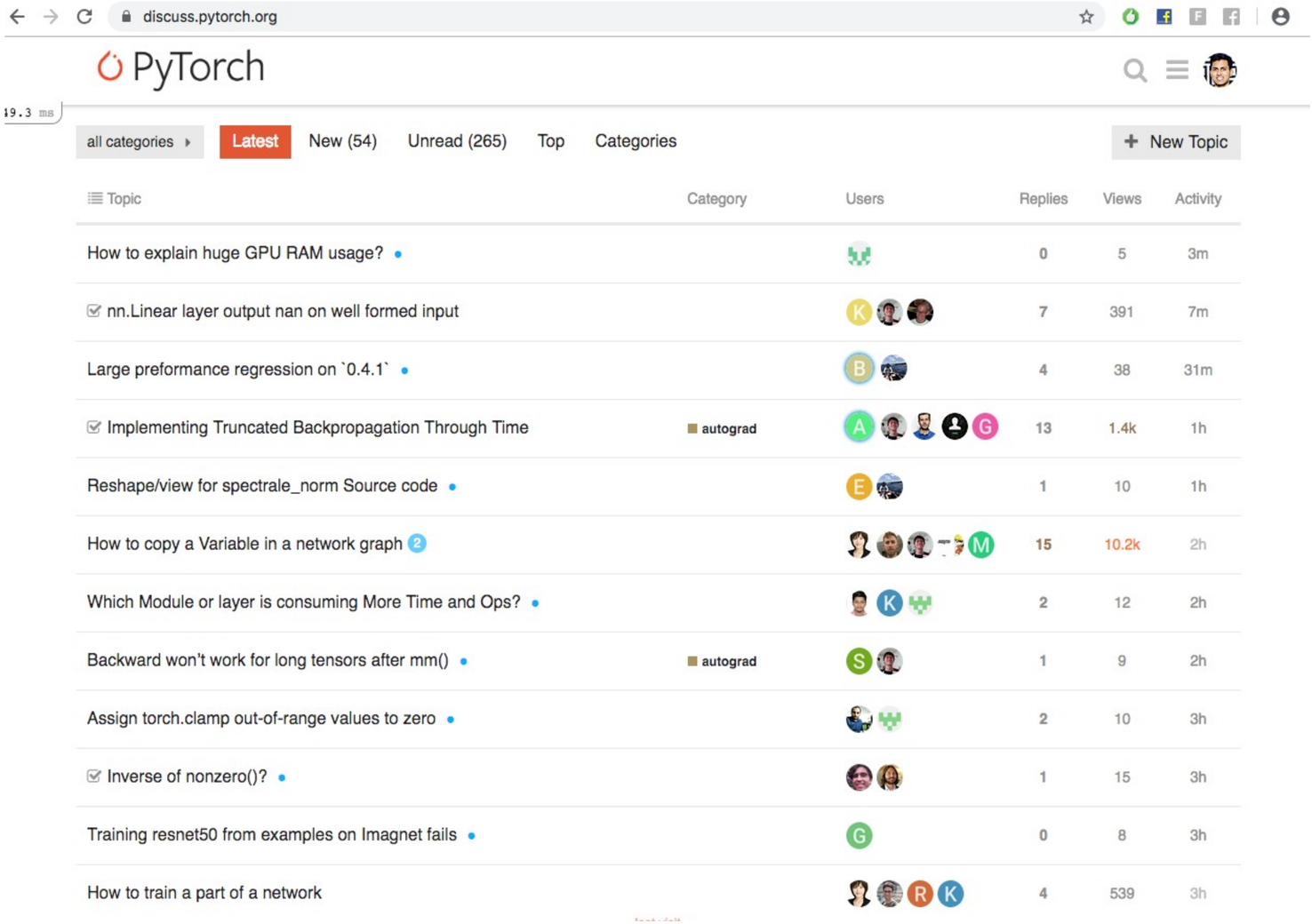
Ecosystem

.FlowNet2: Optical Flow Estimation with Deep Networks

https://github.com/NVIDIA/flownet2-pytorch

Ecosystem

- A strong support system



PyTorch Hub

Discover and publish models to a pre-trained model repository designed for both research exploration and development needs.

PyTorch Hub

Discover and publish models to a pre-trained model repository designed for both research exploration and development needs.

LOADING MODELS

Users can load pre-trained models using `torch.hub.load()` API.

Here's an example showing how to load the `resnet18` entrypoint from the `pytorch/vision` repo.

```
model = torch.hub.load('pytorch/vision',
'resnet18', pretrained=True)
```

PyTorch Hub

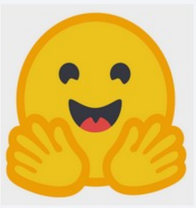
Discover and publish models to a pre-trained model repository designed for both research exploration and development needs.

Deeplabv3-ResNet101

DeepLabV3 model with a ResNet-101 backbone

BERT

Bidirectional Encoder Representations from Transformers.



WaveGlow

WaveGlow model for generating speech from mel spectrograms (generated by Tacotron2)

ResNext WSL

ResNext models trained with billion scale weakly-supervised data.

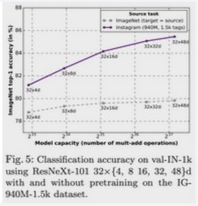


Fig. 5: Classification accuracy on val-IN-1k using ResNeXt-101 32×{4, 8 16, 32, 48}d with and without pretraining on the IG-940M-1.5k dataset.

DCGAN on FashionGen

A simple generative image model for 64x64 images

Progressive Growing of GANs (PGAN)

High-quality image generation of fashion, celebrity faces

PyTorch Hub

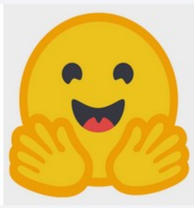
Discover and publish models to a pre-trained model repository designed for both research exploration and development needs.

Deeplabv3-ResNet101

DeepLabV3 model with a ResNet-101 backbone

BERT

Bidirectional Encoder Representations from Transformers.



WaveGlow

WaveGlow model for generating speech from mel spectrograms (generated by Tacotron2)

GPT

Generative Pre-Training (GPT) models for language understanding

GPT-2

Language Models are Unsupervised Multitask Learners

DCGAN on FashionGen

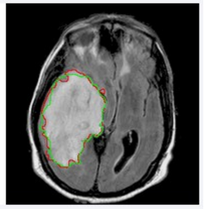
A simple generative image model for 64x64 images

Transformer-XL

Attentive Language Models Beyond a Fixed-Length Context

U-Net for brain MRI

U-Net with batch normalization for biomedical image segmentation with pretrained weights for abnormality segmentation in brain MRI



SSD

Single Shot MultiBox Detector model for object detection

Tacotron 2

The Tacotron 2 model for generating mel spectrograms from text

PyTorch Hub

Discover and publish models to a pre-trained model repository designed for both research exploration and development needs.

Deeplabv3-ResNet101

DeepLabV3 model with a ResNet-101 backbone

BERT

Bidirectional Encoder Representations from Transformers.

WaveGlow

WaveGlow model for generating speech from mel spectrograms (generated by Tacotron2)

GPT

Generative Pre-Training (GPT) models for language understanding

GPT-2

Language Models are Unsupervised Multitask Learners

DCGAN on FashionGen

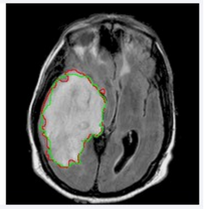
A simple generative image model for 64x64 images

Transformer-XL

Attentive Language Models Beyond a Fixed-Length Context

U-Net for brain MRI

U-Net with batch normalization for biomedical image segmentation with pretrained weights for abnormality segmentation in brain MRI



SSD

Single Shot MultiBox Detector model for object detection

Tacotron 2

The Tacotron 2 model for generating mel spectrograms from text

https://pytorch.org/hub

https://pytorch.org

facebook

NVIDIA

salesforce

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIE
PARIS INSTITUTE OF TECHNOLOGY

Carnegie Mellon University

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

Digital Reasoning

Stanford University

UNIVERSITY OF OXFORD

NYU

Inria

ENS
ÉCOLE NORMALE SUPÉRIEURE

EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Berkeley
UNIVERSITY OF CALIFORNIA

UBER