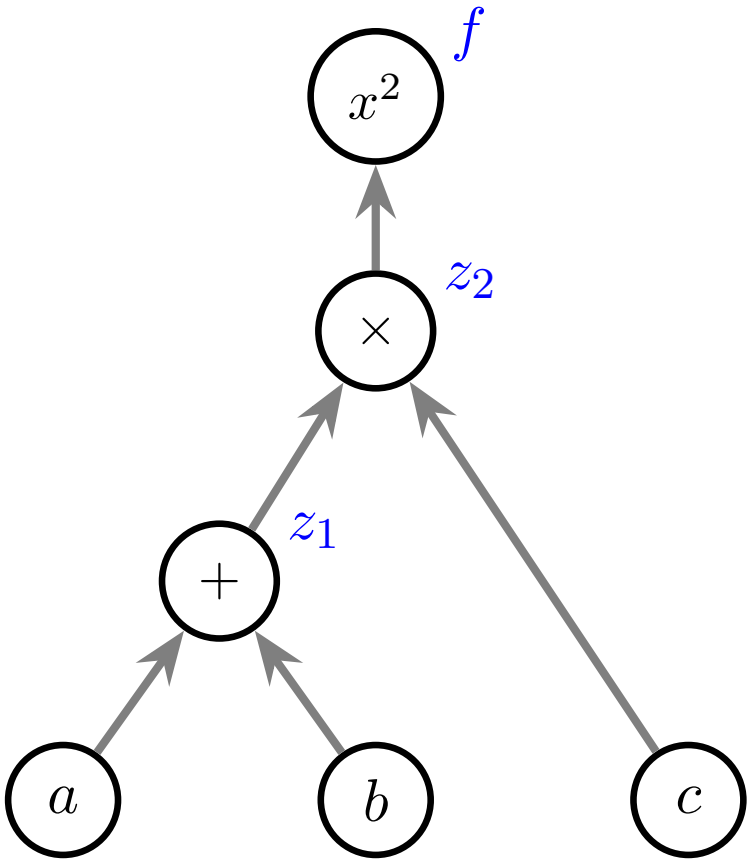
Topic 13: DIFFERENTIABLE COMPUTING

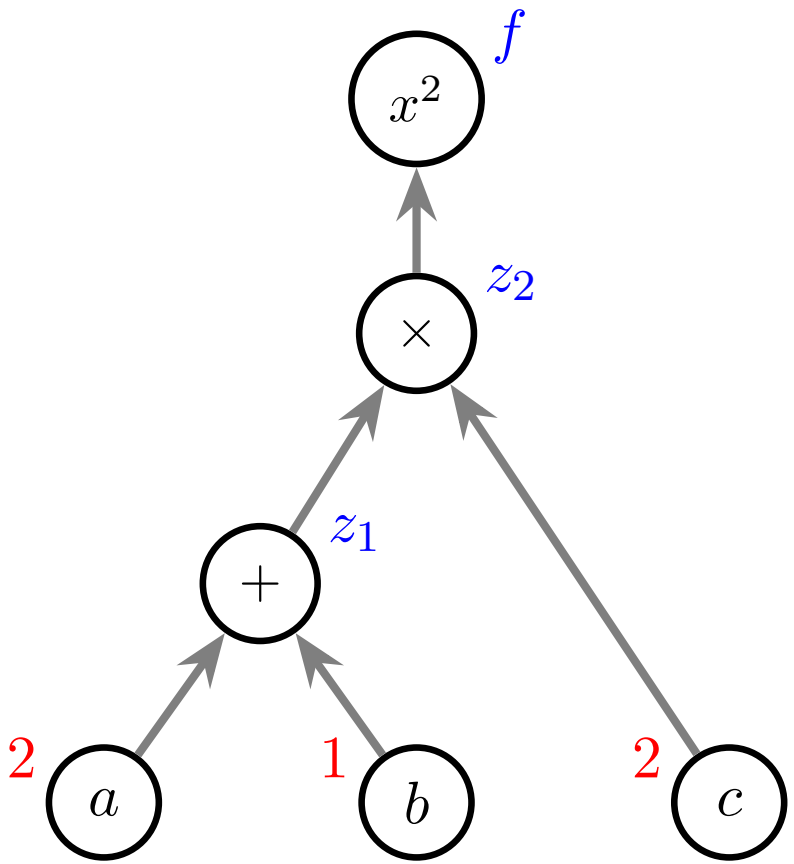
CMSC 35400/STAT 37710 Machine Learning
Risi Kondor, The University of Chicago



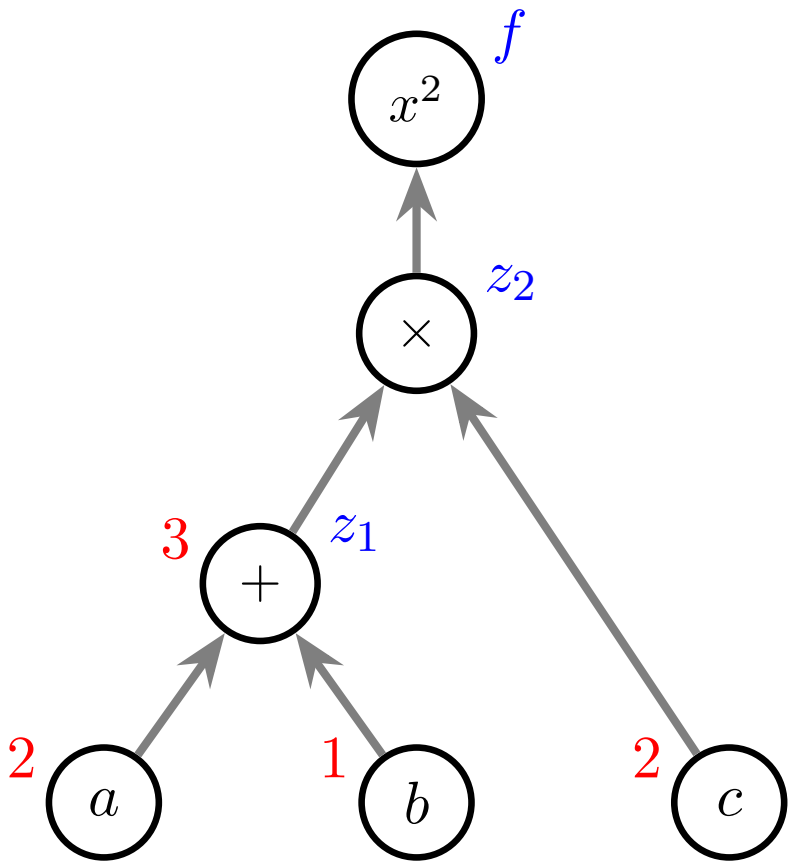
$$f(a, b, c) = ((a + b)c)^2$$

Forward and backward computations

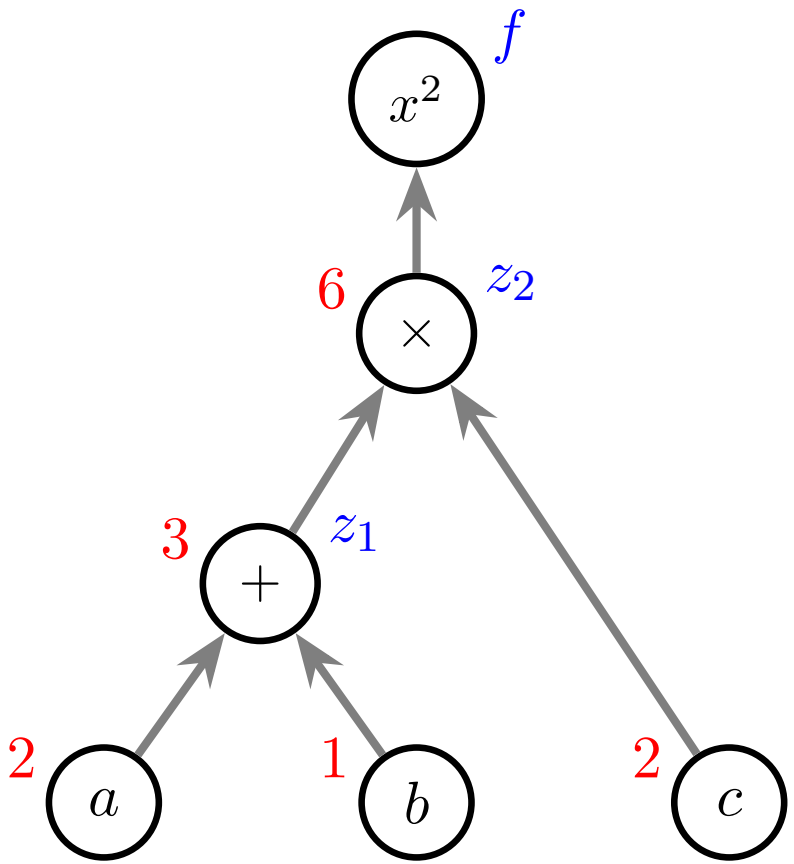
Forward



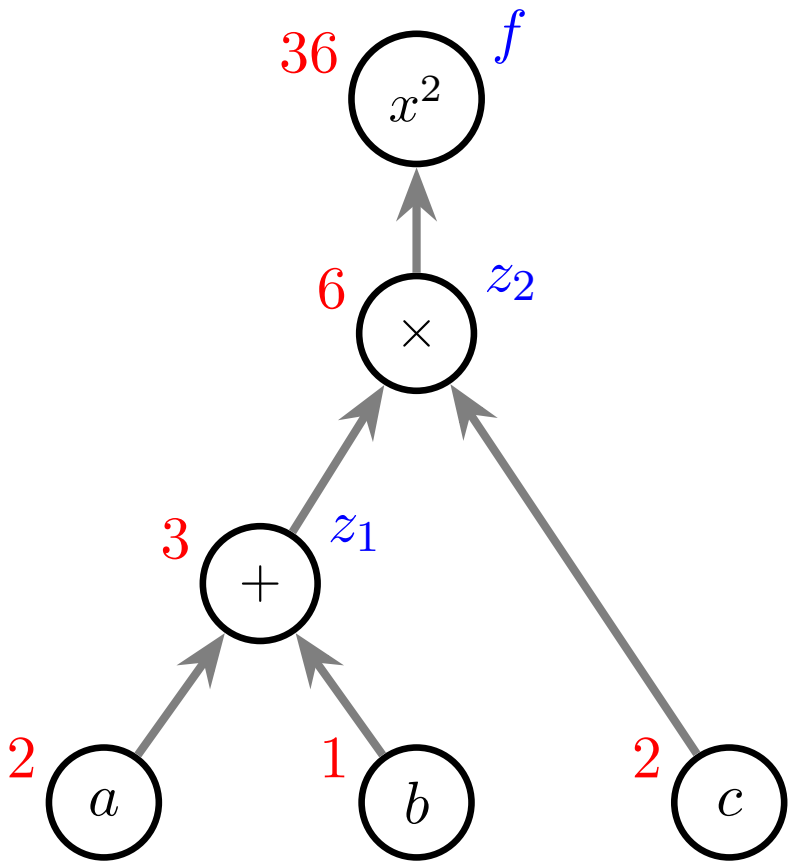
Forward

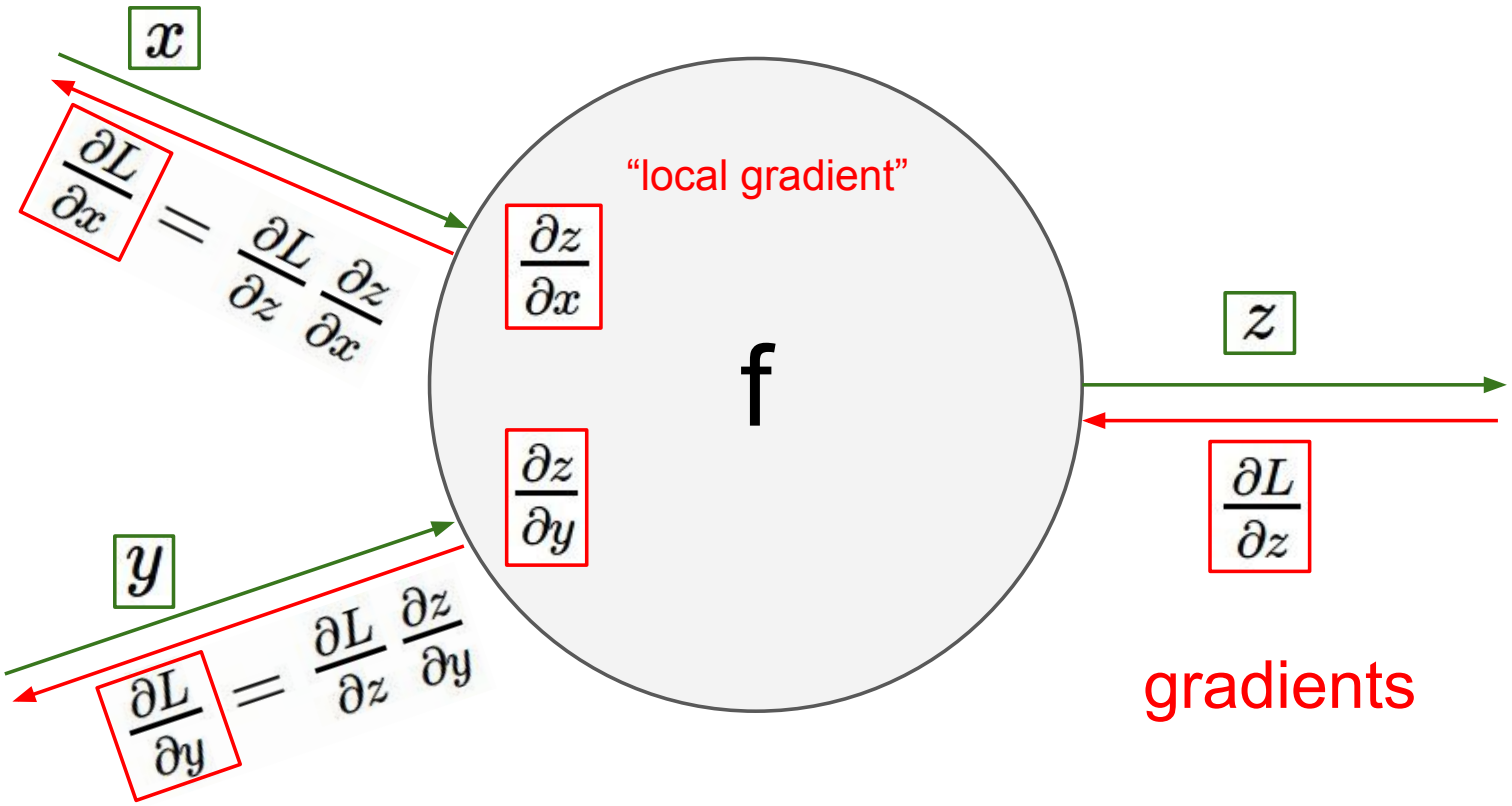


Forward

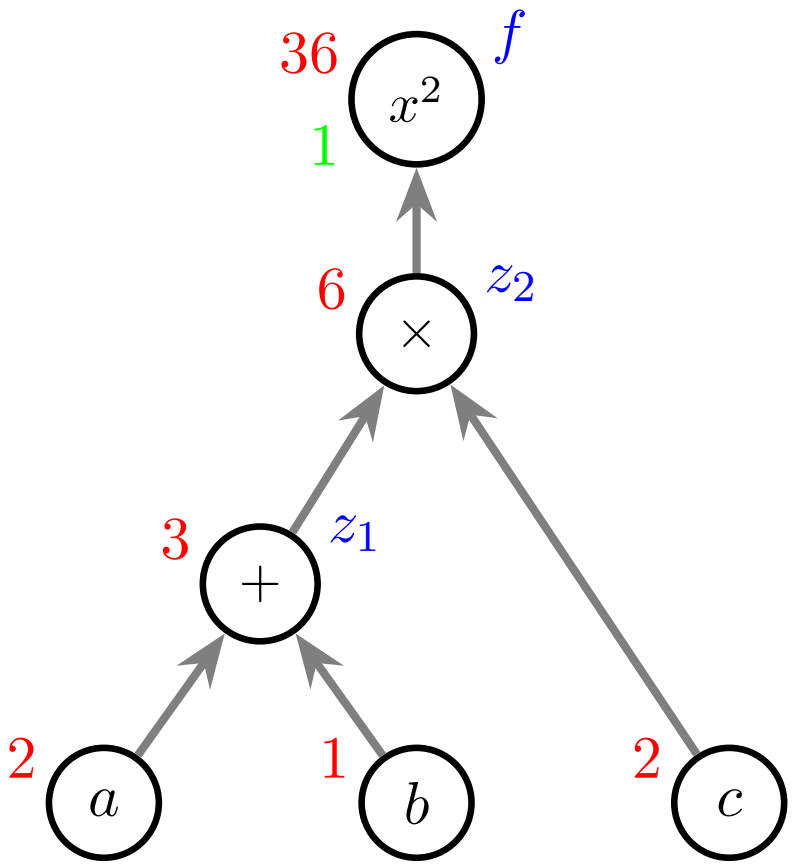


Forward



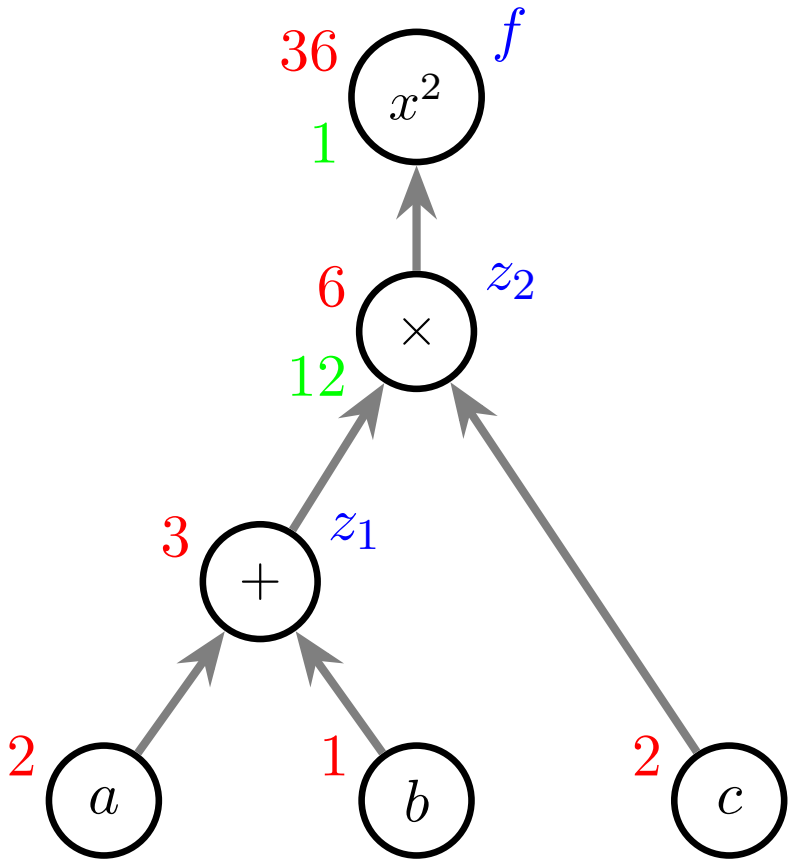

x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
z
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients

Backward



$$\frac{\partial f}{\partial f} = 1$$

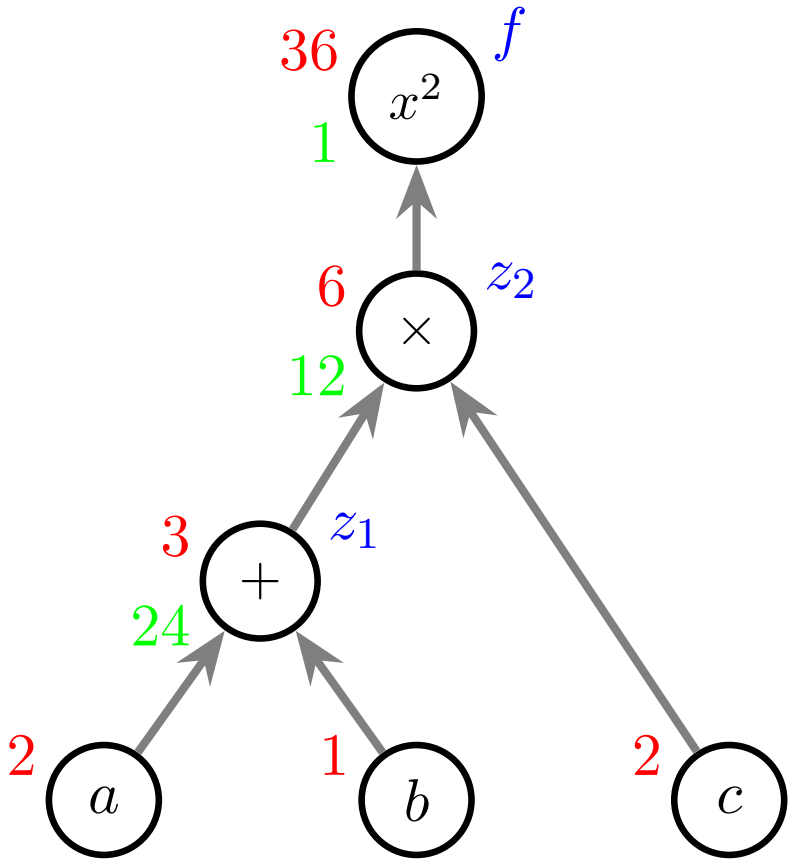
Backward



$$f = z_2^2$$

$$\frac{\partial f}{\partial z_2} = \underbrace{\frac{\partial f}{\partial f}}_{1} \underbrace{\frac{\partial f}{\partial z_2}}_{2z_2=12} = 12$$

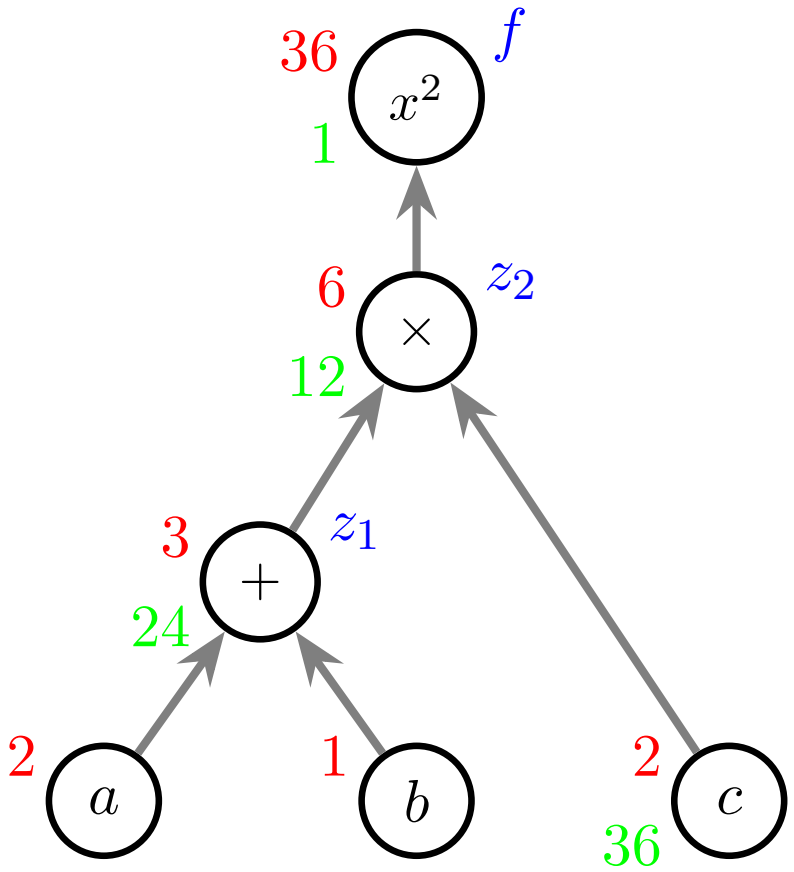
Backward



$$z_2 = z_1 \cdot c$$

$$\frac{\partial f}{\partial z_1} = \underbrace{\frac{\partial f}{\partial z_2}}_{12} \underbrace{\frac{\partial z_2}{\partial z_1}}_{c} = 24$$

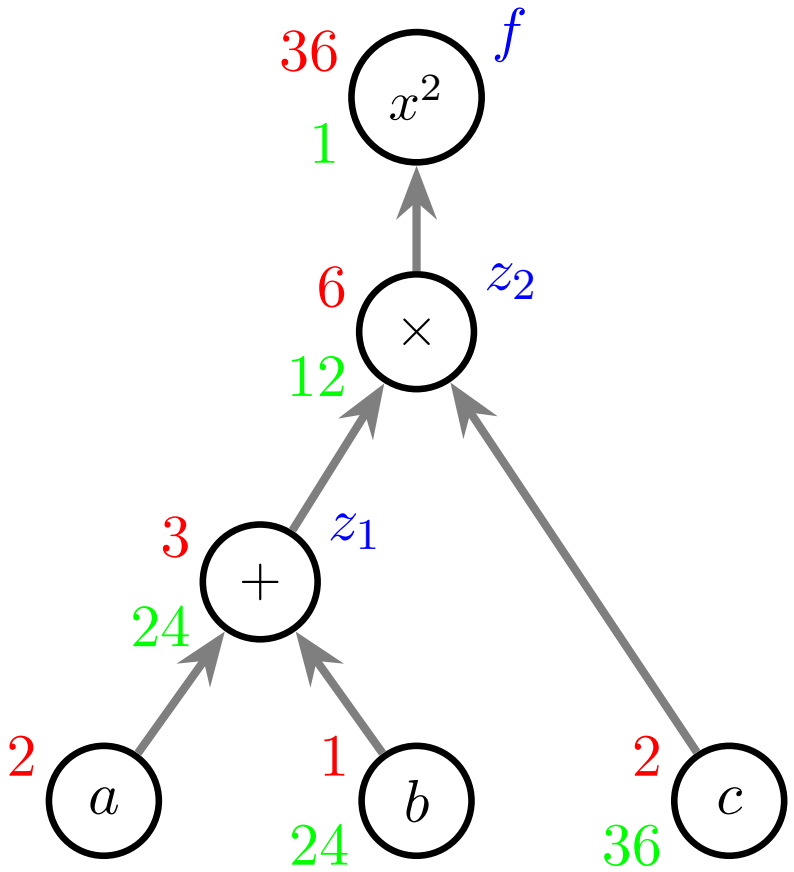
Backward



$$z_2 = z_1 \cdot c$$

$$\frac{\partial f}{\partial c} = \underbrace{\frac{\partial f}{\partial z_2}}_{12} \underbrace{\frac{\partial z_2}{\partial c}}_{z_1} = 36$$

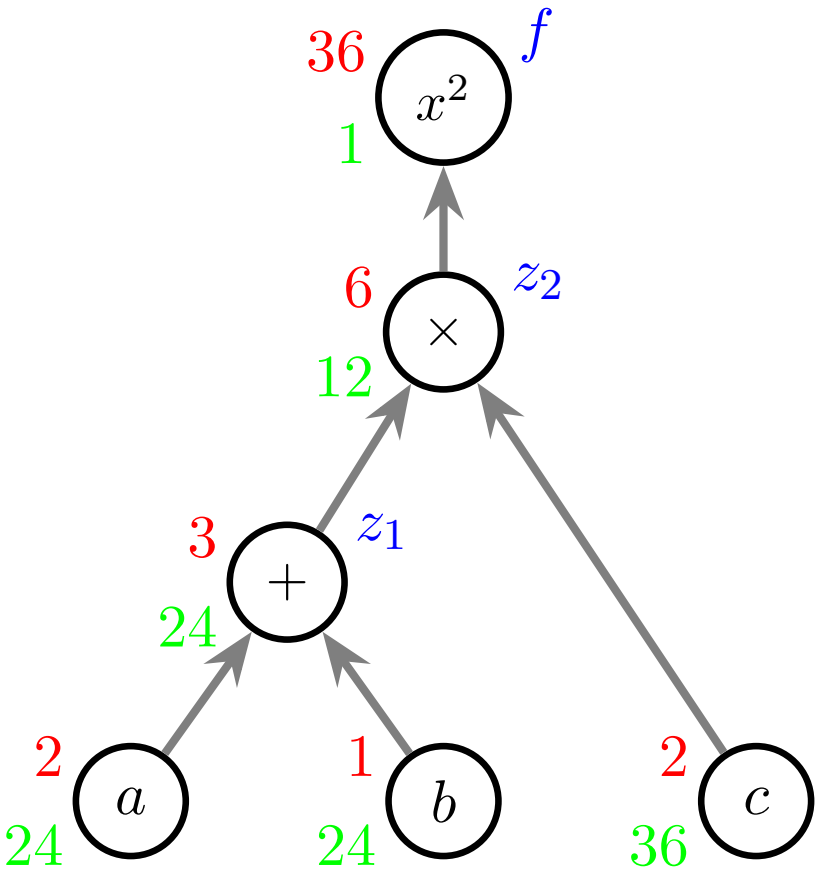
Backward



$$z_1 = a + b$$

$$\frac{\partial f}{\partial b} = \underbrace{\frac{\partial f}{\partial z_1}}_{24} \underbrace{\frac{\partial z_1}{\partial b}}_{1} = 24$$

Backward

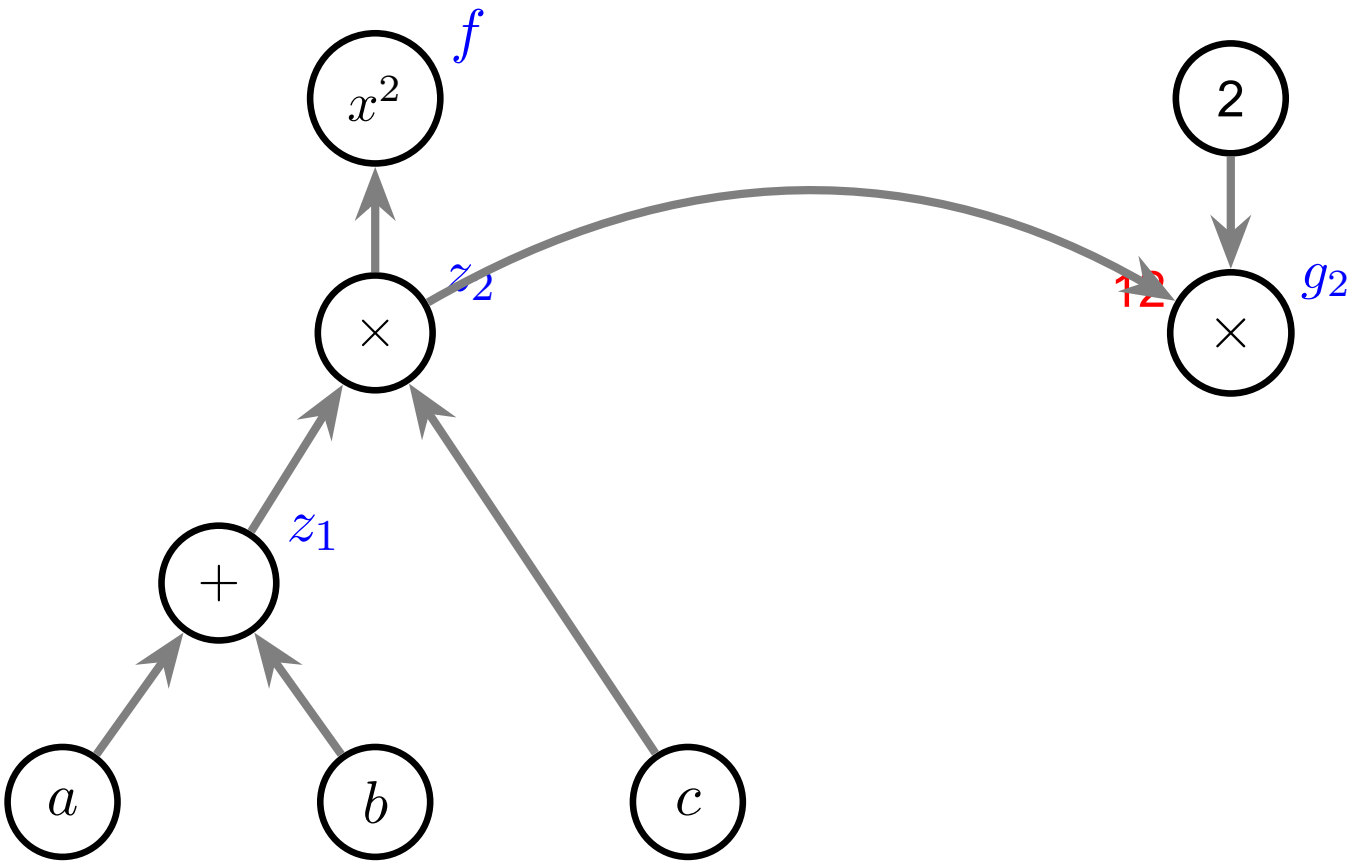


$$z_1 = a + b$$

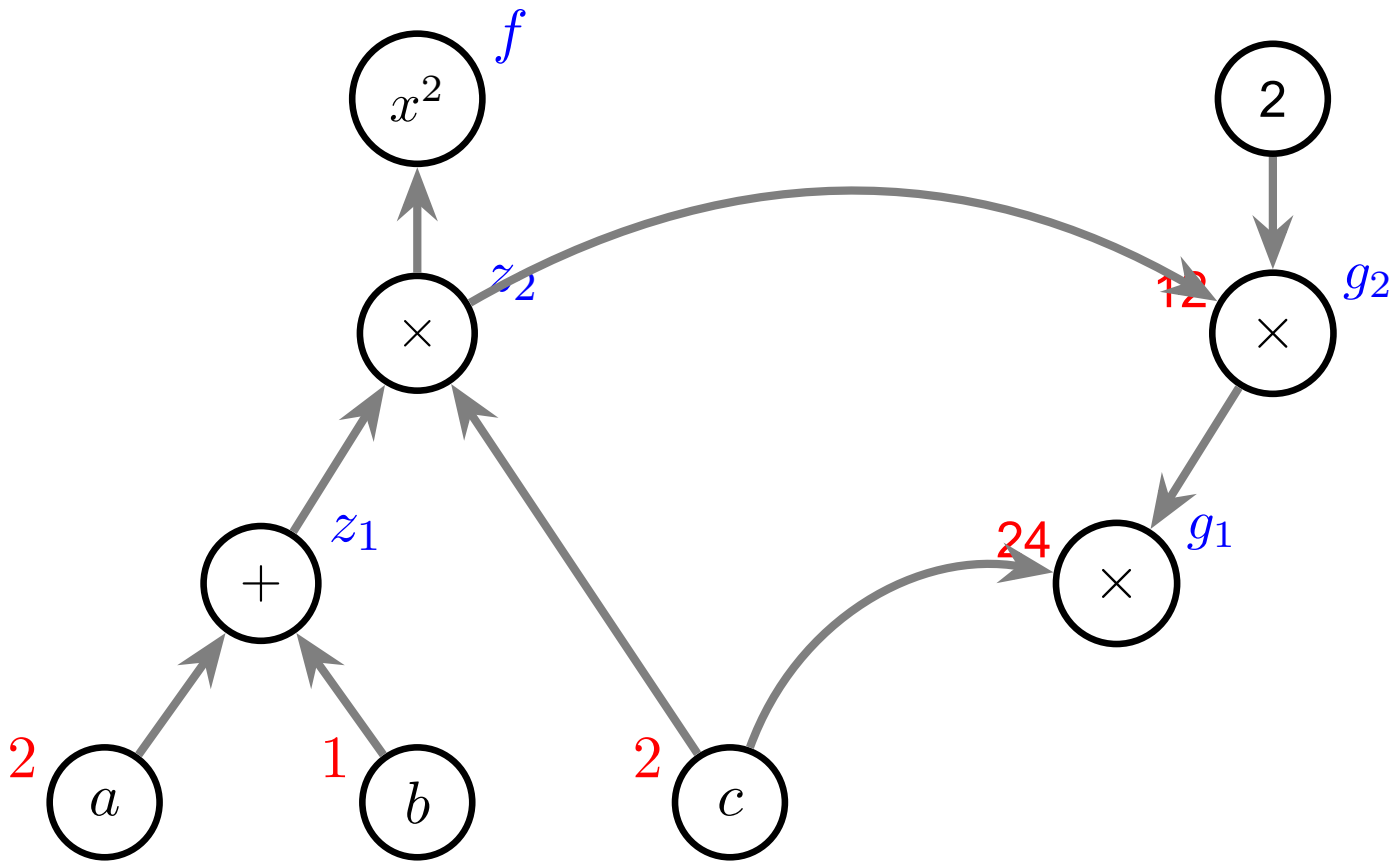
$$\frac{\partial f}{\partial a} = \underbrace{\frac{\partial f}{\partial z_1}}_{24} \underbrace{\frac{\partial z_1}{\partial a}}_{1} = 24$$

Symbolic differentiation

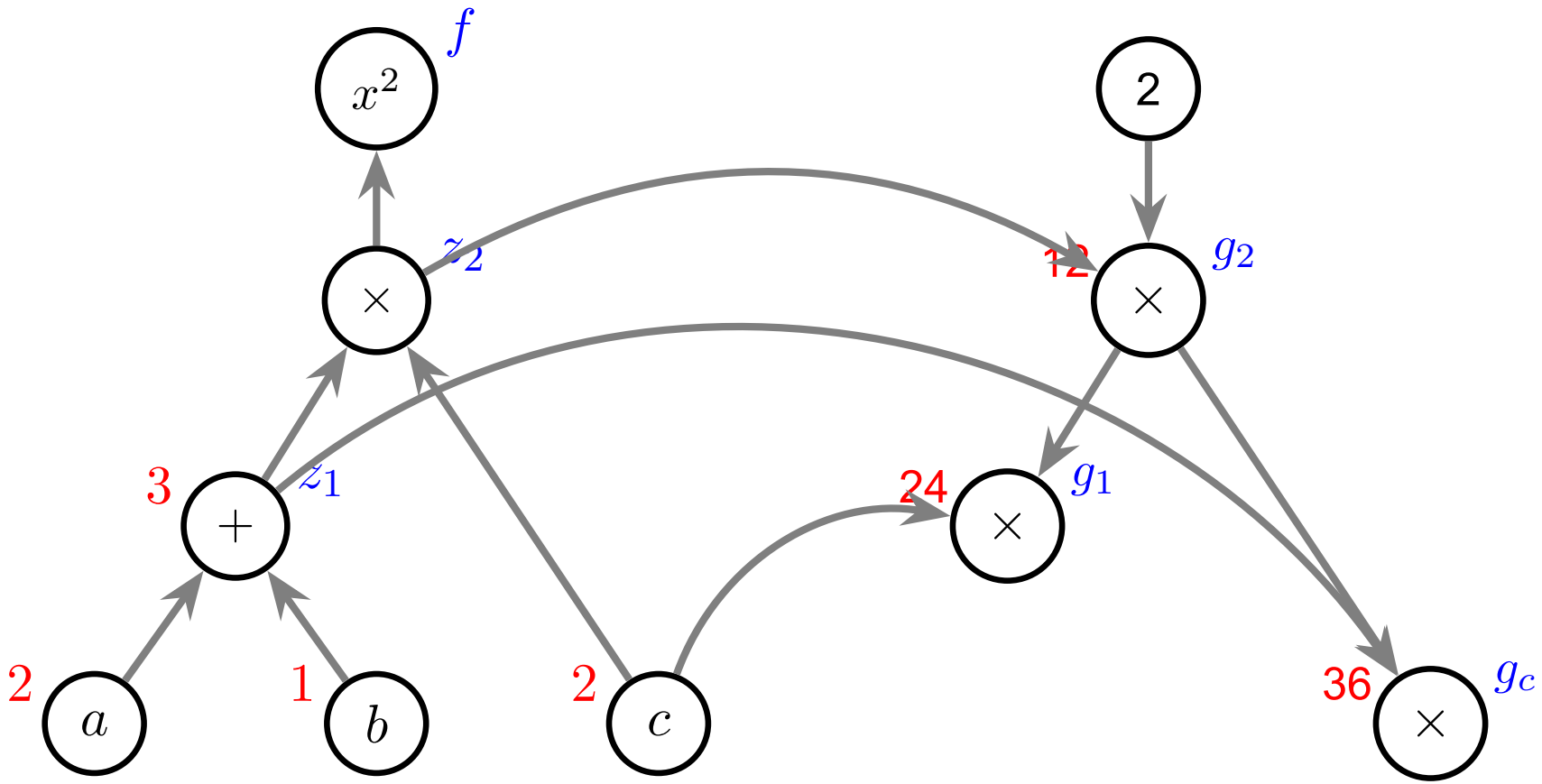
$$g_2 = \frac{\partial f}{\partial z_2} = 2 \cdot z_2$$



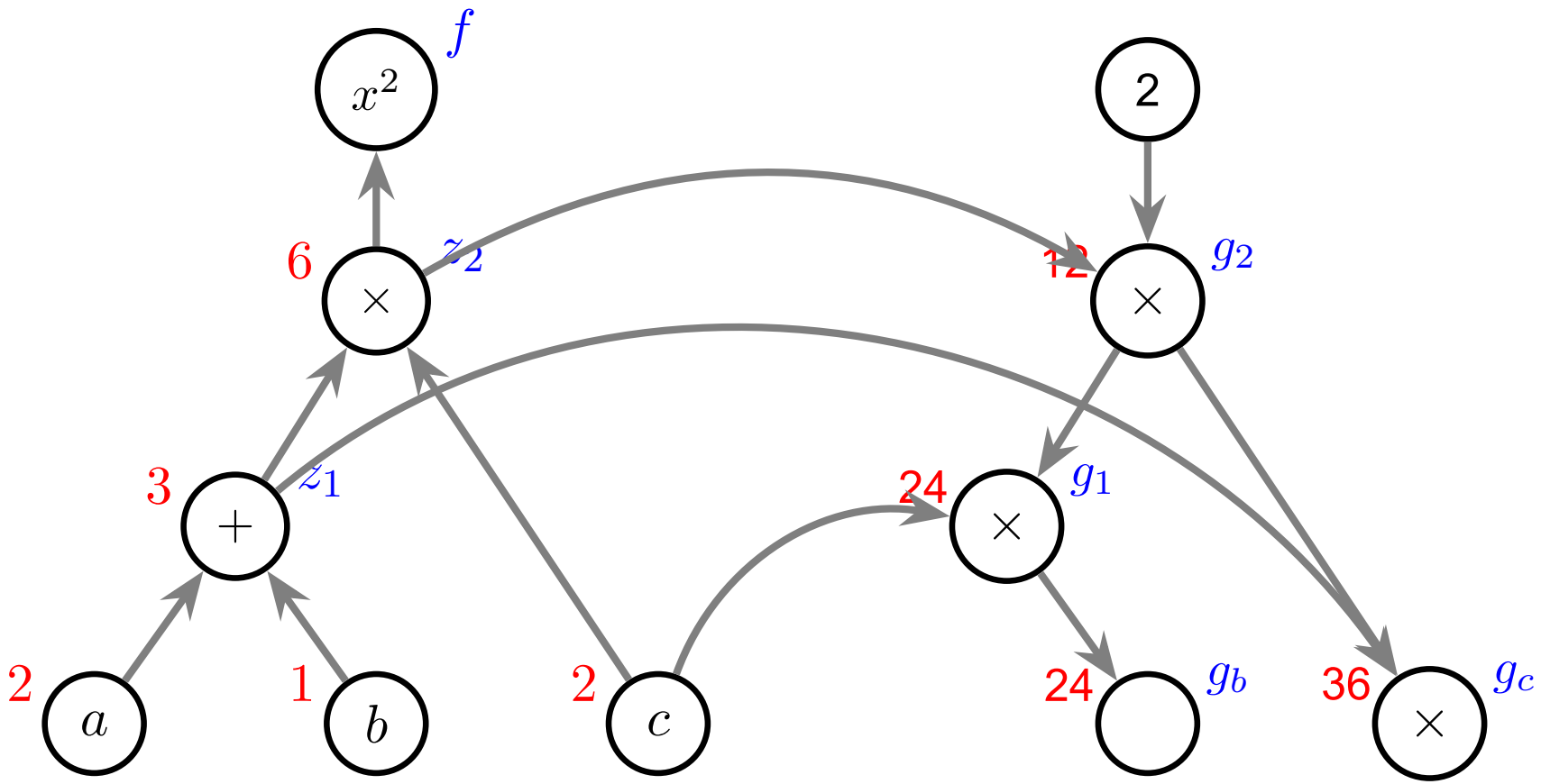
$$g_1 = \frac{\partial f}{\partial z_1} = \frac{\partial f}{\partial z_2} \underbrace{\frac{\partial z_2}{\partial z_1}}_{c}$$



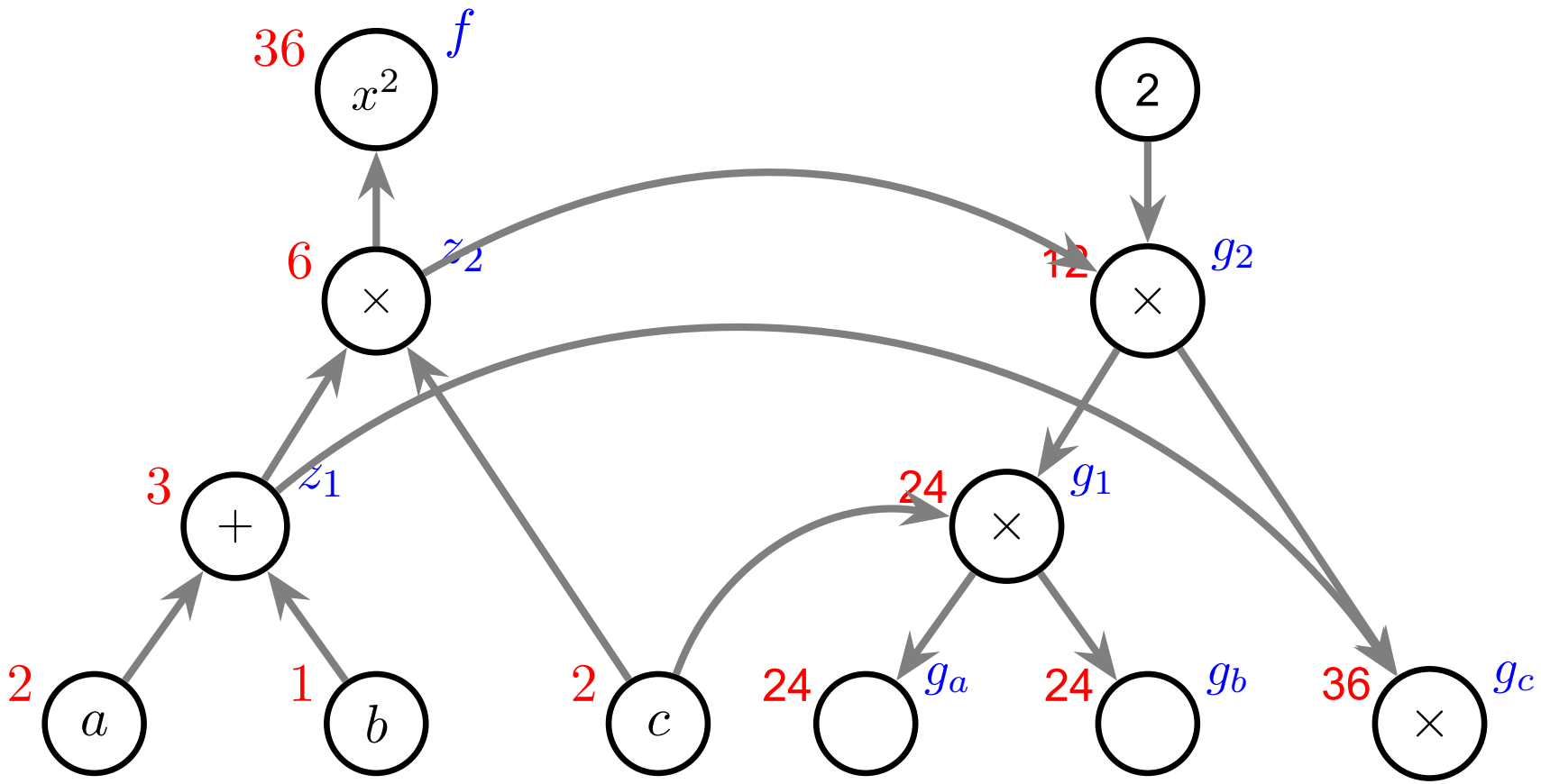
$$g_c = \frac{\partial f}{\partial c} = \frac{\partial f}{\partial z_2} \underbrace{\frac{\partial z_2}{\partial c}}_{z_1}$$



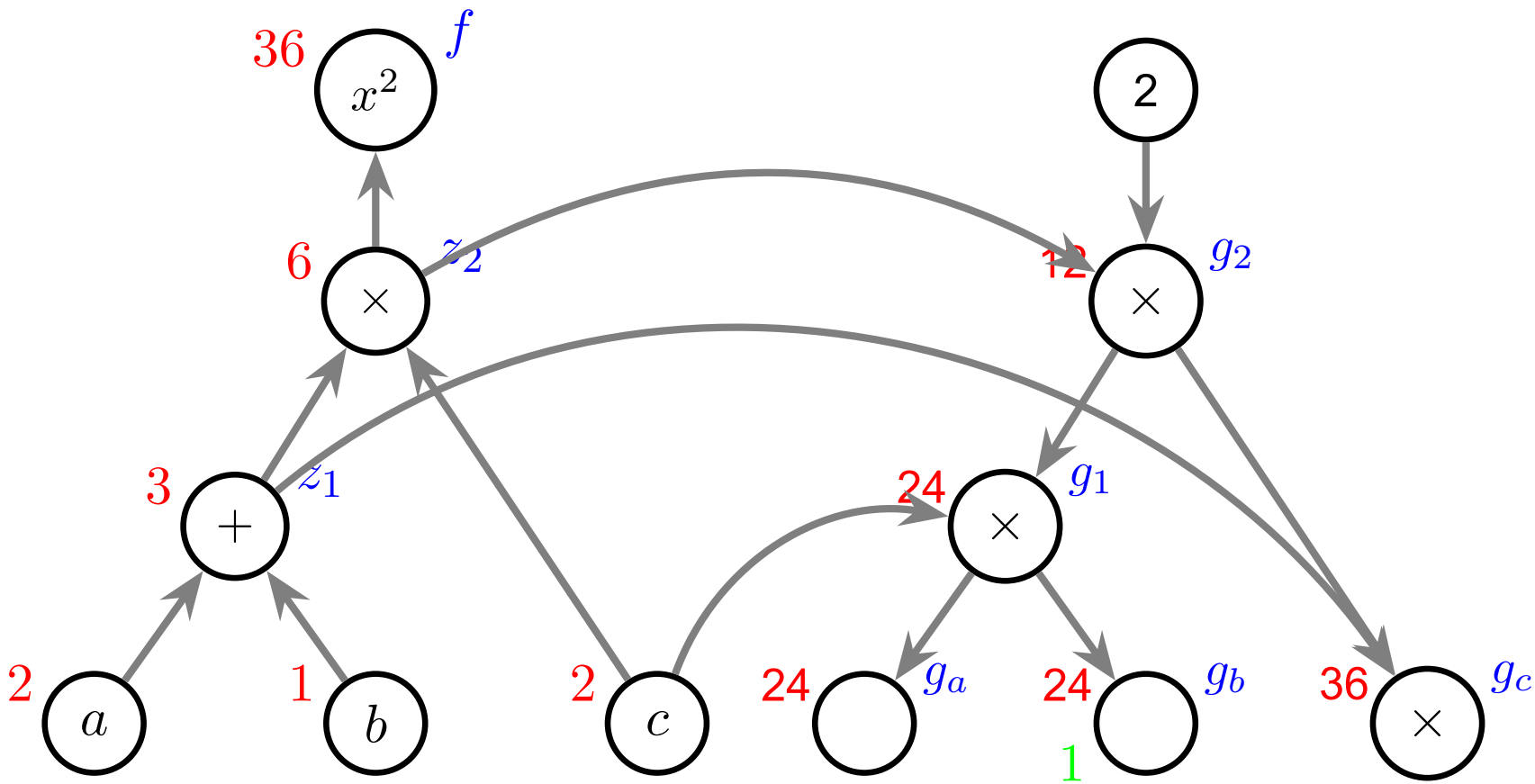
$$g_b = \frac{\partial f}{\partial b} = \frac{\partial f}{\partial z_1} \underbrace{\frac{\partial z_1}{\partial b}}_{1}$$



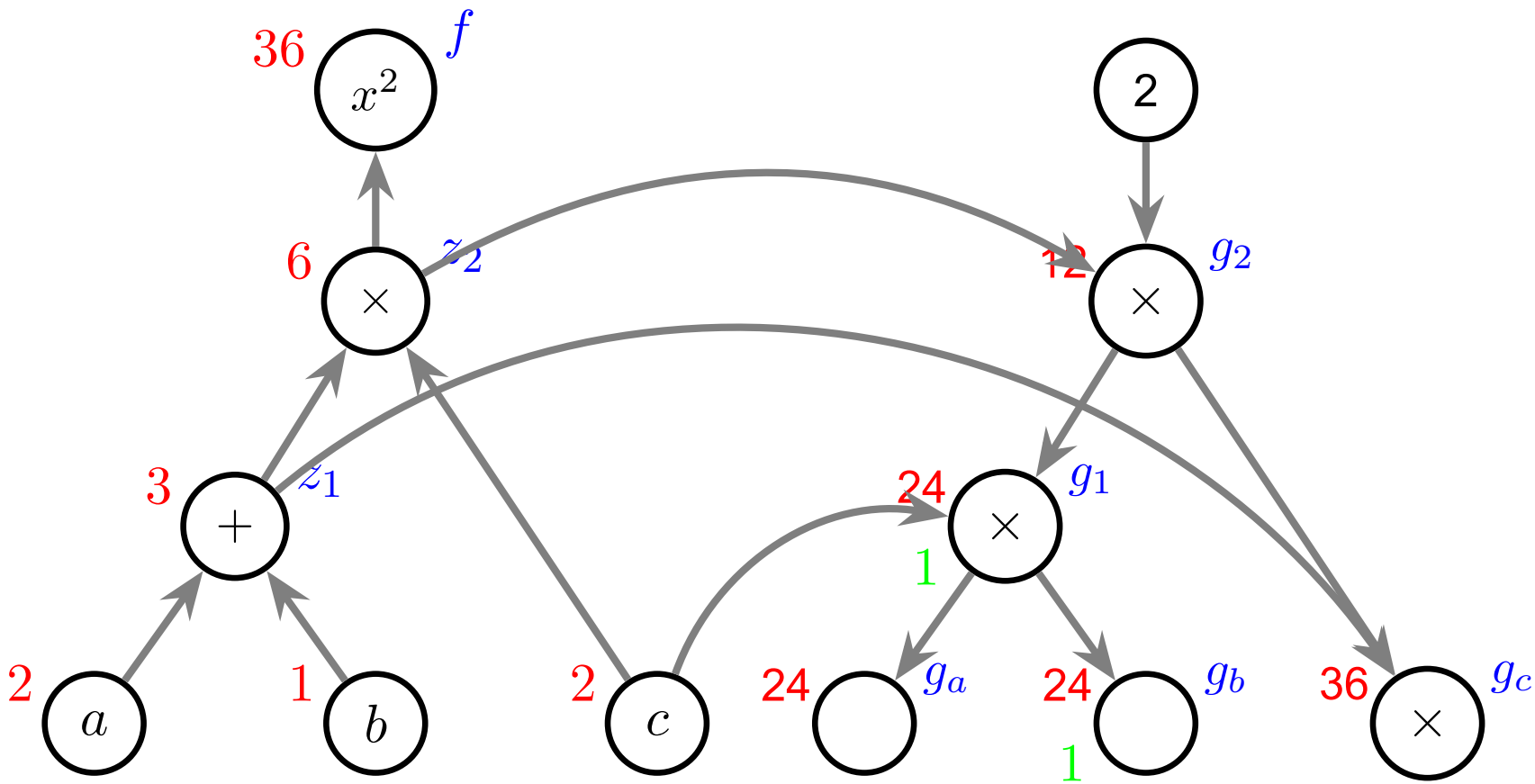
$$g_a = \frac{\partial f}{\partial a} = \frac{\partial f}{\partial z_1} \underbrace{\frac{\partial z_1}{\partial a}}_{1}$$



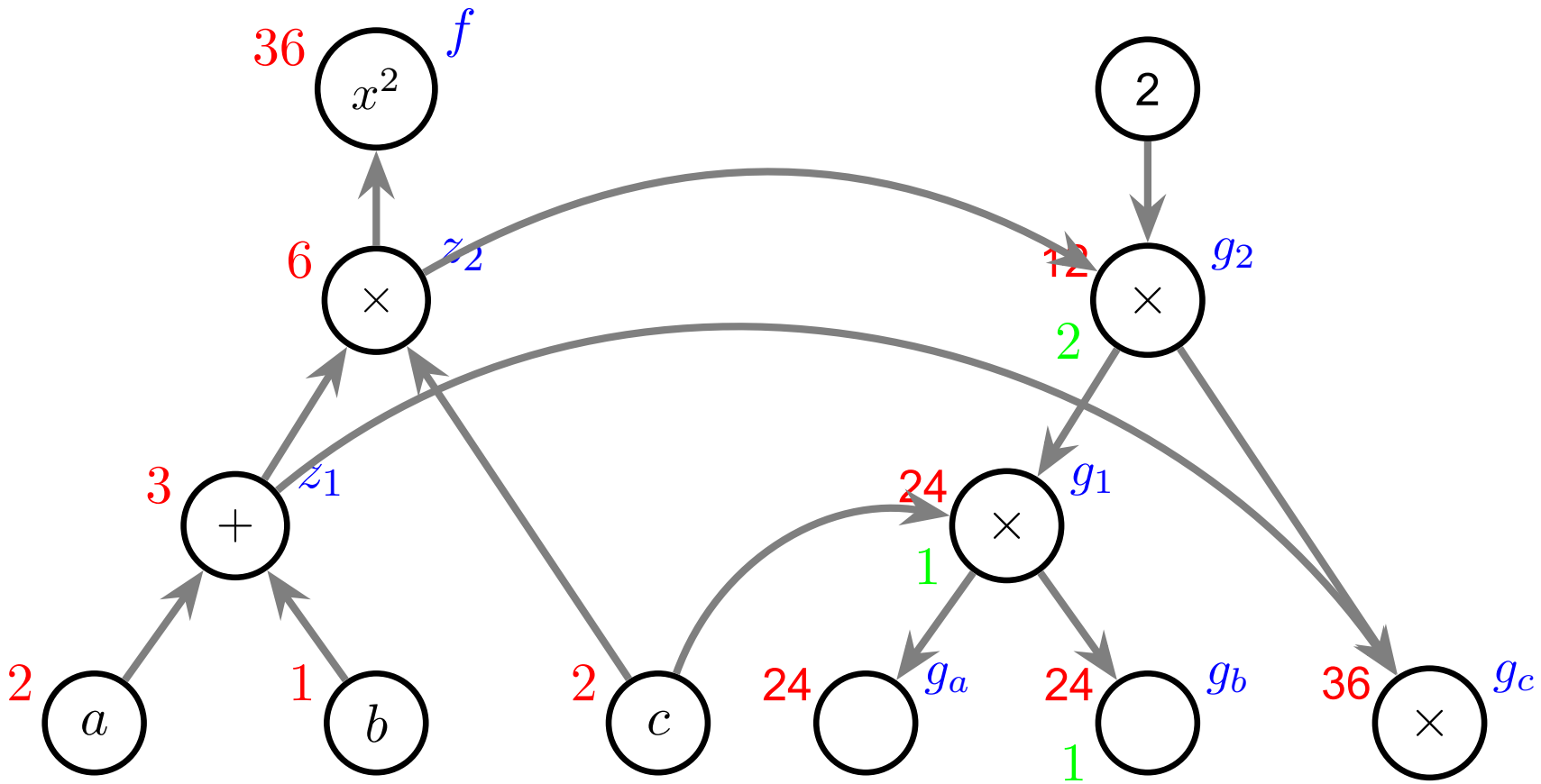
$$\frac{\partial g_b}{\partial g_b} = 1$$



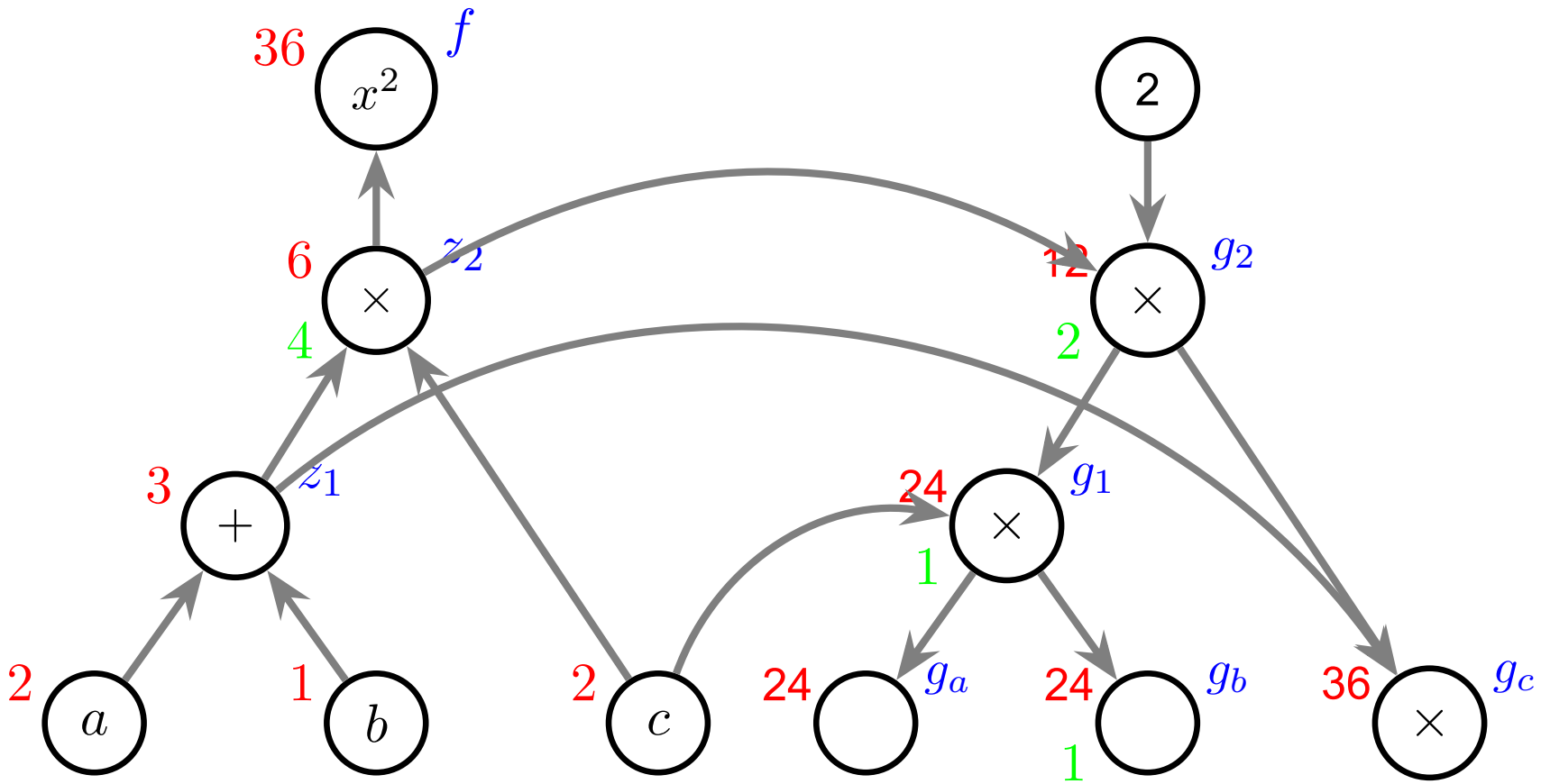
$$\frac{\partial g_b}{\partial g_1} = \frac{\partial g_b}{\partial g_b} = 1$$



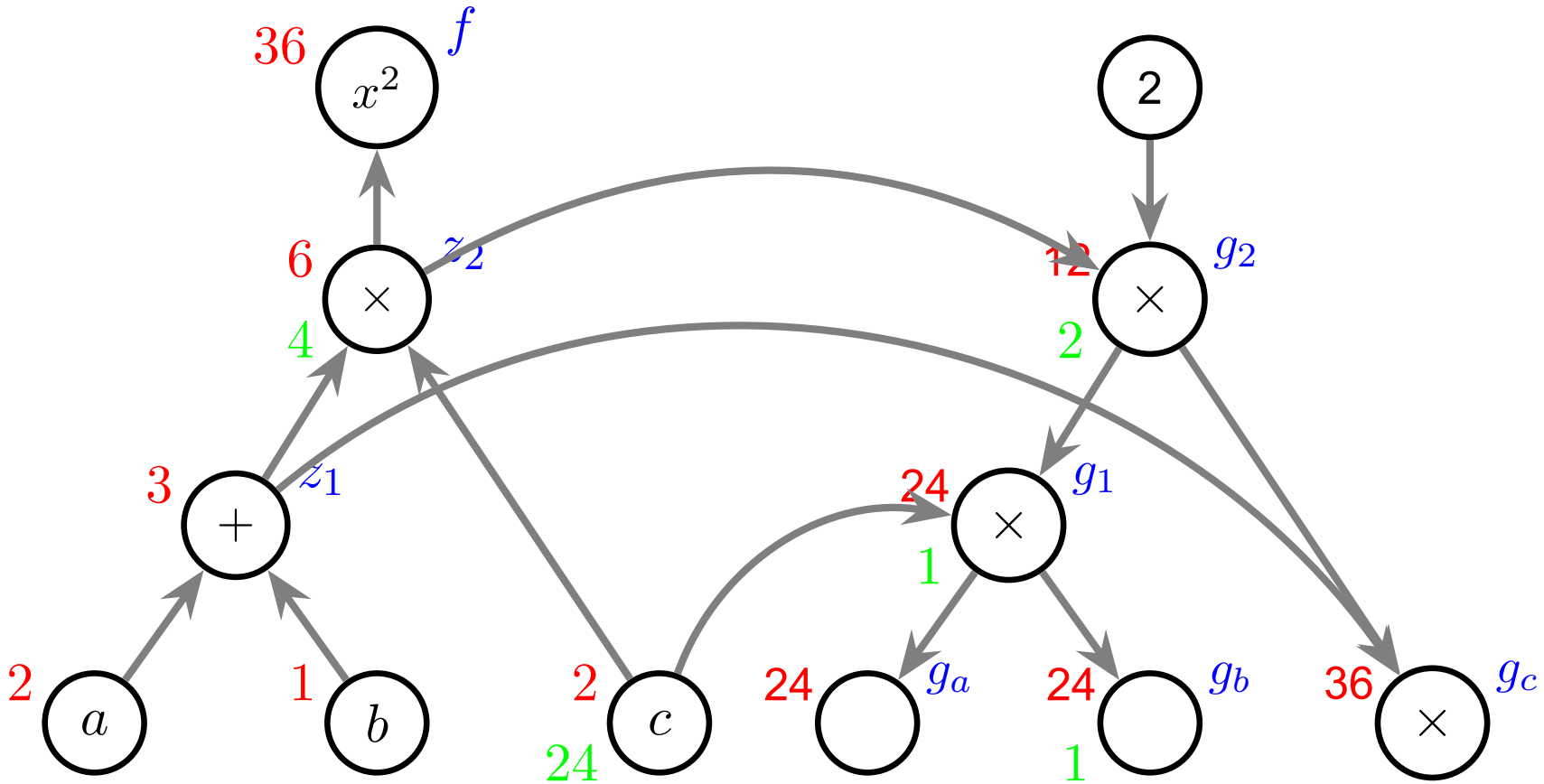
$$\frac{\partial g_b}{\partial g_2} = \underbrace{\frac{\partial g_b}{\partial g_1}}_{1} \underbrace{\frac{\partial g_1}{\partial g_2}}_{c} = 2$$



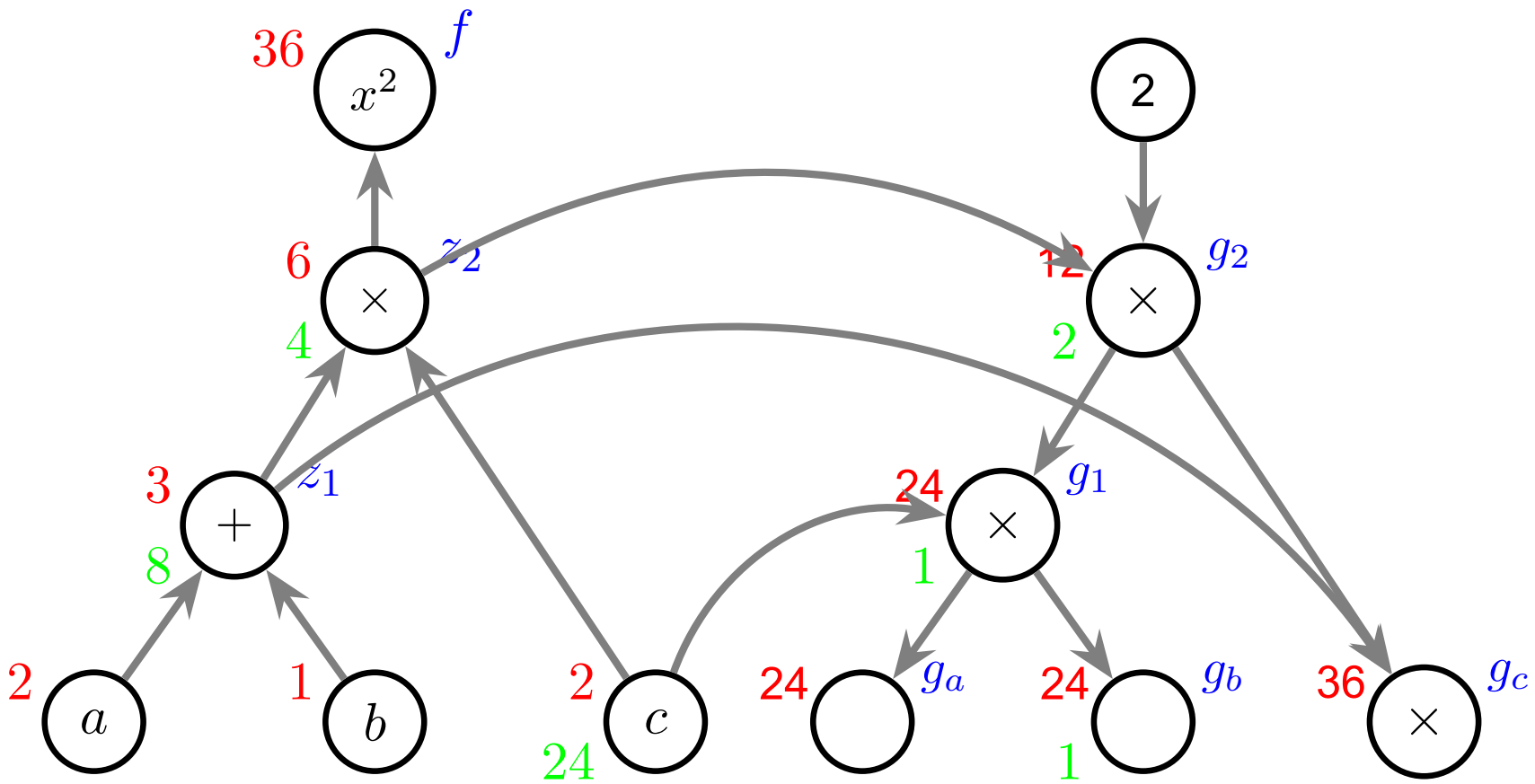
$$\frac{\partial g_b}{\partial z_2} = \underbrace{\frac{\partial g_b}{\partial g_2}}_{2} \underbrace{\frac{\partial g_2}{\partial z_2}}_{2} = 4$$



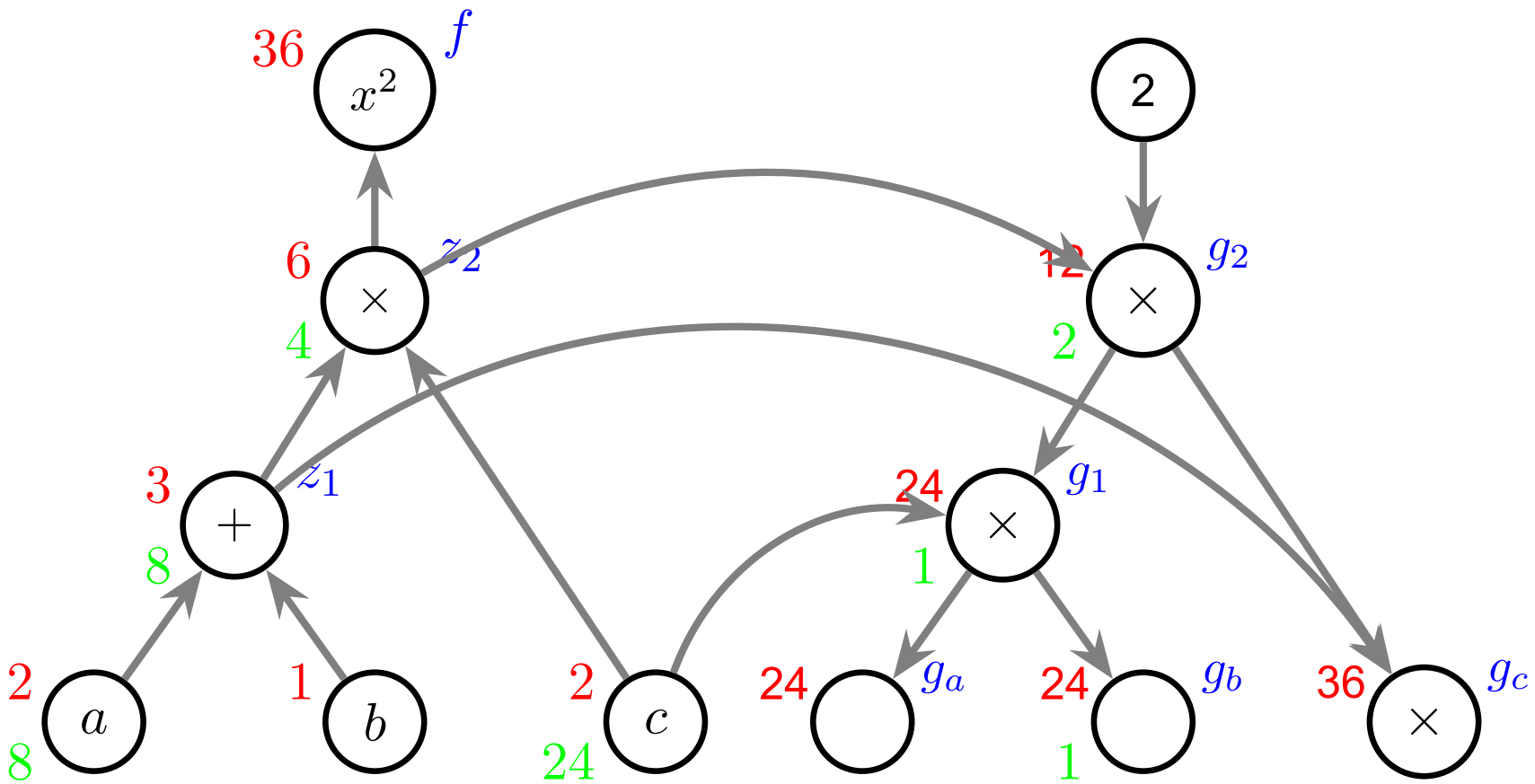
$$\frac{\partial g_b}{\partial c} = \underbrace{\frac{\partial g_b}{\partial z_2}}_{4}\underbrace{\frac{\partial z_2}{\partial c}}_{z_1} + \underbrace{\frac{\partial g_b}{\partial g_1}}_{1}\underbrace{\frac{\partial g_1}{\partial c}}_{g_2} = 4{\cdot}3{+}1{\cdot}12 = 24$$



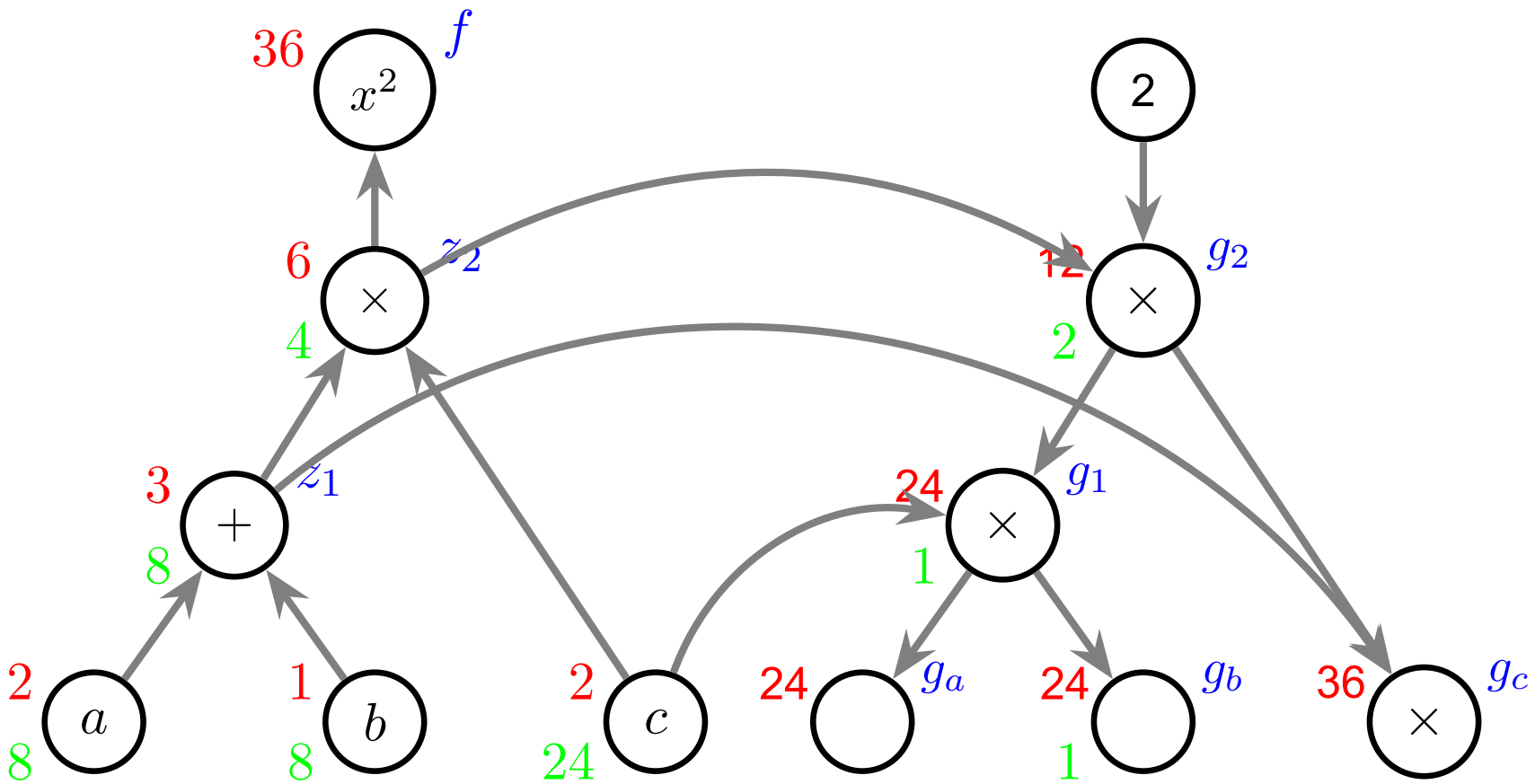
$$\frac{\partial g_b}{\partial z_1} = \underbrace{\frac{\partial g_b}{\partial z_2}}_{4} \underbrace{\frac{\partial z_2}{\partial z_1}}_{c} = 8$$



$$\frac{\partial g_b}{\partial a} = \underbrace{\frac{\partial g_b}{\partial z_1}}_{8} \underbrace{\frac{\partial z_1}{\partial a}}_{1} = 8$$



$$\frac{\partial g_b}{\partial b} = \underbrace{\frac{\partial g_b}{\partial z_1}}_{8} \underbrace{\frac{\partial z_1}{\partial b}}_{1} = 8$$



Verification

$$f = ((a+b)c)^2$$

$$\frac{\partial f}{\partial b} = 2(a+b)c^2$$

$$\frac{\partial f}{\partial a \partial b} = 2c^2 = 8$$

$$\frac{\partial f}{\partial a \partial c} = 2(a+b)c = 12$$

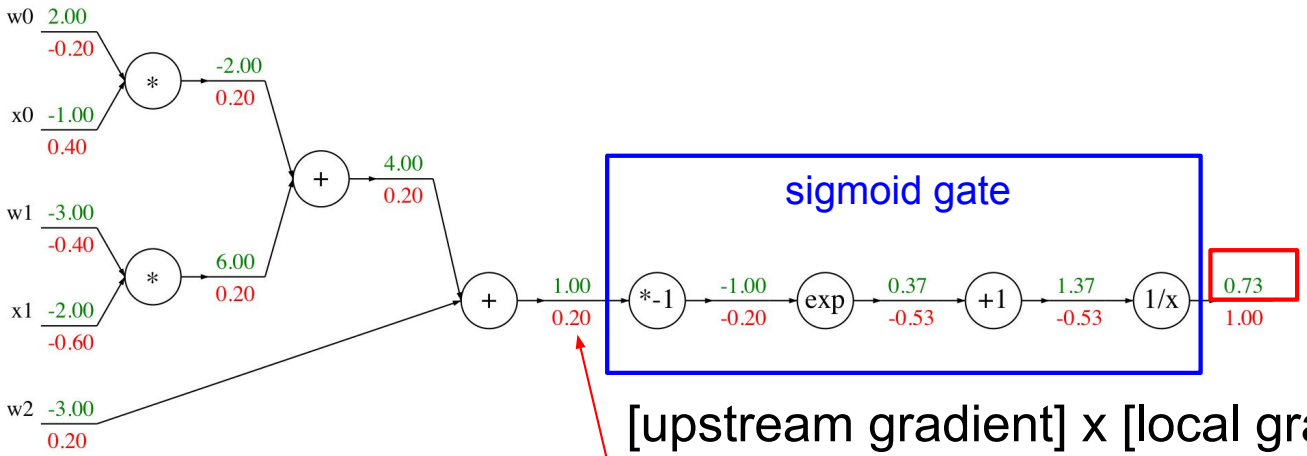
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

sigmoid function

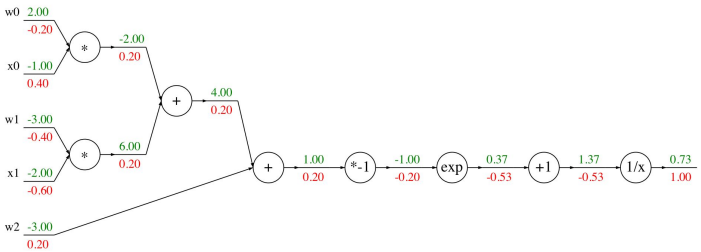
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



[upstream gradient] x [local gradient]

[1.00] x [(1 - 0.73) (0.73)]= 0.2

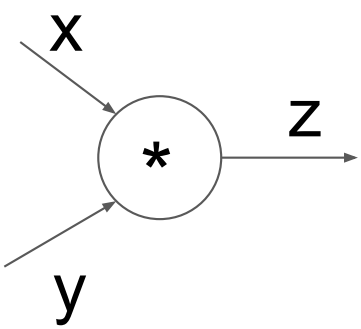
Modularized implementation: forward / backward API



Graph (or Net) object *(rough pseudo code)*

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Modularized implementation: forward / backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Local gradient

Upstream gradient variable

Implementations

Spot the GPUs!

(graphics processing unit)

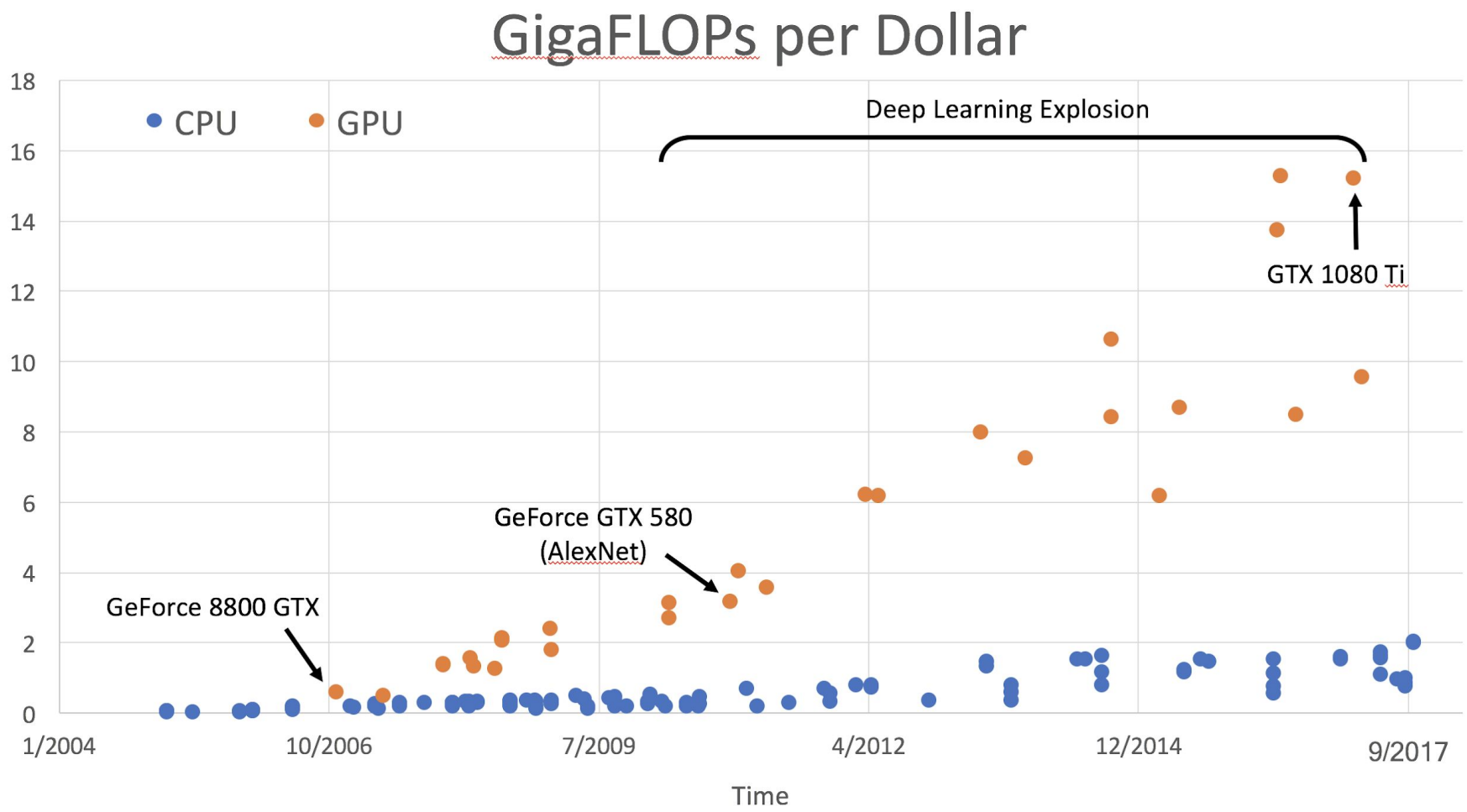
This image is in the public domain

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 TFLOPs FP32

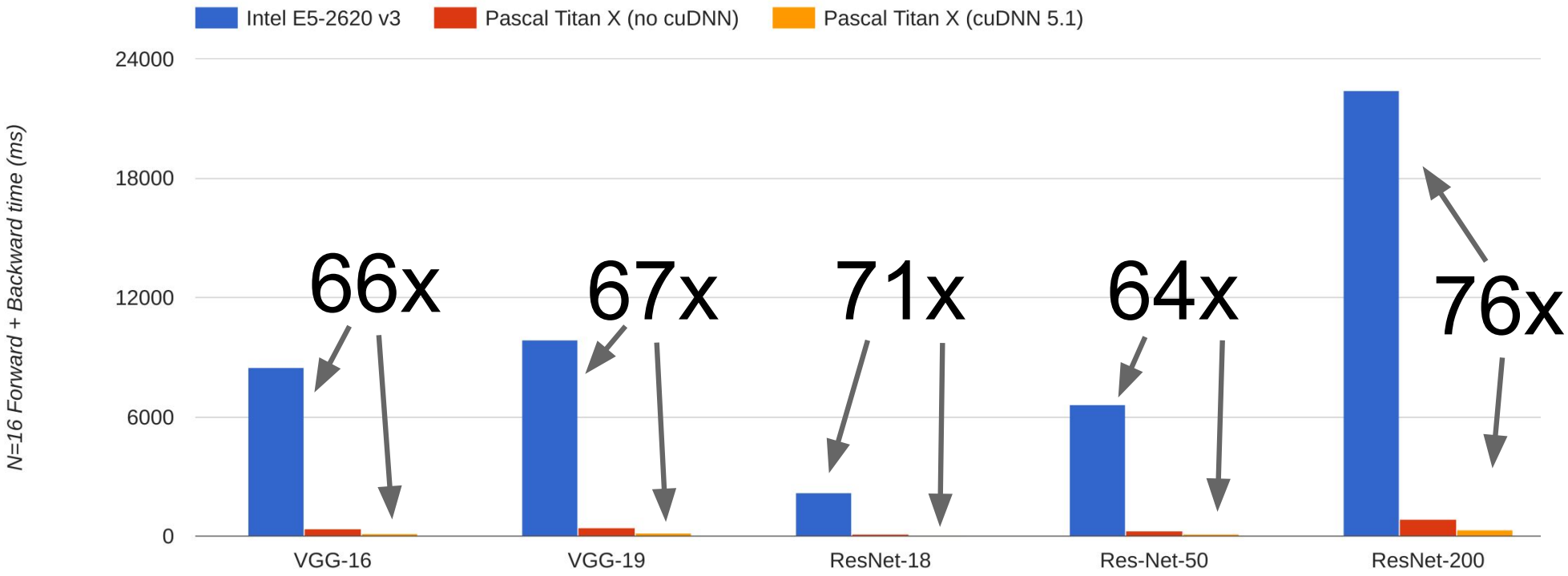
CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks


GigaFLOPs per Dollar
CPU
GPU
Deep Learning Explosion
GTX 1080 Ti
GeForce GTX 580
(AlexNet)
GeForce 8800 GTX
18
16
14
12
10
8
6
4
2
0
1/2004
10/2006
7/2009
4/2012
12/2014
9/2017
Time

CPU vs GPU in practice

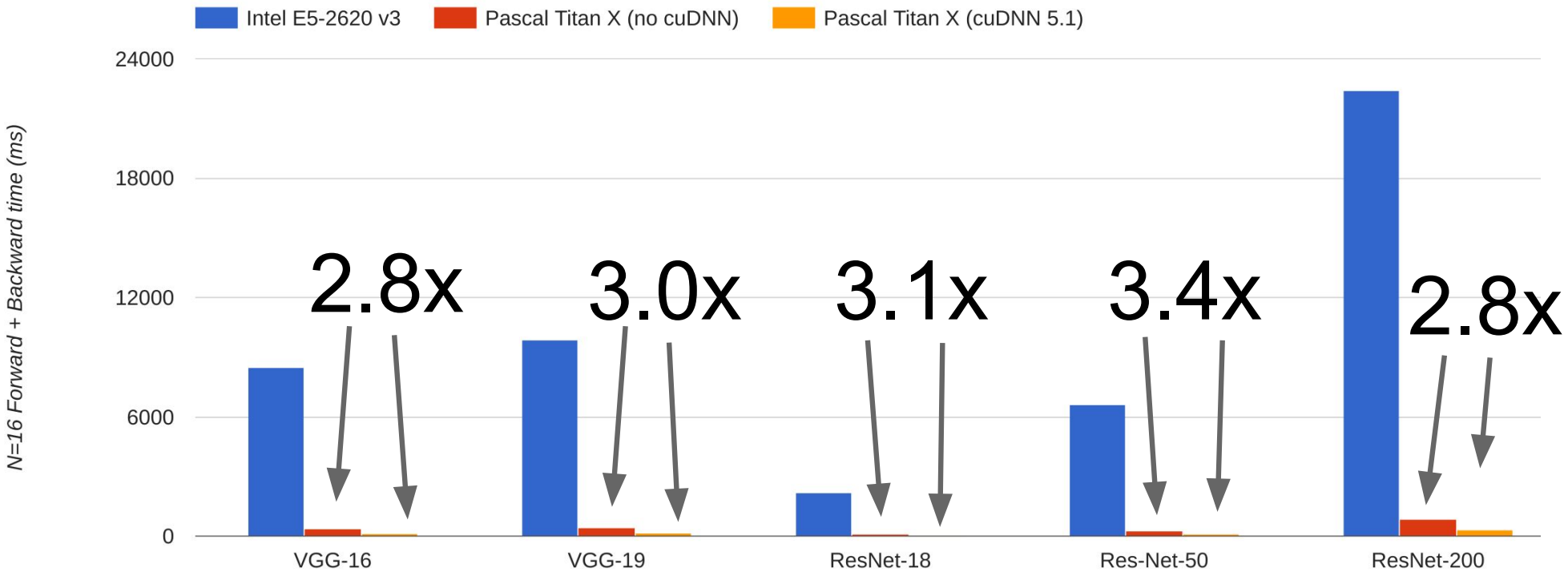
(CPU performance not well-optimized, a little unfair)



Data from https://github.com/jcjohnson/cnn-benchmarks

CPU vs GPU in practice

cuDNN much faster than "unoptimized" CUDA



Data from https://github.com/jcjohnson/cnn-benchmarks

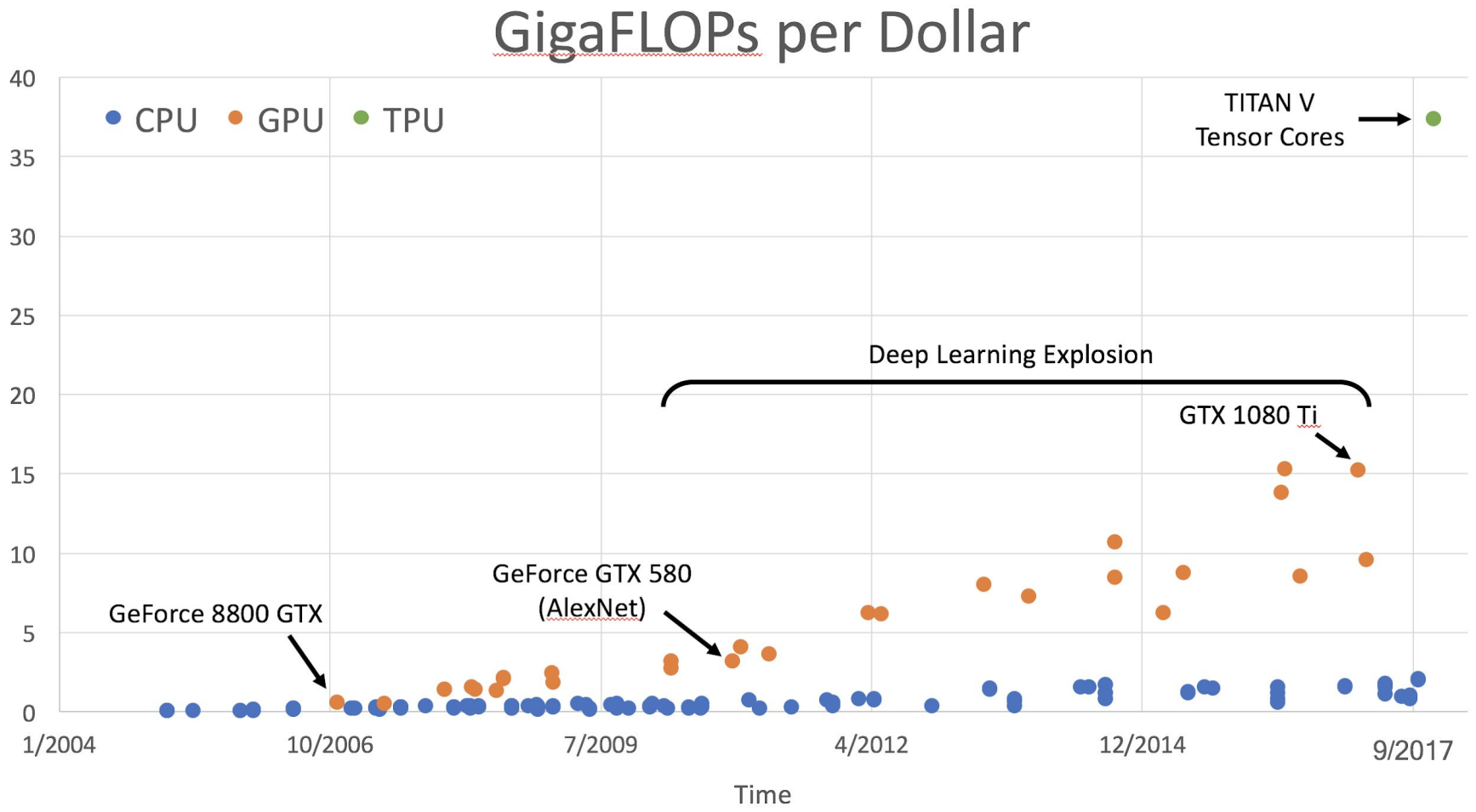
CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 TFLOPs FP32
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	$6.50 per hour	~180 TFLOP

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks

TPU: Specialized hardware for deep learning


GigaFLOPs per Dollar
CPU
GPU
TPU
TITAN V
Tensor Cores
Deep Learning Explosion
GTX 1080 Ti
GeForce GTX 580
(AlexNet)
GeForce 8800 GTX
40
35
30
25
20
15
10
5
0
1/2004
10/2006
7/2009
4/2012
12/2014
9/2017
Time

A zoo of frameworks!

PaddlePaddle
(Baidu)

Chainer

Caffe
(UC Berkeley)

Caffe2
(Facebook)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Deeplearning4j

Theano
(U Montreal)

TensorFlow
(Google)

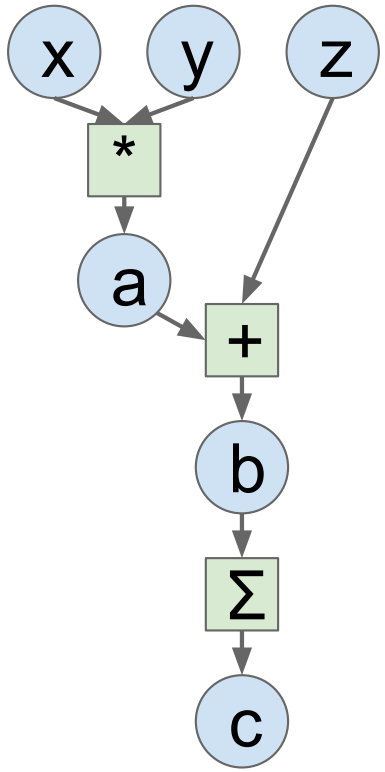
We'll focus on these

And others...

Computational Graphs

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Good:

- Clean API, easy to write numeric code

Bad:

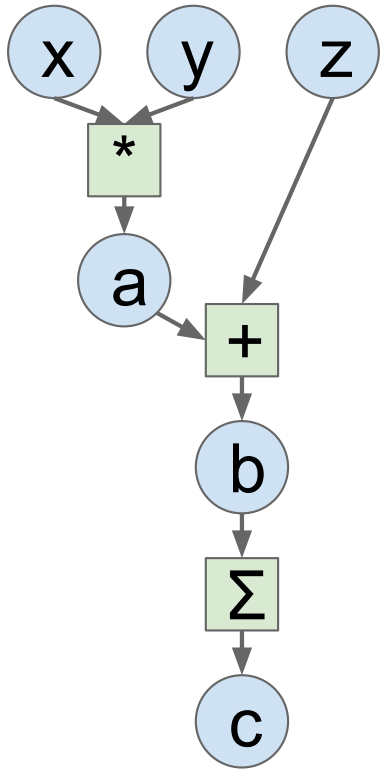
- Have to compute our own gradients
- Can't run on GPU

Computational Graphs

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```

```
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

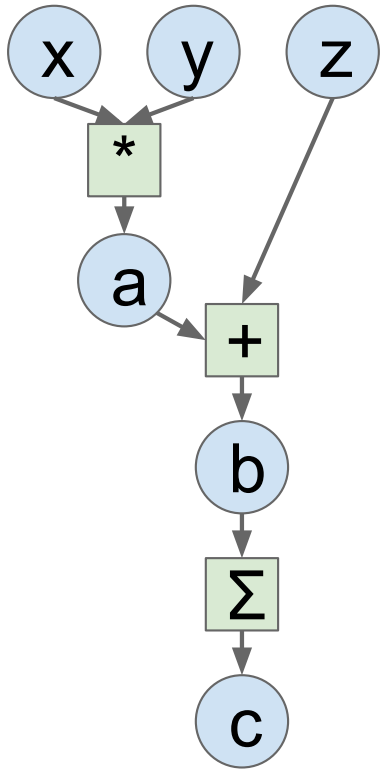
```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

Computational Graphs

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch

device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

PyTorch: Autograd

Forward pass looks exactly the same as before, but we don't need to track intermediate values - PyTorch keeps track of them for us in the graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Very similar to modular layers in A2! Use ctx object to “cache” values for the backward pass, just like cache objects from A2

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

PyTorch: nn

Define our model as a sequence of layers; each layer is an object that holds learnable weights

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: optim

Use an **optimizer** for different update rules

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: nn
Define new Modules

Initializer sets up two children (Modules can contain modules)

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: DataLoaders

Iterate over loader to form minibatches

```
import torch
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size=8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
for epoch in range(20):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

PyTorch: Pretrained Models

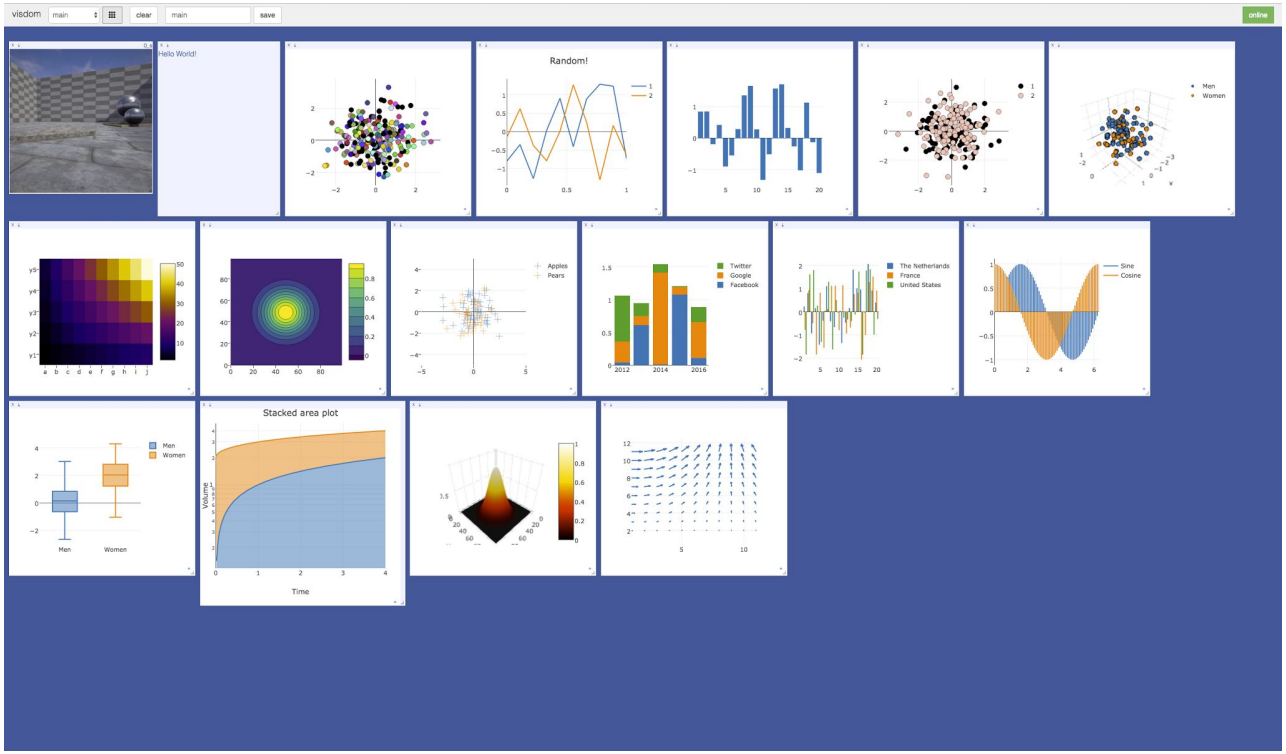
Super easy to use pretrained models with torchvision
https://github.com/pytorch/vision

```
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```

PyTorch: Visdom

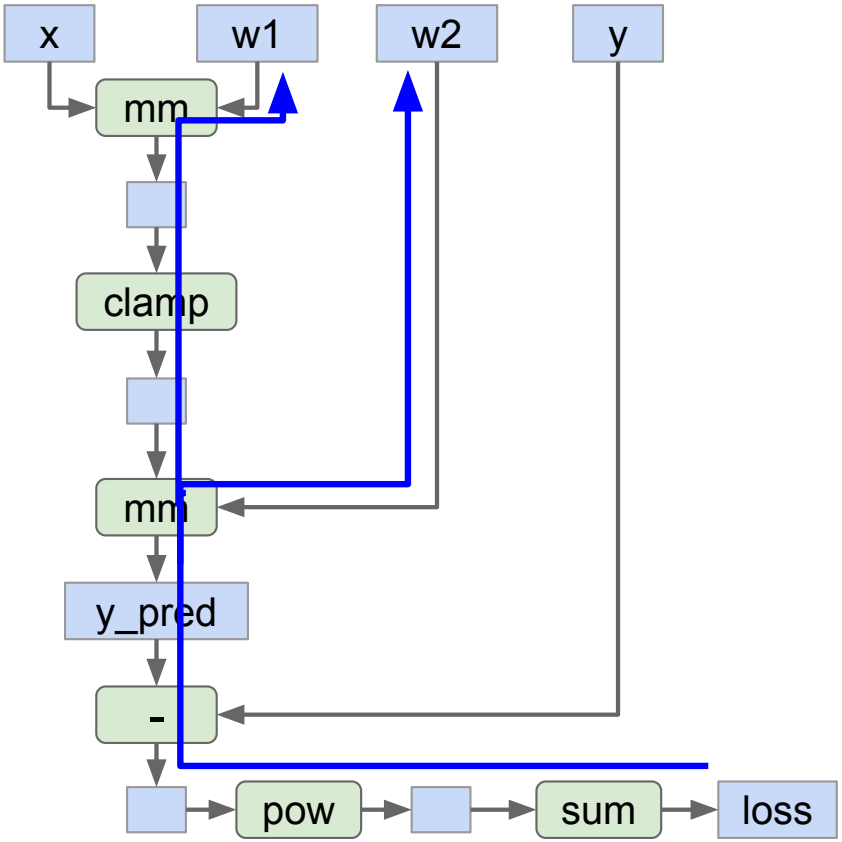
Visualization tool: add logging to your code, then visualize in a browser

Can't visualize computational graph structure (yet?)



https://github.com/facebookresearch/visdom

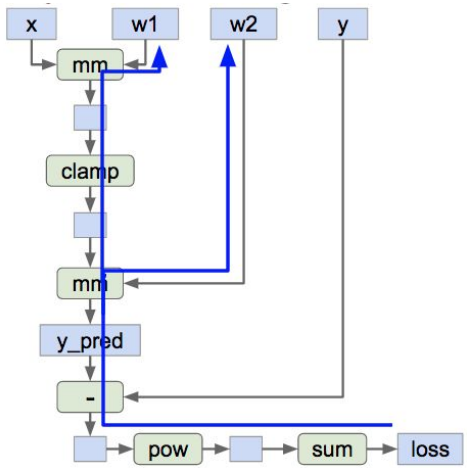
PyTorch: **Dynamic** Computation Graphs




```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Search for path between loss and w1, w2 (for backprop) AND perform computation

Static Computation Graphs



Alternative: **Static** graphs

Step 1: Build computational graph describing our computation (including finding paths for backprop)

Step 2: Reuse the same graph on every iteration

```
graph = build_graph()

for x_batch, y_batch in loader:
    run_graph(graph, x=x_batch, y=y_batch)
```

TensorFlow: Neural Net

First **define** computational graph

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

Then **run** the graph many times

```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net

Change w1 and w2 from **placeholder** (fed on each call) to **Variable** (persists in the graph between calls)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

Keras: High-Level Wrapper

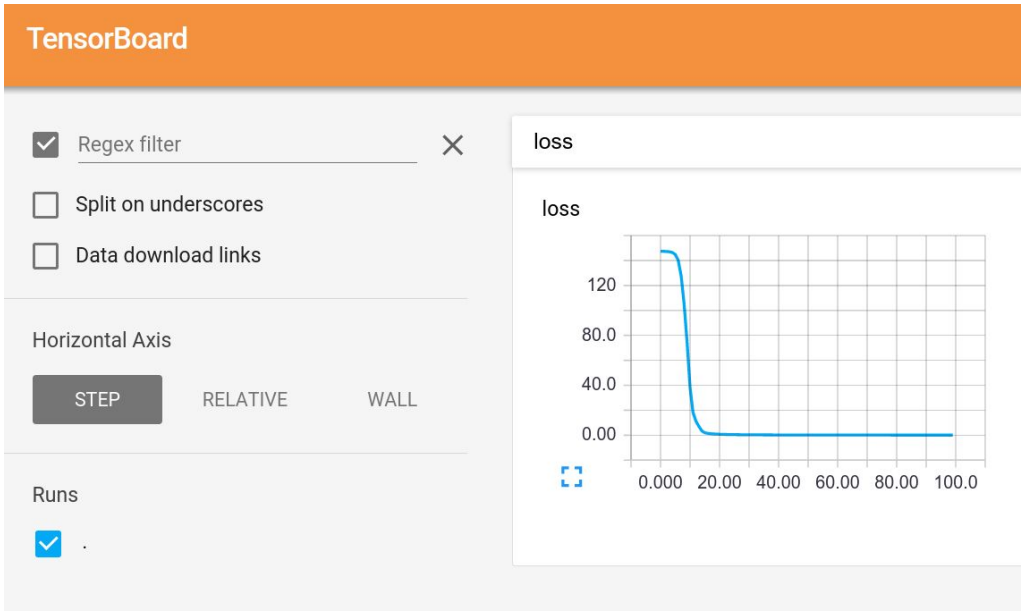
Define model as a sequence of layers

Get output by calling the model

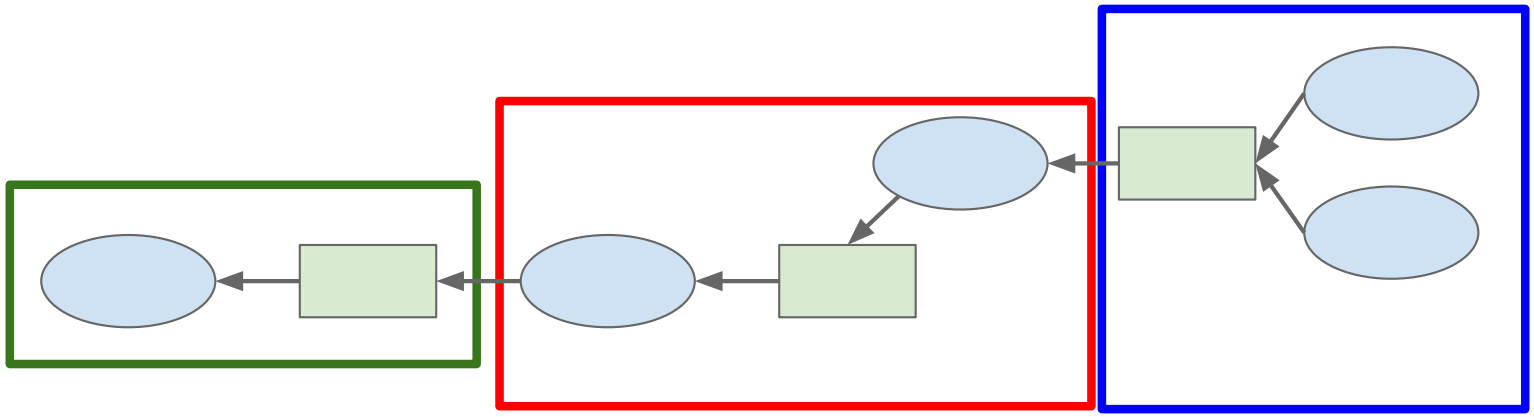
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
y_pred = model(x)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

TensorFlow: Tensorboard

Add logging to code to record loss, stats, etc
Run server and get pretty graphs!



TensorFlow: Distributed Version



Split one graph over multiple machines!

https://www.tensorflow.org/deploy/distributed

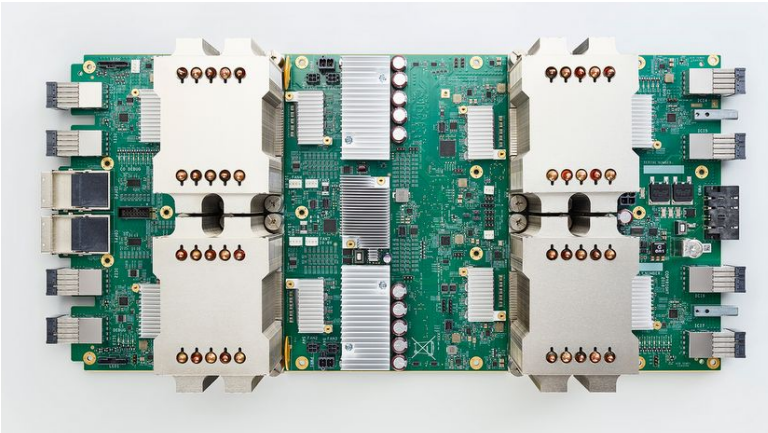
TensorFlow: Tensor Processing Units

Google Cloud TPU
= 180 TFLOPs of compute!

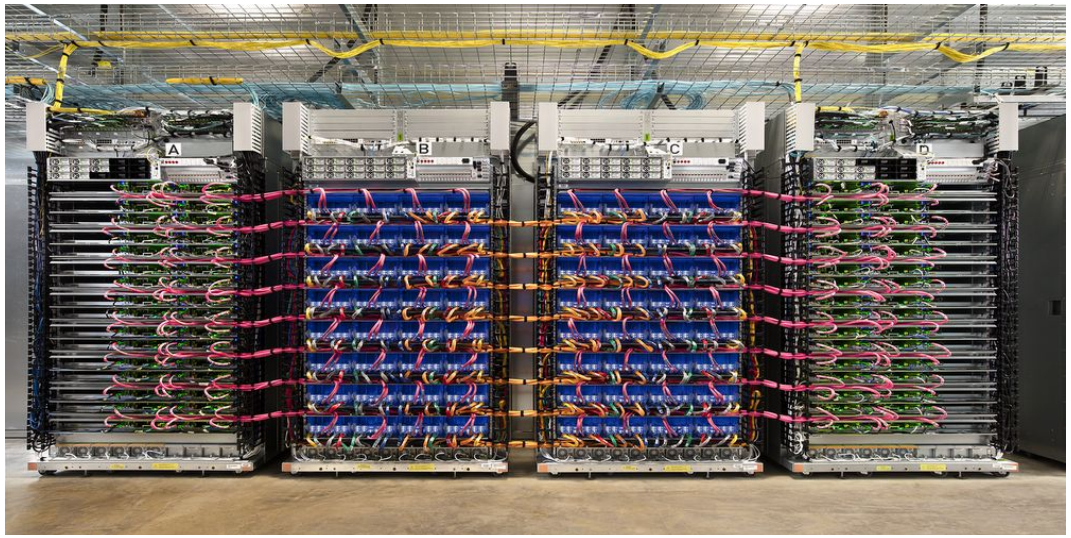
NVIDIA Tesla V100
= 125 TFLOPs of compute

NVIDIA Tesla P100 = 11 TFLOPs of compute
GTX 580 = 0.2 TFLOPs

TensorFlow: Tensor Processing Units



Google Cloud TPU
= 180 TFLOPs of compute!



Google Cloud TPU Pod
= 64 Cloud TPUs
= 11.5 PFLOPs of compute!

https://www.tensorflow.org/versions/master/programmers_guide/using_tpu

Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static)**

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic)**

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

New graph each iteration

Static vs Dynamic: Conditional

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

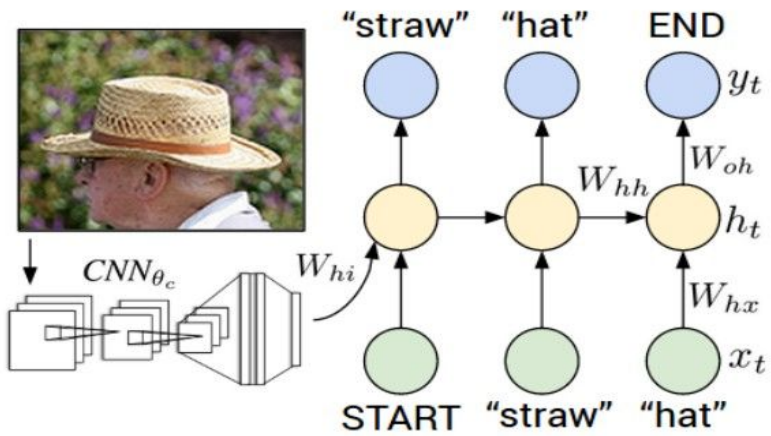
```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

TensorFlow: Special TF control flow operator!

```
N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(D, H))

def f1(): return tf.matmul(x, w1)
def f2(): return tf.matmul(x, w2)
y = tf.cond(tf.less(z, 0), f1, f2)

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        z: 10,
        w1: np.random.randn(D, H),
        w2: np.random.randn(D, H),
    }
    y_val = sess.run(y, feed_dict=values)
```

Dynamic Graph Applications

- Recurrent networks



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks

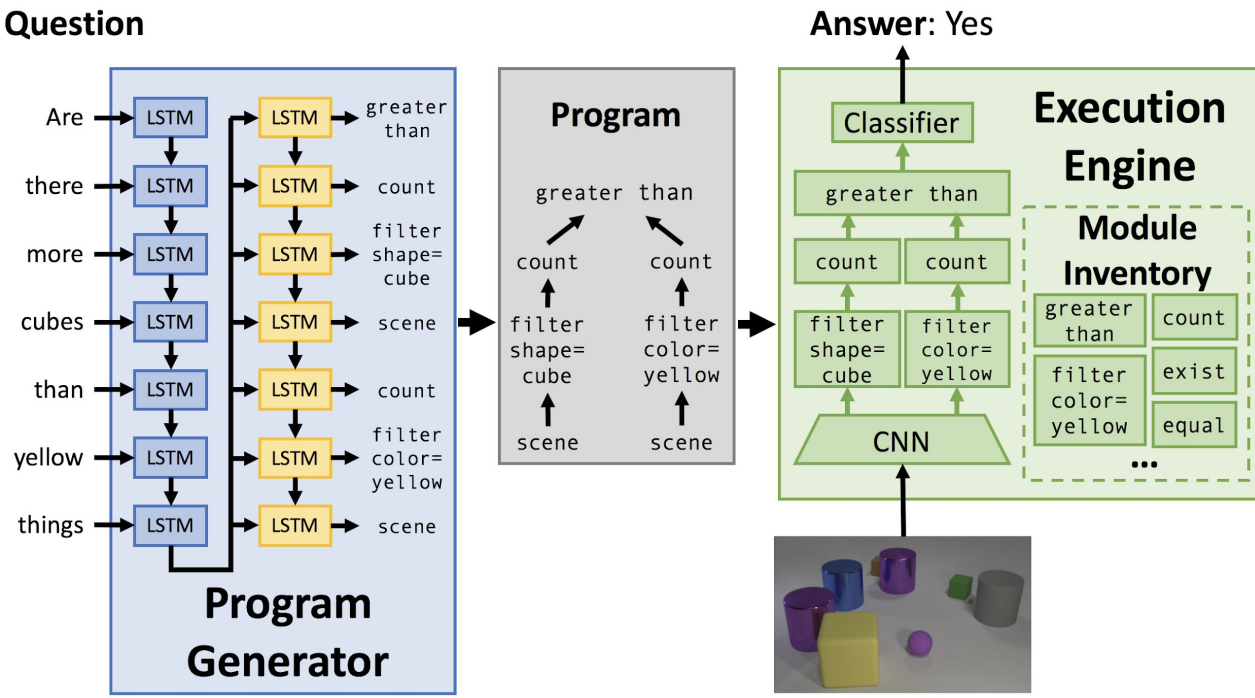


Figure copyright Justin Johnson, 2017. Reproduced with permission.

Andreas et al, "Neural Module Networks", CVPR 2016
Andreas et al, "Learning to Compose Neural Networks for Question Answering", NAACL 2016
Johnson et al, "Inferring and Executing Programs for Visual Reasoning", ICCV 2017

My Advice:

PyTorch is my personal favorite. Clean API, dynamic graphs make it very easy to develop and debug. Can build model in PyTorch then export to Caffe2 with ONNX for production / mobile

TensorFlow is a safe bet for most projects. Not perfect but has huge community, wide usage. Can use same framework for research and production. Probably use a high-level framework. Only choice if you want to run on TPUs.