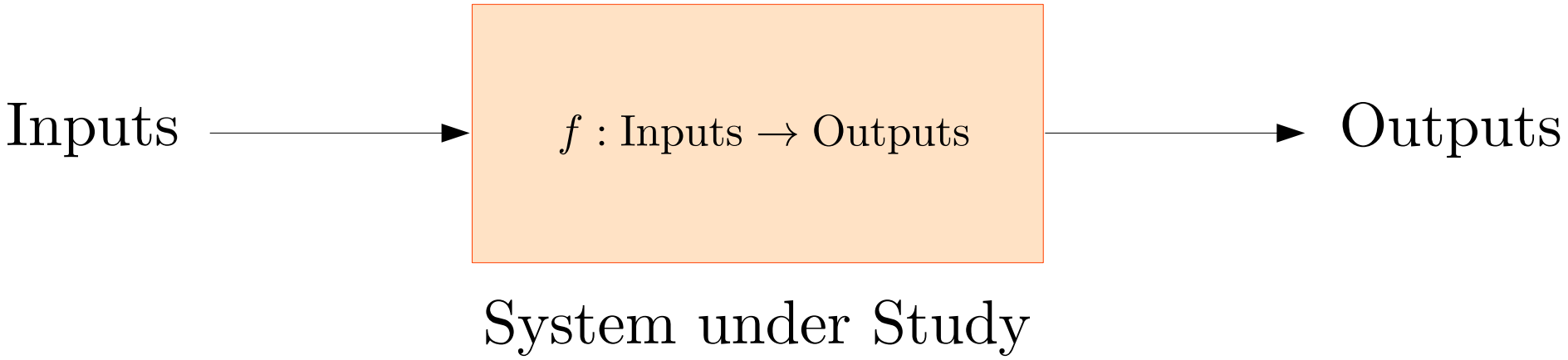
What is Machine Learning?

Sanjay Arora
AI Center of Excellence
Red Hat

Red Hat

Why and how does something work?



Question: what are the rules governing the system i.e. what is f?

The Scientific Method

Most successful approach till now:

- Observe system with different inputs and measure outputs
- Guess rules or guess f
- Make predictions from guessed rules
- Compare predictions to outputs
- If predictions “match” outputs, you **might** have the correct rules
- If predictions don’t match outputs, you are definitely wrong

Caveats

- A scientific theory can never be **proven** to be correct. There’s no proof the sun will rise tomorrow – just an overwhelming possibility.
- Comparison of the predictions and outputs is more subtle than it appears. It depends on the degree of accuracy required. Rules might predict outputs within 10% but not within 1% and that might be sufficient to explain the core elements of the system under study.
- Process not linear in time!
- The process for guessing rules is crucial.
- Simplicity is desirable. The best (in terms of predictive power) scientific theories tend to be simple.

How to guess the rules?

No standard way but each field has its own intuitions, methods, ideas that worked in the past that can be tried again

One way:
Collect data and scan for patterns. How Kepler found the laws for planetary motion

What is Machine Learning?

- Collect data and find patterns to discover rules.
- Let computers scan through several guesses/hypotheses and find the best ones.
- Need to invent algorithms that can scan multiple hypotheses efficiently (in time, memory, amount of data required).

Terminology: Model

Our guess of the rules

Terminology: Learning

Process of scanning data to find the best rules

Algorithms

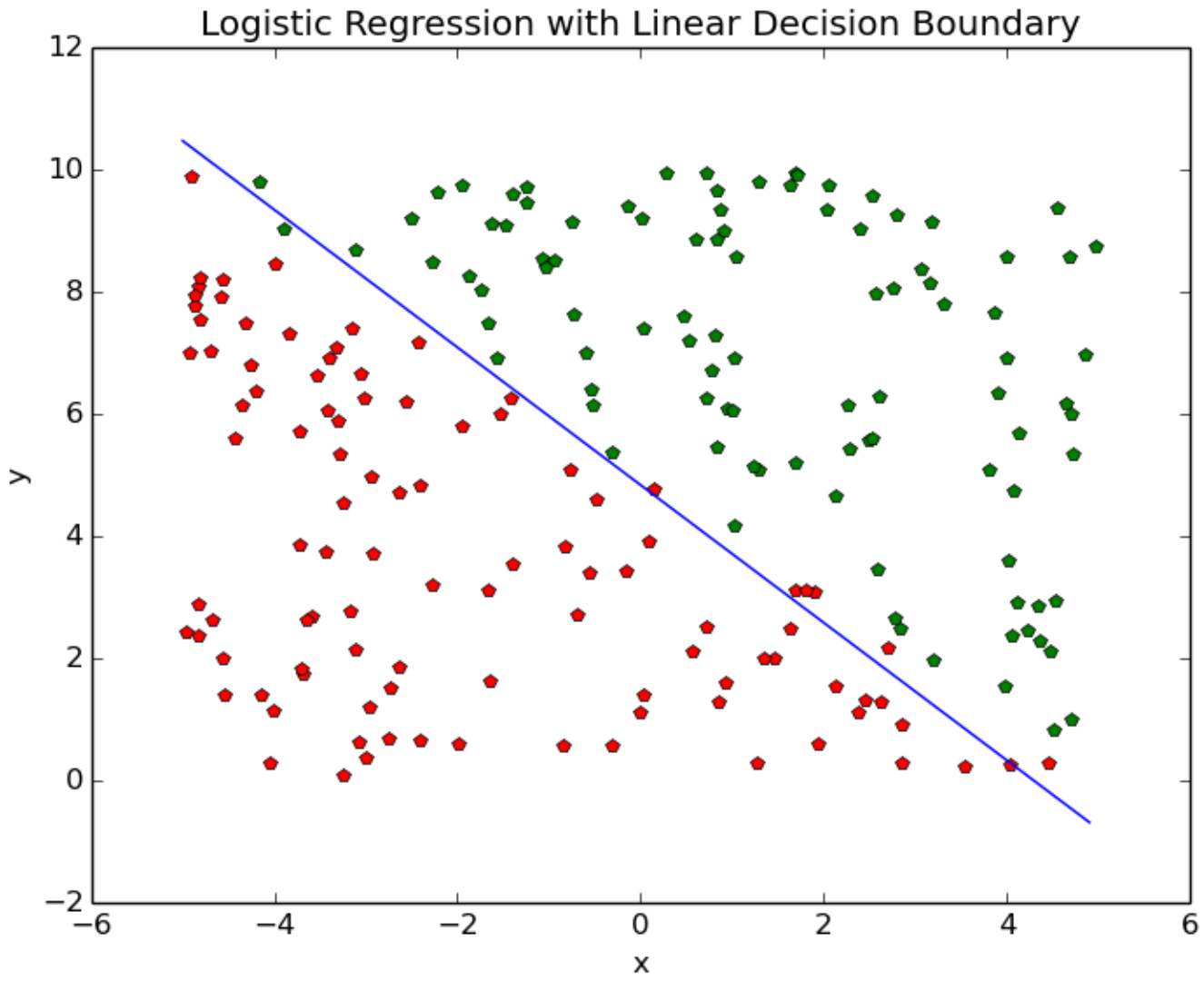
Every field has a few simple examples that help one build intuition

No mathematical details here → Just pictures

Problem Statement

$$\text{Given features} : x_1, x_2, \ldots, x_n$$

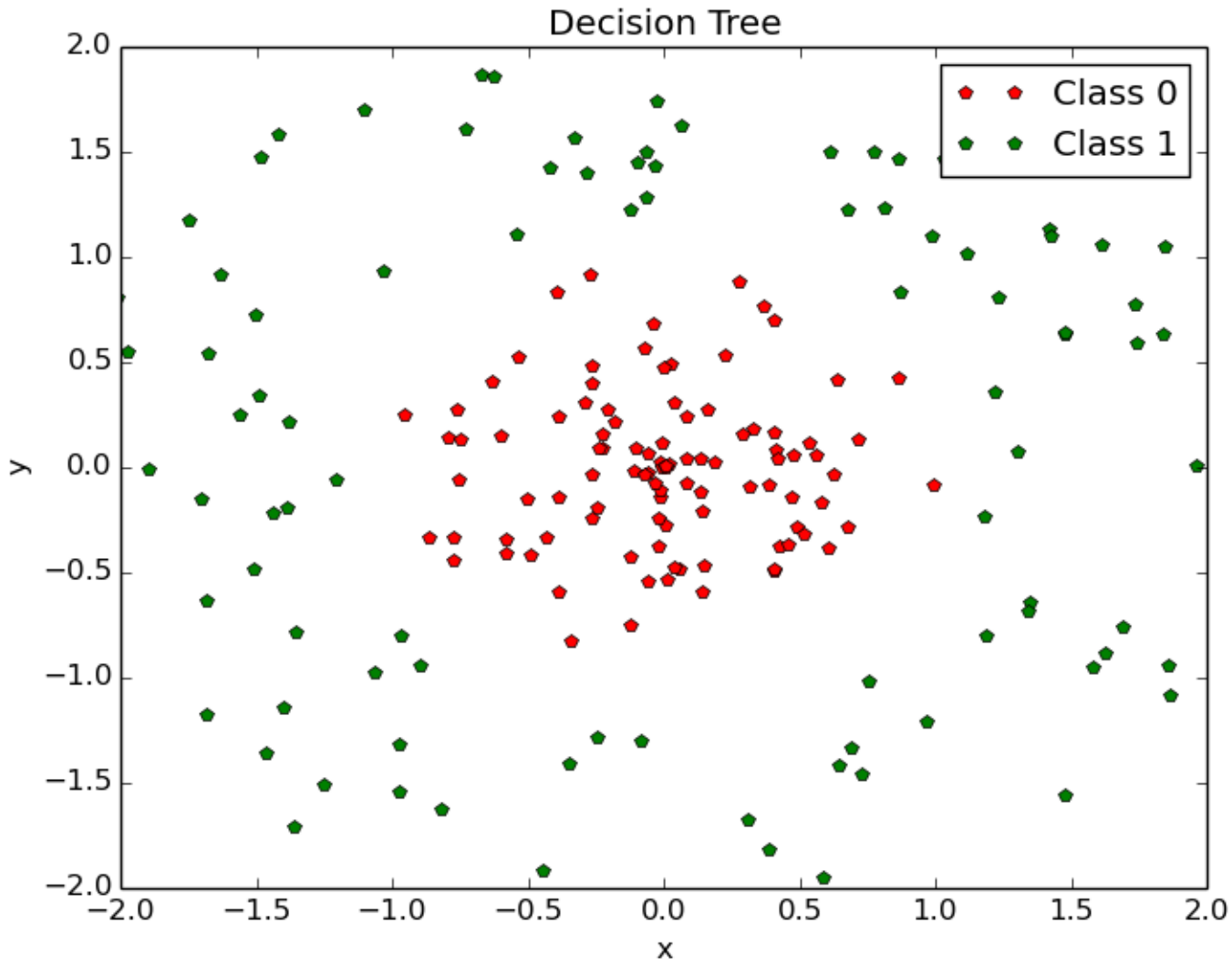
$$\text{predict class or label, y} = 0 \text{ or } 1$$

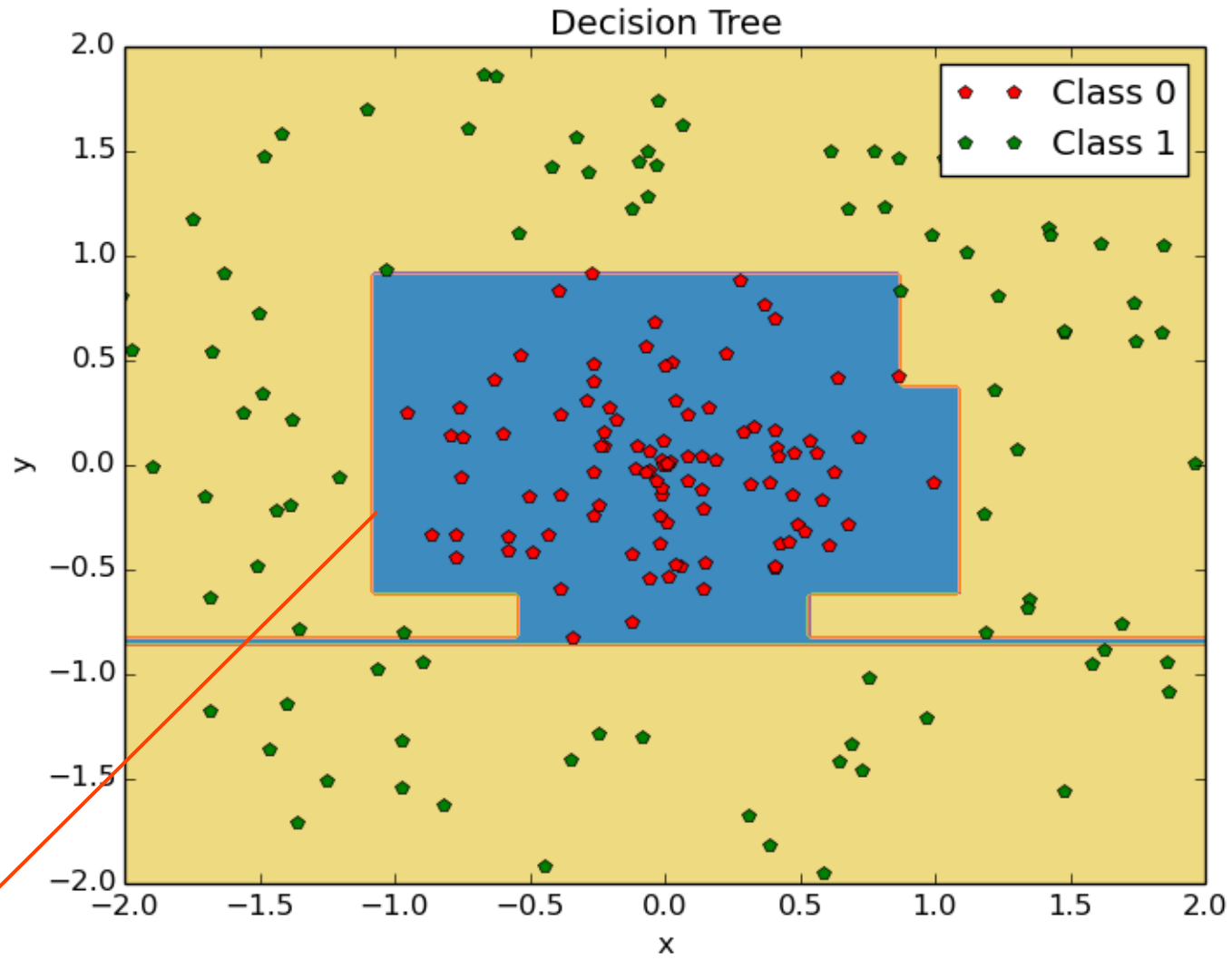


Logistic Regression = “Draw a line separating red from green”

(Linear) Support Vector Machine = “Draw a road separating red from green with highest width”

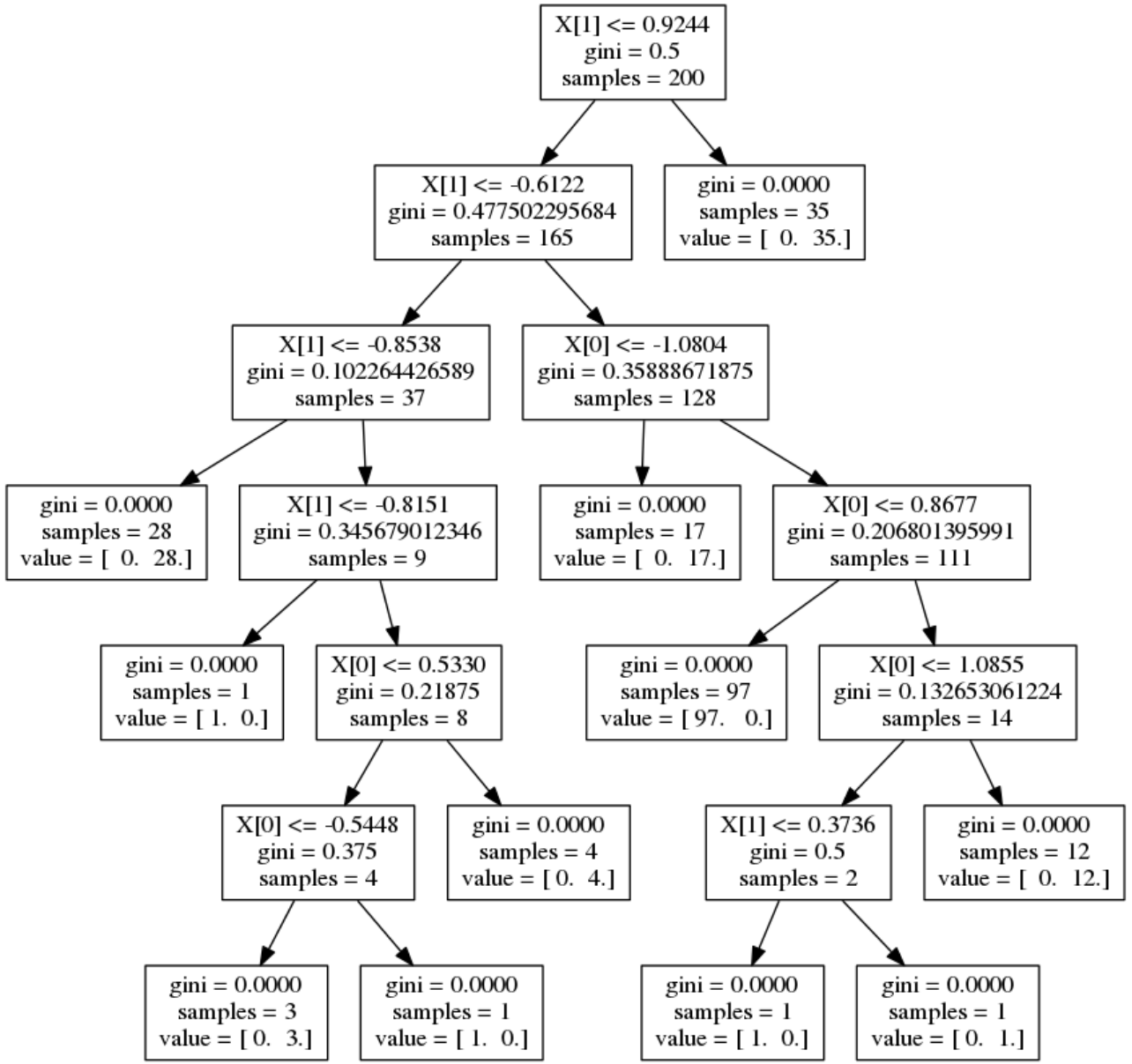
Red Hat

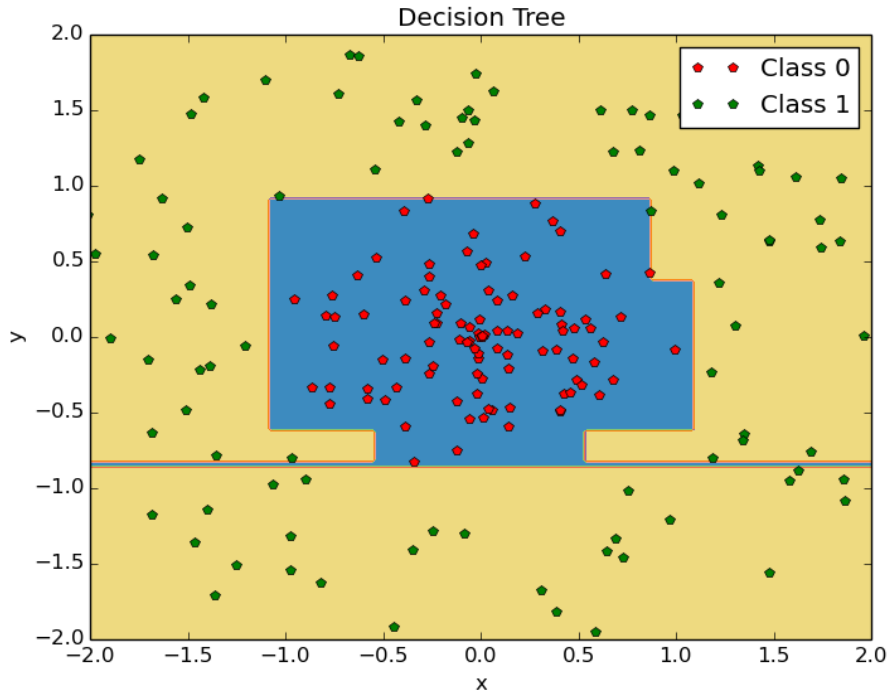

Decision Tree
Class 0
Class 1
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x



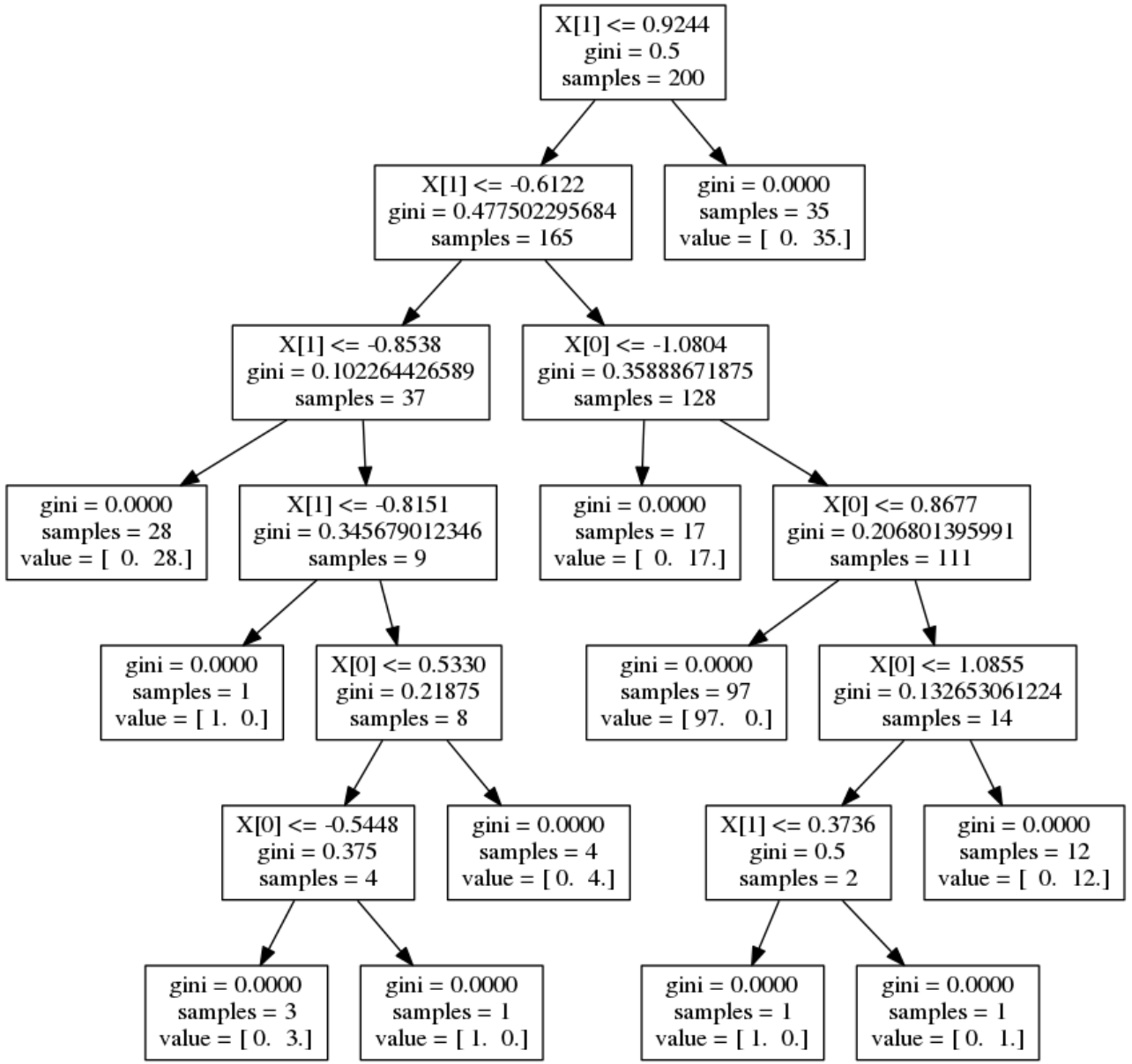
Any if-else statement on x or y ⟺ Horizontal or vertical line

Red Hat

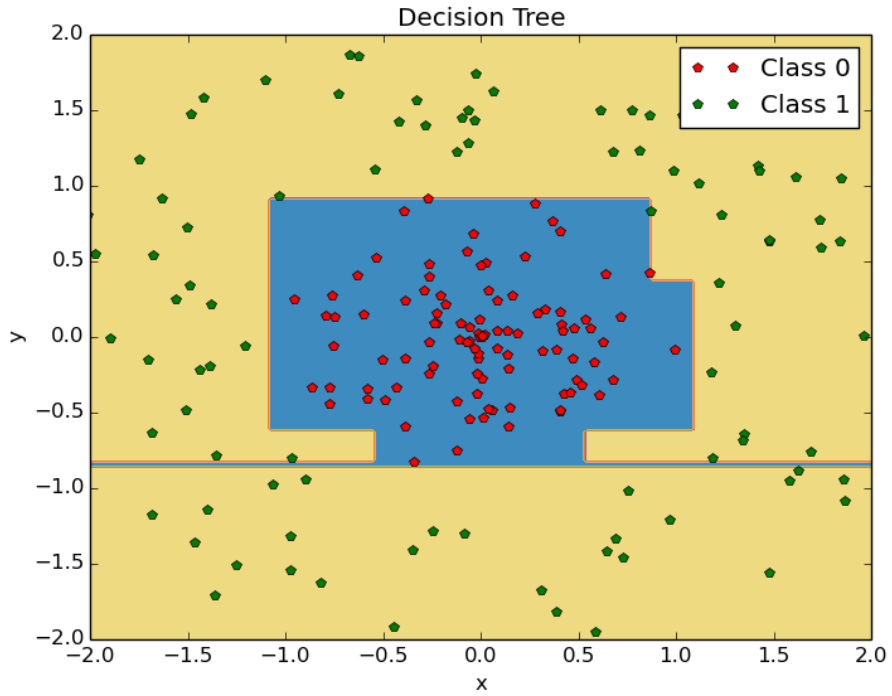

X[1] <= 0.9244
gini = 0.5
samples = 200
X[1] <= -0.6122
gini = 0.477502295684
samples = 165
gini = 0.0000
samples = 35
value = [0. 35.]
X[1] <= -0.8538
gini = 0.102264426589
samples = 37
X[0] <= -1.0804
gini = 0.35888671875
samples = 128
gini = 0.0000
samples = 28
value = [0. 28.]
X[1] <= -0.8151
gini = 0.345679012346
samples = 9
gini = 0.0000
samples = 17
value = [0. 17.]
X[0] <= 0.8677
gini = 0.206801395991
samples = 111
gini = 0.0000
samples = 1
value = [1. 0.]
X[0] <= 0.5330
gini = 0.21875
samples = 8
gini = 0.0000
samples = 97
value = [97. 0.]
X[0] <= 1.0855
gini = 0.132653061224
samples = 14
X[0] <= -0.5448
gini = 0.375
samples = 4
gini = 0.0000
samples = 4
value = [0. 4.]
X[1] <= 0.3736
gini = 0.5
samples = 2
gini = 0.0000
samples = 12
value = [0. 12.]
gini = 0.0000
samples = 3
value = [0. 3.]
gini = 0.0000
samples = 1
value = [1. 0.]
gini = 0.0000
samples = 1
value = [1. 0.]
gini = 0.0000
samples = 1
value = [0. 1.]


Decision Tree
Class 0
Class 1
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

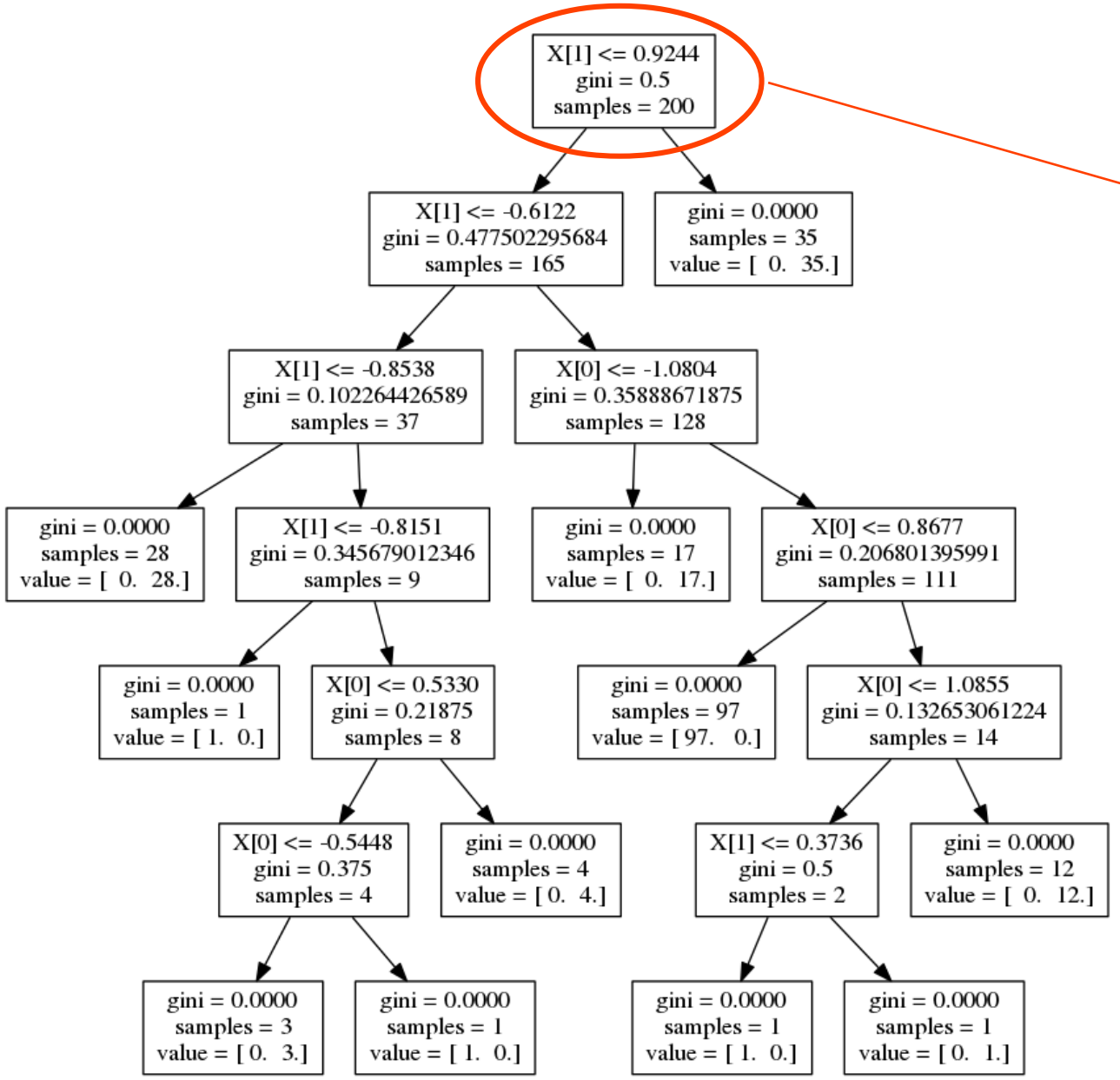
Red Hat



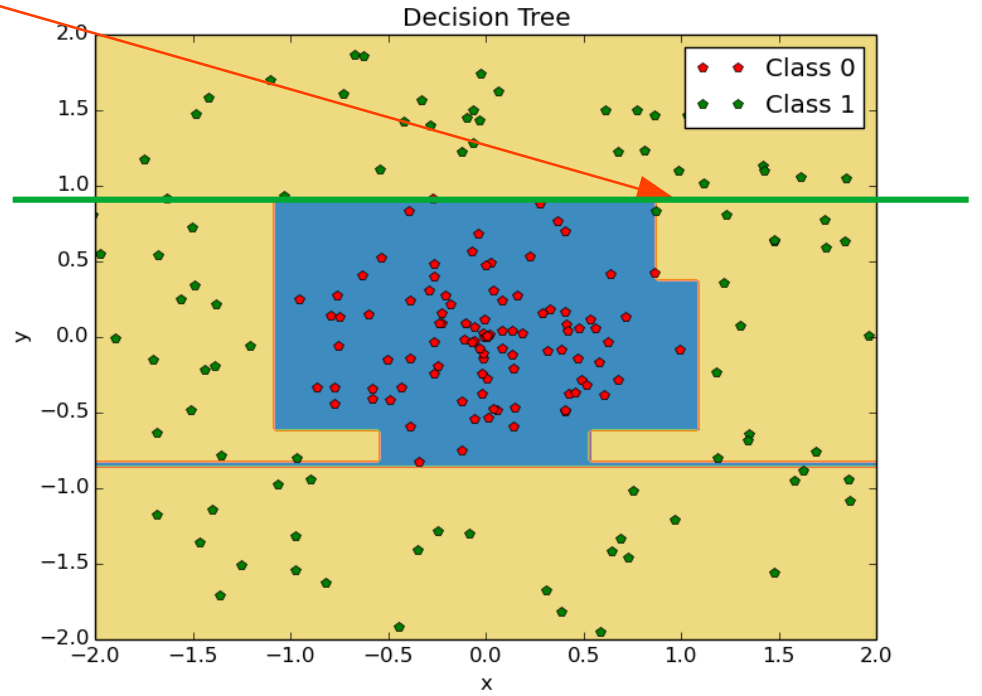
X[0] = x
X[1] = y



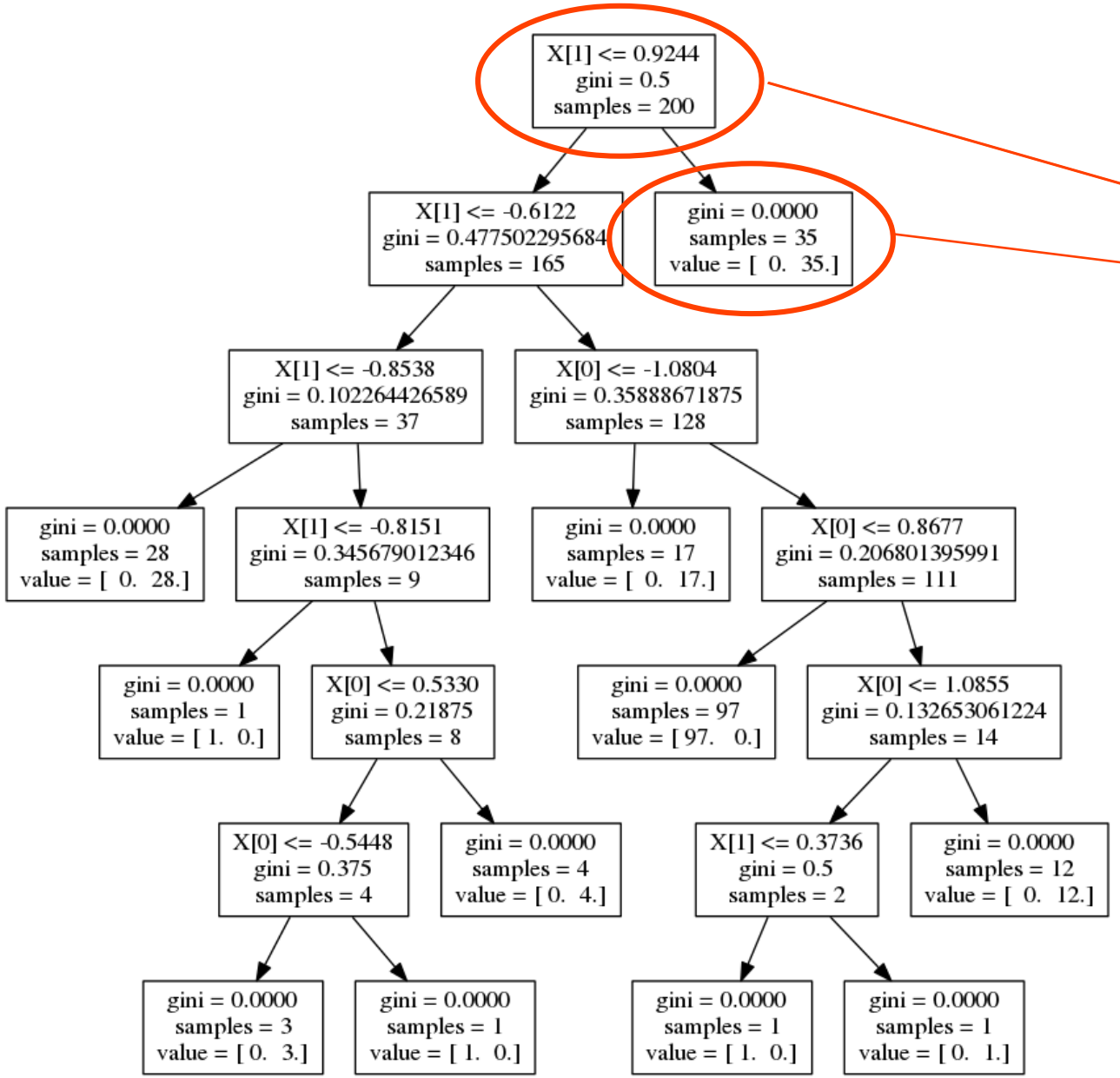
Red Hat



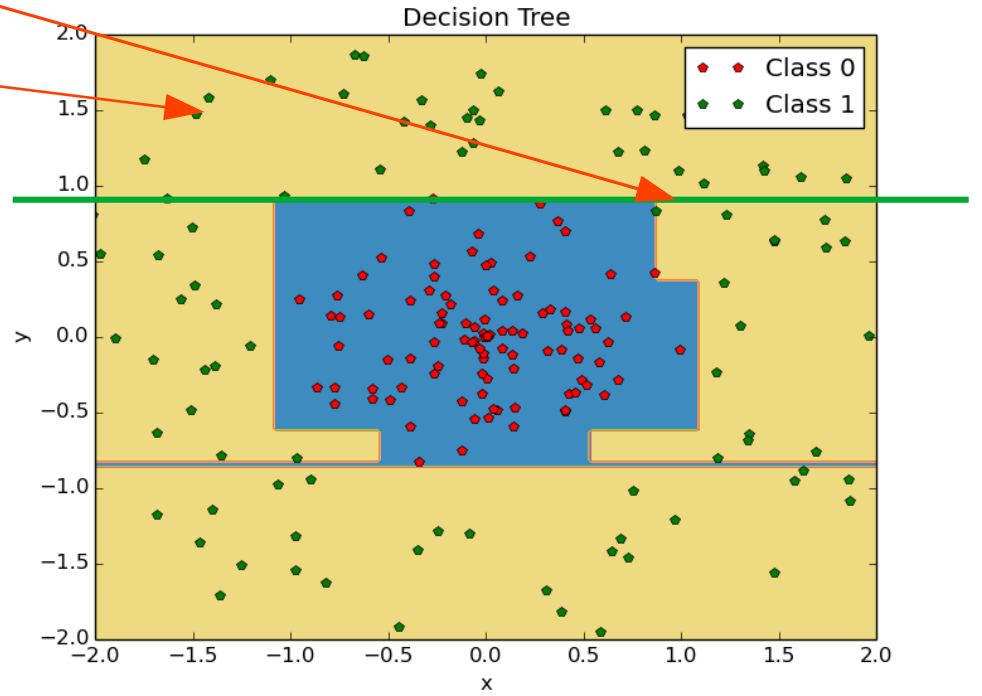
X[0] = x
X[1] = y



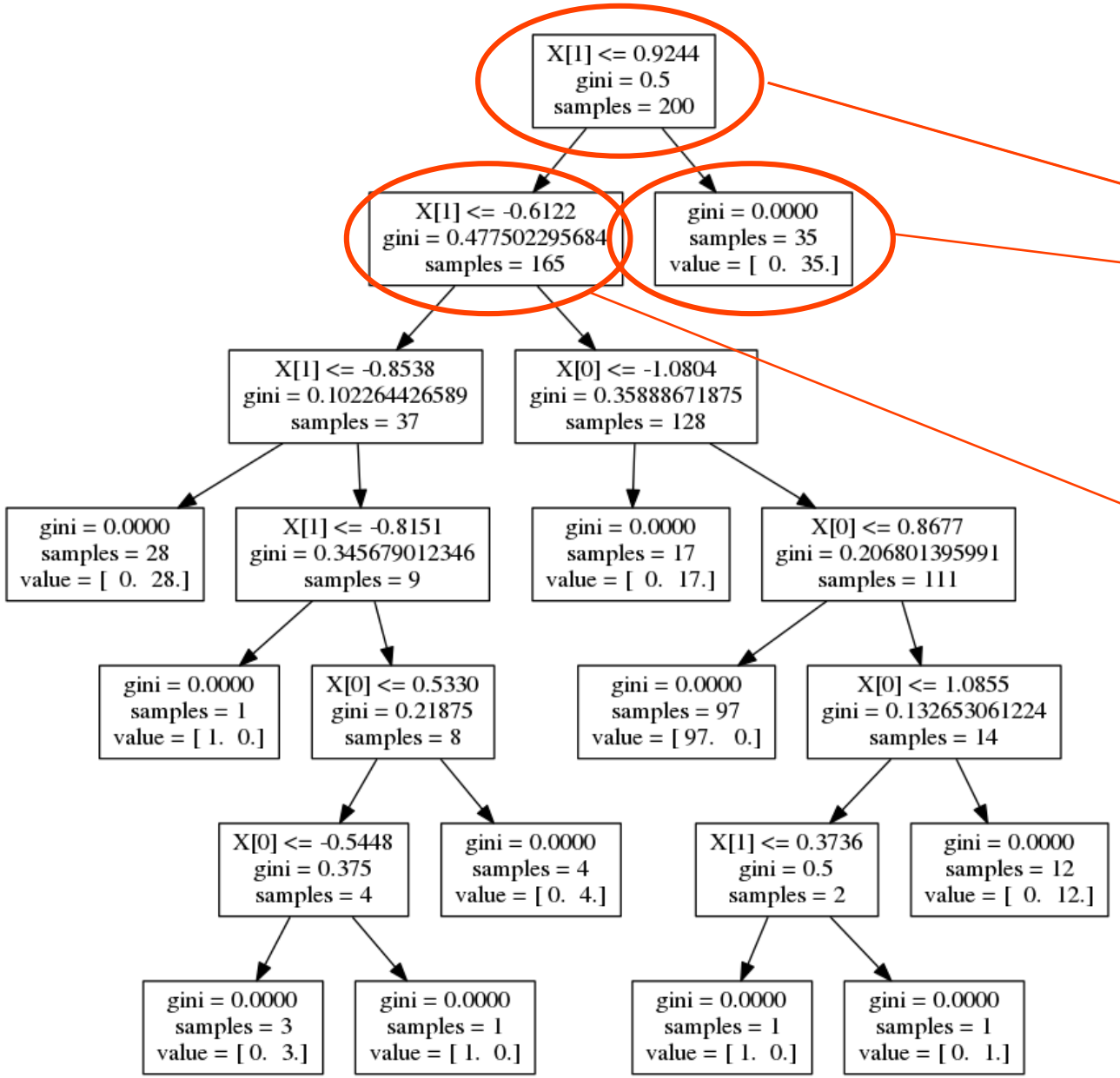
Red Hat



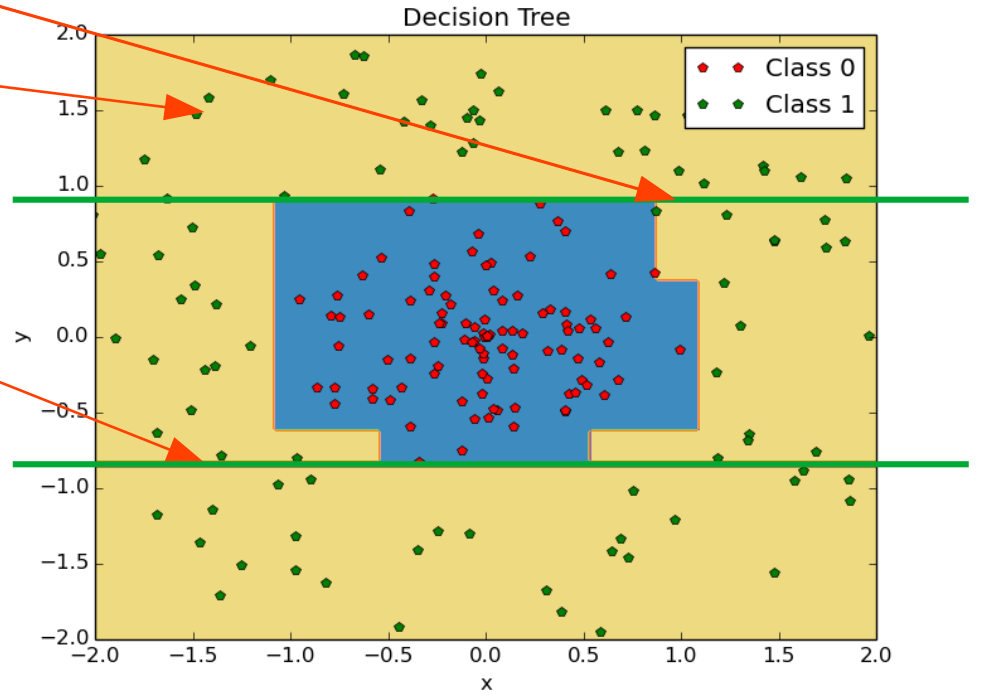
X[0] = x
X[1] = y



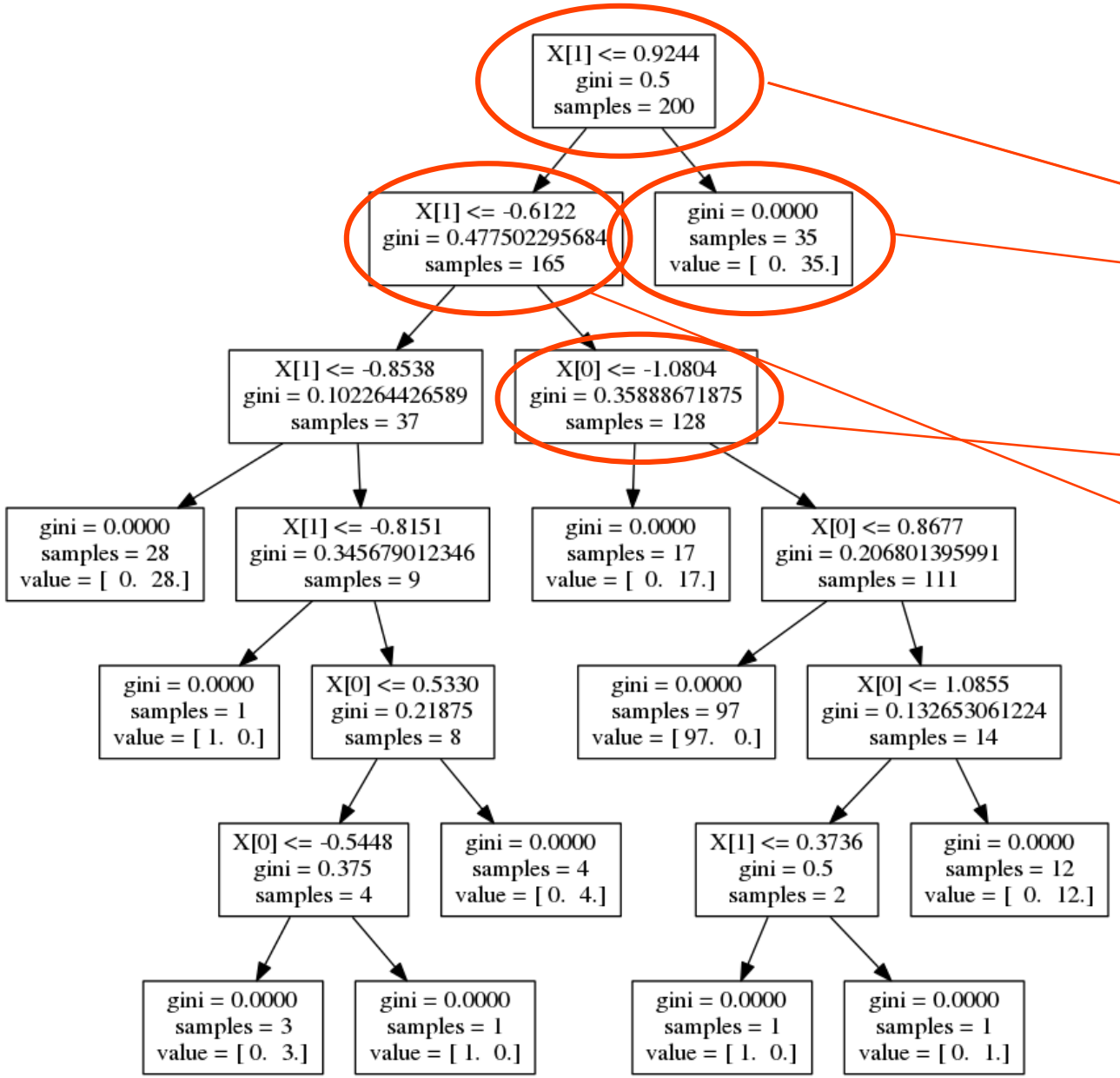
Red Hat



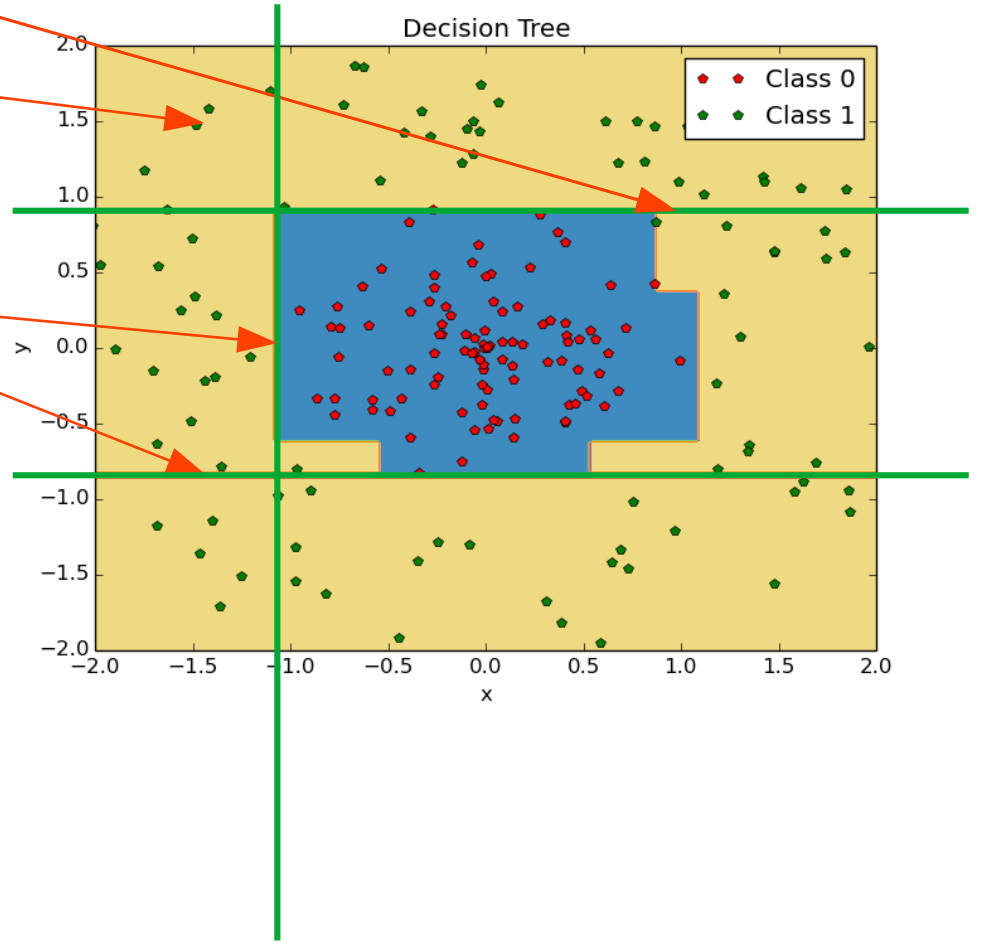
X[0] = x
X[1] = y



Red Hat

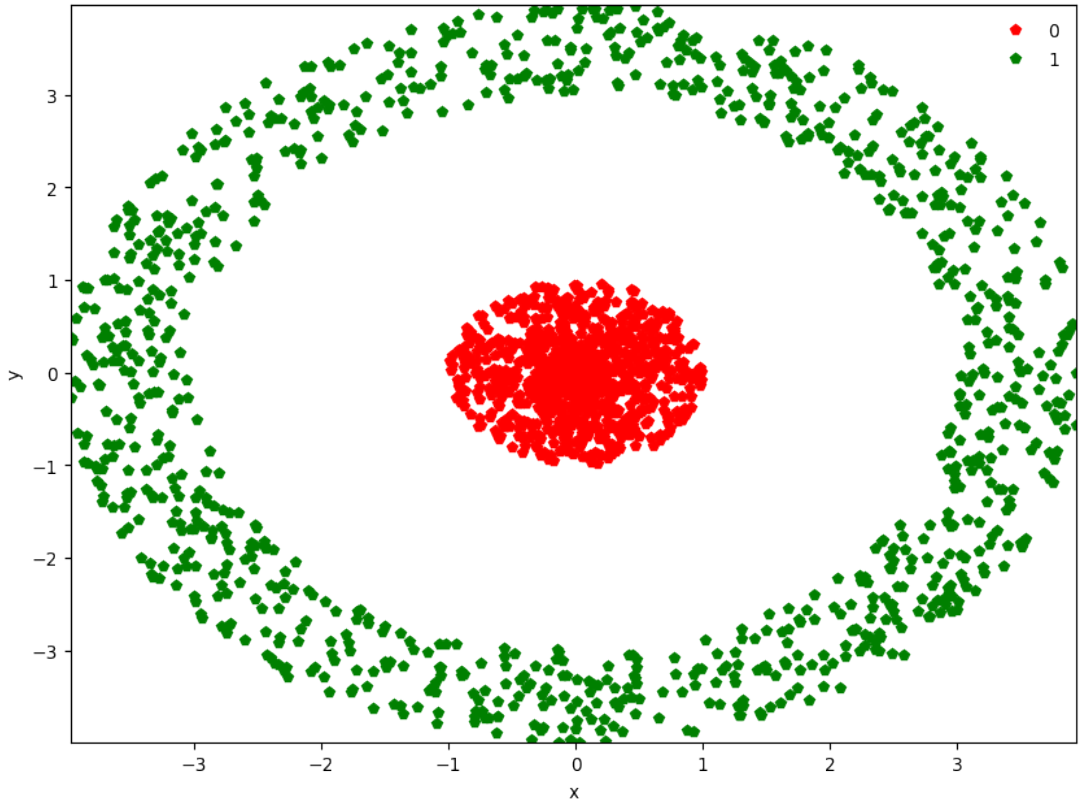


X[0] = x
X[1] = y



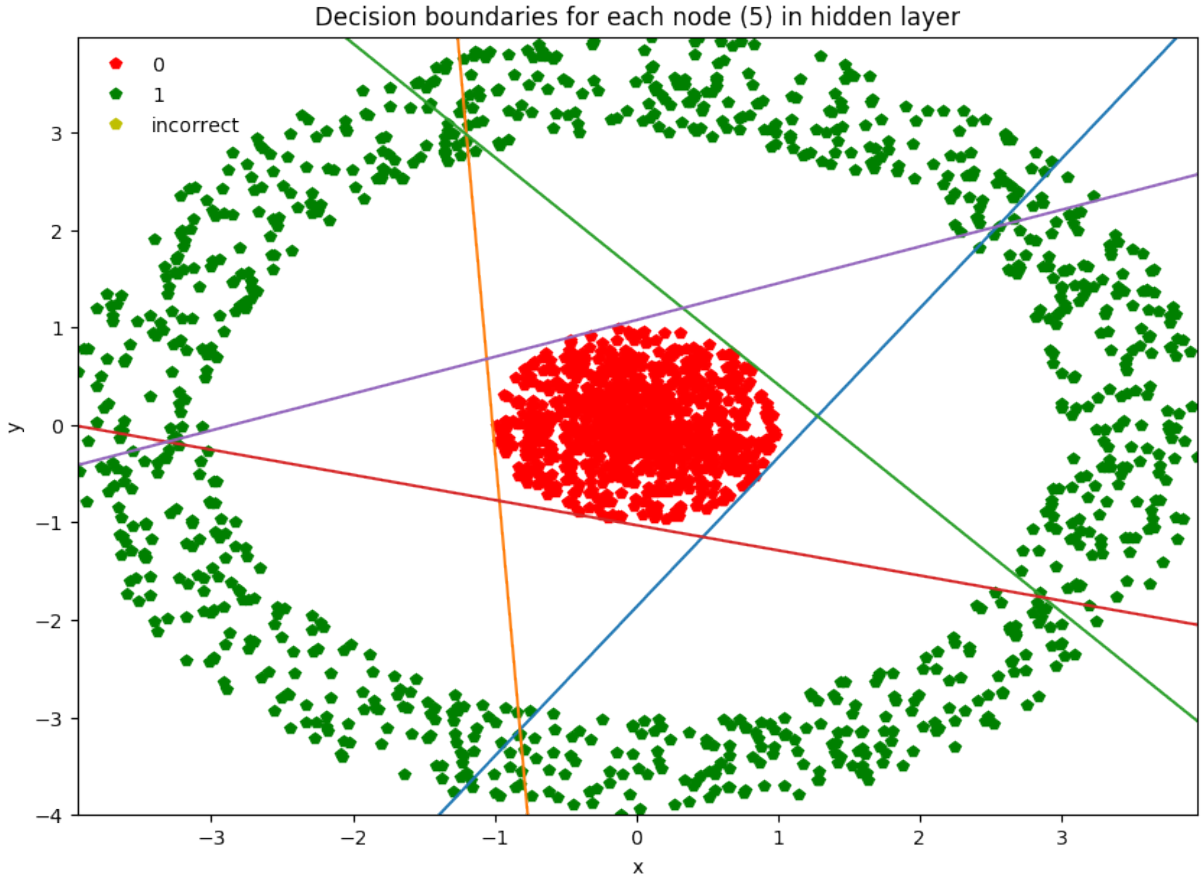
Red Hat

Problem



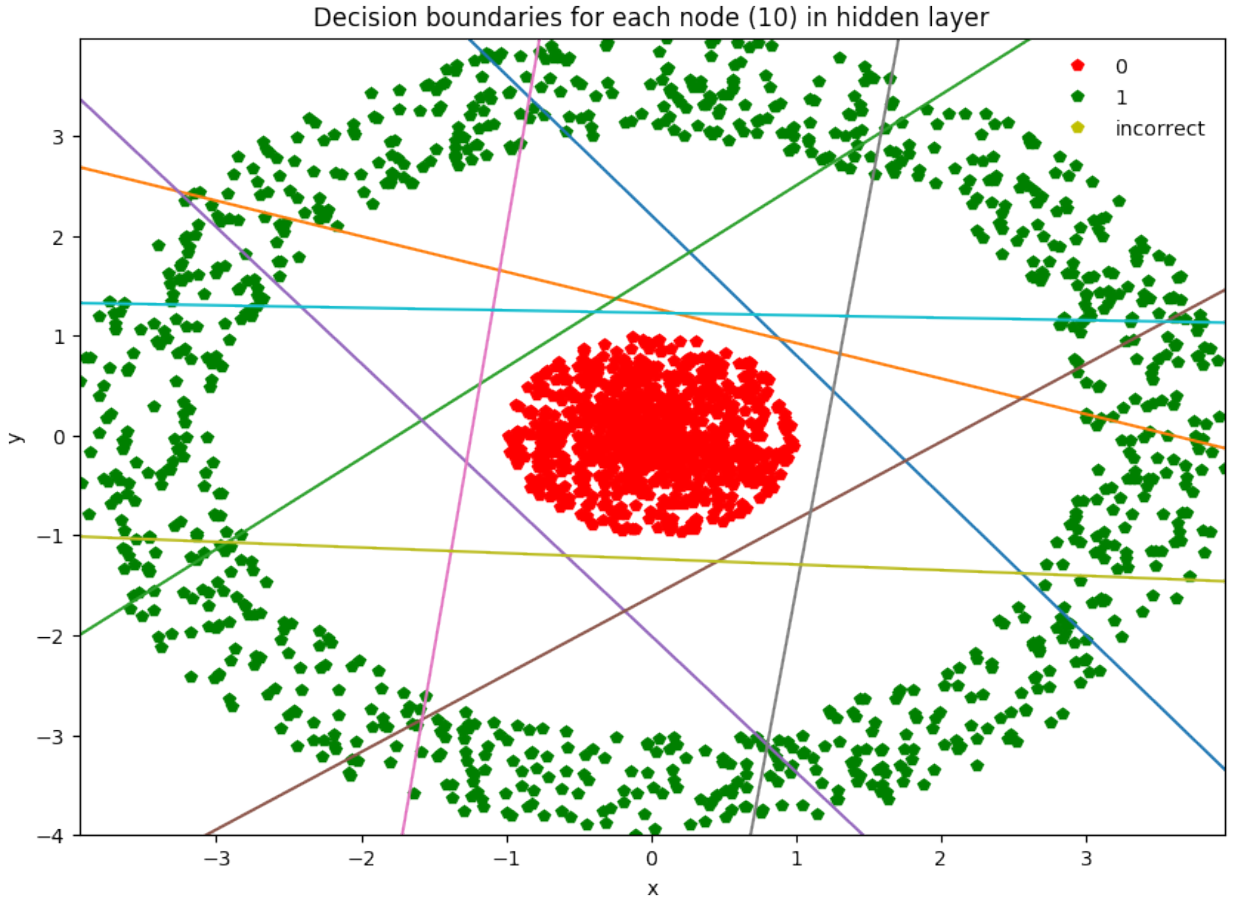
Binary Classification: Given (x, y), predict class 0 or 1

Hidden Nodes = 5



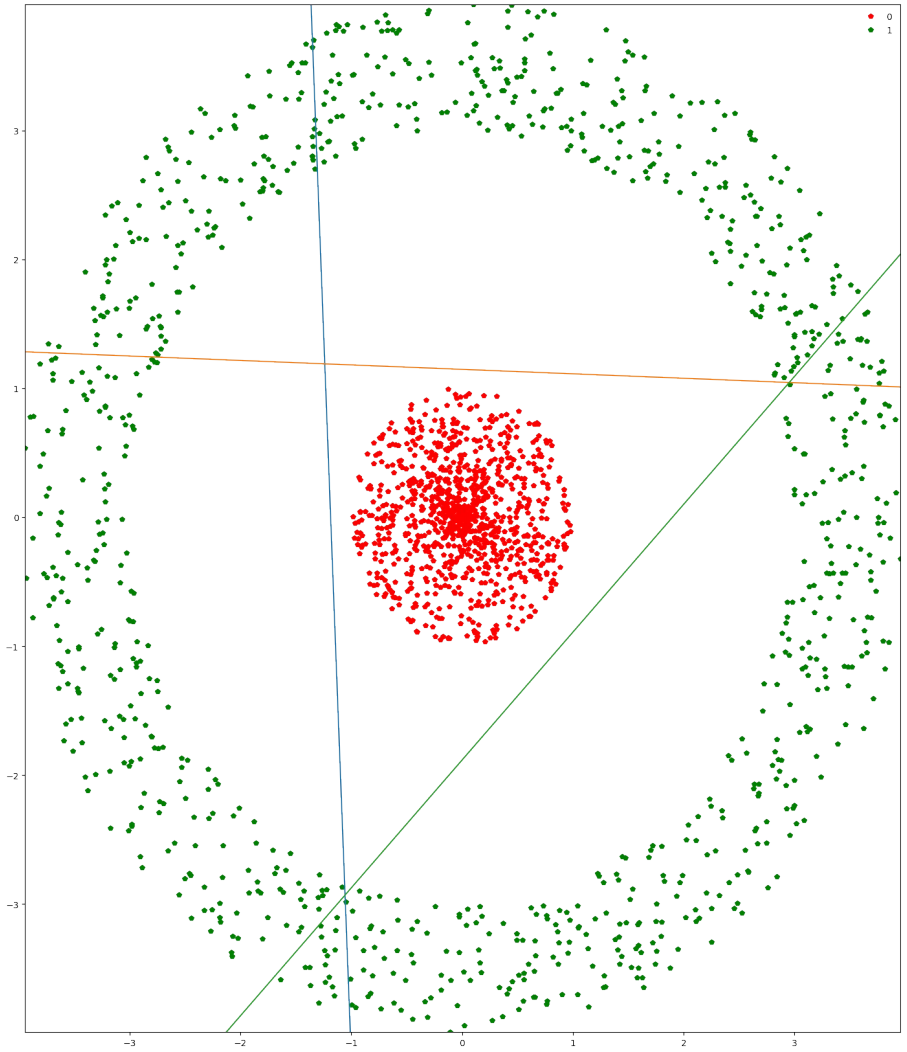
Red Hat

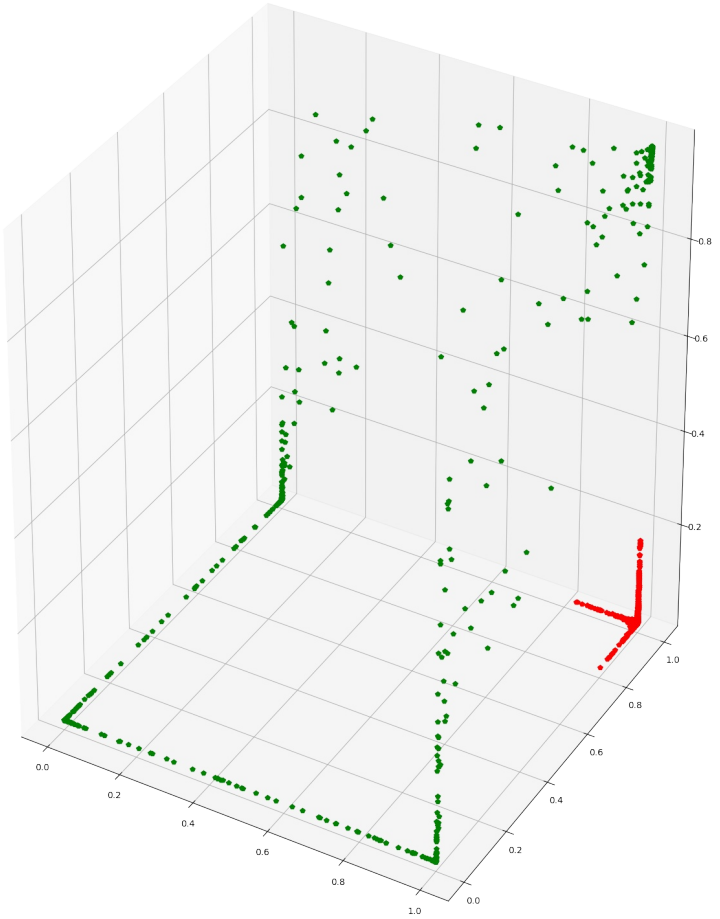
Hidden Nodes = 10



Red Hat

Map 2-dim input to 3-dim space





Red Hat

Many more options:

Combine multiple (weak) trees → Bagging, Random Forests

Sequentially train trees → Boosting

Don’t have to be trees → any set of models

Pick K-nearest neighbors and use their results

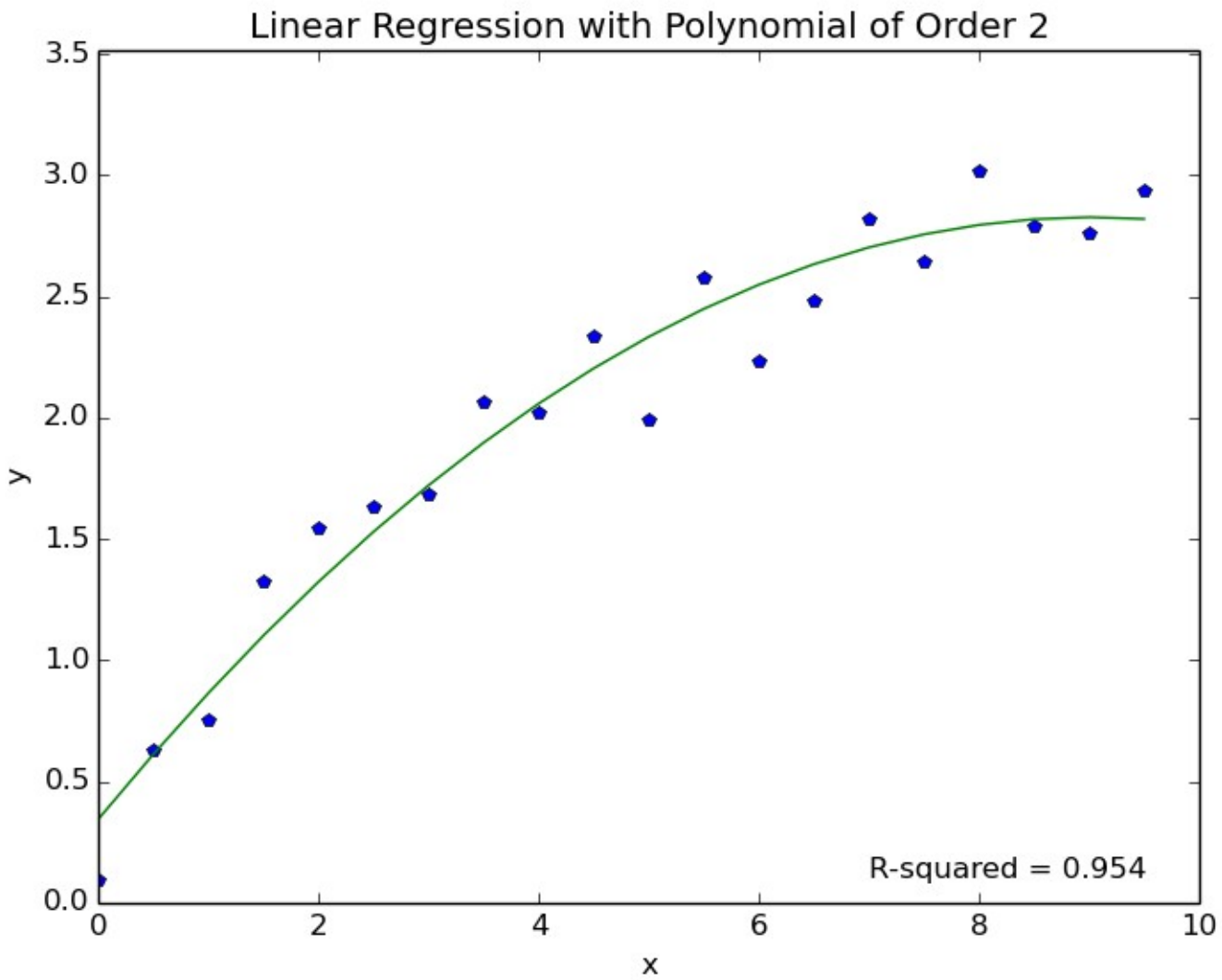
Things get trickier in higher dimensions!

Problem Statement

$$\text{Given features} : x_1, x_2, \ldots, x_n$$

$$\text{predict real number y}$$

Example



$$y = w_0 + w_1 x + w_2 x^2 = 0.344 + 0.551x - 0.031x^2$$

Can use all the techniques mentioned before with some modifications

All of this is called **supervised learning**

Supervised = have both inputs and outputs available

OR

Have **feedback** (output) for **every example** (input)

Supervised Learning:

Optimization: Turn learning problem into optimization problem

$$\min_{x} f(x) \rightarrow \min_{\text{model parameters}} \text{error}(\text{pred}, \text{output})$$

Generalization: should be able to predict accurately on unseen inputs

What if don’t have feedback for every input??

Maybe have only infrequent feedback

What if don’t have feedback for every input??

Maybe have only infrequent feedback

College student taking a course

College student taking a course

Goal: maximize learning by end of semester

College student taking a course

Goal: maximize learning by end of semester

Question: how should the student organize studying?

College student taking a course

Goal: maximize learning by end of semester

Question: how should the student organize studying?

Occasional feedback: quiz grades, exam grade etc.

Reinforcement Learning

$$\text{state} = s_t$$

$$\text{obs (partial state)} = o_t$$

Reinforcement Learning

$$\text{state} = s_t$$

$$\text{obs (partial state)} = o_t$$

ML Model

Policy

$$\pi_\theta(a_t|o_t)$$

probability distribution over actions given obs

Reinforcement Learning

$$\text{state} = s_t$$

$$\text{obs (partial state)} = o_t$$

ML Model

Policy

$$\pi_\theta(a_t|o_t)$$

probability distribution over actions given obs

$$a_t(\text{sample action})$$

Reinforcement Learning

ML Model

Policy

$\pi_\theta(a_t|o_t)$

probability distribution over actions given obs

a_t(sample action)

$\text{state} = s_t$

$\text{obs (partial state)} = o_t$

Dynamics

$\rho(s_{t+1}|s_t, a_t)$

$+$

$\text{Reward} = r_t(s_t, a_t)$

Reinforcement Learning

ML Model

Policy

$\pi_\theta(a_t|o_t)$

probability distribution over actions given obs

a_t(sample action)

$\text{state} = s_t$

$\text{obs (partial state)} = o_t$

Dynamics

$\rho(s_{t+1}|s_t, a_t)$

$+$

$\text{Reward} = r_t(s_t, a_t)$

Maximize: $R = \Sigma_{t=1}^{T} r_t$

Reinforcement Learning

Dynamics are usually not known

Dynamics, ρ and r_t stochastic

$R = \Sigma_{t=1}^{T} r_t$ highly path dependent

More precisely, maximize: $\mathbb{E}[R|s_0]$

Supervised Learning:

Optimization: Turn learning problem into optimization problem

$$\min_{x} f(x) \rightarrow \min_{\text{model parameters}} \text{error}(\text{pred}, \text{output})$$

Generalization: should be able to predict accurately on unseen inputs

Planning:

Optimization: Turn learning problem into optimization problem

$$\min_{x} f(x) \rightarrow \min_{\text{model parameters}} \text{error}(\text{pred}, \text{output})$$

Generalization: should be able to predict accurately on unseen inputs

Delayed Feedback: don’t know if sequence of moves good/bad till experiment ends

Reinforcement Learning:

Optimization: Turn learning problem into optimization problem

$$\min_{x} f(x) \rightarrow \min_{\text{model parameters}} \text{error}(\text{pred}, \text{output})$$

Generalization: should be able to predict accurately on unseen inputs

Delayed Feedback: don’t know if sequence of moves good/bad till experiment ends

Exploration: Rules (dynamics) of system **unknown**. Have to discover them by trial-and-error.

Supervised = have **feedback** (output) for **every example** (input)

Reinforcement Learning = have **infrequent** feedback

What if no feedback?!

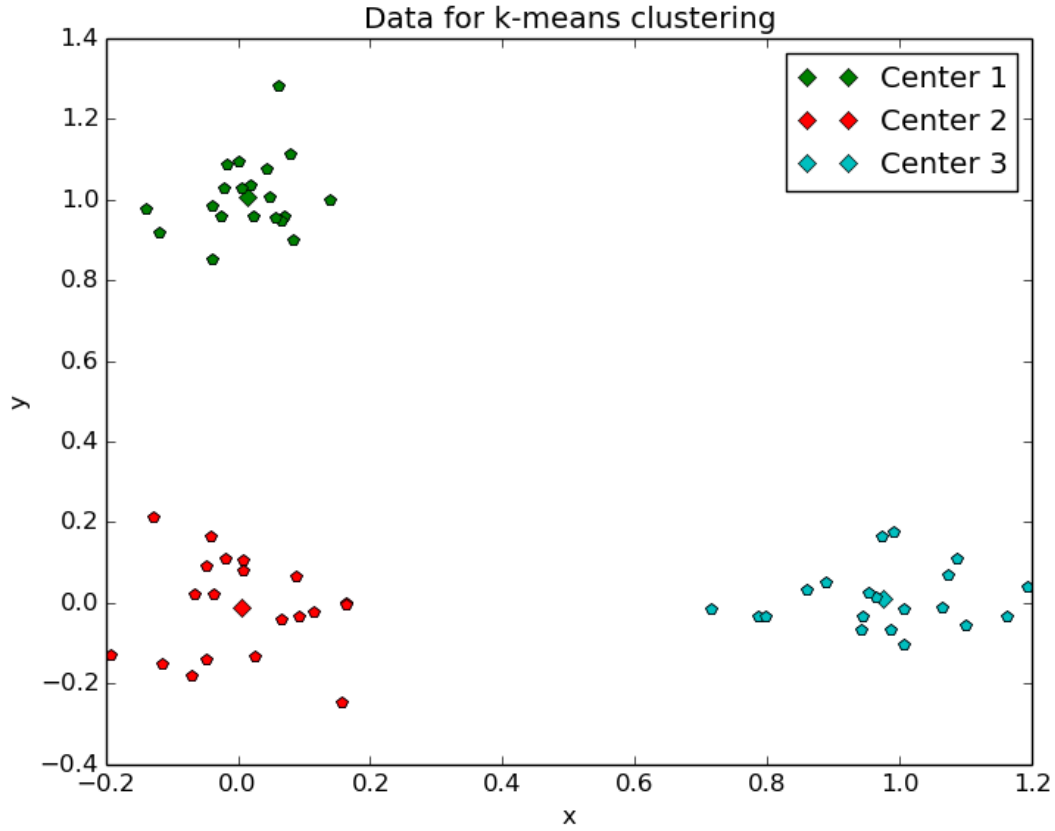
Supervised = have **feedback** (output) for **every example** (input)

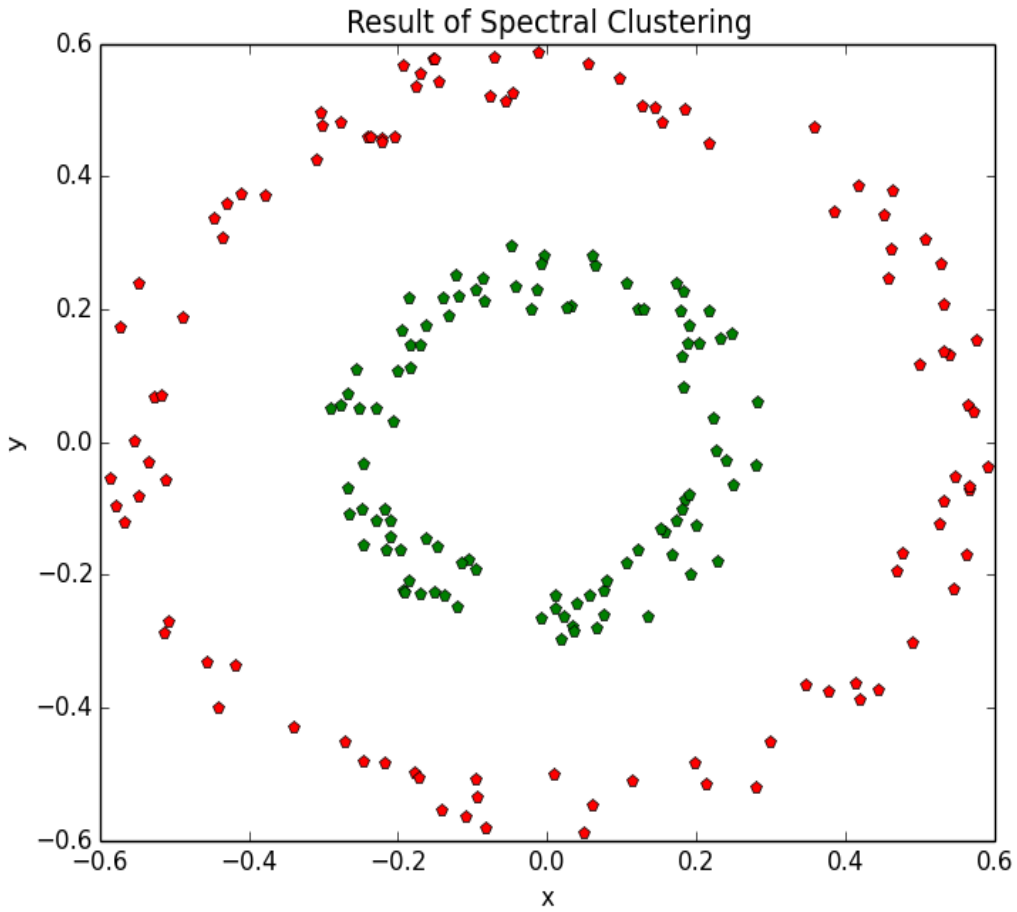
Reinforcement Learning = have **infrequent** feedback

What if no feedback?!

Very common! = just have some data

Unsupervised Learning

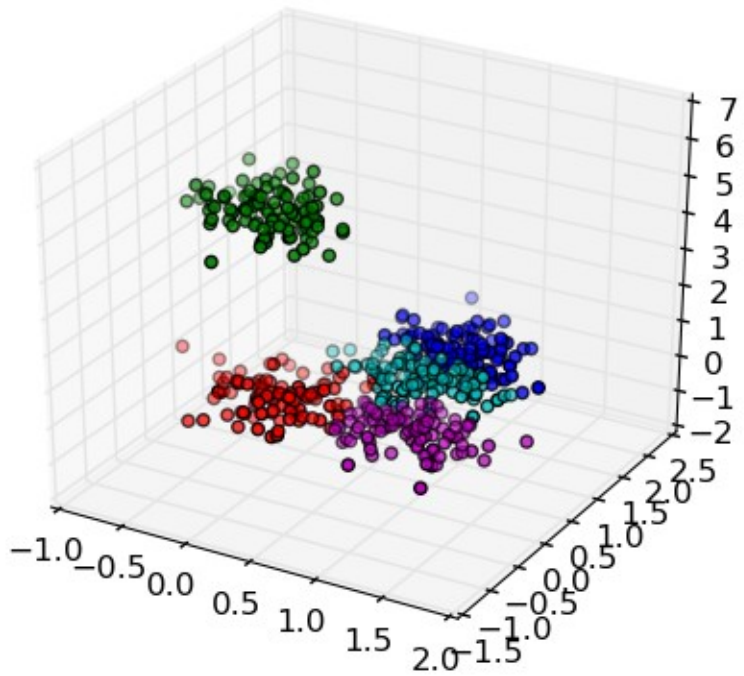


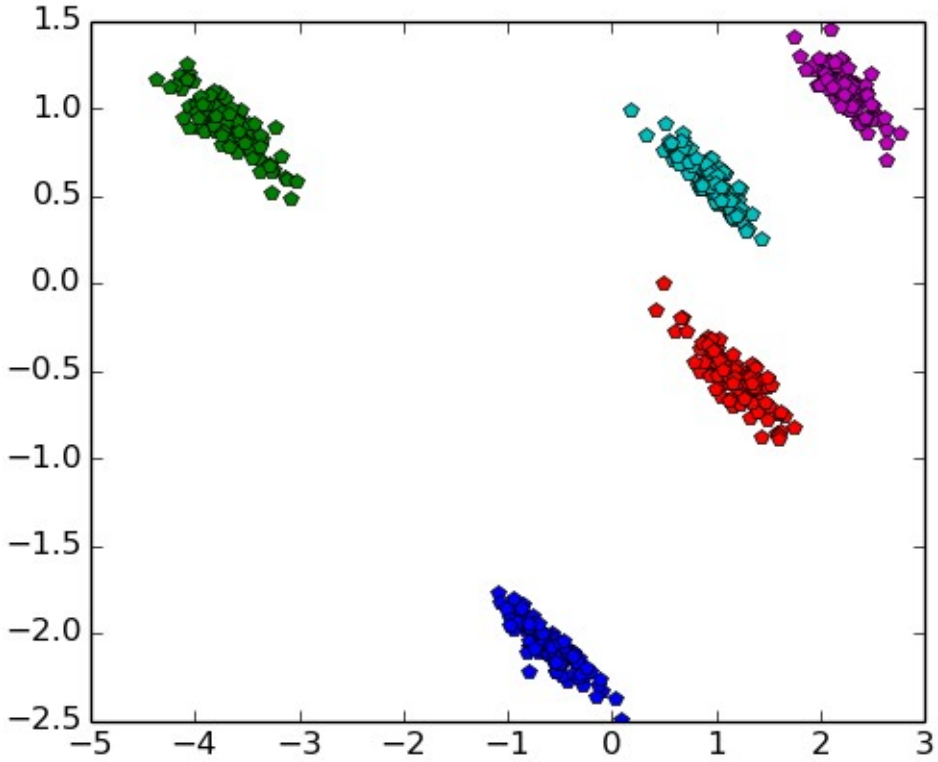


Clustering → assign point/row to one of K groups

Red Hat

Unsupervised Learning

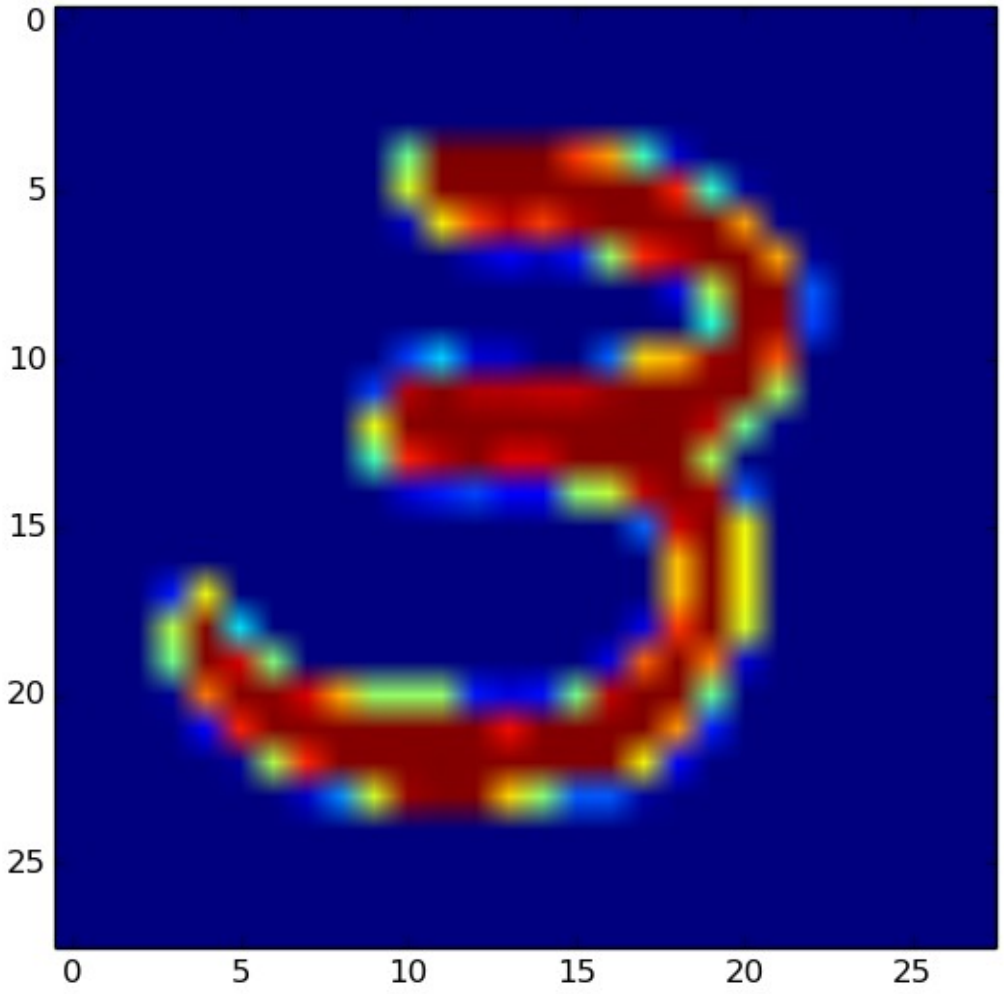




Dimensionality Reduction

Red Hat

Unsupervised Learning

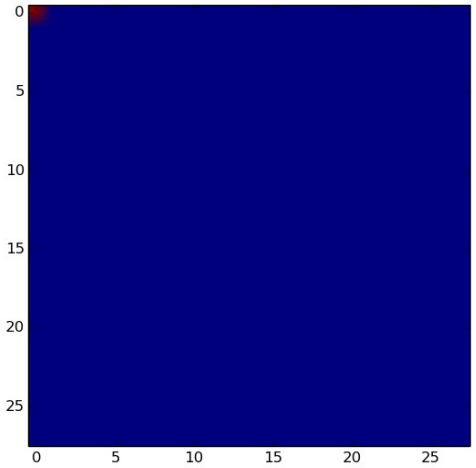


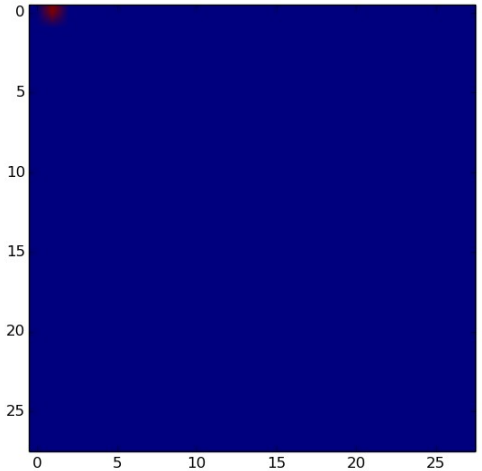
Dimensionality Reduction

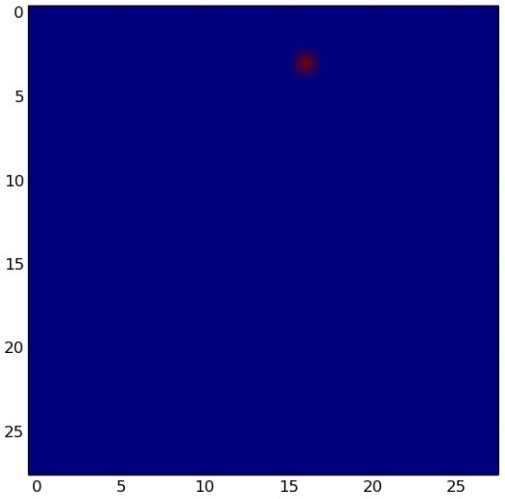
Red Hat

Unsupervised Learning

Old Basis:

Each image = a_1 

$+a_2$ 

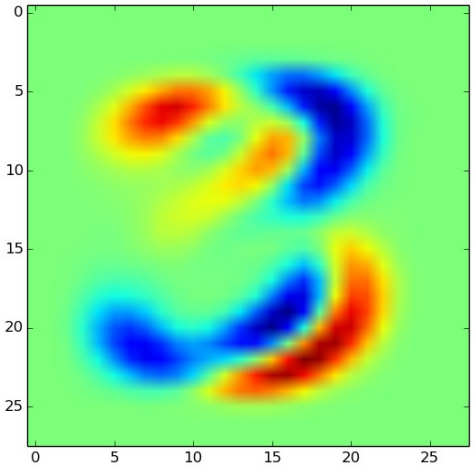
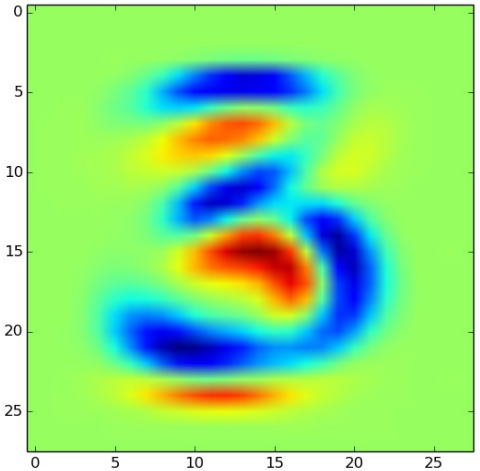
$+\ldots+a_{100}$ 

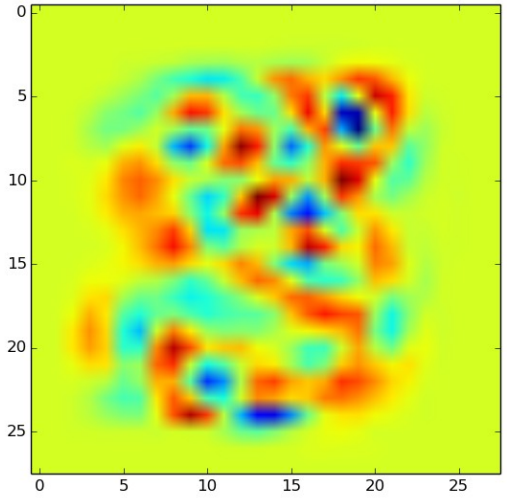
$+\ldots$

Red Hat

Unsupervised Learning

New Basis:

Each image = b_1  $+b_2$ 

$+ \ldots + b_{100}$  $+ \ldots$

Unsupervised Learning

Many more

In some ways, this is the dominant problem

Generally:

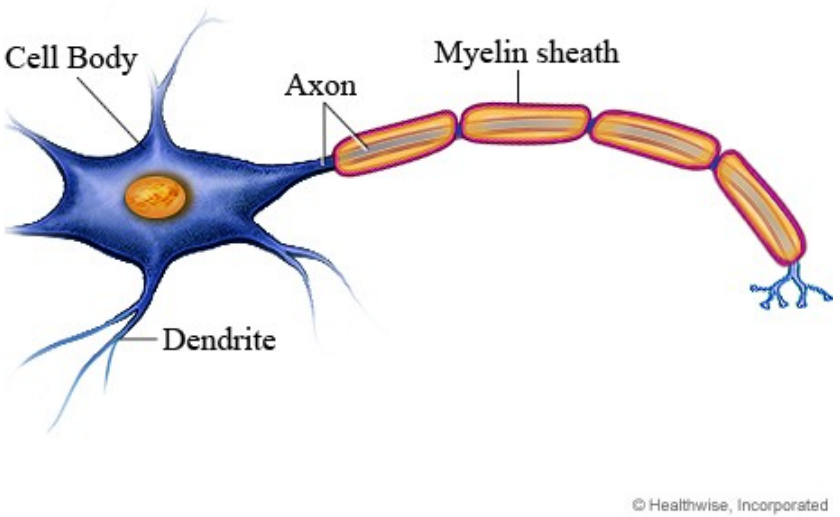
Supervised = straightforward if problem defined. Good implementations available

Unsupervised = Good implementations available often. Problem definition is much harder

Reinforcement Learning = models can be unstable and very sensitive to details (rewards, dynamics, cardinality of state and action spaces ...)

Neural Networks

What is a neural network?



Loosely inspired by neurons in the brain

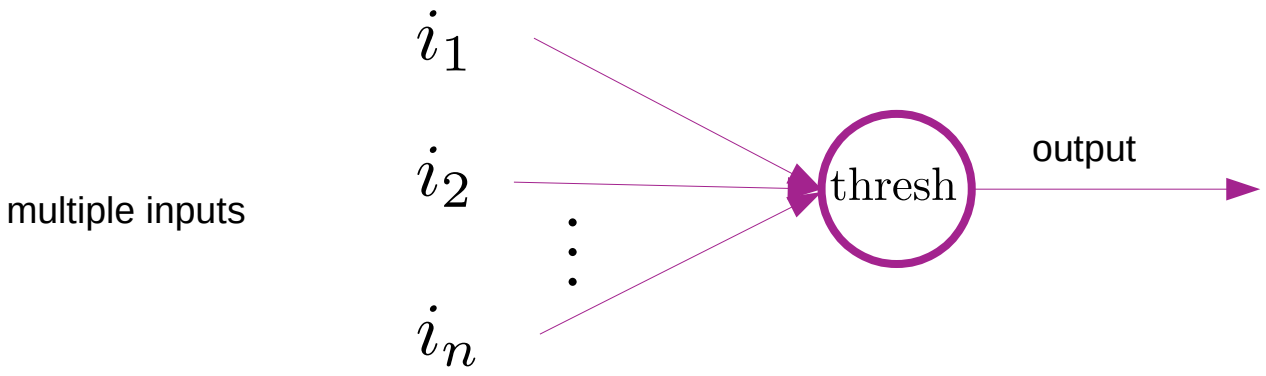
Red Hat

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?

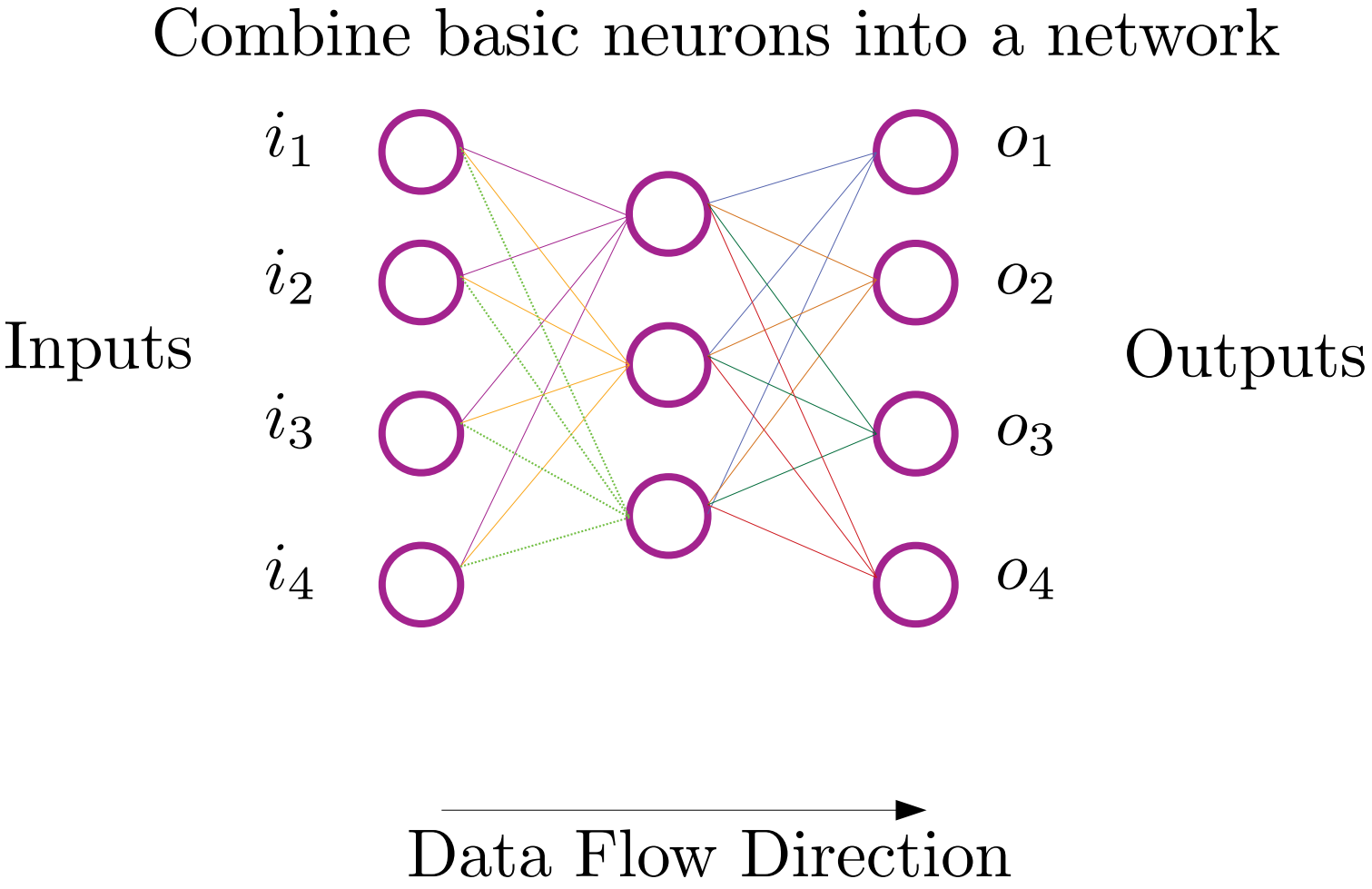


$$\text{input}, i = \underbrace{w_1}_{\text{weight}} i_1 + \underbrace{w_2}_{\text{weight}} i_2 + \ldots + \underbrace{w_n}_{\text{weight}} i_n$$

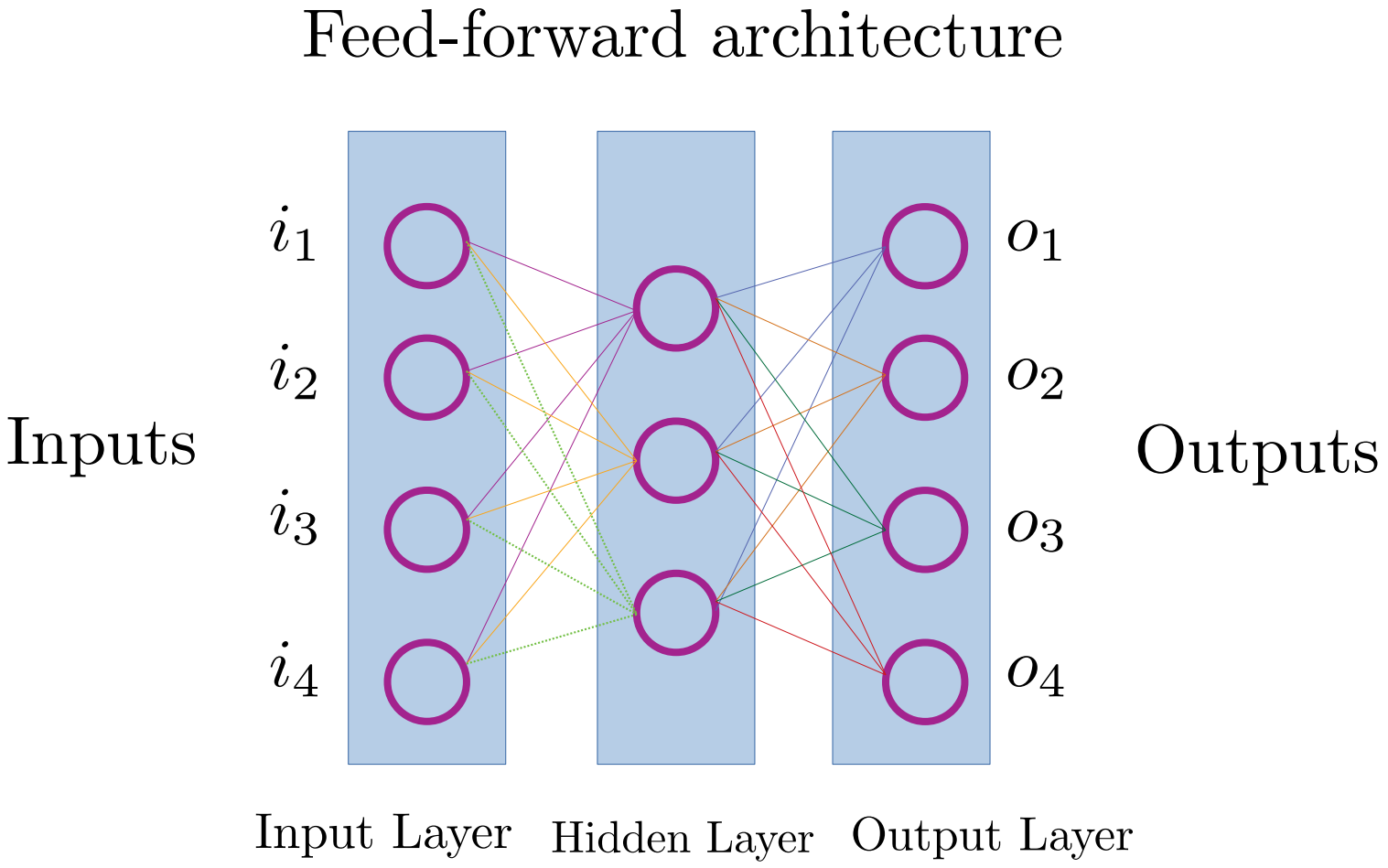
$$\text{input} > \text{threshold} \rightarrow \text{output} = 1$$

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

What is a neural network?



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

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

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

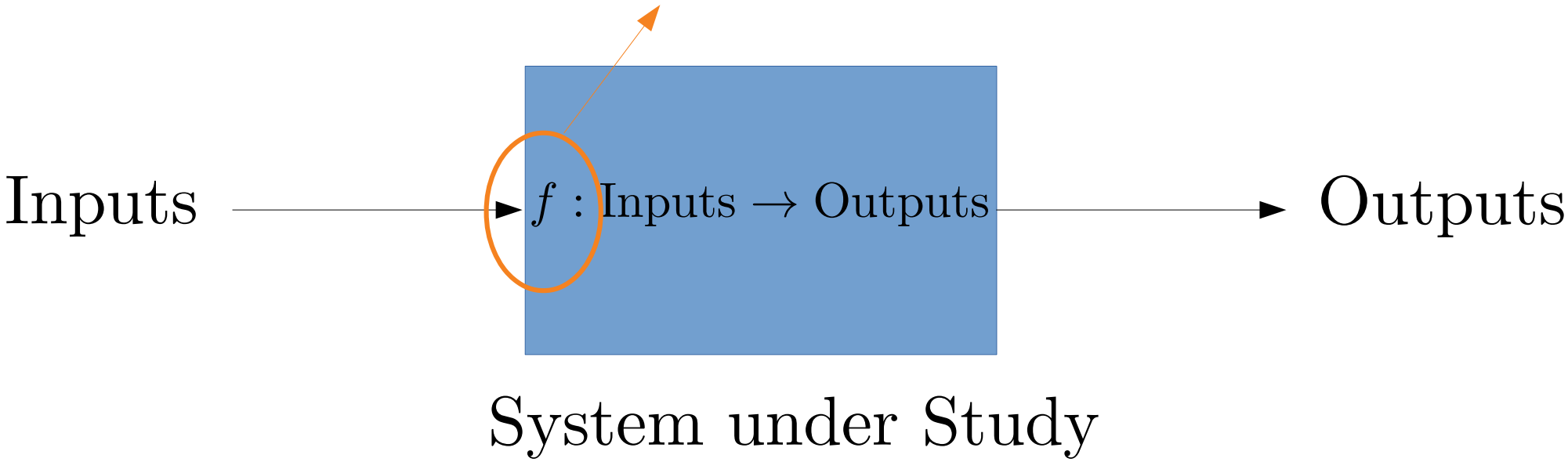
$$| \underbrace{f(x)}_{\text{function to learn}} - \underbrace{\Sigma_{i=1}^{N} w_i \sigma(u_i^T x + b_i)}_{\text{approximation to } f} | < \epsilon$$

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

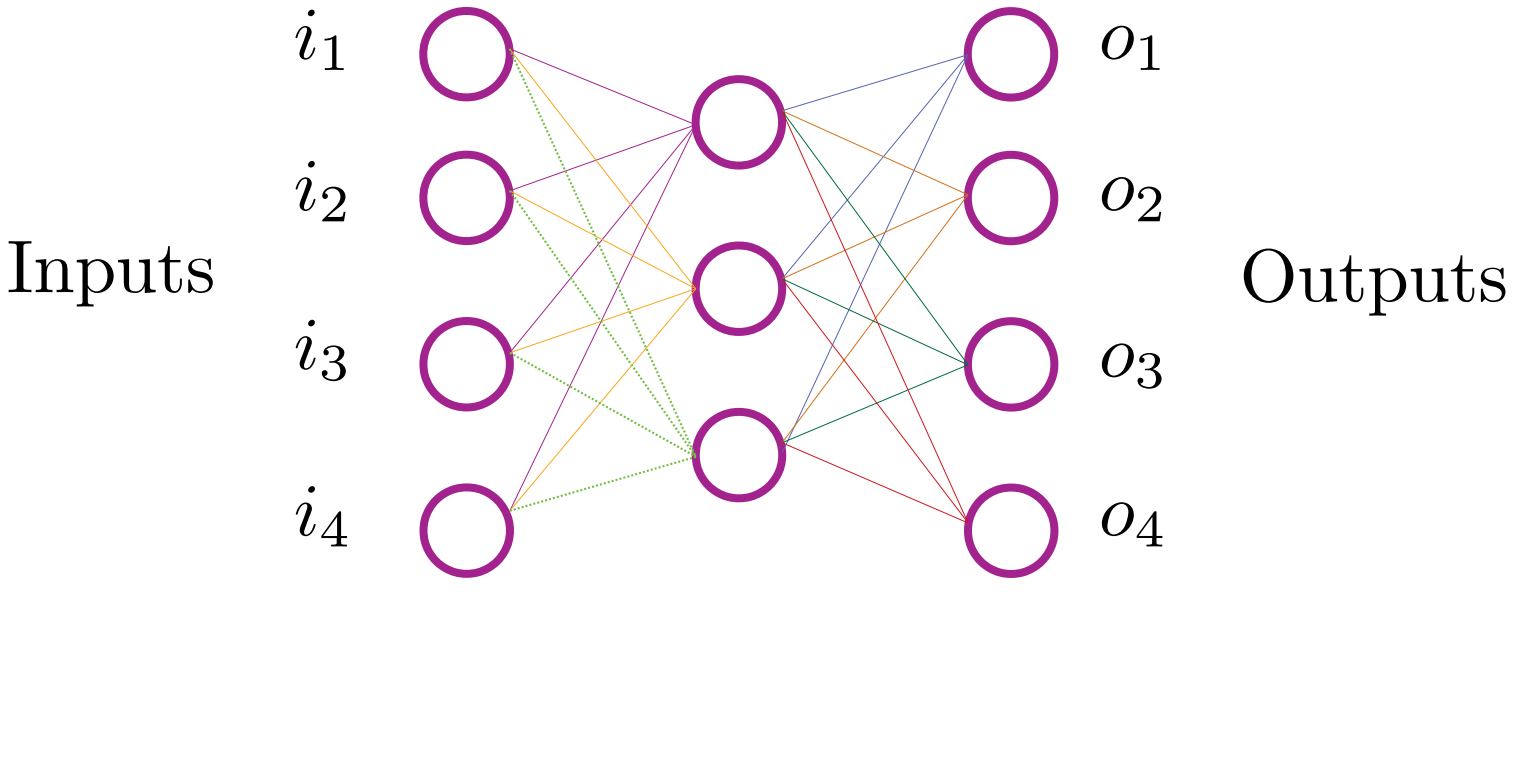
Need non-linear activation function, ϕ in our neural network (we'll see what this means shortly)

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

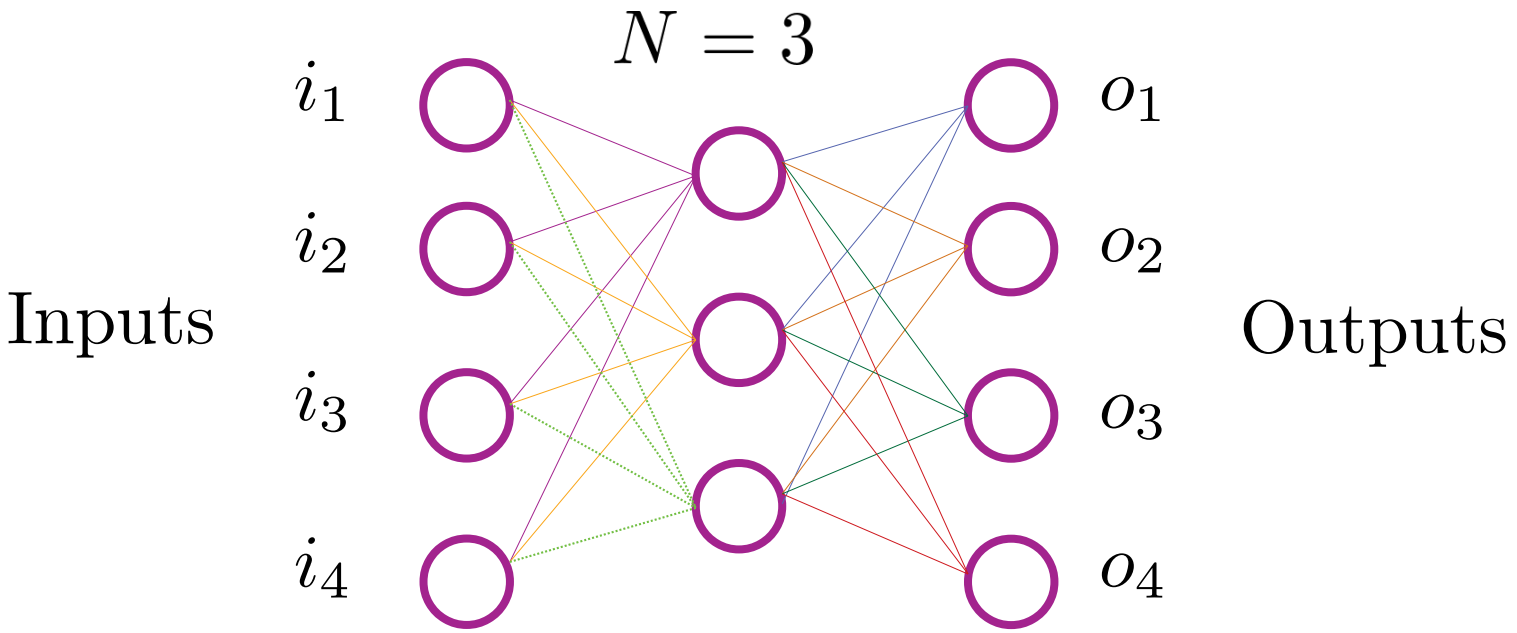
Need non-linear activation function, σ in our neural network
(we'll see what this means shortly)



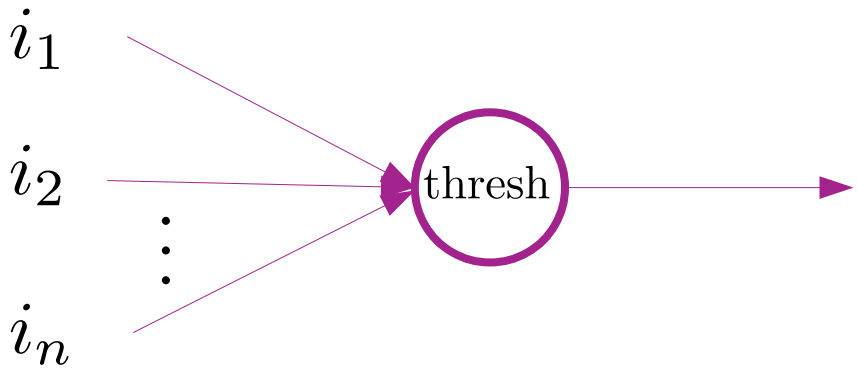
Two Caveats

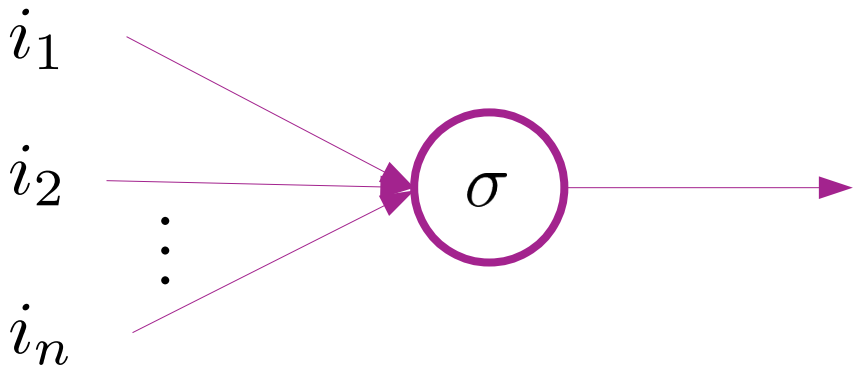
N refers to the number of neurons (nodes) in the hidden layer. This grows exponentially as ϵ decreases

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



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

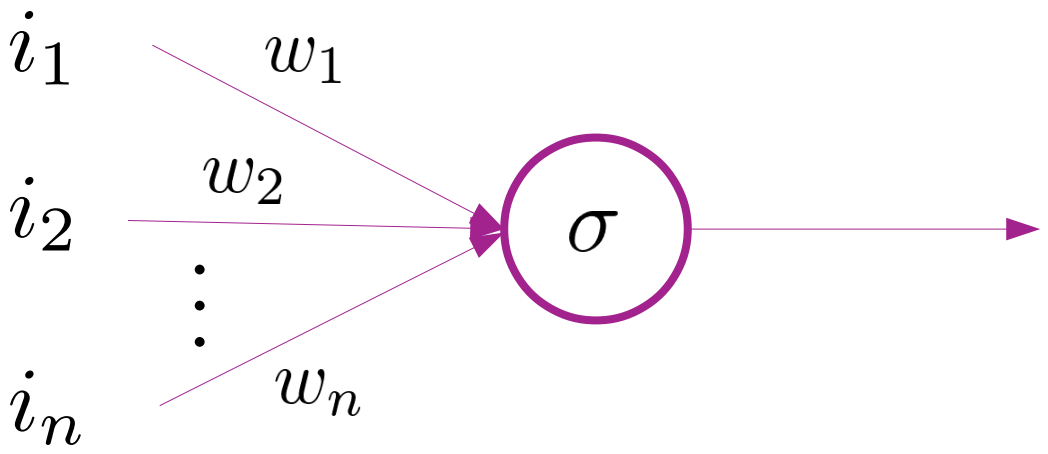
Non-linear

$\log(x)$ 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

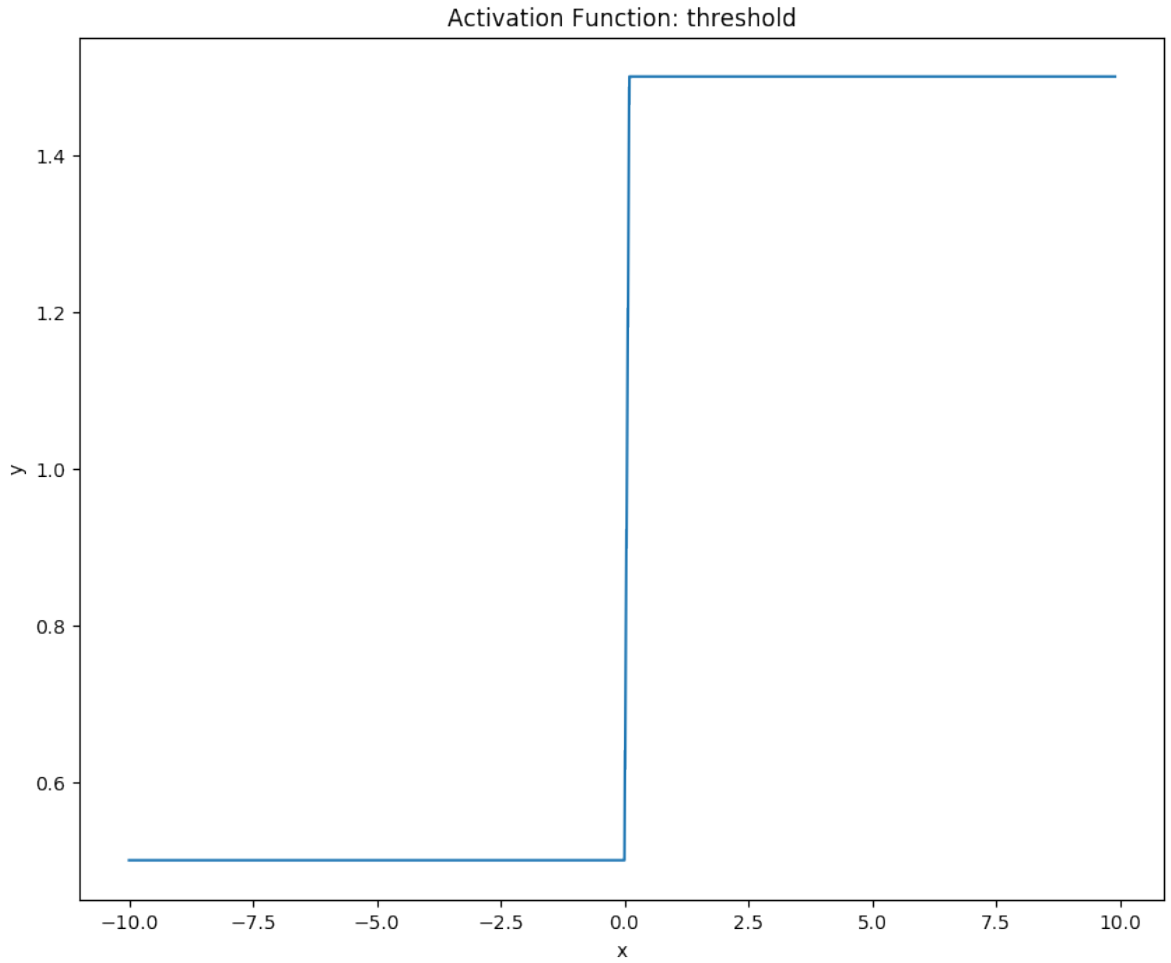
Will need derivatives so hopefully derivatives easy to evaluate

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

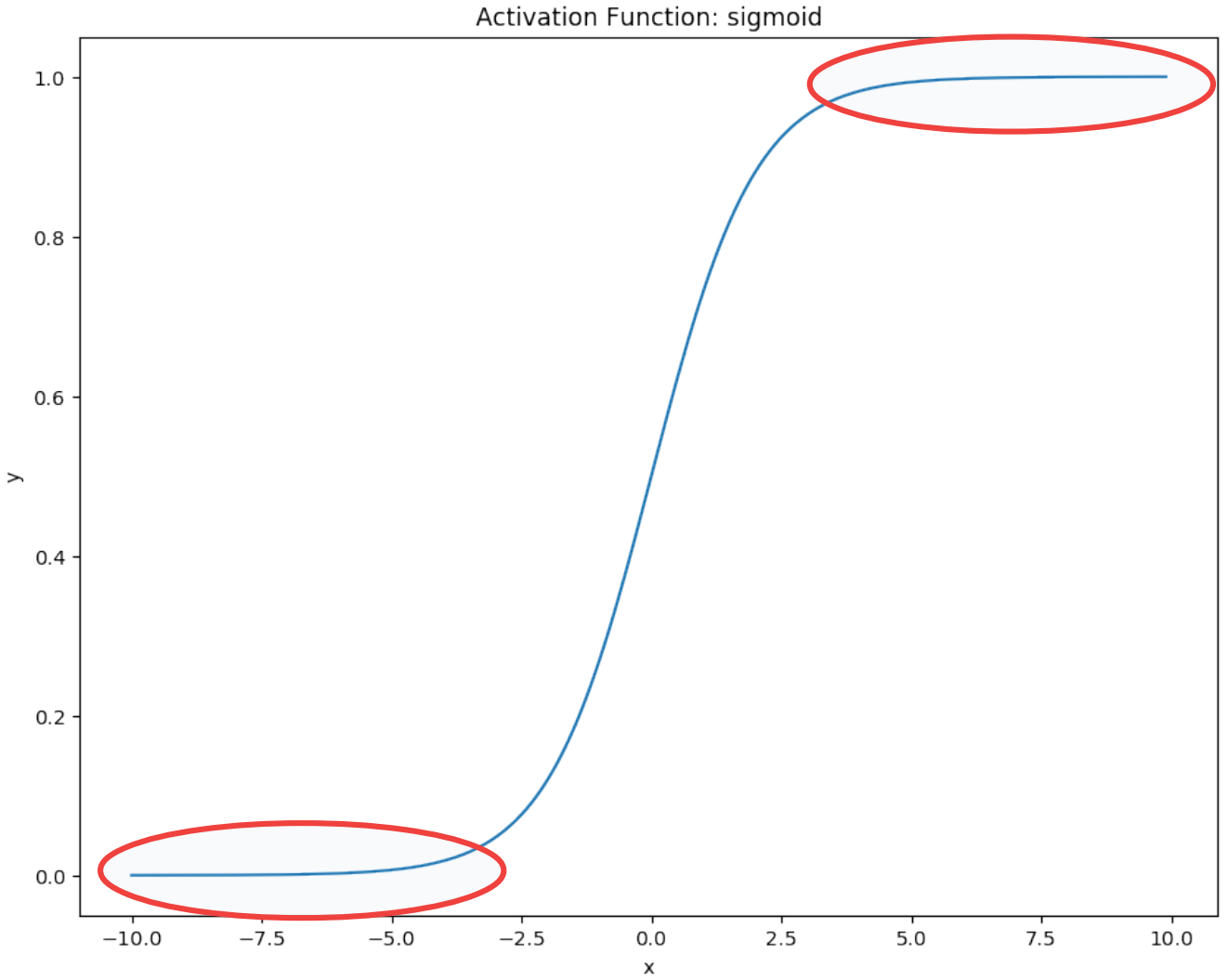
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Continuous switch

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

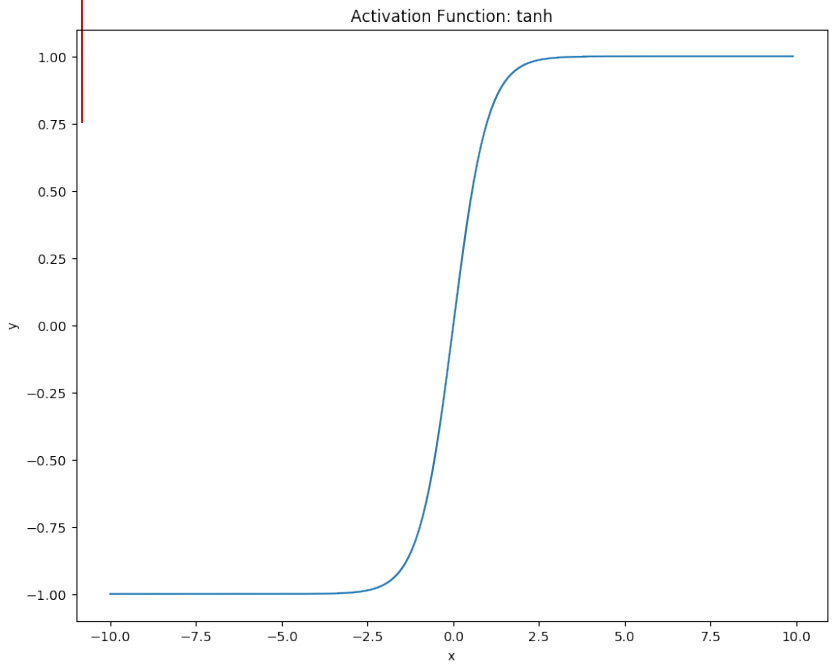
Derivatives in terms of activation itself

Positive x → 1 (on)

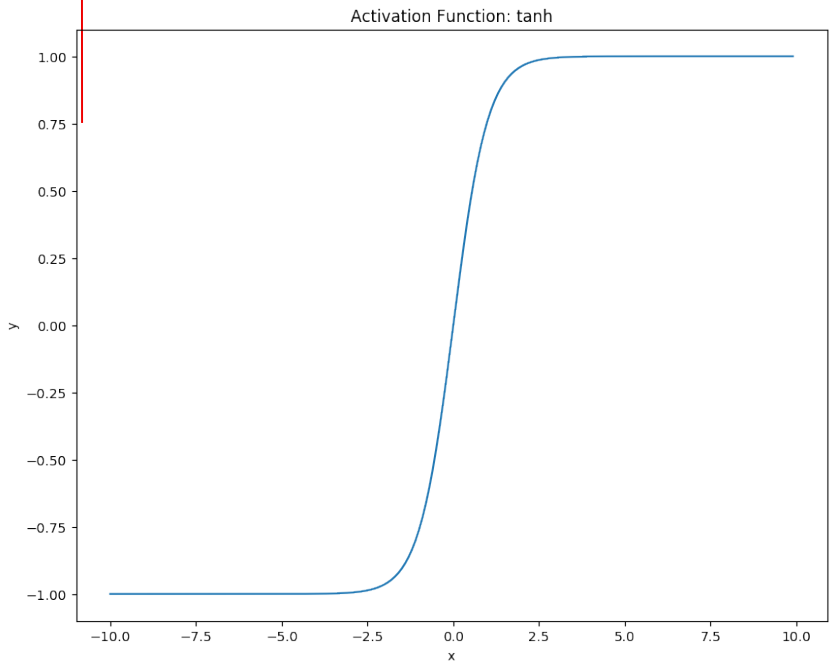


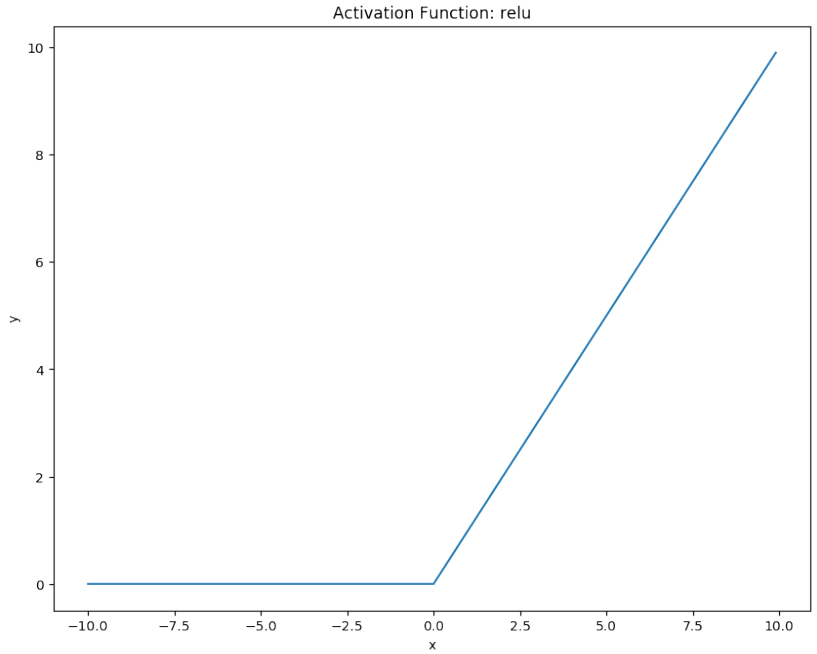
Negative x → 0 (off)

Red Hat

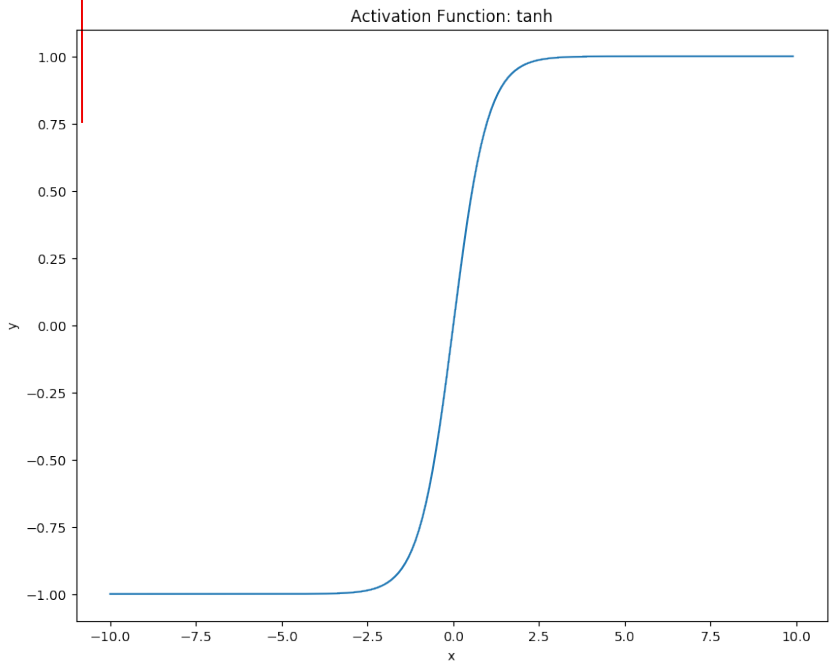

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

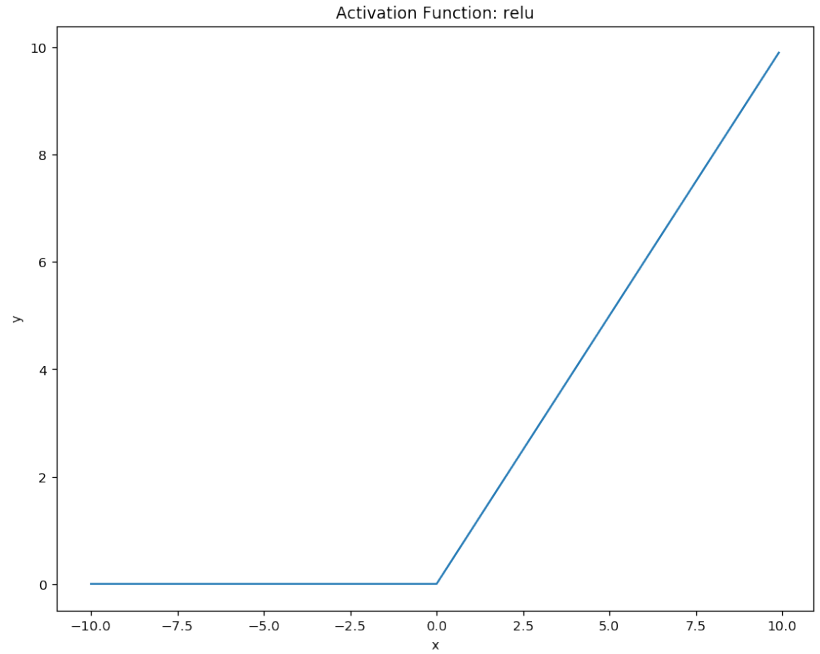
Red Hat

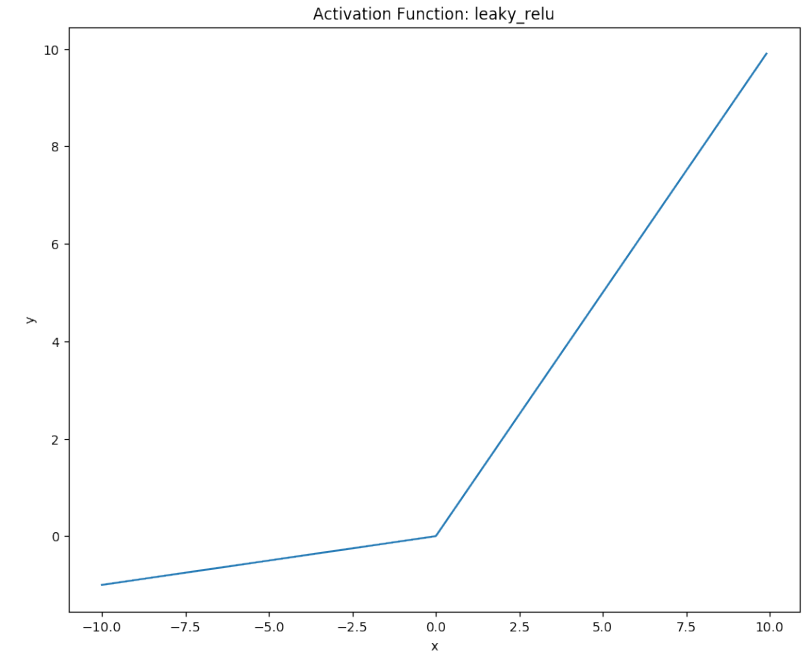

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

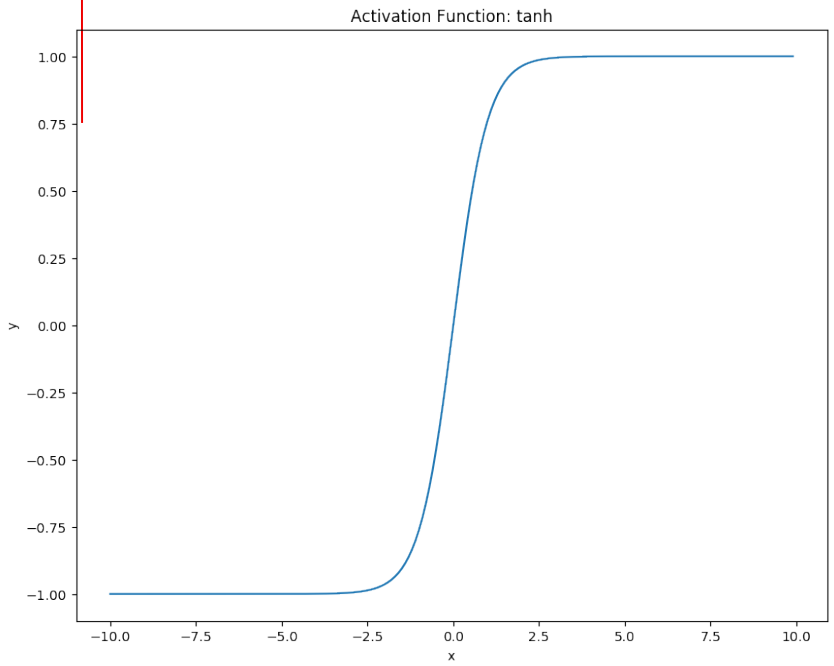

Activation Function: relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

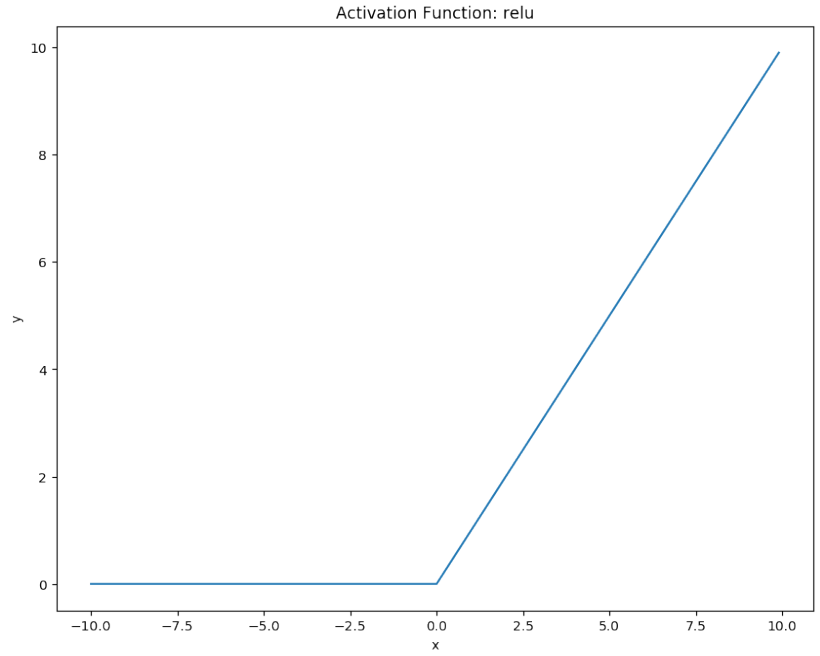
Red Hat

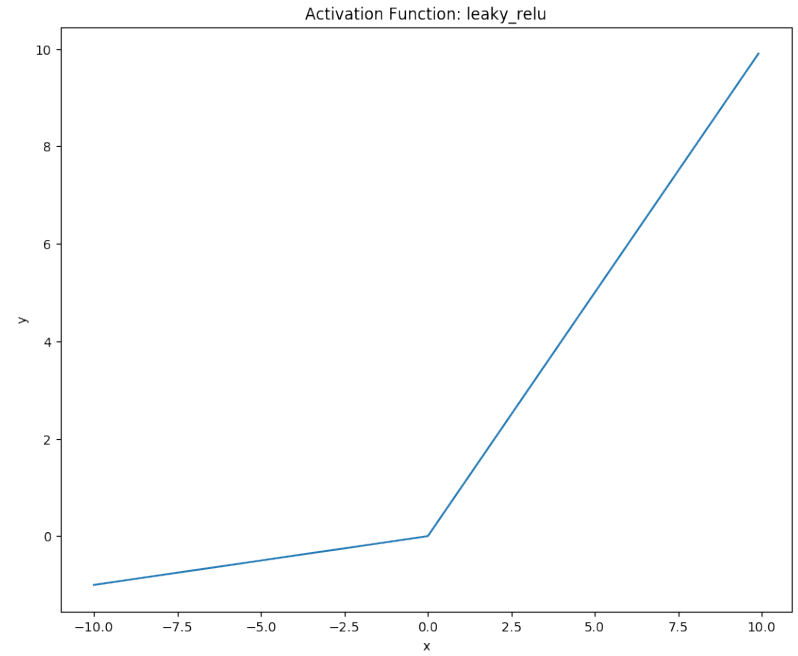

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

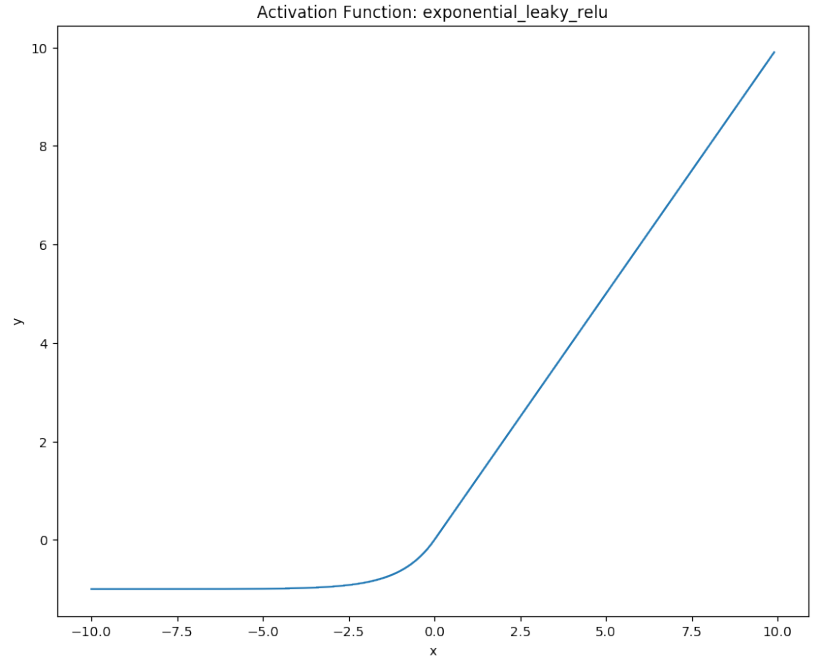

Activation Function: relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x


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

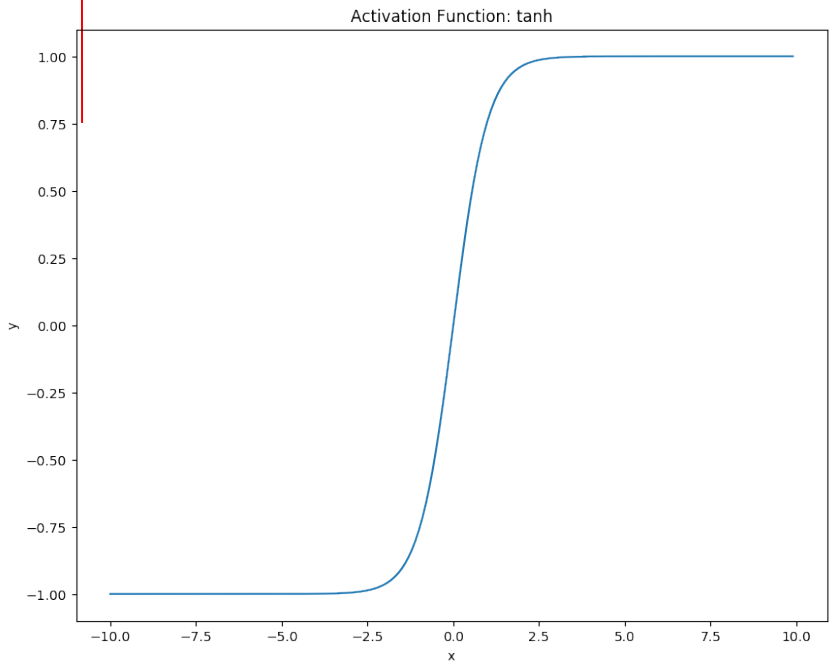

Activation Function: tanh
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

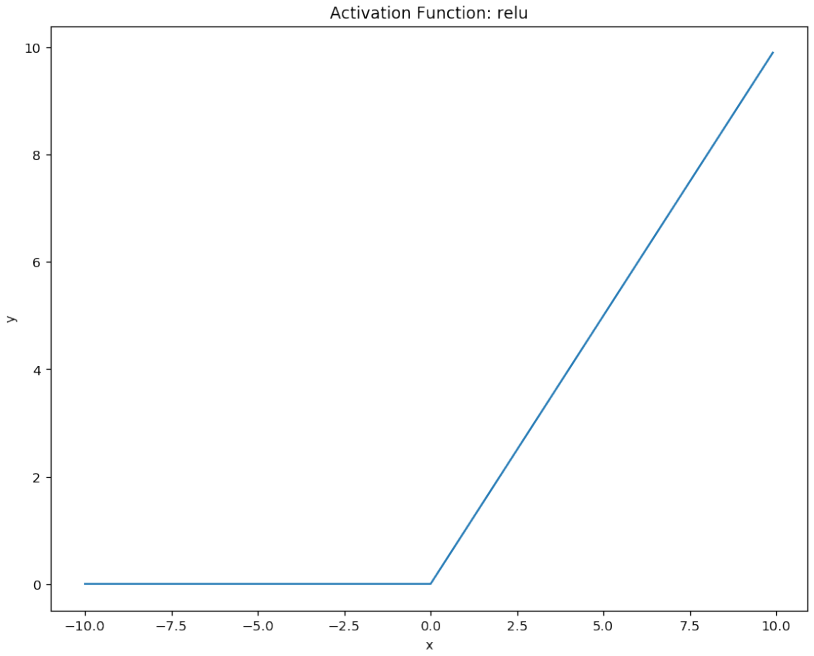

Activation Function: relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

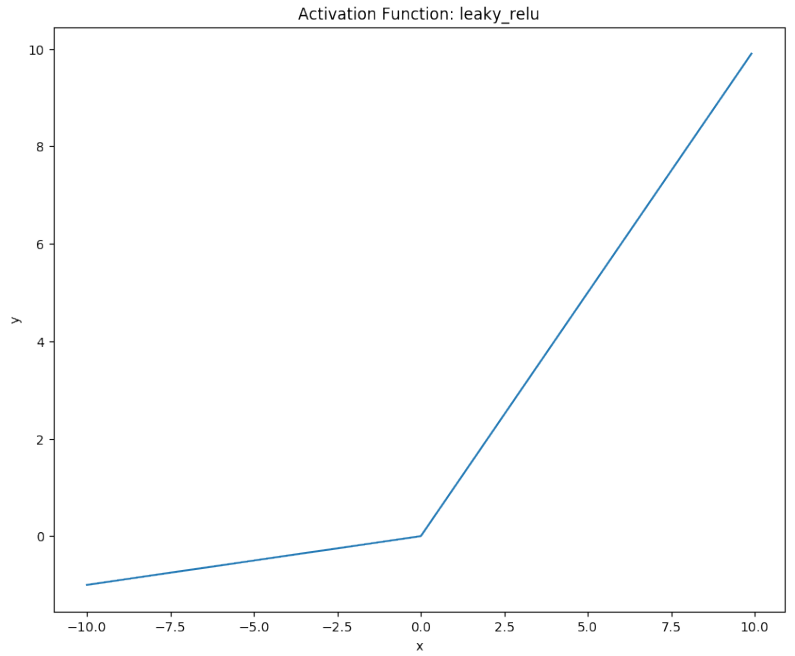

Activation Function: leaky_relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

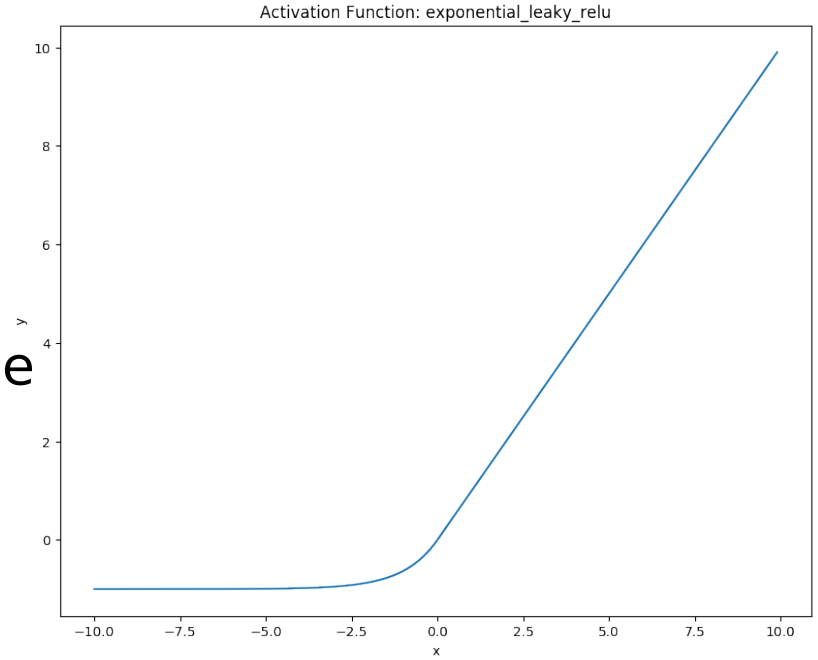

Activation Function: exponential_leaky_relu
10
8
6
4
2
0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

Red Hat









Many more choices

Choices dictated by:

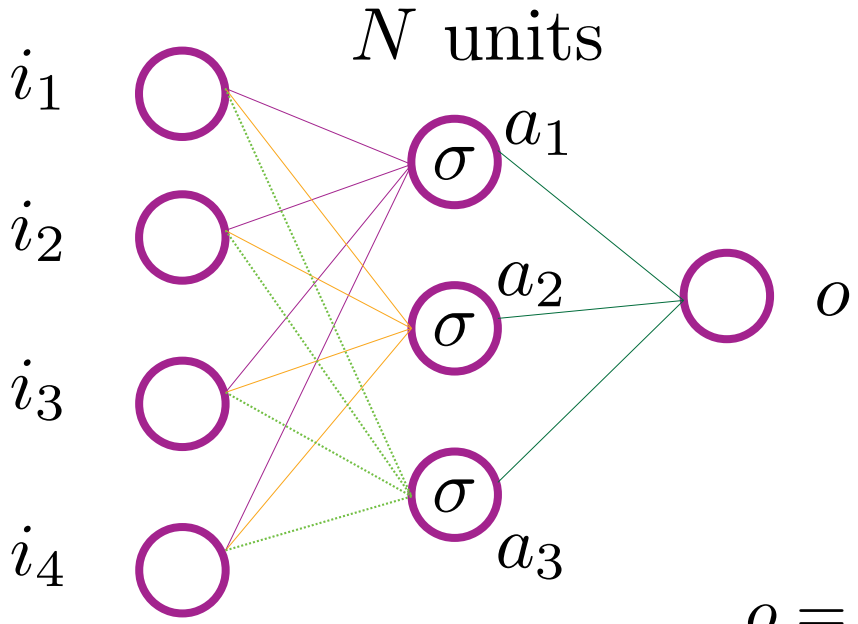
- Issues during training time (saturation of nodes)
- Type of predictions

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

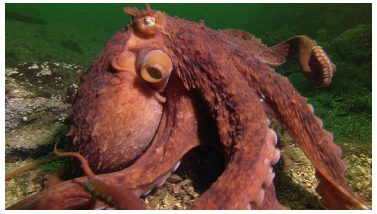
Start with labeled data

elephant

dog

snow leopard

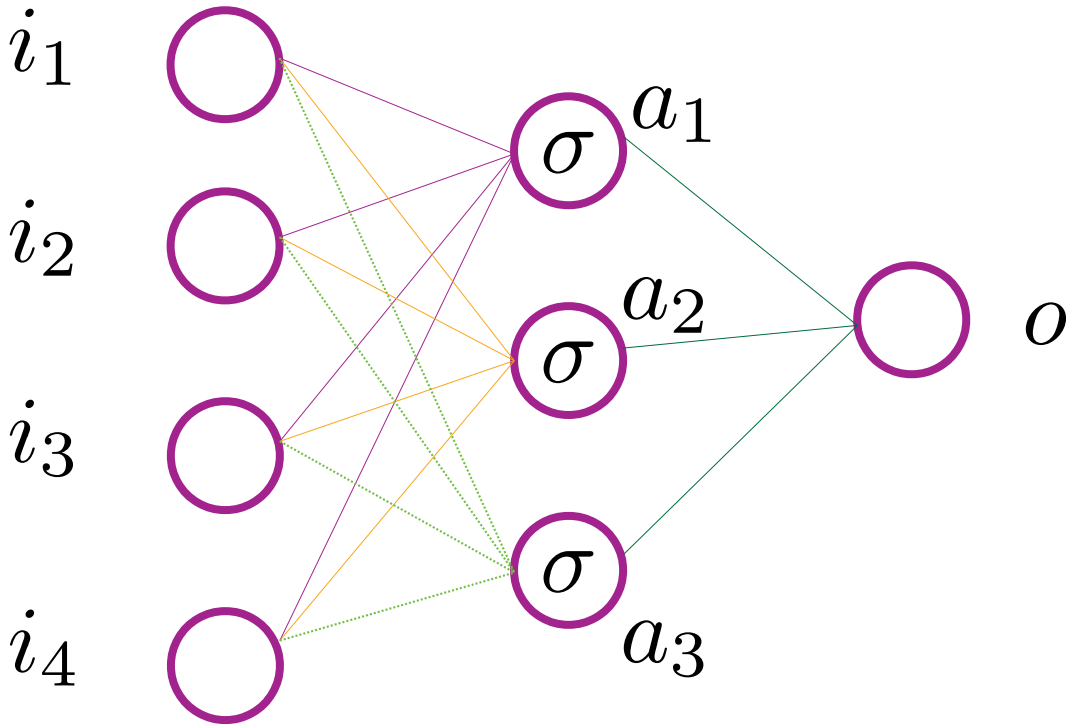
penguin



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)

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

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:

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

$$\underbrace{\qquad\qquad\qquad\qquad}_{\text{Cost for example i}}$$

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

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

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

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

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

Loss Functions: Custom

You can design one based on your problem

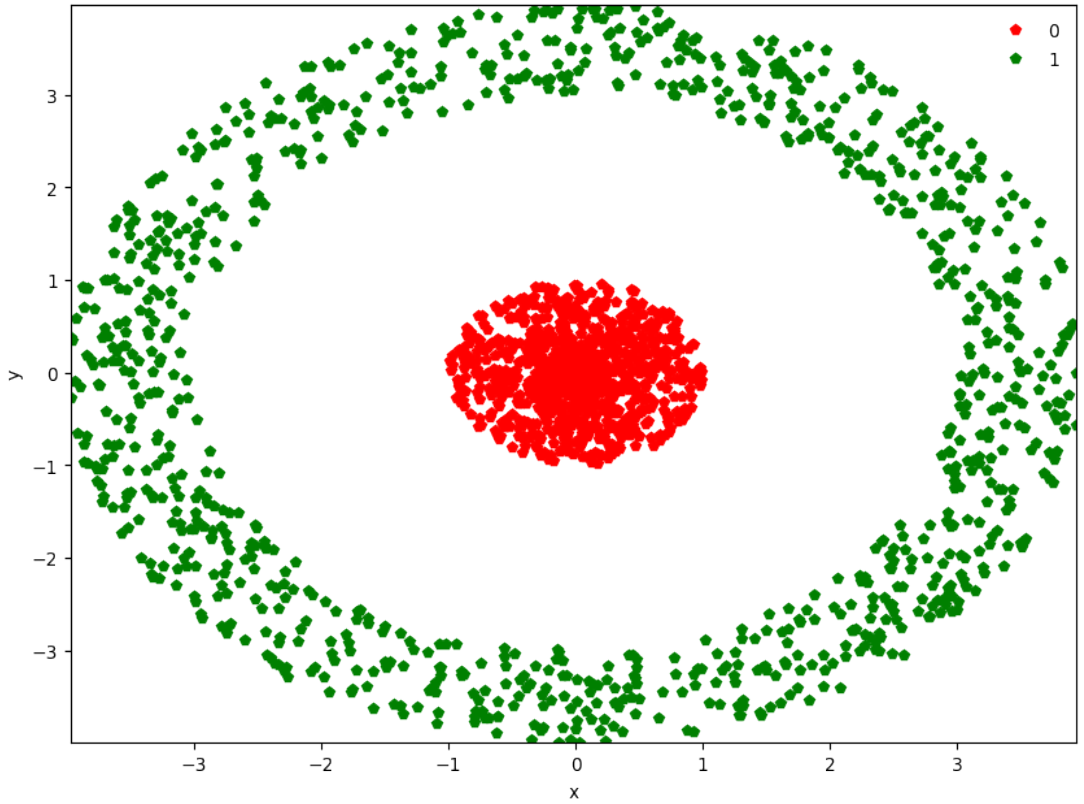
Well-designed loss greatly aids learning/training

Example: Kullback-Leibler divergence for comparing two distributions

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

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

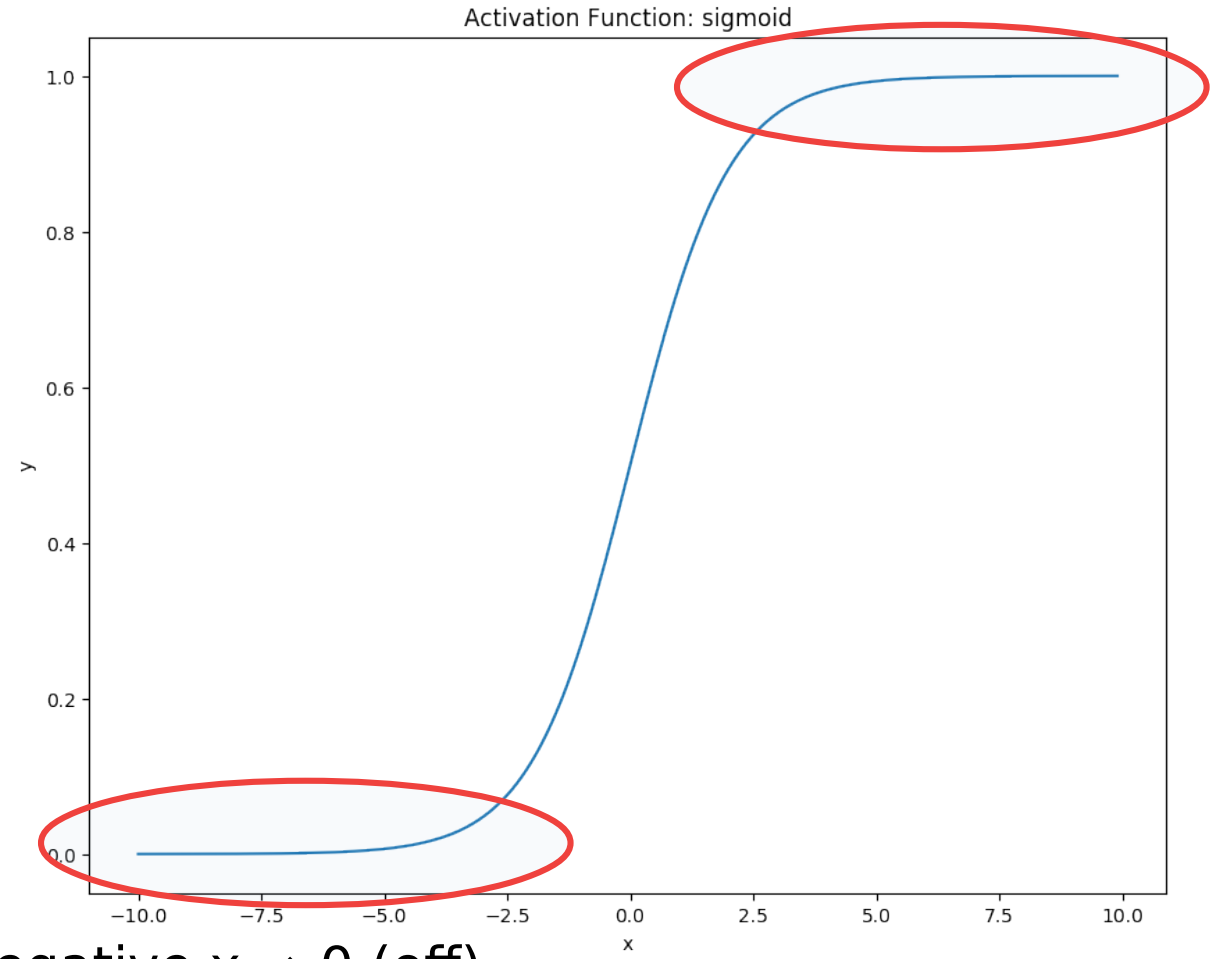
a, b, c found by minimizing cross-entropy loss

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

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)

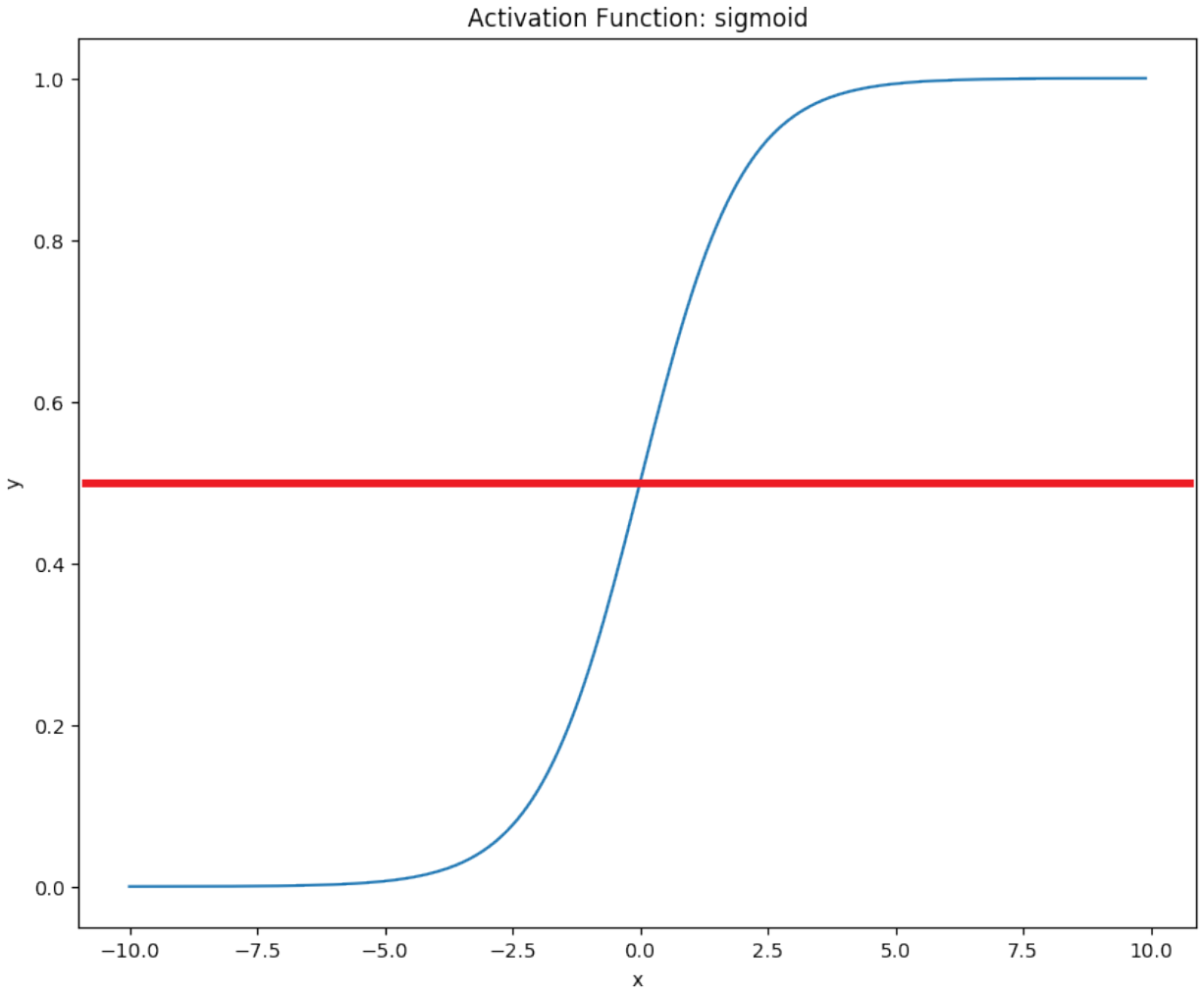
$$\sigma(ax + by + c) = \frac{1}{1 + e^{-(ax+by+c)}}$$

Red Hat

Class 1 (100%)

Probability

Class 0 (0%)

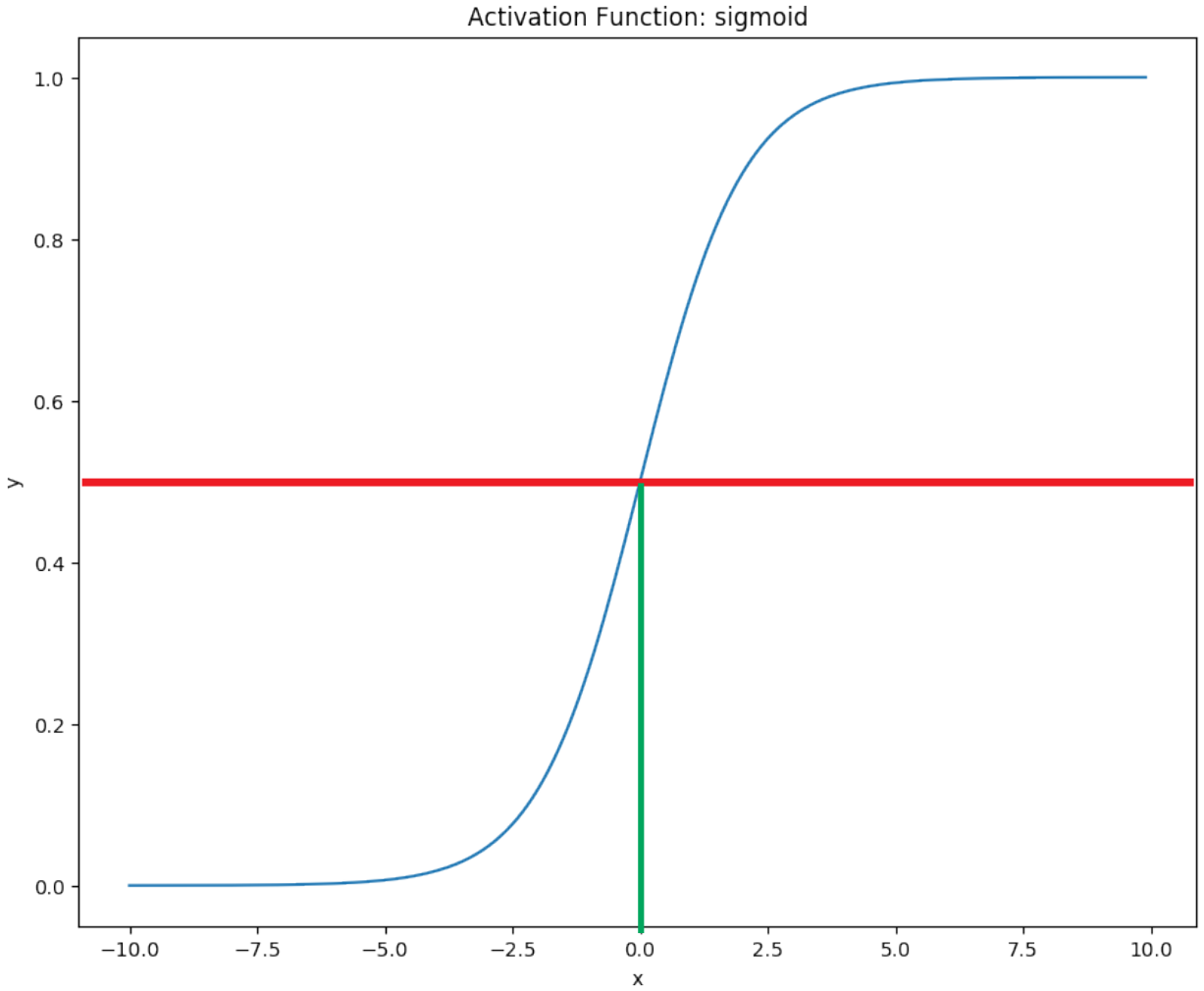


50% →
Decision boundary

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

Class 1 (100%)

Probability



Class 0 (0%)

50% →
Decision boundary

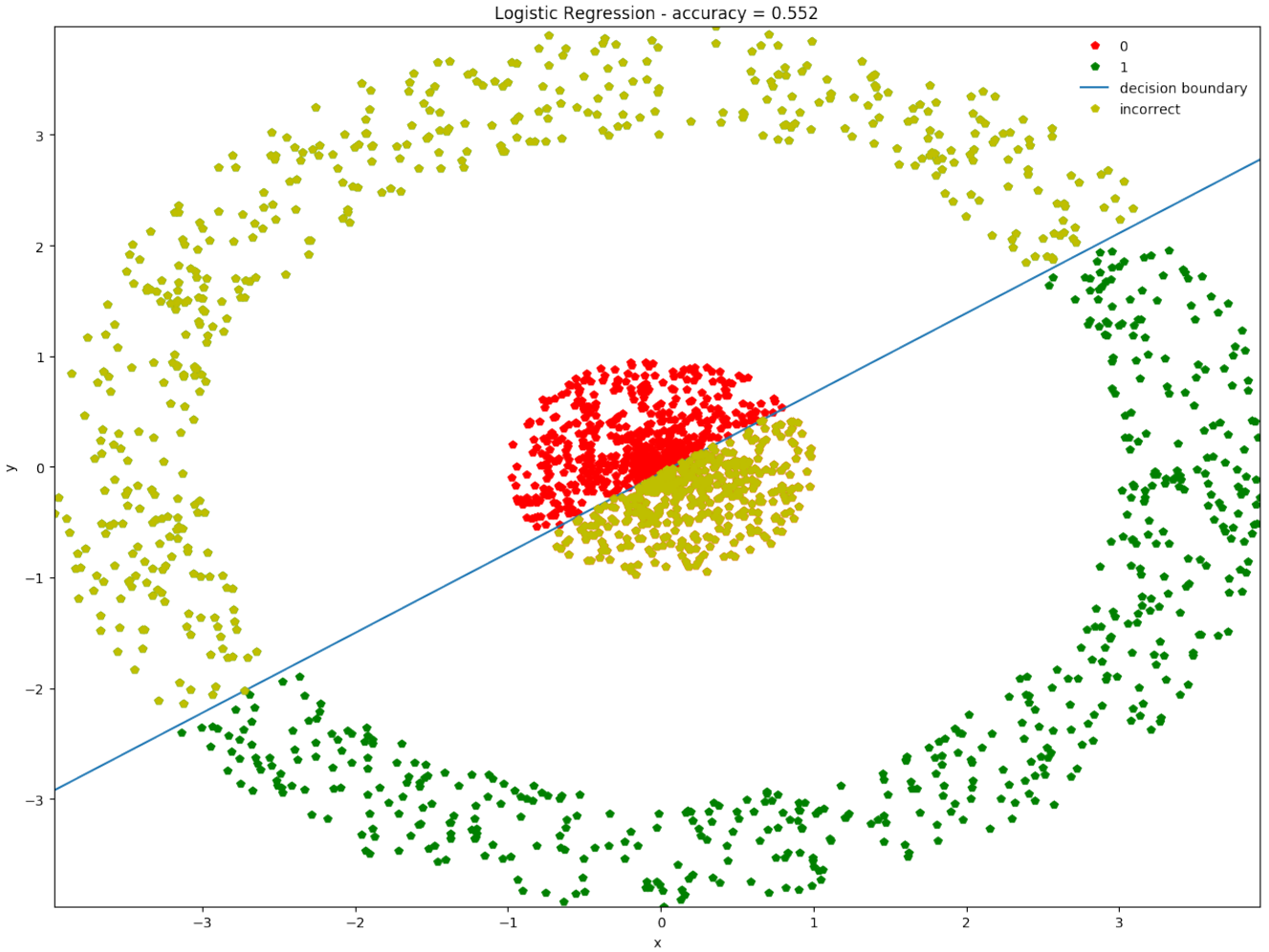
$$\text{y-axis} = \sigma(x) = \text{probability of being in class 1}$$

[redacted]

Red Hat

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

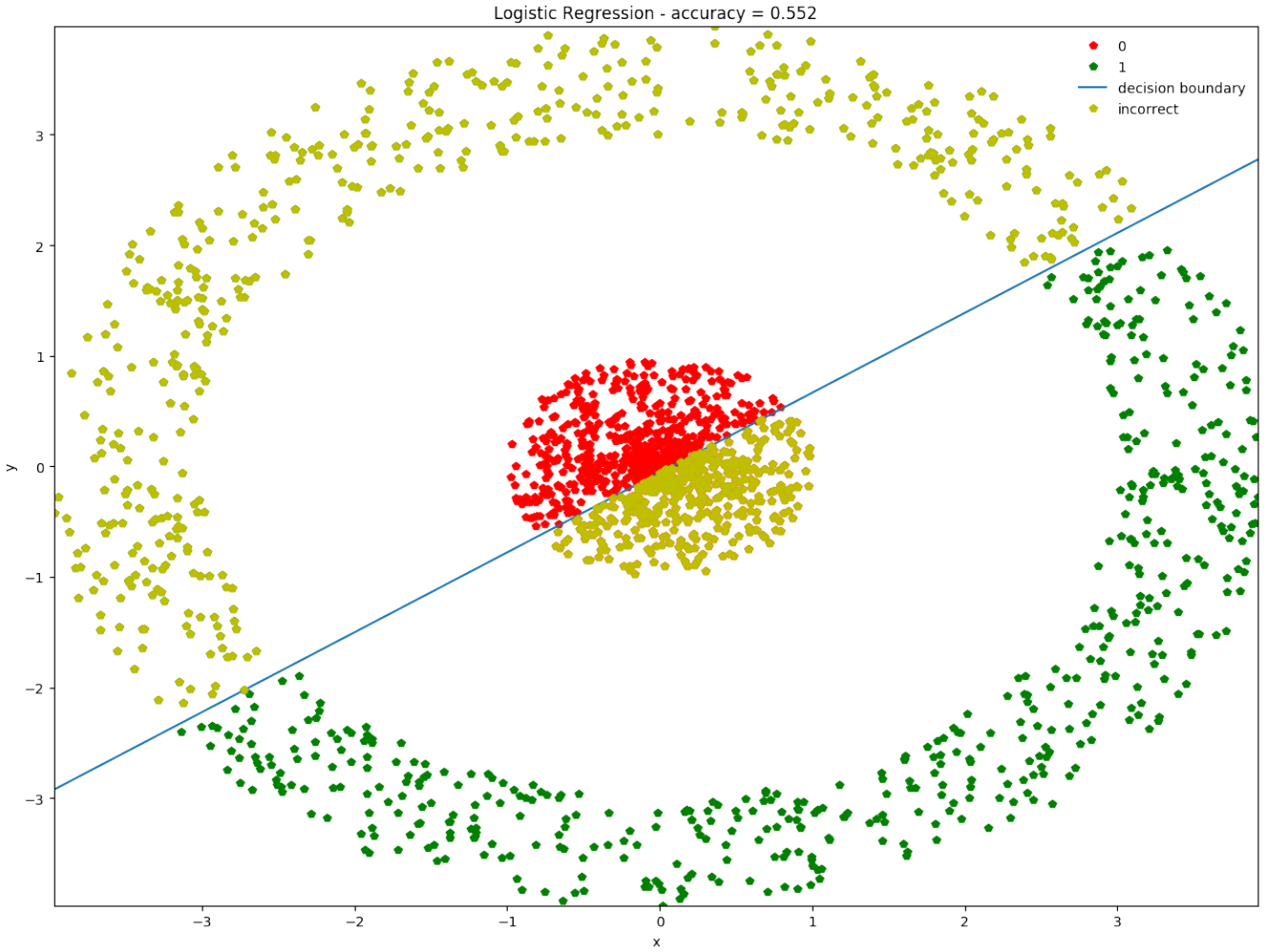
$$ax + by + c = 0 \implies y = -\frac{c}{b} - \frac{a}{b}x$$



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

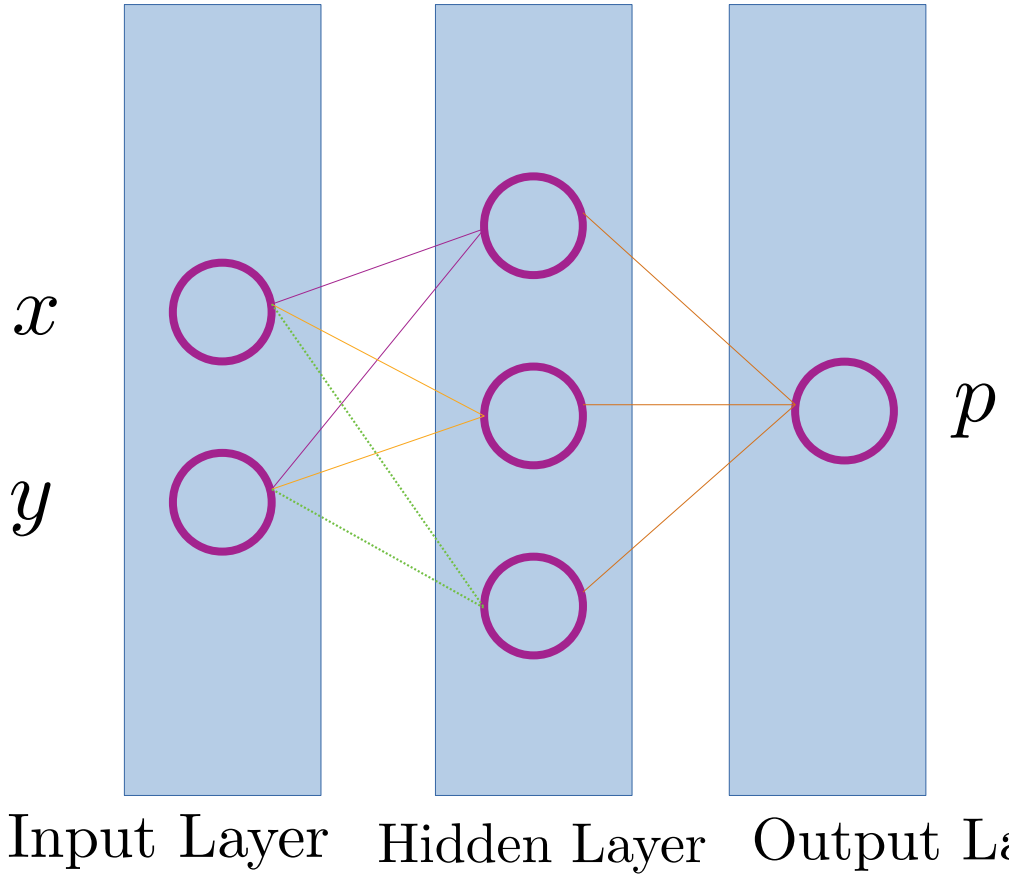
One solution:

Convert to polar coordinates

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

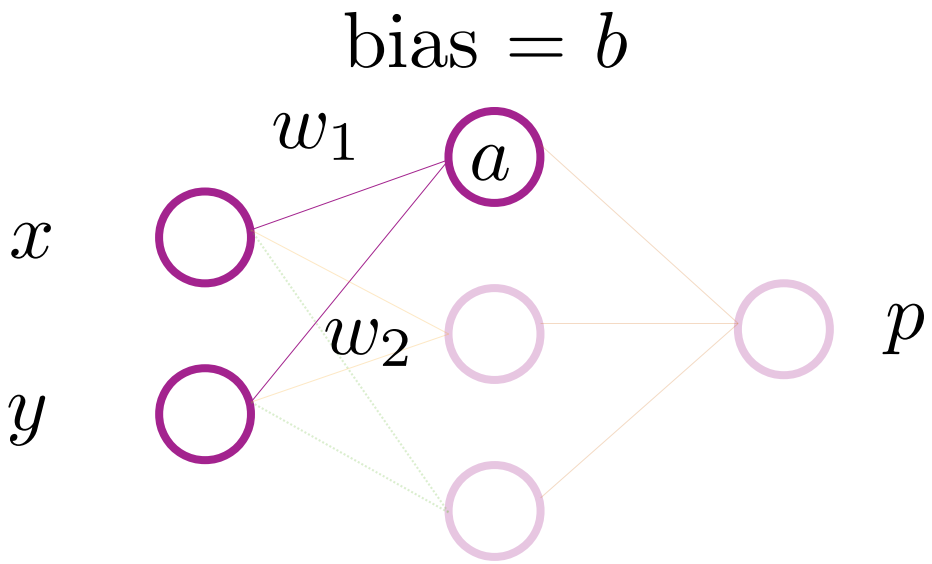
What about high dimensional data when we can't visualize?

Red Hat



Minimize:
Cross-entropy

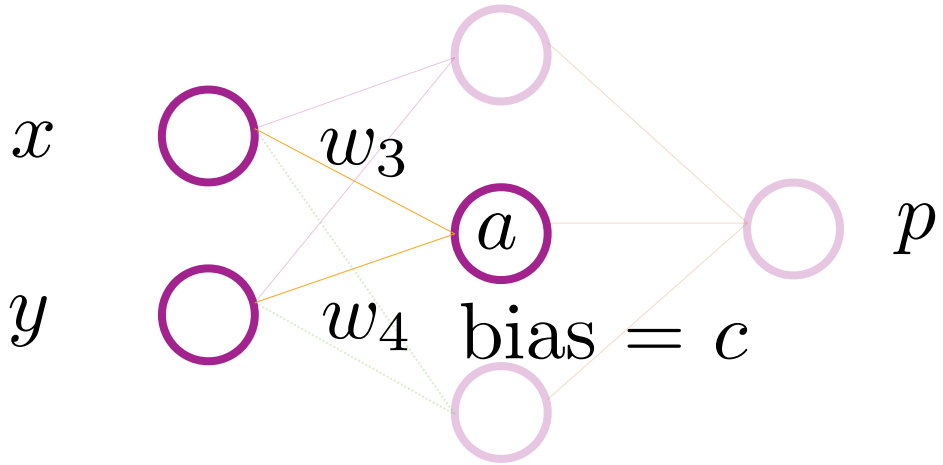
Activation = Sigmoid



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

Decision boundary:

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



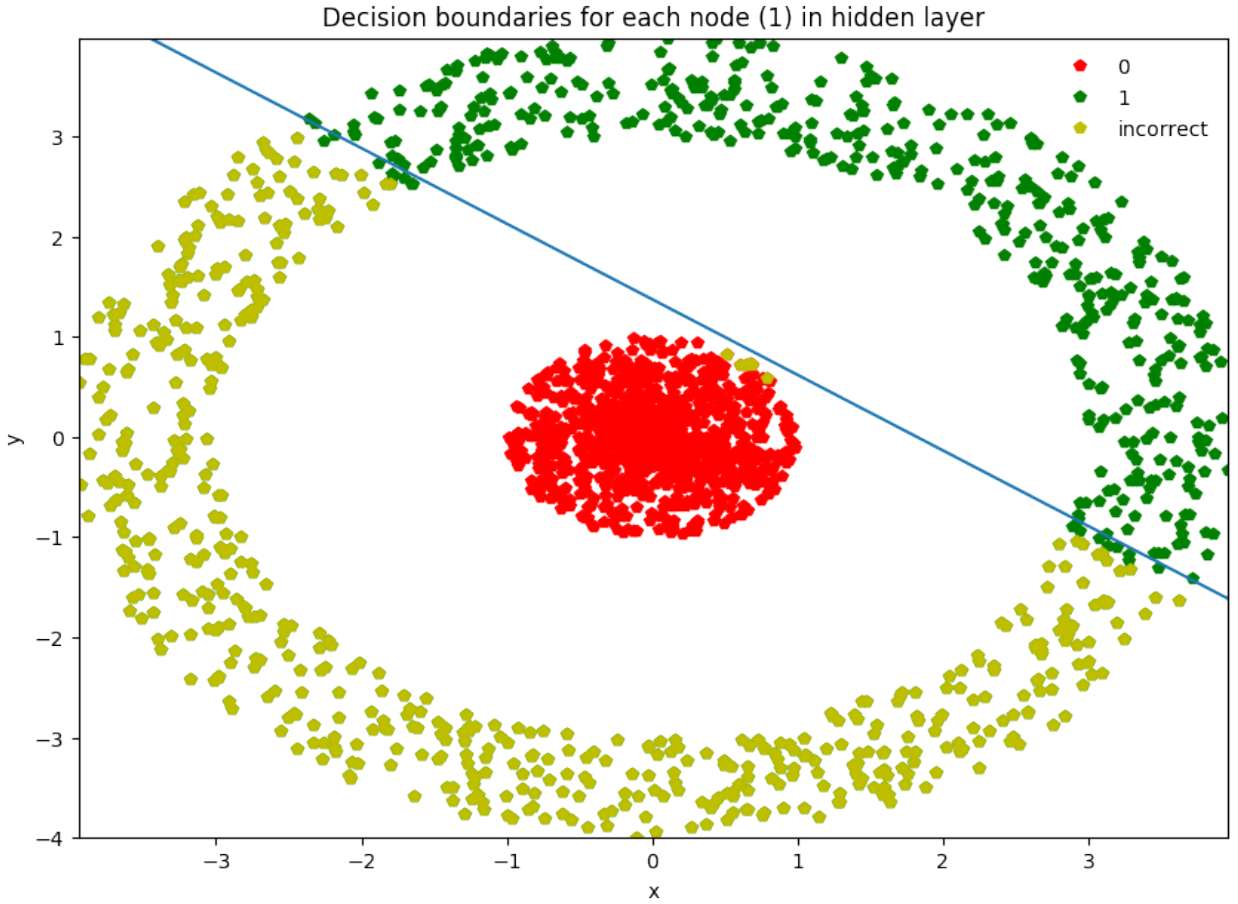
$$\boxed{a = \sigma(w_3 x + w_4 y + c)}$$

Decision boundary:

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

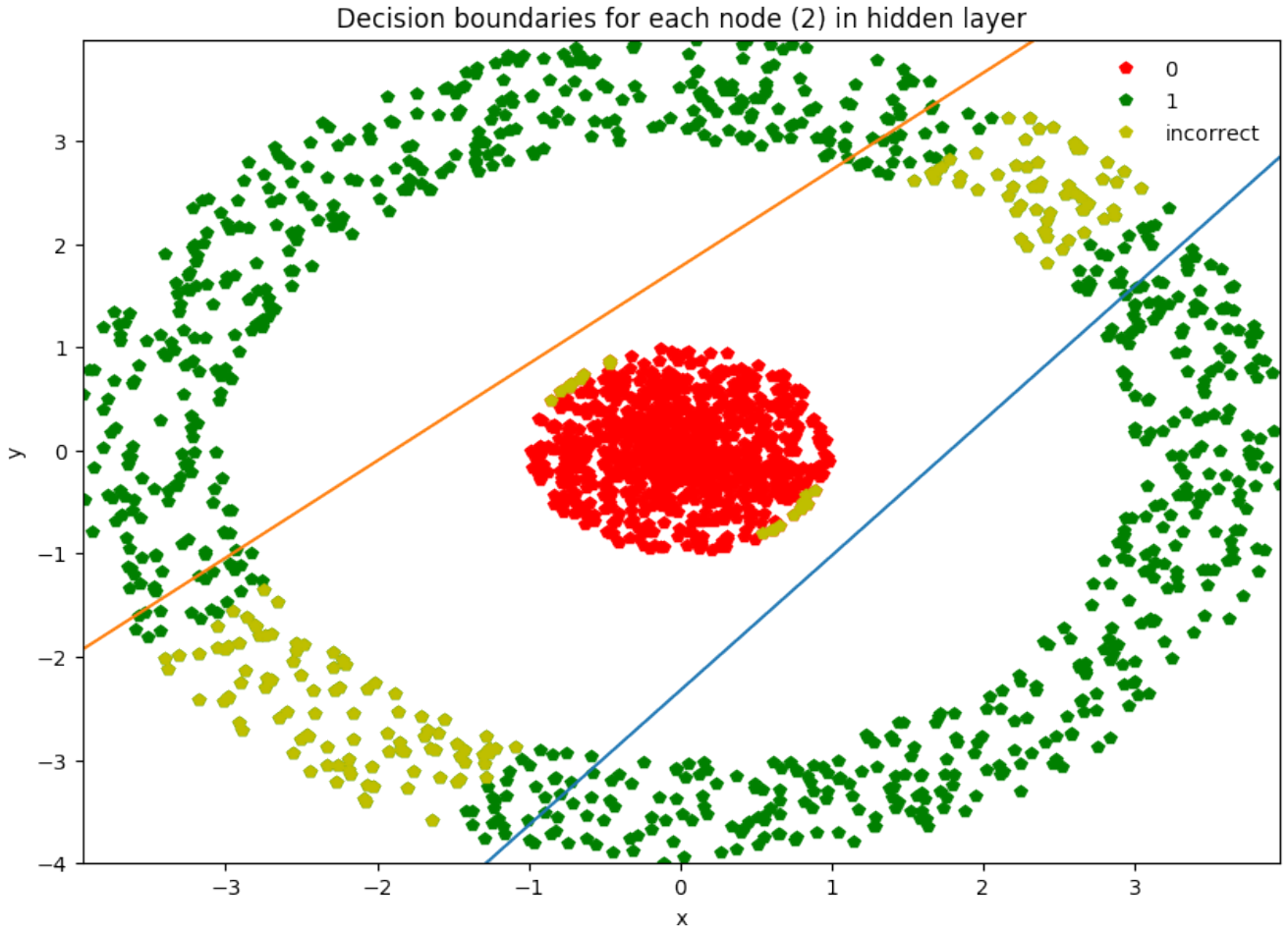
Input Layer Hidden Layer Output Layer

Hidden Nodes = 1



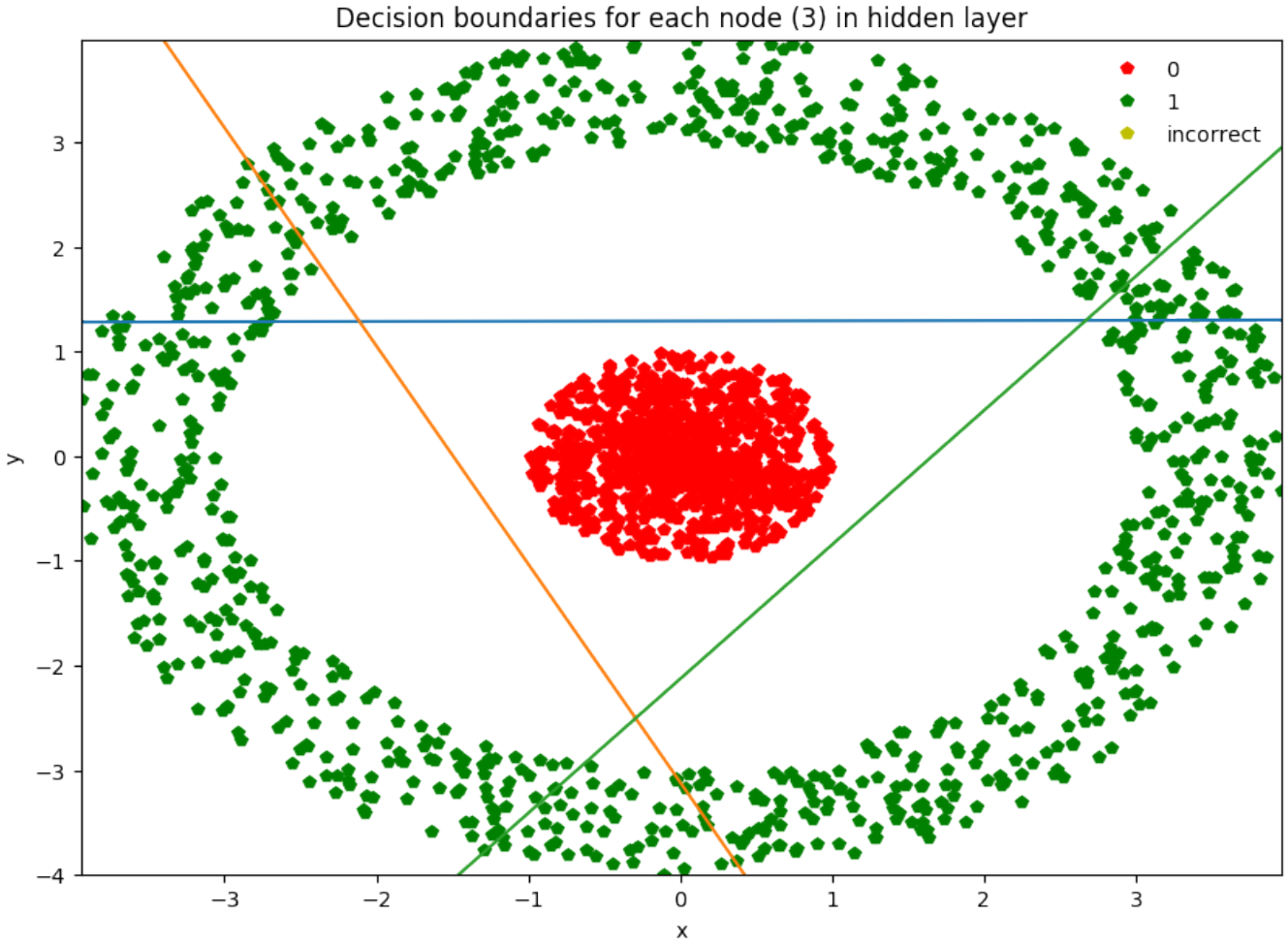
Red Hat

Hidden Nodes = 2



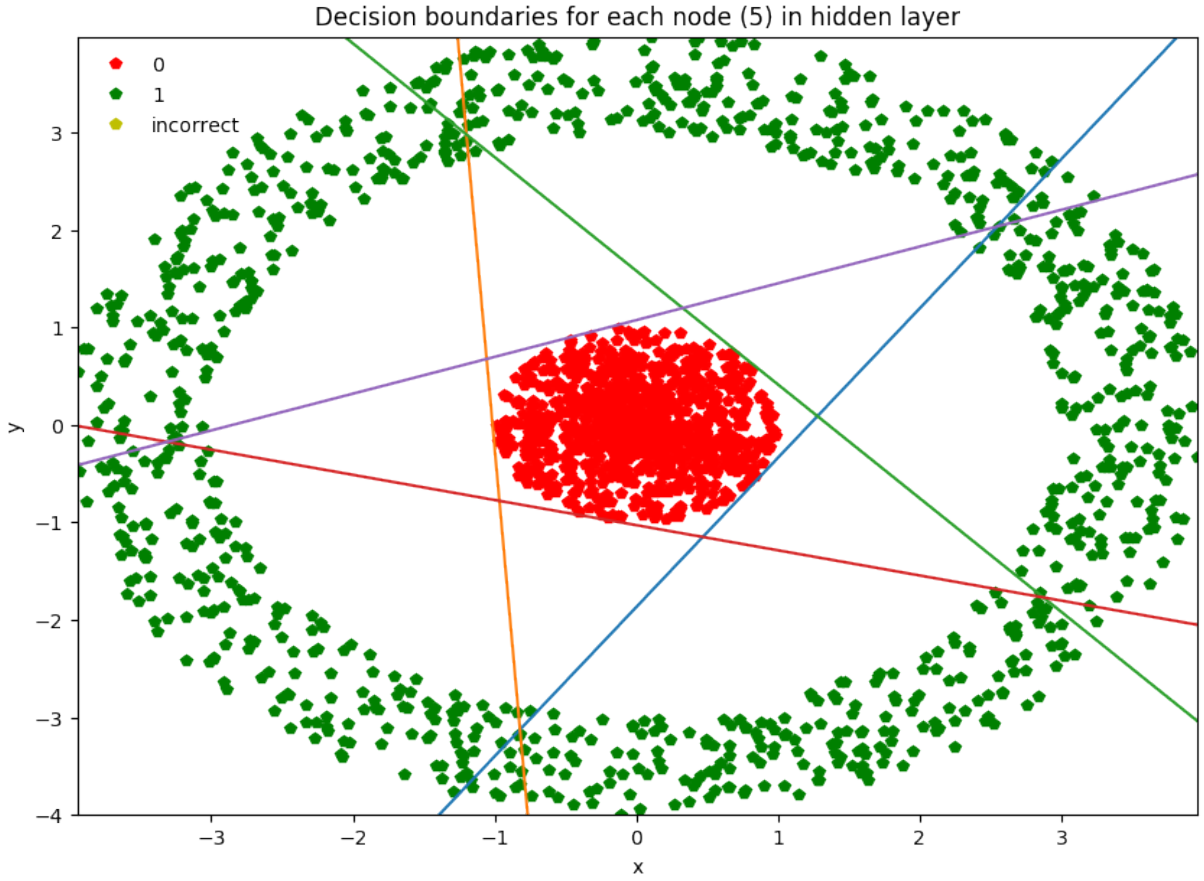
Red Hat

Hidden Nodes = 3



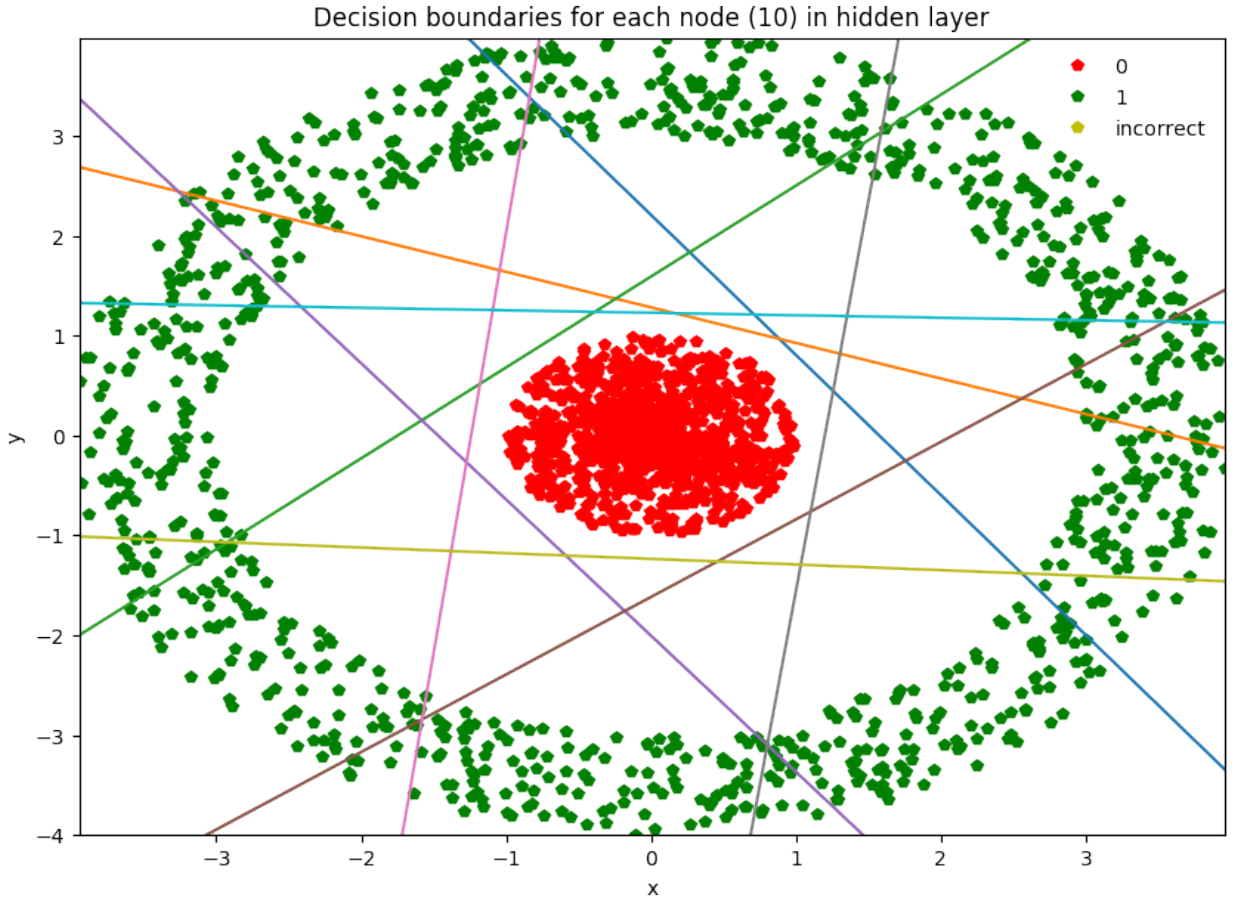
Red Hat

Hidden Nodes = 5



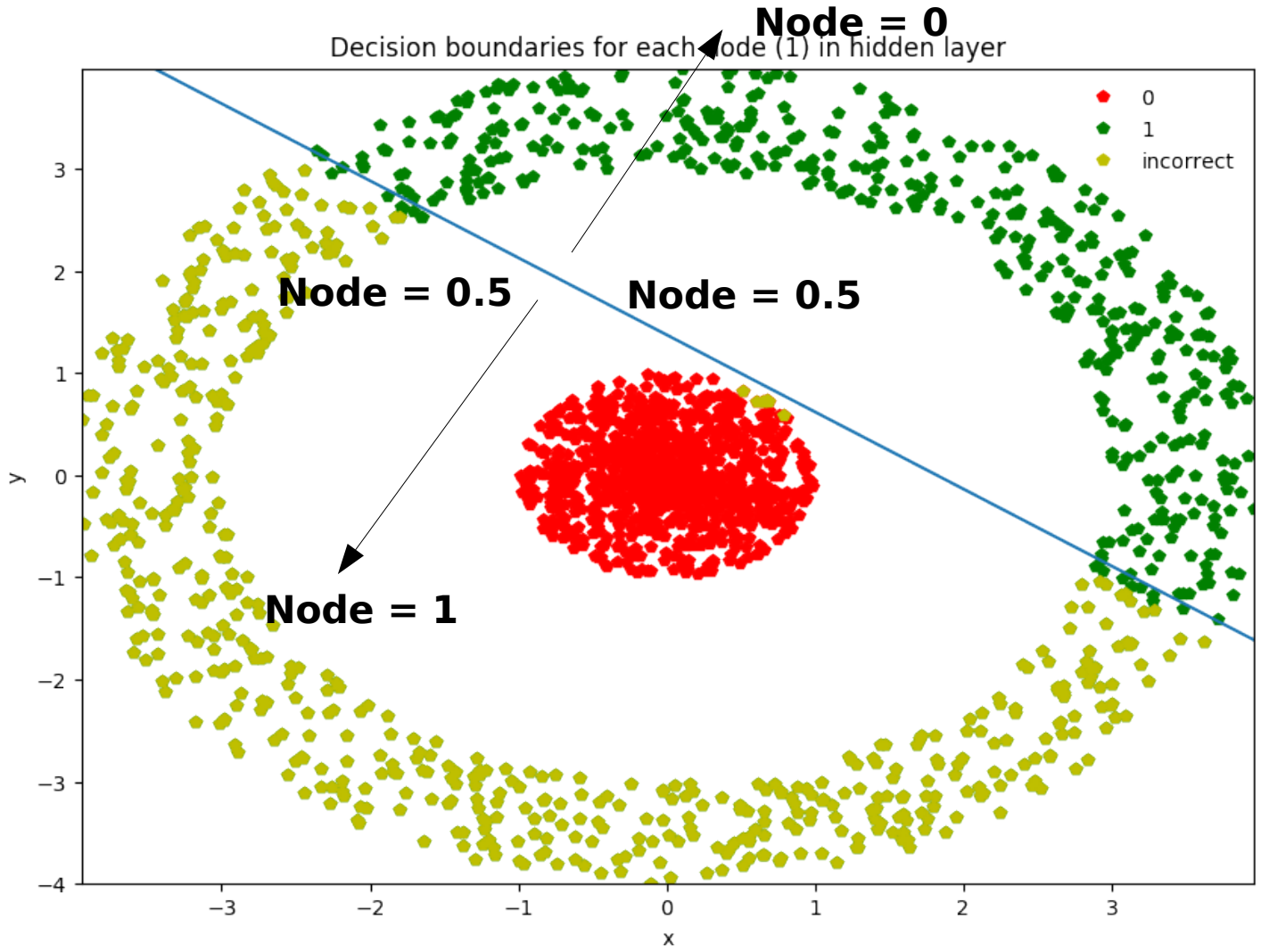
Red Hat

Hidden Nodes = 10



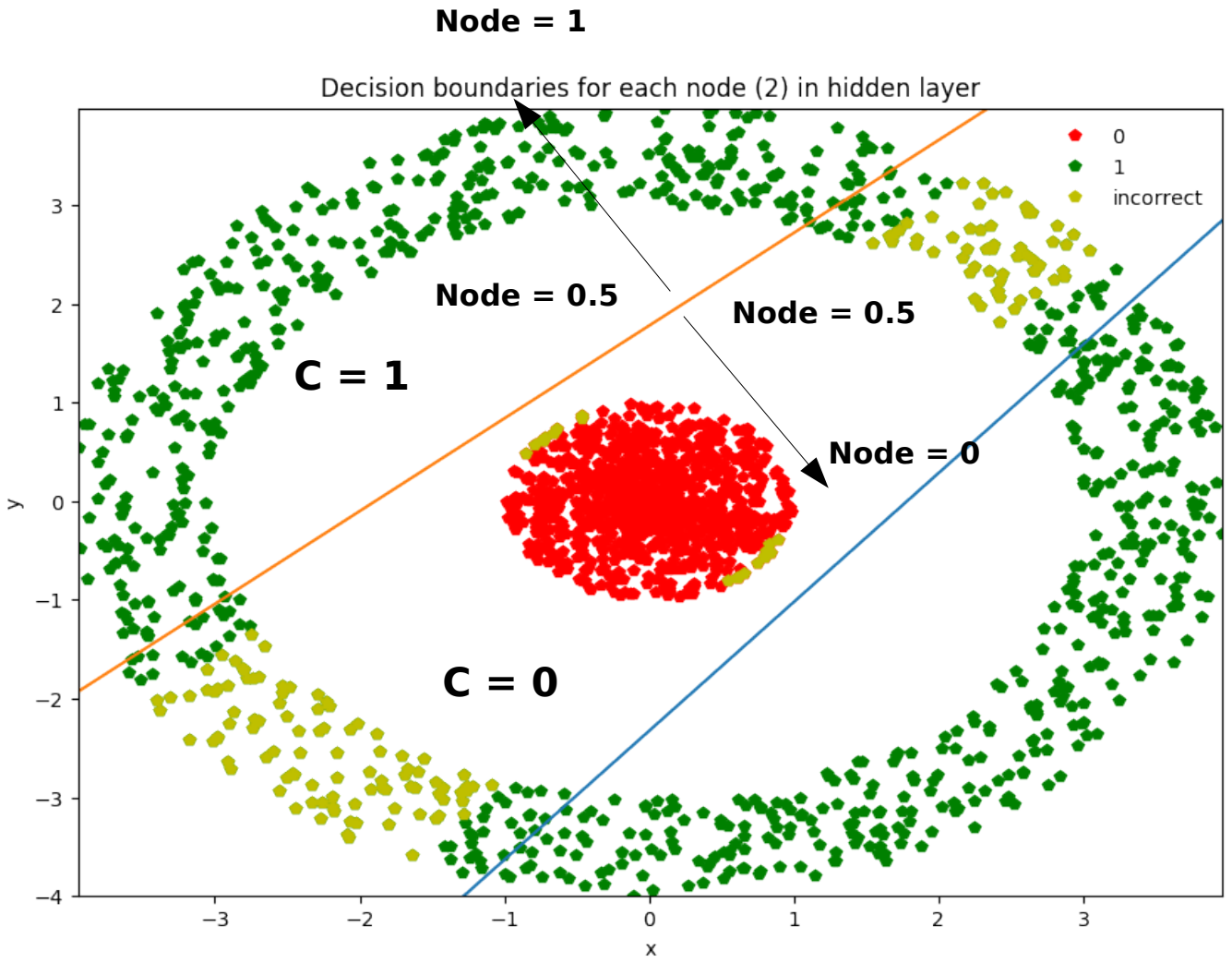
Red Hat

Hidden Nodes = 1



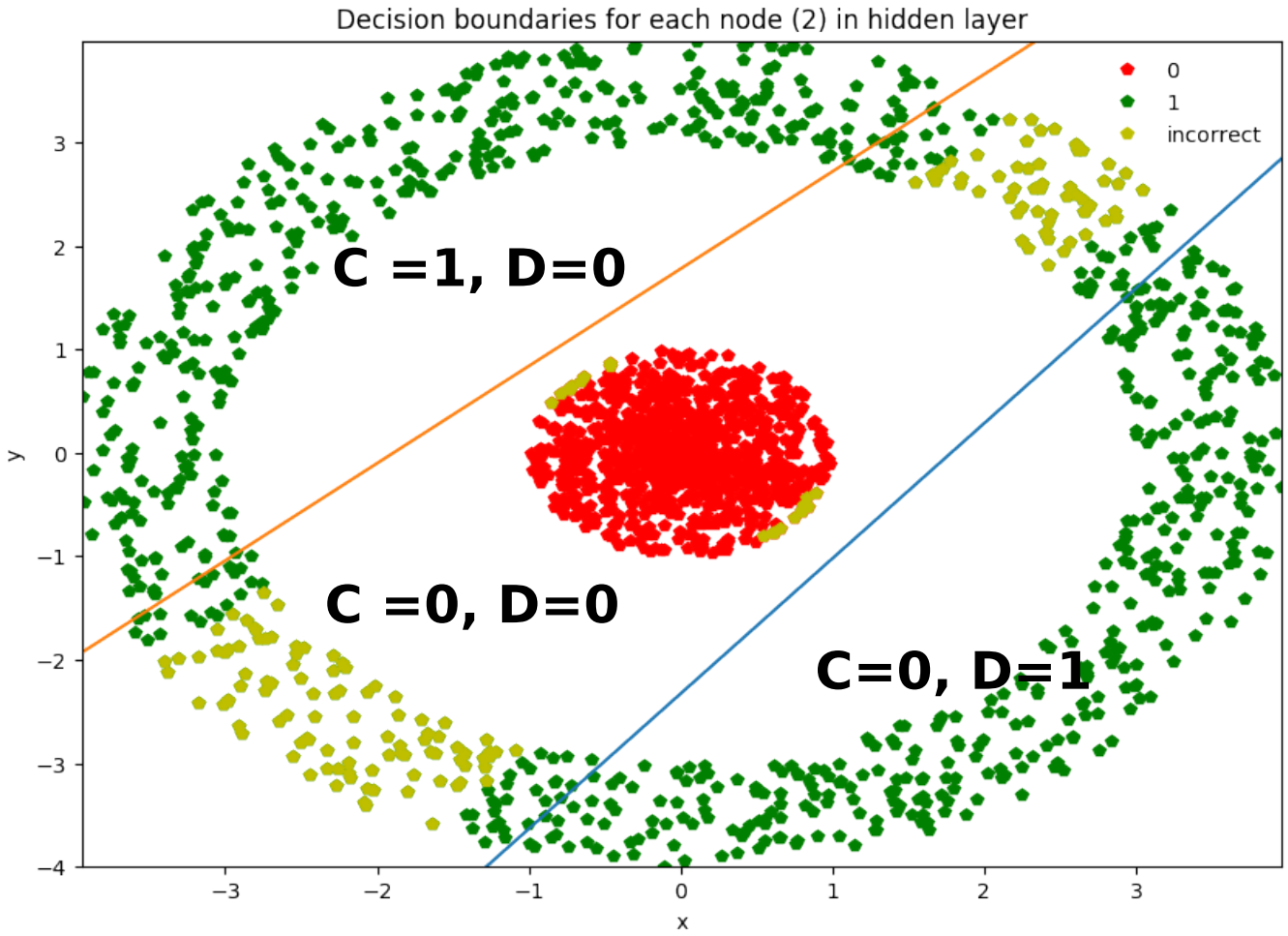
Red Hat

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

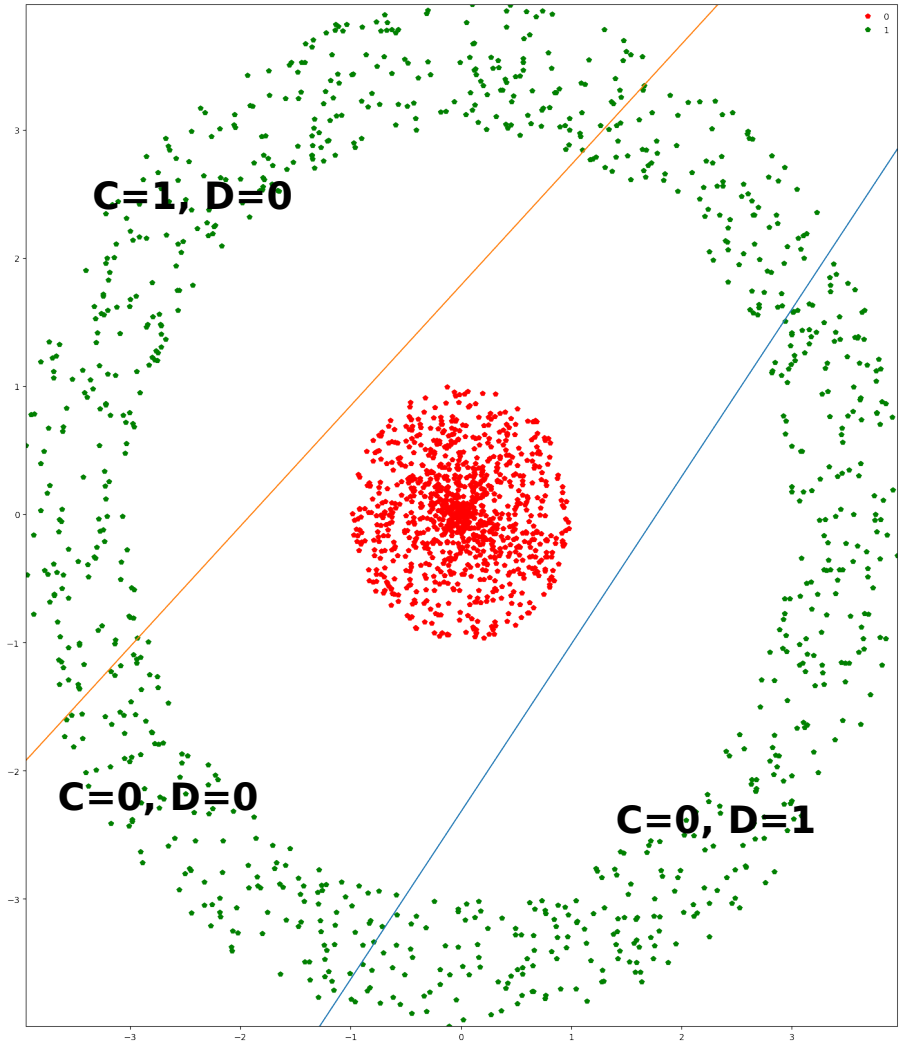
Keep increasing N (without overfitting) till regions ”pure”

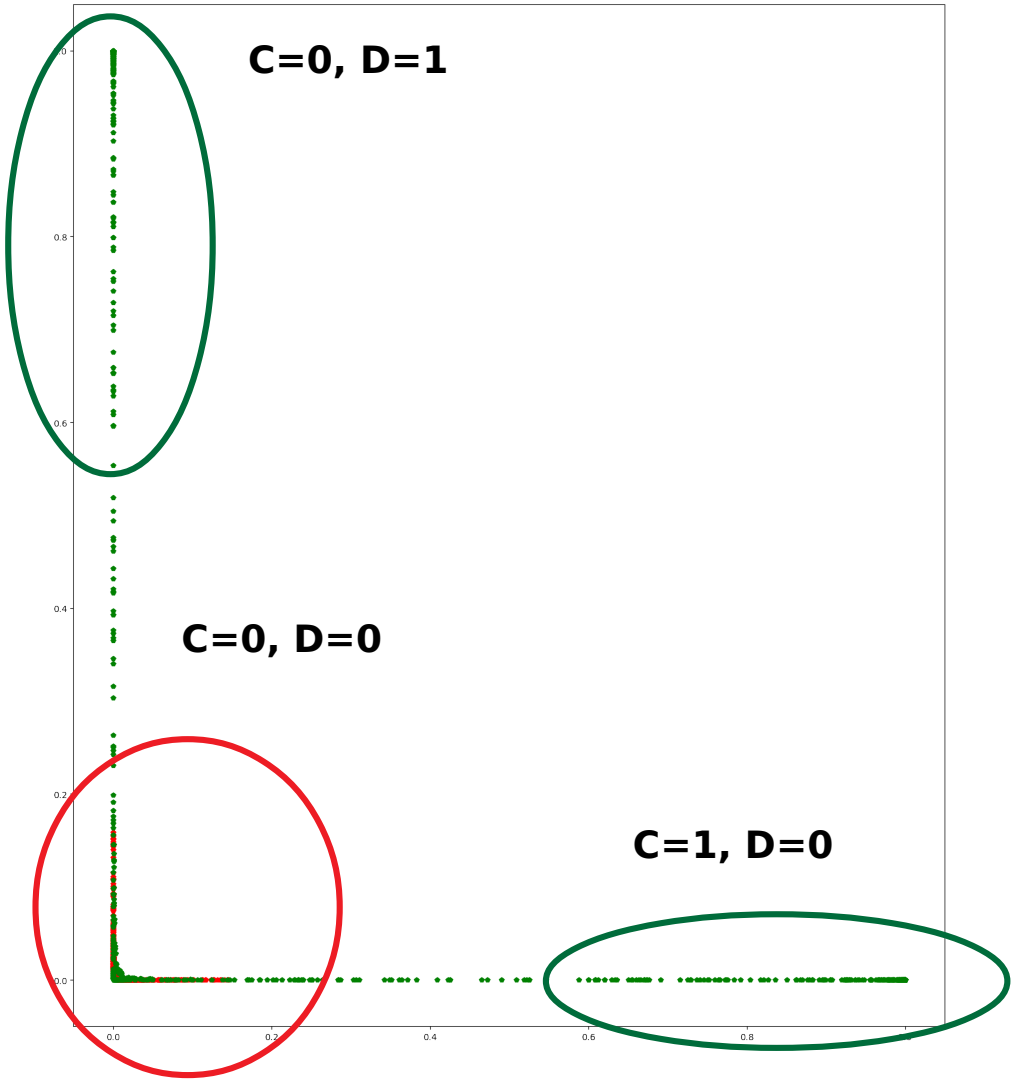
Map 2-dim input to N-dim space

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

in the hope of linearly separating classes

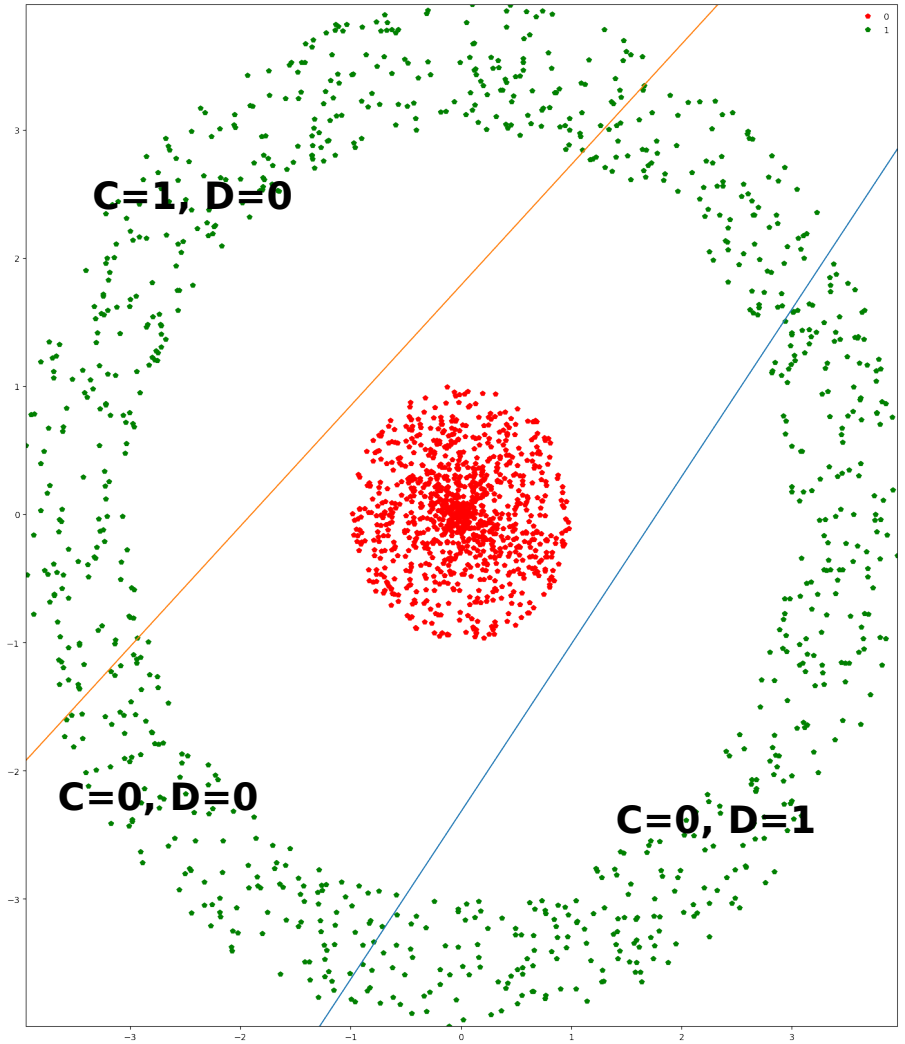
Map 2-dim input to 2-dim space

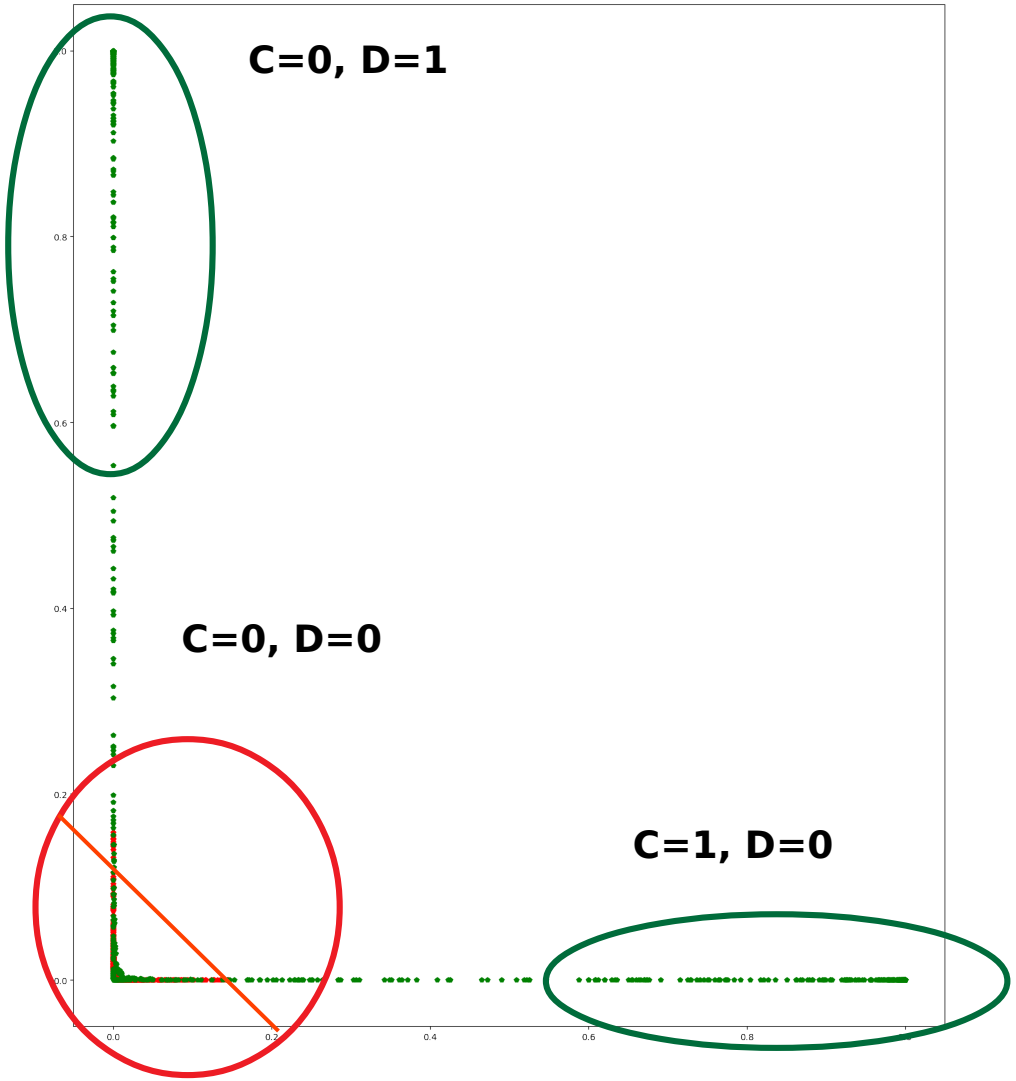




Red Hat

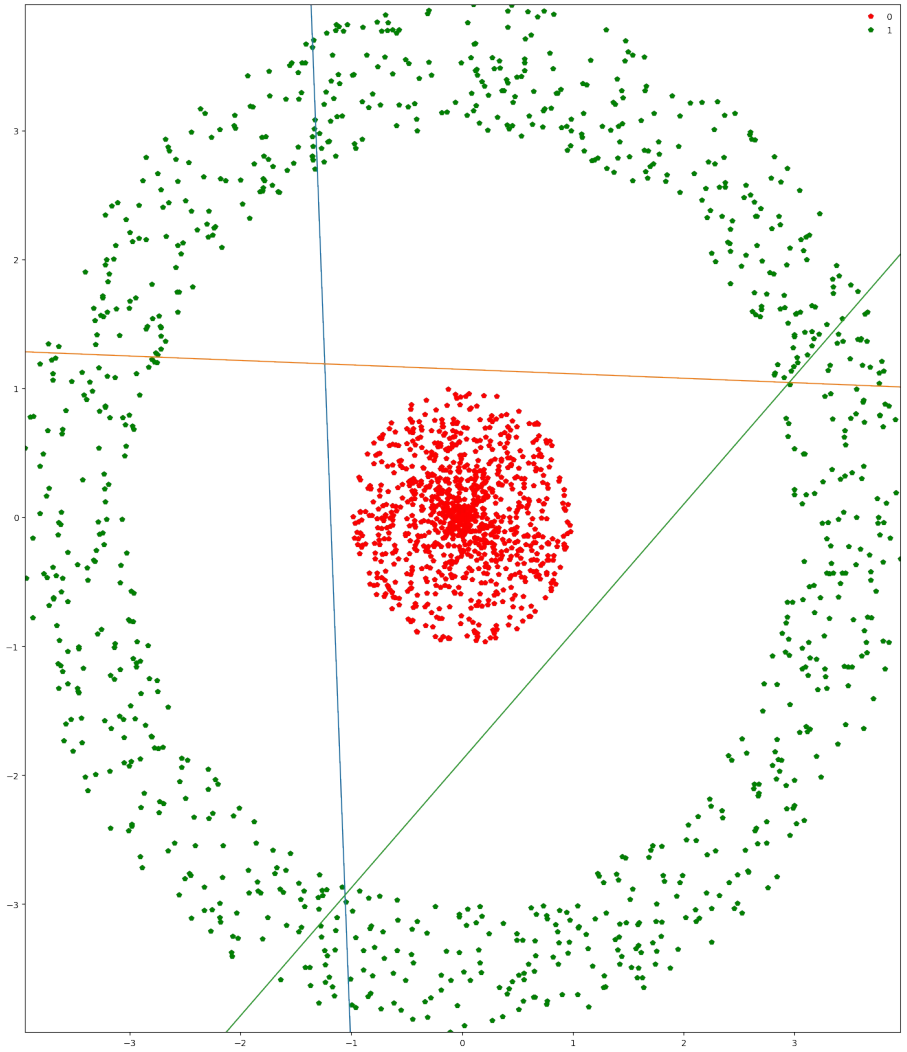
Map 2-dim input to 2-dim space

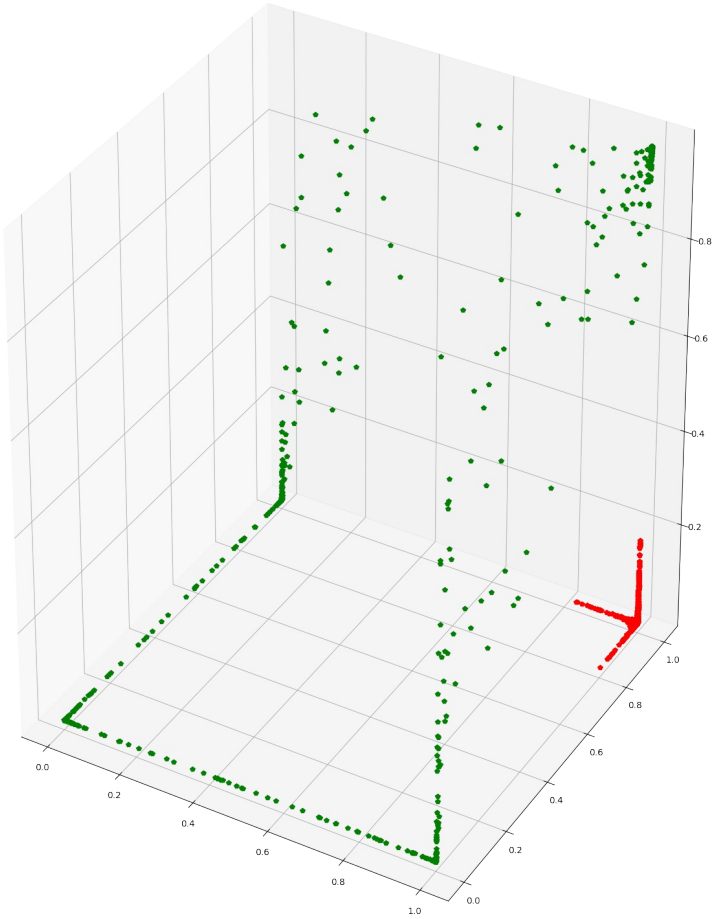




Red Hat

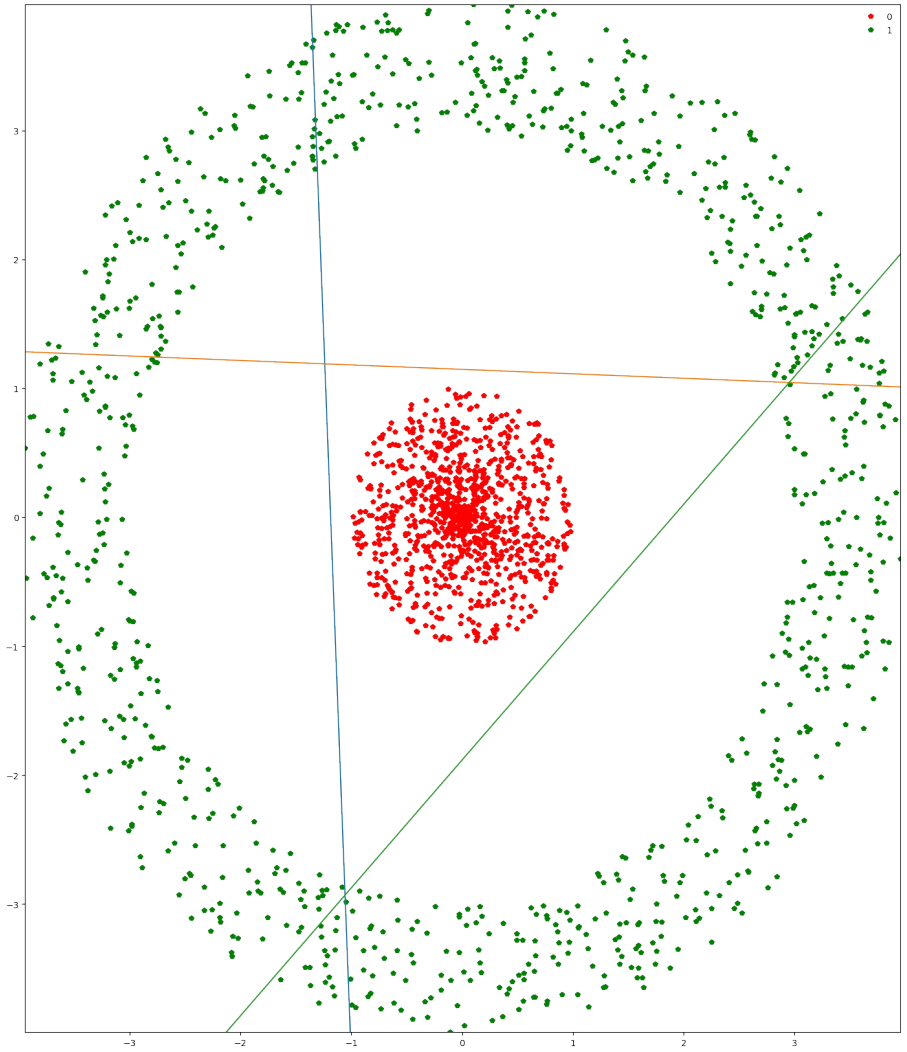
Map 2-dim input to 3-dim space

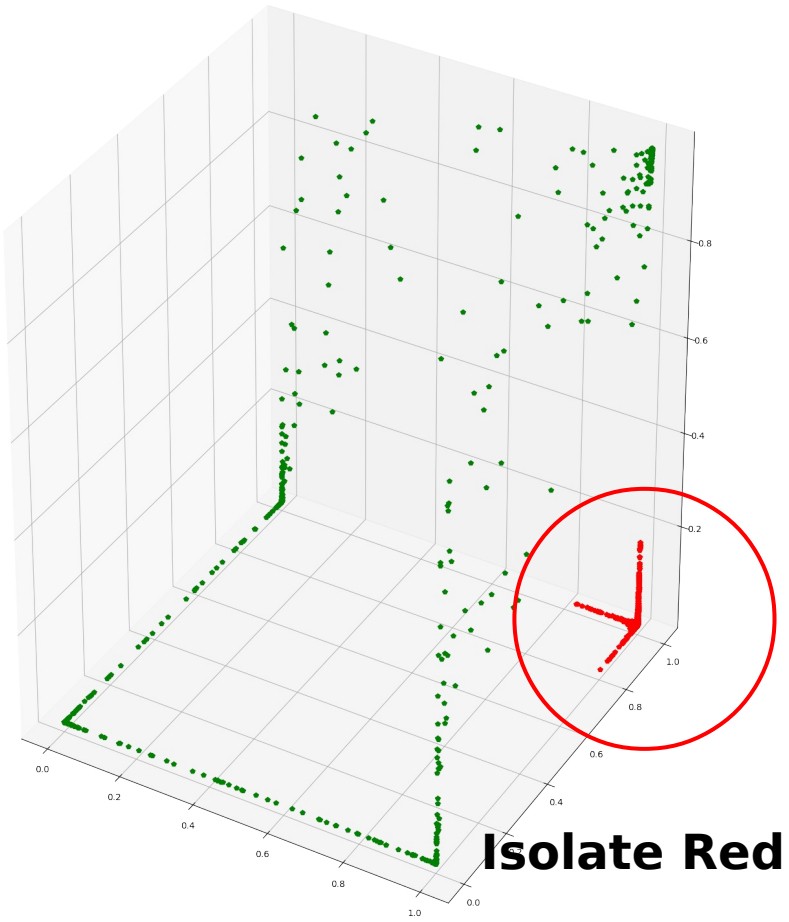




Red Hat

Map 2-dim input to 3-dim space

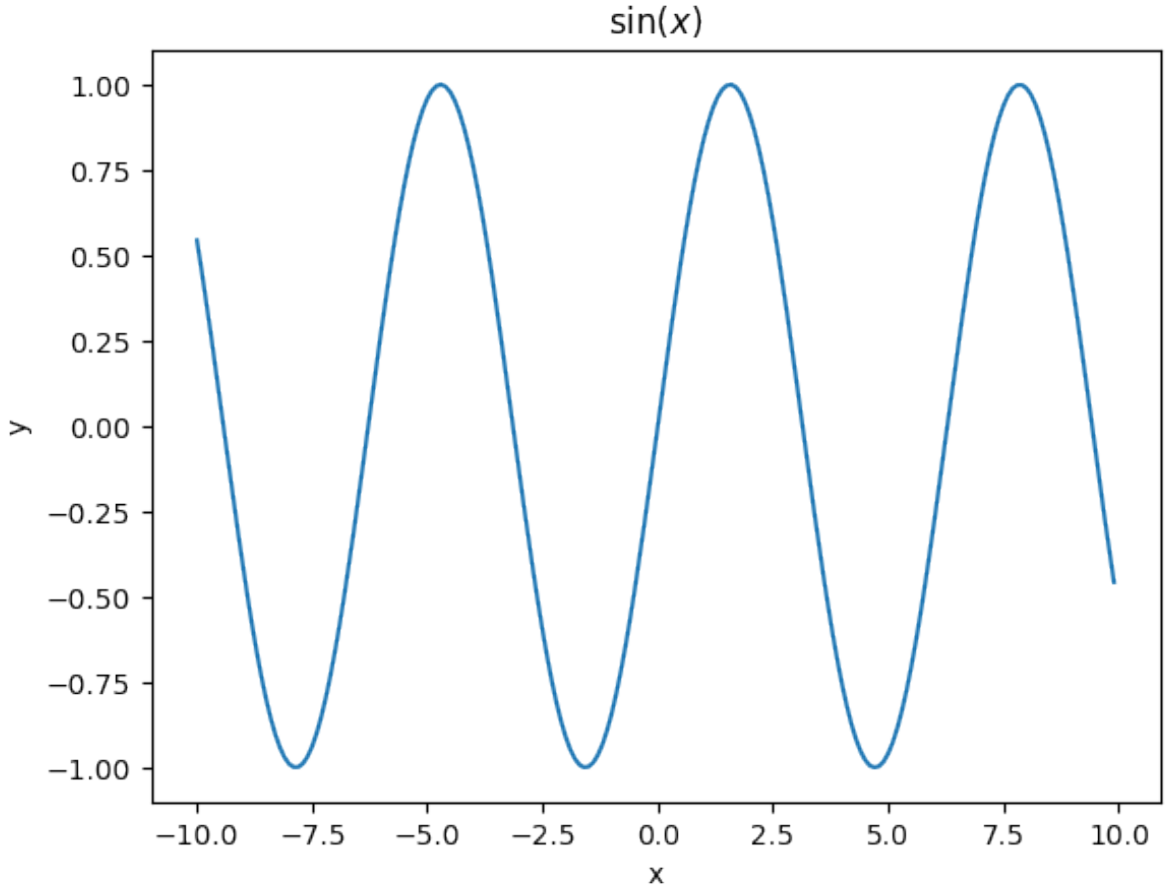




Isolate Red in corner

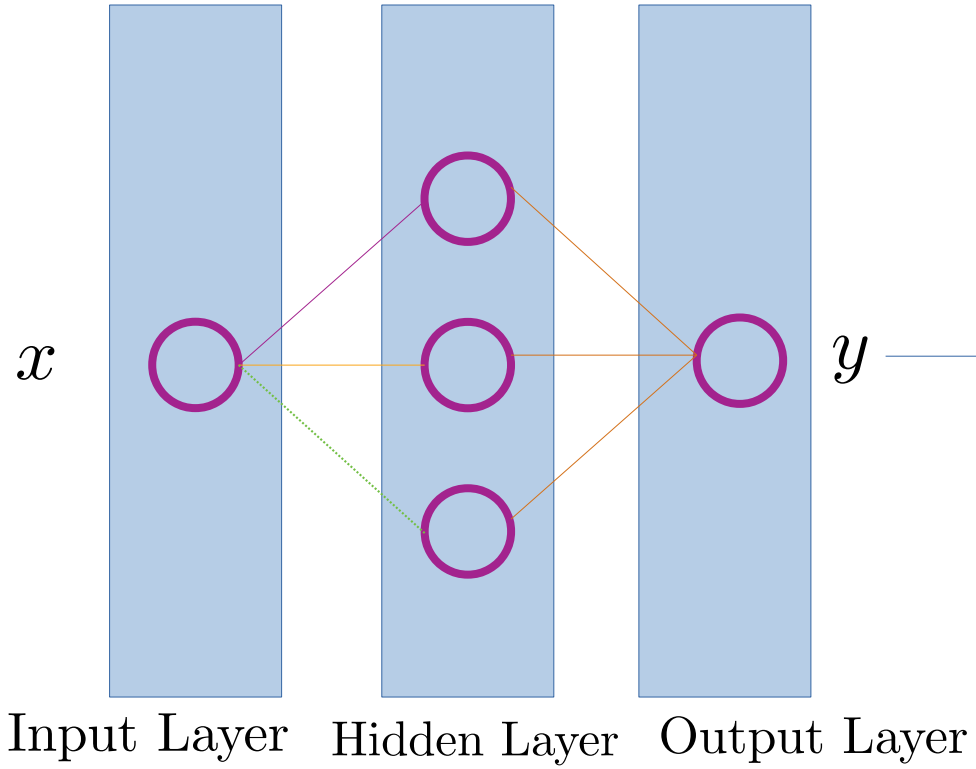
Red Hat

A Second Example: Regression



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

Red Hat

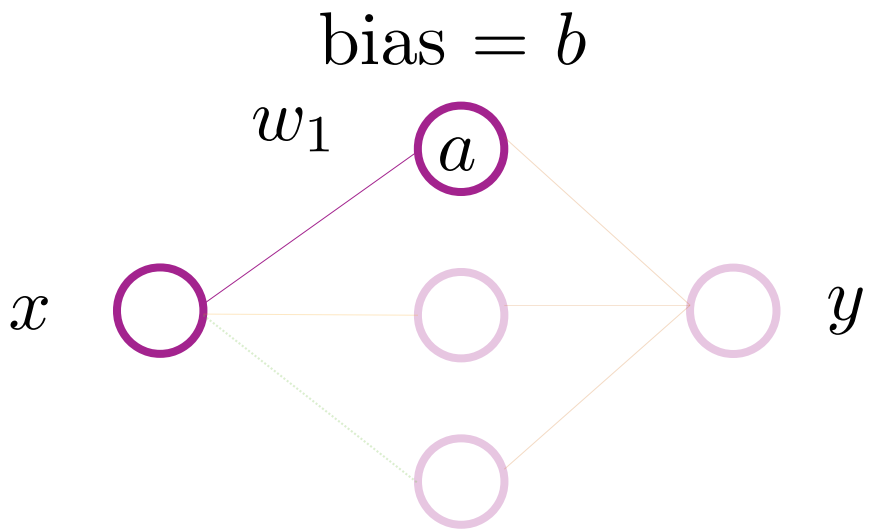


Minimize:

Mean Squared Error

Activation = Sigmoid

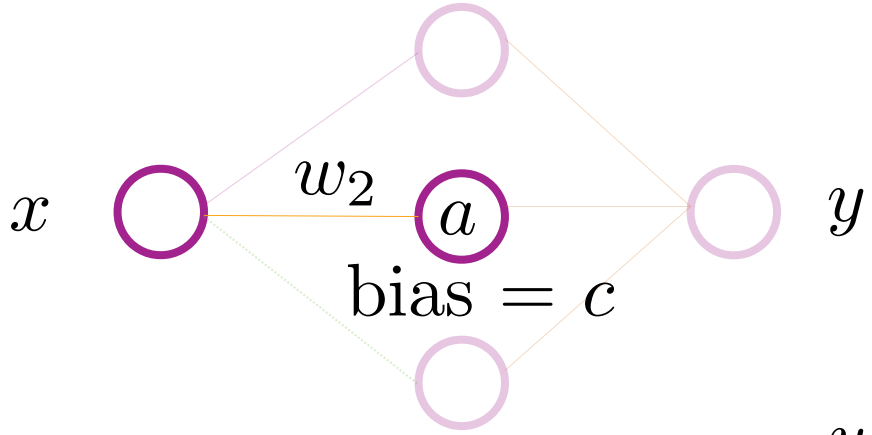
Output = Linear



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

Decision boundary:

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

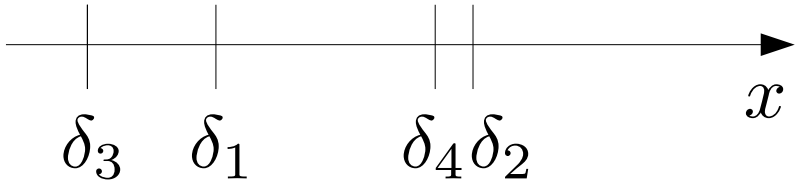


$$\boxed{a = \sigma(w_2 x + c)}$$

Decision boundary:

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

1 decision boundary per hidden node



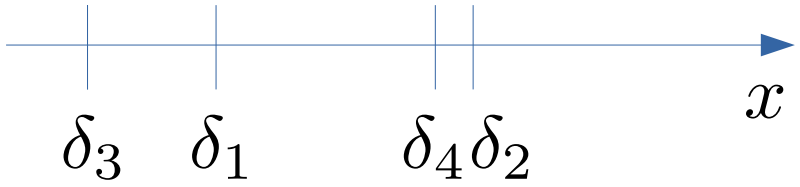
$$\delta_1 = -\frac{b_1}{w_1}$$

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

$$\vdots$$

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

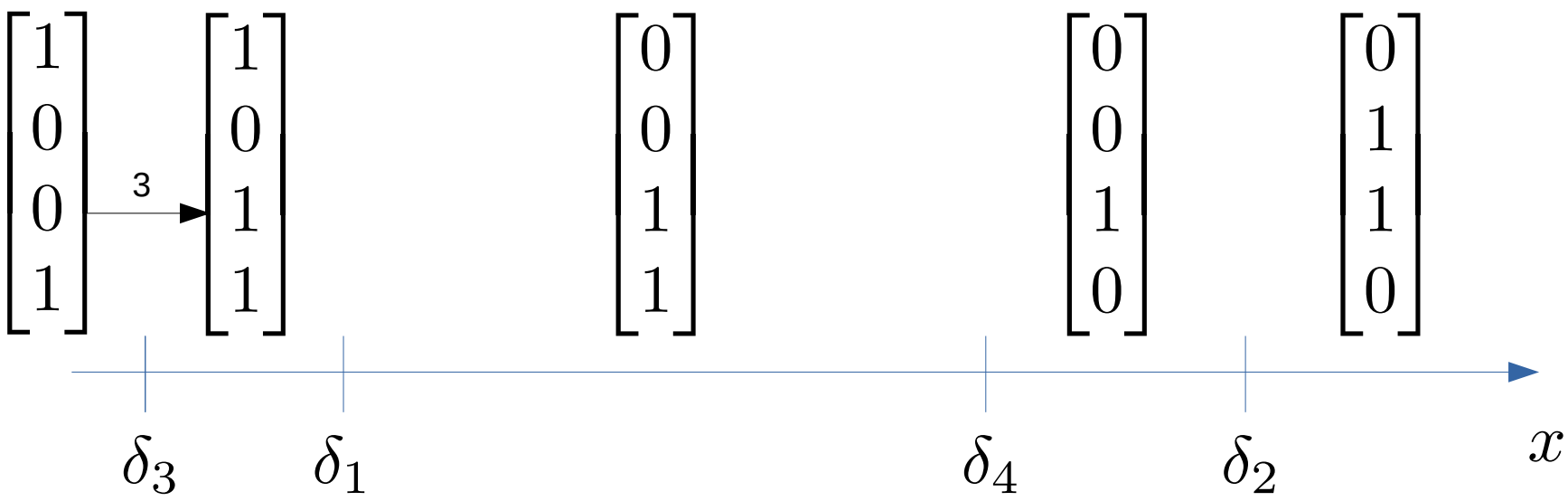
Walk from left to right

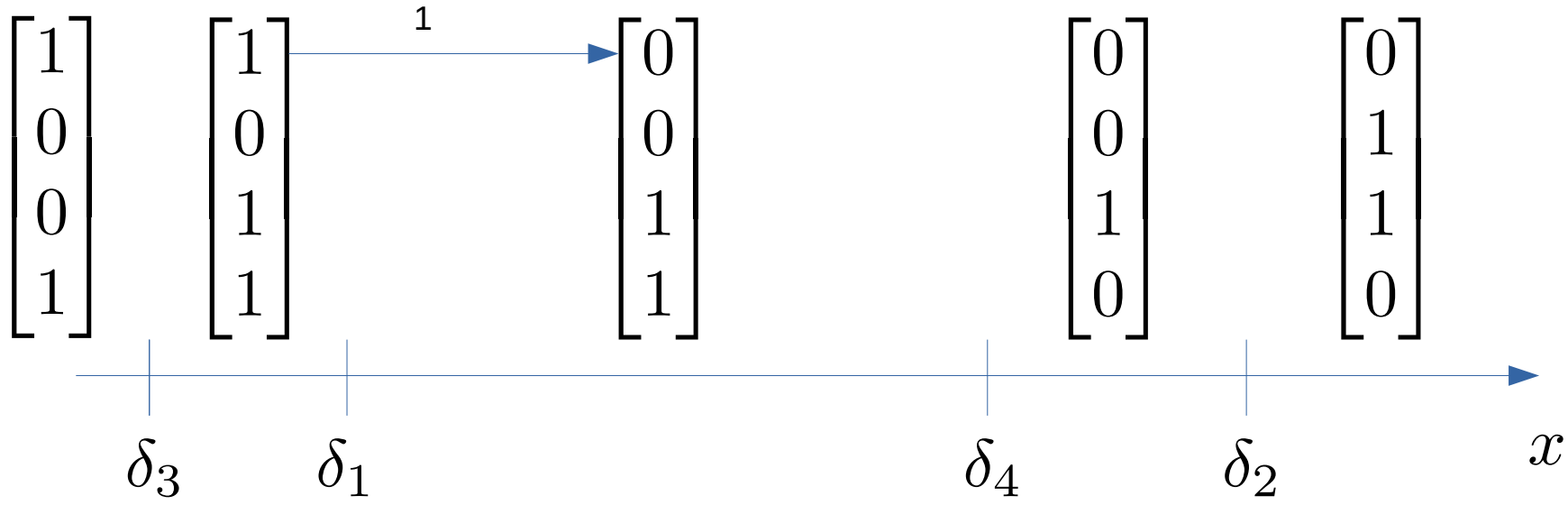


Each time you cross a boundary, a hidden node either:

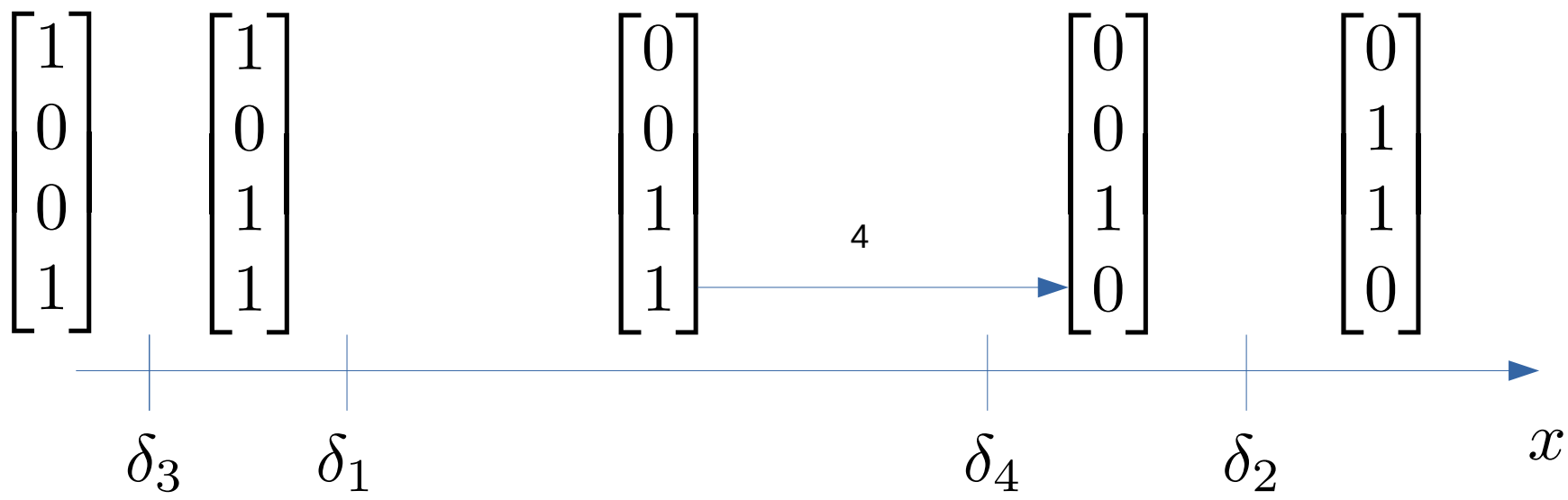
Turns on i.e. goes from 1 → 0

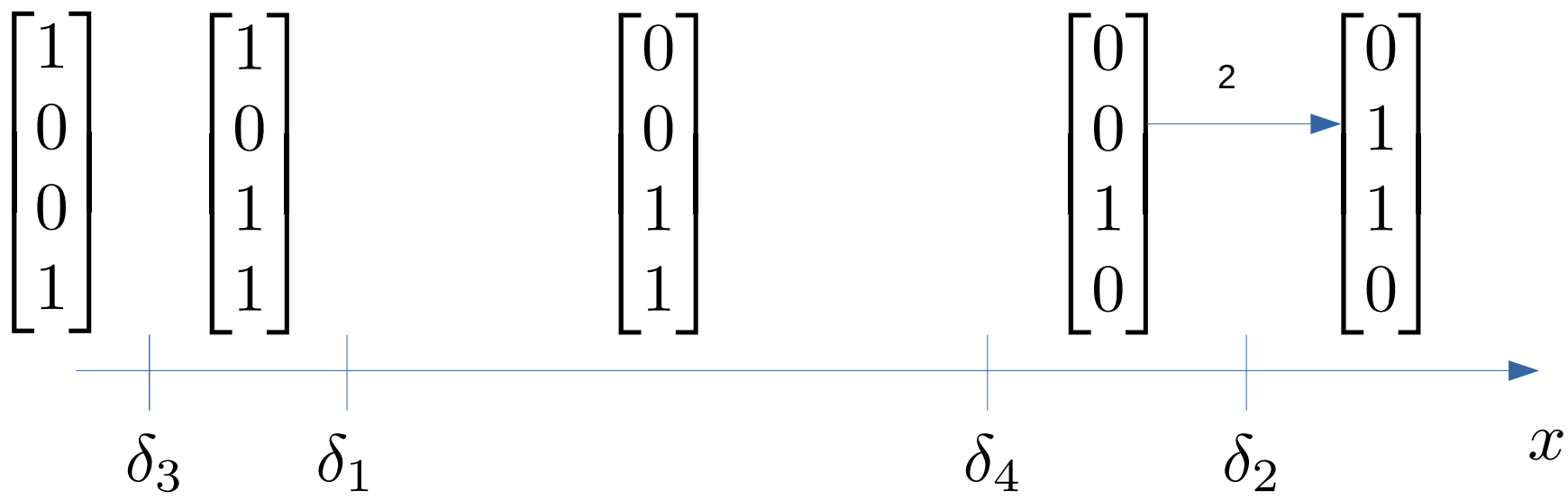
Turns off i.e. goes from 0 → 1



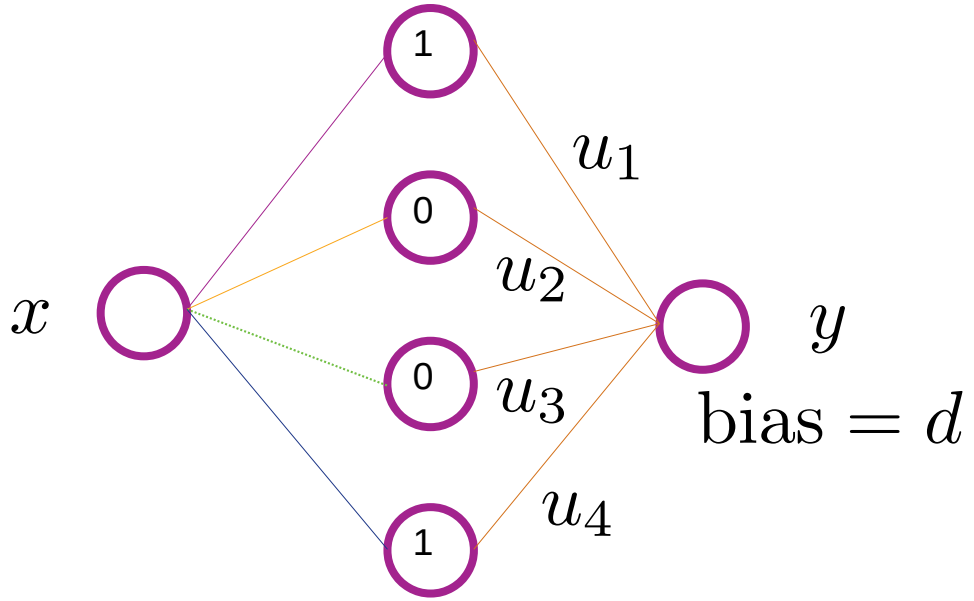

1
$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
δ_3
δ_1
δ_4
δ_2
x

Red Hat

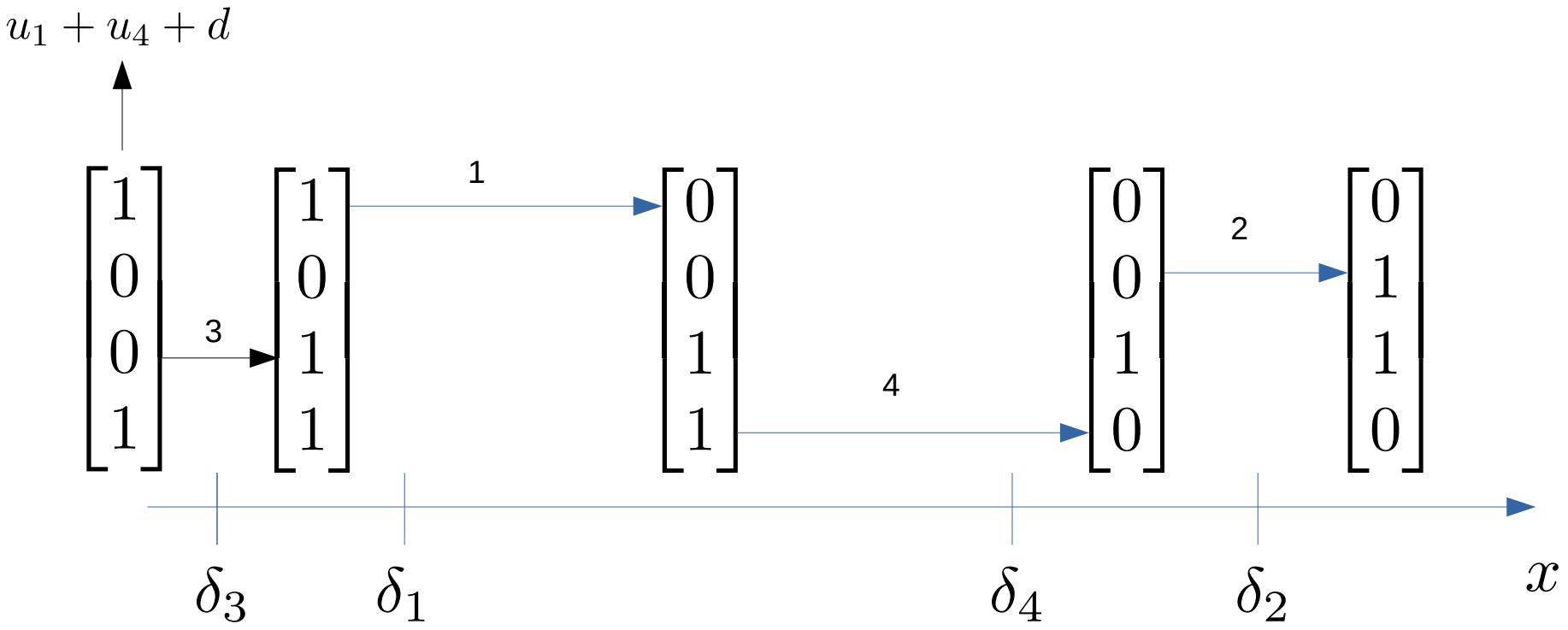


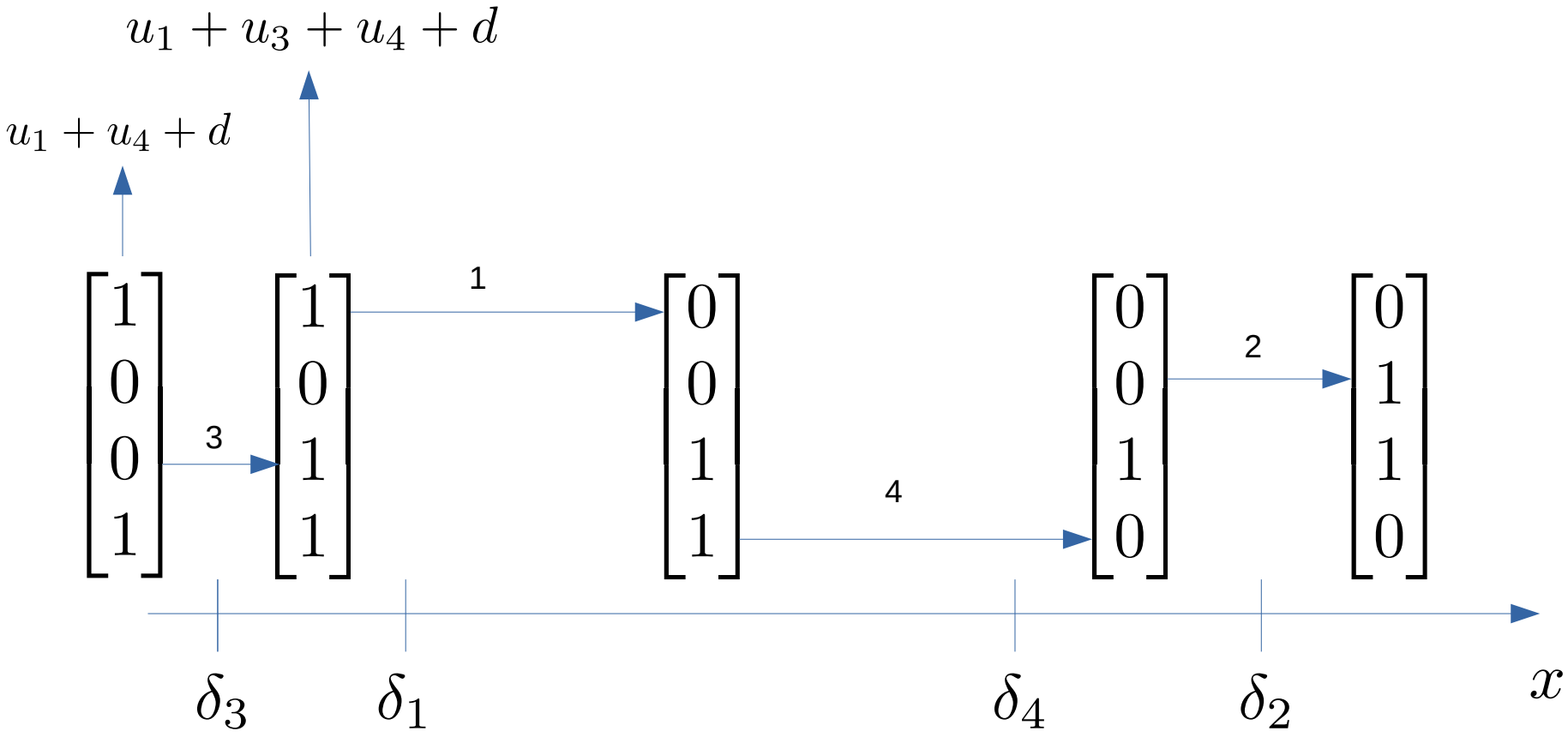

$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
2
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
δ_3
δ_1
δ_4
δ_2
x

Red Hat

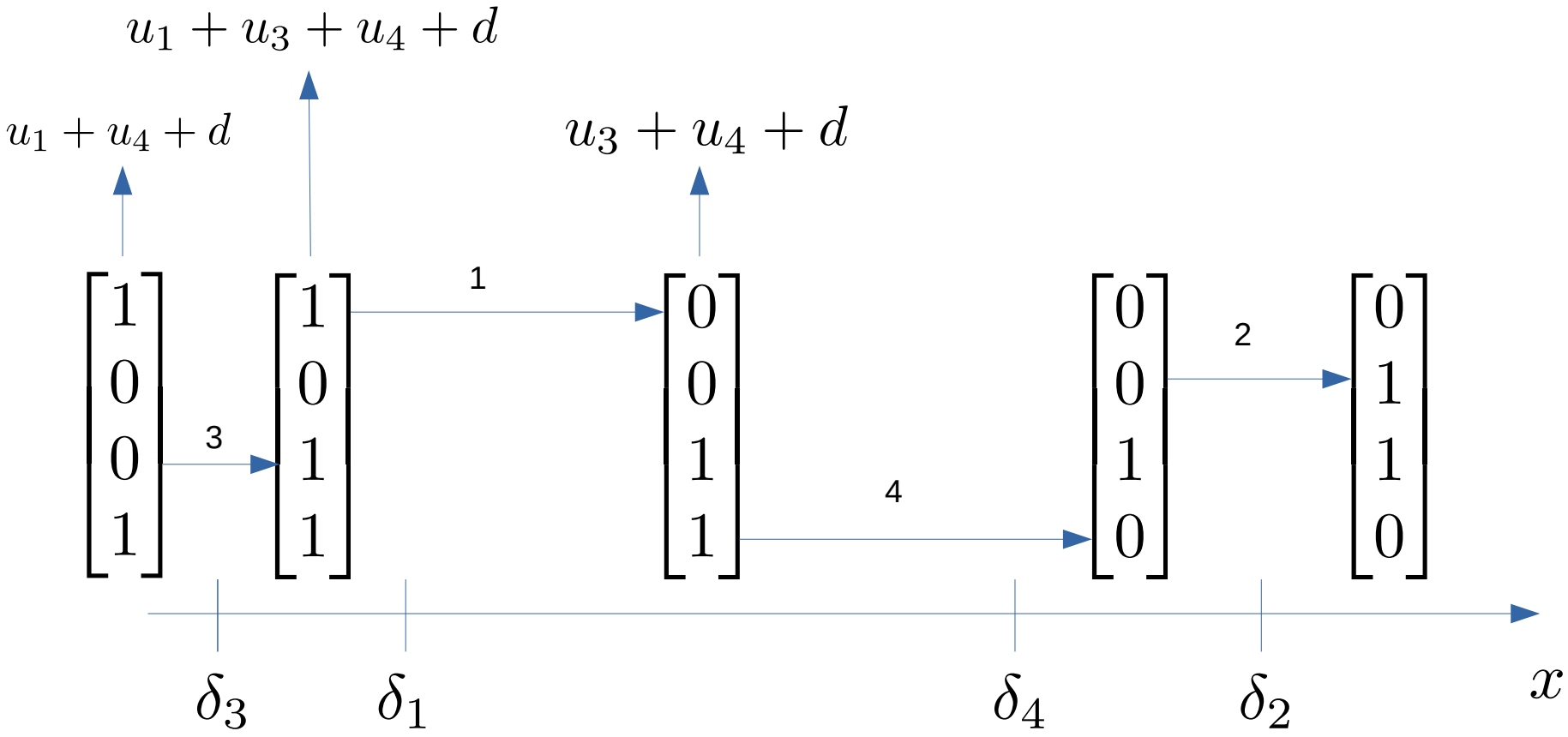


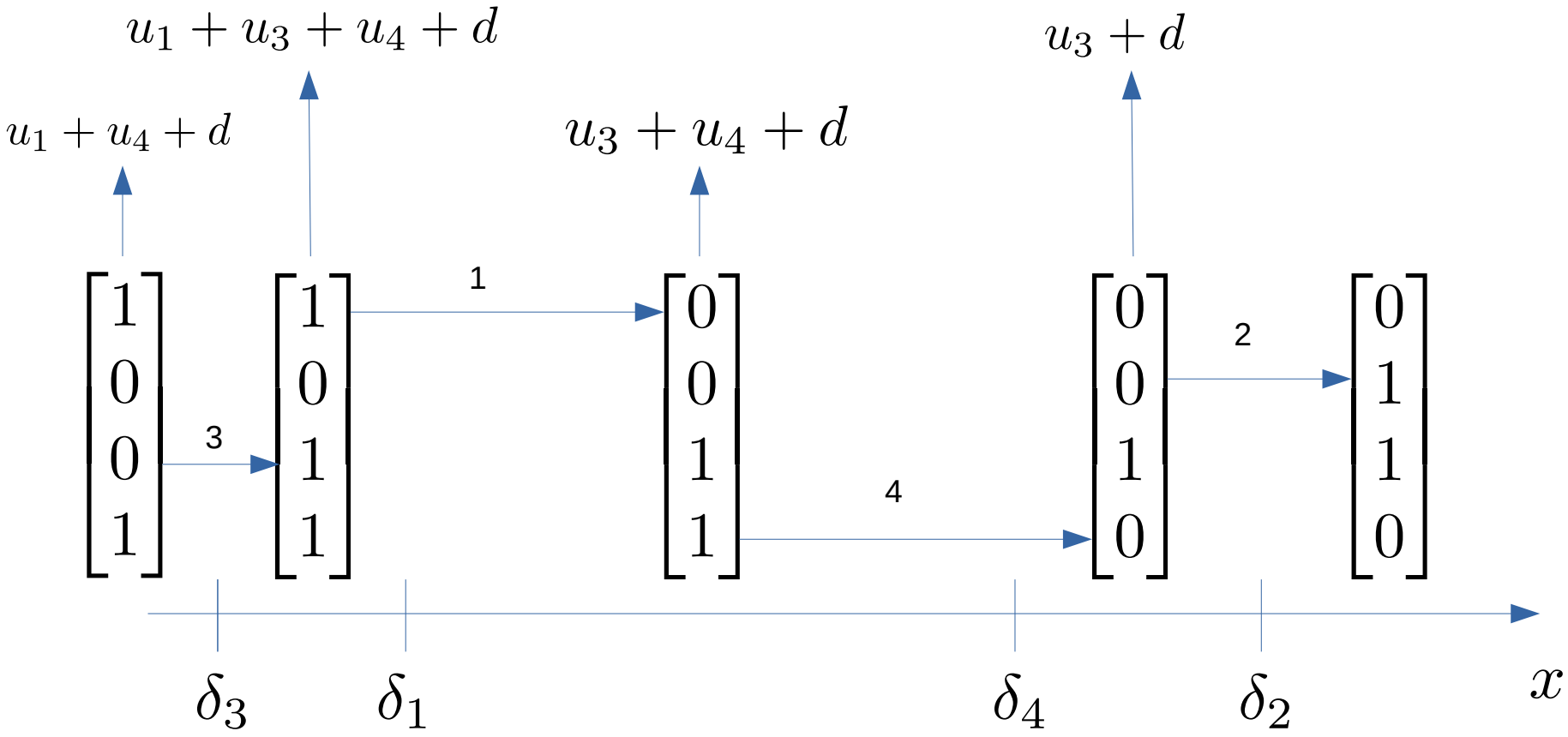
$$\begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow 1 * u_1 + 0 * u_2 + 0 * u_3 + 1 * u_4 + d = \boxed{u_1 + u_4 + d}$$

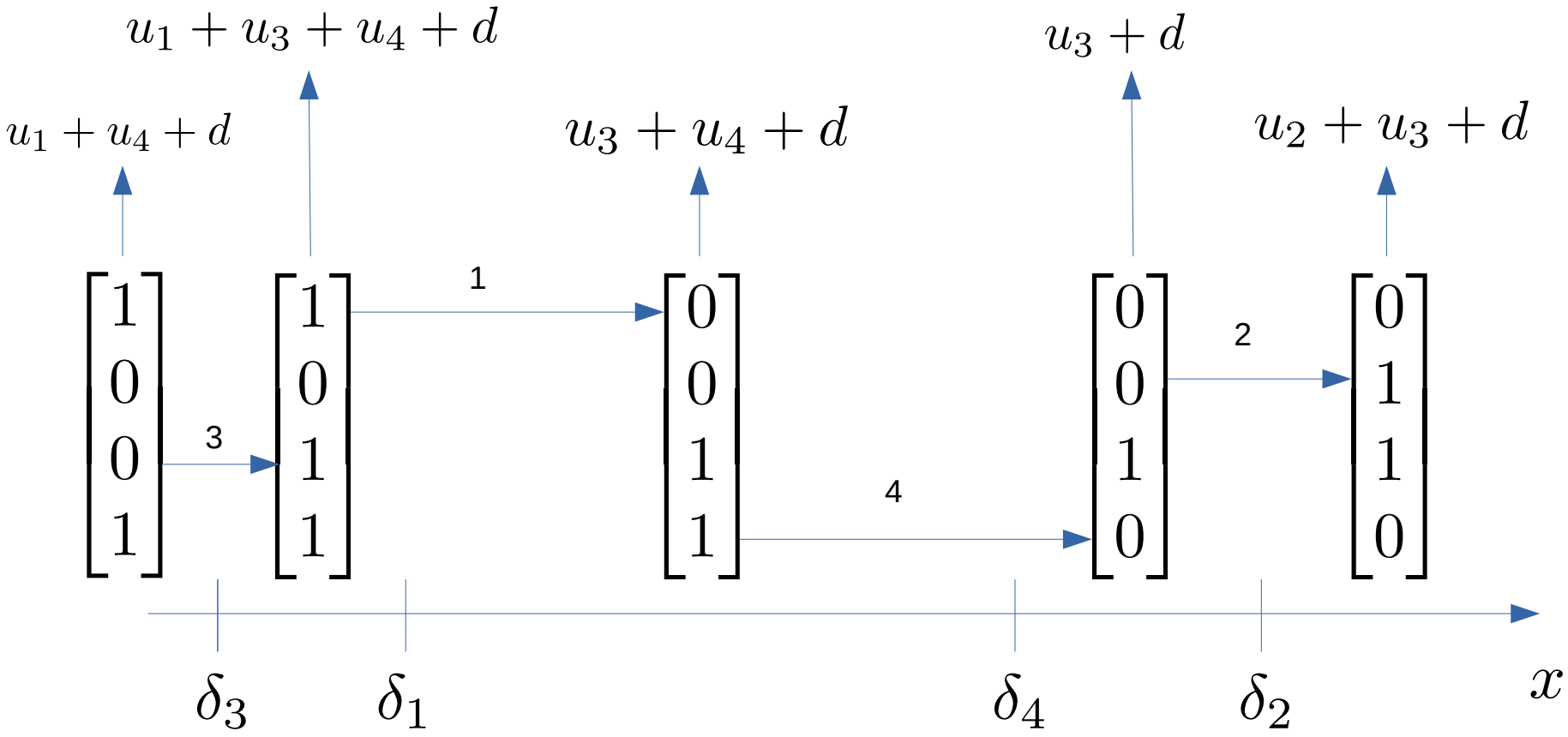

$u_1 + u_4 + d$
$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
3
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
1
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
4
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
2
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
δ_3
δ_1
δ_4
δ_2
x

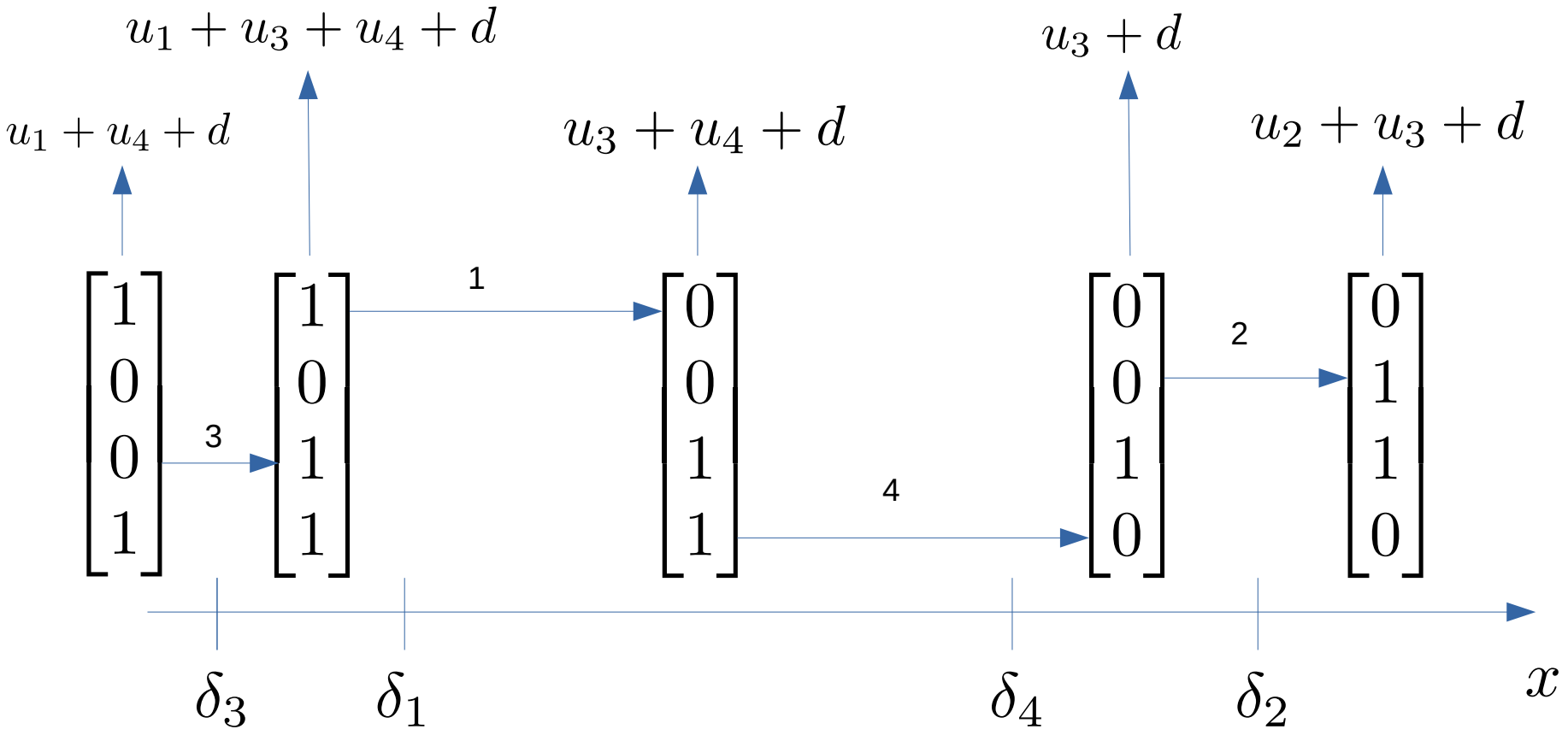

$u_1 + u_3 + u_4 + d$
$u_1 + u_4 + d$
$\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\\1\\0\end{bmatrix}$
3
1
4
2
δ_3
δ_1
δ_4
δ_2
x

Red Hat



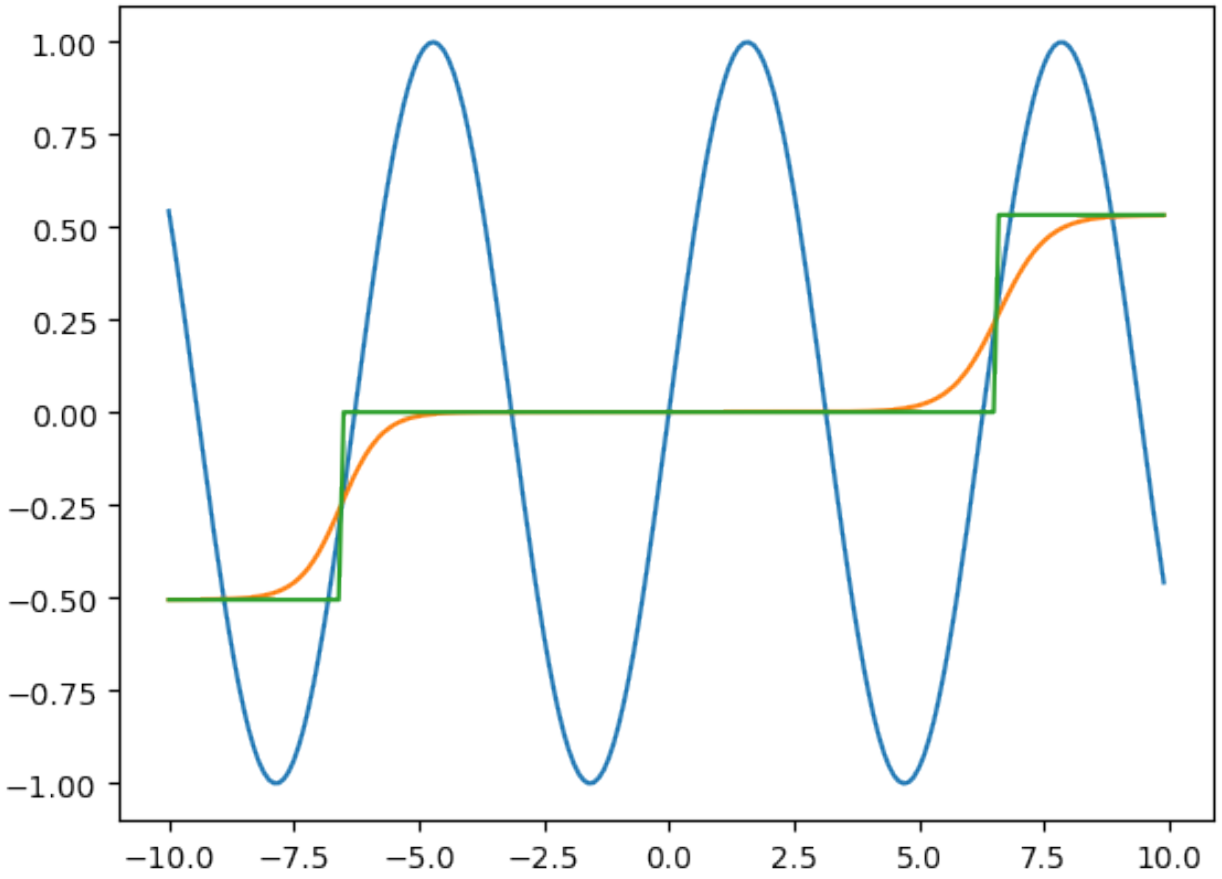



$u_1 + u_3 + u_4 + d$
$u_3 + d$
$u_1 + u_4 + d$
$u_3 + u_4 + d$
$u_2 + u_3 + d$
1
2
3
4
δ_3
δ_1
δ_4
δ_2
x

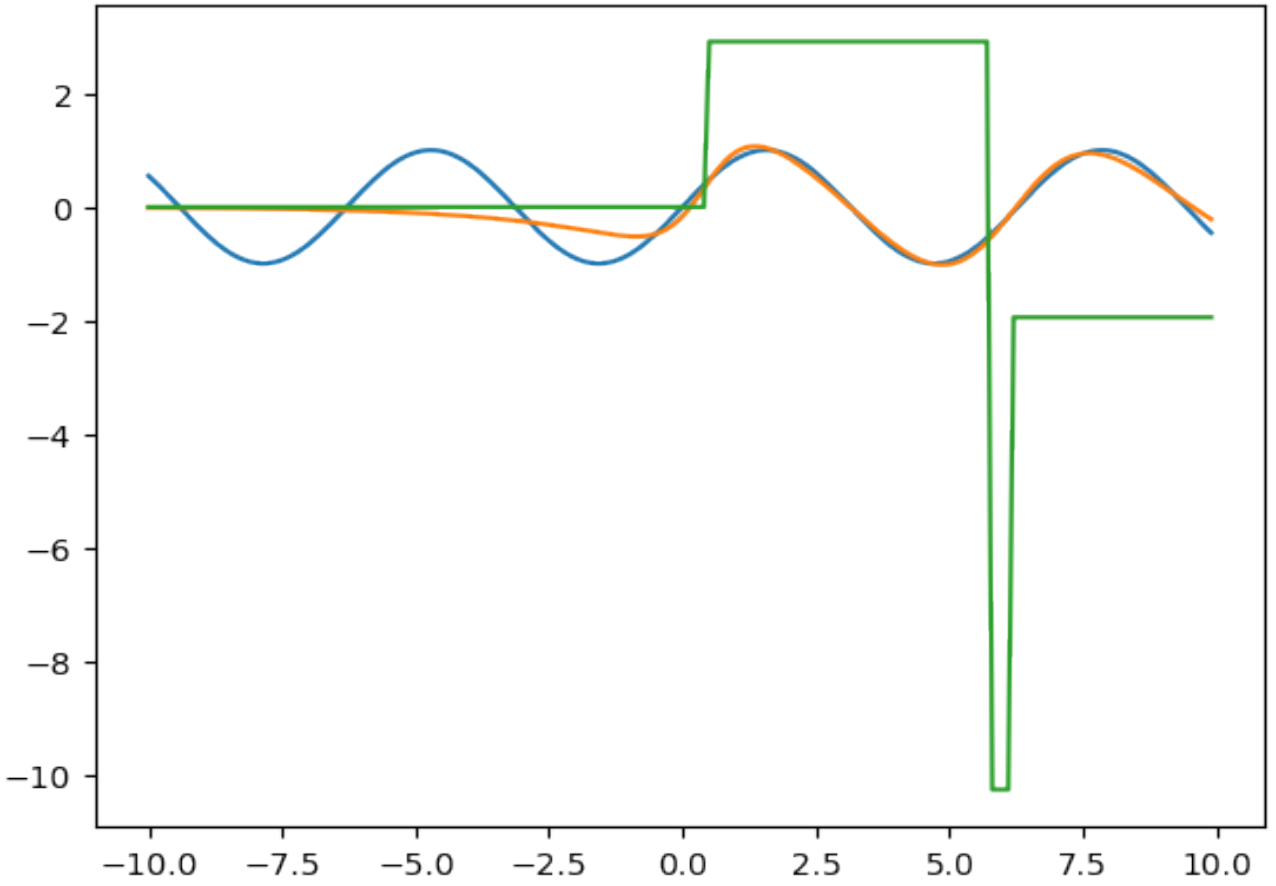


Five distinct regions where NN predicts constant values with sigmoid turn on and turn off

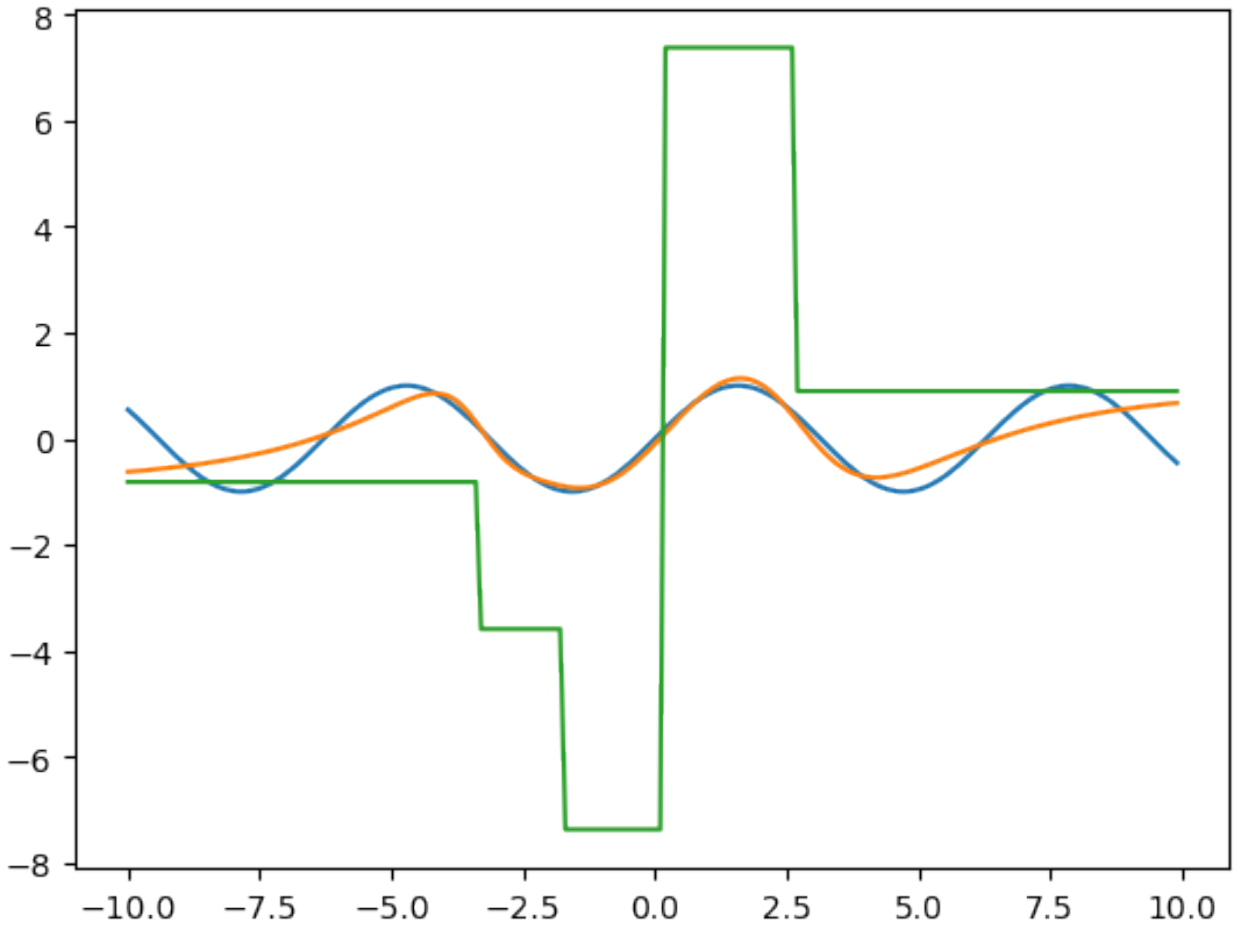
Hidden Nodes = 2



Hidden Nodes = 3

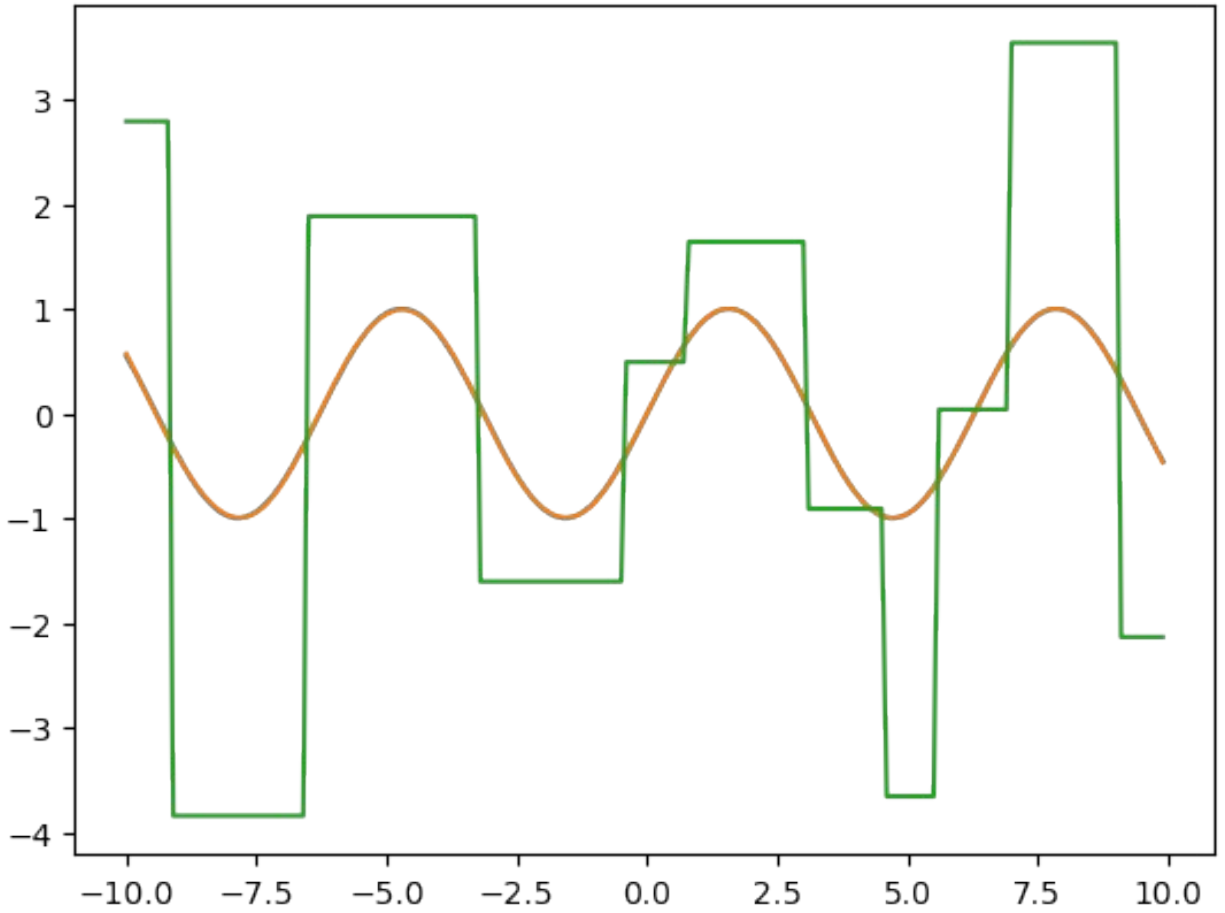


Hidden Nodes = 4

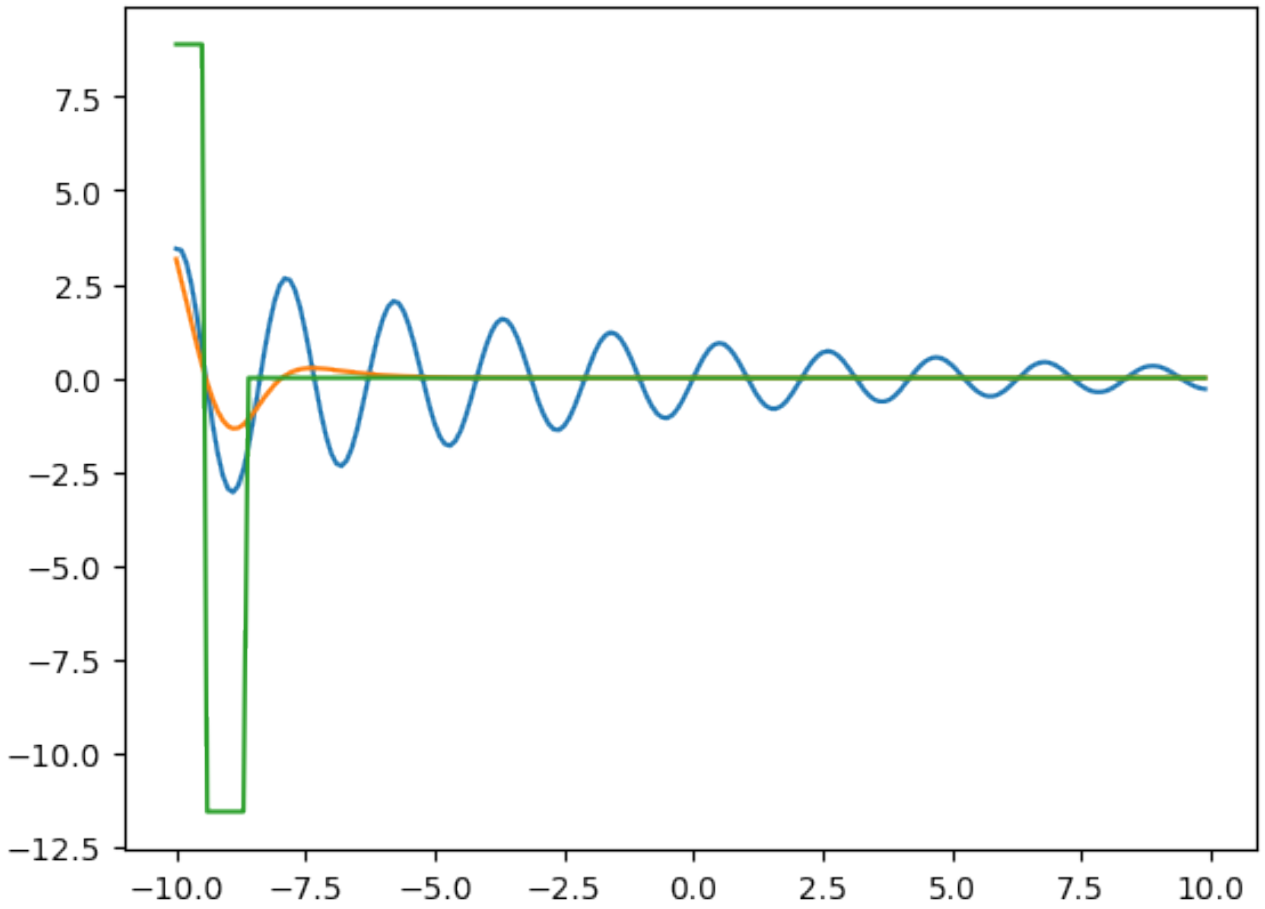


Red Hat

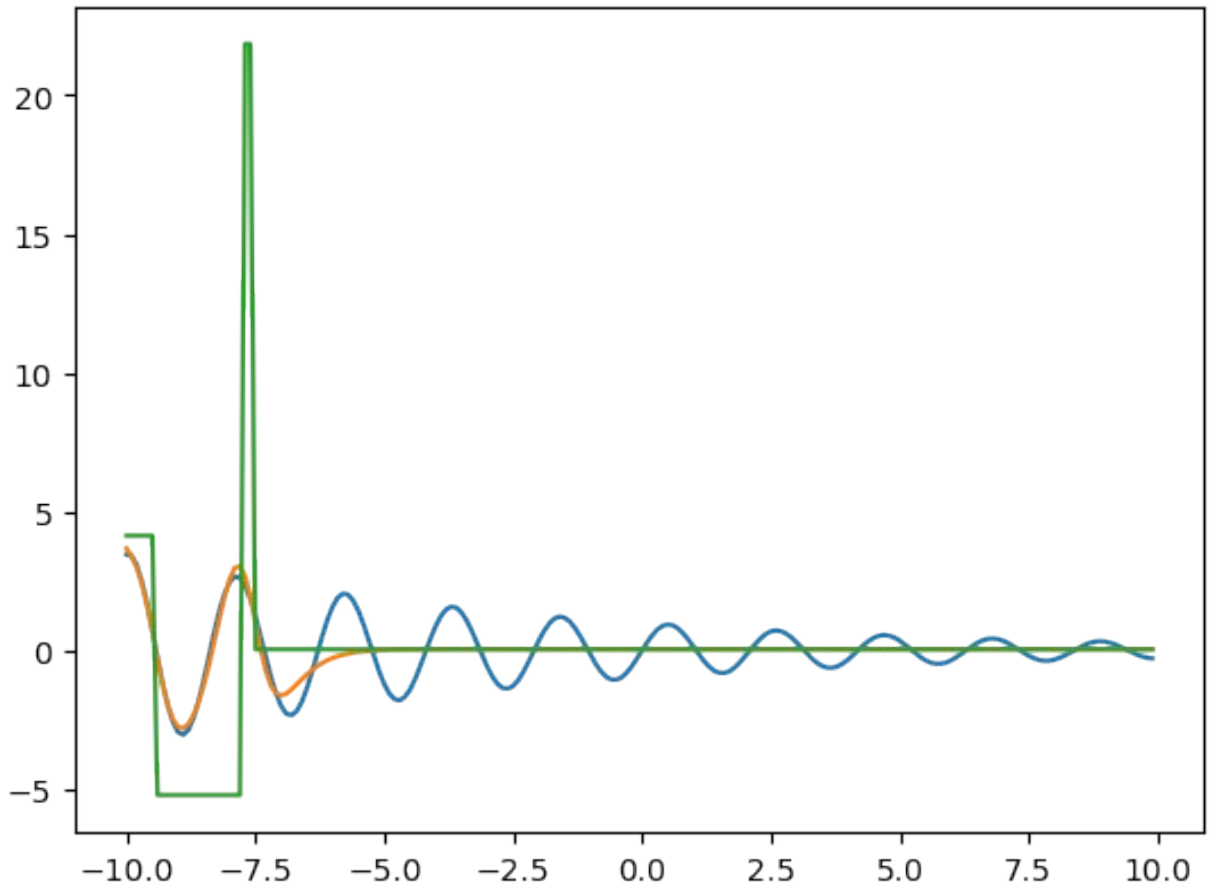
Hidden Nodes = 10



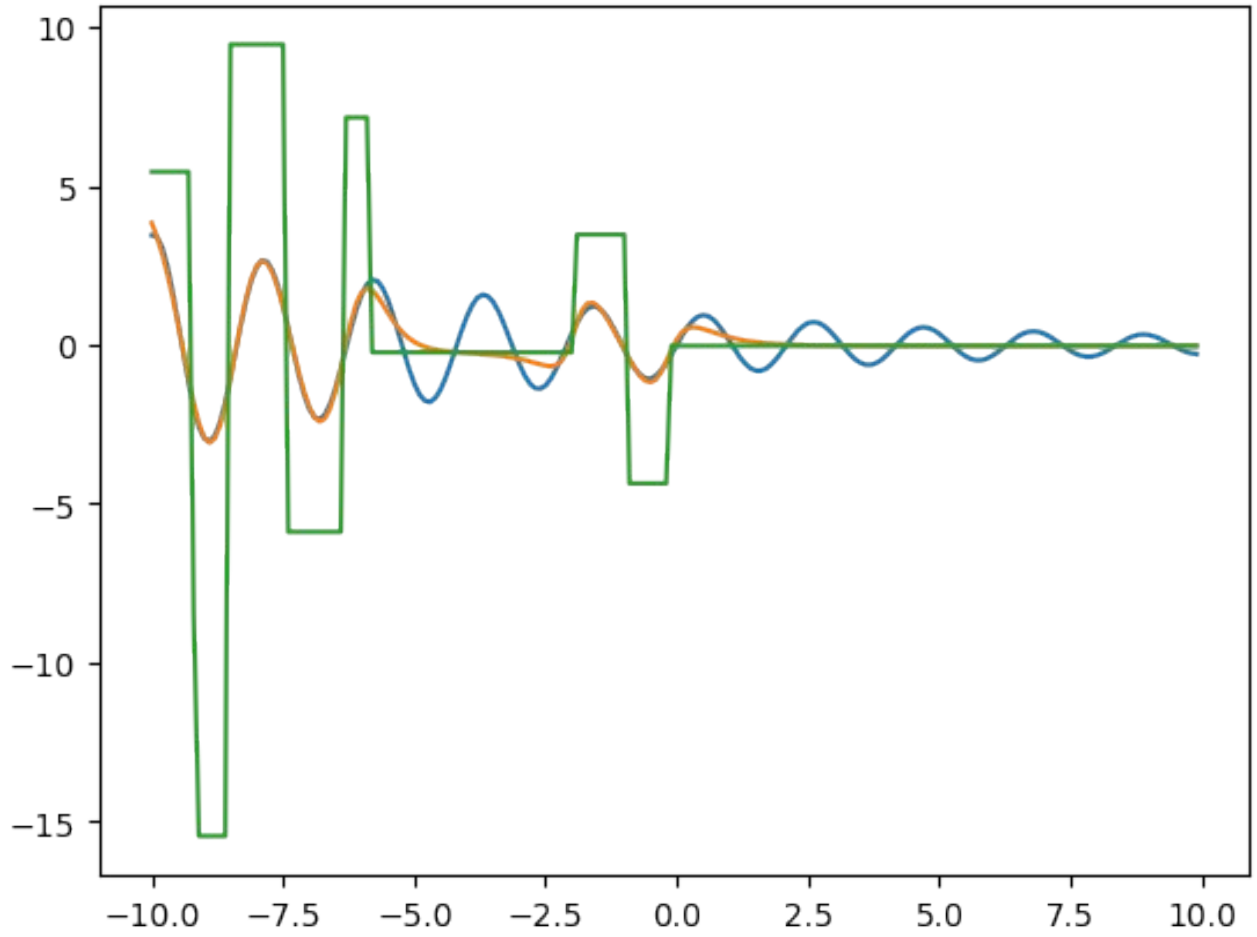
Hidden Nodes = 2



Hidden Nodes = 3

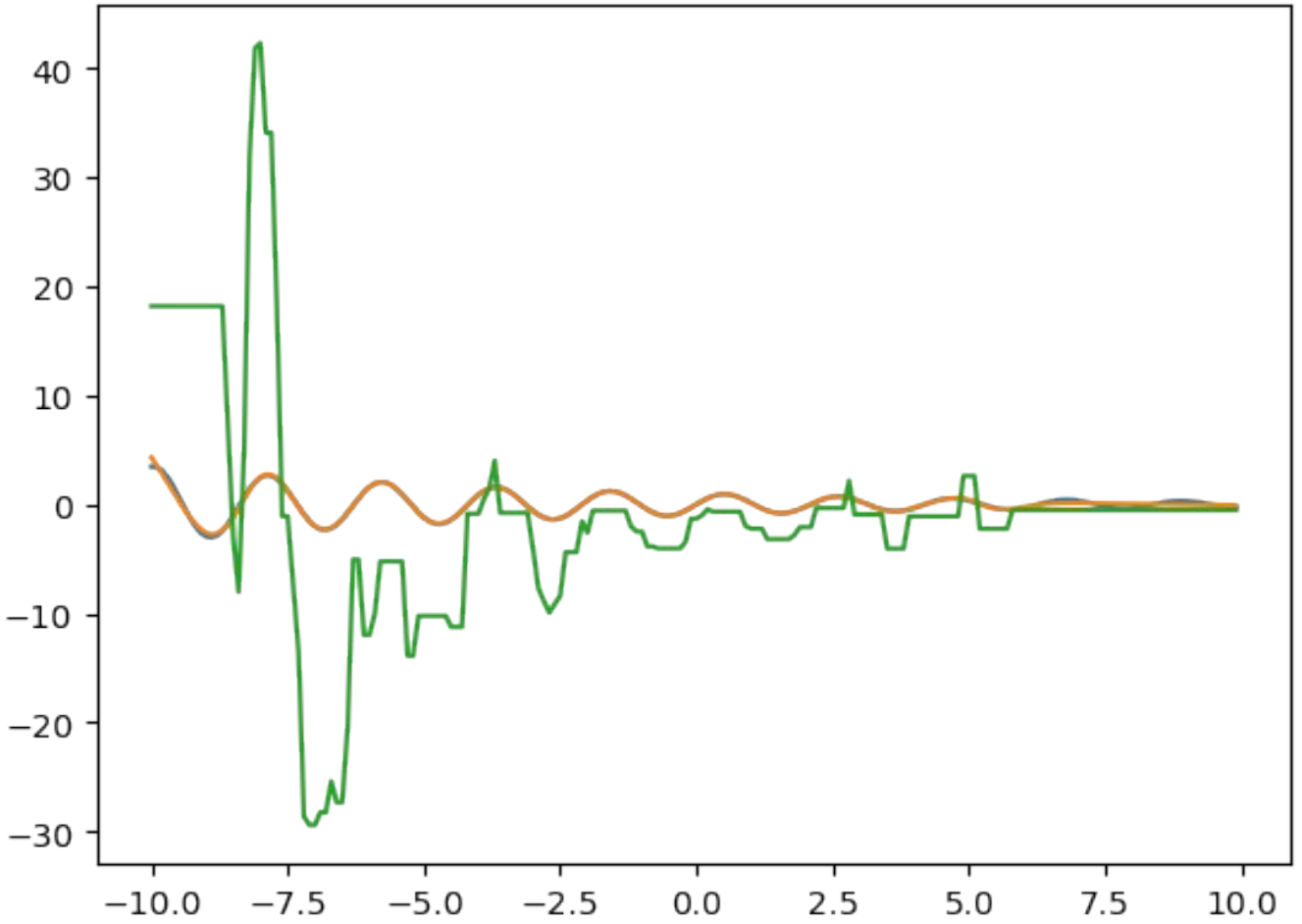


Hidden Nodes = 10



Red Hat

Hidden Nodes = 100

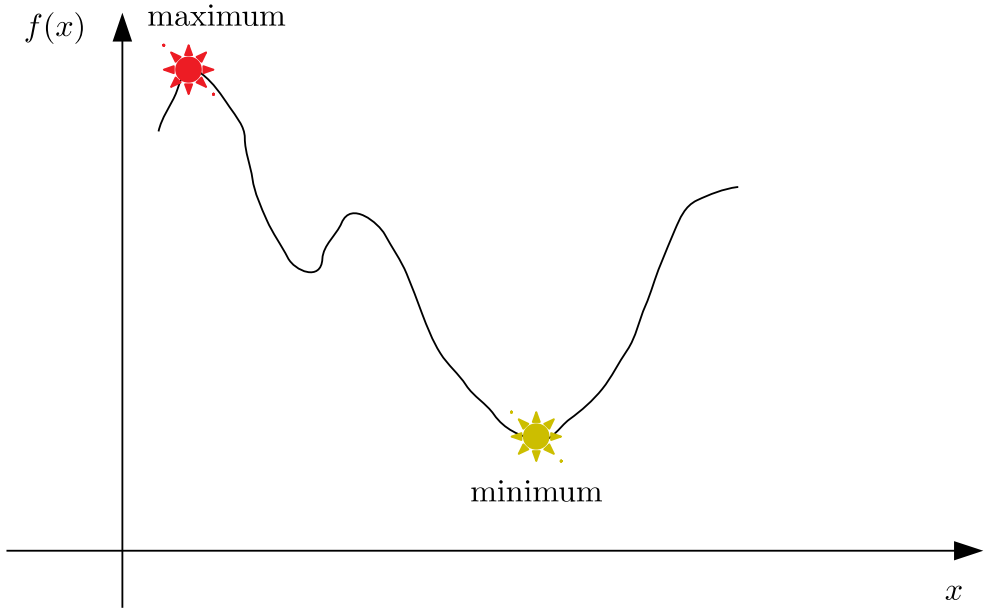


Red Hat

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

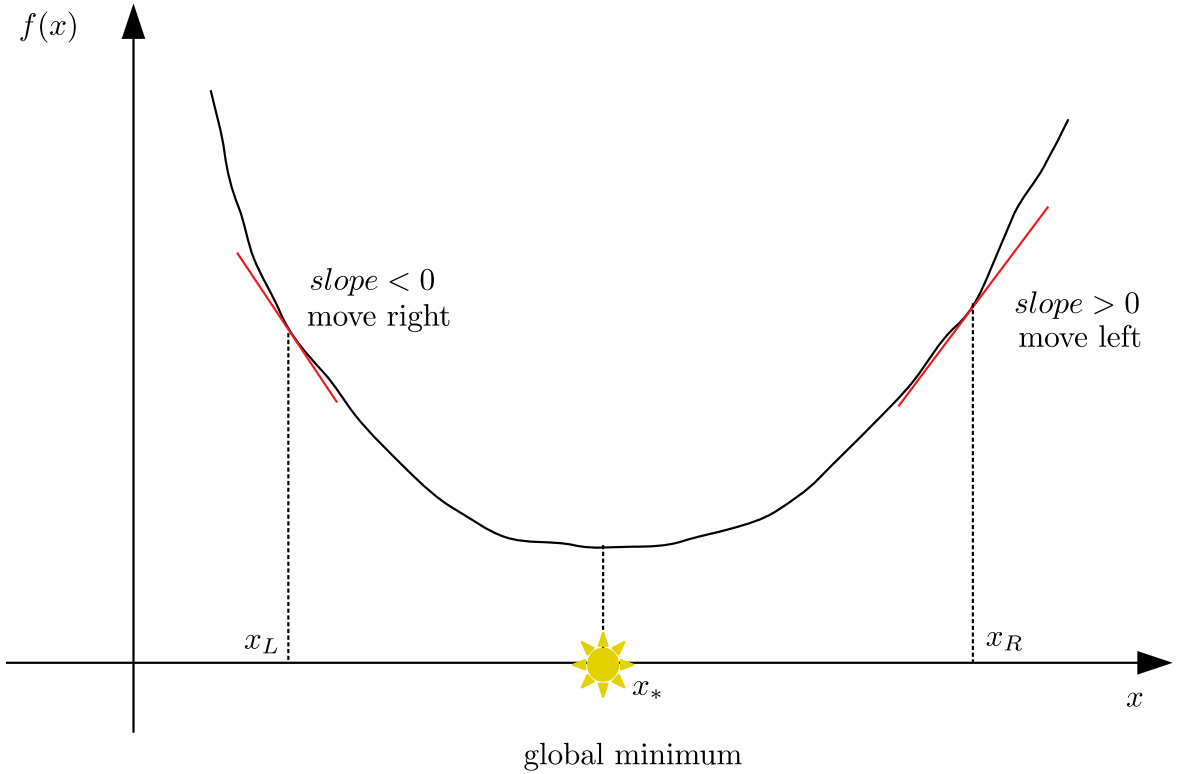
Training: Finding weights that minimize the loss function on the test set (“out of sample”) for a neural network is an optimization problem

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

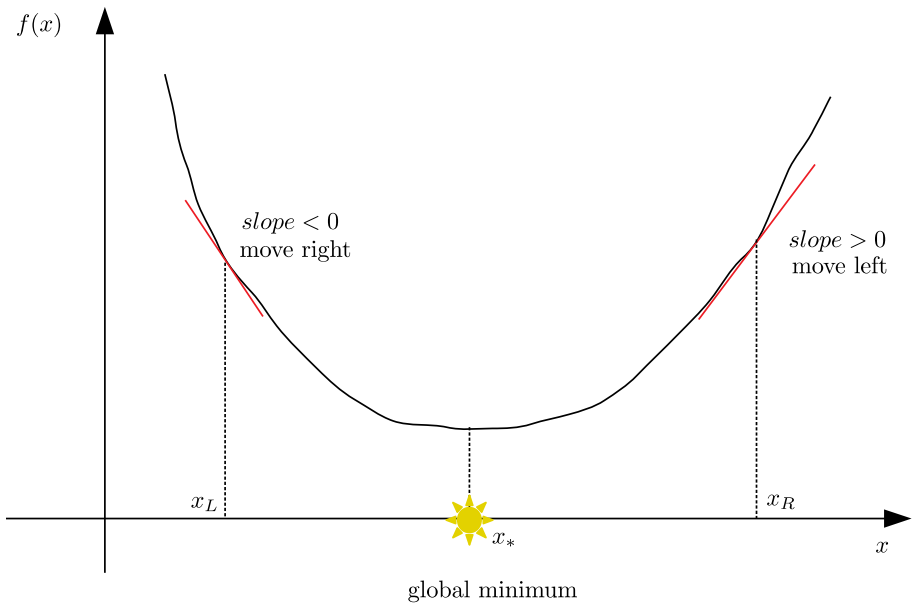
- f might be computationally expensive to evaluate.
- f might be discrete so no way to evaluate derivatives which can guide search for minima
- Even if f is continuous and well-behaved, evaluating higher derivatives (second, third etc.) is very expensive.
- f might have very complex structure with multiple (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

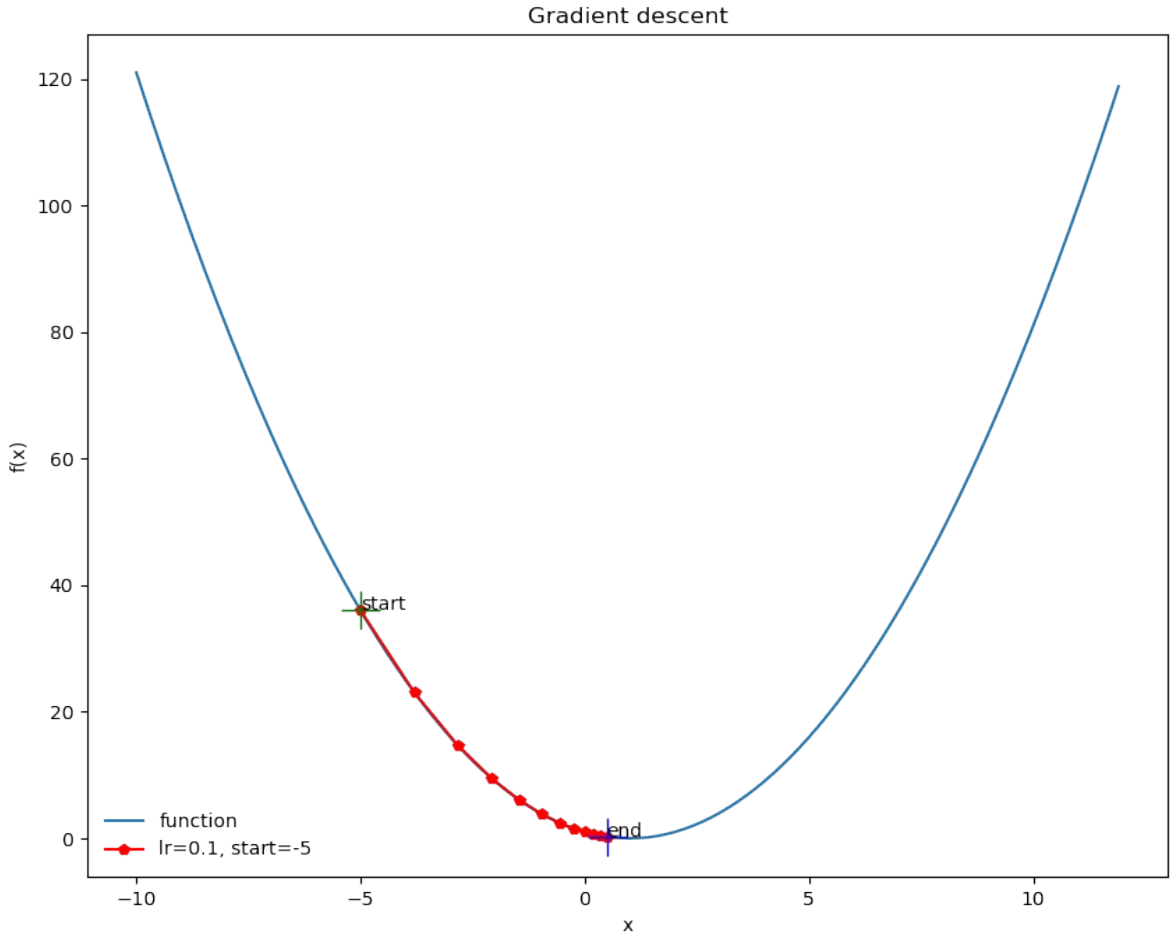
Start at some reasonable point and keep updating

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

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

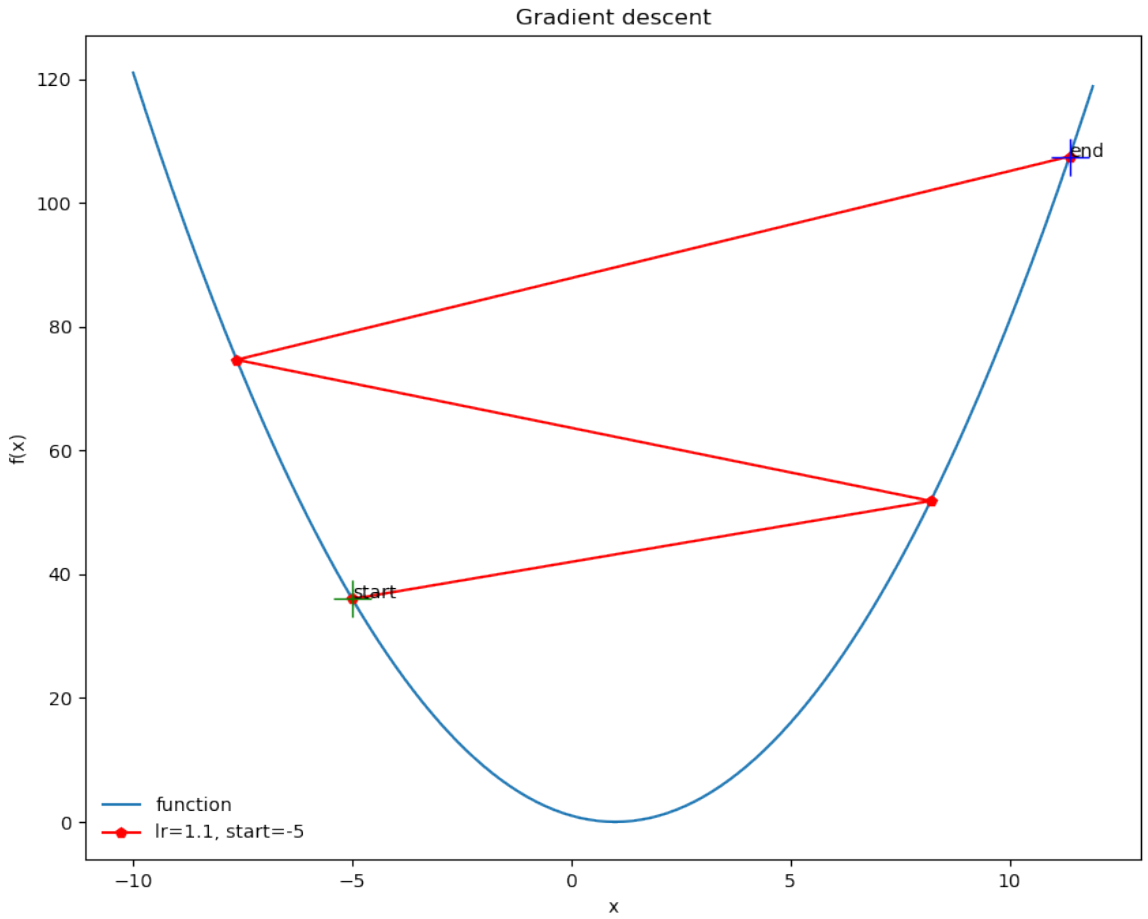
$$\eta = \text{learning rate}$$

Intuition



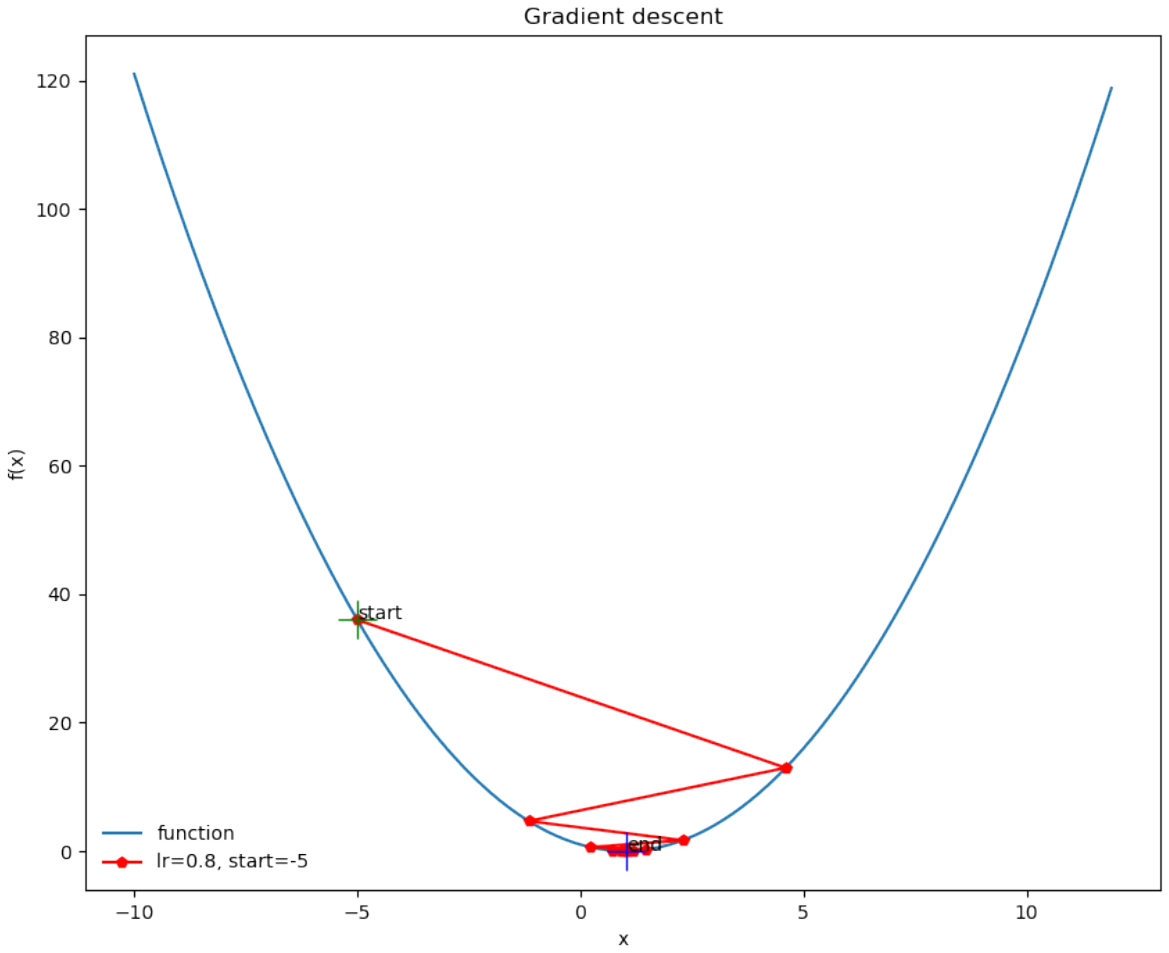
Red Hat

Intuition



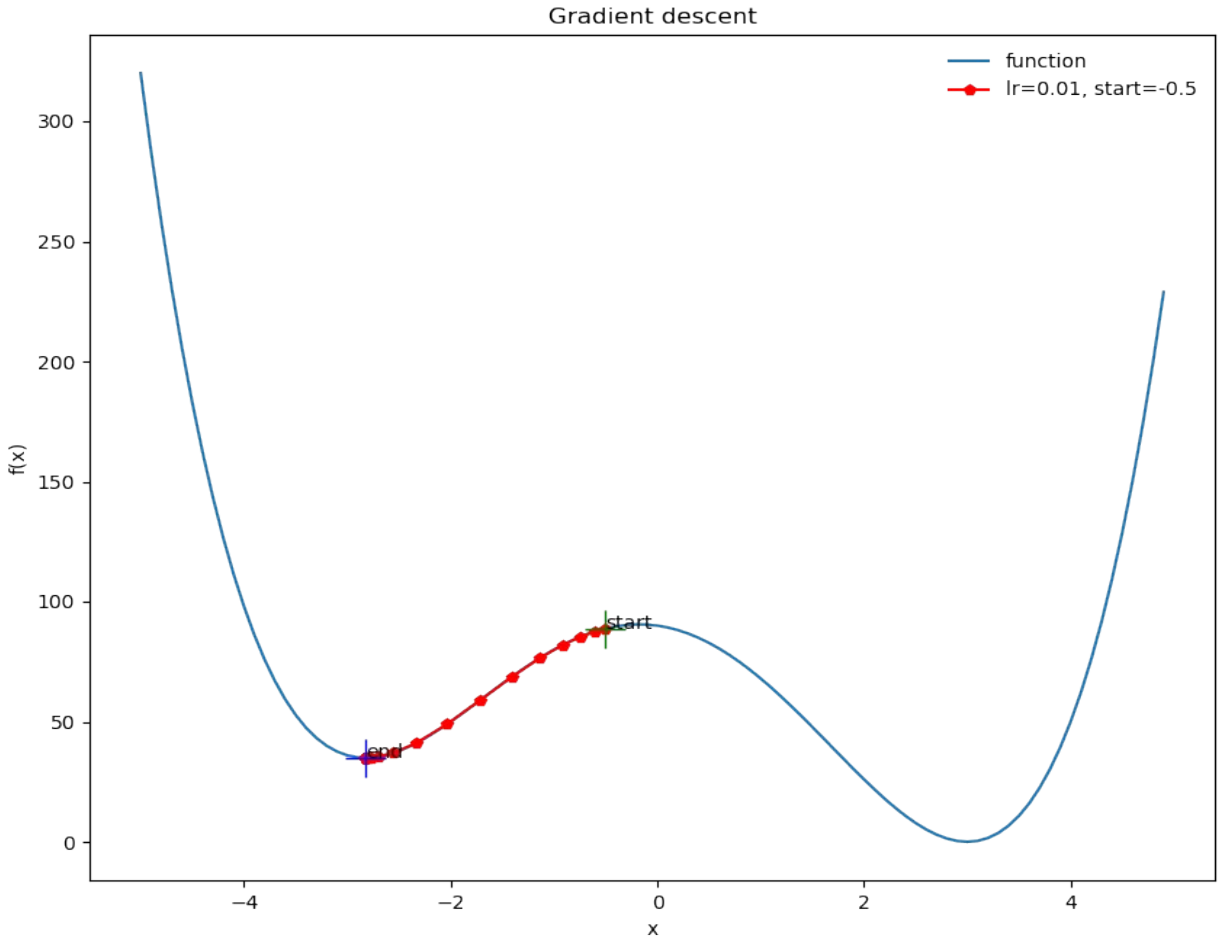
Red Hat

Intuition



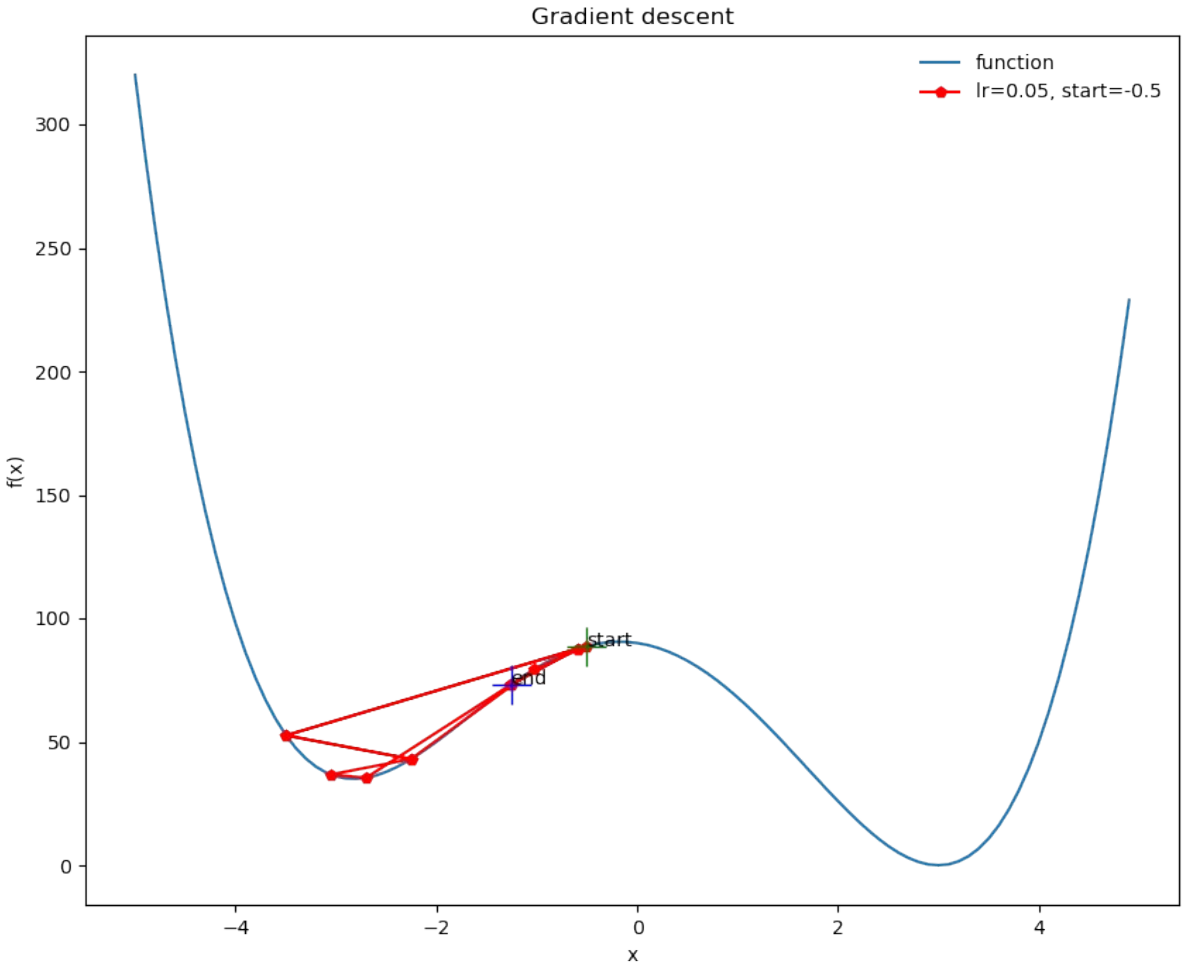
Red Hat

Intuition



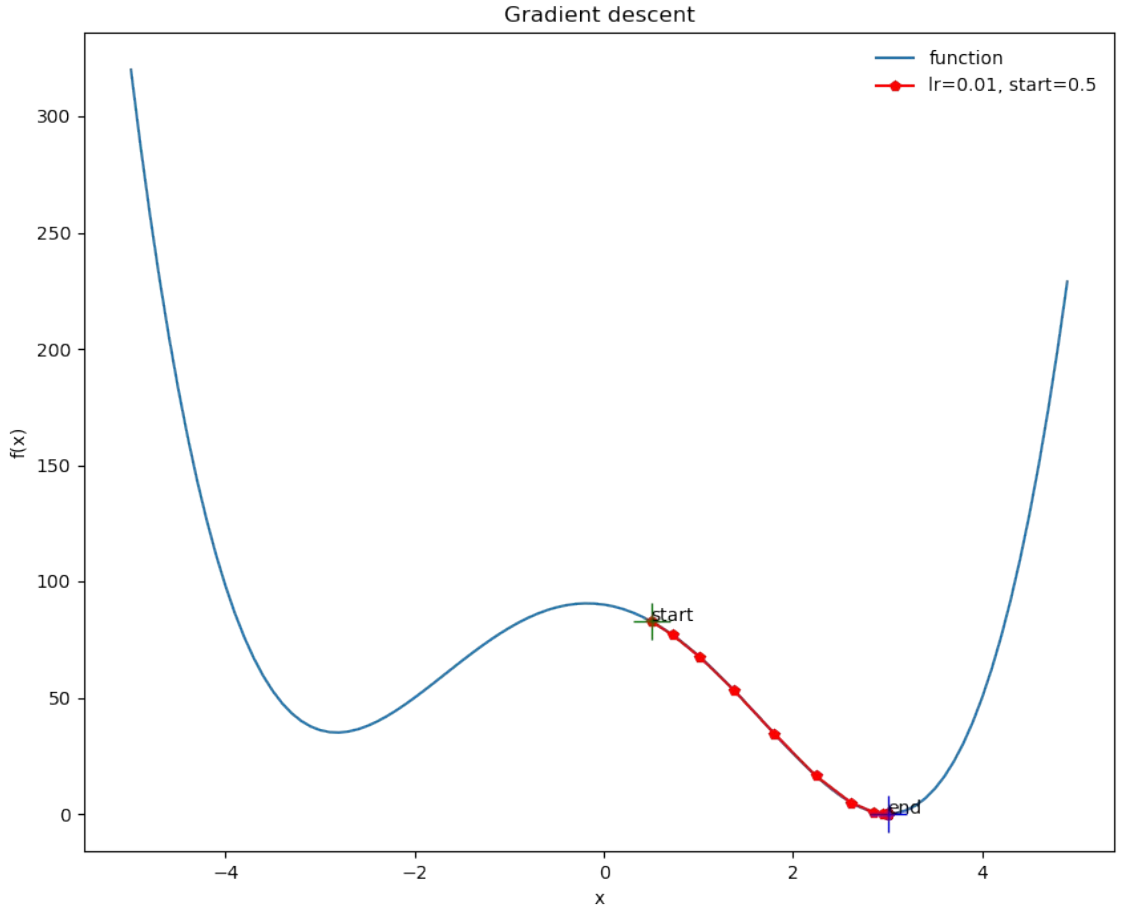
Red Hat

Intuition



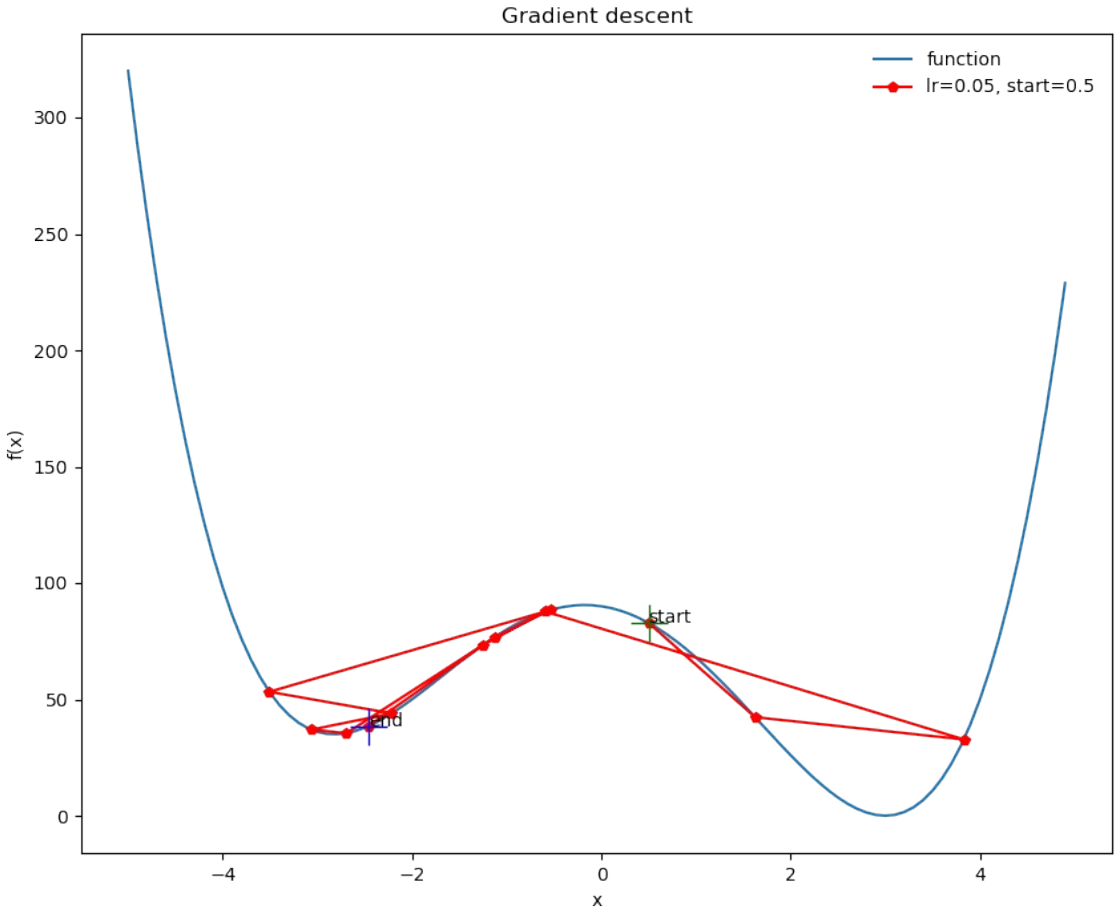
Red Hat

Intuition



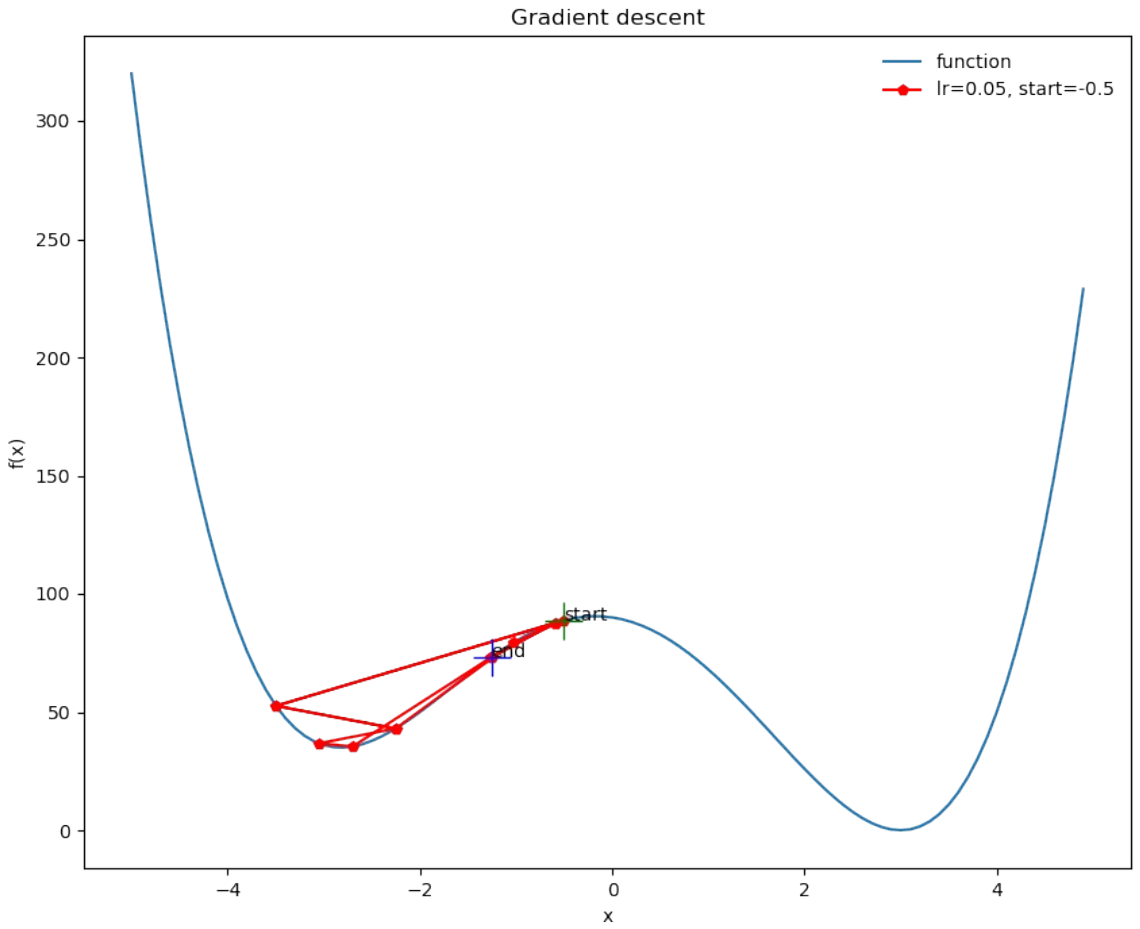
Red Hat

Intuition



Red Hat

Intuition



Red Hat

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