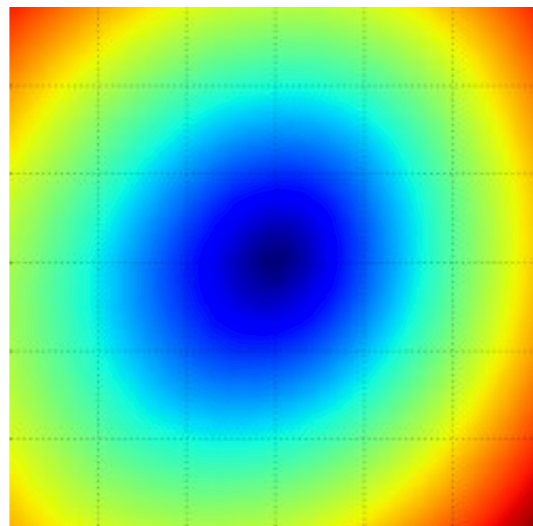


CS4670/5670: Computer Vision

Kavita Bala

Lecture 29: Optimization and Neural Nets



Slides from Andrej Karpathy and Fei-Fei Li
<http://vision.stanford.edu/teaching/cs231n/>

Today

- Optimization
- Today and Monday: Neural nets, CNNs
 - [Mon: http://cs231n.github.io/classification/](http://cs231n.github.io/classification/)
 - [Wed: http://cs231n.github.io/linear-classify/](http://cs231n.github.io/linear-classify/)
 - Today:
 - <http://cs231n.github.io/optimization-1/>
 - <http://cs231n.github.io/optimization-2/>

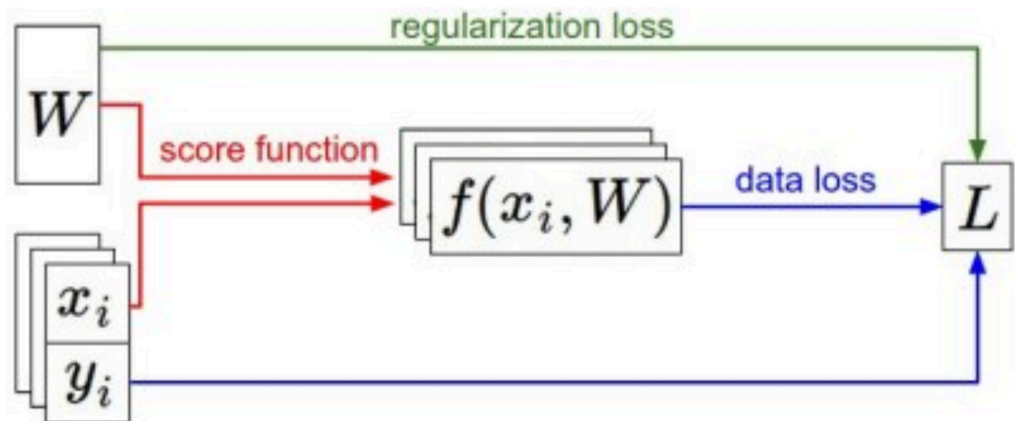
Summary

1. Score function

$$f(x_i, W, b) = Wx_i + b$$

2. Loss function

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda R(W)$$



Other loss functions

- Scores are not very intuitive
- Softmax classifier
 - Score function is same
 - Intuitive output: normalized class probabilities
 - Extension of logistic regression to multiple classes

Softmax classifier

$f(x_i, W) = Wx_i$ score function
is the same

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

softmax function

$$[1, -2, 0] \rightarrow [e^1, e^{-2}, e^0] = [2.71, 0.14, 1] \rightarrow [0.7, 0.04, 0.26]$$

Interpretation: squashes values into range 0 to 1

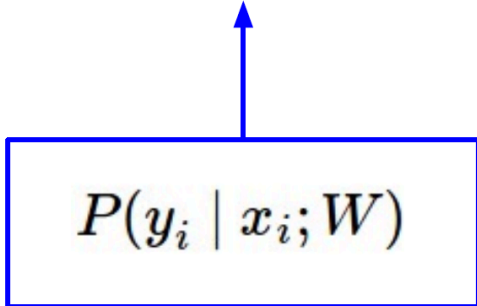
$$P(y_i | x_i; W)$$

Cross-entropy loss

$f(x_i, W) = Wx_i$ score function
is the same

$$L_i = -\log \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right)$$

$$L_i = -f_{y_i} + \log \sum_j e^{f_j}$$


$$P(y_i | x_i; W)$$

i.e. we're minimizing
the negative log
likelihood.

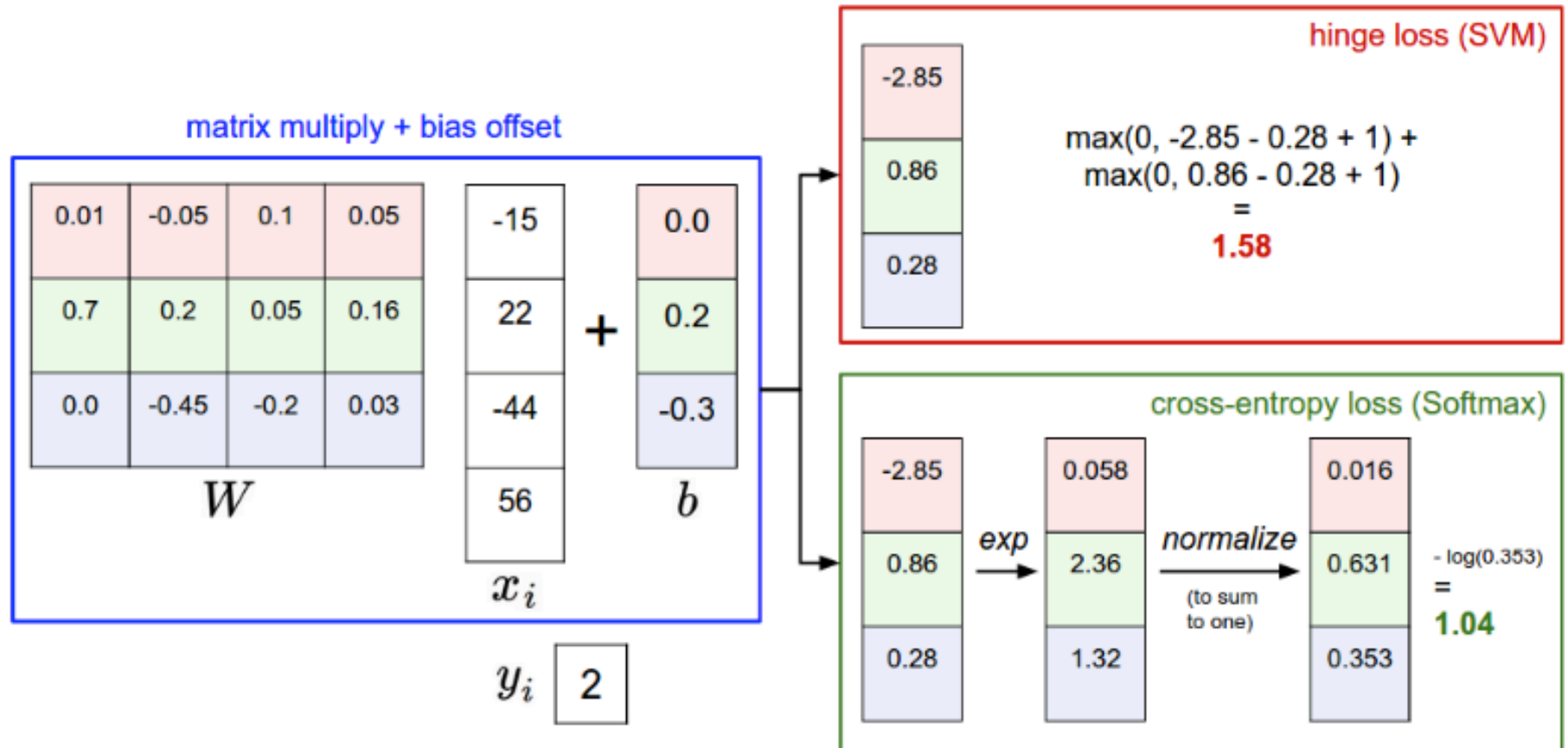
Aside: Loss function interpretation

- Probability
 - Maximum Likelihood Estimation (MLE)
 - Regularization is Maximum a posteriori (MAP) estimation

$$H(p, q) = - \sum_x p(x) \log q(x)$$

- Cross-entropy H
 - p is true distribution (1 for the correct class), q is estimated
 - Softmax classifier minimizes cross-entropy
 - Minimizes the KL divergence (Kullback-Leibler) between the distribution: distance between p and q

SVM vs. Softmax



Example of the difference between the SVM and Softmax classifiers for one datapoint. In both cases we compute the same score vector \mathbf{f} (e.g. by matrix multiplication in this section). The difference is in the interpretation of the scores in \mathbf{f} : The SVM interprets these as class scores and its loss function encourages the correct class (class 2, in blue) to have a score higher by a margin than the other class scores. The Softmax classifier instead interprets the scores as (unnormalized) log probabilities for each class and then encourages the (normalized) log probability of the correct class to be high (equivalently the negative of it to be low). The final loss for this example is 1.58 for the SVM and 1.04 for the Softmax classifier, but note that these numbers are not comparable; They are only meaningful in relation to loss computed within the same classifier and with the same data.

Summary

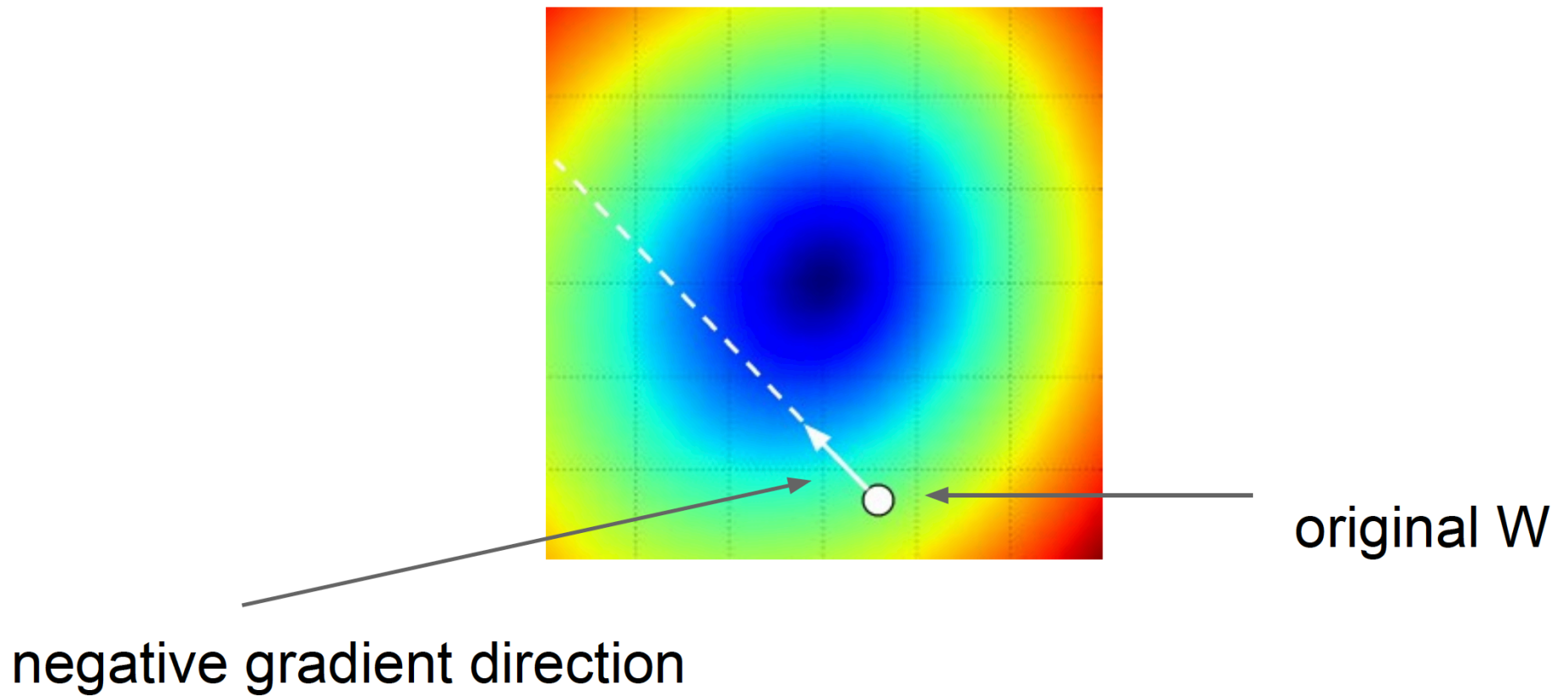
- Have score function and loss function
 - Will generalize the score function
- Find W and b to minimize loss
 - SVM vs. Softmax
 - Comparable in performance
 - SVM satisfies margins, softmax optimizes probabilities

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda \sum_k \sum_l W_{k,l}^2$$

$$L = \frac{1}{N} \sum_i -\log \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) + \lambda \sum_k \sum_l W_{k,l}^2$$

Gradient Descent





Step size: learning rate
Too big: will miss the minimum
Too small: slow convergence

Analytic Gradient

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

$$\nabla_{w_j} L_i = \mathbf{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

$$\nabla_{w_{y_i}} L_i = - \left(\sum_{j \neq y_i} \mathbf{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i$$

In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

=>

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check**.

Gradient Descent

```
# Vanilla Gradient Descent
```

```
while True:
```

```
    weights_grad = evaluate_gradient(loss_fun, data, weights)
```

```
    weights += - step_size * weights_grad # perform parameter update
```

Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

```
# Vanilla Minibatch Gradient Descent  
  
while True:  
    data_batch = sample_training_data(data, 256) # sample 256 examples  
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)  
    weights += - step_size * weights_grad # perform parameter update
```


Common mini-batch sizes are ~100 examples.

e.g. Krizhevsky ILSVRC ConvNet used 256 examples

Stochastic Gradient Descent (SGD)

- use a single example at a time

```
# Vanilla Minibatch Gradient Descent  
  
while True:  
    data_batch = sample_training_data(data, 256) # sample 256 examples  
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)  
    weights += - step_size * weights_grad # perform parameter update
```



(also sometimes called **on-line** Gradient Descent)

Summary

- Always use mini-batch gradient descent
- Incorrectly refer to it as “doing SGD” as everyone else
(or call it batch gradient descent)
- The mini-batch size is a hyperparameter, but it is not very common to cross-validate over it (usually based on practical concerns, e.g. space/time efficiency)

The dynamics of Gradient Descent

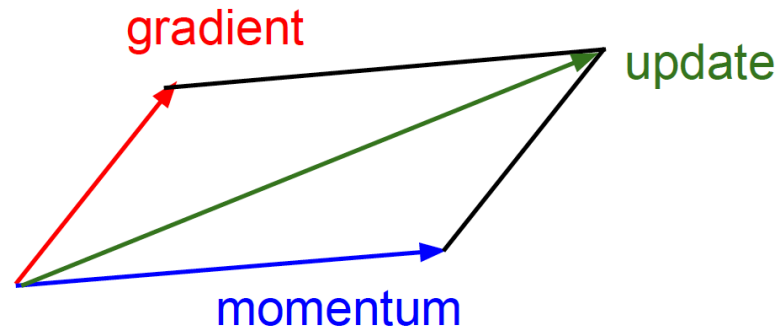
pull some weights up and some down

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda \sum_k \sum_l W_{k,l}^2$$

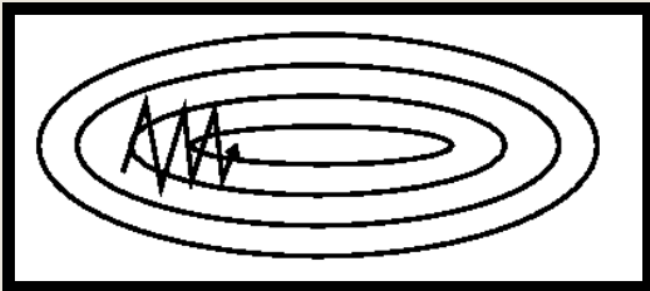
$$L = \frac{1}{N} \sum_i -\log \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) + \lambda \sum_k \sum_l W_{k,l}^2$$

always pull the weights down

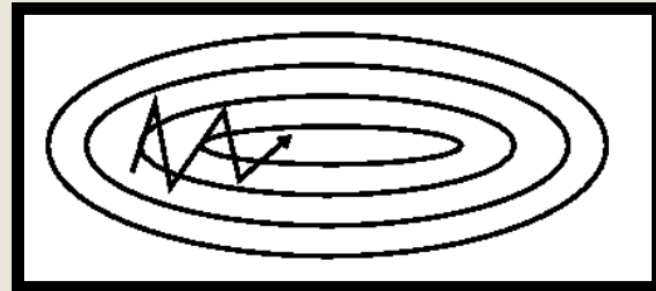
Momentum Update



```
weights_grad = evaluate_gradient(loss_fun, data, weights)
vel = vel * 0.9 - step_size * weights_grad
weights += vel
```



(Fig. 2a)



(Fig. 2b)

Many other ways to perform optimization...

- Second order methods that use the Hessian (or its approximation): BFGS, **LBFGS**, etc.
- Currently, the lesson from the trenches is that well-tuned SGD+Momentum is very hard to beat for CNNs.

Where are we?

- Classifiers: SVM vs. Softmax
- Gradient descent to optimize loss functions
 - Batch gradient descent, stochastic gradient descent
 - Momentum
 - Numerical gradients (slow, approximate), analytic gradients (fast, error-prone)

Derivatives

- Given $f(x)$, where x is vector of inputs
 - Compute gradient of f at x : $\nabla f(x)$

Examples

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

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

$$f(x+h) = f(x) + h \frac{df(x)}{dx}$$

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

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

$$f(x+h) = f(x) + h \frac{df(x)}{dx}$$

Example: $x = 4, y = -3.$ $\Rightarrow f(x,y) = -12$

$$\frac{\partial f}{\partial x} = -3$$

$$\frac{\partial f}{\partial y} = 4$$

partial derivatives

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]$$

gradient

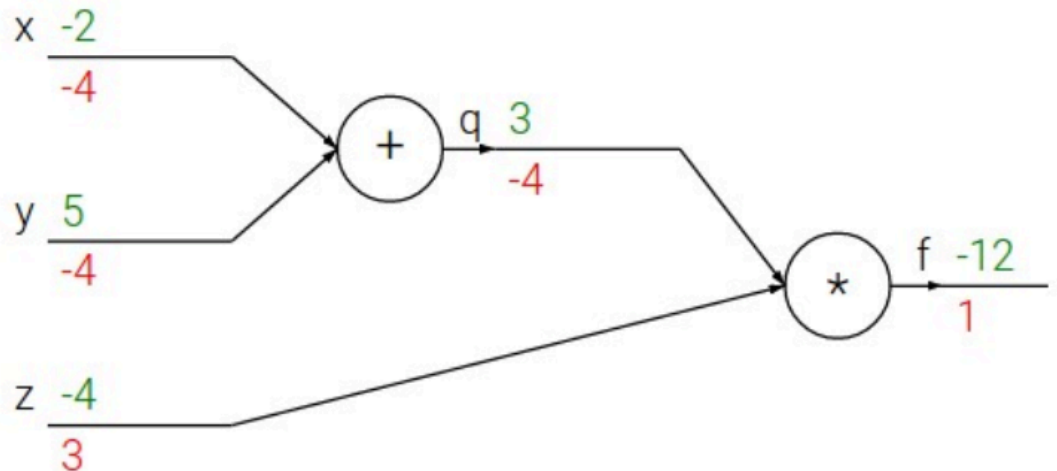
Compound expressions: $f(x, y, z) = (x + y)z$

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

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

Chain rule:

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



Backprop gradients

Every gate during backprop computes, for all its inputs:

$$[\text{LOCAL GRADIENT}] \times [\text{GATE GRADIENT}]$$



Can be computed right away,
even during forward pass



The gate receives this during
backpropagation

- Can create complex stages, but easily compute gradient