ACM Summer School on NLP and ML

Neural Network

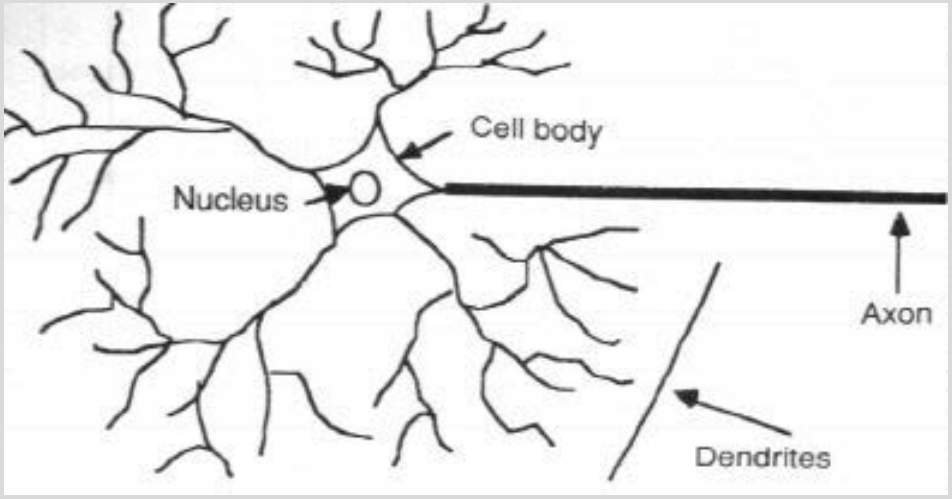
Sudeshna Sarkar
IIT Kharagpur

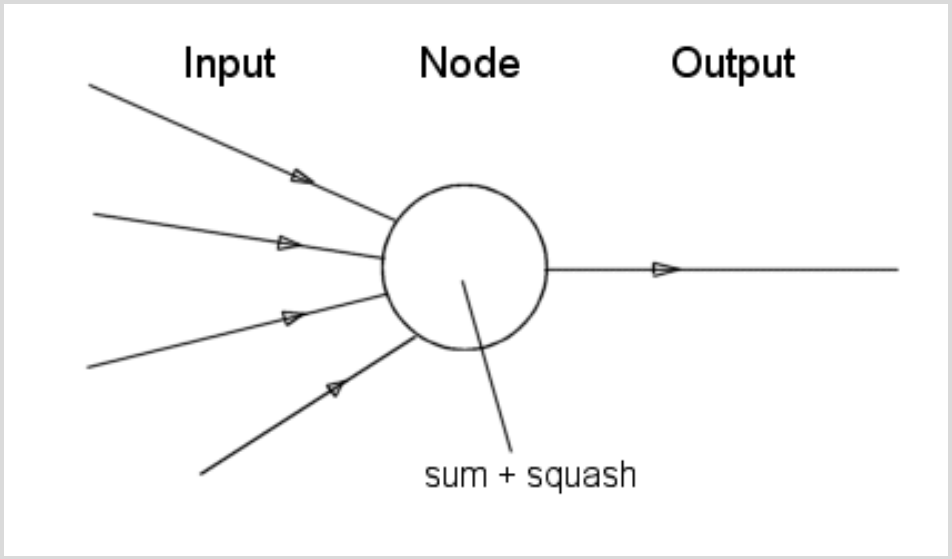
Date: 12 June 2017

Introduction

- Inspired by the human brain.
- Some NNs are models of biological neural networks
- Human brain contains a massively interconnected net of 10^{10}-10^{11} (10 billion) neurons (cortical cells)
  - Massive parallelism – large number of simple processing units
  - Connectionism – highly interconnected
  - Associative distributed memory
    - Pattern and strength of synaptic connections

Neuron

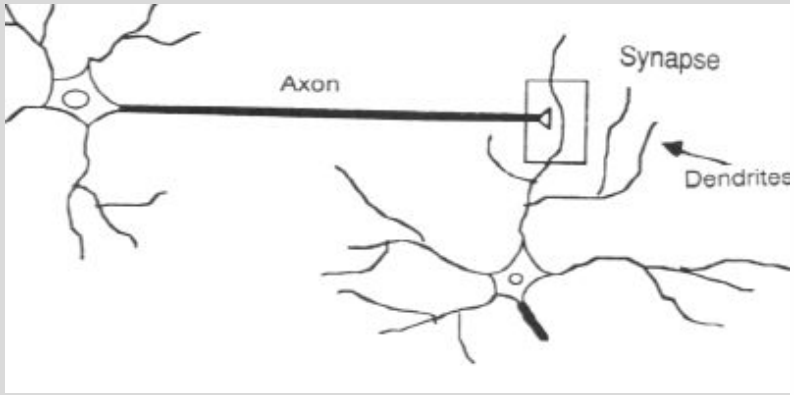


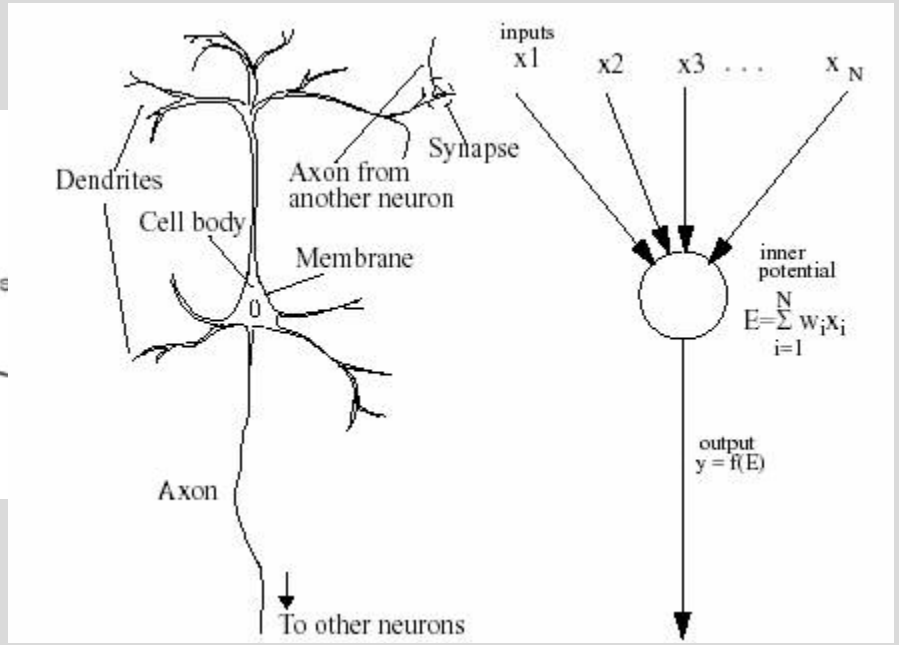


Neural Unit

ANNs

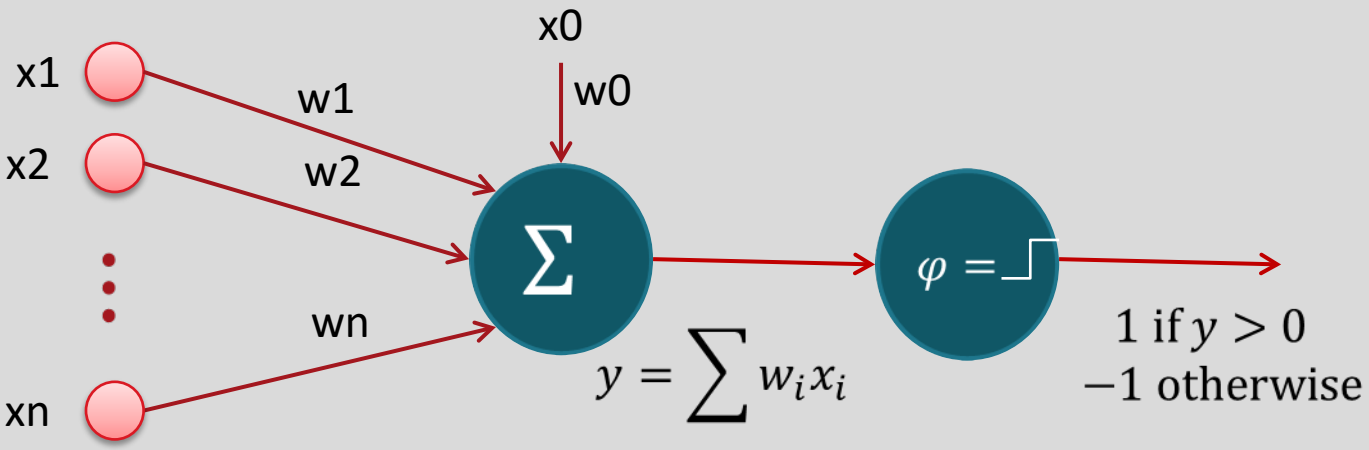
- ANNs incorporate the two fundamental components of biological neural nets:
  1. Nodes - Neurones
  2. Weights - Synapses





Perceptrons

- Basic unit in a neural network: Linear separator
  - N inputs, x1 ... xn
  - Weights for each input, w1 ... wn
  - A bias input x0 (constant) and associated weight w0
  - Weighted sum of inputs, $y = \sum_{i=0}^{n} w_i x_i$
  - A threshold function, i.e., 1 if y > 0, −1 if y <= 0



Perceptron training rule

Updates perceptron weights for a training ex as follows:

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \eta(y - \hat{y})x_i$$

- If the data is linearly separable and η is sufficiently small, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations

The limitations of Perceptrons

- If you are allowed to choose the features by hand and if you use enough features, you can do almost anything.
  - For binary input vectors, we can have a separate feature unit for each of the exponentially many binary vectors and so we can make any possible discrimination on binary input vectors.
    - This type of table look-up won't generalize.
- But once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.

Linear neurons

- The neuron has a real-valued output which is a weighted sum of its inputs

$$\hat{y} = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

- Define the error as the squared residuals summed over all training cases:

$$E = \frac{1}{2} \sum_j (y - \hat{y})^2$$

- Differentiate to get error derivatives for weights

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{j=1..m} \frac{\partial \hat{y}_j}{\partial w_i} \frac{\partial E_j}{\partial \hat{y}_j} = - \sum_{j=1..m} x_{i,j} (y_j - \hat{y}_j)$$

- The batch delta rule changes the weights in proportion to their error derivatives summed over all training cases

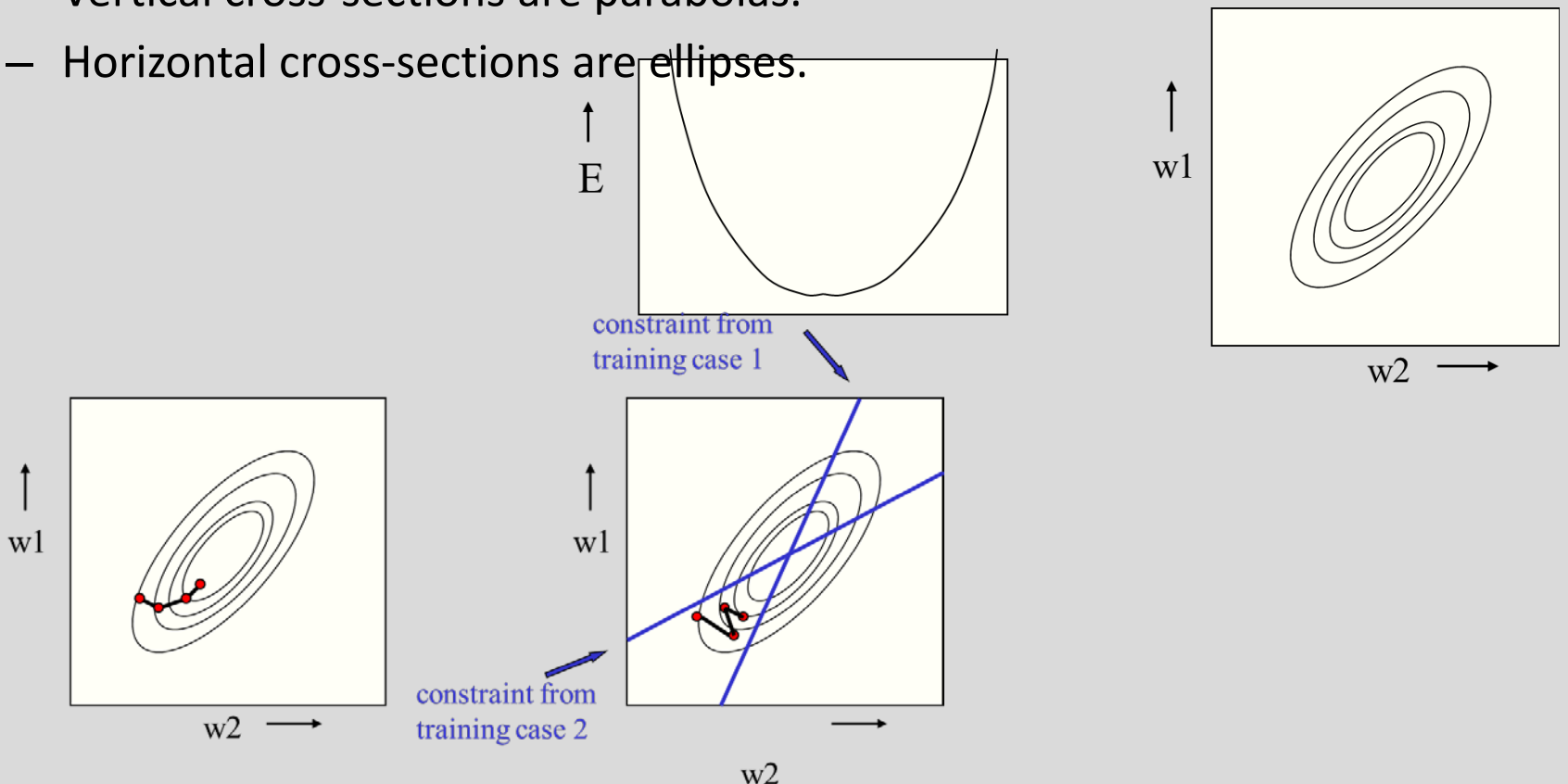
$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

- Perceptron training rule may not converge if points are not linearly separable
- Gradient descent by changing the weights by the total error for all training points.
  - If the data is not linearly separable, then it will converge to the best fit

Error Surface

- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
  - For a linear neuron, it is a quadratic bowl.
  - Vertical cross-sections are parabolas.
  - Horizontal cross-sections are ellipses.



Batch Line and Stochastic Learning

Batch Learning

- Steepest descent on the error surface

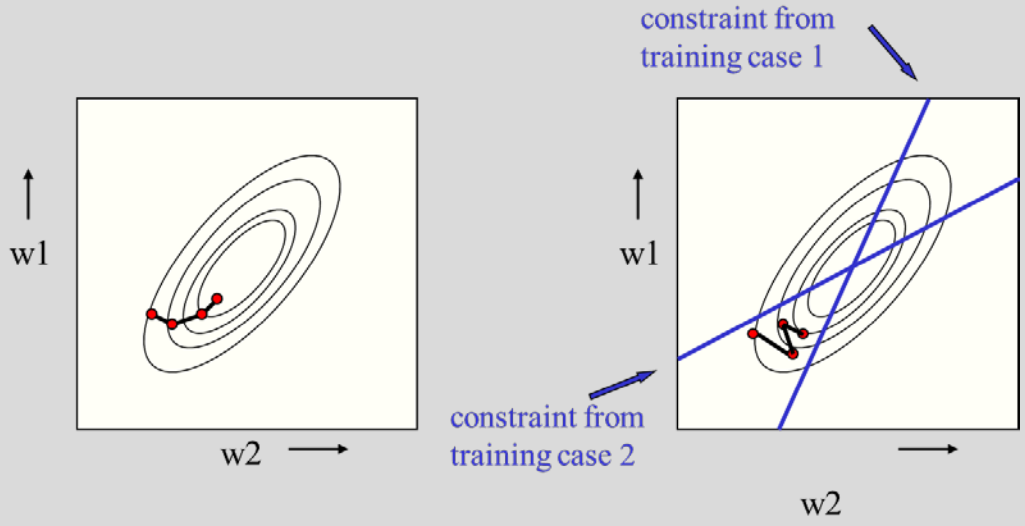
Stochastic/ Online Learning

For each example compute the gradient.

$$E = \frac{1}{2}(y - \hat{y})^2$$

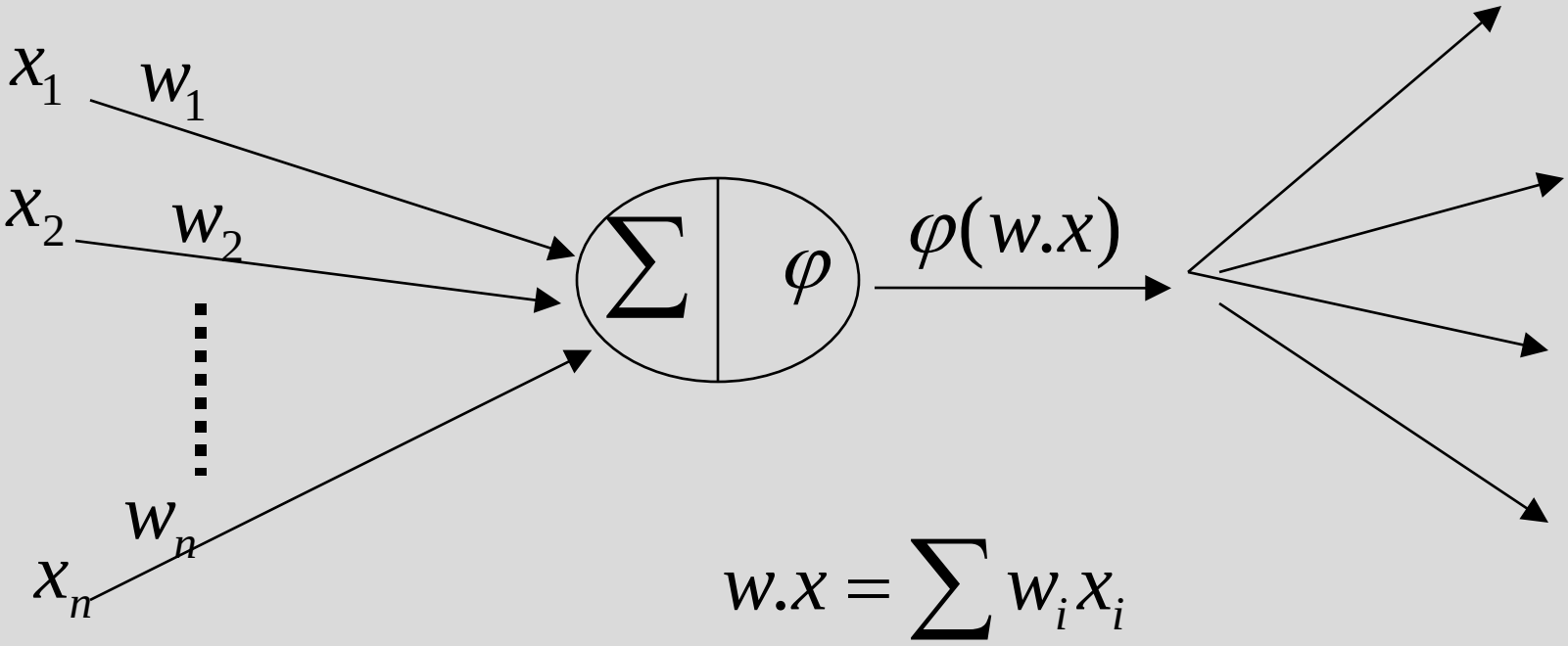
$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\frac{\partial \hat{y}}{\partial w_i}\frac{\partial E_j}{\partial \hat{y}}$$

$$= -x_i(y - \hat{y})$$



Computation at Units

- Compute a 0-1 or a *graded* function of the weighted sum of the inputs
- $\varphi()$ is the *activation* function



Neuron Model: Logistic Unit

$$\varphi(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-w.x}}$$

$$\phi'(z) = \varphi(z)\big(1-\varphi(z)\big)$$

$$E = \frac{1}{2}\sum_d (y-\hat{y})^2 = \frac{1}{2}\sum_d (y-\varphi(w.x_d))^2$$

$$\frac{\partial E}{\partial w_i} = \sum_d \frac{1}{2}\frac{\partial E_d}{\partial \widehat{y_d}}\frac{\partial \widehat{y_d}}{\partial w_i}$$

$$= \sum_d (y_d - \widehat{y_d})\frac{\partial y}{\partial w_i}(y_d - \varphi(w.x_d))$$

$$= -\sum_d (y_d - \widehat{y_d})\,\varphi'(w.x_d)\,x_{i,d}$$

$$= -\sum_d (y_d - \widehat{y_d})\widehat{y_d}\,(1-\widehat{y_d})x_{i,d}$$

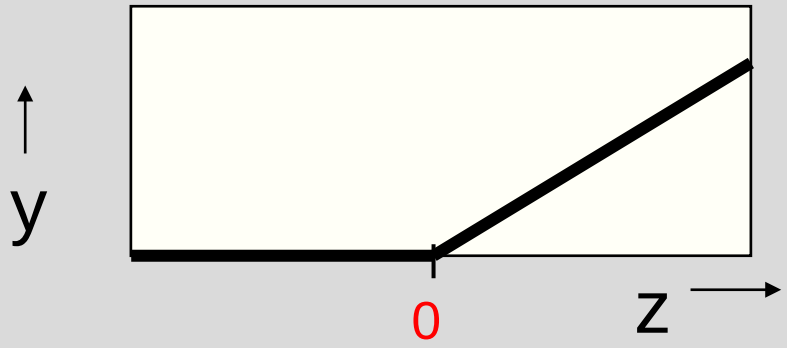
Training Rule: $\Delta w_i = \eta \sum_d (y_d - \widehat{y_d})\widehat{y_d}\,(1-\widehat{y_d})x_{i,d}$

Rectified Linear Units

They compute a linear weighted sum of their inputs.
The output is a non-linear function of the total input.

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

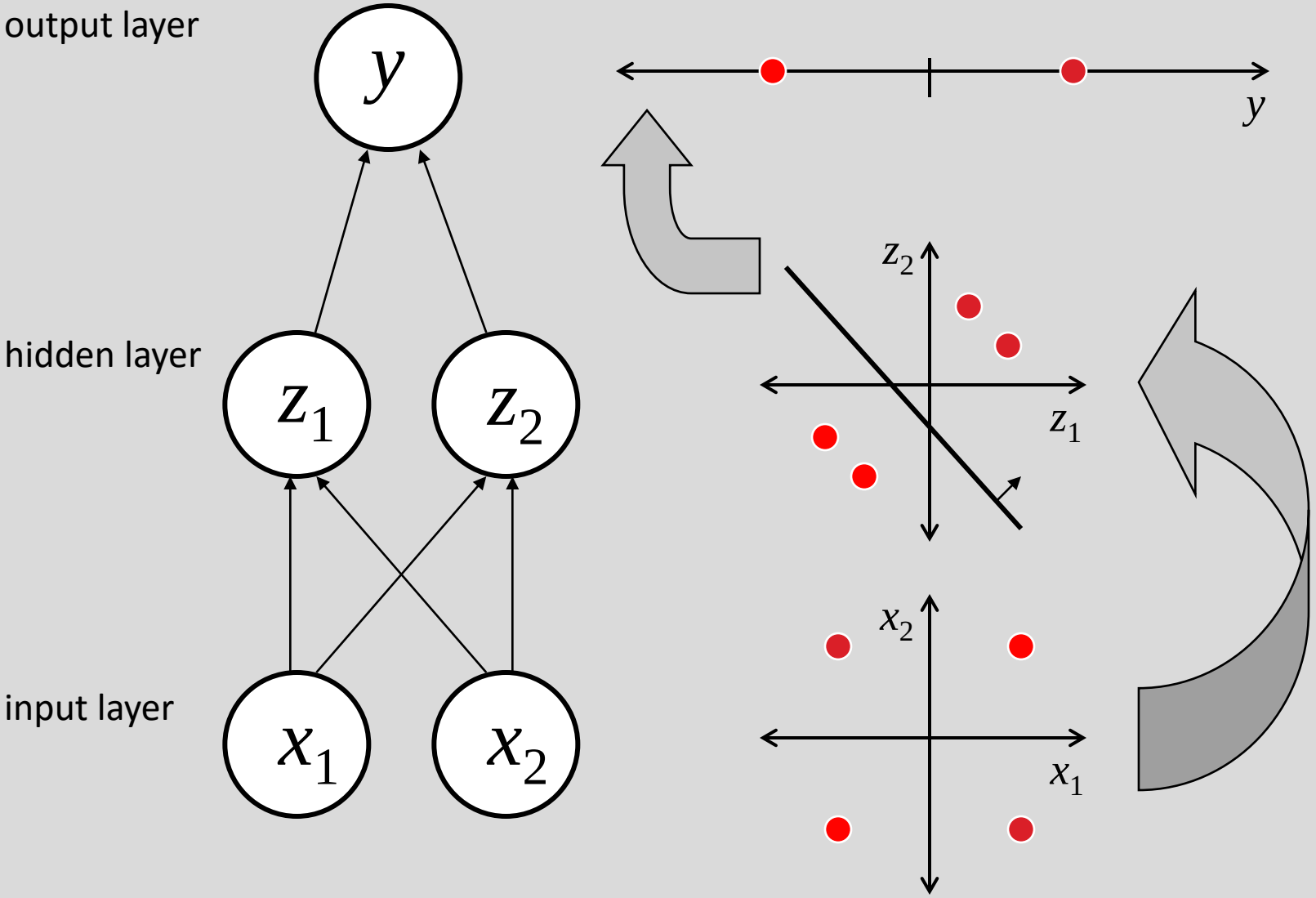


Limitations of Perceptrons

- Perceptrons have a *monotinicity* property:

  If a link has positive weight, activation can only increase as the corresponding input value increases (*irrespective* of other input values)
- Can't represent functions where input *interactions* can cancel one another's effect (e.g. XOR)
- Can represent only linearly separable functions

A solution: multiple layers



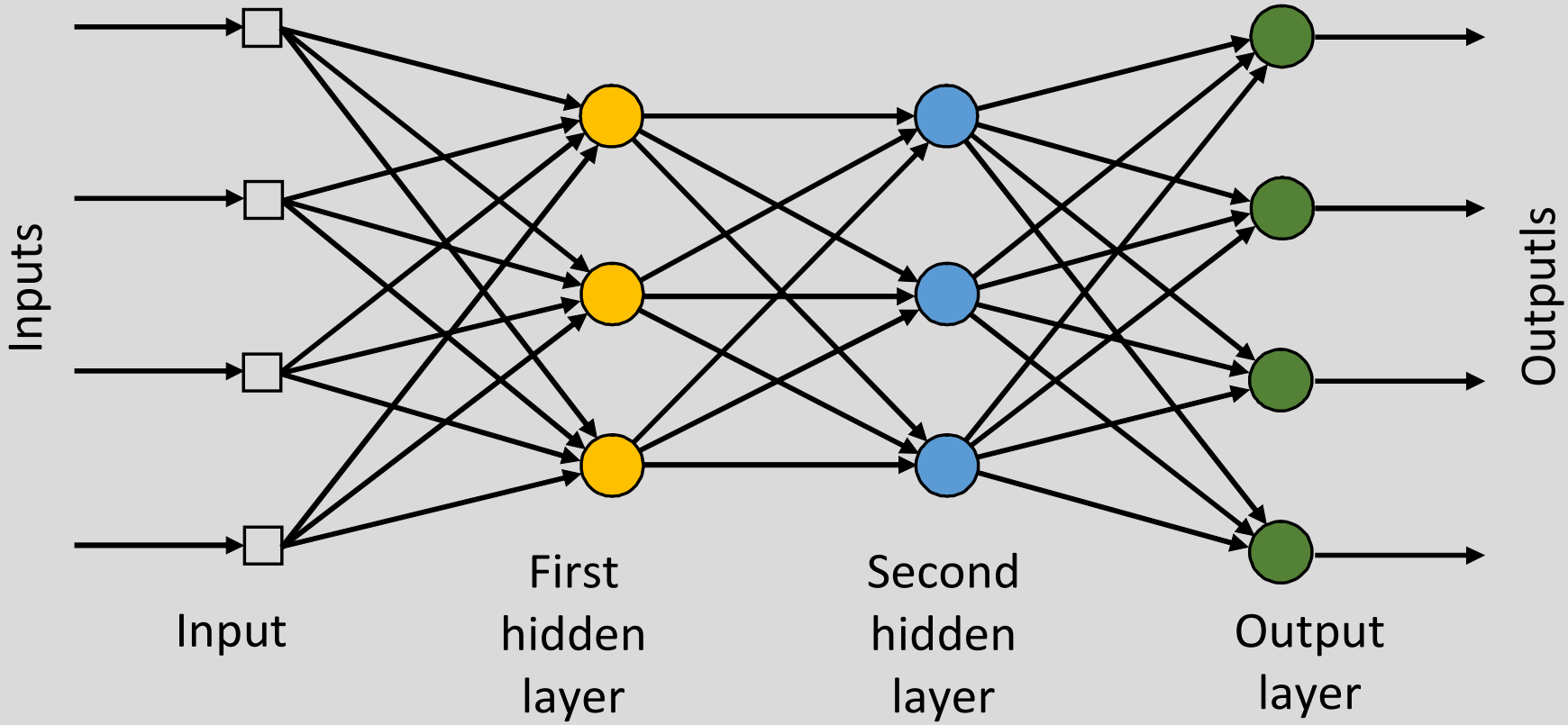
Learning with hidden units

- Networks without hidden units are very limited in the input-output mappings they can learn to model.
- We need multiple layers of adaptive, non-linear hidden units.
- How can we train such nets?
  - We need an efficient way of adapting all the weights.
  - Learning the weights going into hidden units is equivalent to learning features.
  - Nobody is telling us directly what the hidden units should do.

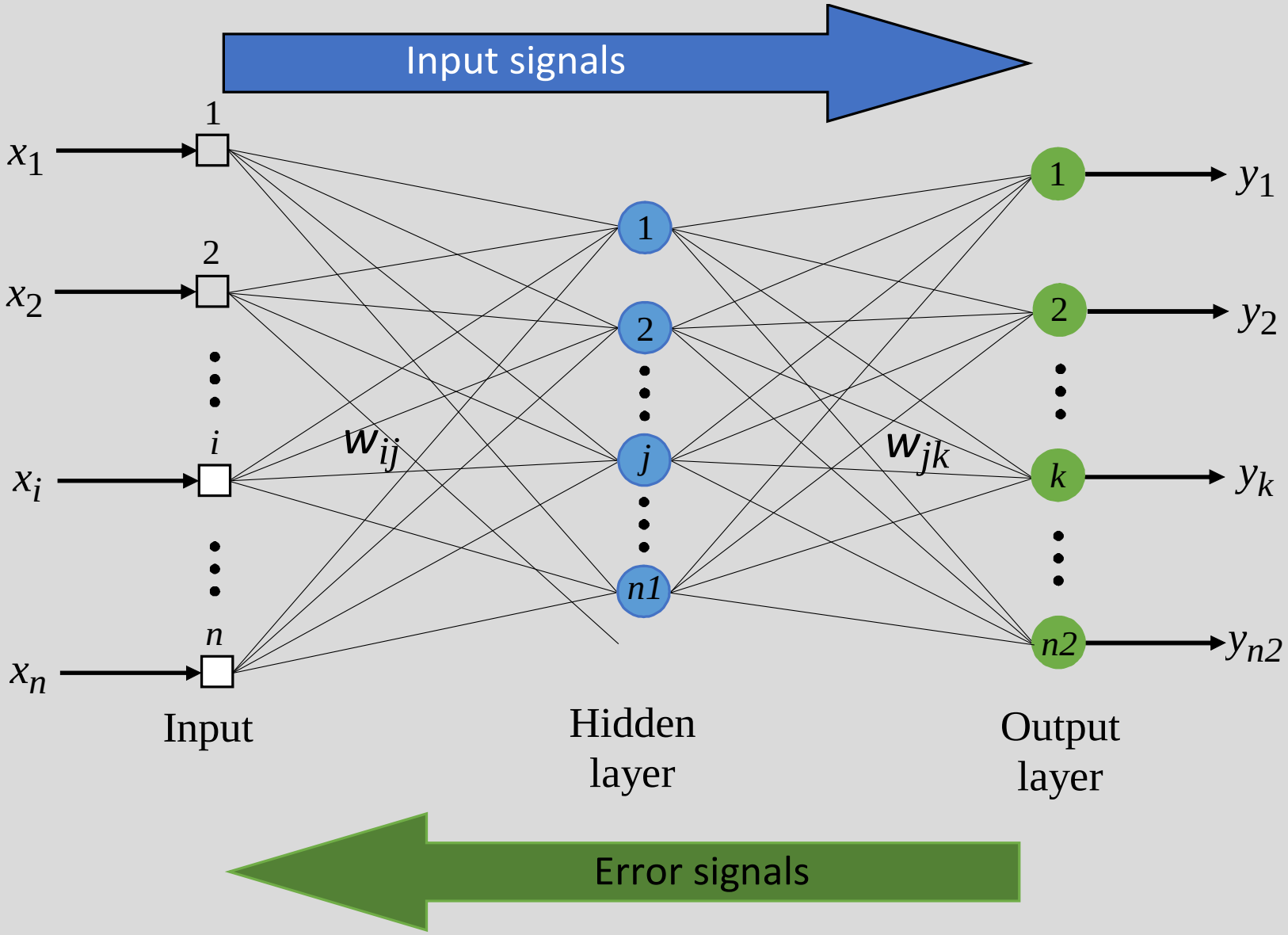
Power/Expressiveness of Multilayer Networks

- Can represent interactions among inputs
- Two layer networks can represent any Boolean function, and continuous functions (within a tolerance) as long as the number of hidden units is sufficient and appropriate activation functions used
- Learning algorithms exist, but weaker guarantees than perceptron learning algorithms

Multilayer Network

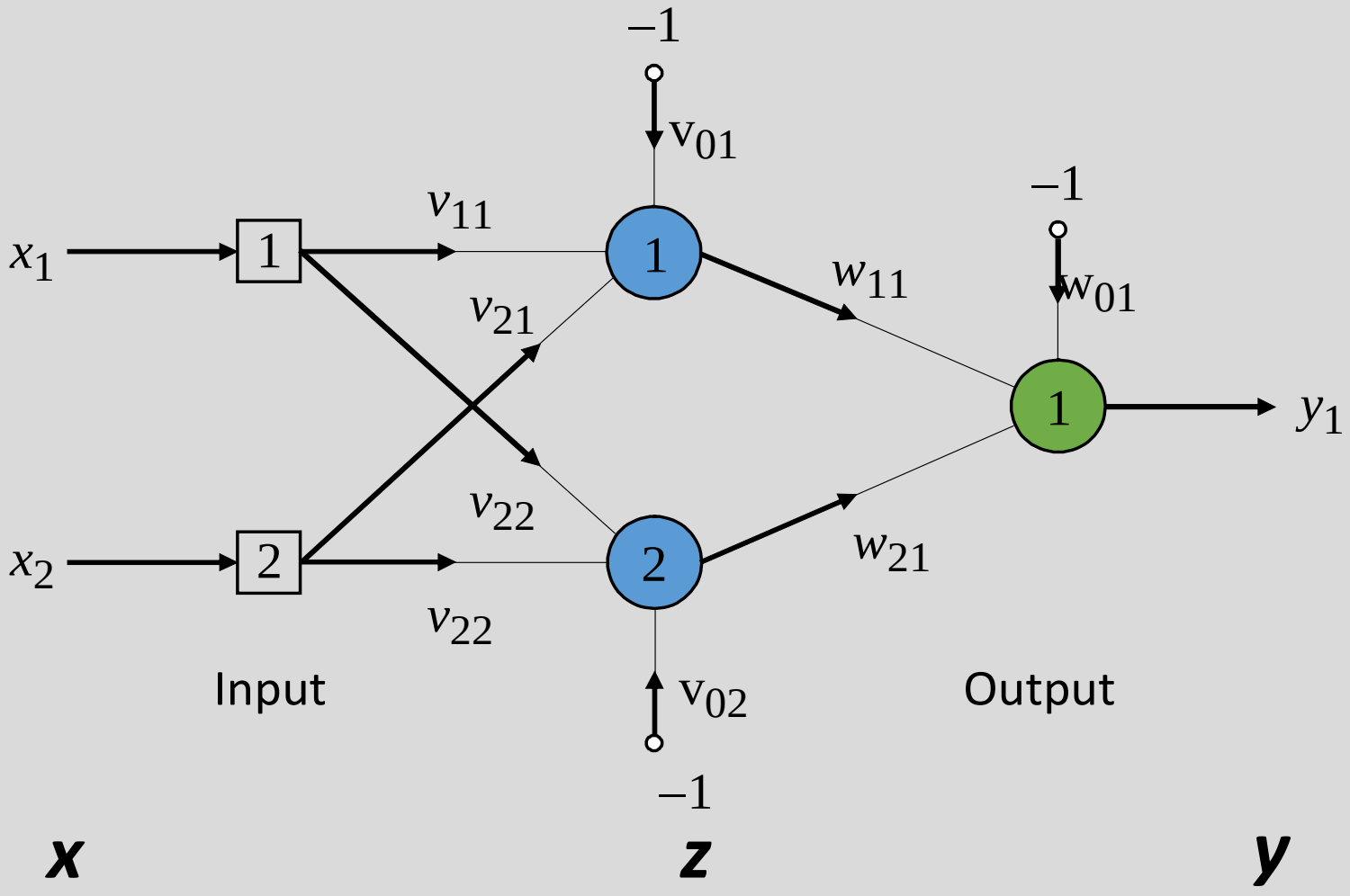


Two-layer back-propagation neural network



The back-propagation training algorithm

- Step 1: Initialisation
  Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range



Loss Function

- Mean square loss
- Hinge Loss
- Cross Entropy Loss

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Backprop

- Initialization
  - Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range
- Forward computing:
  - Apply an input vector ***x*** to input units
  - Compute activation/output vector ***z*** on hidden layer

$$z_j = \varphi(\sum_i v_{ij} x_i)$$

  - Compute the output vector ***y*** on output layer

$$y_k = \varphi(\sum_j w_{jk} z_j)$$

y is the result of the computation.

Backprop

- Backpropagation: compute gradients of expressions through recursive application of chain rule.
- x is a vector of inputs.
- Given some function f(x) such as the loss function (L)
  - compute the gradient of f at x (i.e. $\nabla f(x)$).
- compute the gradient for the parameters (e.g. W,b)
- Use it to perform a parameter update.

Backpropagation

Backpropagation

- Backpropagation: compute gradients of expressions through recursive application of chain rule.
- Given function f(x) such as the loss function (L)
- x is a vector of inputs
- Compute the gradient of f at x (i.e. $\nabla f(x)$).

$$\nabla_W L$$

- compute the gradient for the parameters (e.g. W,b) so that we can use it to perform a parameter update.

Optimization

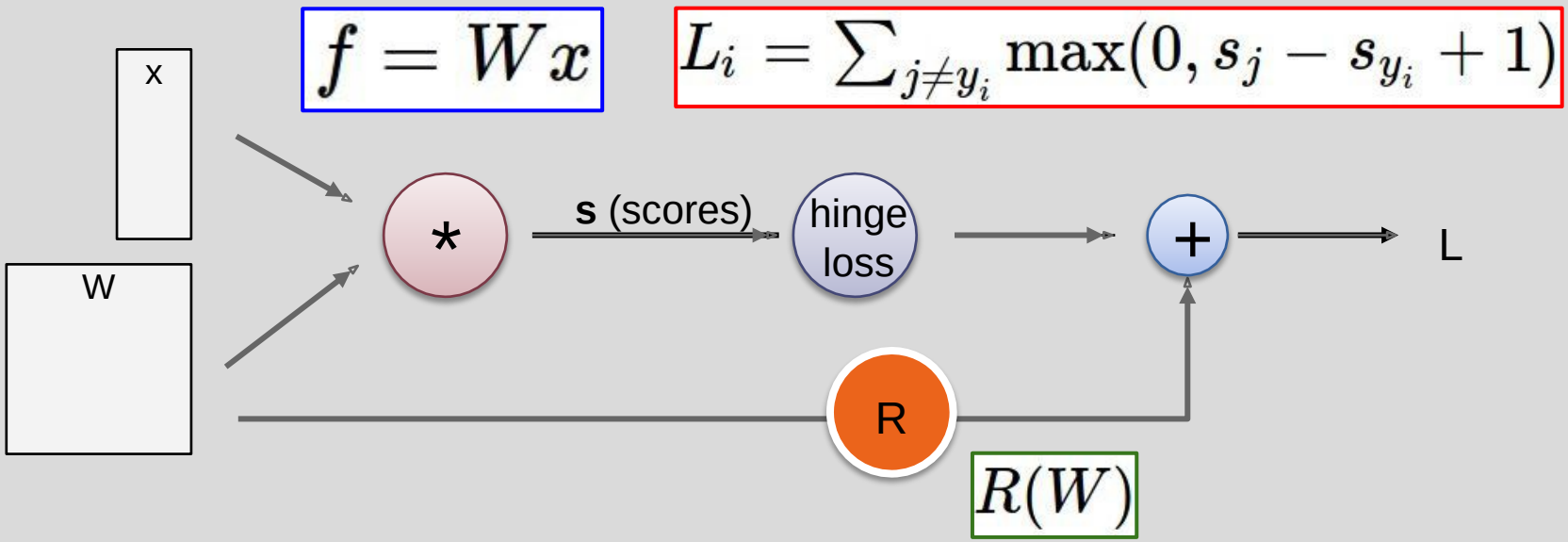
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Gradient Descent

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

- Numerical gradient: slow, approximate, easy to write
- Analytic gradient: fast, exact, error-prone
- In practice: Derive analytic gradient, check your implementation with numerical gradient

Computational Graph

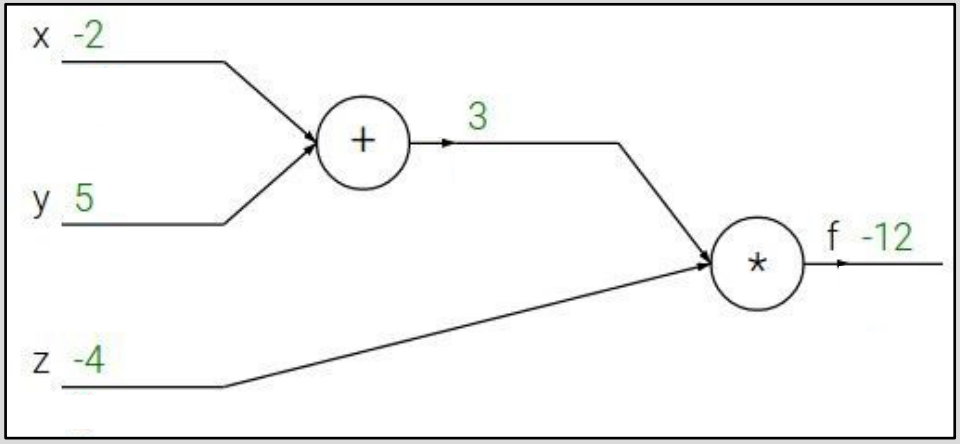


Intuitive understanding of backpropagation

- Backpropagation: local process.
- Every gate gets some inputs and can compute two things:
  1. its output value
  2. the local gradient of its inputs with respect to its output value.
- once the forward pass is over, during backpropagation the gate will eventually learn about the gradient of its output value on the final output of the entire circuit.
- Chain rule : the gate should take that gradient and multiply it into every gradient it normally computes for all of its inputs.

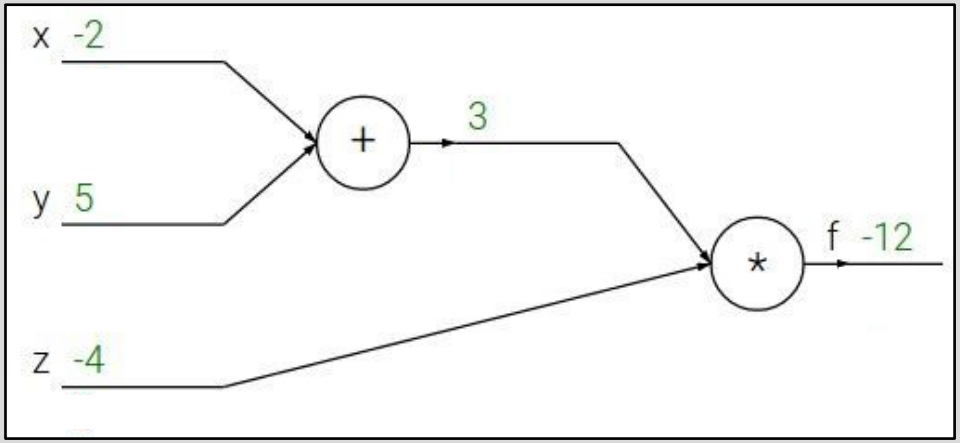
$$f(x, y, z) = (x + y)z$$

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



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

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



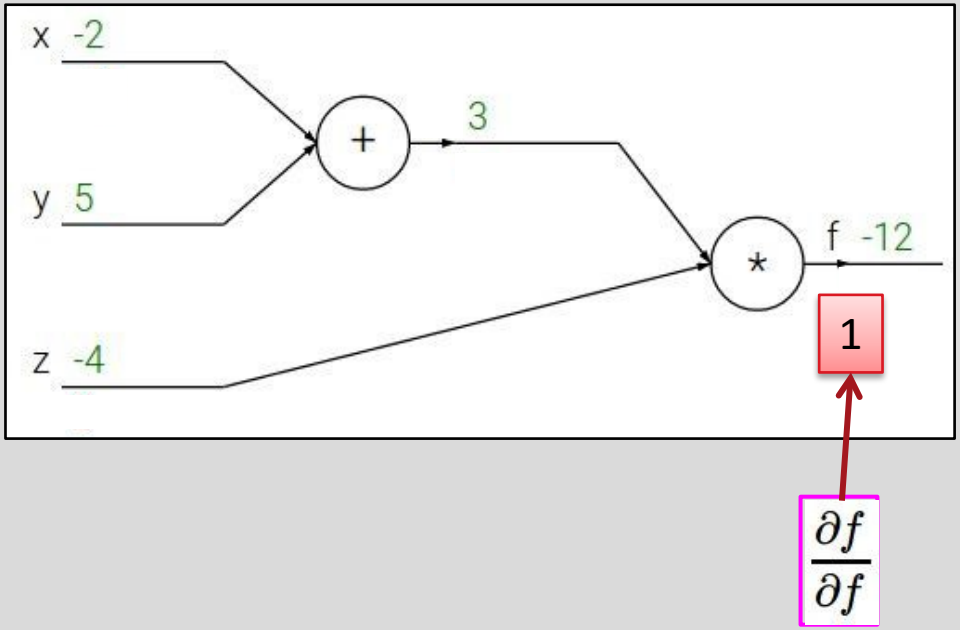
$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

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

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

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

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



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

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

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

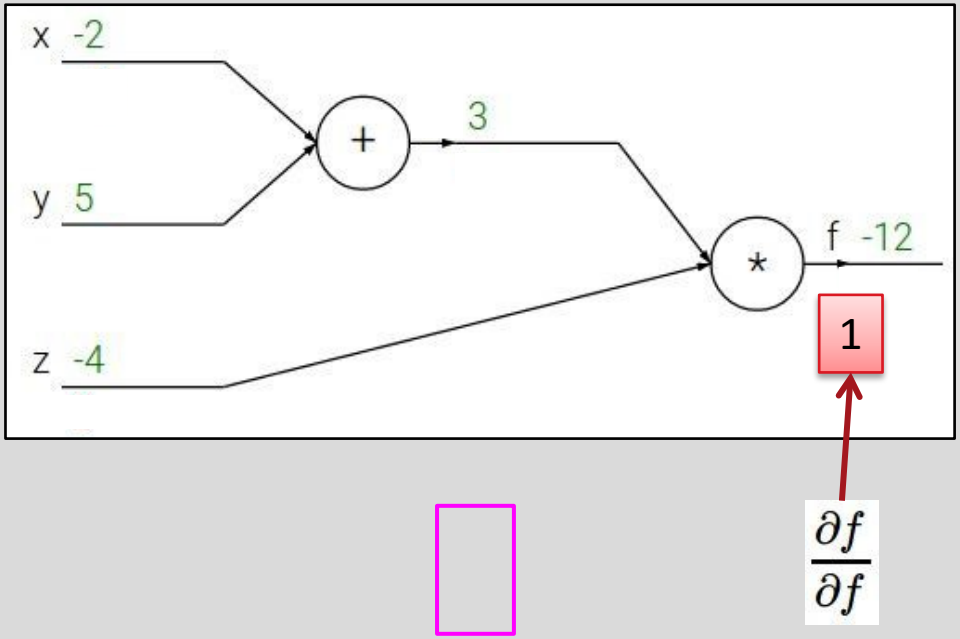
$$f(x, y, z) = (x + y)z$$

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

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

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

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



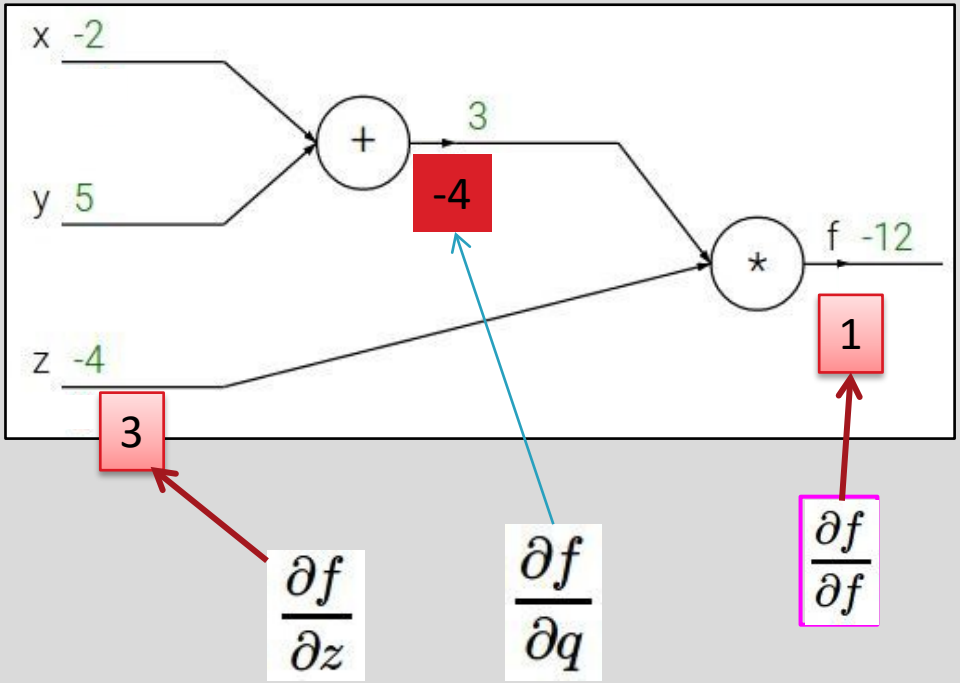
$$f(x, y, z) = (x + y)z$$

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

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

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

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



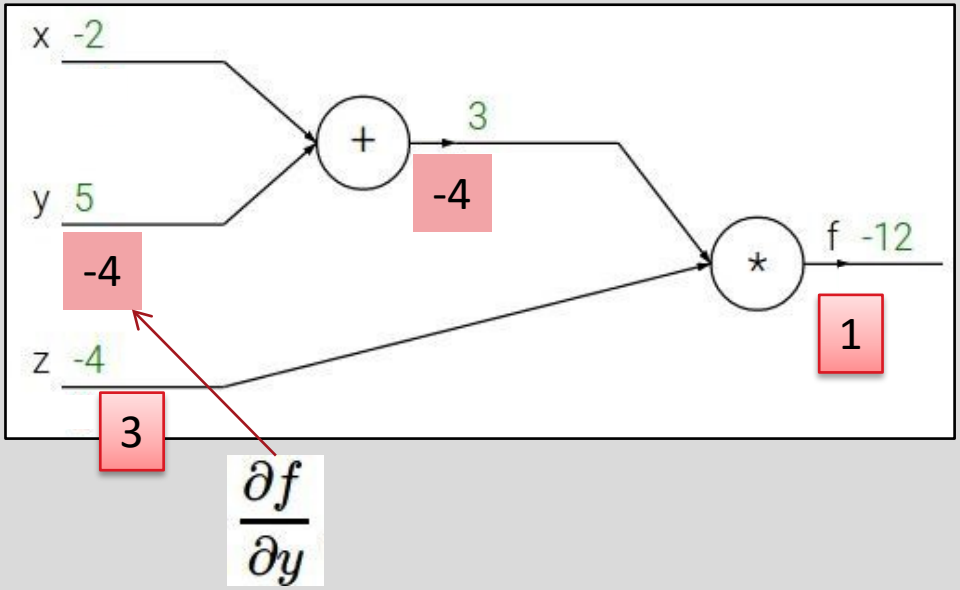
$$f(x, y, z) = (x + y)z$$

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

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

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

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



Chain Rule: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$

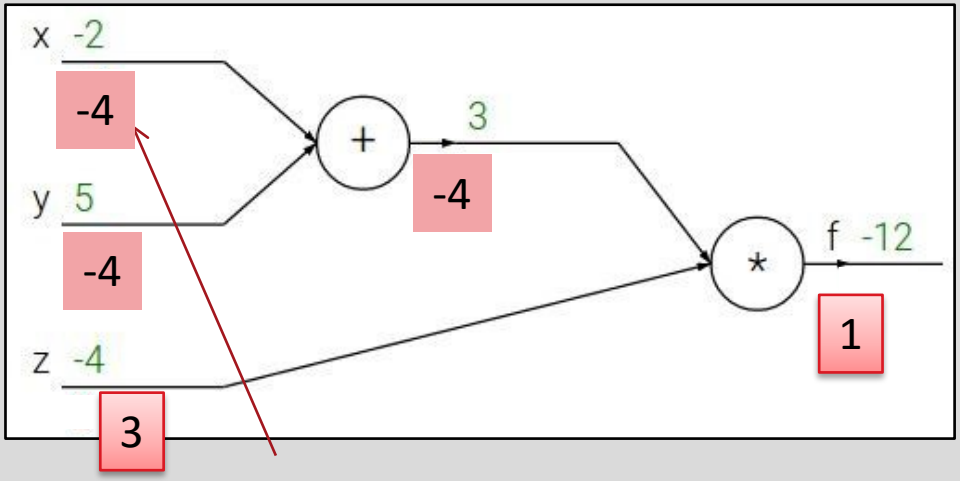
$$f(x, y, z) = (x + y)z$$

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

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

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

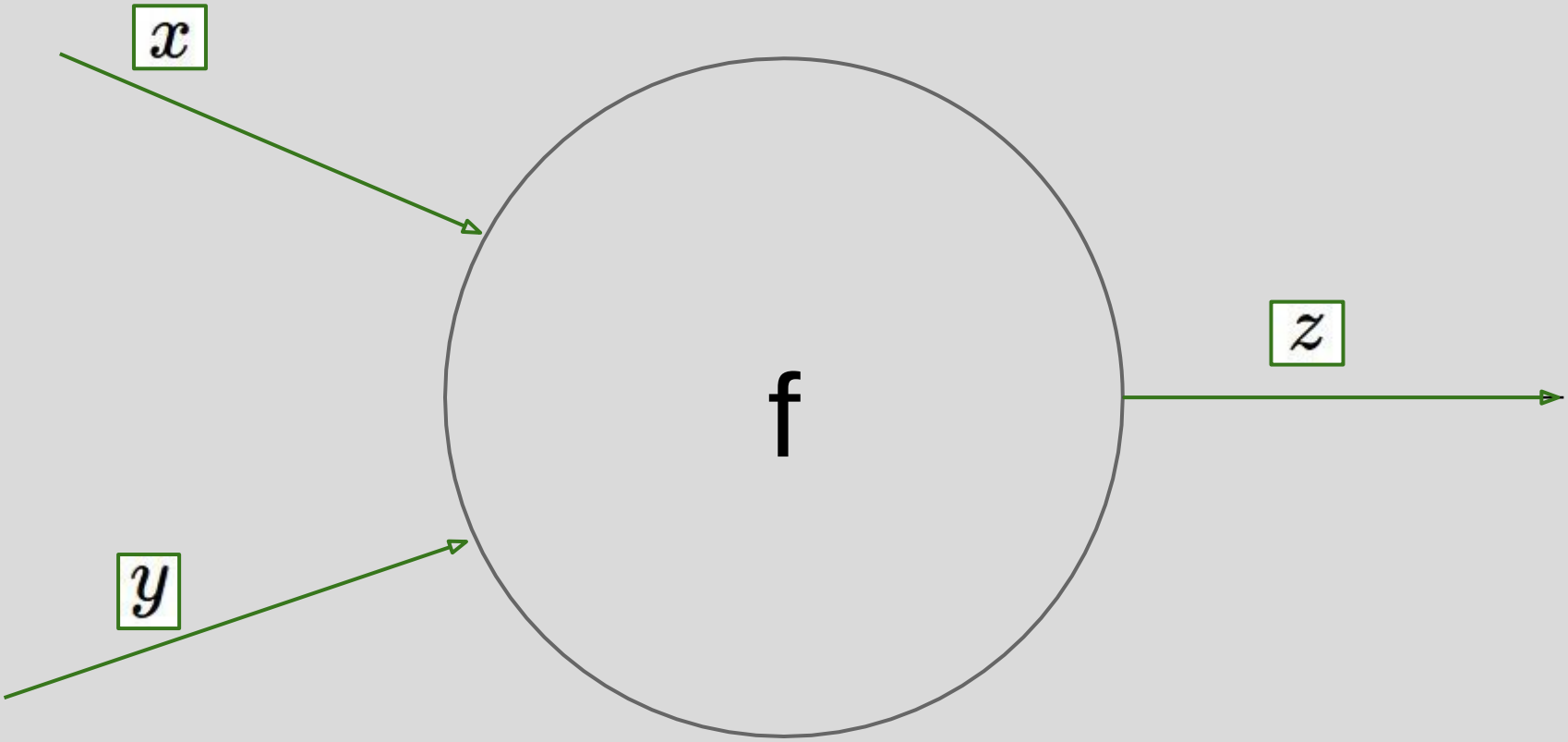
Want $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



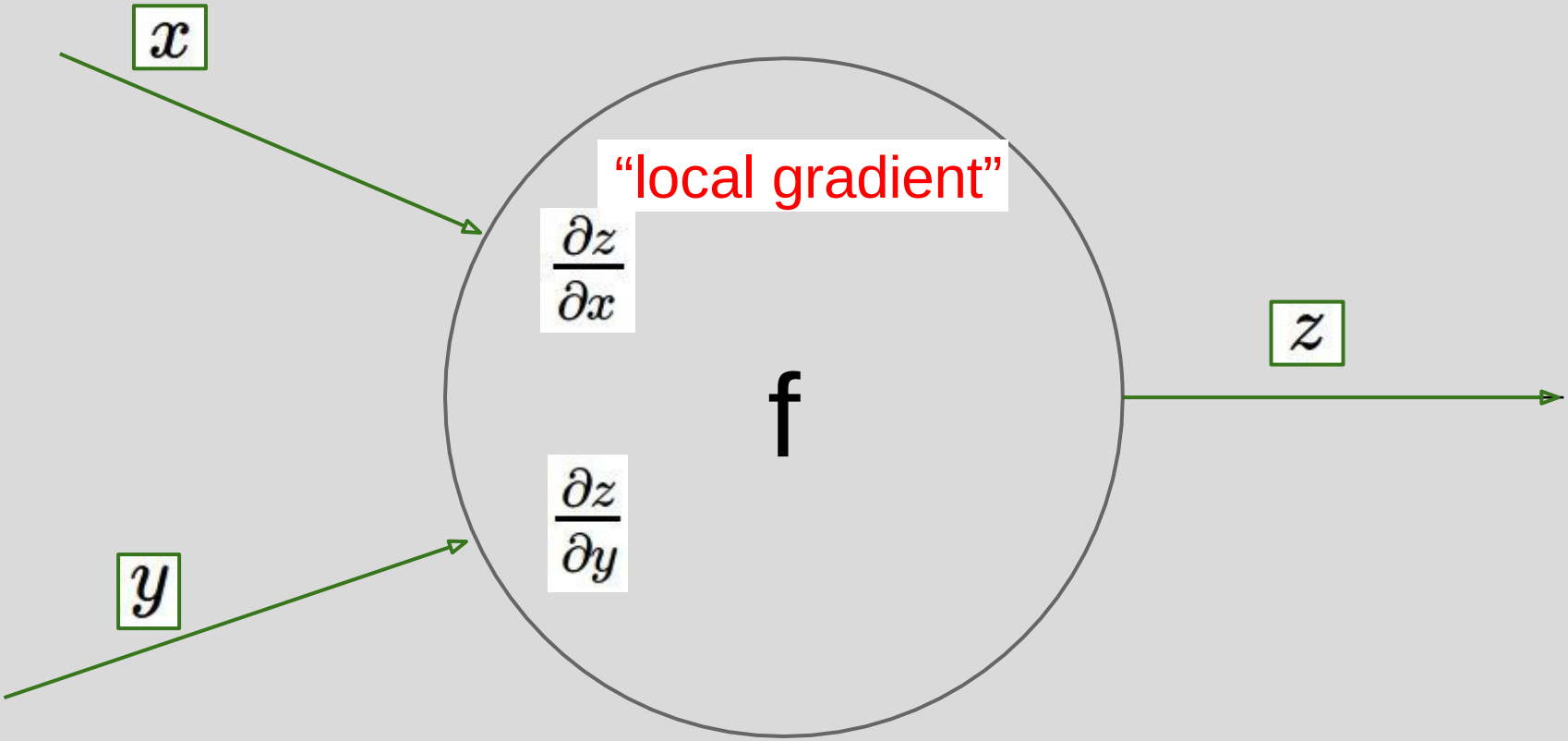
$\frac{\partial f}{\partial x}$

Chain Rule: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$

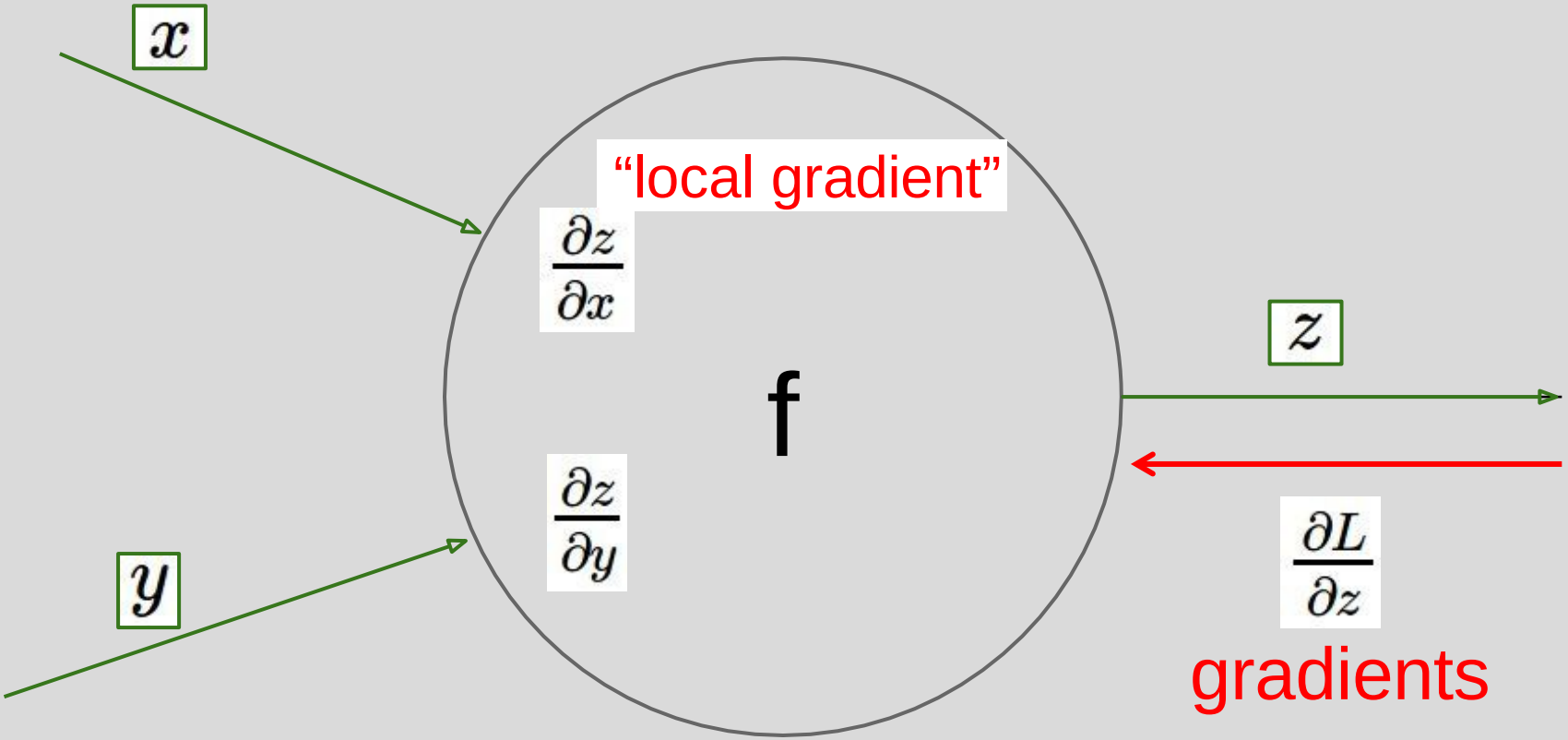
Activations



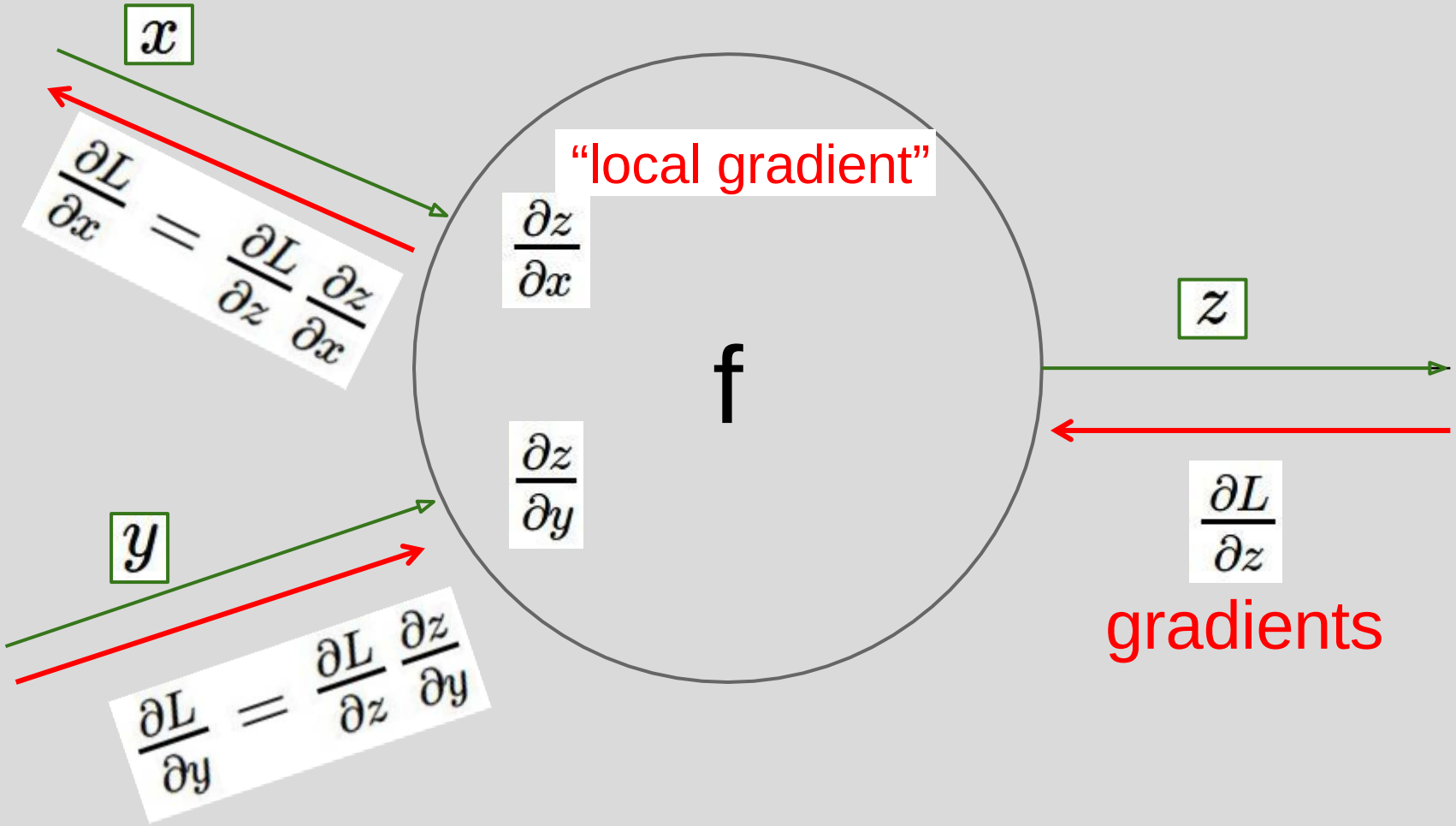
Activations



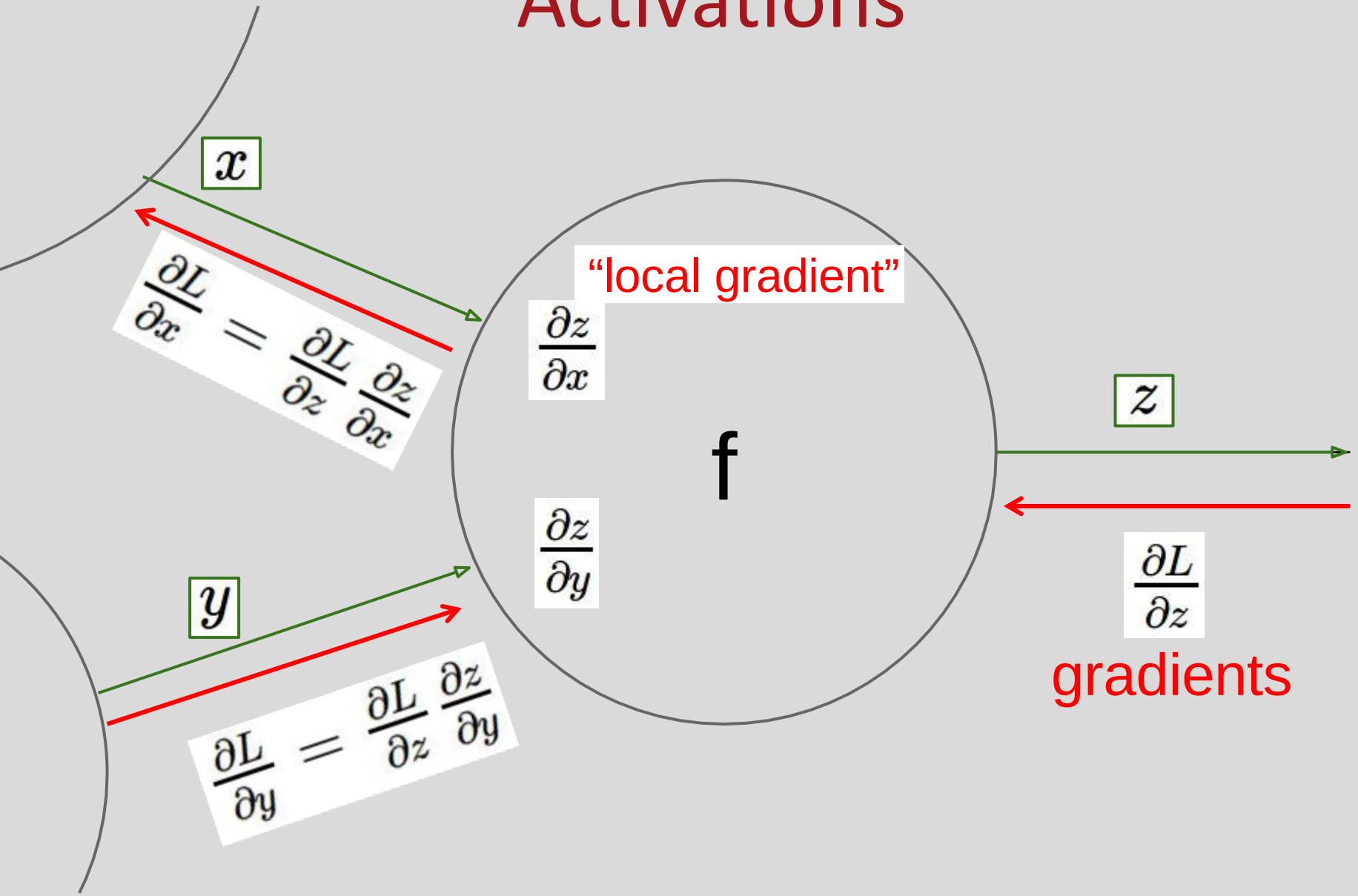
Activations



Activations

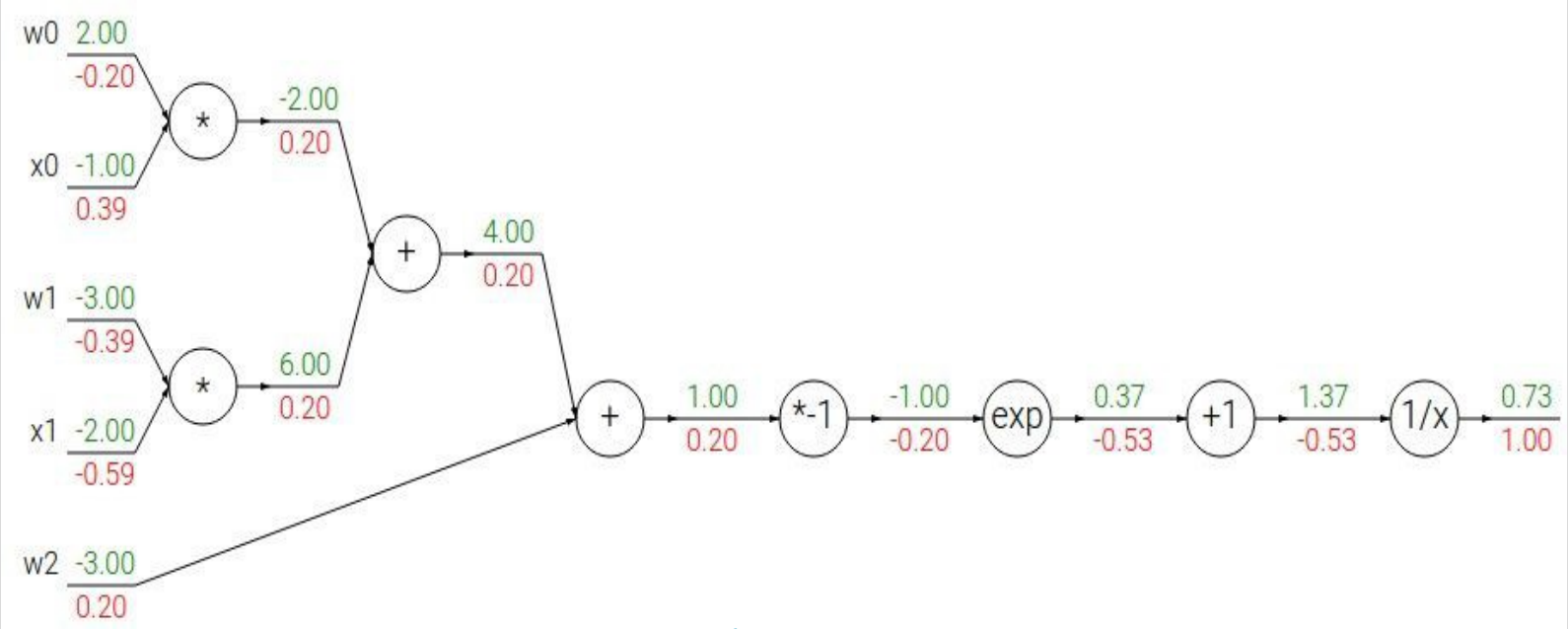


Activations



Another example

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



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

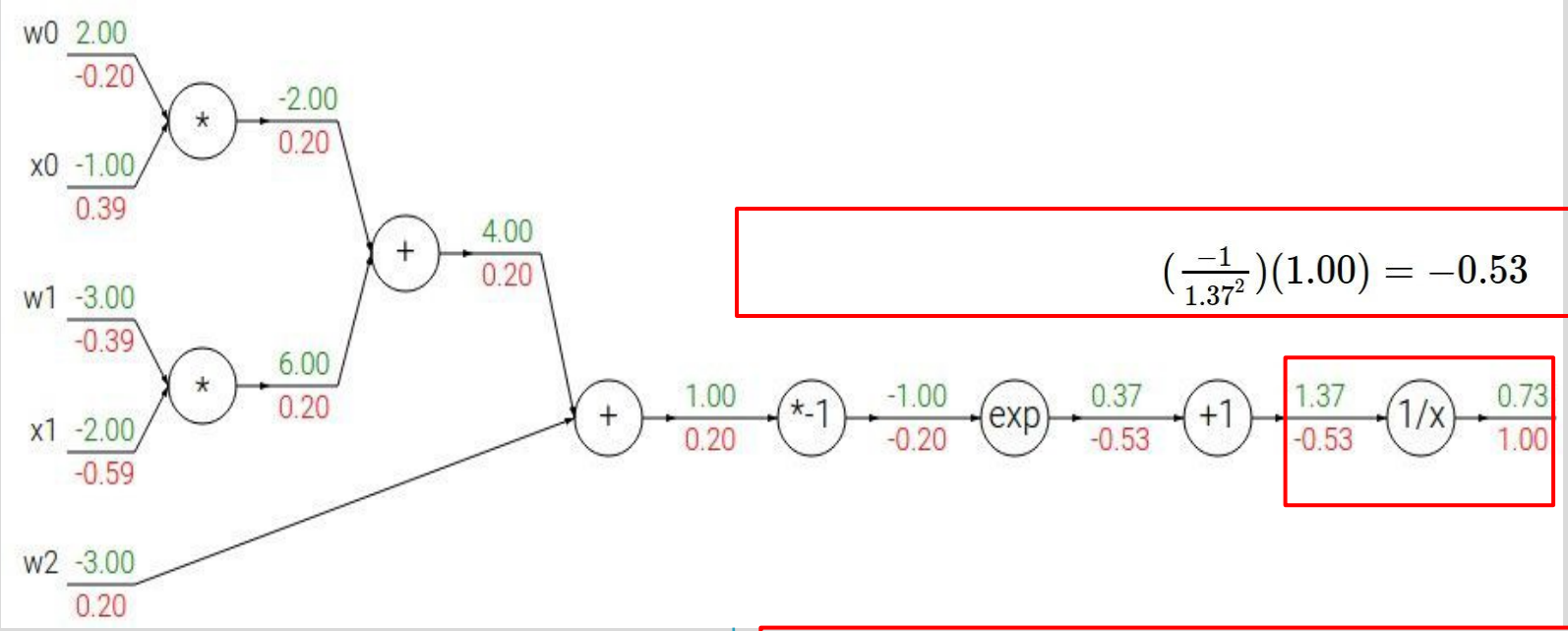
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

Another example

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



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

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

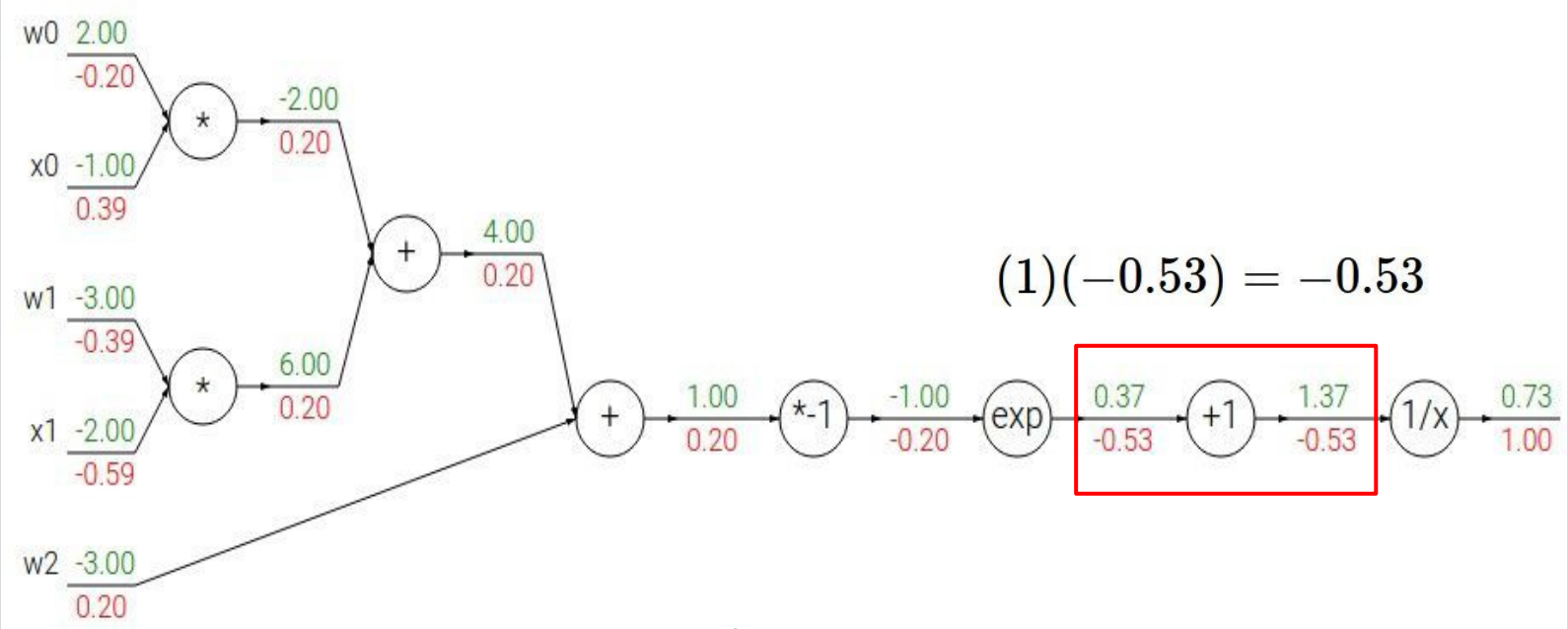
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

Another example

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



$$(1)(-0.53) = -0.53$$

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

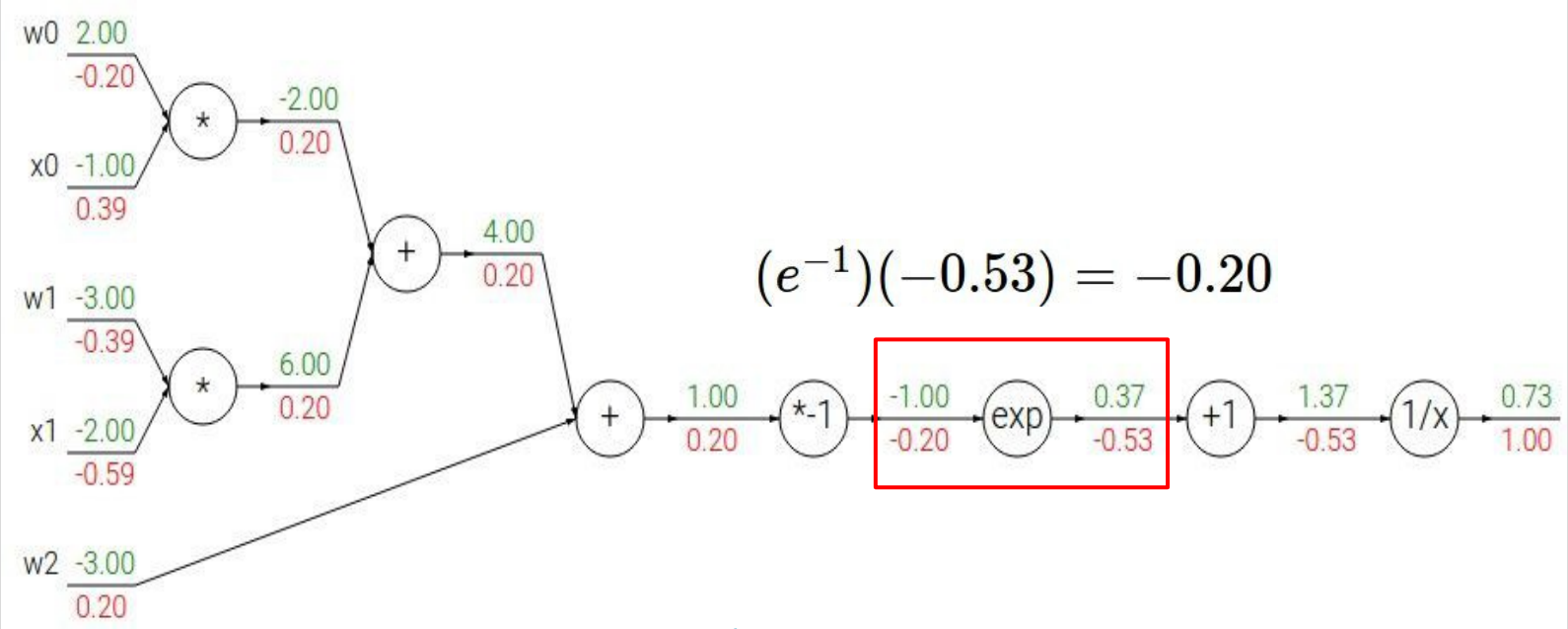
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

Another example

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



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

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

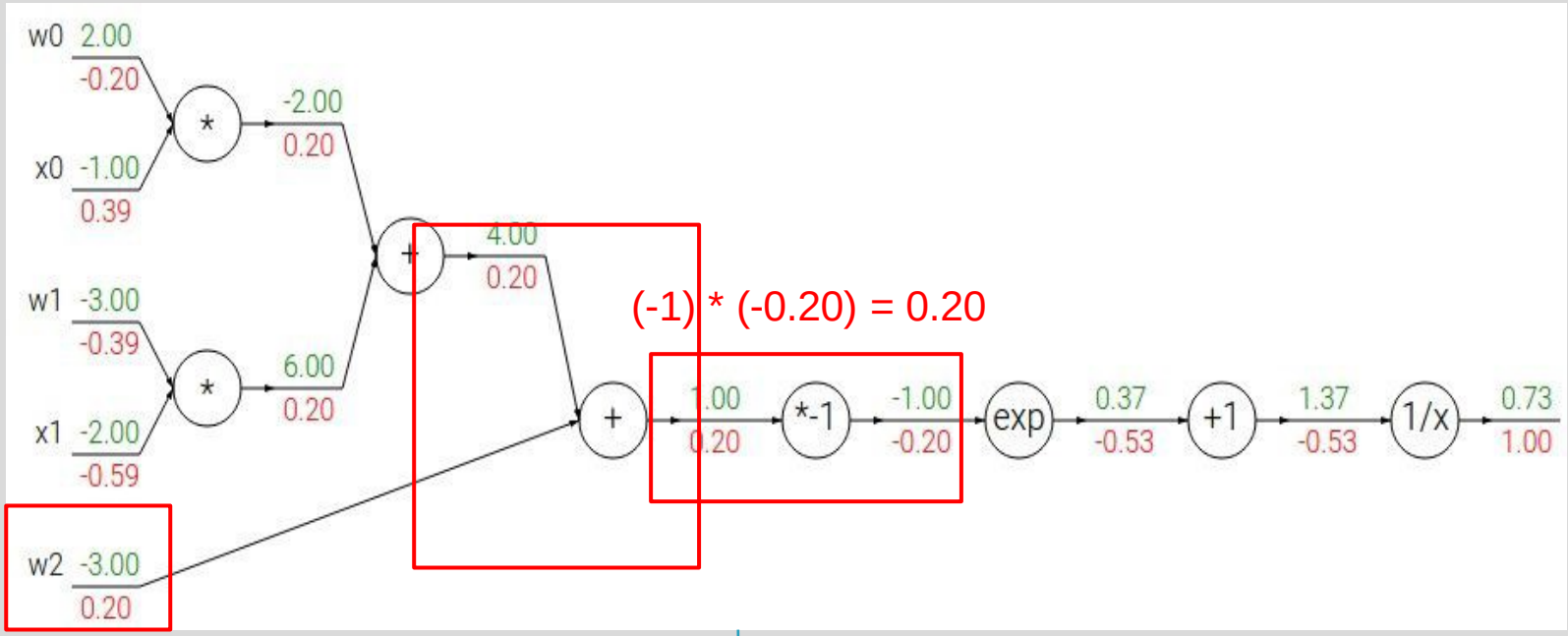
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

Another example

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



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

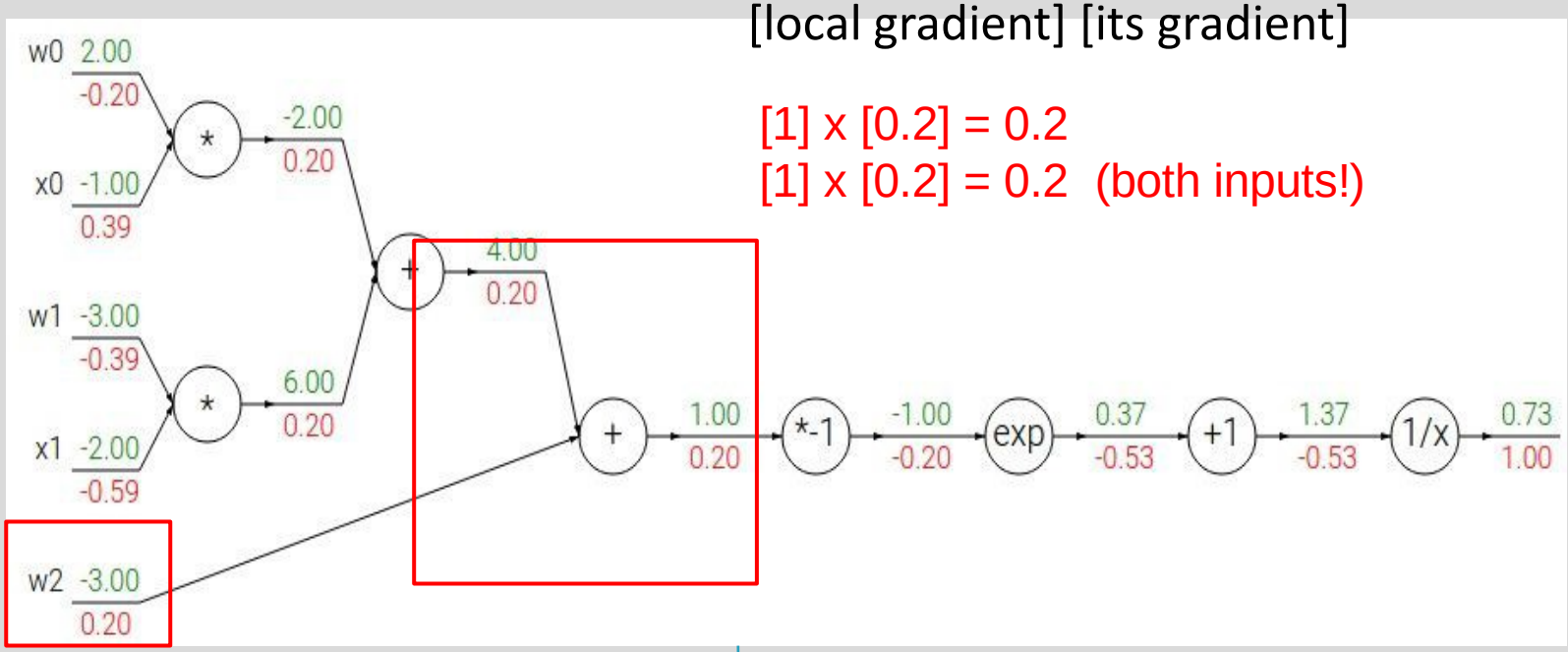
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

Another example

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



[local gradient] [its gradient]

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

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

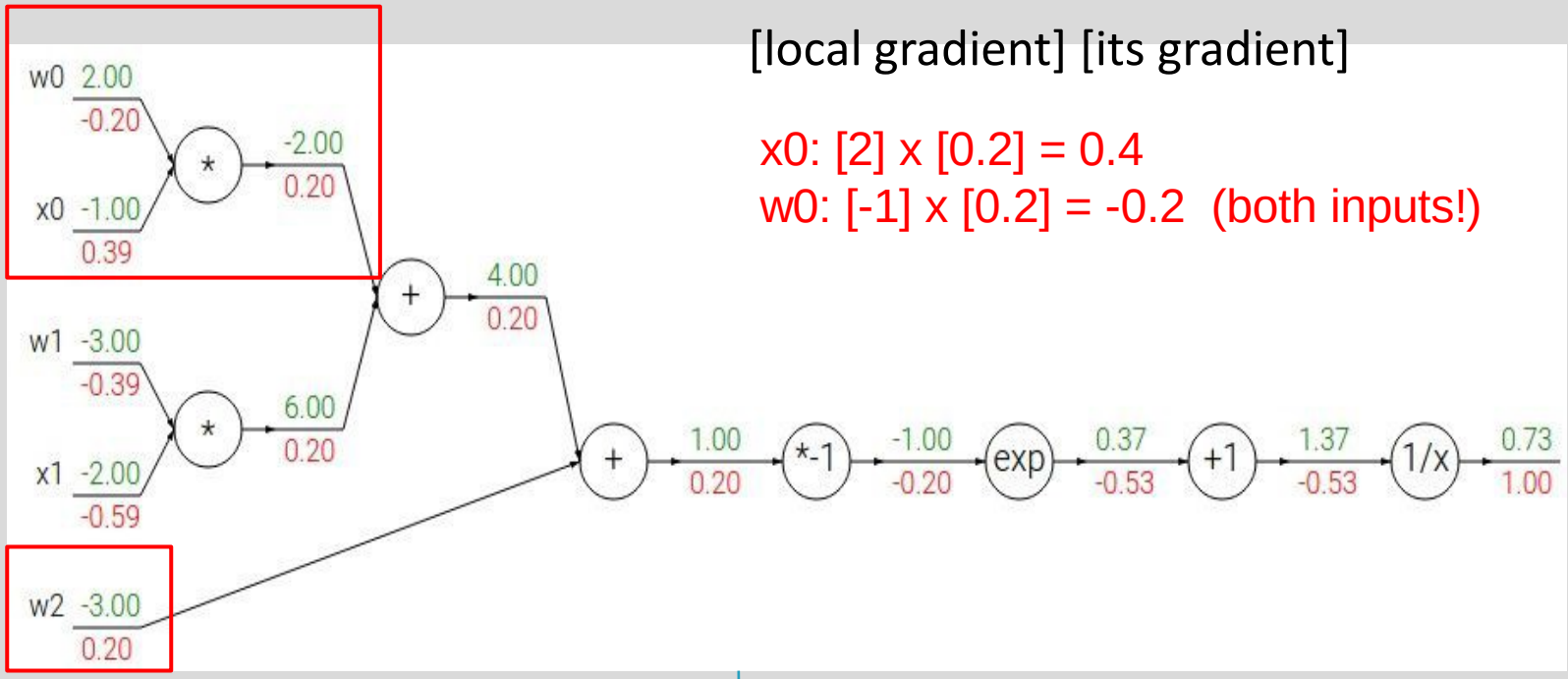
$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

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

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

Another example

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



[local gradient] [its gradient]

x0: [2] x [0.2] = 0.4
w0: [-1] x [0.2] = -0.2 (both inputs!)

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

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

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

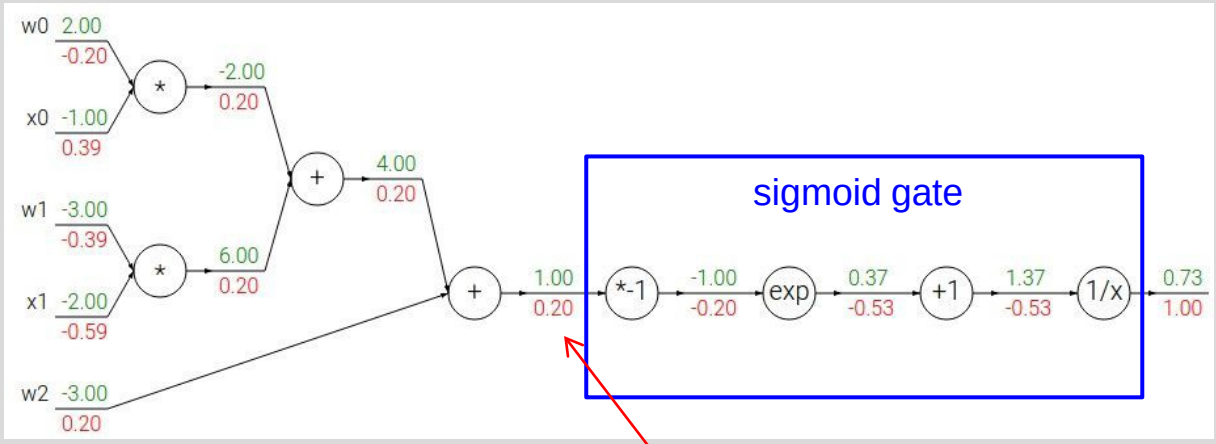
$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

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

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

sigmoid function

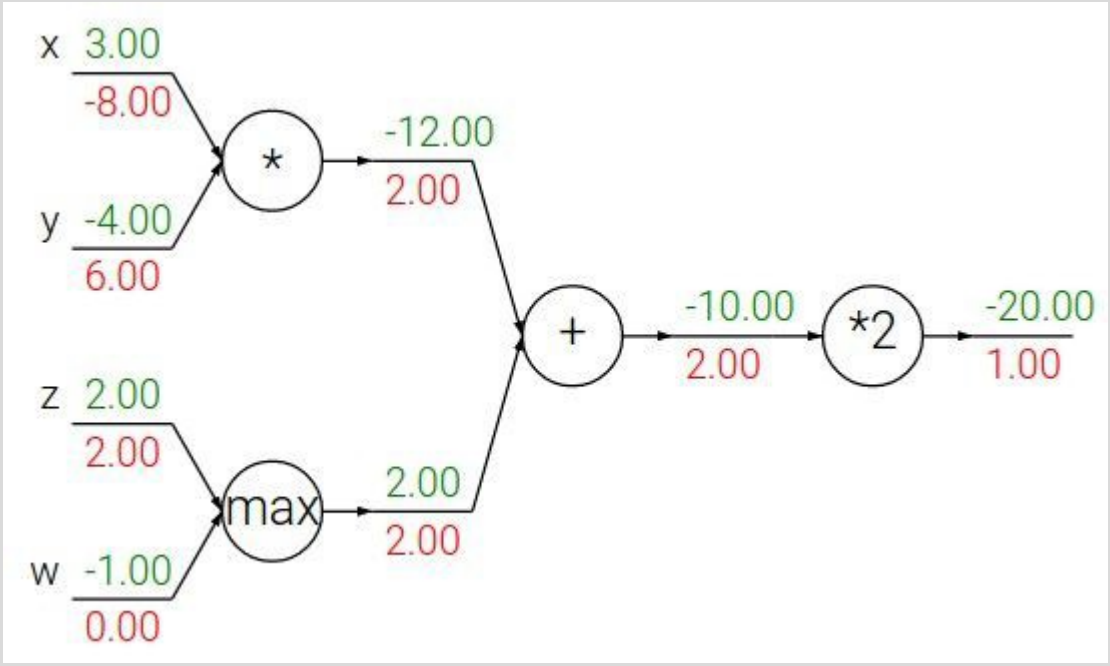
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



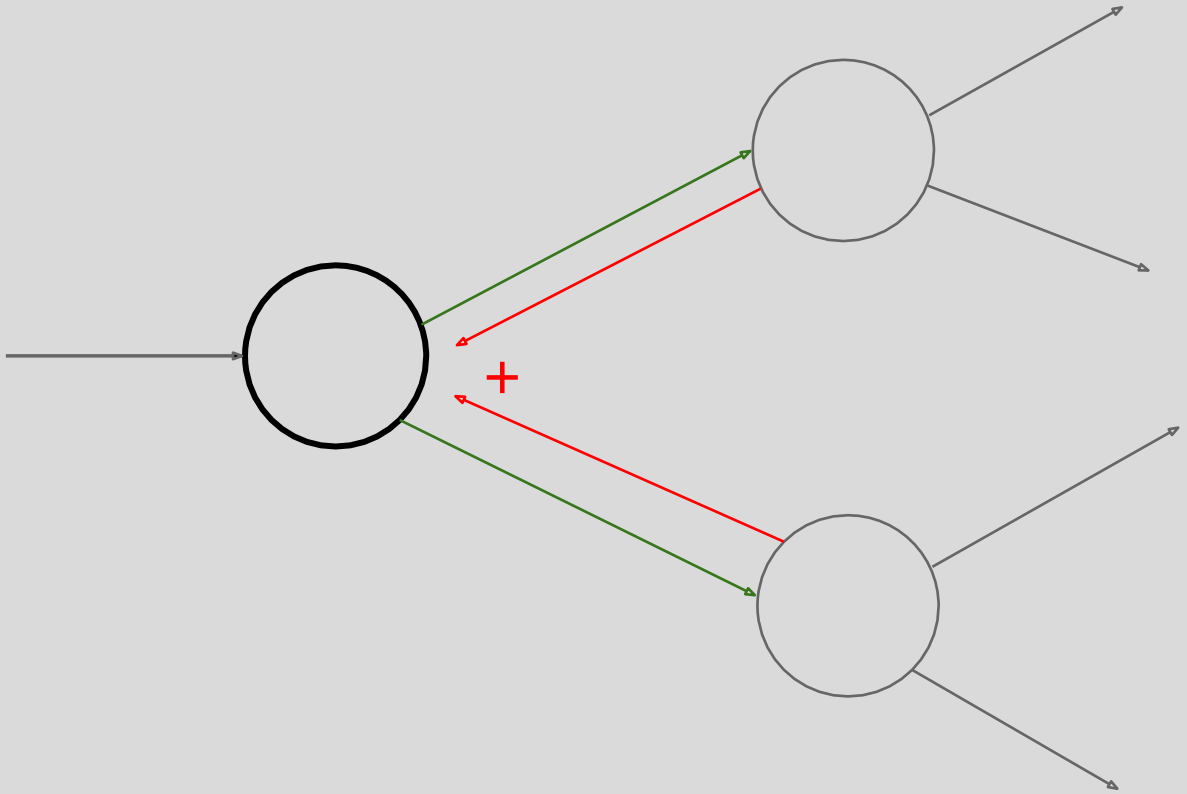
(0.73) * (1 - 0.73) = 0.2

Patterns in backward flow

add gate: gradient distributor
max gate: gradient router
mul gate: gradient… “switcher”?

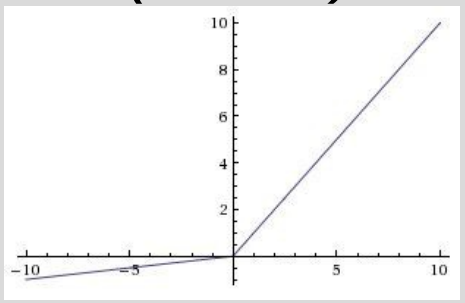


Gradients add at branches

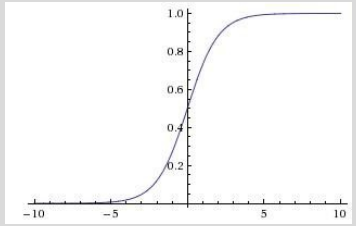


Activation functions

Leaky RELU
max(0.1x, x)



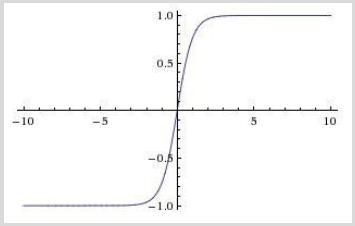
$$\sigma(x) = 1/(1 + e^{-x})$$



Maxout:

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

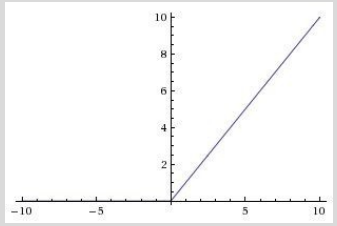
tanh: tanh(x)

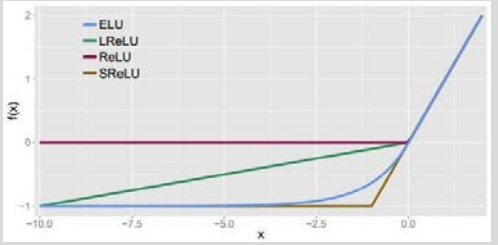


ELU:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

ReLU: max(0,x)





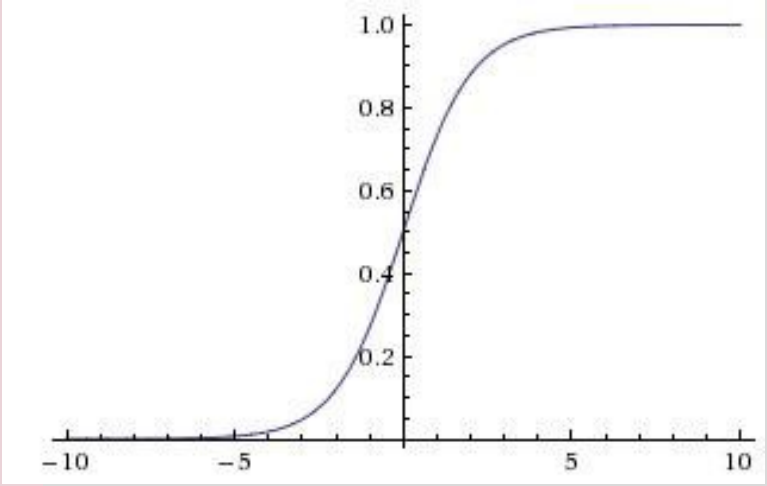
TRAINING NEURAL NETWORKS

Activation Function : Sigmoid

Squashes numbers to range [0,1]

- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

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



3 problems:

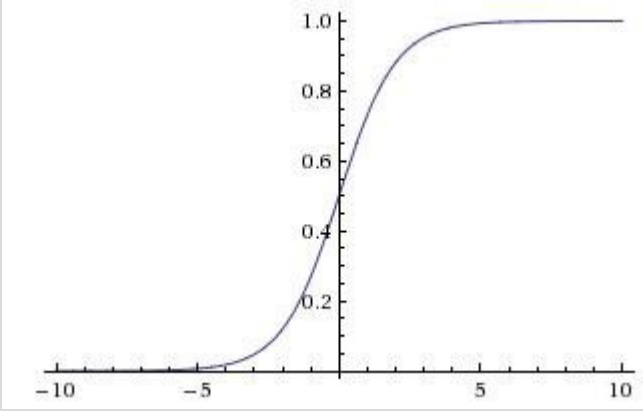
1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are non zero-centred
3. exp() is a bit compute expensive

Sigmoids saturate and kill gradients.

- when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero.
- During backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective.
- Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data.
- Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.

Activation function tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated

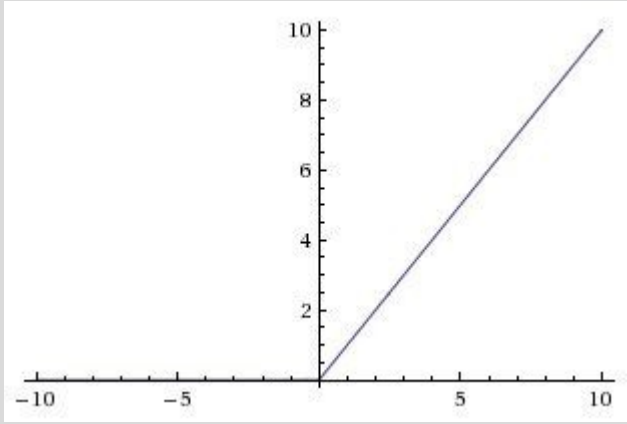


Activation function ReLU

- 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)



- **ReLU**

(Rectified Linear Unit)

Mini-batch SGD

Loop:

1. Sample a batch of data
2. Forward prop it through the graph, get loss
3. Backprop to calculate the gradients
4. Update the parameters using the gradient

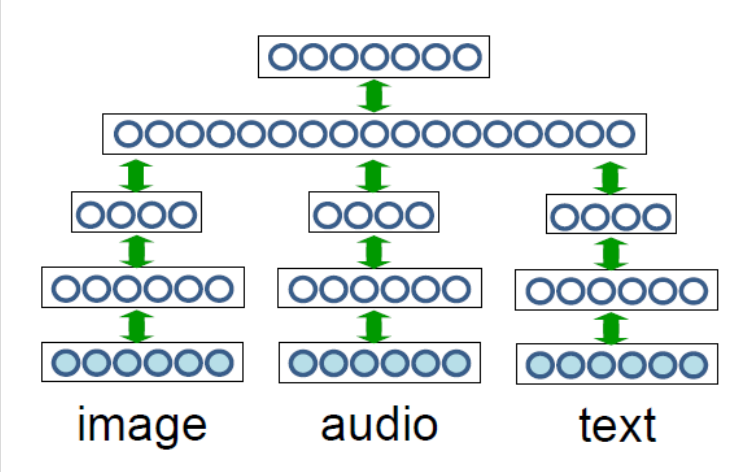
ACM summer school on NLP and ML

Neural Network

Deep Neural Network

Deep Learning

- Breakthrough results in
  - Image classification
  - Speech Recognition
  - Machine Translation
  - Multi-modal learning



Deep Neural Network

- Problem: training networks with many hidden layers doesn’t work very well
- Local minima, very slow training if initialize with zero weights.
- Diffusion of gradient.

Hierarchical Representation

- Hierarchical Representation help represent complex functions.
- NLP: character ->word -> Chunk -> Clause -> Sentence
- Image: pixel > edge -> texton -> motif -> part -> object
- Deep Learning: learning a hierarchy of internal representations
- Learned internal representation at the hidden layers (trainable feature extractor)
- Feature learning

Deep neural network

- Feed forward NN
- Stacked Autoencoders (multilayer neural net with target output = input)
- Stacked restricted Boltzmann machine
- Convolutional Neural Network
- Recurrent Neural Network

A Deep Architecture: Multi-Layer Perceptron

Output Layer

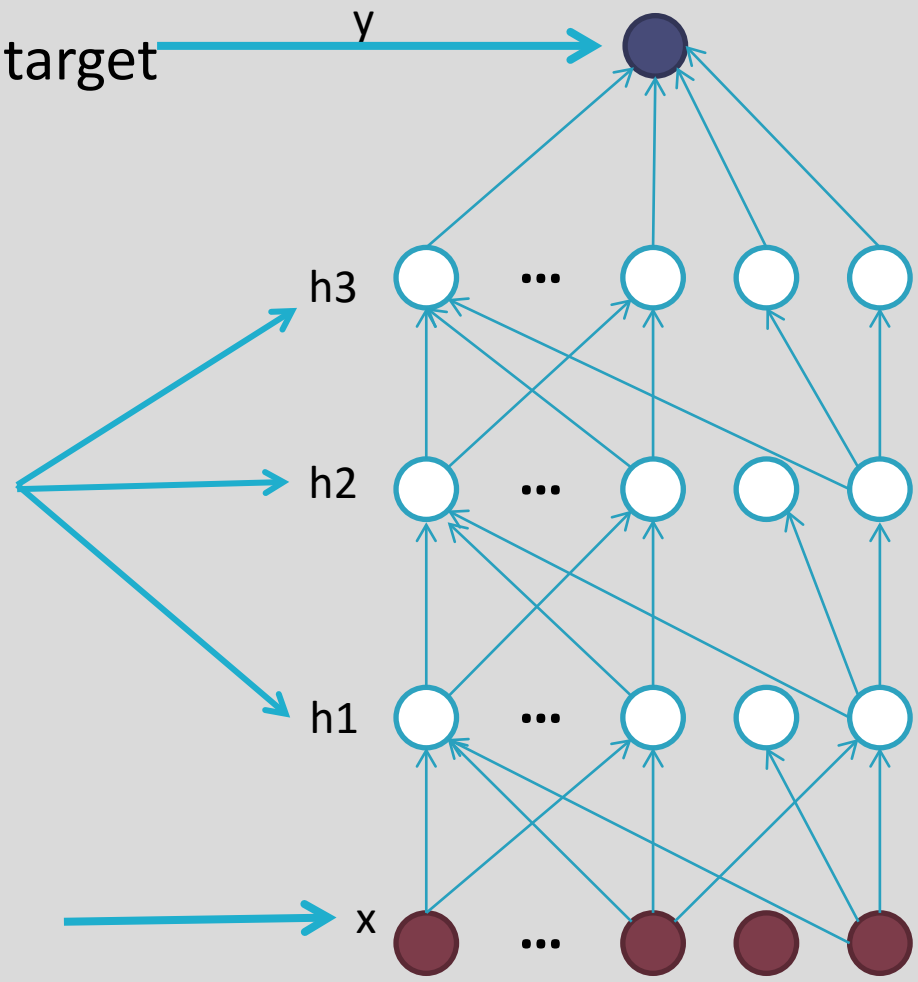
Here predicting a supervised target

Hidden layers

These learn more abstract representations as you head up

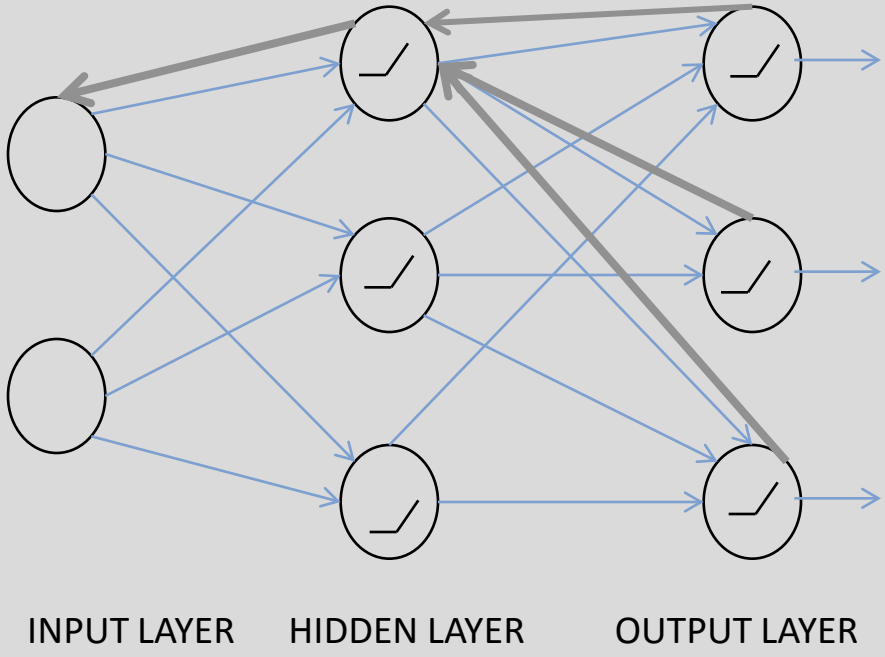
Input layer

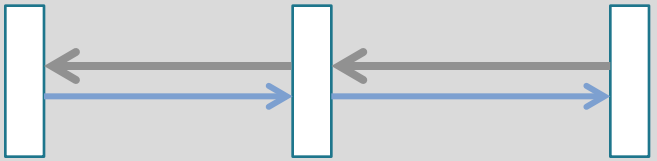
Raw sensory inputs



A Neural Network

- Training : Back Propagation of Error
  - Calculate total error at the top
  - Calculate contributions to error at each step going backwards
  - The weights are modified as the error is propagated



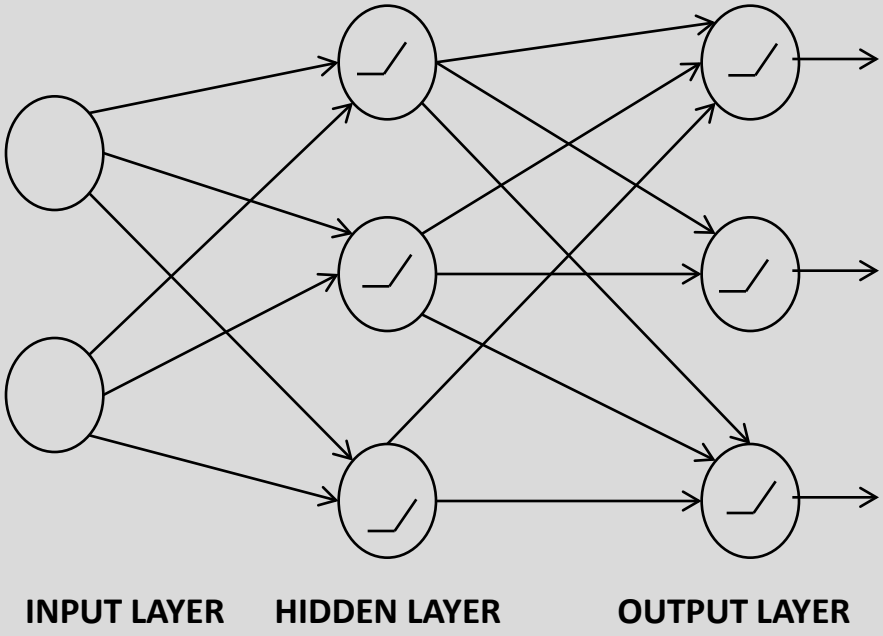


Training Deep Networks

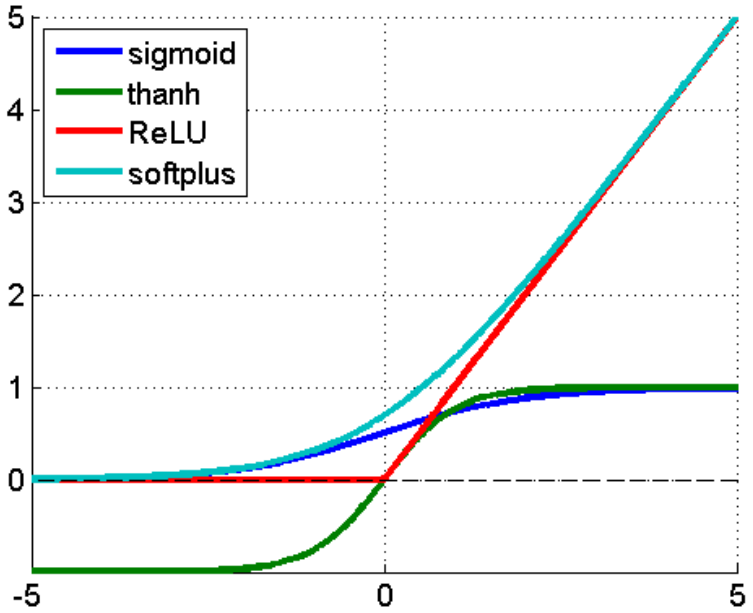
- Difficulties of supervised training of deep networks
  1. Early layers of MLP do not get trained well
     - Diffusion of Gradient – error attenuates as it propagates to earlier layers
     - Leads to very slow training
     - the error to earlier layers drops quickly as the top layers "mostly" solve the task
  2. Often not enough labeled data available while there may be lots of unlabeled data
  3. Deep networks tend to have more local minima problems than shallow networks during supervised training

Training of neural networks

- Forward Propagation :
  - Sum inputs, produce activation
  - feed-forward



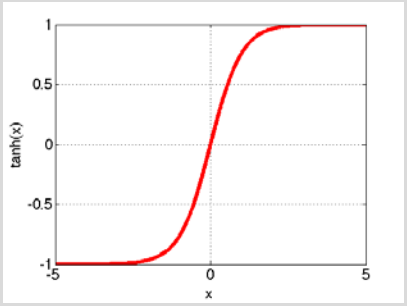
Activation Functions examples



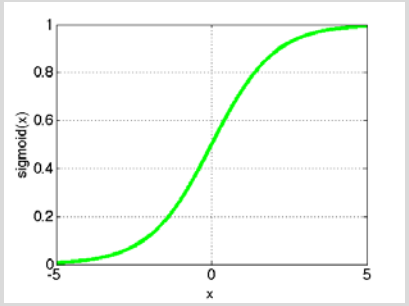
Activation Functions

Non-linearity

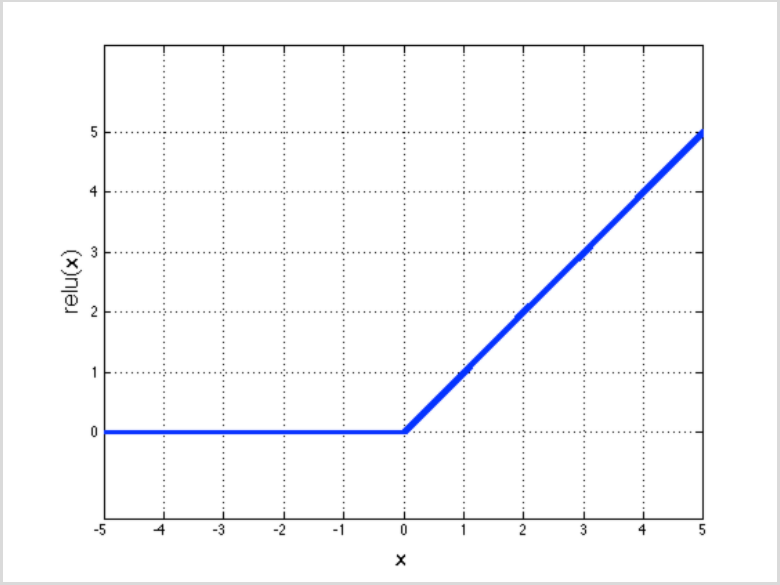
- $\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$



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



- Rectified linear
  relu(x) = max(0,x)



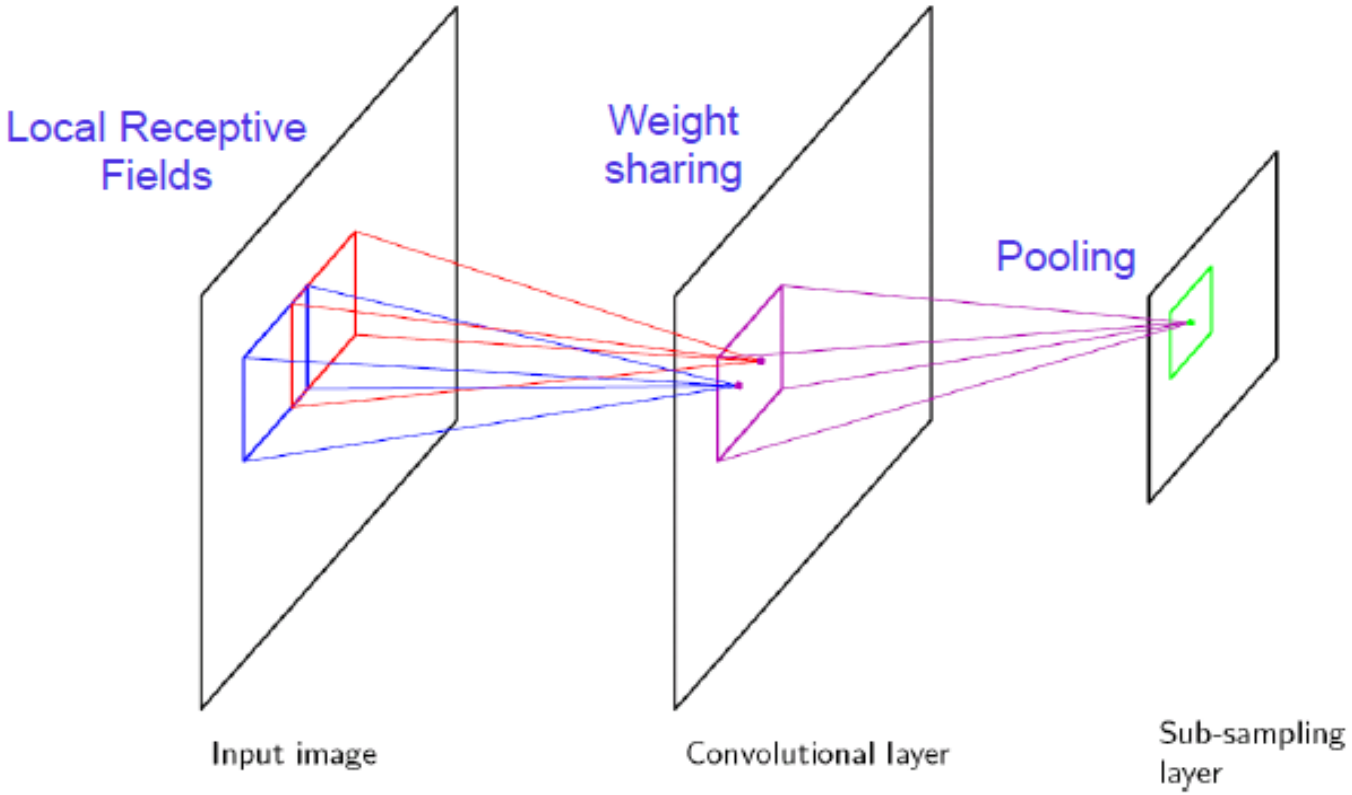
- Simplifies backprop
- Makes learning faster
- Make feature sparse

→ Preferred option

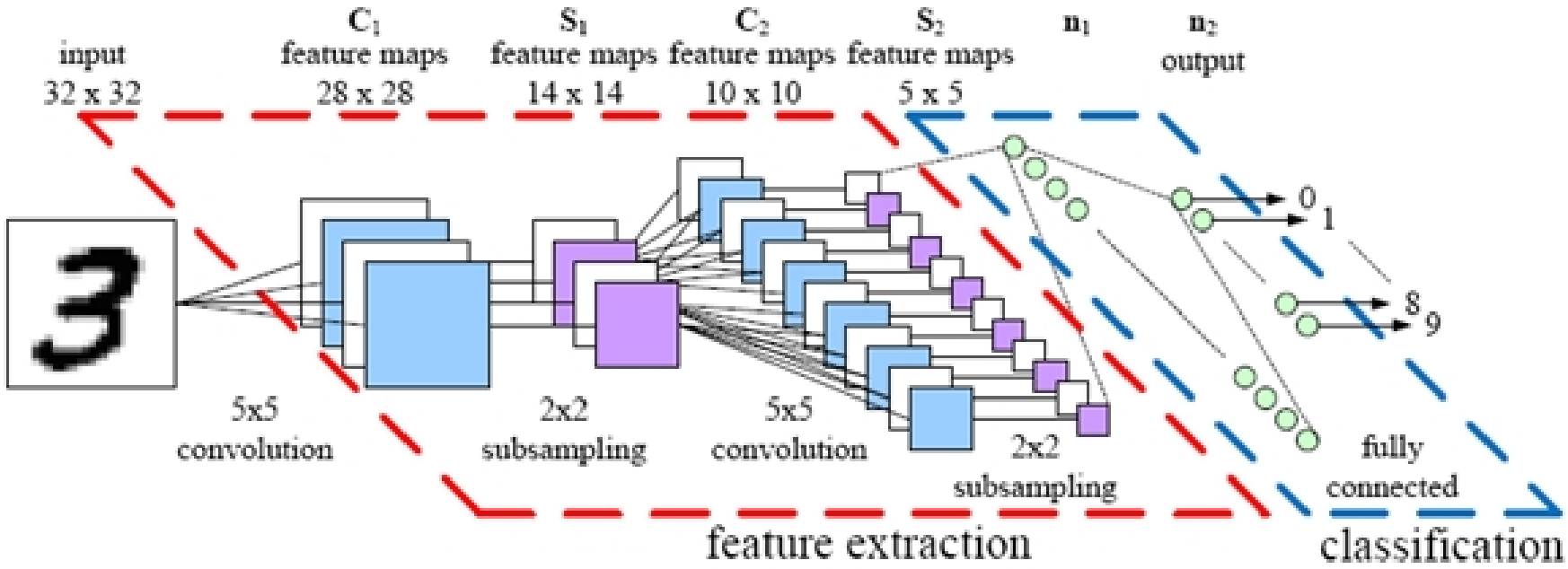
Convolutional Neural netwoks

- A CNN consists of a number of convolutional and subsampling layers.
- Input to a convolutional layer is a m x m x r image where m x m is the height and width of the image and r is the number of channels, e.g. an RGB image has r=3
- Convolutional layer will have k filters (or kernels)
- size n x n x q
- n is smaller than the dimension of the image and,
- q can either be the same as the number of channels r or smaller and may vary for each kernel

Convolutional Neural Networks

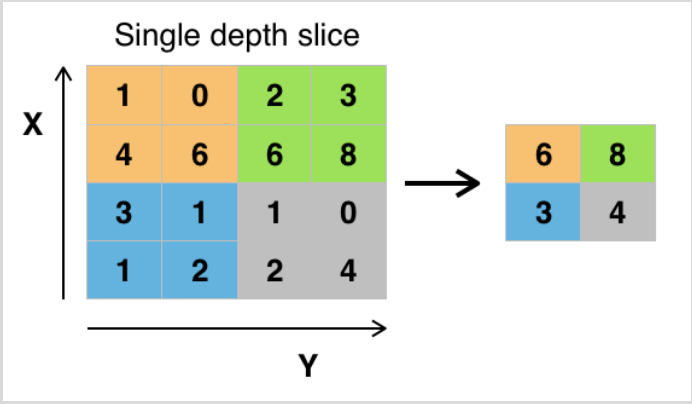


Convolutional layers consist of a rectangular grid of neurons
Each neuron takes inputs from a rectangular section of the previous layer
the weights for this rectangular section are the same for each neuron in the convolutional layer.



Pooling: Using features obtained after Convolution for Classification

The pooling layer takes small rectangular blocks from the convolutional layer and subsamples it to produce a single output from that block : max, average, etc.

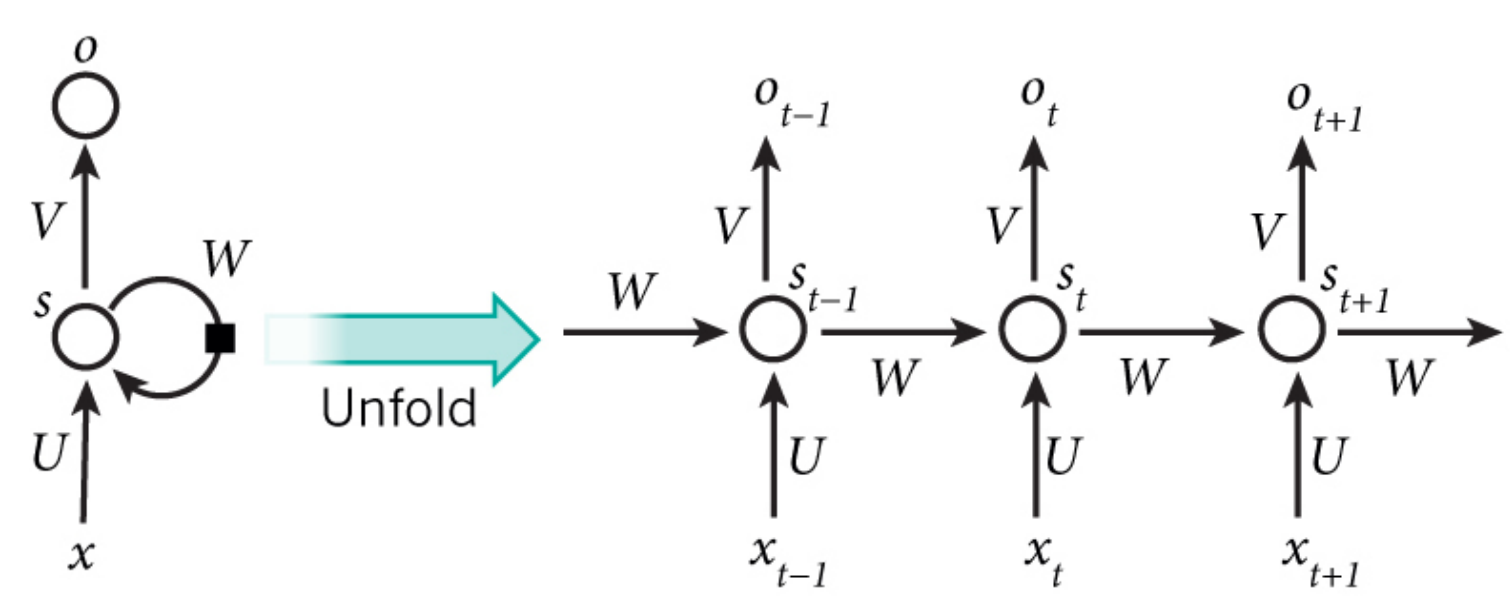


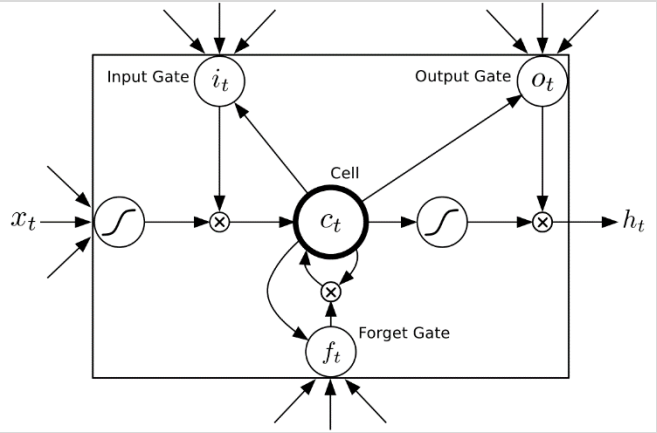
CNN properties

- CNN takes advantage of the sub-structure of the input
- Achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features.

- CNN are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units.

Recurrent Neural Network (RNN)





Thank You