Neural Networks and Deep Learning

Sanjay Arora
AI Center of Excellence
Red Hat

Ulrich Drepper
AI Center of Excellence
Red Hat

Red Hat

Why the hype?

- Very flexible learners – can learn very complicated functions.
- Spectacular success in vision and language tasks but not limited just to these domains.
- Need minimal feature engineering.

Why the hype?

- Very flexible learners – can learn very complicated functions.
- Spectacular success in vision and language tasks but not limited just to these domains.
- Need minimal feature engineering.

  BUT these techniques have been around for decades.
  What changed?

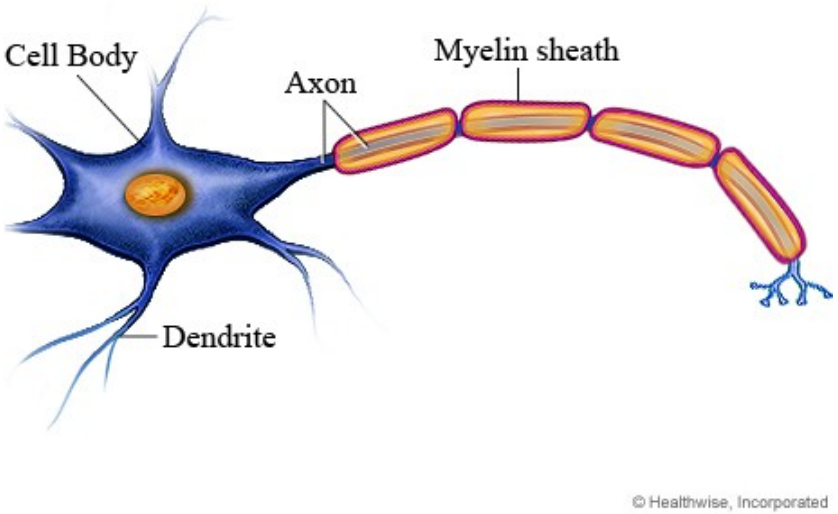
Why the hype?

- Very flexible learners – can learn very complicated functions.
- Spectacular success in vision and language tasks but not limited just to these domains.
- Need minimal feature engineering.

  BUT these techniques have been around for decades.
  What changed?

- Access to large amounts of labeled data.
- Accelerated training on GPUs.

What is a neural network?



Loosely inspired by neurons in the brain

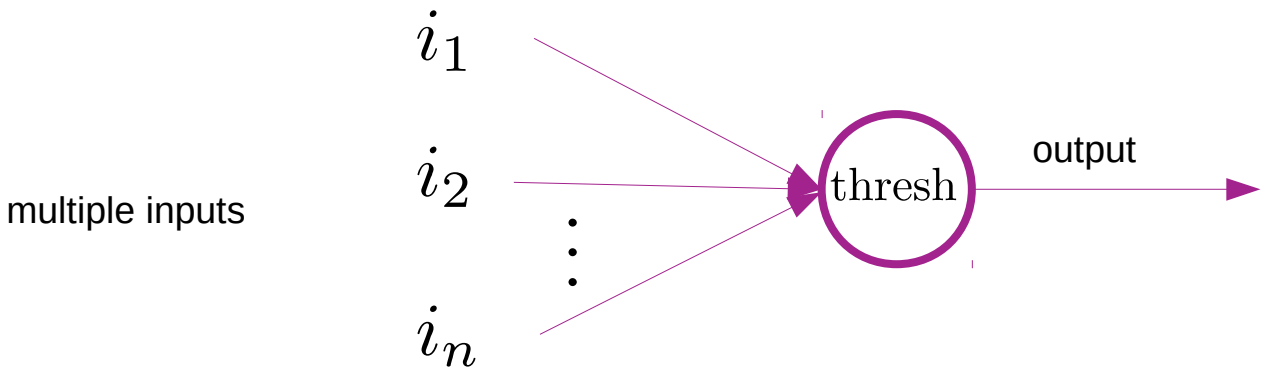
Red Hat

What is a neural network?

$$\text{input} > \text{threshold} \rightarrow \text{output} = 1$$

$$\text{input} <= \text{threshold} \rightarrow \text{output} = 0$$

What is a neural network?



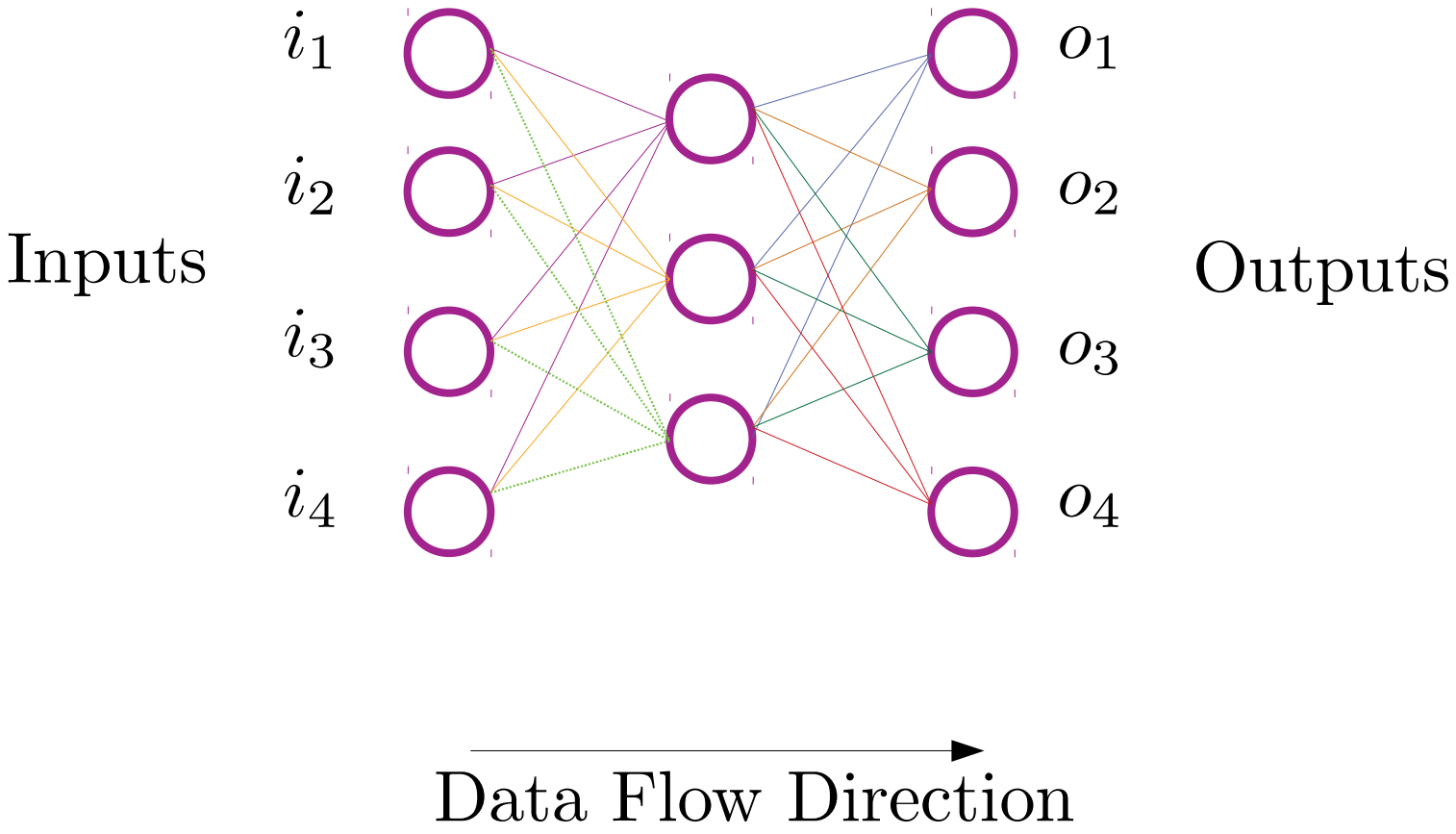
$$\text{input}, i = \underbrace{w_1}_{\text{weight}} i_1 + \underbrace{w_2}_{\text{weight}} i_2 + \ldots + \underbrace{w_n}_{\text{weight}} i_n$$

$$\text{input} > \text{threshold} \rightarrow \text{output} = 1$$

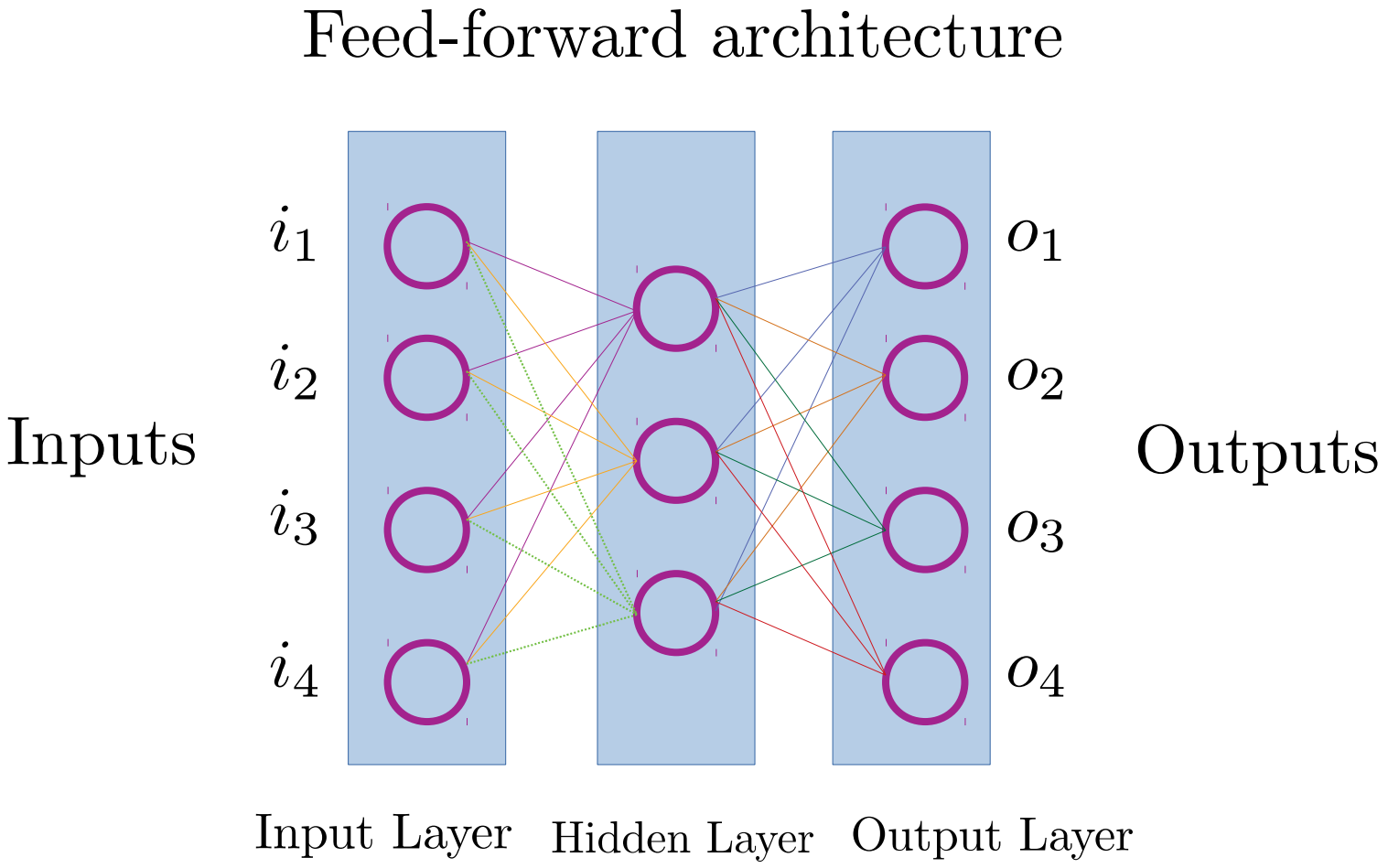
$$\text{input} <= \text{threshold} \rightarrow \text{output} = 0$$

What is a neural network?

Combine basic neurons into a network



What is a neural network?



Why go through all this trouble?

Why is any of this useful?

Can’t I just use linear regression please?

Or random forests?

Why go through all this trouble: Universal Approximation Theorem

Loose version

A (feed-forward) neural network with one hidden layer can approximate any “reasonable” function, f, to arbitrary accuracy

Why go through all this trouble: Universal Approximation Theorem

Precise version

"Activation" function	$\sigma : \mathbb{R} \to \mathbb{R}$ σ non-constant, bounded, continuous
Function to learn	$f : [0,1]^n \to \mathbb{R}$ f continuous on $[0,1]^n$

Why go through all this trouble: Universal Approximation Theorem

Precise version

$$\sigma : \mathbb{R} \to \mathbb{R} \qquad\qquad f : [0,1]^n \to \mathbb{R}$$

$$\text{can find } w_i, u_i, b_i, N$$

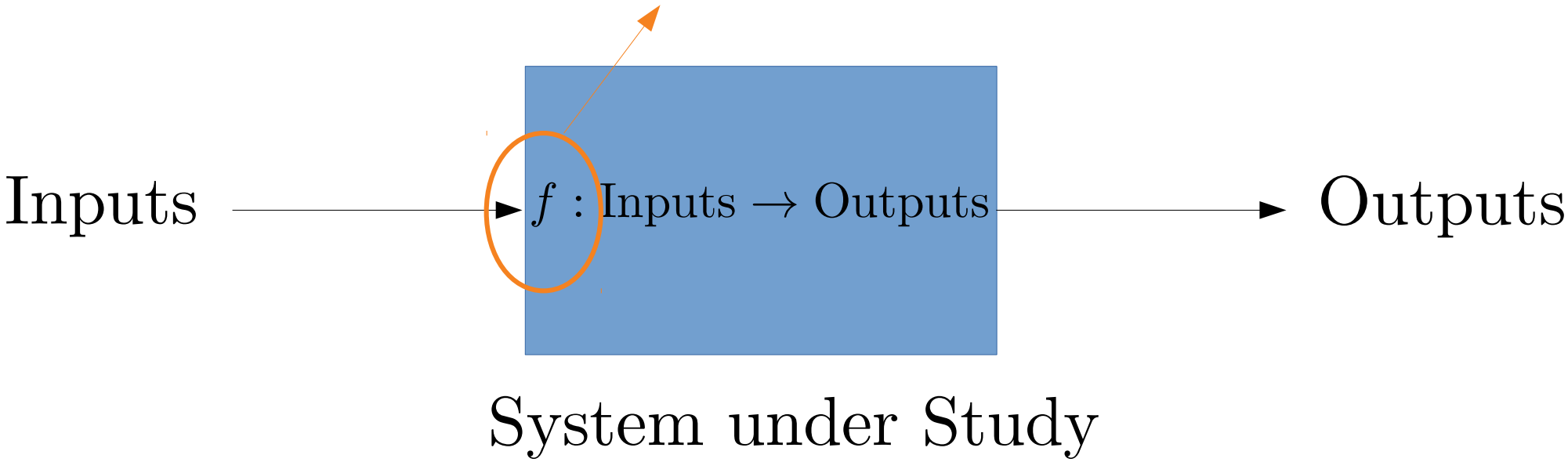
$$| \quad \underbrace{f(x)}_{\text{function to learn}} \quad - \underbrace{\Sigma_{i=1}^{N} w_i \sigma(u_i^T x + b_i)}_{\text{approximation to } f} | < \epsilon$$

Important Technicality: can extend to any compact subset for the domain

Why go through all this trouble: Universal Approximation Theorem

What does this mean for me?

Can approximate to arbitrarily high accuracy with a neural network



Why go through all this trouble: Universal Approximation Theorem

Two caveats

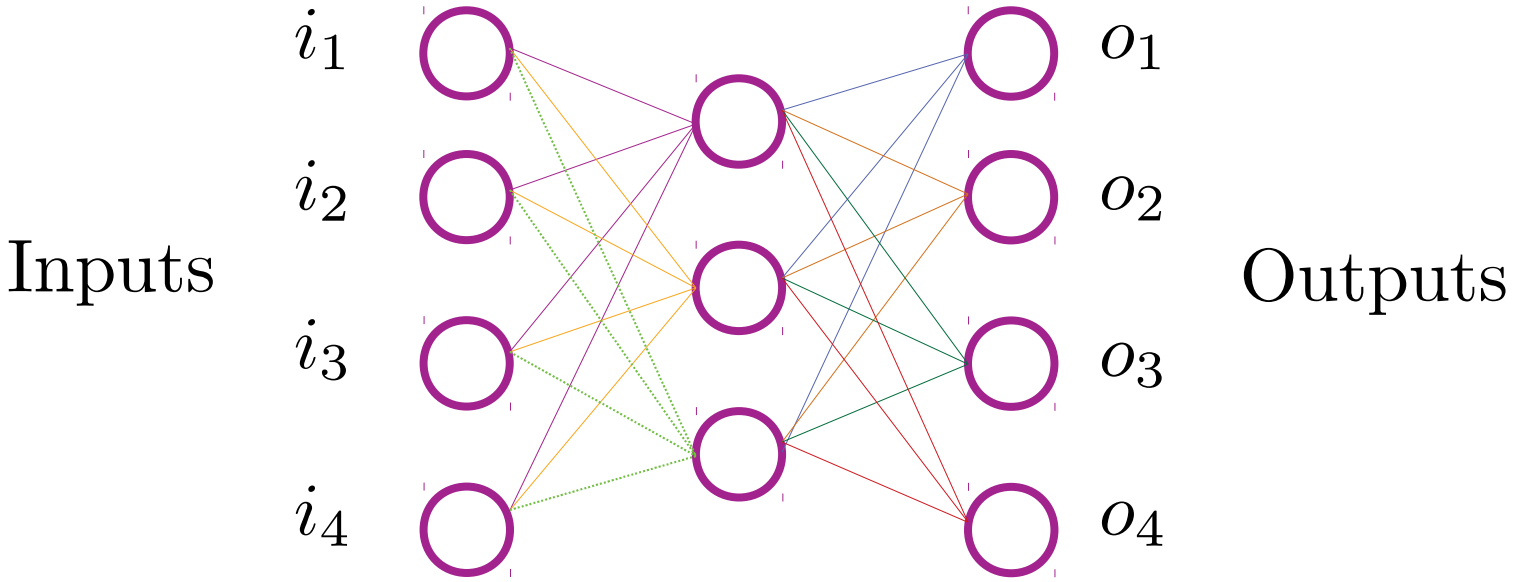
Need non-linear activation function, ϕ in our neural network (we'll see what this means shortly)

N refers to the number of neurons (nodes) in the hidden layer. This grows exponentially as ϵ decreases

$$| \underbrace{f(x)}_{\text{function to learn}} - \underbrace{\Sigma_{i=1}^{N} w_i \sigma(u_i^T x + b_i)}_{\text{approximation to } f} | < \epsilon$$

Two Caveats

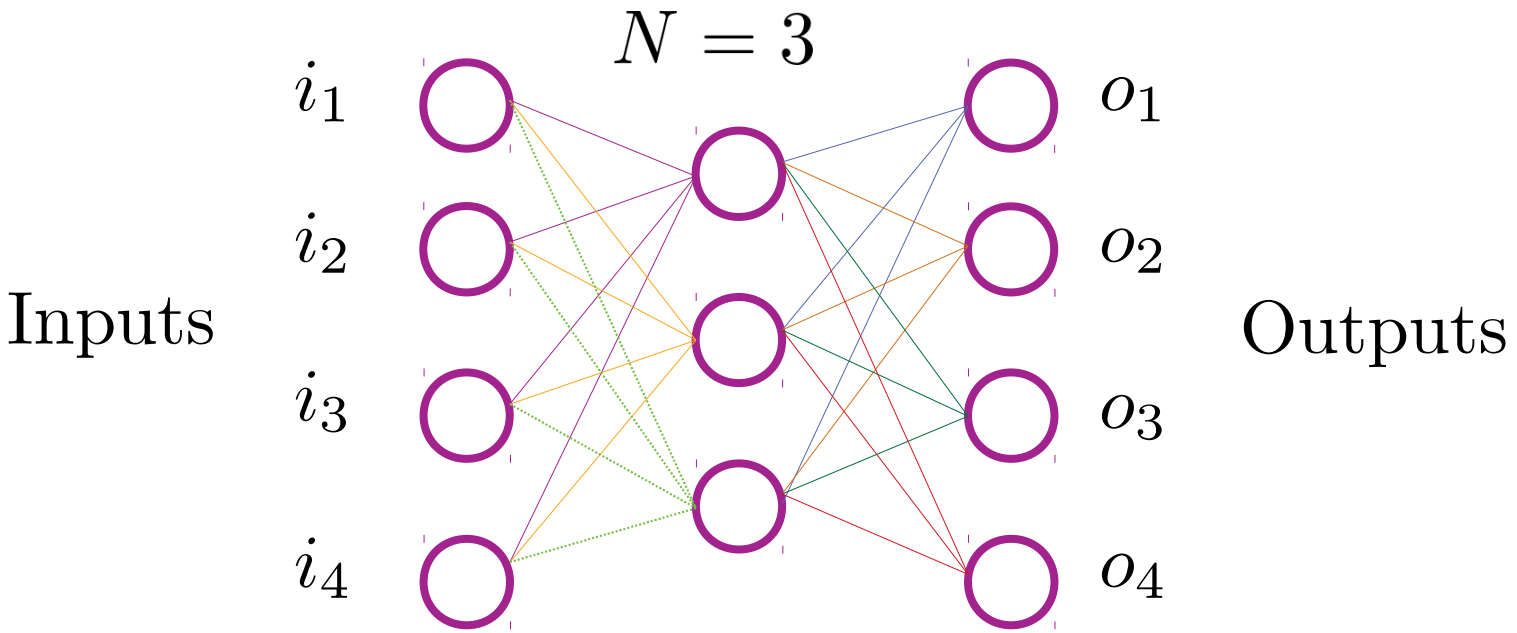
Need non-linear activation function, σ in our neural network (we'll see what this means shortly)



Two Caveats

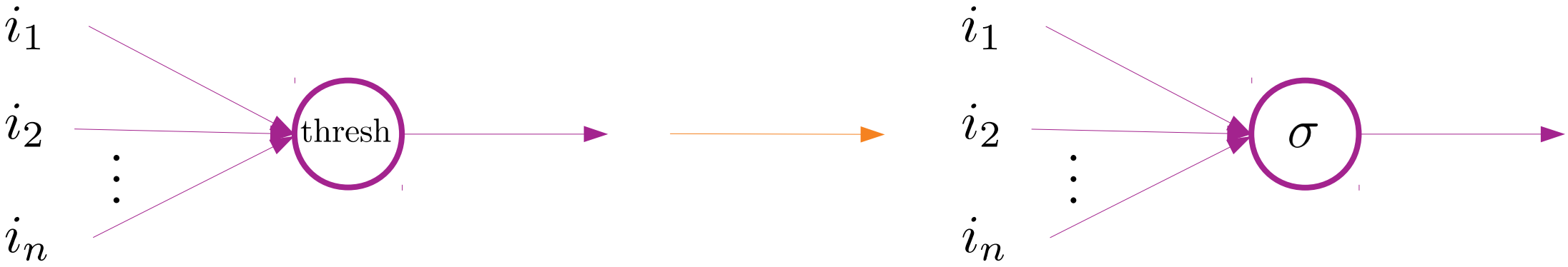
N refers to the number of neurons (nodes) in the hidden layer. This grows exponentially as ϵ decreases

$$| \underbrace{f(x)}_{\text{function to learn}} - \underbrace{\Sigma_{i=1}^{N} w_i \sigma(u_i^T x + b_i)}_{\text{approximation to } f} | < \epsilon$$



Red Hat

Structure of a node (neuron)



$$\text{thresh}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

σ can be one of many non-linear activation functions

Linear vs Non-linear

$$\text{Linear: } \sigma(x) = C_1 x + C_2$$

Applying one function after the other

Composition of two linear functions is linear

$$f(x) = C_1 x + C_2 \qquad g(x) = D_1 x + D_2$$

$$g(f(x)) = D_1 f(x) + D_2 = D_1(C_1 x + C_2) + D_2$$

$$g(f(x)) = \underbrace{(D_1 C_1)}_{E_1} x + \underbrace{(D_1 C_2 + D_2)}_{E_2}$$

Linear vs Non-linear

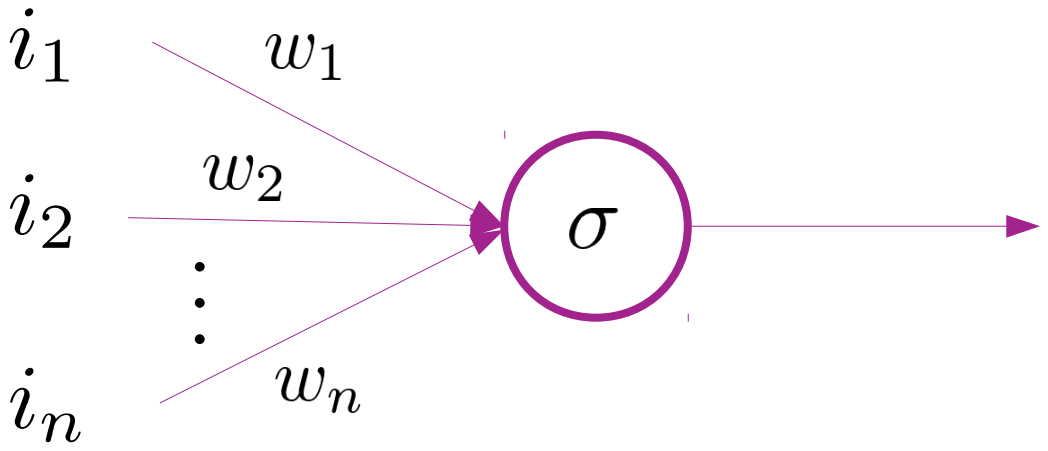
Non-linear

$\log(x)$ e^x x^2 $\sqrt{x}$

infinitely more

Most complex systems behave non-linearly

Structure of a node (neuron)



$$o = \sigma(w_1 i_1 + w_2 i_2 + \ldots w_n i_n + b)$$

Short-hand: $o = \sigma(w^T i + b)$

$$i = \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad w^T = \begin{bmatrix} w_1 & w_2 & \ldots w_n \end{bmatrix}$$

b is called bias or intercept

Activation Functions: Requirements

Non-linear

Simple to evaluate

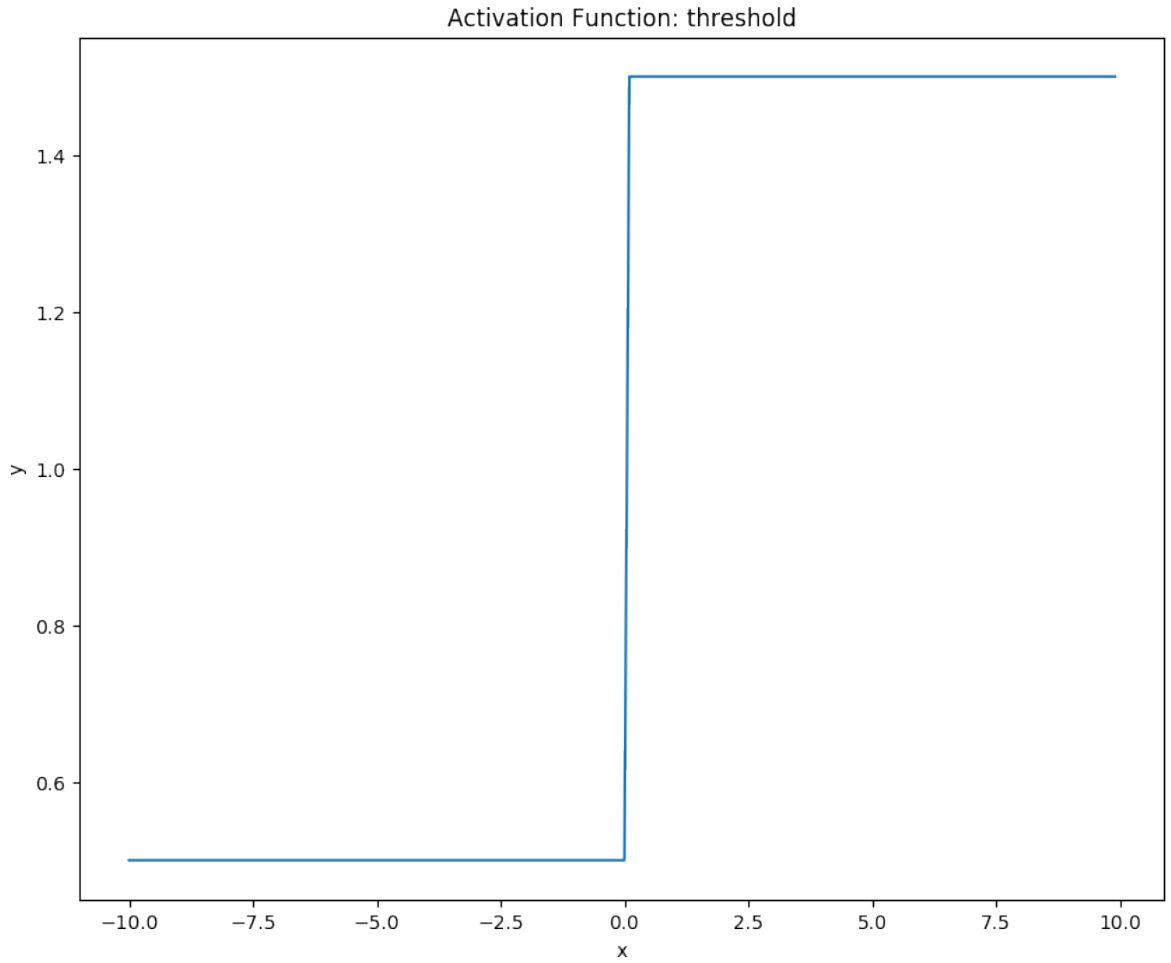
Will need derivatives so hopefully derivatives easy to evaluate

Binary switch

Derivative not defined at $x = 0$
(not a major issue)

Derivative not informative

$\sigma'(x) = 0$ everywhere except at x=0



Red Hat

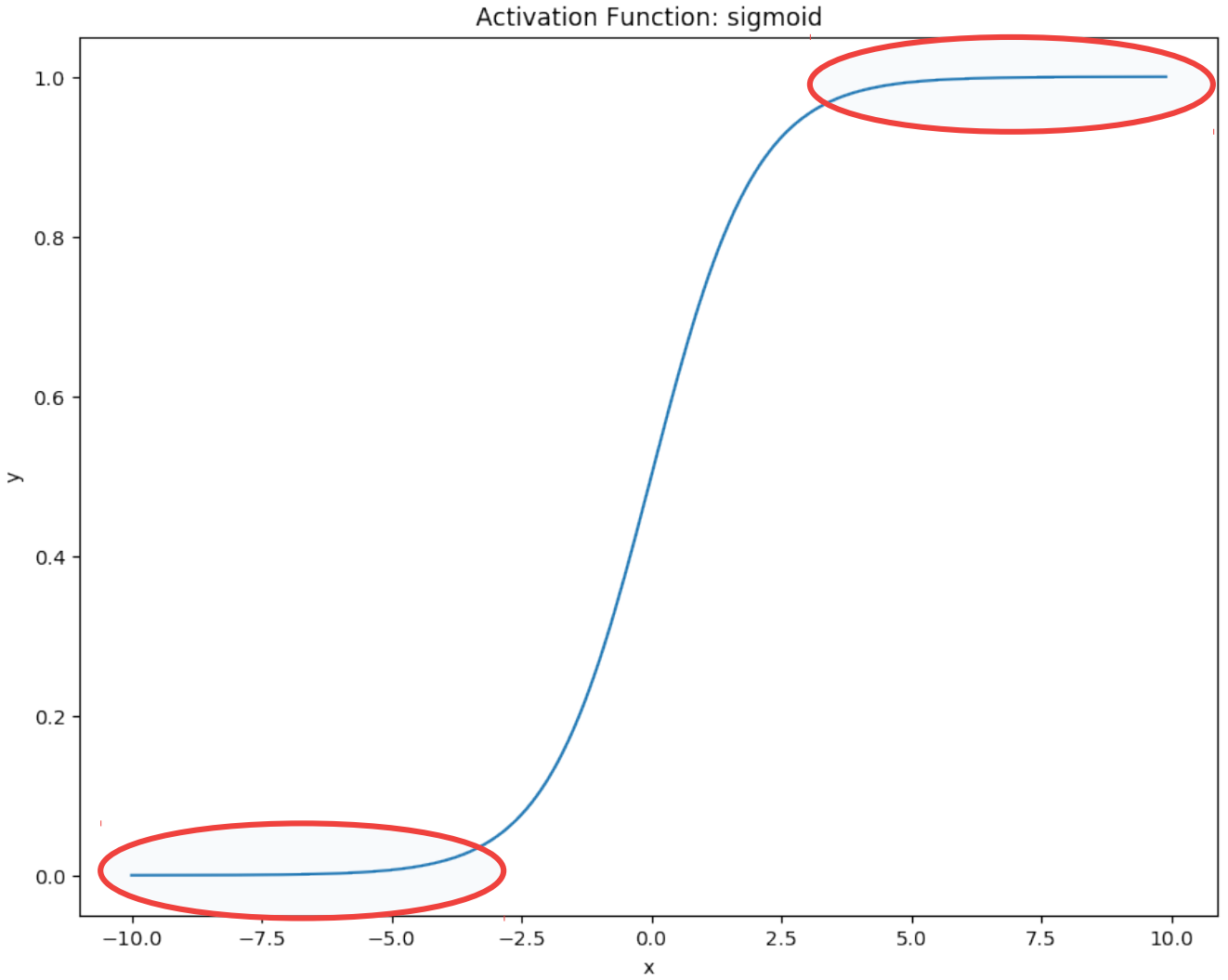
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Continuous switch

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

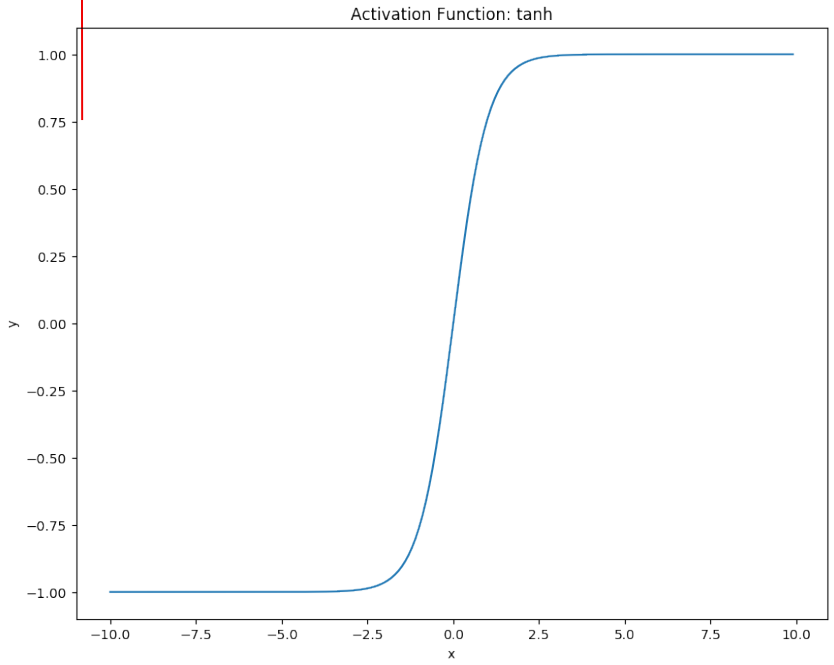
Derivatives in terms of activation itself

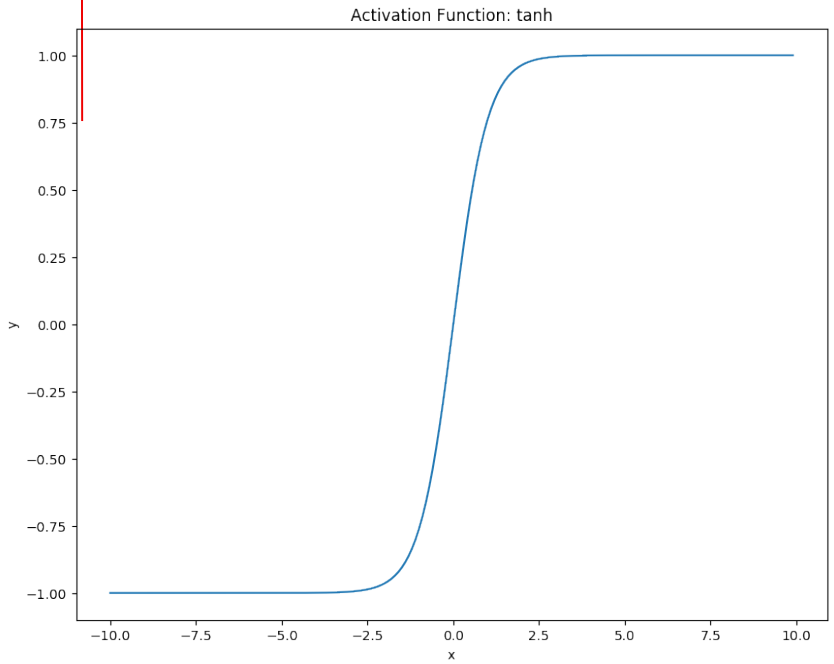
Positive x → 1 (on)

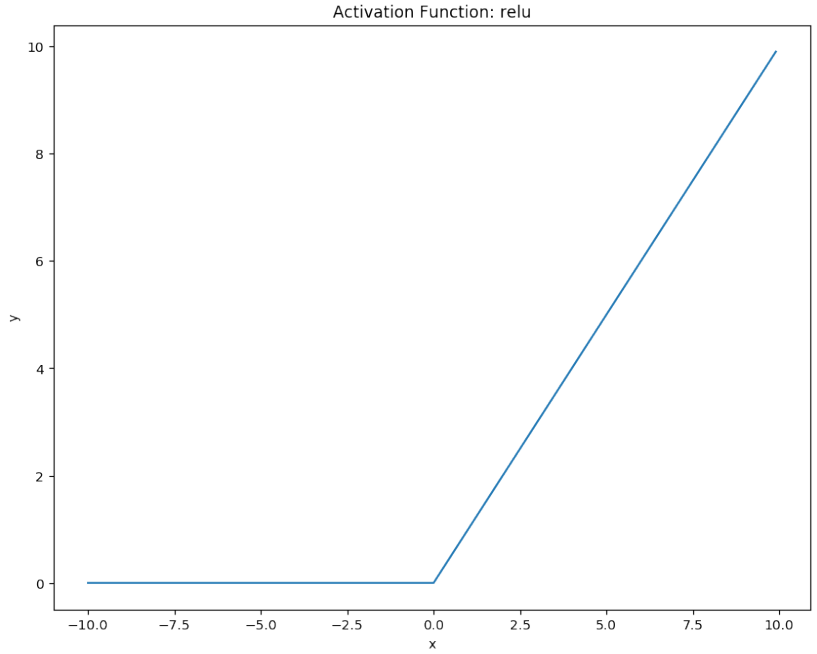


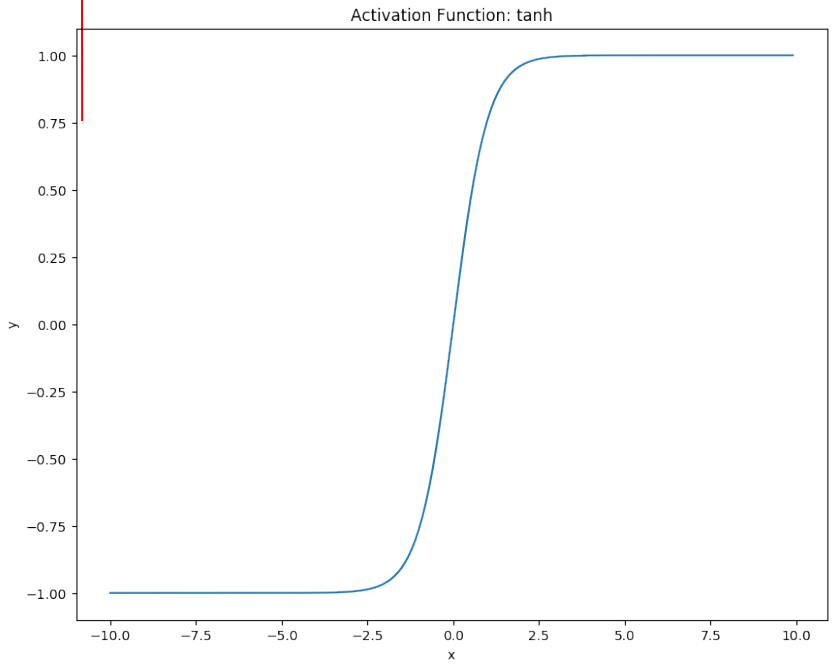
Negative x → 0 (off)

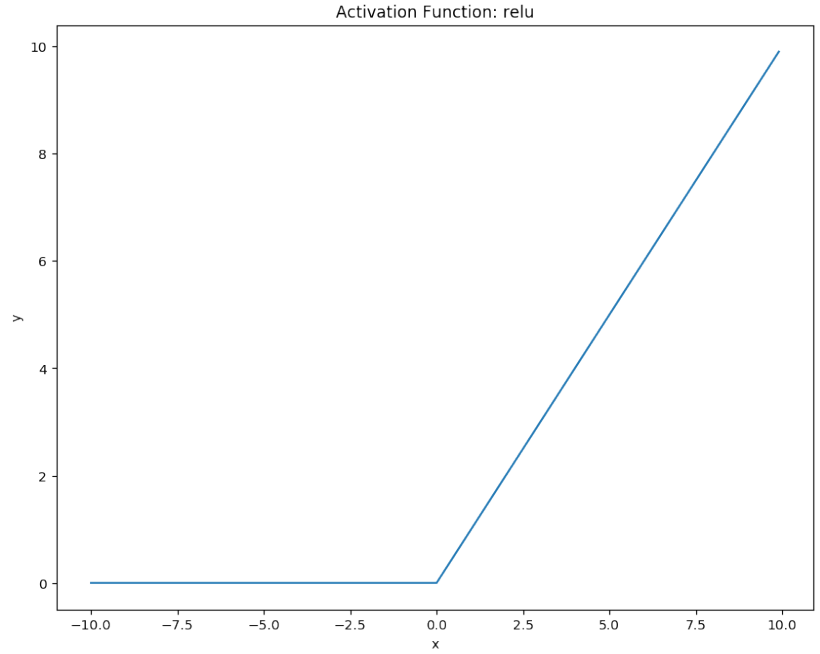
Red Hat

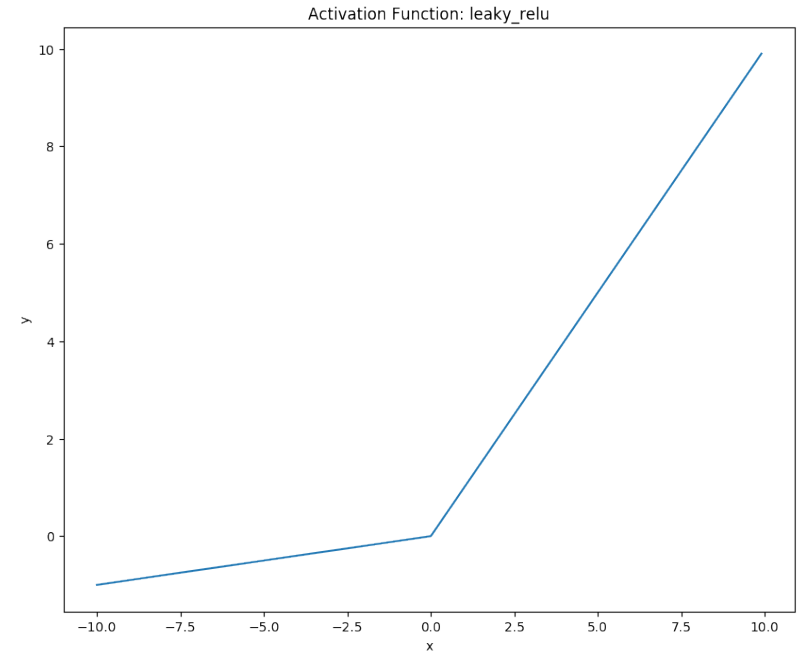

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

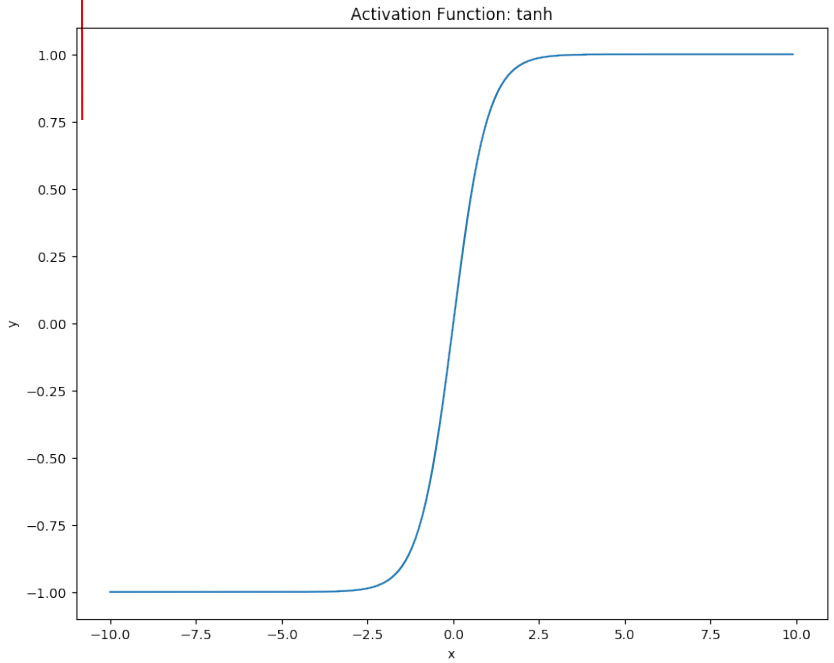

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

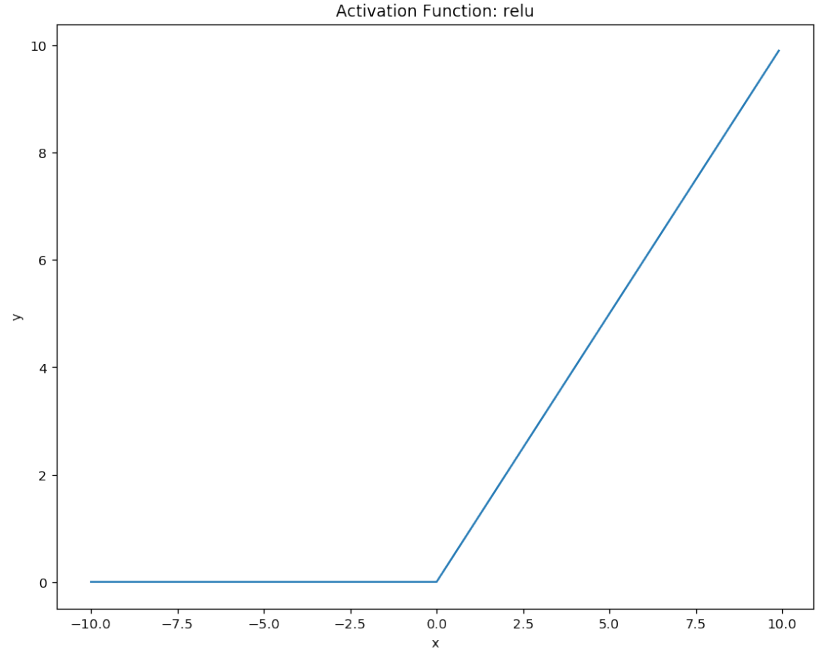

Activation Function: relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

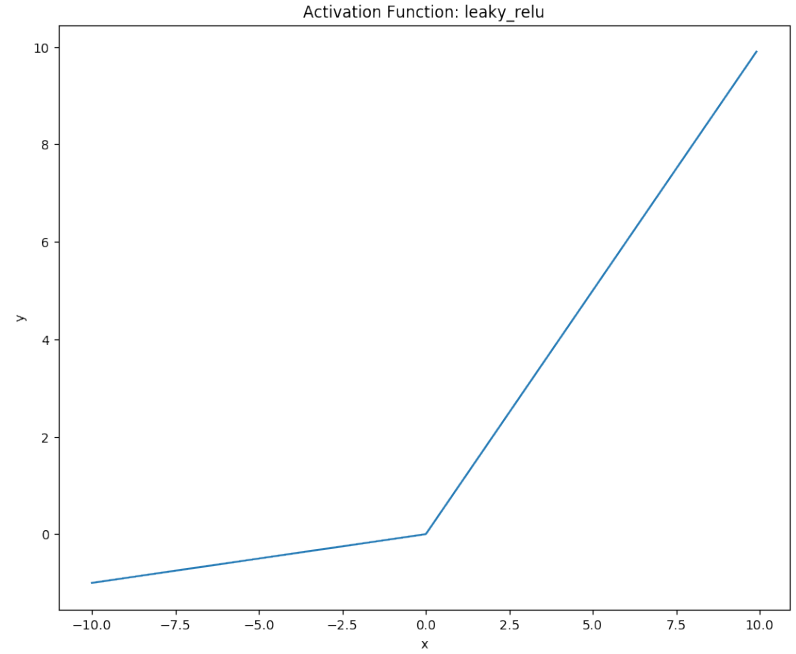

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

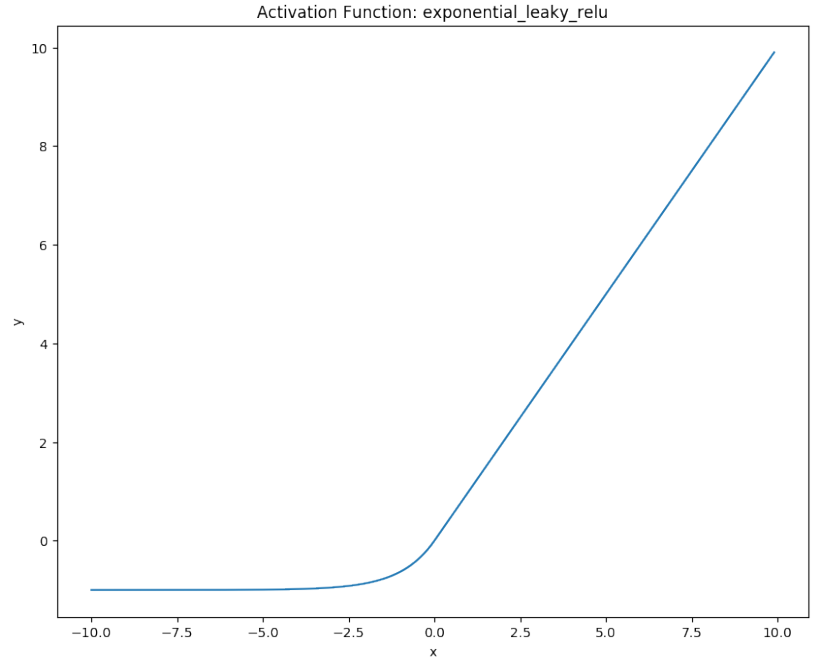

Activation Function: relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x


Activation Function: leaky_relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

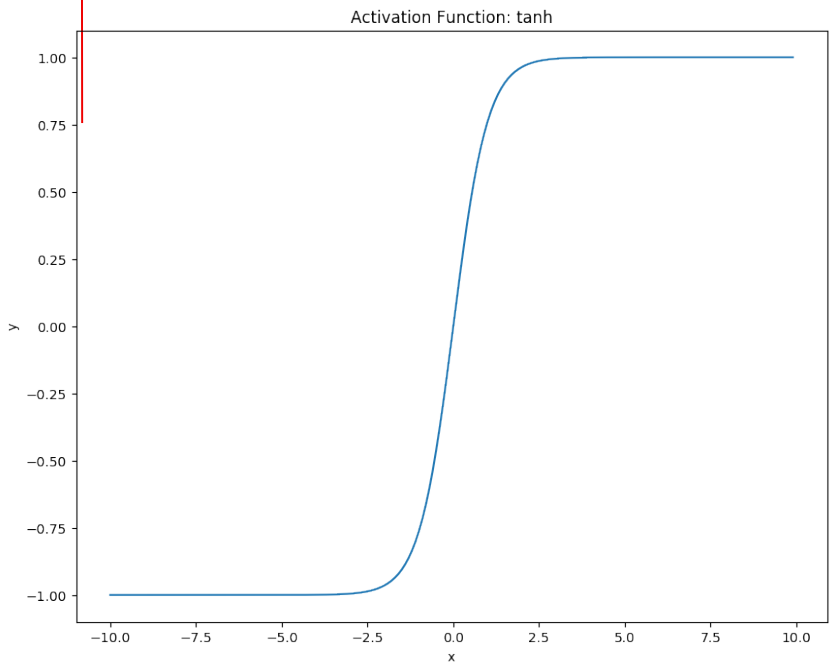

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

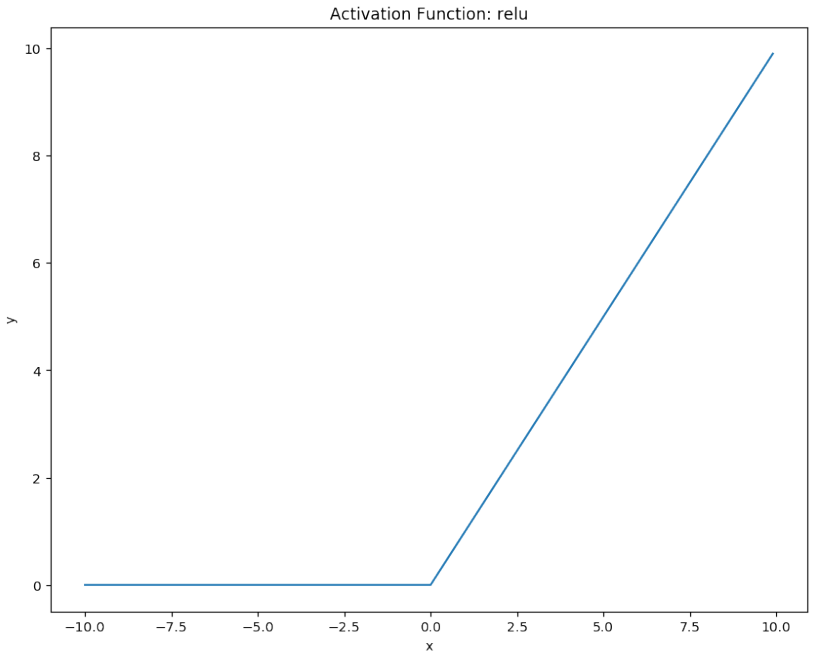

Activation Function: relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

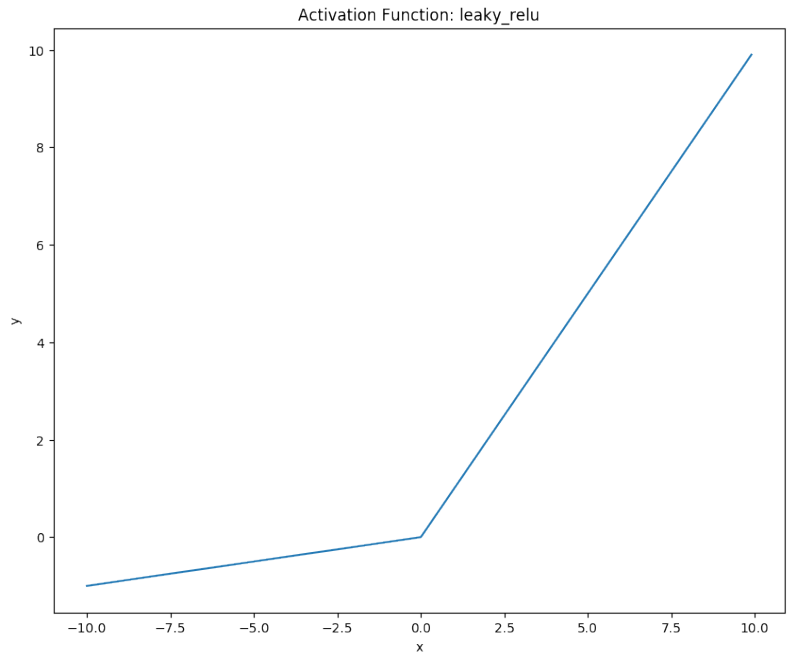

Activation Function: leaky_relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

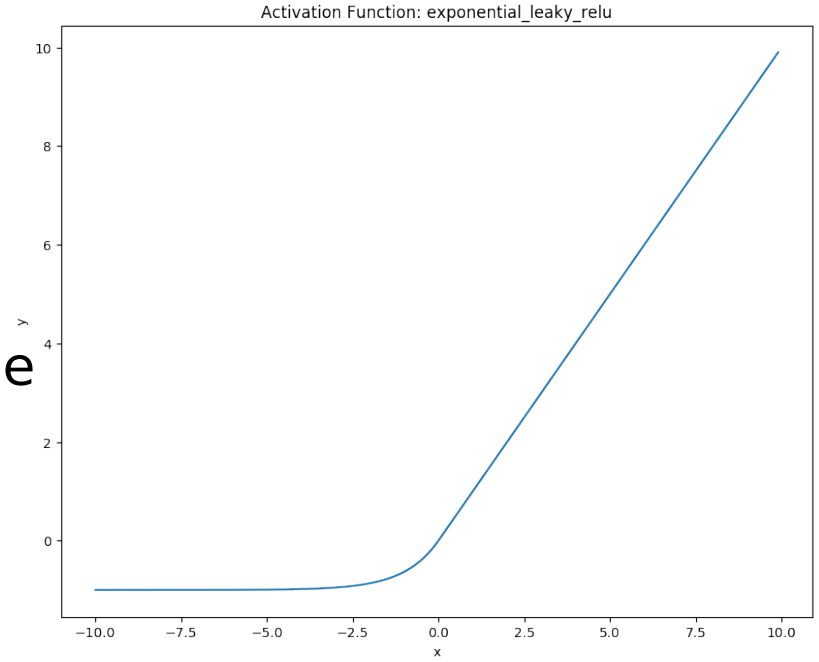

Activation Function: exponential_leaky_relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

Red Hat









Many more choices

Choices dictated by:

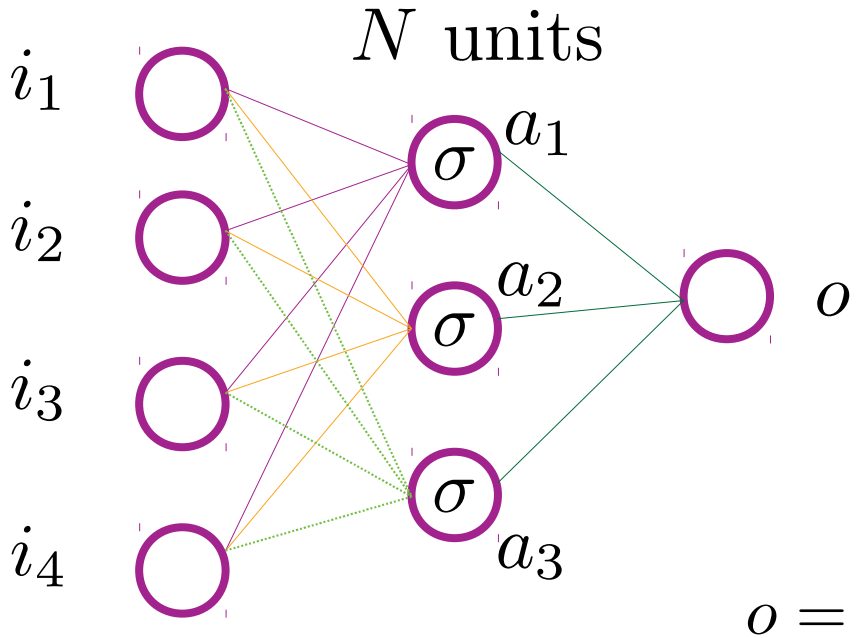
- Issues during training time (saturation of nodes)
- Type of predictions

Red Hat

Recap

- Start with simple neuron
- Need to add non-linear activation to be able to learn/mimic any reasonable function
- Looked at a bunch of activations and added to our neuron
- Time to put neurons together into a network and see how prediction would work.

Full Neural Network



$$\text{Input:} \quad : (i_1, i_2, i_3, i_4)$$

Hidden:

$$a_1 = \sigma(w_1^T i + b_1)$$

$$a_2 = \sigma(w_2^T i + b_2)$$

$$a_3 = \sigma(w_3^T i + b_3)$$

Output:

$$o = v_1 a_1 + v_2 a_2 + v_3 a_3 + c = v^T a + c$$

Every edge has a “weight”

Full Neural Network

Input: $: (i_1, i_2, i_3, i_4)$

Hidden:

$$a_1 = \sigma(w_1^T i + b_1)$$

$$a_2 = \sigma(w_2^T i + b_2)$$

$$a_3 = \sigma(w_3^T i + b_3)$$

Output:

$$o = v_1 a_1 + v_2 a_2 + v_3 a_3 + c = v^T a + c$$

Concise matrix notation

$$o = V\sigma(Wi + b) + c$$

W, V are matrices of weights

b, c are vectors of biases

Imagine implementing this using a matrix library, say numpy in python.

Full Training Cycle

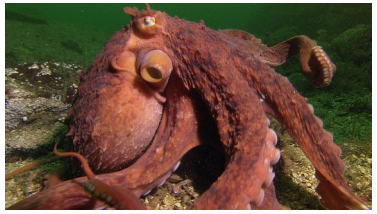
Start with labeled data

elephant

dog

snow leopard

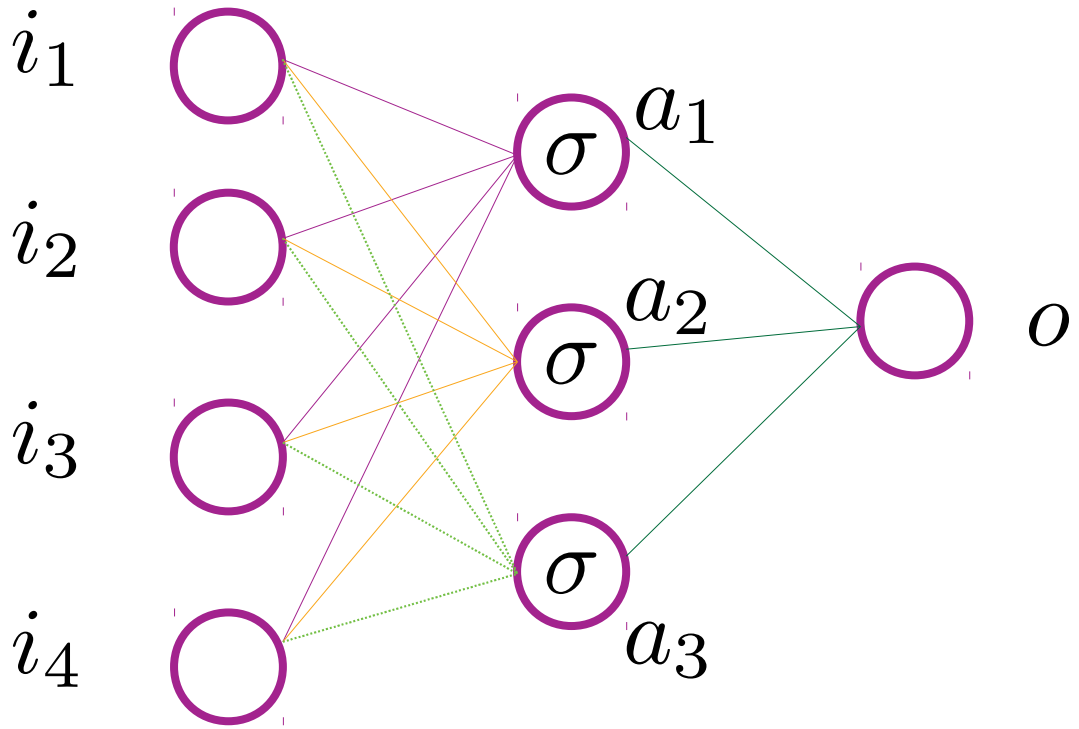
penguin



octopus

Full Training Cycle

Pick network architecture (more details later)



Full Training Cycle

Make predictions (forward propagation)

$$\underbrace{o}_{\text{a number}} = \underbrace{V}_{\text{weight}} \underbrace{\sigma}_{\text{activation}} \left(\underbrace{W}_{\text{weight}} \underbrace{i}_{\text{input}} + \underbrace{b}_{\text{bias}} \right) + \underbrace{c}_{\text{bias}}$$

V, W, b, c unknown

V, W initialized randomly according to some distribution

b, c initialized to vectors of zeros

Full Training Cycle

Make predictions (forward propagation)

$$\underbrace{o}_{\text{a number}} = \underbrace{V}_{\text{weight}} \underbrace{\sigma}_{\text{activation}} \left(\underbrace{W}_{\text{weight}} \underbrace{i}_{\text{input}} + \underbrace{b}_{\text{bias}} \right) + \underbrace{c}_{\text{bias}}$$

Compare output, o to actual value, v using a loss function (exactly like previous models)

Full Training Cycle

Make predictions (forward propagation)

$$\underbrace{o}_{\text{a number}} = \underbrace{V}_{\text{weight}}\ \underbrace{\sigma}_{\text{activation}}\ (\underbrace{W}_{\text{weight}}\ \underbrace{i}_{\text{input}} + \underbrace{b}_{\text{bias}}) + \underbrace{c}_{\text{bias}}$$

Compare output, o to actual value, v using a loss function

$$\text{Loss: } L = \frac{1}{2}(o - v)^2$$

- $= 0$ if $o = v$ (prediction matches value)
- larger the worse predictions get

Loss function depends on your problem

Full Training Cycle

Make predictions (forward propagation)

$$\underbrace{o}_{\text{a number}} = \underbrace{V}_{\text{weight}} \underbrace{\sigma}_{\text{activation}} \left(\underbrace{W}_{\text{weight}} \underbrace{i}_{\text{input}} + \underbrace{b}_{\text{bias}} \right) + \underbrace{c}_{\text{bias}}$$

Tweak V, W, b and c to adjust o and minimize L

$$\text{Minimize: } L = L(V, W, b, c)$$

Full Training Cycle

Predict (Forward propagation)

$$o = V\sigma(Wi + b) + c$$

$o = \text{polar bear}$

$L(o, v)$

Adjust V, W, b, c

Minimize L

$v = \text{bunny}$

Loss Functions

Loss Functions

Measuring deviation between predictions and targets/labels

Each example is treated independently:

$$\text{Loss, } \mathcal{L} = \Sigma_{i=1}^{n} \underbrace{C[\underbrace{\hat{y}_i}_{\text{prediction}}, \underbrace{y_i}_{\text{label}}]}_{\text{Cost for example i}}$$

Loss Functions

$$\text{Loss, } \mathcal{L} = \Sigma_{i=1}^{n} C[\underbrace{\underbrace{\hat{y}_i}_{\text{prediction}}, \underbrace{y_i}_{\text{label}}}_{\text{Cost for example i}}]$$

Choice of C dictated by problem type

C bounded by below i.e. $C \geq 0$

Loss Functions: Regression

Mean Squared Error

$$C[\hat{y}, y] = (\hat{y} - y)^2$$

Any deviation from y is punished

Loss Functions: Binary Classification

Target: 0 or 1

Probability of belonging to class 1 : $p \in [0, 1]$

How to measure deviation?

Loss Functions: Binary Classification

Pick a threshold, say 0.5

Convert p to label:

$$p \leq 0.5 \rightarrow \text{prediction} = 0$$

$$p > 0.5 \rightarrow \text{prediction} = 1$$

Accuracy: % of predictions correct???

Loss Functions: Binary Classification

Accuracy: % of predictions correct???

Problems?

Loss Functions: Binary Classification

Accuracy: % of predictions correct???

Problems:

Dependent on threshold

Not a smooth function of predictions $\rightarrow$ indirect control of weights

Loss Functions: Binary Classification

$$C = (\underbrace{y}_{0 \text{ or } 1} - \underbrace{p}_{\in[0,1]})^2 \quad ???$$

Problems?

Loss Functions: Binary Classification

$$C = (\underbrace{y}_{0 \text{ or } 1} - \underbrace{p}_{\in[0,1]})^2 \quad ???$$

Unnatural to compare class label to probability

Penalty bounded above by 1 $\rightarrow$ too mild

Loss Functions: Binary Classification

Given p, y : what's the **likelihood** of getting y?

$$y = 0 : 1 - p$$

Loss Functions: Binary Classification

Given p, y : what's the **likelihood** of getting y?

$$y = 1 : p$$

Loss Functions: Binary Classification

Given p, y : what's the **likelihood** of getting y?

$$\boxed{p^y(1-p)^{1-y}}$$

$$y = 0 : p^0(1-p)^{1-0} = 1 - p$$

$$y = 1 : p^1(1-p)^{1-1} = p$$

Loss Functions: Binary Classification

All samples are independent

Total likelihood: $\Pi_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1-y_i}$

p_i depends on the model: V, W, b, c

Tune $V, W, b, c \rightarrow$ Change $p_i \rightarrow$ Max Likelihood

Loss Functions: Binary Classification

$$\mathcal{L} = \Pi_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1-y_i}$$

Maximizing $\mathcal{L}$ equivalent to maximizing $\log \mathcal{L}$

(log is a monotonically increasing function)

$$\log \mathcal{L} = \Sigma_i^n y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

Nice: change things to a sum

Loss Functions: Binary Classification

Usually have algorithms to minimize functions

Maximizing f equivalent to minimizing $-f$

Minimize:

$$\mathcal{L}' = -\log \mathcal{L} = \boxed{-\Sigma_i^n y_i \log(p_i) + (1 - y_i) \log(1 - p_i)}$$

Loss Functions: Binary Classification

$$\mathcal{L} = -\Sigma_i^n y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

$$\text{For one example, if } y = 1 : \mathcal{L} = -\log(p_i)$$

$$y = 0 : \mathcal{L} = -\log(1 - p_i)$$

Loss Functions: Binary Classification

$$y = 1 : \mathcal{L} = -\log(p_i)$$

$$p_i = 1(\text{correct})\mathcal{L} = -\log(1) = 0$$

$$p_i = 0(\text{incorrect})\mathcal{L} = -\log(0) \rightarrow \infty$$

Loss Functions: Binary Classification

$$y = 0 : \mathcal{L} = -\log(1 - p_i)$$

$$p_i = 0(\text{correct},)\mathcal{L} = -\log(1 - 0) = 0$$

$$p_i = 1(\text{incorrect})\mathcal{L} = -\log(1 - 1) \rightarrow \infty$$

Loss Functions: Multi-class Classification

$$y \in 1, 2, \ldots, n$$

Loss Functions: Multi-class Classification

$$y \in 1, 2, \ldots, n$$

$$\mathcal{L} = -\Sigma_{i=1}^{n_{examples}} y_{i1} \log(p_{i1}) + y_{i2} \log(p_{i2}) + \ldots + y_{in} \log(p_{in})$$

Loss Functions: Multi-class Classification

$$y \in 1, 2, \ldots, n$$

$$\mathcal{L} = -\Sigma_{i=1}^{n_{examples}} y_{i1} \log(p_{i1}) + y_{i2} \log(p_{i2}) + \ldots + y_{in} \log(p_{in})$$

$$\mathcal{L} = -\Sigma_{i=1}^{n_{\text{examples}}} \Sigma_{j=1}^{n} y_{ij} \log(p_{ij})$$

Loss Functions: Multi-class Classification

$$y \in 1, 2, \ldots, n$$

$$\mathcal{L} = -\Sigma_{i=1}^{n_{examples}} y_{i1} \log(p_{i1}) + y_{i2} \log(p_{i2}) + \ldots + y_{in} \log(p_{in})$$

$$\mathcal{L} = -\Sigma_{i=1}^{n_{\text{examples}}} \Sigma_{j=1}^{n} y_{ij} \log(p_{ij})$$

$$\text{Note: } p_{i1} + \ldots + p_{in} = 1, y_{i1} + \ldots + y_{in} = 1, y_{ij} \in 0, 1$$

Loss Functions: Multi-class Classification

Exactly what we used in the trip classification problem in the notebooks!

Loss Functions: Custom

You can design one based on your problem

Well-designed loss greatly aids learning/training

Example: Kullback-Leibler divergence for comparing two distributions

Exercise: Go to kaggle and look at evaluation metrics for 10 competitions

Loss Functions: Multi-class Classification

$$y \in 1, 2, \ldots, n$$

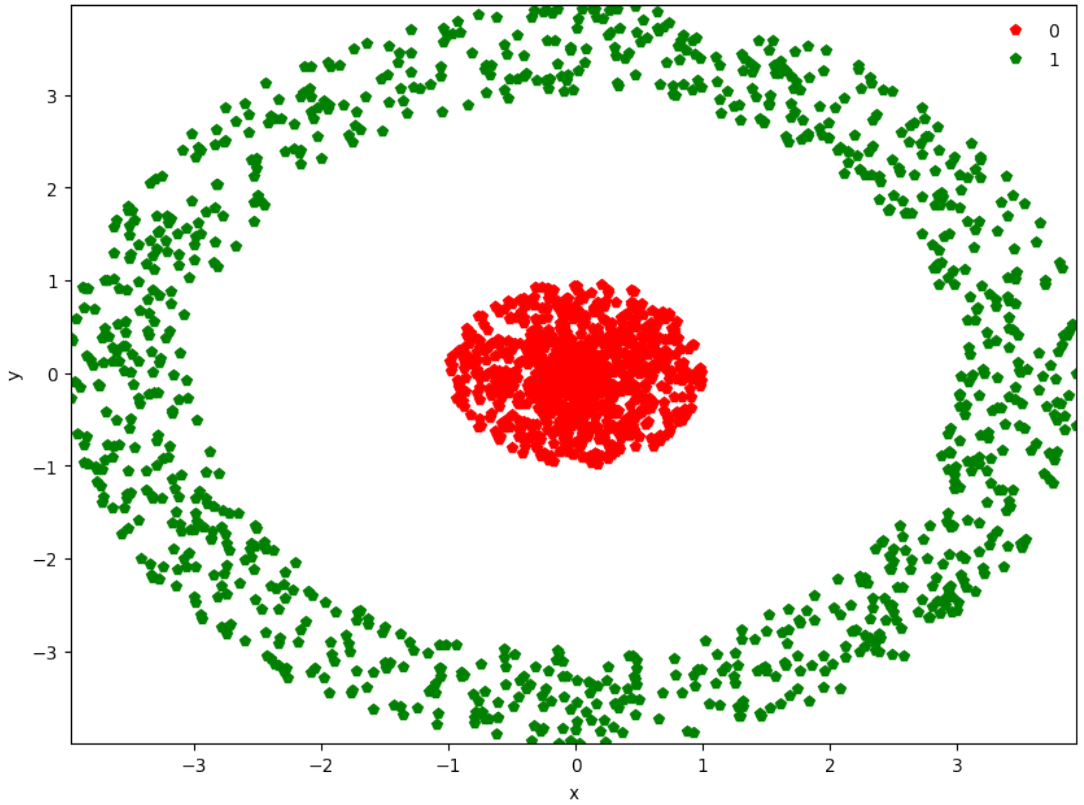
$$\mathcal{L} = -\Sigma_{i=1}^{n_{examples}} y_{i1} \log(p_{i1}) + y_{i2} \log(p_{i2}) + \ldots + in \log(p_{in})$$

$$\mathcal{L} = -\Sigma_{i=1}^{n_{\text{examples}}} \Sigma_{j=1}^{n} y_{ij} \log(p_{ij})$$

$$\text{Note: } p_{i1} + \ldots + p_{in} = 1, y_{i1} + \ldots + y_{in} = 1, y_{ij} \in 0, 1$$

Intuition: Why do neural networks work?

Problem



Binary Classification: Given (x, y), predict class 0 or 1

Problem

Logistic Regression

$$\text{Compute: } \sigma(x, y) = \frac{1}{1 + e^{-(ax+by+c)}}$$

Probability of belonging to class 1

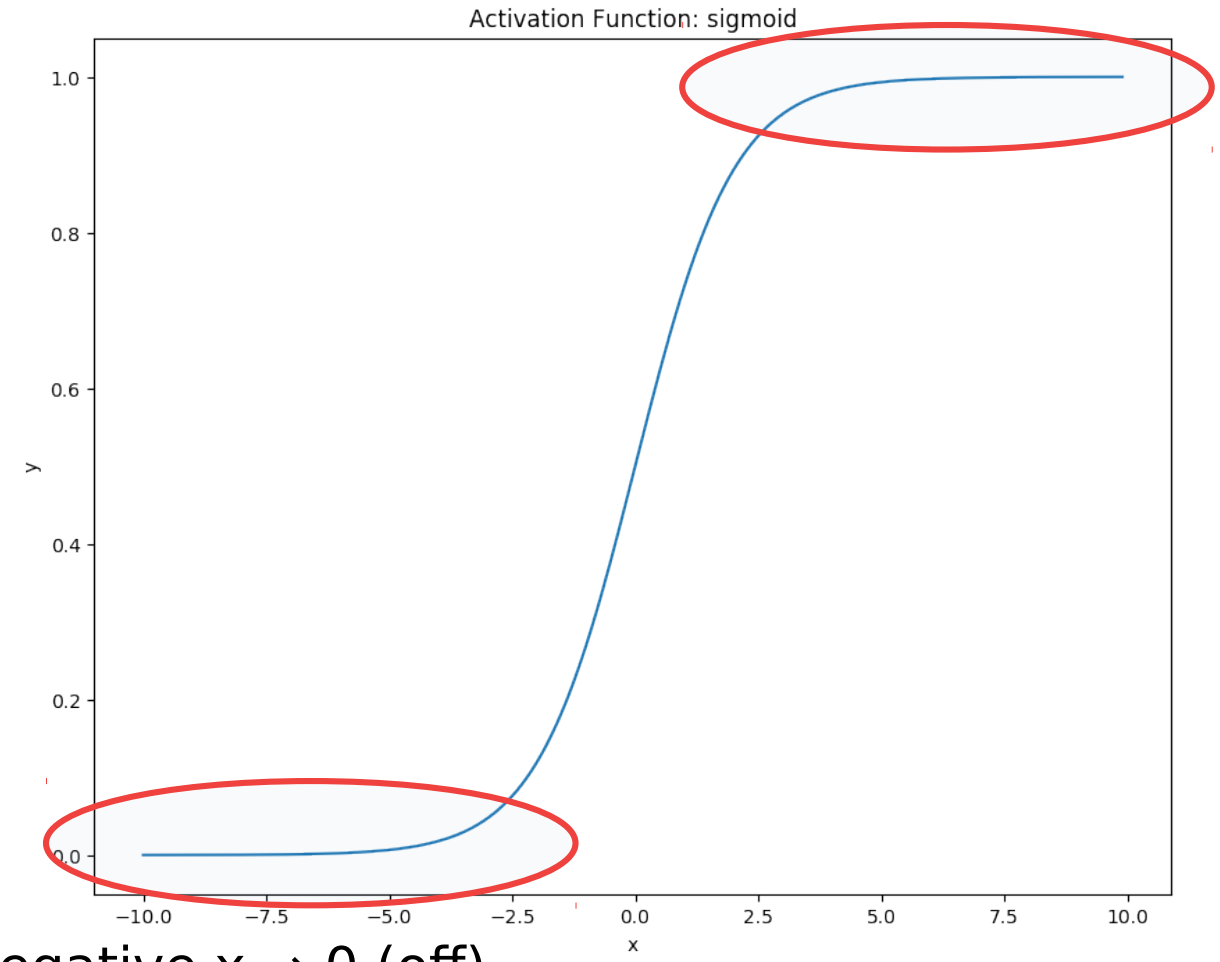
a, b, c found by minimizing cross-entropy loss

$$\mathcal{L} = -\Sigma_{i=1}^{n}(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Label $y_i = 0$ or $y_i = 1$

Prediction: $p_i = p_i(a, b, c) \in [0, 1]$

Positive x → 1 (on)



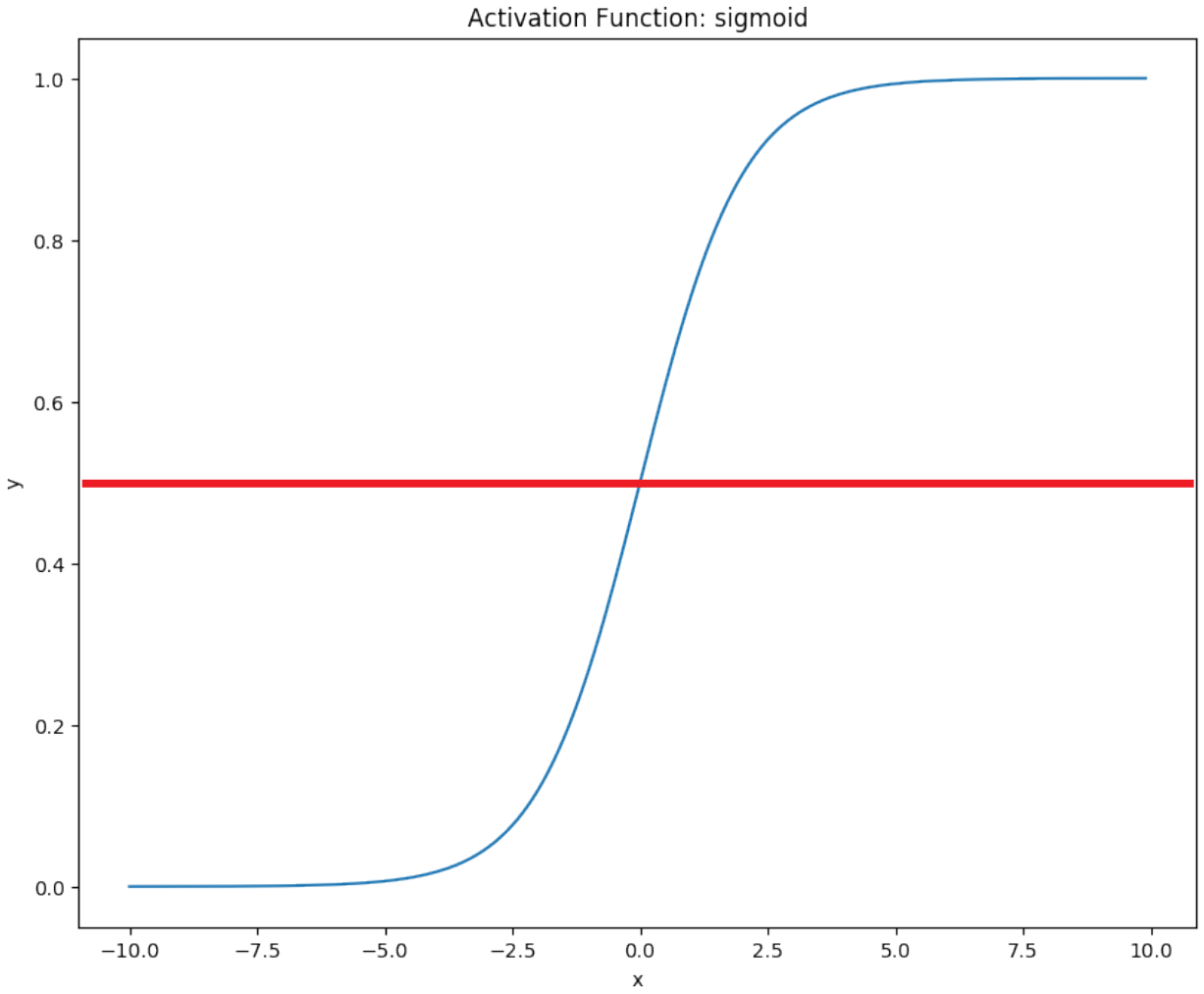
Negative x → 0 (off)

$$\sigma(ax + by + c) = \frac{1}{1 + e^{-(ax+by+c)}}$$

Class 1 (100%)

Probability

Class 0 (0%)



50% →
Decision boundary

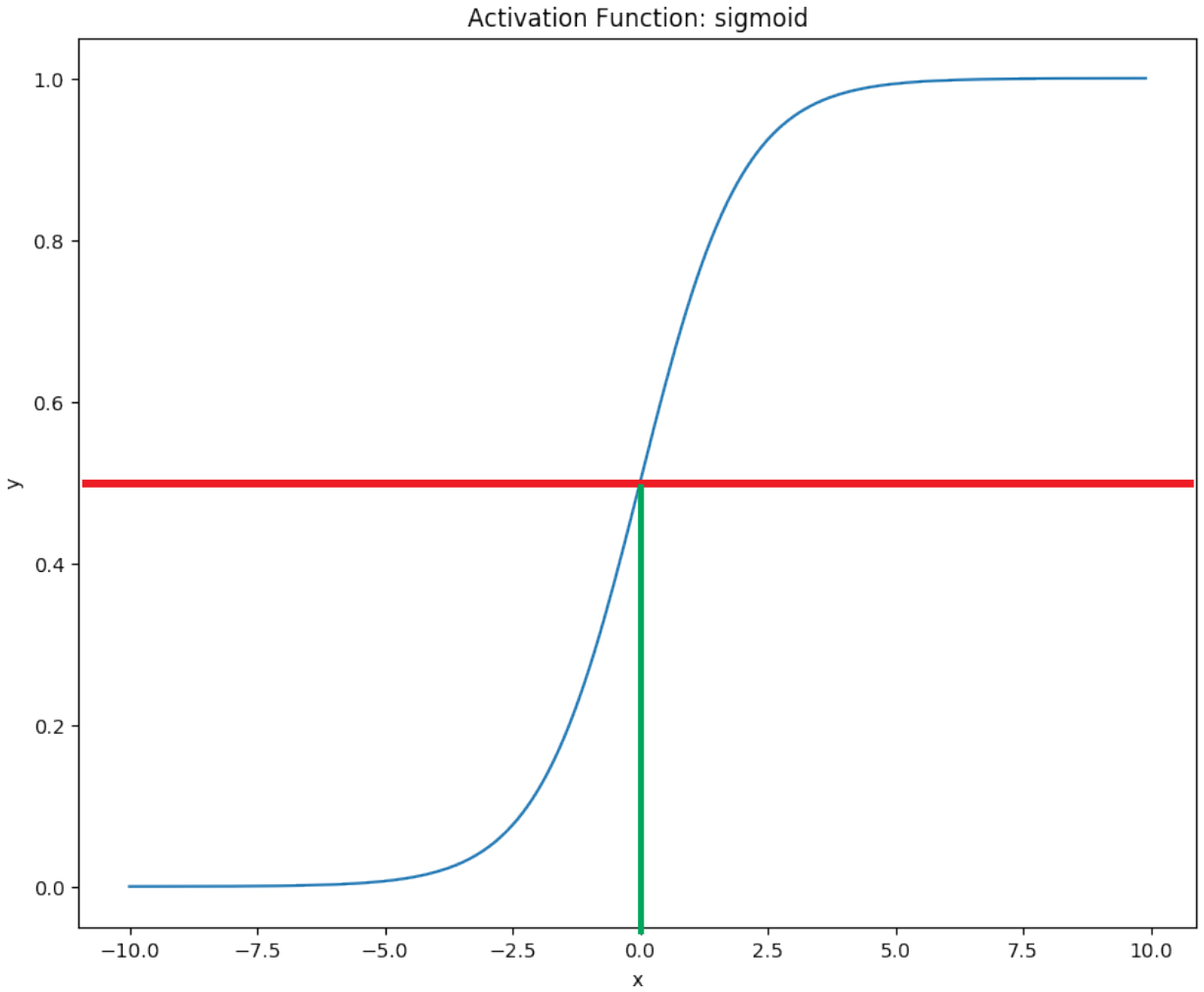
$$\text{y-axis} = \sigma(x) = \text{probability of being in class 1}$$

Red Hat

Class 1 (100%)

Probability

Class 0 (0%)



50% →
Decision boundary

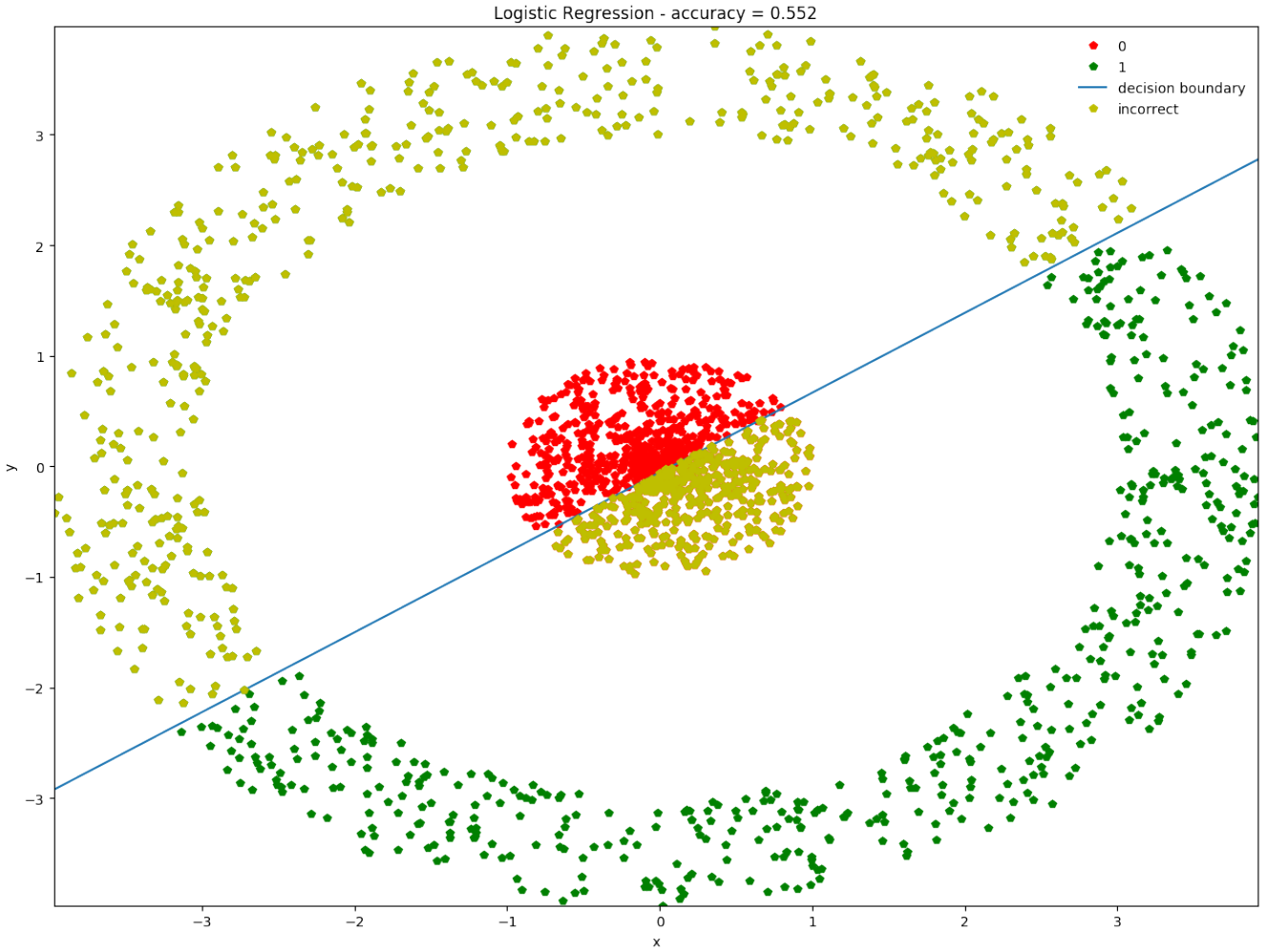
$$\text{y-axis} = \sigma(x) = \text{probability of being in class 1}$$

$$\sigma(x) = 0.5 \implies x(\text{input to sigmoid}) = 0$$

Red Hat

$$\sigma(ax + by + c) = 50\% \rightarrow \underbrace{ax + by + c = 0}_{\text{Straight Line}}$$

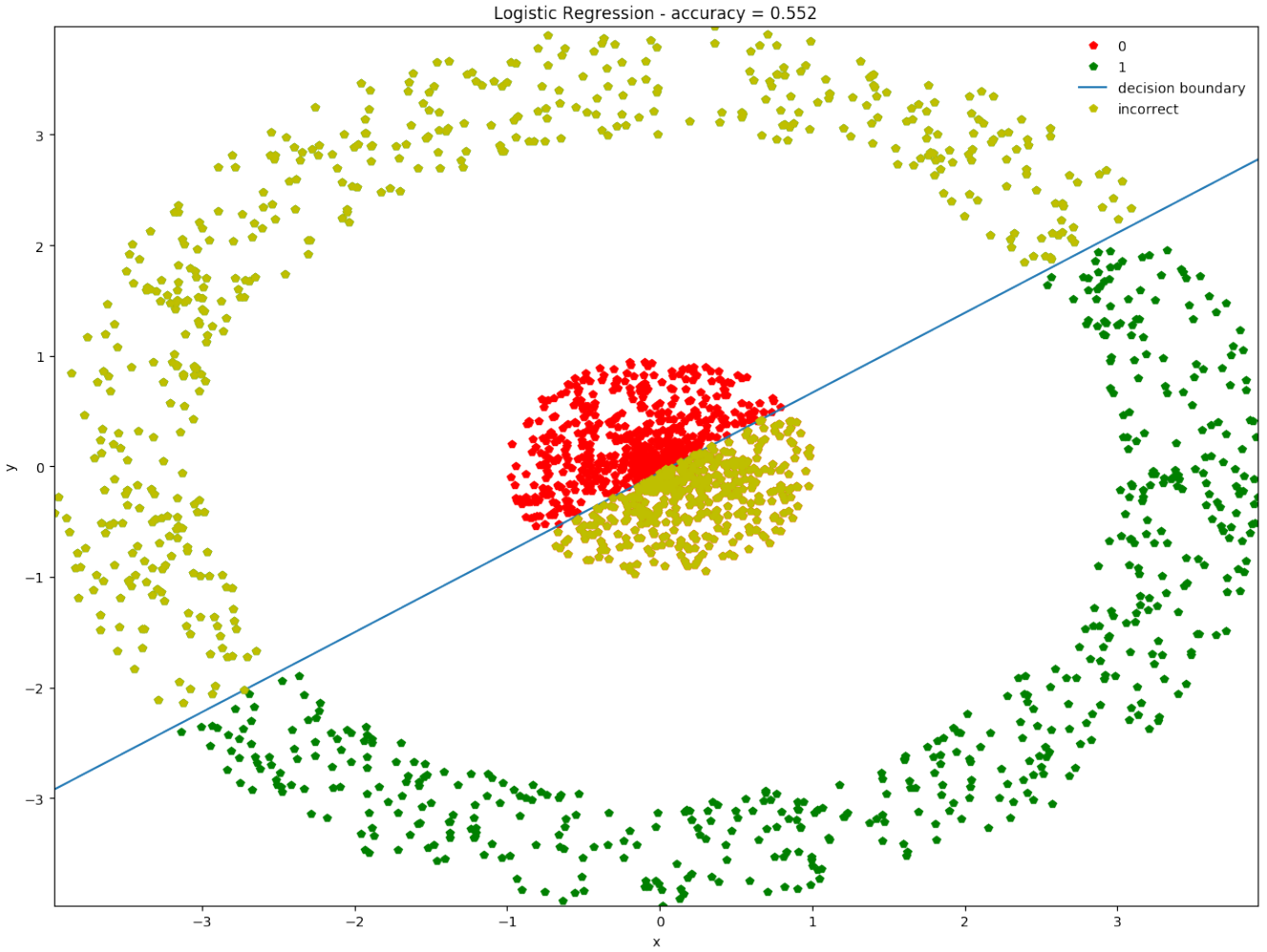
$$ax + by + c = 0 \implies y = -\frac{c}{b} - \frac{a}{b}x$$



No straight line can separate class 0 from class 1

Not **linearly separable**

Red Hat



No straight line can separate class 0 from class 1

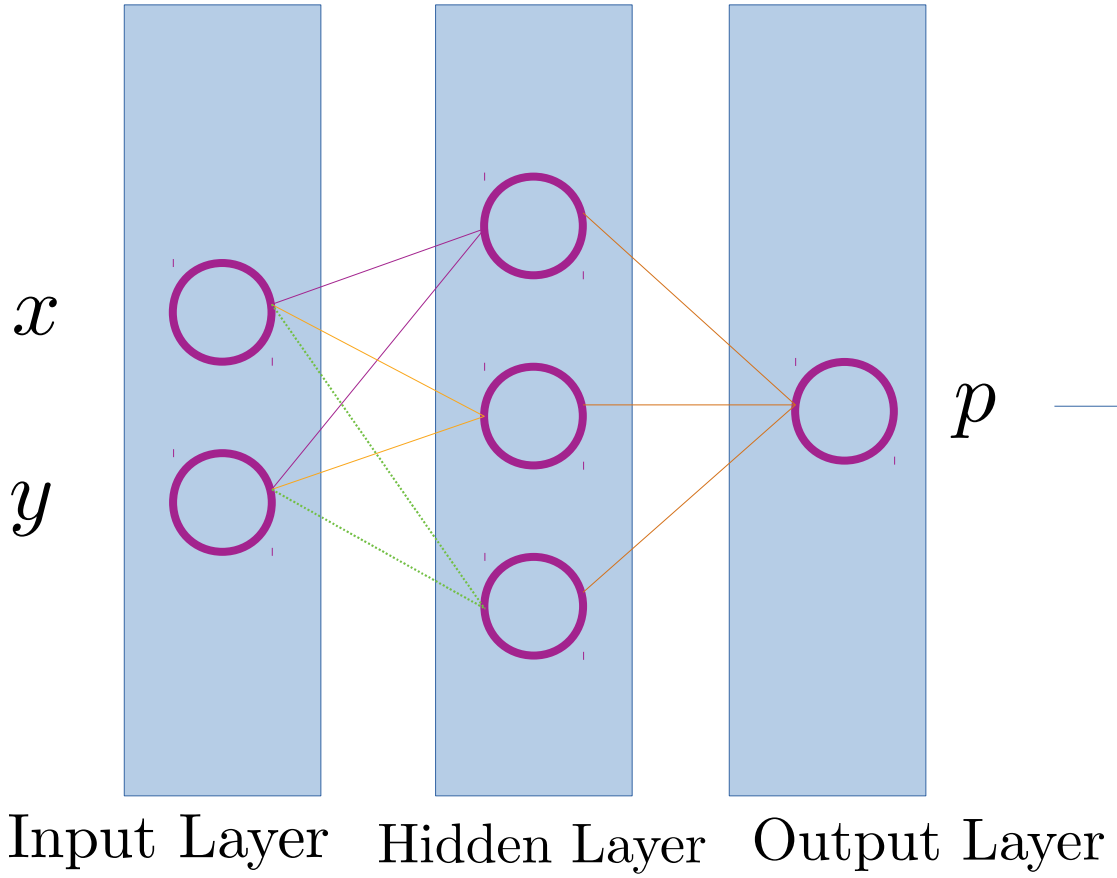
One solution:

Convert to polar coordinates

$$r^2 = x^2 + y^2$$

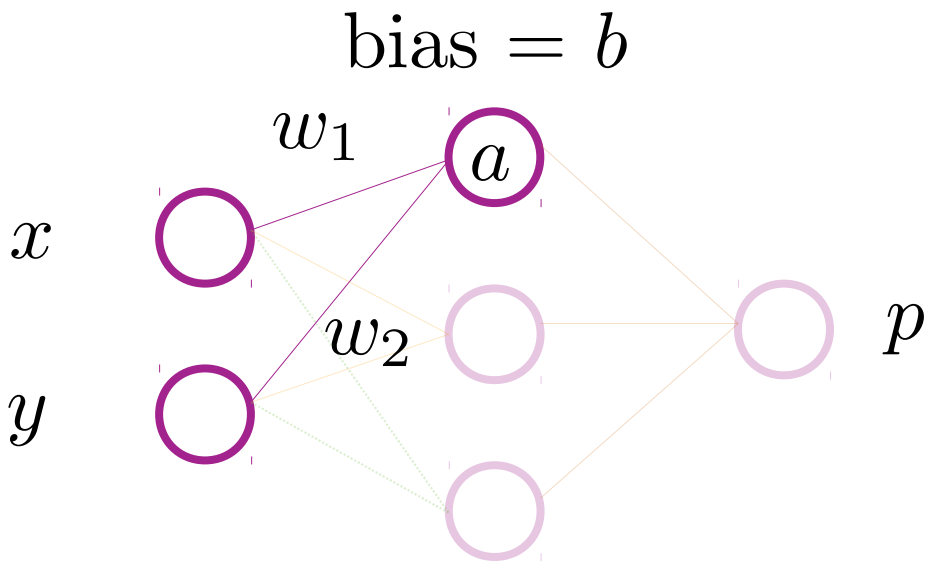
What about high dimensional data when we can't visualize?

Red Hat



Minimize:
Cross-entropy

Activation = Sigmoid

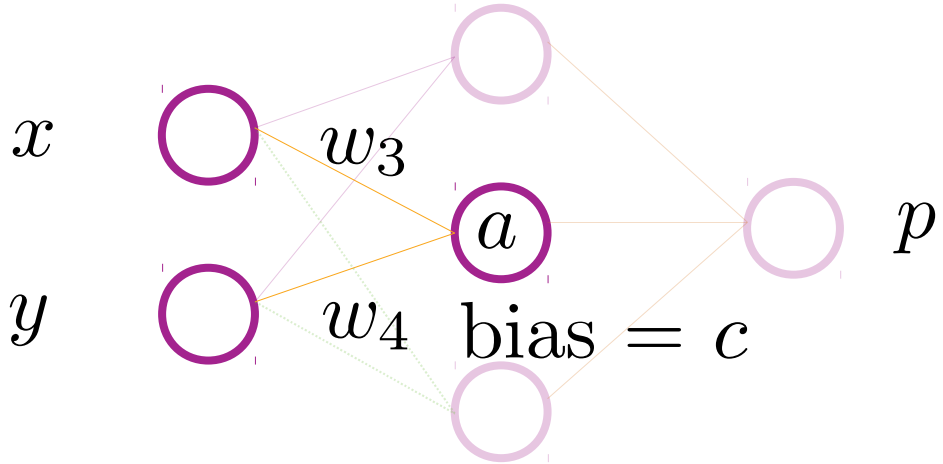


$$a = \sigma(w_1 x + w_2 y + b)$$

Decision boundary:

$$w_1 x + w_2 y + b = 0$$

Input Layer Hidden Layer Output Layer



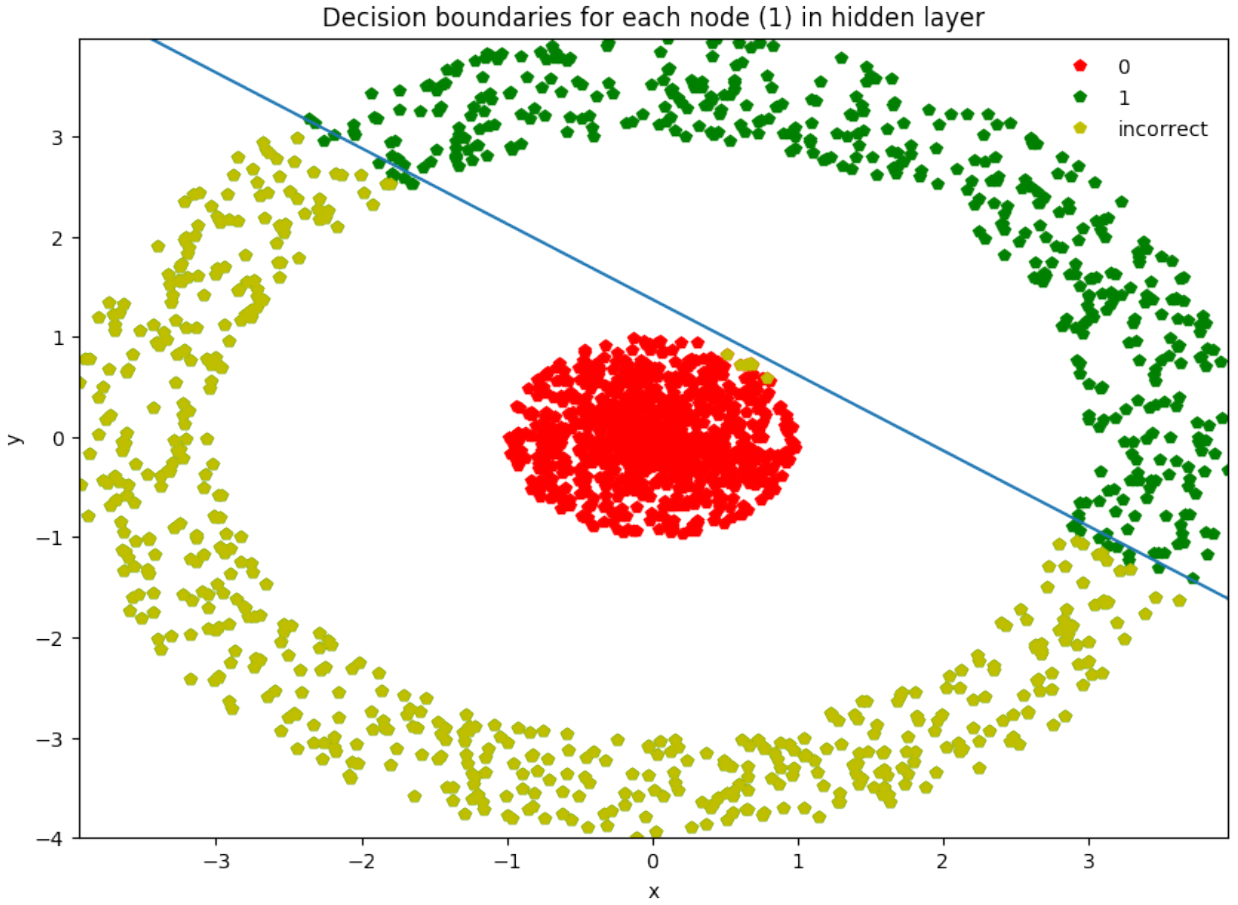
$$\boxed{a = \sigma(w_3 x + w_4 y + c)}$$

Decision boundary:

$$w_3 x + w_4 y + c = 0$$

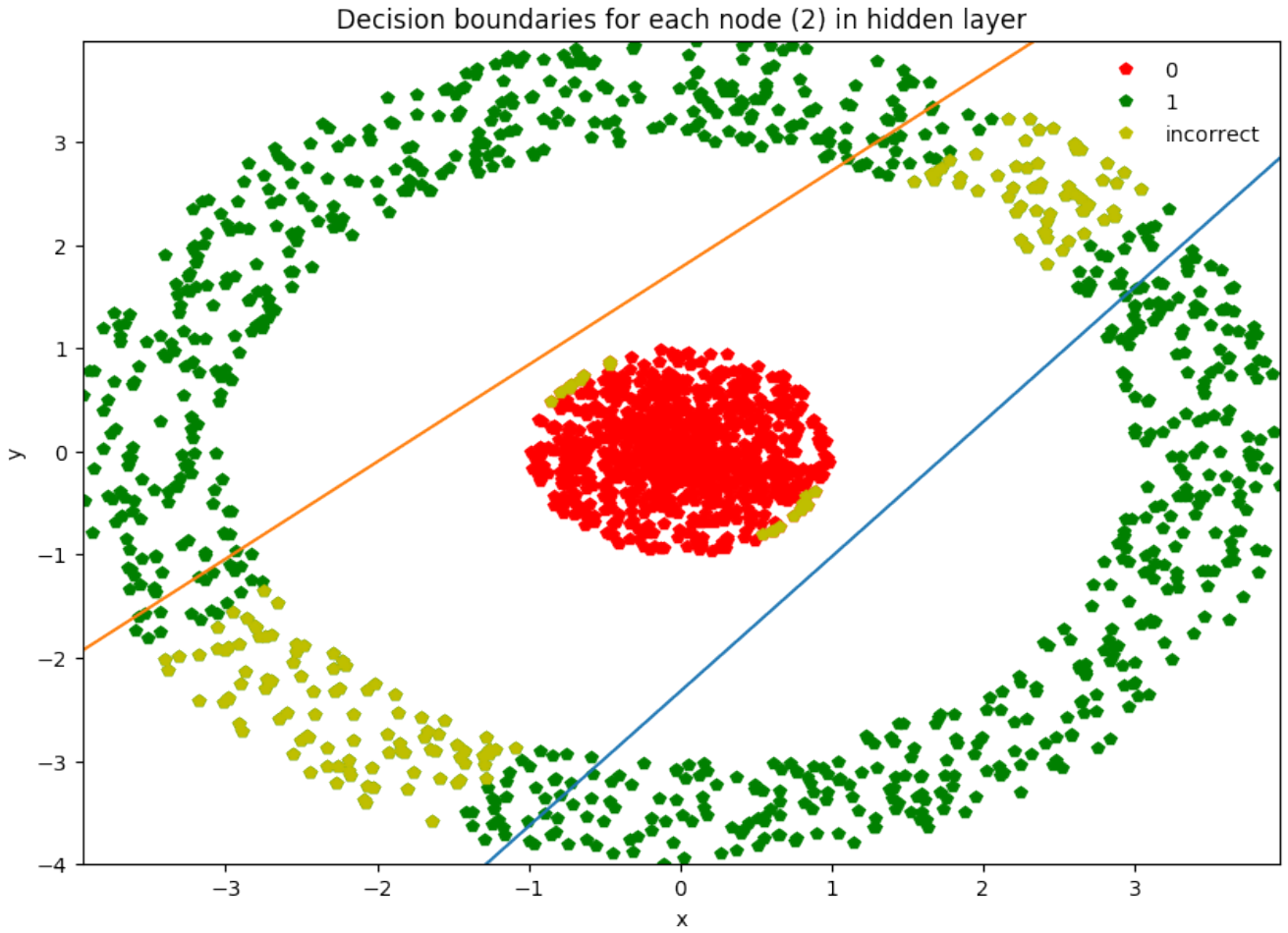
Input Layer Hidden Layer Output Layer

Hidden Nodes = 1



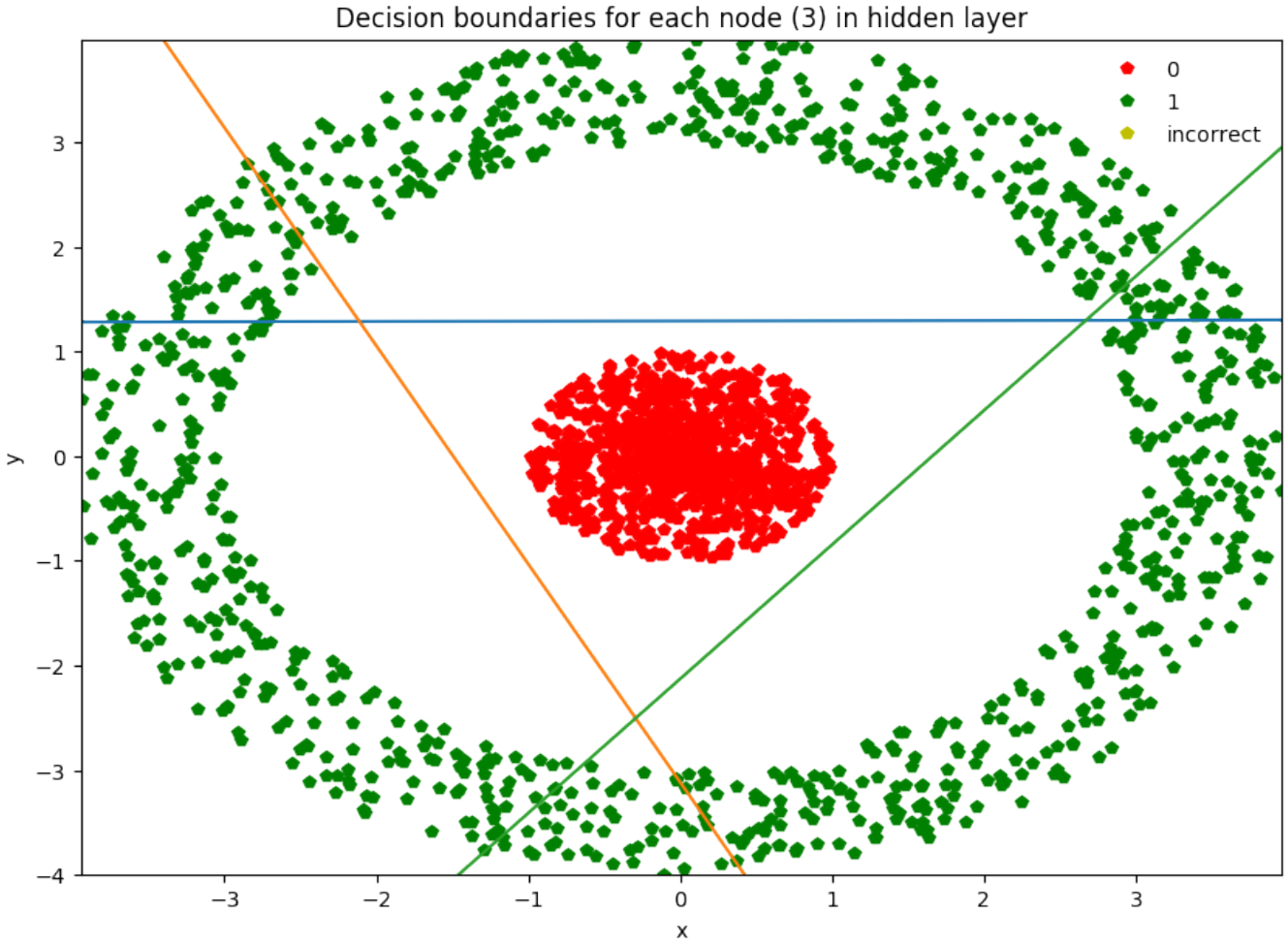
Red Hat

Hidden Nodes = 2



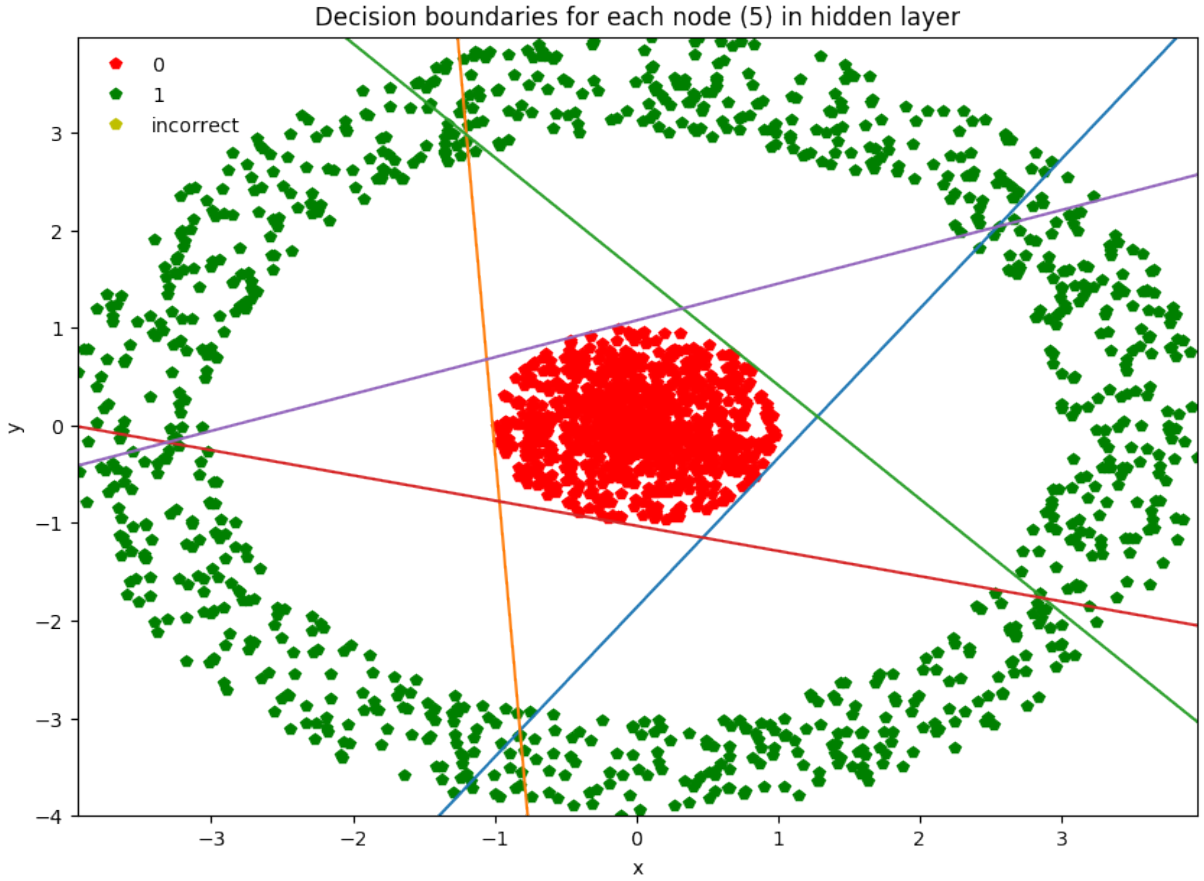
Red Hat

Hidden Nodes = 3



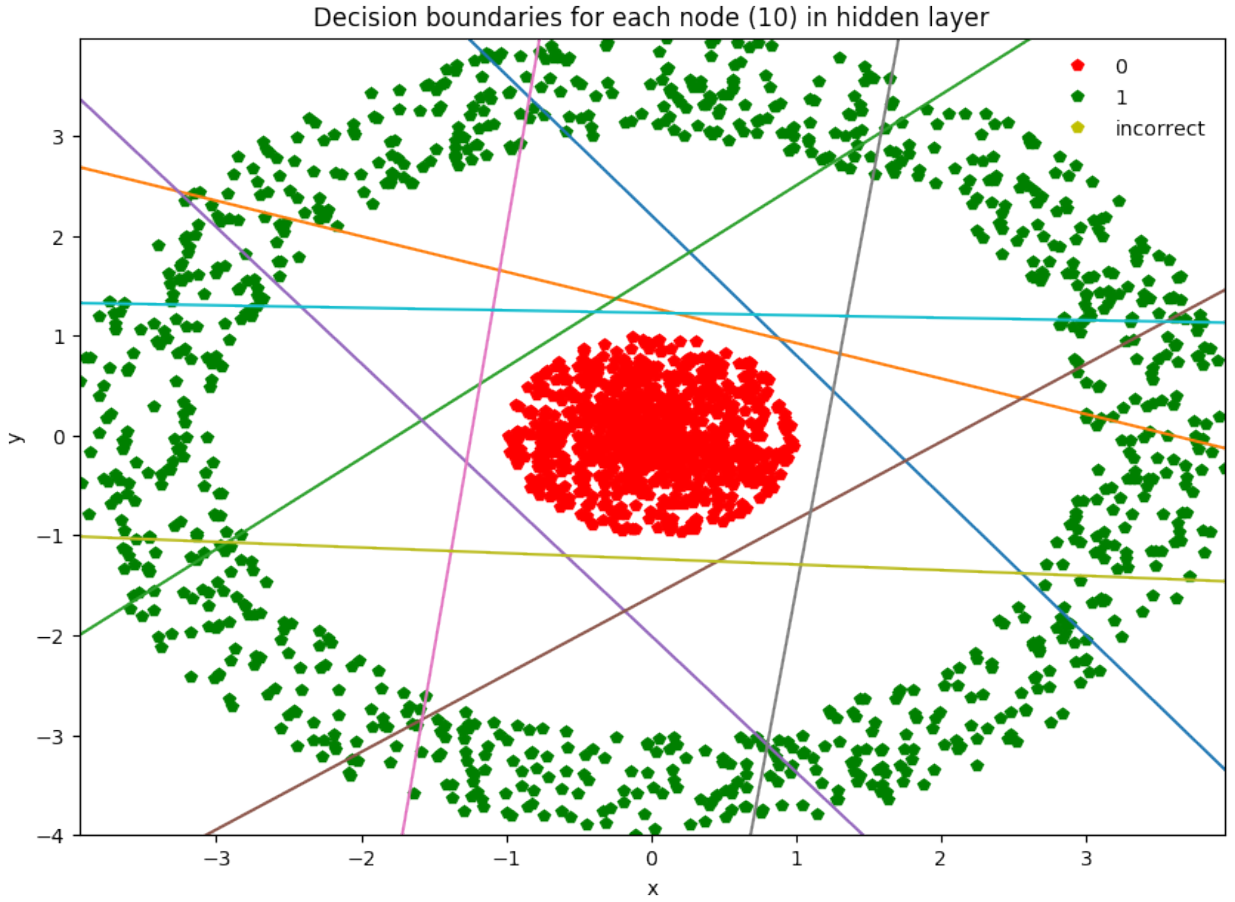
Red Hat

Hidden Nodes = 5



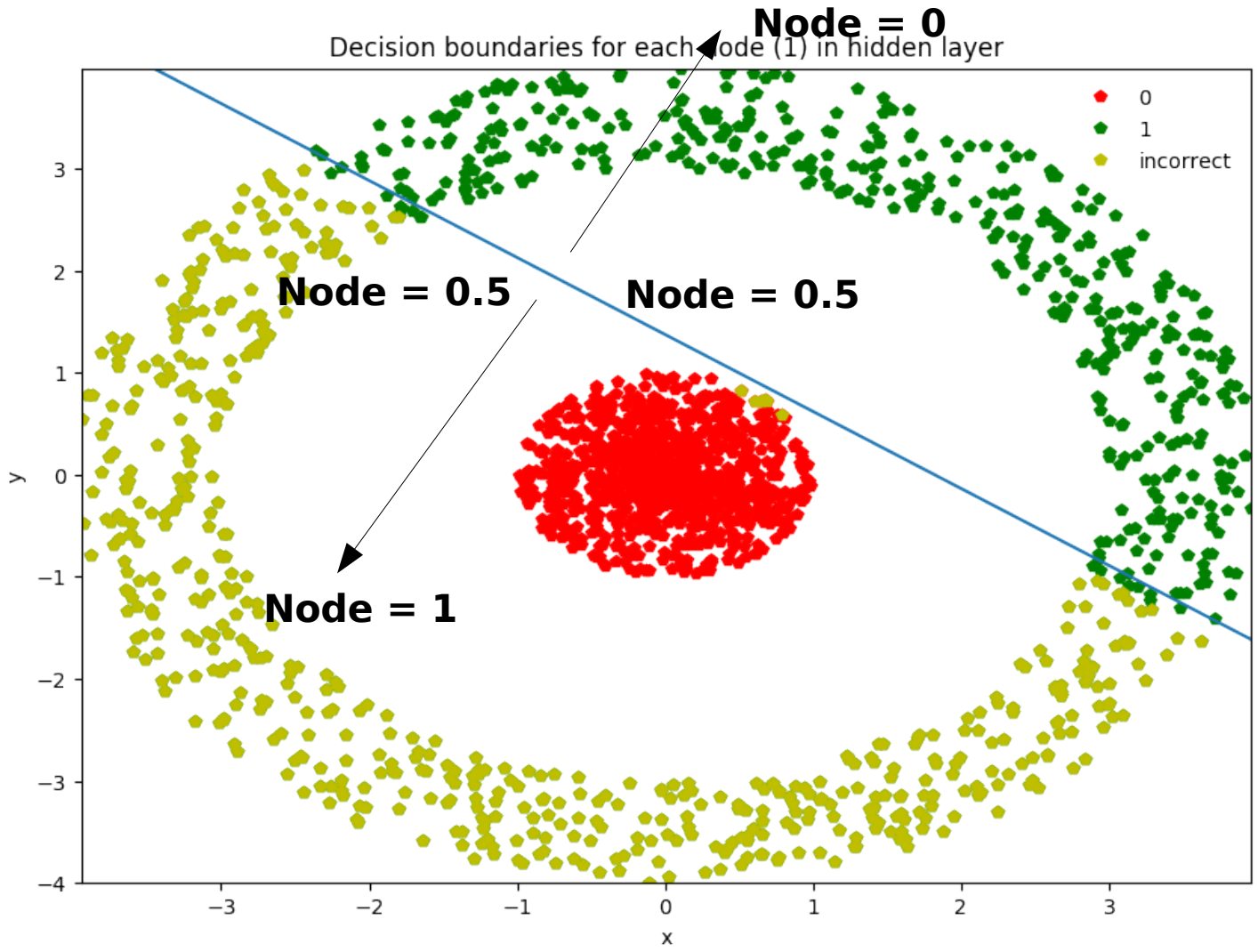
Red Hat

Hidden Nodes = 10



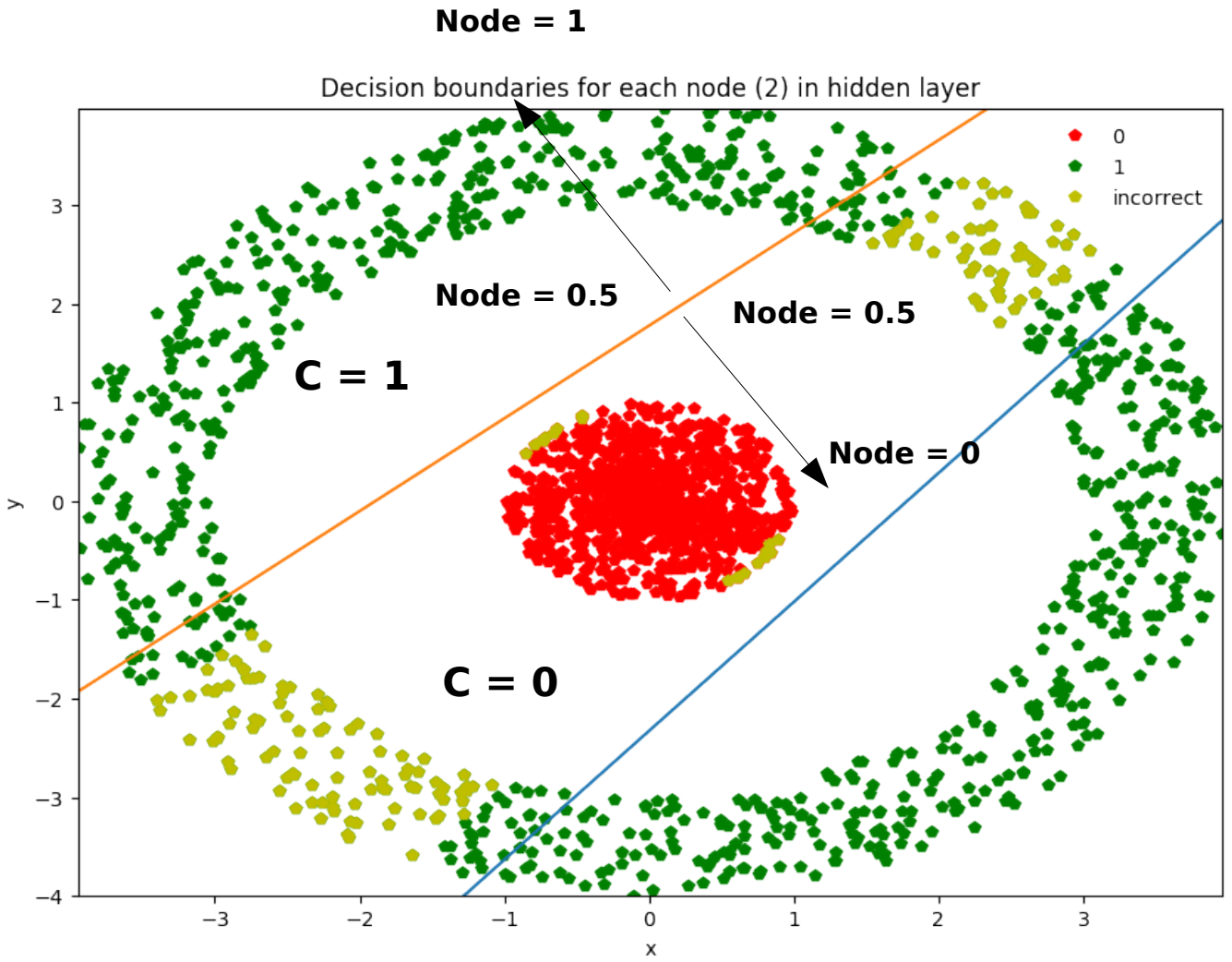
Red Hat

Hidden Nodes = 1



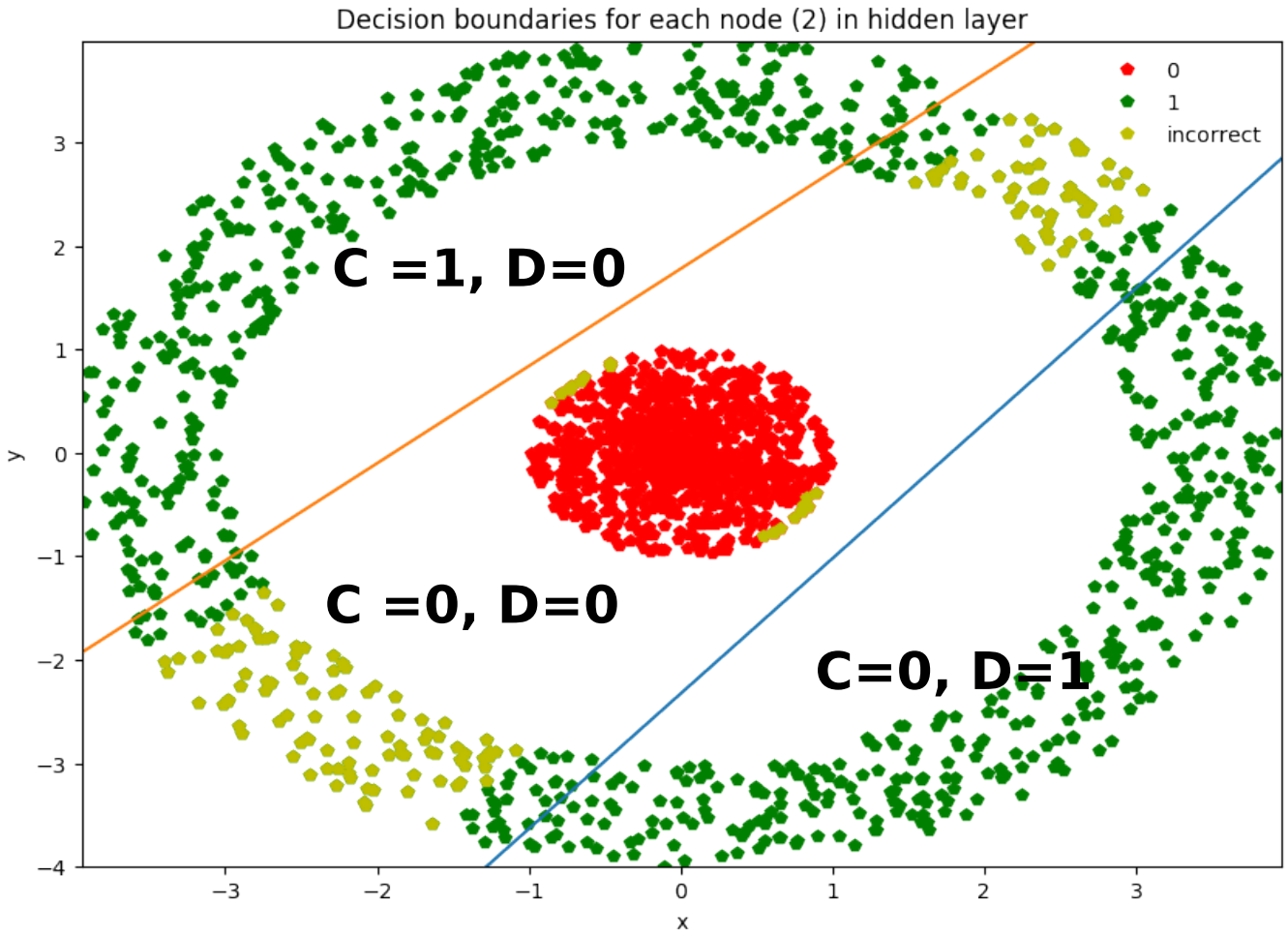
Red Hat

Hidden Nodes = 2



Red Hat

Hidden Nodes = 2



Red Hat

Split input space into 3 regions

$$C = 0, D = 0 \longrightarrow \text{Mixed - both class 0 and class 1}$$

$$C = 1, D = 0 \longrightarrow \text{Pure class 1}$$

$$C = 0, D = 1 \longrightarrow \text{Pure class 1}$$

N hidden binary nodes can split input into 2^N regions

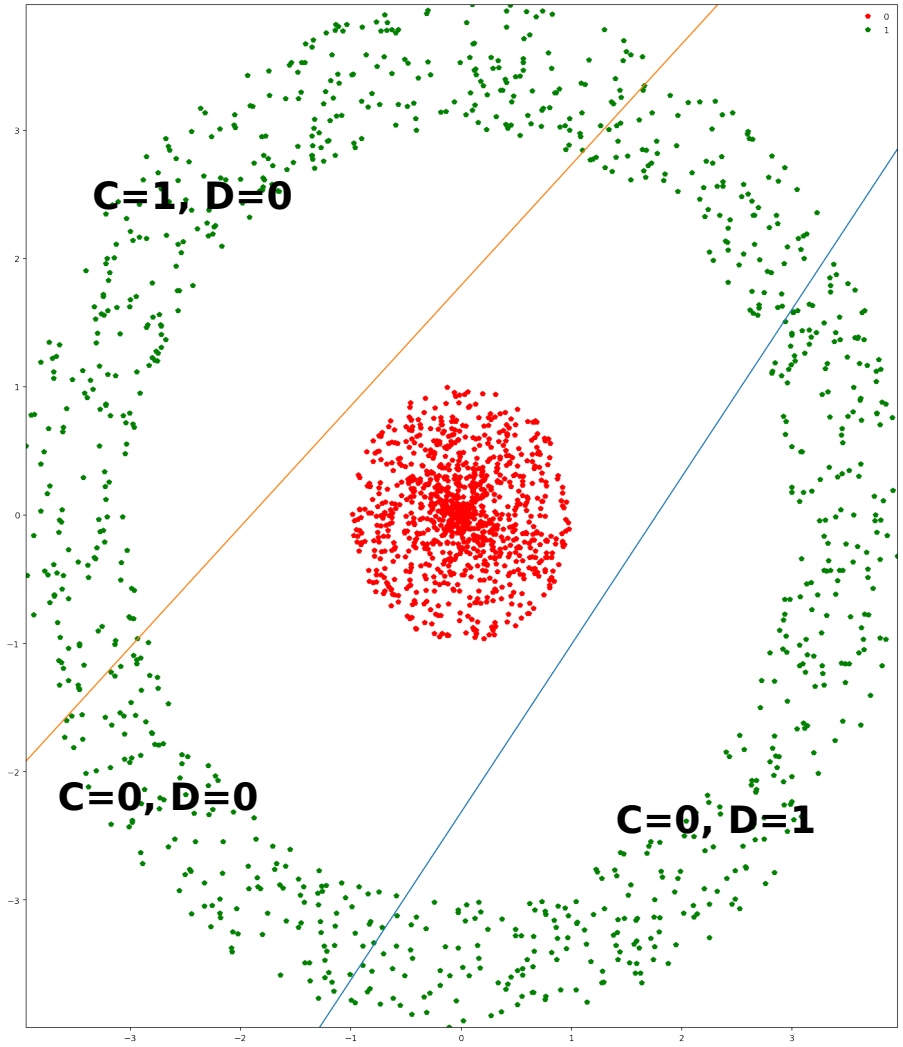
Keep increasing N (without overfitting) till regions "pure"

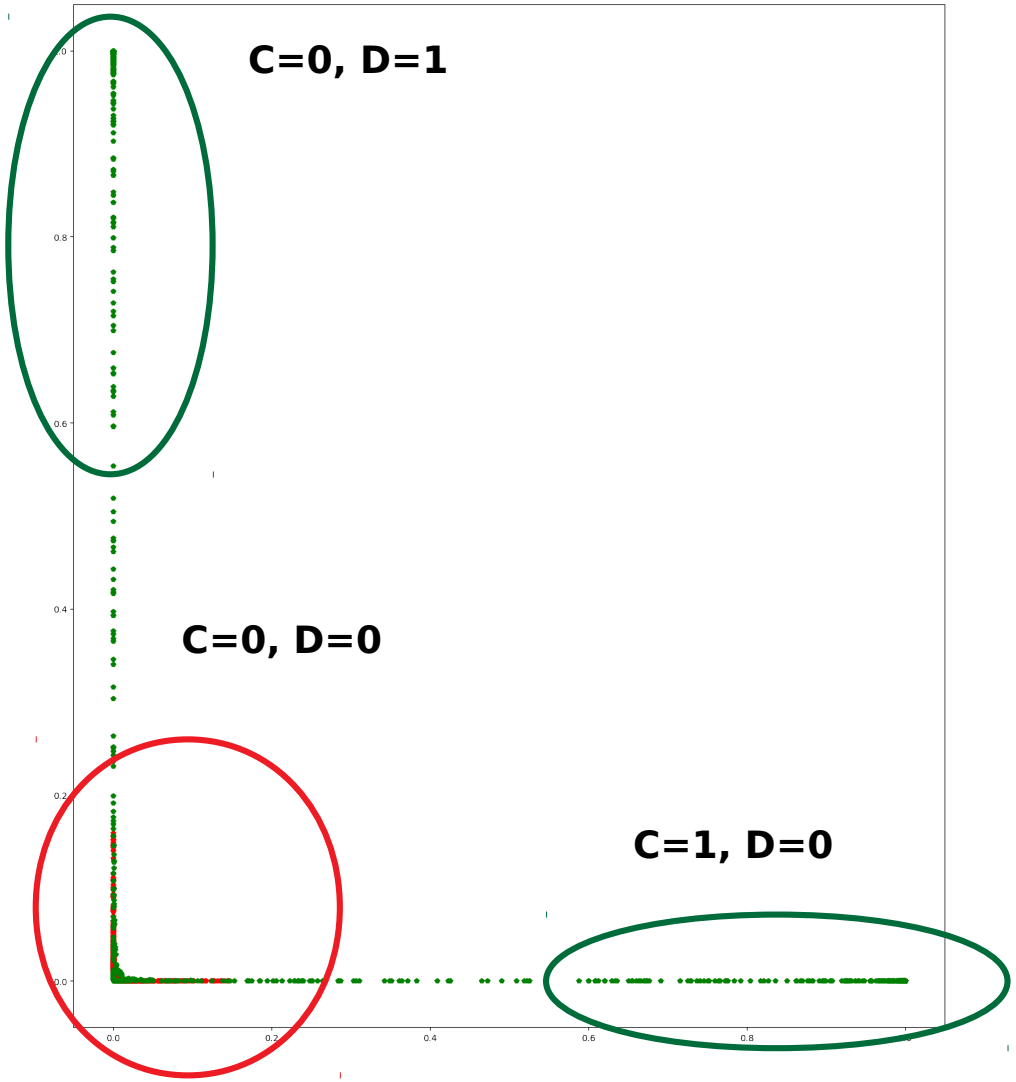
Map 2-dim input to N-dim space

$$(x, y) \rightarrow (a_1, a_2, \ldots, a_N) = \underbrace{(1, 0, \ldots, 1)}_{\text{corners of N-dim hypercube)}}$$

in the hope of linearly separating classes

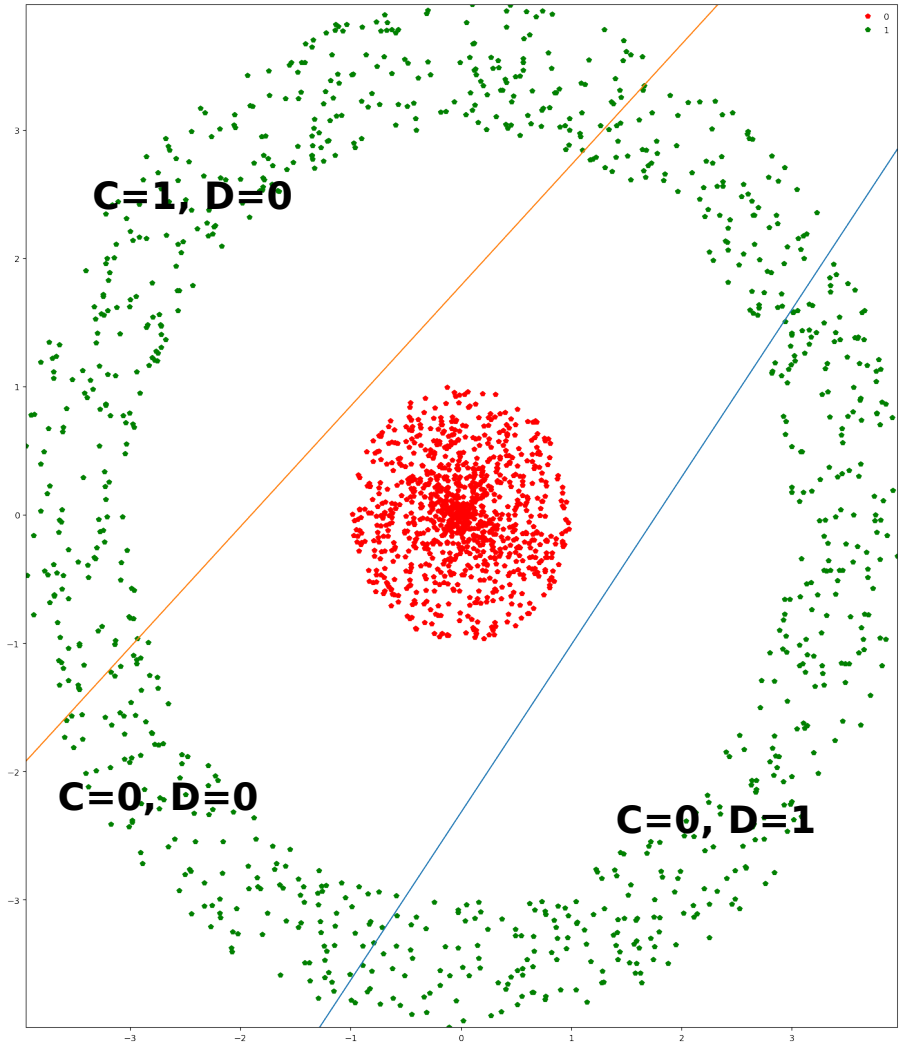
Map 2-dim input to 2-dim space

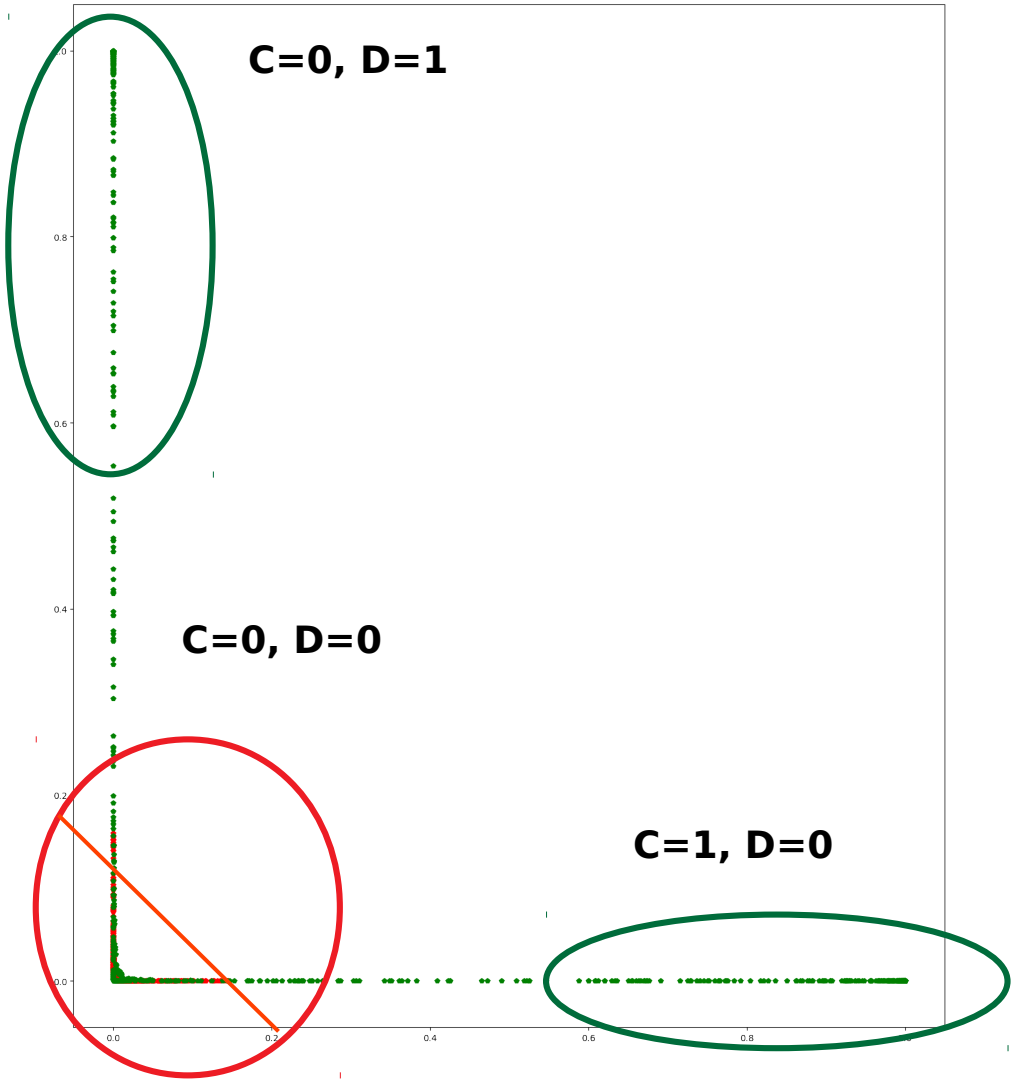




Red Hat

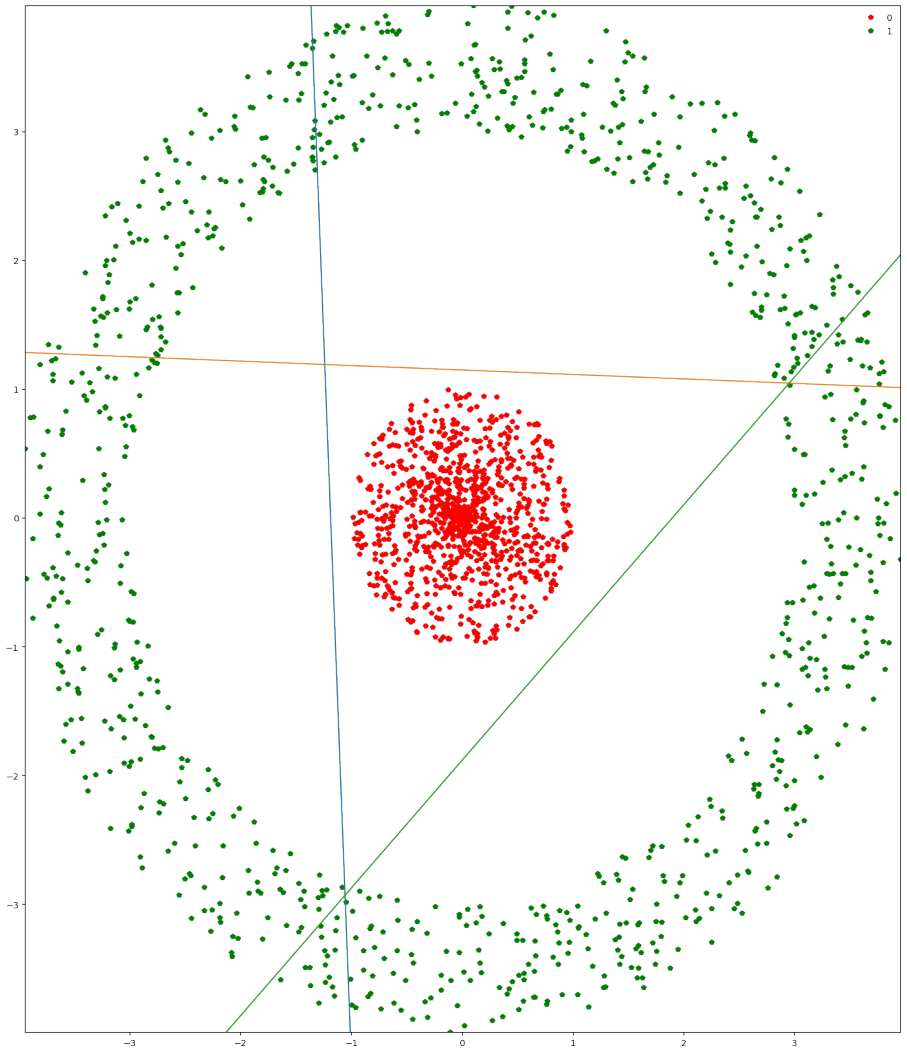
Map 2-dim input to 2-dim space

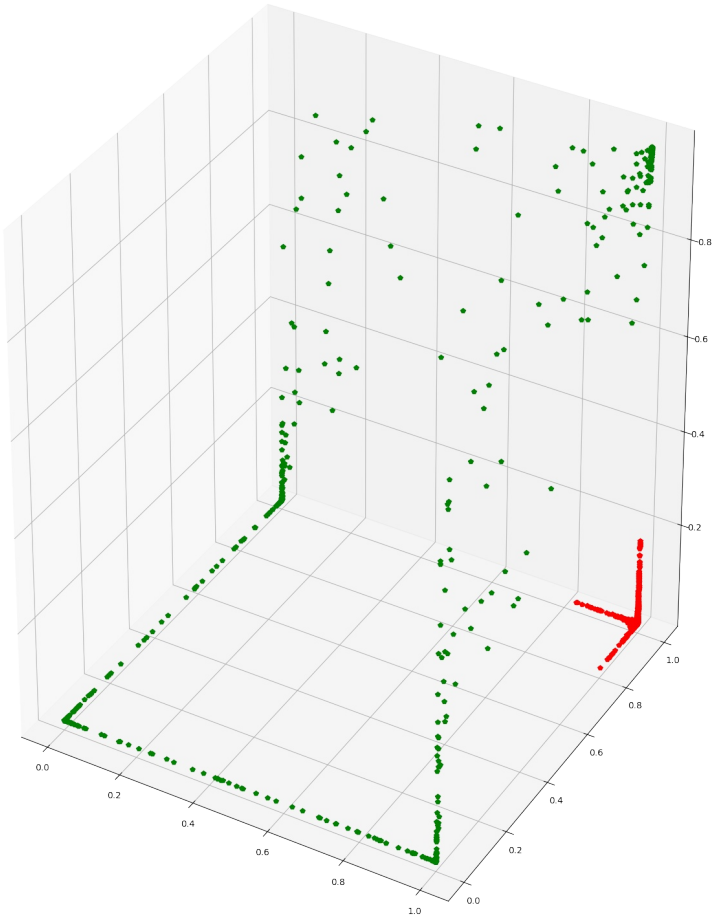




Red Hat

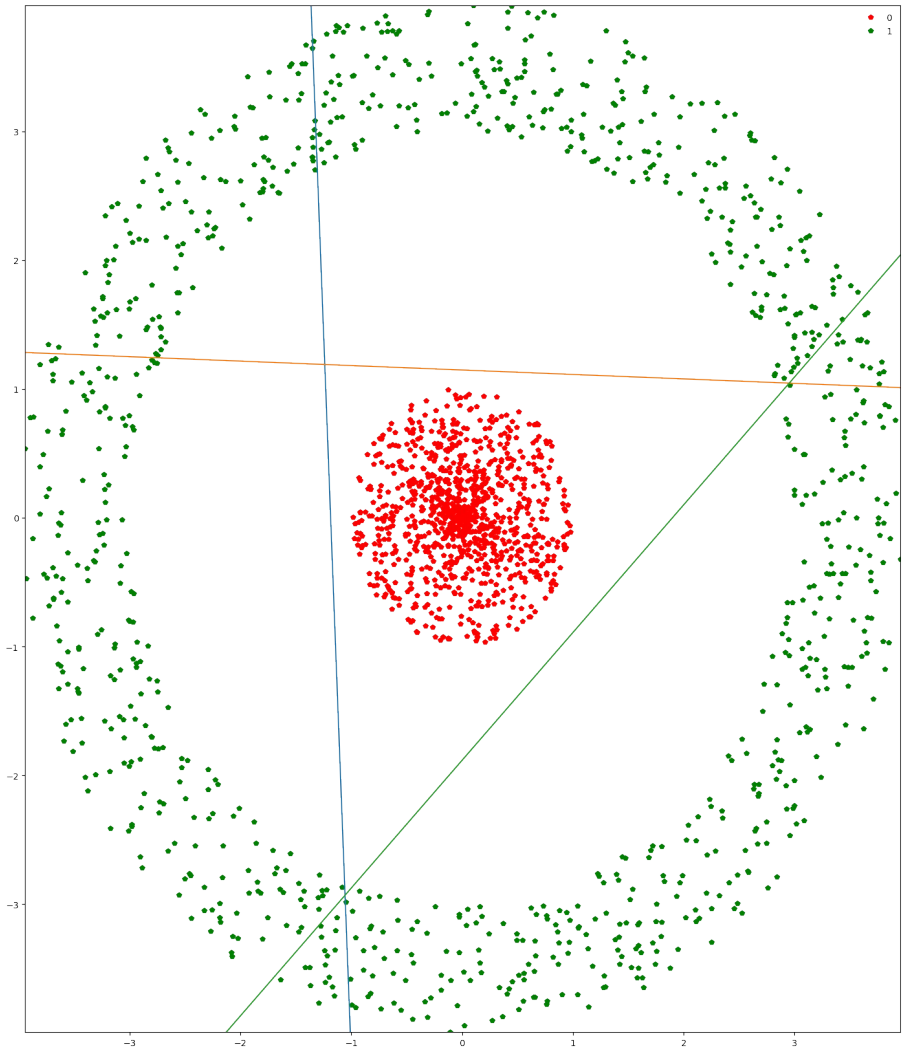
Map 2-dim input to 3-dim space

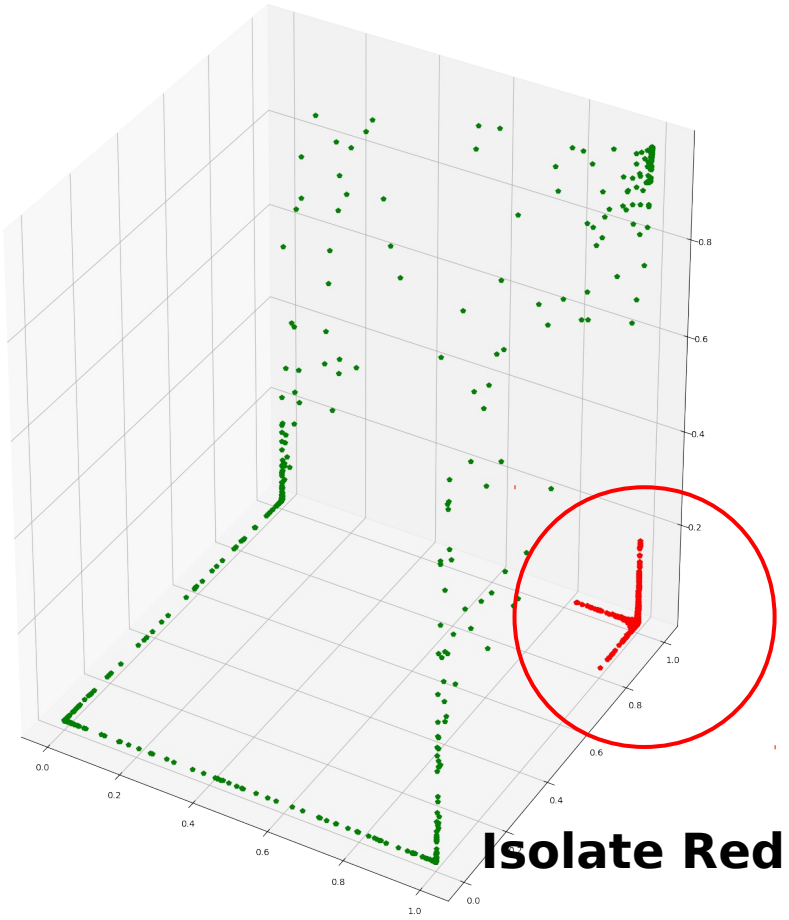




Red Hat

Map 2-dim input to 3-dim space

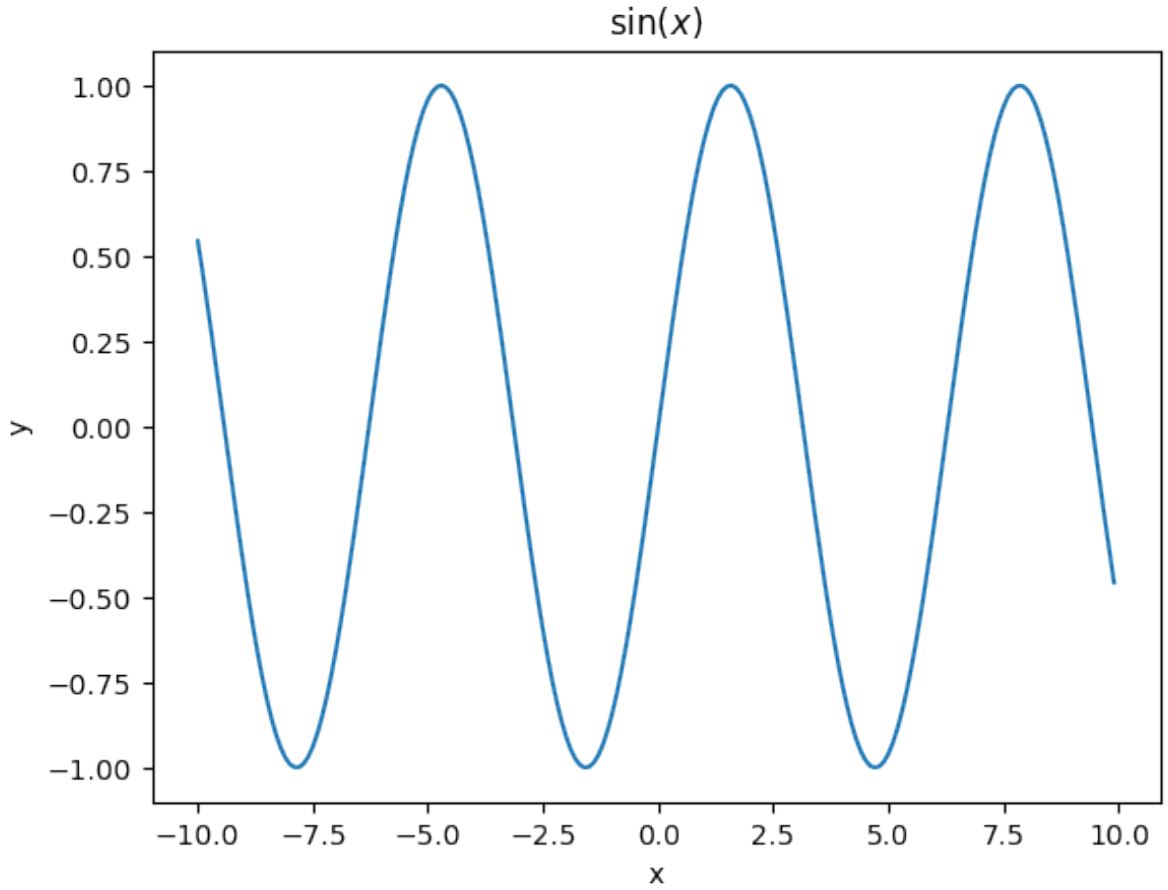




Isolate Red in corner

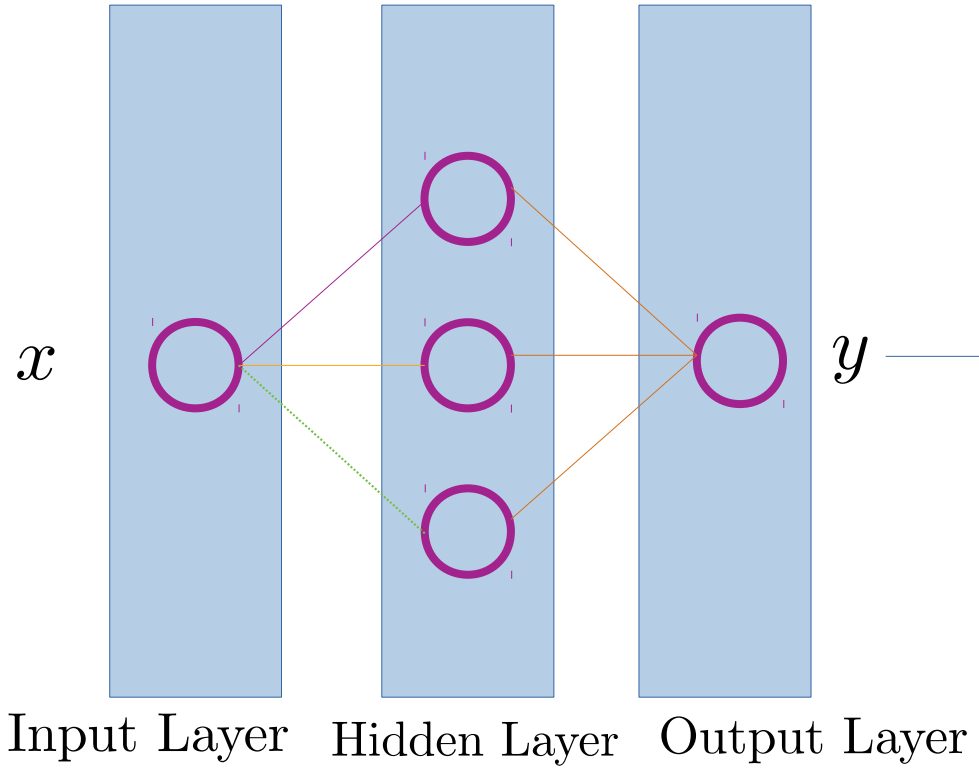
Red Hat

A Second Example: Regression



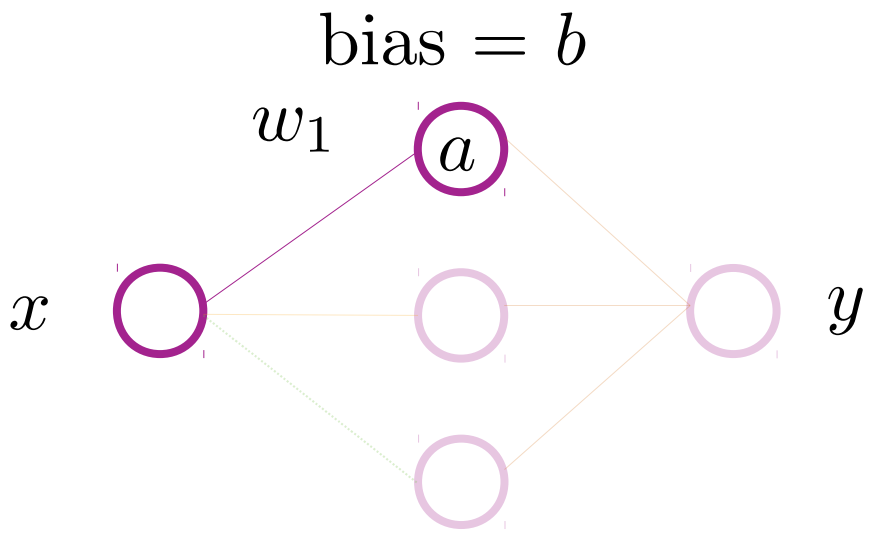
$$\text{Given } x, \text{ predict y } = \sin(x)$$

Red Hat



Minimize:
Mean Squared Error

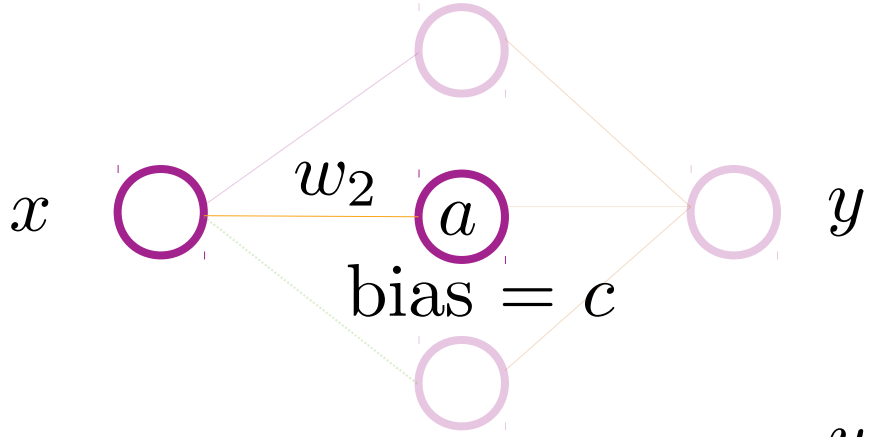
Activation = Sigmoid
Output = Linear



$$\boxed{a = \sigma(w_1 x + b)}$$

Decision boundary:

$$w_1 x + b = 0 \implies x = -\frac{b}{w_1}$$

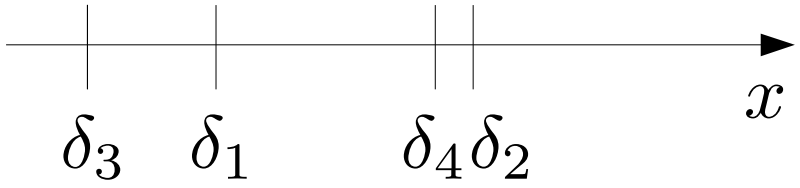


$$\boxed{a = \sigma(w_2 x + c)}$$

Decision boundary:

$$w_2 x + c = 0 \implies x = -\frac{c}{w_2}$$

1 decision boundary per hidden node



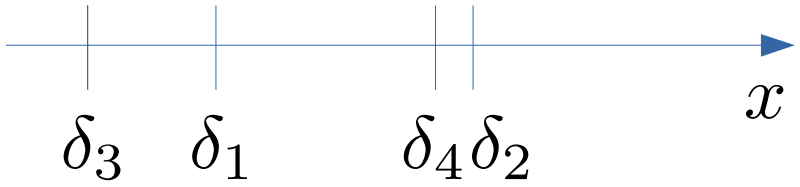
$$\delta_1 = -\frac{b_1}{w_1}$$

$$\delta_2 = -\frac{b_2}{w_2}$$

$$\vdots$$

$$\delta_n = -\frac{b_n}{w_n}$$

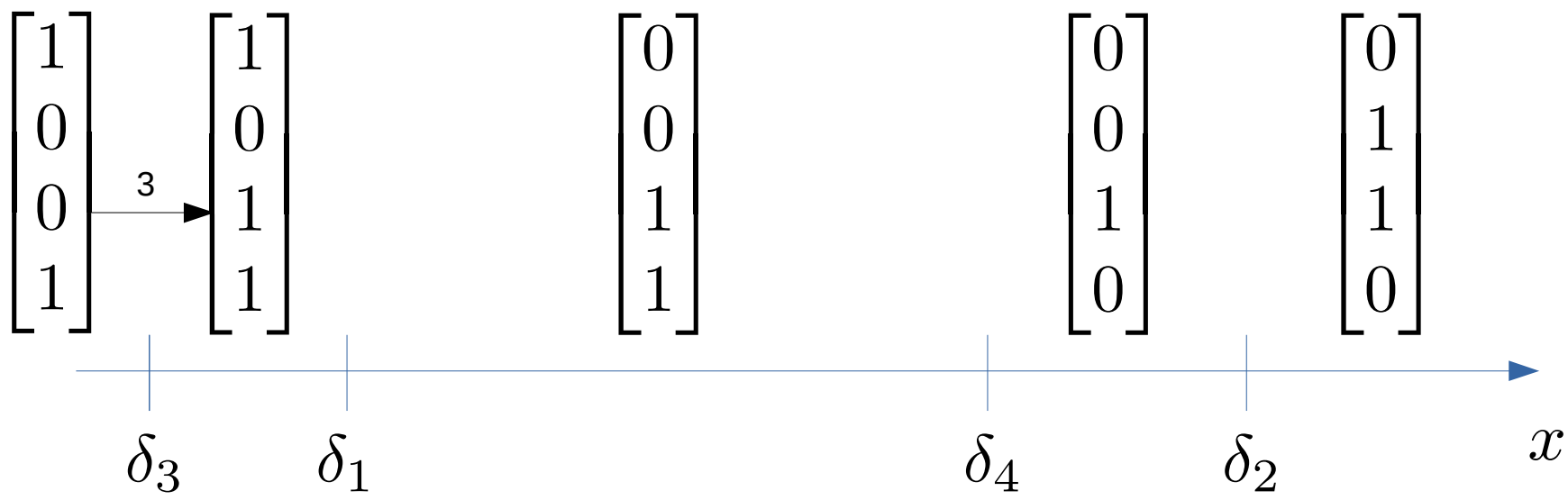
Walk from left to right



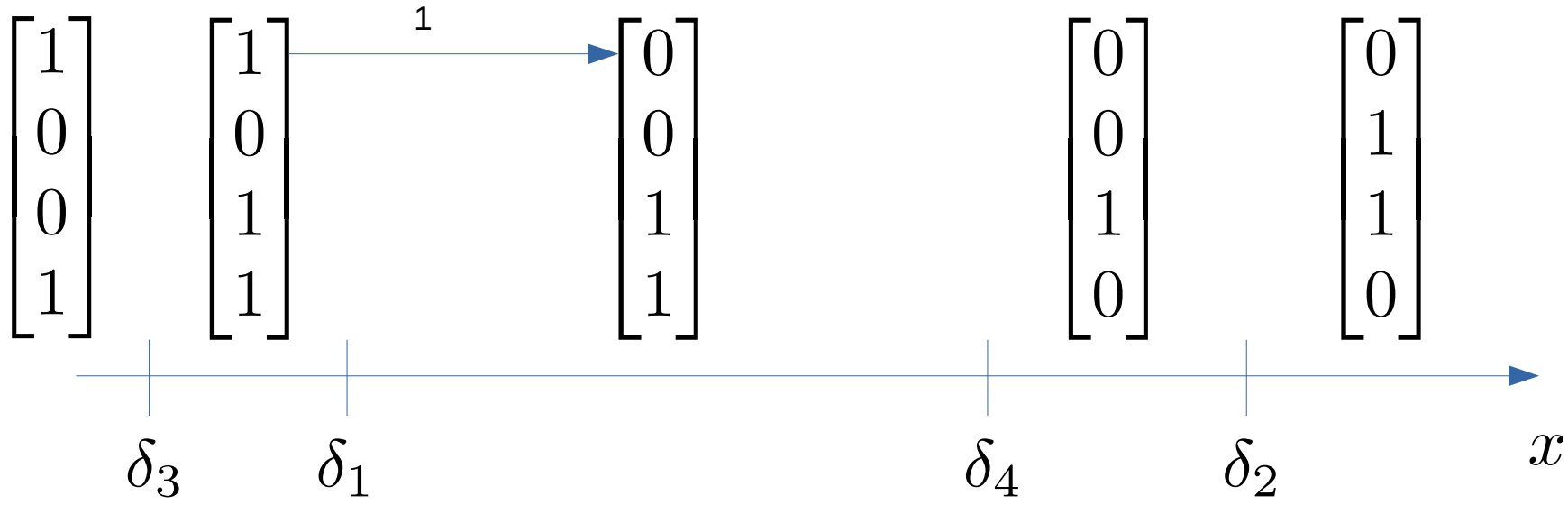
Each time you cross a boundary, a hidden node either:

Turns on i.e. goes from 1 → 0

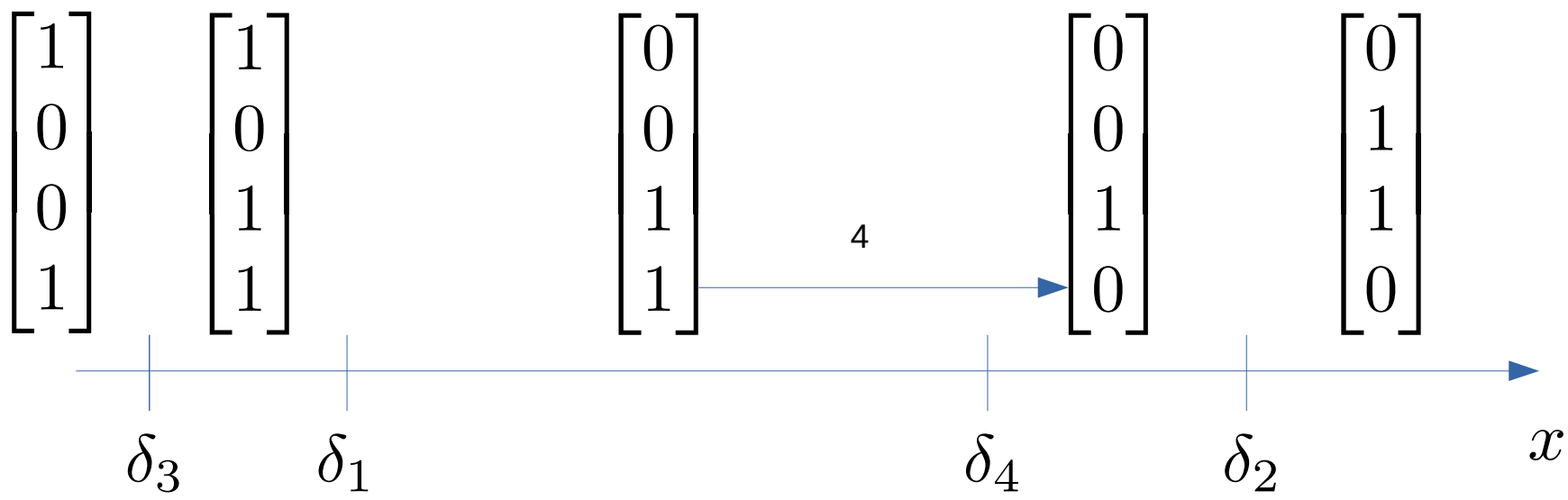
Turns off i.e. goes from 0 → 1

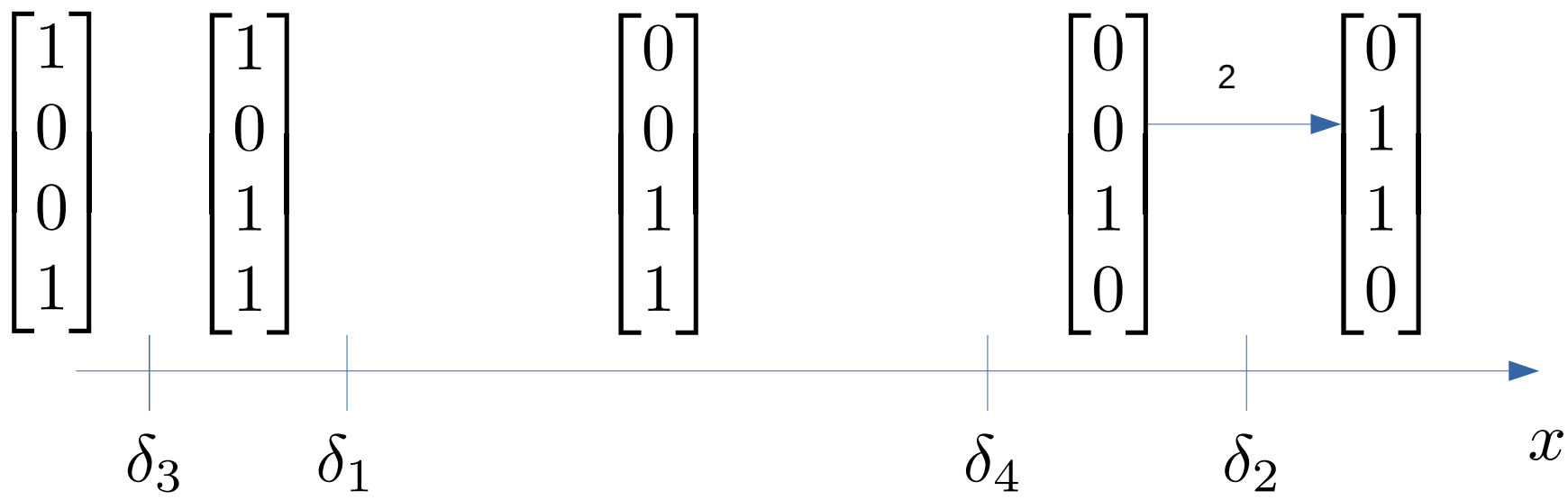


Red Hat

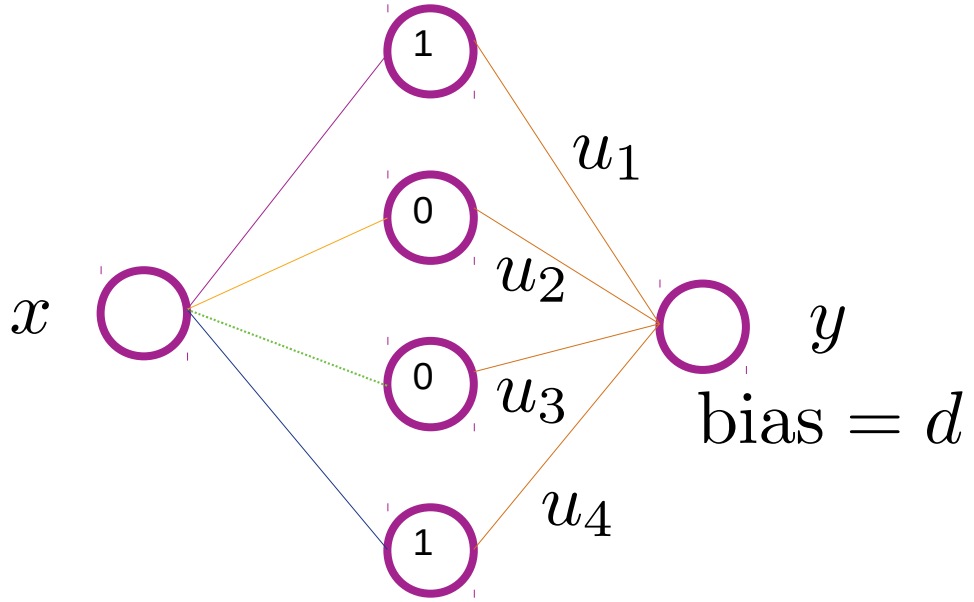

1
$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
δ_3
δ_1
δ_4
δ_2
x

Red Hat

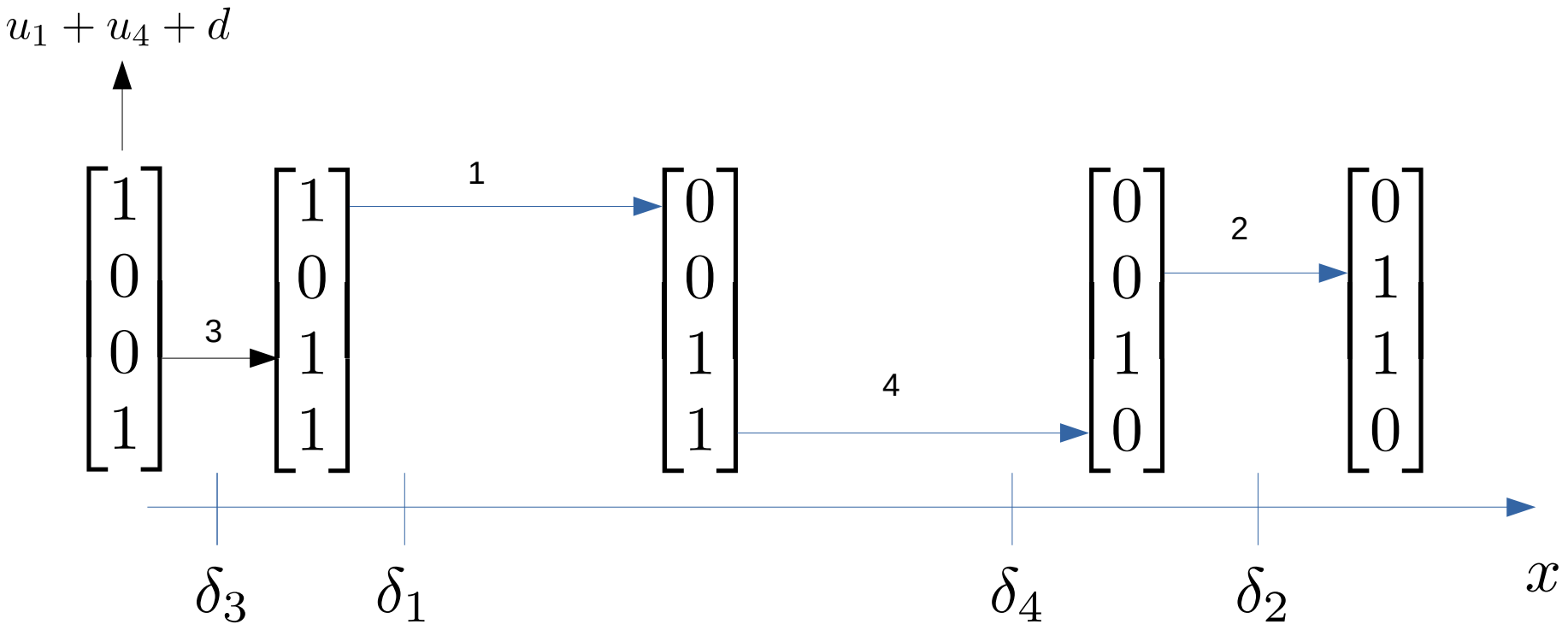

$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
4
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
δ_3
δ_1
δ_4
δ_2
x

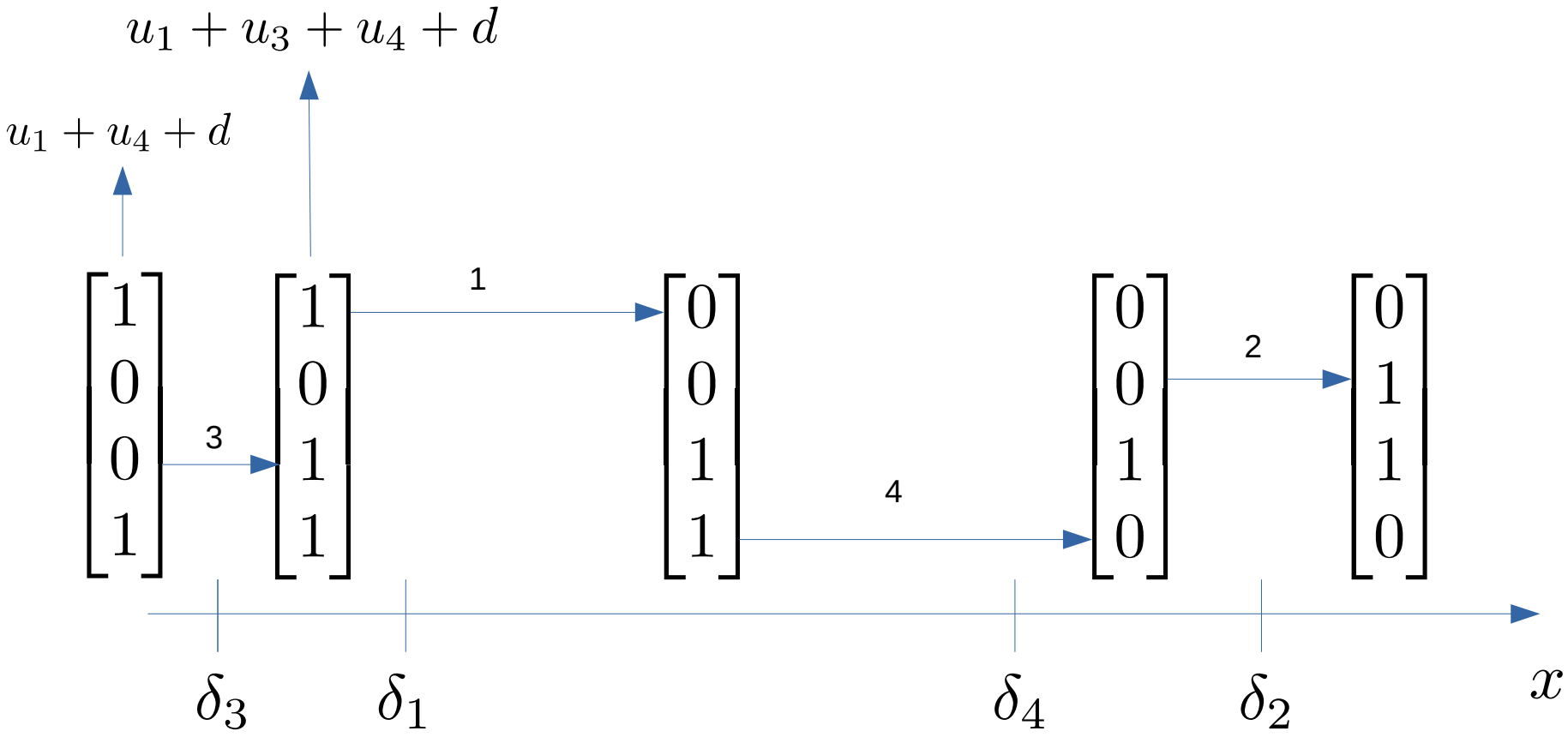

$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
2
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
δ_3
δ_1
δ_4
δ_2
x

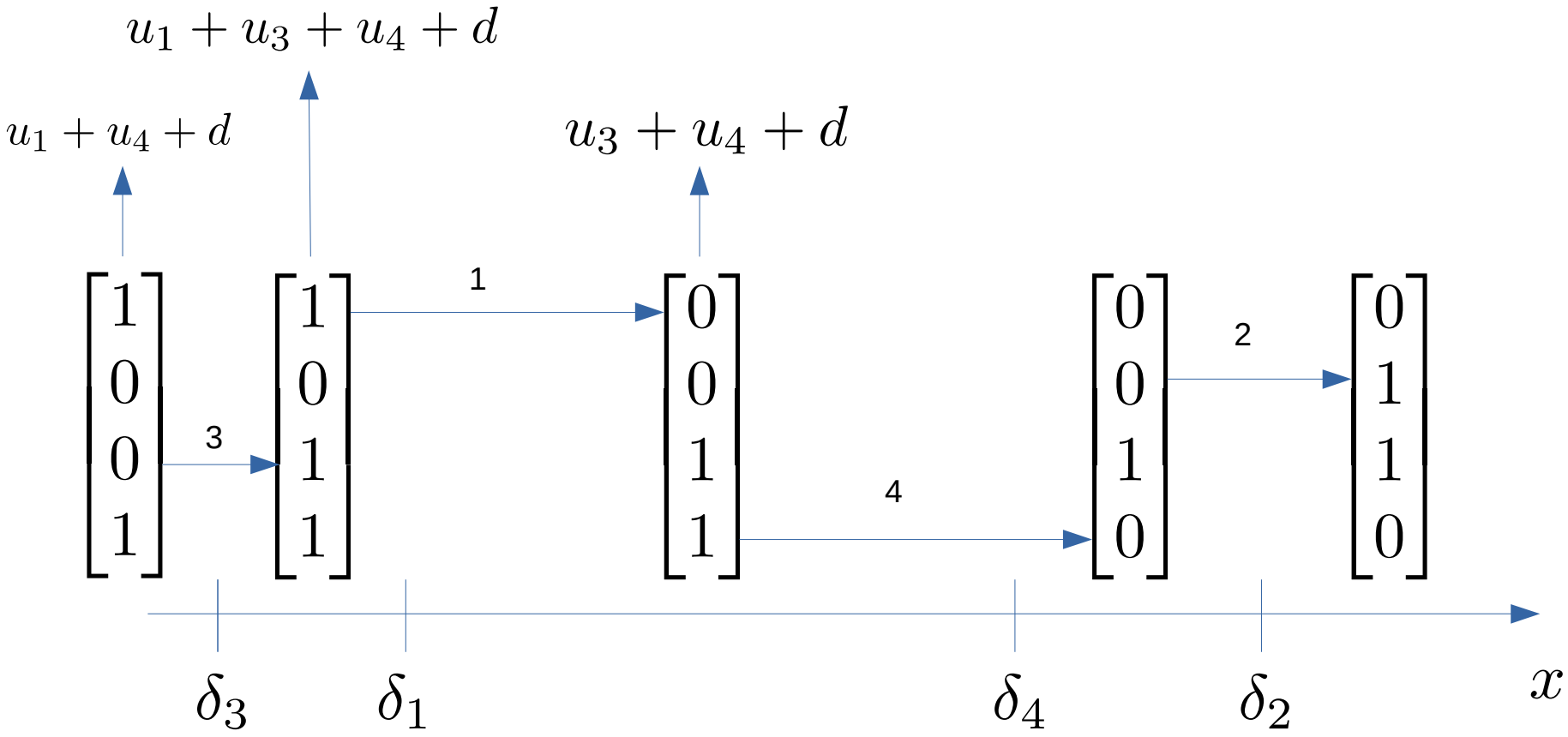
Red Hat

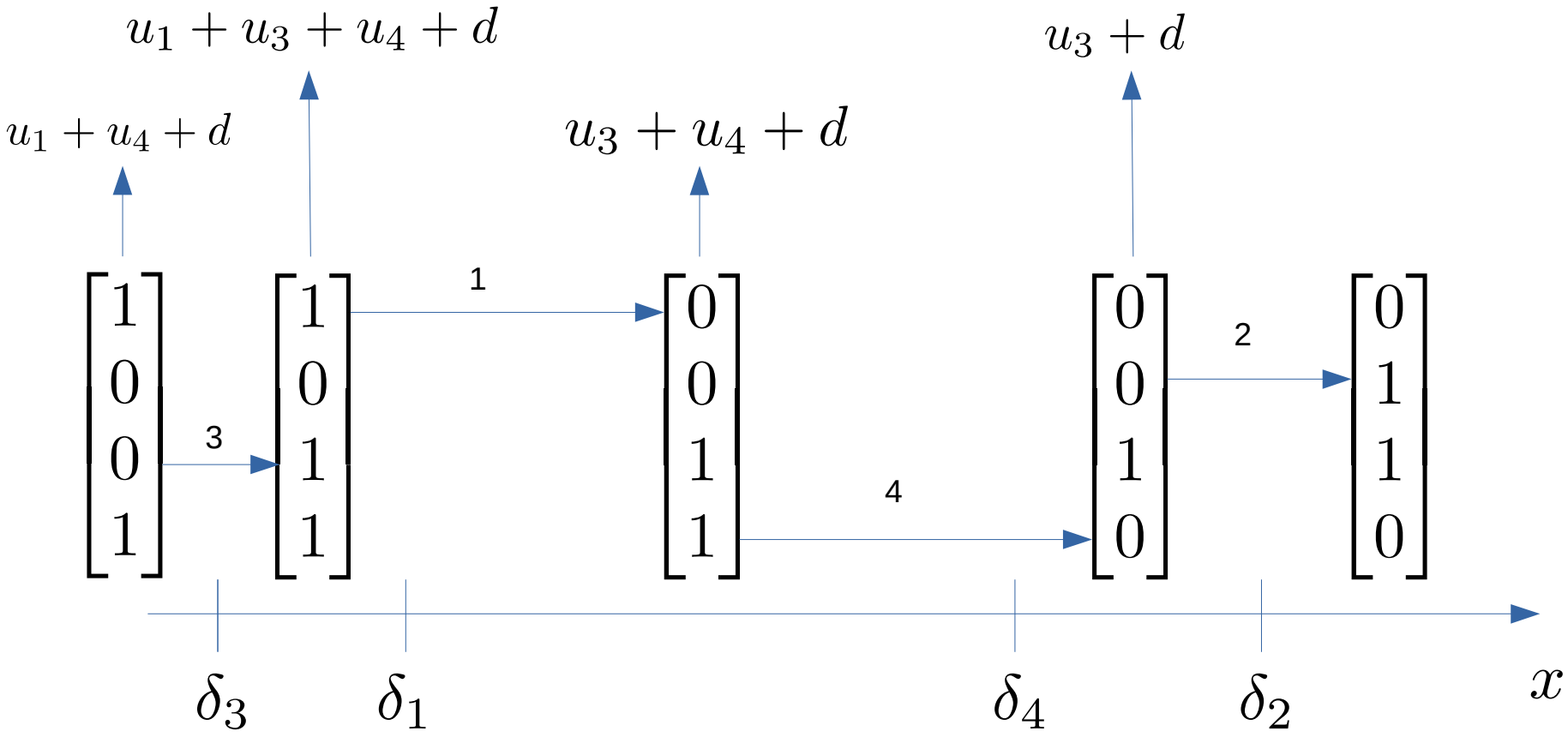


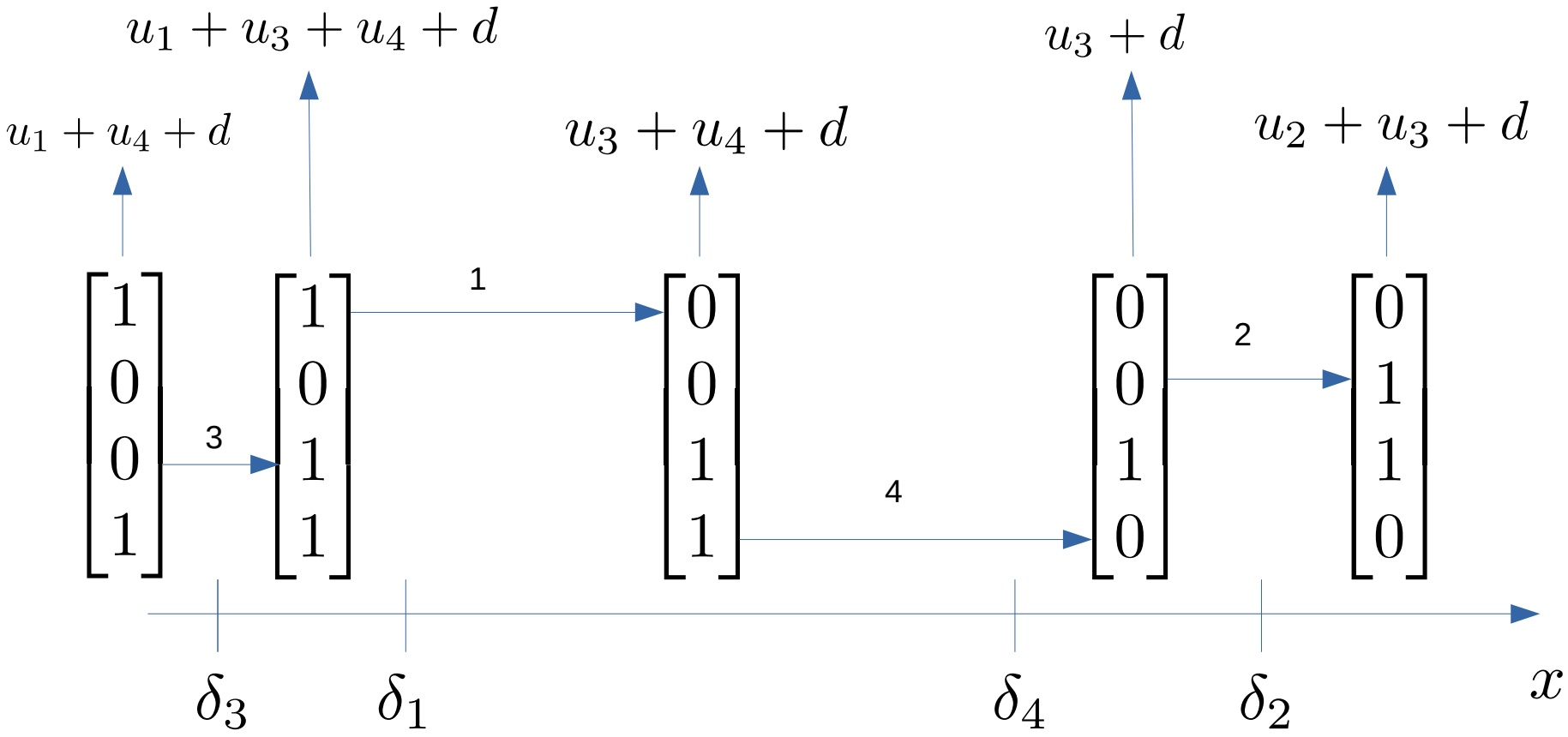
$$\begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow 1 * u_1 + 0 * u_2 + 0 * u_3 + 1 * u_4 + d = \boxed{u_1 + u_4 + d}$$

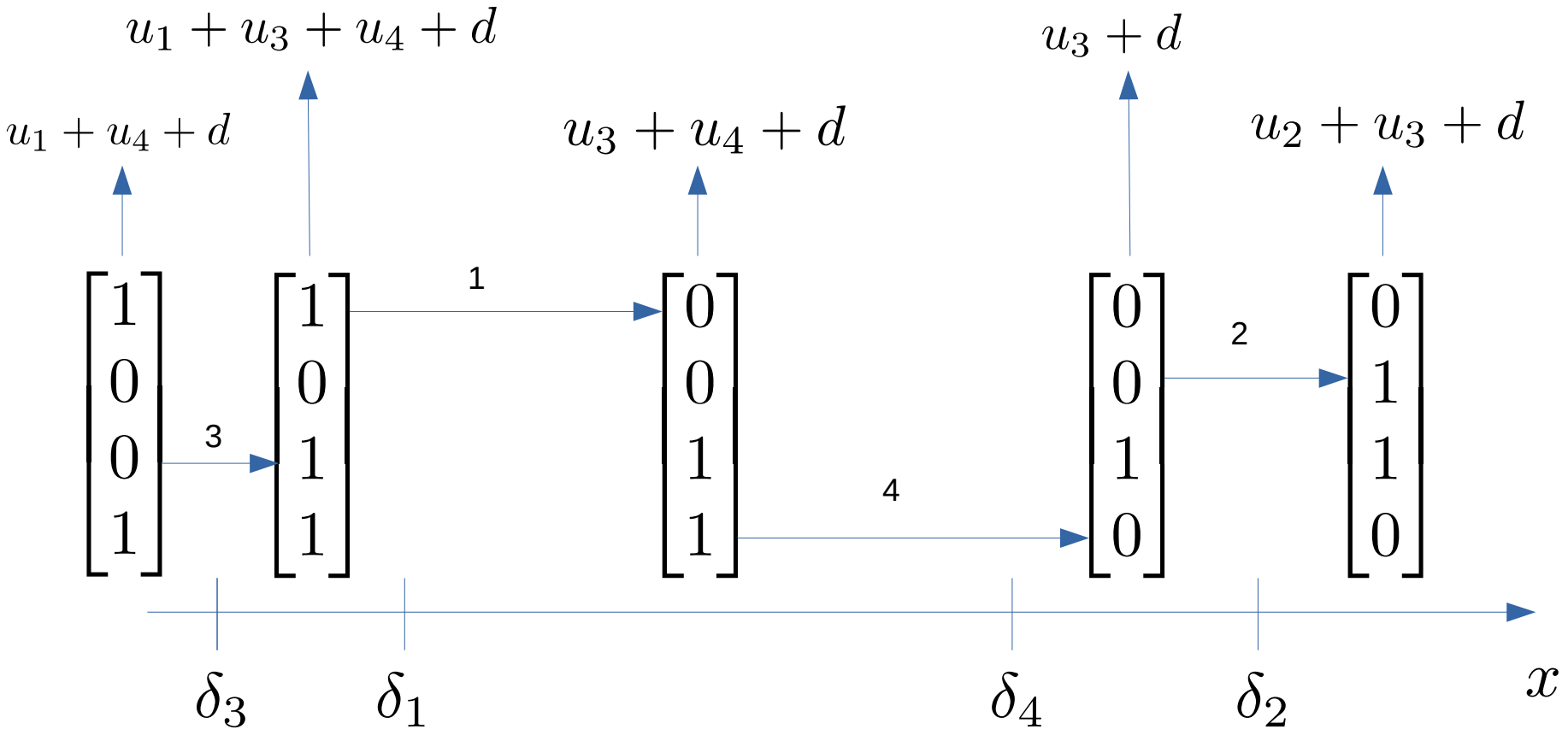

$u_1 + u_4 + d$
$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
3
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
1
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
4
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
2
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
δ_3
δ_1
δ_4
δ_2
x


$u_1 + u_3 + u_4 + d$
$u_1 + u_4 + d$
$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
3
1
4
2
δ_3
δ_1
δ_4
δ_2
x



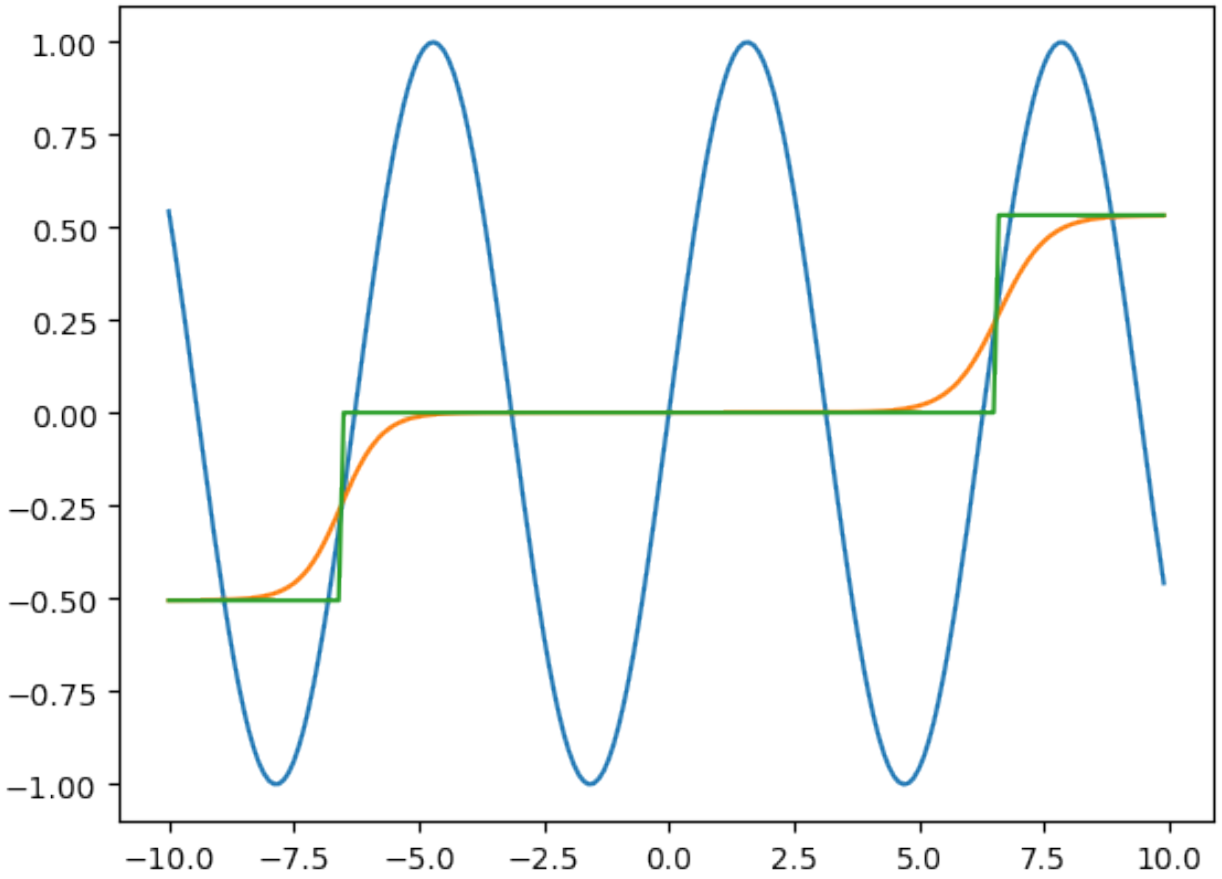



$u_1 + u_3 + u_4 + d$
$u_3 + d$
$u_1 + u_4 + d$
$u_3 + u_4 + d$
$u_2 + u_3 + d$
1
2
3
4
δ_3
δ_1
δ_4
δ_2
x



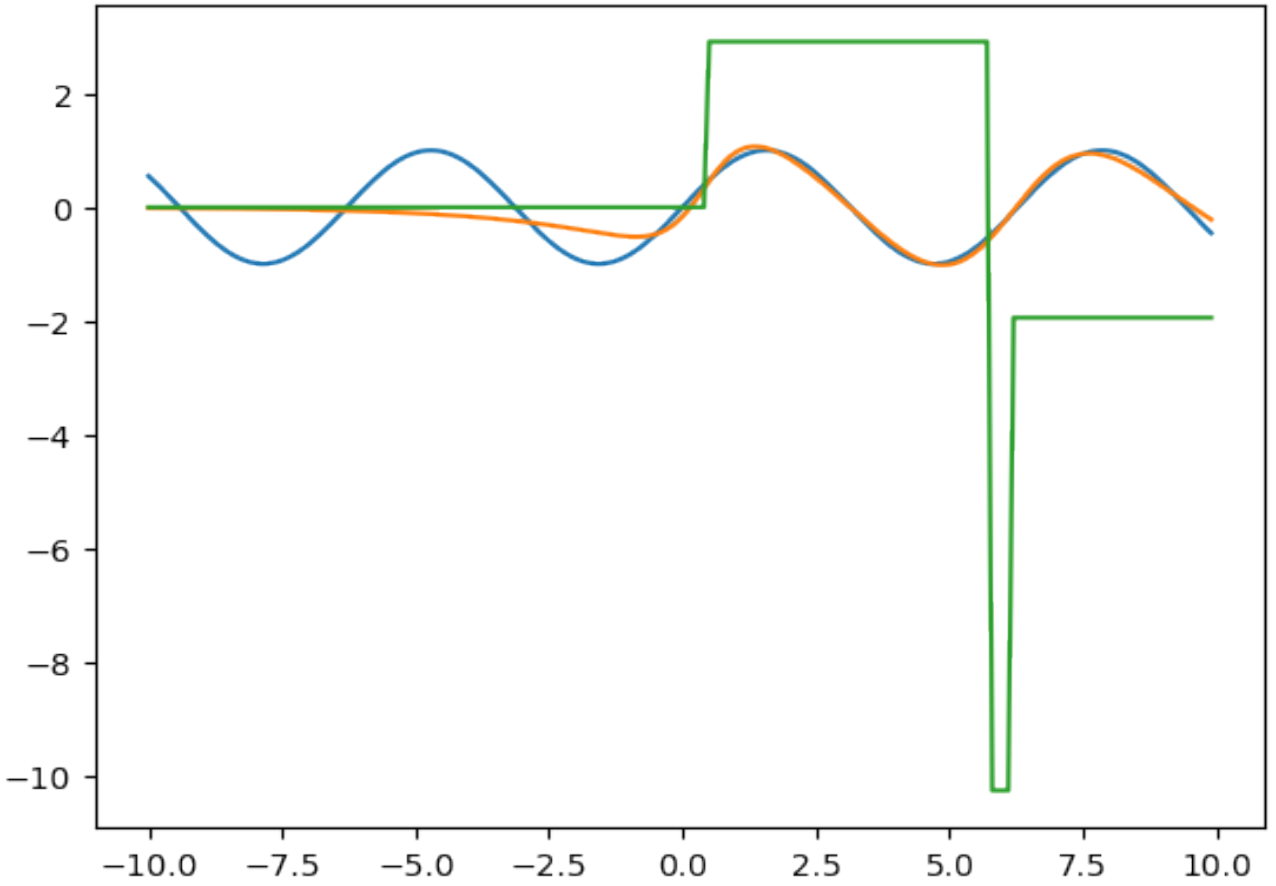
Five distinct regions where NN predicts constant values with sigmoid turn on and turn off

Hidden Nodes = 2



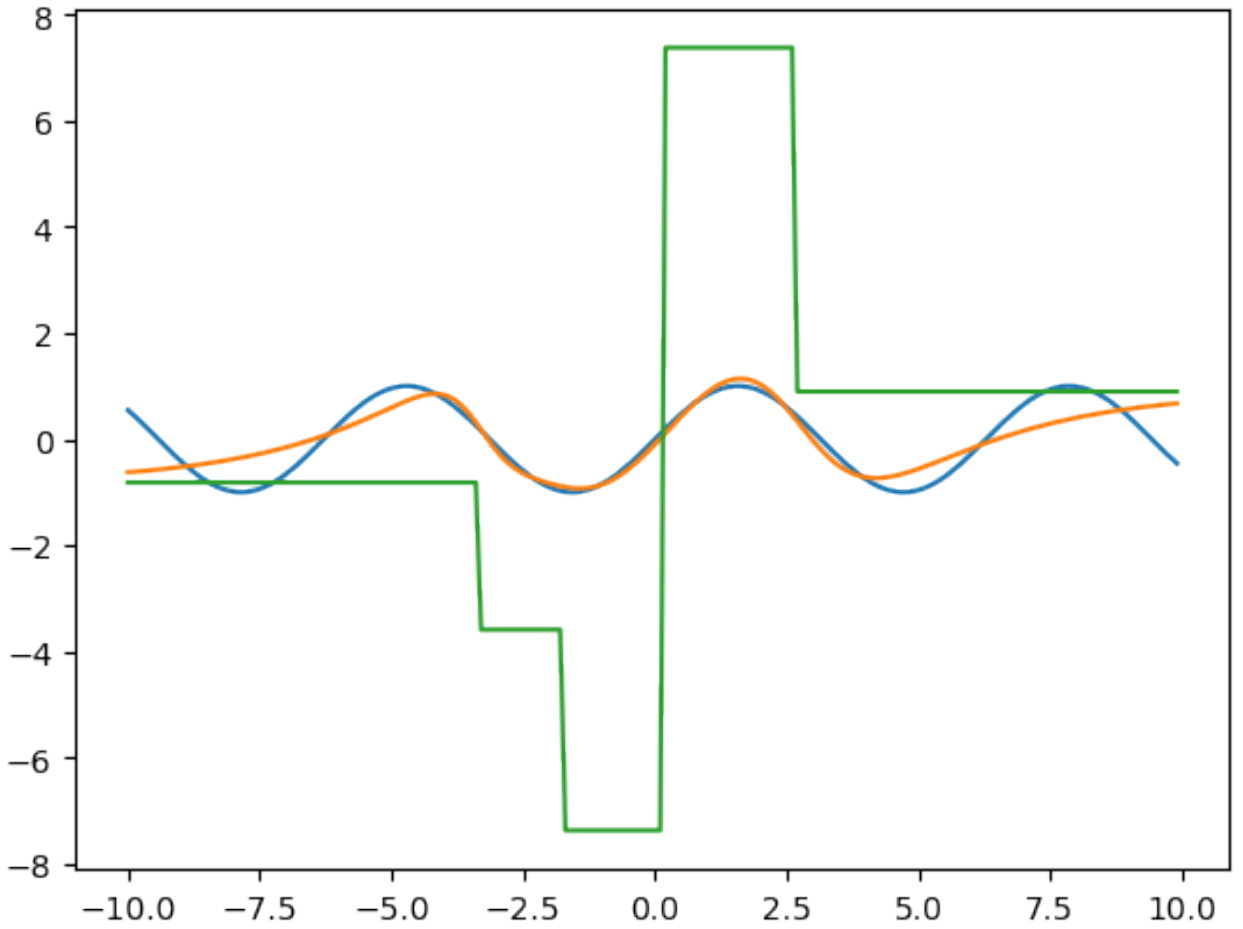
Red Hat

Hidden Nodes = 3

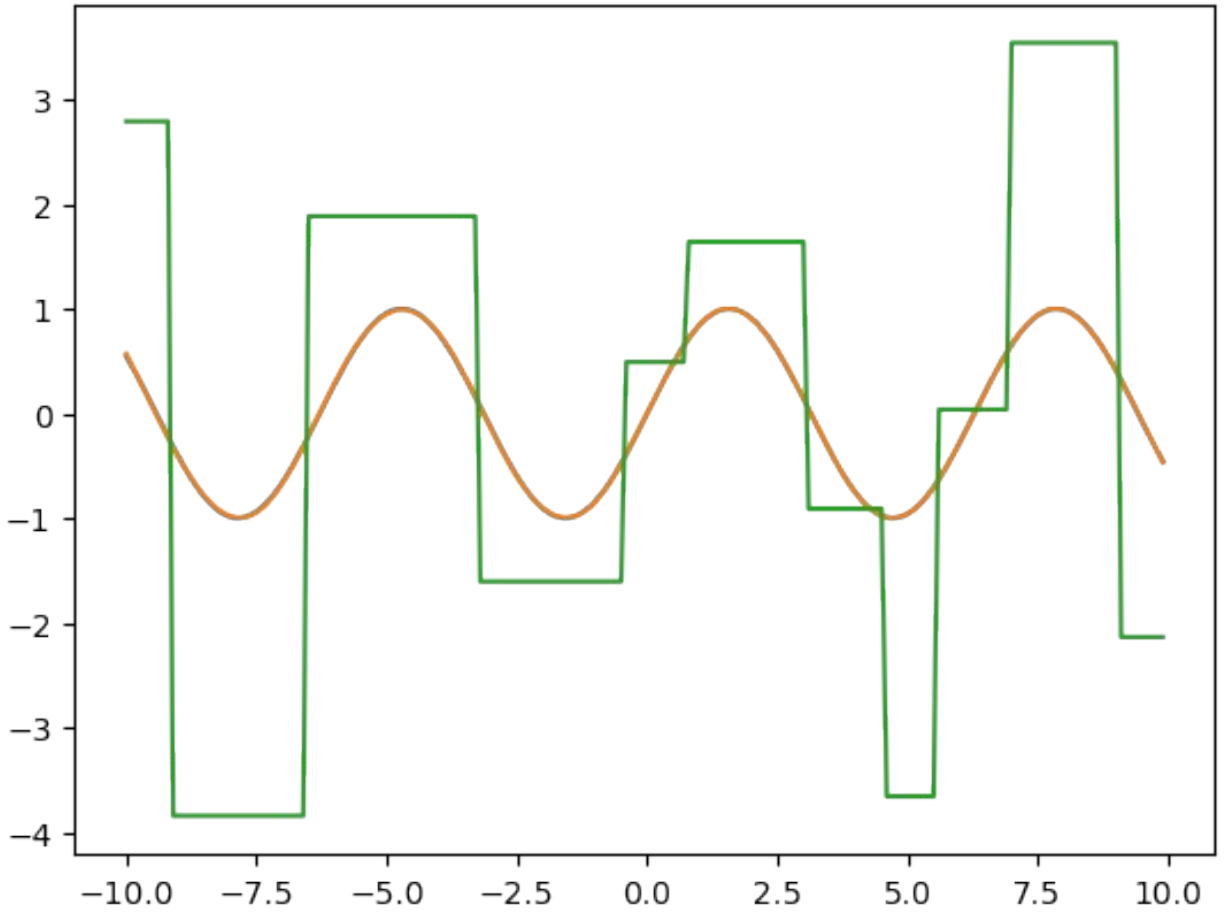


Red Hat

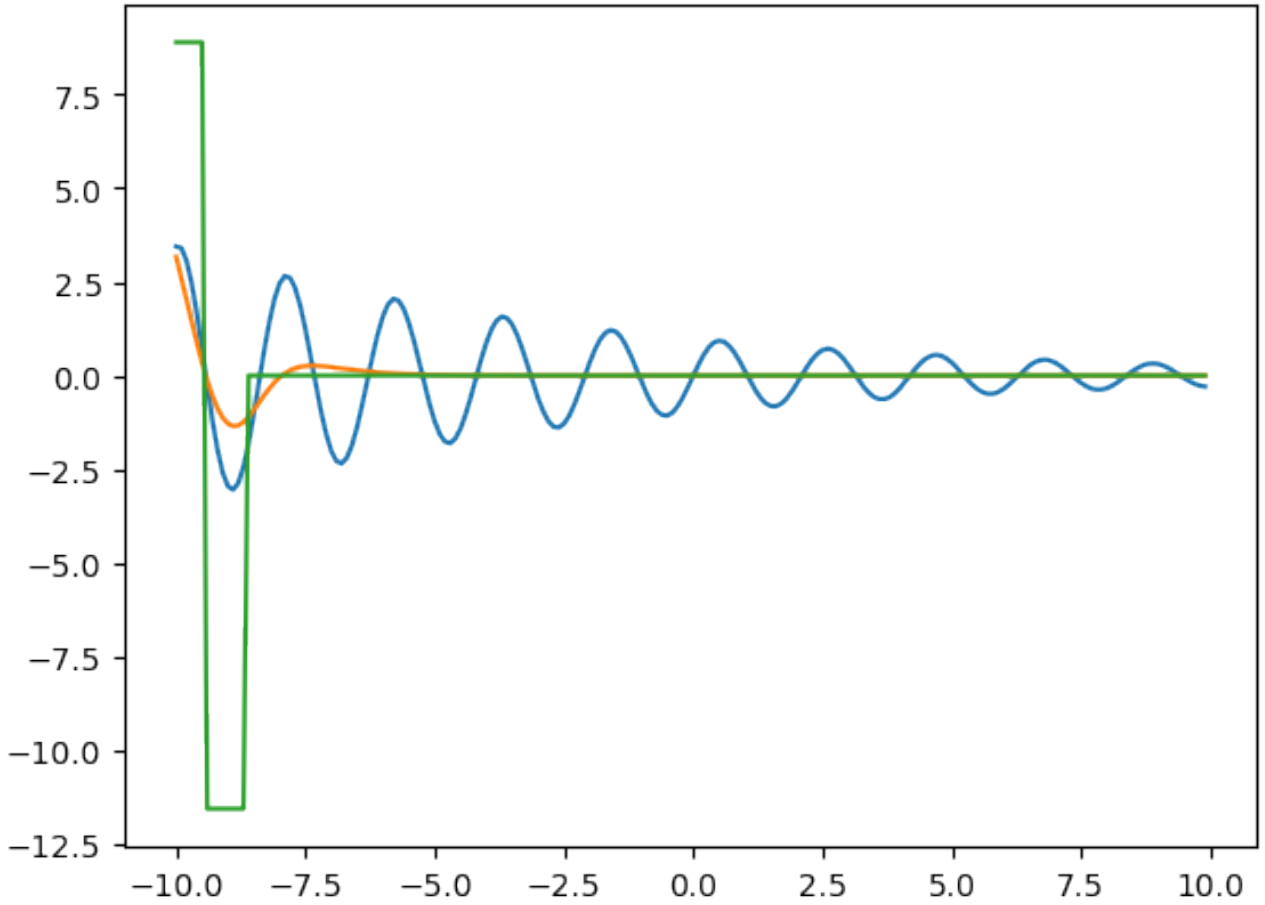
Hidden Nodes = 4



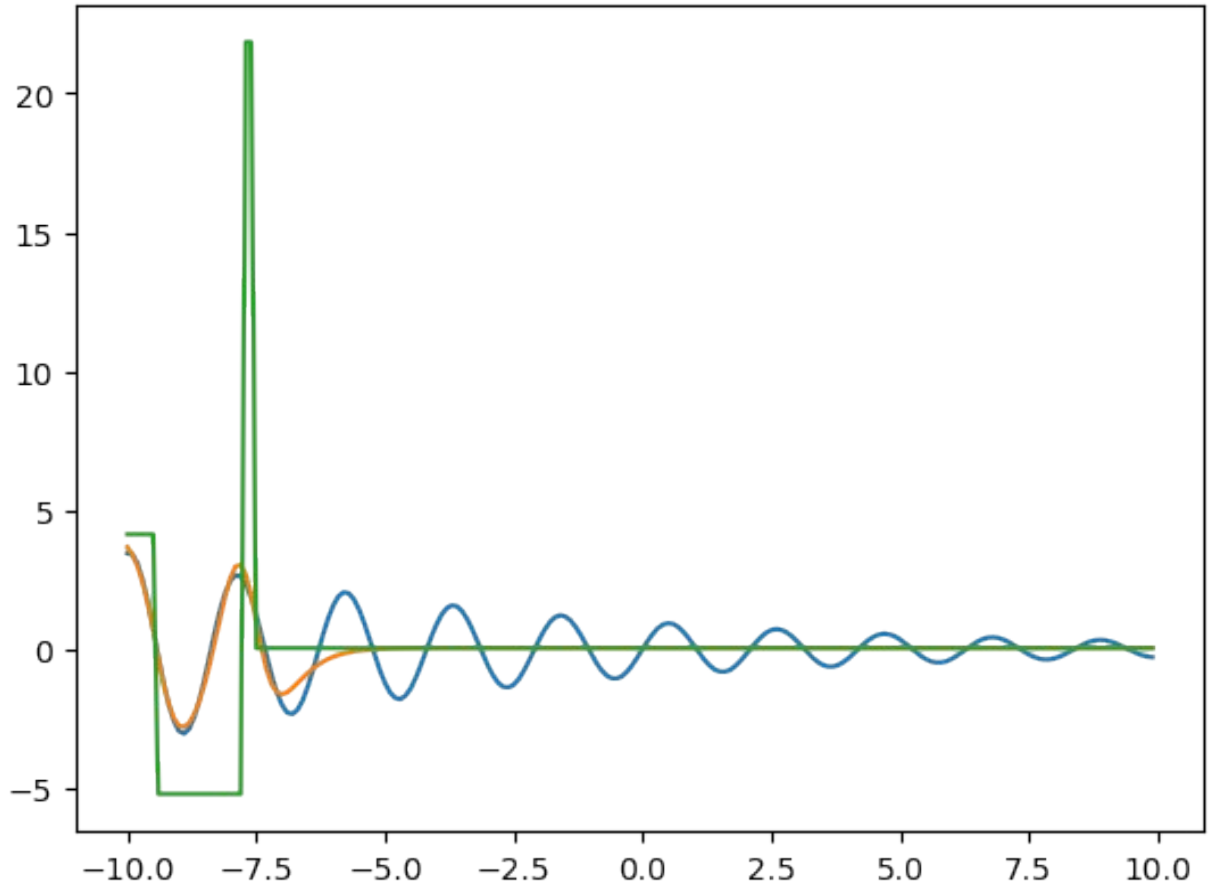
Hidden Nodes = 10



Hidden Nodes = 2

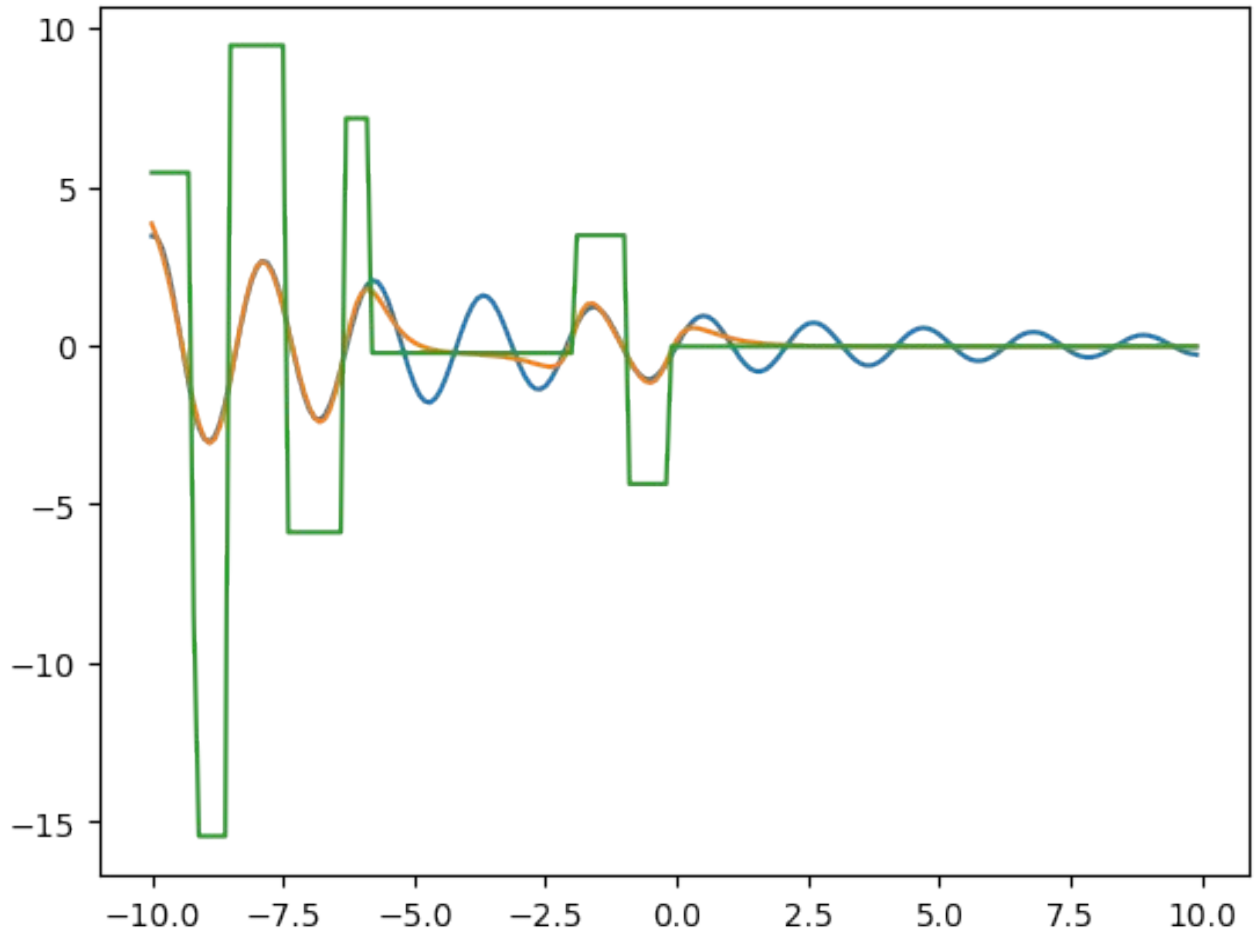


Hidden Nodes = 3



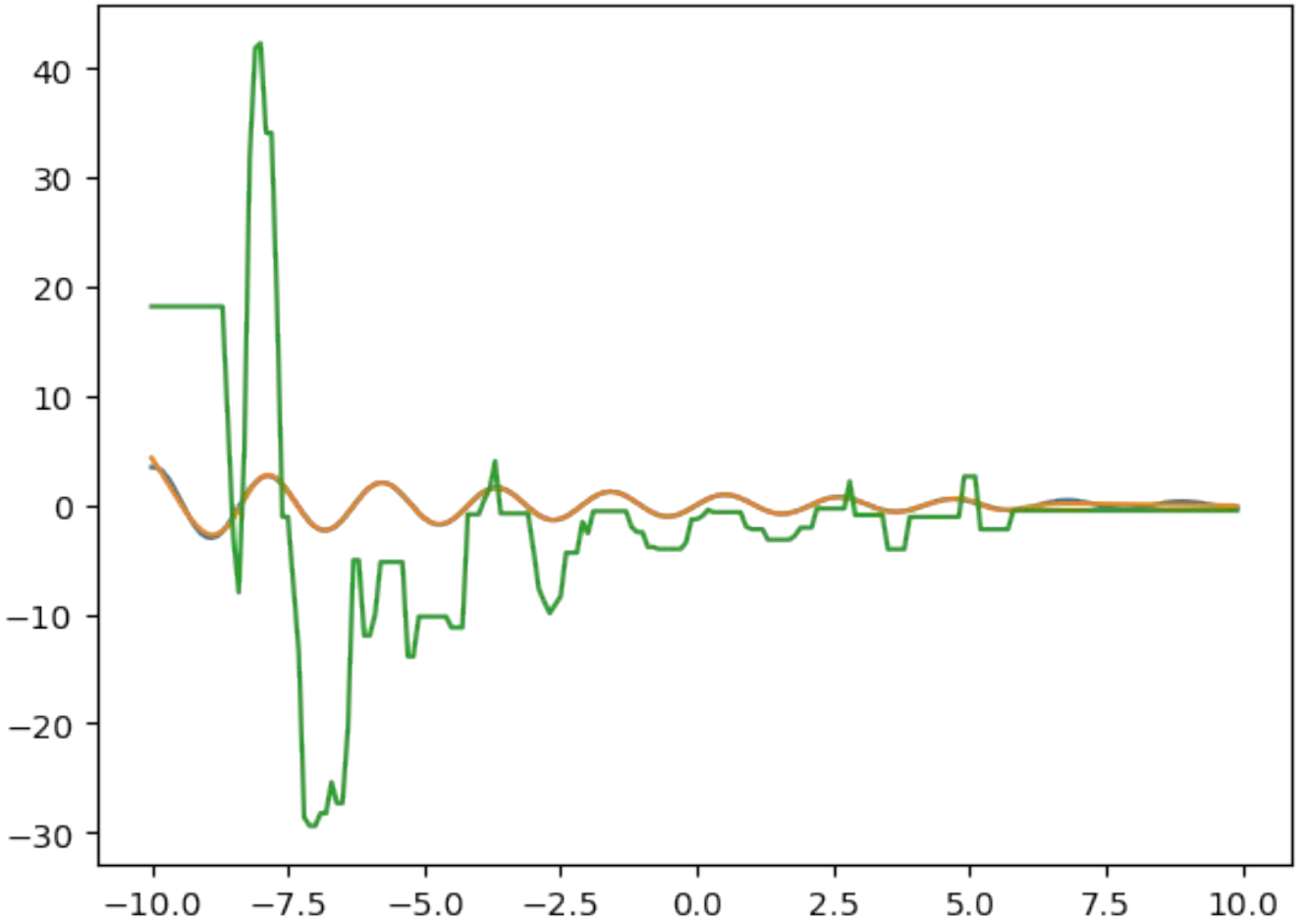
Red Hat

Hidden Nodes = 10



Red Hat

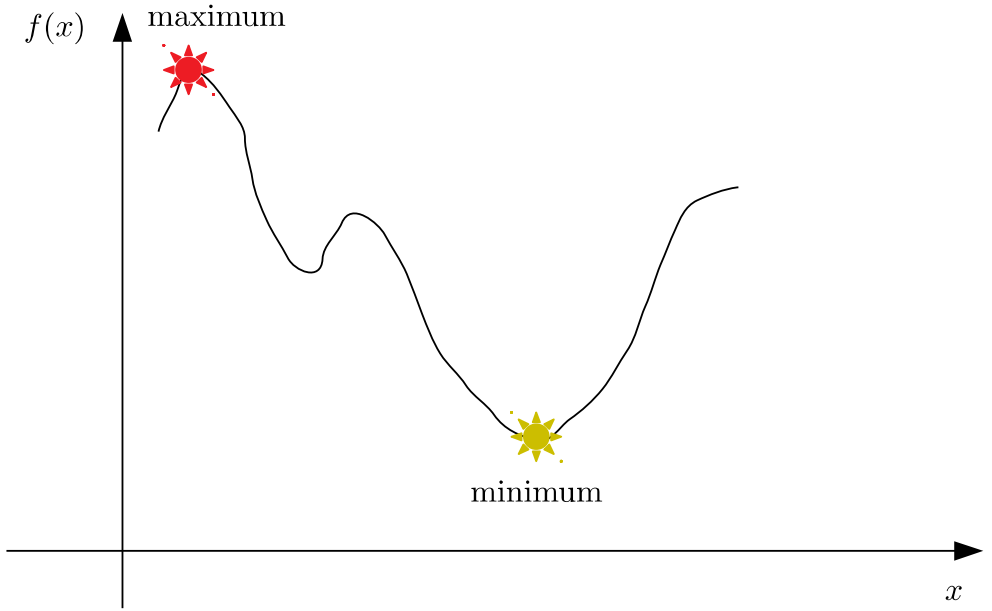
Hidden Nodes = 100



Mathematical Optimization

What is Optimization?

Mathematical and numerical techniques to find the maximum or minimum of a function



Red Hat

Terminology

Global maximum: the maximum value the function takes across its domain

Local maximum: the maximum value of the function in a small neighborhood around an x (input value)

Similar definitions for minima

Terminology

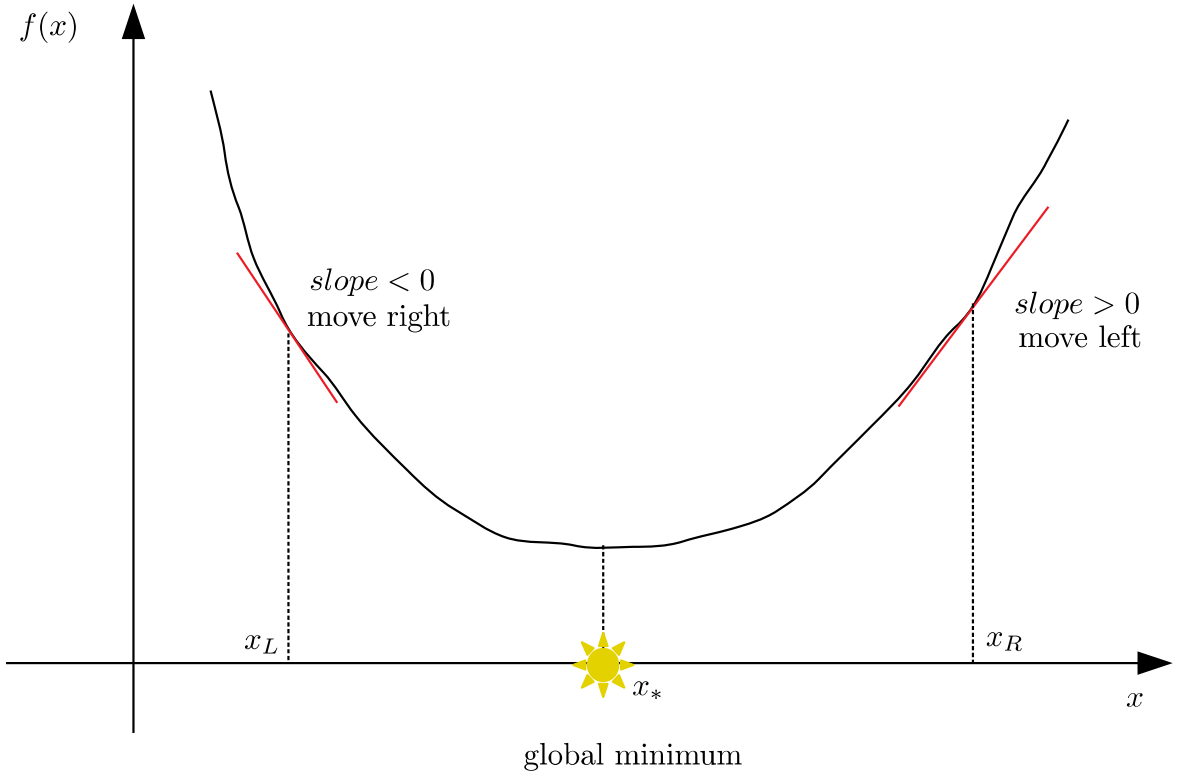
Training: Finding weights that minimize the loss function on the train set for a neural network is an optimization problem

Hyperparameter tuning: Finding the number of layers, the activation function, the number of nodes in each layer, and other parameters is also an optimization problem

Obstacles

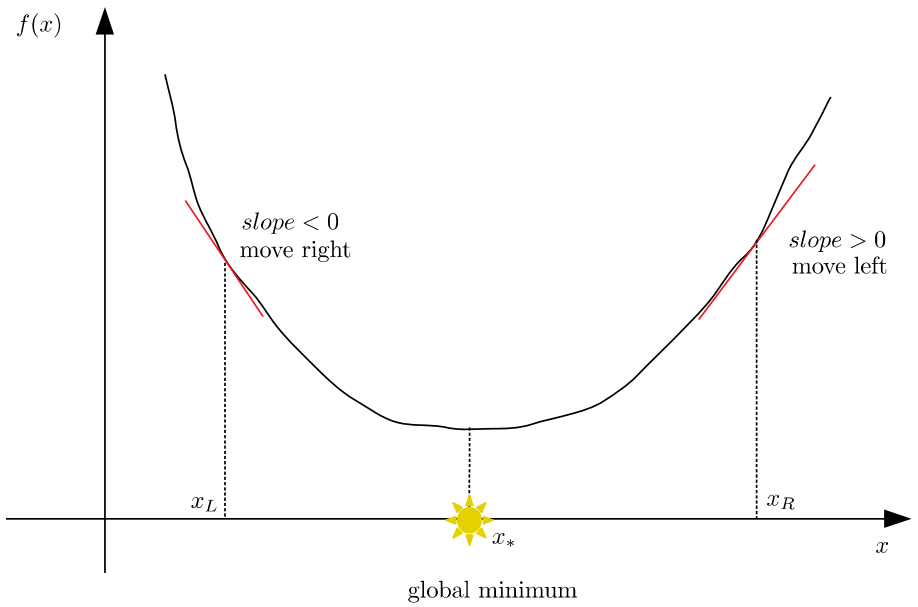
- f might be computationally expensive to evaluate.
- f might be discrete so no way to evaluate derivatives which can guide search for minima
- Even if f is continuous and well-behaved, evaluating higher derivatives (second, third etc.) is very expensive.
- f might have very complex structure with multiple (possibly infinite) local minima
- f might be very high-dimensional i.e. it has a large number of inputs and hence we are searching for the minima in a high-dimensional space with many more directions to explore.

Gradient Descent



Red Hat

Gradient Descent



Iterative method

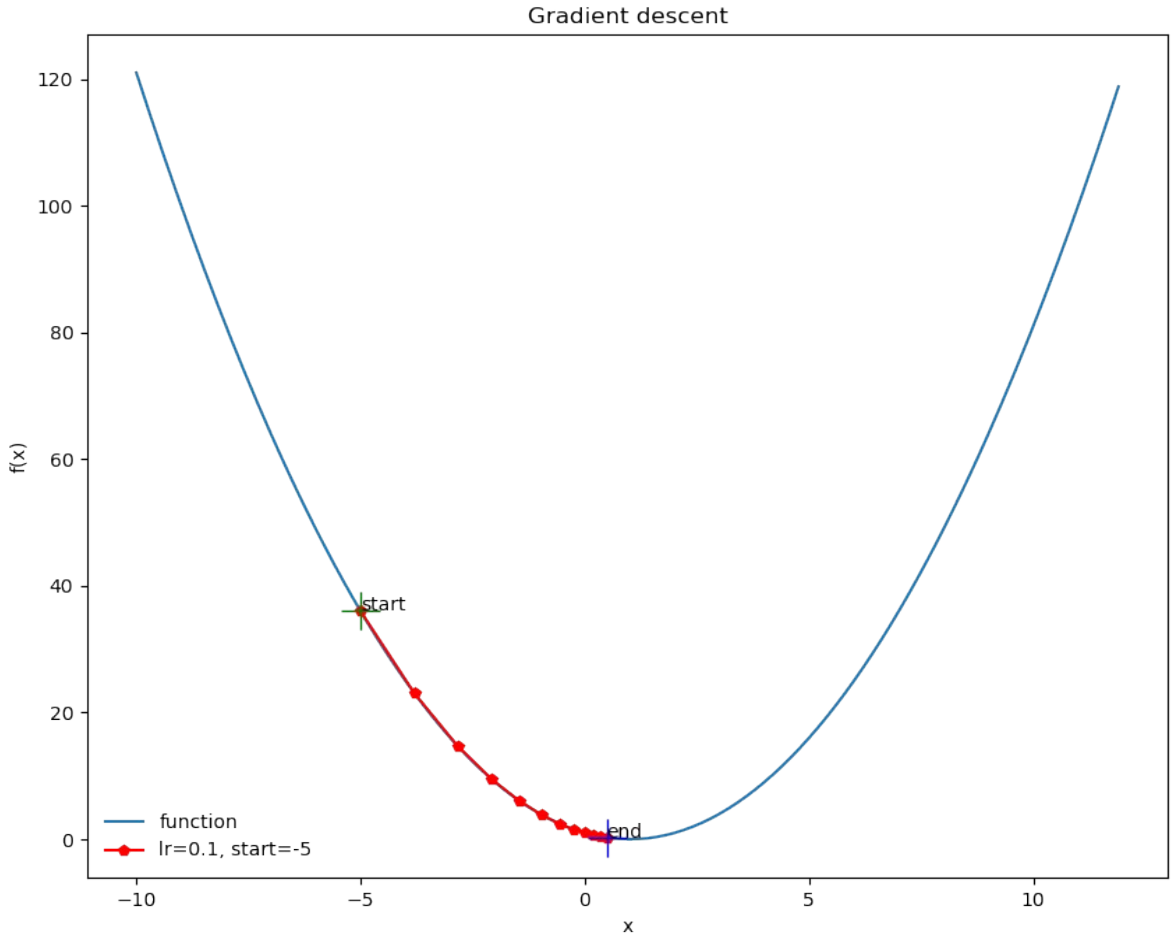
Start at some reasonable point and keep updating

$$x^{(t+1)} = x^{(t)} + \text{update}$$

$$x^{(t+1)} = x^{(t)} - \eta \frac{df}{dx}(x^{(t)})$$

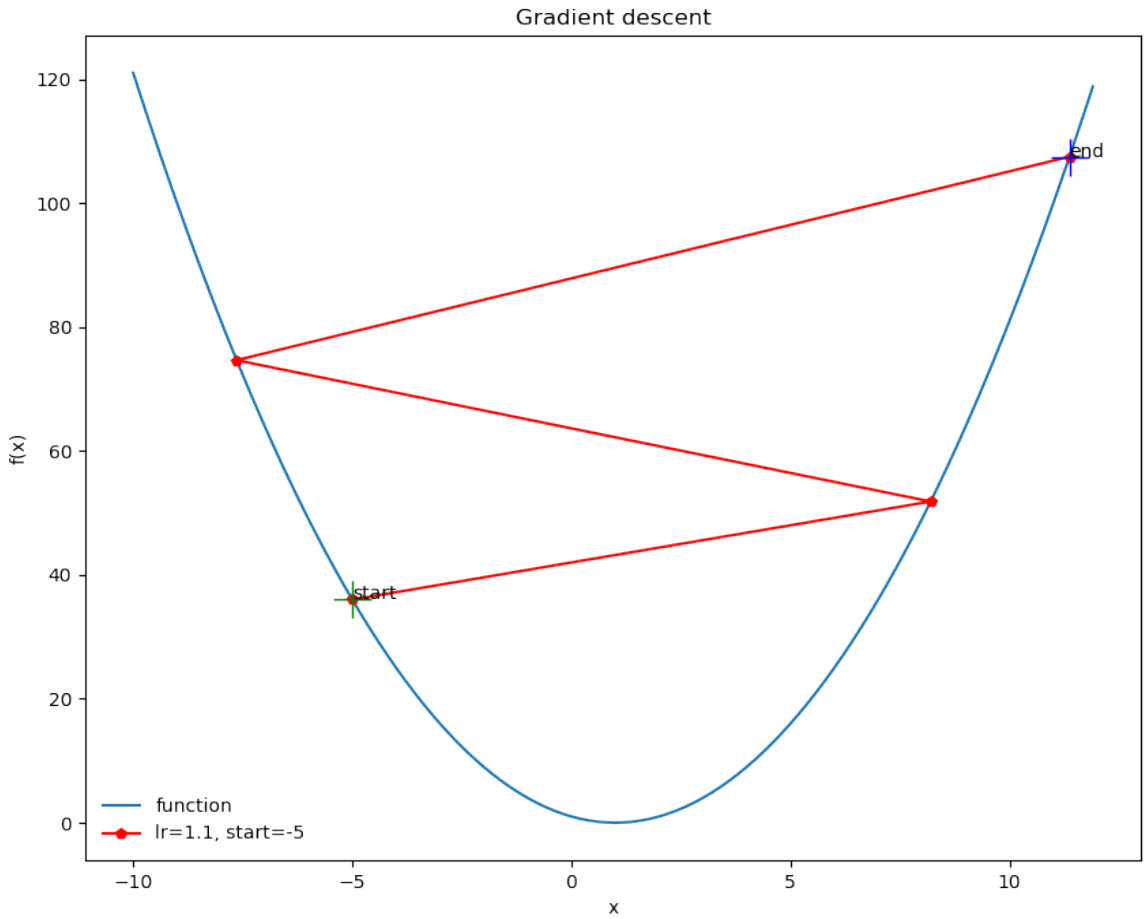
$$\eta = \text{learning rate}$$

Intuition



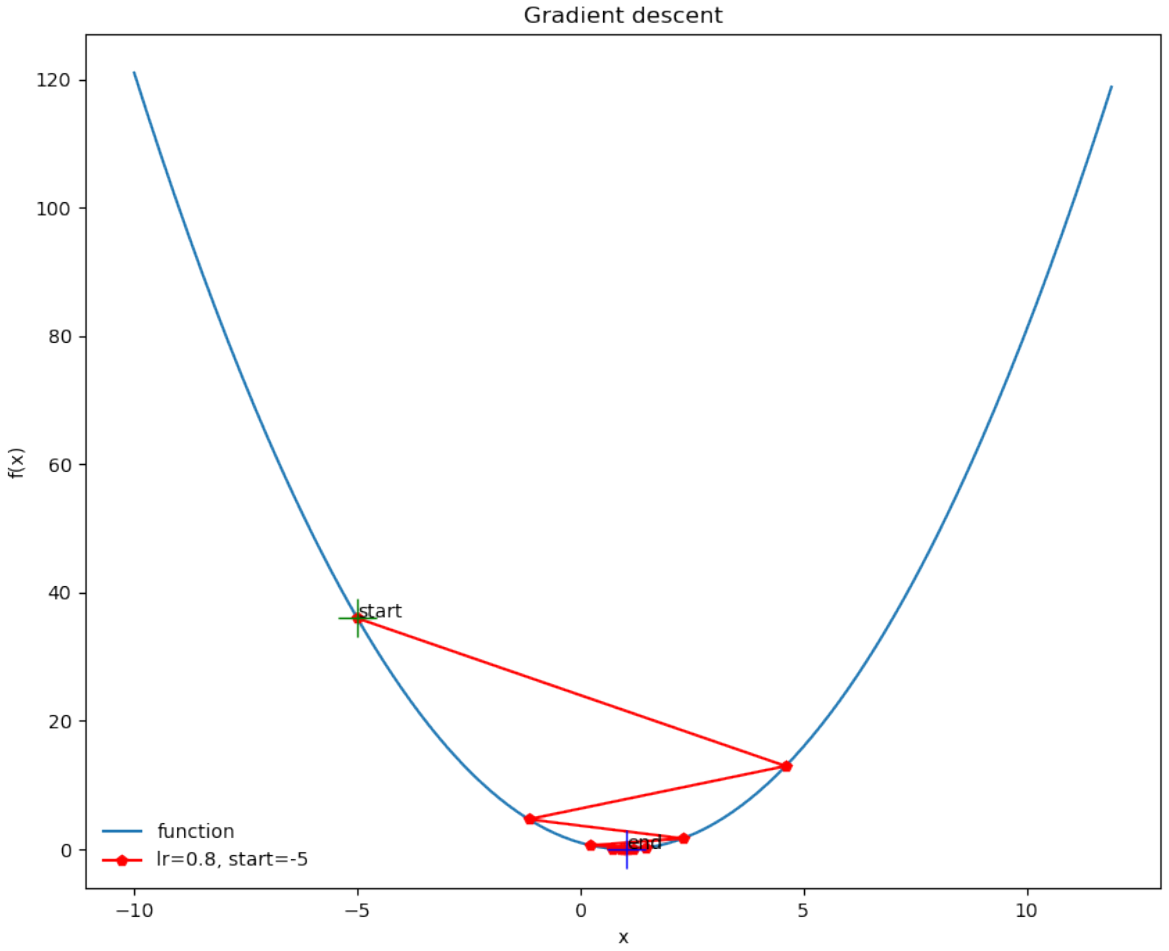
Red Hat

Intuition



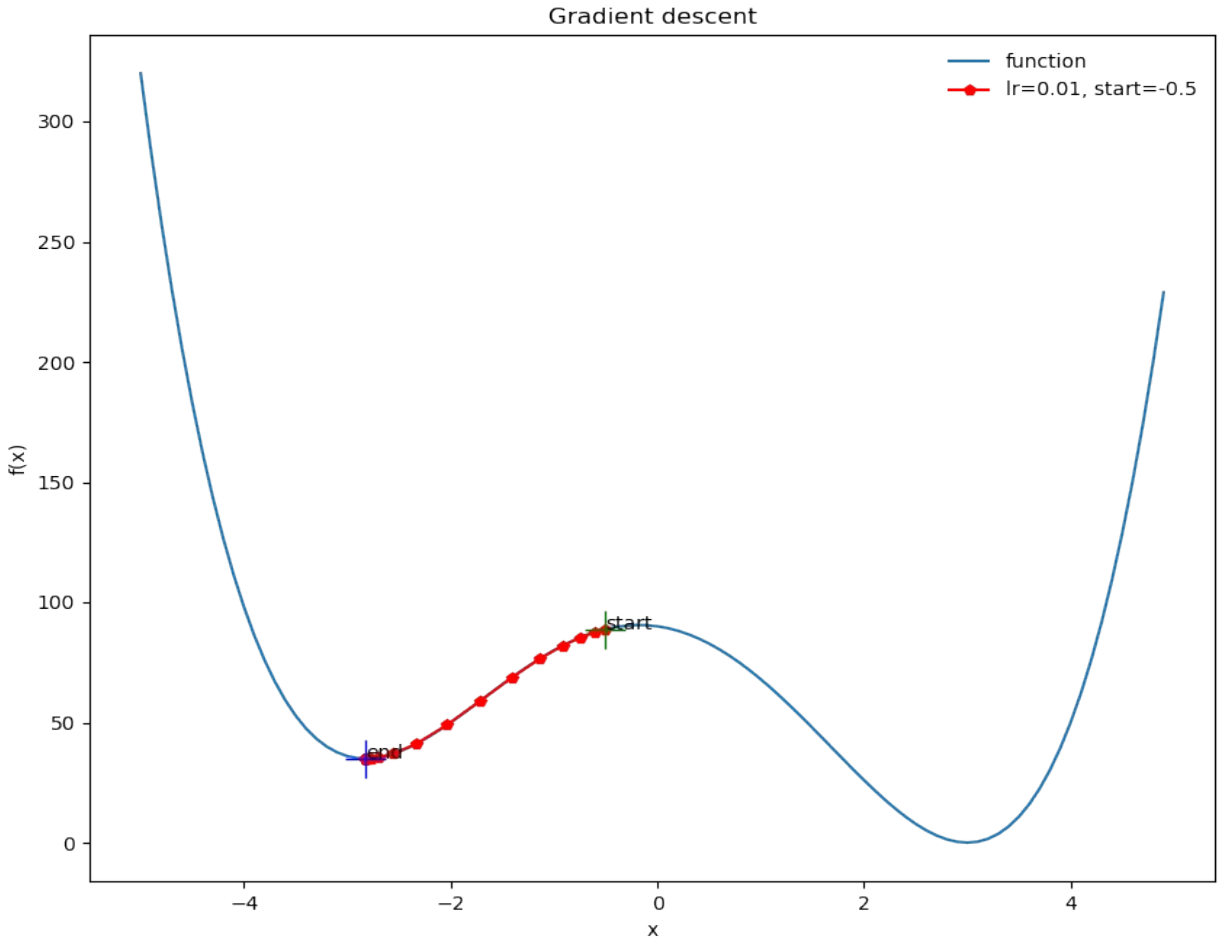
Red Hat

Intuition



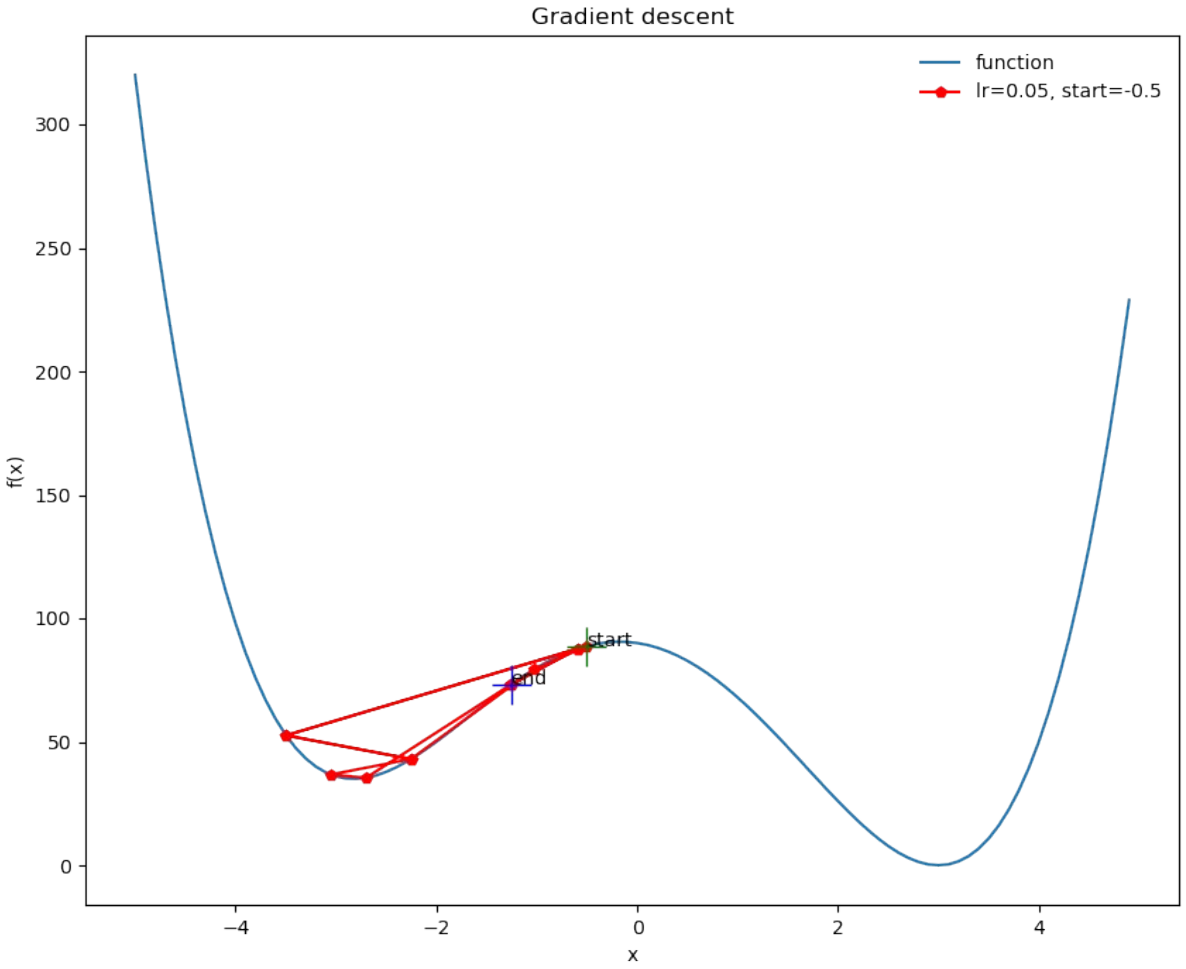
Red Hat

Intuition



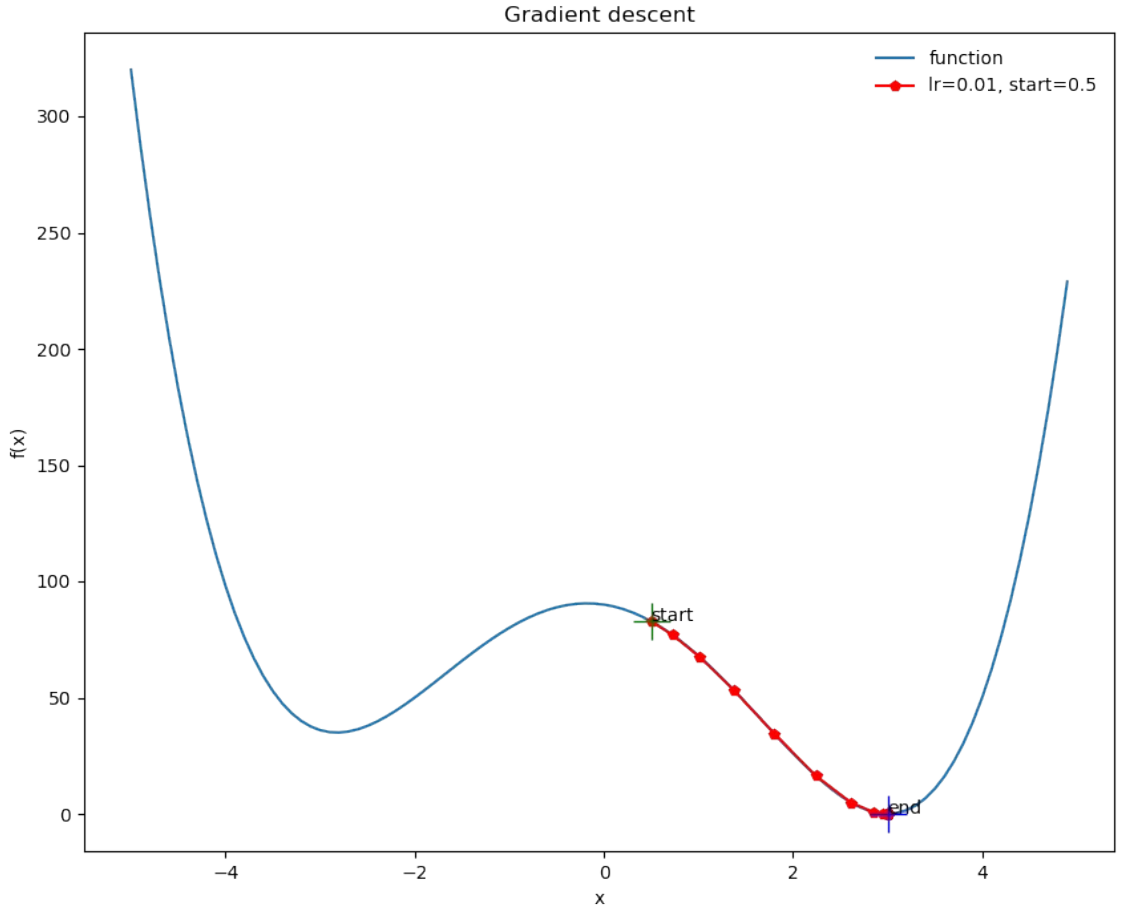
Red Hat

Intuition



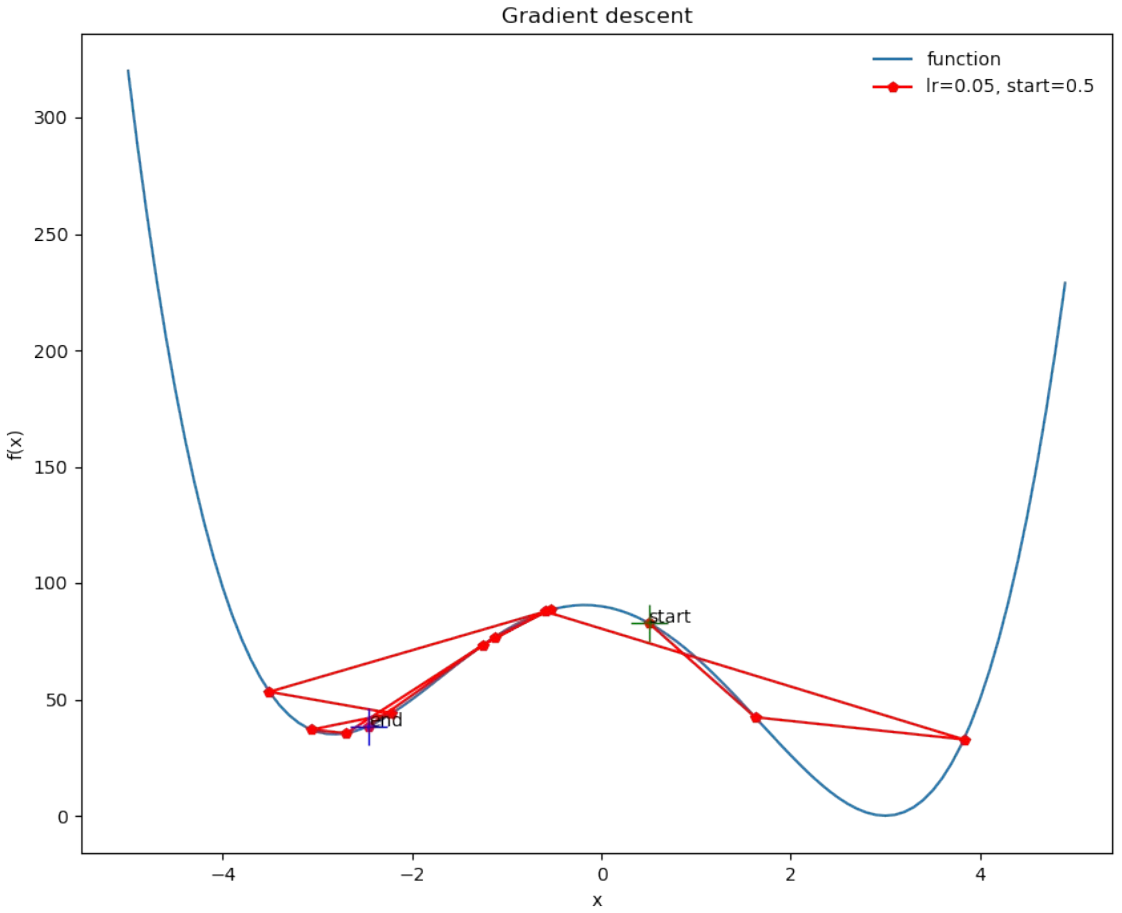
Red Hat

Intuition



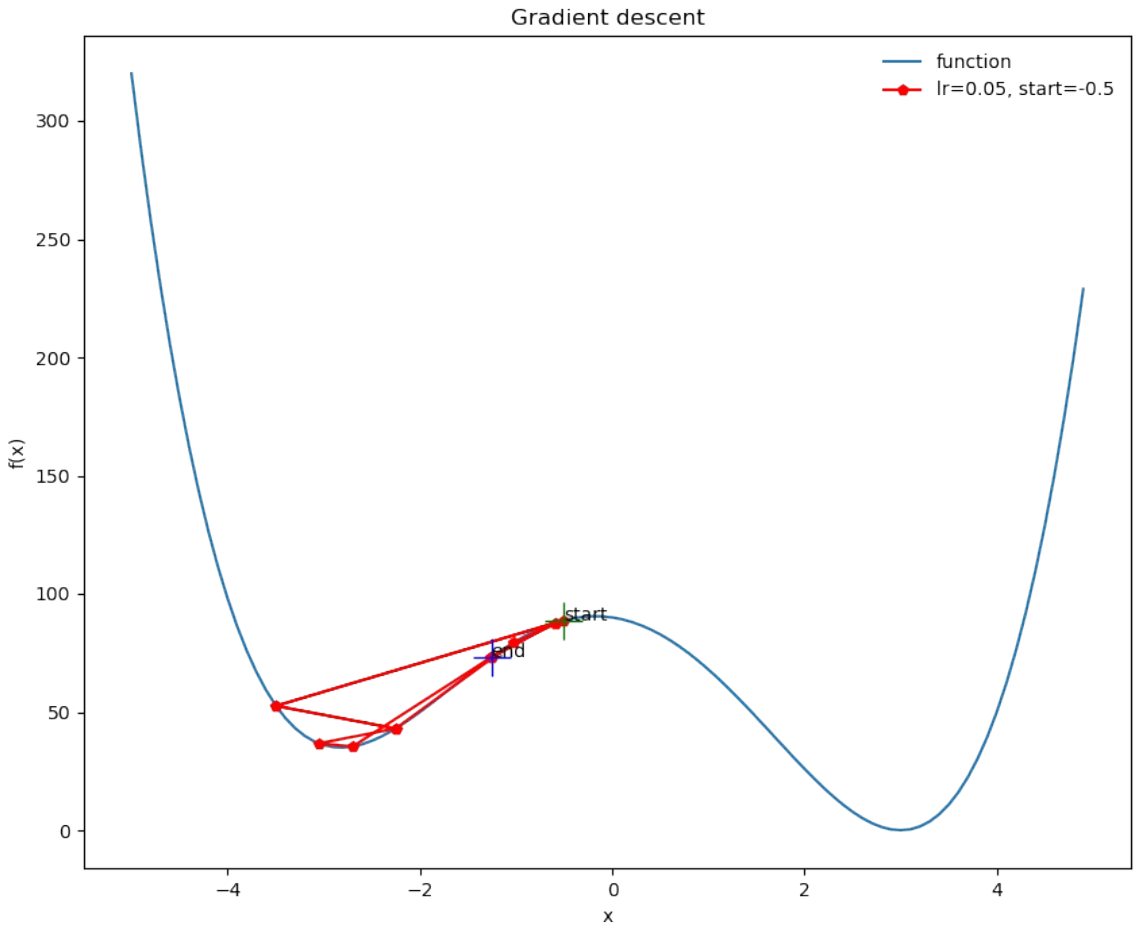
Red Hat

Intuition



Red Hat

Intuition



Red Hat

Intuition

- Convergence very sensitive to learning rate
- Learning rate too small:
  - Will converge to **local** minimum
  - Might take a long time
- Learning rate too large:
  - Will bounce around or even escape valley one started in
- If function decreases gradually, will take a long time to converge.
- Obvious question:
  - Can learning rate be **adaptive** i.e. change in response to location?

Convex Convergence

Every smooth function can be locally approximated as a quadratic (convex) function

$$\mathrm{f(x)} = \underbrace{f(x_{min}) + f'(x_{min})(x - x_{min}) + \frac{1}{2}f''(x_{min})(x - x_{min})^2 + \mathcal{O}((x - x_{min})^3)}_{a+b(x-c)^2}$$

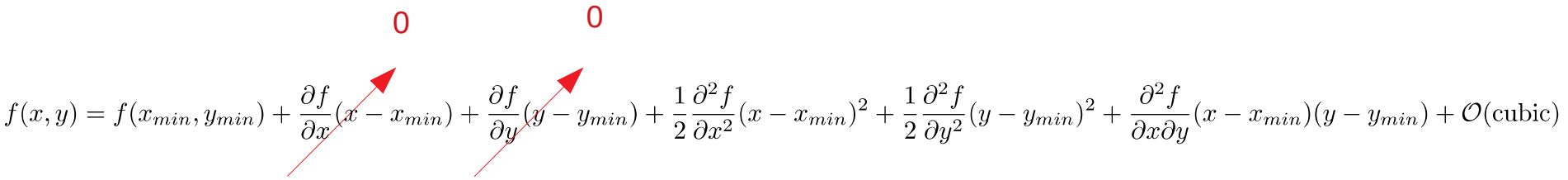
0

Taylor expansion around local minimum

Function of one variable

Convex Convergence

Every smooth function can be locally approximated as a quadratic (convex) function



$$f(x,y) = a + b(x - x_{min})^2 + d(y - y_{min})^2 + g(x - x_{min})(y - y_{min})$$

Taylor expansion around local minimum

All derivatives evaluated at (x_{min}, y_{min})

Convex Convergence

$$f(x,y) = a + b(x - x_{min})^2 + d(y - y_{min})^2 + g(x - x_{min})(y - y_{min})$$

Change of coordinates: $x' = x - x_{min}, y' = y - y_{min}$

$$f(x,y) = a + \begin{bmatrix} x' & y' \end{bmatrix} \begin{bmatrix} b & g\epsilon \\ g(1-\epsilon) & d \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$\epsilon = \frac{1}{2} \rightarrow \text{diagonalize symmetric matrix} \rightarrow f(x'', y'') = a + \alpha x''^2 + \beta y''^2$$

Convex Convergence

$$f(x) = a + \frac{b}{2}(x - c)^2$$

Start point: $x^{(0)}$

Update: $x^{(t)} = x^{(t-1)} - \eta \frac{df}{dx}(x^{(t-1)})$

$$\boxed{x^{(t)} = x^{(t-1)} - \eta b(x^{(t-1)} - c)}$$

Convex Convergence

$$x^{(t)} = x^{(t-1)} - \eta b(x^{(t-1)} - c)$$

$$\text{Minimum at: } x = c$$

$$\text{Consider Error: } \epsilon_t \equiv |x^{(t)} - c|$$

$$\underbrace{|x^{(t)} - c|}_{\epsilon_t} = |x^{(t-1)} - \eta b(x^{(t-1)} - c) - c| = \underbrace{|(x^{(t-1)} - c)|}_{\epsilon_{t-1}} |1 - \eta b|$$

$$\boxed{\epsilon_t = \epsilon_{t-1}|1 - \eta b|}$$

Convex Convergence

$$\epsilon_t = \epsilon_{t-1}|1 - \eta b|$$

$$\boxed{\epsilon_t = \epsilon_0|1 - \eta b|^t}$$

Distance from minimum at time t (ideally close to 0)

Distance from minimum at time 0

Convex Convergence

$$\boxed{\epsilon_t = \epsilon_0 |1 - \eta b|^t} \qquad 0 < 1 - \eta b < 1$$

$$\text{Suppose we want } \frac{\epsilon_t}{\epsilon_0} = \delta, \text{ a small number}$$

$$t = \frac{\log \delta}{\log |1 - \eta b|} \approx \frac{-\log \delta}{\eta b} \qquad \eta b \text{ small}$$

Backpropagation: Derivatives for Gradient Descent

Backpropagation: Derivatives

Gradient descent and its variants need the **first derivatives**:

$$\text{Minimize: } f(x_1, x_2, \ldots, x_n)$$

$$\text{Initial guess: } (x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)})$$

$$\text{Update: } x_i^{(t+1)} = x_i^{(t)} - \eta \boxed{\frac{\partial f}{\partial x_i}(x_1^{(t)}, x_2^{(t)}, \ldots, x_n^{(t)})}$$

Backpropagation: Derivatives

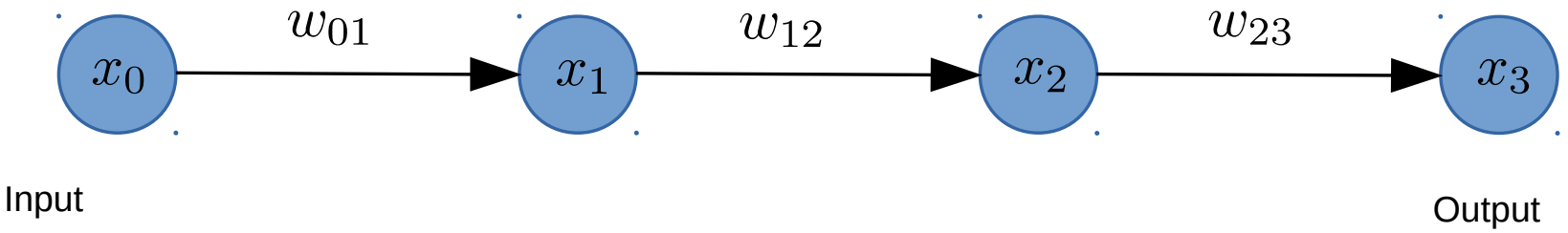
- Neural network has cost or loss function that depends on weights:

$$C[\vec{w}] = C[w_1, w_2, \ldots, w_n]$$

- Need to compute:

$$\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_n}$$

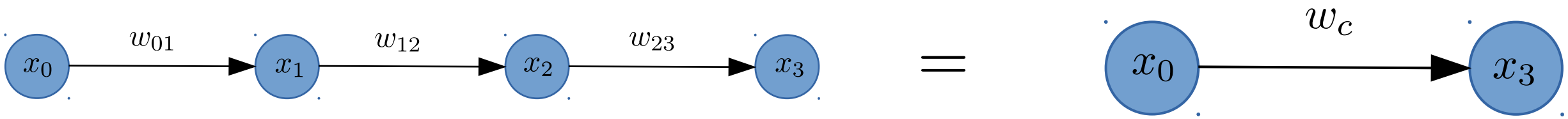
Backpropagation: Toy Example



Forward propagation:

$$x_1 = w_{01}x_0$$
$$x_2 = w_{12}x_1 \implies x_3 = w_{23}w_{12}w_{01}x_0$$
$$x_3 = w_{23}x_2$$

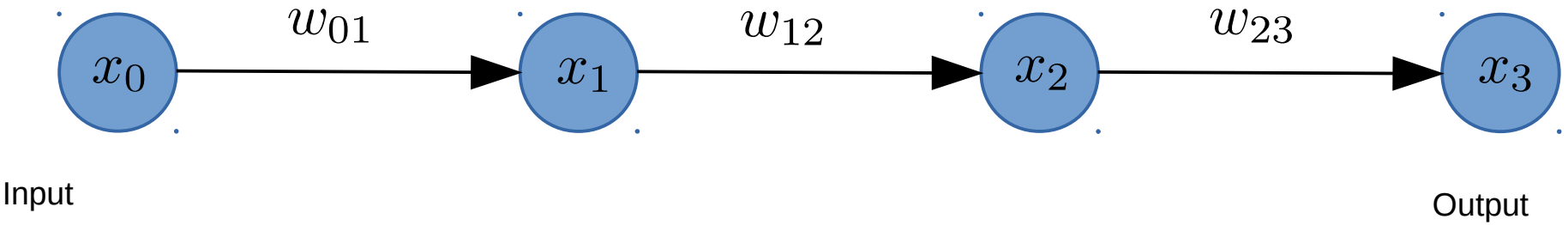
Backpropagation: Toy Example



$$x_3 = \underbrace{w_{23}w_{12}w_{01}}_{w_c} x_0$$

Ignore for now: we'll generalize this in a moment

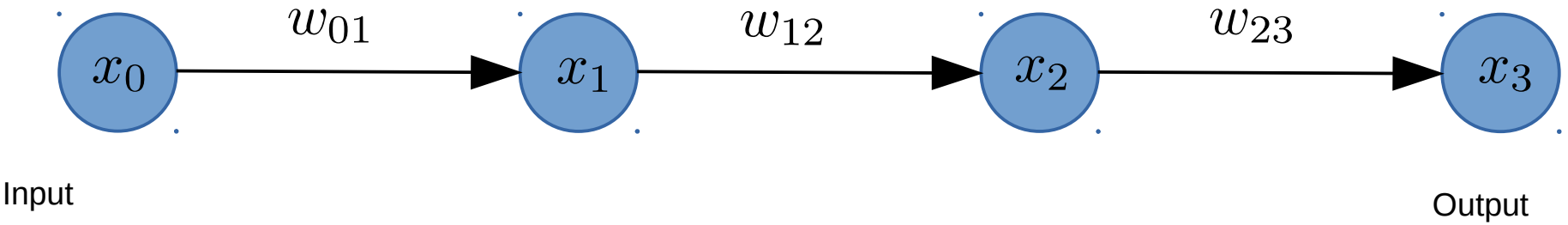
Backpropagation: Toy Example



$$\text{Cost} = \frac{1}{2}(x_3 - y)^2$$

$$x_3 = \text{prediction}, y = \text{label/actual value/target}$$

Backpropagation: Toy Example



$$\text{Cost} = \frac{1}{2}(x_3 - y)^2$$

$$\text{Question: what are } \frac{\partial C}{\partial w_{01}}, \frac{\partial C}{\partial w_{12}}, \frac{\partial C}{\partial w_{23}}?$$

Backpropagation: Toy Example

$$C = \frac{1}{2}(\underbrace{w_{23}w_{12}w_{01}x_0}_{x_3} - y)^2$$

$$\frac{\partial C}{\partial w_{01}} = (x_3 - y)w_{23}w_{12}x_0 = (x_3 - y)w_{23}w_{12}\mathbf{w_{01}}x_0$$

$$\frac{\partial C}{\partial w_{12}} = (x_3 - y)w_{23}w_{01}x_0 = (x_3 - y)w_{23}\mathbf{w_{12}}w_{01}x_0$$

$$\frac{\partial C}{\partial w_{23}} = (x_3 - y)w_{12}w_{01}x_0 = (x_3 - y)\mathbf{w_{23}}w_{12}w_{01}x_0$$

Backpropagation: Toy Example

Define forward chains:

$$x_0$$
$$w_{01}x_0 = x_1$$
$$w_{12}w_{01}x_0 = x_2 = w_{12}x_1$$
$$w_{23}w_{12}w_{01}x_0 = x_3 = w_{23}x_2$$

Define backward chains:

$$(x_3 - y) = \Delta_0$$
$$(x_3 - y)w_{23} = \Delta_1 = w_{23}\Delta_0$$
$$(x_3 - y)w_{23}w_{12} = \Delta_2 = w_{12}\Delta_1$$
$$(x_3 - y)w_{23}w_{12}w_{01} = \Delta_3 = w_{01}\Delta_2$$

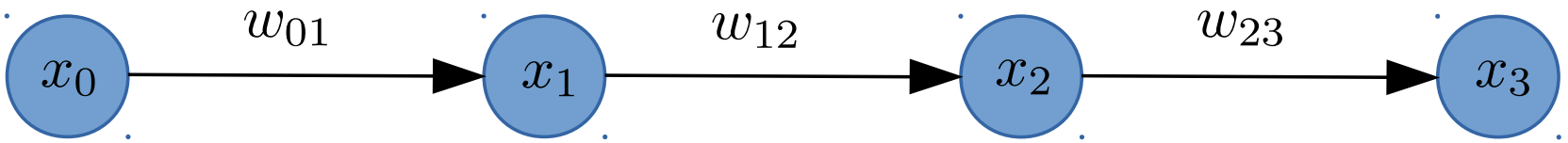
Backpropagation: Toy Example

$$\frac{\partial C}{\partial w_{01}} = \underbrace{(x_3 - y) w_{23} w_{12}}_{\Delta_2} w_{01} x_0 = \Delta_2 x_0$$

$$\frac{\partial C}{\partial w_{12}} = \underbrace{(x_3 - y) w_{23}}_{\Delta_1} w_{12} \underbrace{w_{01} x_0}_{x_1} = \Delta_1 x_1$$

$$\frac{\partial C}{\partial w_{12}} = \underbrace{(x_3 - y)}_{\Delta_0} w_{23} \underbrace{w_{12} w_{01} x_0}_{x_2} = \Delta_0 x_2$$

What's so great?

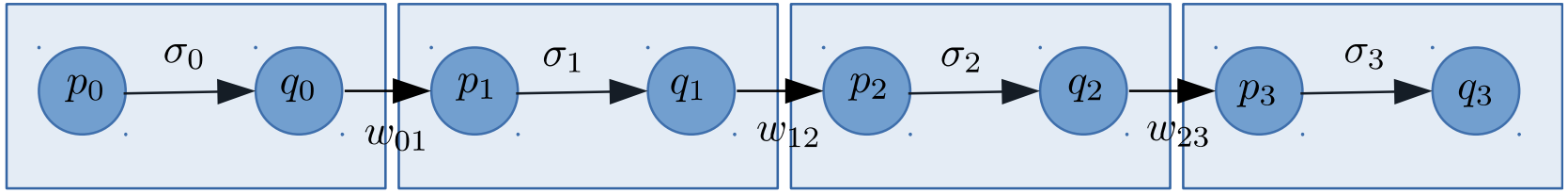


Forward propagation: calculate x_i

Backward propagation: calculate Δ_i

Calculations iterative: just multiply by weights

Backpropagation: Non-linear Toy Example



Forward propagation:

$$
\begin{aligned}
&\text{Layer 0:} && p_0 && q_0 = \sigma_0(p_0) \\
&\text{Layer 1:} && p_1 = w_{01}q_0 && q_1 = \sigma_1(p_1) \\
&\text{Layer 2:} && p_2 = w_{12}q_1 && q_2 = \sigma_2(p_2) \\
&\text{Layer 3:} && p_3 = w_{23}q_2 && q_3 = \sigma_3(p_3)
\end{aligned}
$$

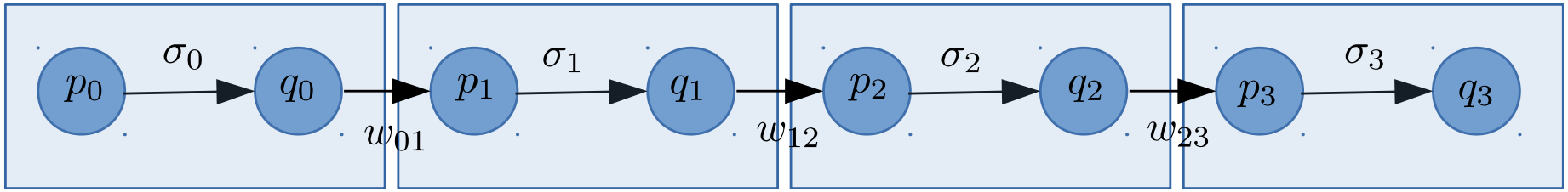
$$\boxed{q_3 = \sigma_3[w_{23}\sigma_2[w_{12}\sigma_1[w_{01}\sigma_0[p_0]]]]}$$

p_i = pre-activation $\rightarrow$ q_i = post-activation

q comes after p in the alphabet

multiplication by weight, $w_{i,i+1}$ takes $q_i \rightarrow p_{i+1}$

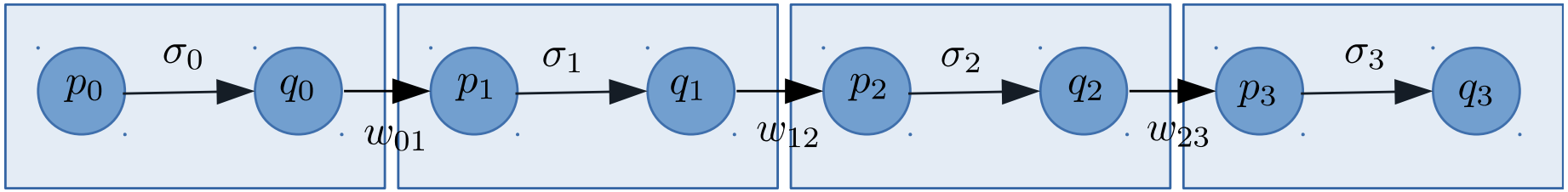
Backpropagation: Non-linear Toy Example



$$\text{Cost} = \frac{1}{2}(q_3 - y)^2$$

$$q_3 = \text{prediction}, y = \text{label/actual value/target}$$

Backpropagation: Non-linear Toy Example



Input

Output

$$\text{Cost} = \frac{1}{2}(q_3 - y)^2$$

$$\text{Question: what are } \frac{\partial C}{\partial w_{01}}, \frac{\partial C}{\partial w_{12}}, \frac{\partial C}{\partial w_{23}}?$$

Backpropagation: Non-linear Toy Example

$$C = \frac{1}{2}(\underbrace{\sigma_3[w_{23}\sigma_2[w_{12}\sigma_1[w_{01}\sigma_0[p_0]]]]}_{q_3} - y)^2$$

$$\frac{\partial C}{\partial w_{23}} = \boxed{(q_3 - y)\sigma_3'(p_3)}\boxed{q_2}$$

$$\frac{\partial C}{\partial w_{12}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)}\boxed{q_1}$$

$$\frac{\partial C}{\partial w_{01}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)w_{12}\sigma_1'(p_1)}\boxed{q_0}$$

Backpropagation: Non-linear Toy Example

Define forward chains:

$$
\begin{aligned}
& q_0 \\
q_1 &= \sigma_1(p_1) = \sigma_1(w_{01} q_0) \\
q_2 &= \sigma_2(p_2) = \sigma_2(w_{12} q_1) \\
q_3 &= \sigma_3(p_3) = \sigma_3(w_{23} q_2)
\end{aligned}
$$

Define backward chains:

$$
\begin{aligned}
(q_3 - y)\sigma_3'(p_3) &= \Delta_0 \\
(q_3 - y)\sigma_3'(p_3) w_{23} \sigma_2'(p_2) &= \Delta_1 = w_{23}\sigma_2'(p_2)\Delta_0 \\
(q_3 - y)\sigma_3'(p_3) w_{23} \sigma_2'(p_2) w_{12} \sigma_1'(p_1) &= \Delta_2 = w_{12}\sigma_1'(p_1)\Delta_1 \\
(q_3 - y)\sigma_3'(p_3) w_{23} \sigma_2'(p_2) w_{12} \sigma_1'(p_1) w_{01} \sigma_0'(p_0) &= \Delta_3 = \sigma_0'(p_0) w_{01} \Delta_2
\end{aligned}
$$

Backpropagation: Non-linear Toy Example

$$\frac{\partial C}{\partial w_{23}} = \boxed{(q_3 - y)\sigma_3'(p_3)}\,\boxed{q_2} = \Delta_0 q_2$$

$$\frac{\partial C}{\partial w_{12}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)}\,\boxed{q_1} = \Delta_1 q_1$$

$$\frac{\partial C}{\partial w_{01}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)w_{12}\sigma_1'(p_1)}\,\boxed{q_0} = \Delta_2 q_0$$

q_i evaluated during forward pass

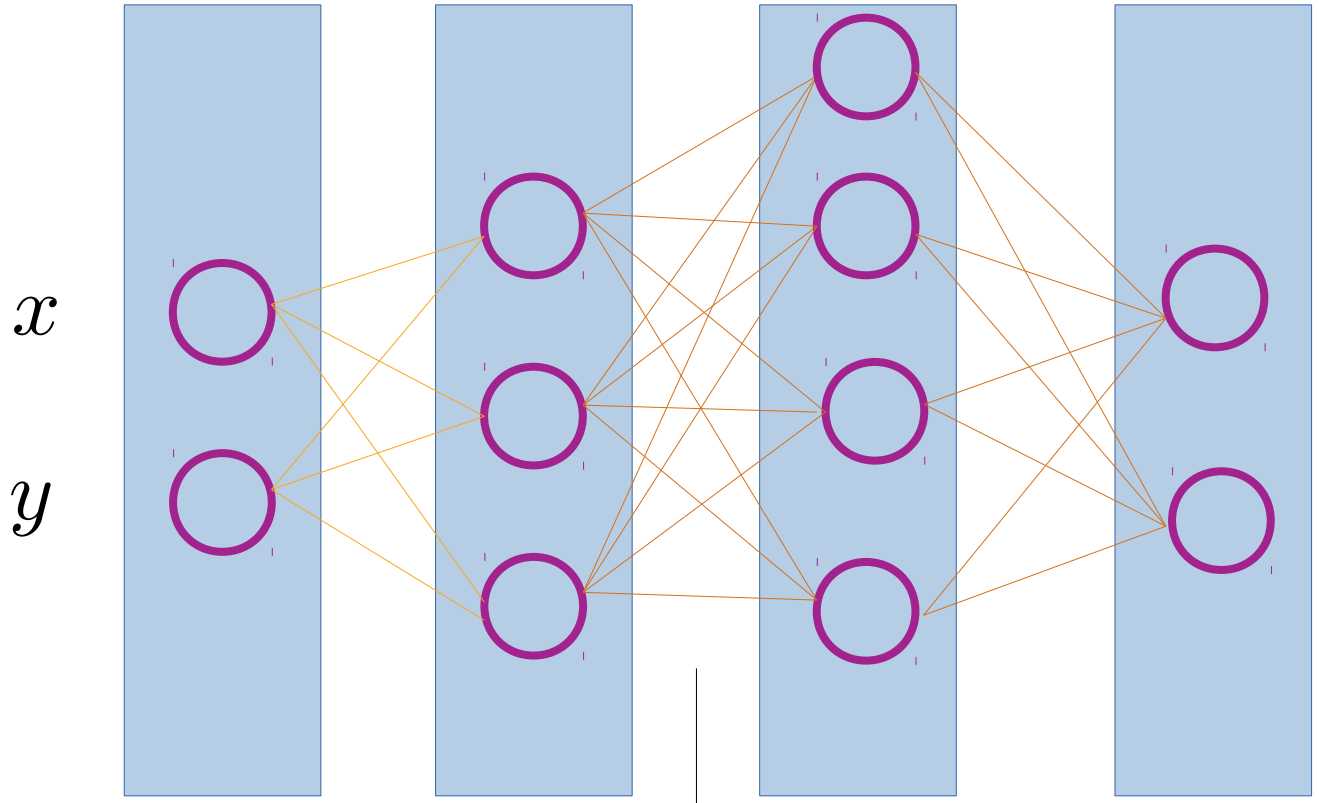
Δ_i iteratively evaluated once forward pass completed

A Whirlwind Tour of Neural Net Architectures

Time to Zoom Out

- It is very important to understand backpropagation and gradient descent (as well as other optimization techniques).
- But as far as applications are concerned, you can go very far by working at a higher level of abstraction.
- Let’s take a look at the kinds of layers/building blocks that are very useful in modern applications of deep learning.

Dense Layers



torch.nn.Linear in PyTorch

All nodes are pairwise connected from one block to another

Red Hat

Images?

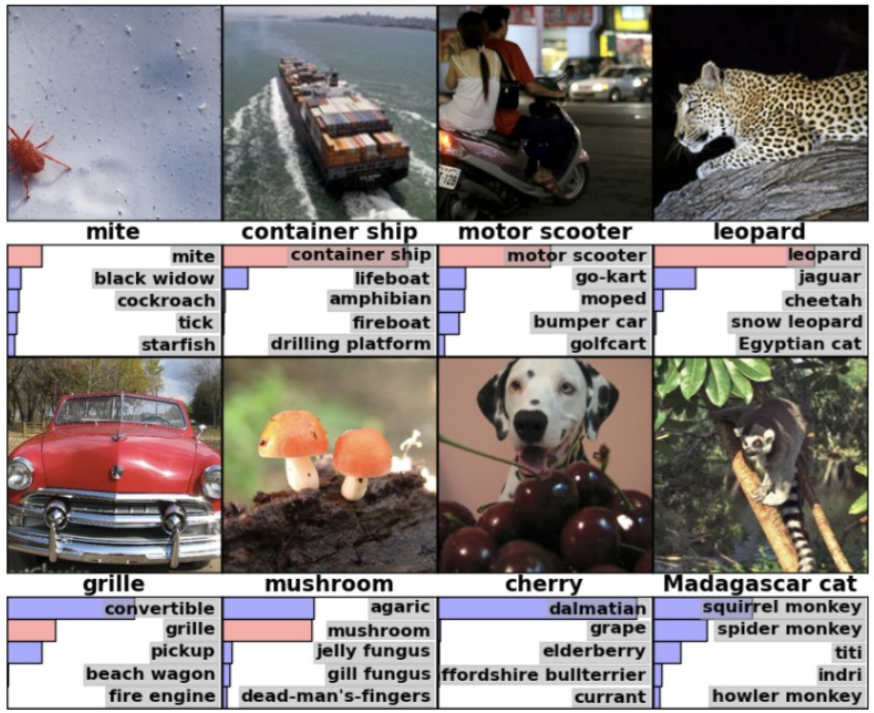


Image Classification

- One of the major applications of deep learning is to images.
- The inputs to the network are the **raw pixel values** (with minimal preprocessing like normalization).
- A dense layer uses **all** the input pixel values to compute the activation in the first hidden layer.
- But objects in real-life images are **local** i.e. a pixel's value is strongly correlated with its neighbors' values.
- Can we impose such locality in the layer?

Filters and Convolutions

An image is a matrix

$$\begin{bmatrix} 1 & 4 & 3 & 2 \\ 1 & 2 & 8 & 15 \\ 21 & 3 & 0 & 0 \\ 100 & 1 & 50 & 2 \\ 23 & 18 & 29 & 45 \end{bmatrix}$$

5x4

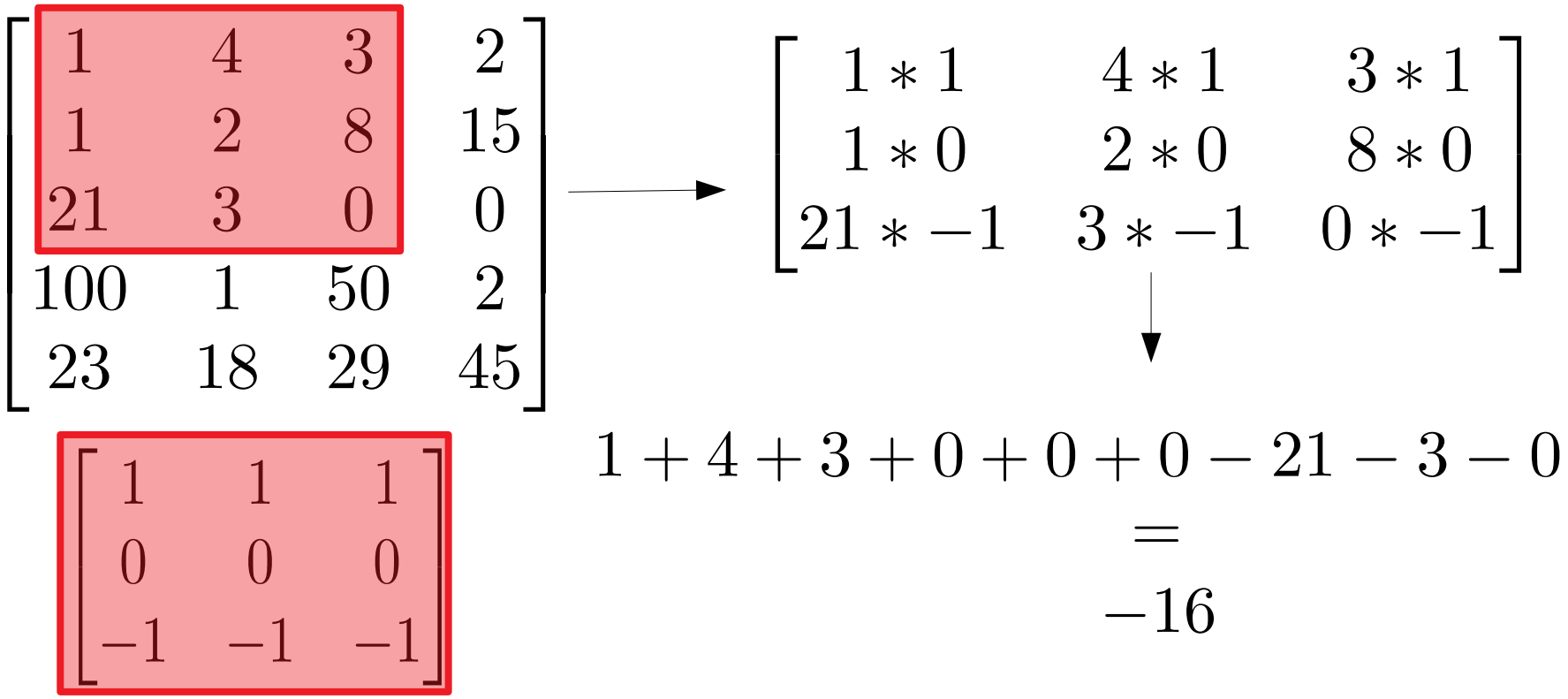
A filter is a matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

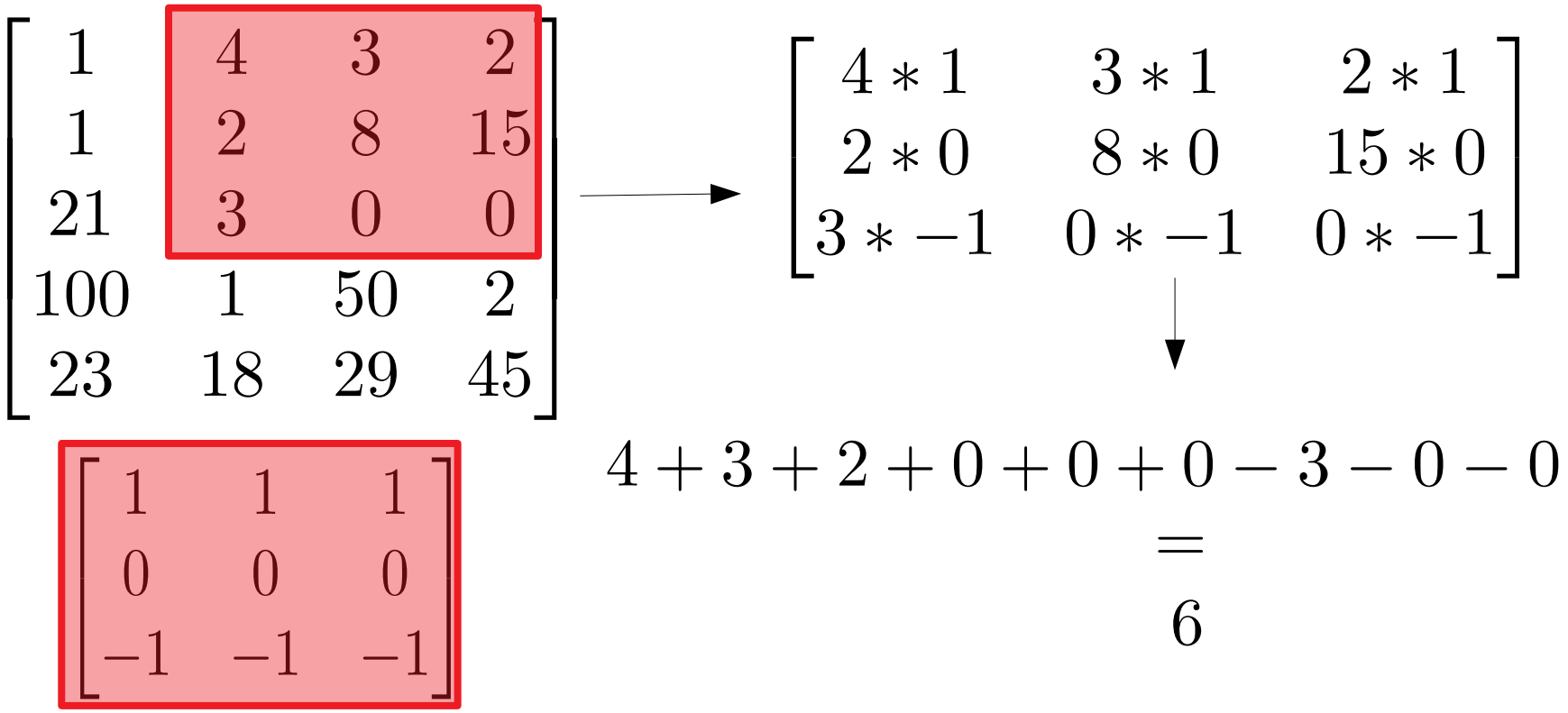
designed to extract specific Information from image

3x3

Filters and Convolutions



Filters and Convolutions



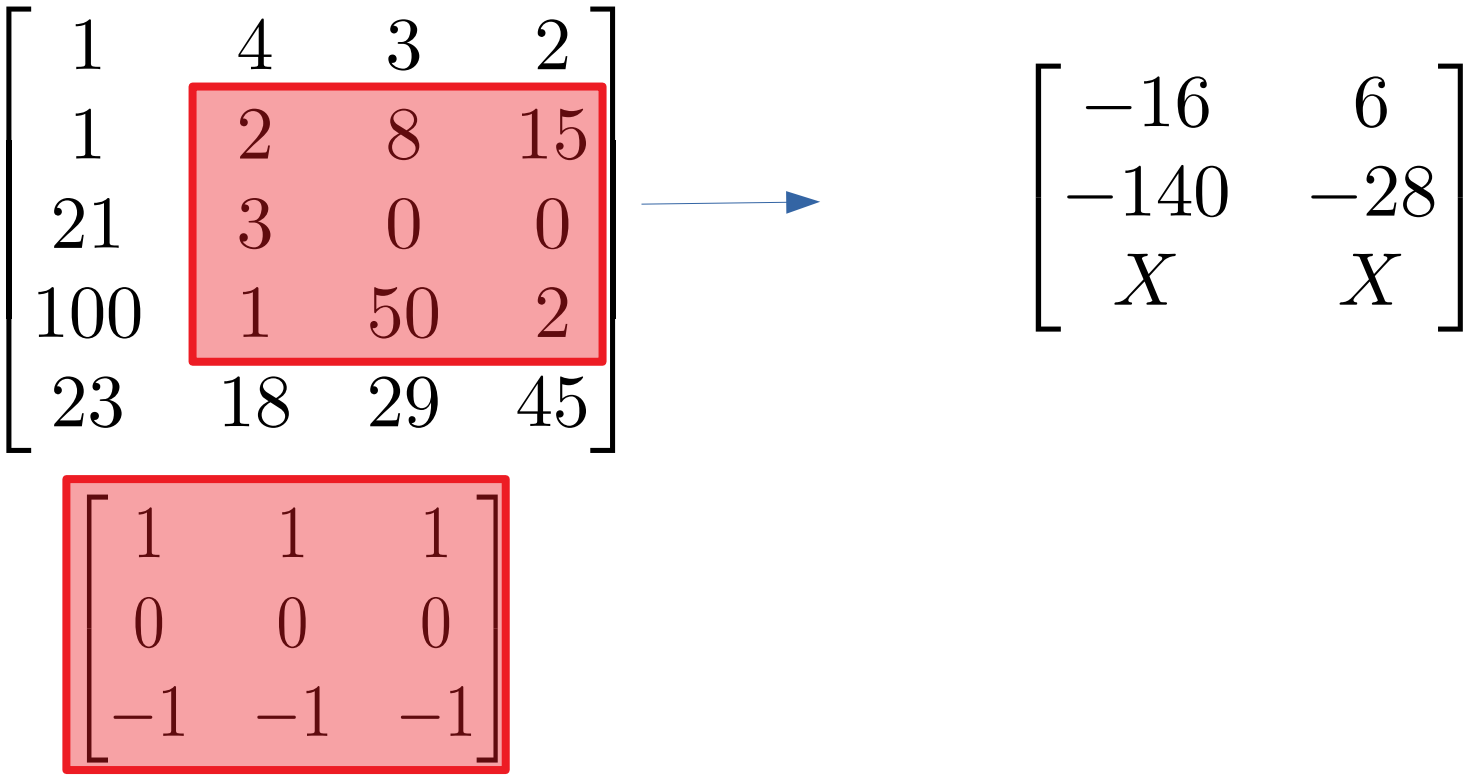
Filters and Convolutions

$$\begin{bmatrix} 1 & 4 & 3 & 2 \\ 1 & 2 & 8 & 15 \\ 21 & 3 & 0 & 0 \\ 100 & 1 & 50 & 2 \\ 23 & 18 & 29 & 45 \end{bmatrix}$$

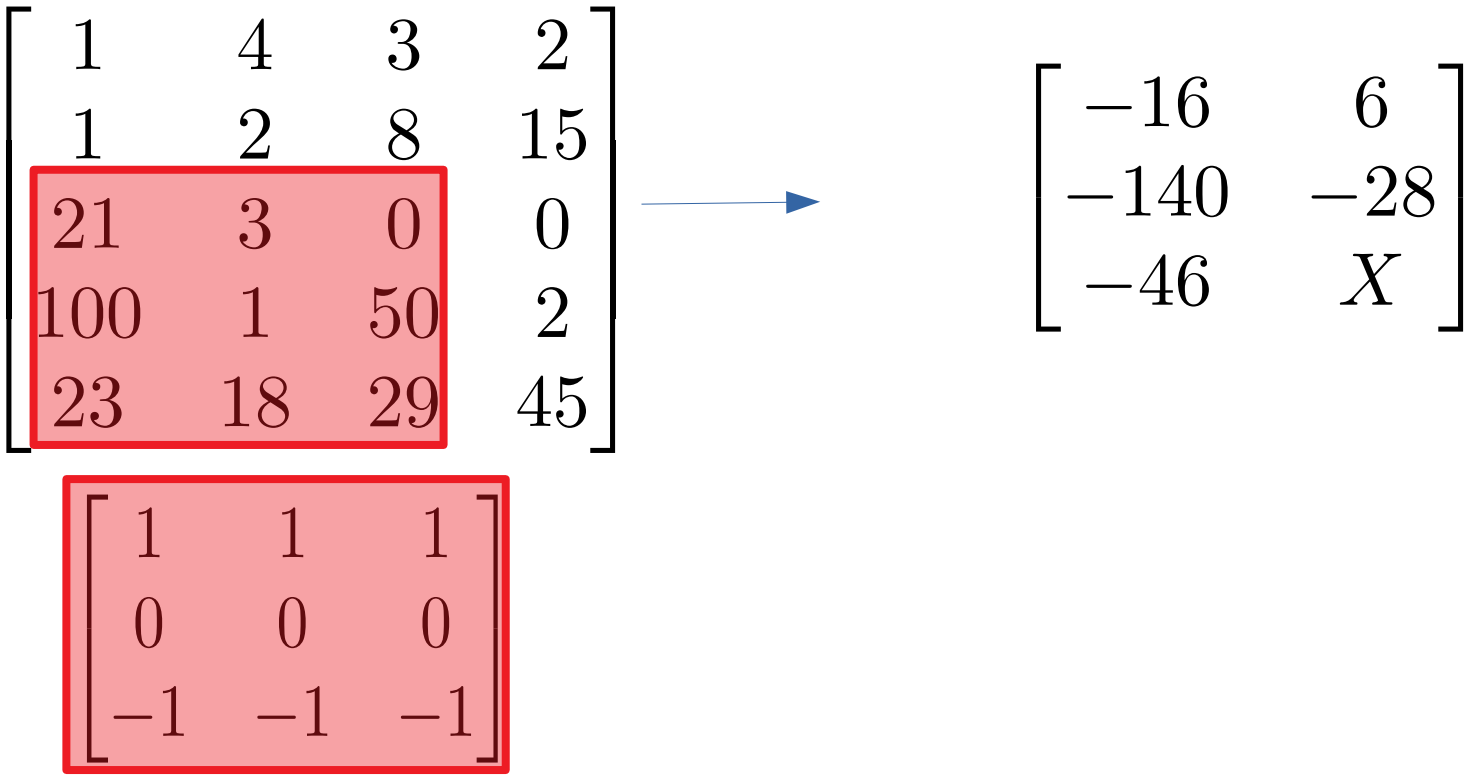
$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} -16 & 6 \\ -140 & X \\ X & X \end{bmatrix}$$

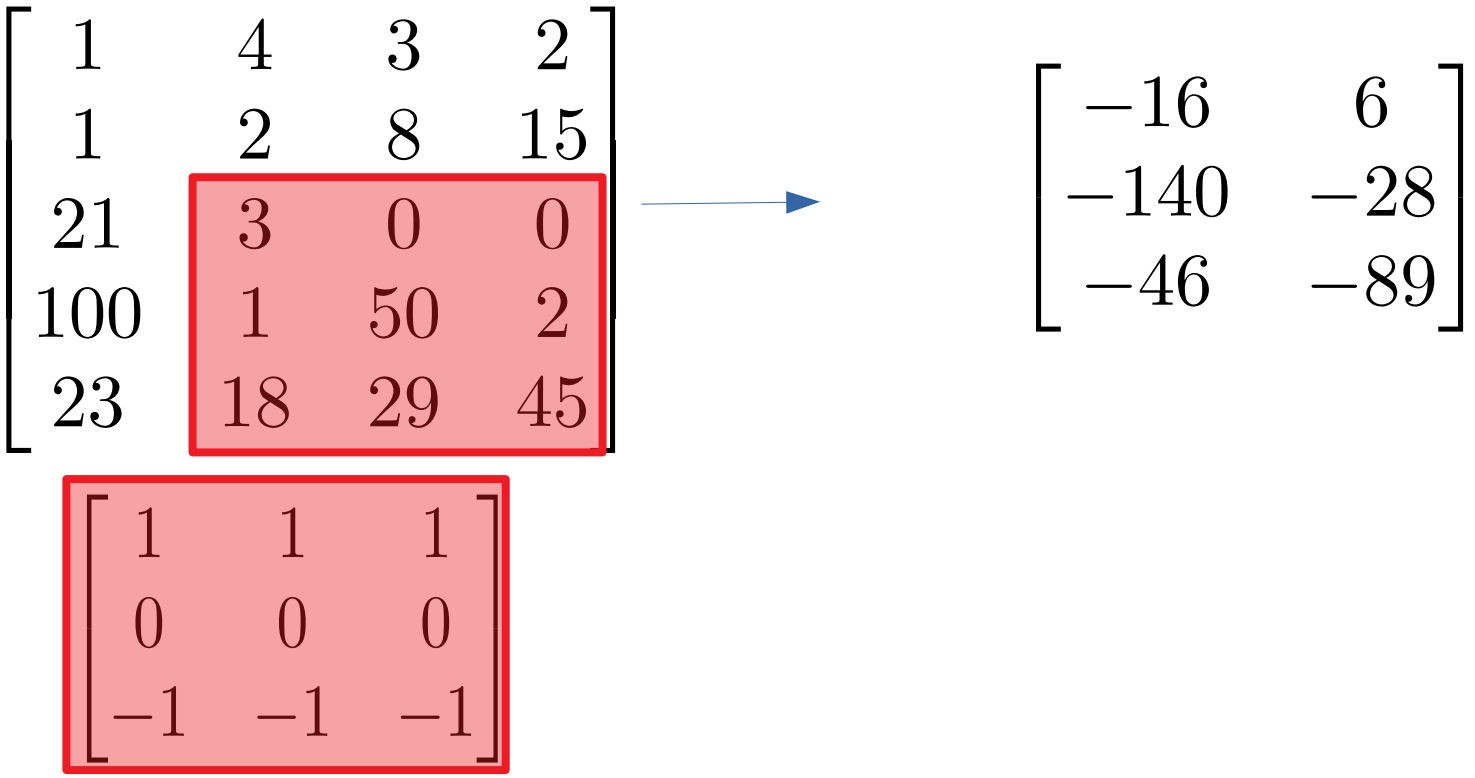
Filters and Convolutions



Filters and Convolutions



Filters and Convolutions



Red Hat

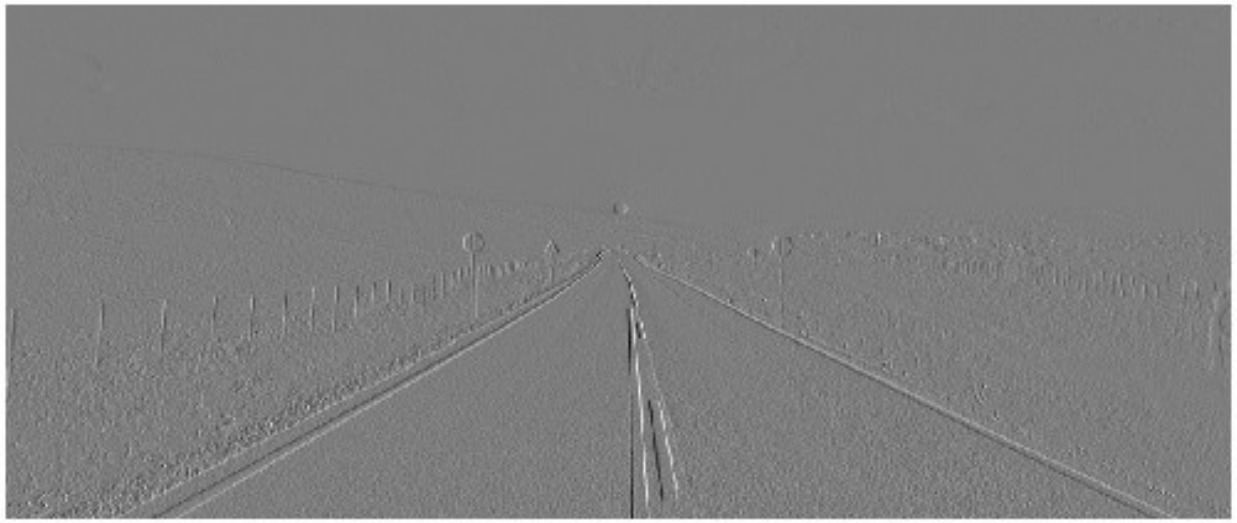
Edge Detection: Hand-crafted Filters

grayscale

Edge Detection: Hand-crafted Filters

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \rightarrow \text{picks up vertical intensity changes i.e. horizontal edges}$$

Edge Detection: Hand-crafted Filters



$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \rightarrow \text{picks up horizontal intensity changes i.e. vertical edges}$$

Edge Detection: Hand-crafted Filters

Use previous two “gradients” to deduce edges

Learning Kernels?

Well-designed kernels serve as very powerful feature extractors

They are local

Natural Question:
Can we “learn” new kernels instead of picking them by hand?

Yes! Convolutional Layers

Won’t go into more details here but see notebooks for a practical example

Default way for computer vision (as well as some natural language) applications

Sequential Data

$$x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_t$$

Data ordered in time

Language, stock prices, machine metrics, speech etc.

Sequential Data

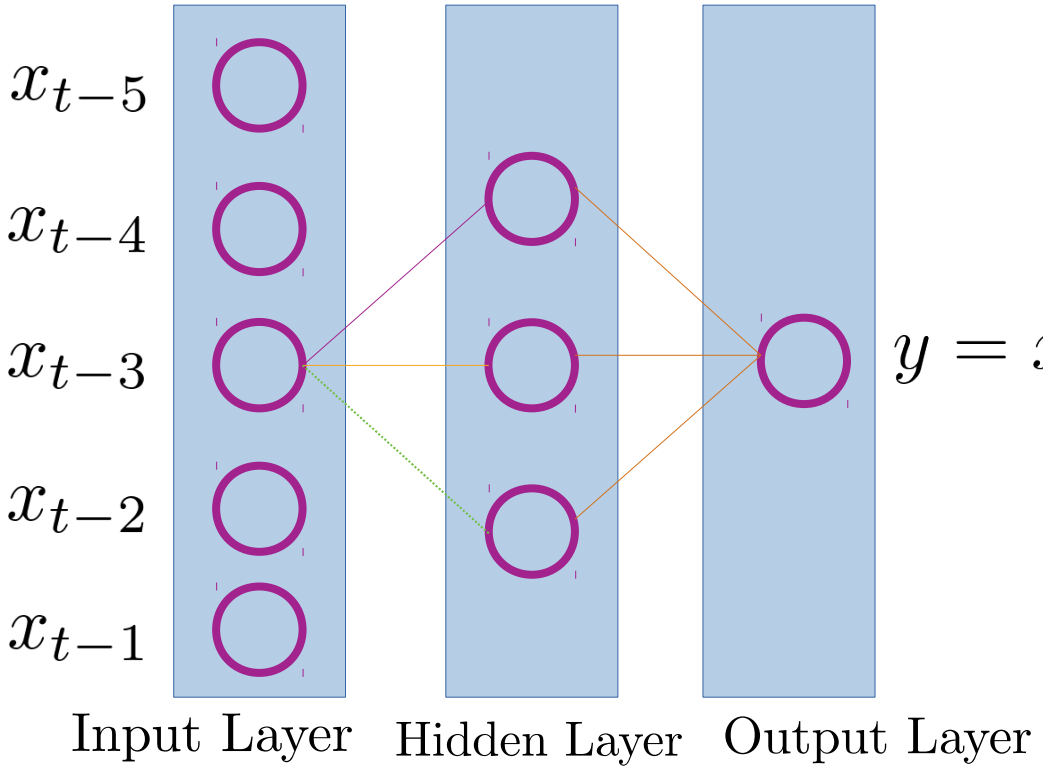
$$x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_t$$

Interesting problems:

Predict future value

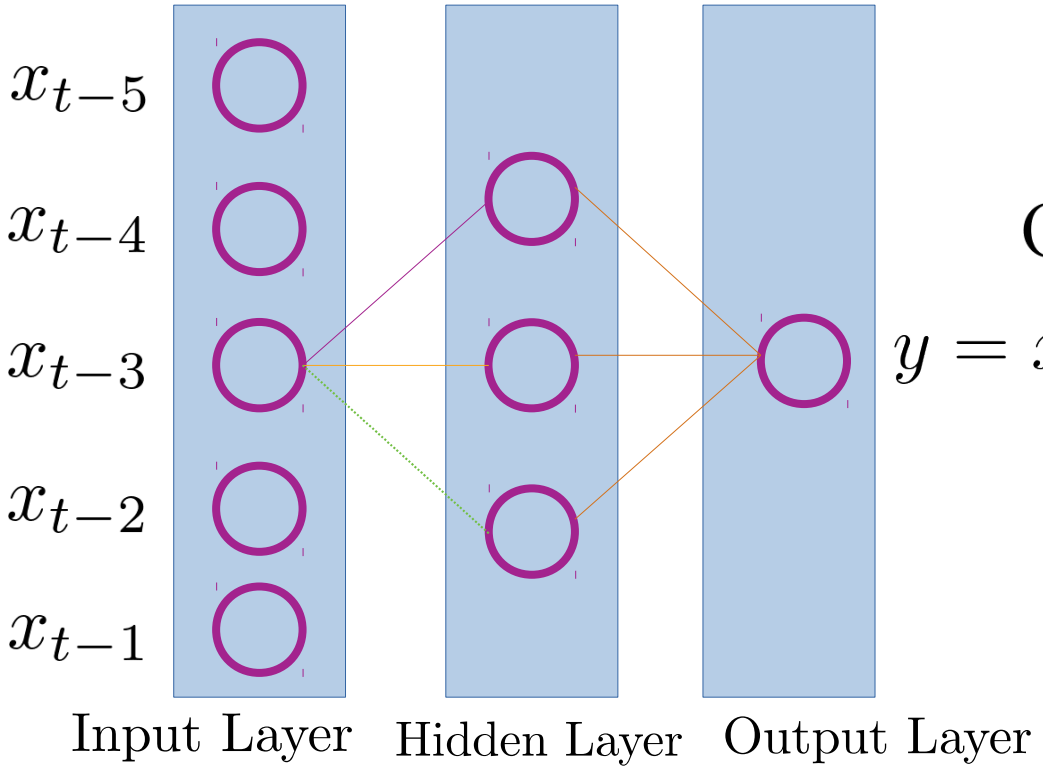
Predict another time-series: speech recognition = sound waves to text

Sequential Data



Maybe this will work?

Sequential Data



Problems:

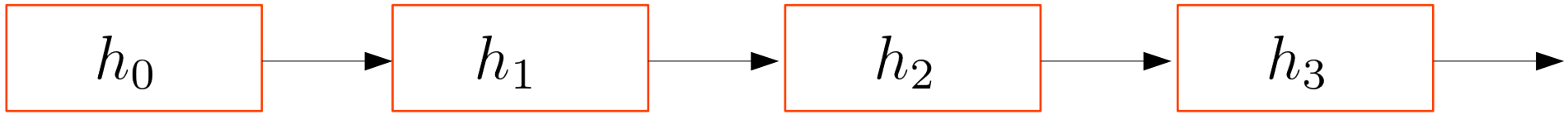
Only use limited and fixed history

No notion of state or memory

Recurrent Neural Networks

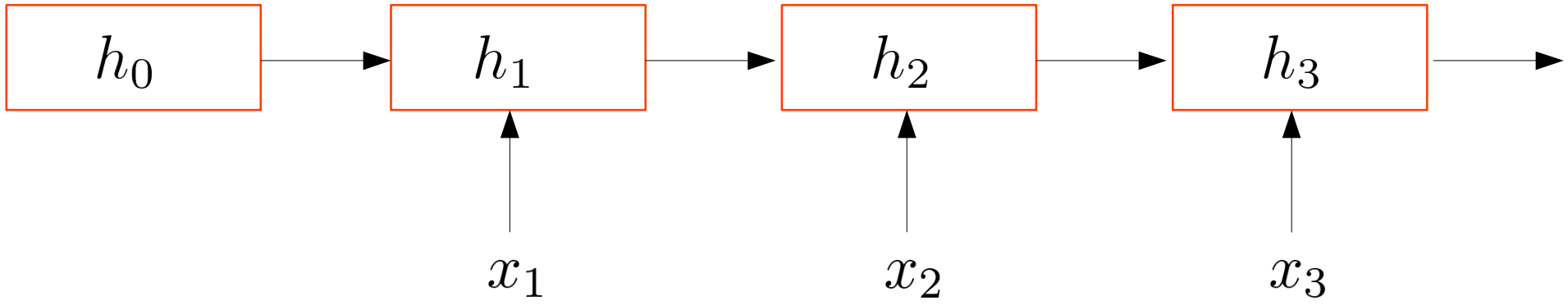
Introduce state at time t: h_t (a vector of numbers)

h_t evolves in time



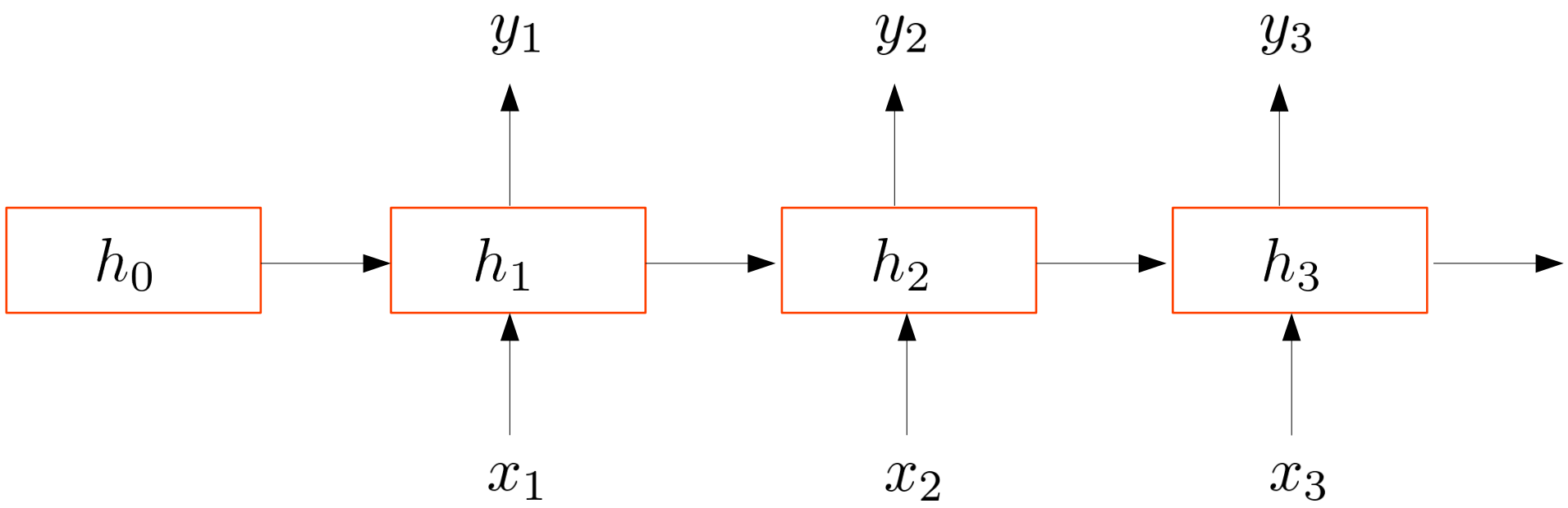
Recurrent Neural Networks

Of course, need inputs: x_t



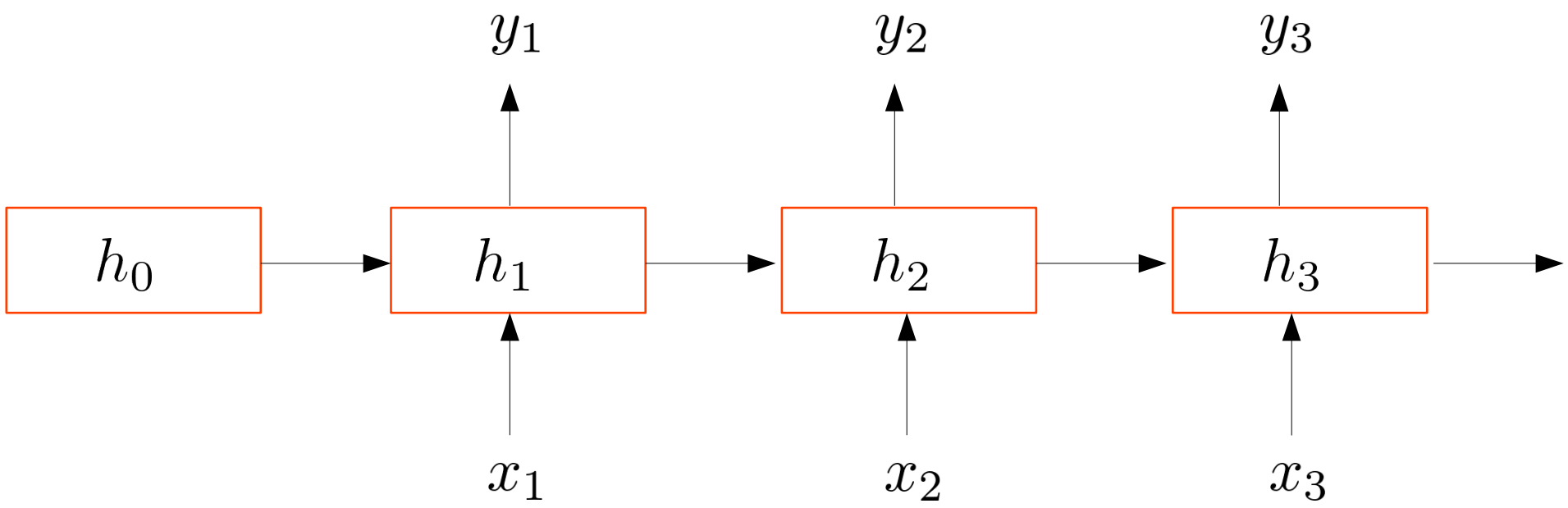
Recurrent Neural Networks

and outputs: y_t



Recurrent Neural Networks

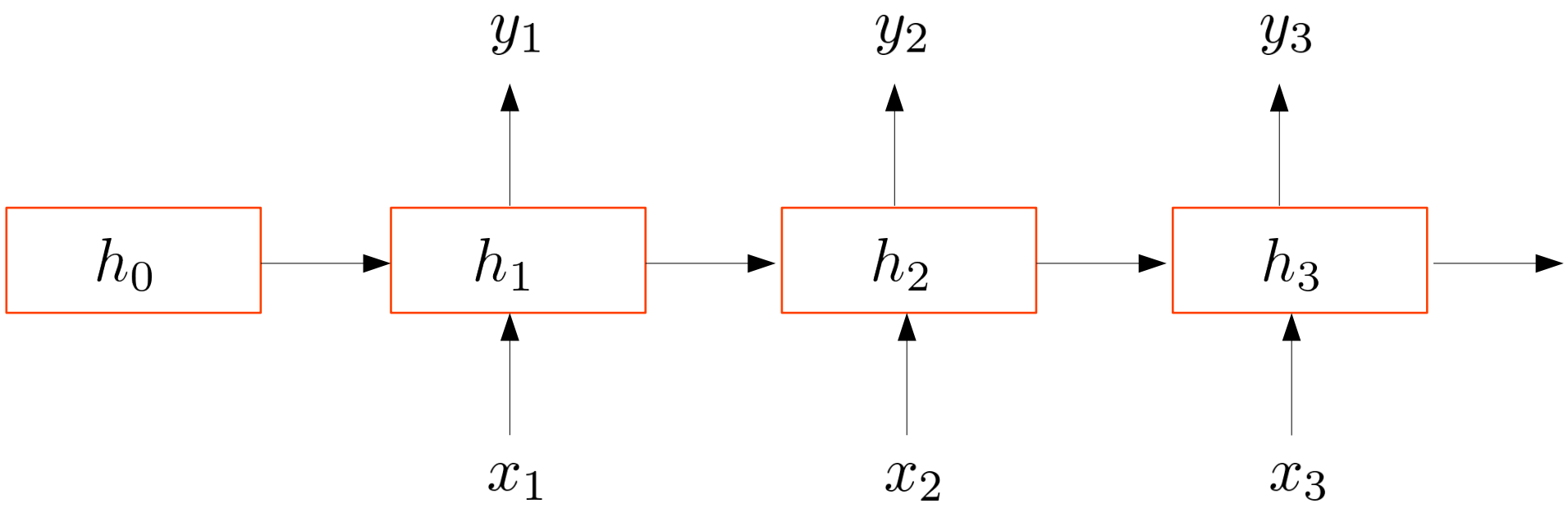
and outputs: y_t



E.g. $y_t = x_{t+k}$ for fixed k > 0

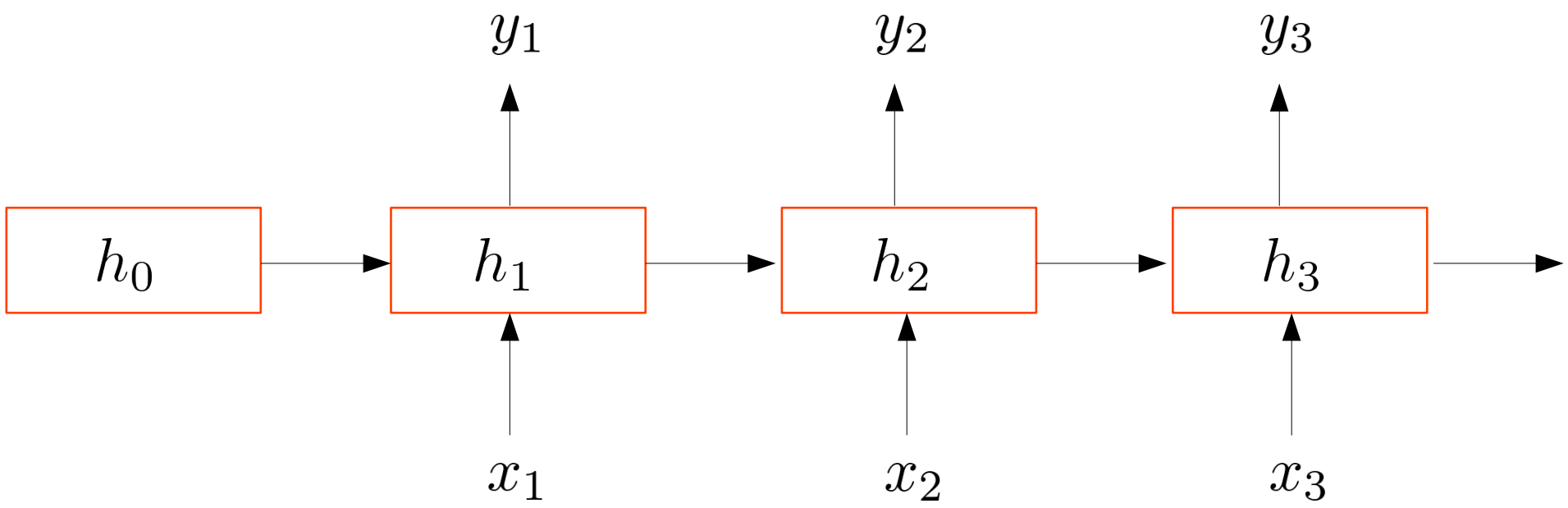
E.g. $y_t =$ sentiment at t

Recurrent Neural Networks



What is forward propagation here?

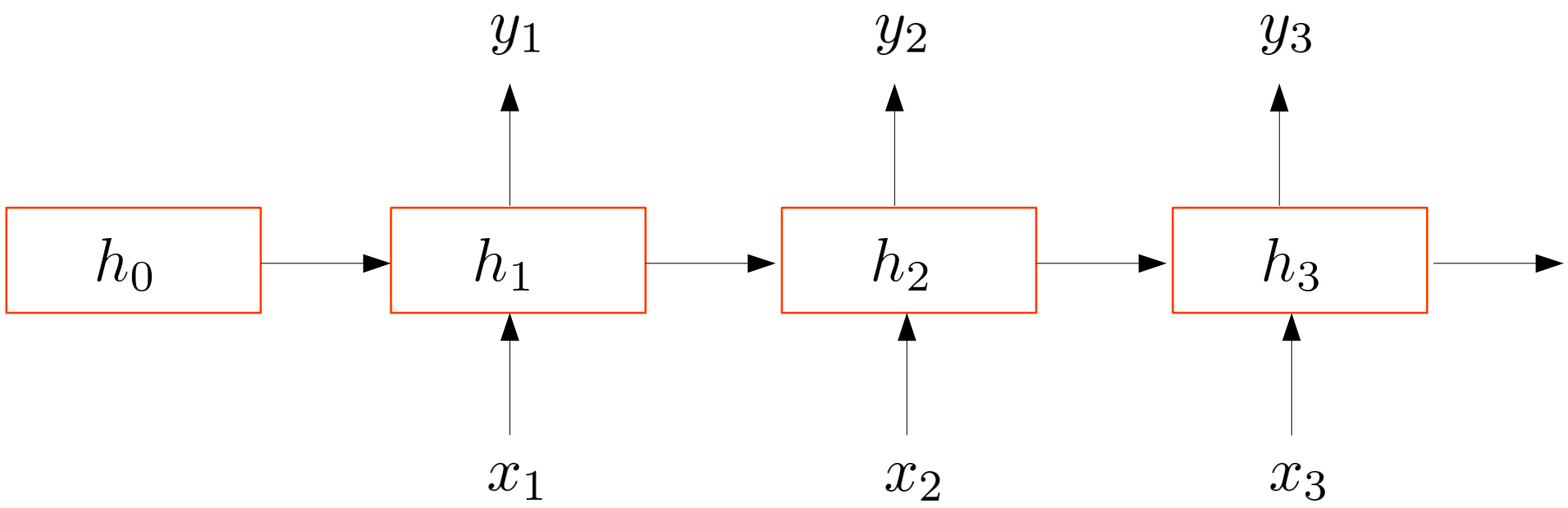
Recurrent Neural Networks



$$\text{Arrow} \rightarrow \sigma(\text{matrix} * \text{vector} + \text{vector})$$

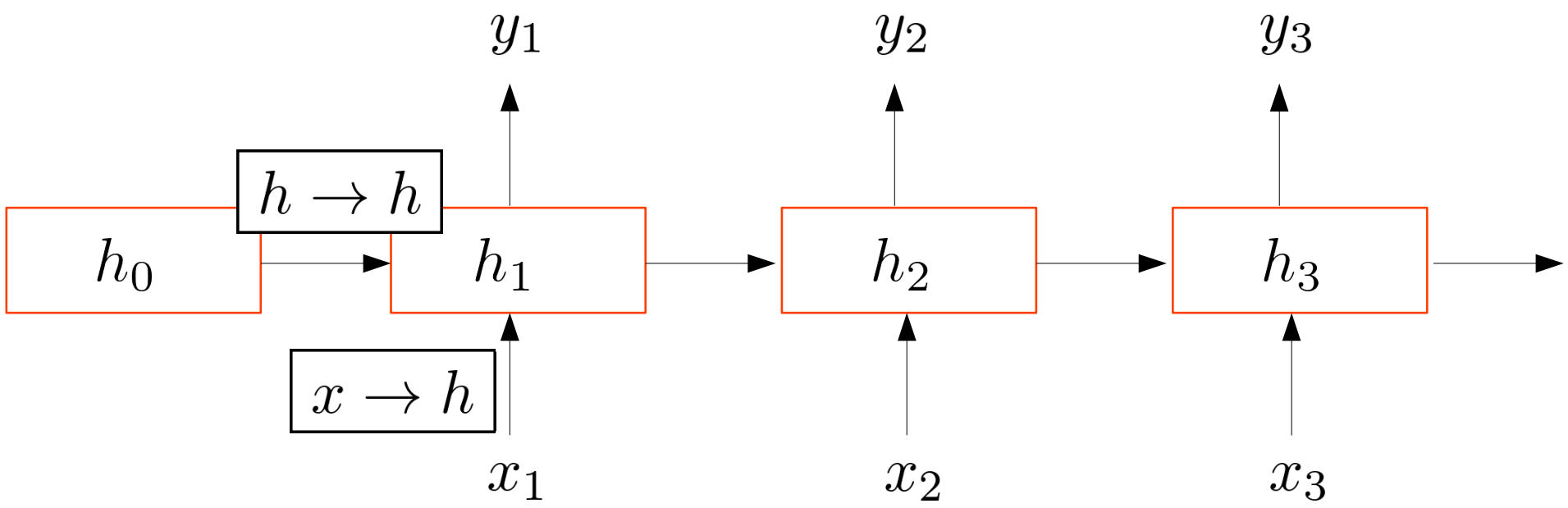
Exactly like a dense layer

Recurrent Neural Networks



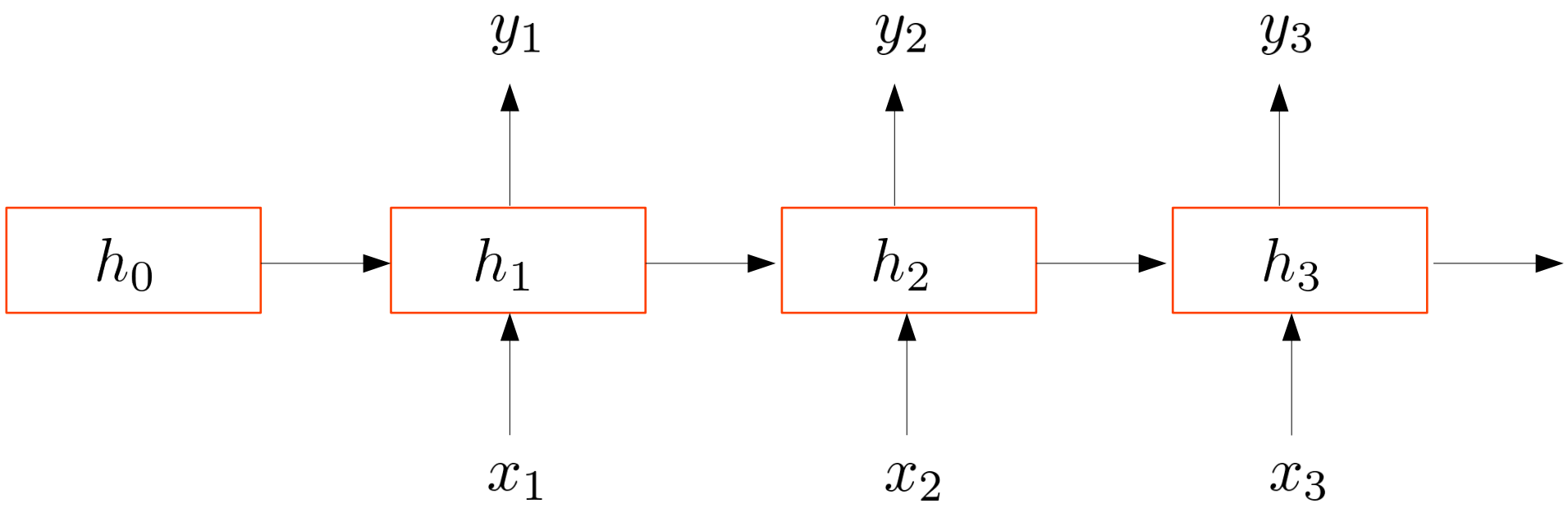
$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

Recurrent Neural Networks



$$\underbrace{h_t}_{\text{h-space}} = \sigma(\underbrace{W_{hx}}_{\text{x-space to h-space}} \underbrace{x_t}_{\text{x-space}} + \underbrace{W_{hh}}_{\text{h-space to h-space}} \underbrace{h_{t-1}}_{\text{h-space}} + \underbrace{b_h}_{\text{h-space bias}})$$

Recurrent Neural Networks



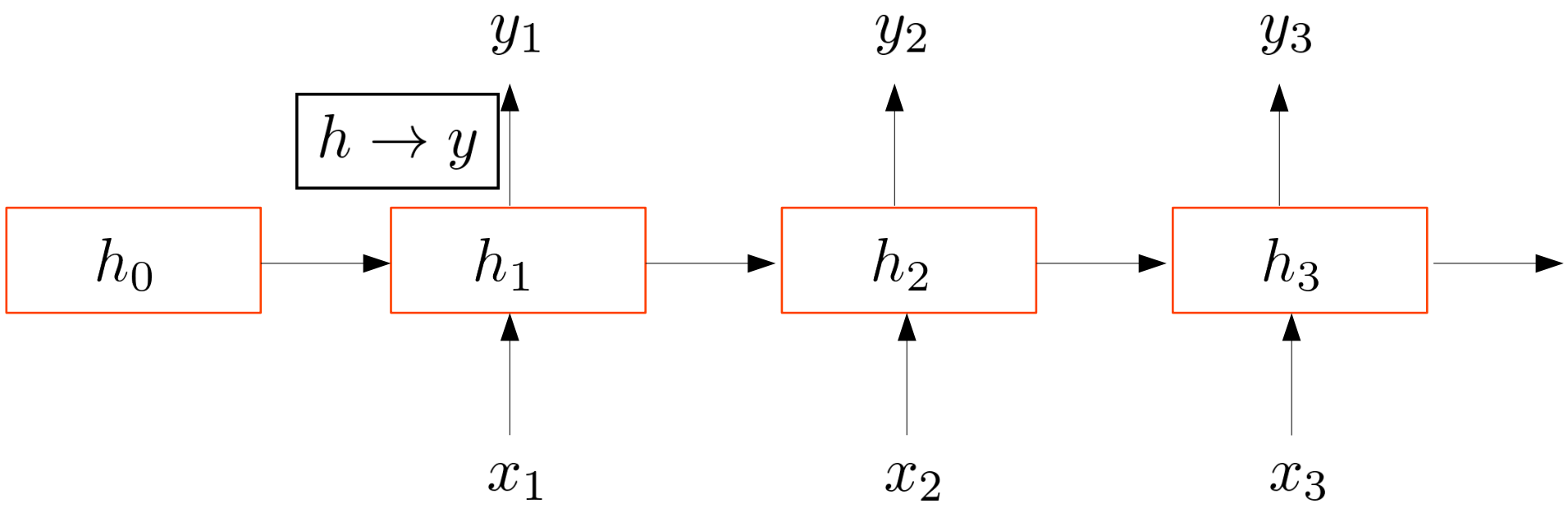
$$h_t = \sigma(\underbrace{W_{hx}}_{\text{x-space to h-space}} \underbrace{x_t}_{\text{x-space}} + \underbrace{W_{hh}}_{\text{h-space to h-space}} \underbrace{h_{t-1}}_{\text{h-space}} + \underbrace{b_h}_{\text{h-space bias}})$$

Recurrent Neural Networks

Good notation helps to keep track of variables

$$\underbrace{h_t}_{\text{h-space}} = \sigma\Big(\underbrace{W_{hx}}_{\text{x-space to h-space}} \underbrace{x_t}_{\text{x-space}} + \underbrace{W_{hh}}_{\text{h-space to h-space}} \underbrace{h_{t-1}}_{\text{h-space}} + \underbrace{b_h}_{\text{h-space bias}} \Big)$$

Recurrent Neural Networks



$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \rho(W_{yh}h_t + b_y)$$

Recurrent Neural Networks

Good notation helps to keep track of variables

$$\underbrace{h_t}_{\text{h-space}} = \sigma(\underbrace{W_{hx}}_{\text{x-space to h-space}} \underbrace{x_t}_{\text{x-space}} + \underbrace{W_{hh}}_{\text{h-space to h-space}} \underbrace{h_{t-1}}_{\text{h-space}} + \underbrace{b_h}_{\text{h-space bias}})$$

$$\underbrace{y_t}_{\text{y-space}} = \rho(\underbrace{W_{yh}}_{\text{h-space to y-space}} \underbrace{h_t}_{\text{h-space}} + \underbrace{b_y}_{\text{y-space}})$$

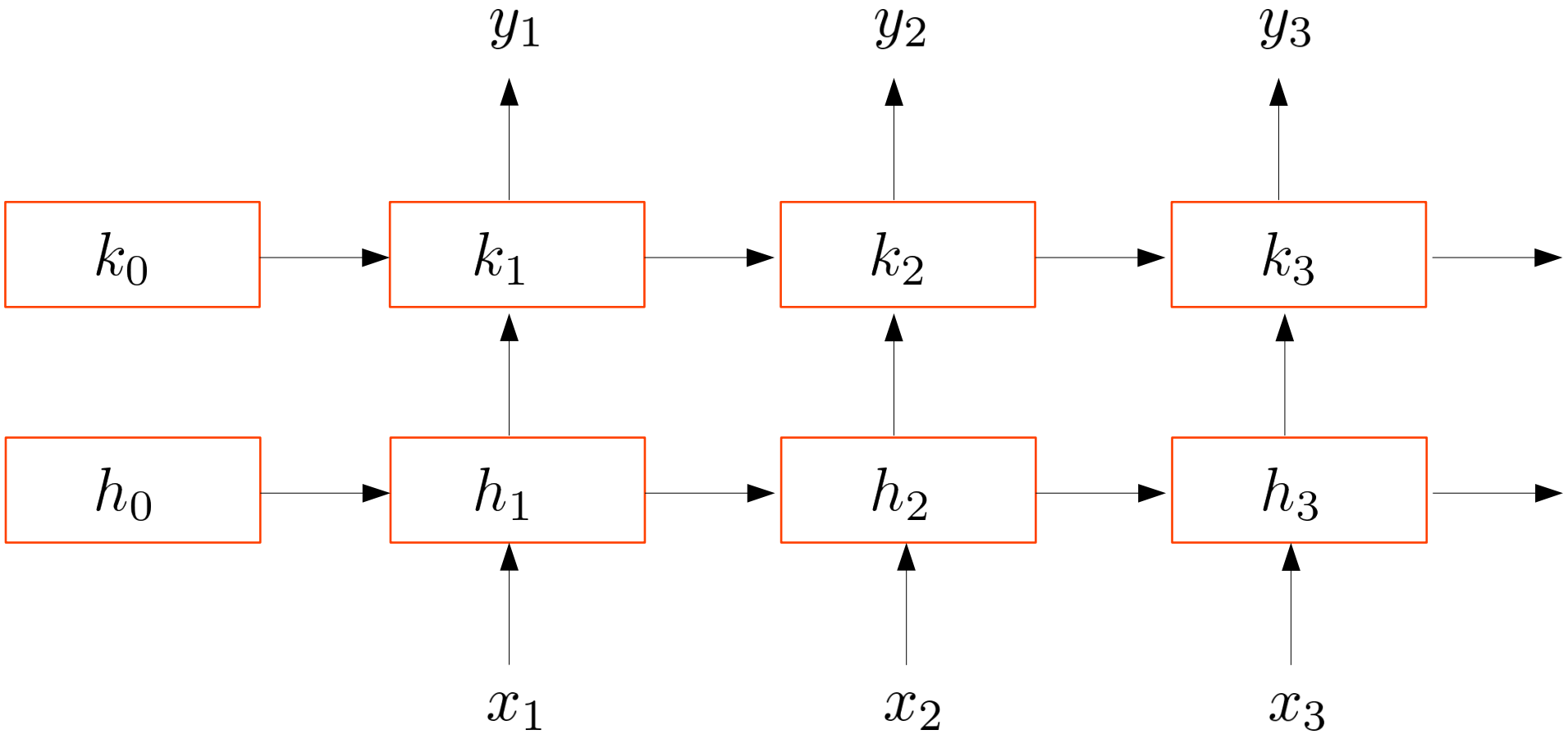
Recurrent Neural Networks

Weights and biases give flexibility in learning

$$\underbrace{h_t}_{\text{h-space}} = \sigma(\underbrace{W_{hx}}_{\text{x-space to h-space}} \underbrace{x_t}_{\text{x-space}} + \underbrace{W_{hh}}_{\text{h-space to h-space}} \underbrace{h_{t-1}}_{\text{h-space}} + \underbrace{b_h}_{\text{h-space bias}})$$

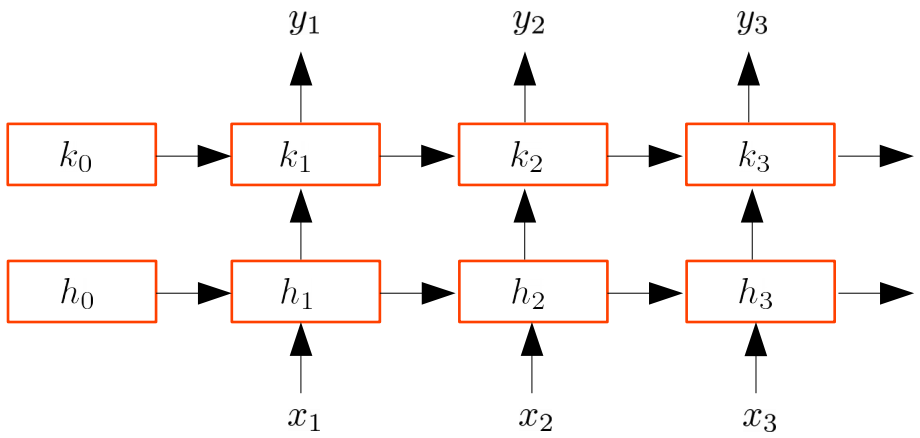
$$\underbrace{y_t}_{\text{y-space}} = \rho(\underbrace{W_{yh}}_{\text{h-space to y-space}} \underbrace{h_t}_{\text{h-space}} + \underbrace{b_y}_{\text{y-space}})$$

Recurrent Neural Networks



Can now start stacking layers together

Recurrent Neural Networks



$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

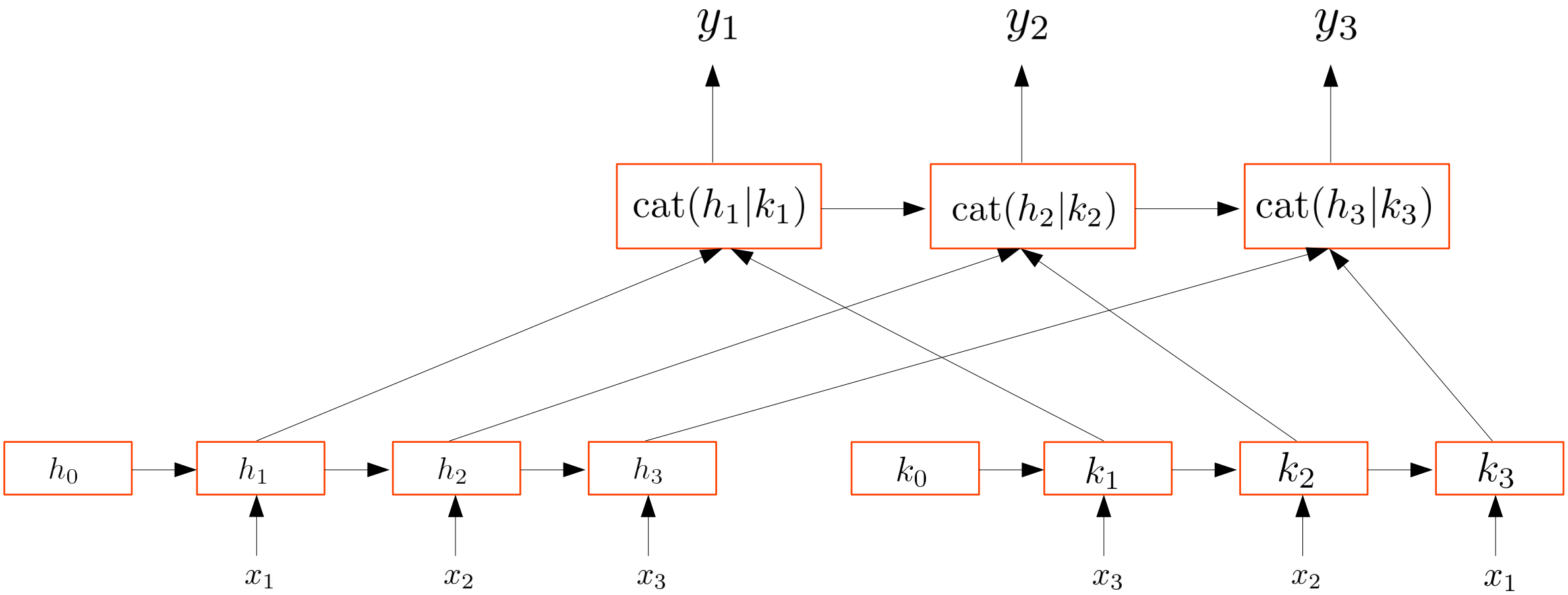
$$k_t = \eta(W_{kh}h_t + W_{kk}k_{t-1} + b_k)$$

$$y_t = \rho(W_{yk} + b_y)$$

Do as many times as you like (up to some training issues)

ρ depends on type of problem: classification, regression

Recurrent Neural Networks



Sometimes full input sequence known

Bi-directional RNNs

Recurrent Neural Networks

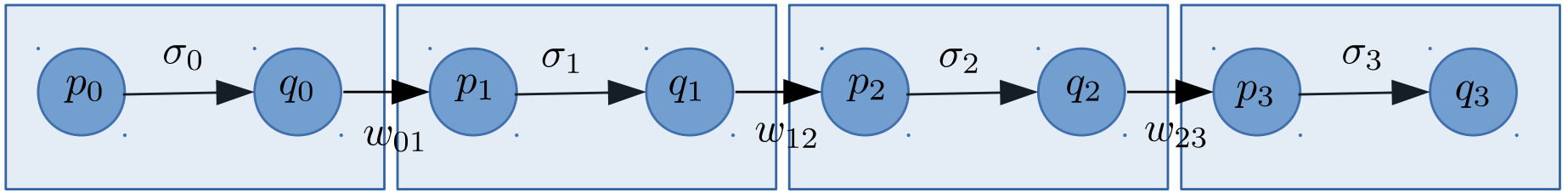
Won't go into training details (at least for this iteration of the course)

But RNNs have trouble with long time-sequences

Recurrent Neural Networks

Recall from our backprop computation

Input



Output

$$\frac{\partial C}{\partial w_{23}} = \boxed{(q_3 - y)\sigma_3'(p_3)}\boxed{q_2}$$

$$\frac{\partial C}{\partial w_{12}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)}\boxed{q_1}$$

$$\frac{\partial C}{\partial w_{01}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)w_{12}\sigma_1'(p_1)}\boxed{q_0}$$

Need these derivatives to do gradient descent

Recurrent Neural Networks

Can think of every layer as a new time-step

$\implies$ $w_{t,t+1}$ propagates us from time t to t+1

for long sequences, have many such weights

$$\frac{\partial C}{\partial w_{23}} = \boxed{(q_3 - y)\sigma_3'(p_3)}\boxed{q_2}$$

$$\frac{\partial C}{\partial w_{12}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)}\boxed{q_1}$$

$$\frac{\partial C}{\partial w_{01}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)w_{12}\sigma_1'(p_1)}\boxed{q_0}$$

Recurrent Neural Networks

These chains would get long

$$\frac{\partial C}{\partial w_{01}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)w_{12}\sigma_1'(p_1)}\boxed{q_0}$$

Multiplying many weights and many activation derivatives together

Easy to get either:

$$0.1 * 0.3 * -0.2 = -0.006(\text{small})$$

$$10 * -40 * 50 = -2000(\text{large})$$

Recurrent Neural Networks

These chains would get long

$$\frac{\partial C}{\partial w_{01}} = \boxed{(q_3 - y)\sigma_3'(p_3)w_{23}\sigma_2'(p_2)w_{12}\sigma_1'(p_1)}\boxed{q_0}$$

Multiplying many weights and many activation derivatives together

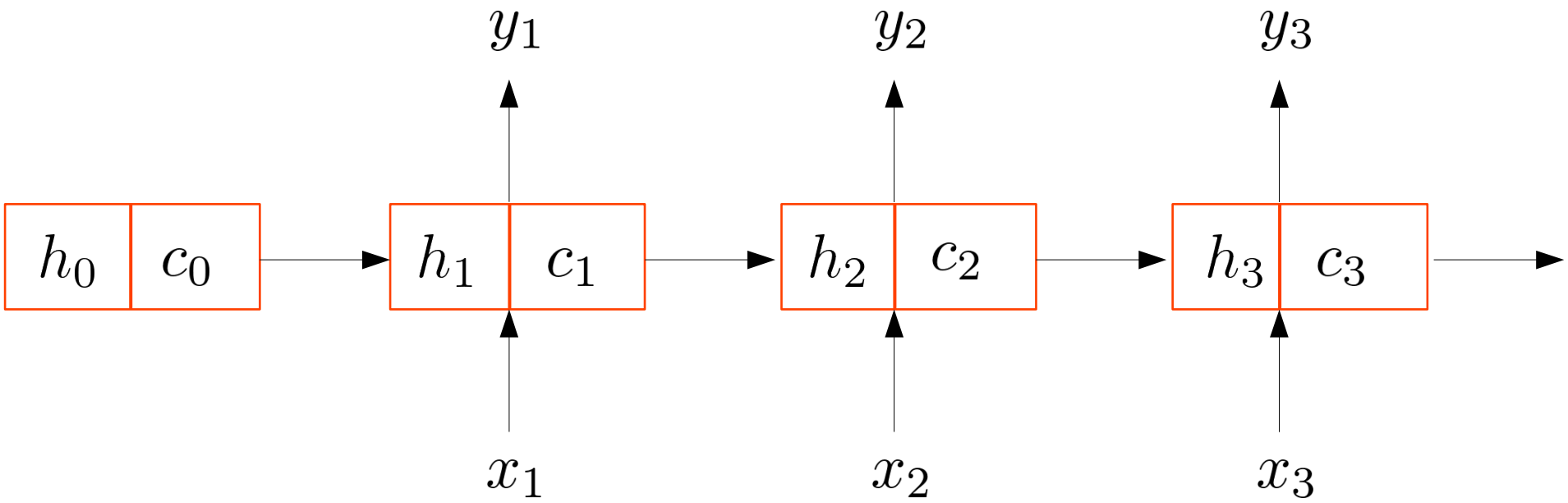
Easy to get either:

$$0.1 * 0.3 * -0.2 = -0.006(\text{small}) \implies \text{Vanishing gradients} \implies \text{no learning}$$

$$10 * -40 * 50 = -2000(\text{large}) \implies \text{Exploding gradients} \implies \text{Bounce in weight space}$$

LSTM: Long Short Term Memory

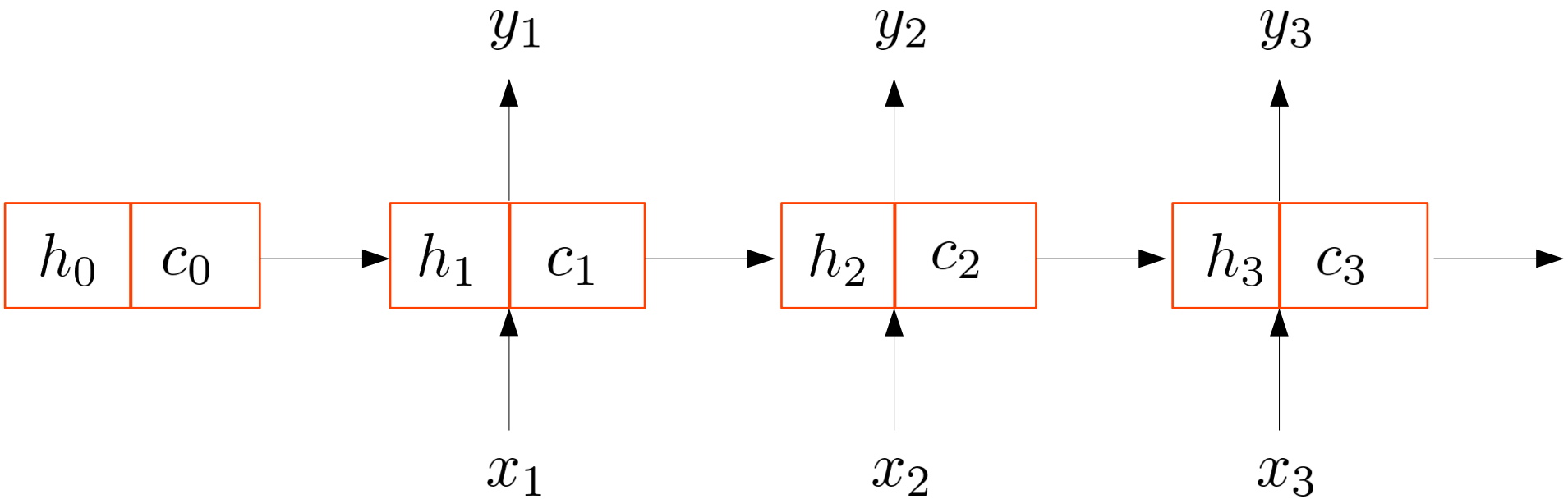
Solution: Introduce explicit long-term "memory"



h_t = immediate hidden state (as before)

c_t = long-term memory (new)

LSTM: Long Short Term Memory



h_t = immediate hidden state (as before) c_t = long-term memory (new)

Questions:

How does c_t get updated?

What is the relationship between short-term memory, h_t and long-term memory, c_t?

LSTM: Long Short Term Memory

Generate candidate for new long-term memory (cell state) :

$$\tilde{c_t} = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

LSTM: Long Short Term Memory

Generate candidate for new long-term memory (cell state) :

$$\tilde{c_t} = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

Have previous cell state, c_t

LSTM: Long Short Term Memory

Generate candidate for new long-term memory (cell state) :

$$\tilde{c_t} = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

Have previous cell state, c_t

What part of previous cell to **forget**? $f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$

What part of candidate, $\tilde{c_t}$ to write/**input** into cell, c_t? $i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$

LSTM: Long Short Term Memory

Note: These all have the same structure! Just different weights and biases

They only depend on the input, x_t and the previous short-term memory, h_{t-1}

$$\tilde{c_t} = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

LSTM: Long Short Term Memory

Update long-term memory

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

$* =$ element-wise multiplication

$f_t, i_t =$ vectors of numbers between 0 and 1 (soft bits)

$$\underbrace{c_t}_{\text{new cell}} = \underbrace{f_t * c_{t-1}}_{\text{forget parts of old cell}} + \underbrace{i_t * \tilde{c}_t}_{\text{write cell candidate elements into old cell}}$$

LSTM: Long Short Term Memory

Recap :

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

$$\tilde{c_t} = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c_t}$$

LSTM: Long Short Term Memory

Recap :

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

$$\tilde{c_t} = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c_t}$$

One last missing piece: Updating h_t

LSTM: Long Short Term Memory

One last missing piece: Updating h_t

Generate similar output mask: $o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$

$$h_t = \underbrace{o_t * \rho(c_t)}_{\text{pick parts of cell to write to short-term memory}}$$

LSTM: Long Short Term Memory

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$

Compute decisions:

Forget (delete)

Input (write)

Output (read)

Depend on: current input + previous short-term memory

$$\tilde{c}_t = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

Compute new long-term memory candidate

Delete parts of long-term memory

Write from candidate to parts of long-term memory

$$h_t = o_t * \rho(c_t)$$

Update short-term memory for next iteration

GRU: Gated Recurrent Units

Not the only choice!

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$

Compute decisions:

Forget (delete)

Input (write)

Output (read)

Depend on: current input + previous short-term memory

$$\tilde{c}_t = \rho(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

Compute new long-term memory candidate

Delete parts of long-term memory

Write from candidate to parts of long-term memory

$$h_t = o_t * \rho(c_t)$$

Update short-term memory for next iteration

GRU: Gated Recurrent Units

Much simpler candidate

Only one memory state: h_t

$$\text{Update: } h_t = \underbrace{(1 - u_t) * h_{t-1}}_{\text{what to copy}} + \underbrace{u_t * \tilde{h}_t}_{\text{what to write}}$$

More appealing: Can interpret u_t as probability of writing to to vector component

GRU: Gated Recurrent Units

$$\text{Update: } h_t = \underbrace{(1 - u_t) * h_{t-1}}_{\text{what to copy}} + \underbrace{u_t * \tilde{h}_t}_{\text{what to write}}$$

$$\tilde{h}_t = \rho(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

same structure as always

Not quite: add flexibility to choose what parts of h_{t-1} to use

$$\tilde{h}_t = \rho(W_{hx}x_t + W_{hh}(h_{t-1} * r_t) + b_h)$$

GRU: Gated Recurrent Units

$$\text{Update: } h_t = \underbrace{(1 - u_t) * h_{t-1}}_{\text{what to copy}} + \underbrace{u_t * \tilde{h}_t}_{\text{what to write}}$$

$$\tilde{h}_t = \rho(W_{hx}x_t + W_{hh}(h_{t-1} * r_t) + b_h)$$

$$\text{Update gate: } u_t = \sigma(W_{ux}x_t + W_{uh}h_{t-1} + b_u)$$

$$\text{Reset gate: } r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r)$$

GRU: Gated Recurrent Units

$$\text{Update: } h_t = \underbrace{(1 - u_t) * h_{t-1}}_{\text{what to copy}} + \underbrace{u_t * \tilde{h}_t}_{\text{what to write}}$$

$$\tilde{h}_t = \rho(W_{hx}x_t + W_{hh}(h_{t-1} * r_t) + b_h)$$

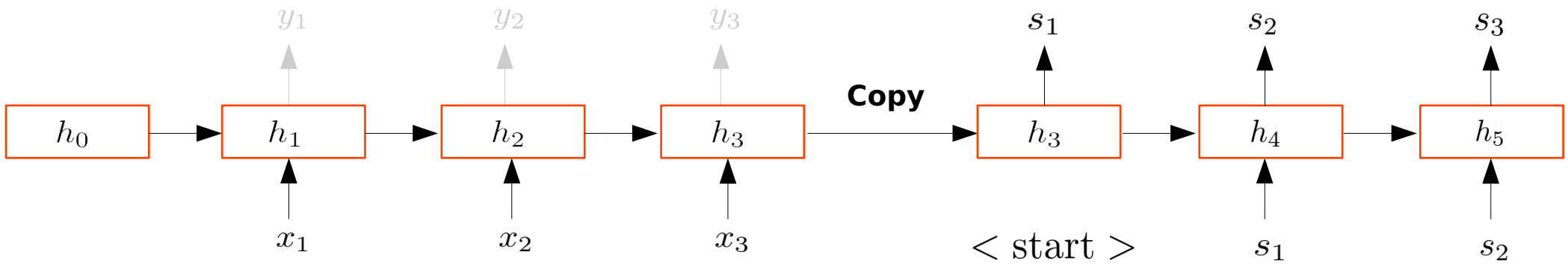
$$\text{Update gate: } u_t = \sigma(W_{ux}x_t + W_{uh}h_{t-1} + b_u)$$

$$\text{Reset gate: } r_t = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r)$$

Much more computationally efficient than LSTMs

How do I use all this?: Language Translation

Generate Spanish Sentence Tokens = “Decoder”



English Sentence Tokens = “Encoder”

Red Hat

Conclusion

- Many more components: attention, transformers, etc.
- These building blocks make modern neural networks very flexible for a wide variety of tasks.
- Many avenues of research and many questions:
  - Do neural networks actually “understand” language?
  - How do they learn? Mathematical foundations of probability and optimization in very high-dimensions.
  - Can we attack such networks by passing malformed inputs?
  - How can they be made more robust?
  - And many more….

Thank you

Red Hat is the world’s leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Red Hat