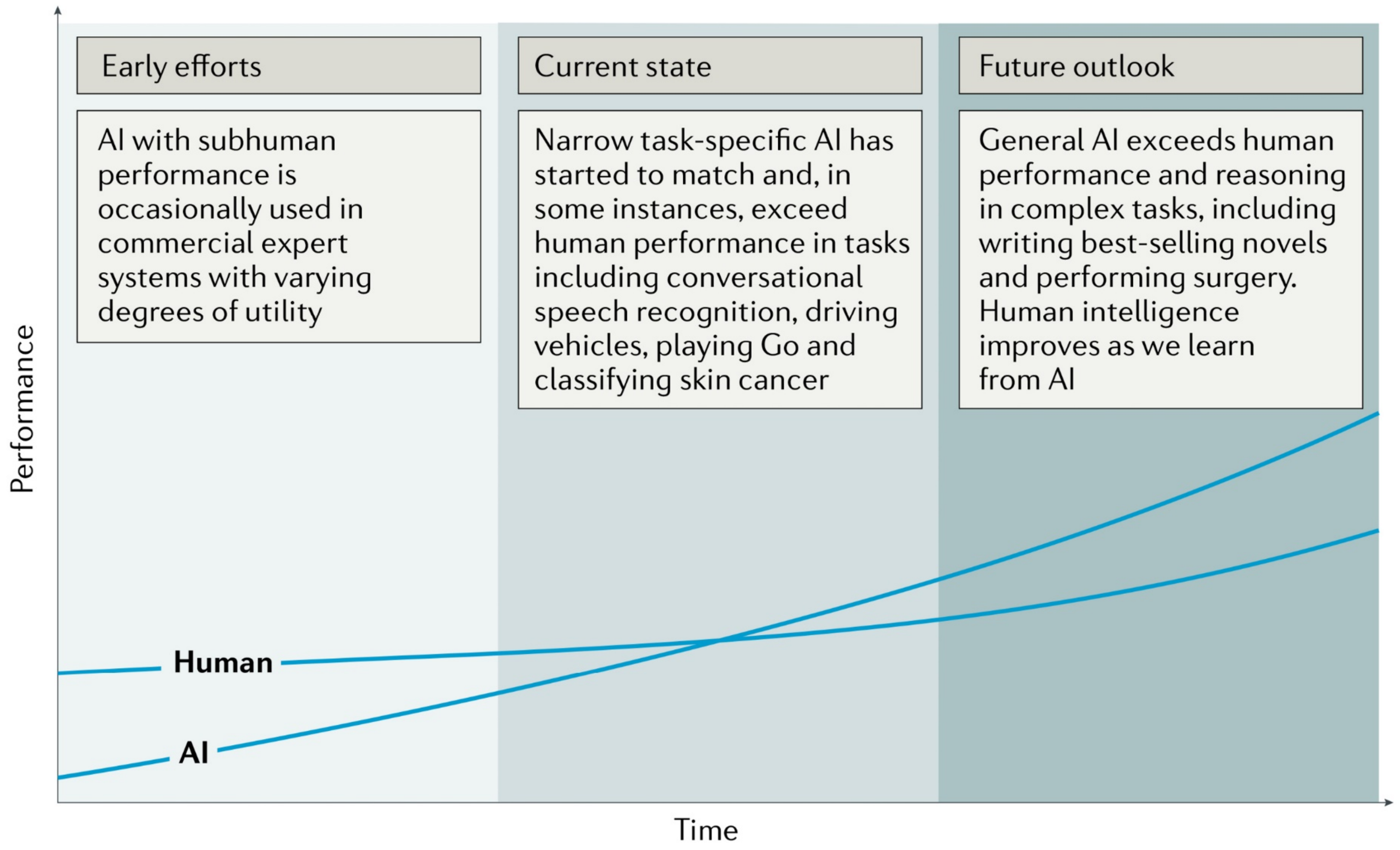
Deep Learning Intuition

HARVARD MEDICAL SCHOOL | BRIGHAM AND WOMEN'S HOSPITAL | DANA-FARBER CANCER INSTITUTE

Ahmed Hosny

Deep Learning

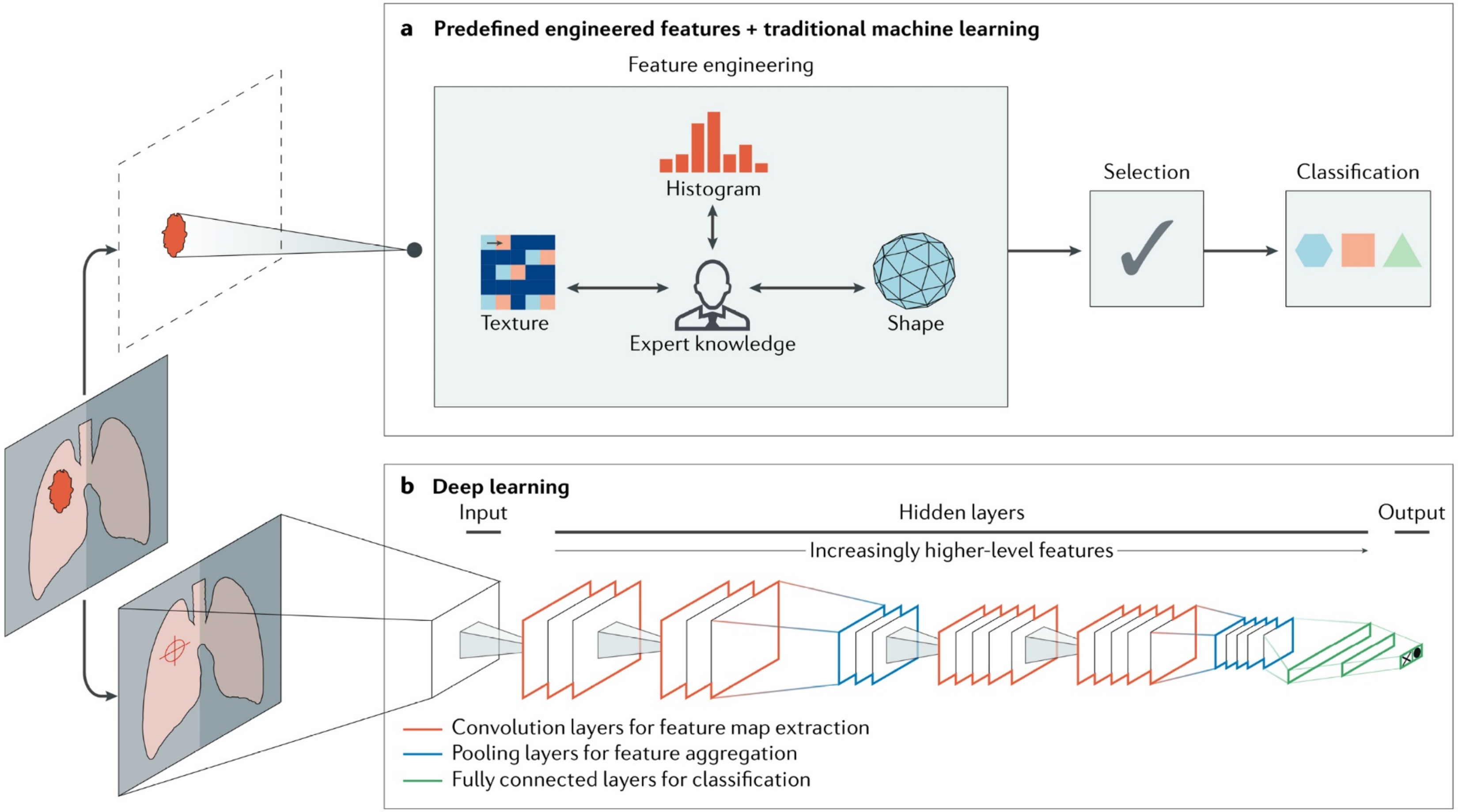


Ahmed Hosny , Chintan Parmar, John Quackenbush , Lawrence H Schwartz and Hugo JWL Aerts

Artificial Intelligence in Radiology

Nature Reviews Cancer - 2018

Deep Learning



Ahmed Hosny , Chintan Parmar, John Quackenbush , Lawrence H Schwartz and Hugo JWL Aerts

Artificial Intelligence in Radiology

Nature Reviews Cancer - 2018

What is the intuition behind neural networks?

How do neural networks learn?

How to train neural networks?

What is the intuition behind neural networks?

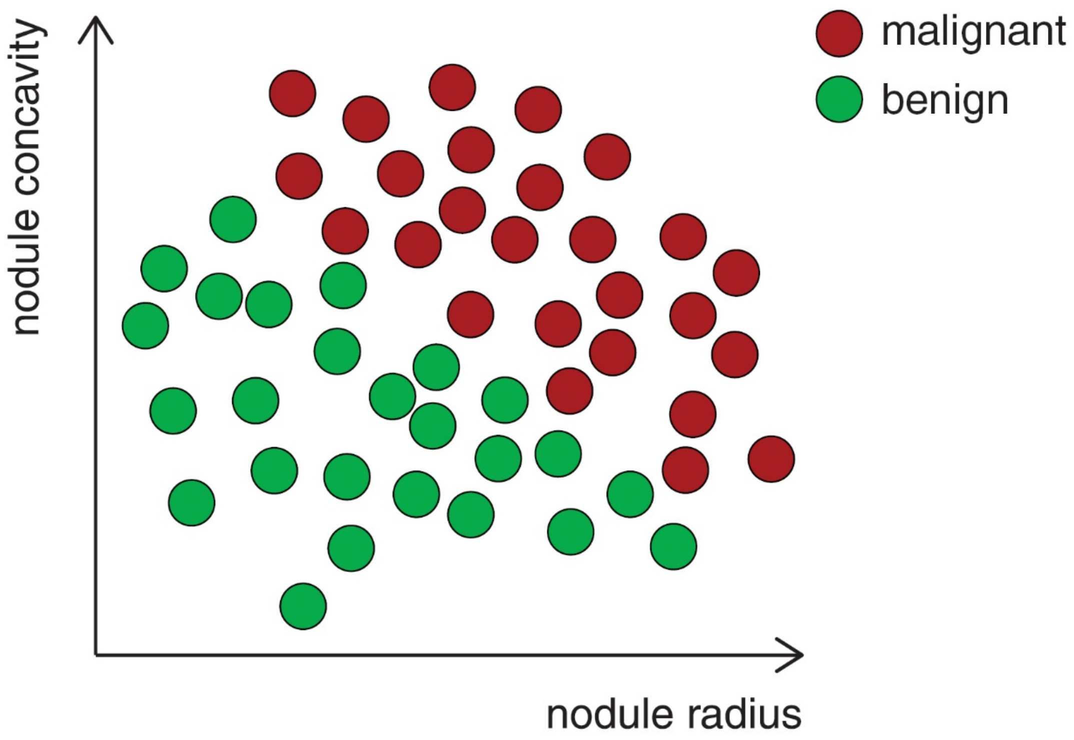
How do neural networks learn?

How to train neural networks?

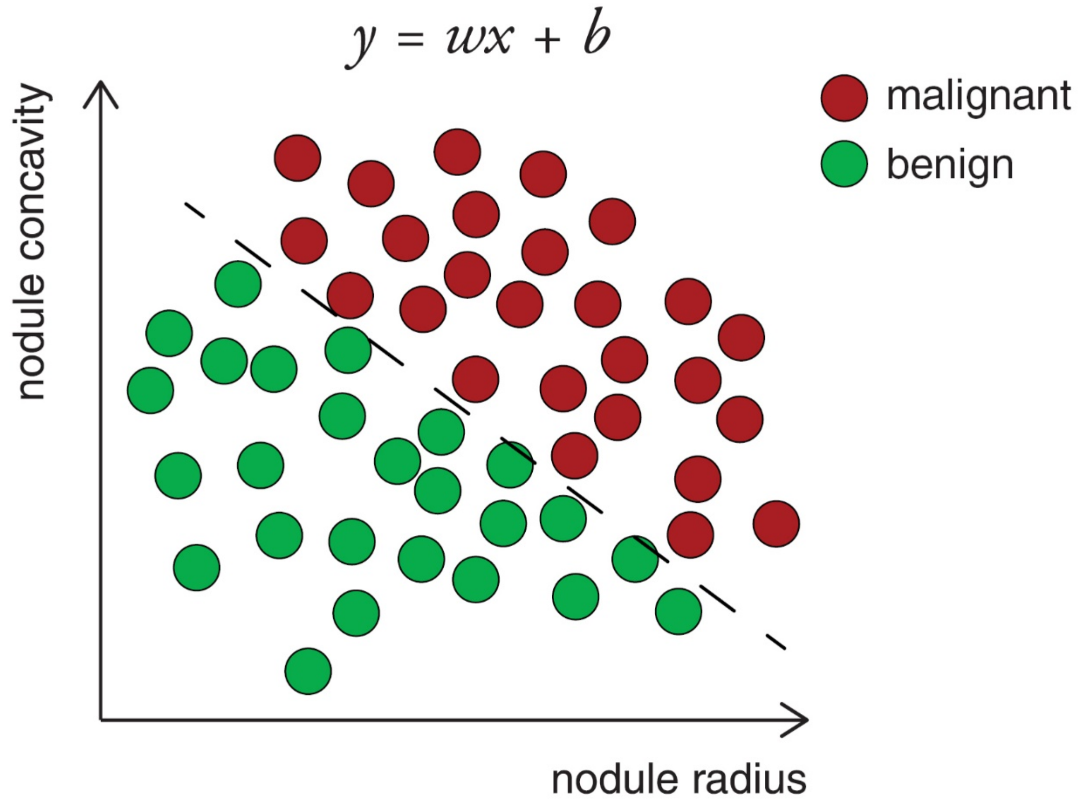
Machine Learning: 4 Main Components

- Data
- Model/Representation
- Cost/Error/Loss of model
- Model Optimizer

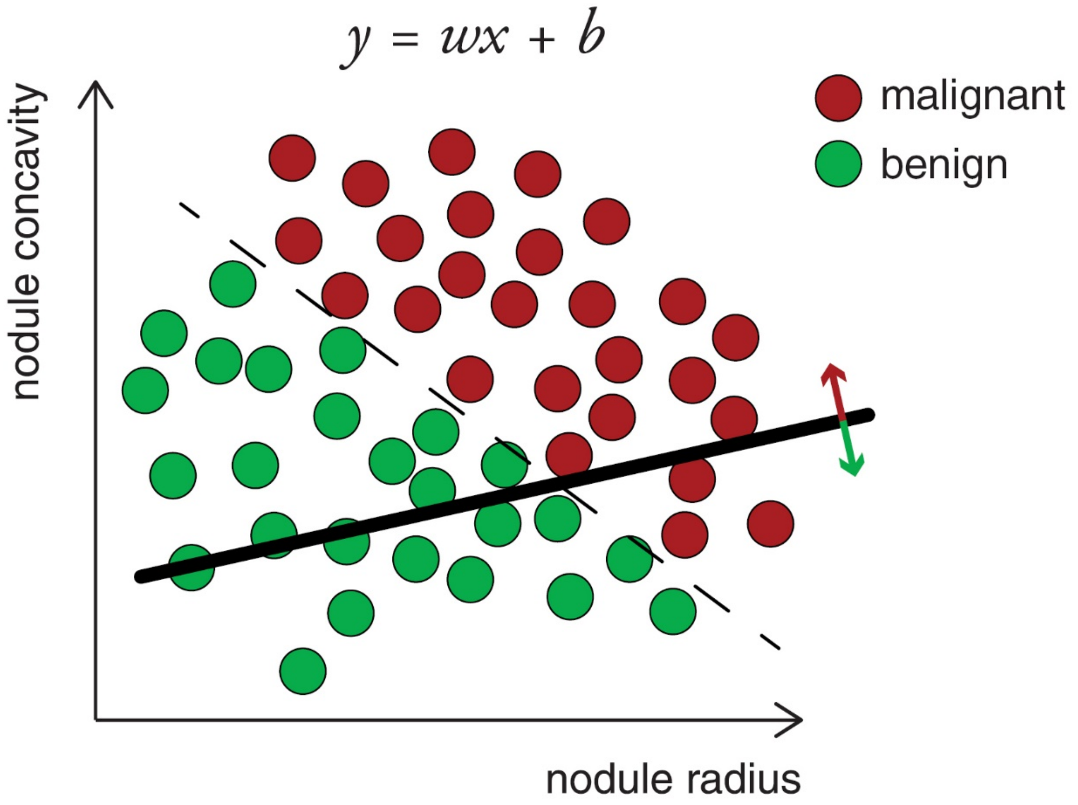
Logistic Regression



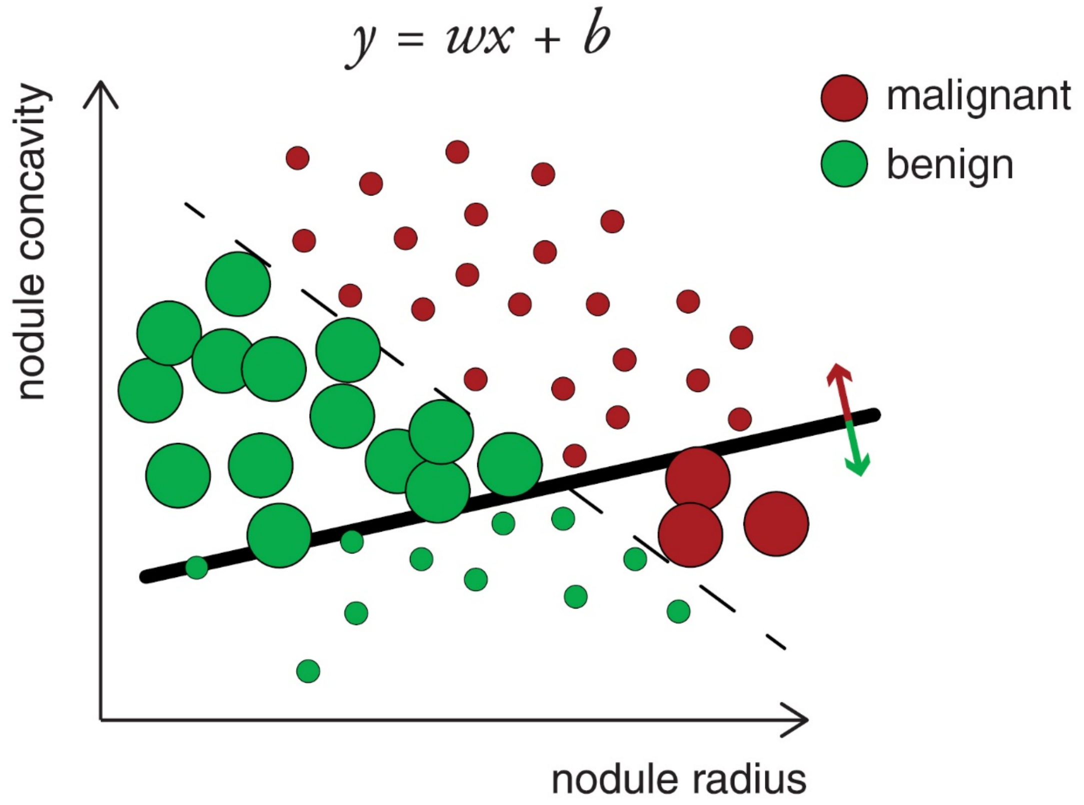
Logistic Regression



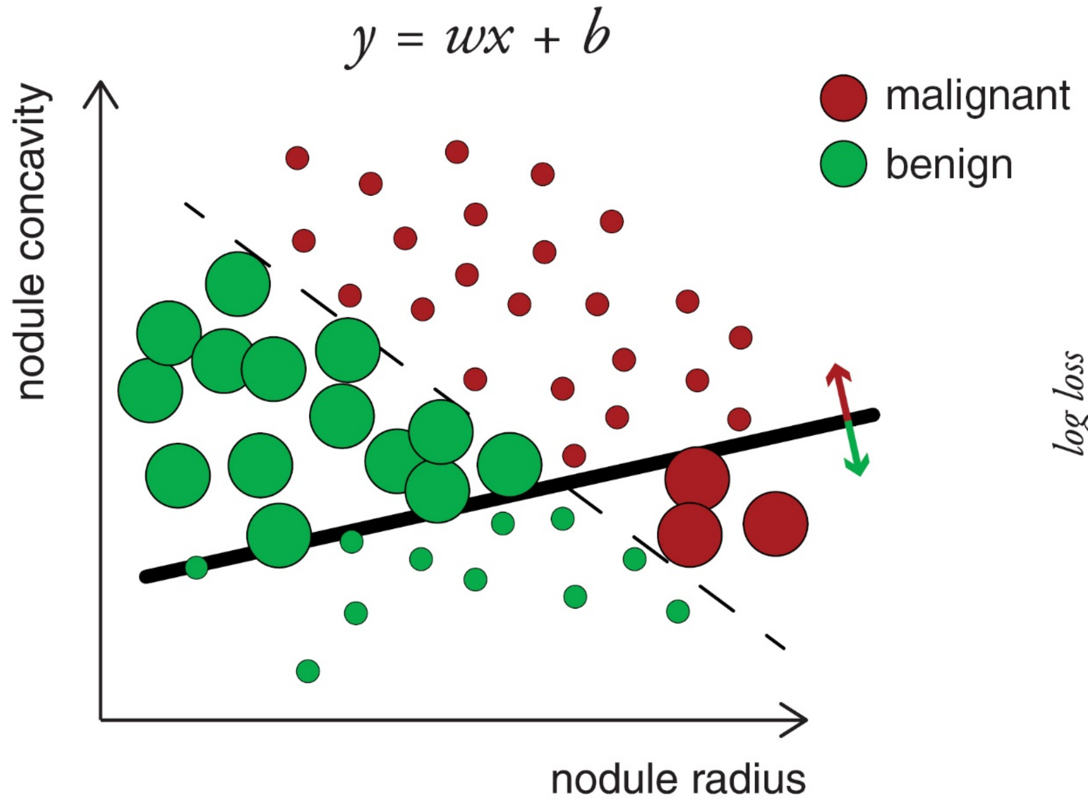
Logistic Regression

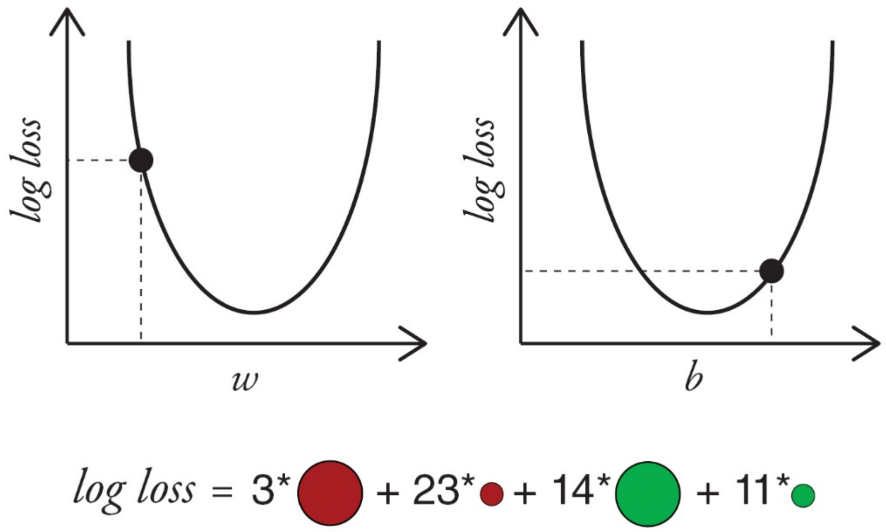


Logistic Regression

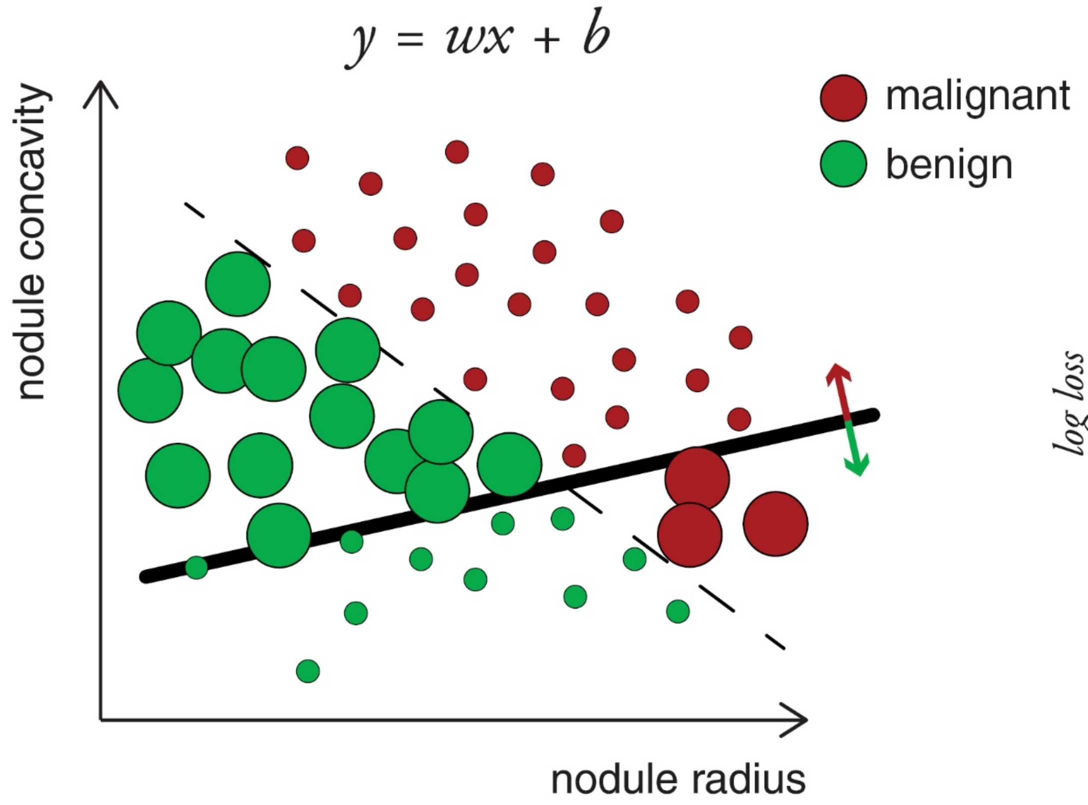


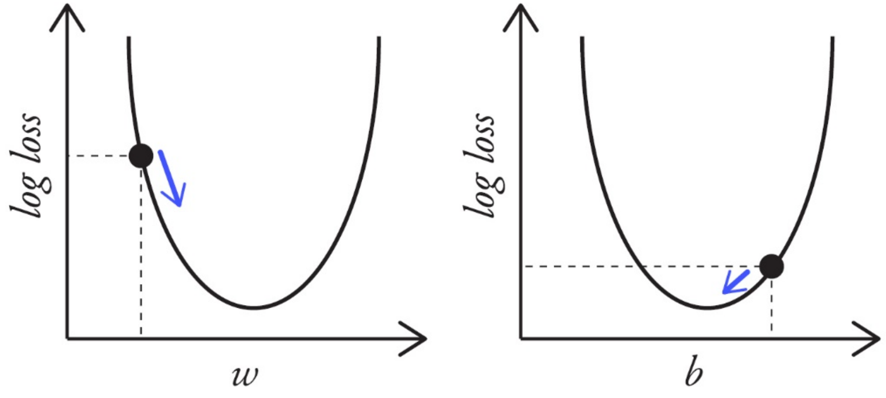
Logistic Regression





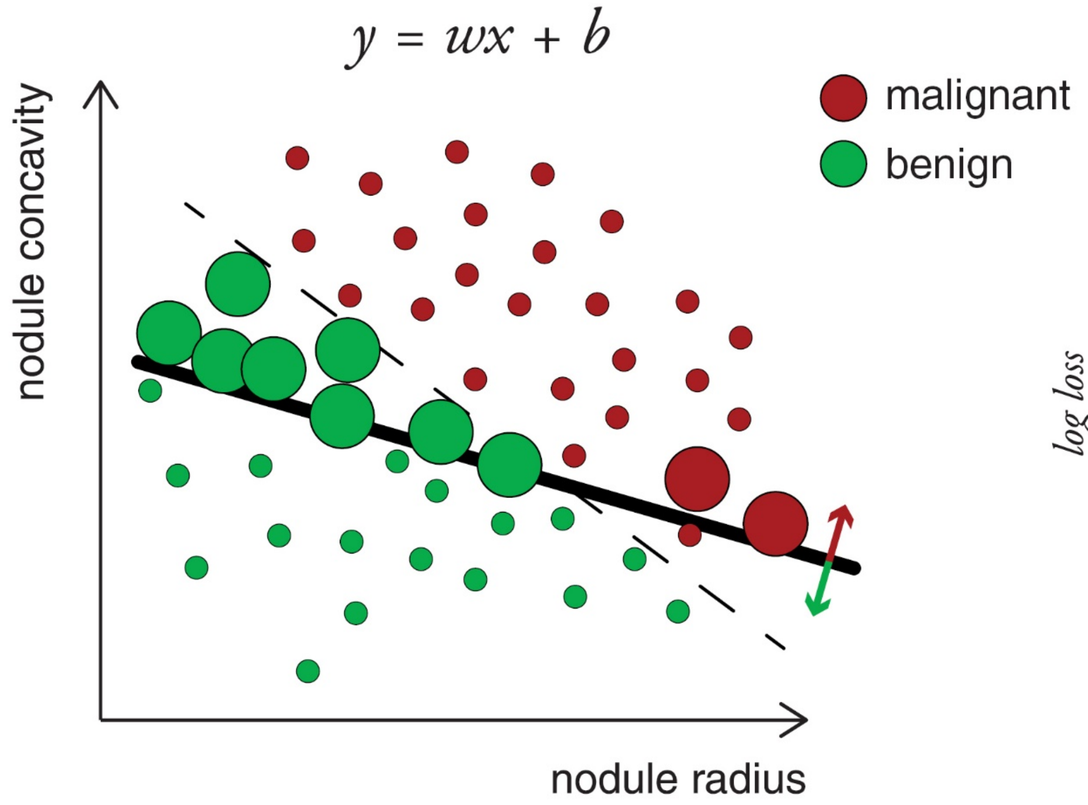
Logistic Regression

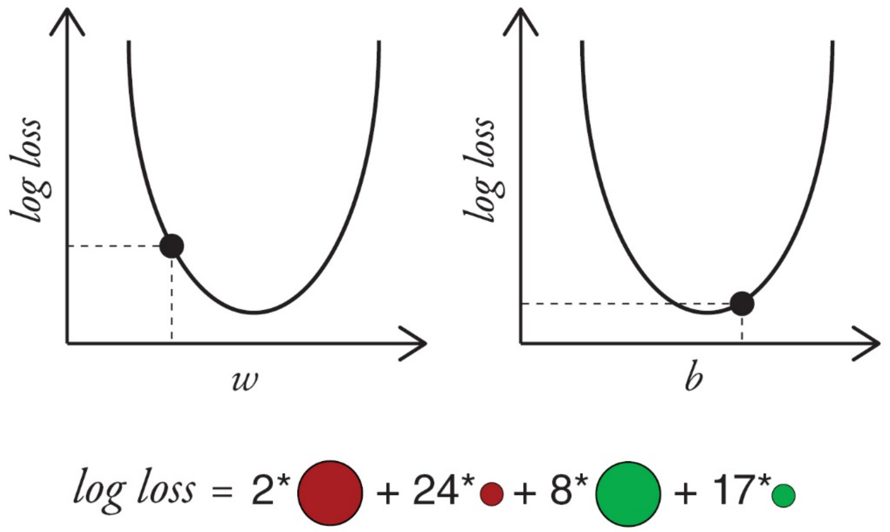




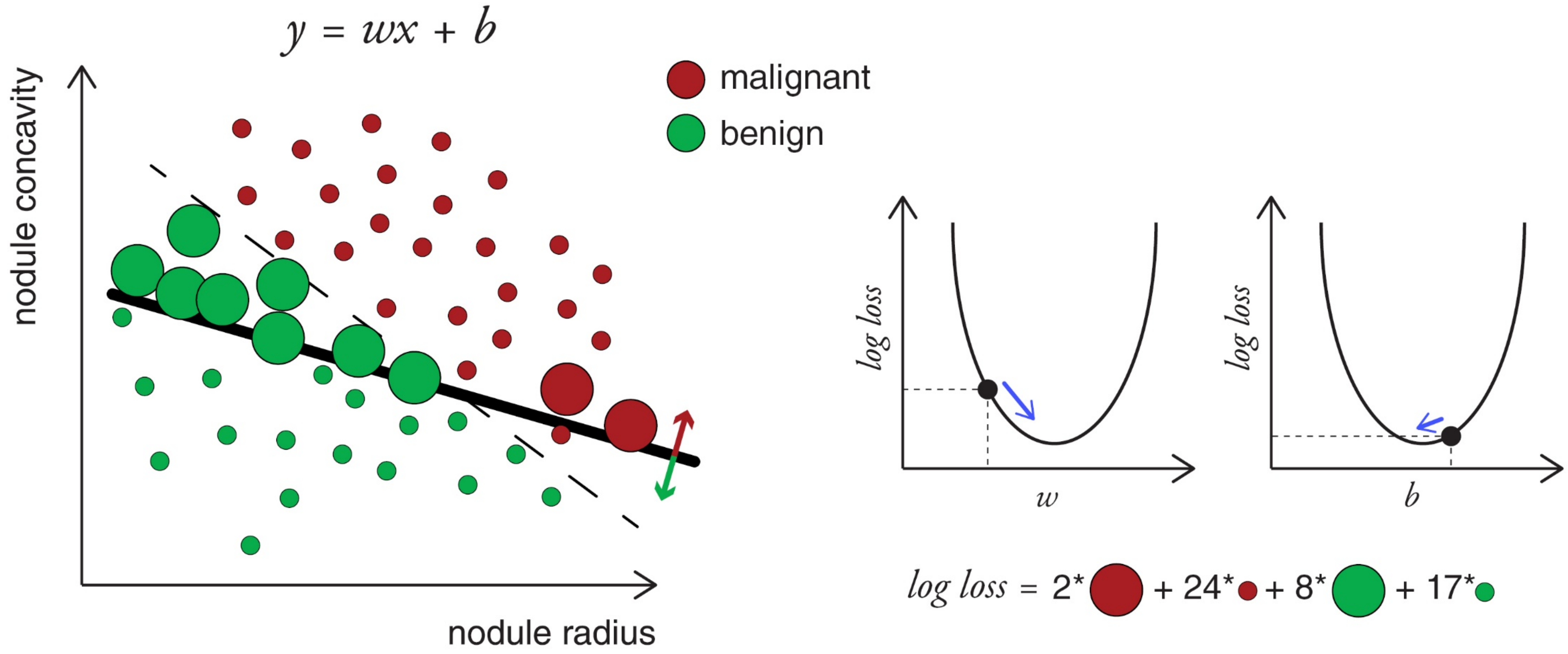
log loss = 3* (large malignant) + 23* (small malignant) + 14* (large benign) + 11* (small benign)

Logistic Regression

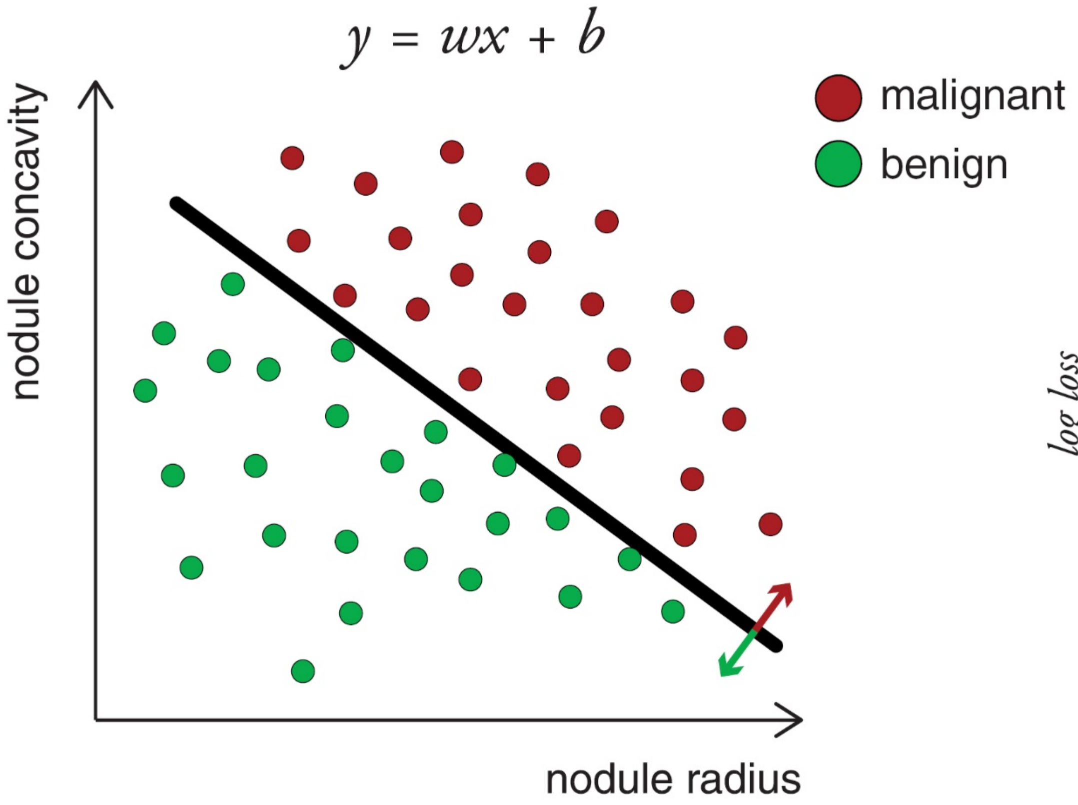


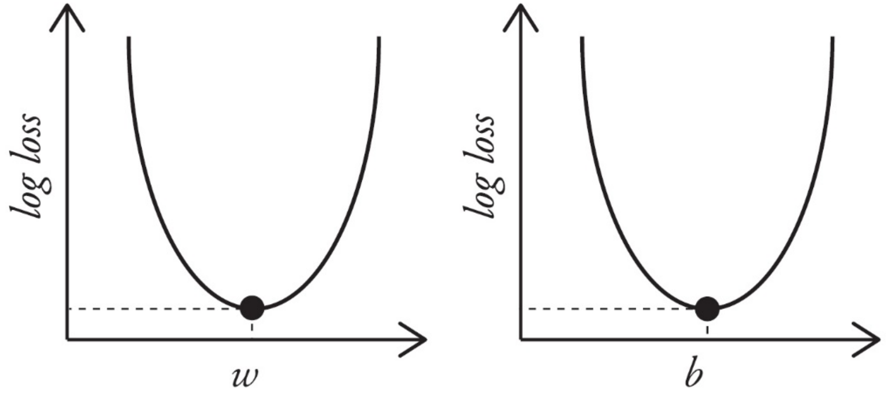


Logistic Regression



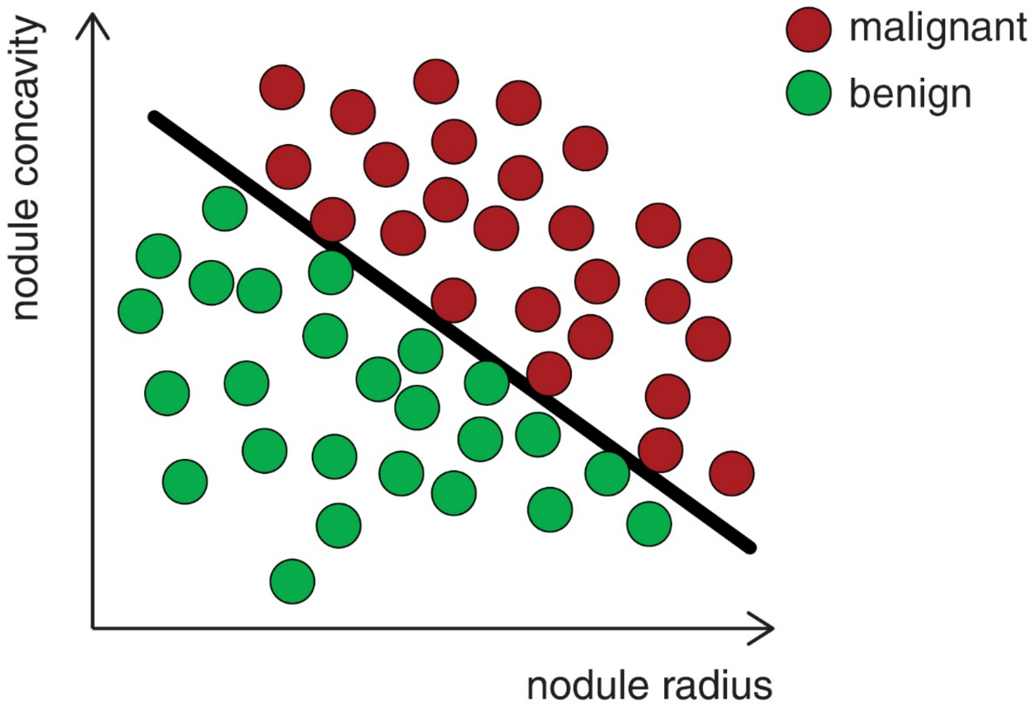
Logistic Regression



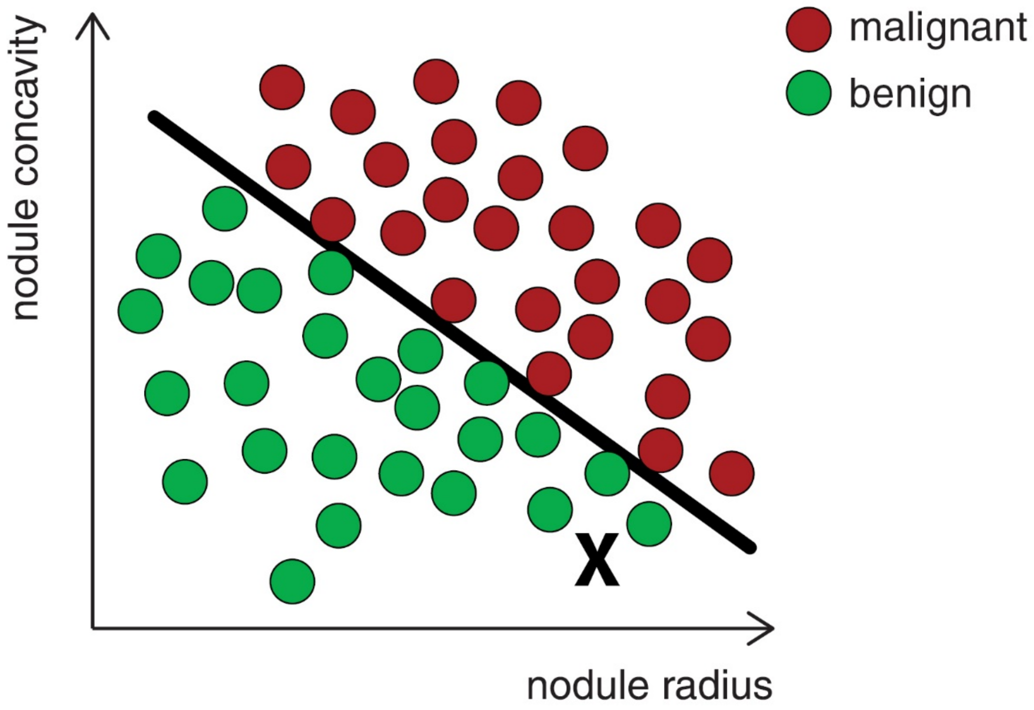


log loss = 26*● + 25*●

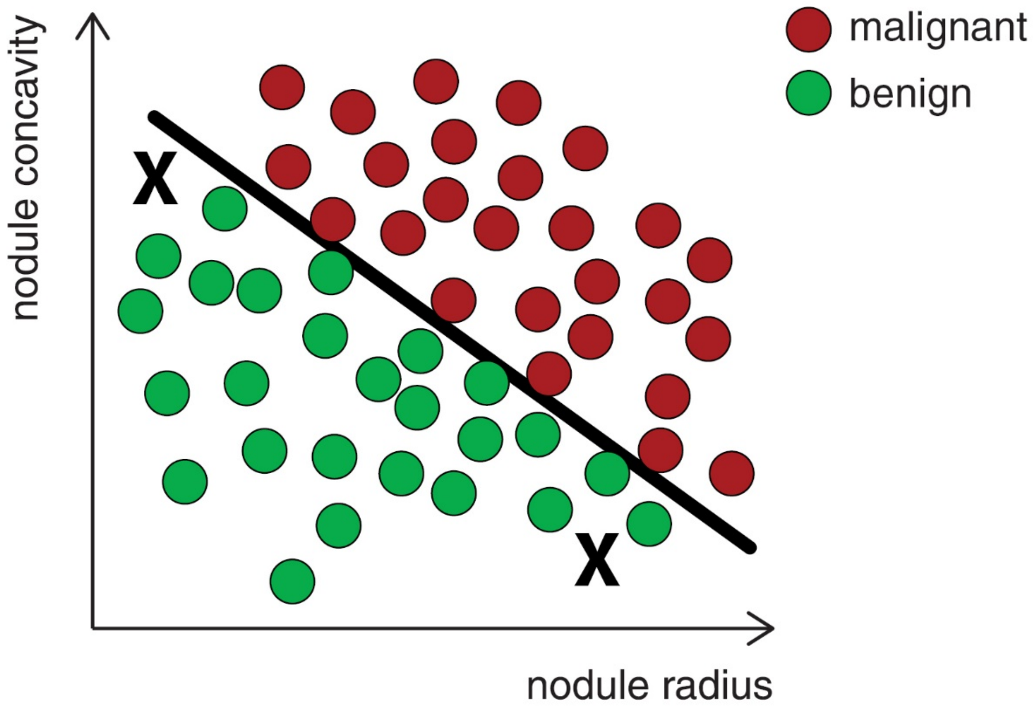
Dealing with Edge Conditions



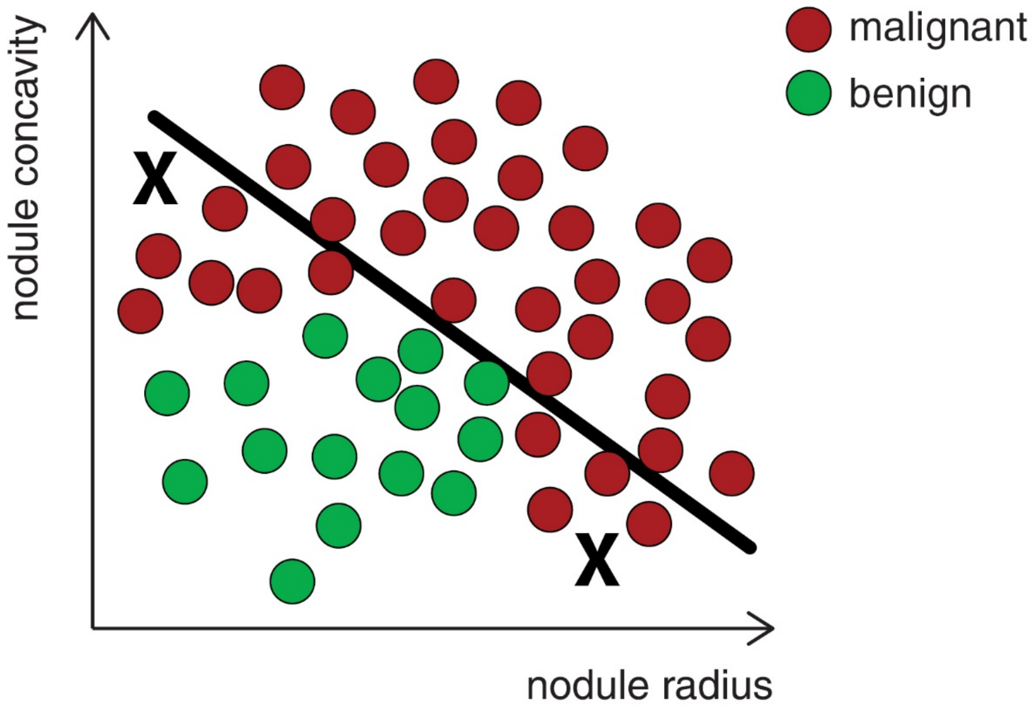
Dealing with Edge Conditions



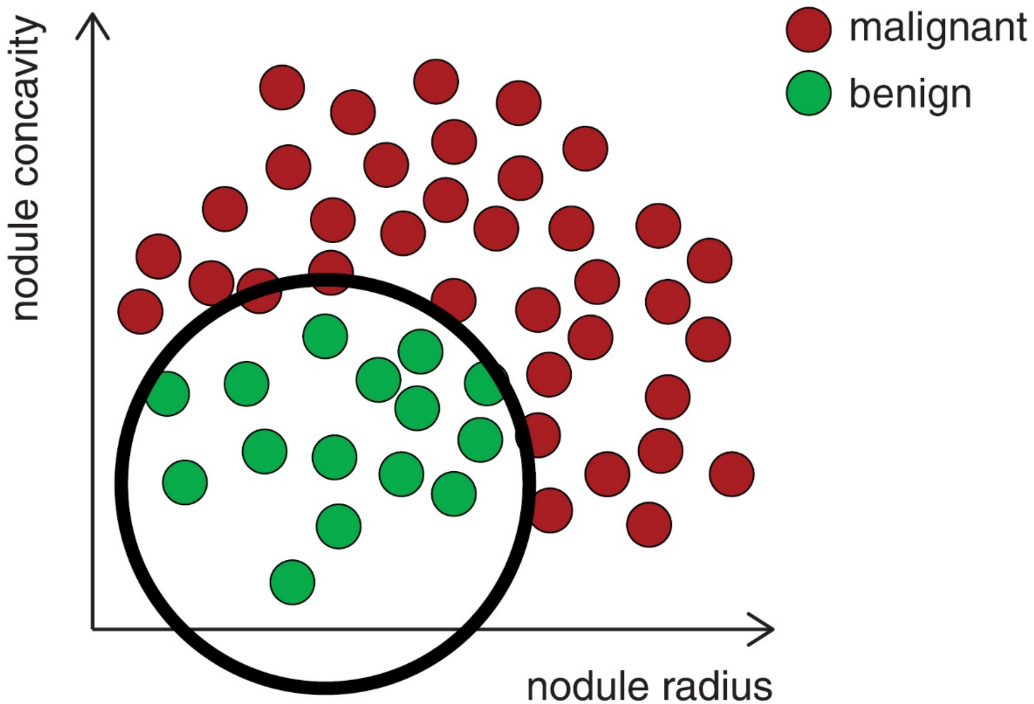
Dealing with Edge Conditions



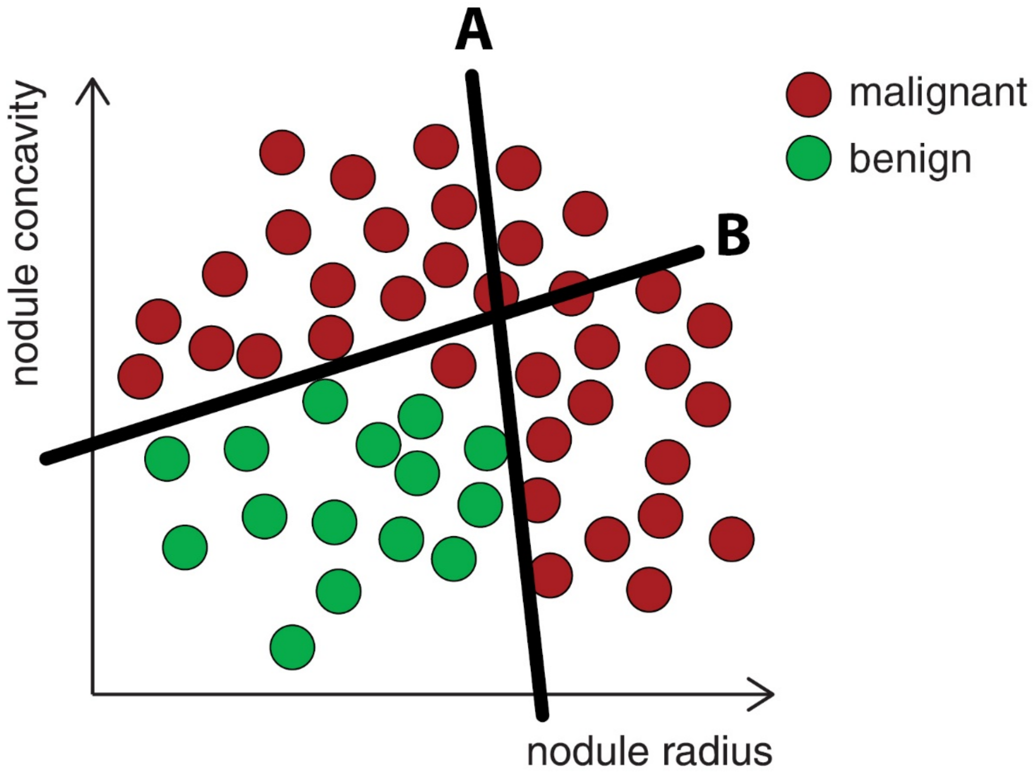
Dealing with Edge Conditions



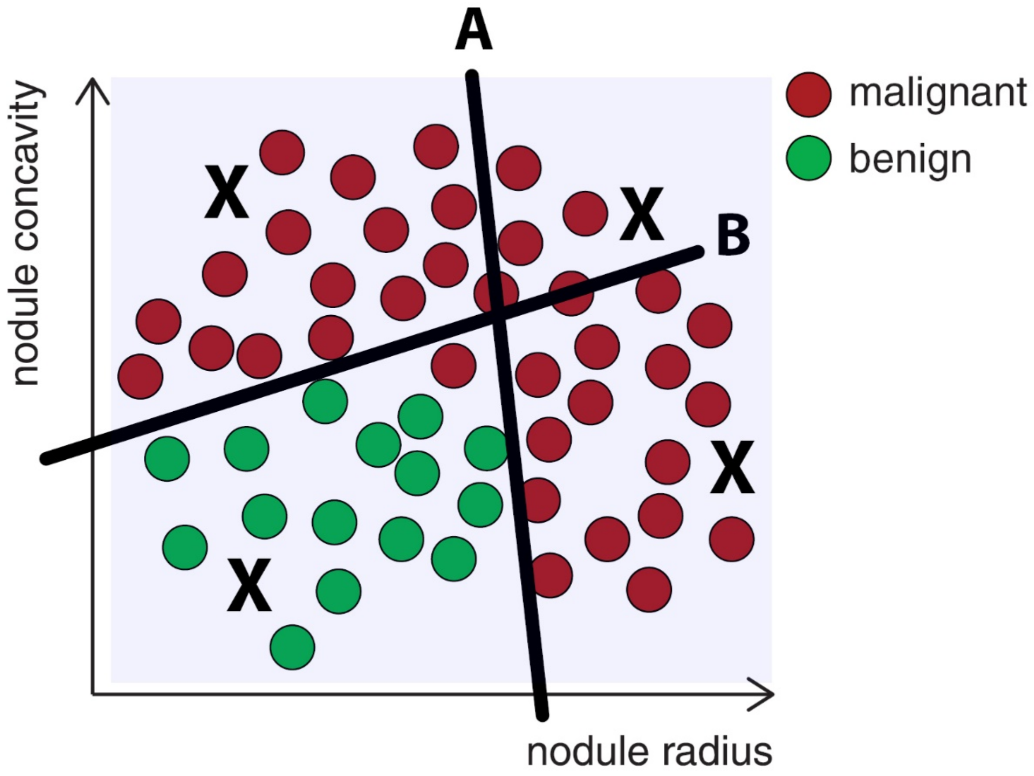
Dealing with Edge Conditions



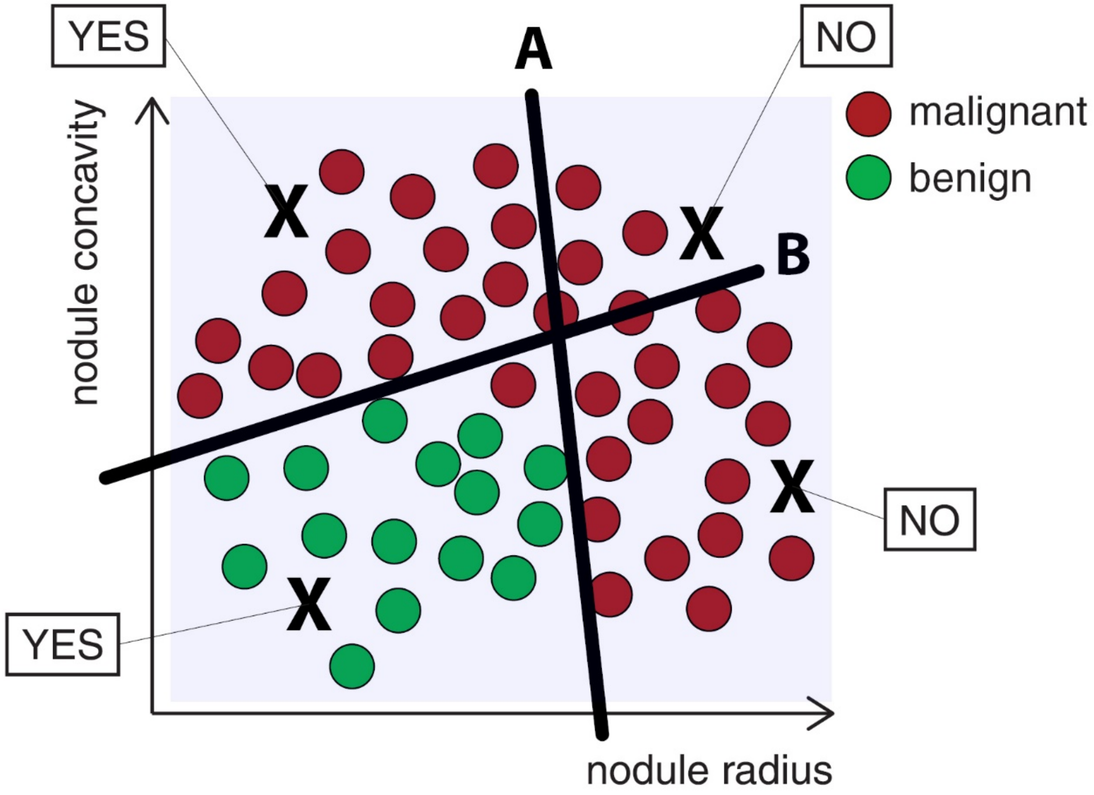
Dealing with Edge Conditions



Quadrant Questioning

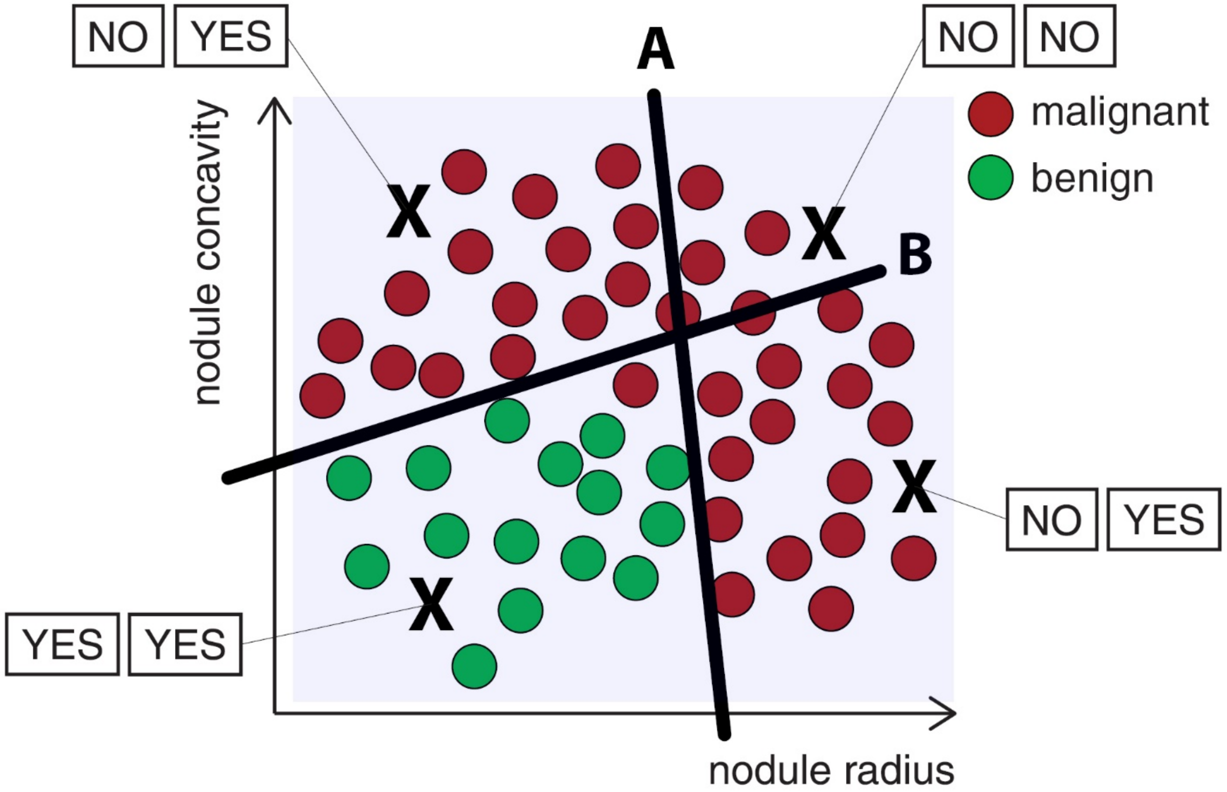


Quadrant Questioning



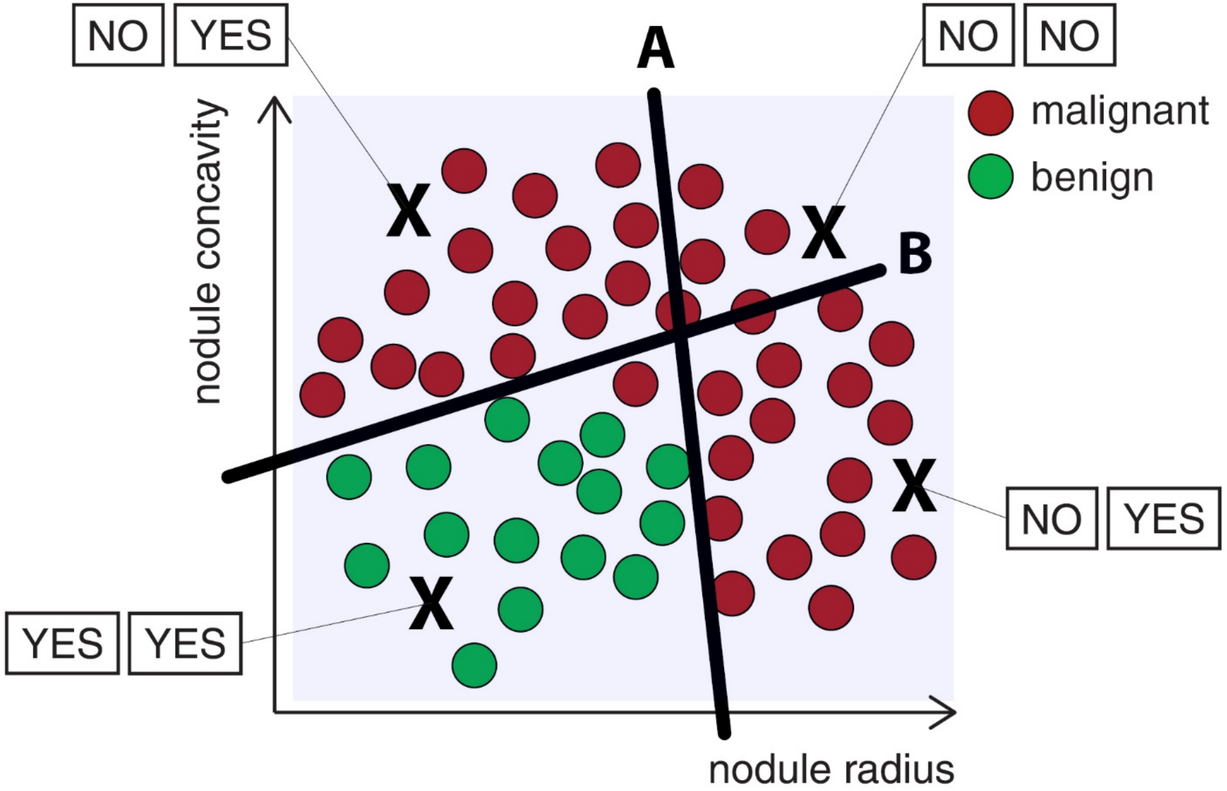
- Line A, over or under?

Quadrant Questioning



- Line A, over or under?
- Line B, over or under?

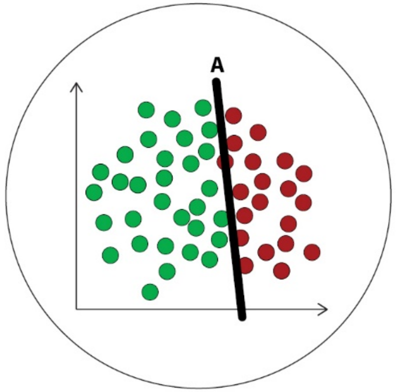
Quadrant Questioning

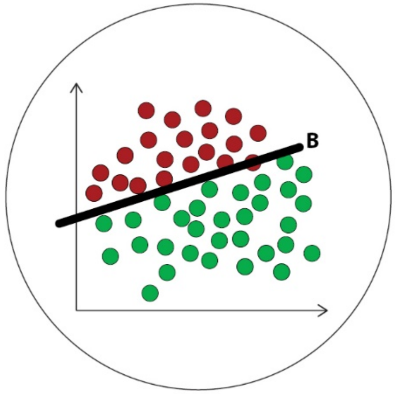


- Line A, over or under?
- Line B, over or under?
- Both YES?

Graph Representation

Line A, over or under?

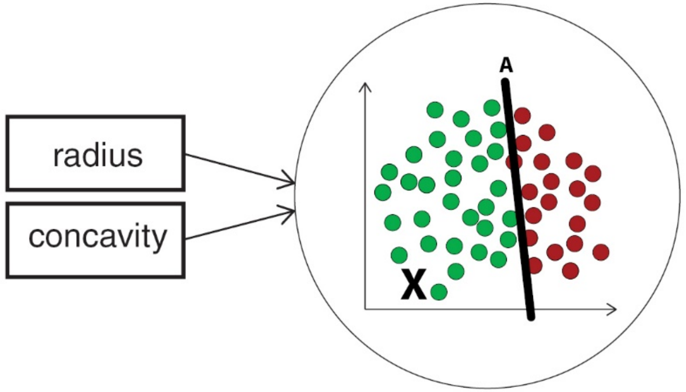


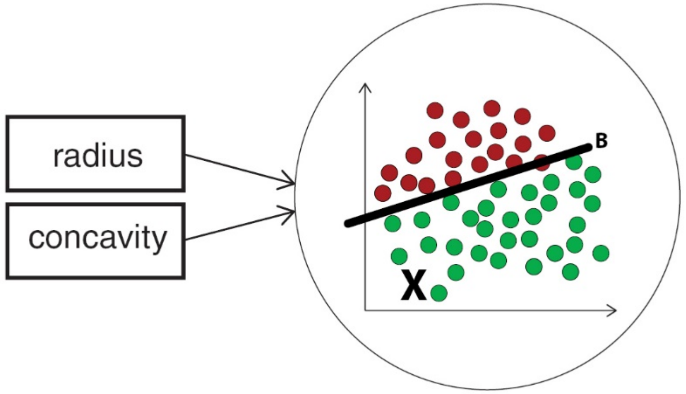


Line B, over or under?

Graph Representation

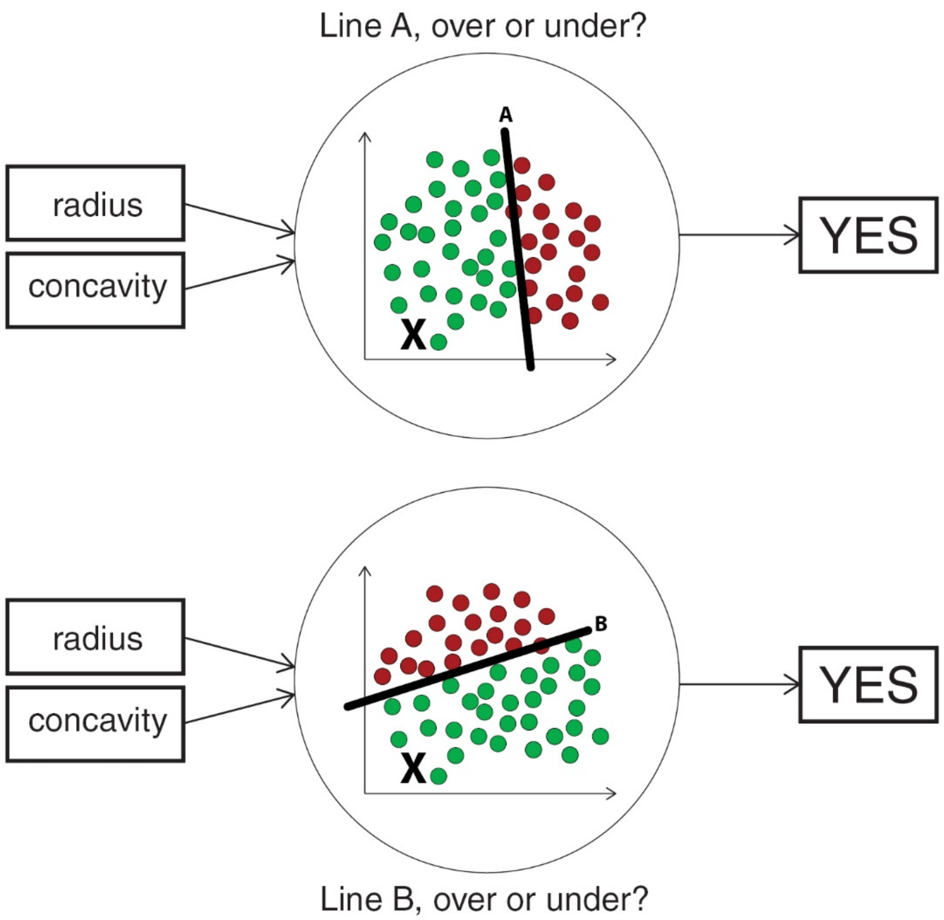
Line A, over or under?



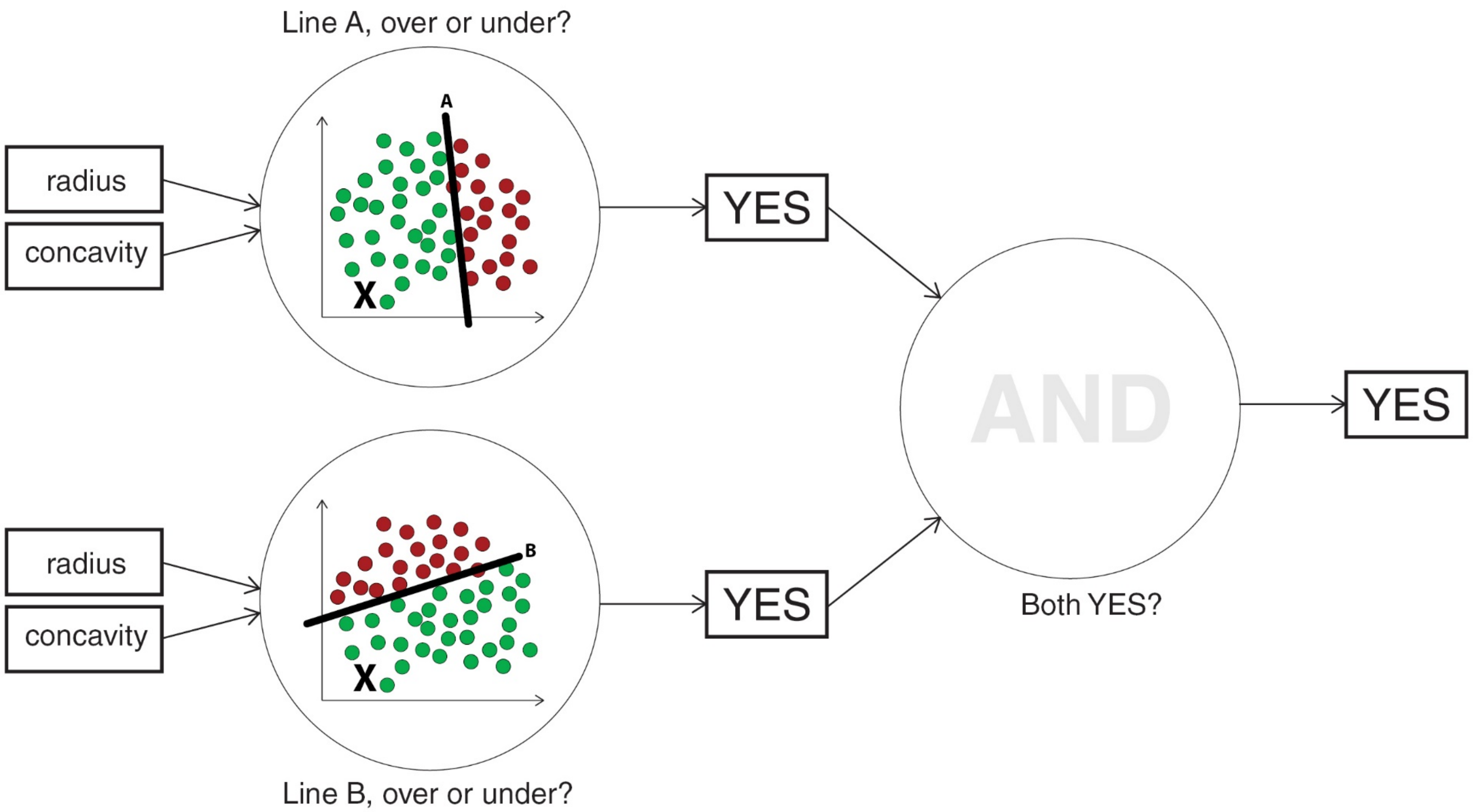


Line B, over or under?

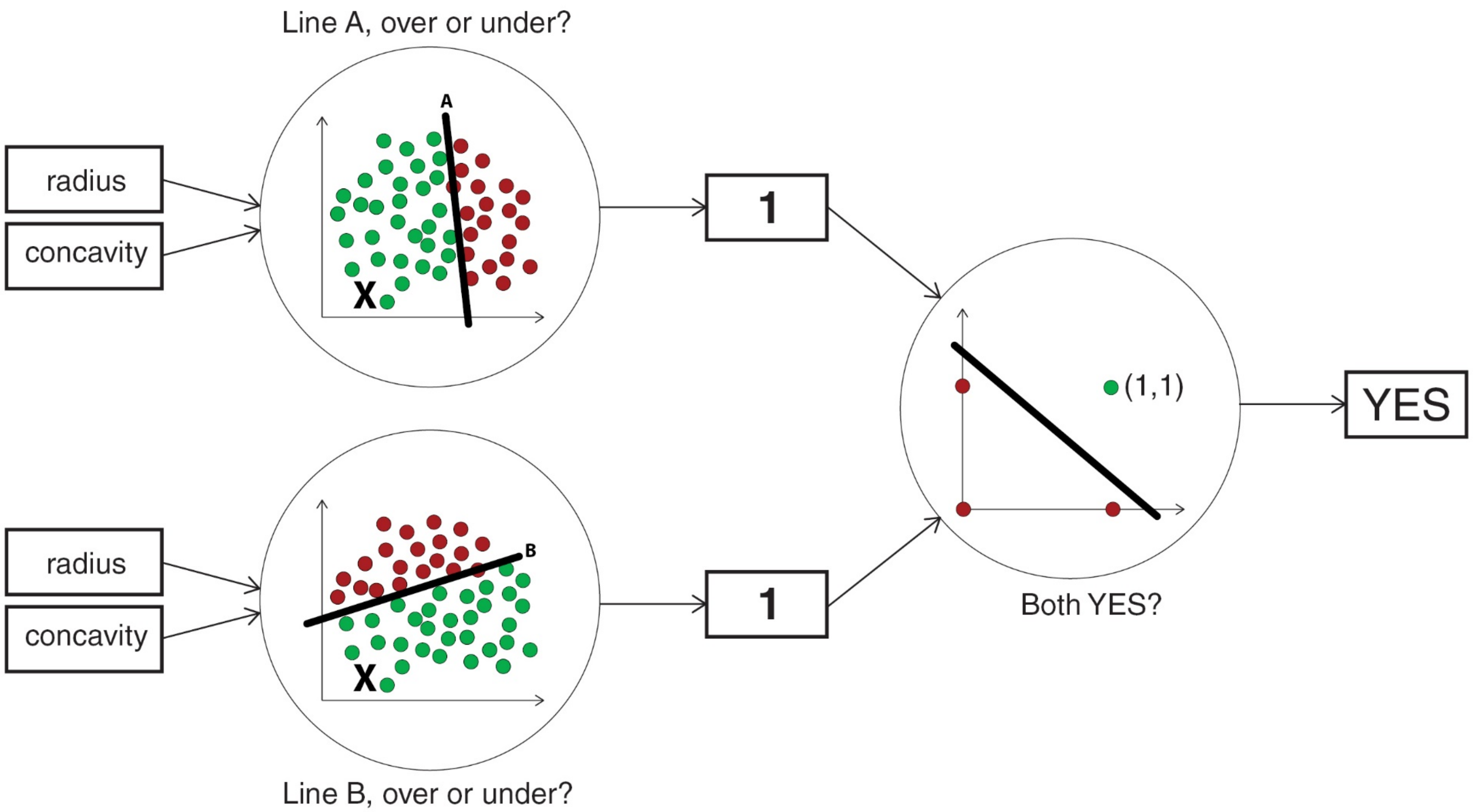
Graph Representation



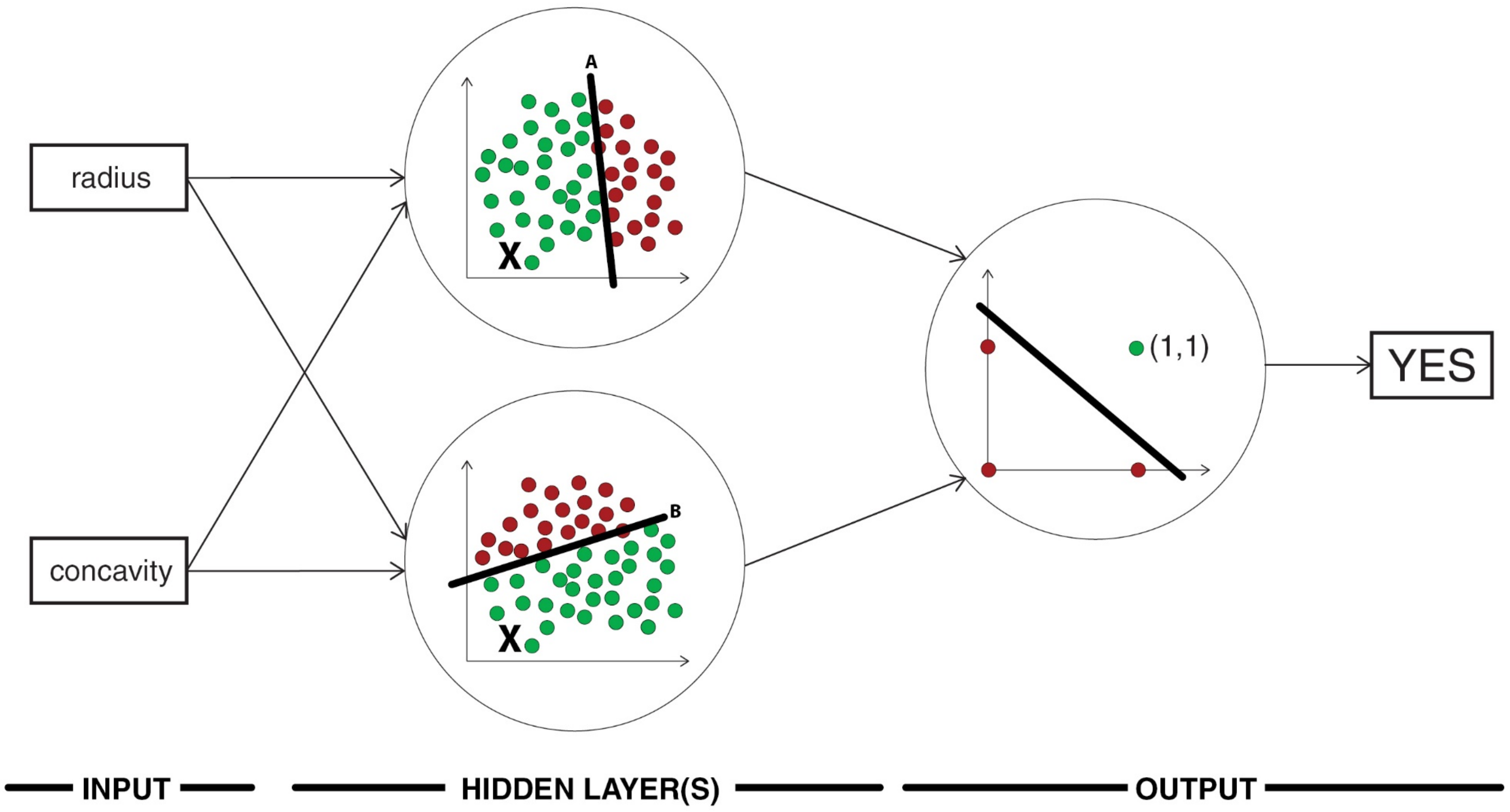
Graph Representation



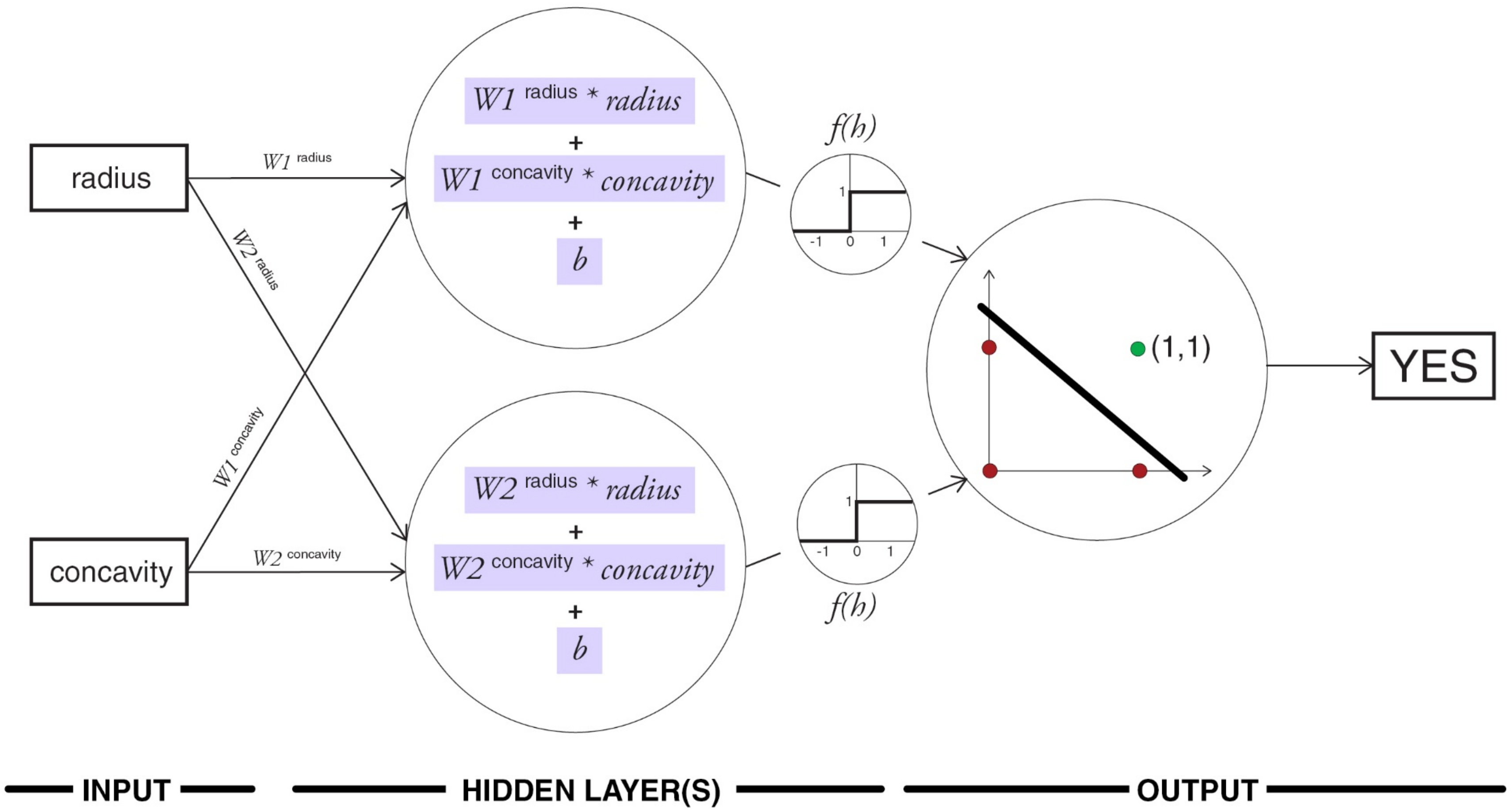
Graph Representation



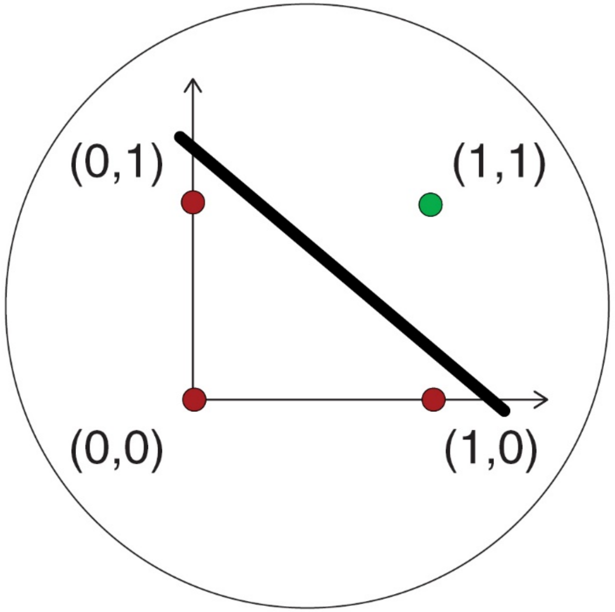
A Neural Network



A Neural Network

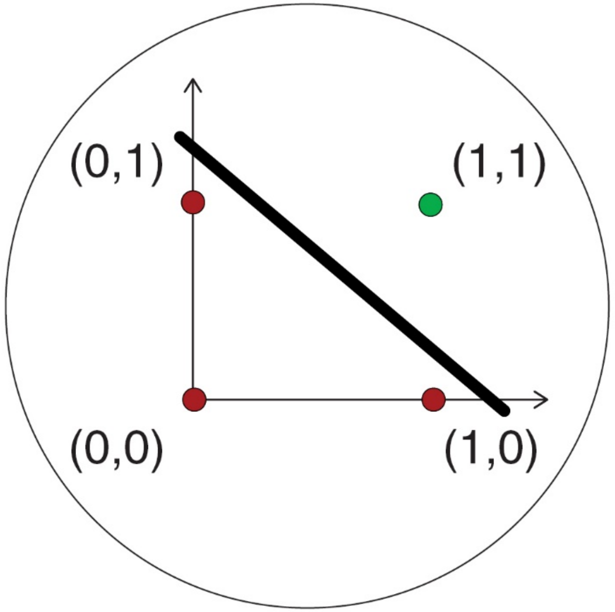


XOR Perceptron

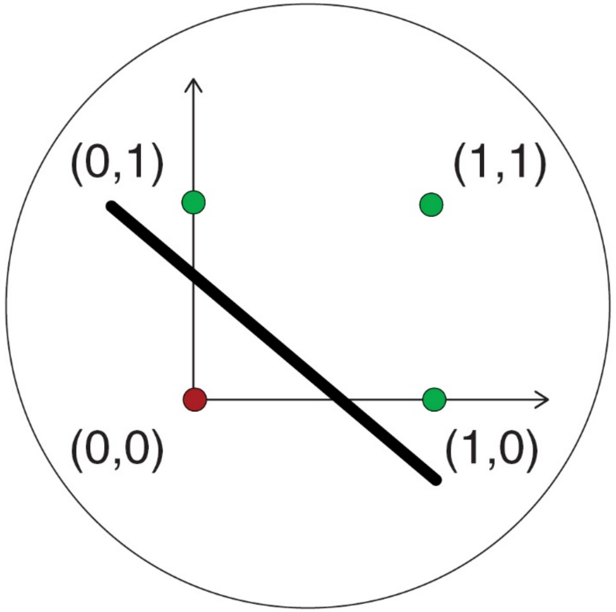


AND

XOR Perceptron

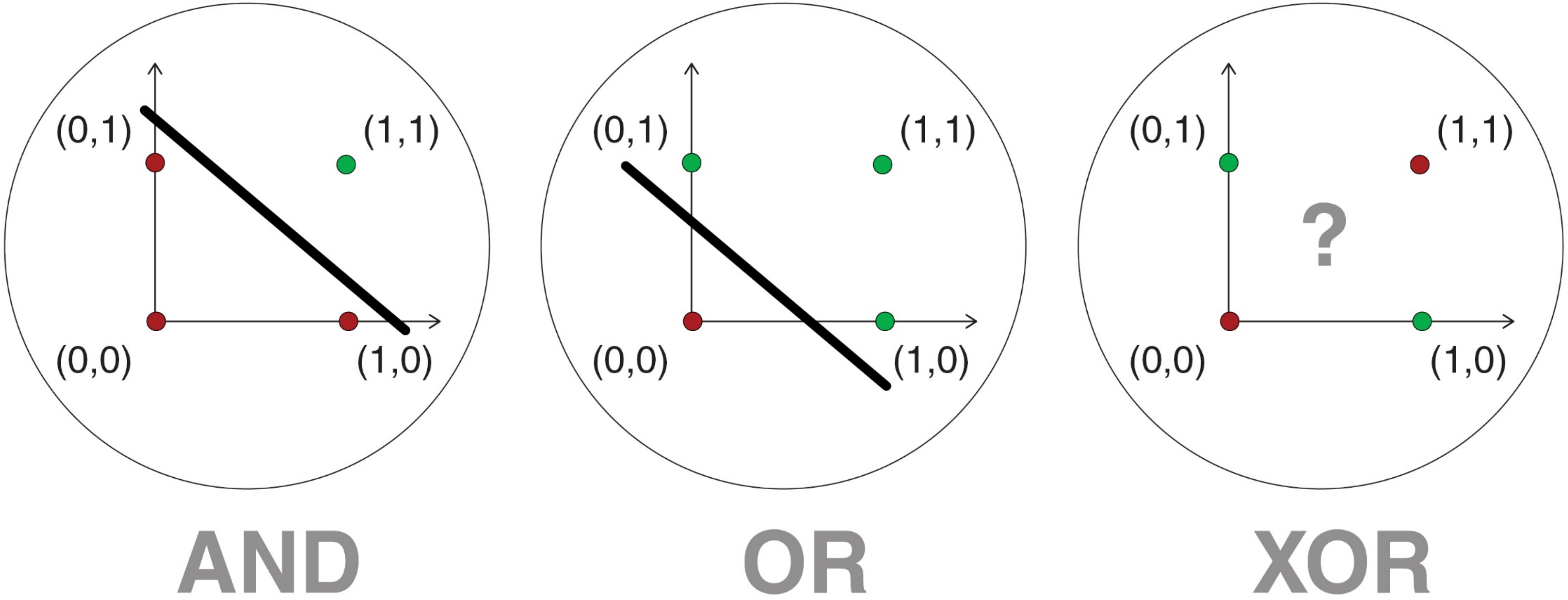


AND

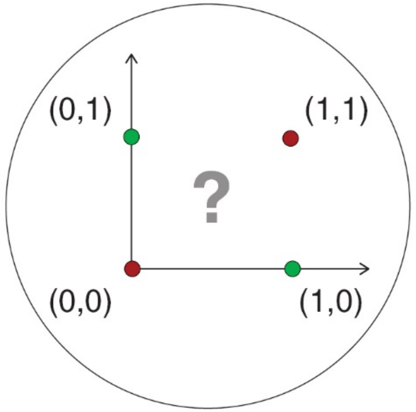


OR

XOR Perceptron

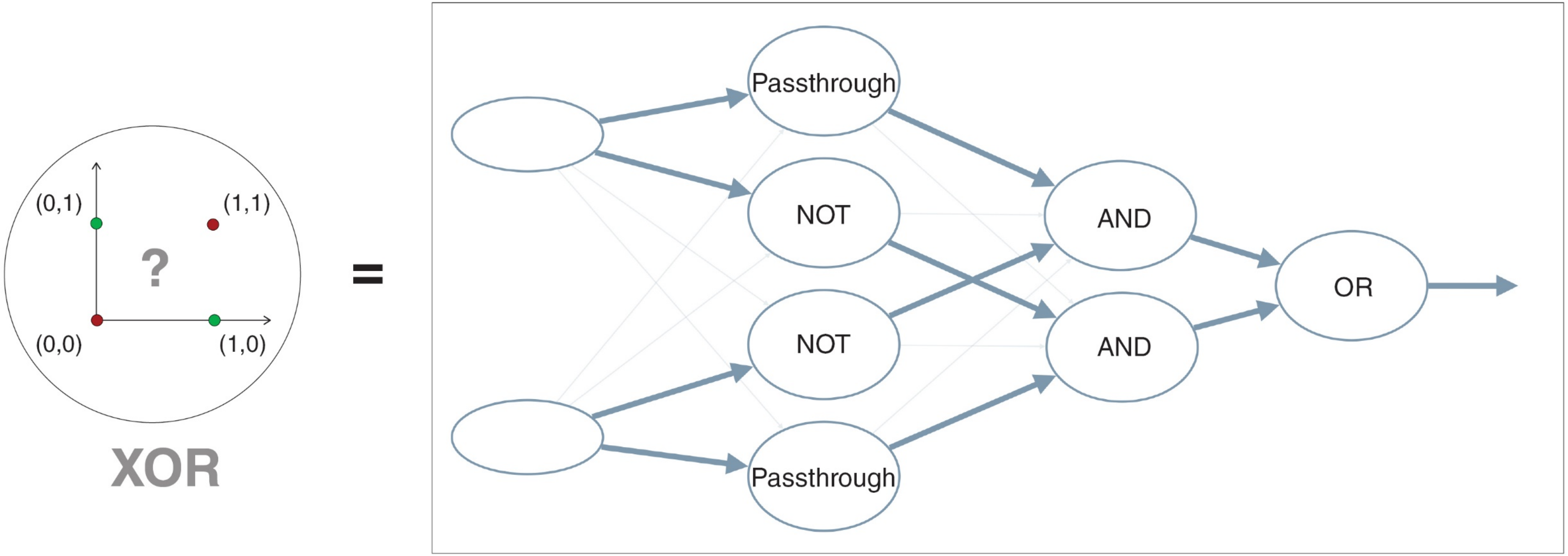


XOR Perceptron

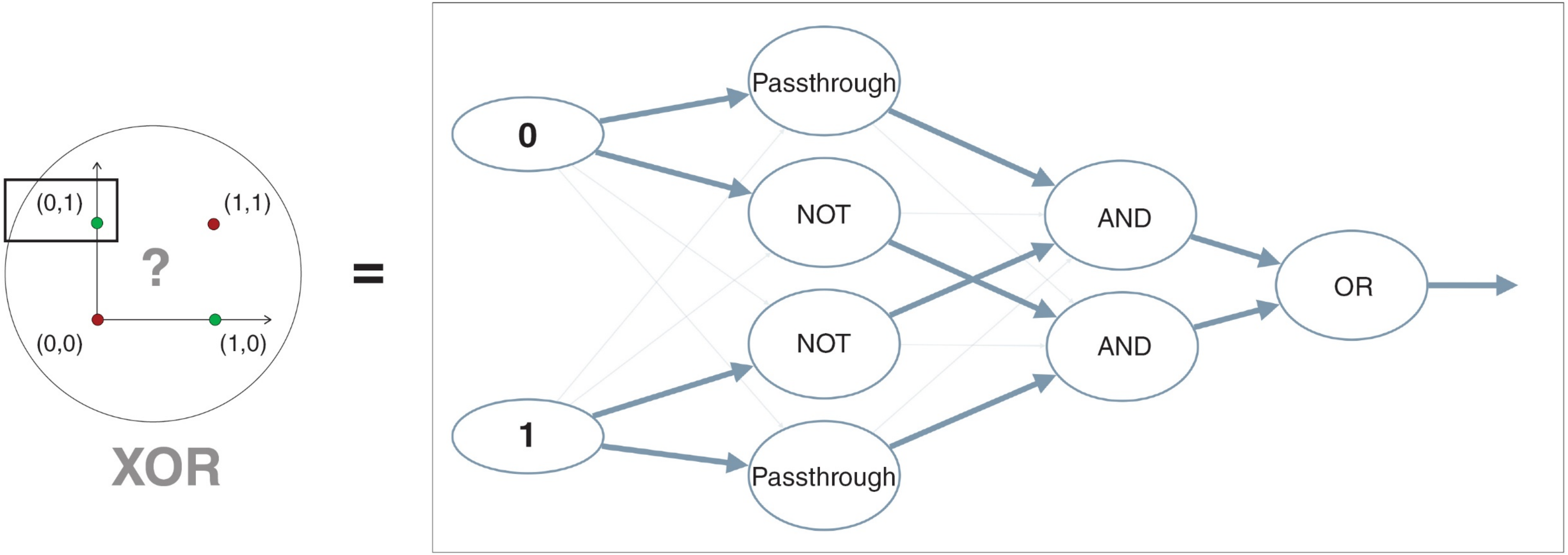


XOR

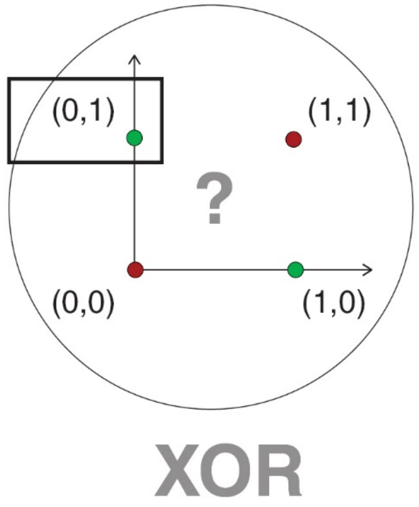
XOR Perceptron



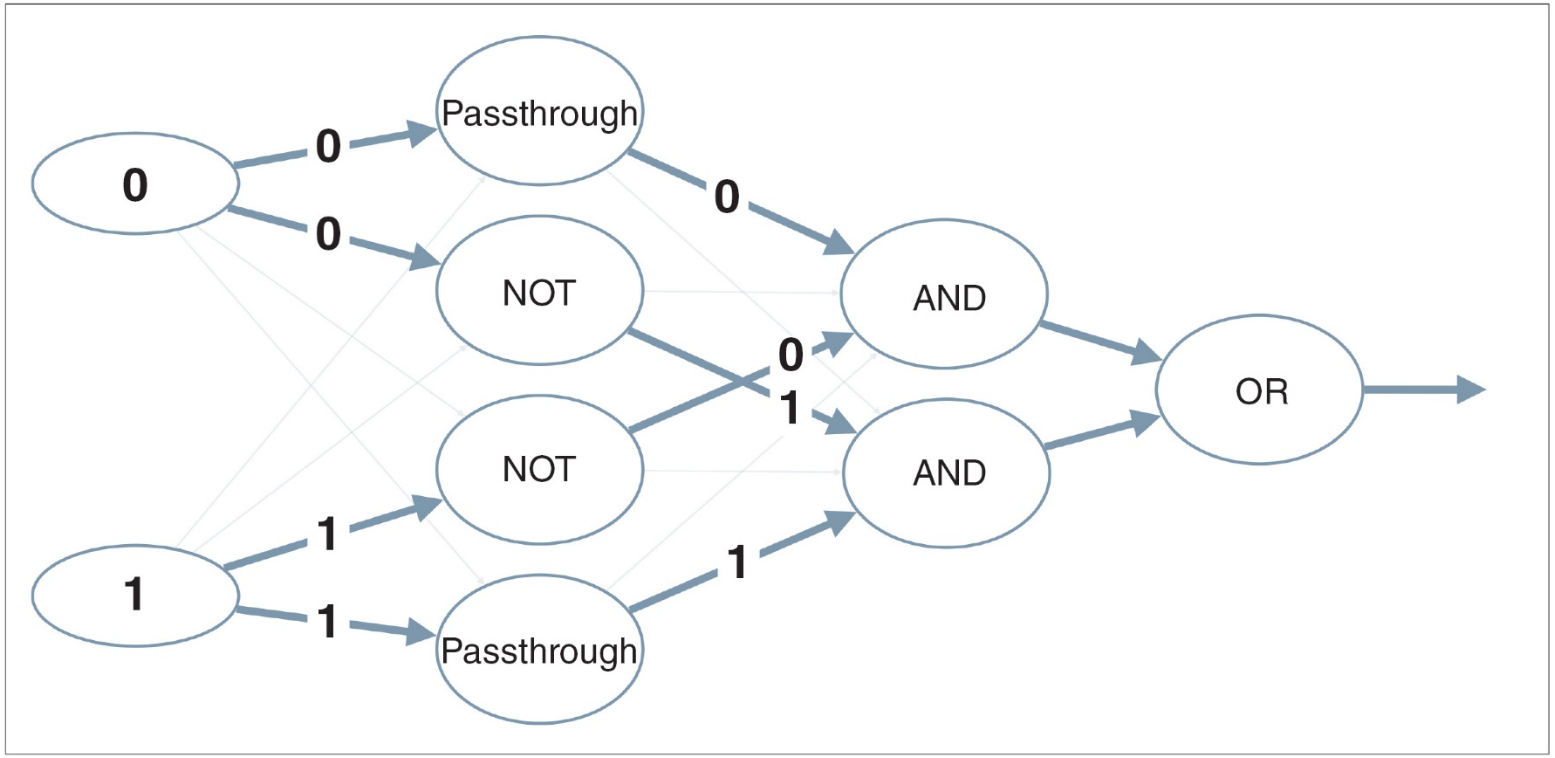
XOR Perceptron



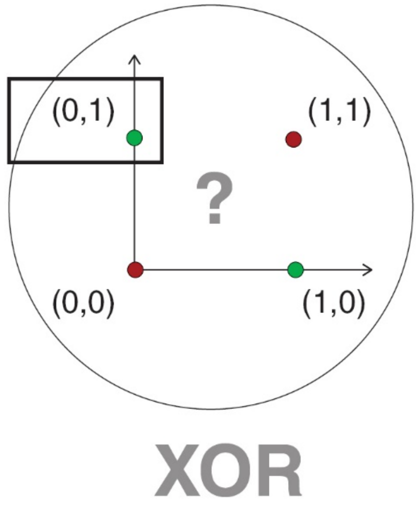
XOR Perceptron



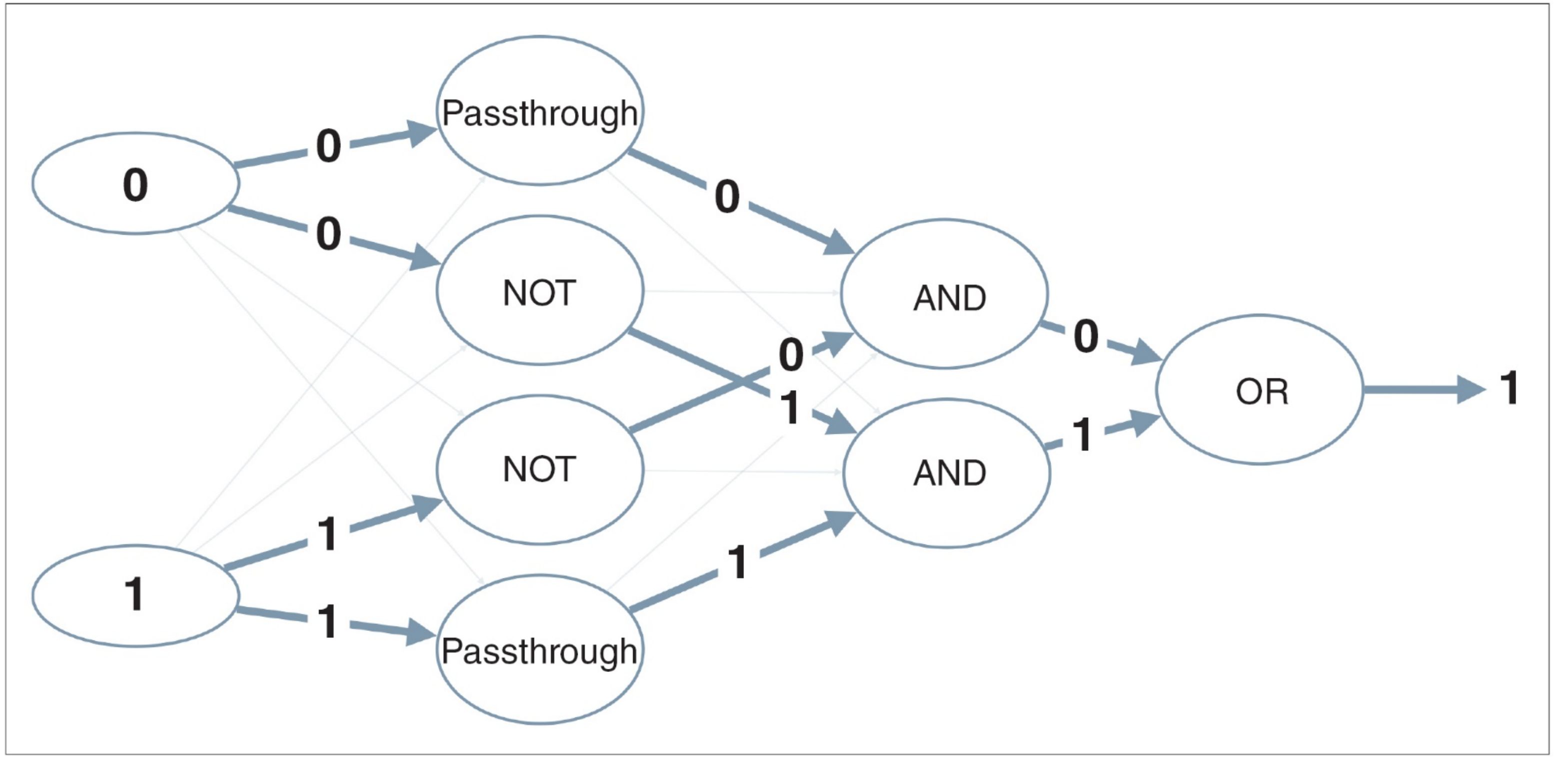
=



XOR Perceptron



=

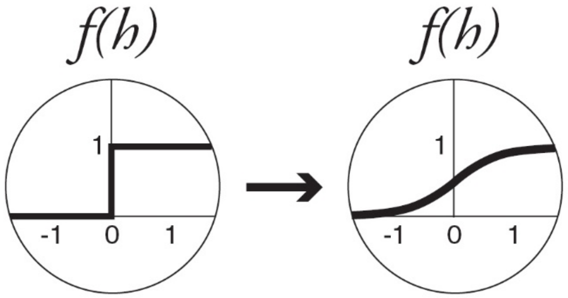


What is the intuition behind neural networks?

How do neural networks learn?

How to train neural networks?

Backpropagation & Gradient Descent in Neural Networks



NATURE VOL. 323 9 OCTOBER 1986 — LETTERS TO NATURE — 533

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \qquad (2)$$

† To whom correspondence should be addressed

534 — LETTERS TO NATURE — NATURE VOL. 323 9 OCTOBER 1986

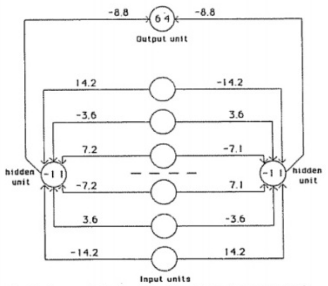


Fig. 1 A network that has learned to detect mirror symmetry in the input vector. The numbers on the arcs are weights and the numbers inside the nodes are biases. The learning required 1,425 sweeps through the set of 64 possible input vectors, with the weights being adjusted on the basis of the accumulated gradient after each sweep. The values of the parameters in equation (9) were $\varepsilon = 0.1$ and $\alpha = 0.9$. The initial weights were random and were uniformly distributed between −0.3 and 0.3. The key property of this solution is that for a given hidden unit, weights that are symmetric about the middle of the input vector are equal in magnitude and opposite in sign. So if a symmetrical pattern is presented, both hidden units will receive a net input of 0 from the input units, and, because the hidden units have a negative bias, both will be off. In this case the output unit, having a positive bias, will be on. Note that the weights on each side of the midpoint are in the ratio 1:2:4. This ensures that each of the eight patterns that can occur above the midpoint sends a unique activation sum to each hidden unit, so the only pattern below the midpoint that can exactly balance this sum is the symmetrical one. For all non-symmetrical patterns, both hidden units will receive non-zero activations from the input units. The two hidden units have identical patterns of weights but with opposite signs, so for every non-symmetric pattern one hidden unit will come on and suppress the output unit.

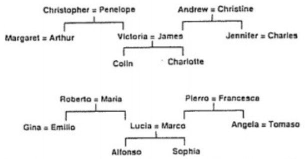


Fig. 2 Two isomorphic family trees. The information can be expressed as a set of triples of the form ⟨person 1⟩⟨relationship⟩⟨person 2⟩, where the possible relationships are {father, mother, husband, wife, son, daughter, uncle, aunt, brother, sister, nephew, niece}. A layered net can be said to 'know' these triples if it can produce the third term of each triple when given the first two. The first two terms are encoded by activating two of the input units, and the network must then complete the proposition by activating the output unit that represents the third term.

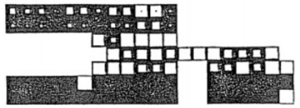


Fig. 3 Activity levels in a five-layer network after it has learned. The bottom layer has 24 input units on the left for representing ⟨person 1⟩ and 12 input units on the right for representing the relationship. The white squares inside these two groups show the activity levels of the units. There is one active unit in the first group representing Colin and one in the second group representing the relationship 'has-aunt'. Each of the two input groups is totally connected to its own group of 6 units in the second layer. These groups learn to encode people and relationships as distributed patterns of activity. The second layer is totally connected to the central layer of 12 units, and these are connected to the penultimate layer of 6 units. The activity in the penultimate layer must activate the correct output units, each of which stands for a particular ⟨person 2⟩. In this case, there are two correct answers (marked by black dots) because Colin has two aunts. Both the input units and the output units are laid out spatially with the English people in one row and the isomorphic Italians immediately below.

It is not necessary to use exactly the functions given in equations (1) and (2). Any input–output function which has a bounded derivative will do. However, the use of a linear function for combining the inputs to a unit before applying the nonlinearity greatly simplifies the learning procedure.

The aim is to find a set of weights that ensure that for each input vector the output vector produced by the network is the same as (or sufficiently close to) the desired output vector. If there is a fixed, finite set of input–output cases, the total error in the performance of the network with a particular set of weights can be computed by comparing the actual and desired output vectors for every case. The total error, E, is defined as

$$E = \tfrac{1}{2}\sum_c \sum_j (y_{j,c} - d_{j,c})^2 \qquad (3)$$

grad descent on E
$\partial E / \partial w$
Σ

where c is an index over cases (input–output pairs), j is an index over output units, y is the actual state of an output unit and d is its desired state. To minimize E by gradient descent it is necessary to compute the partial derivative ~~of E~~ with respect to each weight in the network. This is simply the sum of the partial derivatives for each of the input–output cases. For a given case, the partial derivatives of the error with respect to each weight are computed in two passes. We have already described the forward pass in which the units in each layer have their states determined by the input they receive from units in lower layers using equations (1) and (2). The backward pass which propagates derivatives from the top layer back to the bottom one is more complicated.

The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, c, and suppressing the index c gives

$$\partial E/\partial y_j = y_j - d_j \qquad (4)$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot dy_j/dx_j$$

Differentiating equation (2) to get the value of dy_j/dx_j and substituting gives

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1-y_j) \qquad (5)$$

This means that we know how a change in the total input x to an output unit will affect the error. But this total input is just a linear function of the states of the lower level units and it is also a linear function of the weights on the connections, so it is easy to compute how the error will be affected by changing these states and weights. For a weight w_{ji}, from i to j the derivative is

$$\partial E/\partial w_{ji} = \partial E/\partial x_j \cdot \partial x_j/\partial w_{ji} = \partial E/\partial x_j \cdot y_i \qquad (6)$$

and for the output of the i^{th} unit the contribution to $\partial E/\partial y_i$

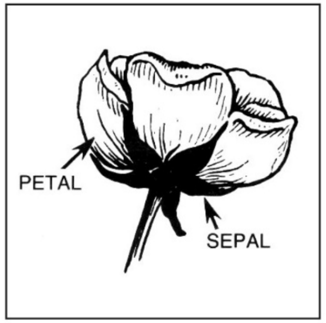
David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Learning Representations by Back-propagating Errors

Nature - 1986

Iris Dataset

Inputs



petal length & width
sepal length & width

Outputs

Iris Versicolor

Iris Setosa

Iris Virginica

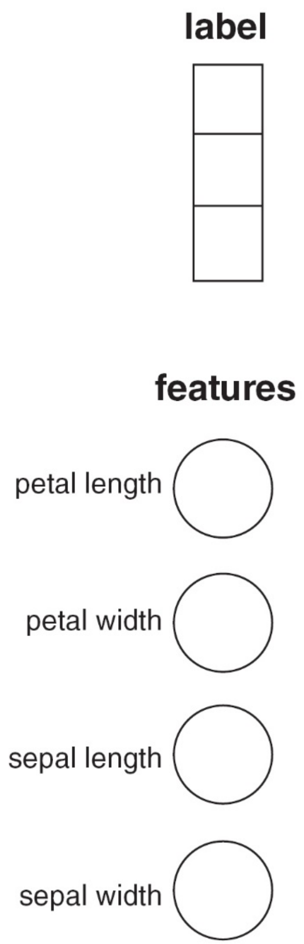
R.A. Fisher

The Use of Multiple Measurements in Taxonomic Problems

Annual Eugenics - 1936

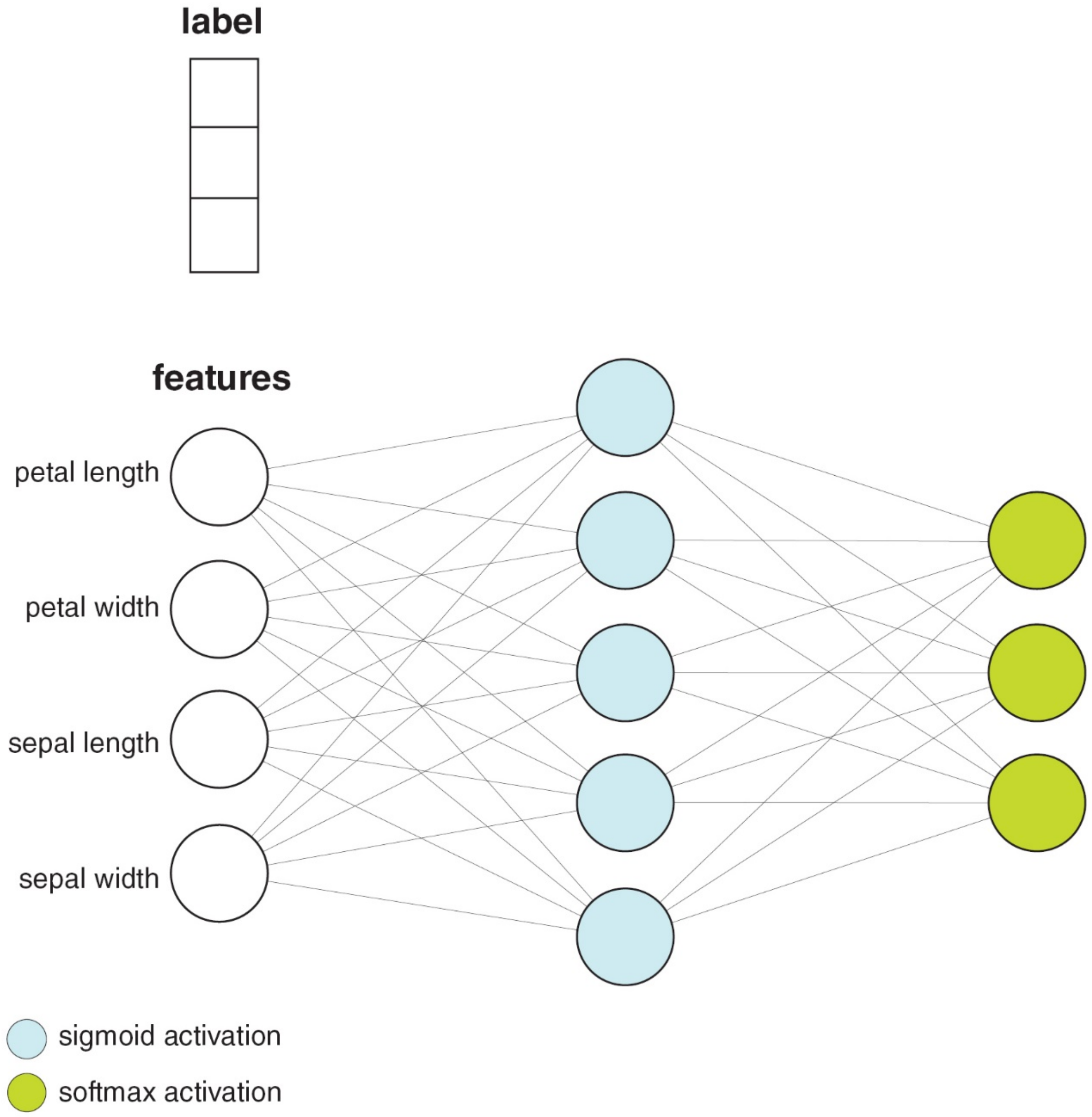
How Neural Networks Learn

✓ **Data:** iris dataset



How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network



sigmoid

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

softmax

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$ for $j = 1, \ldots, K$.

How Neural Networks Learn

✓ **Data:** iris dataset
✓ **Model:** 3-layer neural network
✓ **Loss:** cross entropy

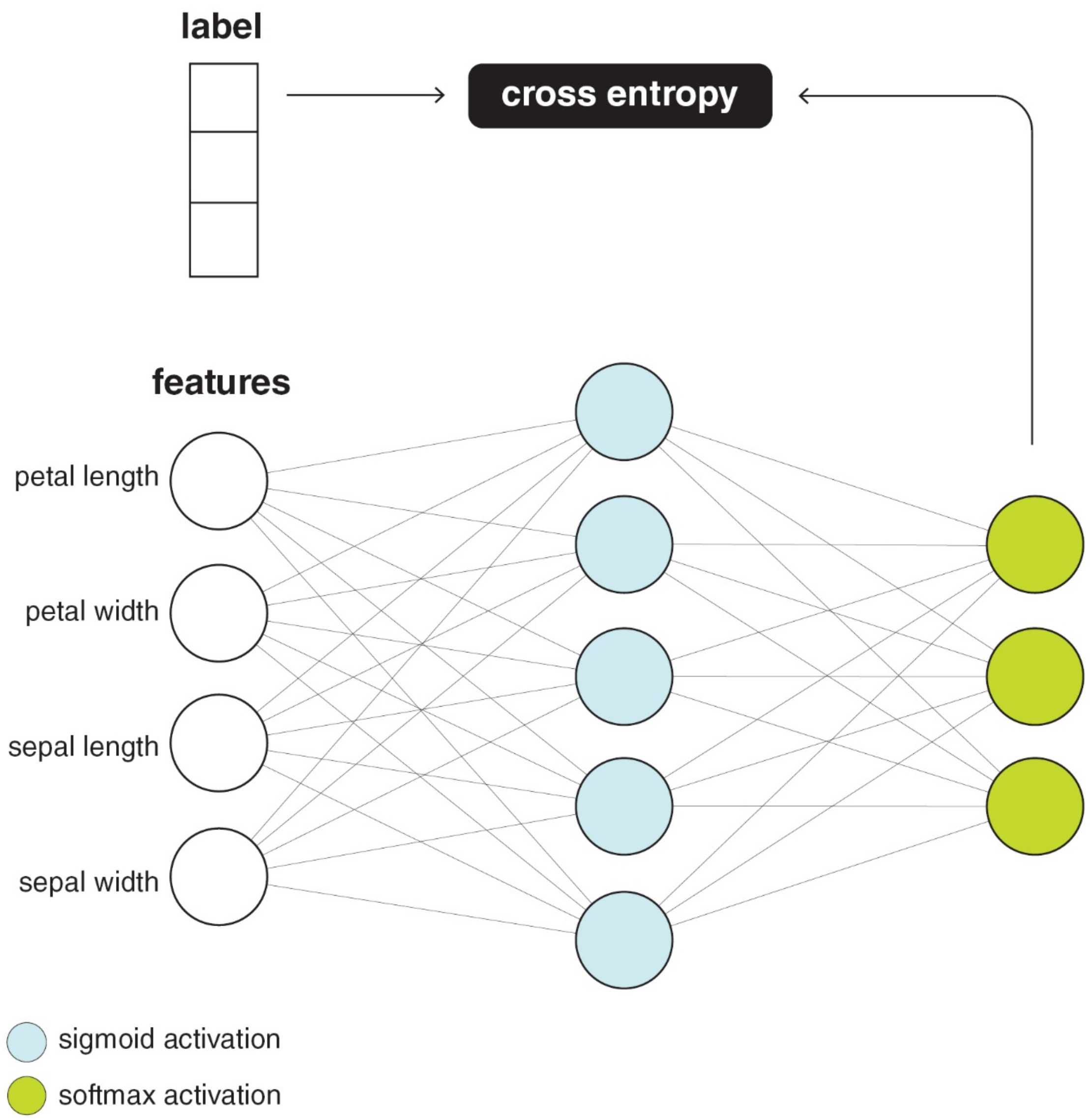
↑ **likelihood**

=

↑ **log(likelihood)**

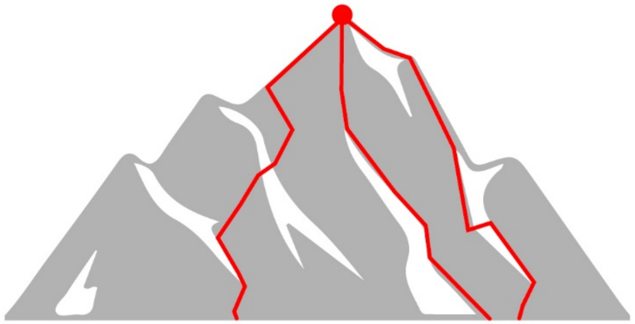
=

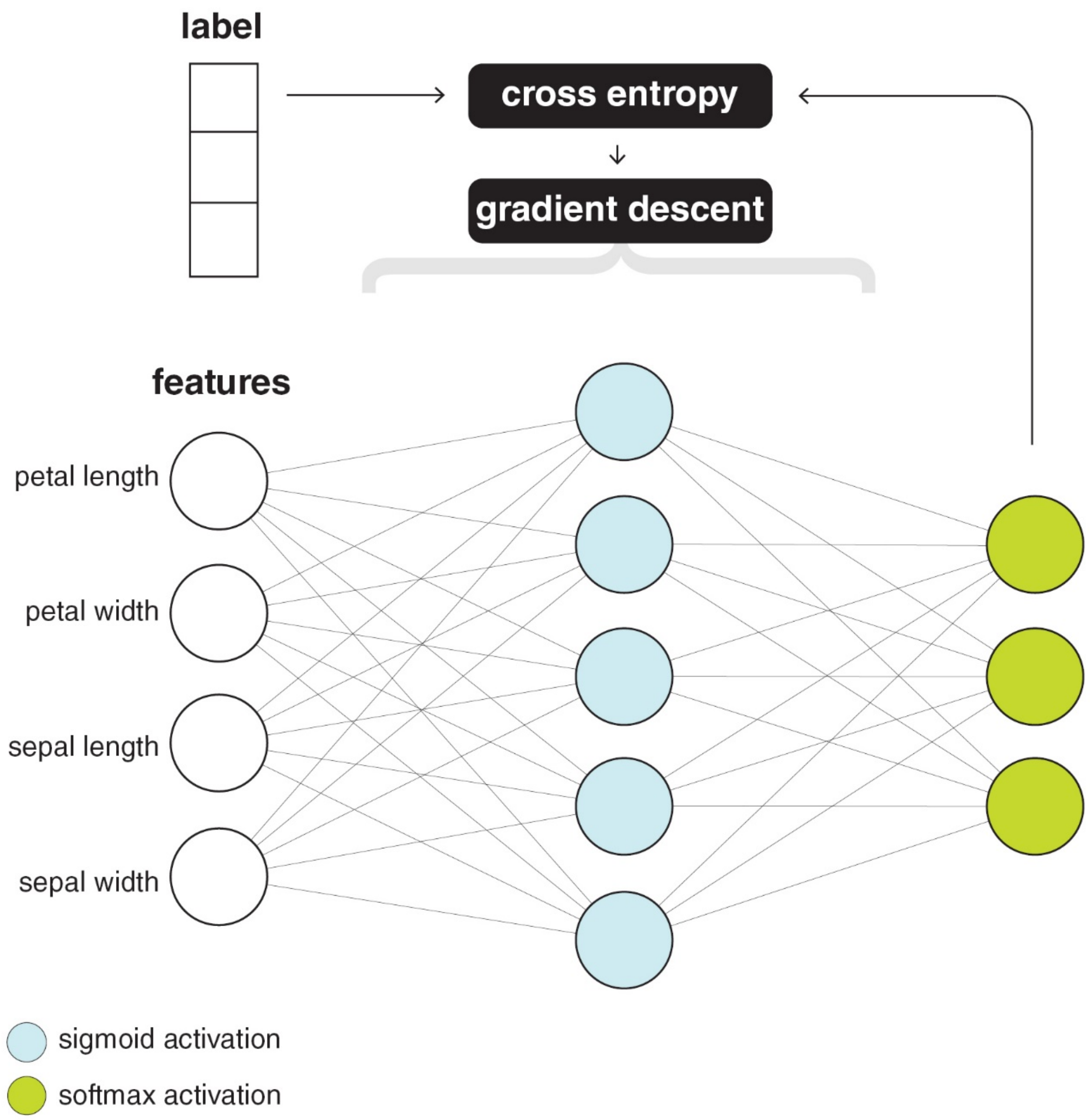
↓ **-log(likelihood)**



How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent



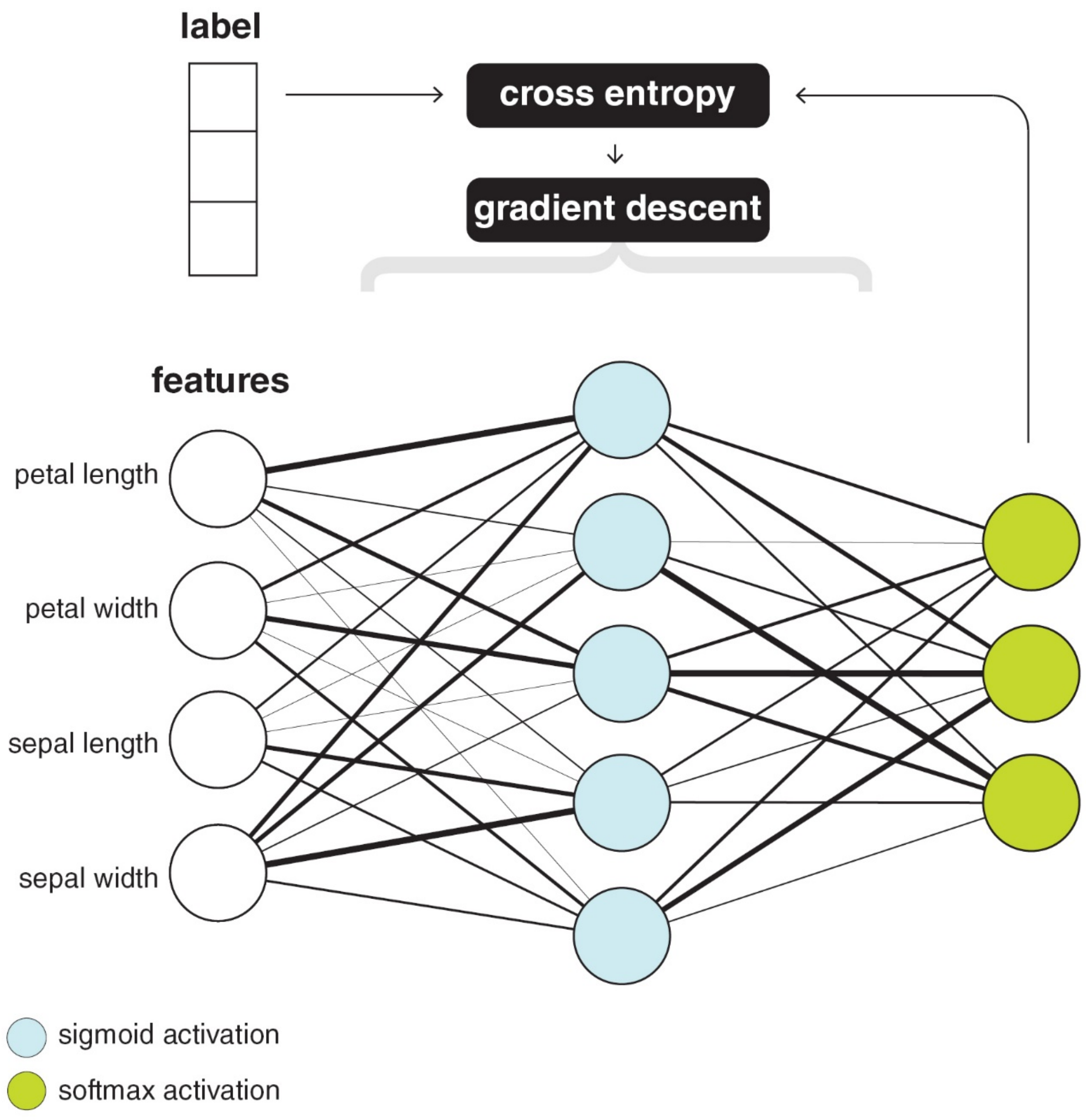


How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization

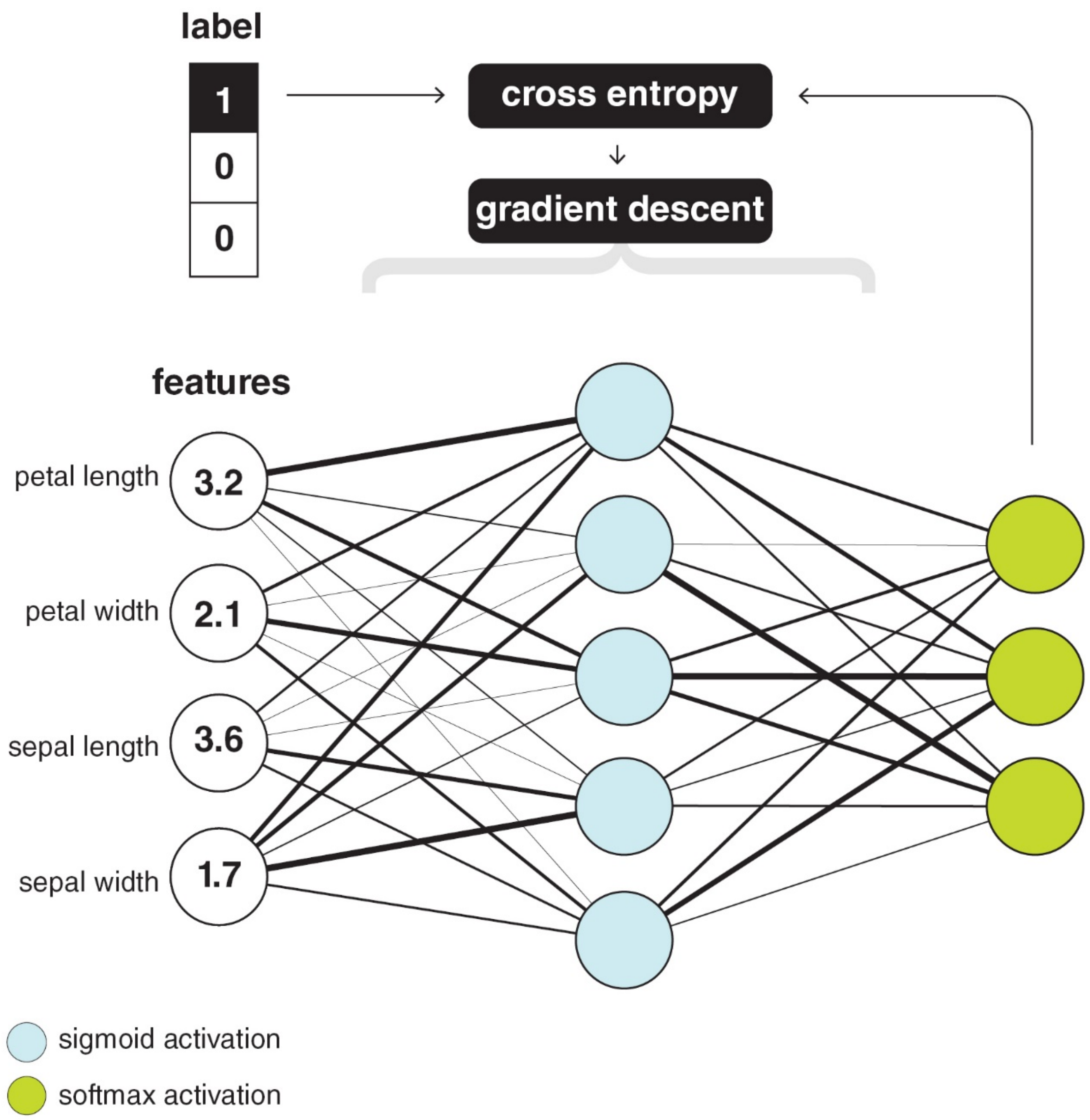


How Neural Networks Learn

✓ **Data:** iris dataset
✓ **Model:** 3-layer neural network
✓ **Loss:** cross entropy
✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input

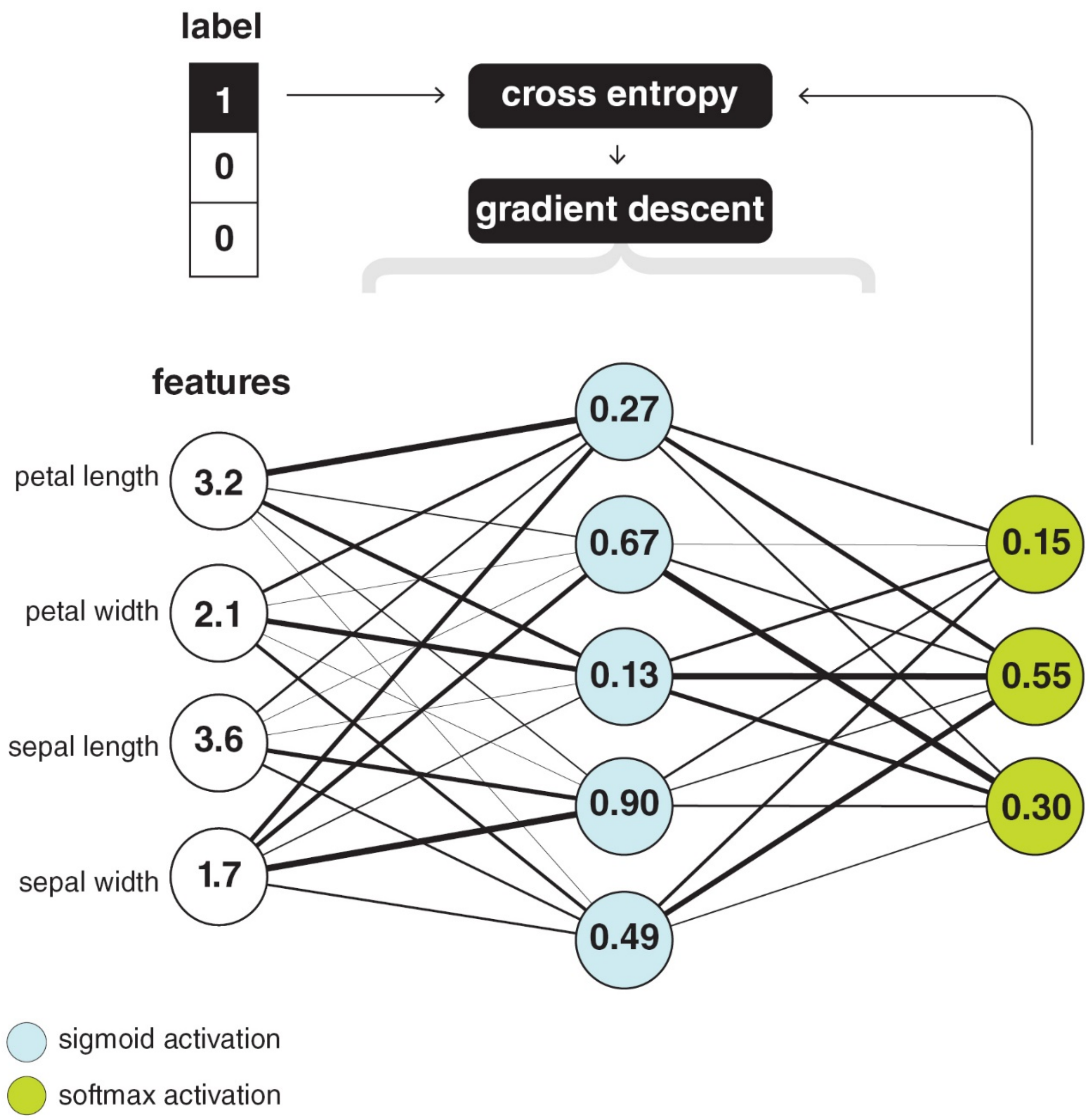


How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation

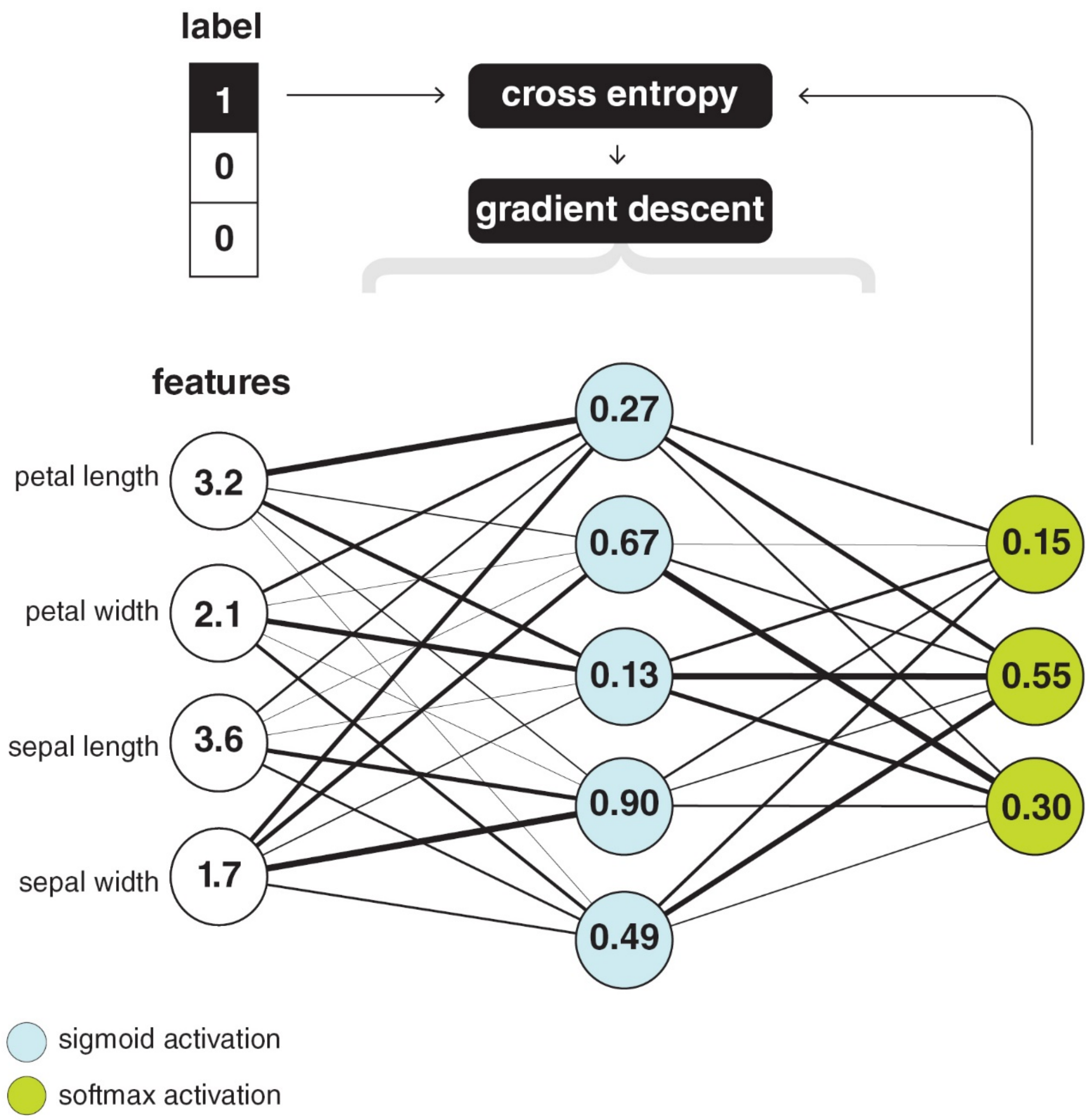


How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation

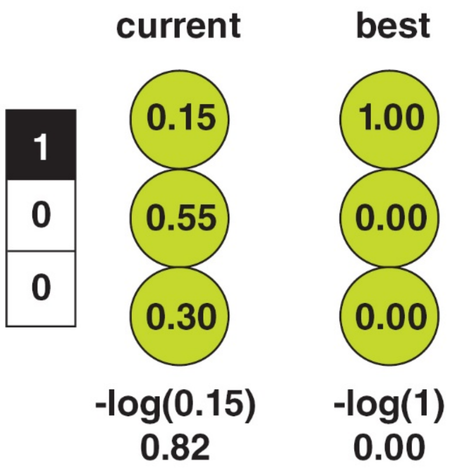


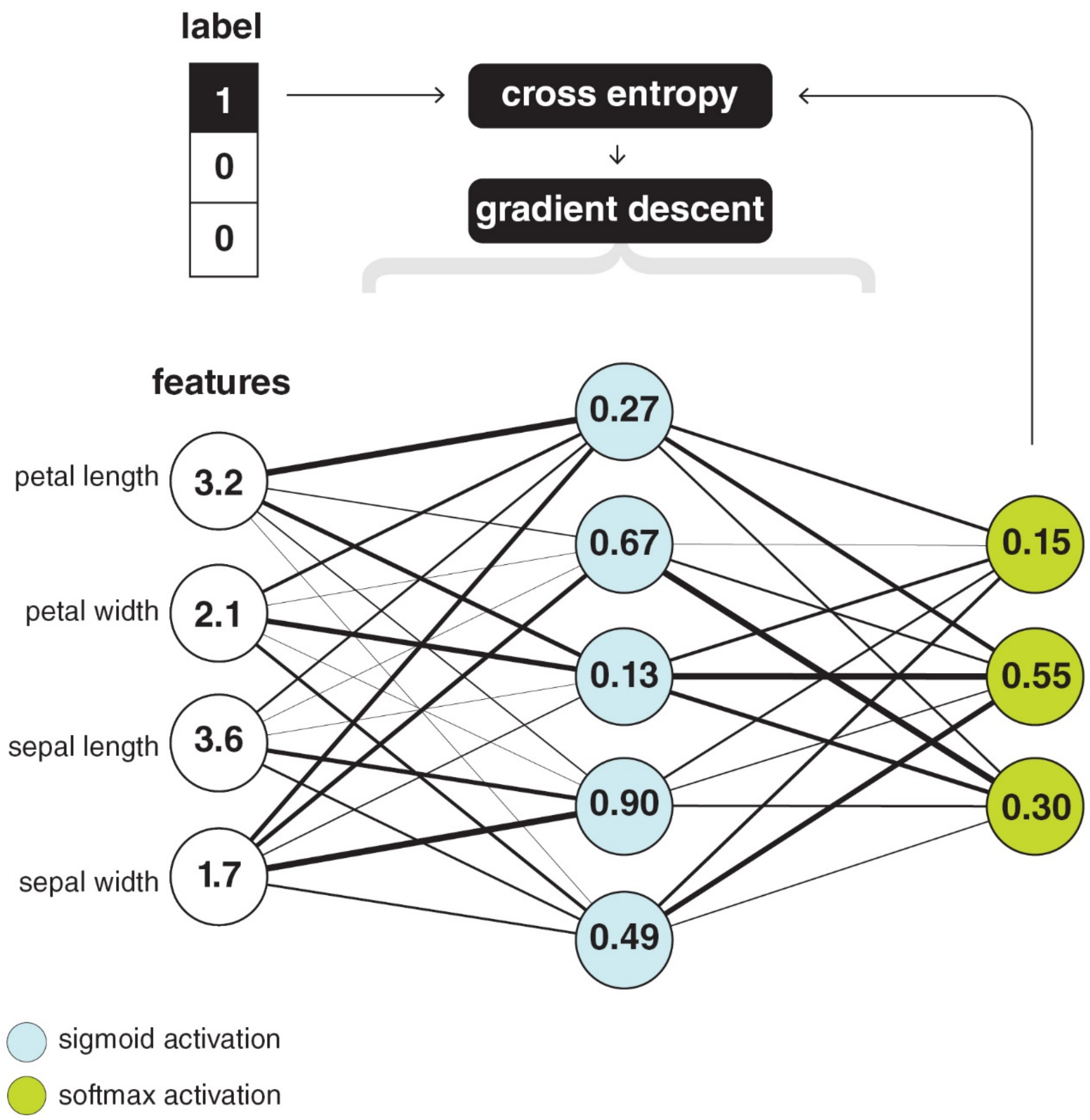
How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation



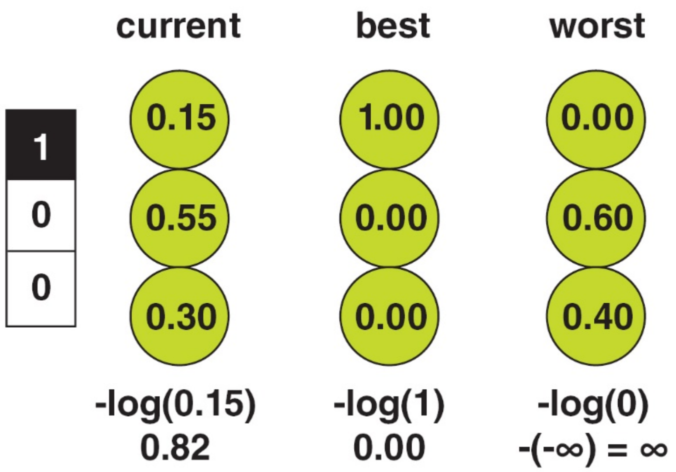


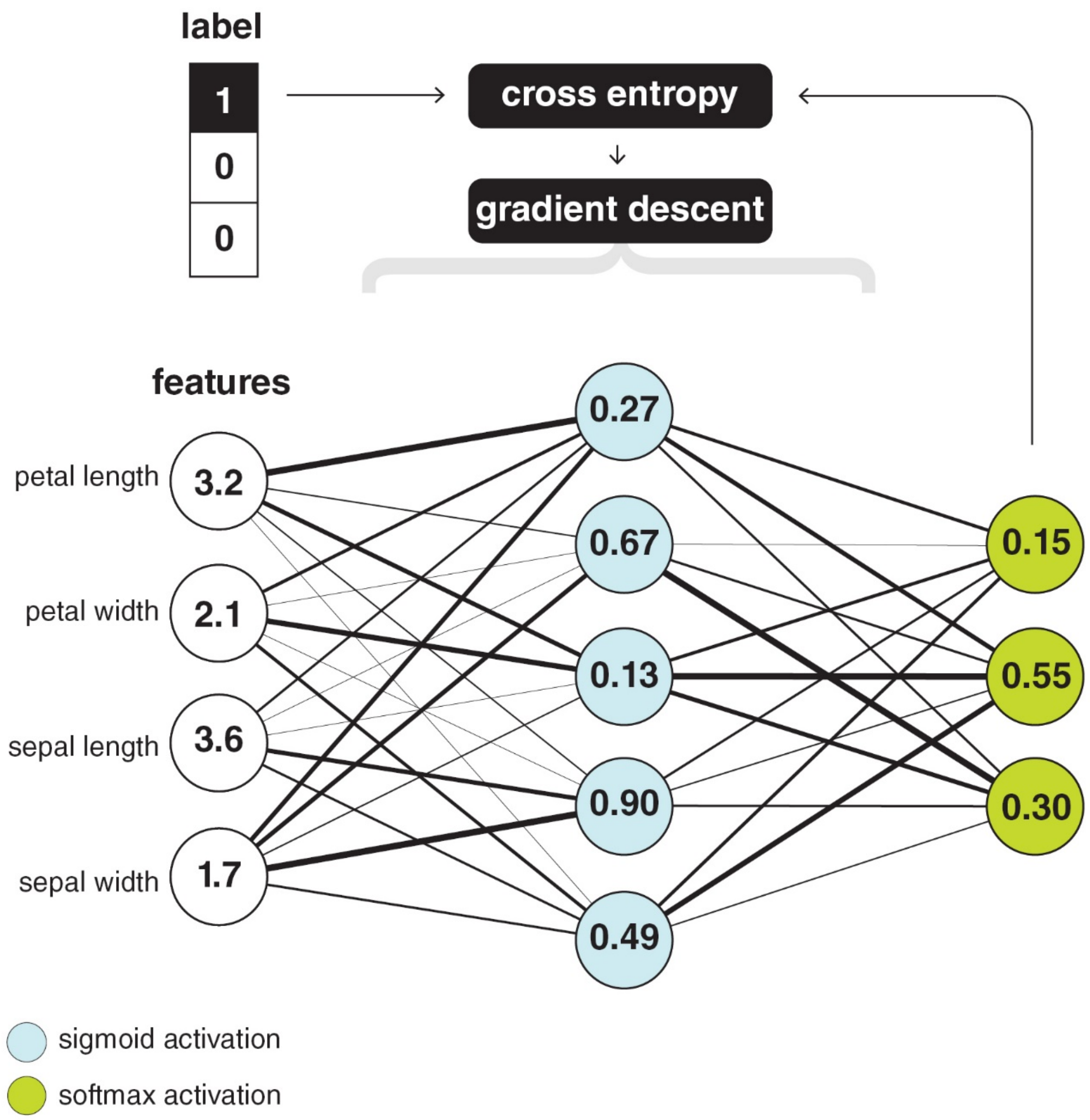
How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation





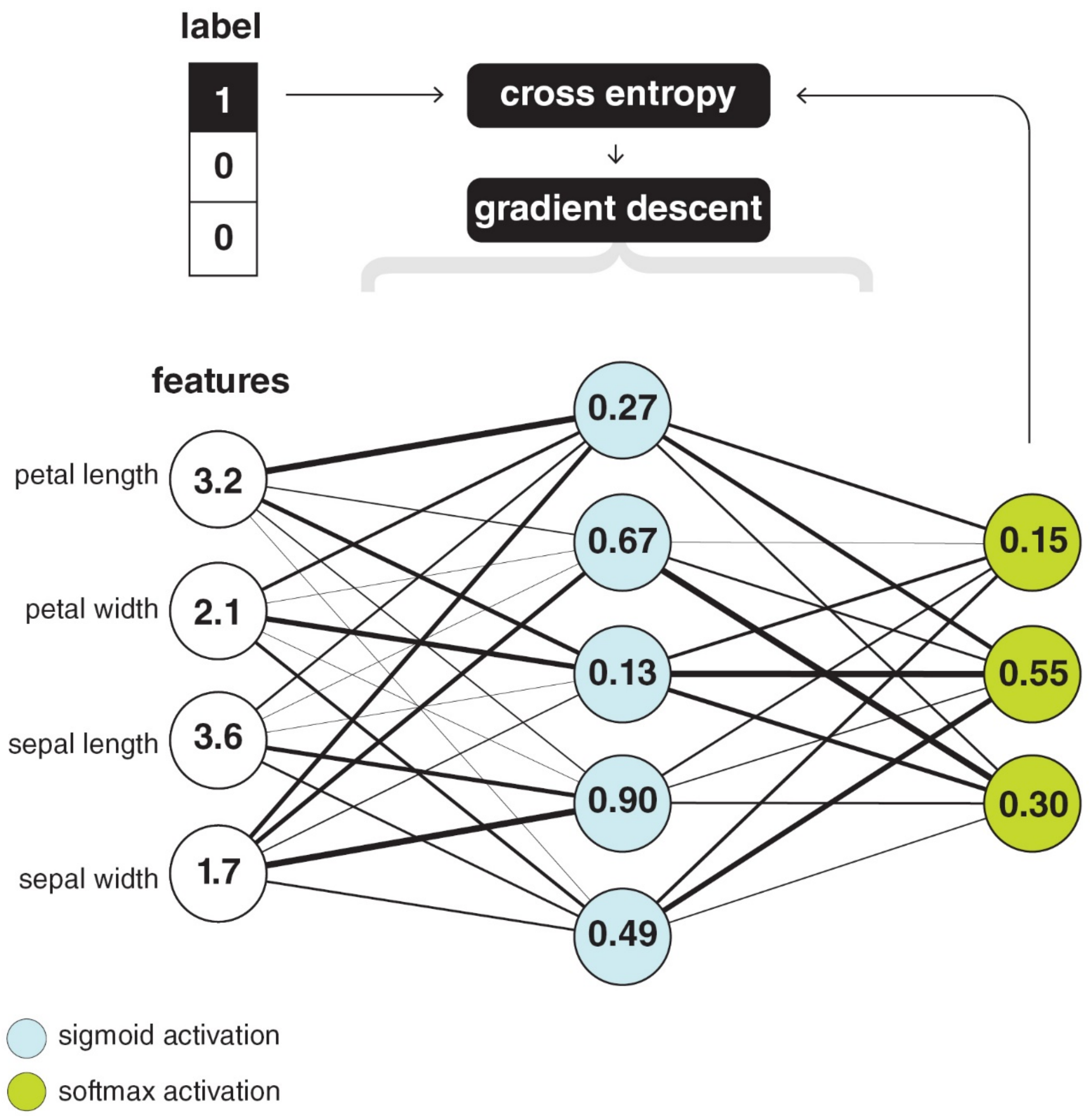
How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates

$$\Delta w \propto - \text{gradient}$$
$$= - \frac{\partial L}{\partial w}$$
$$= - \eta \frac{\partial L}{\partial w}$$



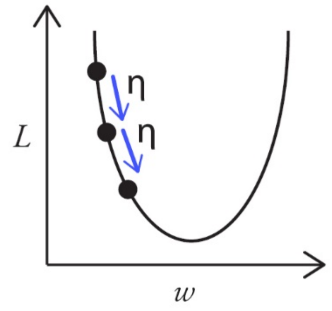
How Neural Networks Learn

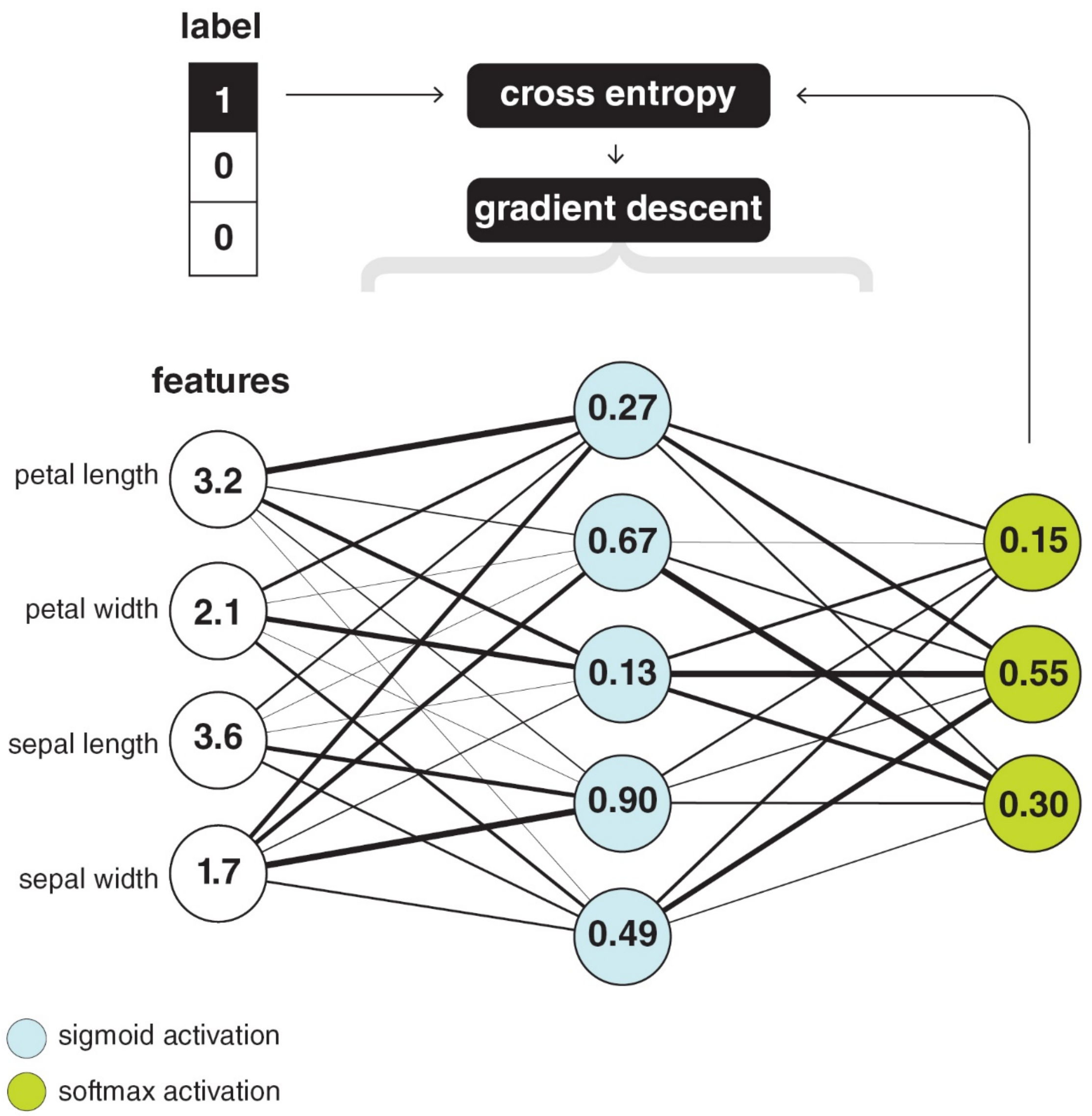
- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates

$$\Delta w \propto - \text{gradient} = -\frac{\partial L}{\partial w} = -\eta \frac{\partial L}{\partial w}$$





How Neural Networks Learn

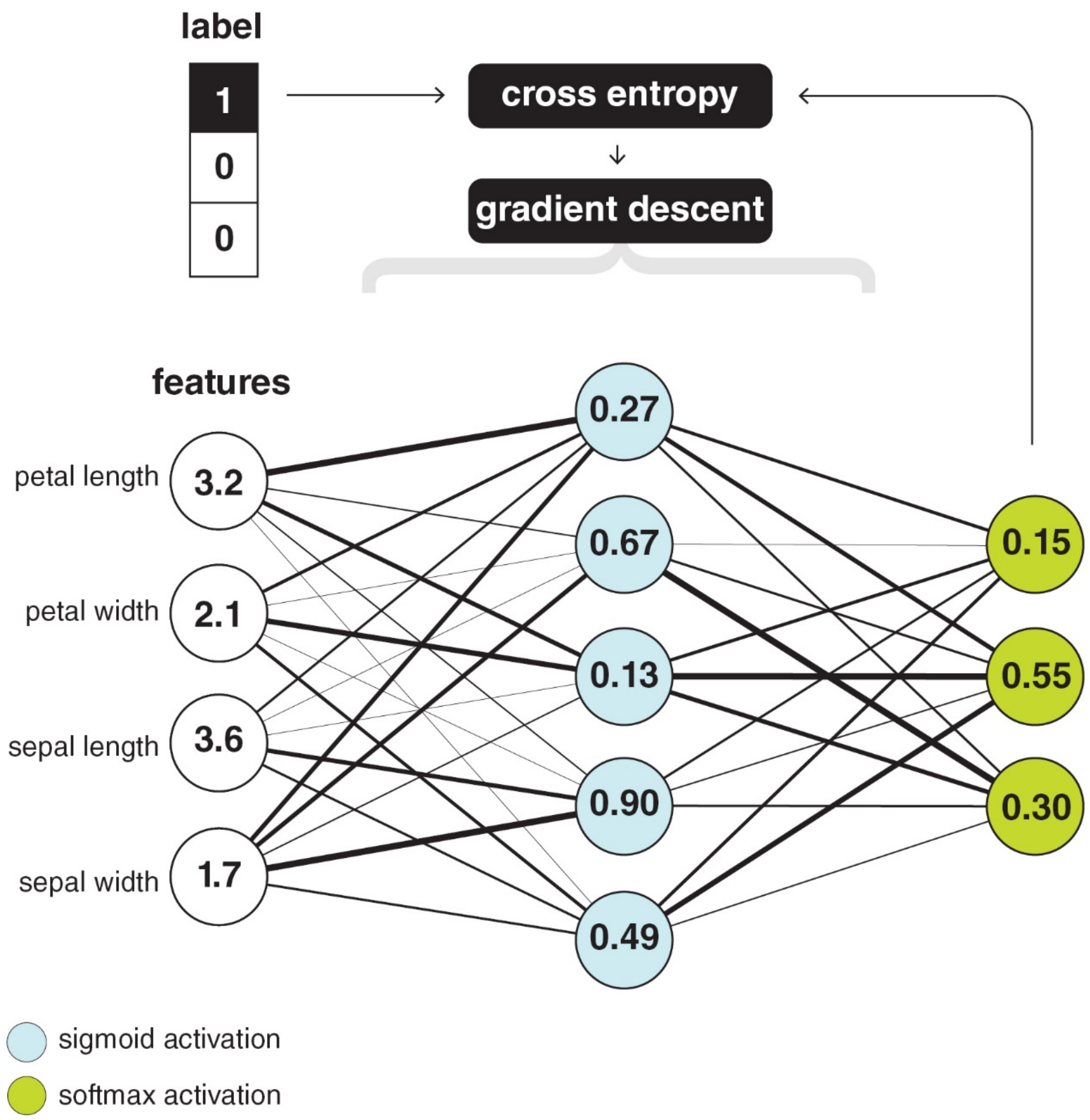
- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates

---

f(f(input))

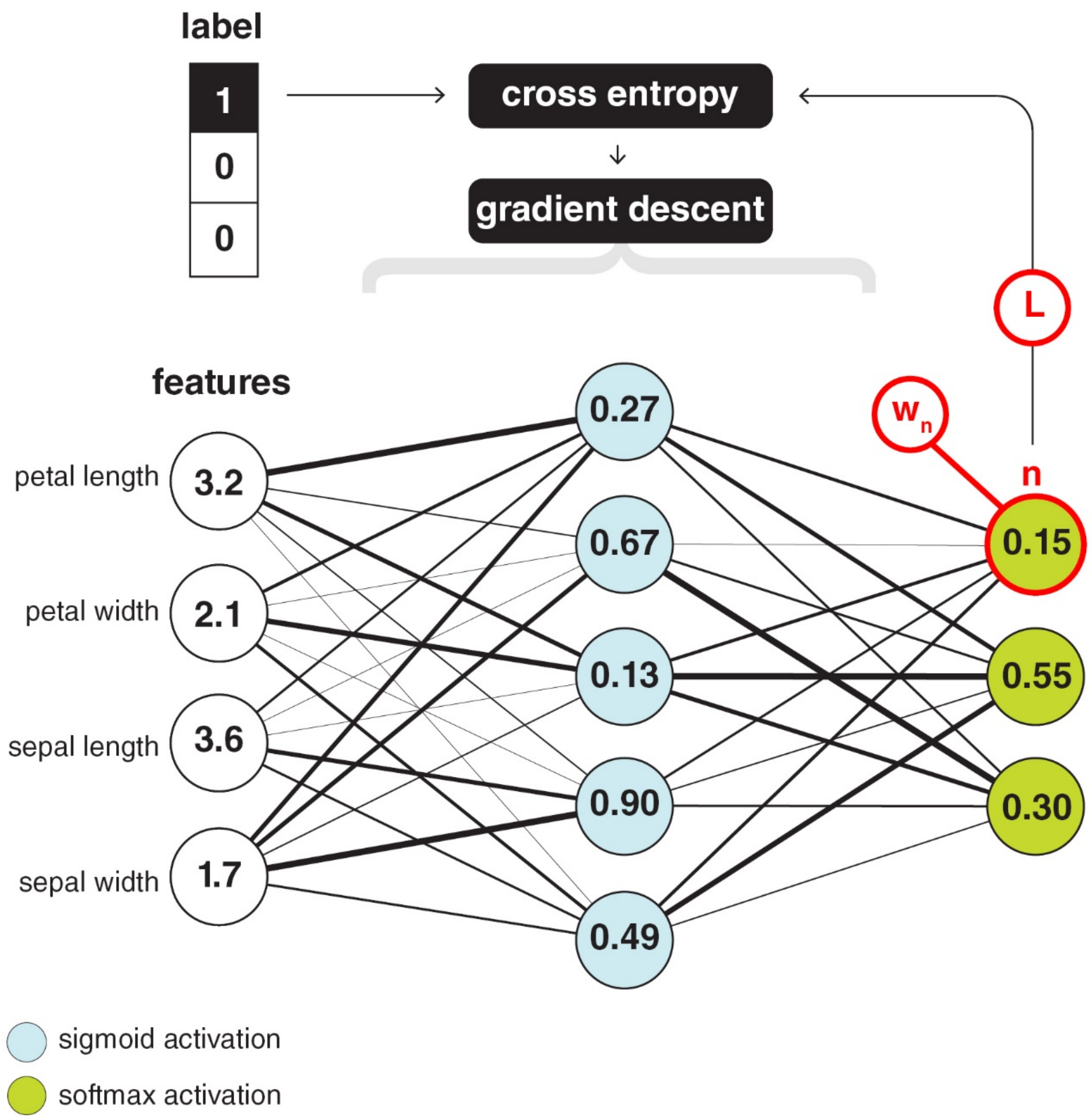


How Neural Networks Learn

✓ **Data:** iris dataset
✓ **Model:** 3-layer neural network
✓ **Loss:** cross entropy
✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates



How Neural Networks Learn

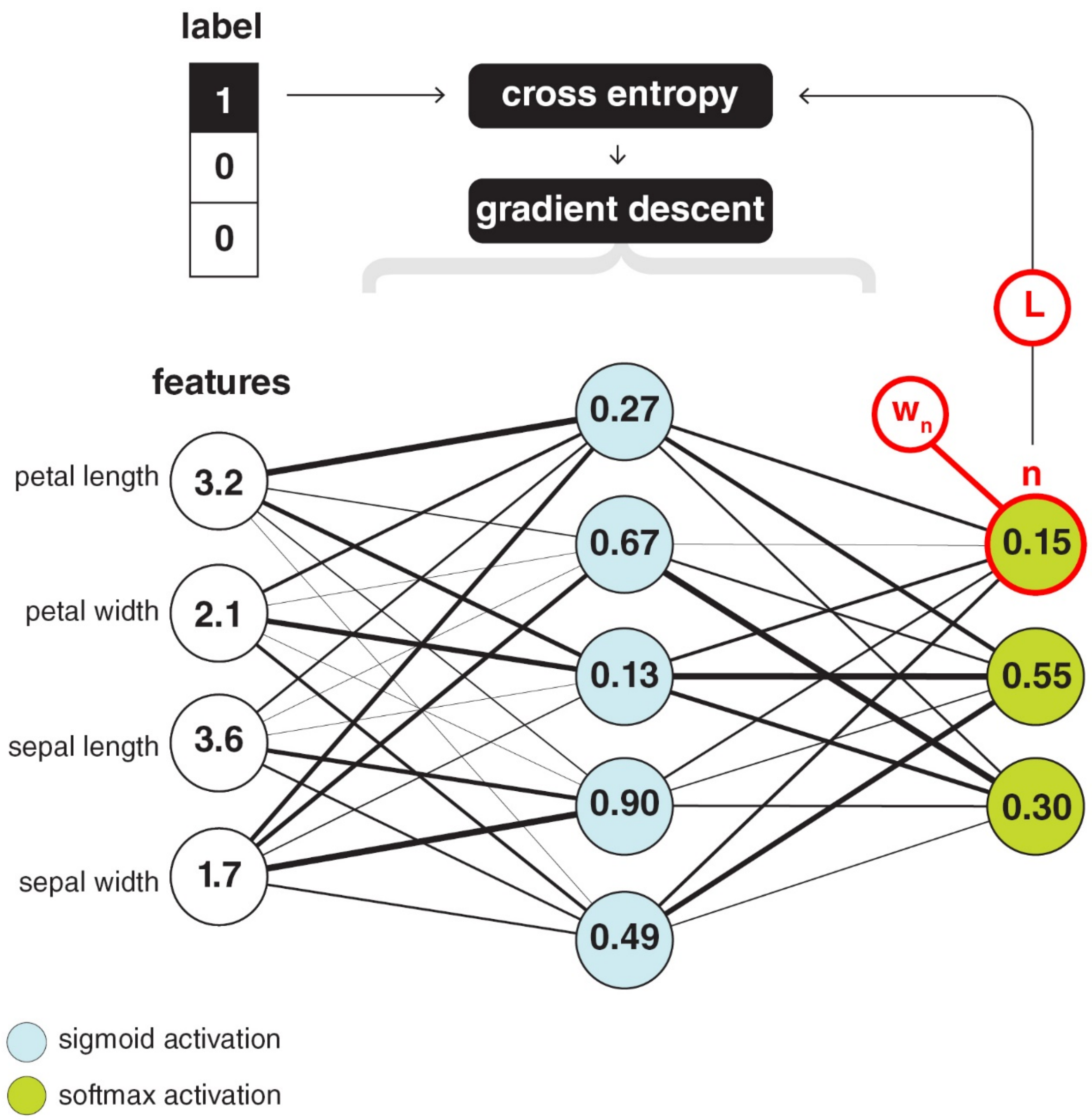
- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates

$$\Delta w_n = -\eta \frac{\partial L}{\partial w_n}$$

$$= -\eta \frac{\partial L}{\partial so} \frac{\partial so}{\partial n} \frac{\partial n}{\partial w_n}$$

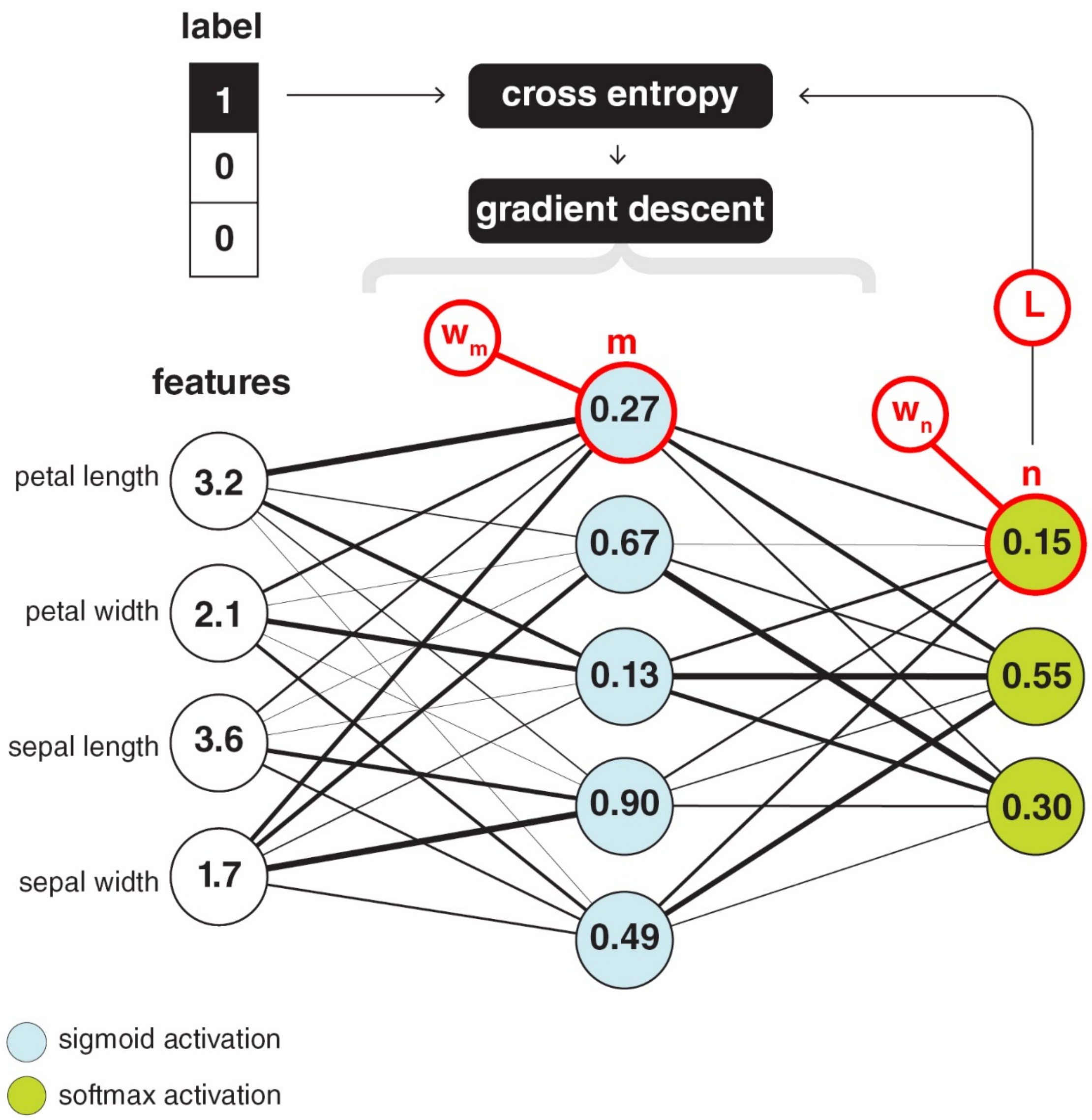


How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates



How Neural Networks Learn

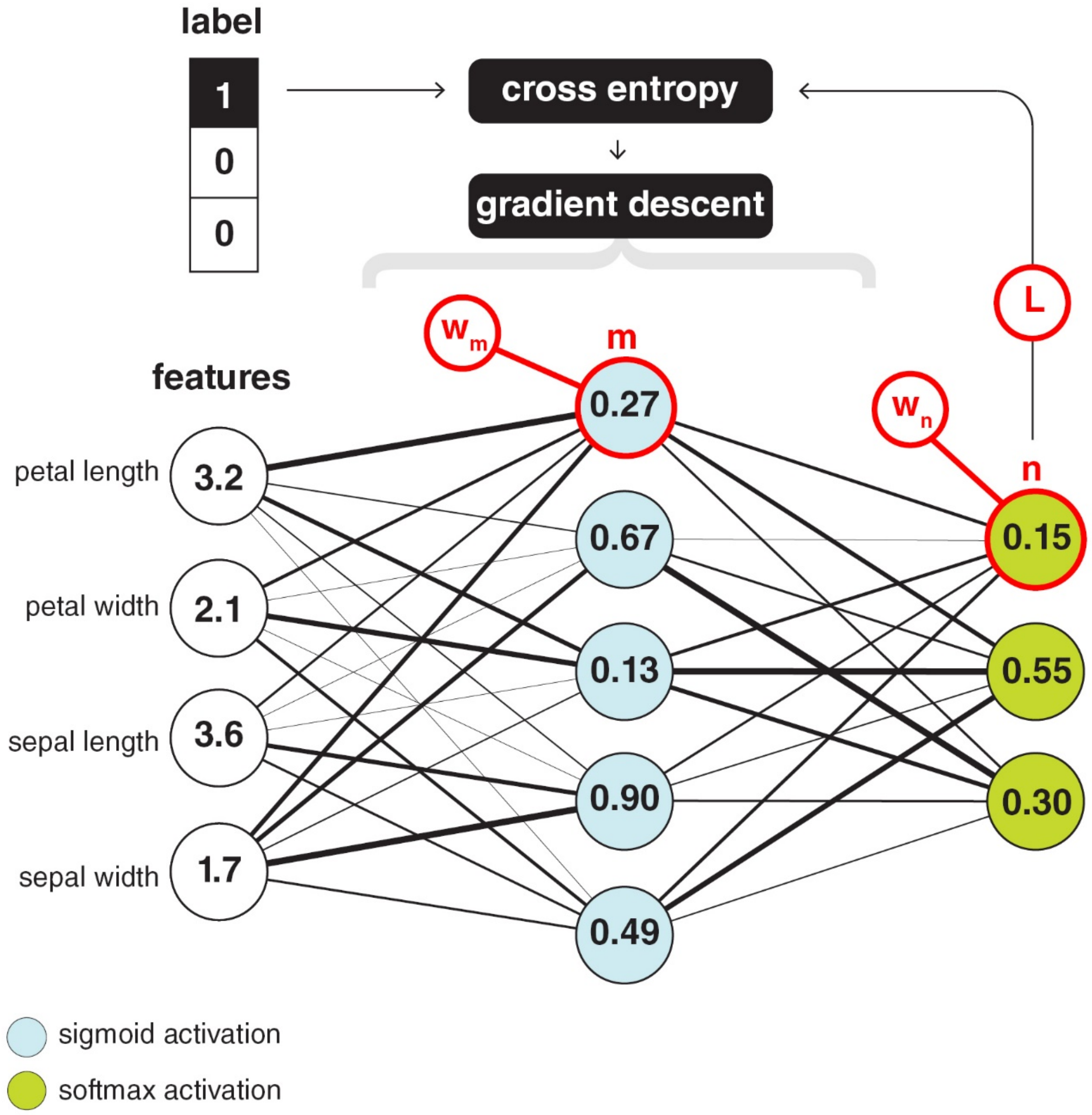
- ✔ **Data:** iris dataset
- ✔ **Model:** 3-layer neural network
- ✔ **Loss:** cross entropy
- ✔ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates

$$\Delta w_m = -\eta \frac{\partial L}{\partial w_m}$$

$$= -\eta \frac{\partial L}{\partial so} \frac{\partial so}{\partial n} \frac{\partial n}{\partial si} \frac{\partial si}{\partial m} \frac{\partial m}{\partial w_m}$$

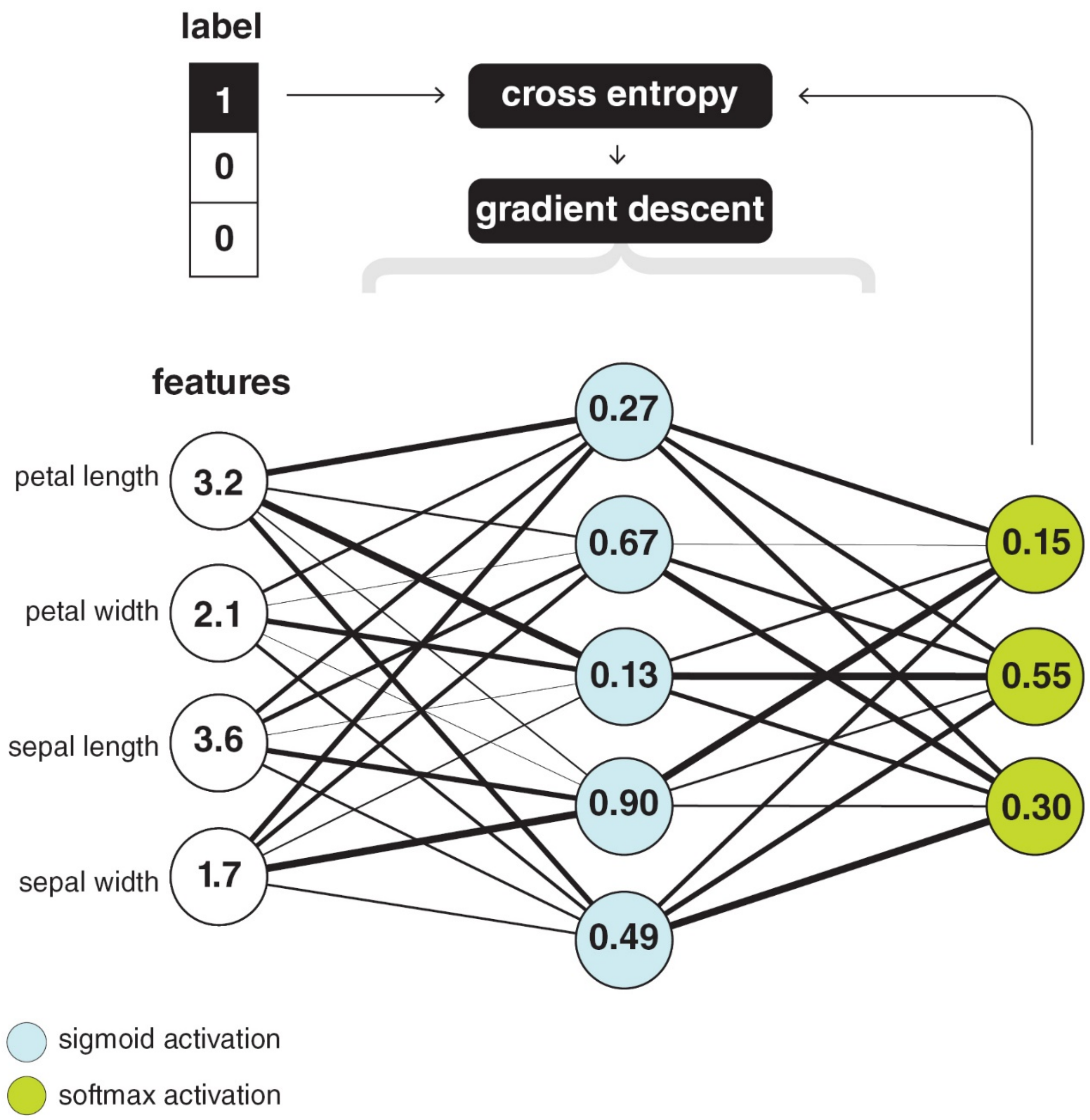


How Neural Networks Learn

- ✓ **Data:** iris dataset
- ✓ **Model:** 3-layer neural network
- ✓ **Loss:** cross entropy
- ✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates

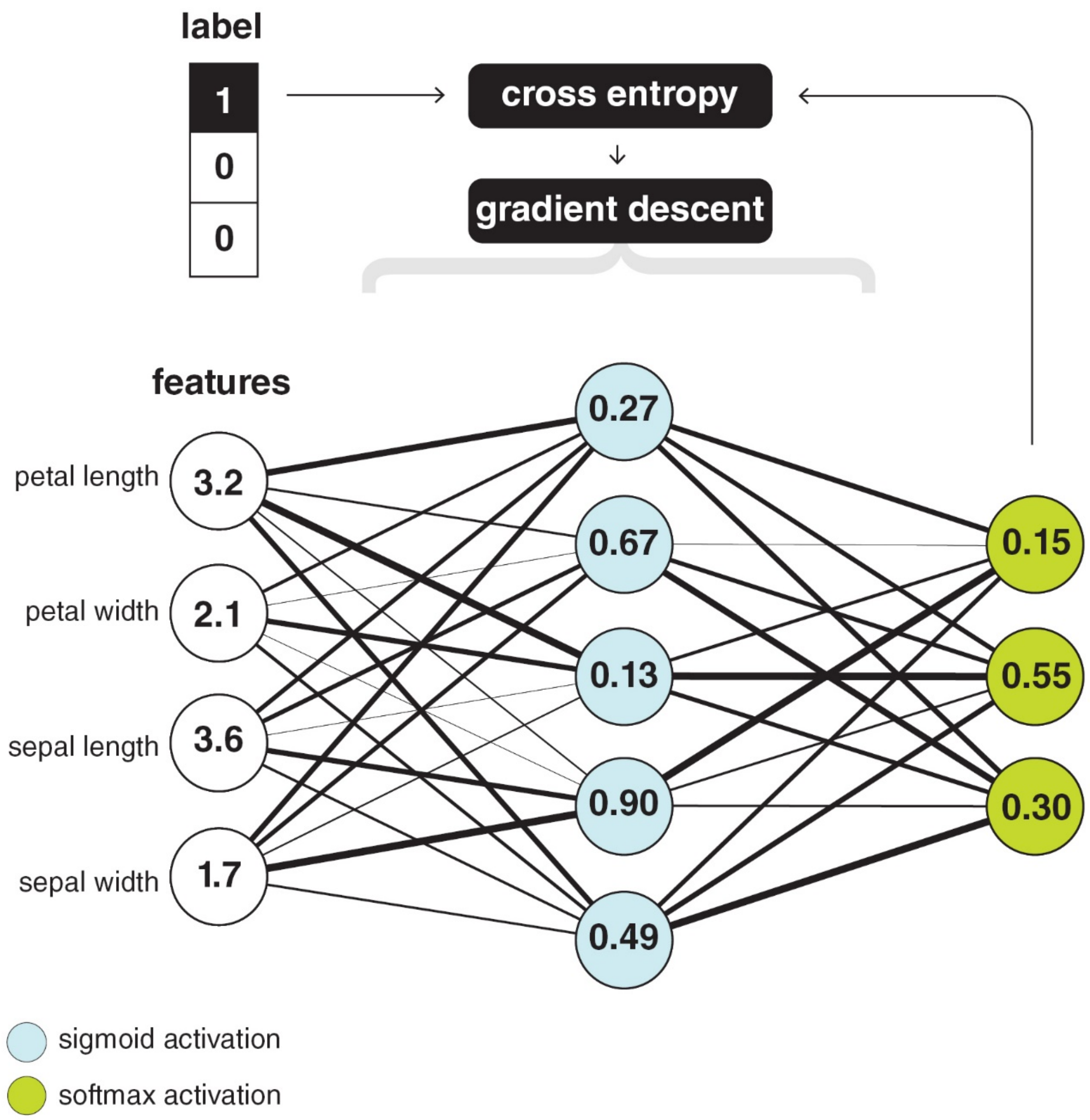


How Neural Networks Learn

✓ **Data:** iris dataset
✓ **Model:** 3-layer neural network
✓ **Loss:** cross entropy
✓ **Optimizer:** gradient descent

---

1. parameter initialization
2. data input
3. forward propagation
4. loss calculation
5. backpropagation + updates
6. repeat 2,3,4, & 5



Gradient Descent Flavors

vanilla gradient descent - entire dataset
stochastic gradient descent - random batch of samples (IID)
online gradient descent - (need not be IID)

Gradient Descent Flavors

vanilla gradient descent - entire dataset

stochastic gradient descent - random batch of samples (IID)

online gradient descent - (need not be IID)

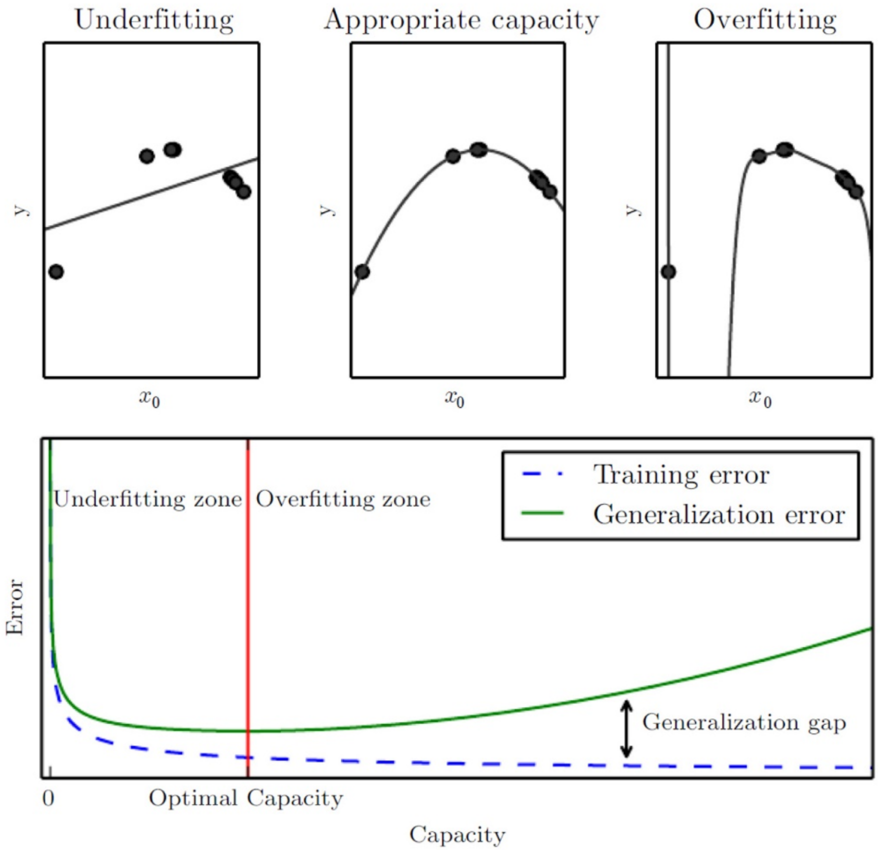
learning rate | **batch size** | **# of epochs**

What is the intuition behind neural networks?

How do neural networks learn?

How to train neural networks?

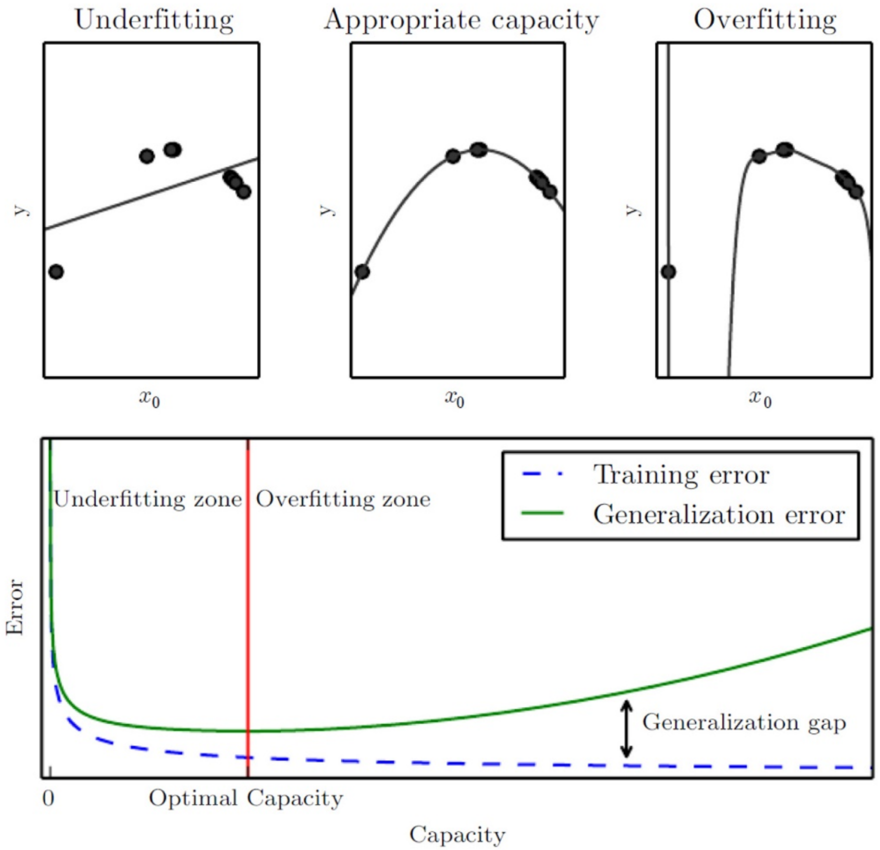
The Perfect Fit



Ian Goodfellow, Yoshua Bengio & Aaron Courville

Deep Learning
MIT Press - 2016

The Perfect Fit



parameters vs **hyperparameters**

Ian Goodfellow, Yoshua Bengio & Aaron Courville

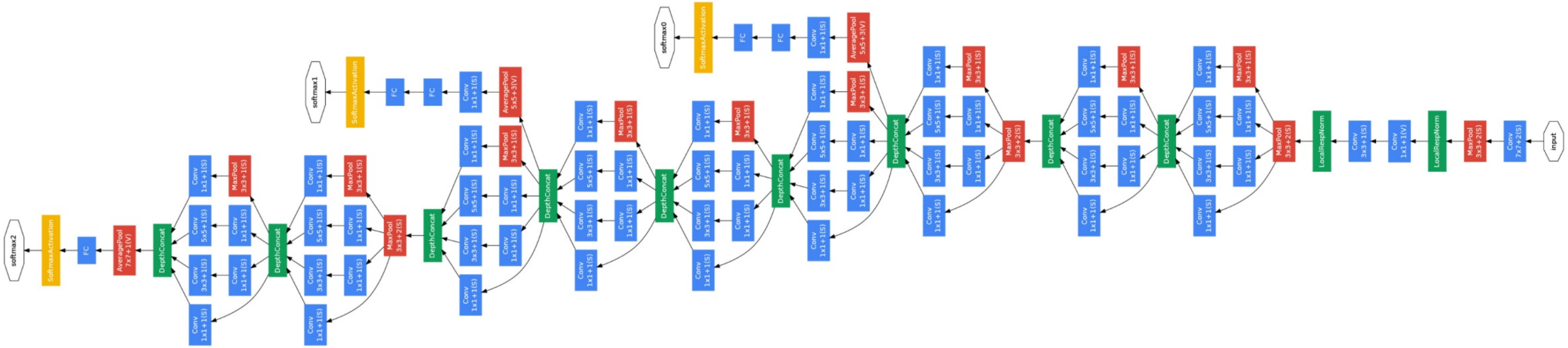
Deep Learning
MIT Press - 2016

Hyperparameters

model-specific	vs	optimizer-specifc
architecture		learning rate
activations		batch size
initializations		# of epochs
loss functions		
optimizers		
regularizers		

Architecture

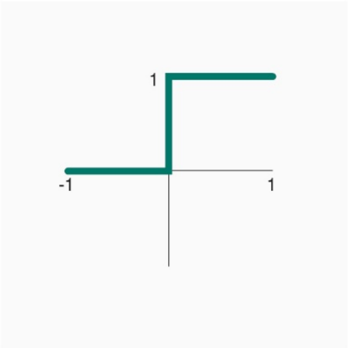
of layers
# of units/layer



Christian Szegedy, Wei Liu, Yangqing Jia, et al.

Going Deeper with Convolutions (GoogleNet/Inception)
CVPR - 2015

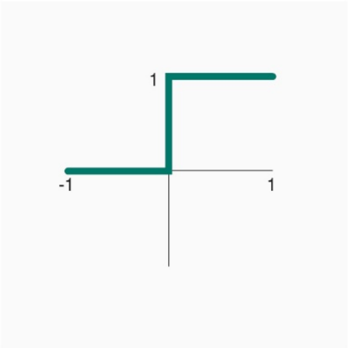
Activations



step

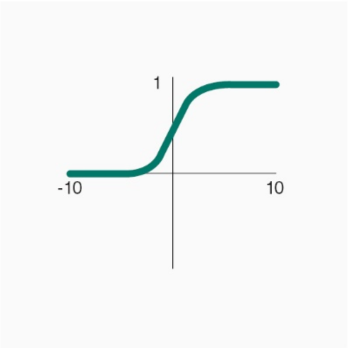
✗ non-differentiable

Activations



step

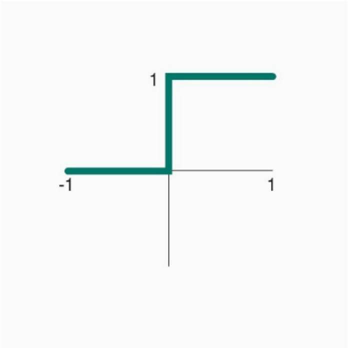
✗ non-differentiable



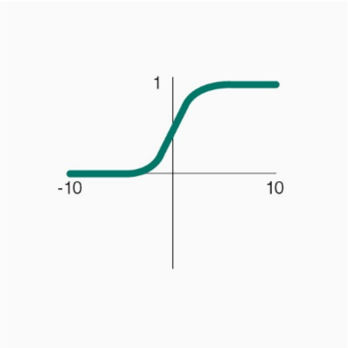
sigmoid

✓ smooth + step-like
✓ good activations close to 0
✓ activations are bound 0~1
✗ vanishing gradients

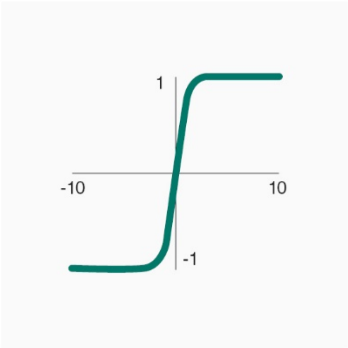
Activations



step
✗ non-differentiable

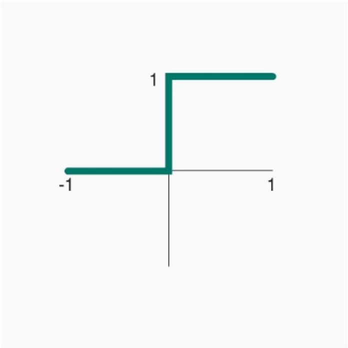


sigmoid
✓ smooth + step-like
✓ good activations close to 0
✓ activations are bound 0~1
✗ vanishing gradients



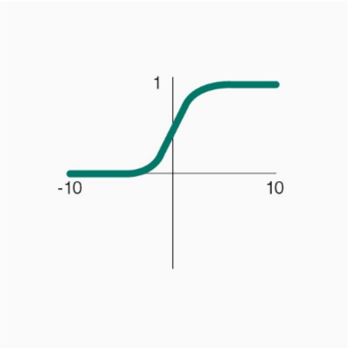
tanh
✓ scaled sigmoid
✓ stronger activations
✗ vanishing gradients

Activations



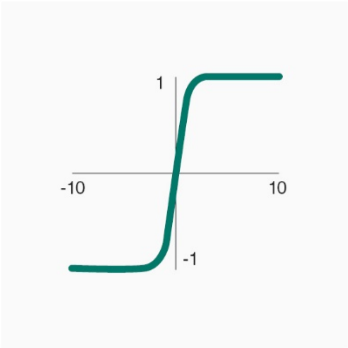
step

✗ non-differentiable



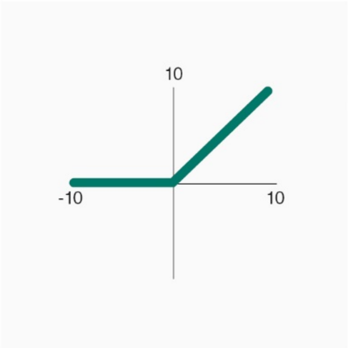
sigmoid

✓ smooth + step-like
✓ good activations close to 0
✓ activations are bound 0~1
✗ vanishing gradients



tanh

✓ scaled sigmoid
✓ stronger activations
✗ vanishing gradients



ReLU

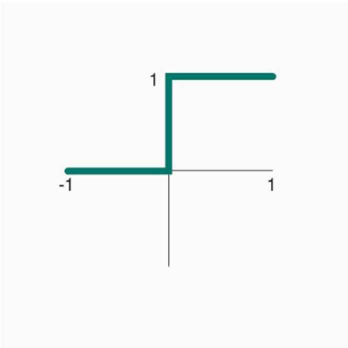
✓ sparse activations
✓ efficient
✗ dead nodes
✗ not bound

Alex Krizhevsky, Ilya Sutskever & Geoffrey E. Hinton

ImageNet Classification with Deep Convolutional Neural Networks

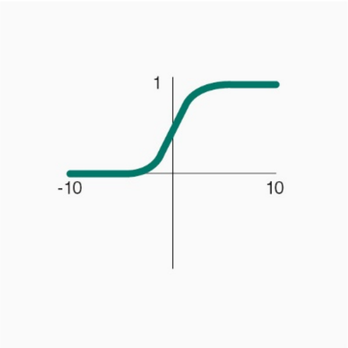
Advances in Neural Information Processing - NIPS 2012

Activations



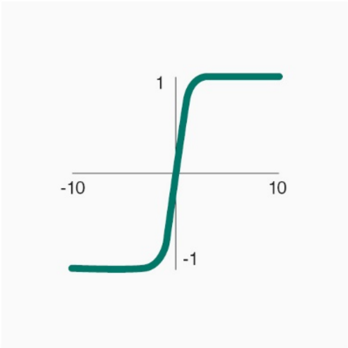
step

- ✗ non-differentiable



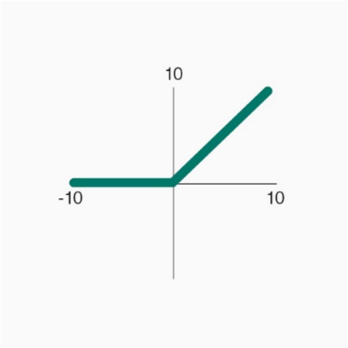
sigmoid

- ✓ smooth + step-like
- ✓ good activations close to 0
- ✓ activations are bound 0~1
- ✗ vanishing gradients



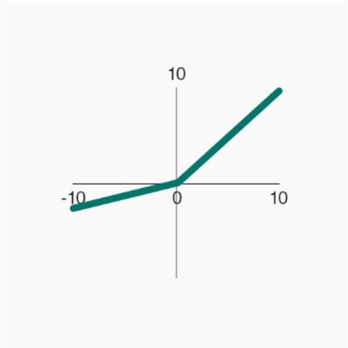
tanh

- ✓ scaled sigmoid
- ✓ stronger activations
- ✗ vanishing gradients



ReLU

- ✓ sparse activations
- ✓ efficient
- ✗ dead nodes
- ✗ not bound



Leaky ReLU

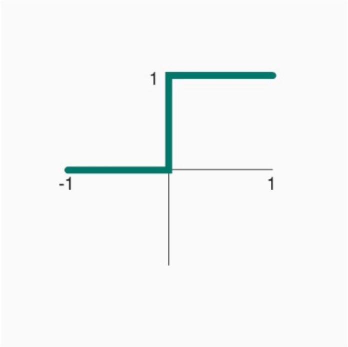
- ✓ no dead nodes

Kaiming He, Xiangyu Zhang, Shaoqing Ren & Jian Sun

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

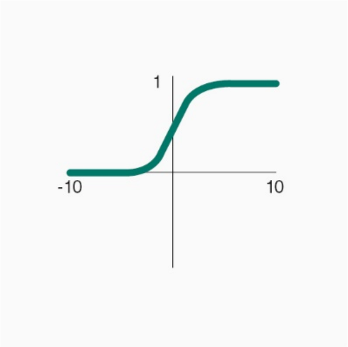
International Conference on Computer Vision - ICCV 2015

Activations



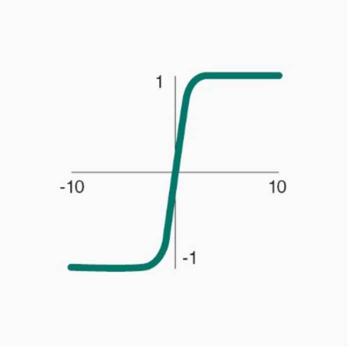
step

- ✗ non-differentiable



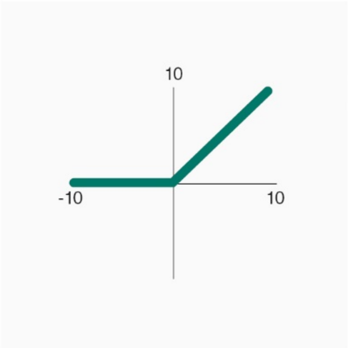
sigmoid

- ✓ smooth + step-like
- ✓ good activations close to 0
- ✓ activations are bound 0~1
- ✗ vanishing gradients



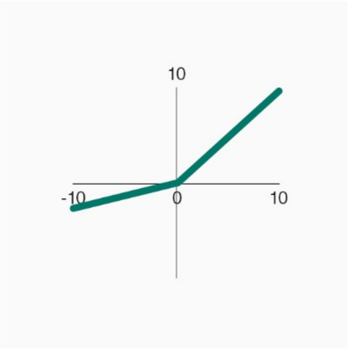
tanh

- ✓ scaled sigmoid
- ✓ stronger activations
- ✗ vanishing gradients



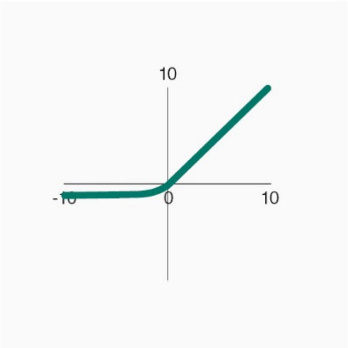
ReLU

- ✓ sparse activations
- ✓ efficient
- ✗ dead nodes
- ✗ not bound



Leaky ReLU

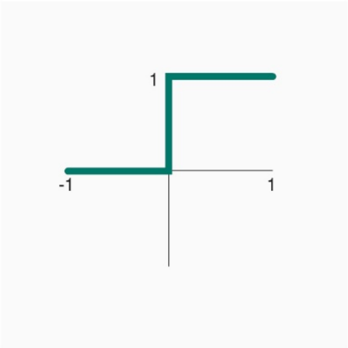
- ✓ no dead nodes



ELU

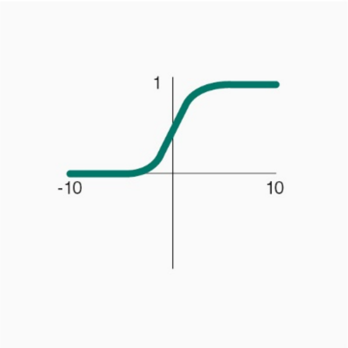
- ✓ robust to noise
- ✗ expensive

Activations



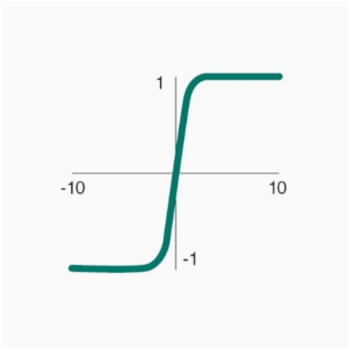
step

✗ non-differentiable



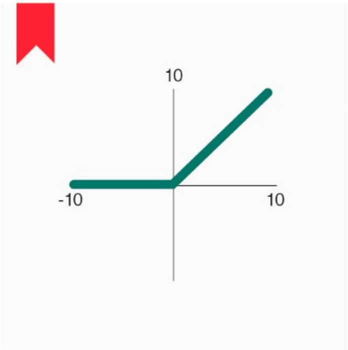
sigmoid

✓ smooth + step-like
✓ good activations close to 0
✓ activations are bound 0~1
✗ vanishing gradients



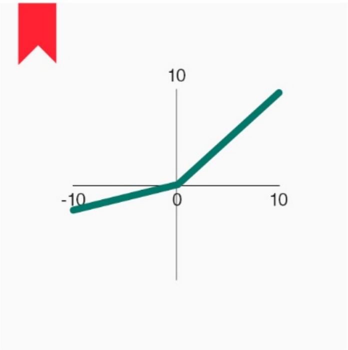
tanh

✓ scaled sigmoid
✓ stronger activations
✗ vanishing gradients



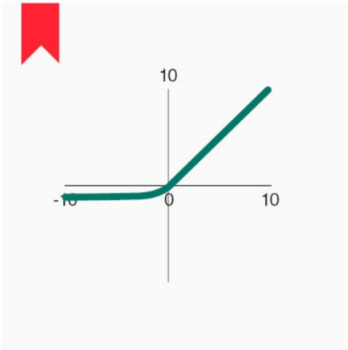
ReLU

✓ sparse activations
✓ efficient
✗ dead nodes
✗ not bound



Leaky ReLU

✓ no dead nodes



ELU

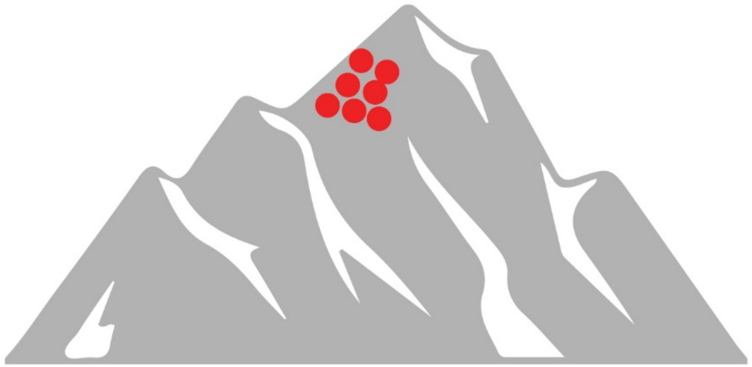
✓ robust to noise
✗ expensive

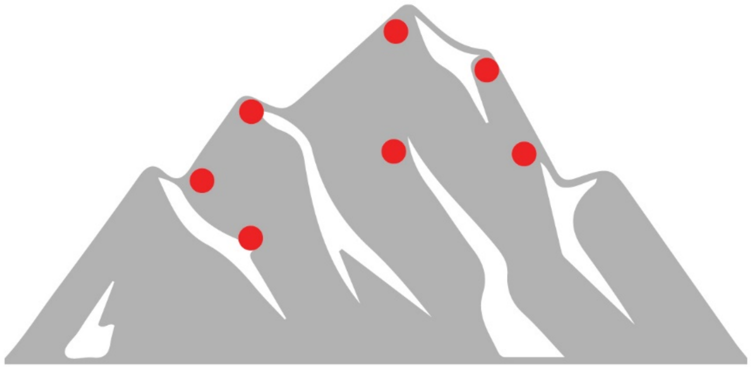
Initializations

0 - stuck at a saddle point

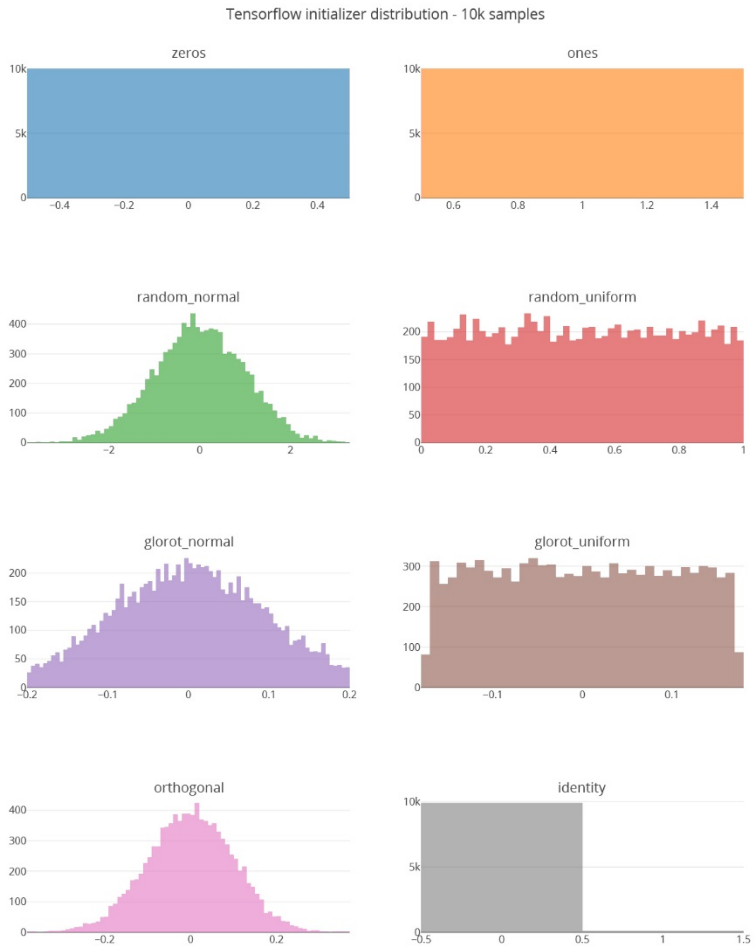
constants - difficult to break the symmetry

large random values - small gradients, slow convergence

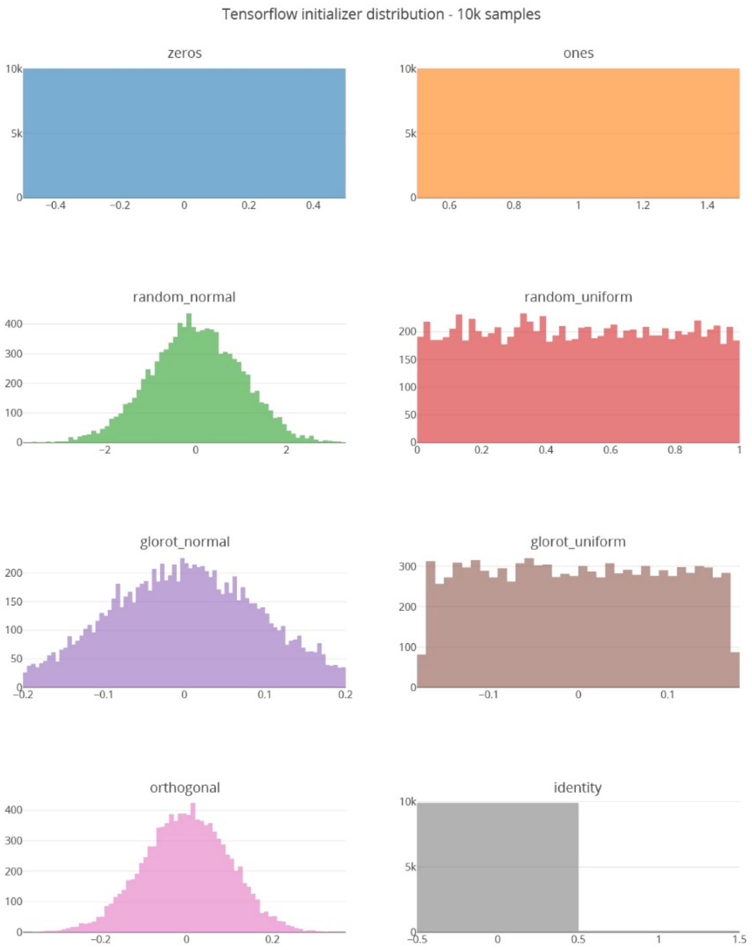


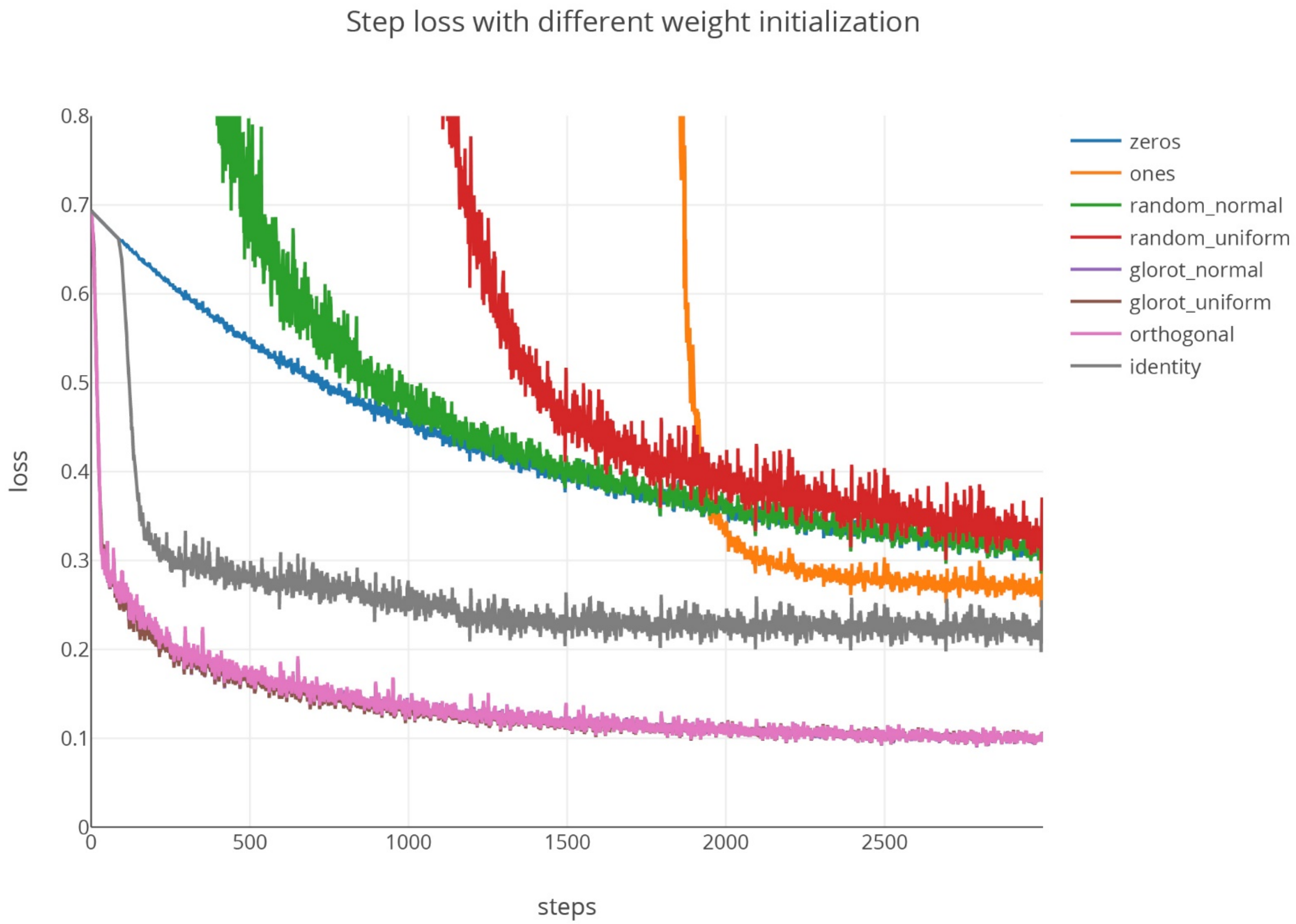


Initializations



Initializations





Initializations

Name	α	β	γ	Reference
Constant	$\alpha = 0$	$\beta = 0$	$\gamma \geq 0$	used by [ZF14]
→ Xavier/Glorot uniform	$\alpha = \sqrt{\frac{6}{n_{in}+n_{out}}}$	$\beta = 0$	$\gamma = 0$	[GB10]
→ Xavier/Glorot normal	$\alpha = 0$	$\beta = \left(\frac{2}{(n_{in}+n_{out})}\right)^2$	$\gamma = 0$	[GB10]
→ He	$\alpha = 0$	$\beta = \frac{2}{n_{in}}$	$\gamma = 0$	[HZRS15b]
Orthogonal	—	—	$\gamma = 0$	[SMG13]
LSUV	—	—	$\gamma = 0$	[MM15]

Table B.2.: Weight initialization schemes of the form $w \sim \alpha \cdot \mathcal{U}[-1, 1] + \beta \cdot \mathcal{N}(0, 1) + \gamma$. n_{in}, n_{out} are the number of units in the previous layer and the next layer. Typically, biases are initialized with constant 0 and weights by one of the other schemes to prevent unit-coadaptation. However, dropout makes it possible to use constant initialization for all parameters.
LSUV and Orthogonal initialization cannot be described with this simple pattern.

Martin Thoma

Analysis and Optimization of Convolutional Neural Network Architectures
https://arxiv.org/pdf/1707.09725.pdf

Loss Functions

regression - mean squared error

multiclass classification - categorical cross entropy

pixel classification - dice/ Wasserstein dice coefficient

Optimizers

stochastic gradient descent + momentum

Optimizers

stochastic gradient descent + momentum

adaptive gradient (AdaGrad)

John Duchi, Elad Hazan & Yoram Singer

Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

Journal of Machine Learning Research - 2011

Optimizers

stochastic gradient descent + momentum

adaptive gradient (AdaGrad)

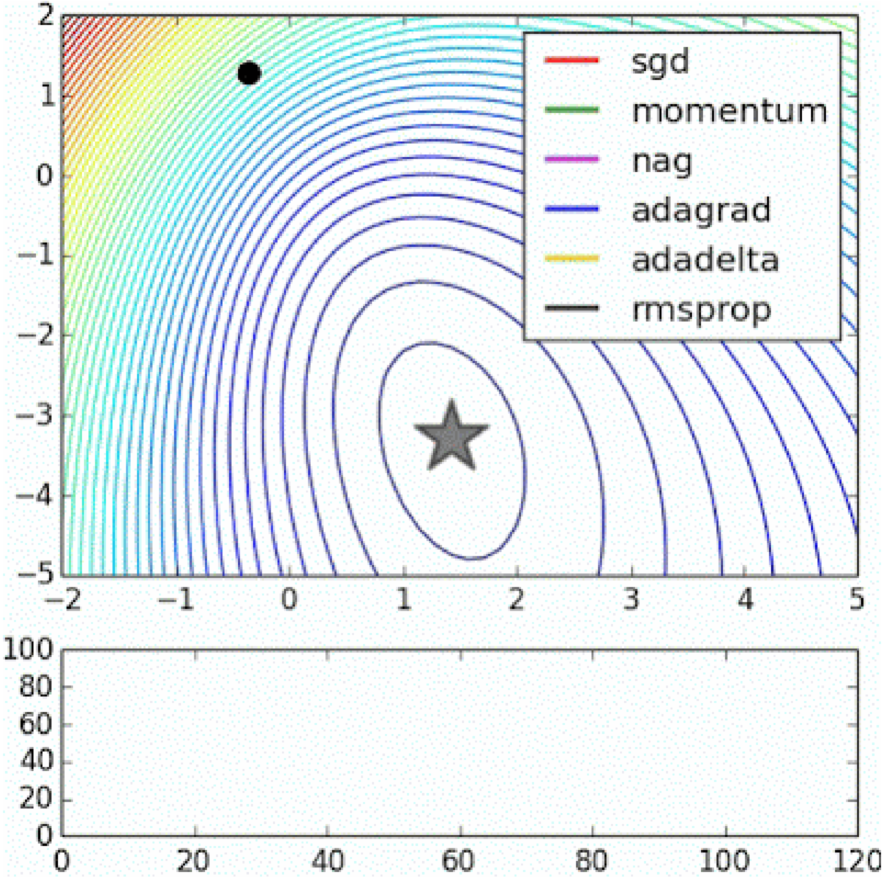
root mean square propagation (RMSProp)

Geoffrey Hinton

Coursera: Neural Networks for Machine Learning - Lecture 6

http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Optimizers



https://imgur.com/a/Hqolp

Regularizers

L1, L2 regularization

$$\mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2}|W|$$

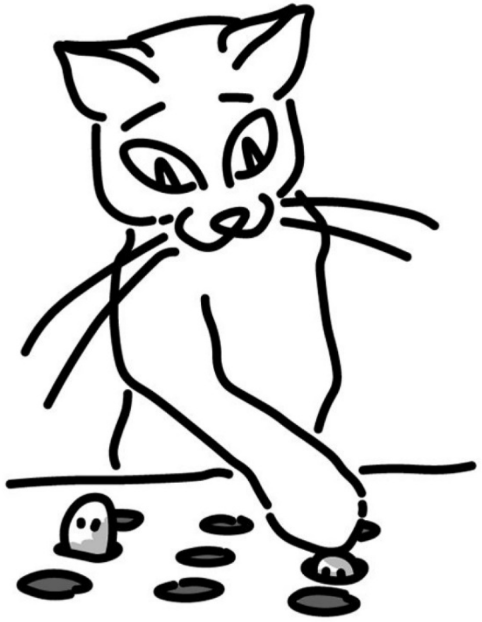
$$\mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2}W^2$$

Regularizers

L1, L2 regularization

$$\mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2}|W|$$

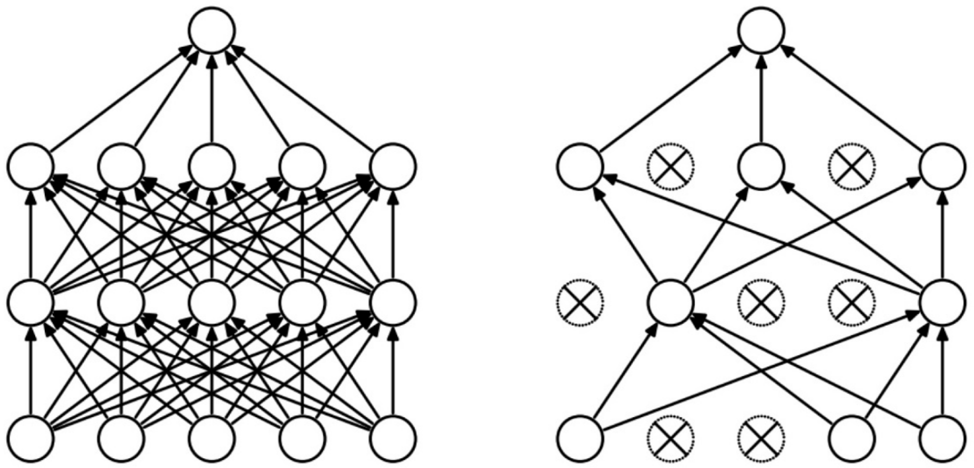
$$\mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2}W^2$$



https://alistapart.com/article/accessibility-whack-a-mole by Eleanor Ratliff

Regularizers

dropout



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

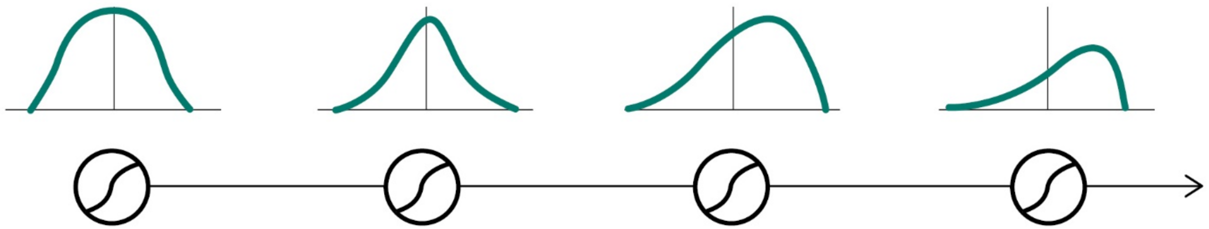
Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever & Ruslan Salakhutdinov

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Journal of Machine Learning Research - 2014

Regularizers

batch normalization



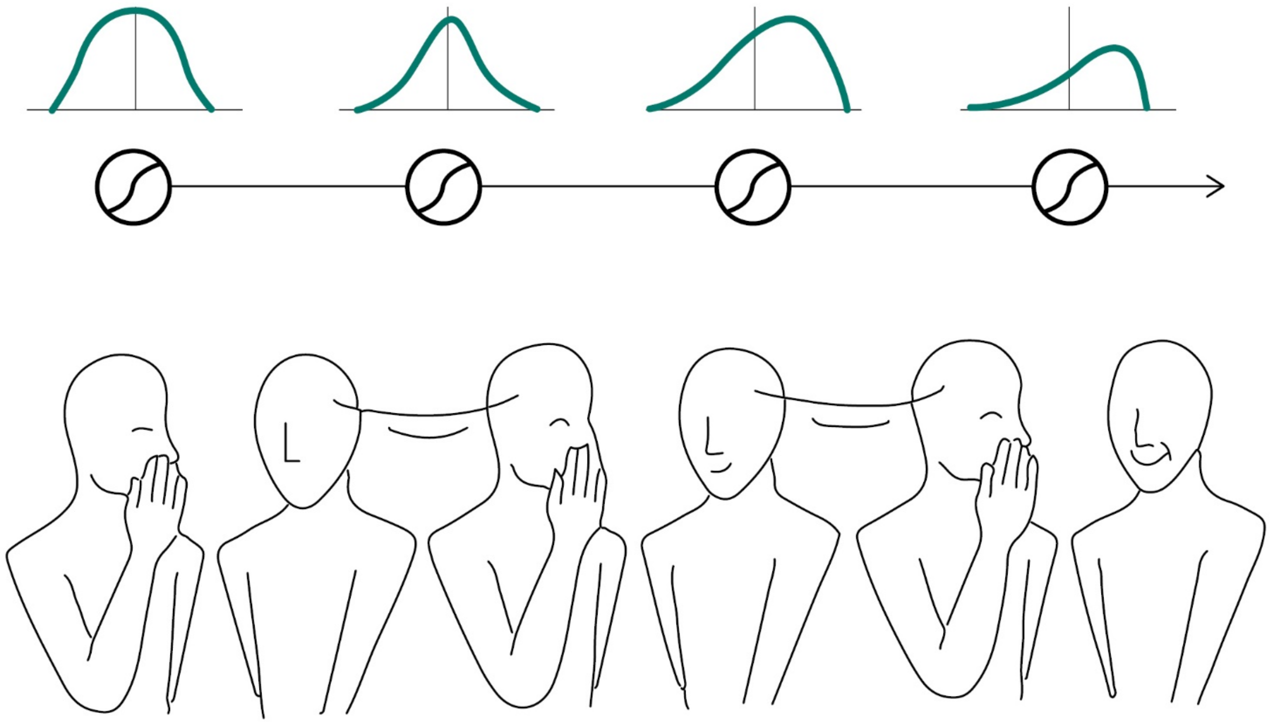
Sergey Ioffe & Christian Szegedy

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Journal of Machine Learning Research - 2014

Regularizers

batch normalization



Sergey Ioffe & Christian Szegedy

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Journal of Machine Learning Research - 2014

Optimizer-specifc Hyperparameters

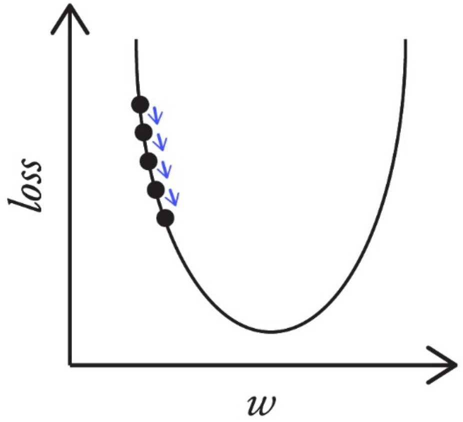
learning rate

0.1, 0.01, 0.001, 0.0001,...

Optimizer-specifc Hyperparameters

learning rate

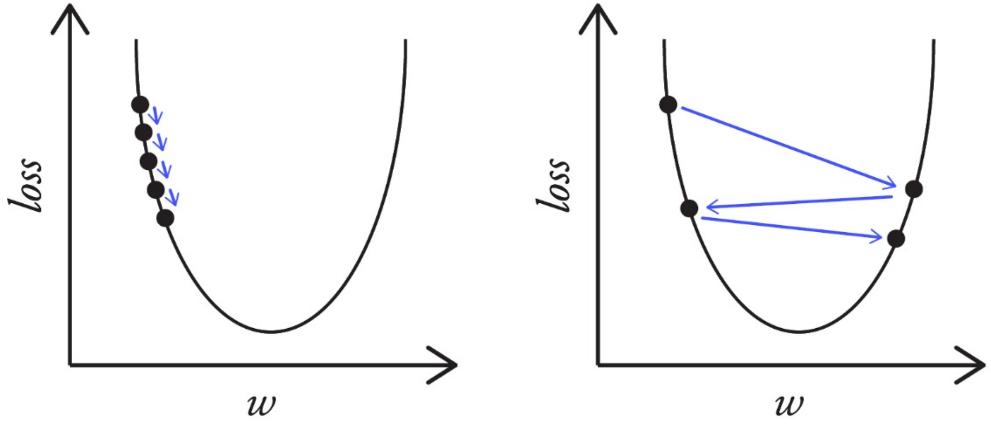
0.1, 0.01, 0.001, 0.0001,...



Optimizer-specifc Hyperparameters

learning rate

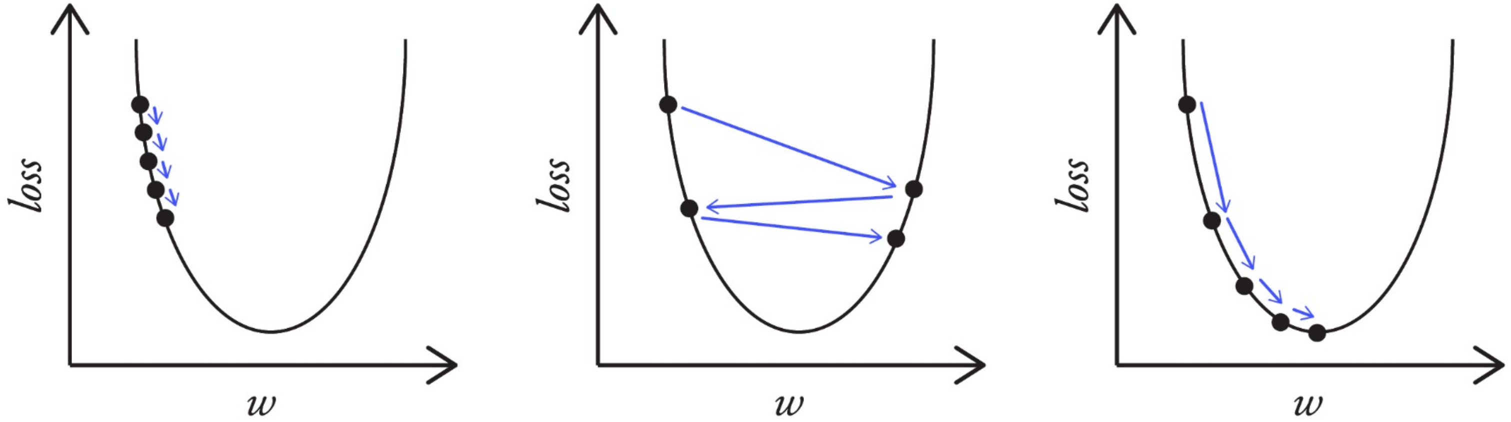
0.1, 0.01, 0.001, 0.0001,...



Optimizer-specifc Hyperparameters

learning rate

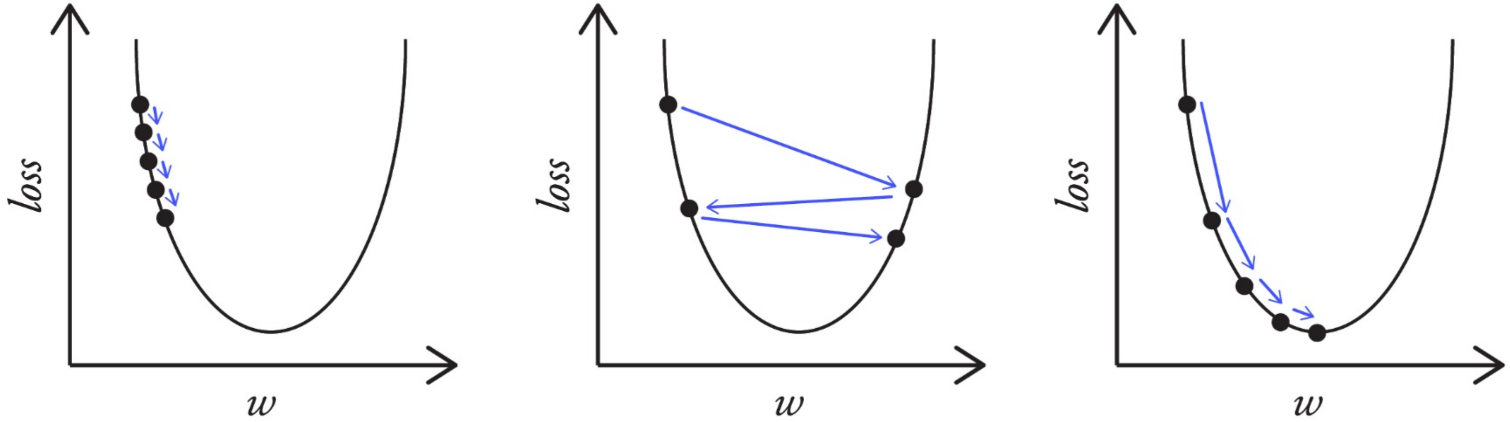
0.1, 0.01, 0.001, 0.0001,...



Optimizer-specifc Hyperparameters

learning rate

0.1, 0.01, 0.001, 0.0001,...



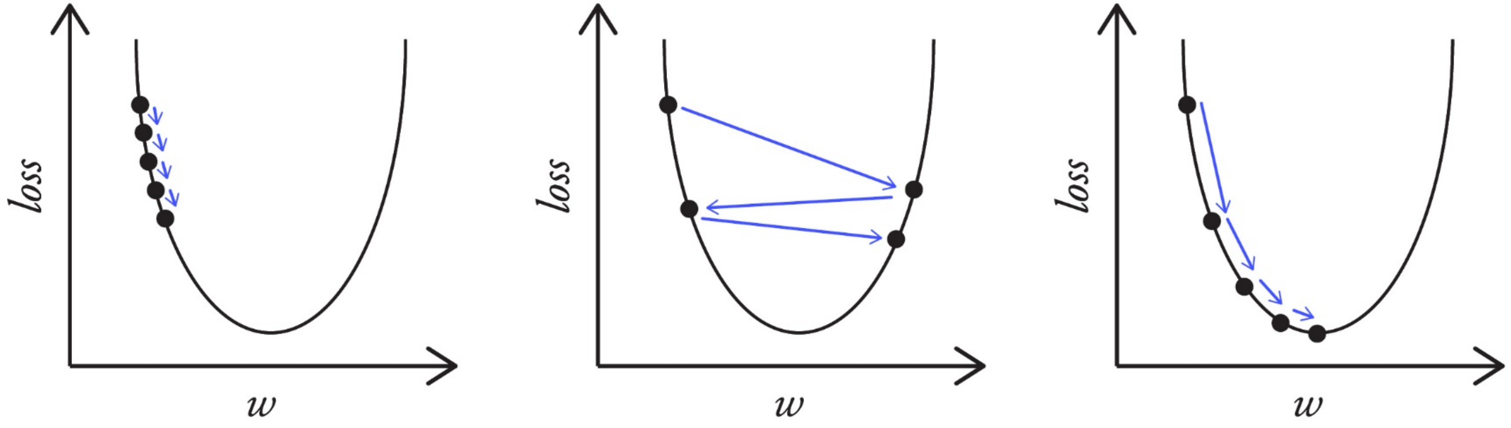
batch size

16, 32, 64, 128,...

Optimizer-specifc Hyperparameters

learning rate

0.1, 0.01, 0.001, 0.0001,...



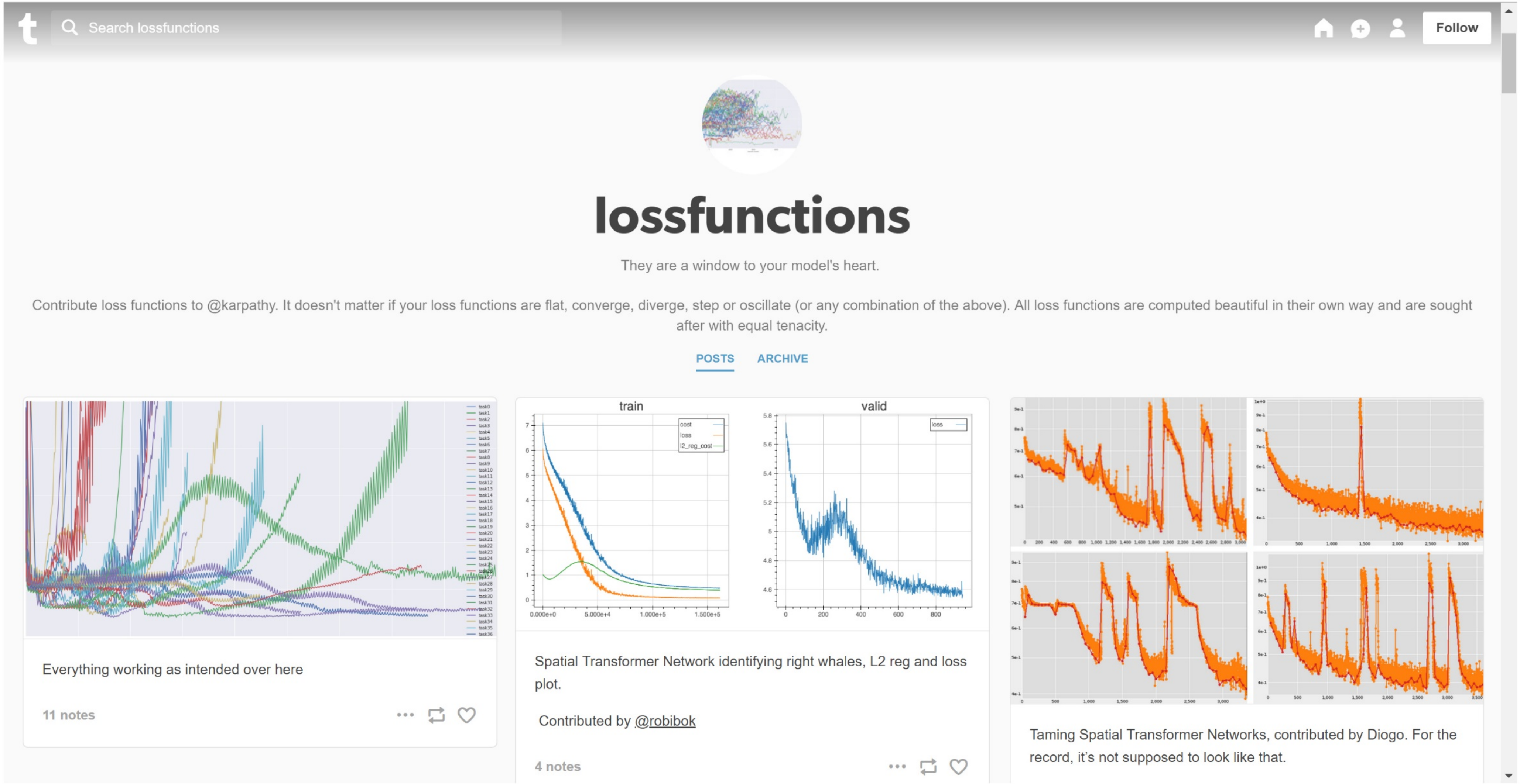
batch size

16, 32, 64, 128,...

# of epochs

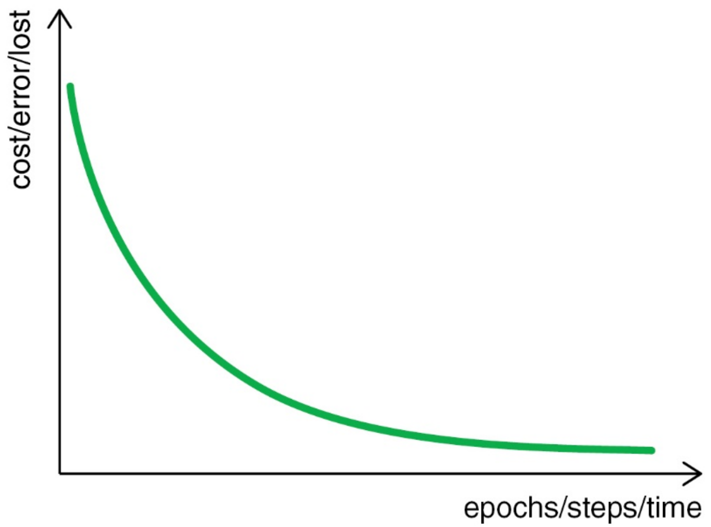
early stopping

Babysitting your Network

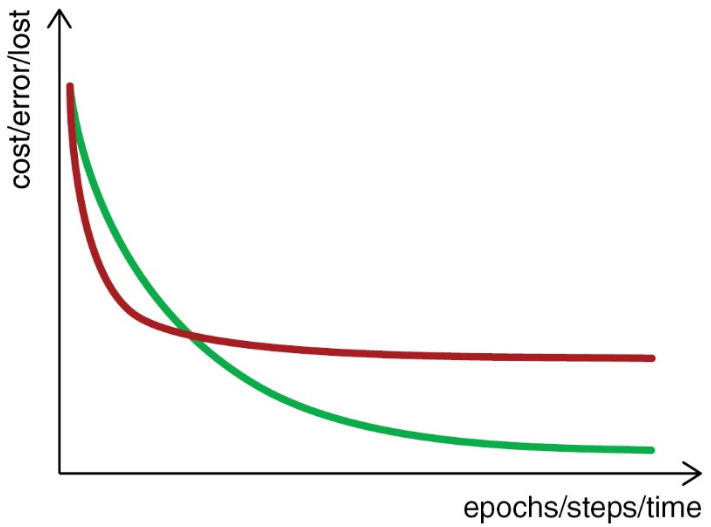


https://lossfunctions.tumblr.com/

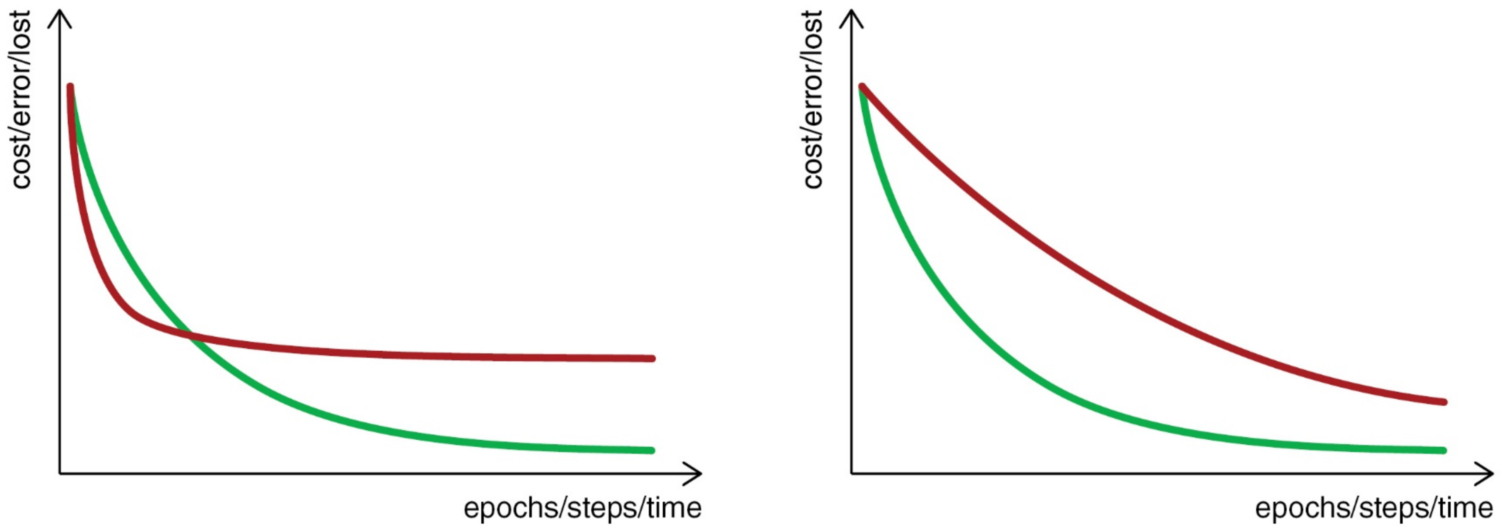
Debugging through Learning Curves



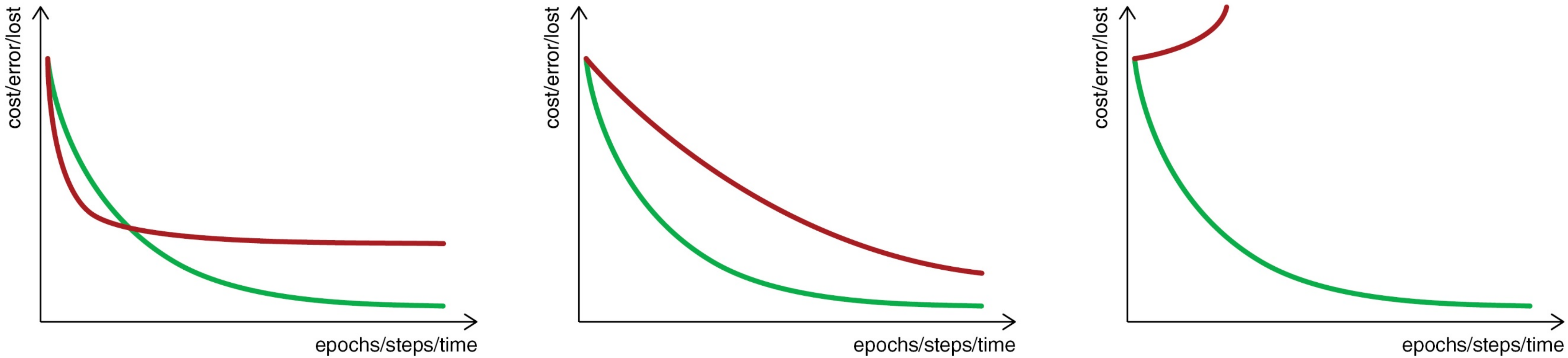
Debugging through Learning Curves



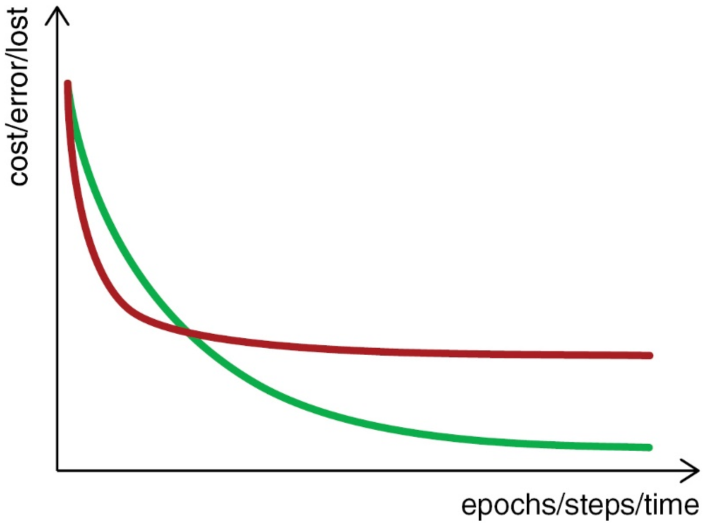
Debugging through Learning Curves

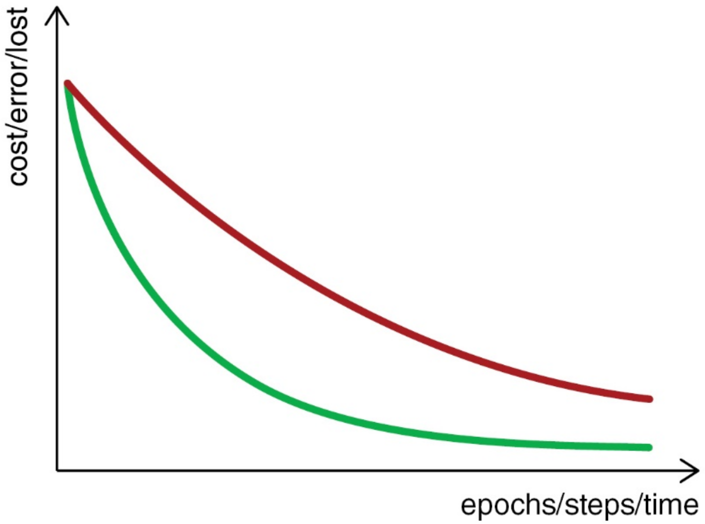


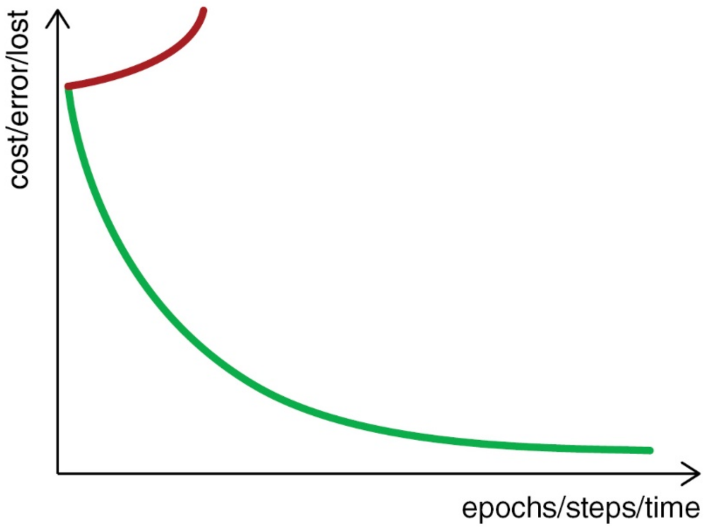
Debugging through Learning Curves

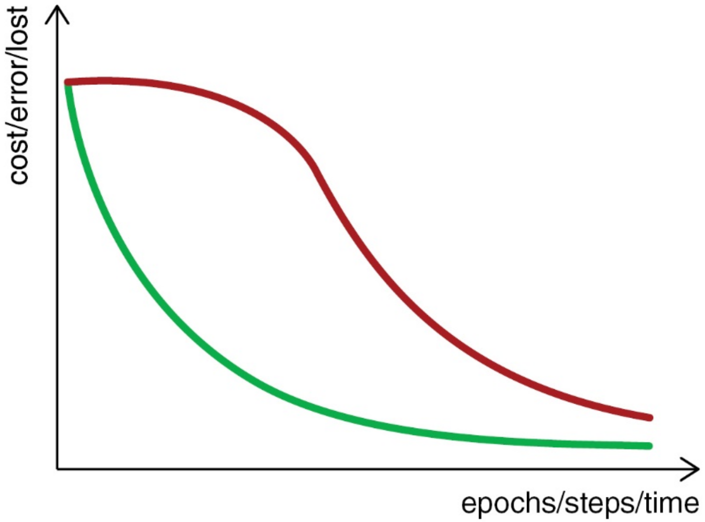


Debugging through Learning Curves

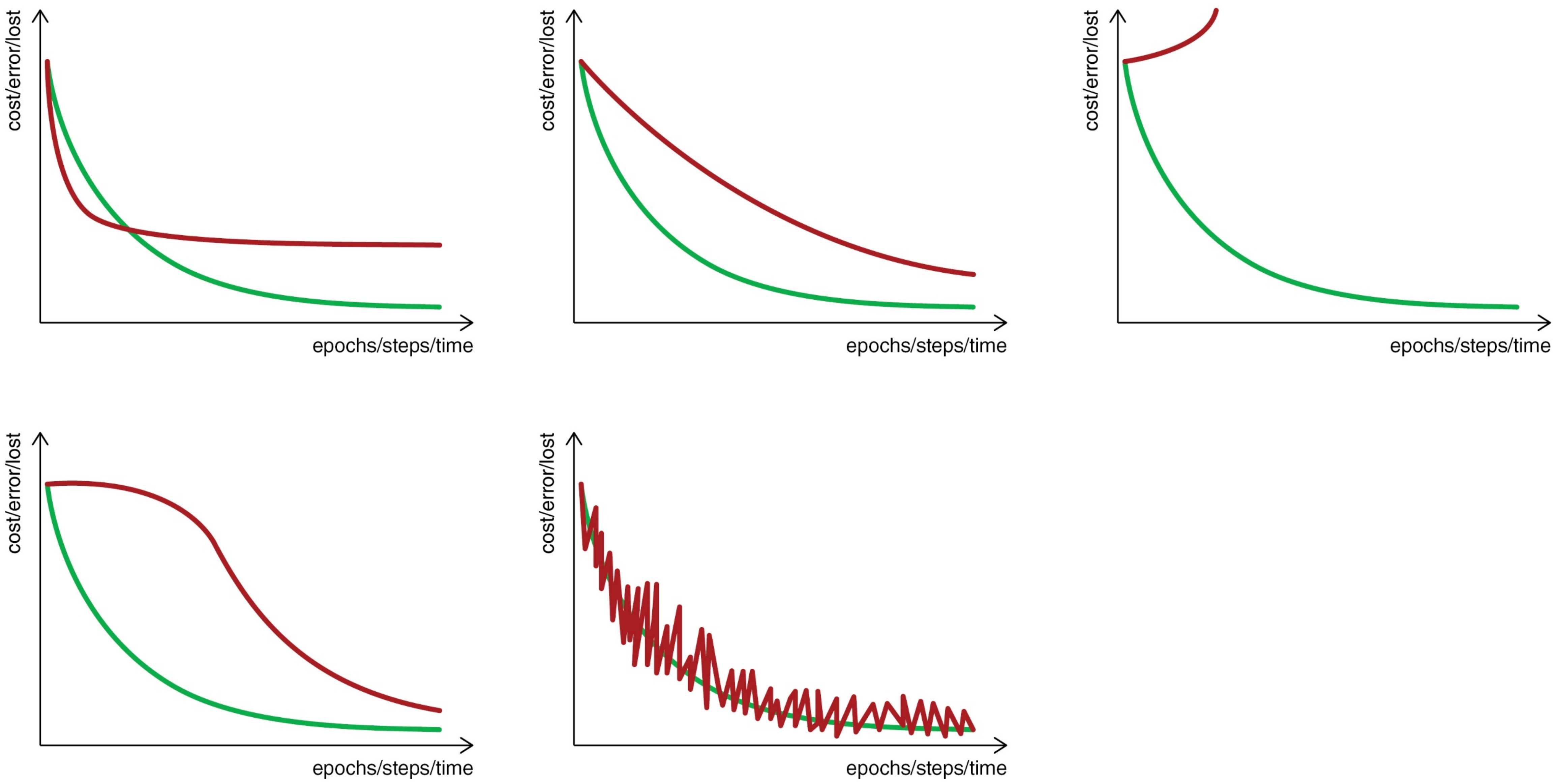




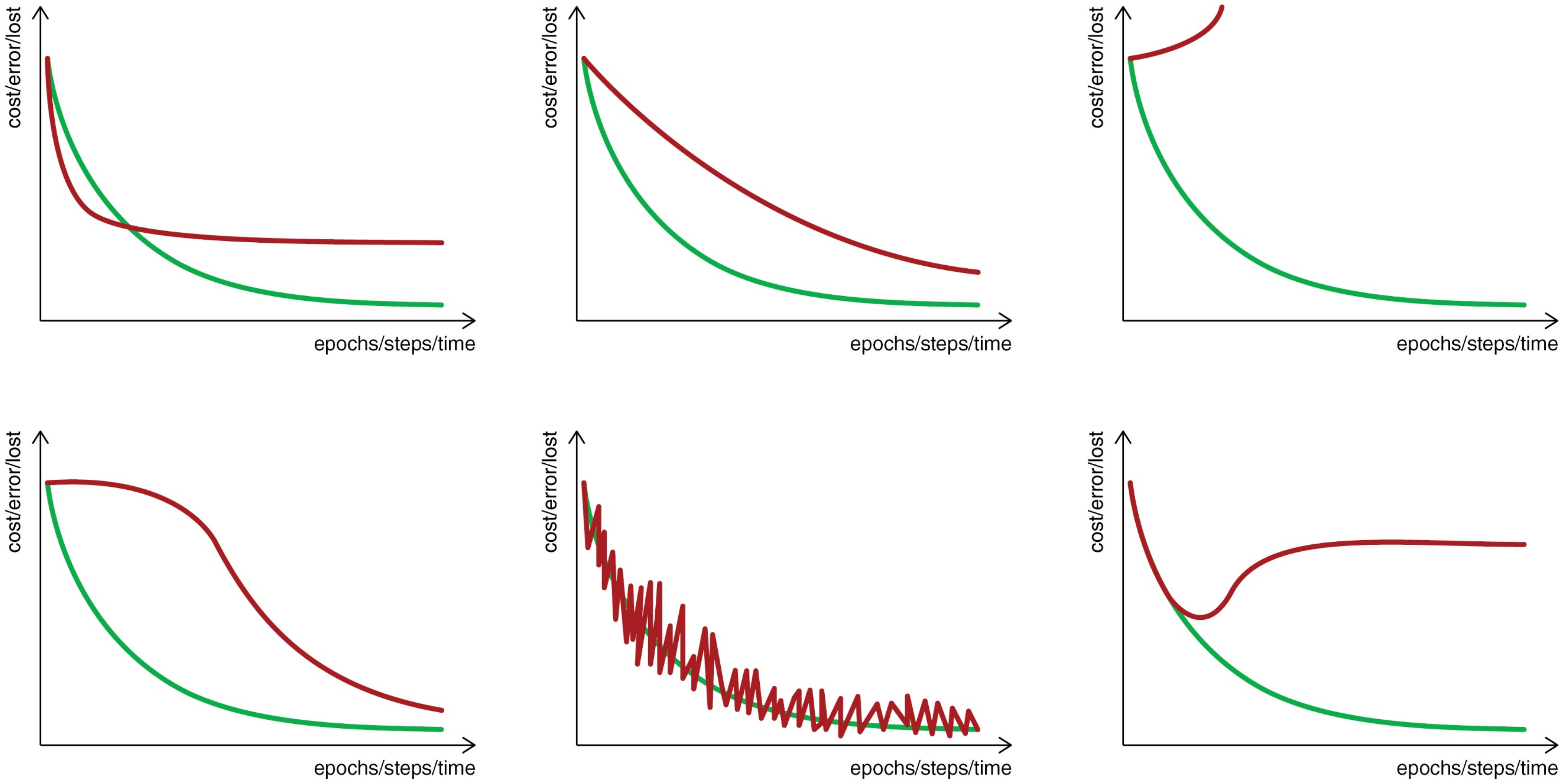




Debugging through Learning Curves



Debugging through Learning Curves



Thank you!