Machine Learning for Image Analysis

Lecture 2: Loss- and activation functions, efficient gradient descent

Dagmar Kainmueller, 23.04.2024

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

MAX DELBRÜCK CENTER

Course Outline

- Introduction to Image Analysis
- **Basics: Neural Networks**
- Convolutional Neural Networks
- Transformers
- Model Interpretability
- Self-supervised Learning
- Generative Models (GANs, Diffusion)

Neural Networks

- Perceptrons
- Multi-layer Perceptrons
- Backpropagation
- Image classification
- Loss- and activation functions for more than two classes
- Better activation functions
- Better gradient descent

Backpropagation

A simple example

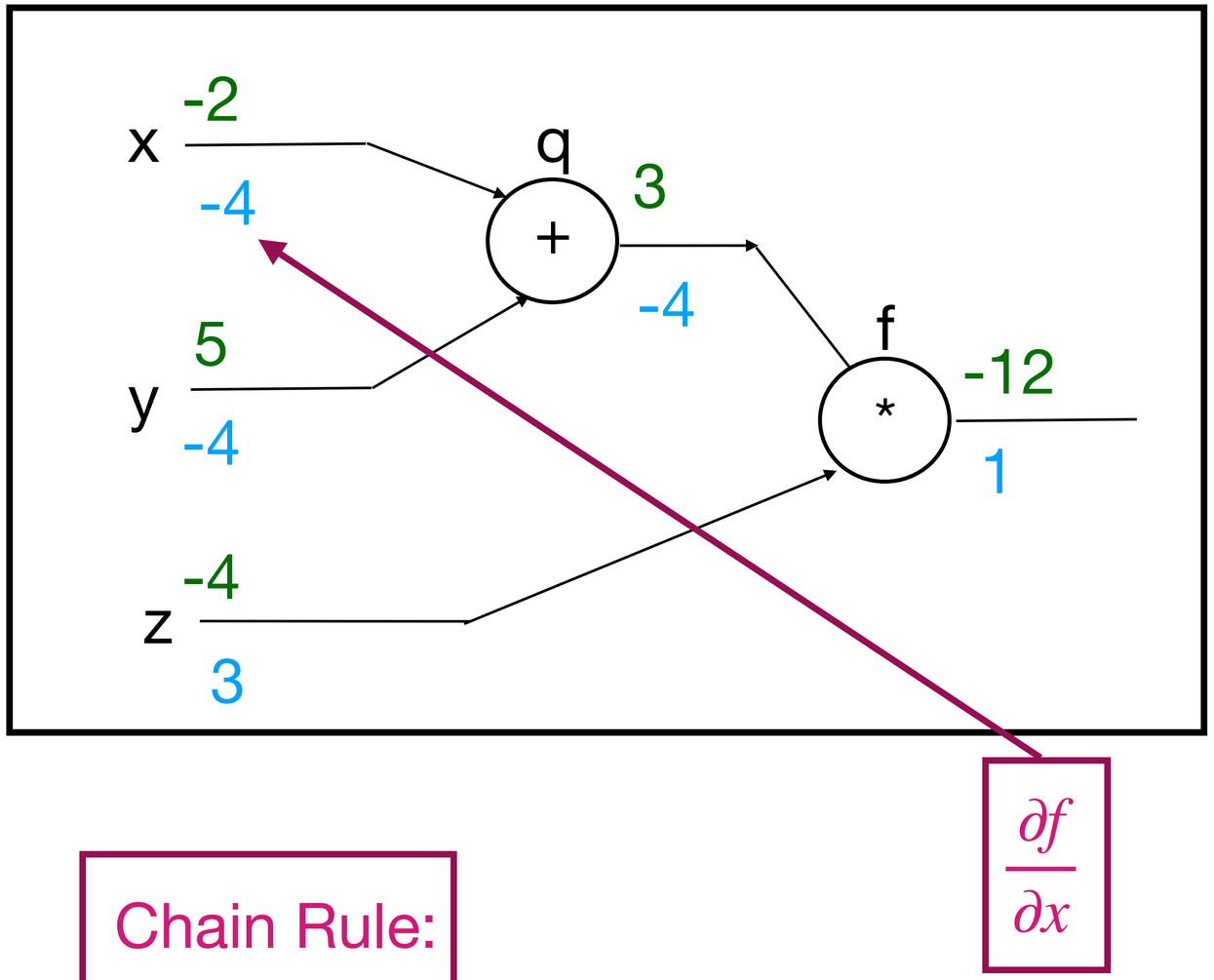
$$f(x, y, z) = (x + y)z$$

Let $x = -2, y = 5, z = -4$

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

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

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



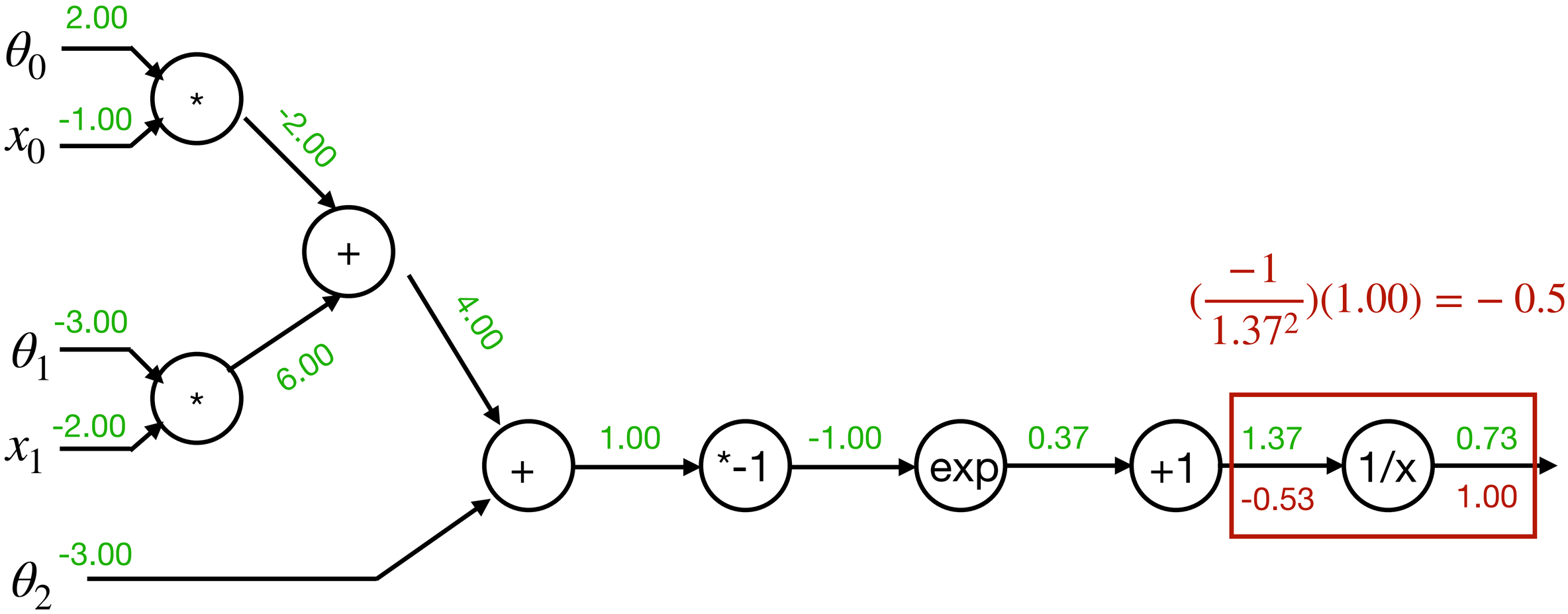
Chain Rule:

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

Back-propagation

Another example

$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$

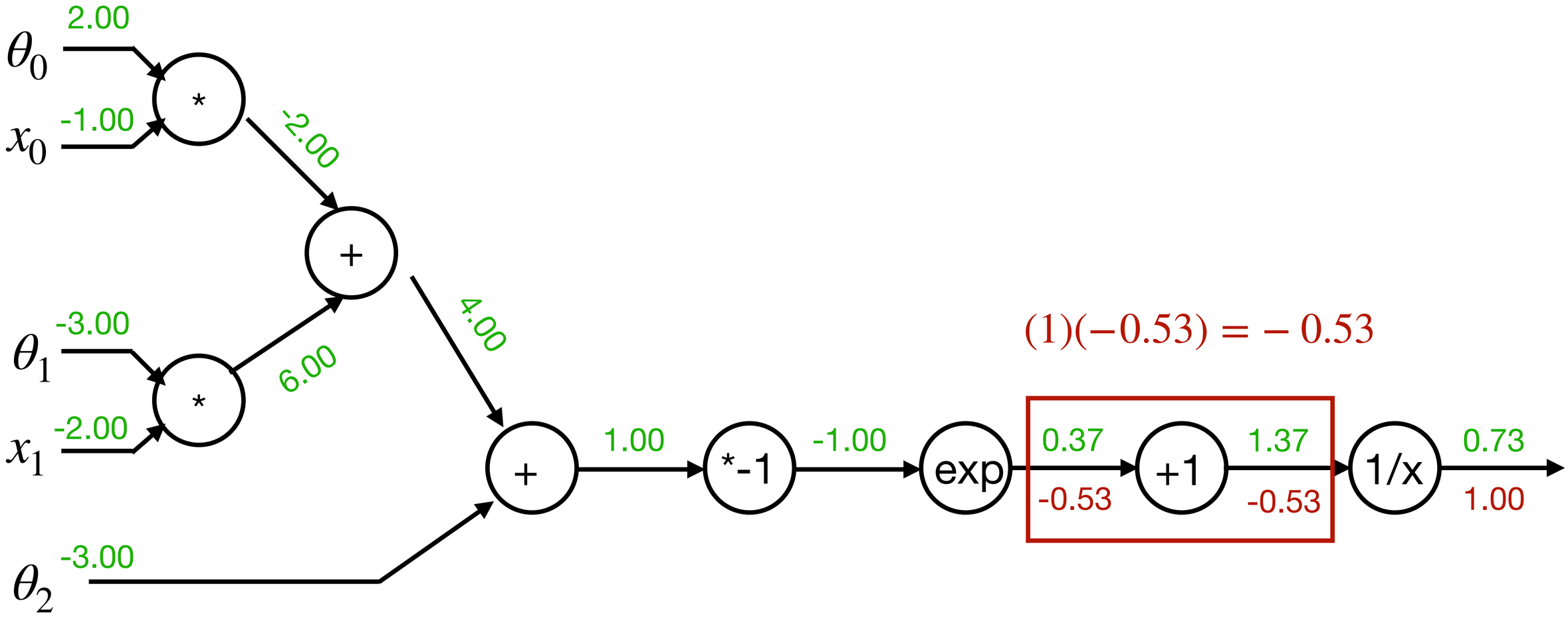


Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Back-propagation

Another example

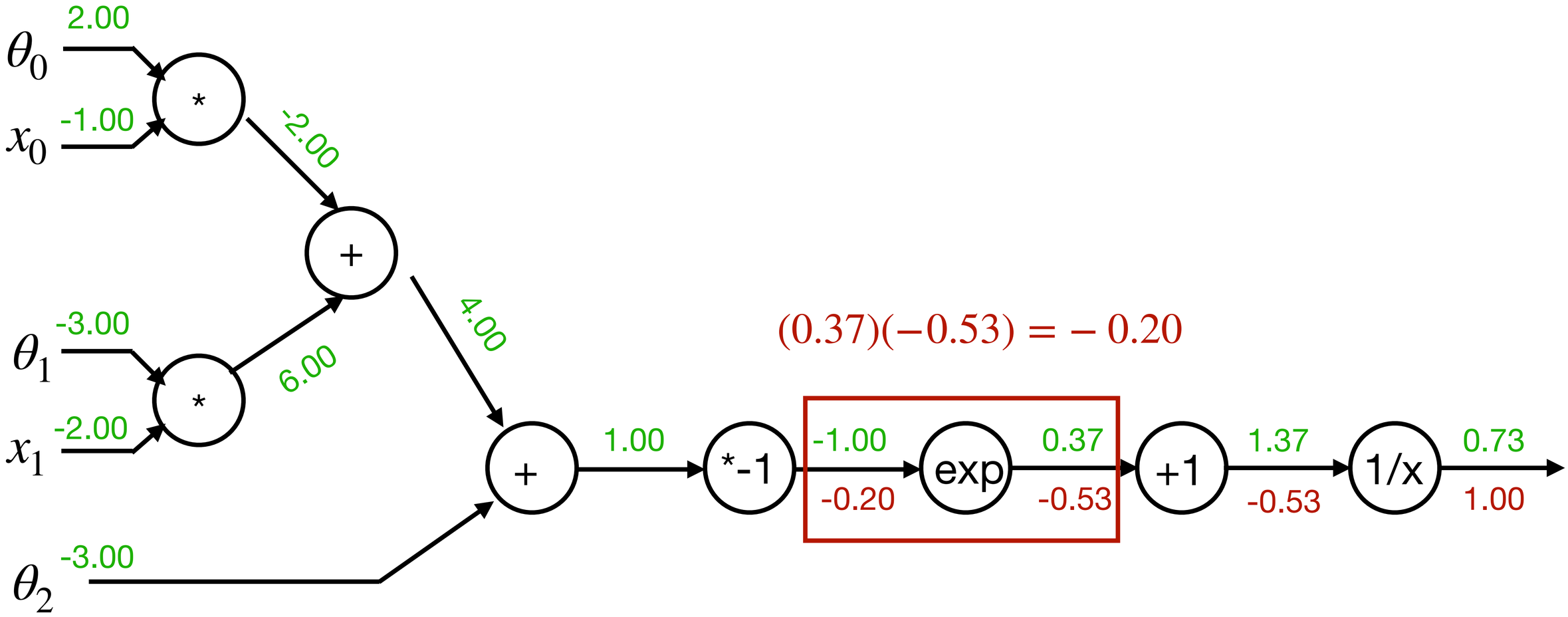
$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$



Back-propagation

Another example

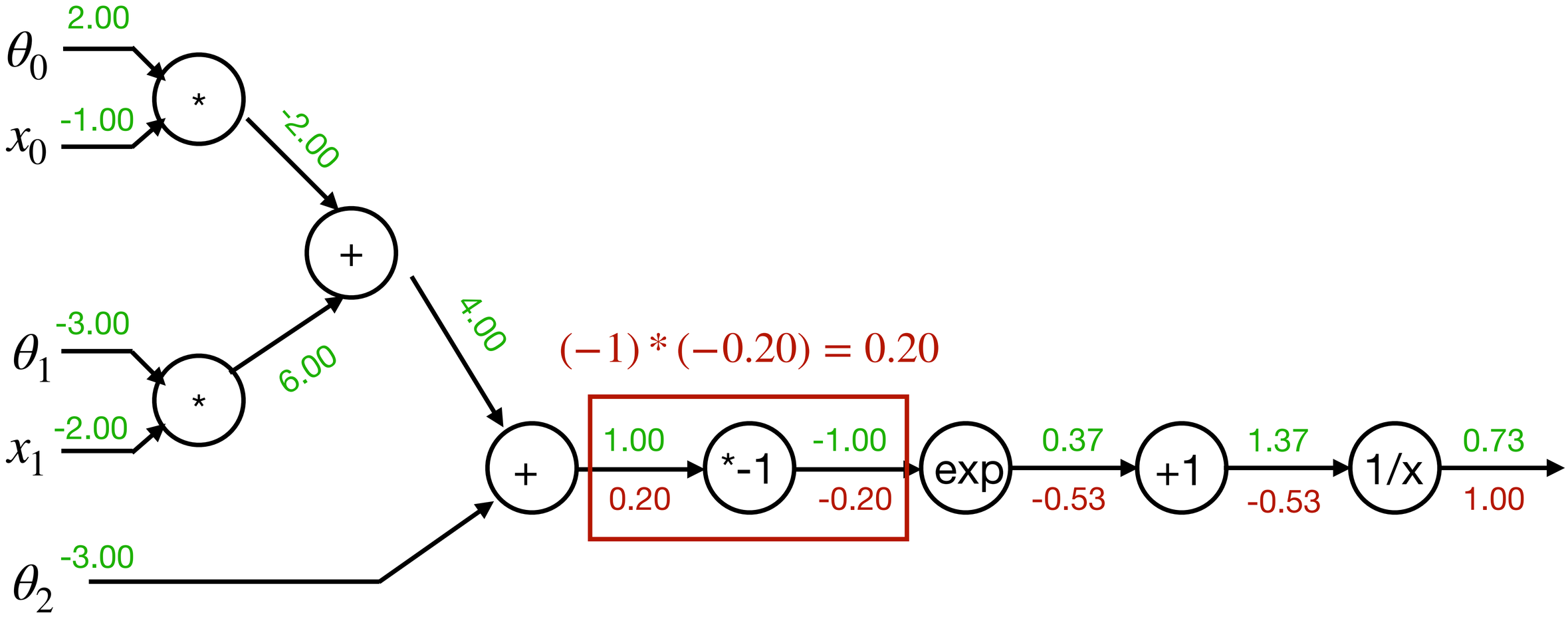
$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$



Back-propagation

Another example

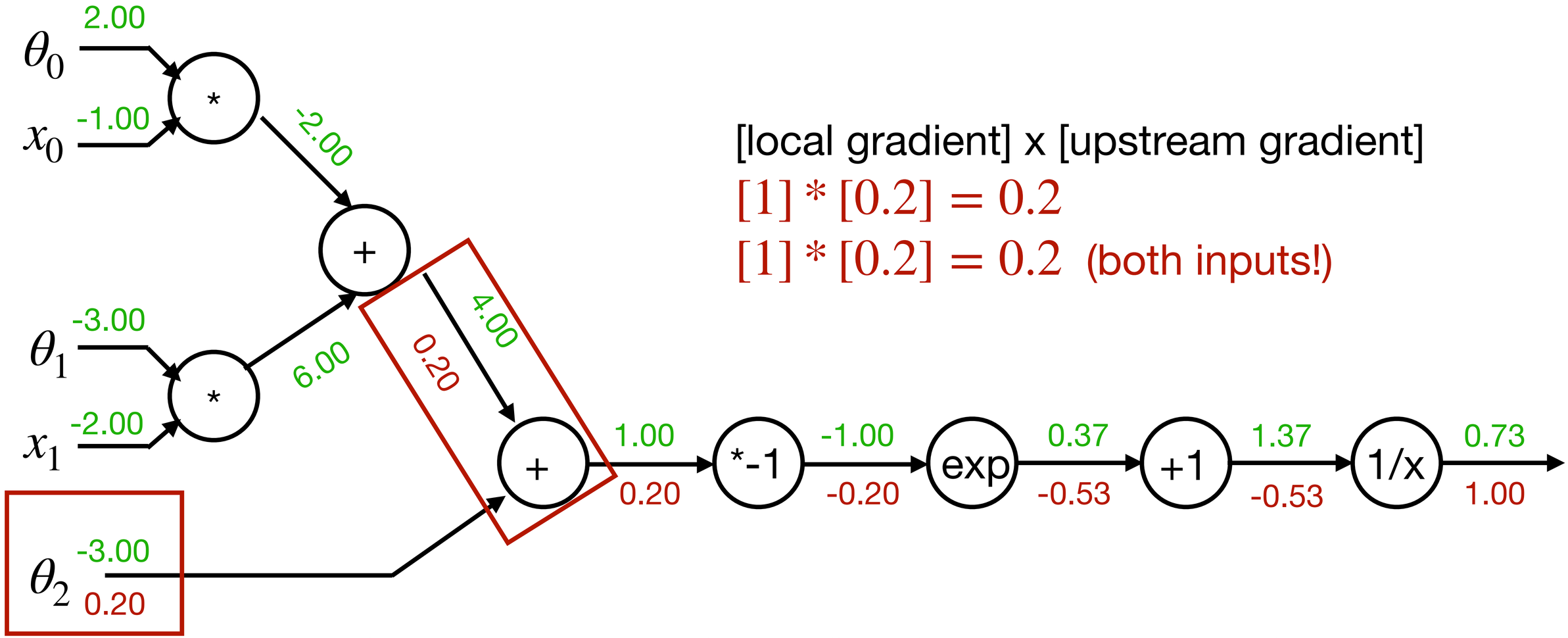
$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$



Back-propagation

Another example

$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$



[local gradient] x [upstream gradient]

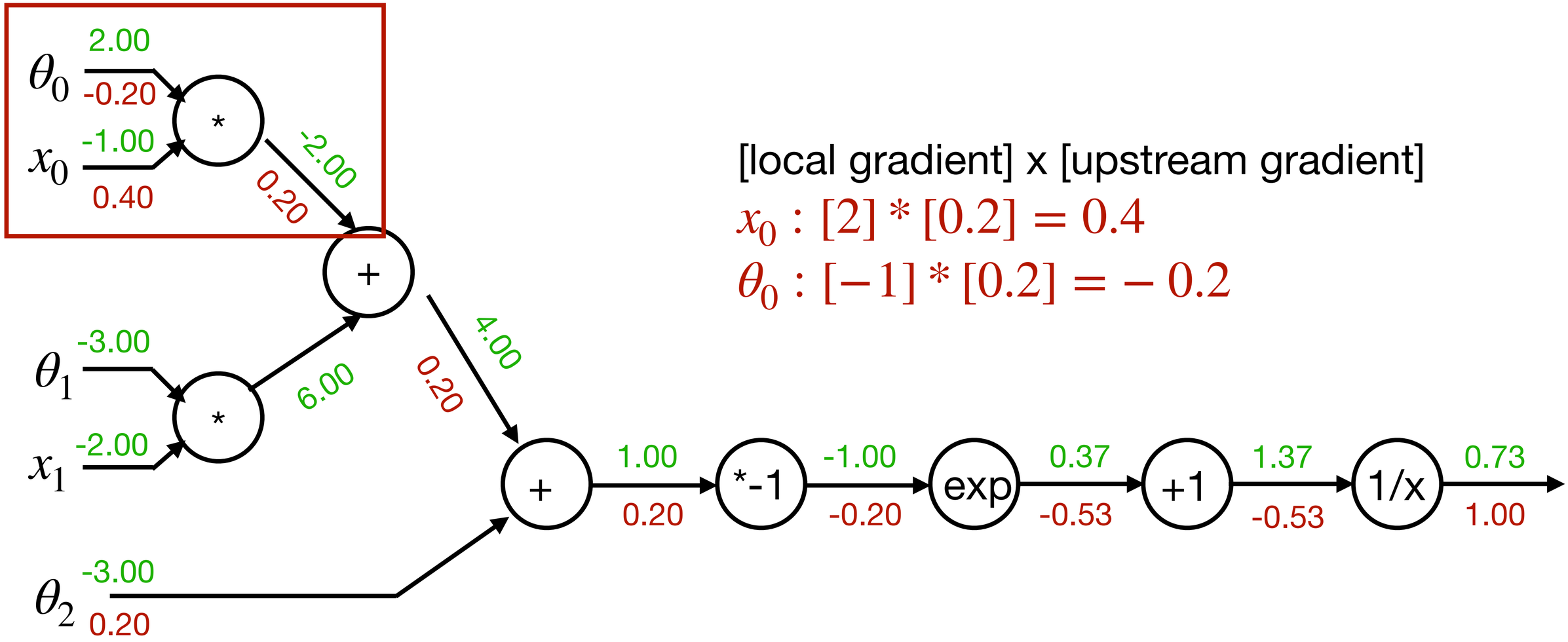
$[1] * [0.2] = 0.2$

$[1] * [0.2] = 0.2$ (both inputs!)

Back-propagation

Another example

$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$



[local gradient] x [upstream gradient]

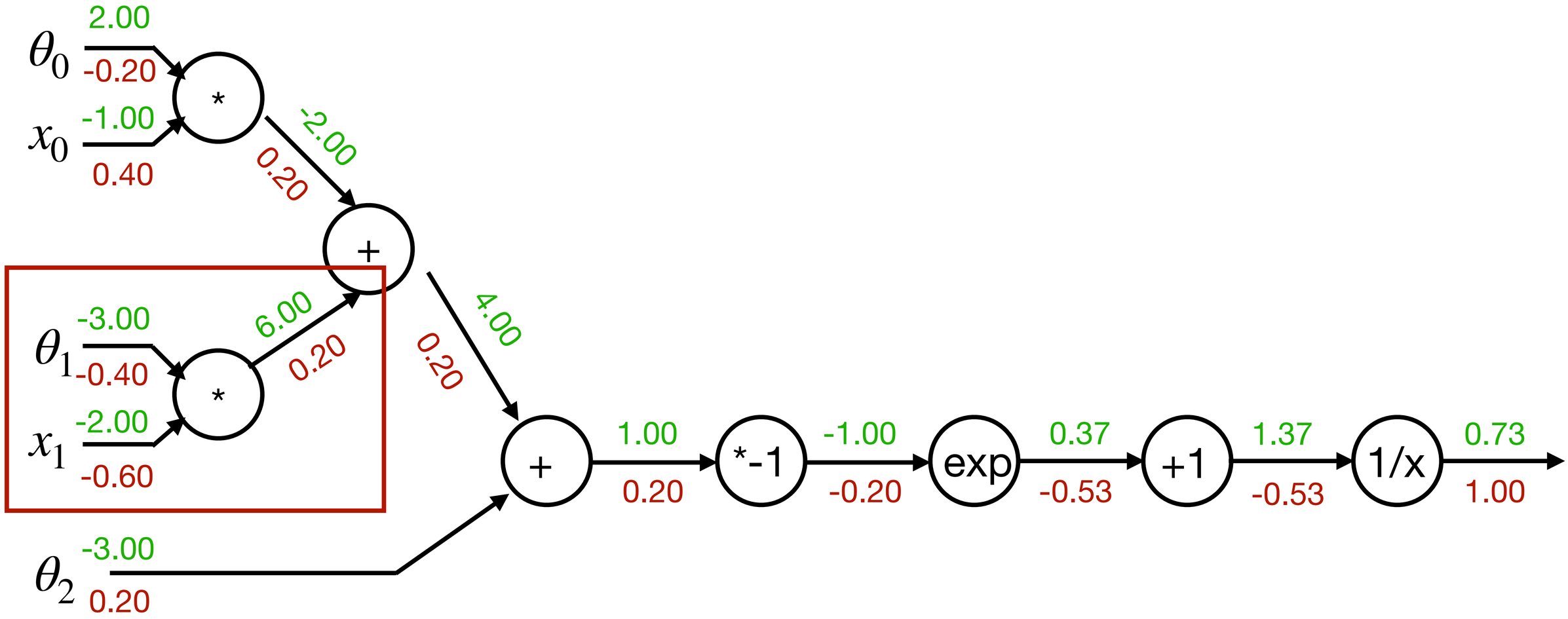
$x_0 : [2] * [0.2] = 0.4$

$\theta_0 : [-1] * [0.2] = -0.2$

Back-propagation

Another example

$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$



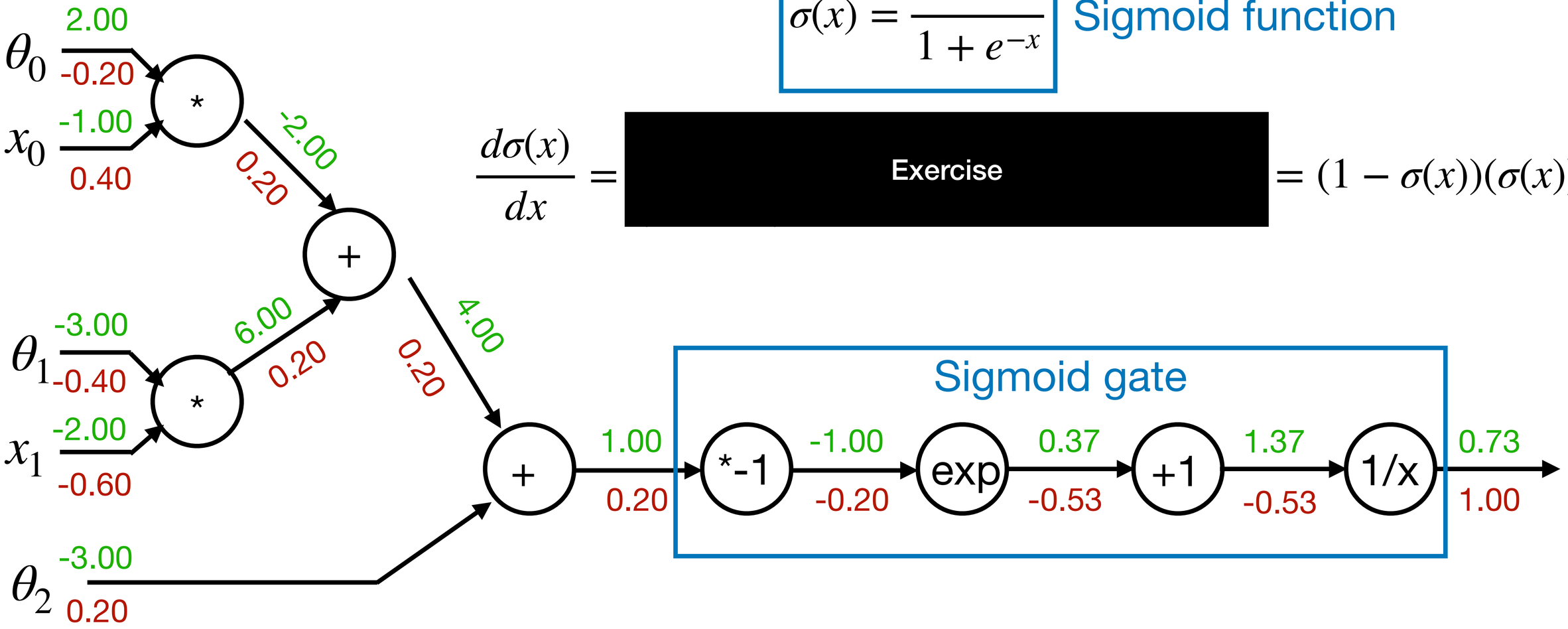
Back-propagation

Another example

$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$

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

$$\frac{d\sigma(x)}{dx} = \text{Exercise} = (1 - \sigma(x))(\sigma(x))$$



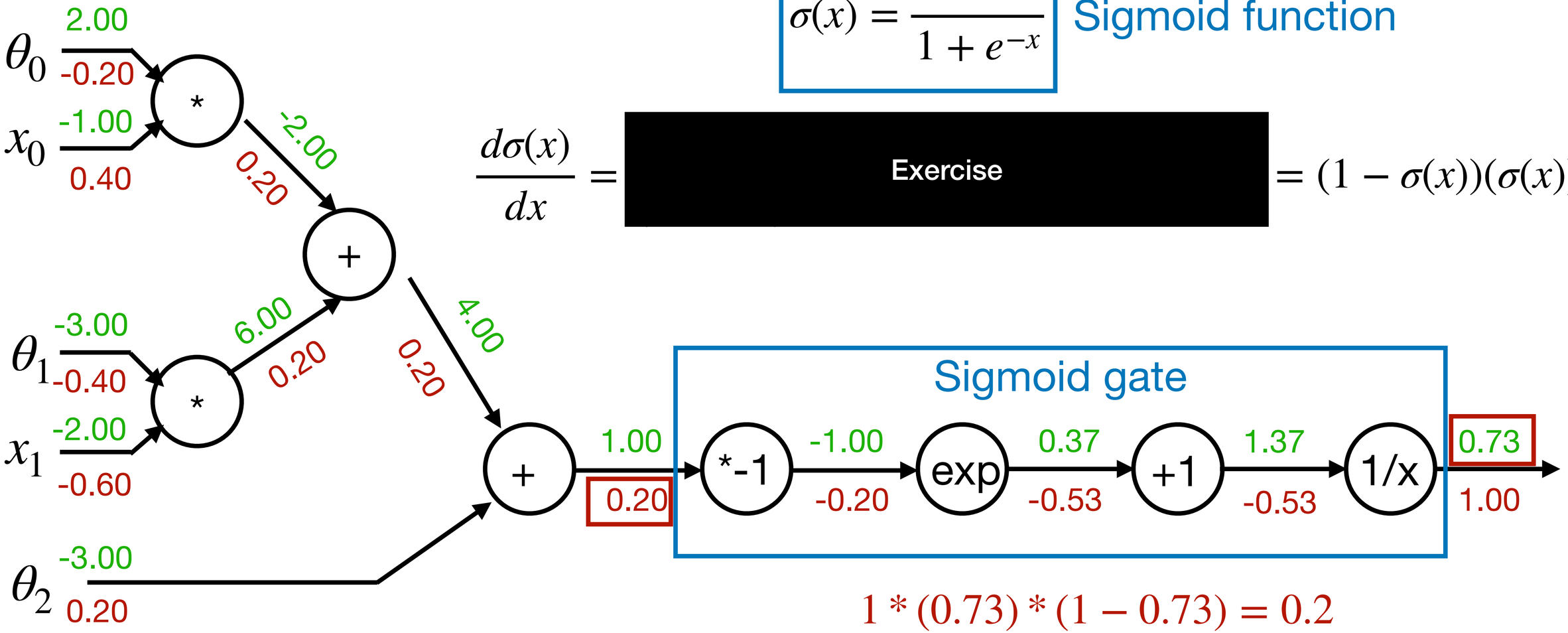
Back-propagation

Another example

$$f(\theta, x) = \frac{1}{1 + e^{-(\theta_0 x_0 + \theta_1 x_1 + \theta_2)}}$$

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

$$\frac{d\sigma(x)}{dx} = \text{Exercise} = (1 - \sigma(x))(\sigma(x))$$

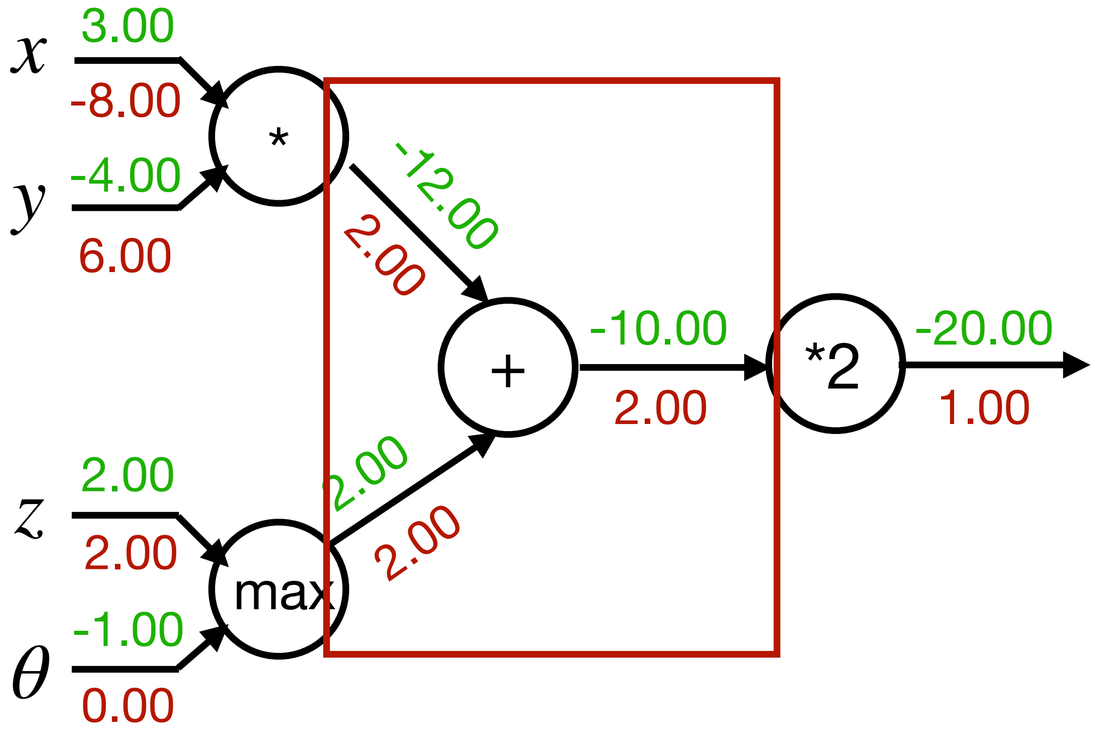


$$1 * (0.73) * (1 - 0.73) = 0.2$$

Patterns in backward flow

Another example

add gate: gradient distributor

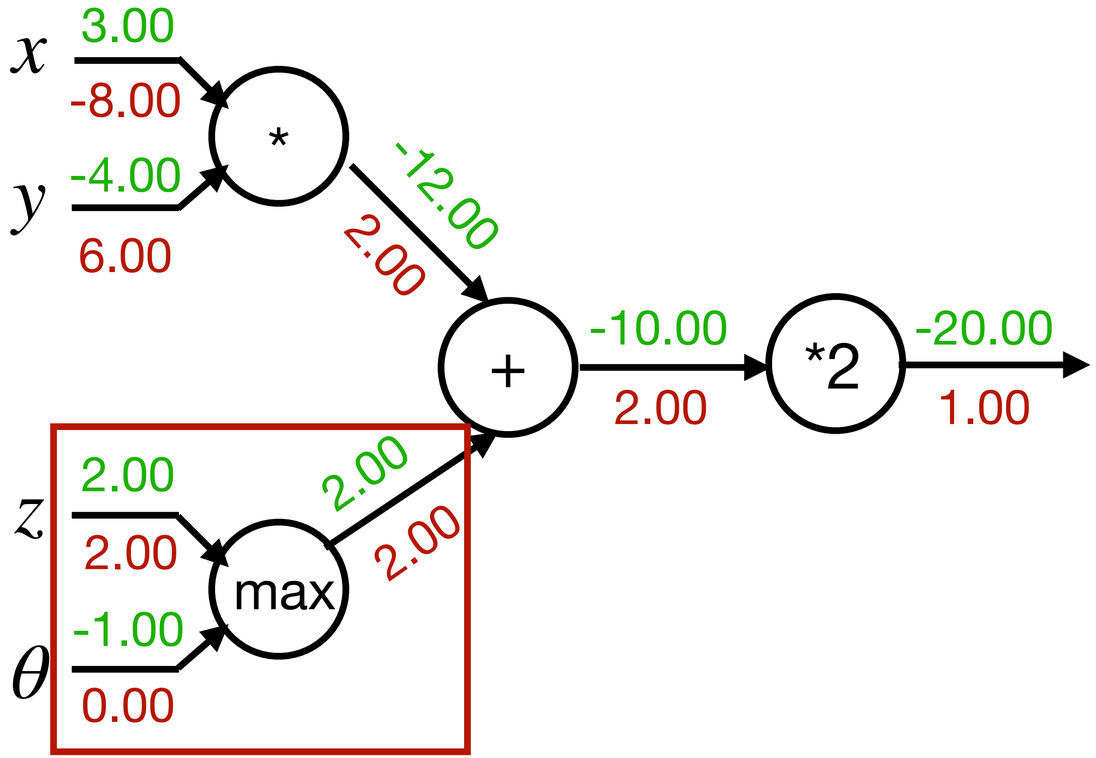


Patterns in backward flow

Another example

add gate: gradient distributor

max gate: gradient router



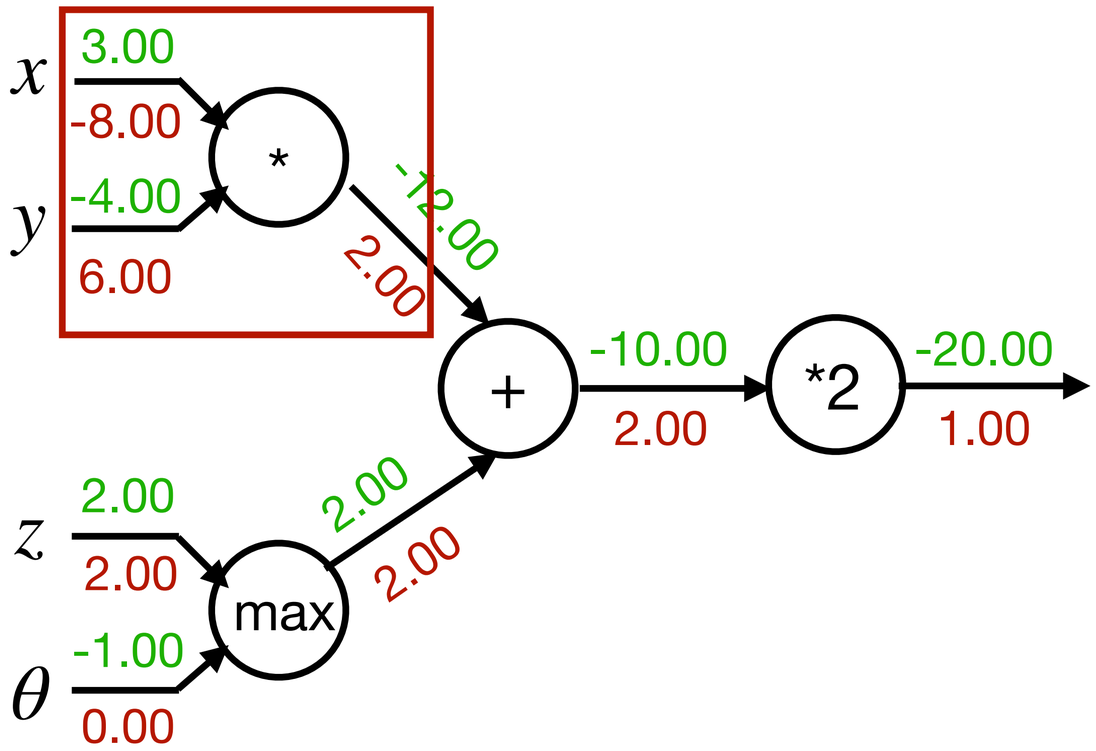
Patterns in backward flow

Another example

add gate: gradient distributor

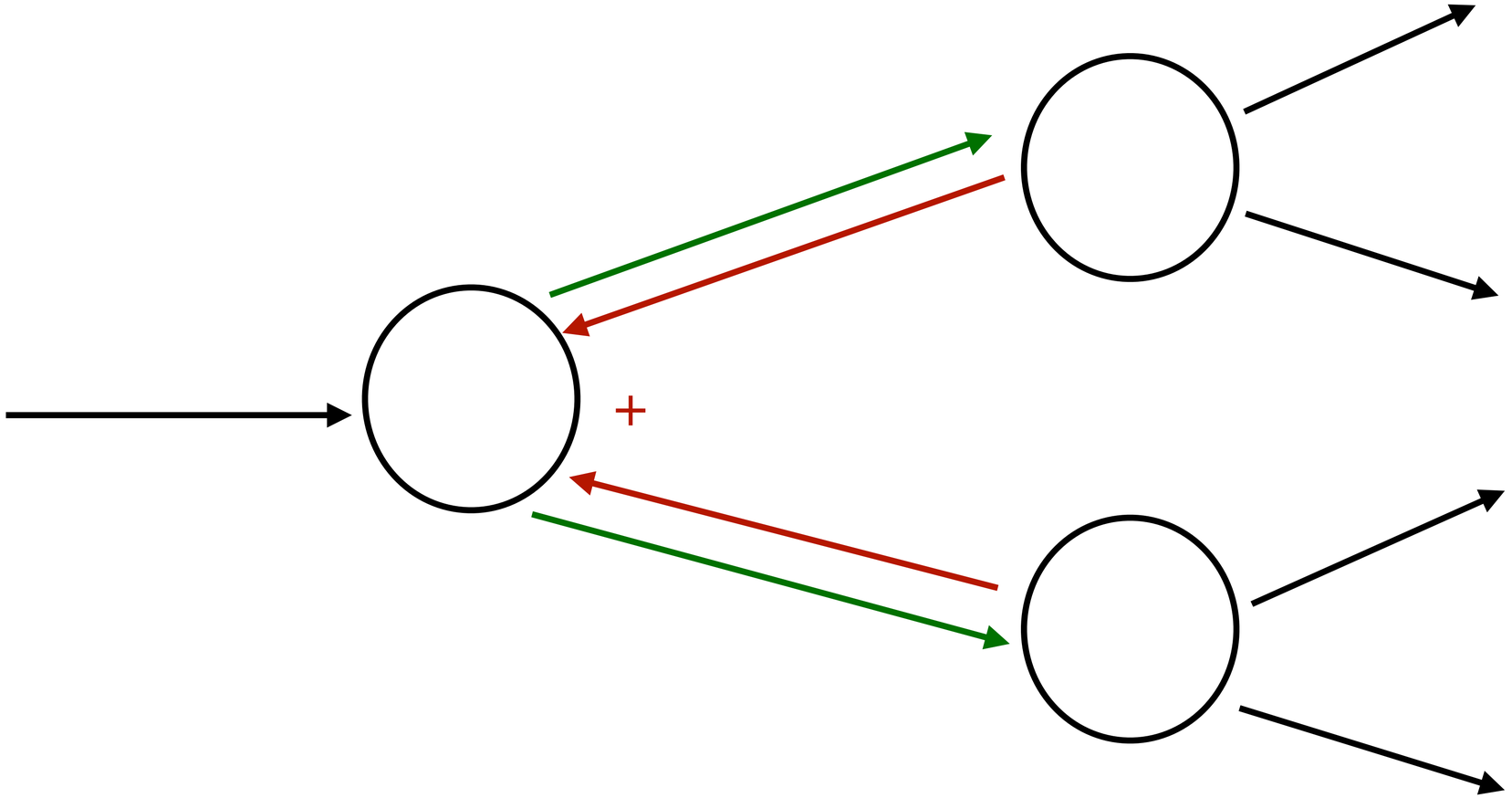
max gate: gradient router

mul gate: input switcher



Gradients add at branches

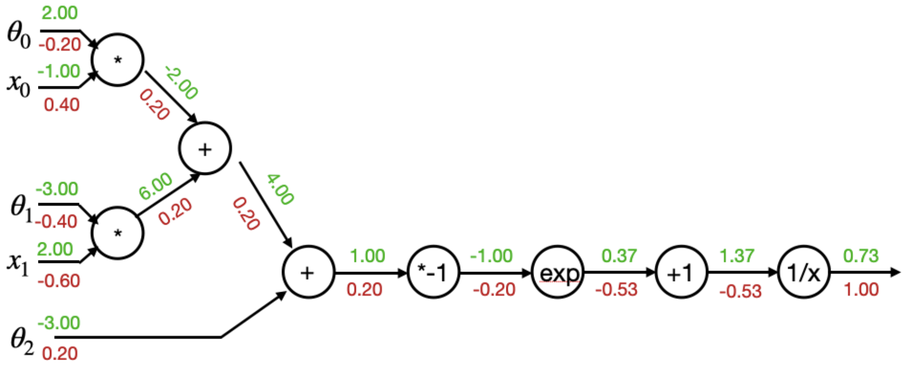
Multivariable chain rule



Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Modularized Implementation

Forward/backward API

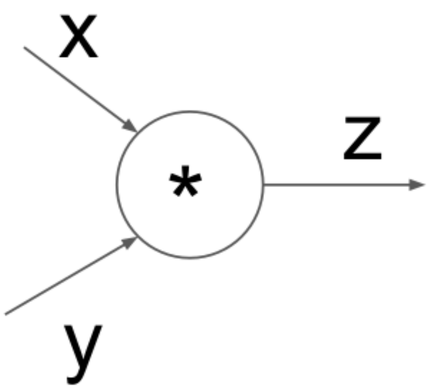


Graph (or Net) object *(rough psuedo code)*

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

Modularized Implementation

Forward/backward API



(x,y,z are scalars)

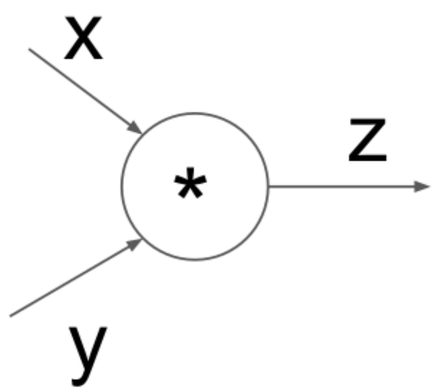
```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$\frac{\partial L}{\partial z}$

$\frac{\partial L}{\partial x}$

Modularized Implementation

Forward/backward API



(x,y,z are scalars)

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

Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Summary so far

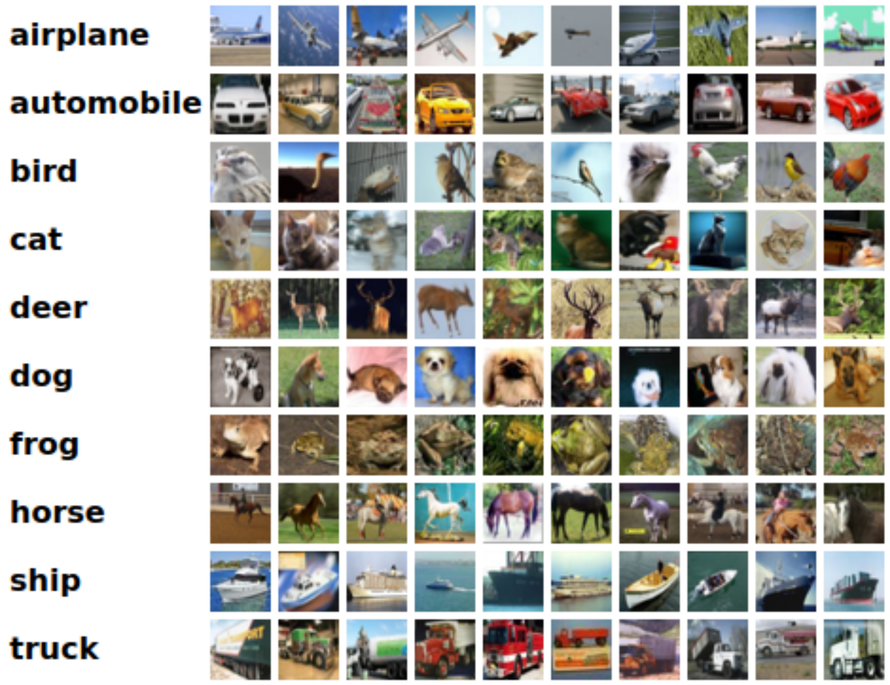
- Neural nets will be very large: impractical to write down gradient formula by hand for all parameters
- Back propagation: recursive application of chain rule along a computational graph to compute gradients of all variables (input / parameters)
- Implementations maintain a graph structure, where the nodes implement the forward( )/backward( ) API
- Forward: compute result of an operation and save intermediates for gradient computation in memory
- Backward: apply chain rule to compute the gradient of the loss function wrt parameters

Introduction to Image Classification: Logistic Regression on Images

CIFAR-10

Collection of images

Here are the classes in the dataset, as well as 10 random images from each:



10 classes

50000 training images

10000 test images

3 channel (RGB) images

Each image is **32x32x3**

Interlude: Nearest neighbor classification

An example using Manhattan/L1 distance

L1 distance $$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

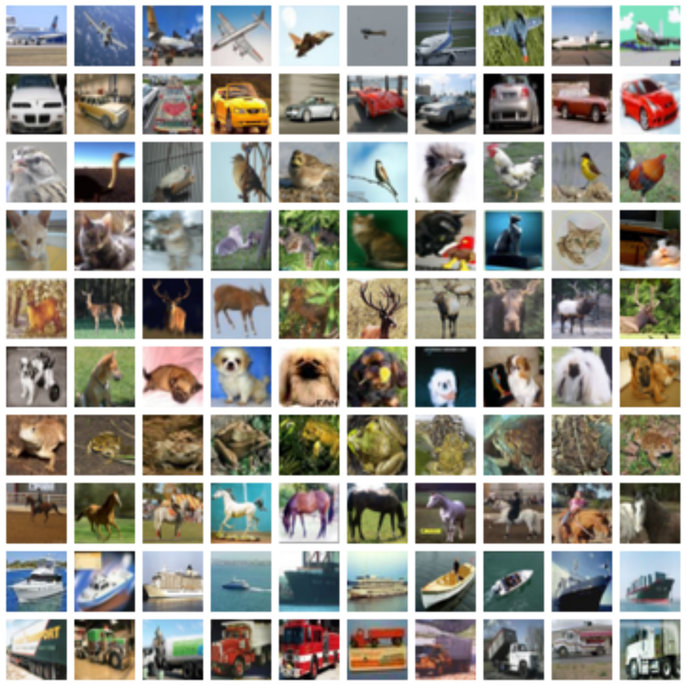
$$\left| \begin{array}{|c|c|c|c|} \hline 56 & 32 & 10 & 18 \\ \hline 90 & 23 & 128 & 133 \\ \hline 24 & 26 & 178 & 200 \\ \hline 2 & 0 & 255 & 220 \\ \hline \end{array} - \begin{array}{|c|c|c|c|} \hline 10 & 20 & 24 & 17 \\ \hline 8 & 10 & 89 & 100 \\ \hline 12 & 16 & 178 & 170 \\ \hline 4 & 32 & 233 & 112 \\ \hline \end{array} \right| = \begin{array}{|c|c|c|c|} \hline 46 & 12 & 14 & 1 \\ \hline 82 & 13 & 39 & 33 \\ \hline 12 & 10 & 0 & 30 \\ \hline 2 & 32 & 22 & 108 \\ \hline \end{array} \xrightarrow{\text{sum}} 456$$

Test image — Training image — pixel-wise absolute differences

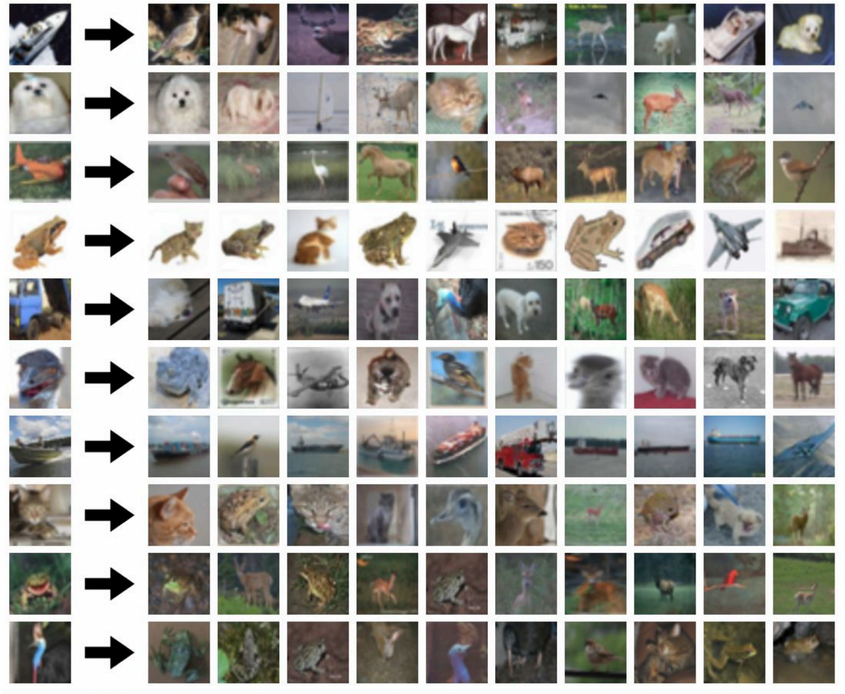
Interlude: Nearest neighbor classification

An example using Manhattan/L1 distance

Here are the classes in the dataset, as well as 10 random images from each:



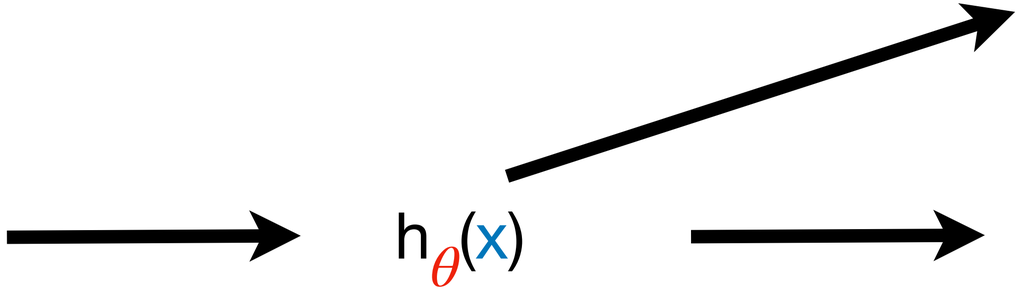
Test images and nearest neighbors:



Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Logistic Regression on Images

Input image



Array of 32x32x3
numbers
3072 values in total

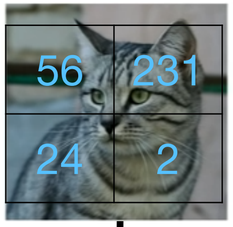
Model: $h_\theta(x) = \sigma(\theta x + b)$

$h_\theta(x)$: 10 x 1; θ: 10 x 3072; x: 3072 x 1; b: 10 x 1

An example image with 4 pixels

3 classes (cat/dog/ship)

Input image



θ

0.2	-0.5	0.1	2.0
1.5	1.3	2.1	0.0
0.0	0.25	0.2	-0.3

x

56
231
24
2

\+

b

1.1
3.2
-1.2

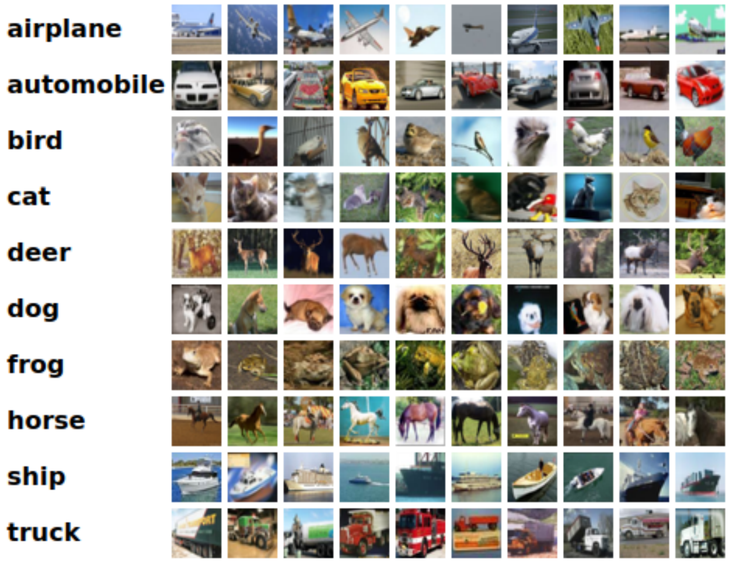
=

-96.8	Cat score
437.9	Dog score
61.95	Ship score

pixels into column vector

Interpreting Linear Classifier

Here are the classes in the dataset, as well as 10 random images from each:

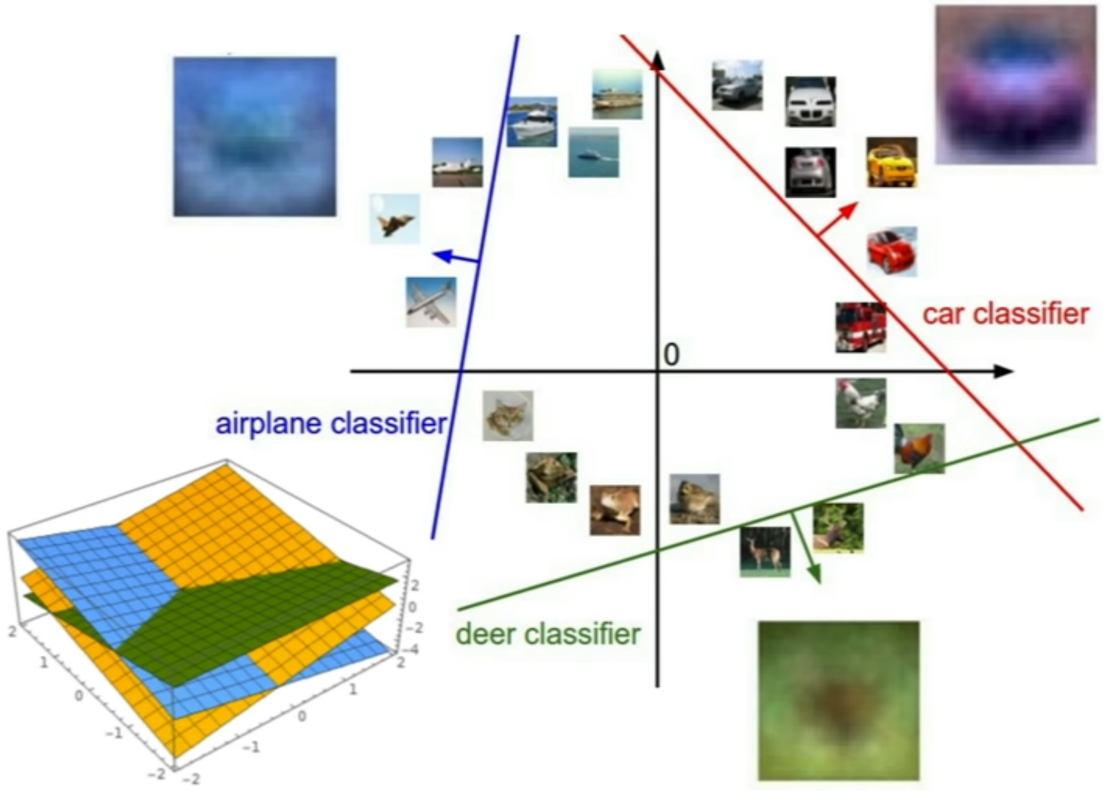


$$h_\theta(x) = \sigma(\theta x + b)$$

Weights of a linear classifier trained on CIFAR-10:

Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Interpreting Linear Classifier



$$h_\theta(x) = \sigma(\theta x + b)$$

One-vs-all classification

Loose end (for next week):

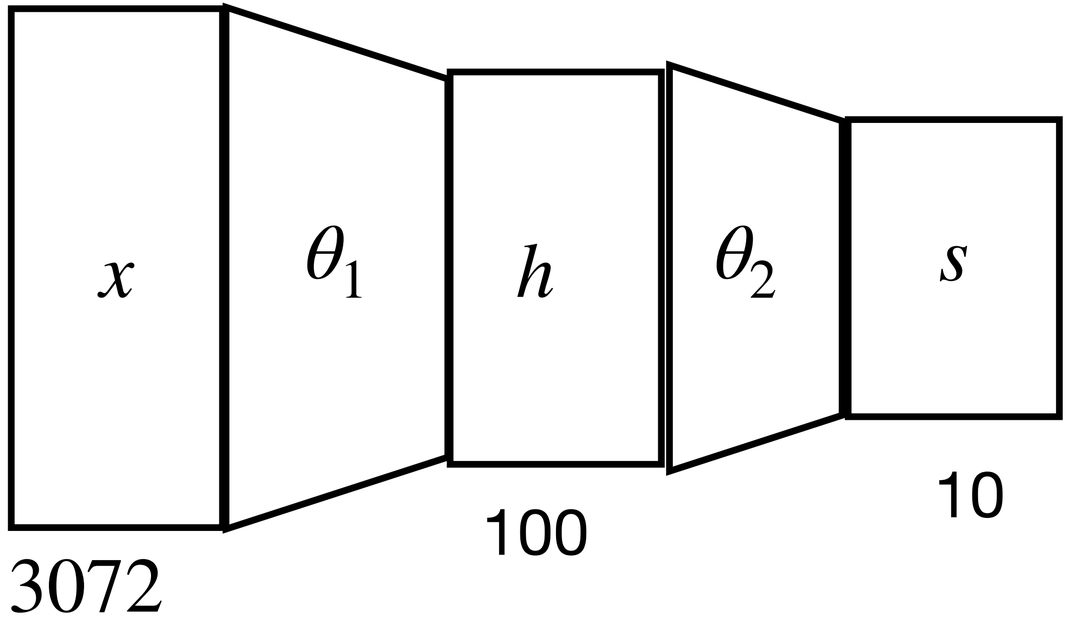
Loss for all classes at once

Neural networks

Linear function: $h = \theta x$

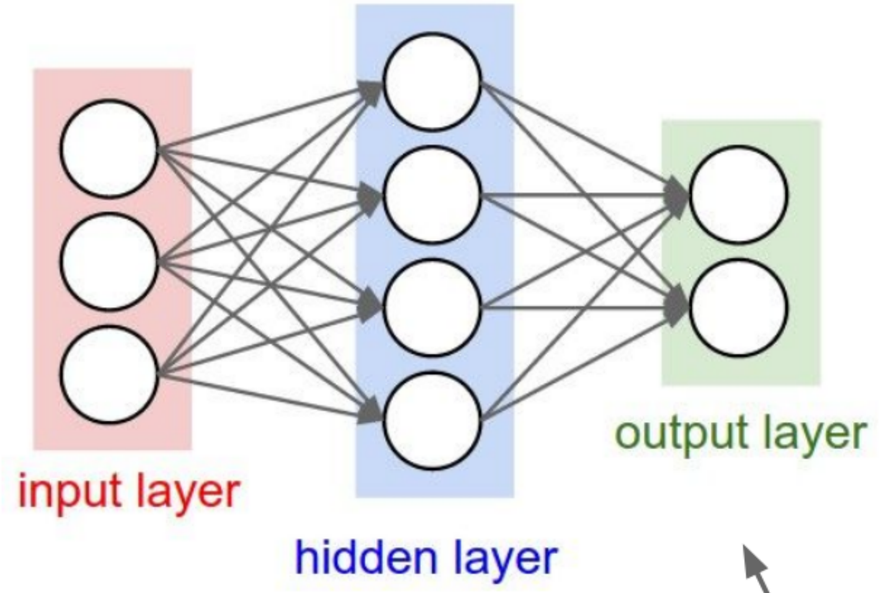
2-layer neural network: $h = \theta_2 g(\theta_1 x)$

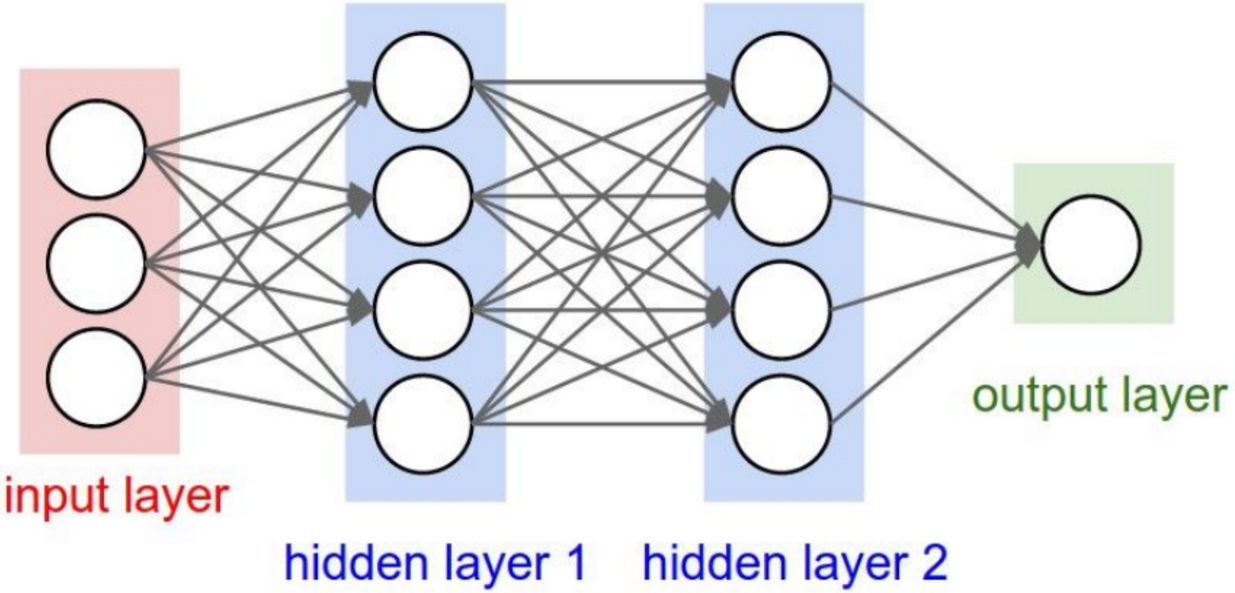
3-layer neural network: $h = \theta_3 g(\theta_2 g(\theta_1 x))$



Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Neural Network: Architectures





"2-layer Neural Net", or
"1-hidden-layer Neural Net"

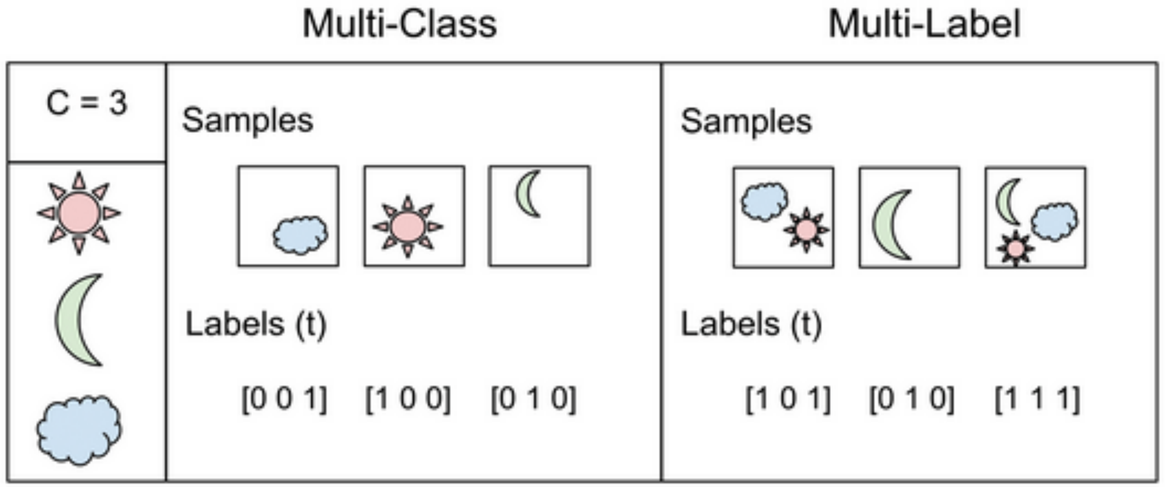
"3-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Loss- and activation functions for classification problems

Types of Classfication Tasks



- Multi-Class classification:

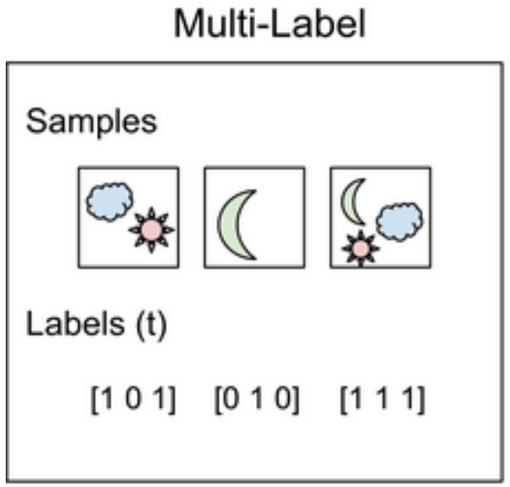
  Each image belongs to exactly one of C classes.
  This is a single classification problem.
  The ground truth (labels) can be one-hot encoded

- Multi-label classification

  Each image can belong to more than one class.
  This problem can be treated as C different binary classification problems

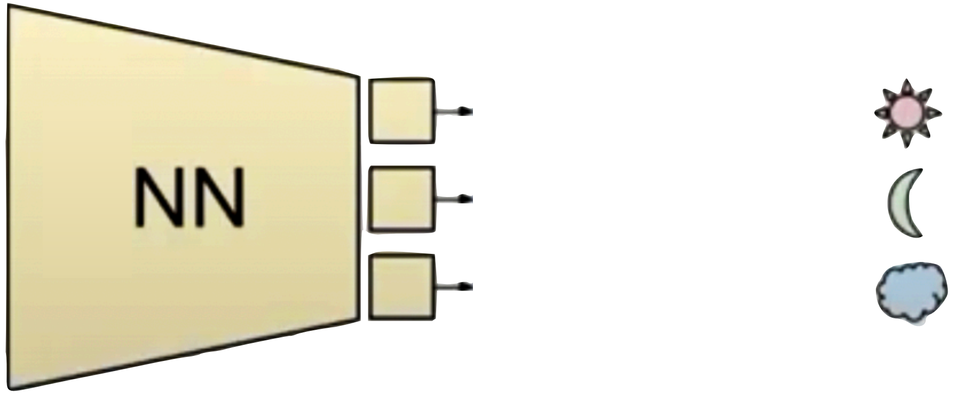
Credit to Raul Gomez (https://youtu.be/635cmrp4z40)

Multi-label classification



Let z_c denote the score obtained from the network for class c.
Sigmoid (aka logistic function):

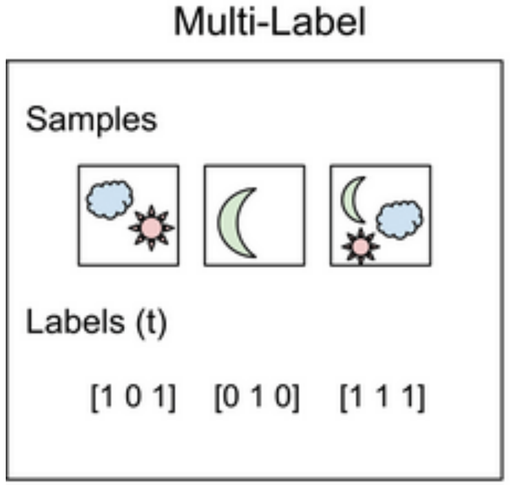
$$\sigma(z_c) = \frac{1}{1 + e^{-z_c}} =: p_c$$



Squishes each score into (0,1) range
—> can be interpreted as class probability
Only depends on a single score
—> appropriate for multi-label classification.

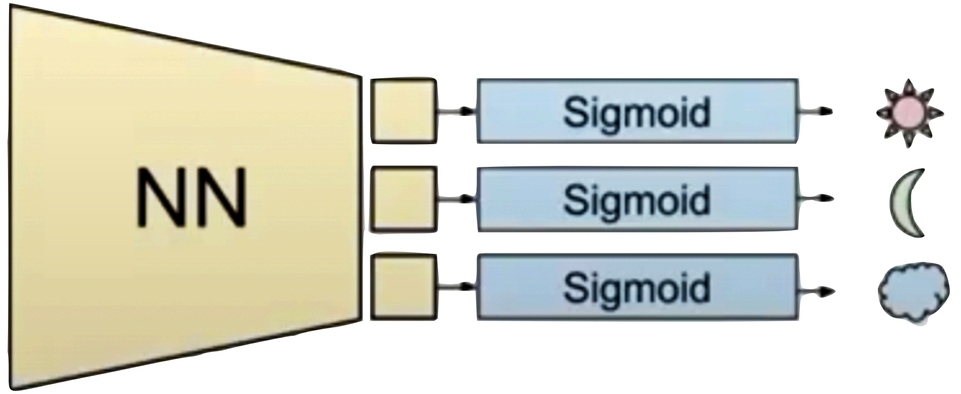
Credit to Raul Gomez (https://youtu.be/635cmrp4z40)

Multi-label classification



Let z_c denote the score obtained from the network for class c.
Sigmoid (aka logistic function):

$$\sigma(z_c) = \frac{1}{1 + e^{-z_c}} =: p_c$$



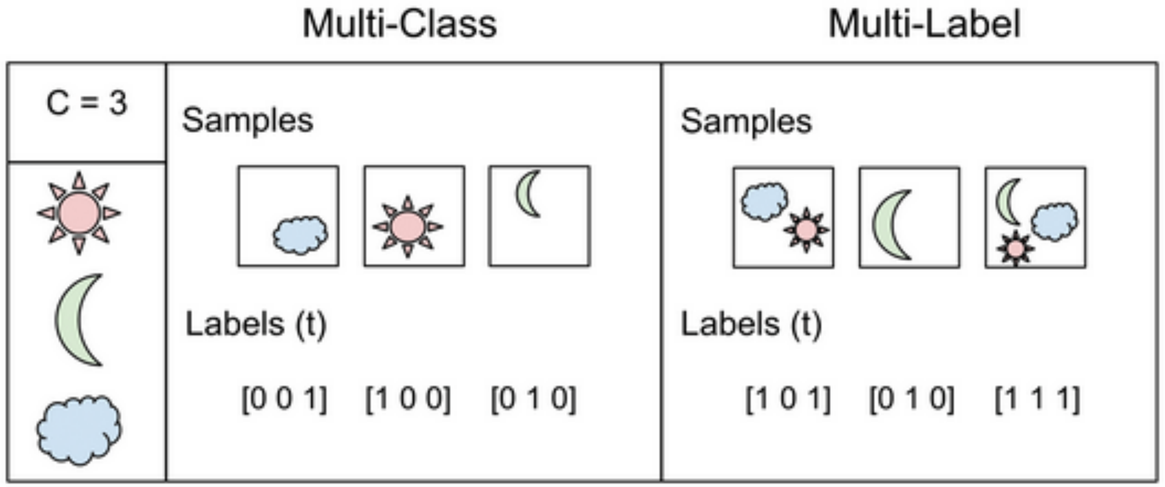
Binary cross-entropy loss per class $c \in C$:

$$L_c = -t_c log(p_c) - (1 - t_c) log(1 - p_c)$$

$$L = \sum_c L_c$$ aka **“sigmoid cross entropy loss”**

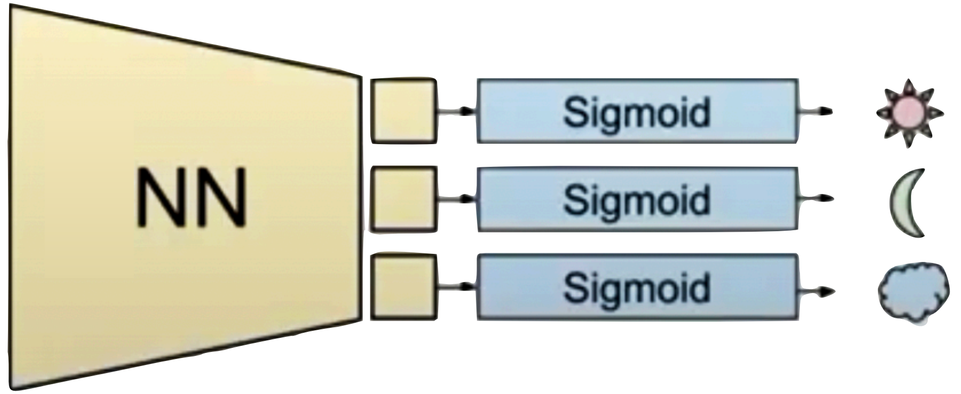
Credit to Raul Gomez (https://youtu.be/635cmrp4z40)

Multi-label classification



Let z_c denote the score obtained from the network for class c.
Sigmoid (aka logistic function):

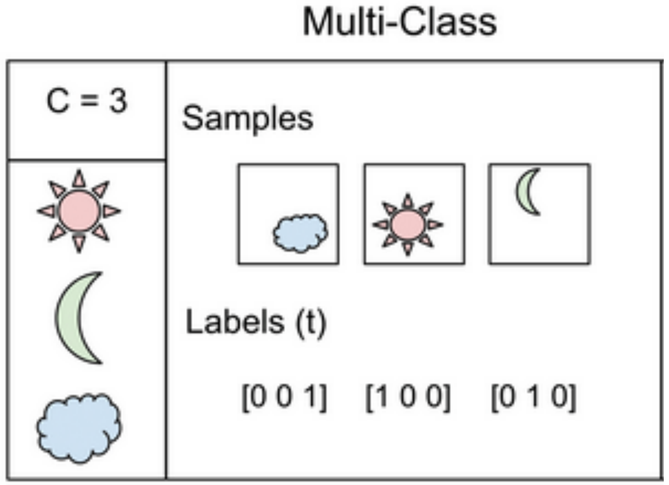
$$\sigma(z_c) = \frac{1}{1 + e^{-z_c}} =: p_c$$



Problem with sigmoid for multi-class classification problems: Mutual exclusivity of labels is not taken into account

Credit to Raul Gomez (https://youtu.be/635cmrp4z40)

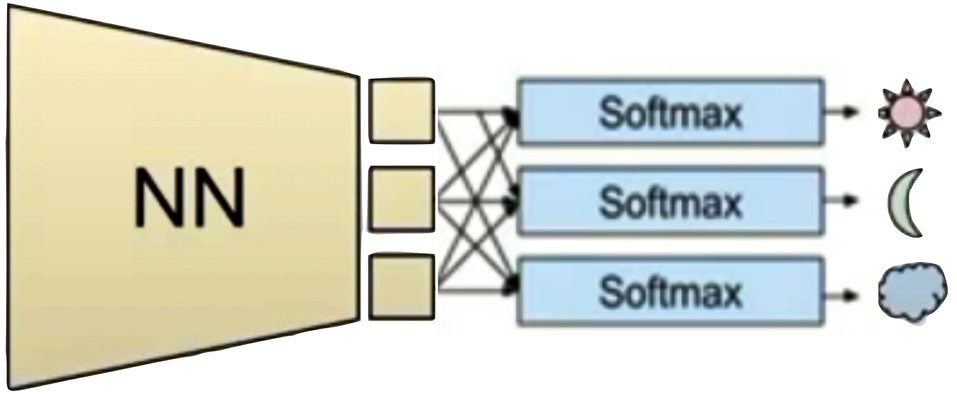
Multi-class classification



Let z_c denote the score obtained from the network for class c.

Softmax:

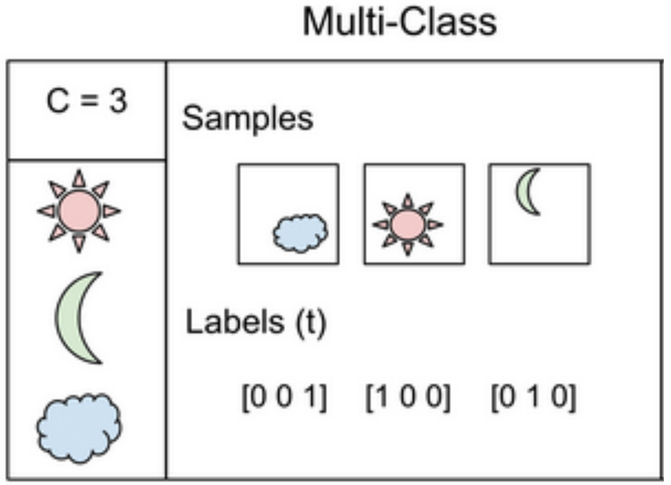
$$\sigma(z)_c = \frac{e^{z_c}}{\sum_{c'}^{C} e^{z_{c'}}} =: p_c$$



Squishes each score into (0,1) range
—> can be interpreted as class probability
Each output depends on all scores
—> appropriate for multi-class classification

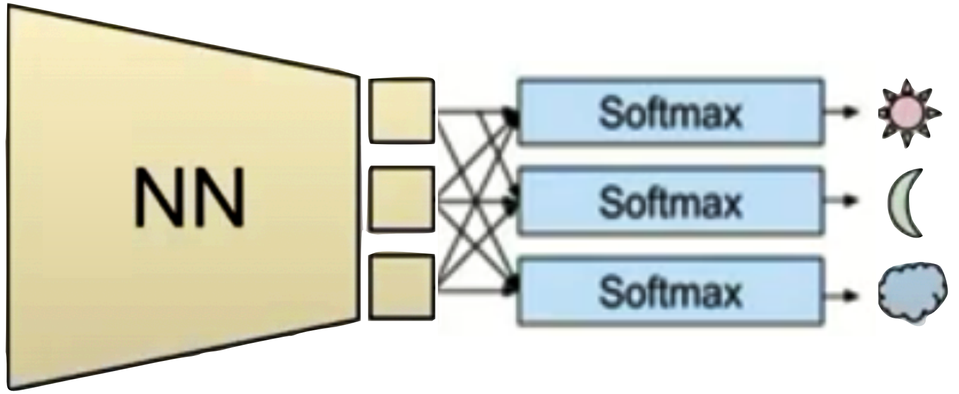
Credit to Raul Gomez (https://youtu.be/635cmrp4z40)

Multi-class classification



Let z_c denote the score obtained from the network for class c.
Softmax:

$$\sigma(z)_c = \frac{e^{z_c}}{\sum_{c'}^{C} e^{z_{c'}}} =: p_c$$

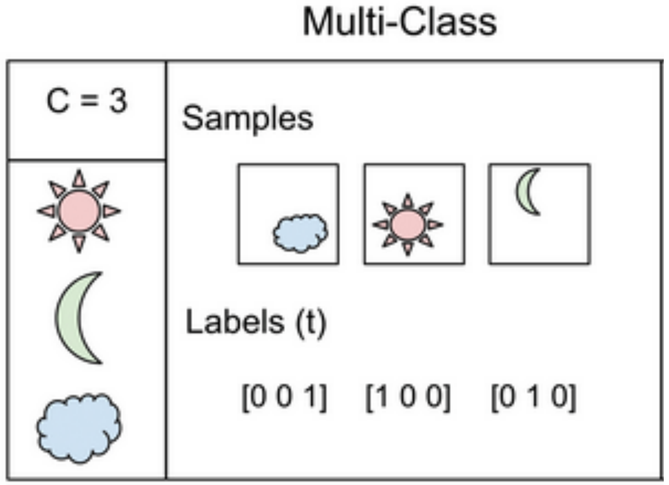


For two classes (i.e. binary classification), softmax is equivalent to sigmoid:

$$\sigma(z)_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2}} = \frac{1}{1 + e^{z_2 - z_1}} \overset{z_2 := 0}{=} \sigma(z_1)$$

Credit to Raul Gomez (https://youtu.be/635cmrp4z40)

Multi-class classification

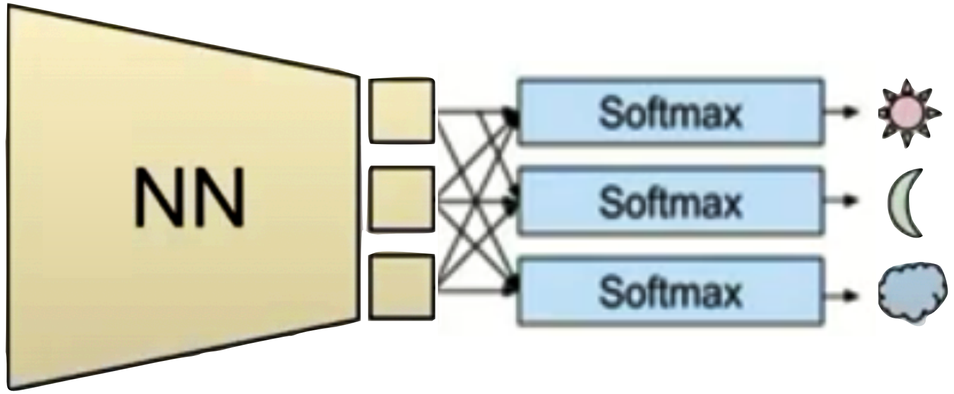


Let z_c denote the score obtained from the network for class c.

Softmax:

$$\sigma(z)_c = \frac{e^{z_c}}{\sum_{c'}^{C} e^{z_{c'}}} =: p_c$$

Backprop through softmax loss: Please try yourselves :)



Cross-entropy loss:

$$L = -\sum_{c}^{C} t_c log(p_c)$$

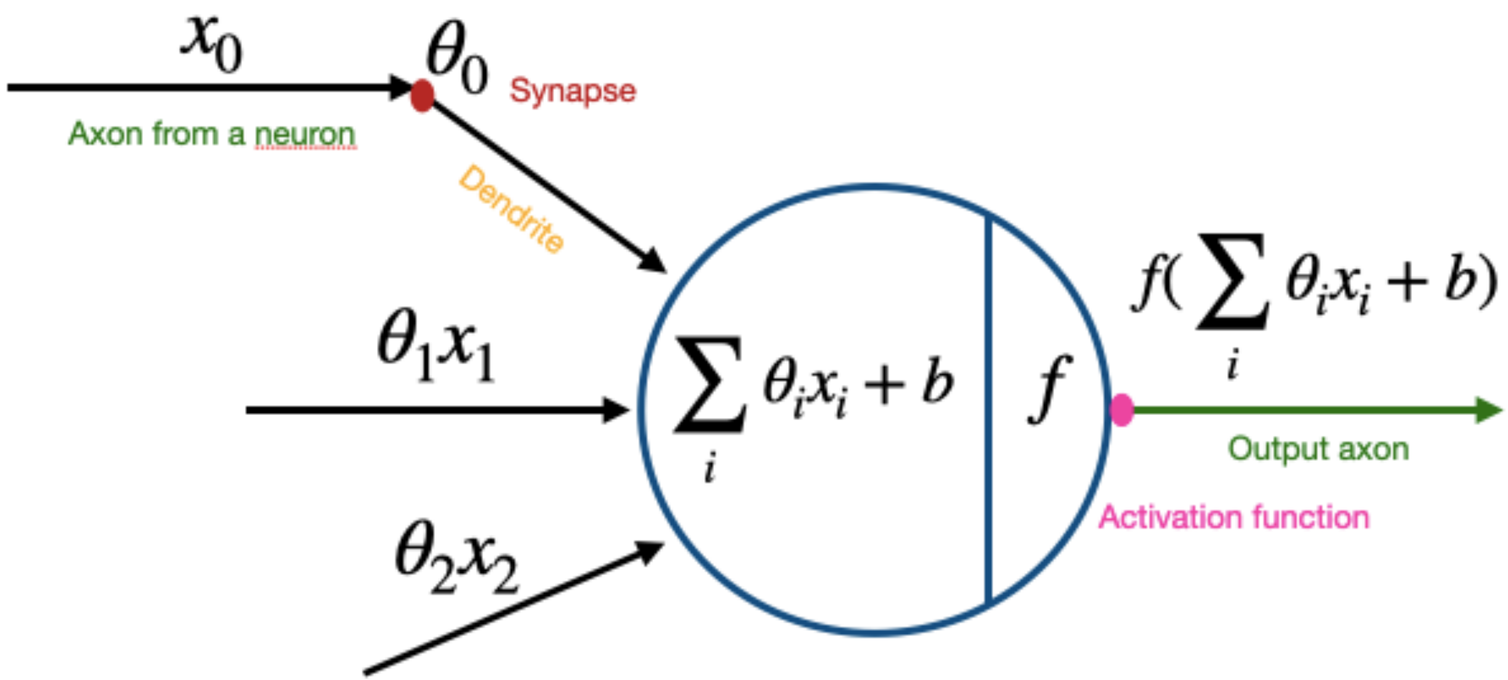
aka **“softmax cross entropy loss”** or **“softmax loss”**

(The binary cross-entropy loss is just the cross-entropy loss for two classes)

Credit to Raul Gomez (https://youtu.be/635cmrp4z40)

Activation Functions
(or: Better activation functions for intermediate layers)

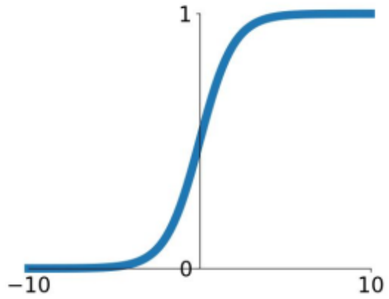
Activation Functions



Activation Functions

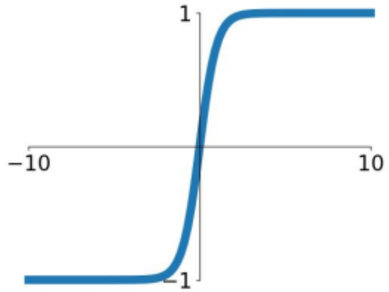
Sigmoid

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

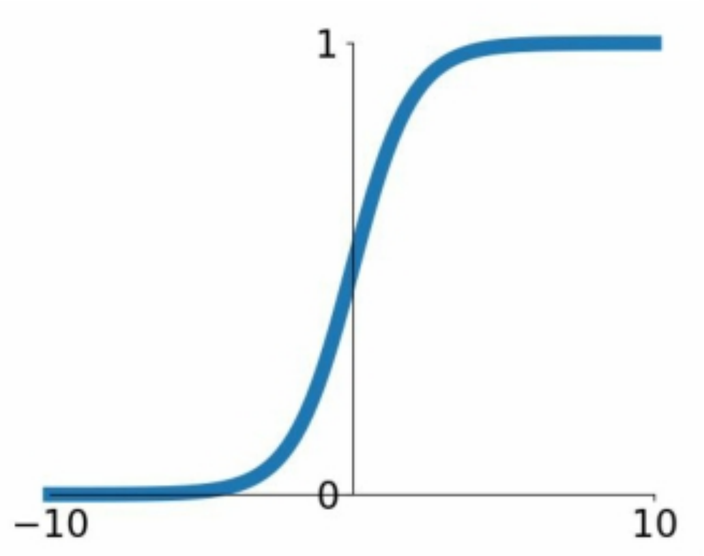


tanh

$$\tanh(x)$$



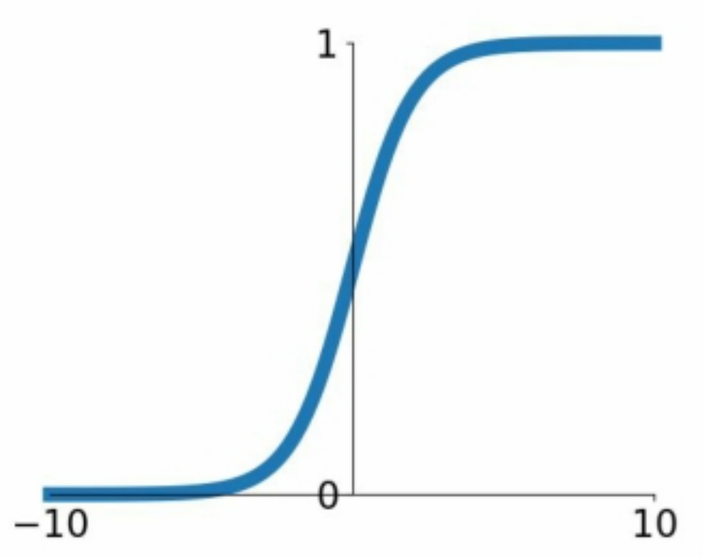
Activation Functions: Sigmoid

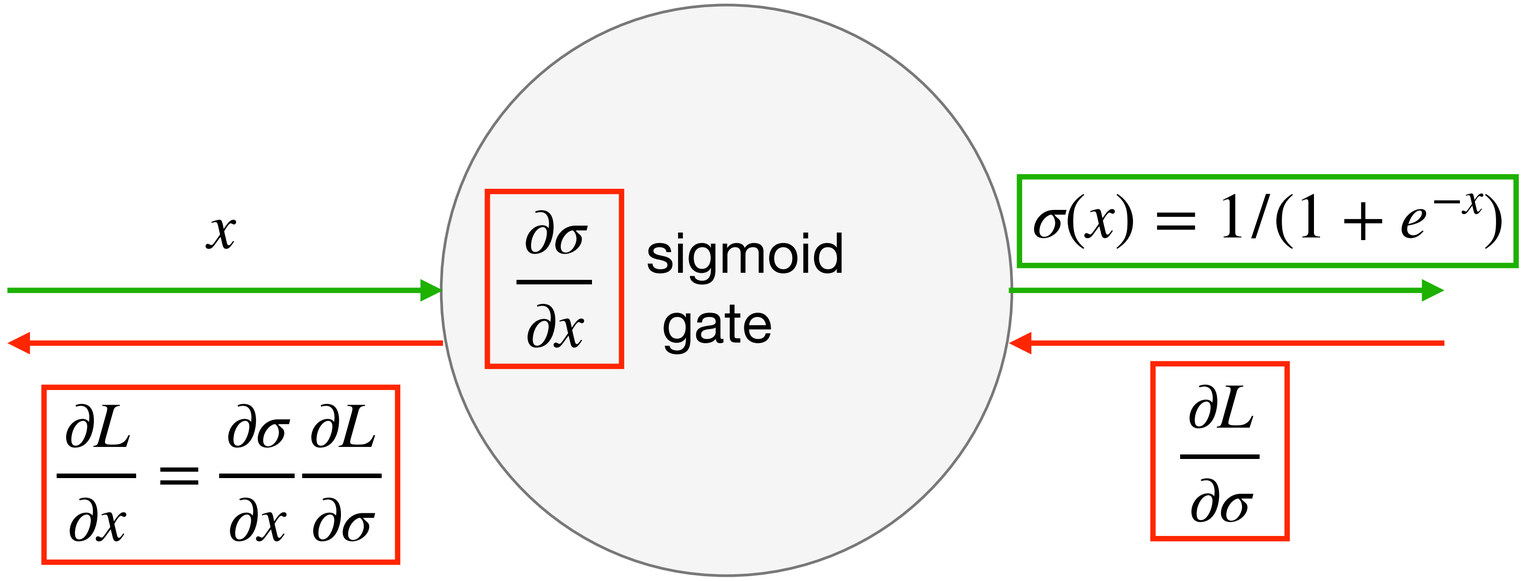


$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squishes inputs to range (0,1)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Sigmoid Problem 1



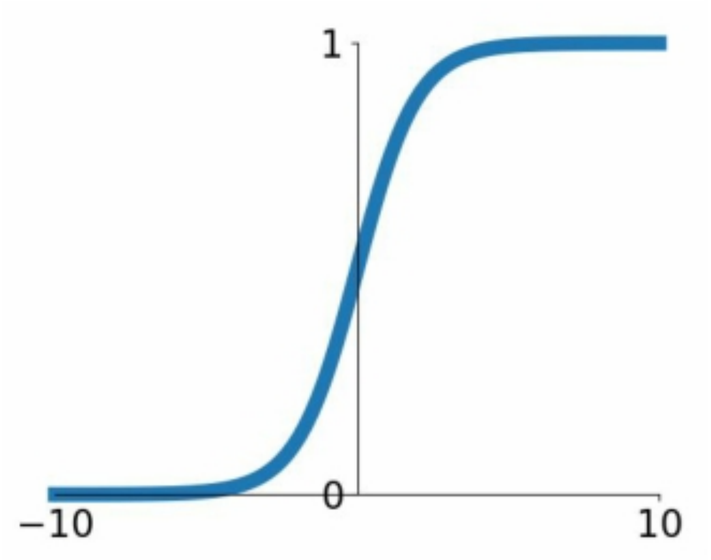


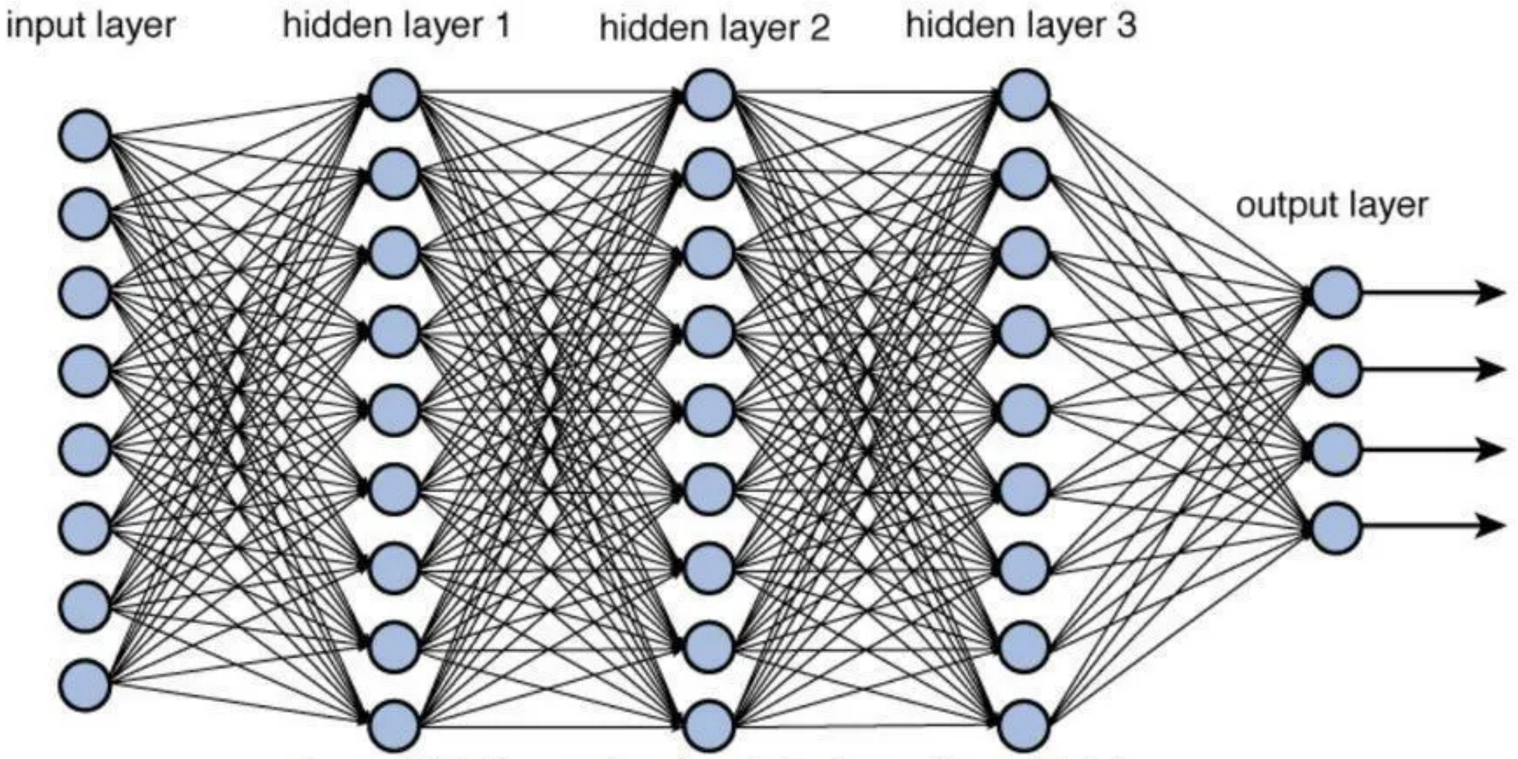
What happens when x = -10?

What happens when x = 0?

What happens when x = 10?

Sigmoid Problem 1



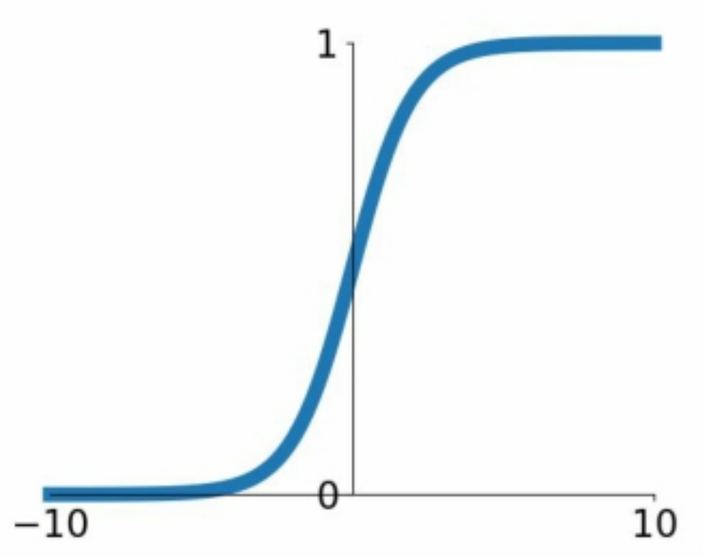


Backprop: Global gradient is multiplied with local gradient in each layer

E.g. best case for 10 layers: $0.25^{10} = 0.000001$

Local gradient $\in (0, 0.25]$

Activation Functions: Sigmoid



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

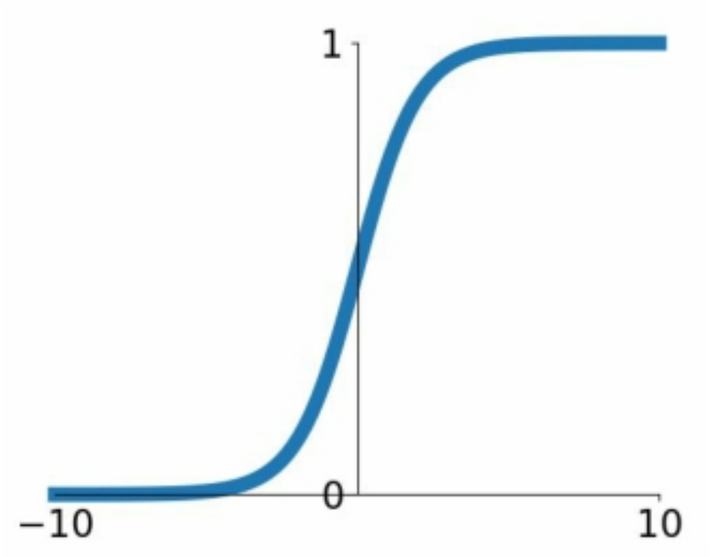
- Squishes inputs to range (0,1)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

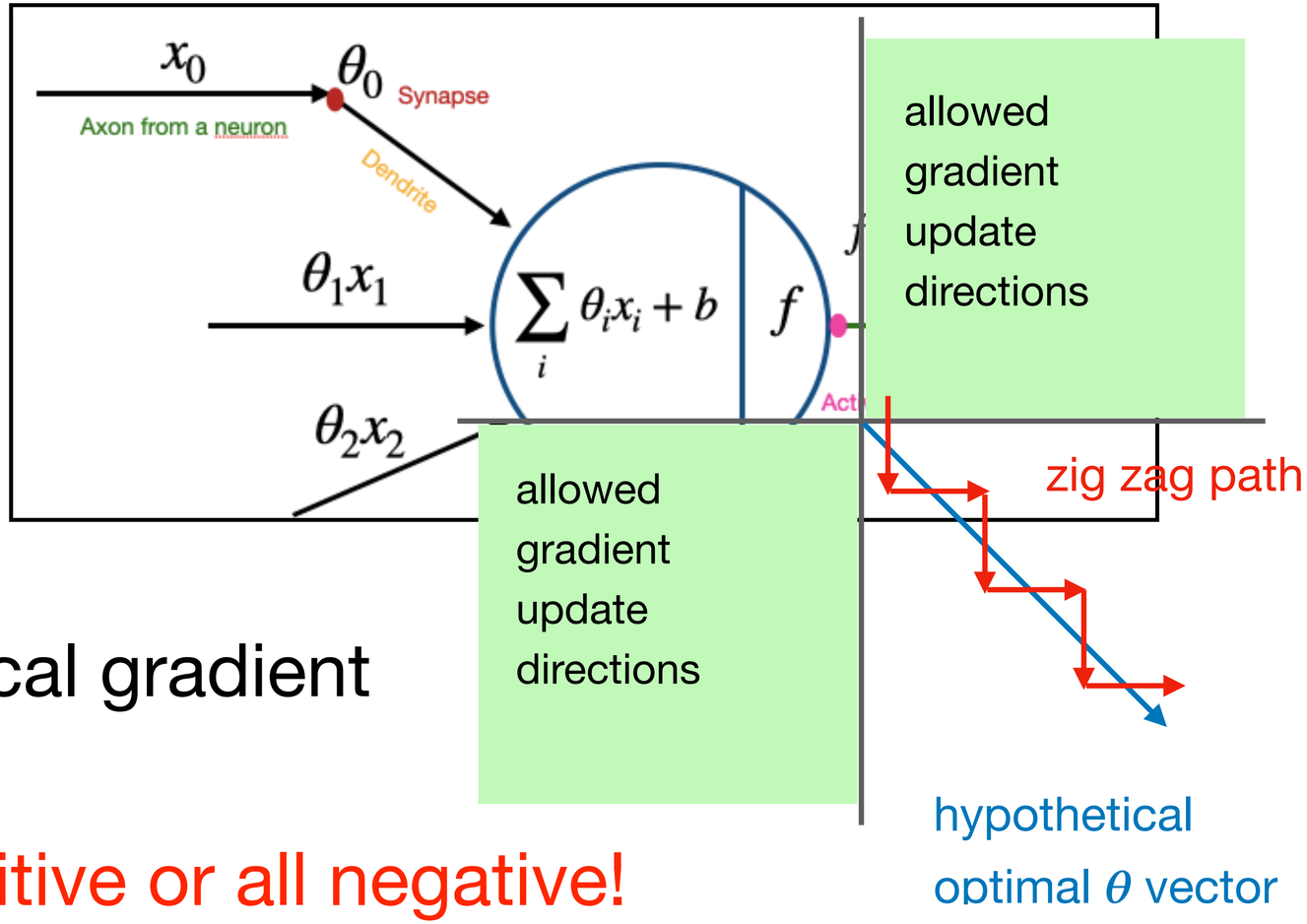
Problems:

1. Saturated neurons “kill” the gradients

Sigmoid Problem 2

What happens if the input x to a neuron is always positive?

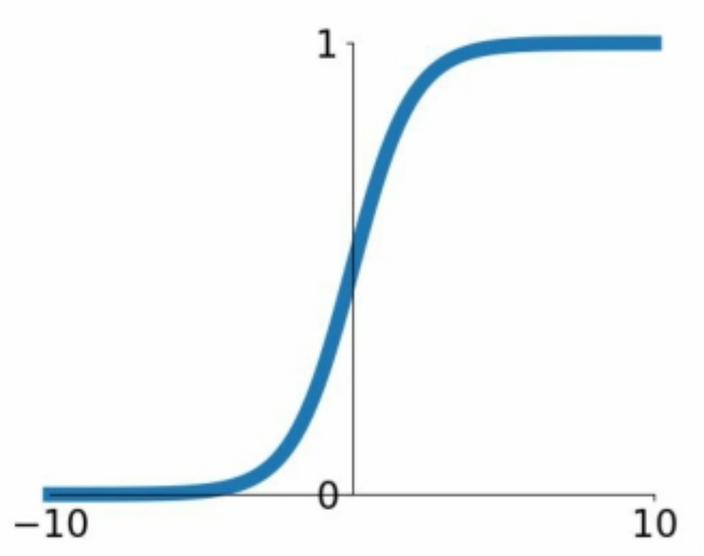




What can we say about the local gradient of the linear function w.r.t. θ ?

Global gradient always all positive or all negative!

Activation Functions: Sigmoid



Sigmoid

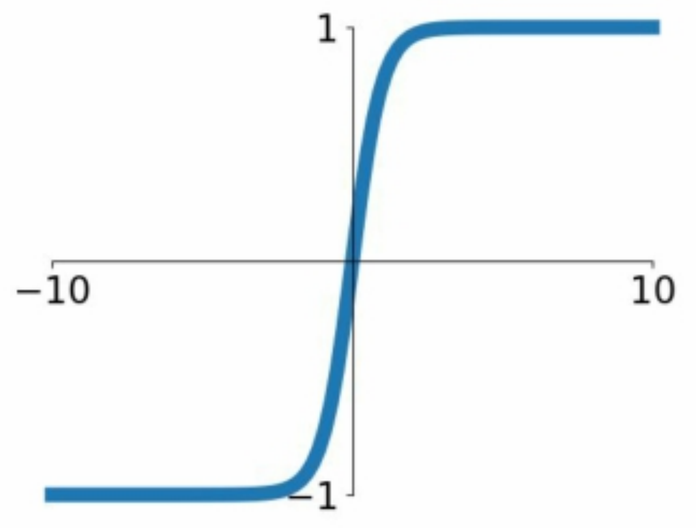
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 Problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Functions: tanh



tanh(x)

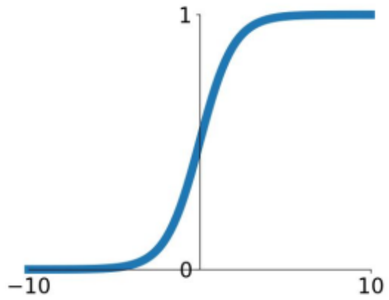
- Squishes numbers to range (-1,1)
- zero-centered
- still kills gradients when saturated

[LeCun et al., 1991]

Activation Functions

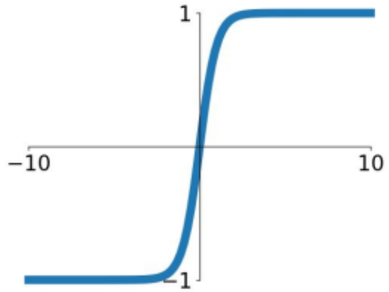
Sigmoid

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



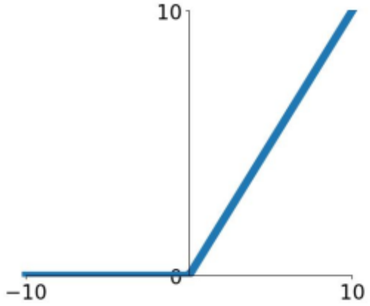
tanh

$$\tanh(x)$$



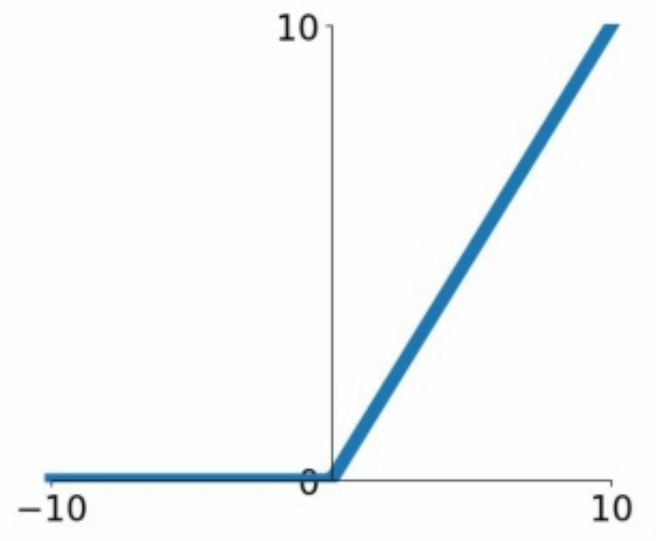
ReLU

$$\max(0, x)$$



Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Activation Functions: ReLU

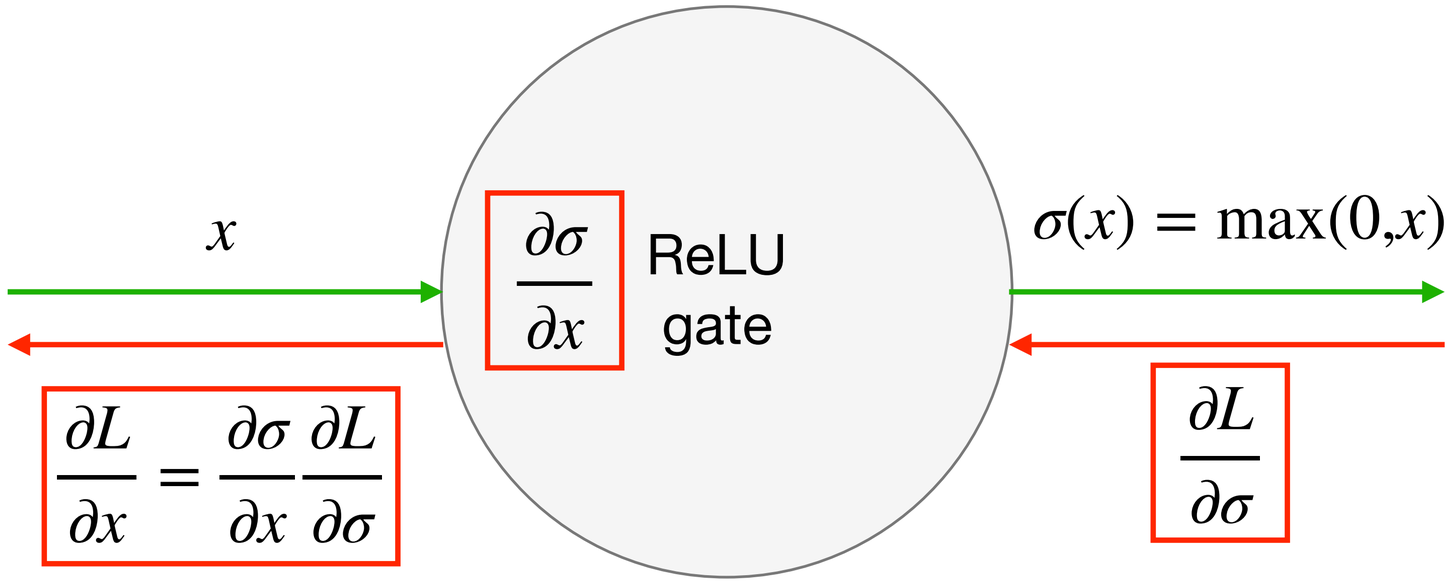


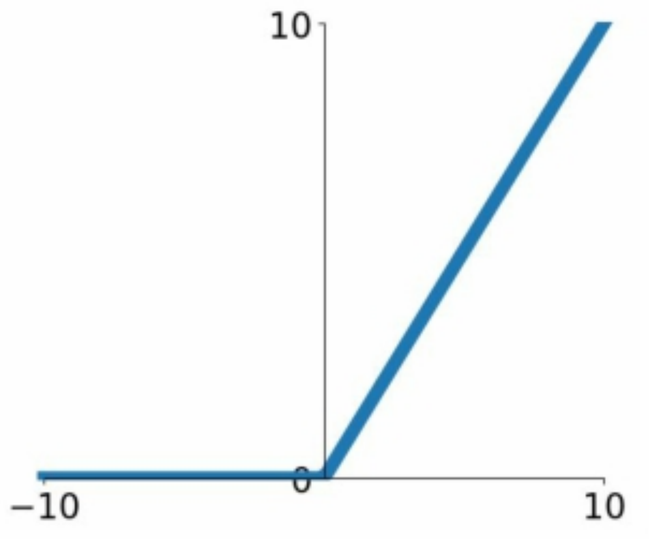
ReLU

(Rectified Linear Unit)

- Computes $f(x) = \max(0,x)$
- Does not saturate (in + region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- (more “biologically plausible” than sigmoid)

[Krizhevsky et al., 2012]



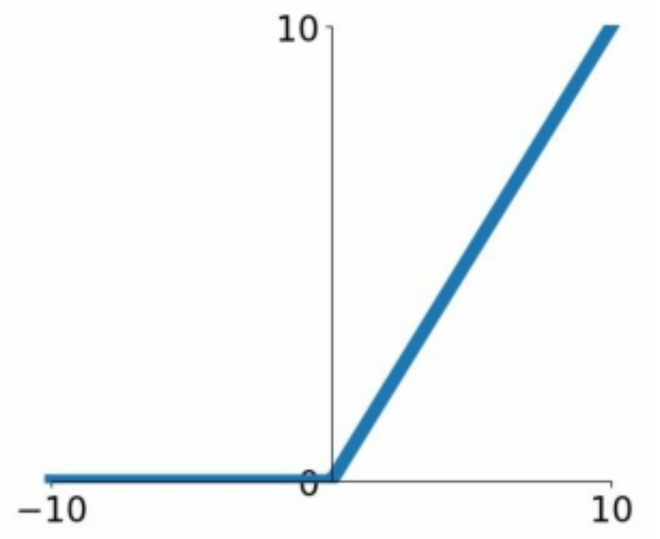


What happens when x = -10?

What happens when x = 0?

What happens when x = 10?

Activation Functions: ReLU



ReLU

(Rectified Linear Unit)

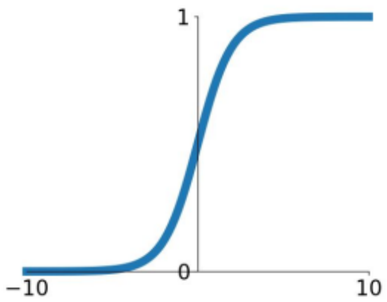
- Computes $f(x) = \max(0,x)$
- Does not saturate (in + region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- (more “biologically plausible” than sigmoid)

- Not zero-centered
- Gradient =0 when x < 0

Activation Functions

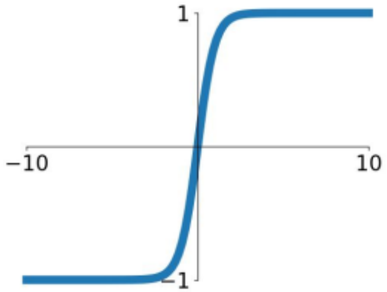
Sigmoid

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



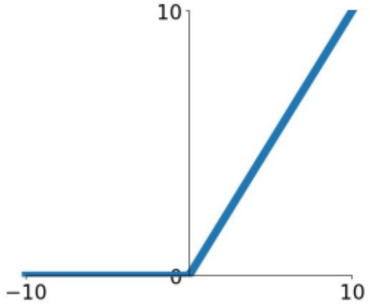
tanh

$$\tanh(x)$$



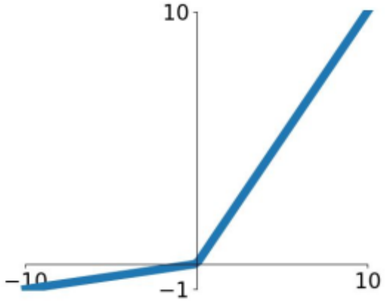
ReLU

$$\max(0, x)$$



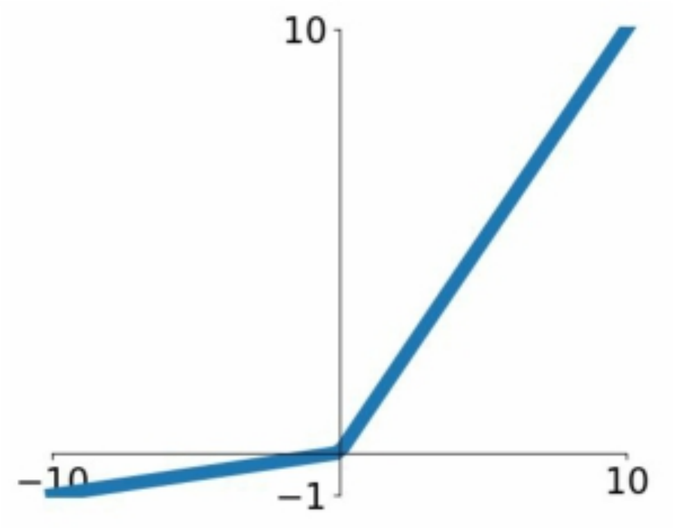
Leaky ReLU

$$\max(0.1x, x)$$



Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Activation Functions: Leaky ReLU



- Does not saturate
- Computationally efficient
- Closer to zero-mean outputs
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- **will not “die”**

Leaky ReLU

$$f(x) = \max(0.1x, x)$$

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into α (parameter)

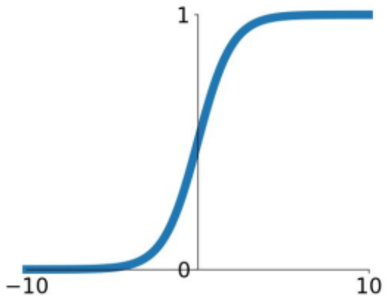
[Mass et al., 2013]

[He et al., 2015]

Activation Functions

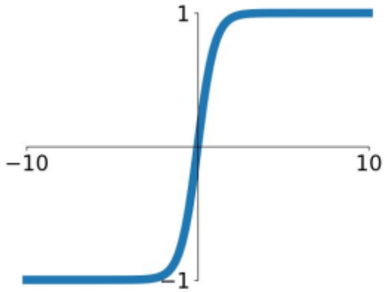
Sigmoid

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



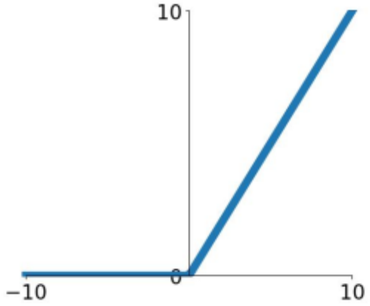
tanh

$$\tanh(x)$$



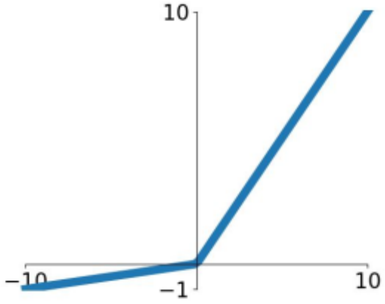
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Activation Functions: Maxout “Neuron”

- Does not have the basic form of dot product ➡ nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

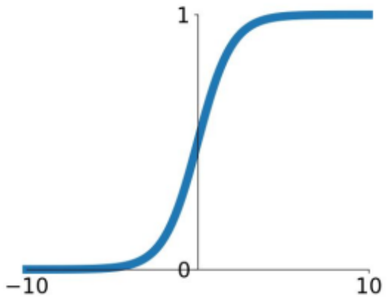
$$\max(\theta_1^T x + b_1, \theta_2^T x + b_2)$$

- Problem: doubles the number of parameters per neuron

Activation Functions

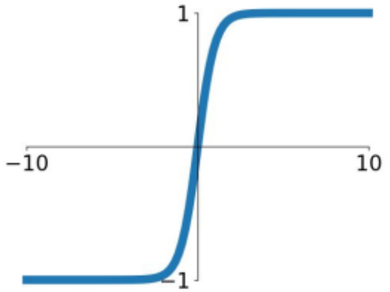
Sigmoid

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



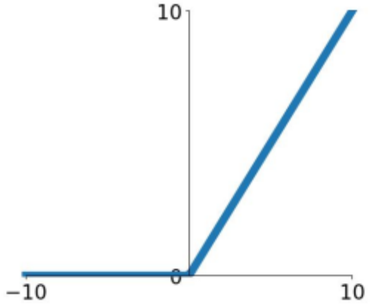
tanh

$$\tanh(x)$$



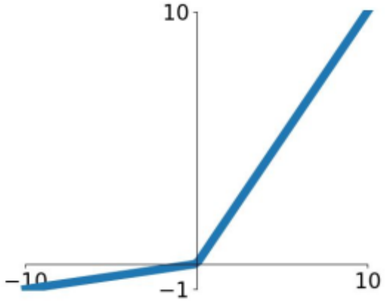
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

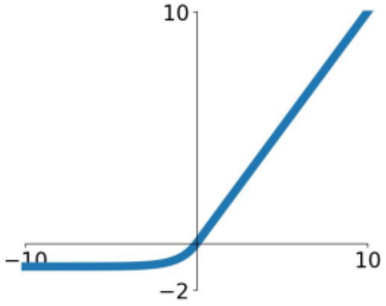


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

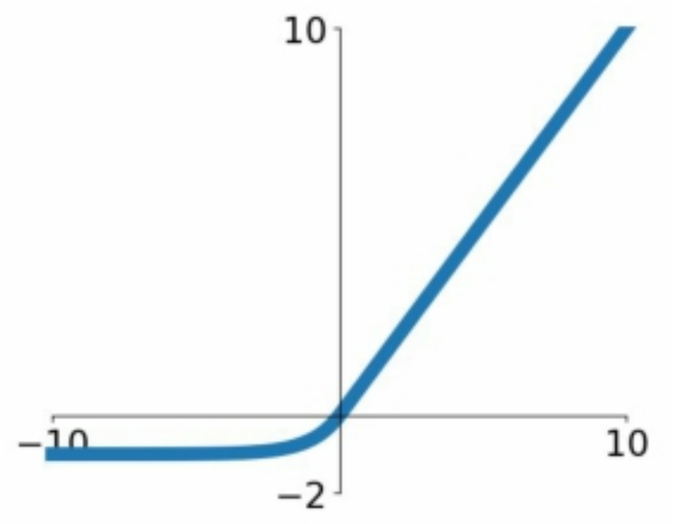
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Activation Functions: ELU



- All benefits of ReLU
- Closer to zero-mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

- Computation requires exp()

Exponential Linear Units (ELU)

$$f(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

[Clevert et al., 2015]

Activation Functions

TLDR: In practice:

- Use ReLU.
- Try out Leaky ReLU / Maxout / ELU
- Don’t expect much from tanh or sigmoid

One more: GELU

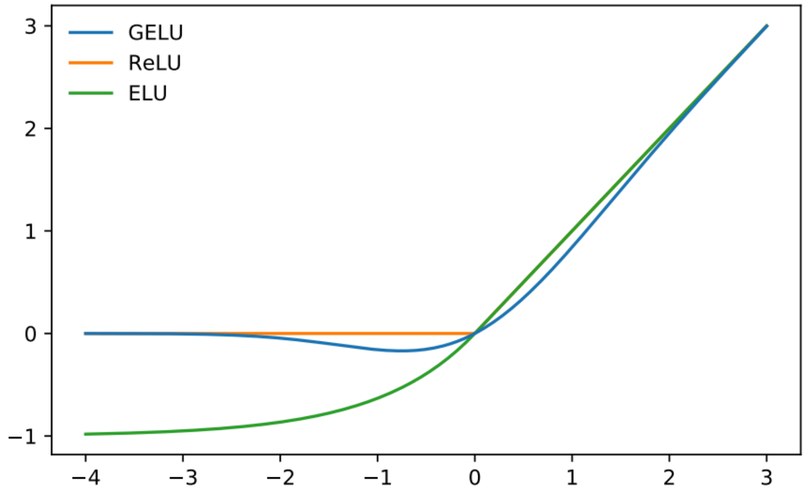


Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

[Hendrycks et al., 2016]

Neural Networks

- Perceptrons
- Multi-layer Perceptrons
- Backpropagation
- … on Images :)
- Loss- and activation functions for more than two classes
- Better activation functions
- Better gradient descent

Stochastic Gradient Descent

Recap

Linear Regression with Gradient Descent

Hypothesis = $h_\theta = \sum_{j=0}^{n} \theta_j x_j$

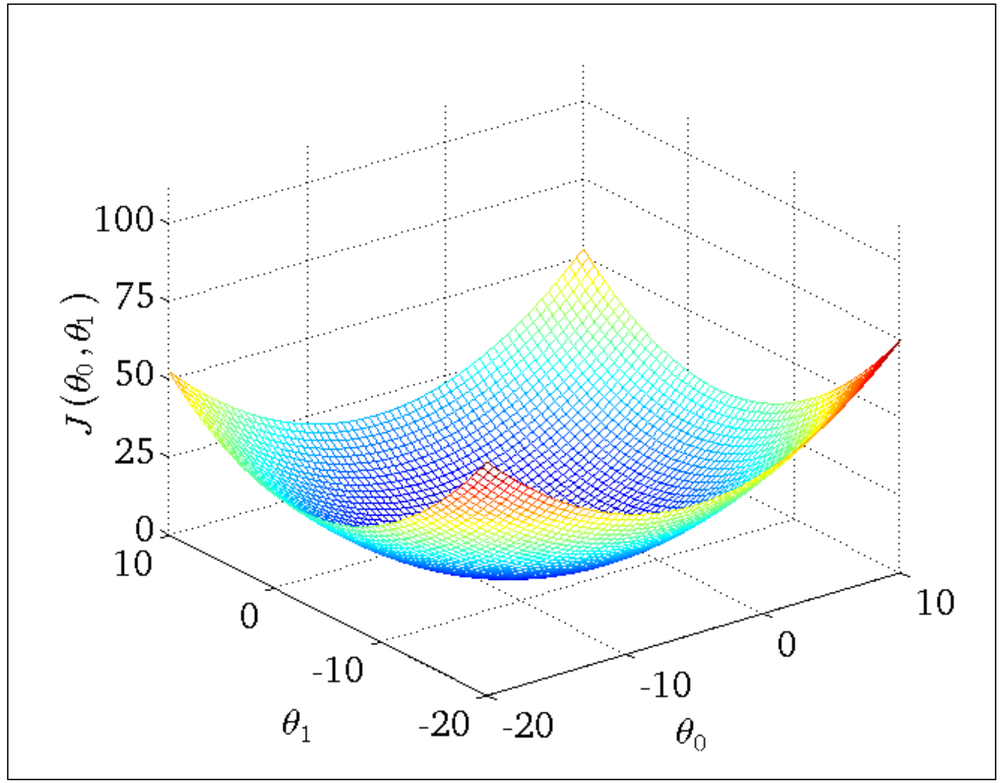
Cost function = $J(\theta) = \frac{1}{2m} \sum_{i=q}^{m} (h_\theta(x^{(i)} - y^{(i)})^2$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

for every $j = 0,1,....,n$

}



Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Recap

Linear Regression with Gradient Descent

Hypothesis = $h_\theta = \sum_{j=0}^{n} \theta_j x_j$

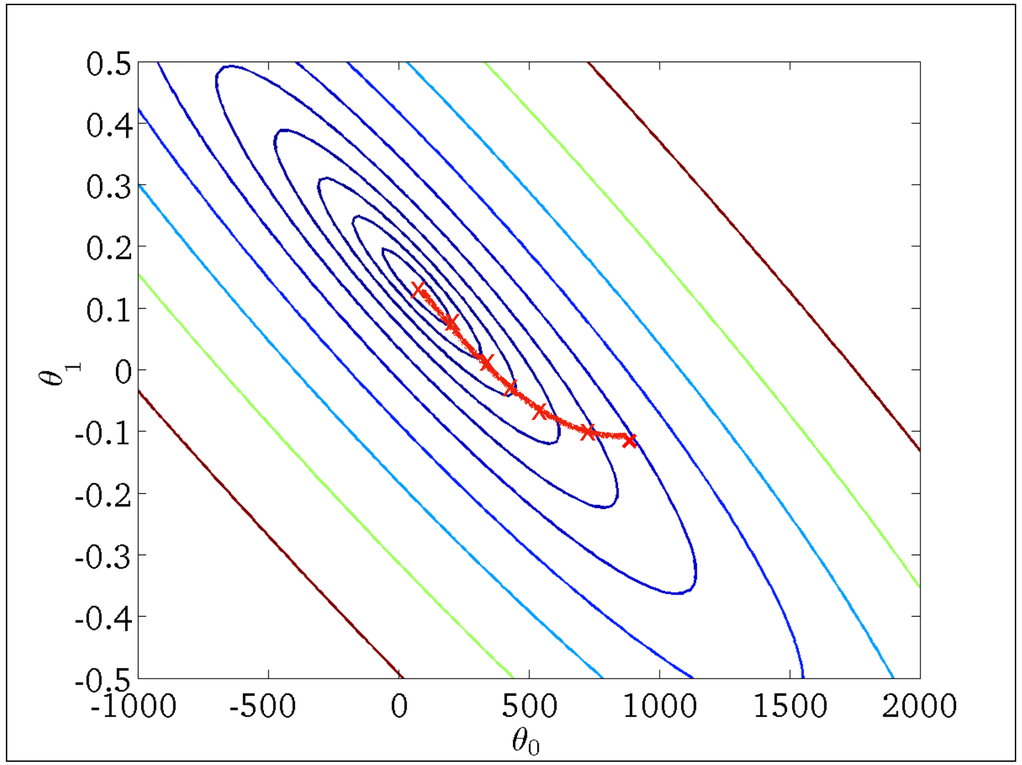
Cost function = $J(\theta) = \frac{1}{2m} \sum_{i=q}^{m} (h_\theta(x^{(i)} - y^{(i)})^2$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

for every $j = 0,1,....,n$

}



Batch gradient descent:
For large datasets, computation is expensive

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Batch- vs. stochastic gradient descent

Batch gradient descent

$$J(\theta) = \frac{1}{2m}\sum_{i=q}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$$

Repeat {

$$\theta_j := \theta_j - \alpha\boxed{\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}}$$

$$j = 0,1,.....,n$$

$$= \frac{\partial}{\partial\theta_j}J(\theta)$$

Stochastic gradient descent

$$cost(\theta,(x^{(i)},y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}cost(\theta,x^{(i)},y^{(i)}))$$

1. Randomly shuffle dataset
2. Repeat {
   for i = 1,...,m {

$$\theta_j := \theta_j - \alpha\boxed{(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}}$$

   for j =0,…n

   } }

$$= \frac{\partial}{\partial\theta_j}cost(\theta,(x^{(i)},y^{(i)}))$$

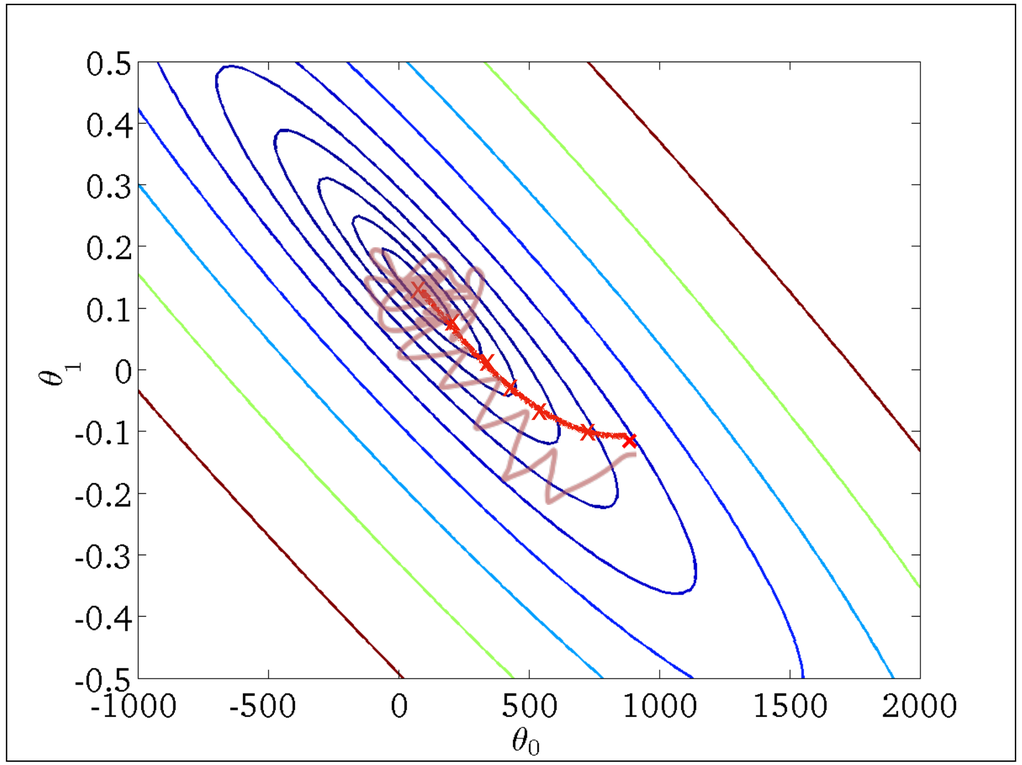
Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Stochastic gradient descent

1. Randomly shuffle dataset
2. Repeat { **= Epoch**
   for i = 1,...,m {

$$\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

   for j =0,…,n
   }
   }



Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- When the training set size m is very large, SGD can be much faster than gradient descent
- The cost function $J_{train}(\theta)$ should go down with every iteration of batch gradient descent (assuming appropriate α) but not necessarily with SGD
- SGD is applicable only to linear regression but not to other models (like logistic regression or neural networks)
- Before beginning the main loop of SGD, it is a good idea to “shuffle” your training data into a random order.

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- When the training set size m is very large, SGD can be much faster than gradient descent
- The cost function $J_{train}(\theta)$ should go down with every iteration of batch gradient descent (assuming appropriate α) but not necessarily with SGD
- SGD is applicable only to linear regression but not to other models (like logistic regression or neural networks)
- Before beginning the main loop of SGD, it is a good idea to “shuffle” your training data into a random order.

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Mini-batch Gradient Descent

Batch gradient descent: Use all *m* examples in each iteration
Stochastic gradient descent: Use 1 example in each iteration
Mini-batch gradient descent: Use b examples in each iteration

b = mini-batch size e.g., b= 10

$$\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{i=k \cdot b}^{(k+1) \cdot b - 1} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

vectorized computation, parallelize

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

- Suppose you use mini-batch gradient descent on a training set of size m, and you use a mini-batch size of B. The algorithm becomes the same as batch gradient descent if:

  - B = 1
  - B = m/2
  - B = m
  - NOTA

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

- Suppose you use mini-batch gradient descent on a training set of size m, and you use a mini-batch size of B. The algorithm becomes the same as batch gradient descent if:

  - B = 1
  - B = m/2
  - B = m
  - NOTA

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Stochastic gradient descent convergence

How to set the learning rate?

Batch gradient descent:

Plot loss as a function of the number of gradient steps

Stochastic gradient descent:

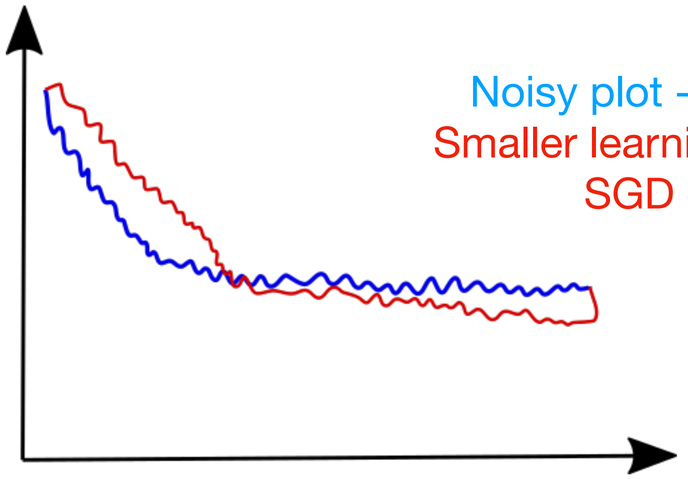
Compute loss contribution of current sample in each gradient step

Every 100 iterations, plot losses averaged over the last 100 samples

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

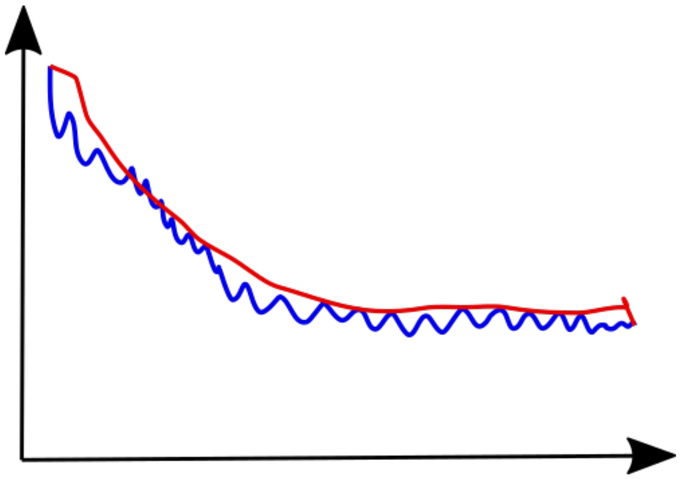
Checking for convergence

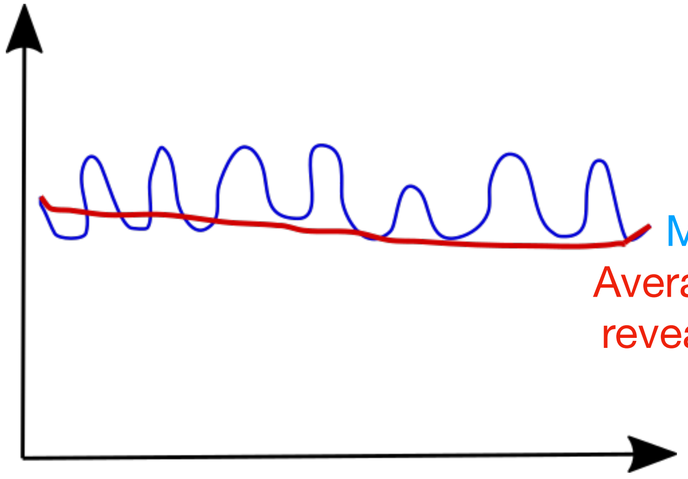
Plot loss averaged over last (e.g.) 1000 samples



Noisy plot - cost is going down
Smaller learning rate -learns slowly
SGD oscillates less

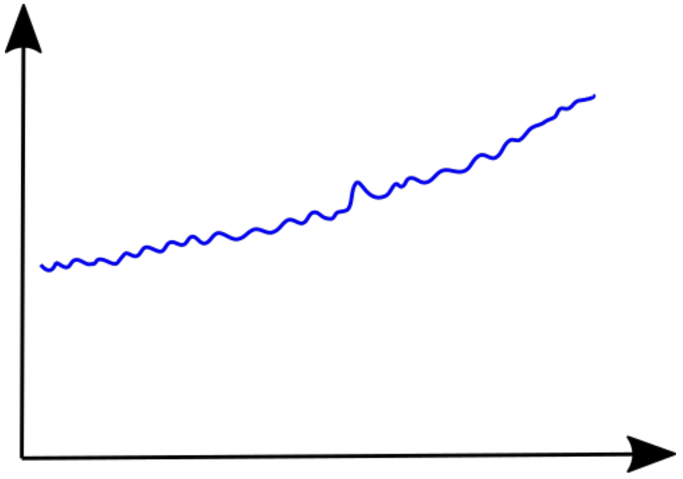
Smoother curve by averaging
over more samples





Model is not learning?
Averaging over more samples
reveals that cost going down

Diverging model
Decrease learning rate



Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- Picking a learning rate α that is very small has no disadvantage and can only speed up learning
- If we reduce the learning rate α (and run SGD), its possible that we may find a set of better parameters than with larger α
- If we want SGD to converge to a (local) minimum rather than wander or "oscillate" around it, we should slowly increase α over time.
- If we plot $cost(\theta, (x^{(i)}, y^{(i)}))$ (averaged over the last 1000 examples) and SGD does not seem to be reducing the cost, one possible problem may be that the learning rate α is poorly tuned

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- Picking a learning rate α that is very small has no disadvantage and can only speed up learning
- If we reduce the learning rate α (and run SGD), its possible that we may find a set of better parameters than with larger α
- If we want SGD to converge to a (local) minimum rather than wander or "oscillate" around it, we should slowly increase α over time.
- If we plot $cost(\theta, (x^{(i)}, y^{(i)}))$ (averaged over the last 1000 examples) and SGD does not seem to be reducing the cost, one possible problem may be that the learning rate α is poorly tuned

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Neural Networks

- Perceptrons
- Multi-layer Perceptrons
- Backpropagation
- Image classification
- Loss- and activation functions for more than two classes
- Better activation functions
- Better gradient descent

Course Outline

- Introduction to Image Analysis
- Basics: Neural Networks
- **Convolutional Neural Networks**
- Transformers
- Model Interpretability
- Self-supervised Learning
- Generative Models (GANs, Diffusion)