Deep Learning Using

Masoud Pourreza

Pourreza.masoud@gmail.com

@masoudpz

HAMIM

Torch

- Open source machine learning library
- Scientific computing framework
- Based on the Lua !!
- Research Companies and Labs
- Facebook AI research, Google + Deep mind , Twitter, NVIDIA
- Started at 2002
- website

PyTorch

- PyTorch is a python package that provides two high-level features:
  - Tensor computation (like numpy) with strong GPU acceleration
  - Deep Neural Networks built on a tape-based autograd system
- PyTorch provides Tensors that can live either on the CPU or the GPU
- PyTorch is a python library
- More Pythonic (imperative)
- Flexible
- Intuitive and cleaner code
- Easy to debug
- More Neural Networkic
- Write code as the network works
- forward/backward

PyTorch packages

Package	Description
torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.multiprocessing	python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience

Installation requirements

Python on ubuntu

Python on windows

Pip on ubuntu

Pip on windows

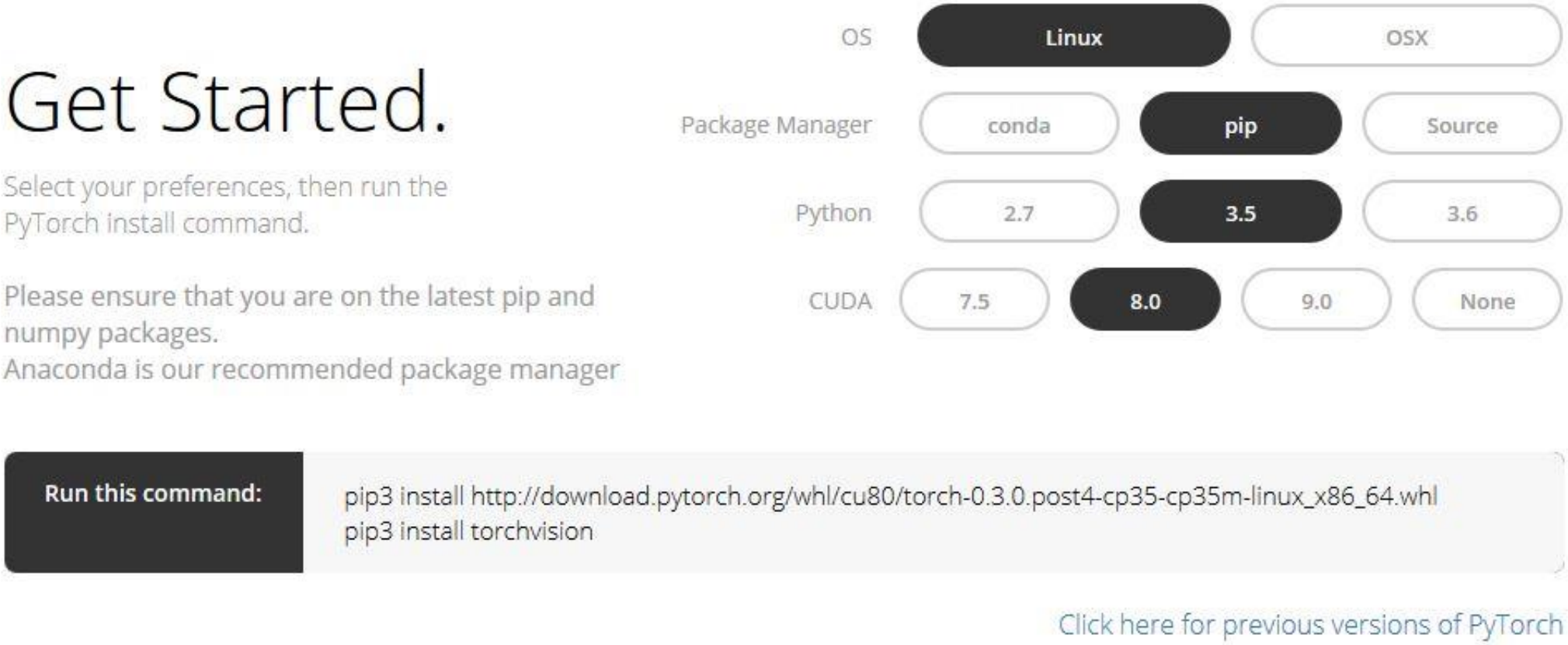
conda on ubuntu

conda on windows

Compile pytorch from source

PyTorch Installation

http://pytorch.org



Installation on windows using codna

HAMIM

Linear Model Example

What would be the grade if I study 4 hours?

Hours (x)	Points (y)
1	2
2	4
3	6
4	?

Supervised Learning

Training data

Test data

Linear Model

- What would be the best model for the data? Linear?

Hours (x)	Points (y)
1	2
2	4
3	6
4	?

$$\hat{y} = x * w + b$$

x → Linear → $\hat{y}$

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=3)	Loss (w=3)
1	2	3	1
2	4	6	4
3	6	9	9
			mean=14/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=4)	Loss (w=4)
1	2	4	4
2	4	8	16
3	6	12	36
			mean=56/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=0)	Loss (w=0)
1	2	0	4
2	4	0	16
3	6	0	36
			mean=56/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=1)	Loss (w=1)
1	2	1	1
2	4	2	4
3	6	3	9
			mean=14/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=2)	Loss (w=2)
1	2	2	0
2	4	4	0
3	6	6	0
			mean=0

Training Loss (error)

MSE mean square error

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

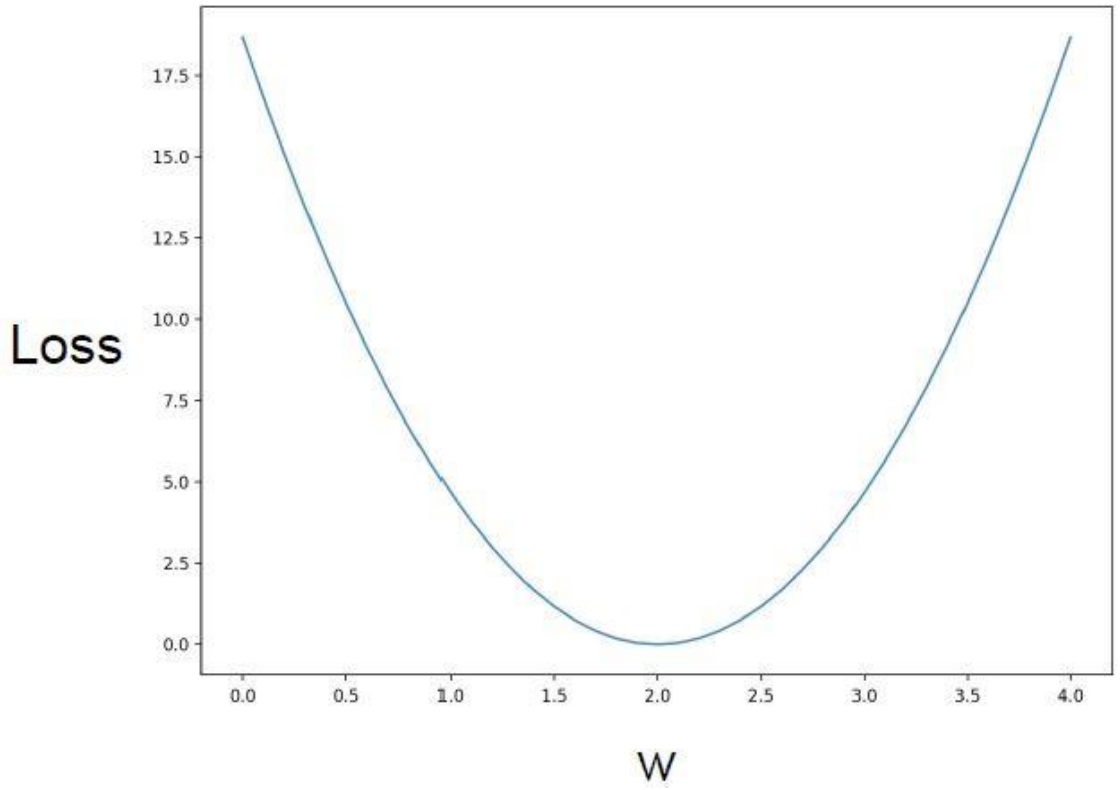
$$loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

Hours, x	Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
1	4	1	0	1	4
2	16	4	0	4	16
3	36	9	0	9	36
	MSE=56/3=18.7	MSE=14/3=4.7	MSE=0	MSE=14/3=4.7	MSE=56/3=18.7

Loss Graph

$$loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2$$

Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
mean=56/3=18.7	mean=14/3=4.7	mean=0	mean=14/3=4.7	mean=56/3=18.7



HAMIM

Model & Loss

$$\hat{y} = x * w$$

```
w = 1.0  # a random guess: random value

# our model for the forward pass
def forward(x):
    return x * w
```

$$loss = (\hat{y} - y)^2$$

```
# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)
```

HAMIM©

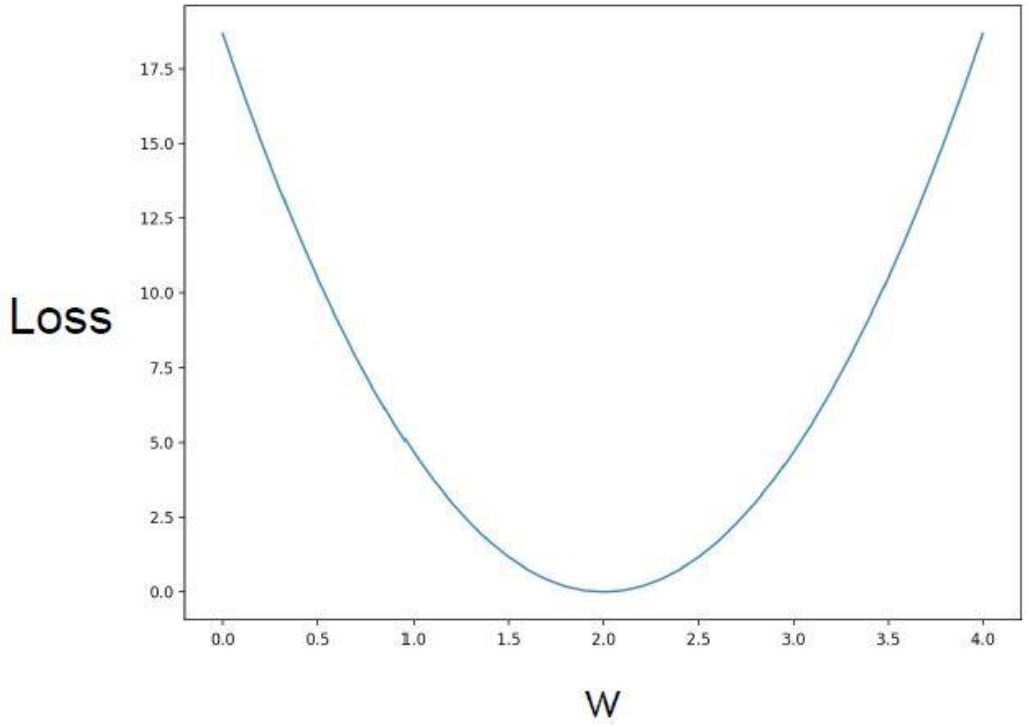
Compute loss for W

```
for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred_val = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred_val, l)

    print("MSE=", l_sum / 3)
```

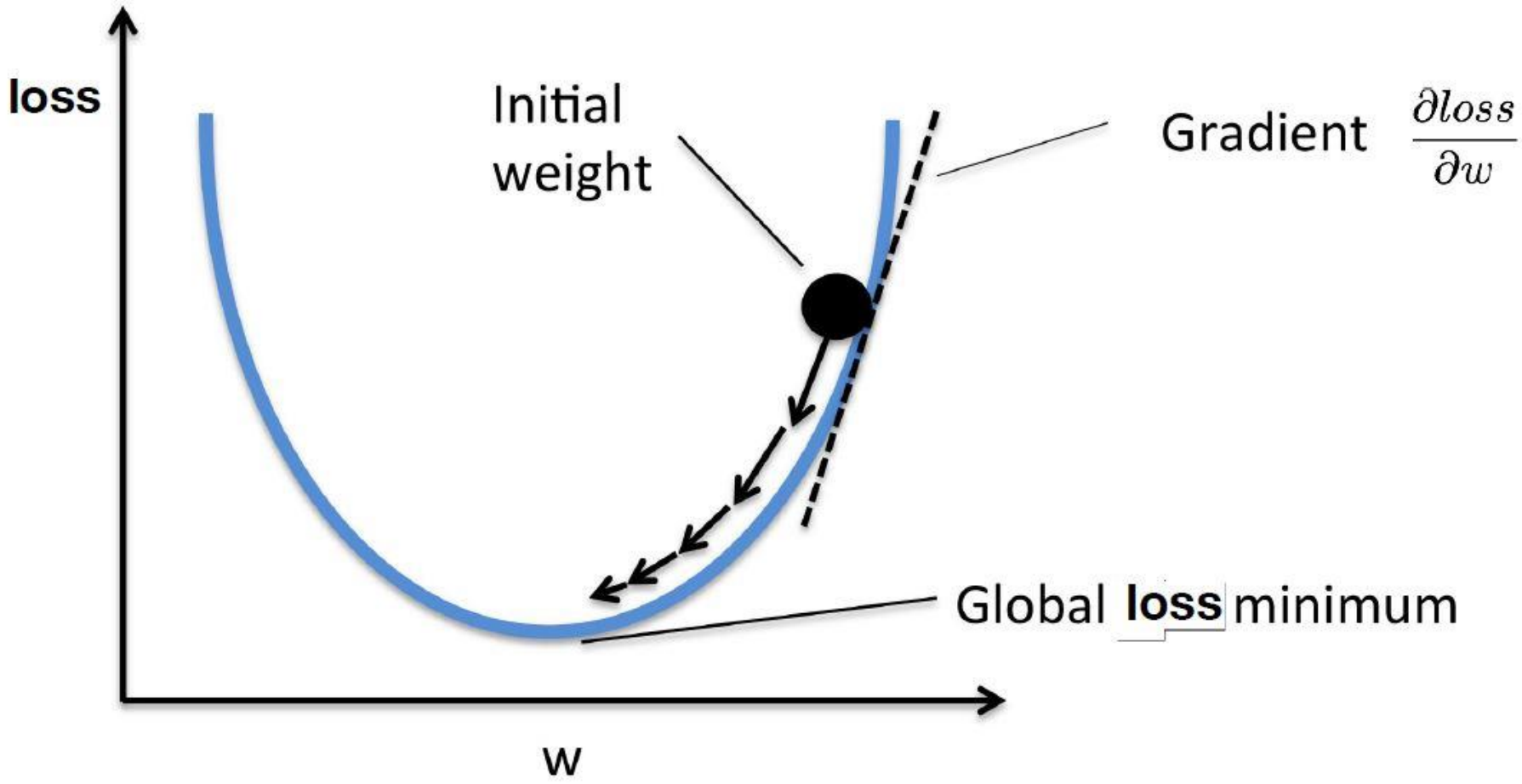
```
w= 0.0
	 1.0 2.0 0.0 4.0
	 2.0 4.0 0.0 16.0
	 3.0 6.0 0.0 36.0
NSE= 18.6666666667
w= 0.1
	 1.0 2.0 0.1 3.61
	 2.0 4.0 0.2 14.44
	 3.0 6.0 0.3 32.49
NSE= 16.8466666667
w= 0.2
	 1.0 2.0 0.2 3.24
	 2.0 4.0 0.4 12.96
	 3.0 6.0 0.6 29.16
NSE= 15.12
w= 0.3
	 1.0 2.0 0.3 2.89
	 2.0 4.0 0.6 11.56
	 3.0 6.0 0.9 26.01
NSE= 13.4866666667
w= 0.4
	 1.0 2.0 0.4 2.56
	 2.0 4.0 0.8 10.24
	 3.0 6.0 1.2 23.04
NSE= 11.9466666667
w= 0.5
	 1.0 2.0 0.5 2.25
	 2.0 4.0 1.0 9.0
	 3.0 6.0 1.5 20.25
NSE= 10.5
```

Plot Graph

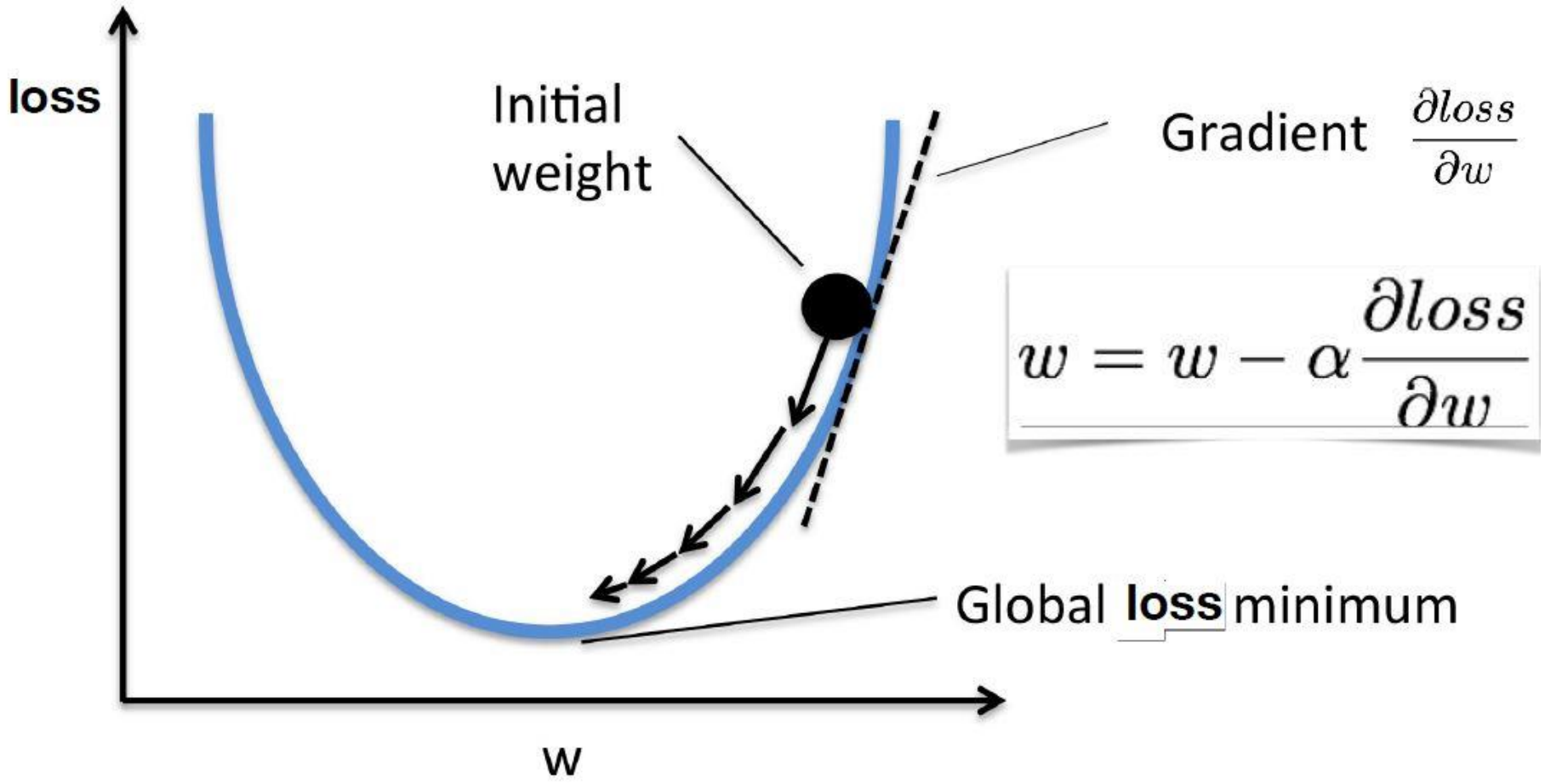
```
w_list = []
mse_list = []
for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred_val = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred_val,

    print("MSE=", l_sum / 3)
    w_list.append(w)
    mse_list.append(l_sum / 3)

plt.plot(w_list, mse_list)
plt.ylabel('Loss')
plt.xlabel('w')
plt.show()
```



Gradient Descent



Gradient Descent



Derivative

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

$$w = w - \alpha \frac{\partial loss}{\partial w}$$

$$\frac{\partial loss}{\partial w} = ?$$

Derivative

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

$$\frac{\partial loss}{\partial w} = ?$$

Derivative

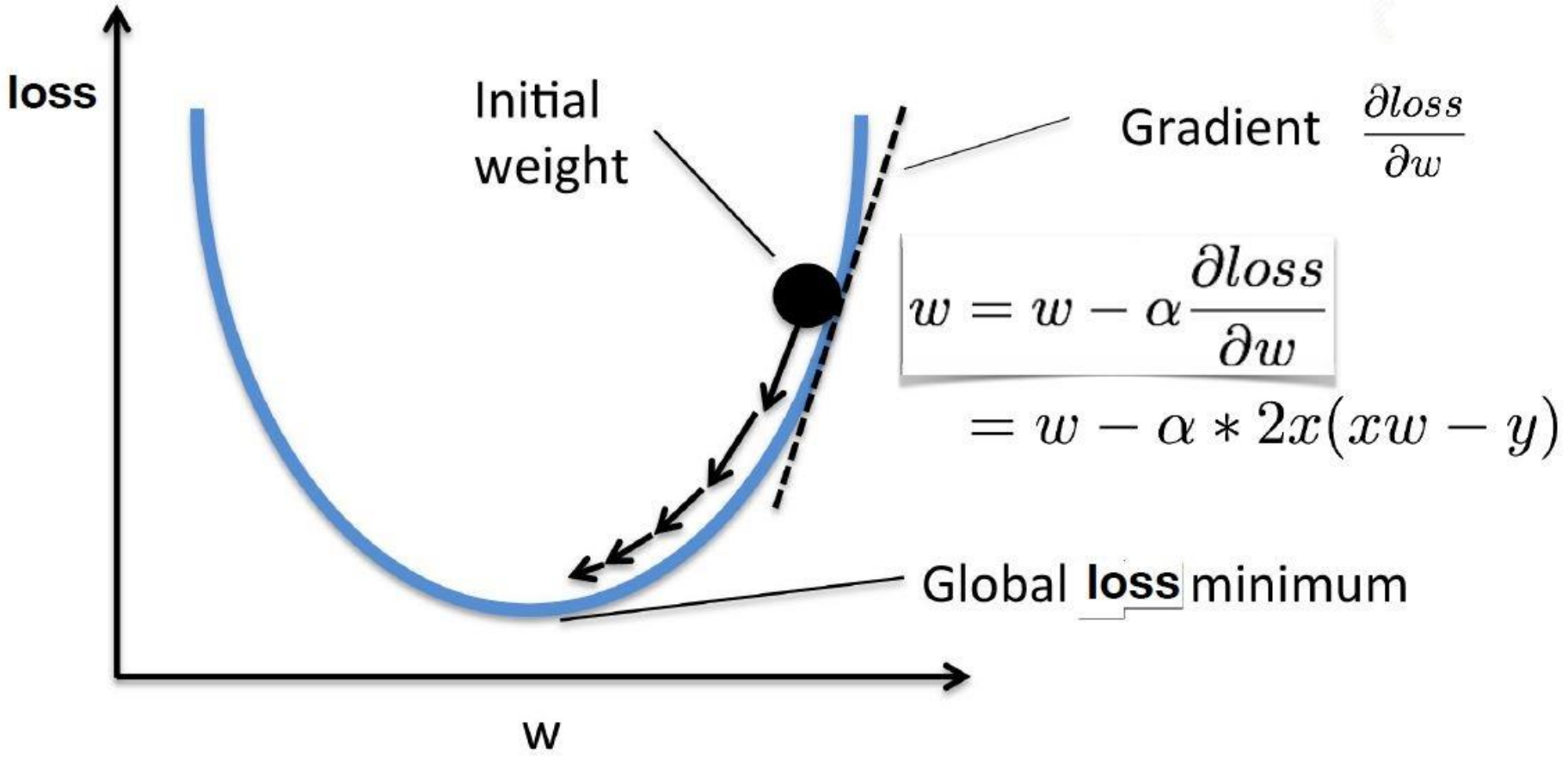
$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

$$\frac{\partial loss}{\partial w} = ?$$

$(U^2)' = 2 \times U \times U'$

$$\frac{\partial loss}{\partial w} = 2\,(x*w - y) * (x)$$

Gradient Descent



HAMIM

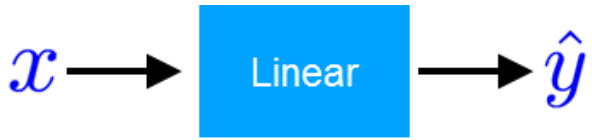
Data, Model , Loss and Gradient

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0  # a random guess: random value

# our model forward pass
def forward(x):
    return x * w

# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)


# compute gradient
def gradient(x, y):  # d_loss/d_w
    return 2 * x * (x * w - y)
```

Refresh W

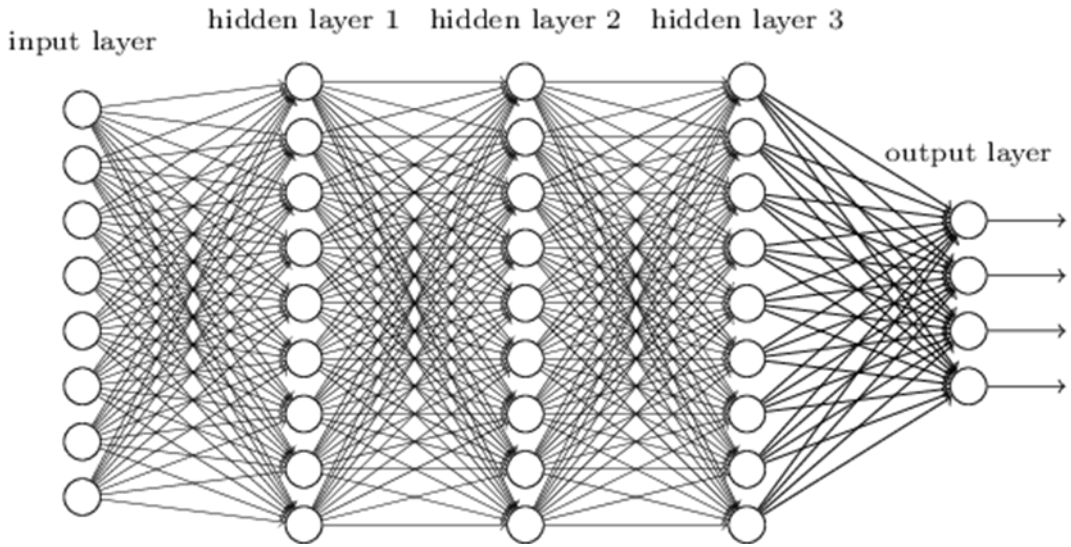
```
# Before training
print("predict (before training)",  4, forward(4))

# Training loop
for epoch in range(100):
    for x_val, y_val in zip(x_data, y_data):
        grad = gradient(x_val, y_val)
        w = w - 0.01 * grad
        print("\tgrad: ", x_val, y_val, grad)
        l = loss(x_val, y_val)

    print("progress:", epoch, "w=", w, "loss=", l)

# After training
print("predict (after training)",  "4 hours", forward(4))
```

Back-propagation & Autograd



Gradient of **loss** with respect to ***w*** $\frac{\partial loss}{\partial w} = ?$

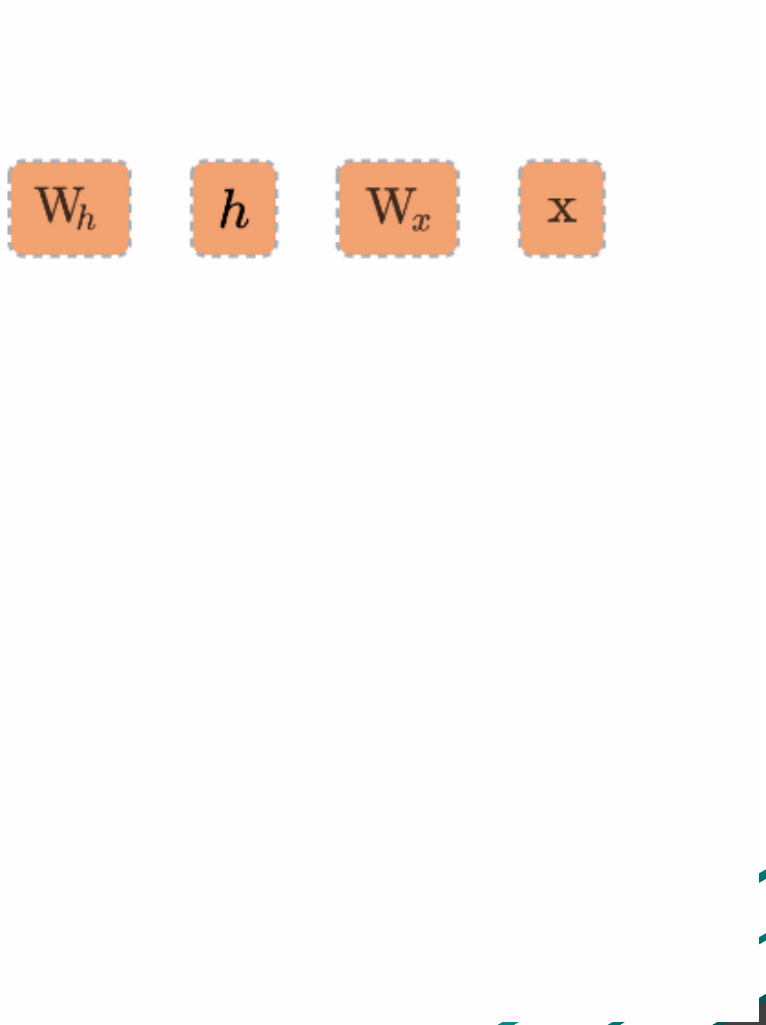
```
# compute gradient
def gradient(x, y):  # d_loss/d_w
    return 2 * x * (x * w - y)
```



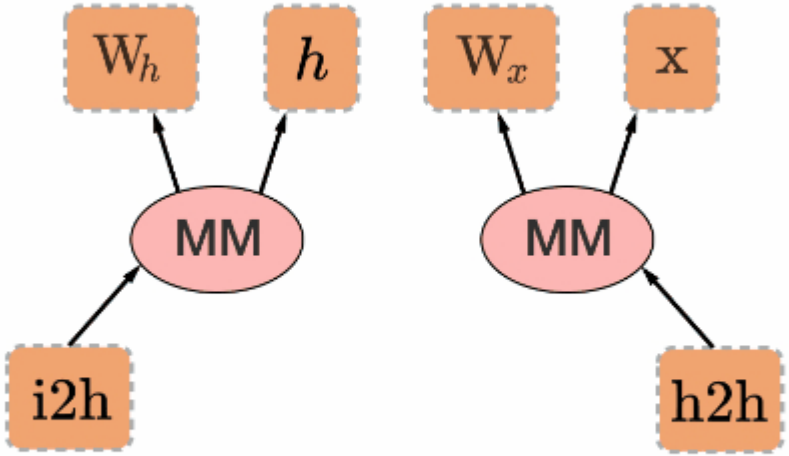
Gradient of **loss** with respect to ***w*** $\frac{\partial loss}{\partial w} = ?$

HAMIM

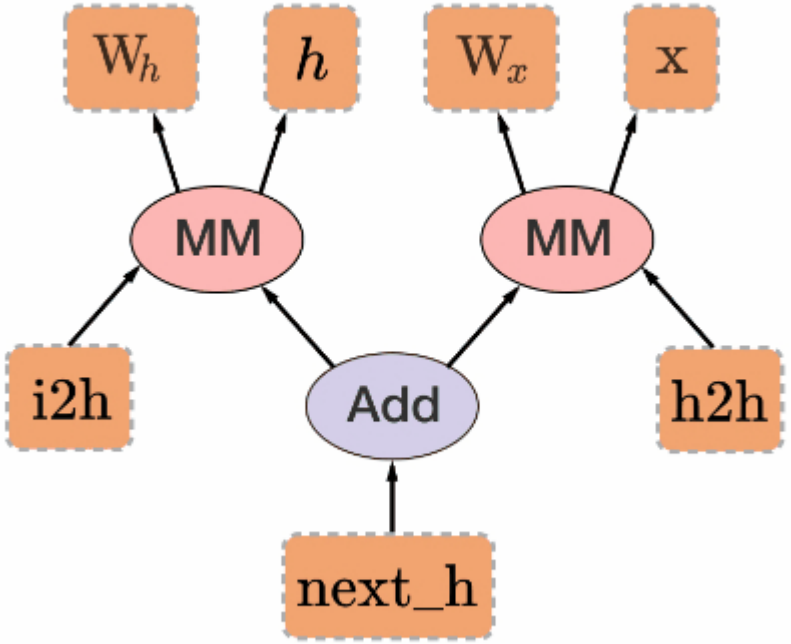
Method? Computational Graph + Chain Rule



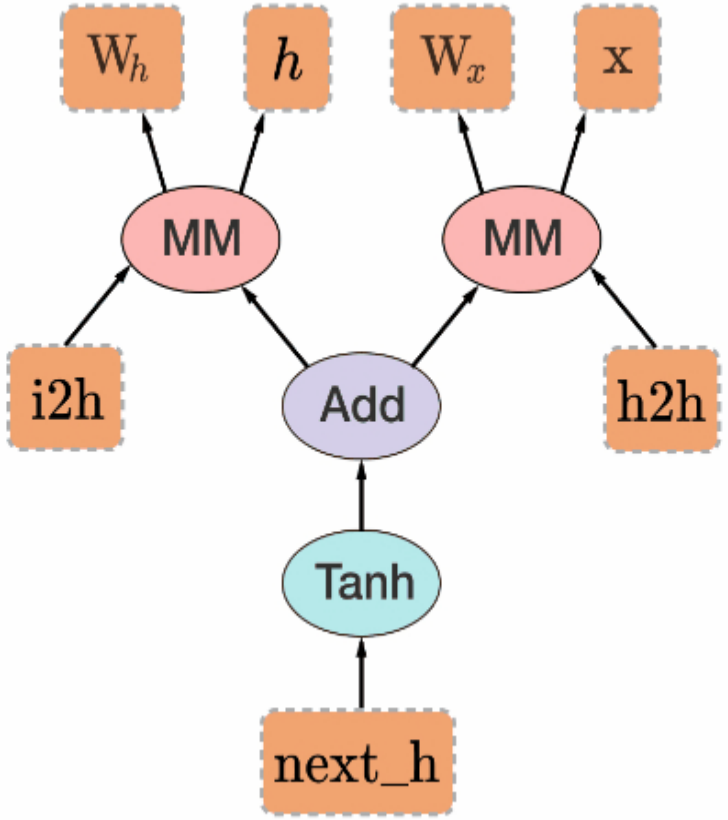
Method? Computational Graph + Chain Rule



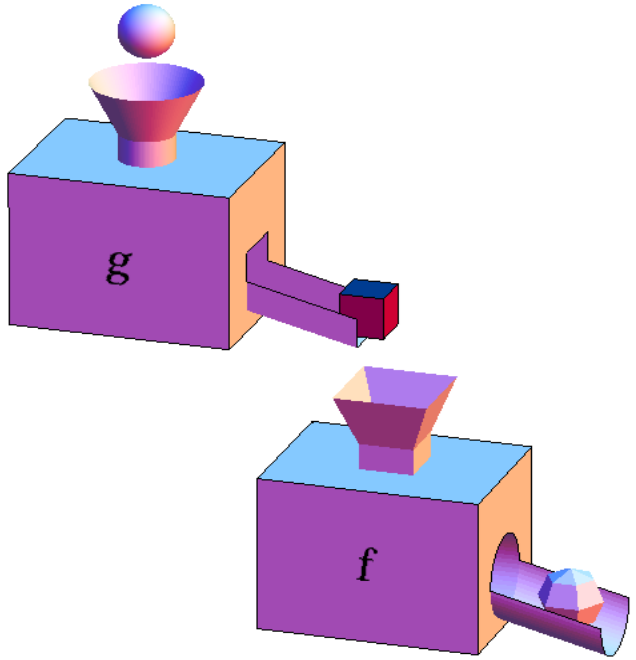
Method? Computational Graph + Chain Rule



Method? Computational Graph + Chain Rule



Chain Rule

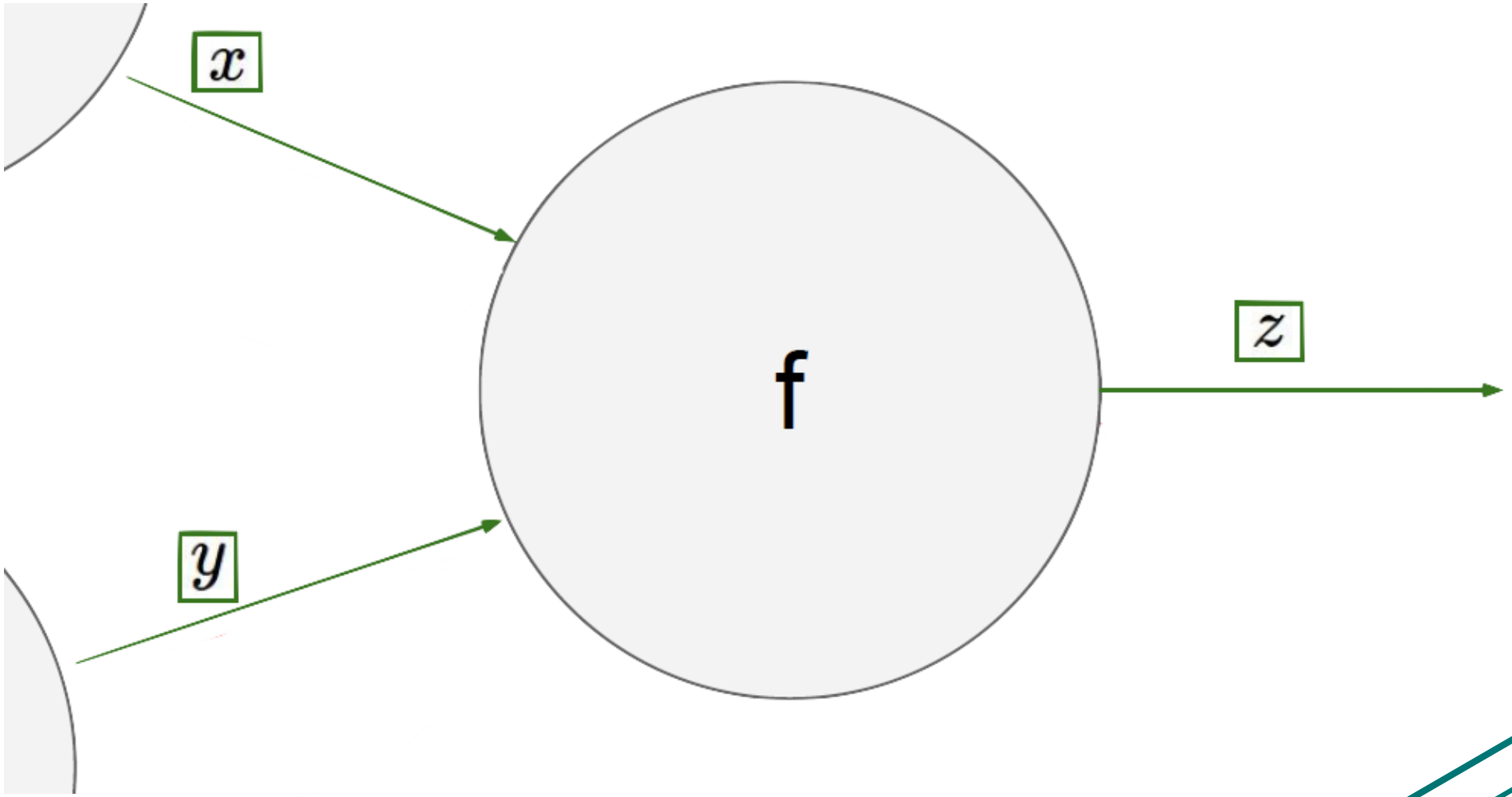


The Chain Rule

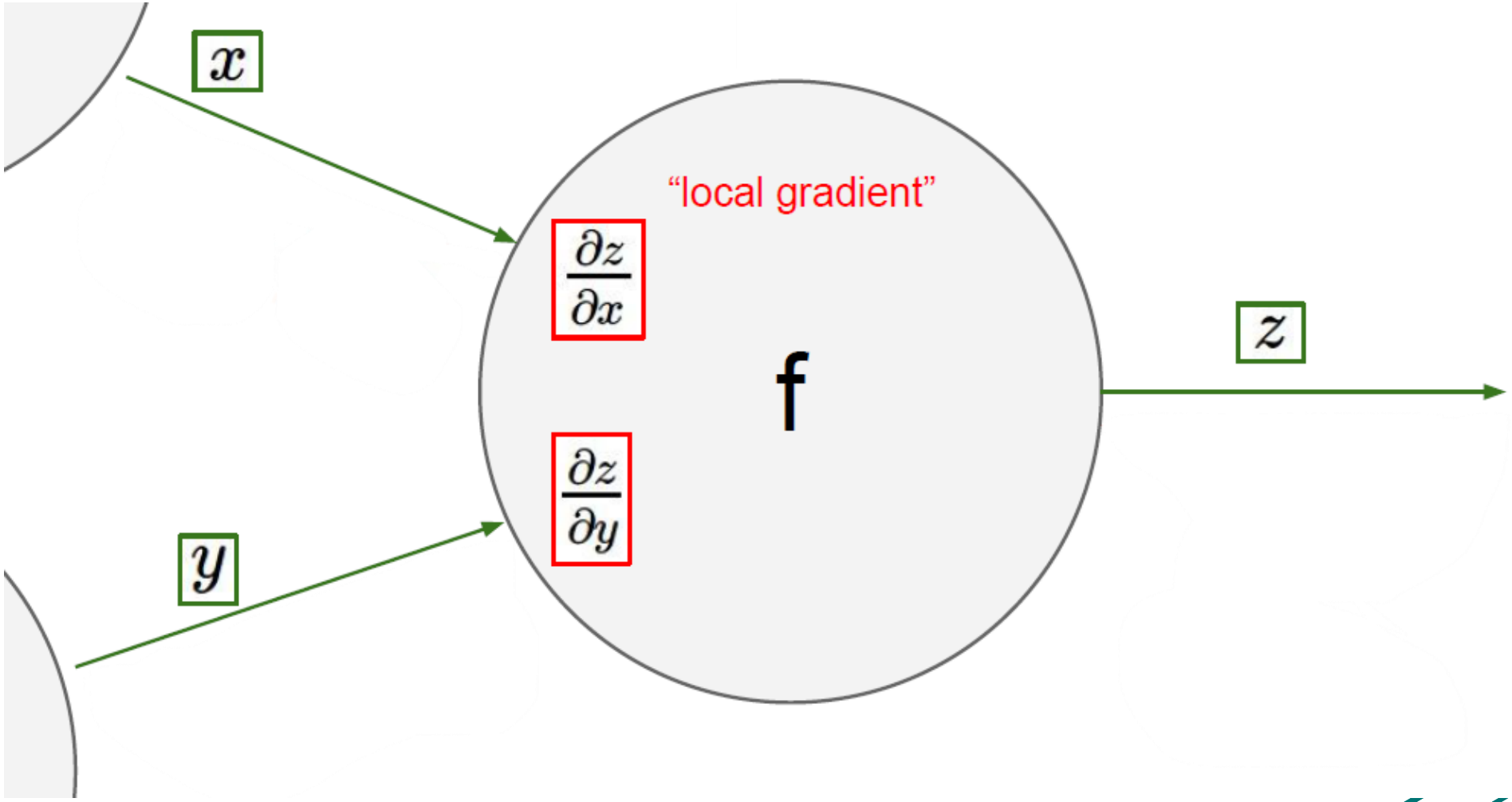
$$f = f(g)\,; \quad g = g(x)$$

$$\frac{df}{dx} = \frac{df}{dg}\,\frac{dg}{dx}$$

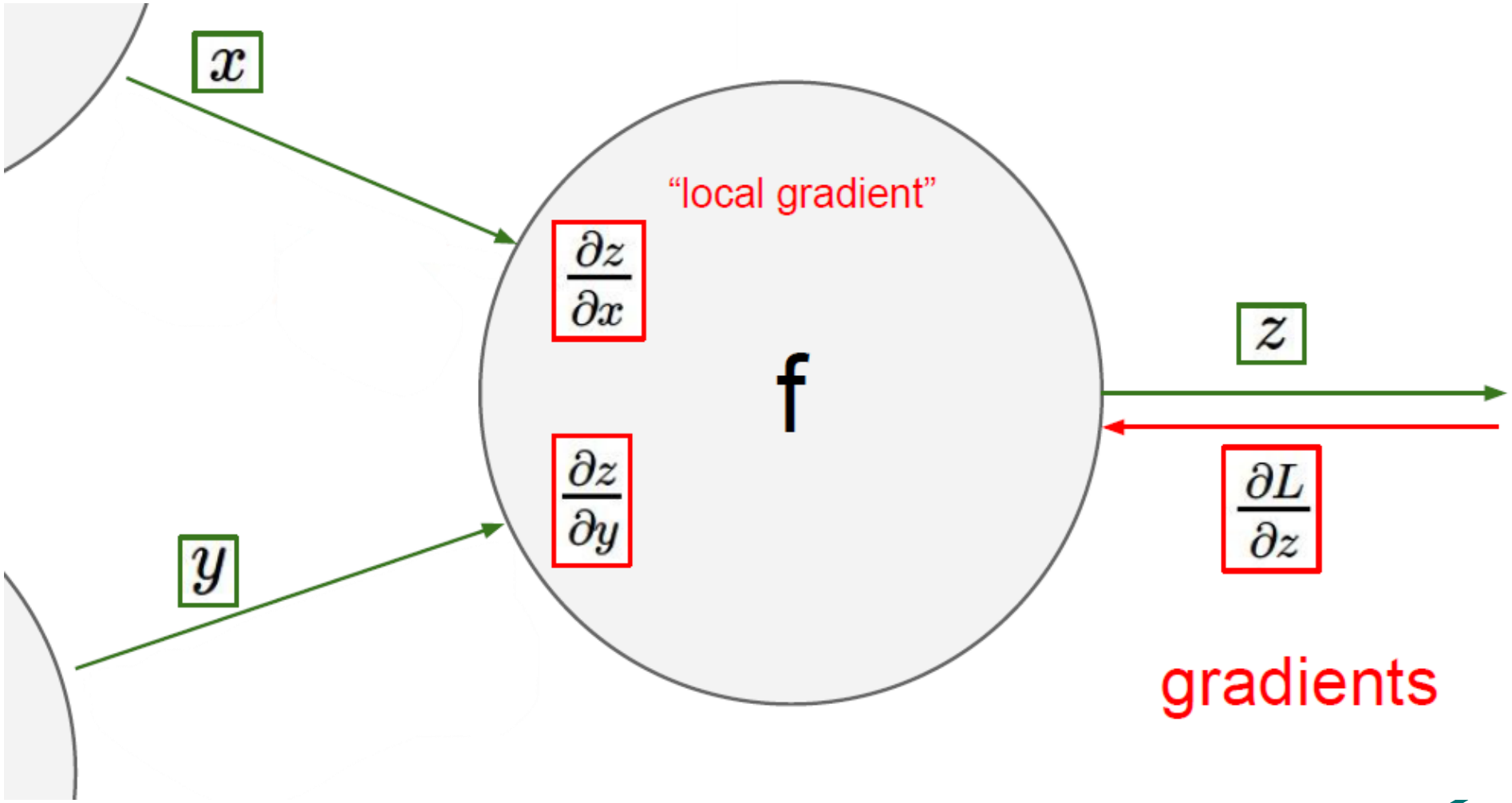
Chain Rule



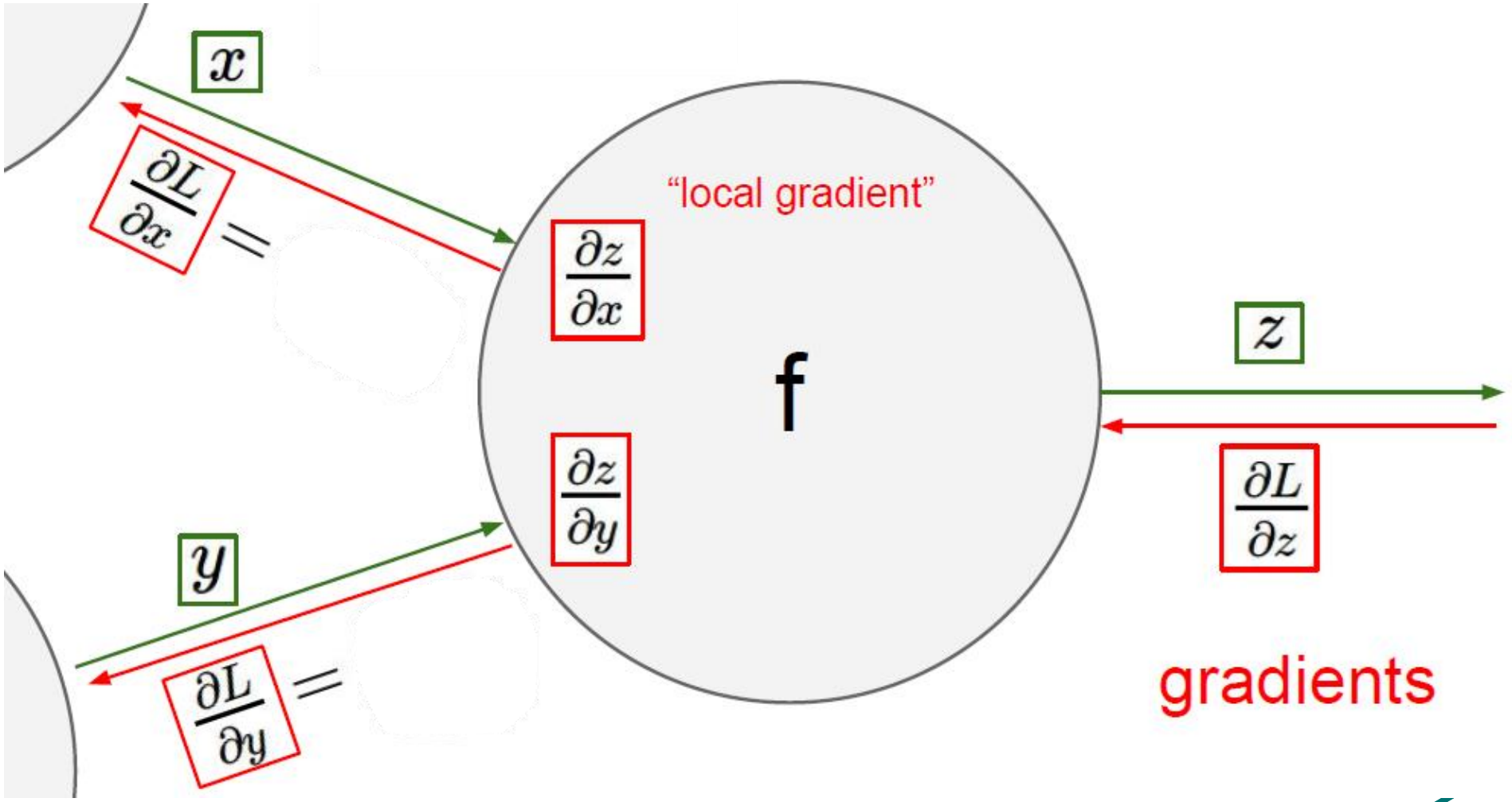
Chain Rule



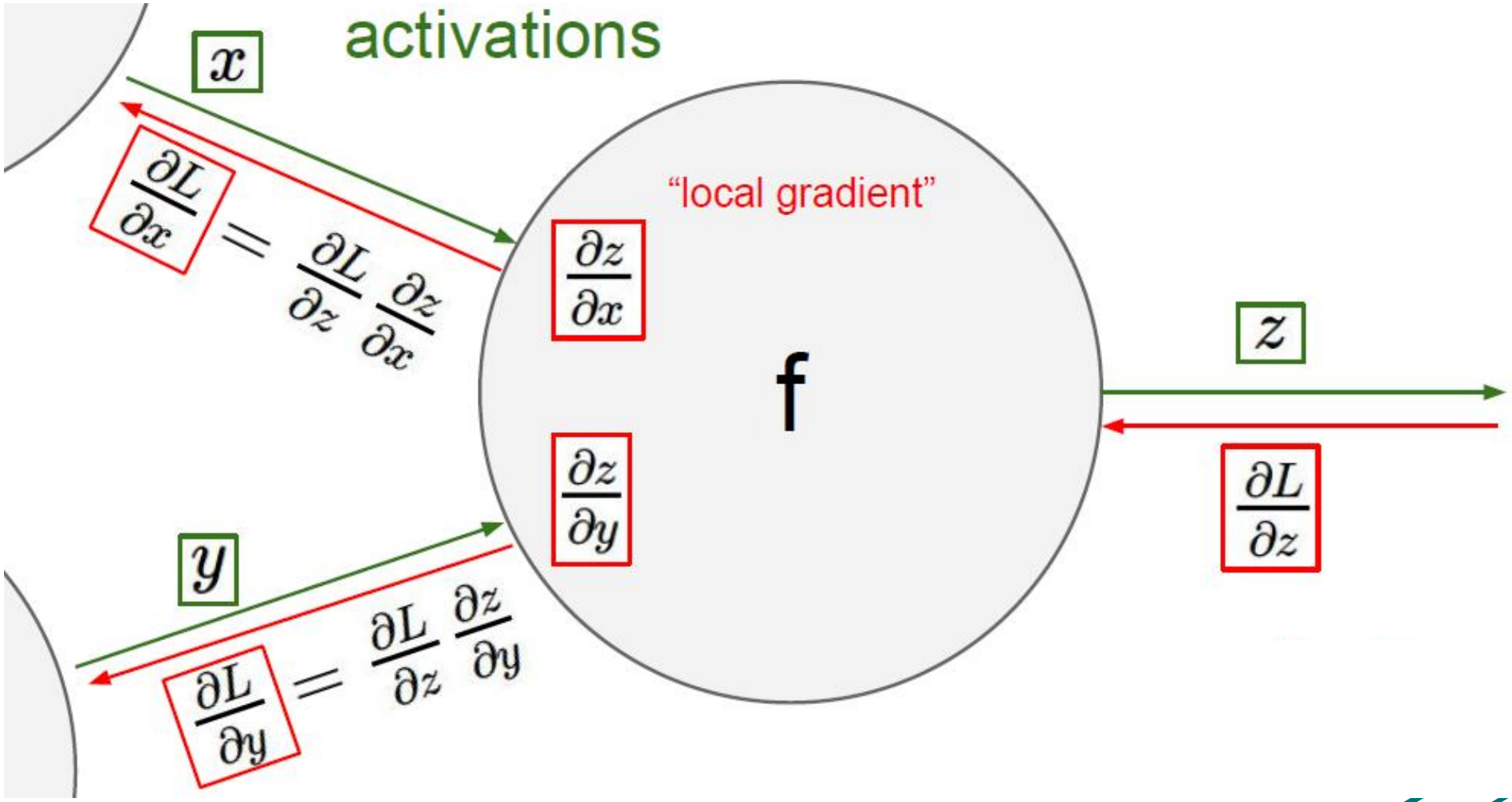
Chain Rule



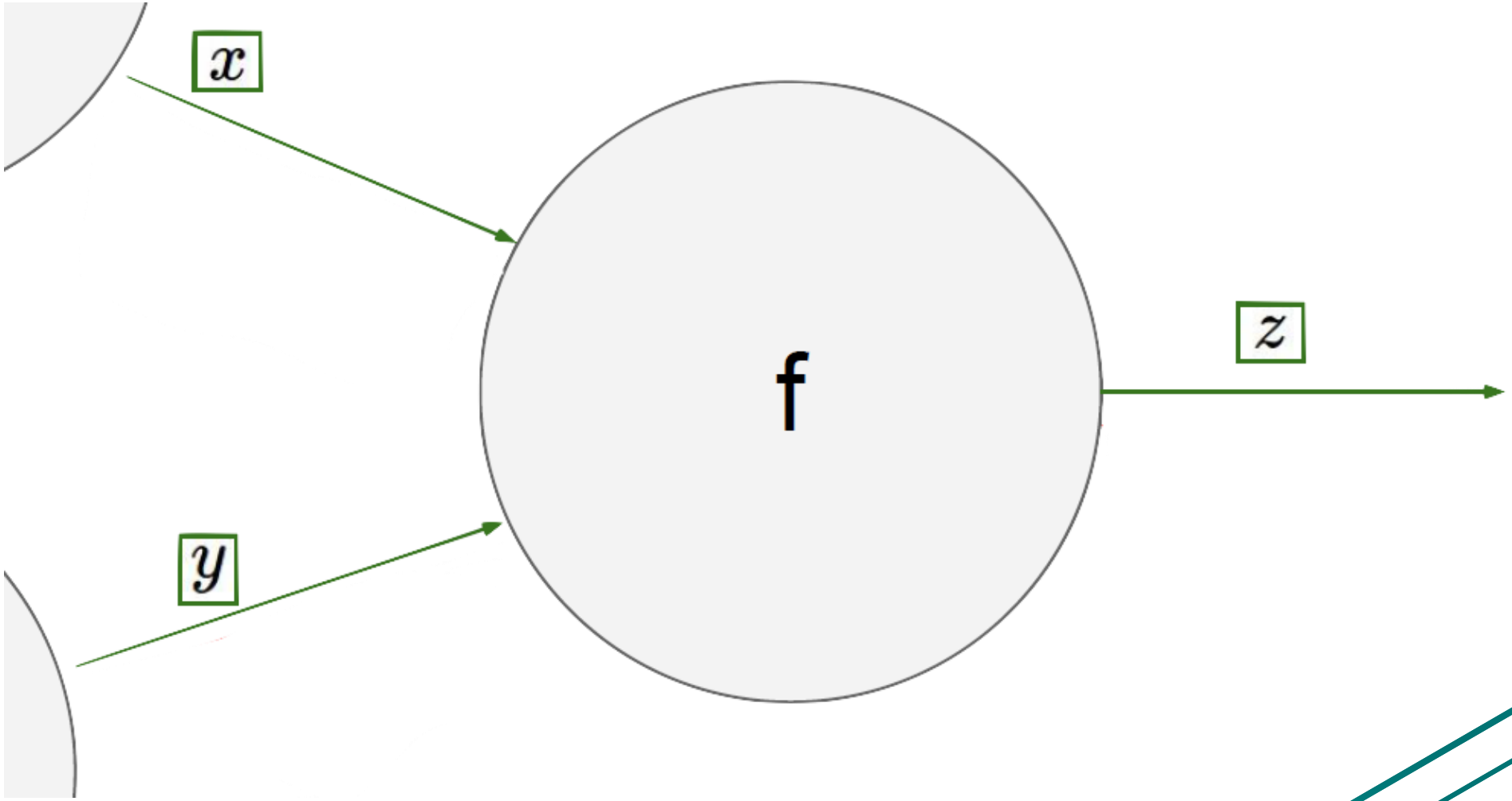
Chain Rule



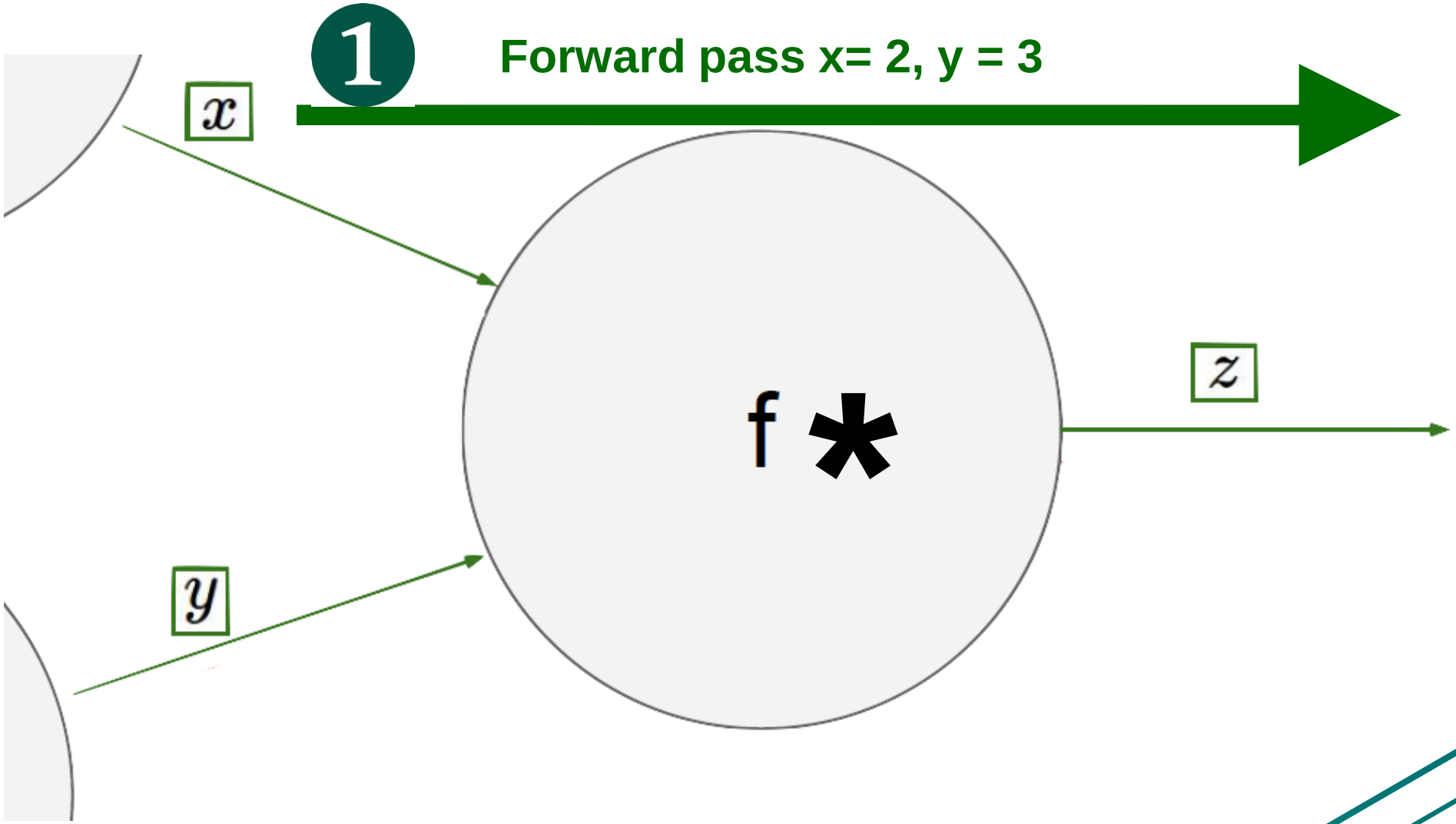
Chain Rule



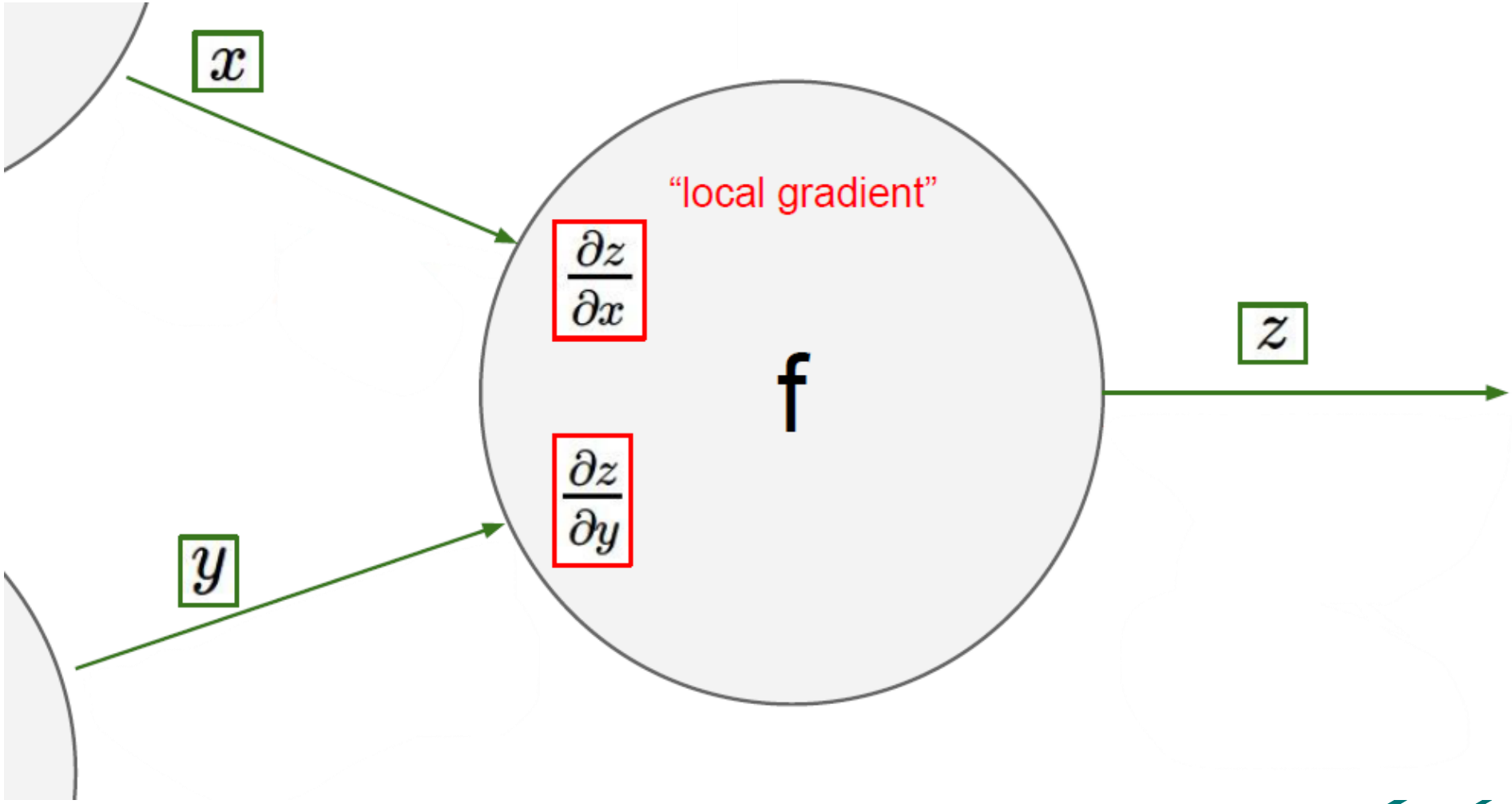
Chain Rule



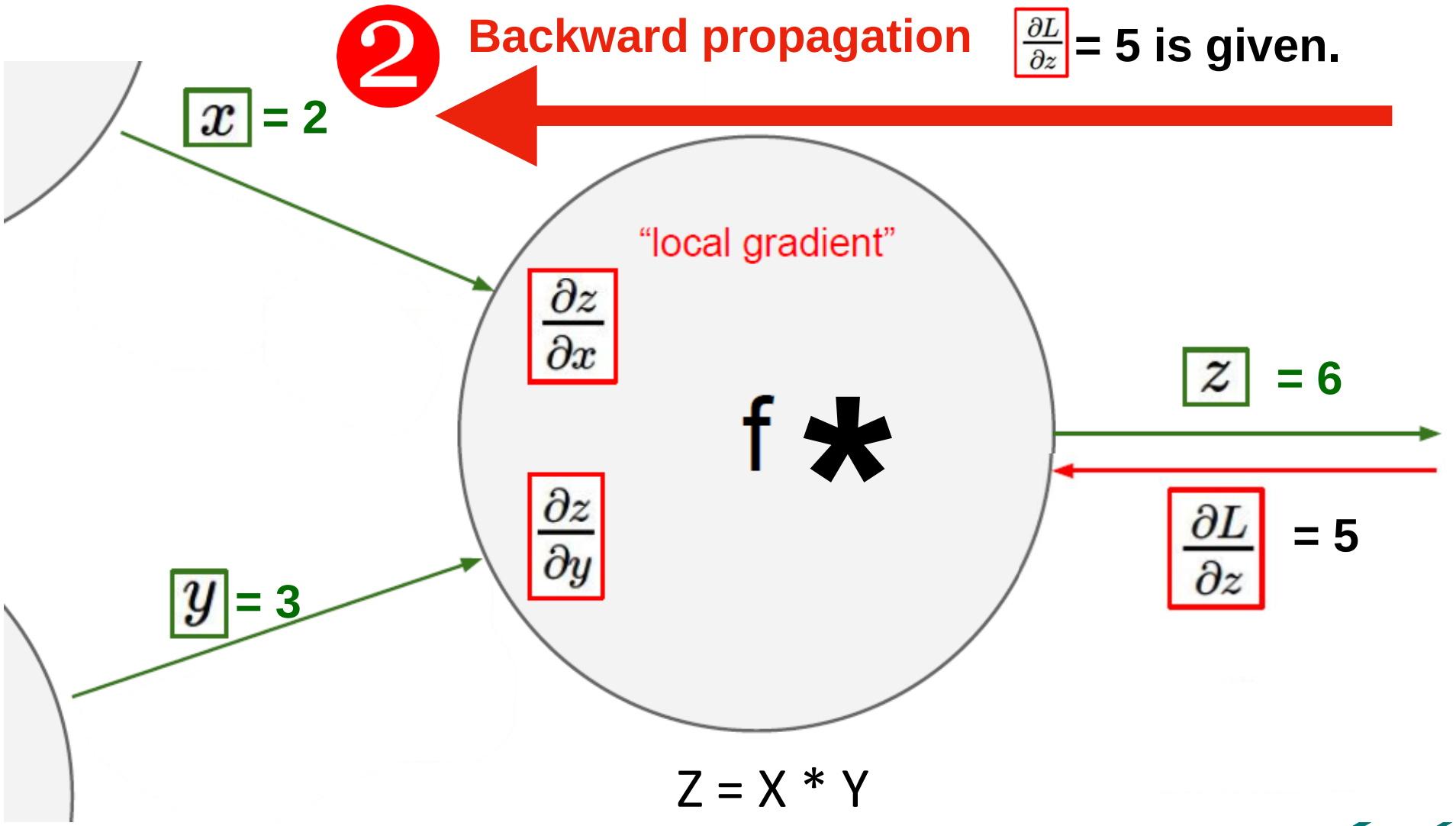
Forward pass



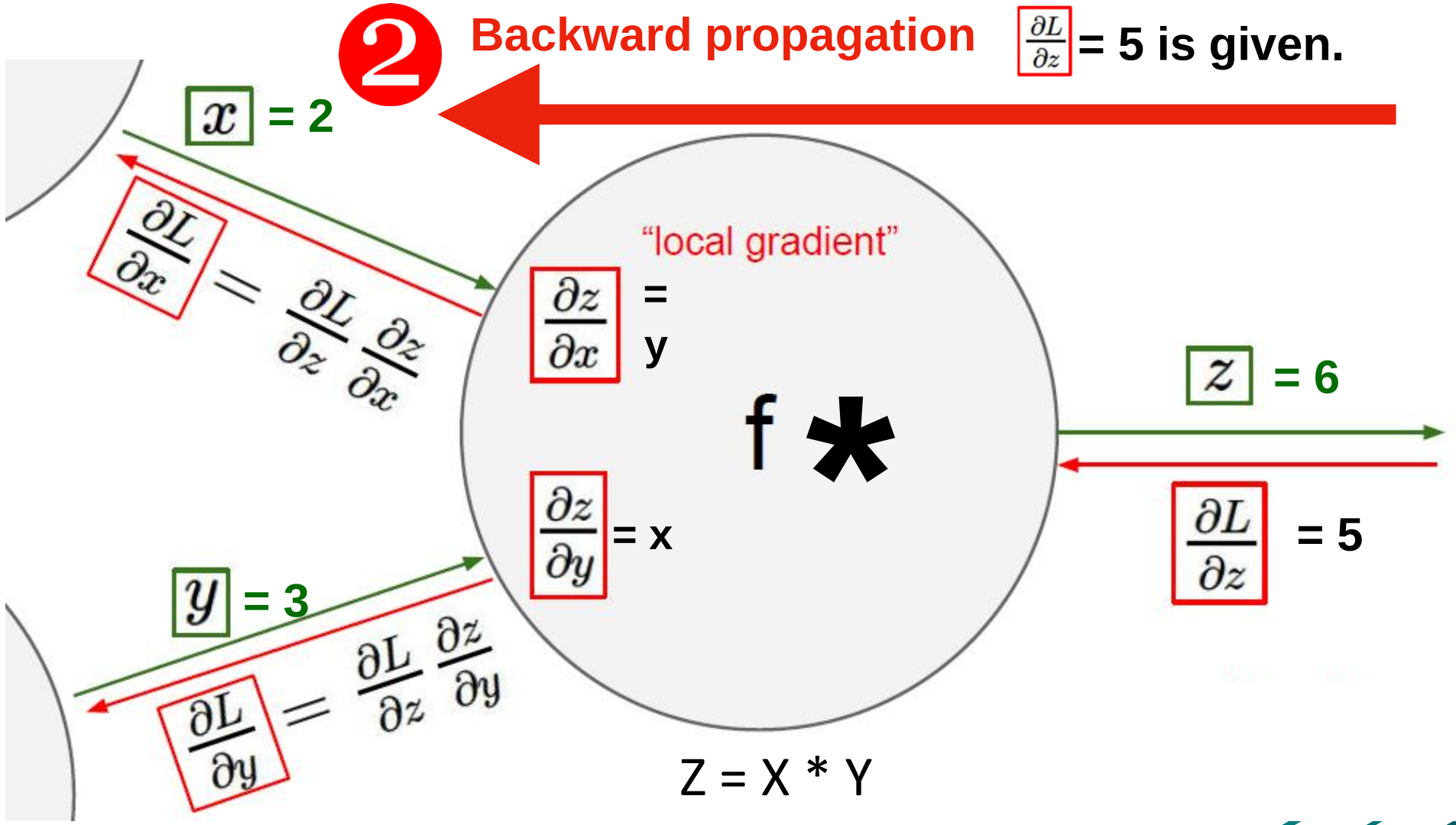
Forward pass



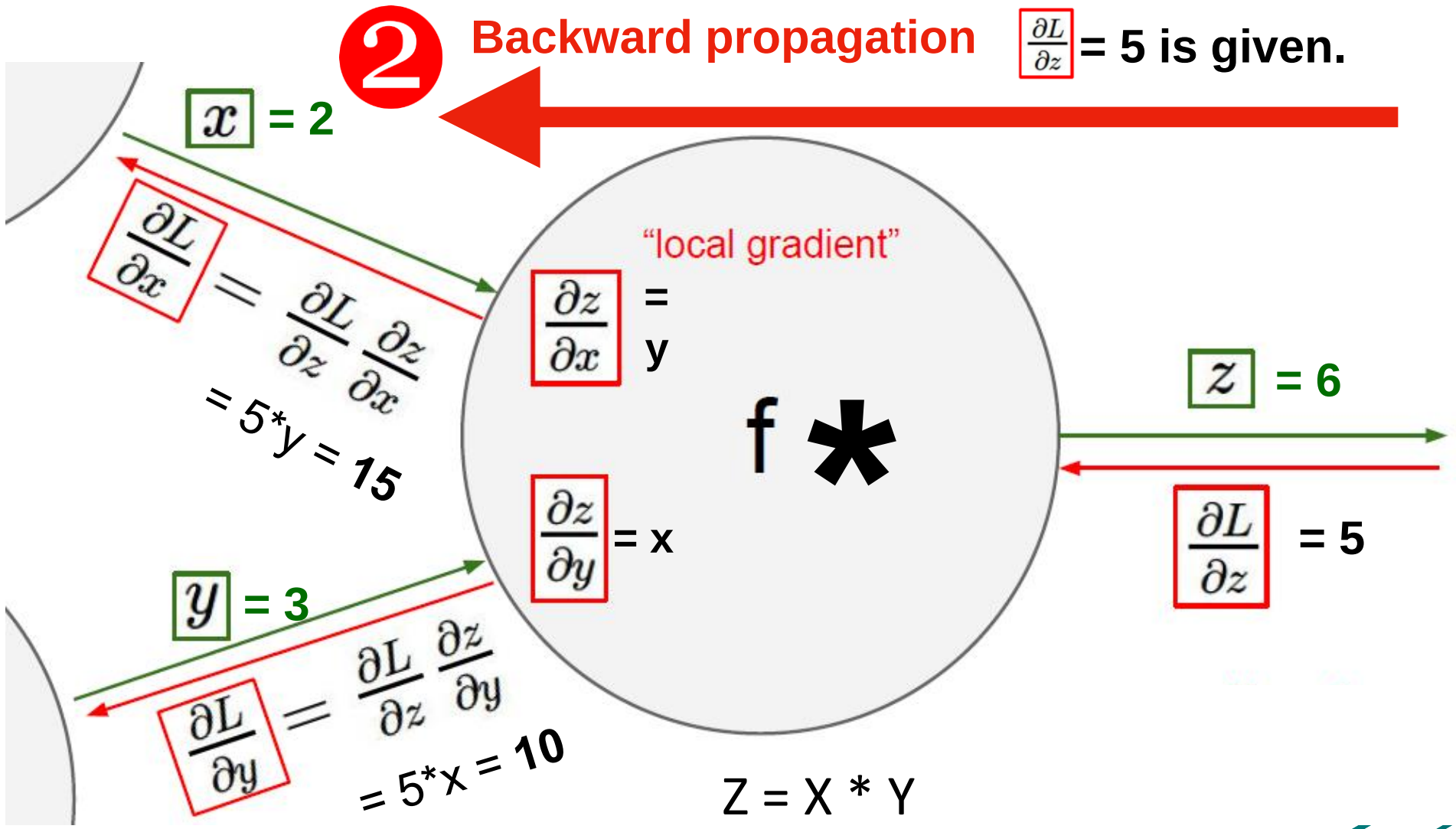
Backward propagation



Backward propagation



Backward Propagation



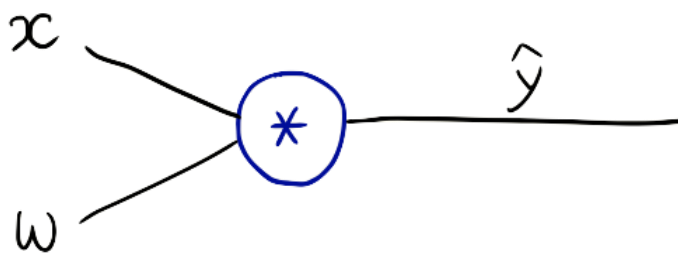
HAMIM

Computational Graph

$$\hat{y} = x * w$$

Computational Graph

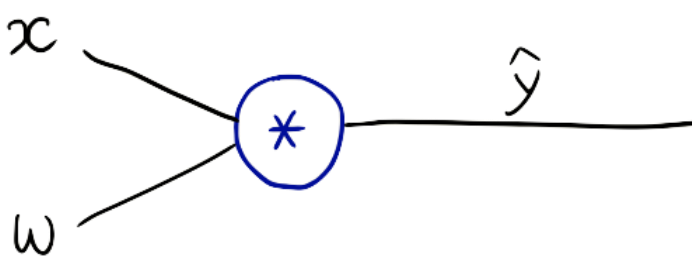
$$\hat{y} = x * w$$



Computational Graph

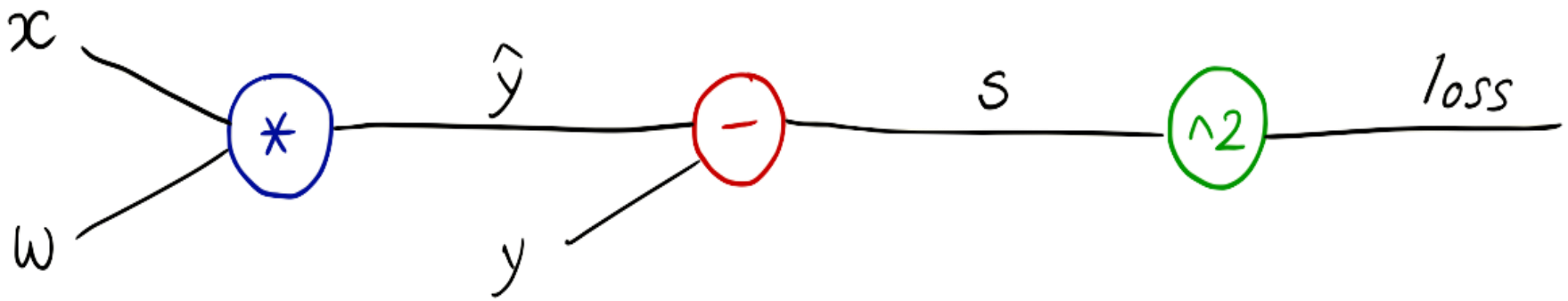
$$\hat{y} = x * w$$

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$



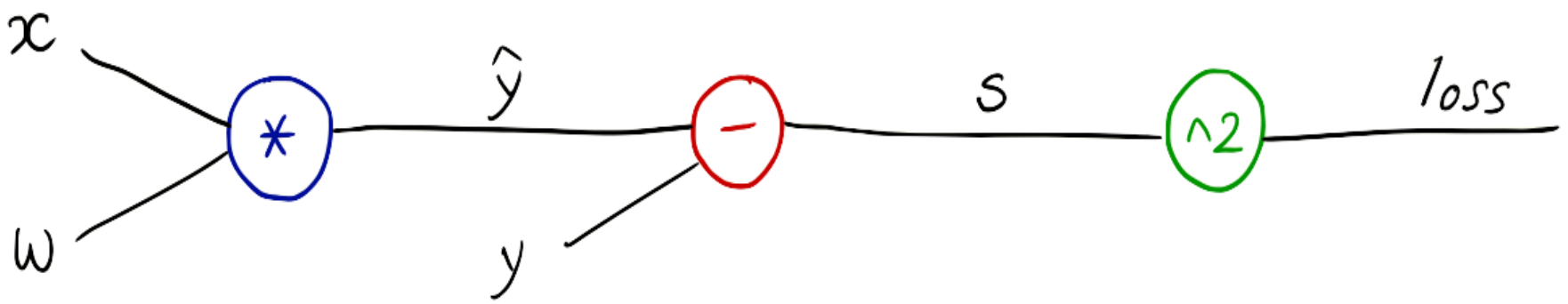
Computational Graph

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$



Computational Graph

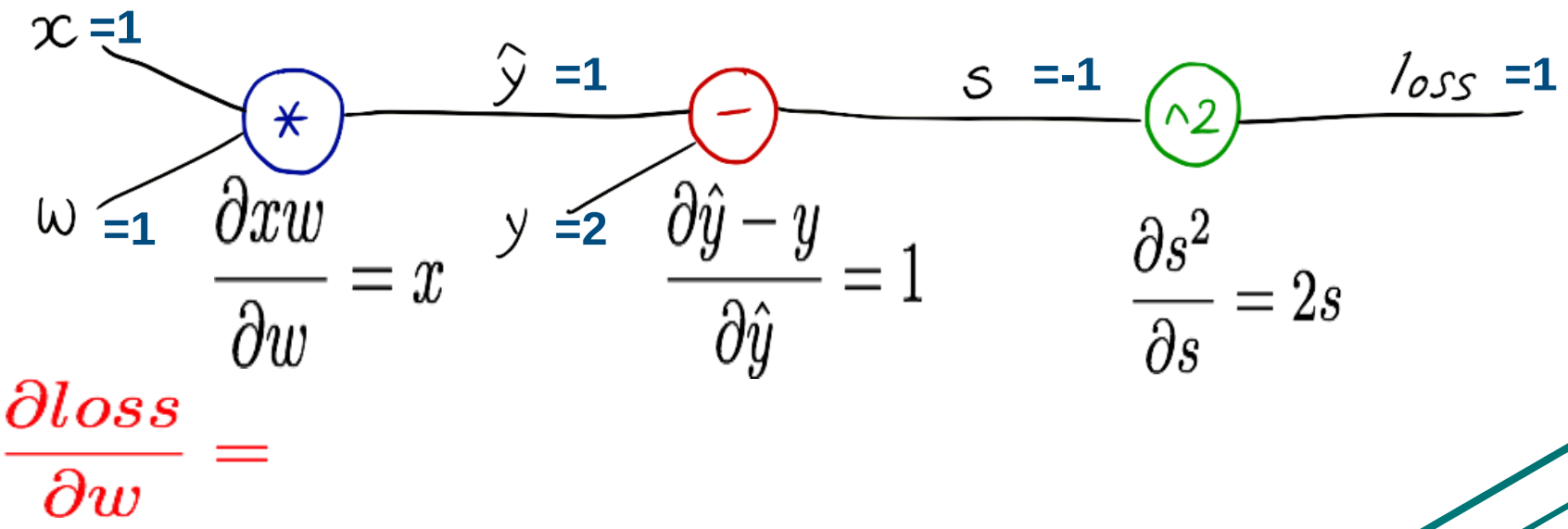
1 **Forward pass x=1, y = 2 where w=1**



Computational Graph

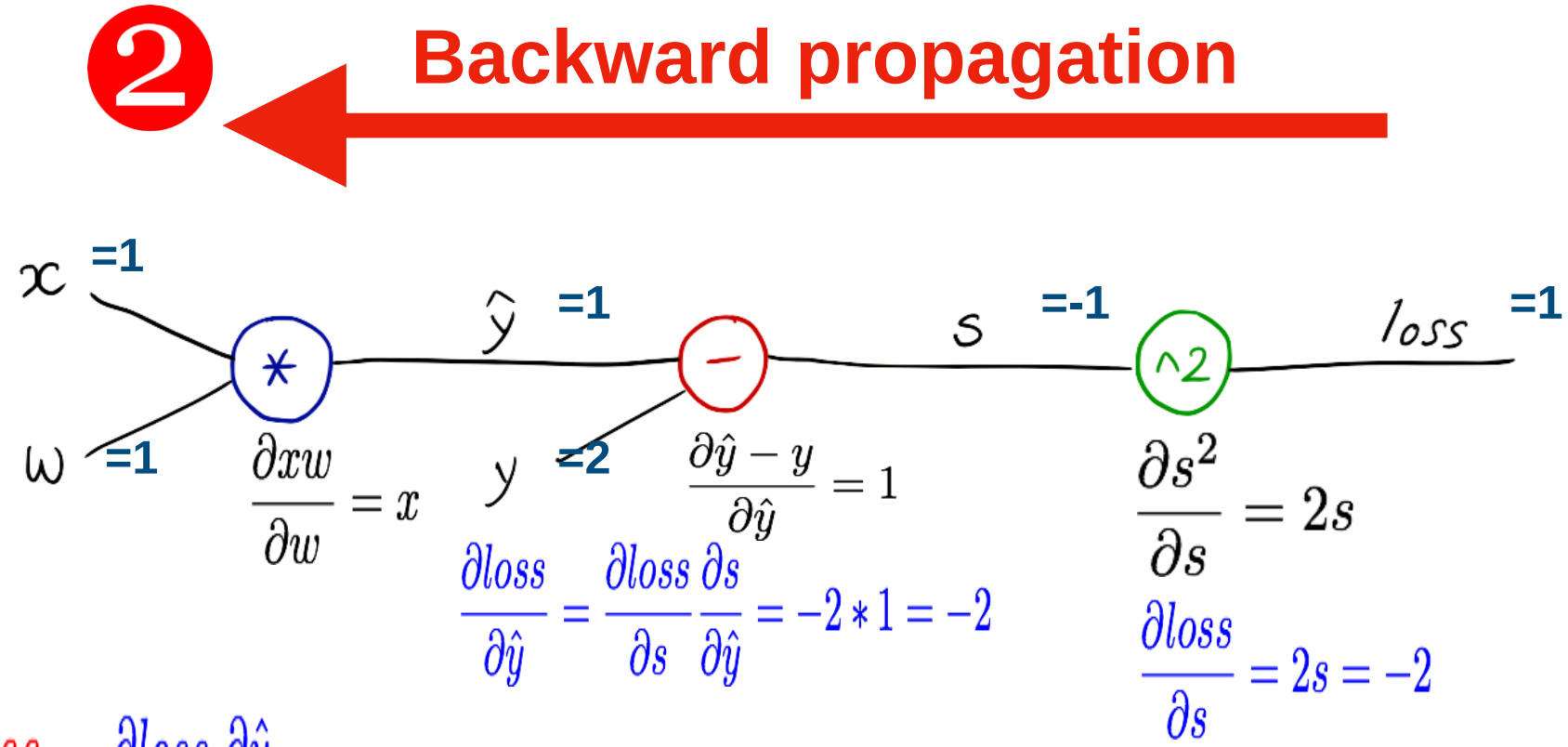
$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

❷ **Backward propagation**



HAMIM©

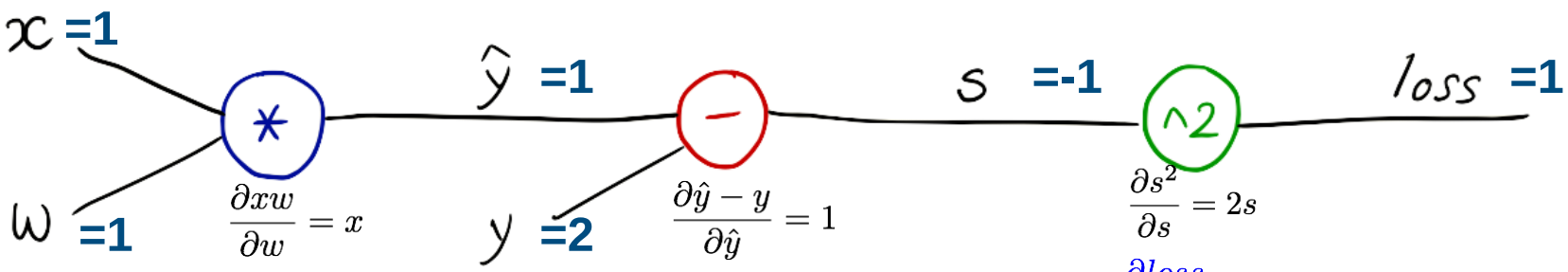
Computational Graph



HAMIM

Computational Graph

❷ Backward propagation



$$\frac{\partial loss}{\partial s} = 2s = -2$$

$$\frac{\partial loss}{\partial \hat{y}} = \frac{\partial loss}{\partial s}\frac{\partial s}{\partial \hat{y}} = -2 * 1 = -2$$

$$\frac{\partial loss}{\partial w} = \frac{\partial loss}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w} = -2 * x = -2 * 1 = -2$$

```
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.84
    grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
```

HAMIM

Data and Variable

```
import torch
from torch.autograd import Variable

x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = Variable(torch.Tensor([1.0]),
requires_grad=True)  # Any random value
```

Data and Variable

A graph is created on the fly

```
from torch.autograd import Variable

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

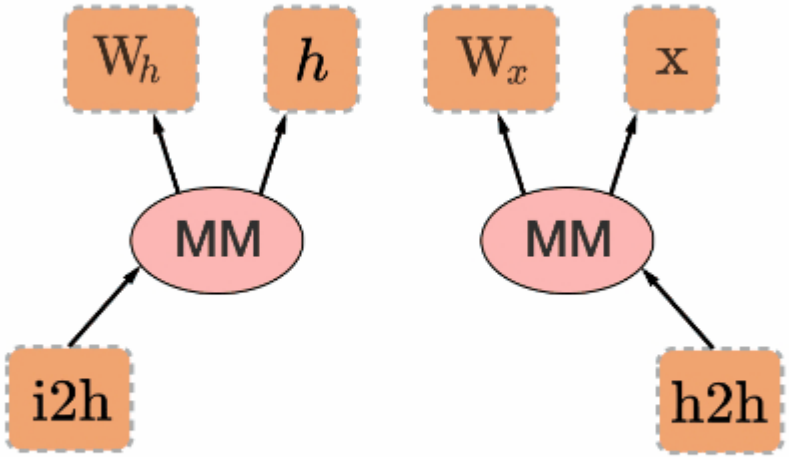
http://pytorch.org/docs/master/notes/autograd.html?highlight=variable

HAMIM

Data and Variable

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

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



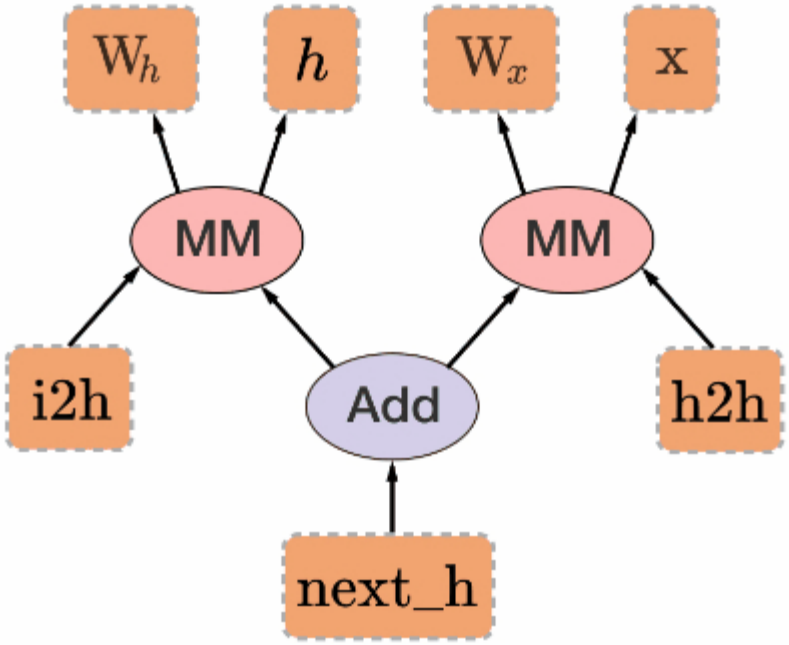
http://pytorch.org/docs/master/notes/autograd.html?highlight=variable

HAMIM

Data and Variable

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

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



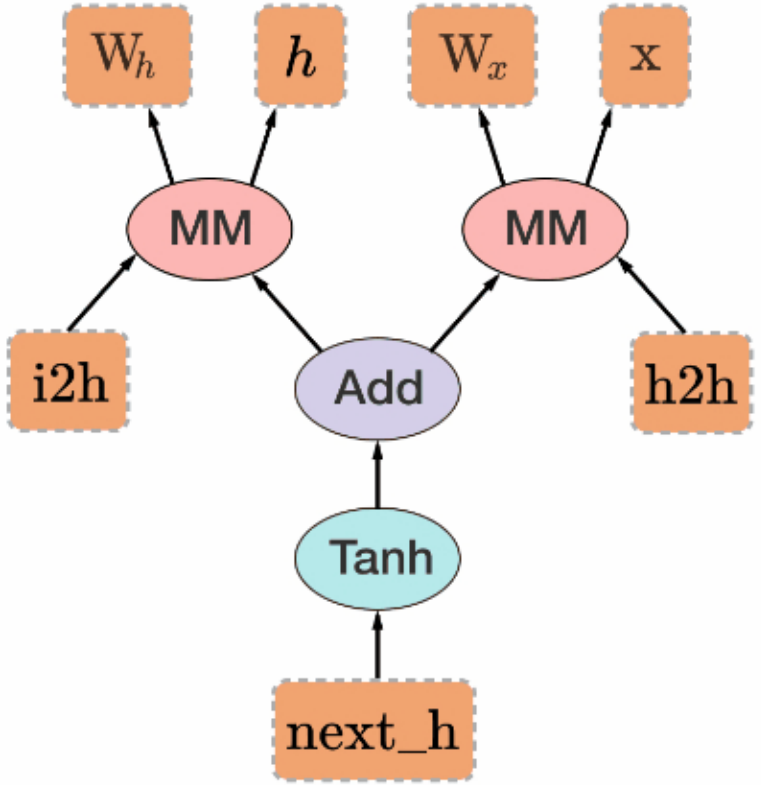
http://pytorch.org/docs/master/notes/autograd.html?highlight=variable

HAMIM

Data and Variable

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

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



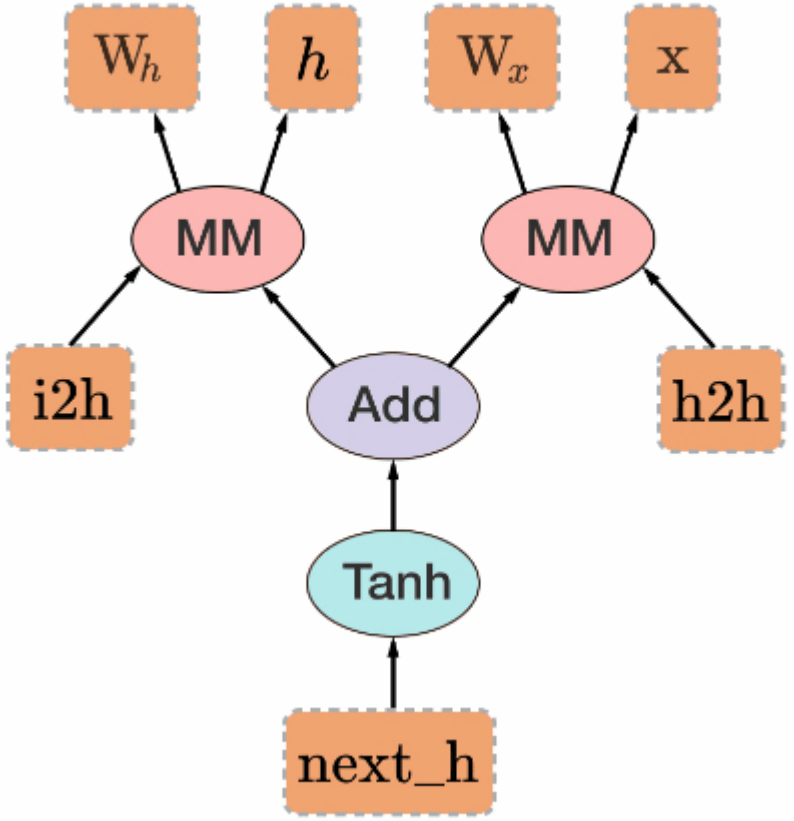
http://pytorch.org/docs/master/notes/autograd.html?highlight=variable

HAMIM

Data and Variable

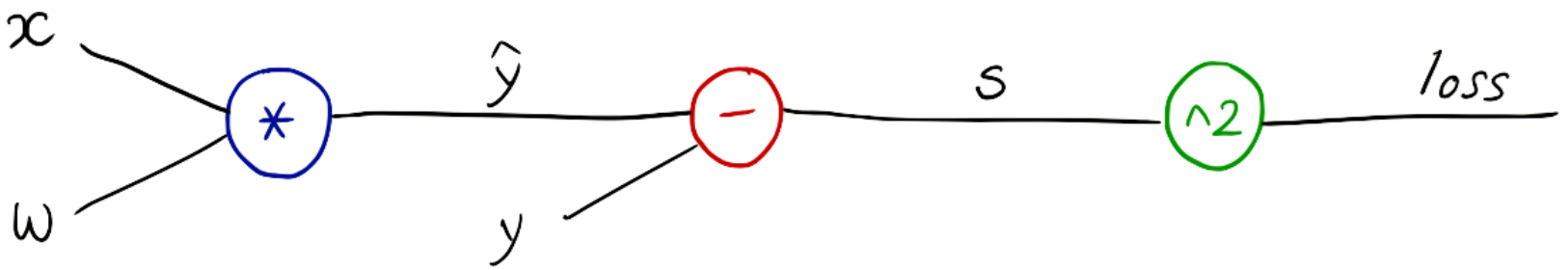
Back-propagation
uses the dynamically built graph

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

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

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



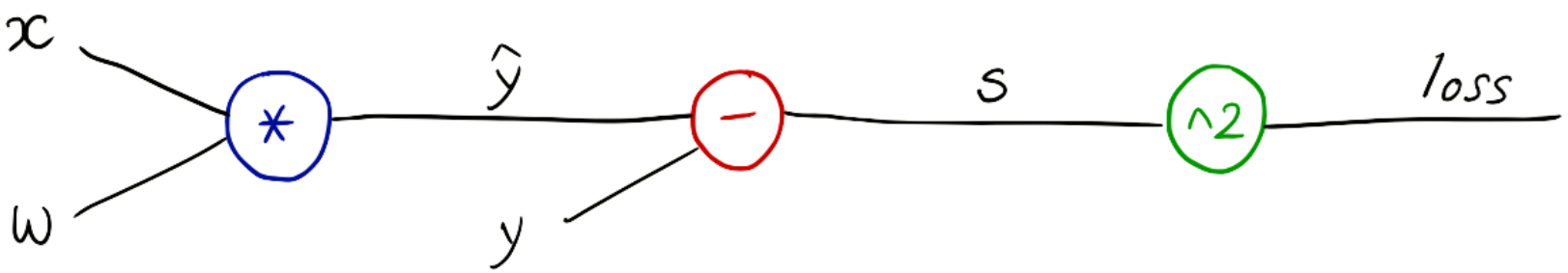
http://pytorch.org/docs/master/notes/autograd.html?highlight=variable

PyTorch forward/backward



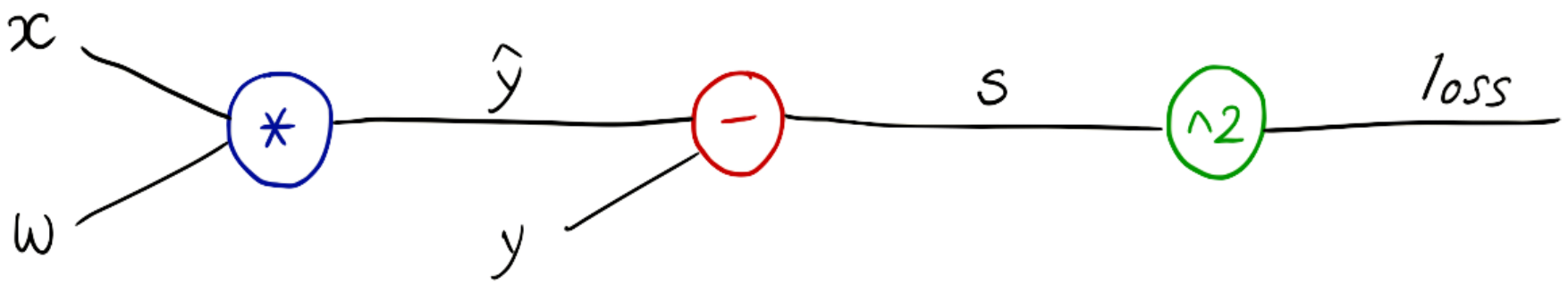
Forward Pass

```
# Any random value
w = Variable(torch.Tensor([1.0]),  requires_grad=True)
l = loss(x=1, y= )
```



$$\frac{\partial loss}{\partial w} =$$

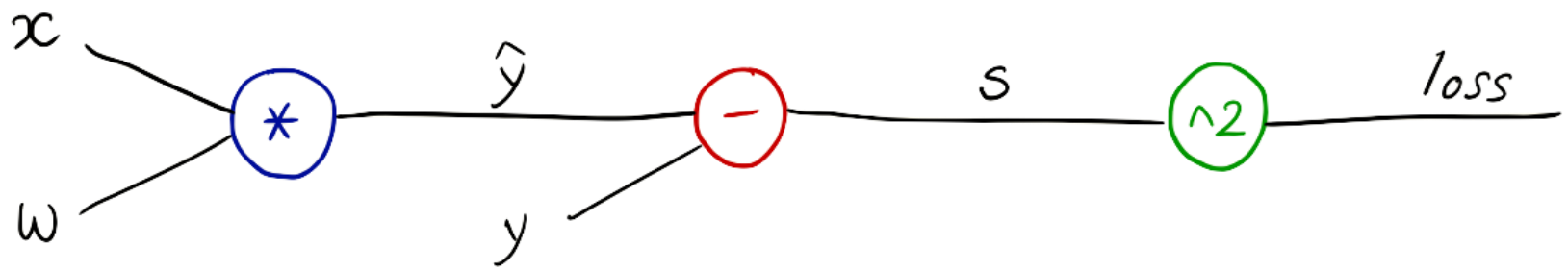
Back propagation : l.backward()



$$\frac{\partial loss}{\partial w} = \texttt{w.grad}$$

Weight update (step)

```
w.data = w.data - 0.01 * w.grad.data
```



$$\frac{\partial loss}{\partial w} = \texttt{w.grad}$$

Model and Loss

```
# our model forward pass
def forward(x):
    return x * w

# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)

# Before training
print("predict (before training)",  4, forward(4).data[0])
```

Training: forward, backward, Update Weight

```
# Training loop
for epoch in range(10):
    for x_val, y_val in zip(x_data, y_data):
        l = loss(x_val, y_val)
        l.backward()
        print("\tgrad: ", x_val, y_val, w.grad.data[0])
        w.data = w.data - 0.01 * w.grad.data

        # Manually zero the gradients after updating weights
        w.grad.data.zero_()

    print("progress:", epoch, l.data[0])

# After training
print("predict (after training)",  4, forward(4).data[0])
```

HAMIM©

Output

```
# Training loop
for epoch in range(10):
    for x_val, y_val in zip(x_data, y_data):
        l = loss(x_val, y_val)
        l.backward()
        print("\tgrad: ", x_val, y_val, w.grad.data[0])
        w.data = w.data - 0.01 * w.grad.data

        # Manually zero the gradients after updating weights
        w.grad.data.zero_()

    print("progress:", epoch, l.data[0])

# After training
print("predict (after training)",  4, forward(4).data[0])
```

```
predict (before training) 4 4.0
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.840000152587891
    grad:  3.0 6.0 -16.228801727294922
progress: 0 7.315943717956543
    grad:  1.0 2.0 -1.478623867034912
    grad:  2.0 4.0 -5.796205520629883
    grad:  3.0 6.0 -11.998146057128906
progress: 1 3.9987640380859375
    grad:  1.0 2.0 -1.0931644439697266
    grad:  2.0 4.0 -4.285204887390137
    grad:  3.0 6.0 -8.870372772216797
progress: 2 2.1856532096862793
    grad:  1.0 2.0 -0.8081896305084229
    grad:  2.0 4.0 -3.1681032180786133
    grad:  3.0 6.0 -6.557973861694336
progress: 3 1.1946394443511963
    grad:  1.0 2.0 -0.5975041389465332
    grad:  2.0 4.0 -2.3422164916992188
    grad:  3.0 6.0 -4.848389625549316
progress: 4 0.6529689431190491
    grad:  1.0 2.0 -0.4417421817779541
    grad:  2.0 4.0 -1.7316293716430664
    grad:  3.0 6.0 -3.58447265625
progress: 5 0.35690122842788696
    grad:  1.0 2.0 -0.3265852928161621
    grad:  2.0 4.0 -1.2802143096923828
    grad:  3.0 6.0 -2.650045394897461
progress: 6 0.195076122879982
    grad:  1.0 2.0 -0.24144840240478516
    grad:  2.0 4.0 -0.9464778900146484
    grad:  3.0 6.0 -1.9592113494873047
progress: 7 0.10662525147199631
    grad:  1.0 2.0 -0.17850565910339355
    grad:  2.0 4.0 -0.699742317199707
    grad:  3.0 6.0 -1.4484672546386719
```

Output (from numeric & gradient Descent)

```
predict (before training) 4 4.0
     grad:  1.0 2.0 -2.0
     grad:  2.0 4.0 -7.84
     grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
     grad:  1.0 2.0 -1.478624
     grad:  2.0 4.0 -5.796206079999999
     grad:  3.0 6.0 -11.998146585599997
progress: 1 2.688769240265834
     grad:  1.0 2.0 -1.093164466688
     grad:  2.0 4.0 -4.285204709416961
     grad:  3.0 6.0 -8.87037374849311
progress: 2 1.4696334962911515
     grad:  1.0 2.0 -0.8081896081960389
     grad:  2.0 4.0 -3.1681032641284723
     grad:  3.0 6.0 -6.557973756745939
progress: 3 0.8032755585999681
     grad:  1.0 2.0 -0.59750427561463
     grad:  2.0 4.0 -2.3422167604093502
     grad:  3.0 6.0 -4.848388694047353
progress: 4 0.43905614881022015
     grad:  1.0 2.0 -0.44174208101320334
     grad:  2.0 4.0 -1.7316289575717576
     grad:  3.0 6.0 -3.584471942173538
progress: 5 0.2399802903801062
     grad:  1.0 2.0 -0.3265852213980338
     grad:  2.0 4.0 -1.2802140678802925
     grad:  3.0 6.0 -2.650043120512205
progress: 6 0.1311689630744999
     grad:  1.0 2.0 -0.241448373202223
     grad:  2.0 4.0 -0.946477622952715
     grad:  3.0 6.0 -1.9592086795121197
progress: 7 0.07169462478267678
     grad:  1.0 2.0 -0.17850567968888198
     grad:  2.0 4.0 -0.6997422643804168
     grad:  3.0 6.0 -1.4484664872674653
progress: 8 0.03918700813247573
     grad:  1.0 2.0 -0.13197139106214673
     grad:  2.0 4.0 -0.5173278529636143
     grad:  3.0 6.0 -1.0708686556346834
progress: 9 0.021418922423117836
predict (after training) 4 7.804863933862125
```

```python
# Before training
print("predict (before training)",  4, forward(4))

# Training loop
for epoch in range(10):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)

    print ("progress:", epoch, l)

# After training
print("predict (after training)",  4, forward(4))
```

HAMIM©

Classification Problem

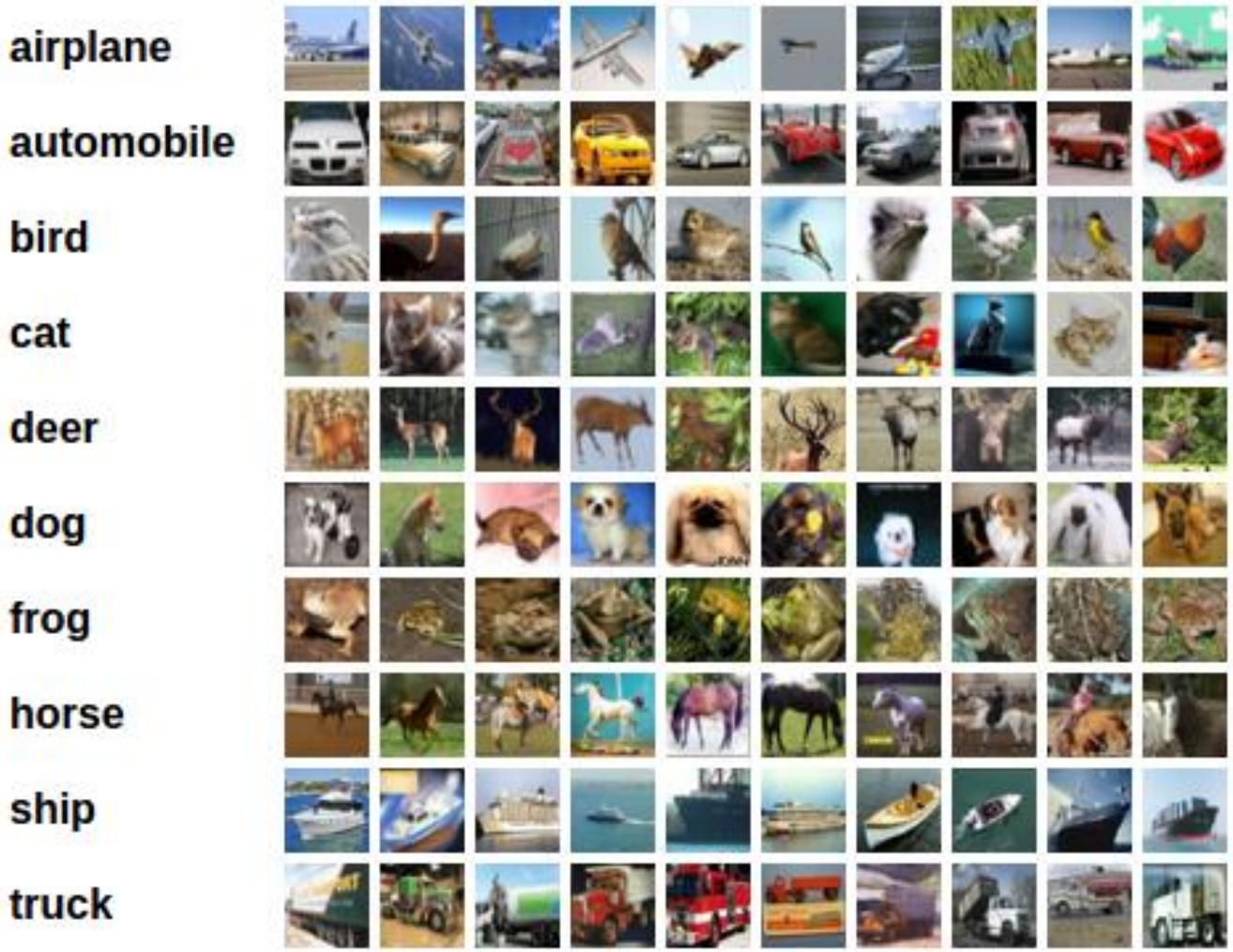


Image Size = 3×32×32

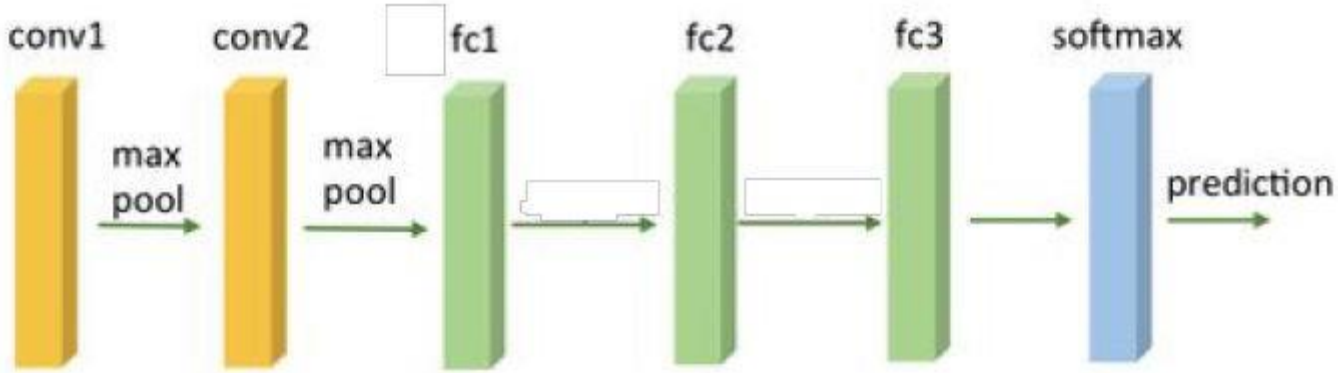
PyTorch Rhythm

1. **Design your model using class with Variables**
2. **Construct loss and optimizer (select from PyTorch API)**
3. **Training cycle (forward, backward, update)**

Classification Problem

1 **Design your model using class with Variables**

```
#########################################################################
# 1. Define a Neural Network
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Copy the neural network from the Neural Networks section before and modify it to
# take 3-channel images (instead of 1-channel images as it was defined).

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()

#########################################################################
```



HAMIM

Cifar10 classification

❷ Construct loss and optimizer (select from PyTorch API)

```
########################################################################
# 2. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#
# Let's use a Classification Cross-Entropy loss and SGD with momentum
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

########################################################################
```

Cifar10 classification

```
########################################################################
# 3. Train the network
# ^^^^^^^^^^^^^^^^^^^^
#
# This is when things start to get interesting.
# We simply have to loop over our data iterator, and feed the inputs to the
# network and optimize
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.data[0]
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

3 **Training cycle (forward, backward, update)**

HAMIM

Torch.nn for Convolution layers structure

- m = nn.Conv2d(16, 33, 3, stride=2,padding=3)
- Parameters:
- in_channels (int) – Number of channels in the input image
- out_channels (int) – Number of channels produced by the convolution
- kernel_size (int or tuple) – Size of the convolving kernel
- stride (int or tuple, optional) – Stride of the convolution. Default: 1
- padding (int or tuple, optional) – Zero-padding added to both sides of the input. Default: 0

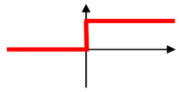
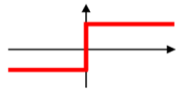
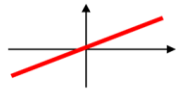
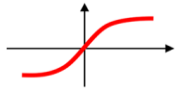
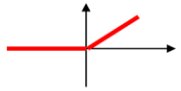
Other Layers

- Convolution Layers
- Pooling Layers
- Padding Layers
- Non-linear Activations
- Normalization Layers
- Recurrent layers
- Linear Layers
- Dropout Layers
- Sparse layers
- Vision layers

Torch Optimization Function

- torch.optim.Adagrad
- torch.optim.Adam
- torch.optim.Adamax
- torch.optim.ASGD
- torch.optim.LBFGS
- torch.optim.RMSprop
- torch.optim.Rprop
- torch.optim.SGD

Activation Functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

http://rasbt.github.io/mlxtend/user_guide/general_concepts/activation-functions/

Non-linear Activations

- ReLU
- ReLU6
- ELU
- SELU
- PReLU
- LeakyReLU
- Threshold
- Hardtanh
- Sigmoid
- Tanh
- LogSigmoid
- Softplus
- Softshrink
- Softsign
- Tanhshrink
- Softmin
- Softmax
- Softmax2d
- LogSoftmax

HAMIM

jupyter

HAMIM©

Data Loader _ Manual Data Feed

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',',
dtype=np.float32)
x_data = Variable(torch.from_numpy(xy[:, 0:-1]))
y_data = Variable(torch.from_numpy(xy[:, [-1]]))

# Training loop
for epoch in range(100):
        # Forward pass: Compute predicted y by passing x to the
model
    y_pred = model(x_data)
    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])
    # Zero gradients, perform a backward pass, and update the
weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Batch (batch size)

```
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
        batch_xs, batch_ys = …
```

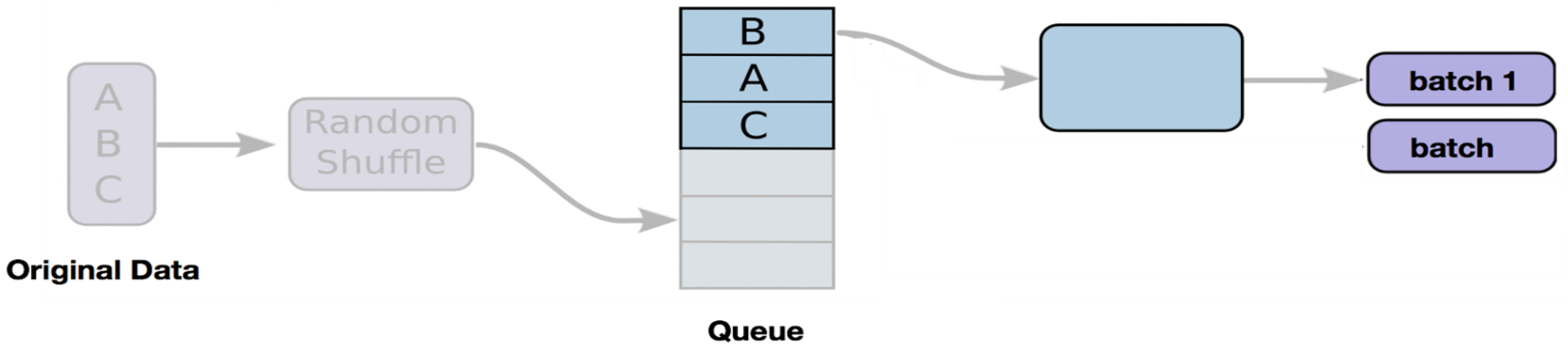
288

In the neural network terminology:

- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Data Loader




```
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs),
Variable(labels)

        # Run your training process
        print(epoch, i, "inputs", inputs.data,
"labels", labels.data)
```

HAMIM

Custom Data Loader

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):


    def __getitem__(self, index):
        return

    def __len__(self):
        return


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

1 download, read data, etc.

2 return one item on the index

3 return the data length

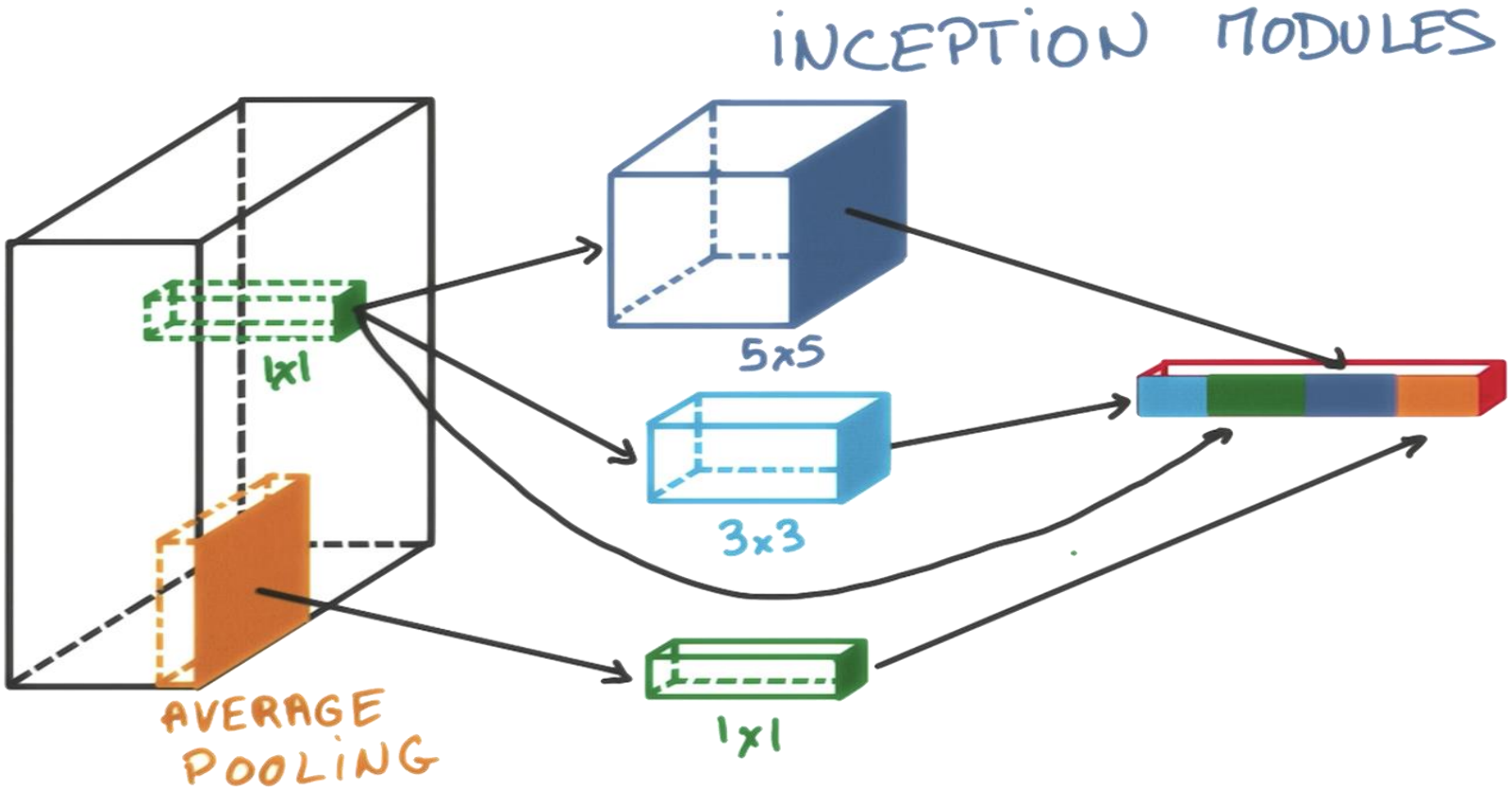
Custom DataLoader

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

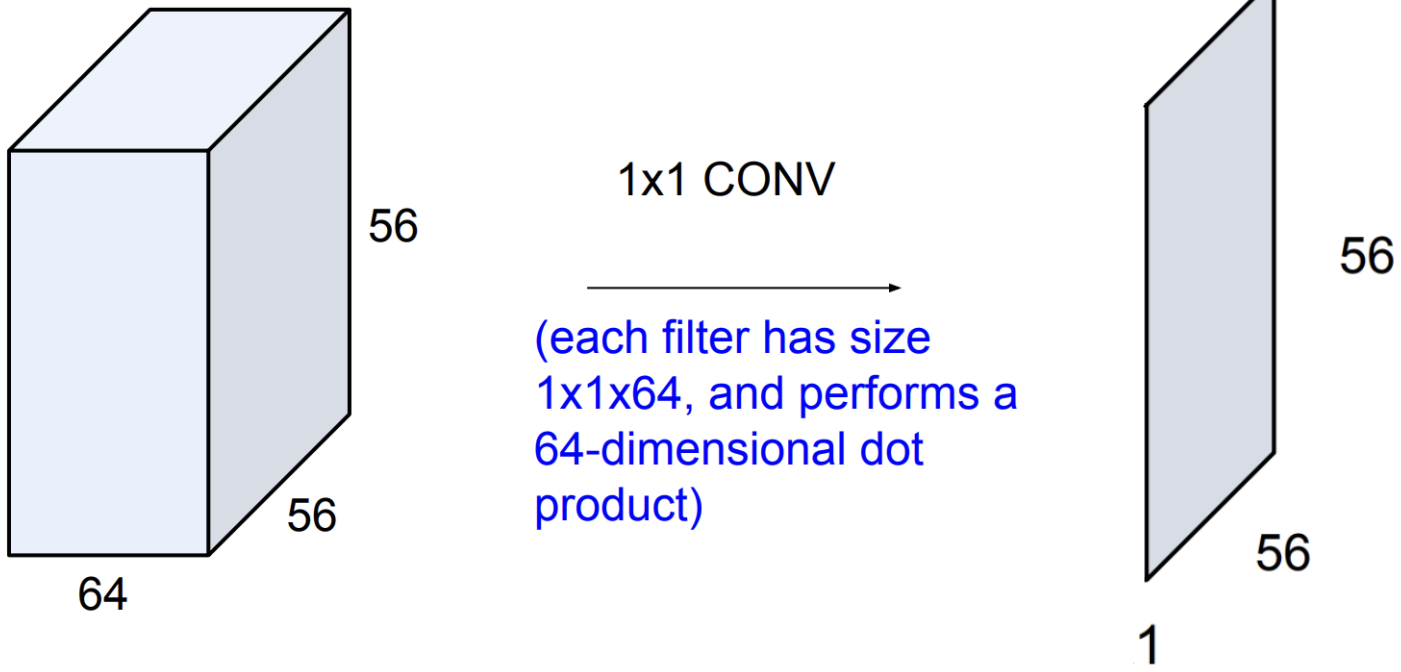
Using DataLoader

```
dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset, batch_size=32, shuffle=True, num_workers=2)


# Training loop
for epoch in range(2):
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # Forward pass: Compute predicted y by passing x to the model
        y_pred = model(inputs)

        # Compute and print loss
        loss = criterion(y_pred, labels)
        print(epoch, i, loss.data[0])

        # Zero gradients, perform a backward pass, and update the weights.
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

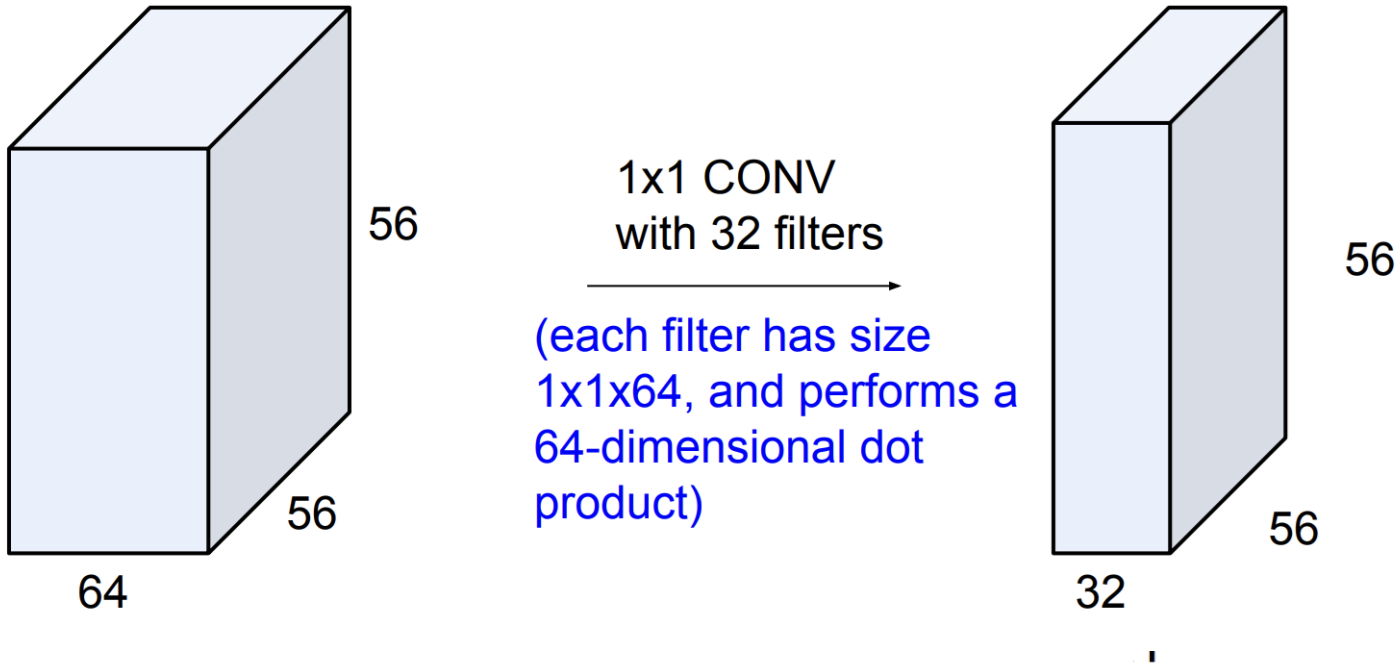
Inception Modules



Why 1x1 convolution?



Why 1x1 convolution?



Why 1x1 convolution?

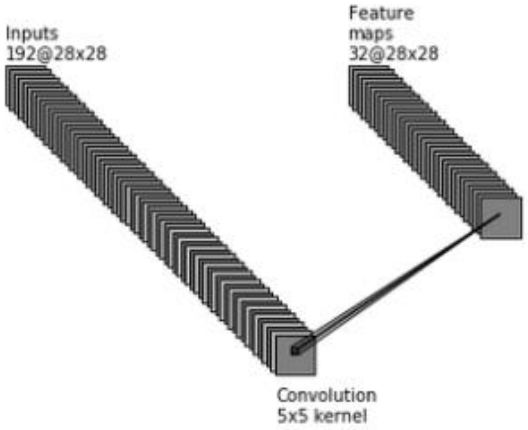


Figure 6. 5×5 convolutions inside the Inception module using the naive model

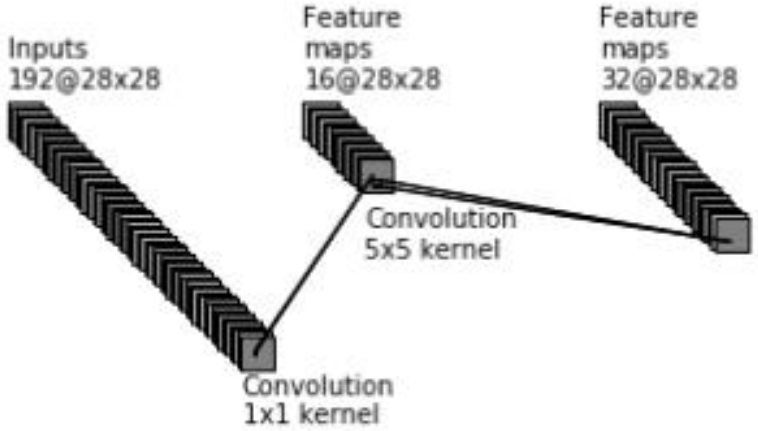


Figure 7. 1×1 convolutions serve as the dimensionality reducers that limit the number of expensive 5×5 convolutions that follow

HAMIM©

Why 1x1 convolution?

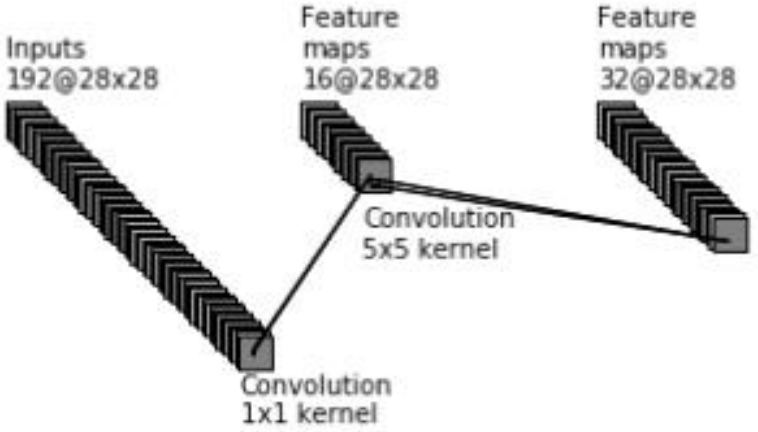


Figure 7. 1×1 convolutions serve as the dimensionality reducers that limit the number of expensive 5×5 convolutions that follow

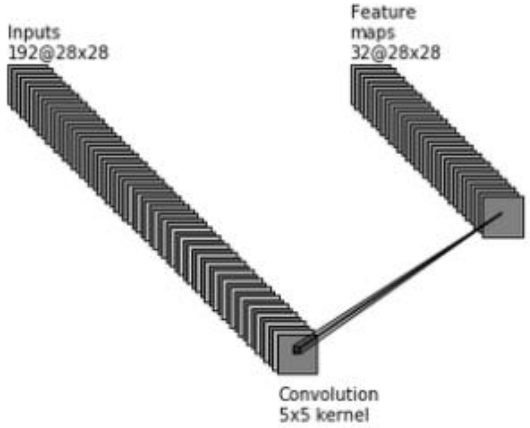


Figure 6. 5×5 convolutions inside the Inception module using the naive model

Without 1x1 convolution:

$$5^2 * 28^2 * 192 * 32 = 120,422,400 \text{ operations}$$

HAMIM

Why 1x1 convolution?

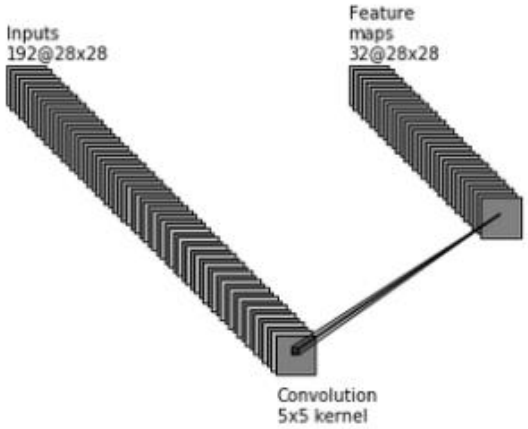


Figure 6. 5×5 convolutions inside the Inception module using the naive model

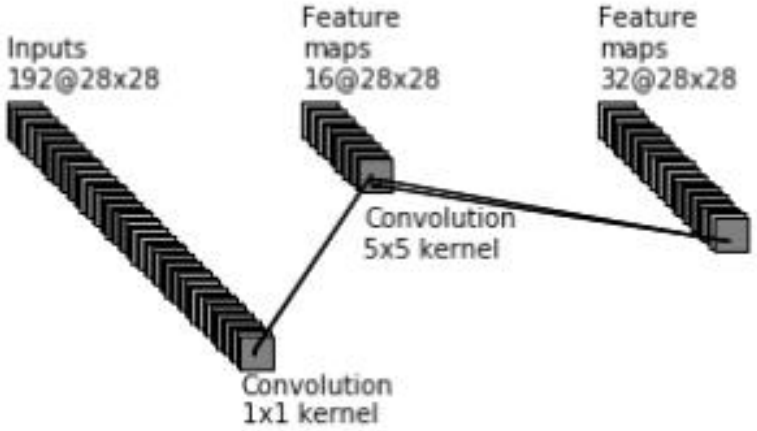


Figure 7. 1×1 convolutions serve as the dimensionality reducers that limit the number of expensive 5×5 convolutions that follow

Without 1x1 convolution:

$5^2 * 28^2 * 192 * 32 = 120,422,400$ operations

With 1x1 convolution:

$1^2 * 28^2 * 192 * 16$

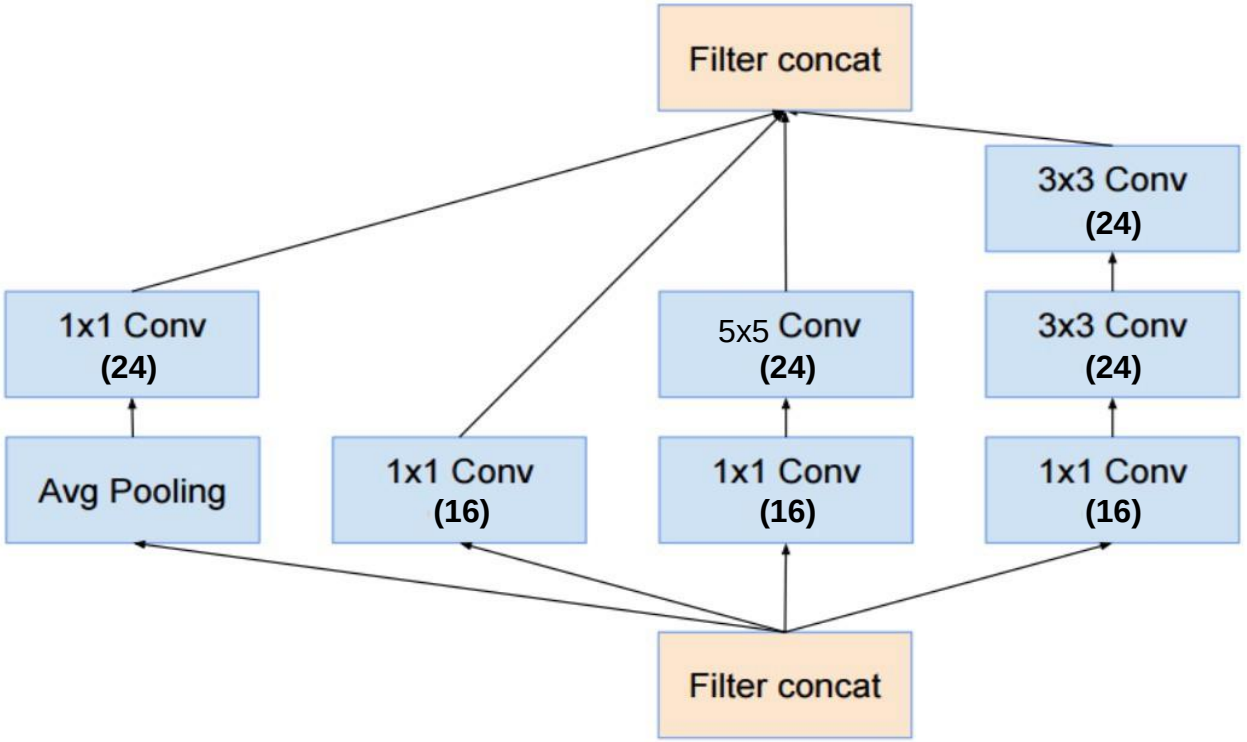
$+ 5^2 * 28^2 * 16 * 32$

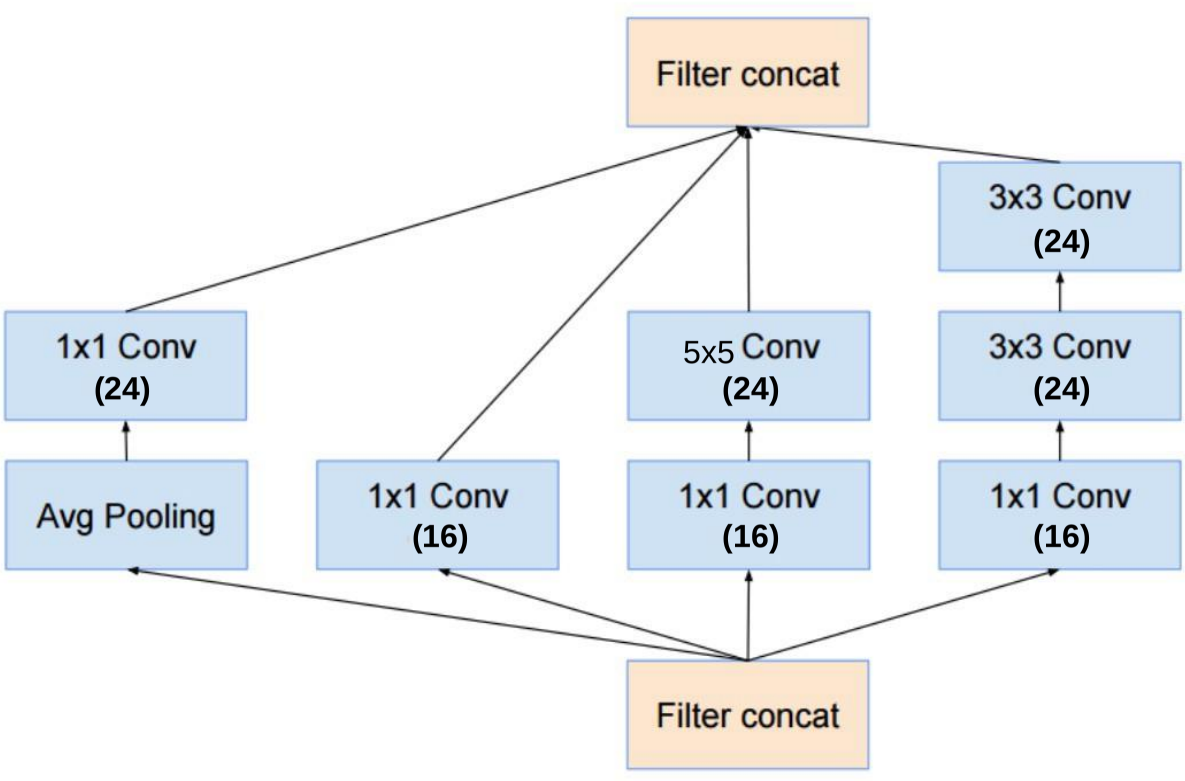
$= 12,443,648$ operations

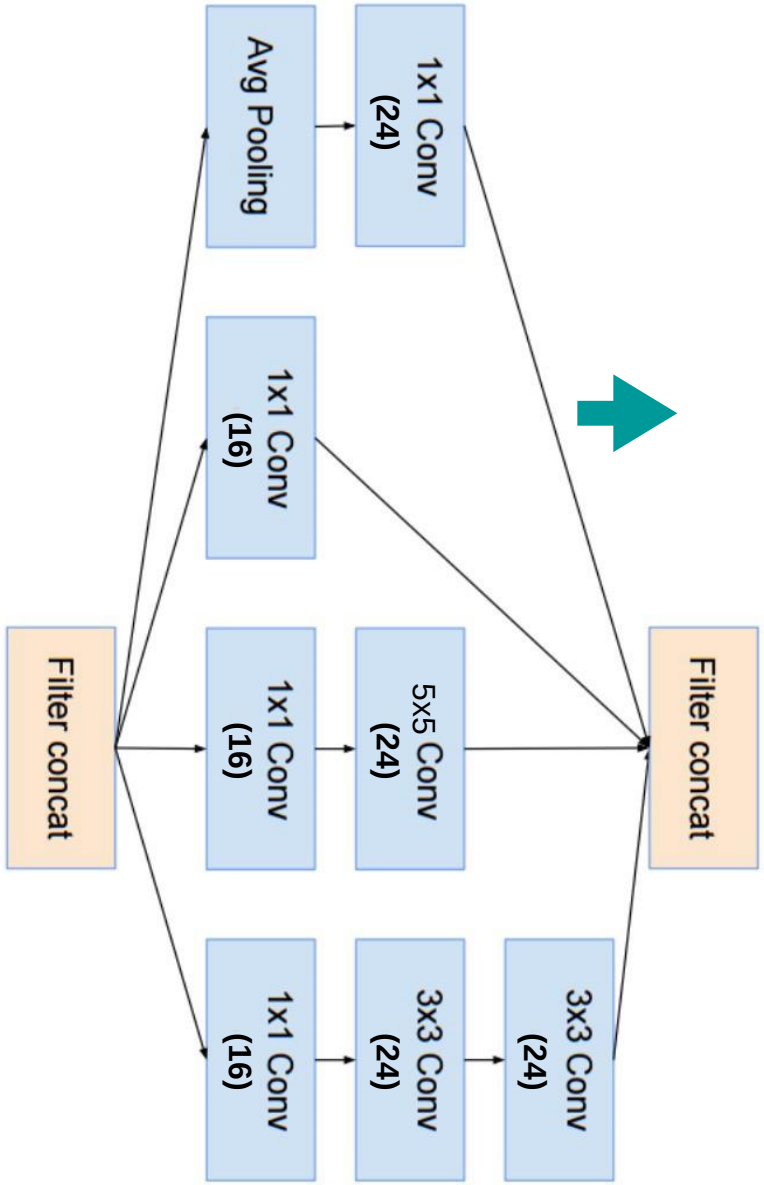
https://hacktilldawn.com/2016/09/25/inception-modules-explained-and-implemented/

HAMIM

Inception Module

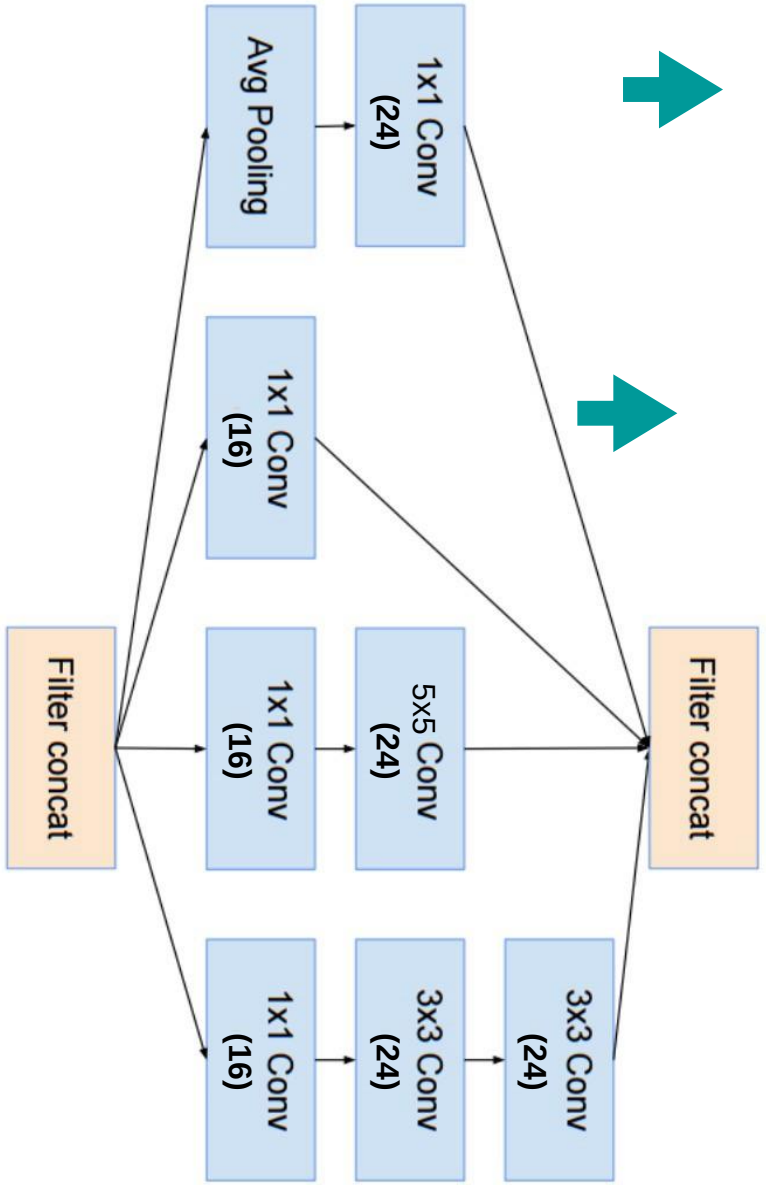



Filter concat
3x3 Conv
(24)
1x1 Conv
(24)
5x5 Conv
(24)
3x3 Conv
(24)
Avg Pooling
1x1 Conv
(16)
1x1 Conv
(16)
1x1 Conv
(16)
Filter concat




```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

branch1x1 = self.branch1x1(x)
```

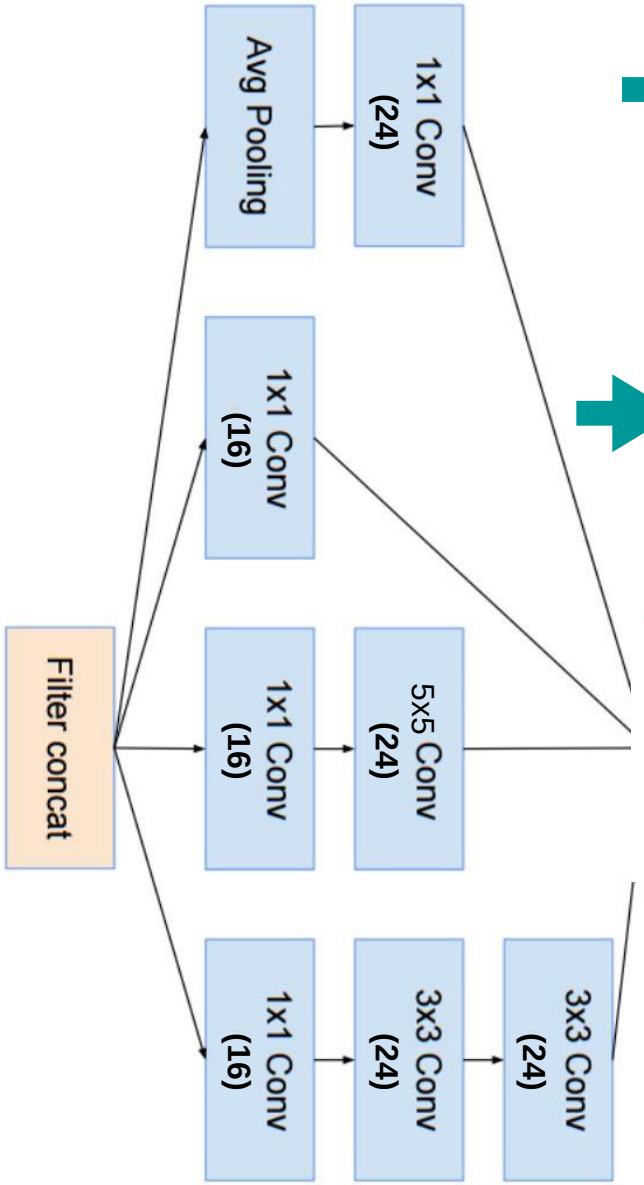
HAMIM




```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)
branch1x1 = self.branch1x1(x)
```

HAMIM

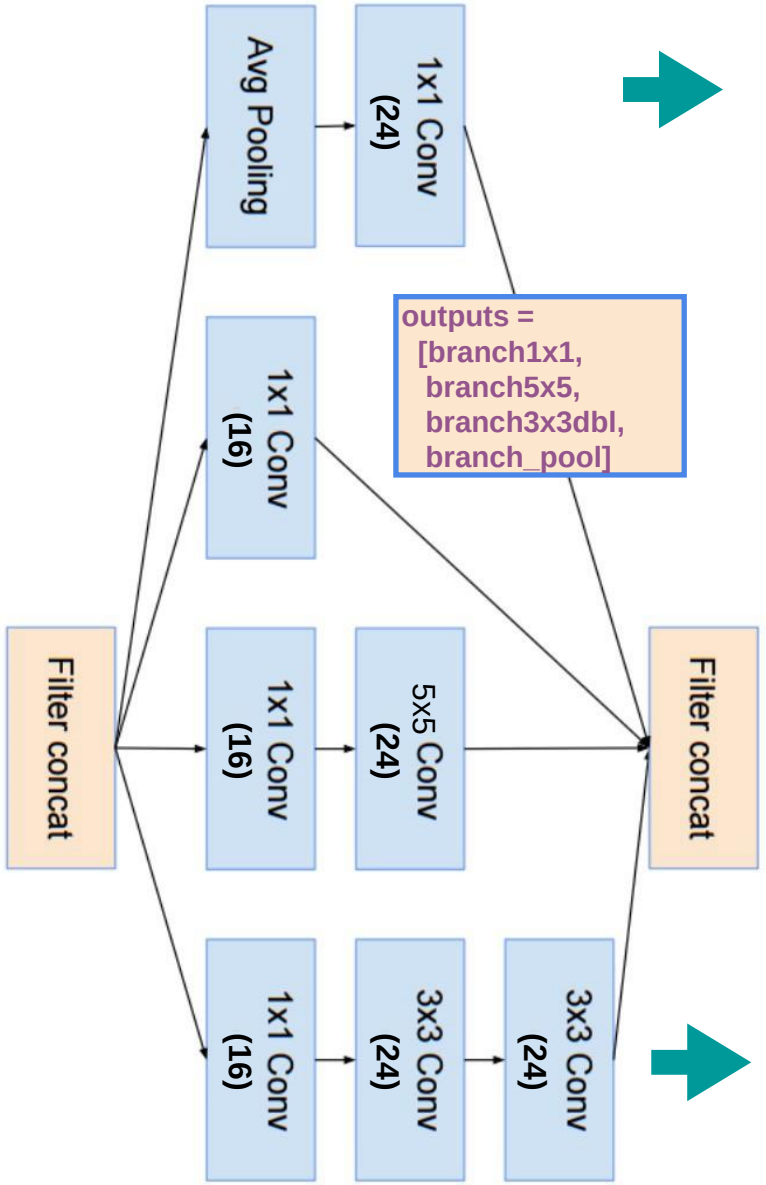



```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)
branch1x1 = self.branch1x1(x)
```

```
self.branch5x5_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

branch5x5 = self.branch5x5_1(x)
branch5x5 = self.branch5x5_2(branch5x5)
```

```
self.branch3x3dbl_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch3x3dbl_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
self.branch3x3dbl_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

branch3x3dbl = self.branch3x3dbl_1(x)
branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
branch3x3dbl = self.branch3x3dbl_3(branch3x3dbl)
```




```
self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
branch_pool = self.branch_pool(branch_pool)
```

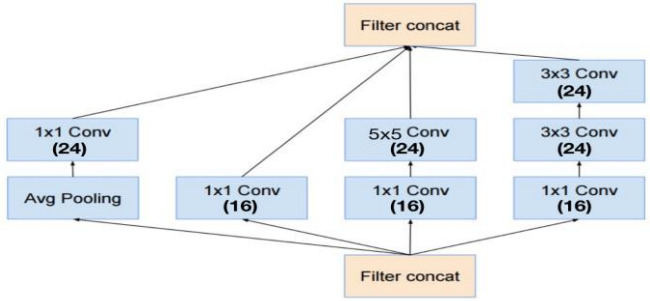
```
self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

branch1x1 = self.branch1x1(x)
```

```
self.branch5x5_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

branch5x5 = self.branch5x5_1(x)
branch5x5 = self.branch5x5_2(branch5x5)
```

```
self.branch3x3dbl_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
self.branch3x3dbl_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
self.branch3x3dbl_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

branch3x3dbl = self.branch3x3dbl_1(x)
branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
branch3x3dbl = self.branch3x3dbl_3(branch3x3dbl)
```

HAMIM

Inception Module




```python
class InceptionA(nn.Module):
    def __init__(self, in_channels):
        super(InceptionA, self).__init__()
        self.branch1x1 = nn.Conv2d(in_channels, 16, kernel_size=1)

        self.branch5x5_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch5x5_2 = nn.Conv2d(16, 24, kernel_size=5, padding=2)

        self.branch3x3dbl_1 = nn.Conv2d(in_channels, 16, kernel_size=1)
        self.branch3x3dbl_2 = nn.Conv2d(16, 24, kernel_size=3, padding=1)
        self.branch3x3dbl_3 = nn.Conv2d(24, 24, kernel_size=3, padding=1)

        self.branch_pool = nn.Conv2d(in_channels, 24, kernel_size=1)

    def forward(self, x):
        branch1x1 = self.branch1x1(x)

        branch5x5 = self.branch5x5_1(x)
        branch5x5 = self.branch5x5_2(branch5x5)

        branch3x3dbl = self.branch3x3dbl_1(x)
        branch3x3dbl = self.branch3x3dbl_2(branch3x3dbl)
        branch3x3dbl = self.branch3x3dbl_3(branch3x3dbl)

        branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
        branch_pool = self.branch_pool(branch_pool)

        outputs = [branch1x1, branch5x5, branch3x3dbl, branch_pool]
        return torch.cat(outputs, 1)
```

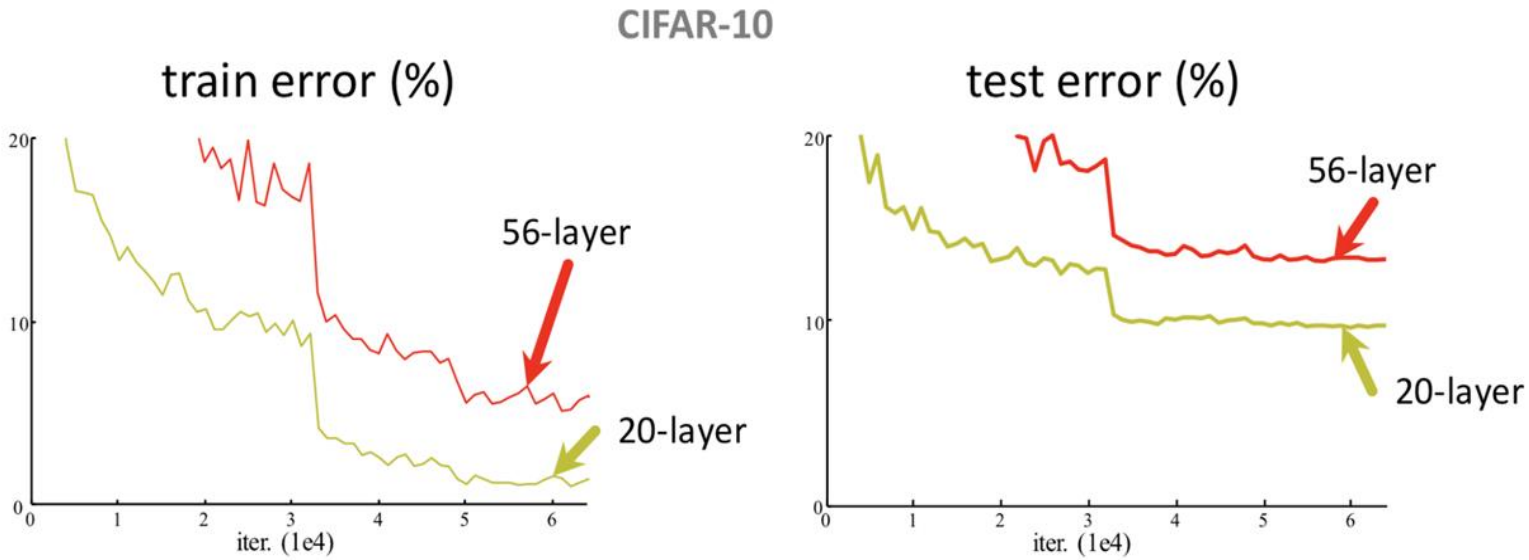
```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10,
kernel_size=5)
        self.conv2 = nn.Conv2d(88, 20,
kernel_size=5)

        self.incept1 = InceptionA(in_channels=10)
        self.incept2 = InceptionA(in_channels=20)

        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(1408, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = self.incept1(x)
        x = F.relu(self.mp(self.conv2(x)))
        x = self.incept2(x)
        x = x.view(in_size, -1)  # flatten the
tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

HAMIM

Deeper ???

Well said Leo, well said

Can we just go deeper, keep stacking layers?



- *Plain* nets: stacking 3x3 conv layers…
- 56-layer net has **higher training error** and test error than 20-layer net

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.