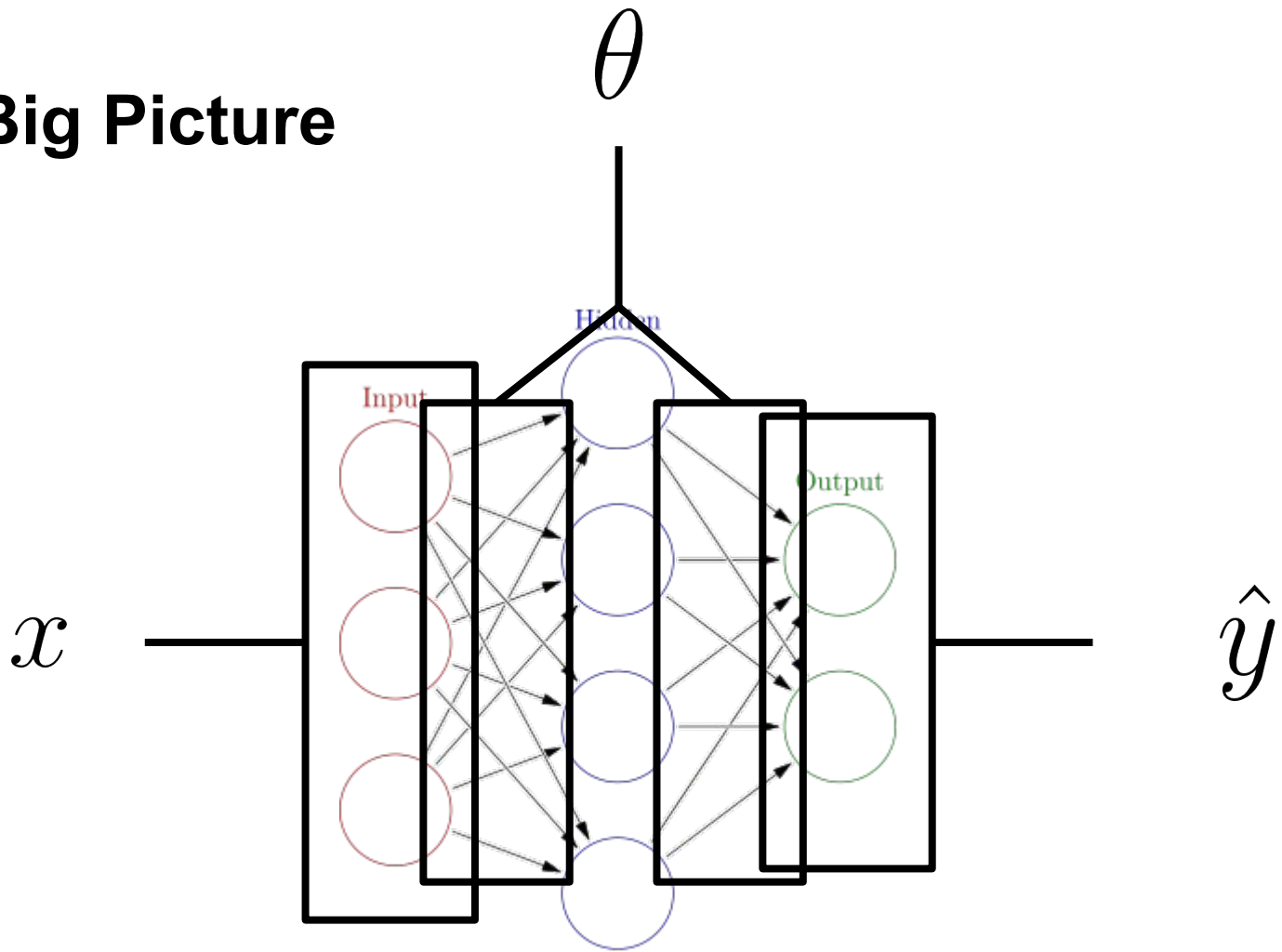
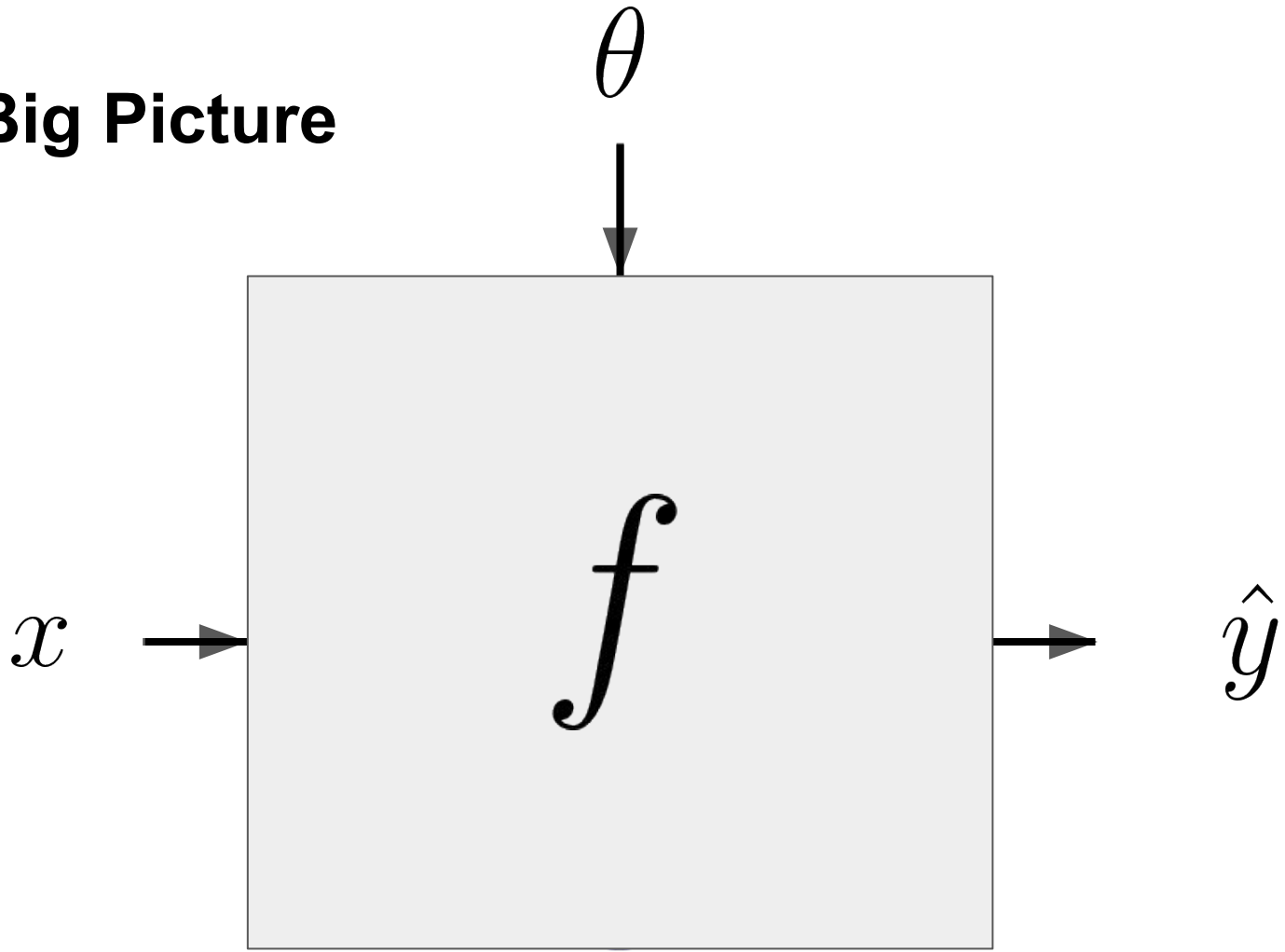
Backpropagation and Computational Graphs

Edward Banner and Brian Spiering

The Big Picture

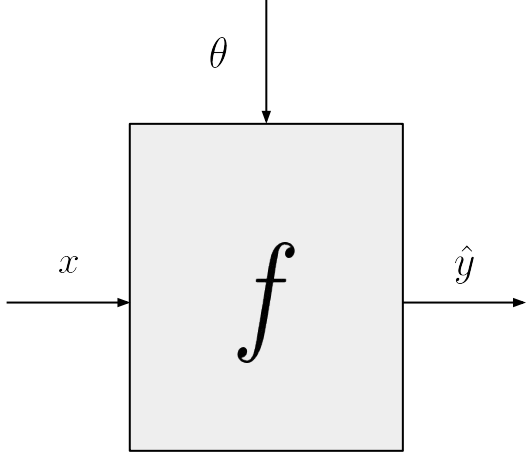


The Big Picture



The Big Picture

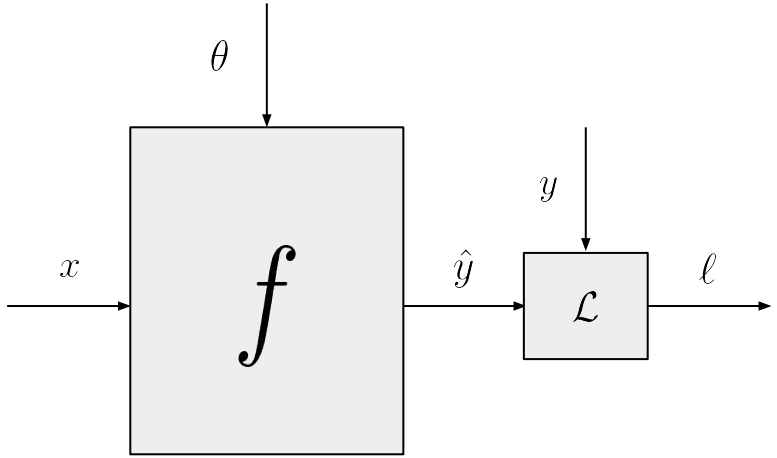
- Neural networks are **functions**
- e.g. 3 hidden-layer perceptron



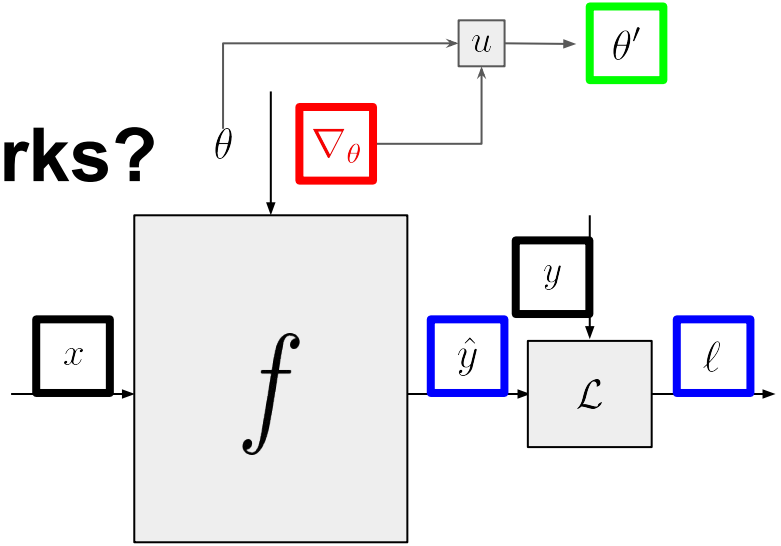
$$f(x, \theta) = \sigma(\sigma(\sigma(w_1 x + b_1)w_2 + b_2)w_3 + b_3)$$

- Hook to **loss function** and minimize loss
- e.g. Squared Loss

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$



How to Train Neural Networks?



Sample a data point and label

Forward prop to get loss

Backprop to calculate gradients

Update parameters using gradients

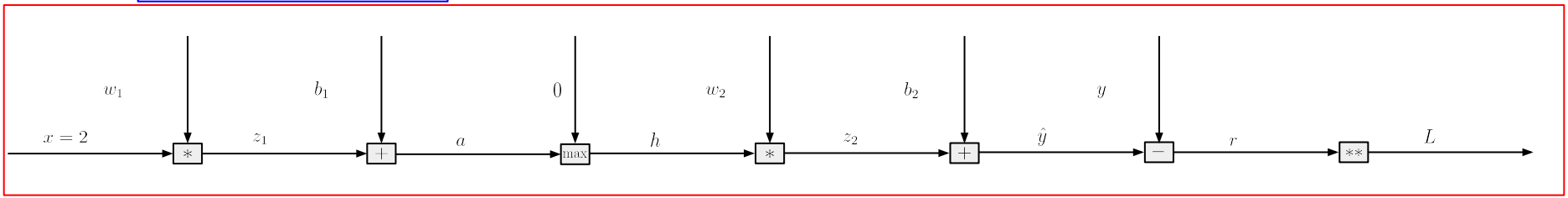
Goal For Today's Class

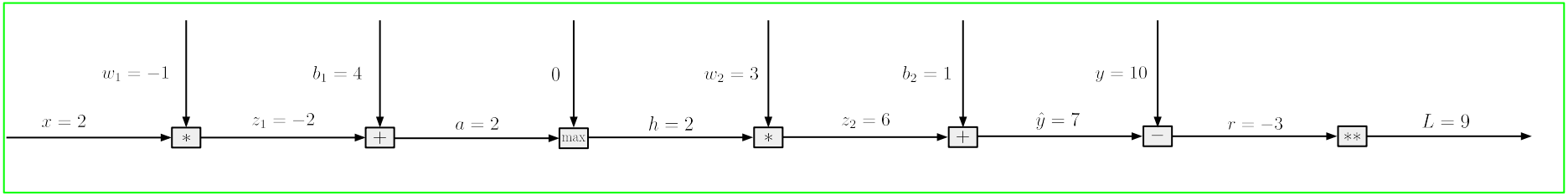
1. Computational Graphs
2. Forward Pass
3. Backward Pass

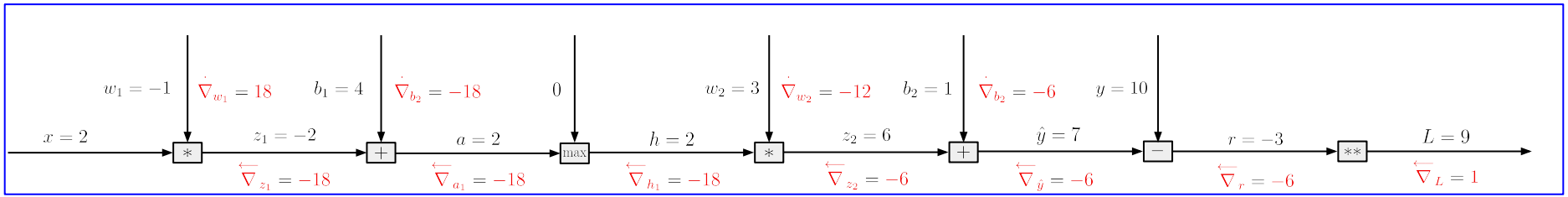
Given $L_{\text{MLP}}(x, y, w_1, b_1, w_2, b_2) = [\max((\max((w_1 x + b_1), 0) w_2 + b_2), 0) - y]^2$

and a particular point (e.g. $L(2, 10, -1, 4, 3, 1)$)

compute $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b_2}$







Why Understand Backpropagation?

- It’s a core algorithm of neural networks (and machine learning)
- Aids in debugging
- Makes (almost all of) the magic go away

Why Study 1D Examples?

- Numbers are simpler than vectors
- 1D backpropagation implementations can “easily” be extended to vectorized and mini-batch implementations

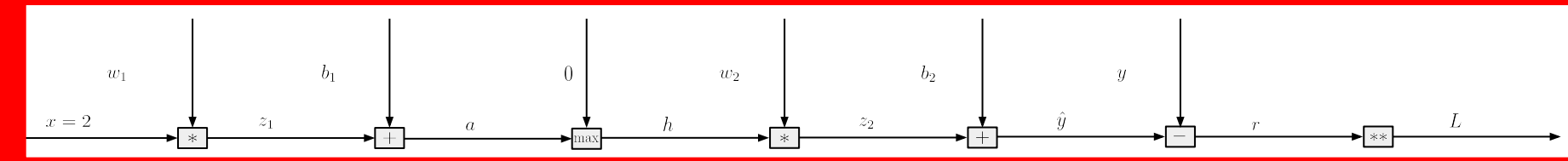
Goal For Today's Class

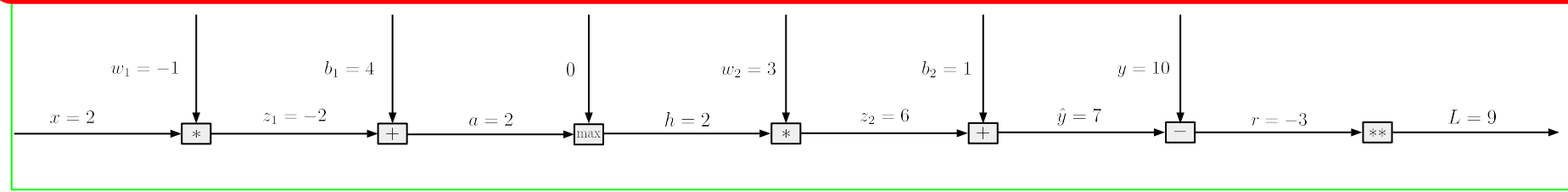
Given $L_{\text{MLP}}(x, y, w_1, b_1, w_2, b_2) = [\max((\max((w_1 x + b_1), 0) w_2 + b_2), 0) - y]^2$

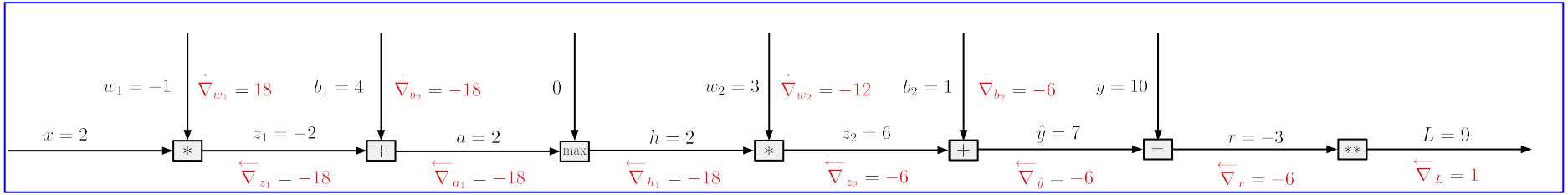
and a particular point (e.g. $L(2, 10, -1, 4, 3, 1)$)

compute $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b_2}$

1. Computational Graphs
2. Forward Pass
3. Backward Pass







What is a Computational Graph?

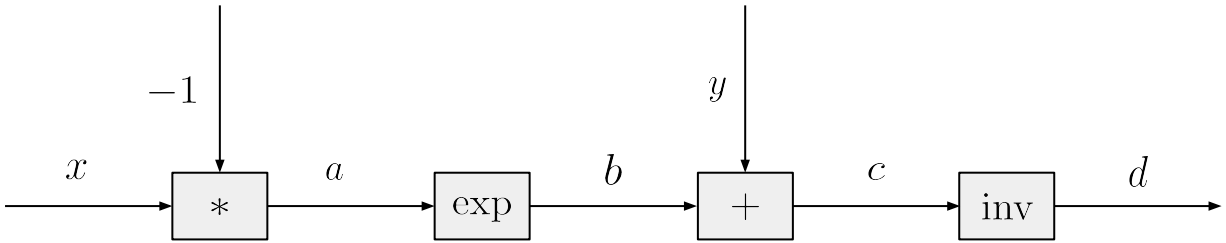
- A computational graph is a graphical representation of a function where the nodes are *operations* and the edges are *intermediate values*

How Do We Construct a Computational Graph?

- Think programmatically
  - How would you write a program to compute the function?

$$\text{Consider } f(x, y) = \frac{1}{y + e^{-x}}$$

```
def f(x, y):
    a = -x
    b = math.e ** a
    c = y + b
    d = 1 / c
    return d
```

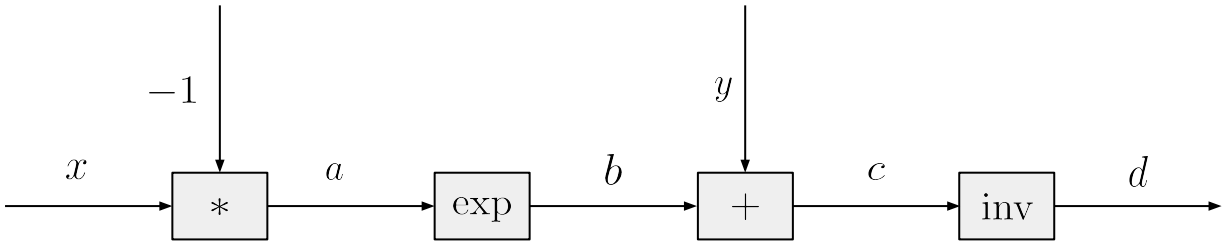


How Do We Construct a Computational Graph?

- How would you write a computer program to compute f?

$$\text{Consider } f(x, y) = \frac{1}{y + e^{-x}}$$

```
def f(x, y):
    a = -x
    b = math.e ** a
    c = y + b
    d = 1 / c
    return d
```

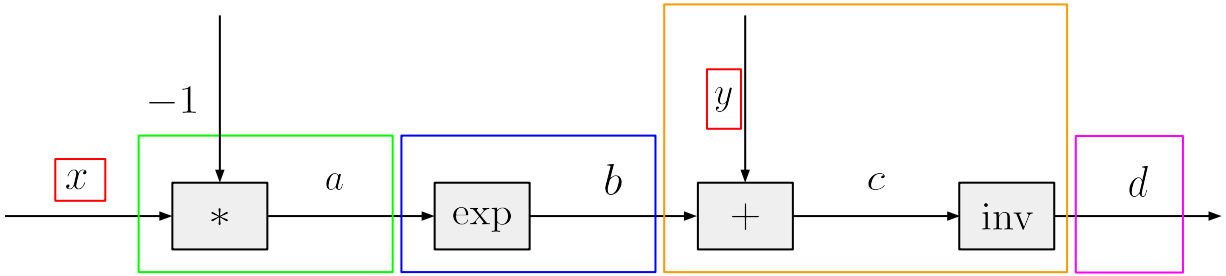


How Do We Construct a Computational Graph?

- How would you write a computer program to compute f?

$$\text{Consider } f(x, y) = \frac{1}{y + e^{-x}}$$

```
def f(x, y):
    a = -x
    b = math.e ** a
    c = y + b
    d = 1 / c
    return d
```

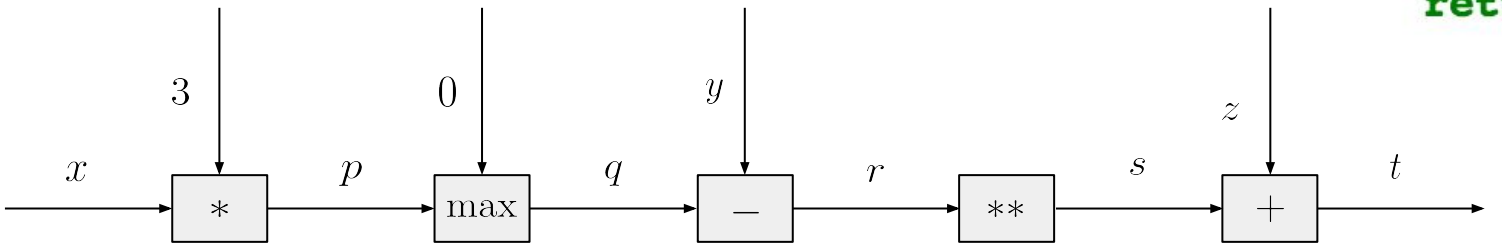


Exercise

$$f(x, y, z) = [\max(3x, 0) - y]^2 + z$$

- Write a function to compute f
- Draw a computational graph for f

```
def f(x, y, z):
    p = x * 3
    q = max(p, 0)
    r = q - y
    s = r**2
    t = s + z
    return t
```



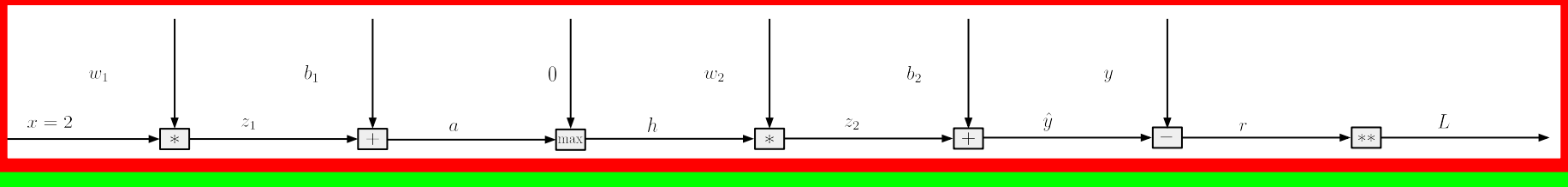
Goal For Today's Class

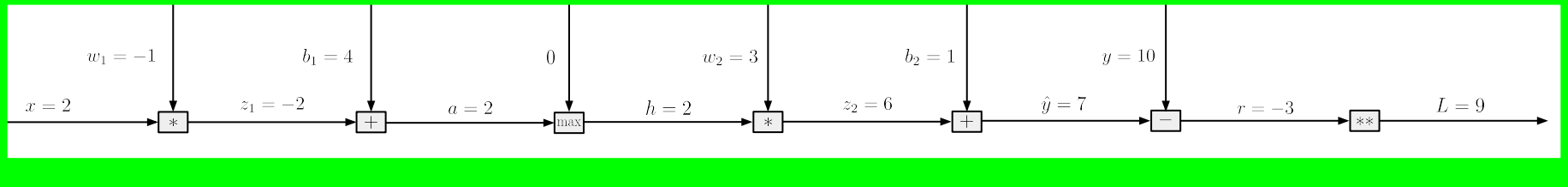
1. Computational Graphs
2. Forward Pass
3. Backward Pass

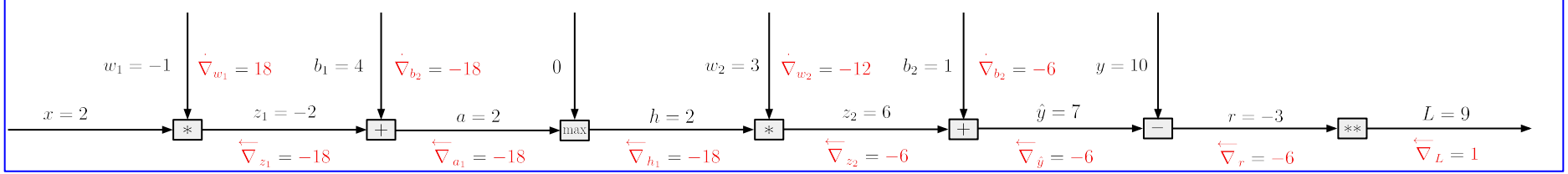
Given $L_{\text{MLP}}(x, y, w_1, b_1, w_2, b_2) = [\max((\max((w_1 x + b_1), 0) w_2 + b_2), 0) - y]^2$

and a particular point (e.g. $L(2, 10, -1, 4, 3, 1)$)

compute $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b_2}$







Forward Pass

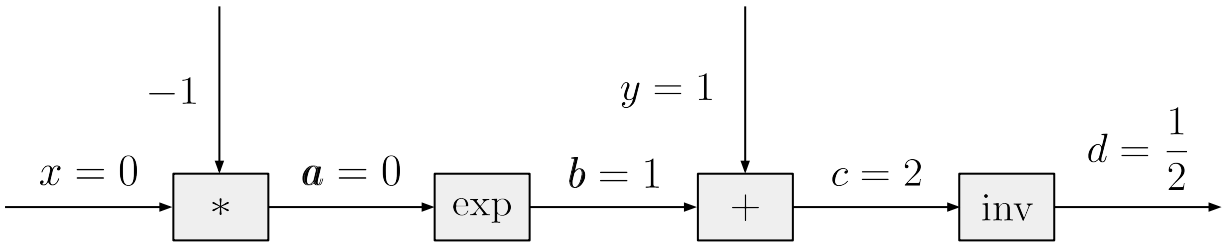
- Functions are to computational graphs as *function application* is to a computational graph forward pass

Forward Pass Example

- Think programmatically
  - How does the *environment* (e.g. the value of variables) change when the function is *invoked* with a set of *arguments*?

```
def f(x, y): # x=0, y=1
    a = -x # a = 0
    b = math.e ** a # b = 1
    c = y + b # c = 2
    d = 1 / c # d = 0.5
    return d
```

$$\text{Given } f(x,y) = \frac{1}{y + e^{-x}} \text{ consider } f(0,1)$$

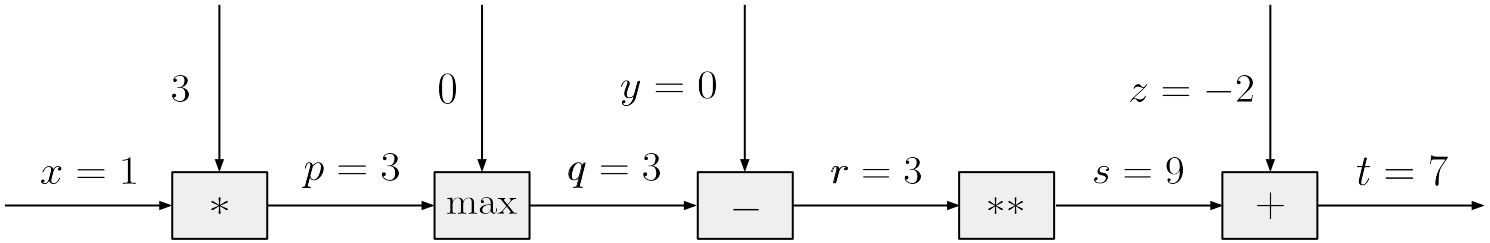


Exercise

$$f(x, y, z) = [\max(3x, 0) - y]^2 + z$$

Write out the forward pass for $f(1, 0, 3)$

```
def f(x, y, z): # x=1, y=0, z=-2
    p = x * 3 # p = 3
    q = max(p, 0) # q = 3
    r = q - y # r = 3
    s = r**2 # s = 9
    t = s + z # t = 7
    return t
```



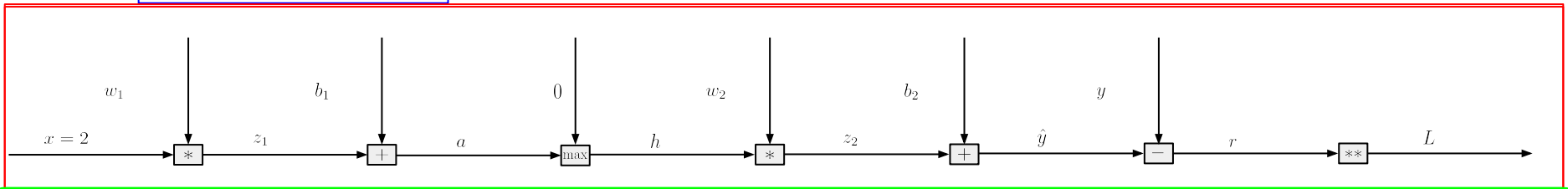
Goal For Today's Class

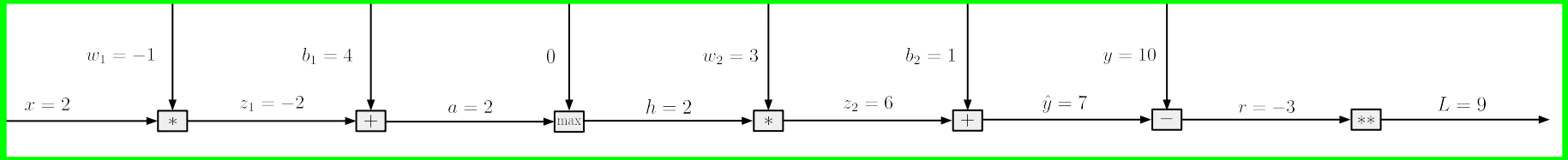
1. Computational Graphs
2. Forward Pass
3. Backward Pass

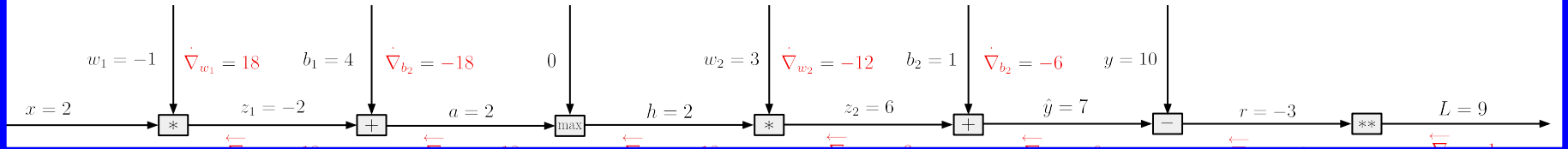
Given $L_{\text{MLP}}(x, y, w_1, b_1, w_2, b_2) = [\max((\max((w_1 x + b_1), 0) w_2 + b_2), 0) - y]^2$

and a particular point (e.g. $L(2, 10, -1, 4, 3, 1)$)

compute $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b_2}$

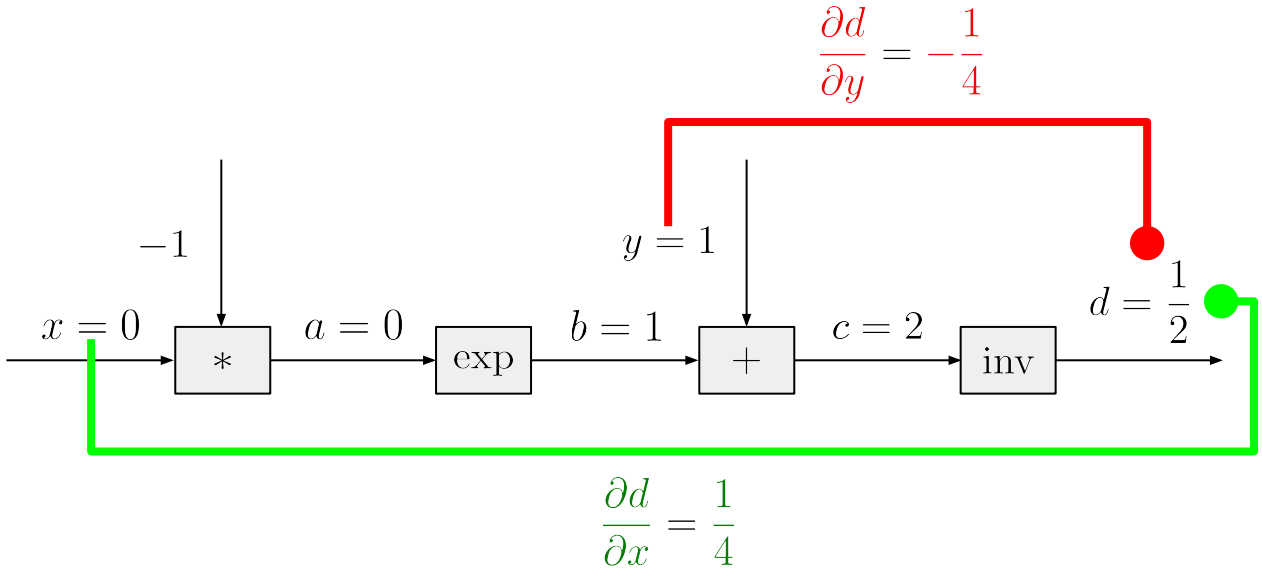






Computing Derivatives in a Computational Graph

Given $f(x, y) = \dfrac{1}{y + e^{-x}}$ consider $f(0, 1)$ and compute $\dfrac{\partial f}{\partial x}$ and $\dfrac{\partial f}{\partial y}$



Algorithm for Computing $\frac{\partial i}{\partial j}$

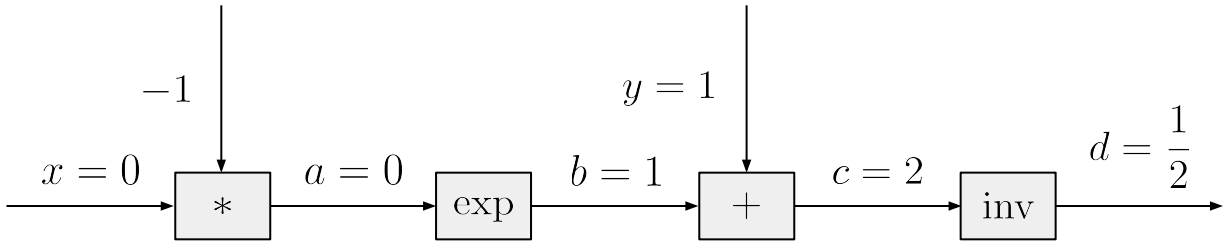
Pick two variables i and j.

Identify a path from i to j.

Given $f(x, y) = \frac{1}{y + e^{-x}}$ consider $f(0, 1)$ and compute $\frac{\partial f}{\partial y}$

Walk from i to j and compute the local derivative at every step of the way.

Take the product of these local derivatives.



Algorithm for Computing $\frac{\partial i}{\partial j}$

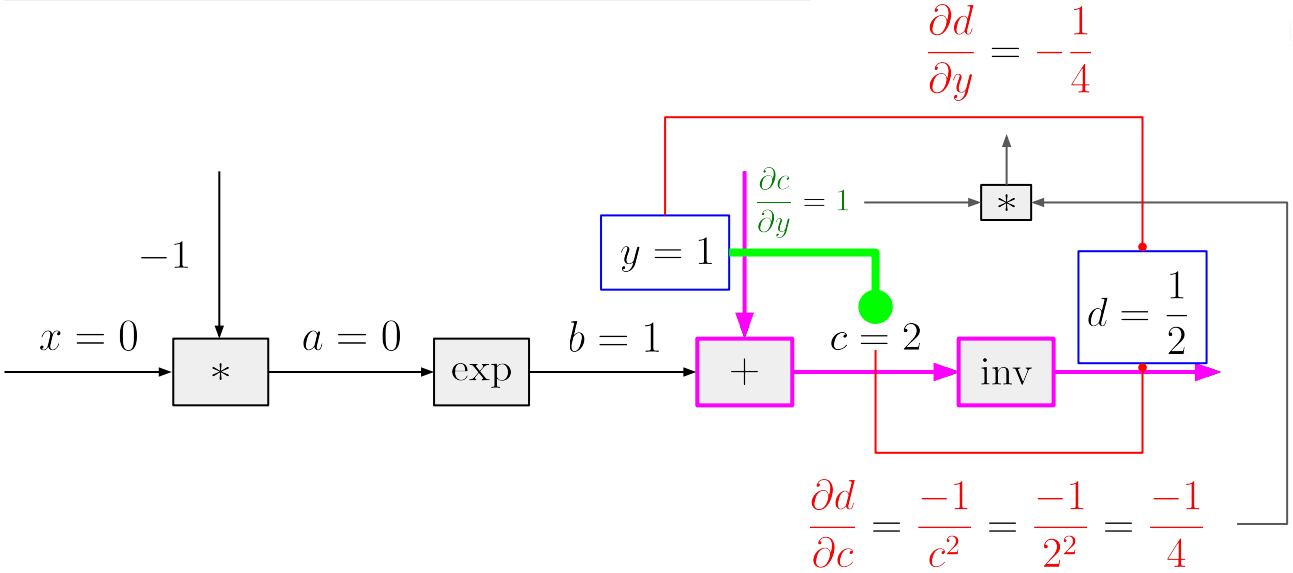
Pick two variables i and j.

Identify a path from i to j.

Walk from i to j and compute the local derivative at every step of the way.

Take the product of these local derivatives.

Given $f(x,y) = \frac{1}{y + e^{-x}}$ consider $f(0,1)$ and compute $\frac{\partial f}{\partial y}$




```
def dfdy(x, y): # x=0, y=1
    a = -x # a = 0
    b = math.e ** a # b = 1
    c = y + b # c = 2
    d = 1 / c # d = 0.5

    dcdy = 1
    dddc = -1 / c**2
    dddy = np.prod([dcdy, dddc])

    return dddy
```

Exercise

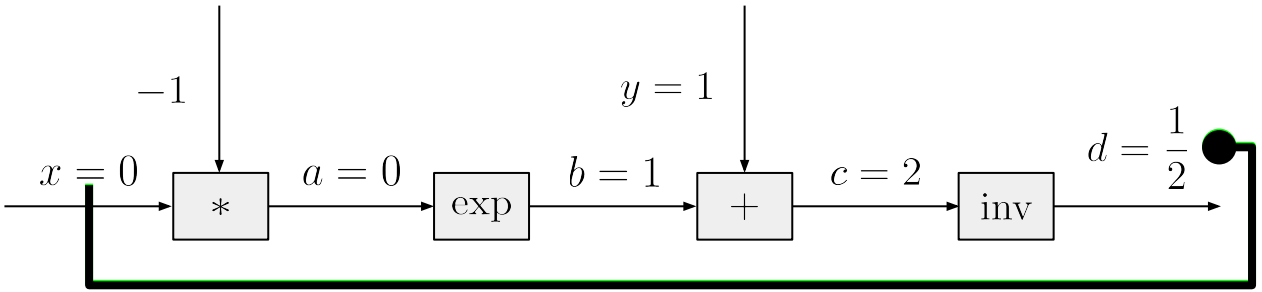
Pick two variables i and j.

Identify a path from i to j.

Given $f(x, y) = \dfrac{1}{y + e^{-x}}$ consider $f(0, 1)$ and compute $\dfrac{\partial f}{\partial x}$

Walk from i to j and compute the local derivative at every step of the way.

Take the product of these local derivatives.



$$\frac{\partial d}{\partial x} = \frac{1}{4}$$

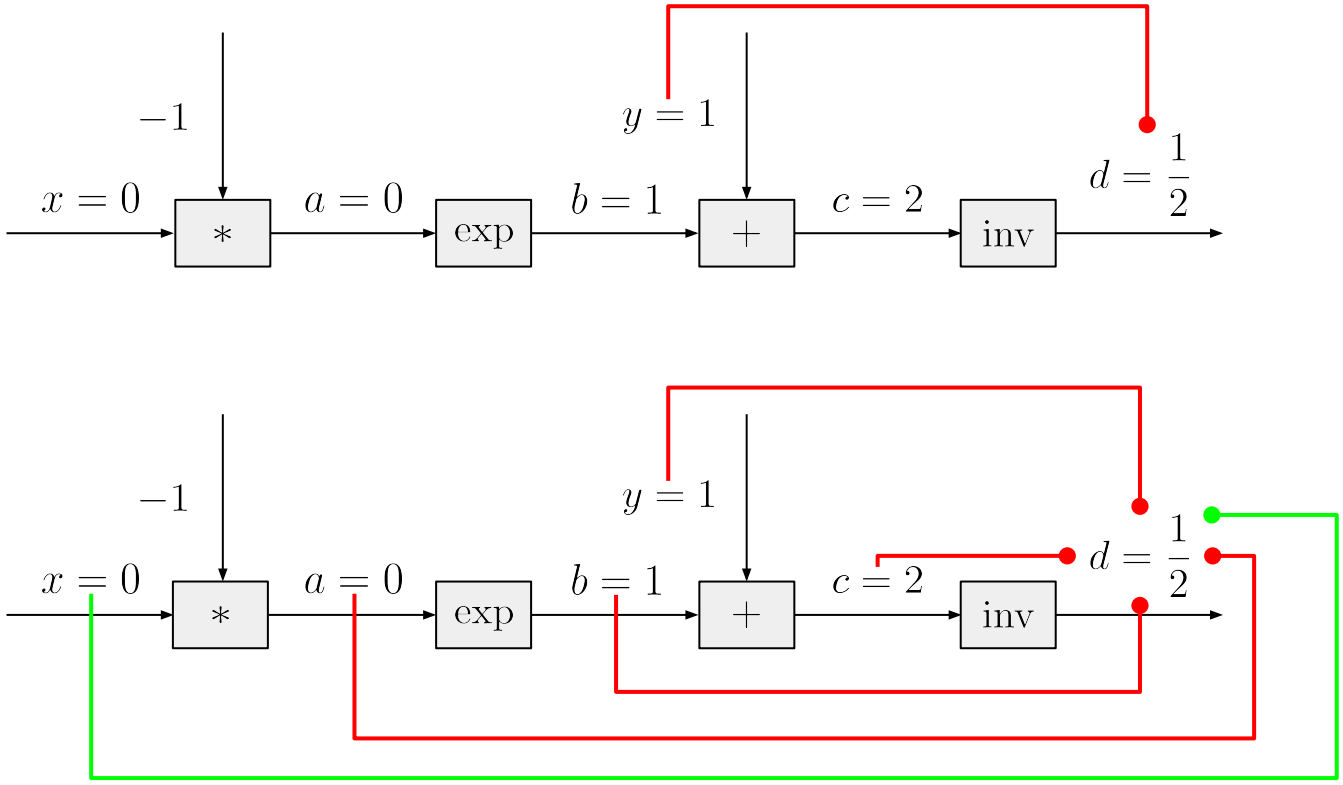
```python
def dfdx(x, y): # x=0, y=1
    a = -x # a = 0
    b = math.e ** a # b = 1
    c = y + b # c = 2
    d = 1 / c # d = 0.5

    dadx = -1
    dbda = math.e**a
    dcdb = 1
    dddc = -1/c**2
    dddx = np.prod([dadx, dbda, dcdb, dddc])

    return dddx
```

Runtime of This Algorithm

For a function $f(x_1, x_2, \ldots, x_n)$
and a computational graph with m nodes,
running this algorithm n times to compute each $\frac{\partial f}{\partial x_i}$
takes $O(nm)$ time and $O(m)$ space.

Backpropagation is an *optimization*
which takes $O(m)$ time and $O(1)$ space.

Algorithm for computing $\frac{\partial f}{\partial i}$ for all intermediate variables i



For g with multiple input variables, do this for each input variable.

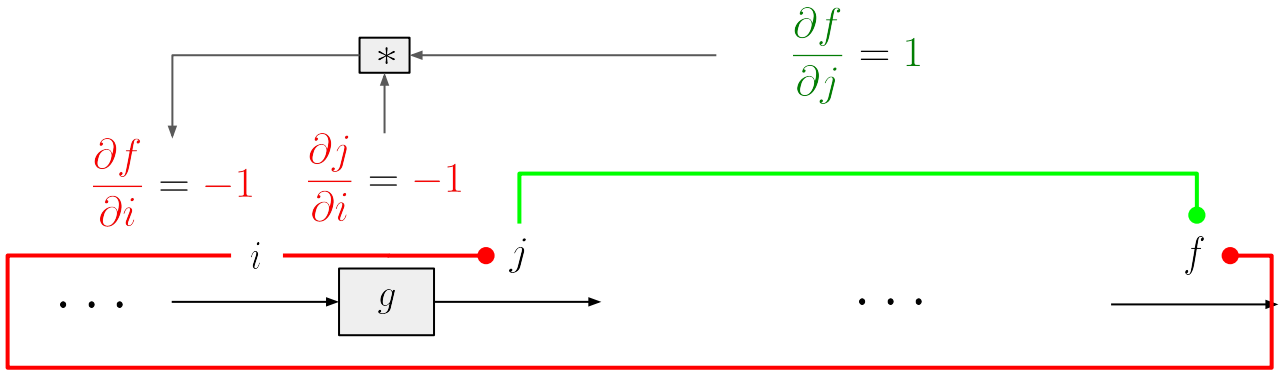
Backpropagation Algorithm

Start at the right and walk leftwards.

At every function g with input variable i and output variable j

$$\text{compute } \frac{\partial f}{\partial i} = \frac{\partial j}{\partial i} \cdot \frac{\partial f}{\partial j}.$$

$\frac{\partial j}{\partial i}$ is the "local" gradient while $\frac{\partial f}{\partial j}$ is the "global" gradient



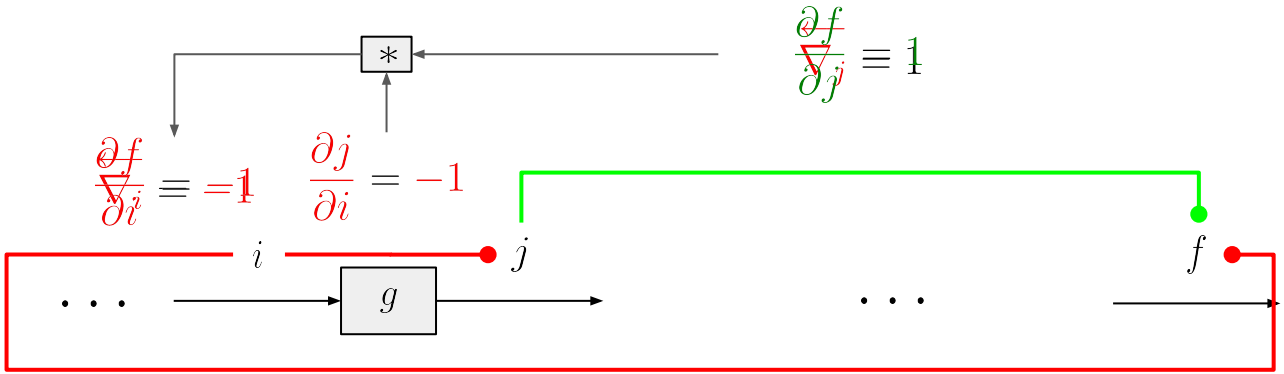
Some Gradient Notation

Define $\nabla_x = \dfrac{\partial f}{\partial x}$.

Start at the right and walk leftwards.

At every function g with input variable i and output variable j

compute $\nabla_i = \dfrac{\partial j}{\partial i} \cdot \nabla_j$

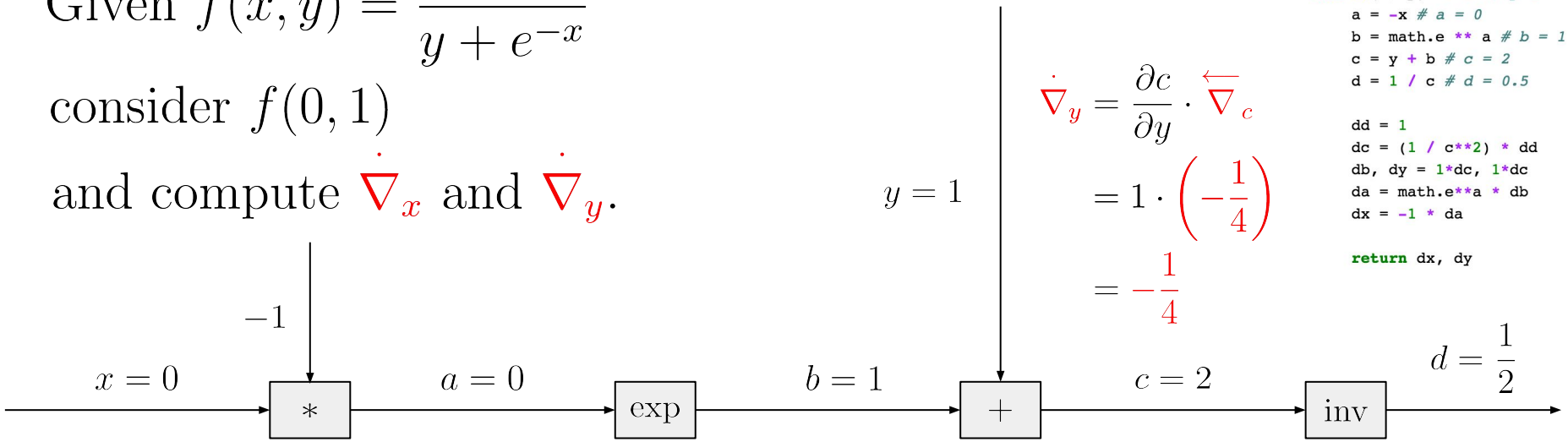


Given $f(x,y) = \dfrac{1}{y + e^{-x}}$

consider $f(0,1)$

and compute $\dot{\nabla}_x$ and $\dot{\nabla}_y$.

```
def df(x, y): # x=0, y=1
    a = -x # a = 0
    b = math.e ** a # b = 1
    c = y + b # c = 2
    d = 1 / c # d = 0.5

    dd = 1
    dc = (1 / c**2) * dd
    db, dy = 1*dc, 1*dc
    da = math.e**a * db
    dx = -1 * da

    return dx, dy
```



$$\dot{\nabla}_x = \frac{\partial a}{\partial x} \cdot \overleftarrow{\nabla}_a = (-1) \cdot \left(-\frac{1}{4}\right) = \frac{1}{4}$$

$$\overleftarrow{\nabla}_a = \frac{\partial b}{\partial a} \cdot \overleftarrow{\nabla}_b = e^a \cdot \left(-\frac{1}{4}\right) = e^0 \cdot \left(-\frac{1}{4}\right) = -\frac{1}{4}$$

$$\overleftarrow{\nabla}_b = \frac{\partial c}{\partial b} \cdot \overleftarrow{\nabla}_c = 1 \cdot \left(-\frac{1}{4}\right) = -\frac{1}{4}$$

$$\overleftarrow{\nabla}_c = \frac{\partial d}{\partial c} \cdot \overleftarrow{\nabla}_d = \left(-\frac{1}{c^2}\right) \cdot 1 = \left(-\frac{1}{4}\right) \cdot 1 = -\frac{1}{4}$$

$$\overleftarrow{\nabla}_d = \frac{\partial d}{\partial d} = 1$$

$$\dot{\nabla}_y = \frac{\partial c}{\partial y} \cdot \overleftarrow{\nabla}_c = 1 \cdot \left(-\frac{1}{4}\right) = -\frac{1}{4}$$

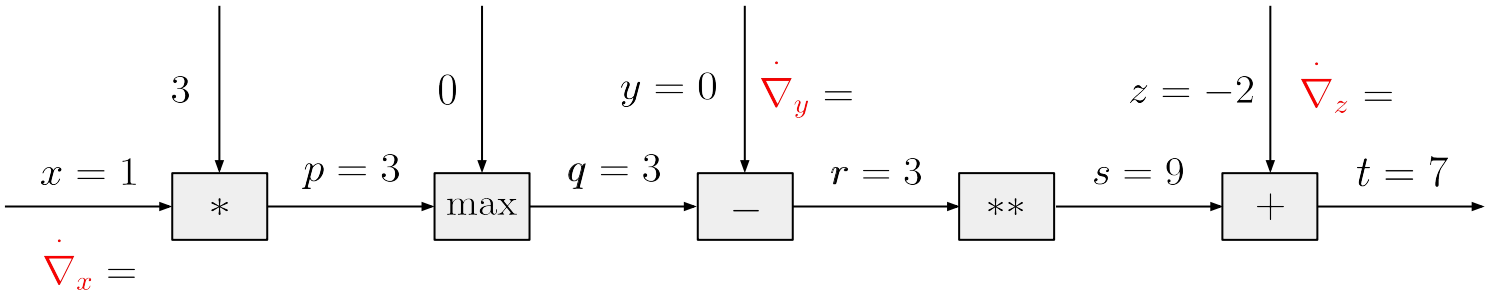
Exercise

Given $f(x, y, z) = [\max(3x, 0) - y]^2 + z$

consider $f(1, 0, -2)$

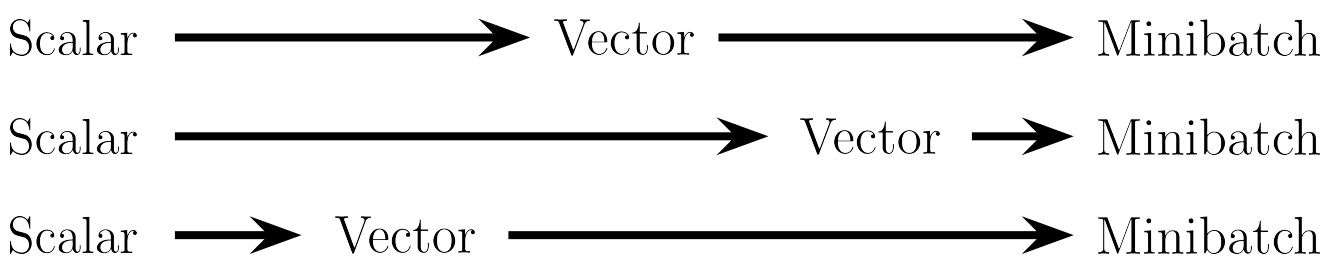
and compute $\dot{\nabla}_x$, $\dot{\nabla}_y$, and $\dot{\nabla}_z$

```
def df(x, y, z): # x=1, y=0, z=-2
    p = x * 3 # p = 3
    q = max(p, 0) # q = 3
    r = q - y # r = 3
    s = r**2 # s = 9
    t = s + z # t = 7

    dt = 1
    ds, dz = 1*dt, 1*dt
    dr = 2*r * ds
    dq, dy = 1*dr, -1*dr
    dp = (p>0) * dq
    dx = 3 * dp

    return dx, dy, dz
```

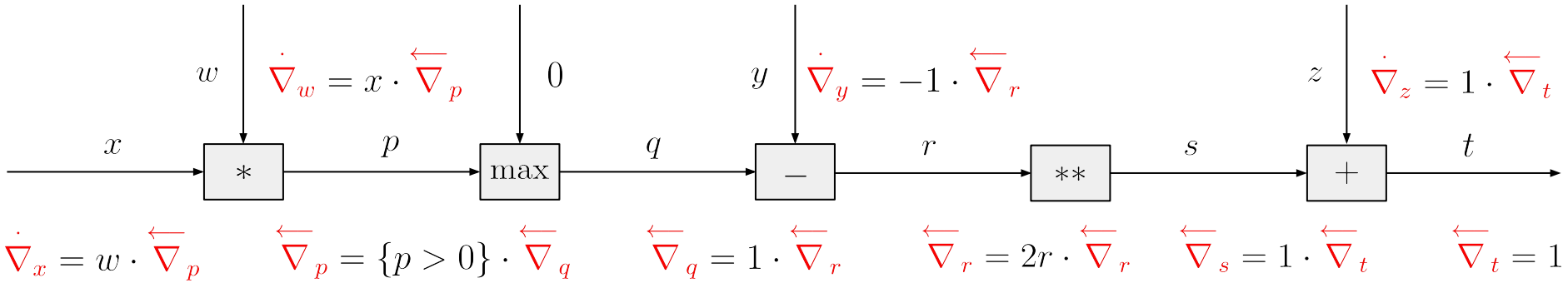


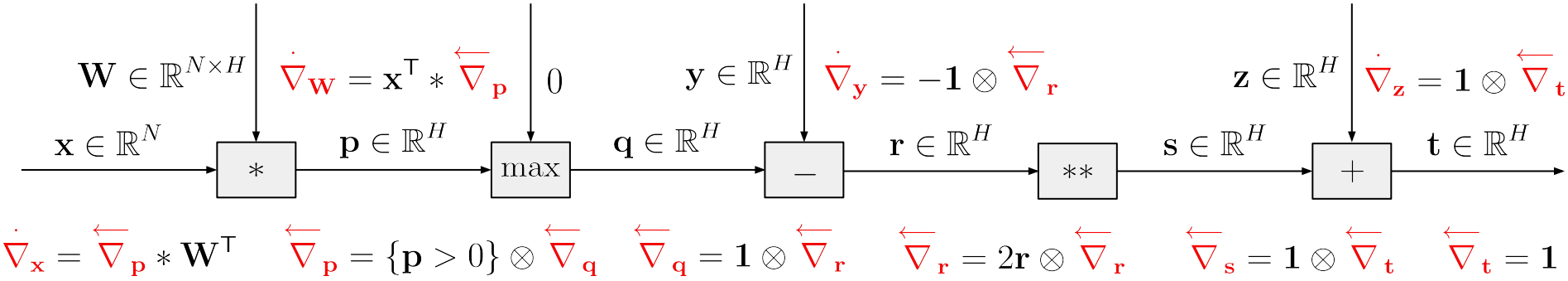
Vectorized Backprop

- How much work is it to extend a 1D backprop implementation to a vectorized one?
- How much work is it to extend a vectorized backprop to support minibatch learning?



Vectorized Backprop





Vectorized Backprop

$$\dot{\nabla}_w = x \cdot \overleftarrow{\nabla}_p \qquad \dot{\nabla}_y = -1 \cdot \overleftarrow{\nabla}_r \qquad \dot{\nabla}_z = 1 \cdot \overleftarrow{\nabla}_t$$

$$\dot{\nabla}_{\mathbf{W}} = \mathbf{x}^\mathsf{T} * \overleftarrow{\nabla}_{\mathbf{p}} \qquad \dot{\nabla}_{\mathbf{y}} = -\mathbf{1} \otimes \overleftarrow{\nabla}_{\mathbf{r}} \qquad \dot{\nabla}_{\mathbf{z}} = \mathbf{1} \otimes \overleftarrow{\nabla}_{\mathbf{t}}$$

$$\dot{\nabla}_x = w \cdot \overleftarrow{\nabla}_p \qquad \overleftarrow{\nabla}_p = \{p > 0\} \cdot \overleftarrow{\nabla}_q \qquad \overleftarrow{\nabla}_q = 1 \cdot \overleftarrow{\nabla}_r \qquad \overleftarrow{\nabla}_r = 2r \cdot \overleftarrow{\nabla}_r \qquad \overleftarrow{\nabla}_s = 1 \cdot \overleftarrow{\nabla}_t \qquad \overleftarrow{\nabla}_t = 1$$

$$\dot{\nabla}_{\mathbf{x}} = \overleftarrow{\nabla}_{\mathbf{p}} * \mathbf{W}^\mathsf{T} \qquad \overleftarrow{\nabla}_{\mathbf{p}} = \{\mathbf{p} > 0\} \otimes \overleftarrow{\nabla}_{\mathbf{q}} \qquad \overleftarrow{\nabla}_{\mathbf{q}} = \mathbf{1} \otimes \overleftarrow{\nabla}_{\mathbf{r}} \qquad \overleftarrow{\nabla}_{\mathbf{r}} = 2\mathbf{r} \otimes \overleftarrow{\nabla}_{\mathbf{r}} \qquad \overleftarrow{\nabla}_{\mathbf{s}} = \mathbf{1} \otimes \overleftarrow{\nabla}_{\mathbf{t}} \qquad \overleftarrow{\nabla}_{\mathbf{t}} = \mathbf{1}$$

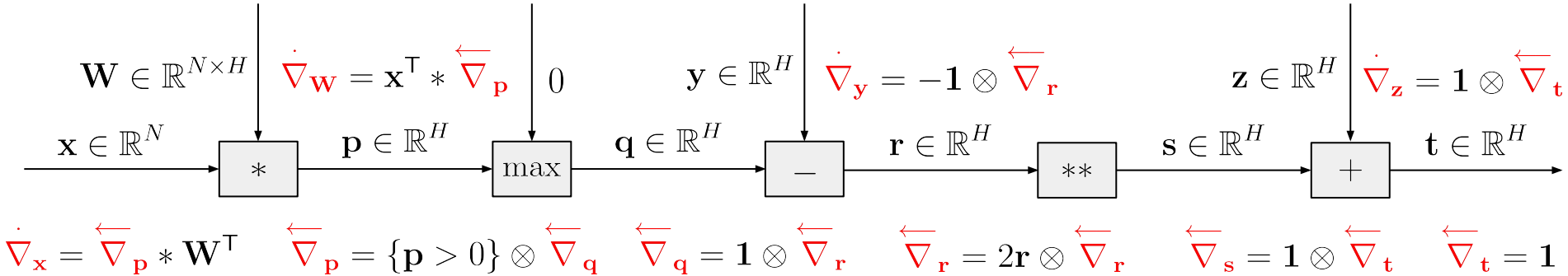
Scalar Input

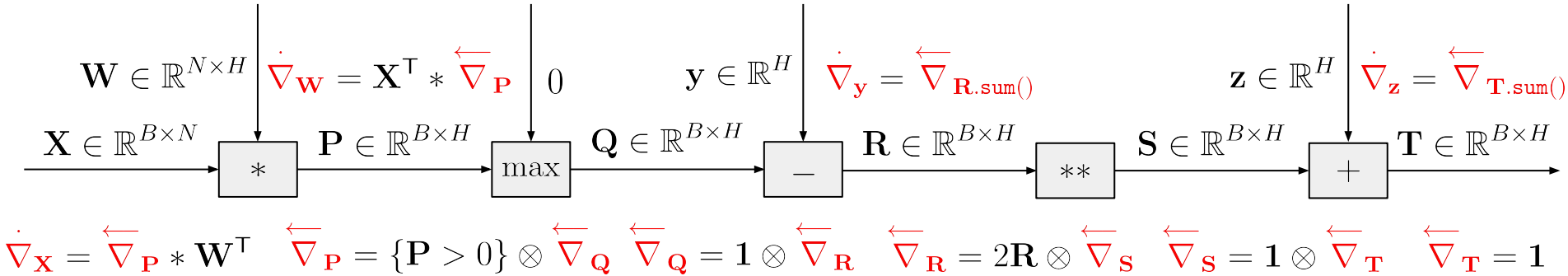
```
def df(x, w, y, z):
    p = x * w
    q = max(p, 0)
    r = q - y
    s = r**2
    t = s + z

    dt = 1
    ds, dz = 1*dt, 1*dt
    dr = 2*r * ds
    dq, dy = 1*dr, -1*dr
    dp = (p>0) * dq
    dx, dw = w*dp, x*dp

    return dx, dw, dy, dz
```

Vector Input

```
def df(x, W, y, z):
    p = x @ W
    q = np.max(p, 0)
    r = q - y
    s = r**2
    t = s + z

    dt = np.ones_like(t)
    ds, dz = 1*dt, 1*dt
    dr = 2*r * ds
    dq, dy = 1*dr, -1*dr
    dp = (p>0) * dq
    dx, dw = dp@W.T, x.T@dp

    return dx, dW, dy, dz
```

Vectorized + Minibatch Backprop





Vectorized + Minibatch Backprop

$$\dot{\nabla}_{\mathbf{W}} = \mathbf{x}^{\mathsf{T}} * \overleftarrow{\nabla}_{\mathbf{p}}$$
$$\dot{\nabla}_{\mathbf{W}} = \mathbf{X}^{\mathsf{T}} * \overleftarrow{\nabla}_{\mathbf{P}}$$

$$\dot{\nabla}_{\mathbf{y}} = -1 \otimes \overleftarrow{\nabla}_{\mathbf{r}}$$
$$\dot{\nabla}_{\mathbf{y}} = \overleftarrow{\nabla}_{\mathbf{R}.\mathrm{sum}()}$$

$$\dot{\nabla}_{\mathbf{z}} = 1 \otimes \overleftarrow{\nabla}_{\mathbf{t}}$$
$$\dot{\nabla}_{\mathbf{z}} = \overleftarrow{\nabla}_{\mathbf{T}.\mathrm{sum}()}$$

$$\dot{\nabla}_{\mathbf{x}} = \overleftarrow{\nabla}_{\mathbf{p}} * \mathbf{W}^{\mathsf{T}} \quad \overleftarrow{\nabla}_{\mathbf{p}} = \{\mathbf{p} > 0\} \otimes \overleftarrow{\nabla}_{\mathbf{q}} \quad \overleftarrow{\nabla}_{\mathbf{q}} = 1 \otimes \overleftarrow{\nabla}_{\mathbf{r}} \quad \overleftarrow{\nabla}_{\mathbf{r}} = 2\mathbf{r} \otimes \overleftarrow{\nabla}_{\mathbf{r}} \quad \overleftarrow{\nabla}_{\mathbf{s}} = 1 \otimes \overleftarrow{\nabla}_{\mathbf{t}} \quad \overleftarrow{\nabla}_{\mathbf{t}} = 1$$

$$\dot{\nabla}_{\mathbf{X}} = \overleftarrow{\nabla}_{\mathbf{P}} * \mathbf{W}^{\mathsf{T}} \quad \overleftarrow{\nabla}_{\mathbf{P}} = \{\mathbf{P} > 0\} \otimes \overleftarrow{\nabla}_{\mathbf{Q}} \quad \overleftarrow{\nabla}_{\mathbf{Q}} = 1 \otimes \overleftarrow{\nabla}_{\mathbf{R}} \quad \overleftarrow{\nabla}_{\mathbf{R}} = 2\mathbf{R} \otimes \overleftarrow{\nabla}_{\mathbf{S}} \quad \overleftarrow{\nabla}_{\mathbf{S}} = 1 \otimes \overleftarrow{\nabla}_{\mathbf{T}} \quad \overleftarrow{\nabla}_{\mathbf{T}} = 1$$

Vector Input

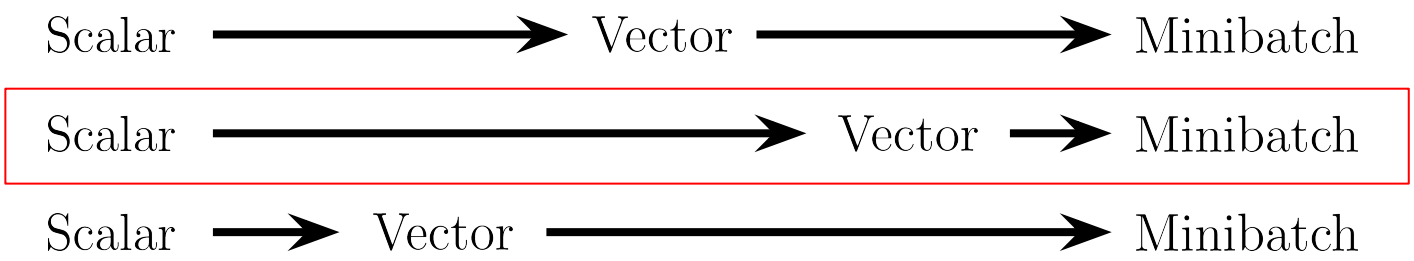
```
def df(x, W, y, z):
    p = x @ W
    q = np.max(p, 0)
    r = q - y
    s = r**2
    t = s + z

    dt = np.ones_like(t)
    ds, dz = 1*dt, 1*dt
    dr = 2*r * ds
    dq, dy = 1*dr, -1*dr
    dp = (p>0) * dq
    dx, dw = dp@W.T, x.T@dp

    return dx, dW, dy, dz
```

Vector + Minibatch Input

```
def df(X, W, y, z):
    P = X @ W
    Q = np.max(P, 0)
    R = Q - y
    S = R**2
    T = S + z

    dT = np.ones_like(T)
    dS, dz = 1*dT, dT.sum(axis=0)
    dR = 2*R * dS
    dQ, dy = 1*dr, -1*dR.sum(axis=0)
    dP = (P>0) * dQ
    dX, dW = dP@W.T, X.T@dP

    return dX, dW, dy, dz
```

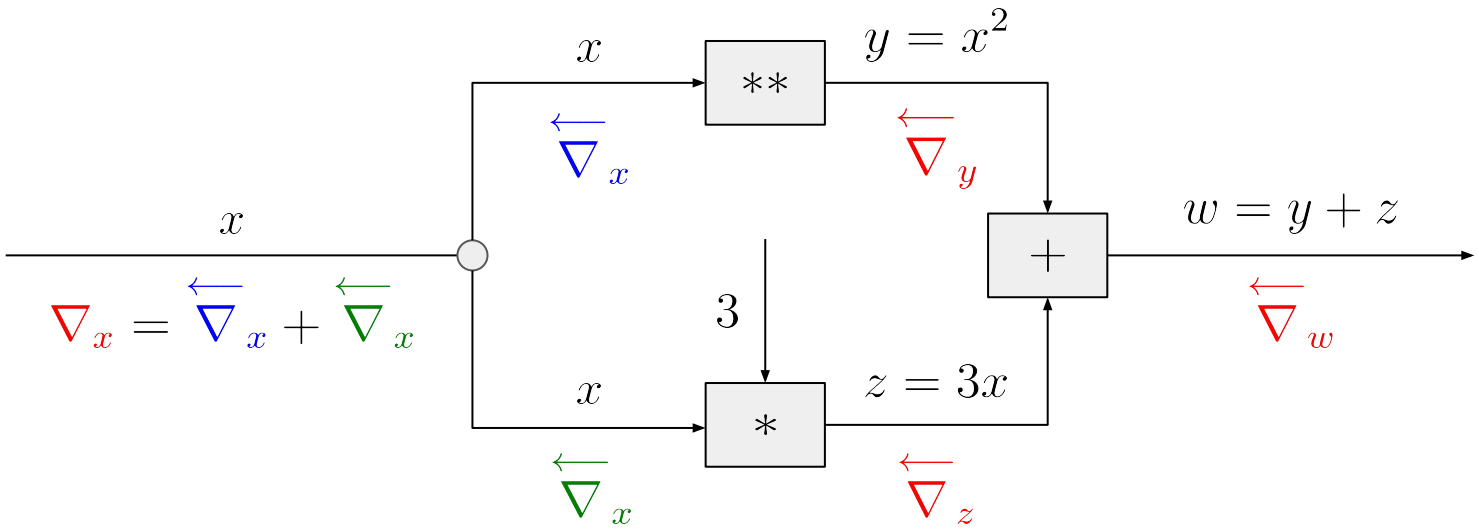
Vectorized Backprop

- How much work is it to extend a 1D backprop implementation to a vectorized one?
- How much work is it to extend a vectorized backprop to support minibatch learning?



Handling Multiple Gradients

Compute ∇_x for $w = f(x) = x^2 + 3x$



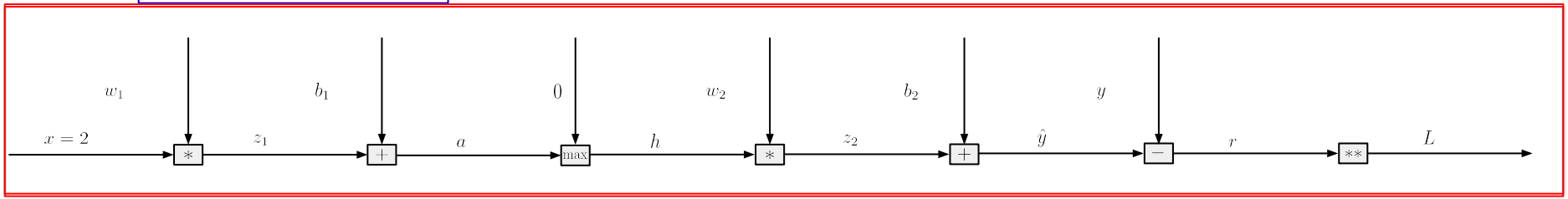
Goal For Today's Class

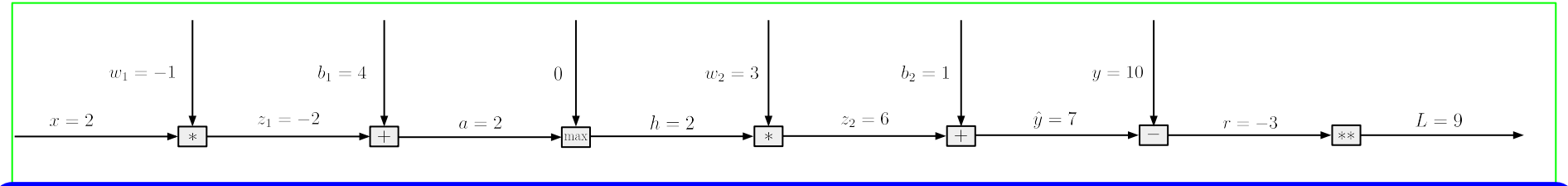
Given $L_{\text{MLP}}(x, y, w_1, b_1, w_2, b_2) = [\max((\max((w_1 x + b_1), 0) w_2 + b_2), 0) - y]^2$

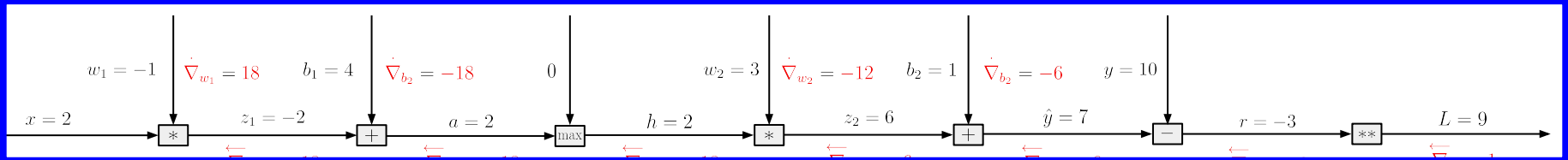
and a particular point (e.g. $L(2, 10, -1, 4, 3, 1)$)

compute $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b_2}$

1. Computational Graphs
2. Forward Pass
3. Backward Pass



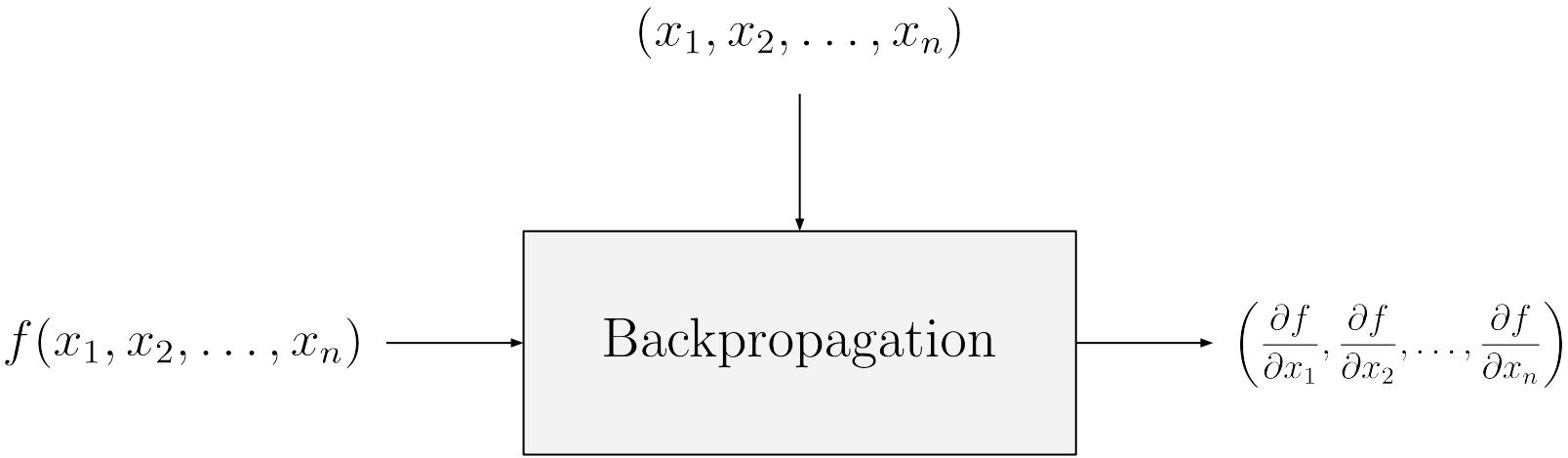




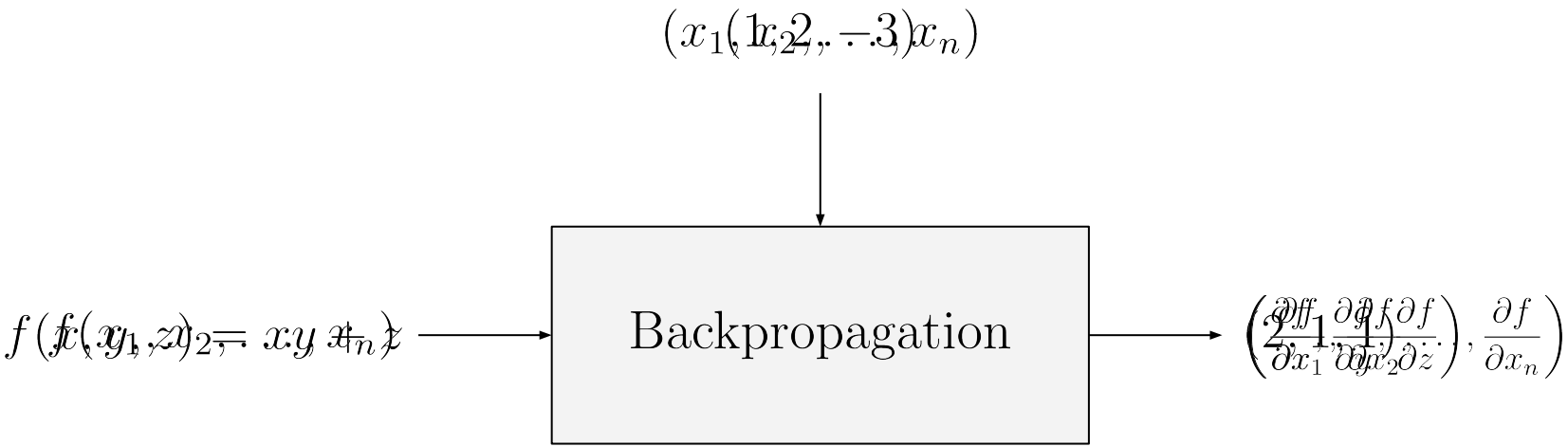
End of Lecture

What is Backpropagation?

- Backpropagation is **an algorithm for computing analytic (i.e. exact) gradients for functions**

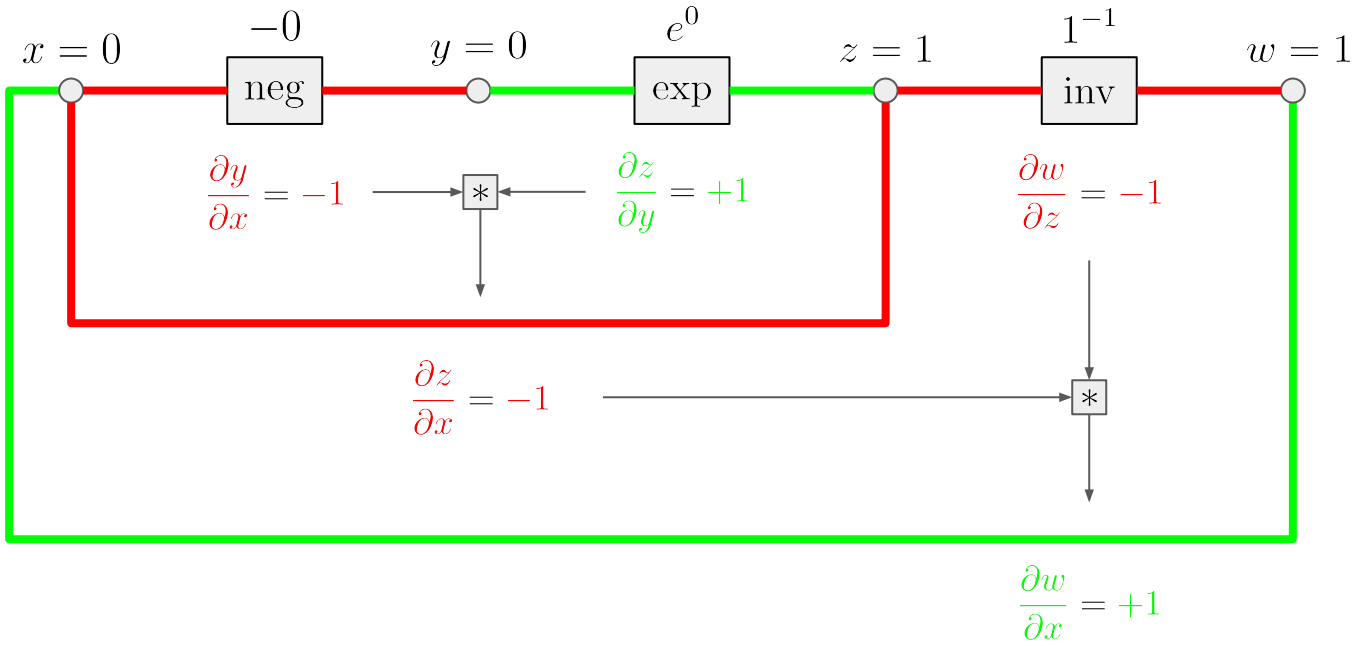


Example



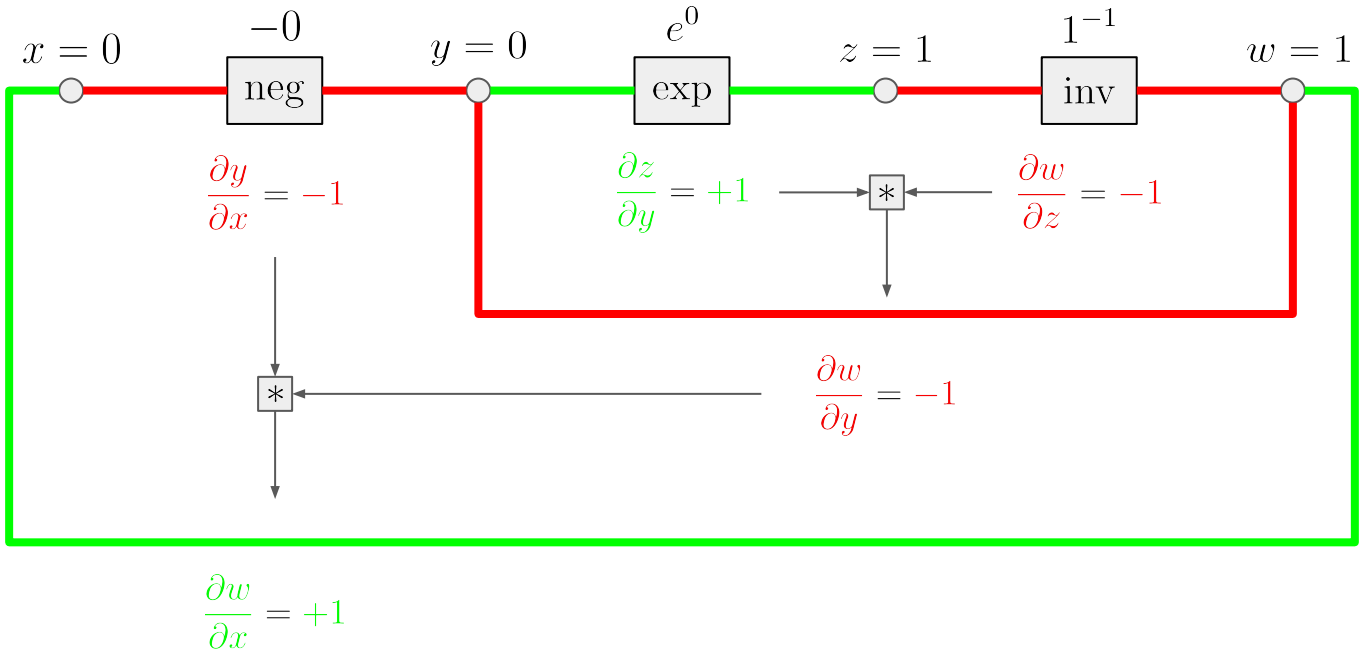
Forward Example

Given $w = f(x) = \dfrac{1}{e^{-x}}$ compute $\dfrac{\partial w}{\partial x}(0)$



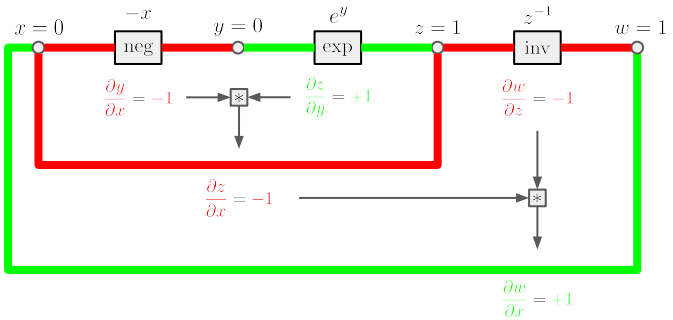
Backward Example

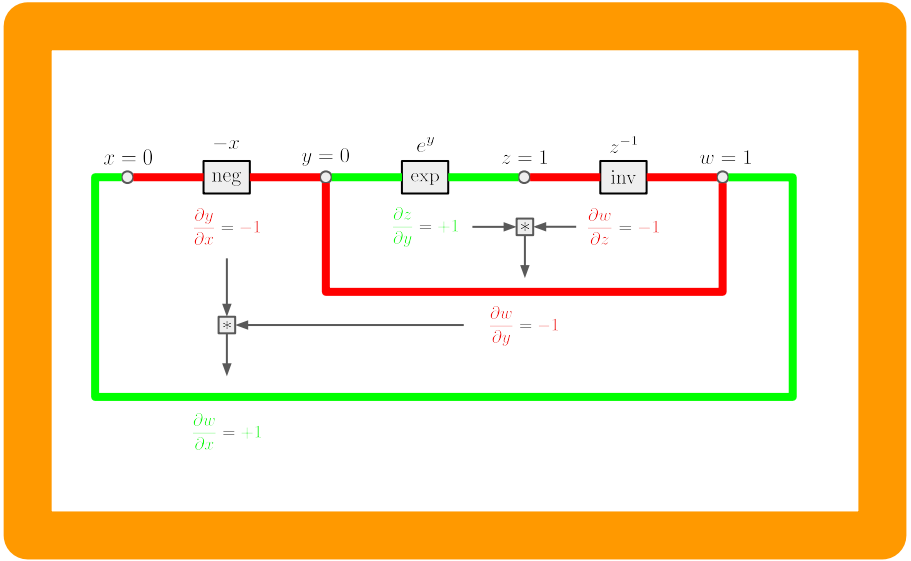
Given $w = f(x) = \dfrac{1}{e^{-x}}$ compute $\dfrac{\partial w}{\partial x}(0)$



Which is Better?

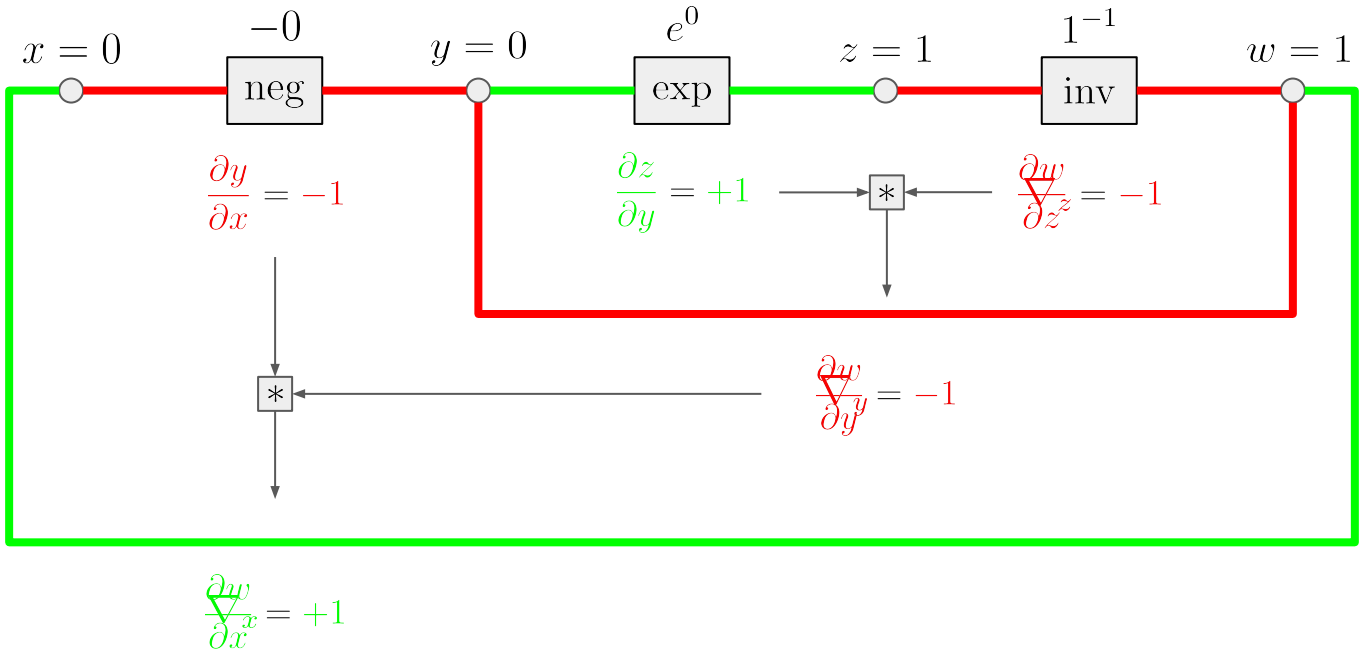
Given $w = f(x) = \dfrac{1}{e^{-x}}$ compute $\dfrac{\partial w}{\partial x}(0)$





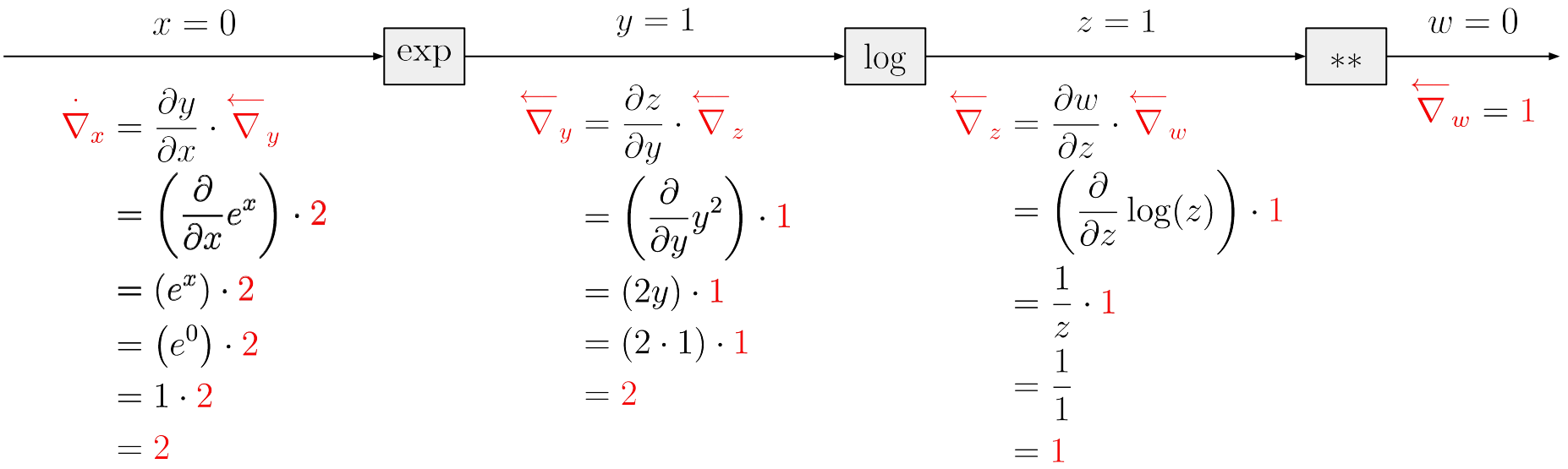
Backward Example

Given $w = f(x) = \dfrac{1}{e^{-x}}$ compute $\dfrac{\partial w}{\partial x}(0)$



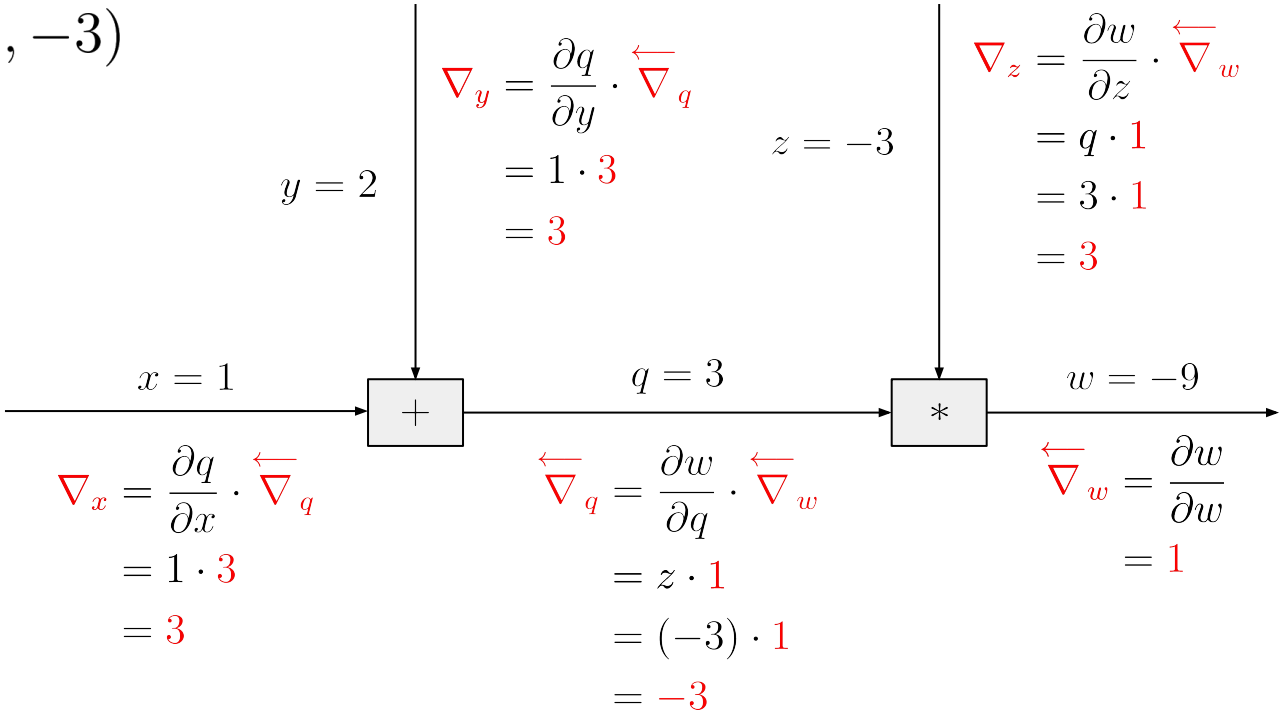
Example with Del Notation

Compute $\nabla_{x=0}$ for $w = f(x) = \log\left([e^x]^2\right)$



Functions of > 1 Variable

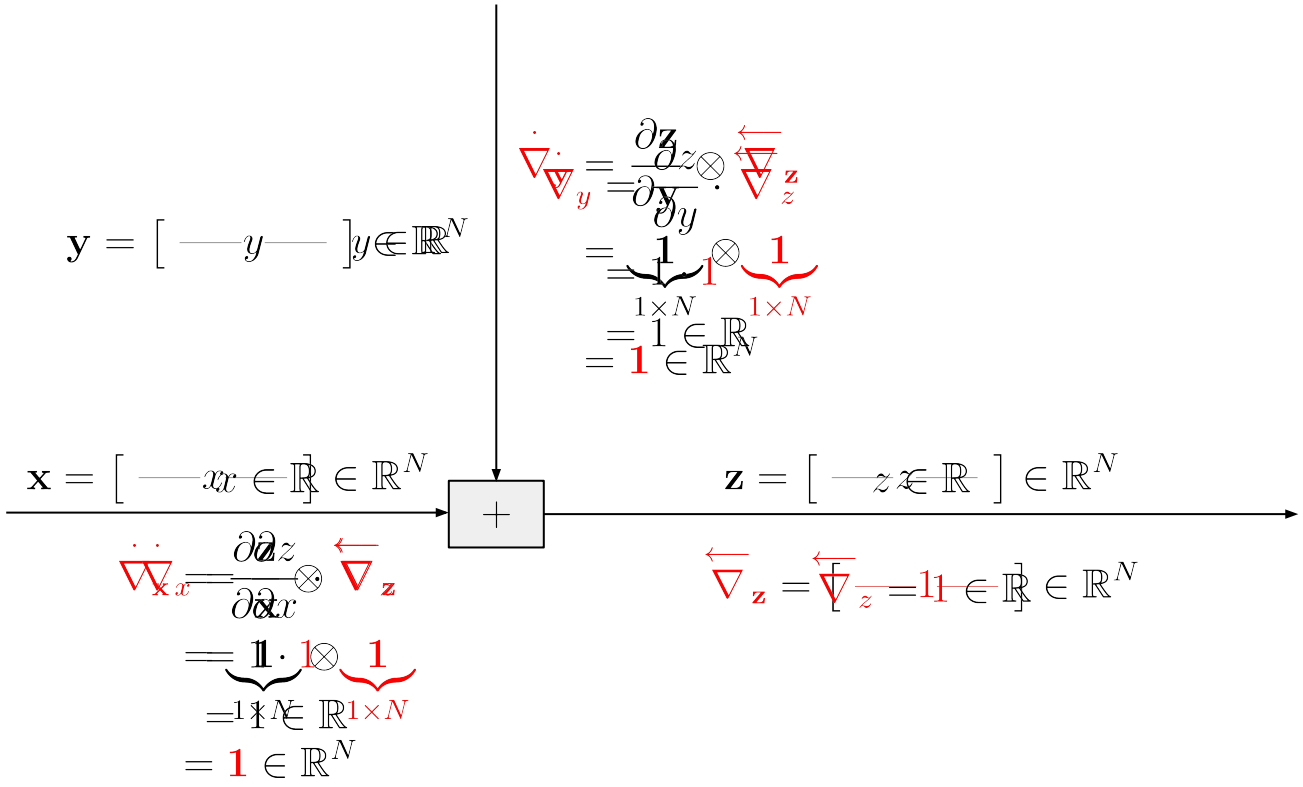
Given $w = f(x, y, z) = (x + y)z$ compute $\nabla_x, \nabla_y,$ and ∇_z for $f(1, 2, -3)$



Vectorized Sum

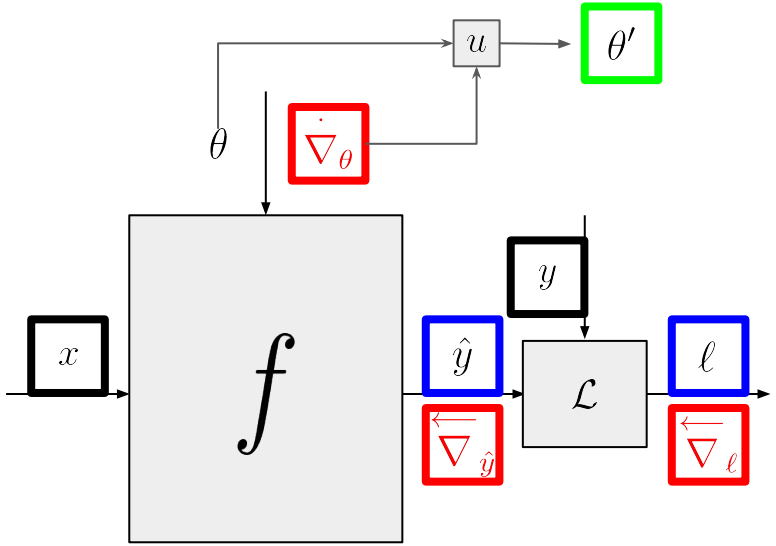
Given $z = f(x, y) = x + y$ compute ∇_x and ∇_y

Given $\mathbf{z} = f(\mathbf{x}, \mathbf{y}) = \mathbf{x} + \mathbf{y}$ compute $\nabla_{\mathbf{x}}$ and $\nabla_{\mathbf{y}}$



Vectorized max

Back to Neural Networks



Sample a data point and label

Forward prop to get loss

Backprop to calculate gradients

Update parameters using gradients