CS 188-2
Discussion-Week 7

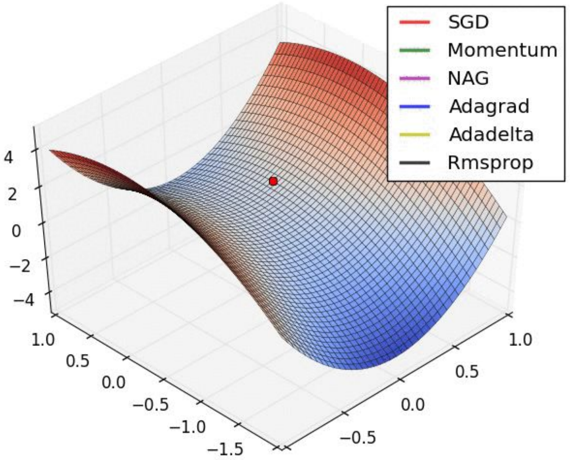
Ali Hatamizadeh
11/08/2019

Gradient Descent

Gradient Descent

- Basic idea: minimize an objective function parameterized by model's parameters by updating the parameters in the opposite direction of the gradient of the objective function w.r.t. to the parameters

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$



Batch Gradient Descent

- Batch gradient descent: compute the gradient of the cost function w.r.t. to the parameters for the entire training dataset:
    - Need to calculate the gradients for the whole dataset to perform just **one** update.
    - Batch gradient descent may not be feasible to use in many cases. **Why** ?

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

- A Pythonic overview

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Batch Gradient Descent

- Batch gradient descent: compute the gradient of the cost function w.r.t. to the parameters for the entire training dataset:
  - Need to calculate the gradients for the whole dataset to perform just **one** update.
  - Batch gradient descent may not be feasible to use in many cases. **Why** ?

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

- A Pythonic overview

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Batch Gradient Descent

- Batch gradient descent: compute the gradient of the cost function w.r.t. to the parameters for the entire training dataset:
  - Need to calculate the gradients for the whole dataset to perform just **one** update.
  - Batch gradient descent may not be feasible to use in many cases. **Why** ?

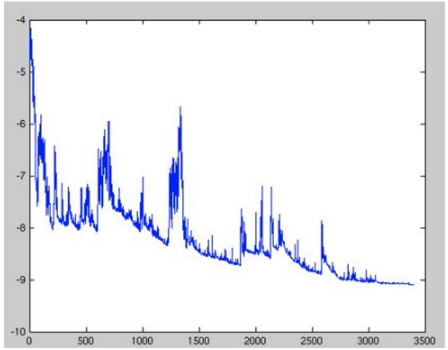
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

- Batch gradient descent is **guaranteed** to converge to the **global minimum** for **convex error surfaces** and to a **local minimum** for **non-convex surfaces**.

Stochastic Gradient Descent

- Stochastic Gradient Descent: performs a parameter update for each training example:
  - Performs redundant computations in large datasets
    - We draw random examples with replacement thus results in independent sampling
    - The loss is approximated from one training example leading to noisy gradient
    - Noisy gradient can be useful for non-convex loss functions
- (it needs to recompute the gradient for very similar examples before making an update)

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$



Stochastic Gradient Descent

- SGD would probably show similar convergence behavior as batch gradient descent if we properly decrease the learning rate

- Pythonic overview

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Mini-Batch Gradient Descent

- Mini-batch gradient descent performs an update for every mini-batch of n training examples (a.k.a batch size):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

- It generally results in more stability in convergence
- It may not guarantee a proper convergence
- Initial learning rate and its schedule needs to be properly chosen
- Minimizing highly non-convex error functions can be challenging due to being trapped in their many suboptimal local minima
- For sparse data-sets, with data samples having different frequencies, having the same learning rate applied to all of them is not optimal.

Mini-Batch Gradient Descent

- Mini-batch gradient descent performs an update for every mini-batch of n training examples (a.k.a batch size):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

- Pythonic overview

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Momentum

- Around local optima, surface curves may be much more steep in one direction than the other.
- SGD has trouble navigating around these areas
    - SGD occilitates across the slopes of ravine making hesitant progress along the bottom towards the local optimum

Image 2: SGD without momentum

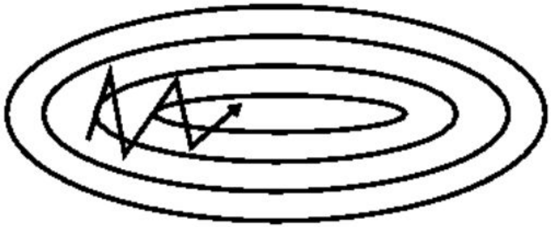


Image 3: SGD with momentum

Momentum

- Momentum accelerates SGD in the relevant direction and dampens oscillations

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Image 2: SGD without momentum

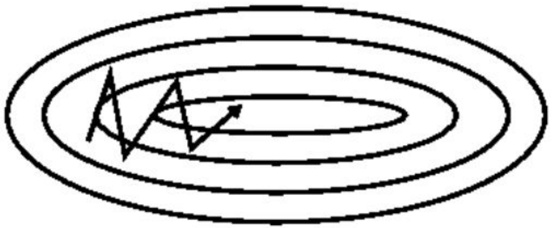


Image 3: SGD with momentum

Adagrad

- It adapts the learning rate to the parameters, performing smaller updates(i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features (very suitable for sparse data)

- One issue with Adagrad is the fact that learning rate will become eventually very small as the accumulated sum of gradient keep growing

Adagrad

- Partial derivative of the objective function w.r.t. to the parameter

$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

- Parameter update

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

- Modify the learning rate for each parameter based on the past gradient computed for that parameter

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

Adagrad

- Partial derivative of the objective function w.r.t. to the parameter

$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

- Parameter update

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

- Vectorized format

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

Adadelta

- An extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate
- Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size
- The sum of gradients is recursively defined as a <u>decaying average</u> of all past squared gradients
- The running average at time step t only depends on previous average and current gradient

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Adadelta

- Putting things together:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

Adadelta

- In order to deal with the issue of unit inconsistency, we define another exponentially decaying average, this time not of squared gradients but of squared parameter updates :

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}.$$

Adadelta

- Eventually :

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

RMSprop

- Similar to Adadelta, in an effort to resolve, Adagrad's radically diminishing learning rates :

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adam

- Adaptive Moment Estimation (Adam)
  - computes adaptive learning rates for each parameter
  - Stores an exponentially decaying average of past squared gradients (like Adadelta and RMSprop)
  - Adam also keeps an exponentially decaying average of past gradients (like momentum)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Back Propagation

Slide credit: some slides adopted from Stanford CS 231n

Back Propagation

- Different way for computing gradient
  - Numerically
    - Slow, and approximate but easy to calculate !
  - Analytically
    - Fast and exact but easy to make mistakes !

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

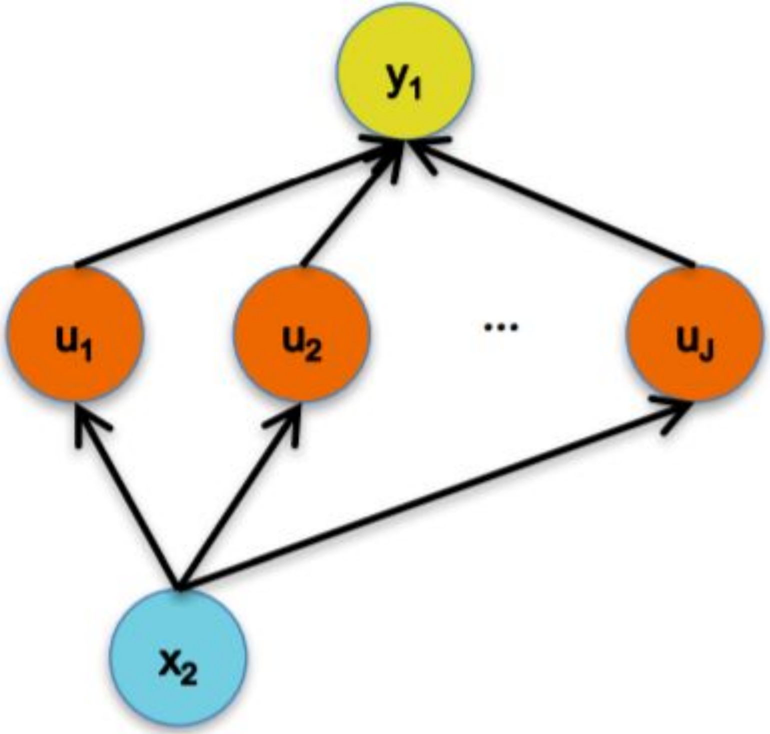
Back Propagation

- Chain Rule

Given: $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$.

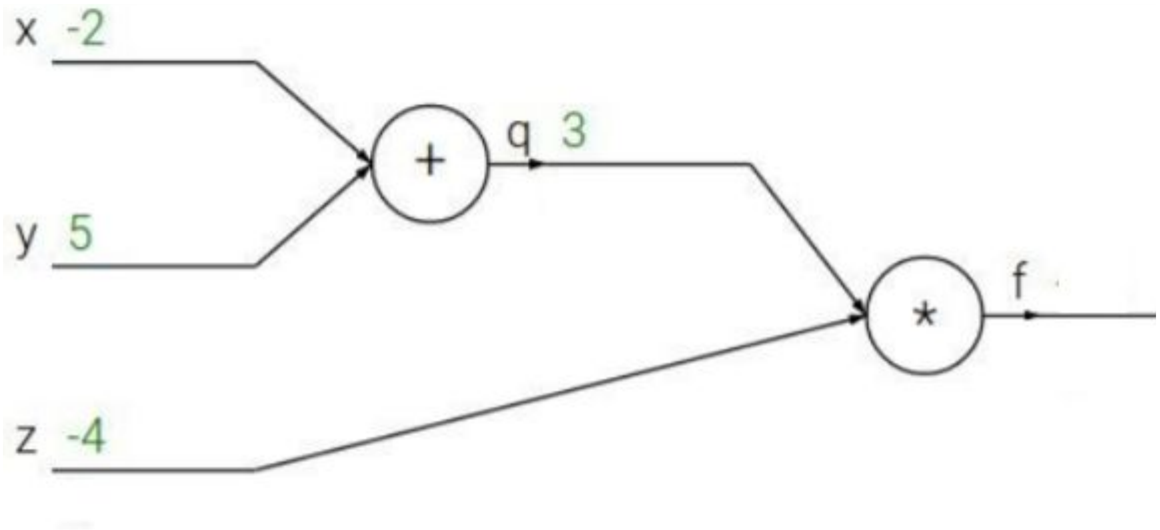
Chain Rule:

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j}\frac{du_j}{dx_k}, \qquad \forall i, k$$



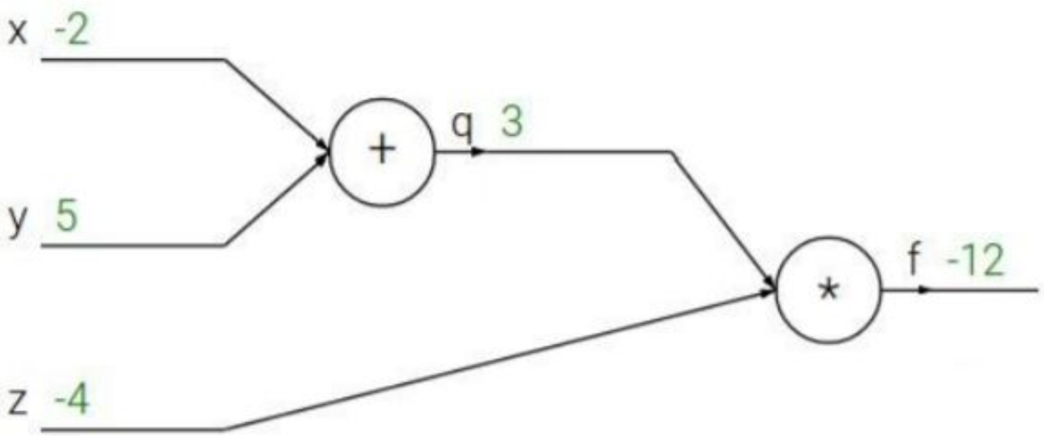
Back Propagation

- Let's do one example ...
  - Compute the forward pass in this computational graph



Back Propagation

- Let’s do one example ...
  - Compute the forward pass in this computational graph



$$f(x, y, z) = (x + y)z$$

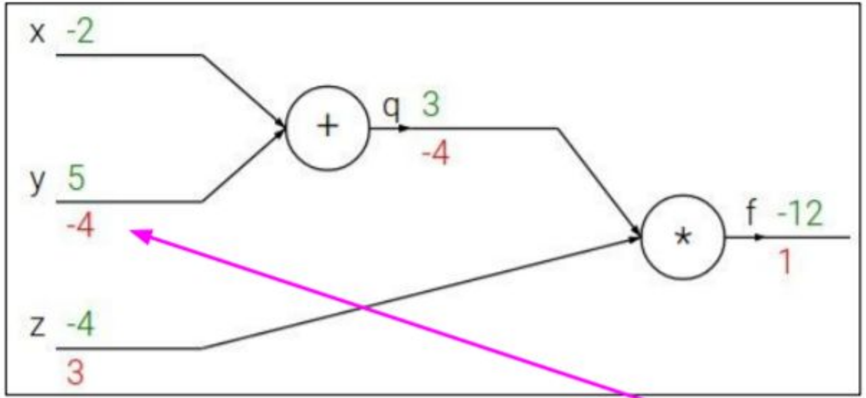
e.g. x = -2, y = 5, z = -4

Back Propagation

- Let's do one example ...
  - Compute $\dfrac{\partial f}{\partial y}$.

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$



$$\frac{\partial f}{\partial y}$$

Back Propagation

- Let's do one example ...
  - Compute $\frac{\partial f}{\partial y}$.

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

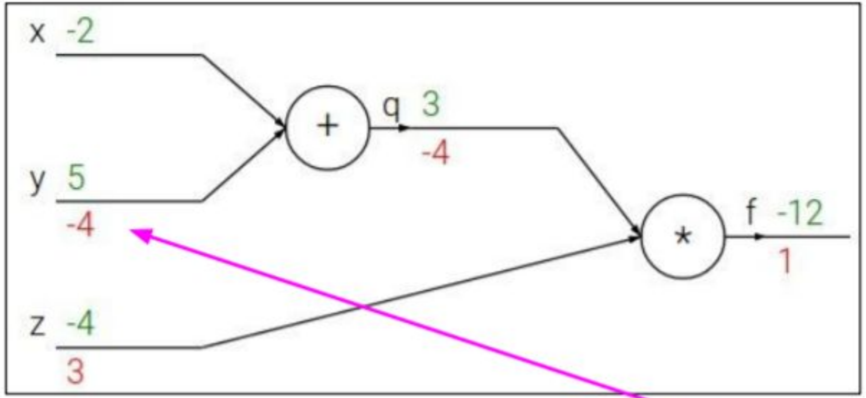
$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Upstream gradient

Local gradient



$$\frac{\partial f}{\partial y}$$

Back Propagation

- Let's do one example ...
  - Compute $\frac{\partial f}{\partial y}$.

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

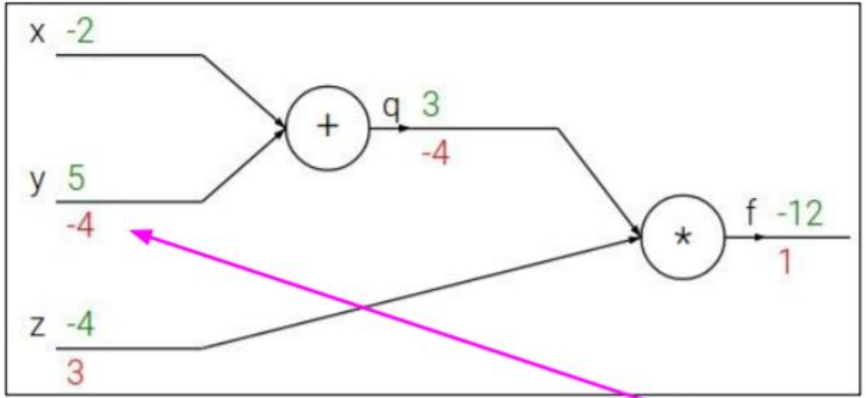
$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule:

$$\boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}}$$

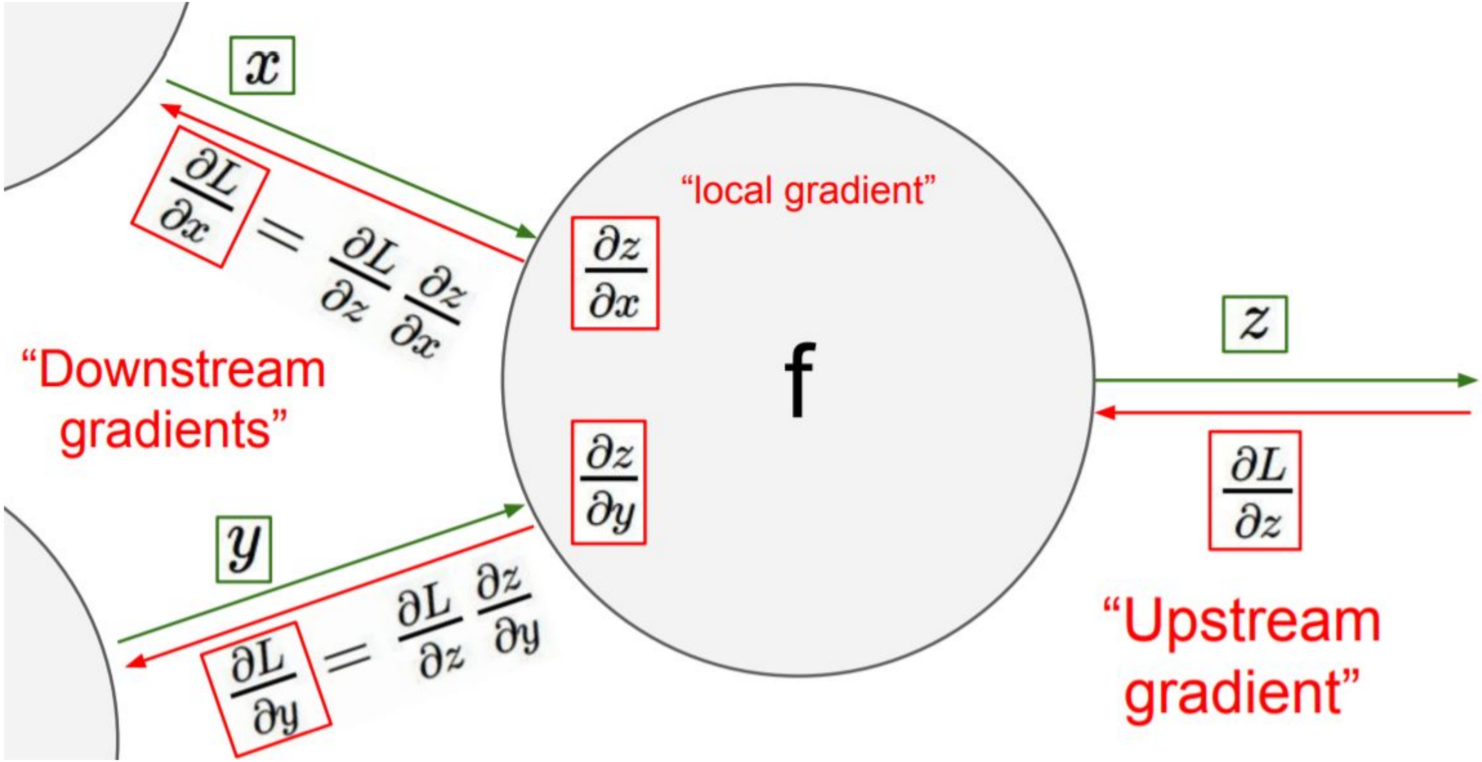
Upstream gradient

Local gradient



$$\frac{\partial f}{\partial y}$$

Back Propagation



Back Propagation

- Automatic Differentiation
  - Forward Pass
    - For each node in the computational graph (in topological order), assuming variable u_i and inputs v_1,...v_N:
      - Compute the u_i=g_i(v_1,...V_N) and cache the results.
  - Backward Pass
    - Calculate all local gradients.
    - For each node in the computational graph (in reverse topological order), for variable u_i=g_i(v_1,...V_N):
      - Use chain rule to calculate dy/dv_j=(dy/du_i)(du_i/dv_j)

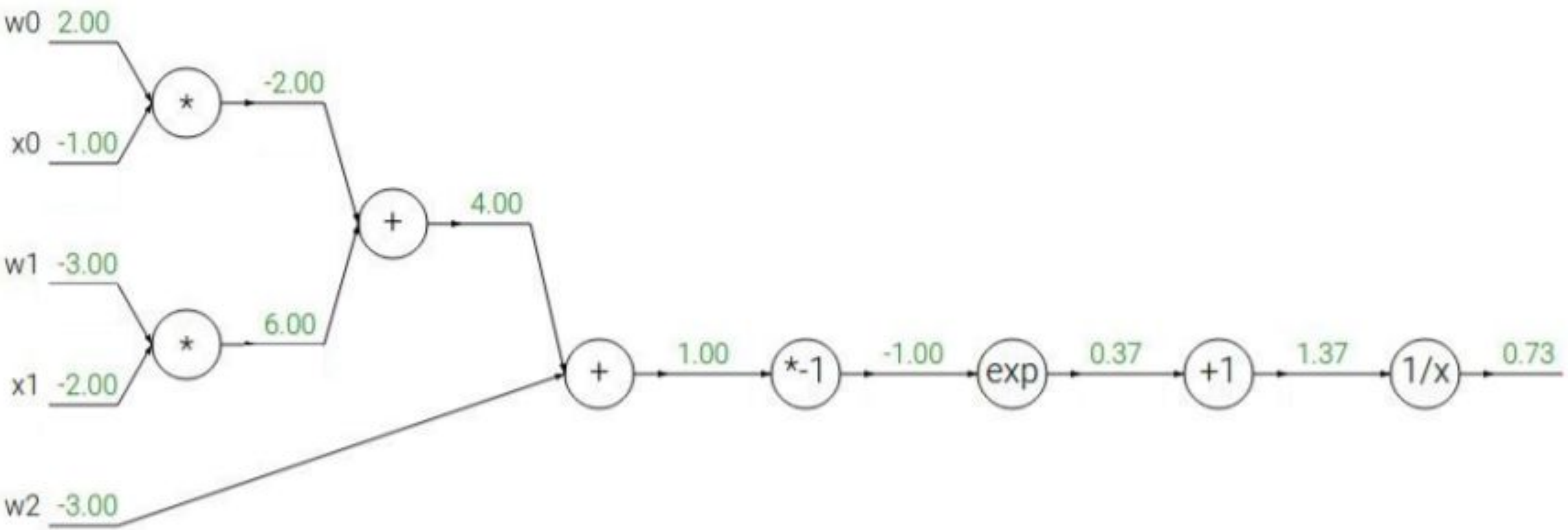
Back Propagation

- Use a computational graph to calculate the gradient of f with respect to x_i and w_i

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

Back Propagation

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



UCLA

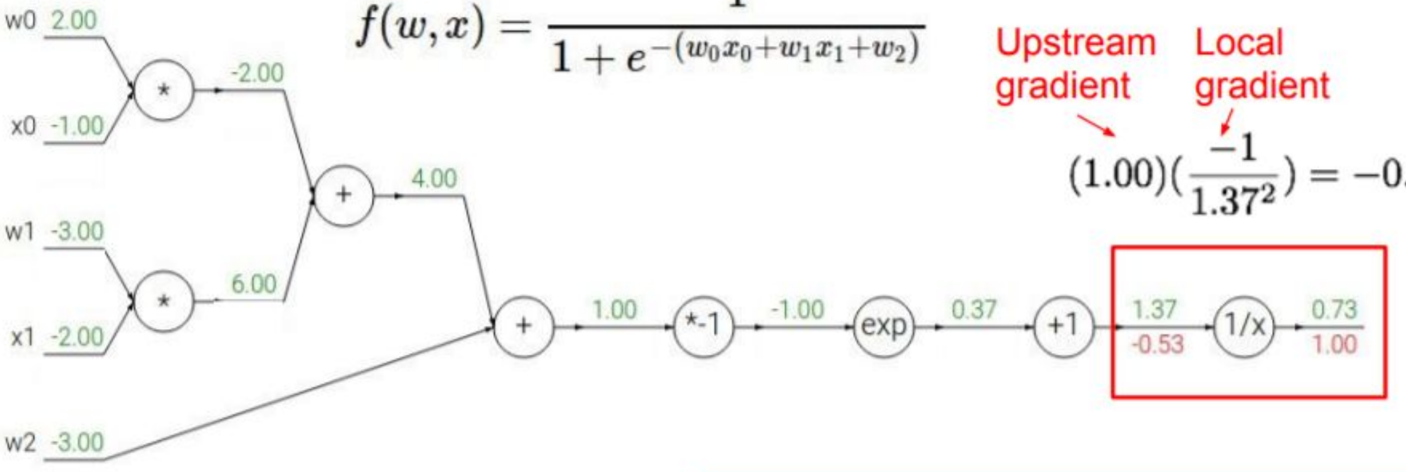
Back Propagation

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

Upstream gradient

Local gradient

$$(1.00)\left(\frac{-1}{1.37^2}\right) = -0.53$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

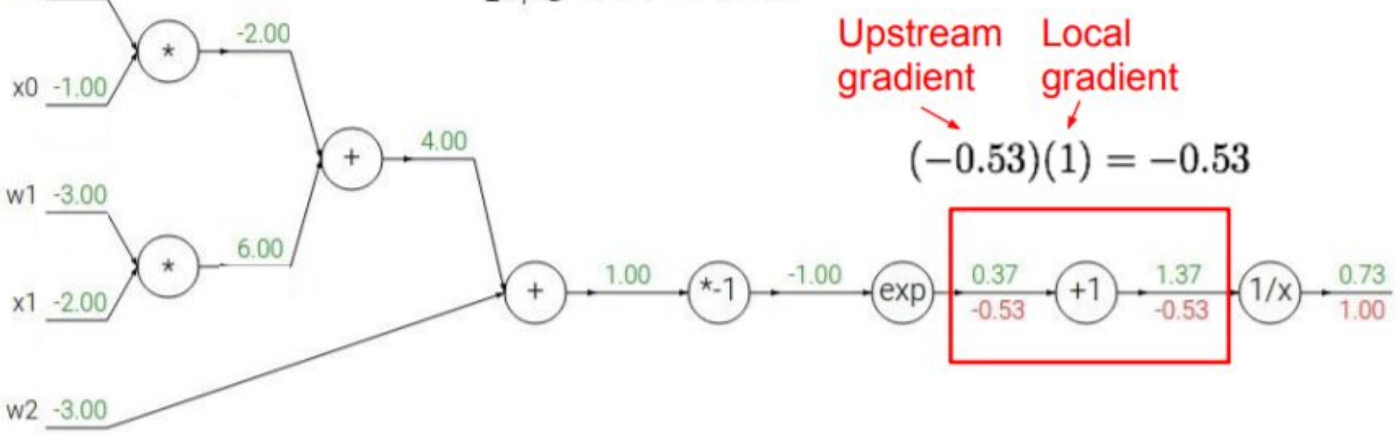
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Back Propagation

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



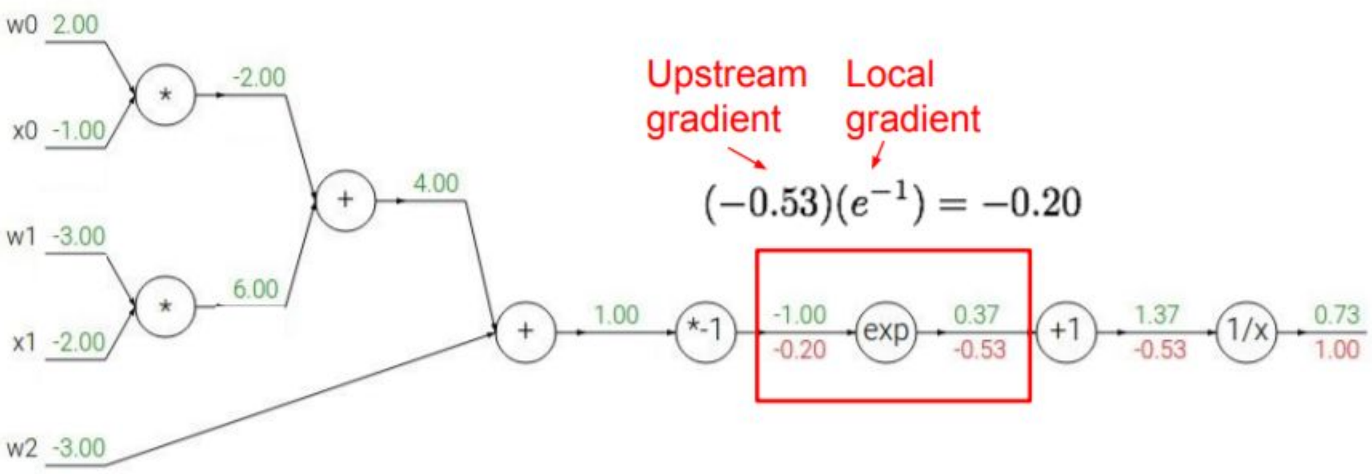
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$\boxed{f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1}$$

Back Propagation



$$(-0.53)(e^{-1}) = -0.20$$

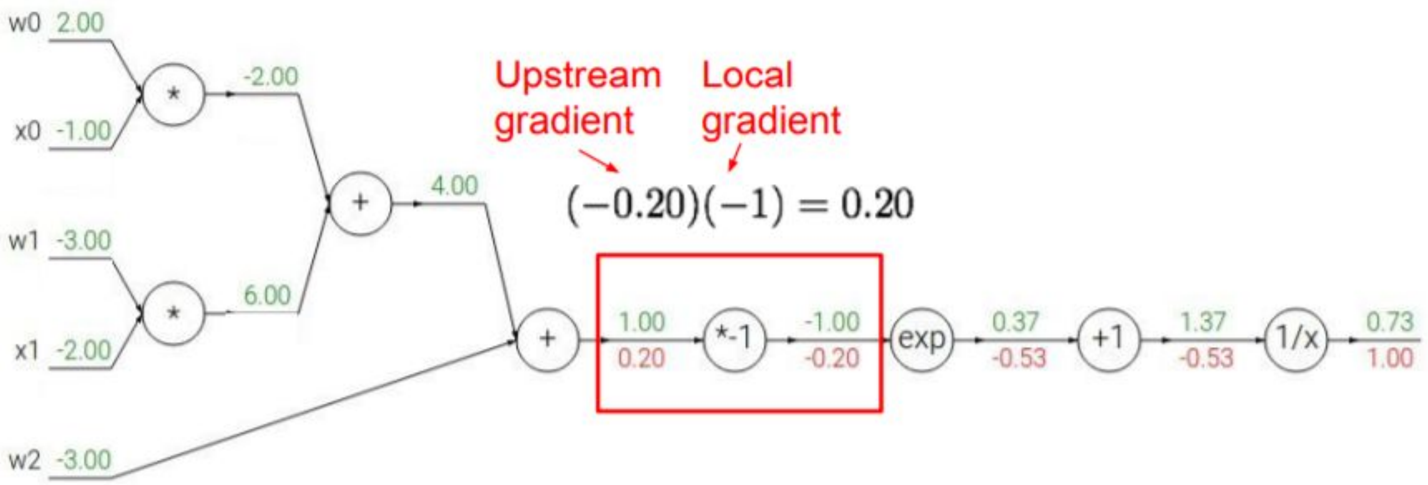
$$\boxed{f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x}$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Back Propagation



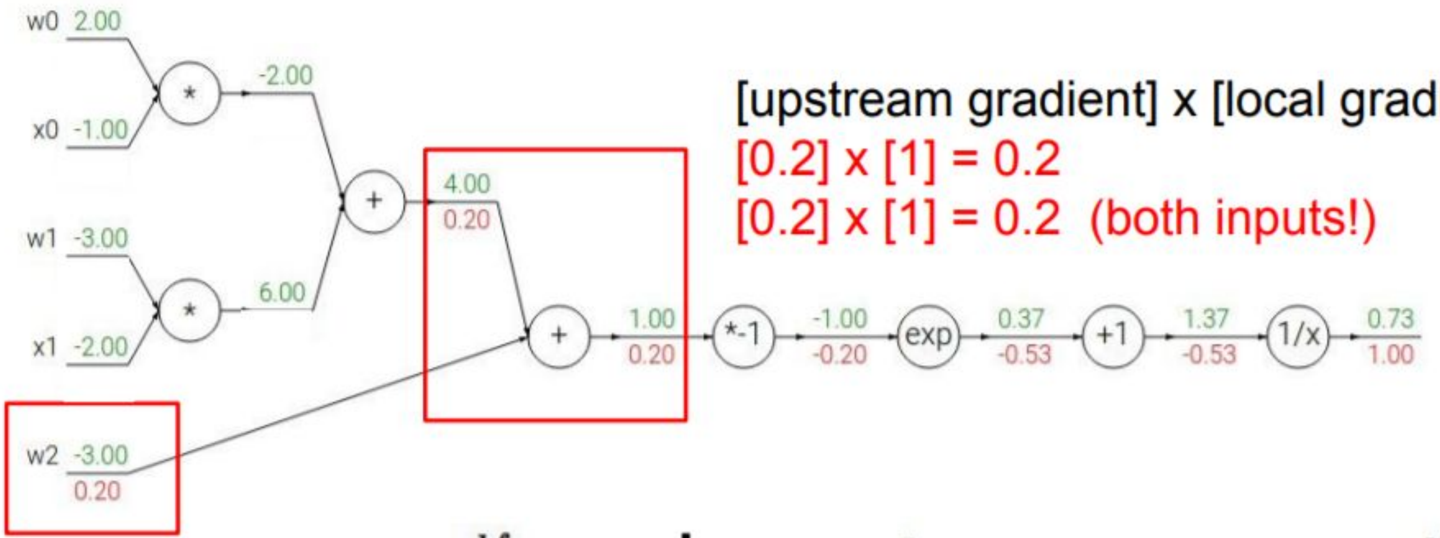
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$\boxed{f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a}$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Back Propagation



[upstream gradient] x [local gradient]

[0.2] x [1] = 0.2

[0.2] x [1] = 0.2 (both inputs!)

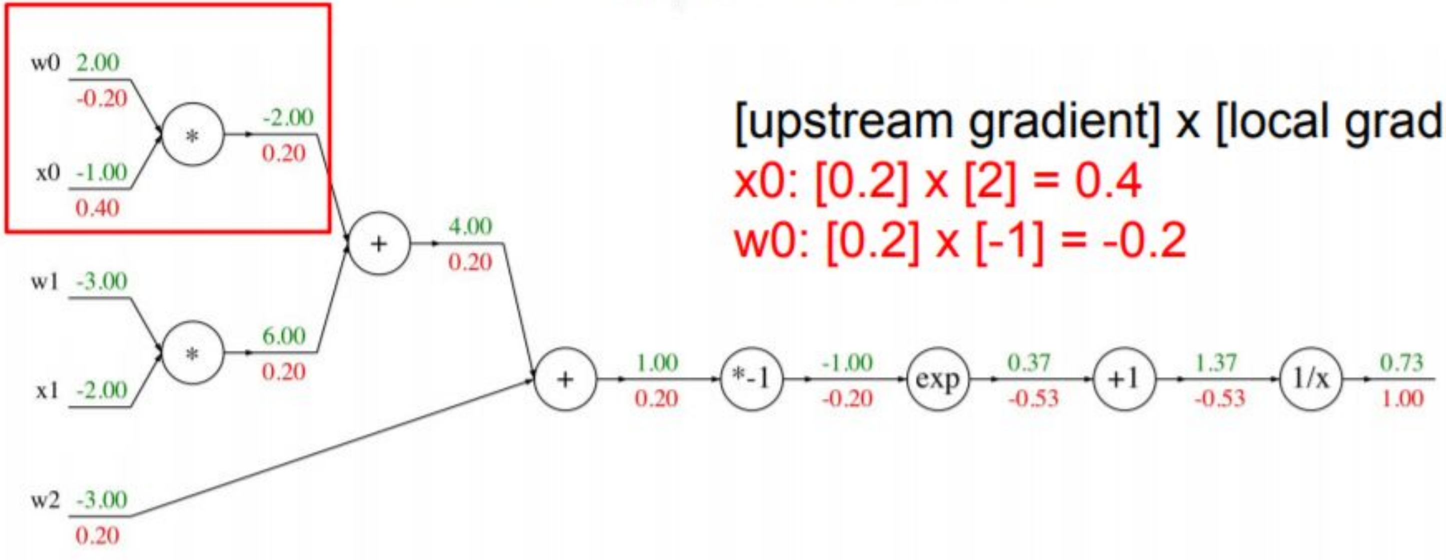
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Back Propagation



[upstream gradient] x [local gradient]

x0: [0.2] x [2] = 0.4

w0: [0.2] x [-1] = -0.2

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Thank you!