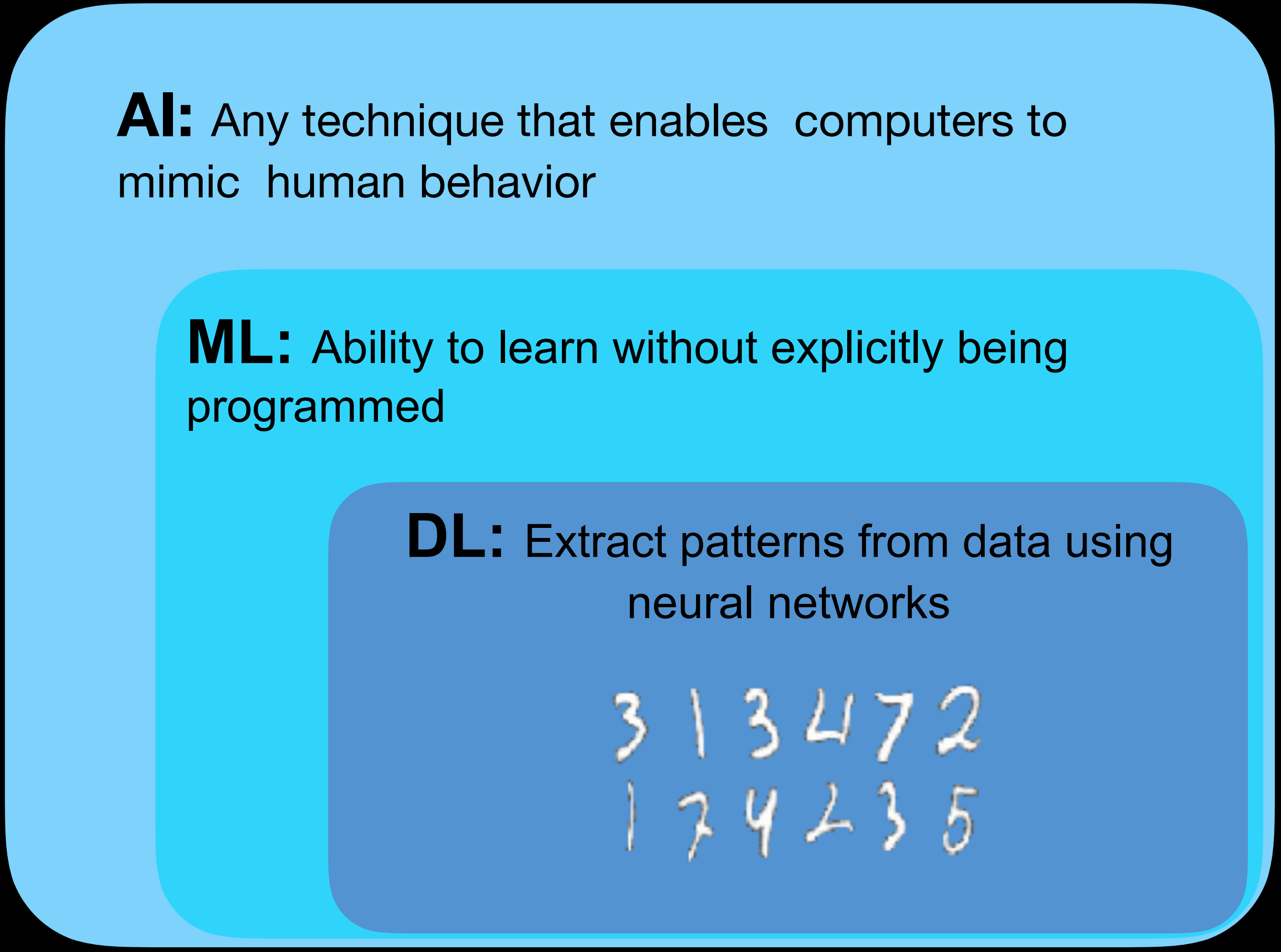
Introduction to Deep Learning

WM CS
Zeyi (Tim) Tao
11/01/2019

References

- Introduction to Deep Learning CMU 11-785 http://deeplearning.cs.cmu.edu/
- Introduction to Deep Learning MIT 6.S191 http://introtodeeplearning.com/
- MIT Deep Learning Collections https://deeplearning.mit.edu/
- Deep Learning Stanford CS230 (Andrew Ng) https://cs230.stanford.edu/
- Convolutional Neural Networks Stanford CS231n http://cs231n.stanford.edu/
- Others (Books, Papers, Talks, Videos, etc

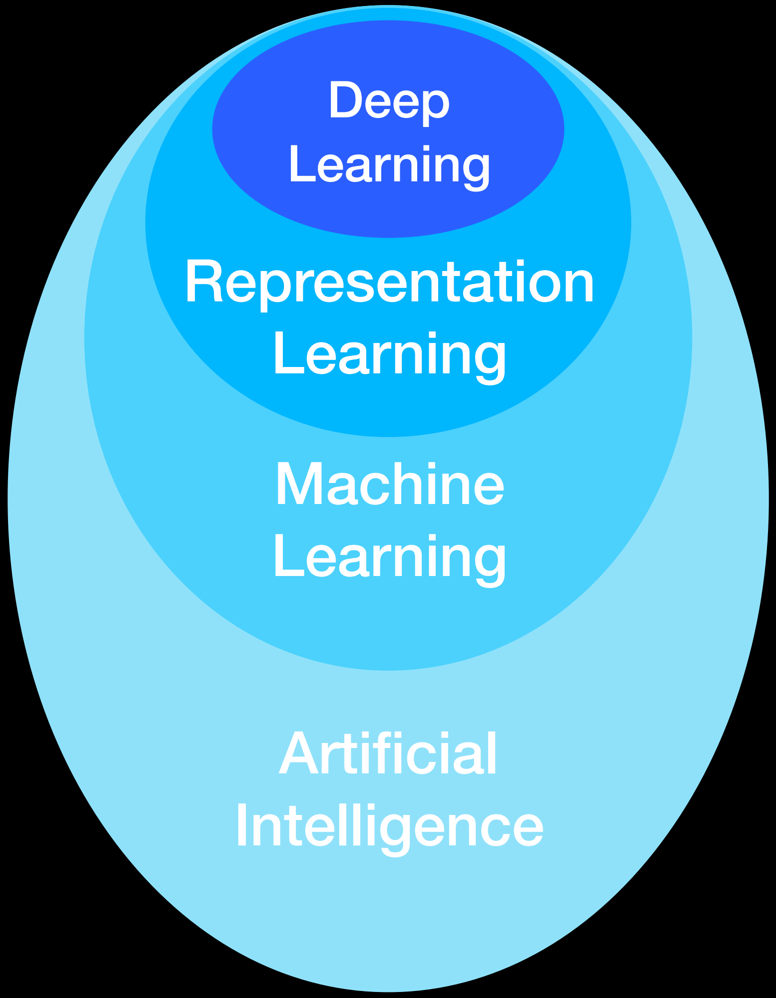
What is Deep Learning

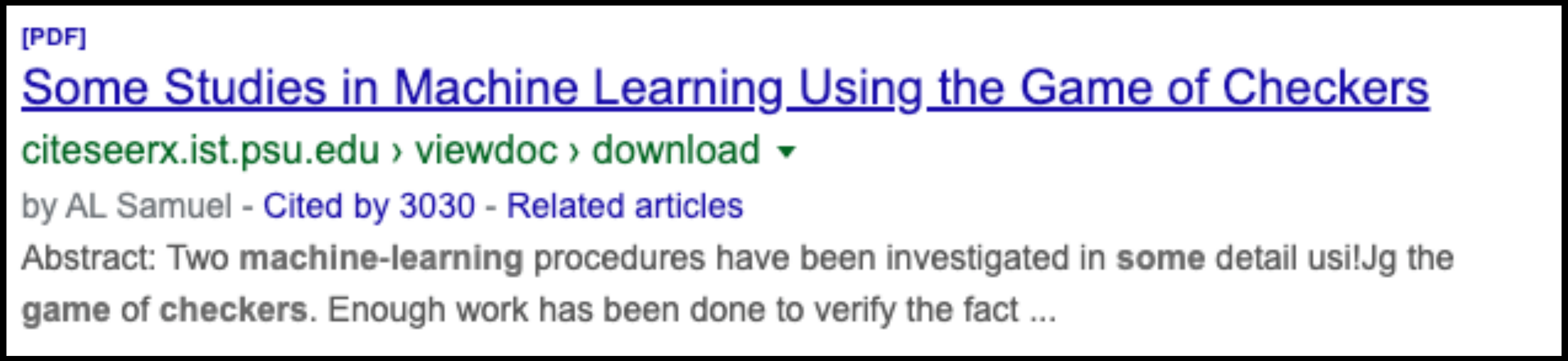


AI: Any technique that enables computers to mimic human behavior

ML: Ability to learn without explicitly being programmed

DL: Extract patterns from data using neural networks





[PDF]

Some Studies in Machine Learning Using the Game of Checkers

citeseerx.ist.psu.edu › viewdoc › download ▾

by AL Samuel - Cited by 3030 - Related articles

Abstract: Two **machine-learning** procedures have been investigated in **some** detail usi!Jg the **game of checkers**. Enough work has been done to verify the fact ...

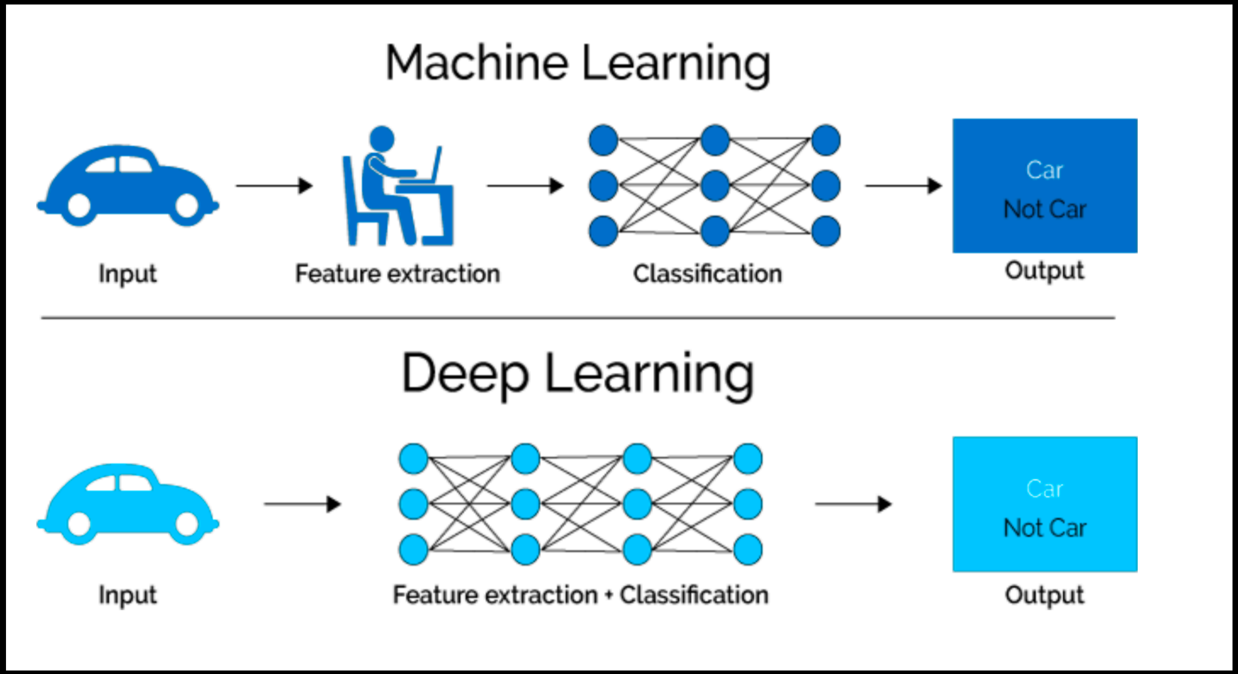
Deep Learning: state-of-the-art

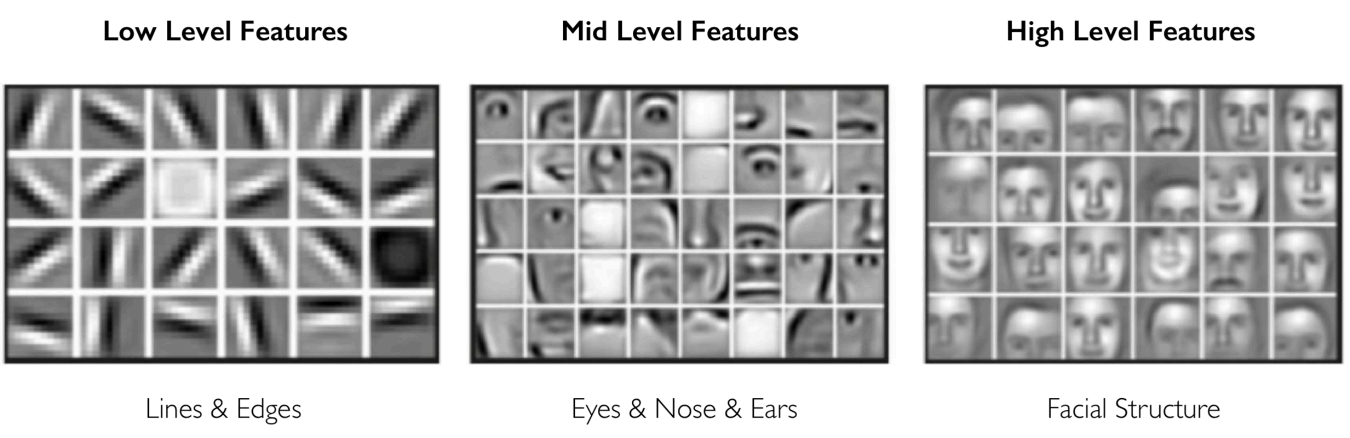
Exciting Progress:

- Face recognition
- Image classification
- Speech recognition
- Text-to-speech generation
- Handwriting transcription
- Machine translation
- Medical diagnosis
- Cars: drivable area, lane keeping
- Digital assistants
- Ads, search, social recommendations
- Game playing with deep RL

Art generation (Neural Style Transfer)

Traditional Machine Learning





Hand engineered features are time consuming, brittle and not scalable in practice.

Can we learn the **underlying features** directly from data?

History of Deep Learning Ideas

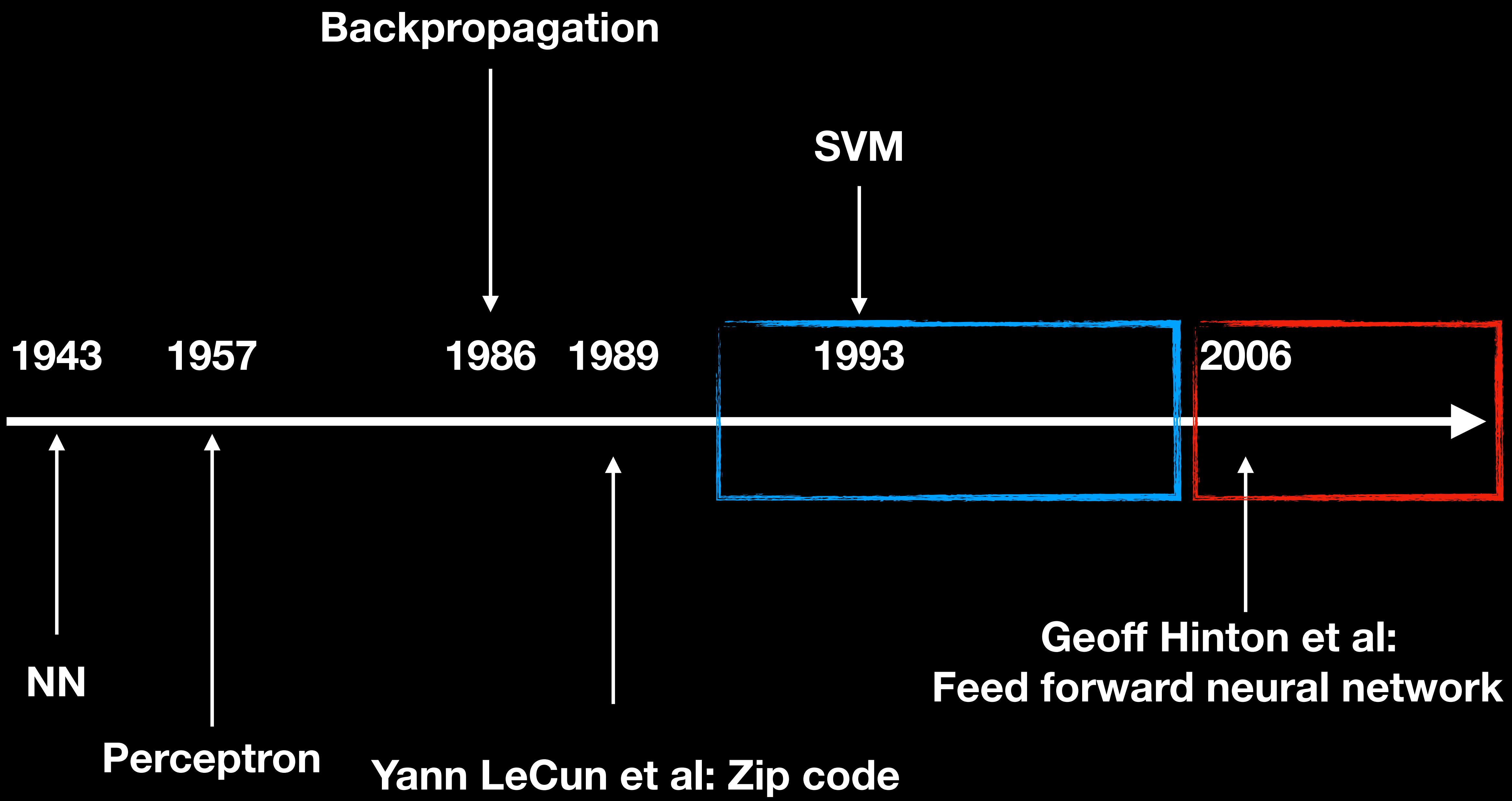
History of Deep Learning Ideas and Milestones

- 1943: Neural Networks
- 1957: Perceptron
- 1974-86: Backpropagation, RNN
- 1989-98: CNN, MNIST, LSTM, Bidirectional RNN
- 2006: “Deep Learning”, DBN, by Geoff Hinton et al
- 2009: ImageNet
- 2012: AlexNet, Dropout
- 2014: GANs
- 2014: DeepFace
- 2016: AlphaGo
- 2017: AlphaZero, Capsule Networks
- 2018: BERT

History of DL Tools

- Mark 1 Perceptron – 1960
- Torch – 2002
- CUDA – 2007
- Theano – 2008
- Caffe – 2014
- DistBelief – 2011
- TensorFlow 0.1 – 2015
- PyTorch 0.1 – 2017
- TensorFlow 1.0 – 2017
- PyTorch 1.0 – 2017
- TensorFlow 2.0 – 2019

History of Deep Learning Ideas



Backpropagation

SVM

1943 1957 1986 1989 1993 2006

NN

Perceptron

Yann LeCun et al: Zip code

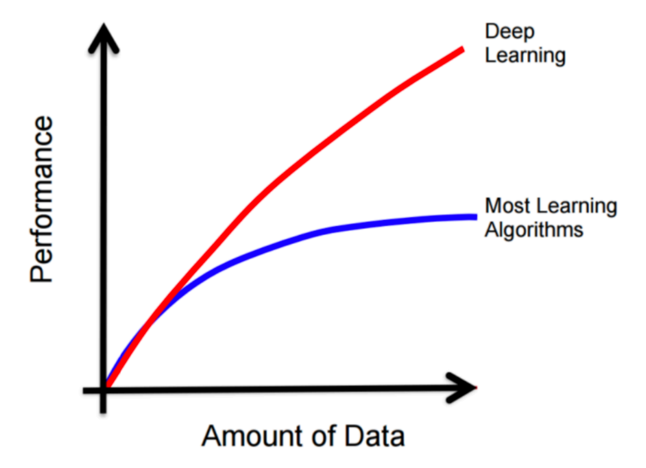
Geoff Hinton et al:
Feed forward neural network

Deep Learning Today

Big Data

Larger Datasets

Easier Collection & Storage

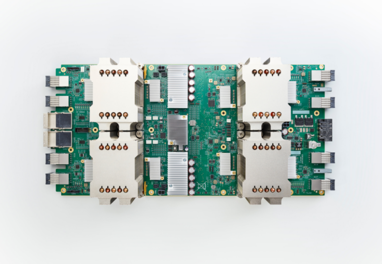


Scale drives deep learning progress

Hardware

Graphics Processing Units (GPUs)

Massively Parallelizable



Software

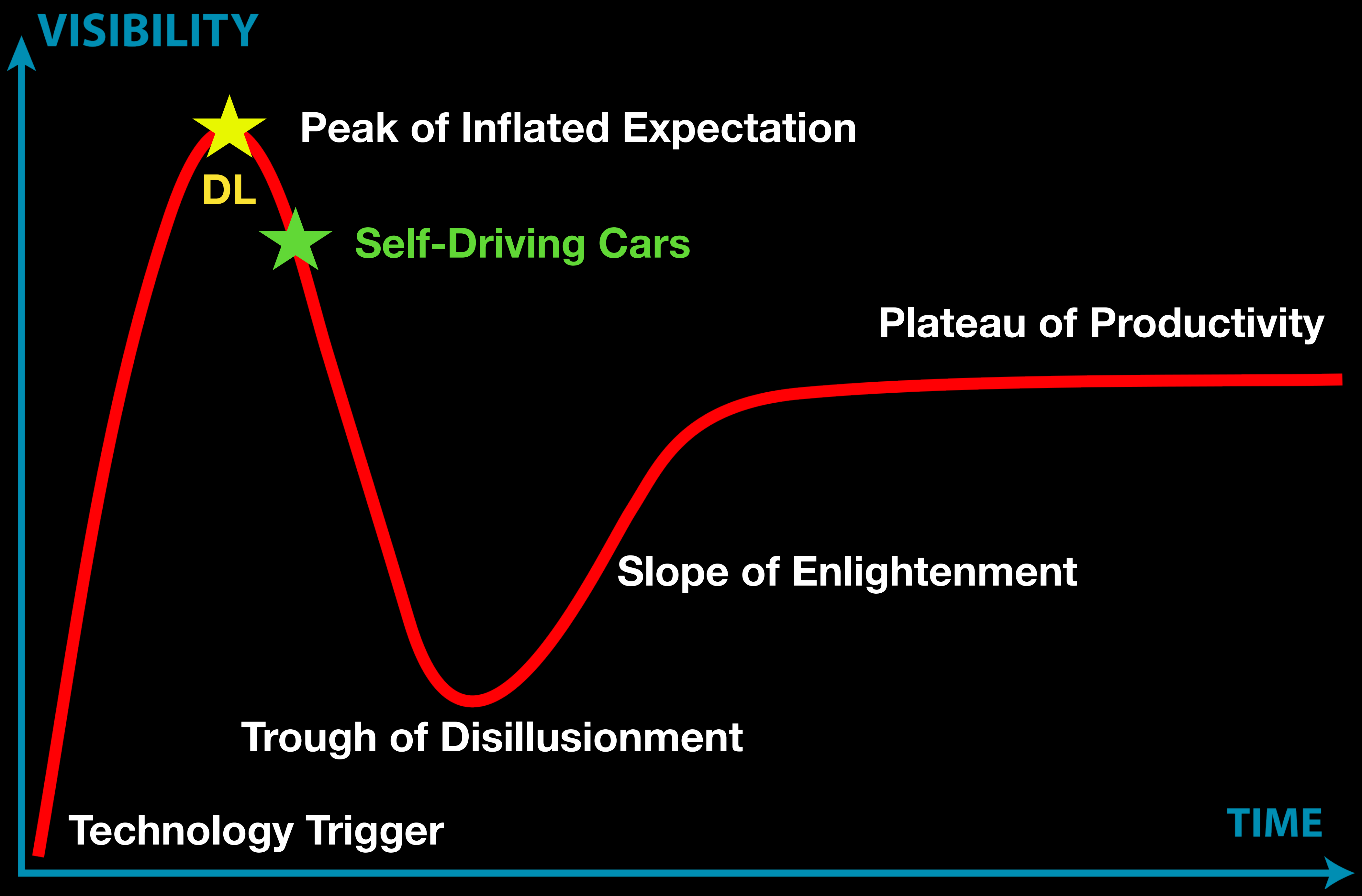
Improved Techniques

New Models Toolboxes

TensorFlow

PyTorch

5 Stages in Gartner Hype Cycle

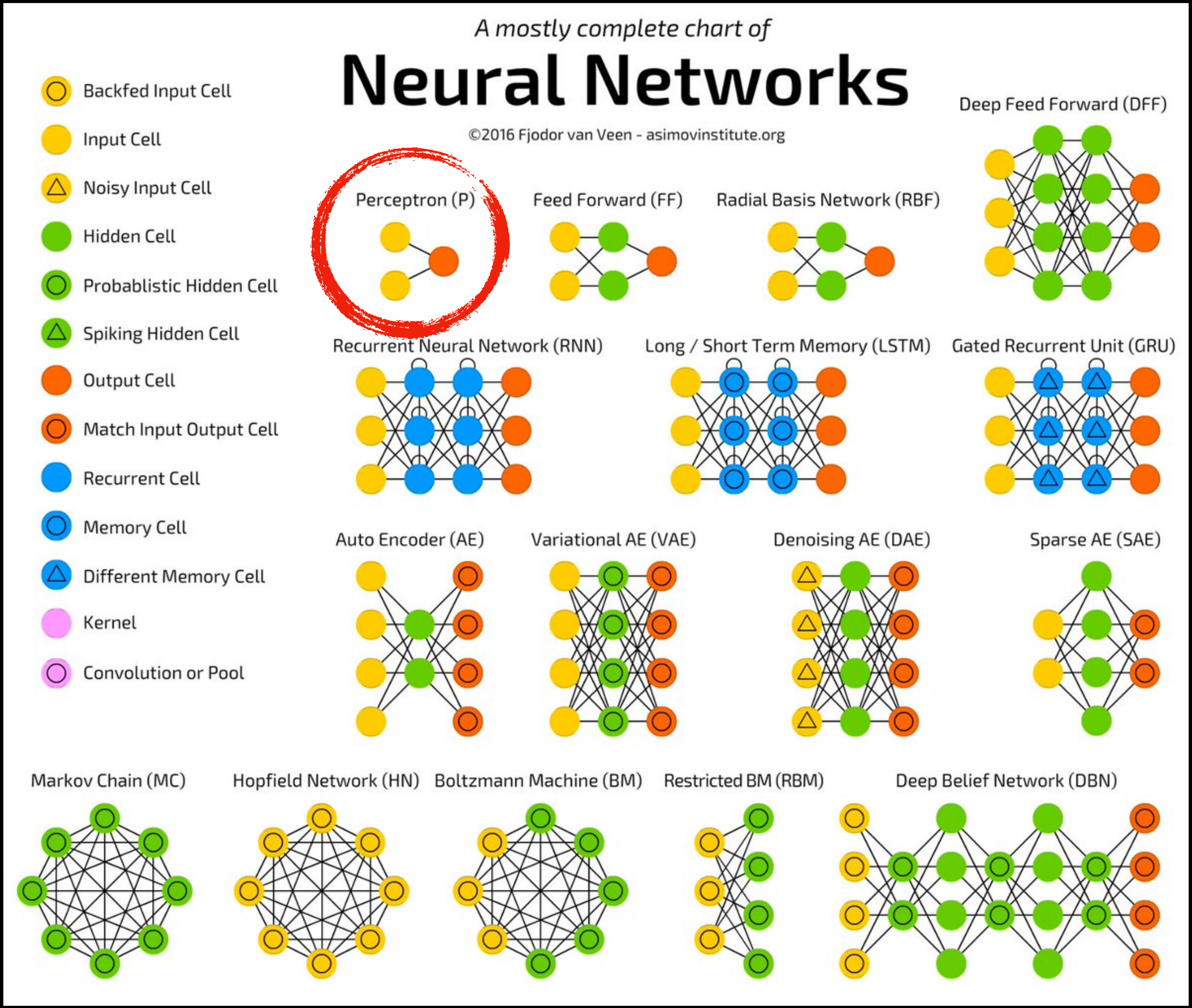


Again, What is Deep Learning?

Input **Model = Architecture + Parameters** **Output**

$x \sim \mathscr{D} \subset \mathscr{R}^m$ $f(\cdot) : \mathscr{R}^m \rightarrow \mathscr{R}^c$ $\hat{y} = f(x)$

Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher level features from the raw input.(wiki)

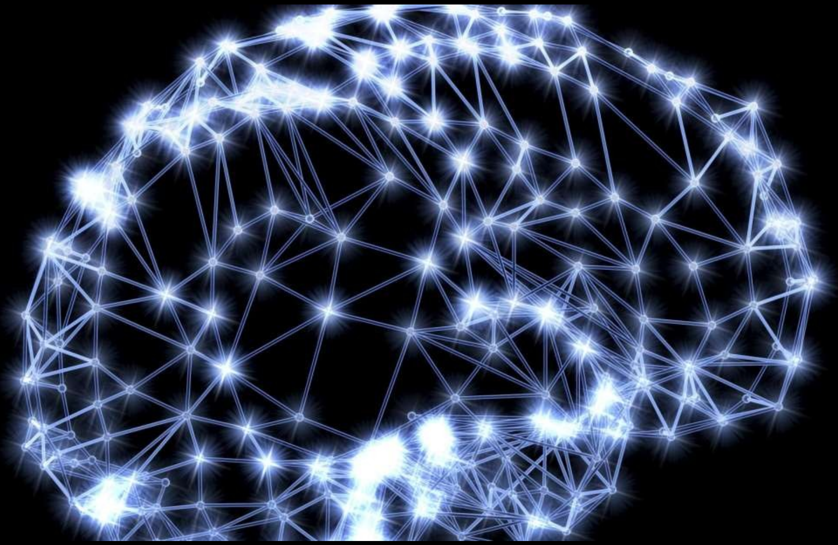


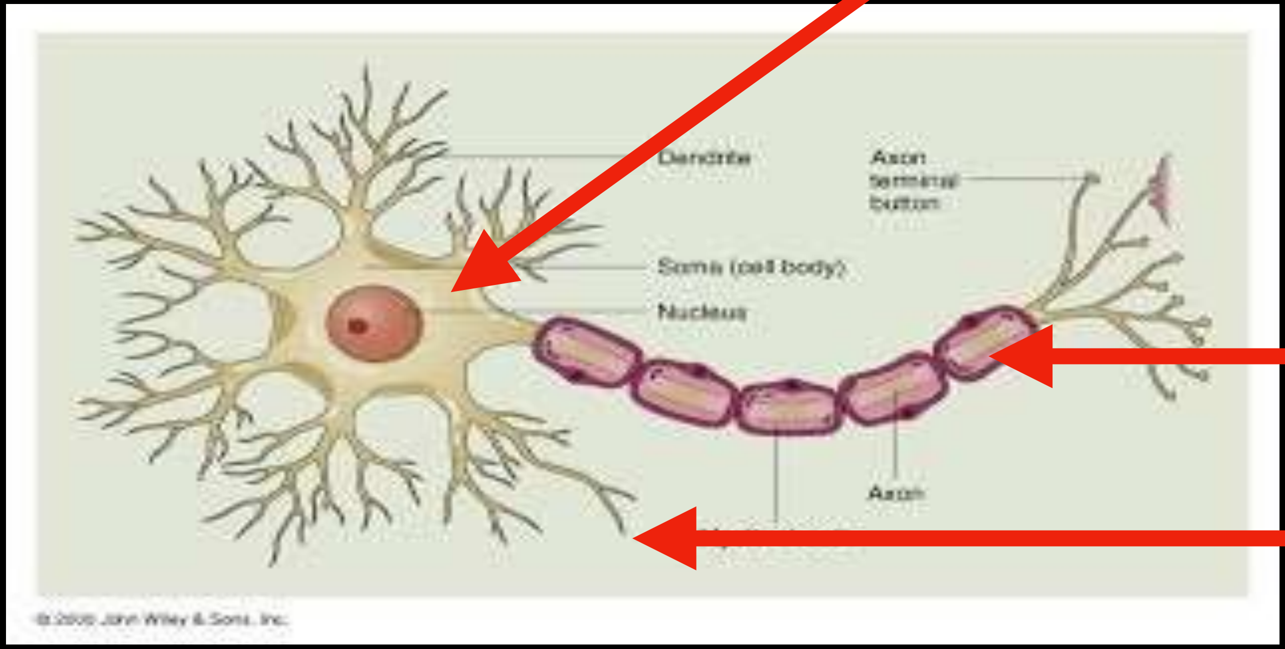
A mostly complete chart of

Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

Observations: The Brain and Neurons





© 2000 John Wiley & Sons, Inc.

Soma

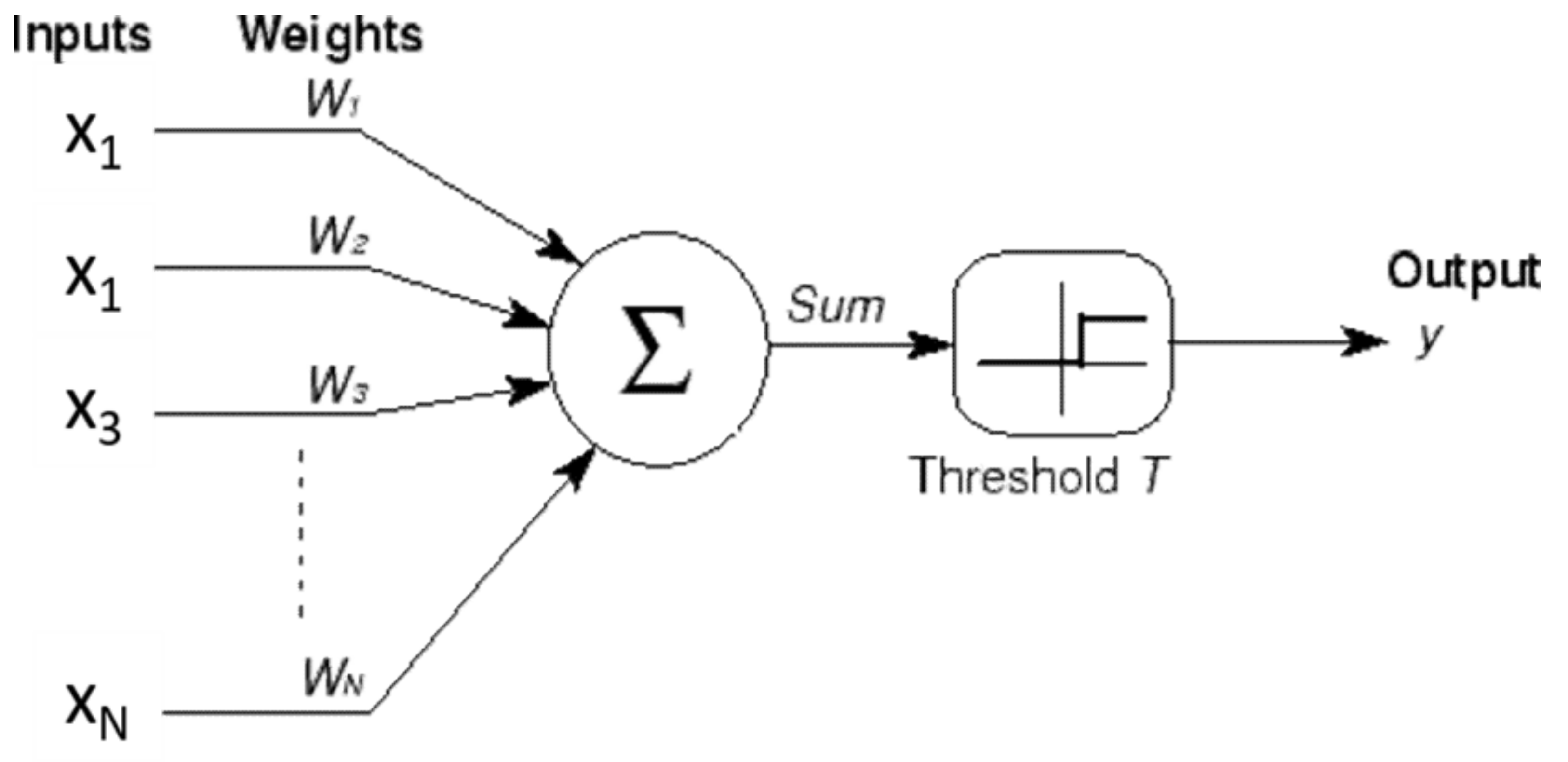
Axon

Dendrites

The Perceptron

- Frank Rosenblatt
  - Psychologist, Logician
  - Inventor of the solution to everything, aka the Perceptron (1957)

Rosenblatt’s Perceptron (math model)

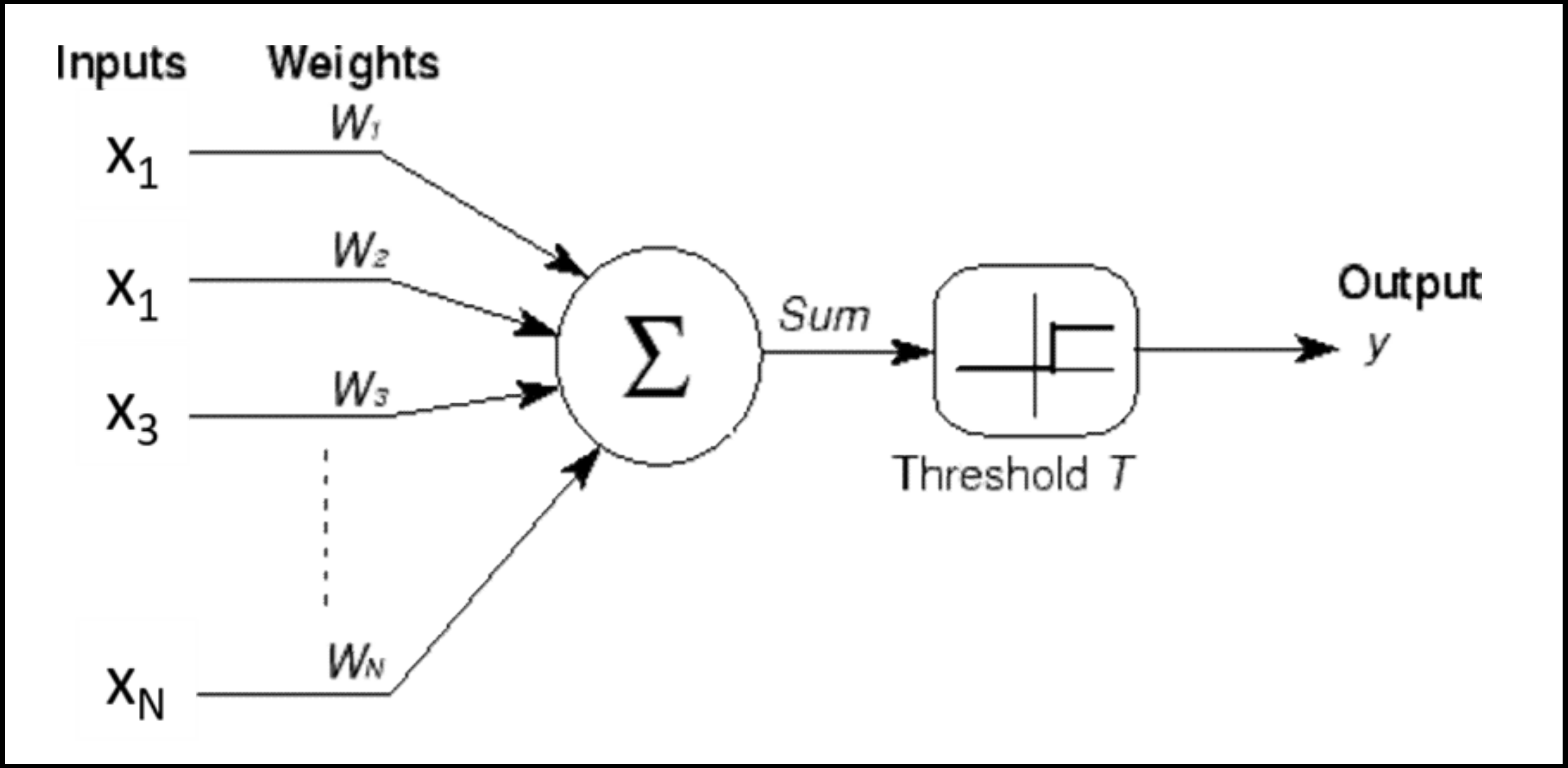


$$Y = \begin{cases} 1 & if \quad \sum_i w_i x_i - T > 0 \\ 0 & else \end{cases}$$

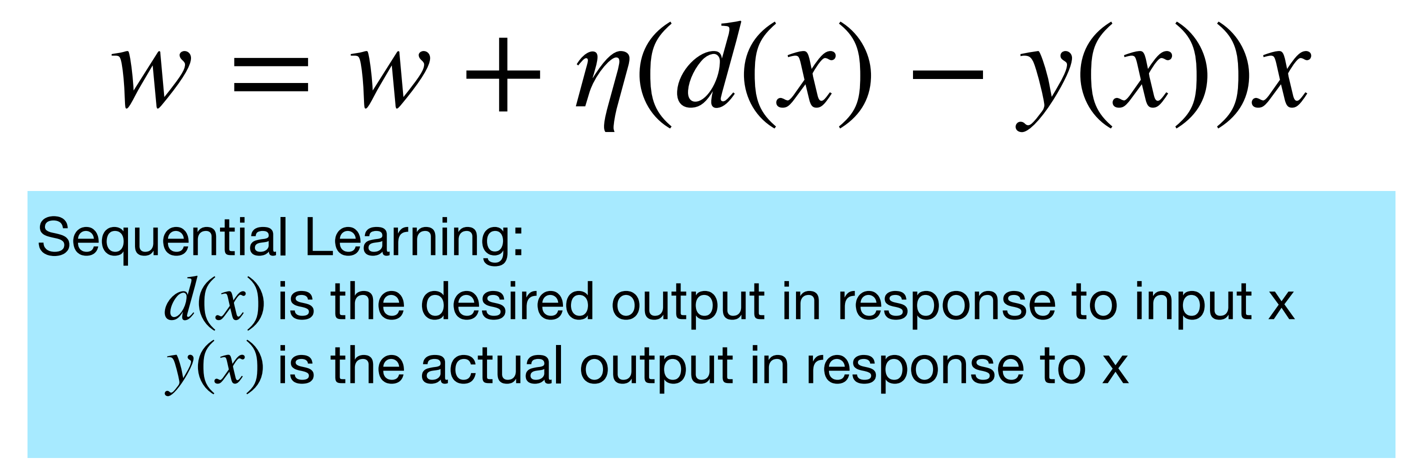
- Number of inputs combine linearly
  - Threshold logic: Fire if combined input exceeds threshold

Rosenblatt’s Perceptron

- Originally assumed could represent any Boolean circuit and perform any logic
  - “the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence,” New York Times (8 July) 1958
  - “Frankenstein Monster Designed by Navy That Thinks,” Tulsa, Oklahoma Times 1958



Rosenblatt’s Learning Algorithm



$$w = w + \eta(d(x) - y(x))x$$

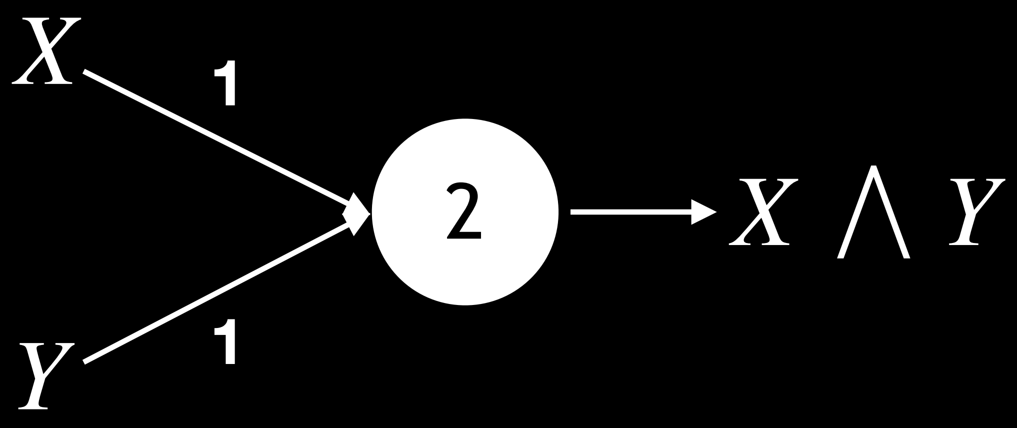
Sequential Learning:

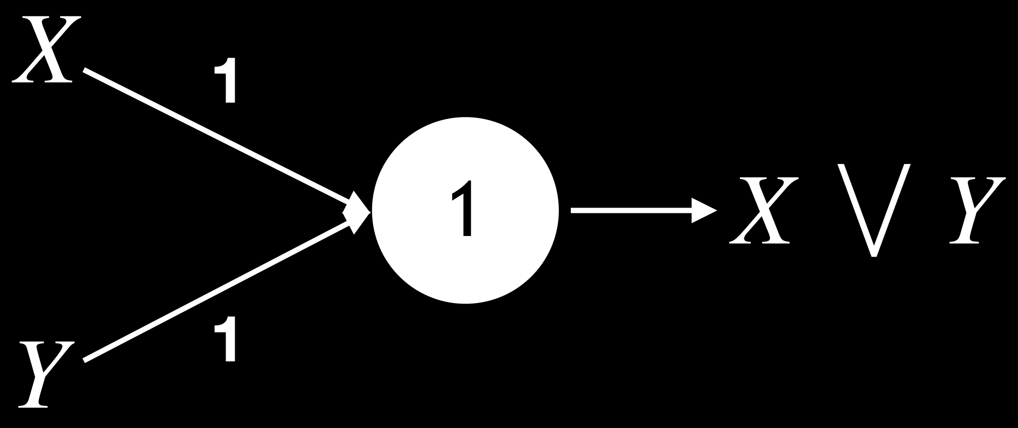
$d(x)$ is the desired output in response to input x

$y(x)$ is the actual output in response to x

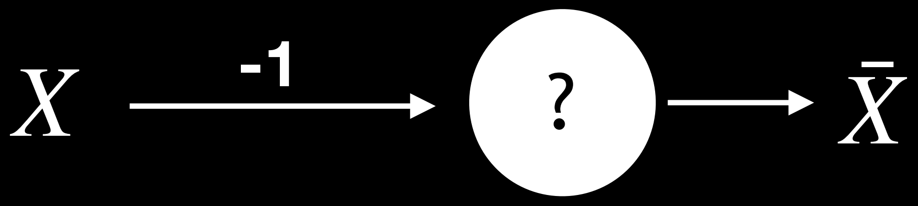
- Boolean tasks
- Update the weights whenever the perceptron output is wrong
- Proved convergence for linearly separable classes

The Perceptron (gate)

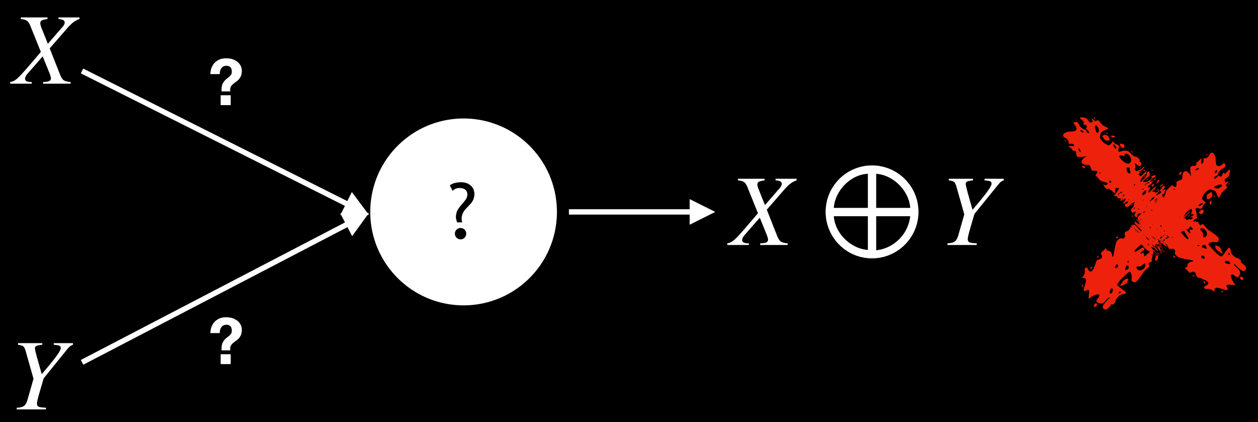




Values shown on edges are weights, numbers in the circles are thresholds

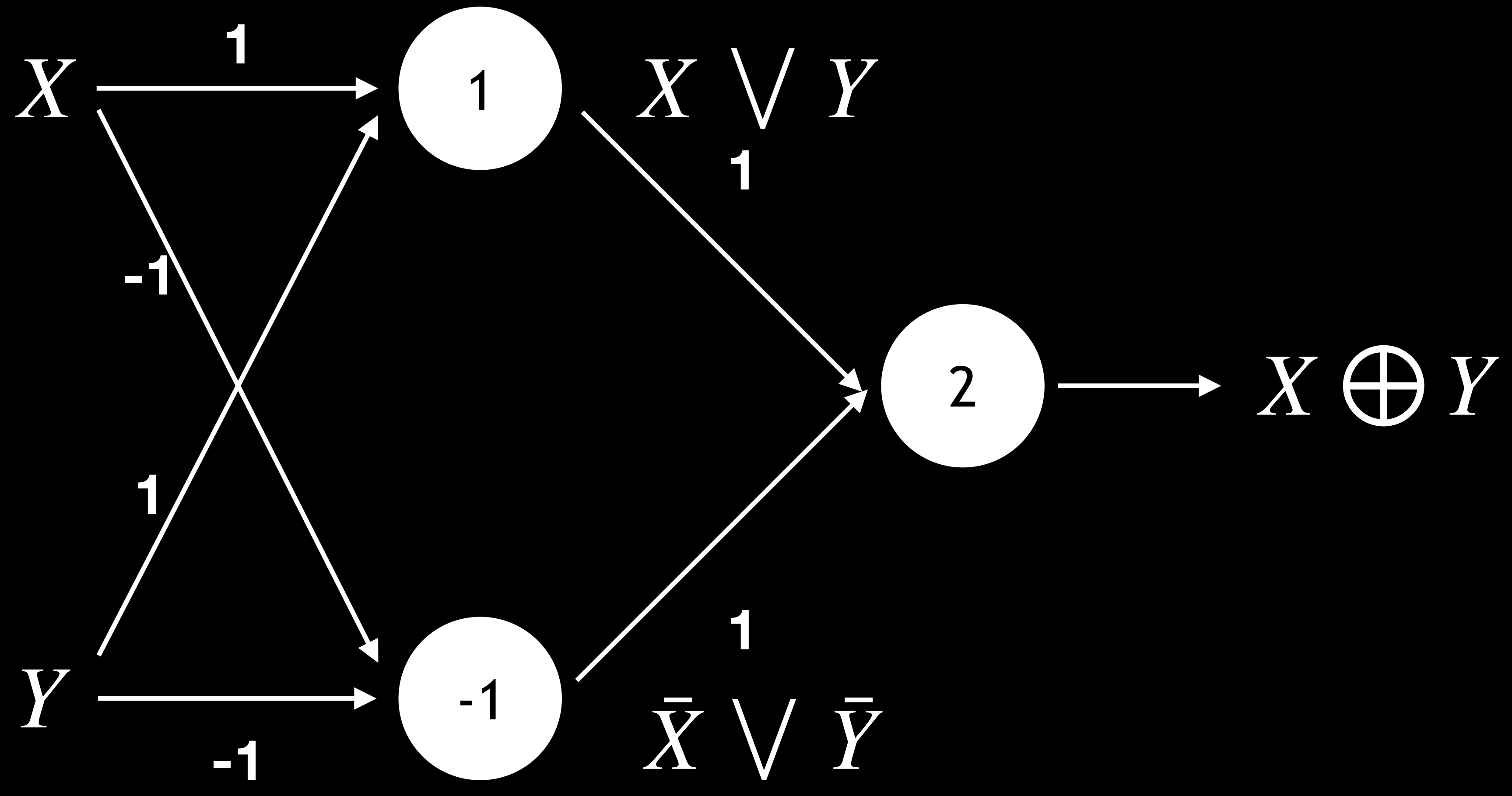


However, the single perceptron …



No solution for XOR!
Not universal!

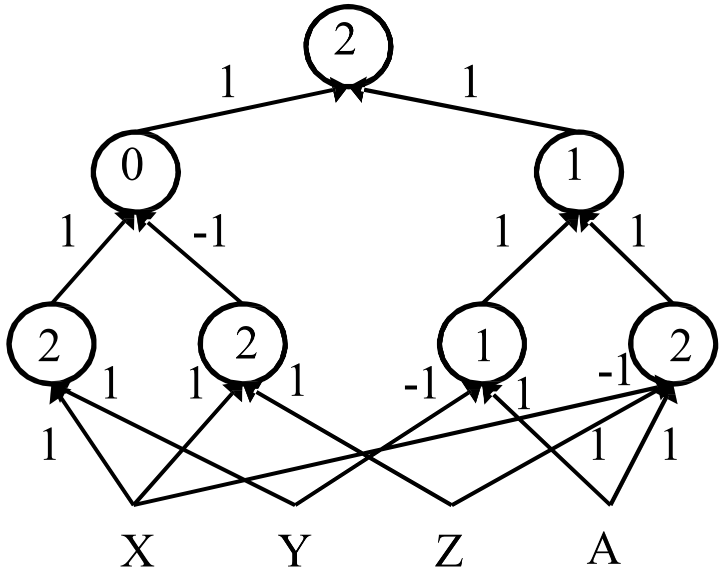
The Multi-layer Perceptron (gate)



- XOR
  - The first layer is a “hidden” layer
  - Also originally suggested by Minsky and Papert 1968

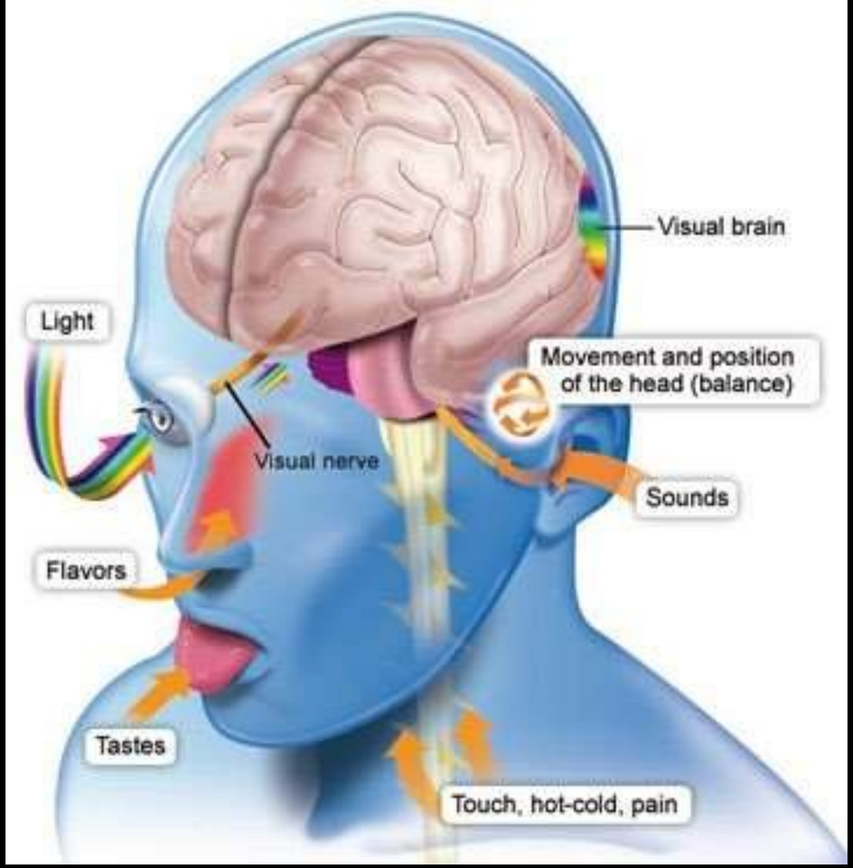
The Multi-layer Perceptron (gate)

$$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$$



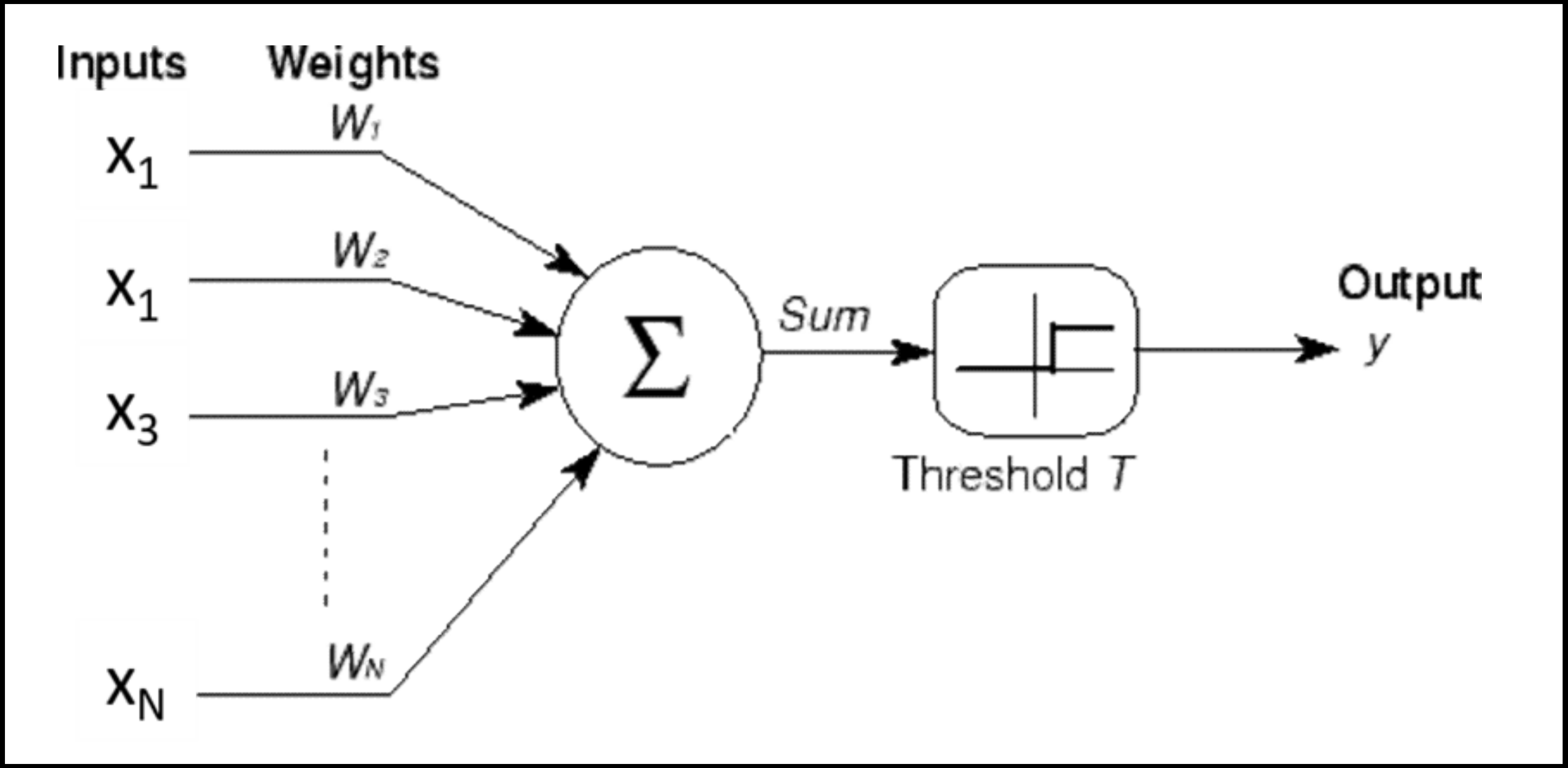
- A“multi-layer” perceptron
- Can compose arbitrarily complicated Boolean functions!
  - – In cognitive terms: Can compute arbitrary Boolean functions over sensory input
  - – More on this in the next class

But our brain is not Boolean



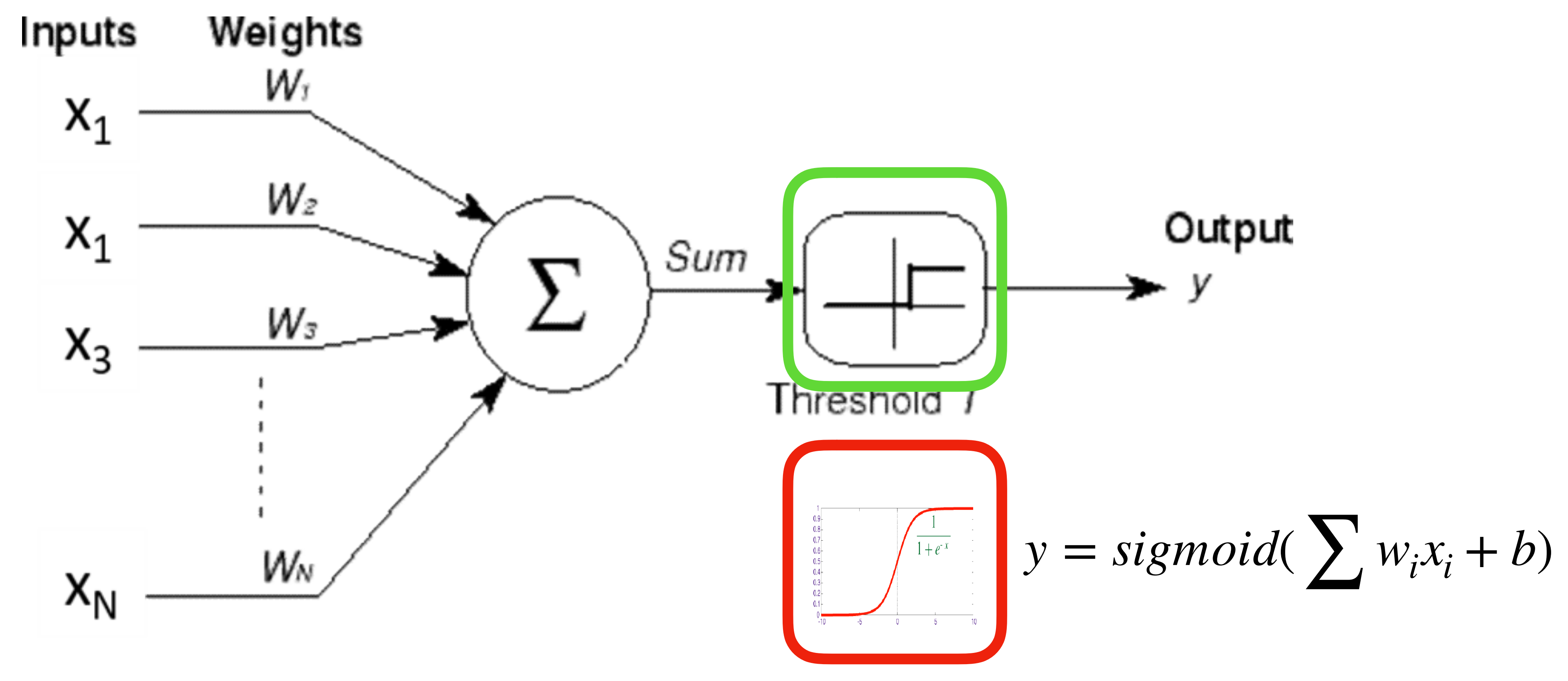
- We have real inputs
- We make non-Boolean inferences/predictions

The Perceptron (real inputs)



- $x_1 \ldots x_N$ are real valued
- $w_1 \ldots w_N$ are real valued
- Unit “fires” if weighted input exceeds a threshold

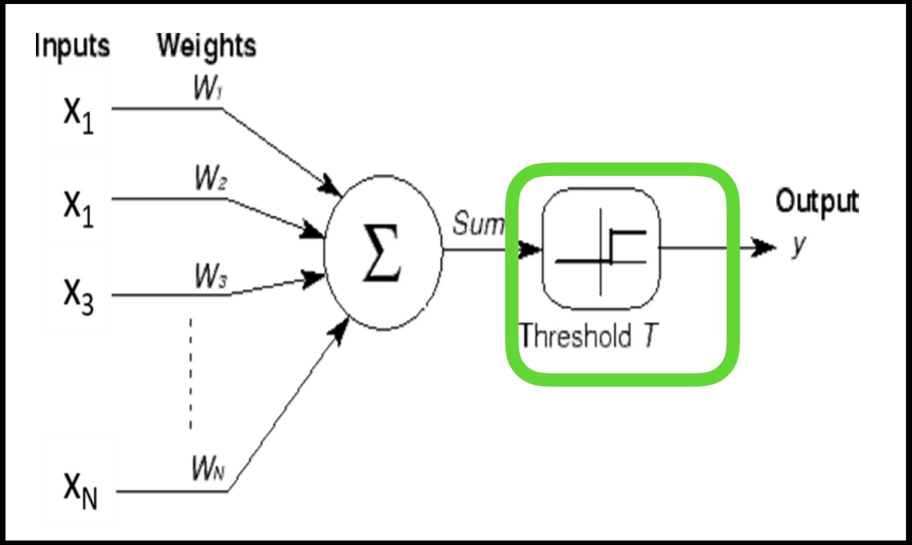
The Perceptron (real inputs)



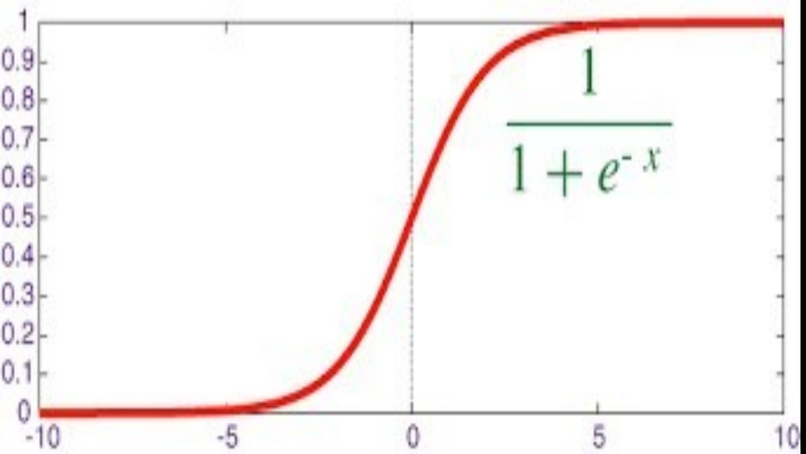
$$y = sigmoid(\sum w_i x_i + b)$$

- $x_1 \ldots x_N$ are real valued
- $w_1 \ldots w_N$ are real valued
- Unit “fires” if weighted input exceeds a threshold
- The output y can be real valued
  - Sometimes viewed as the “probability” of firing

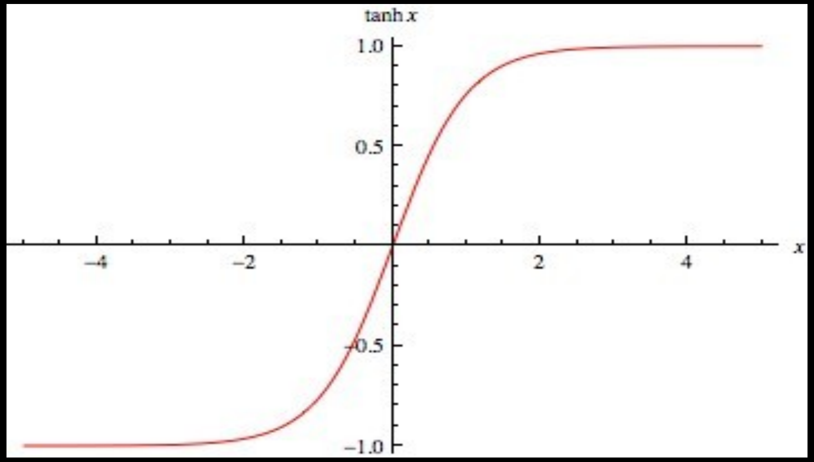
Other activations



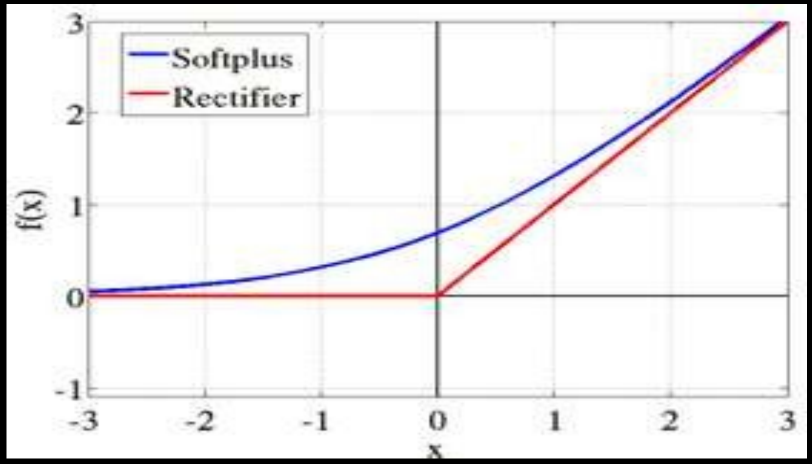
sigmoid



Tanh

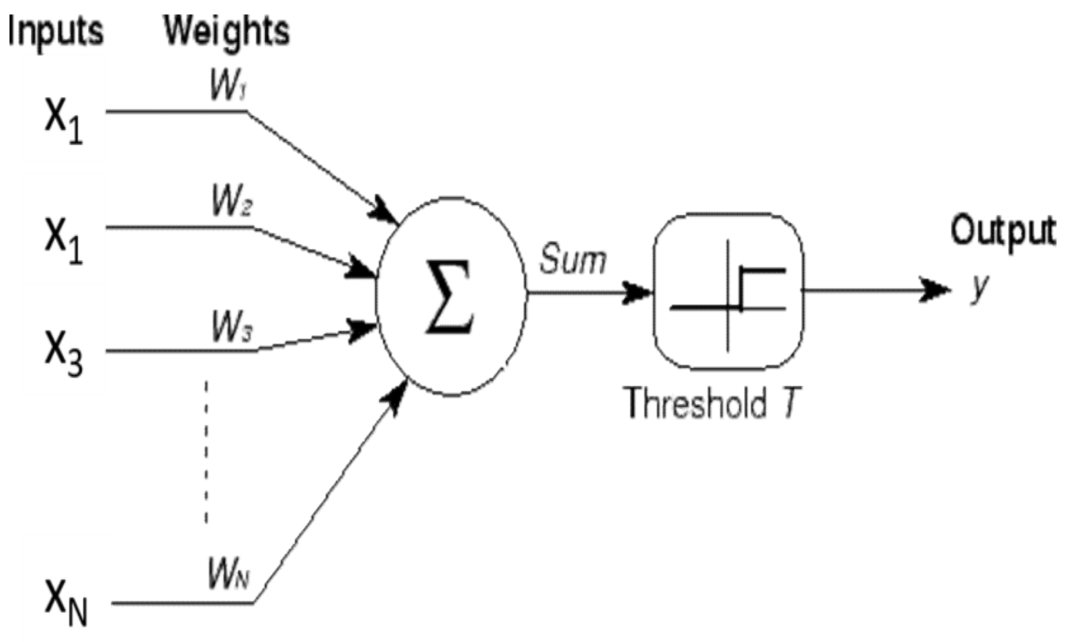


ReLU

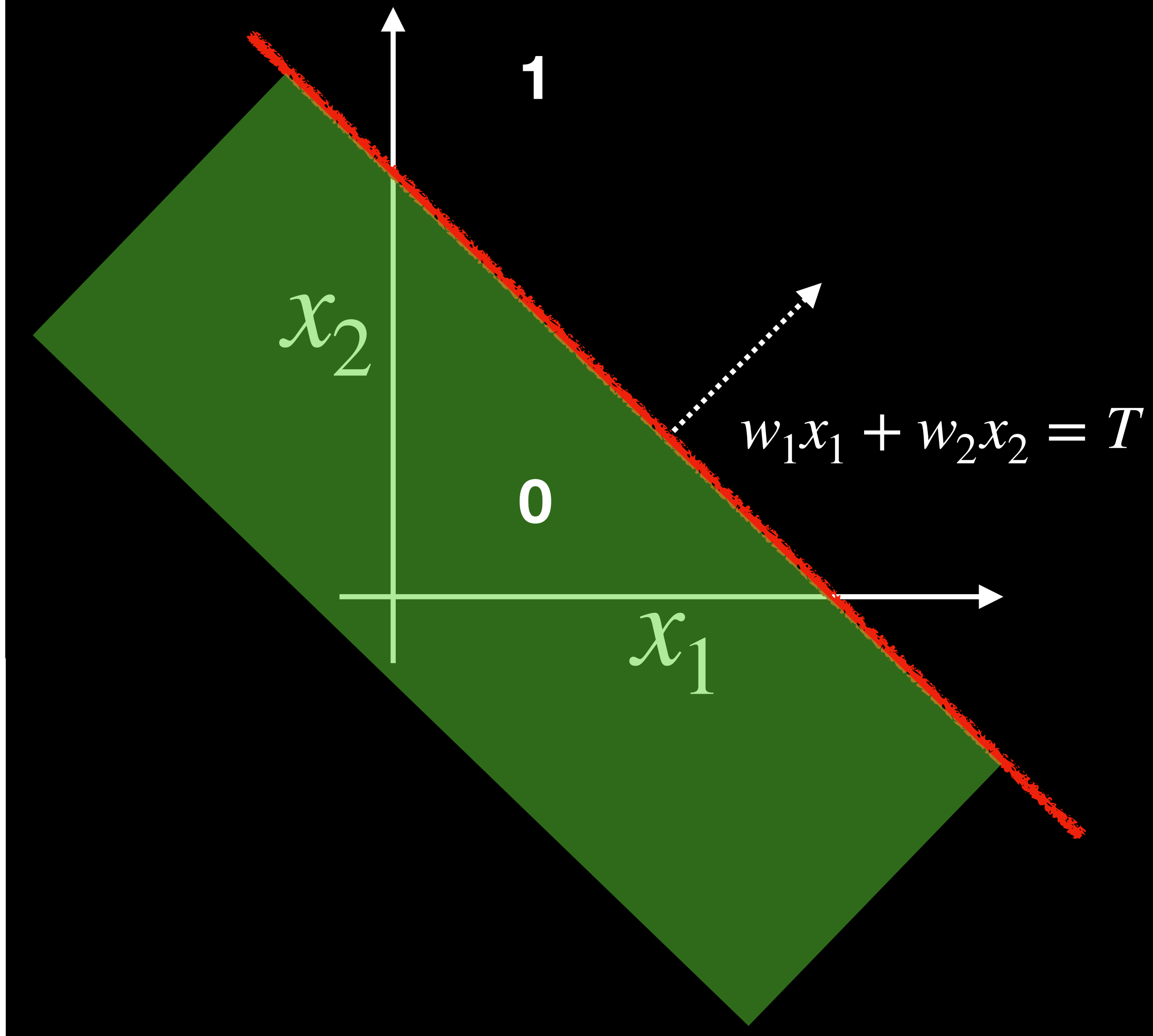


- Does not always have to be a squashing function
  - We will hear more about activations later
- We will continue to assume a “threshold” activation right now

The Perceptron (real inputs)



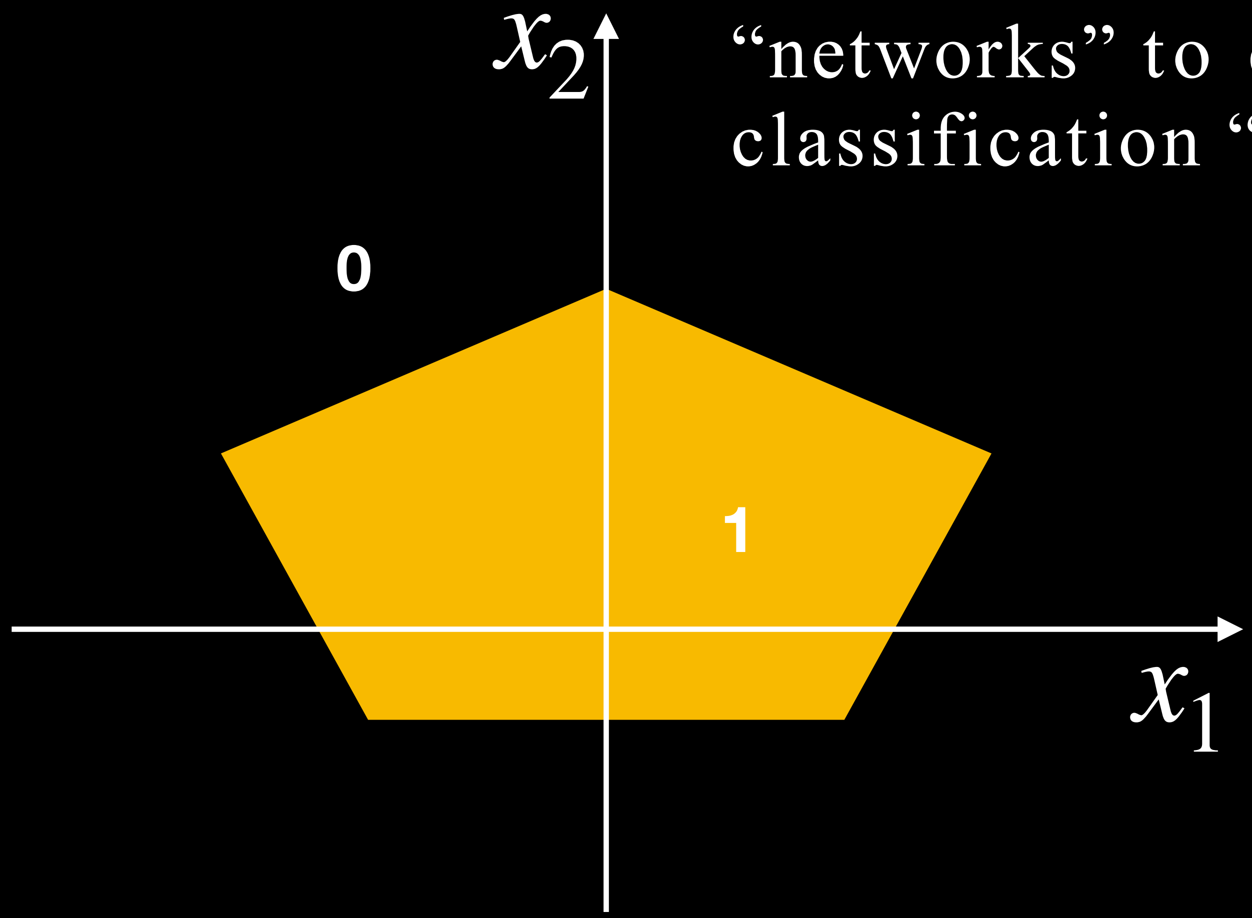
$$Y = \begin{cases} 1 & if \quad \sum_i w_i x_i - T > 0 \\ 0 & else \end{cases}$$



– This is a linear classifier

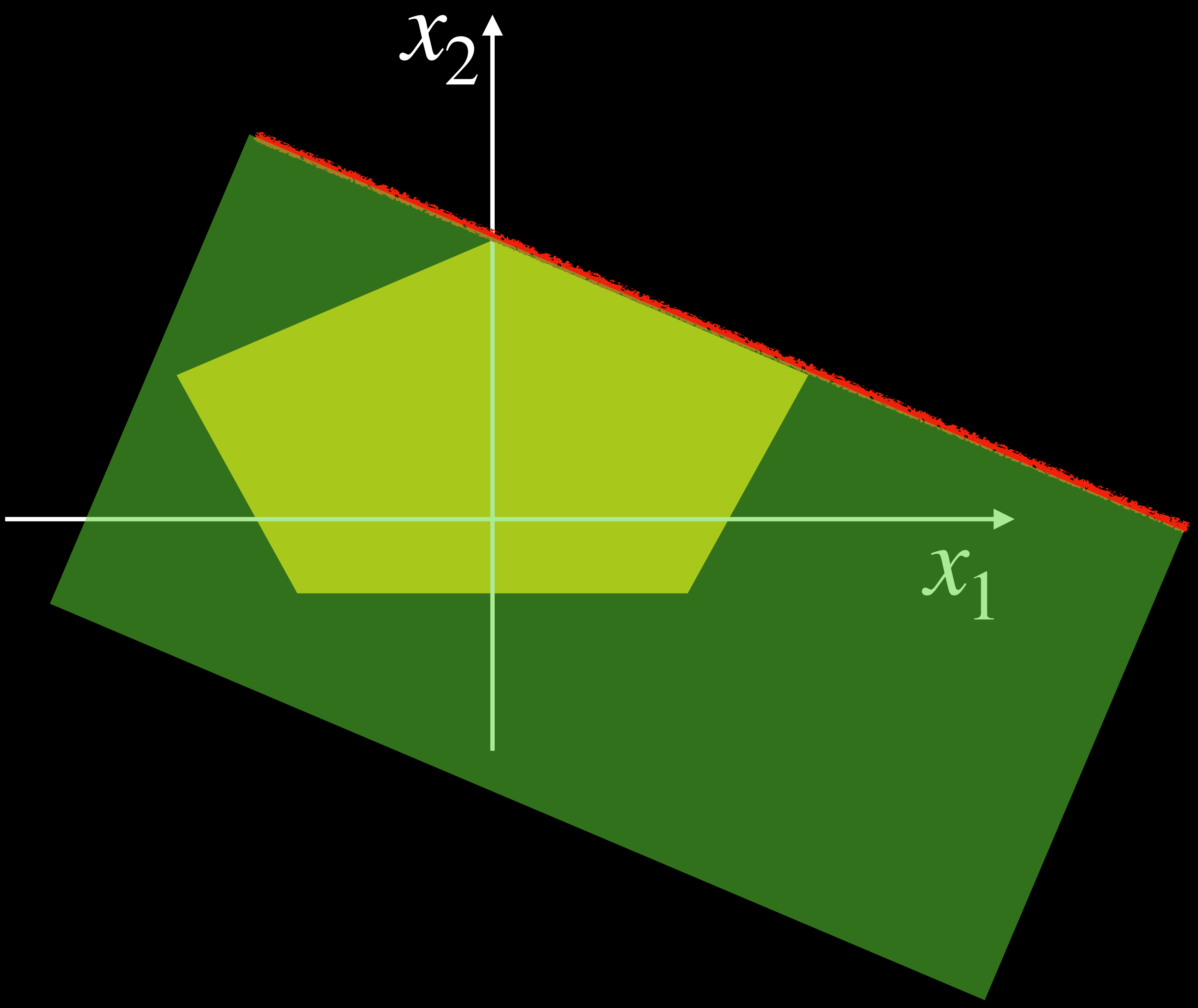
"Decision" Boundaries

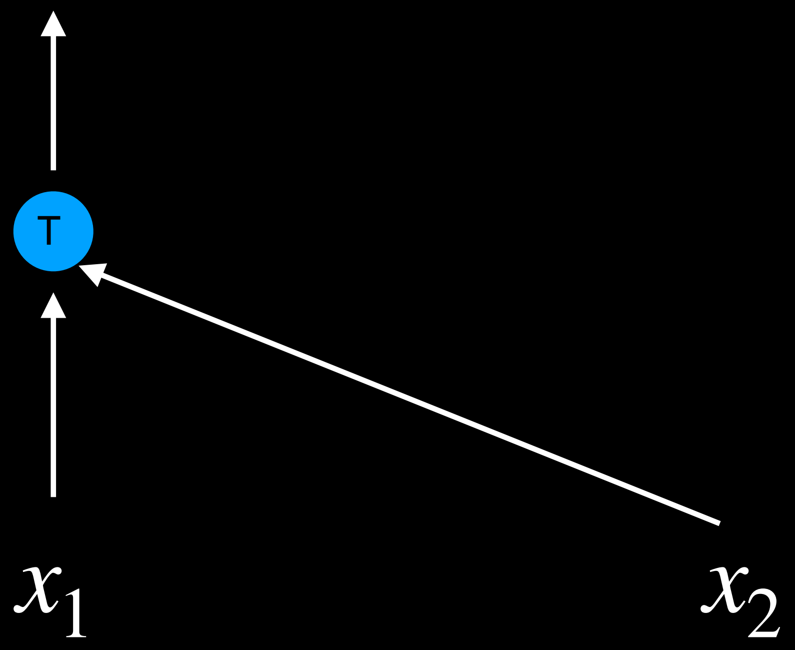
Can now be composed into "networks" to compute arbitrary classification "boundaries"



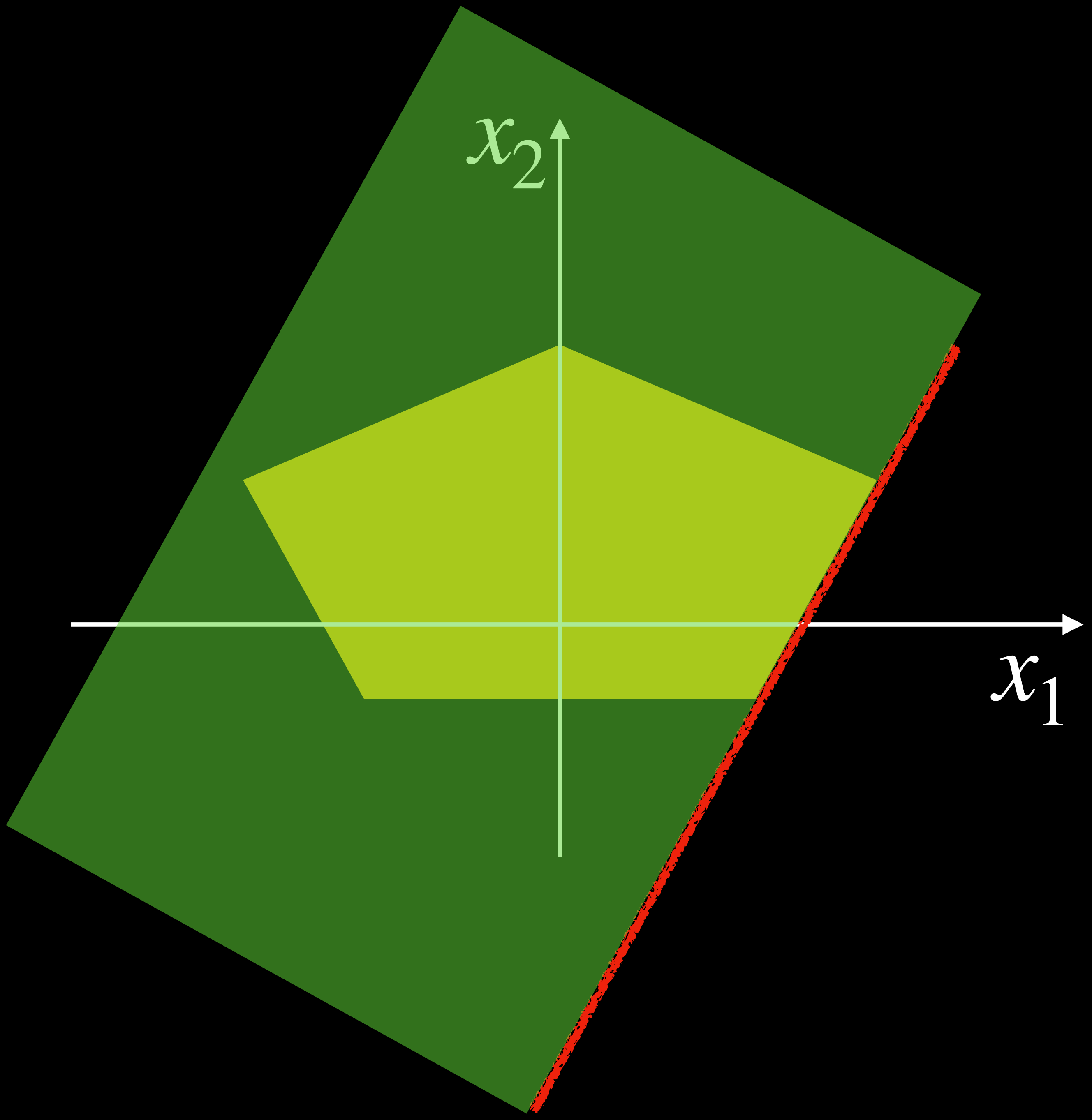
- Build a network of units with a single output that fires if the input is in the coloured area

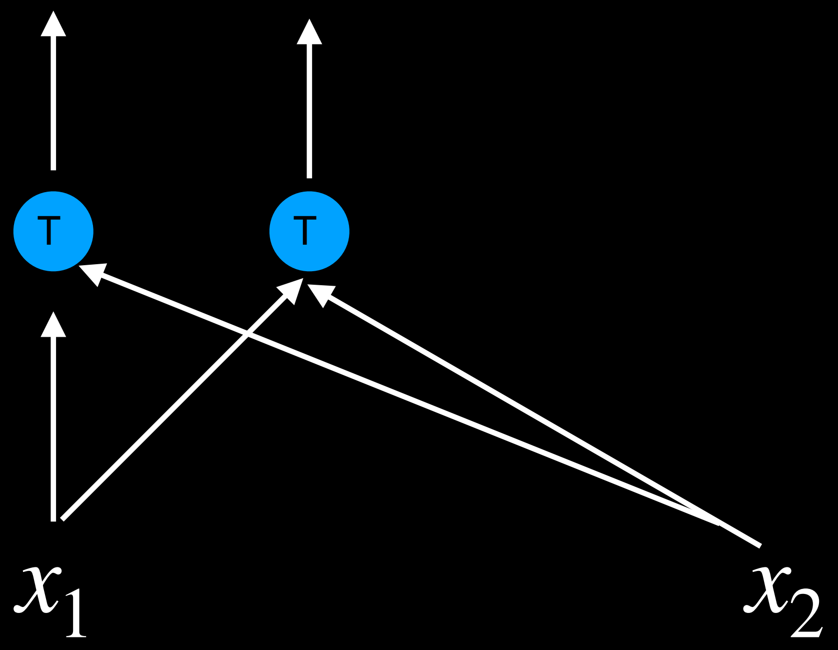
“Decision” Boundaries



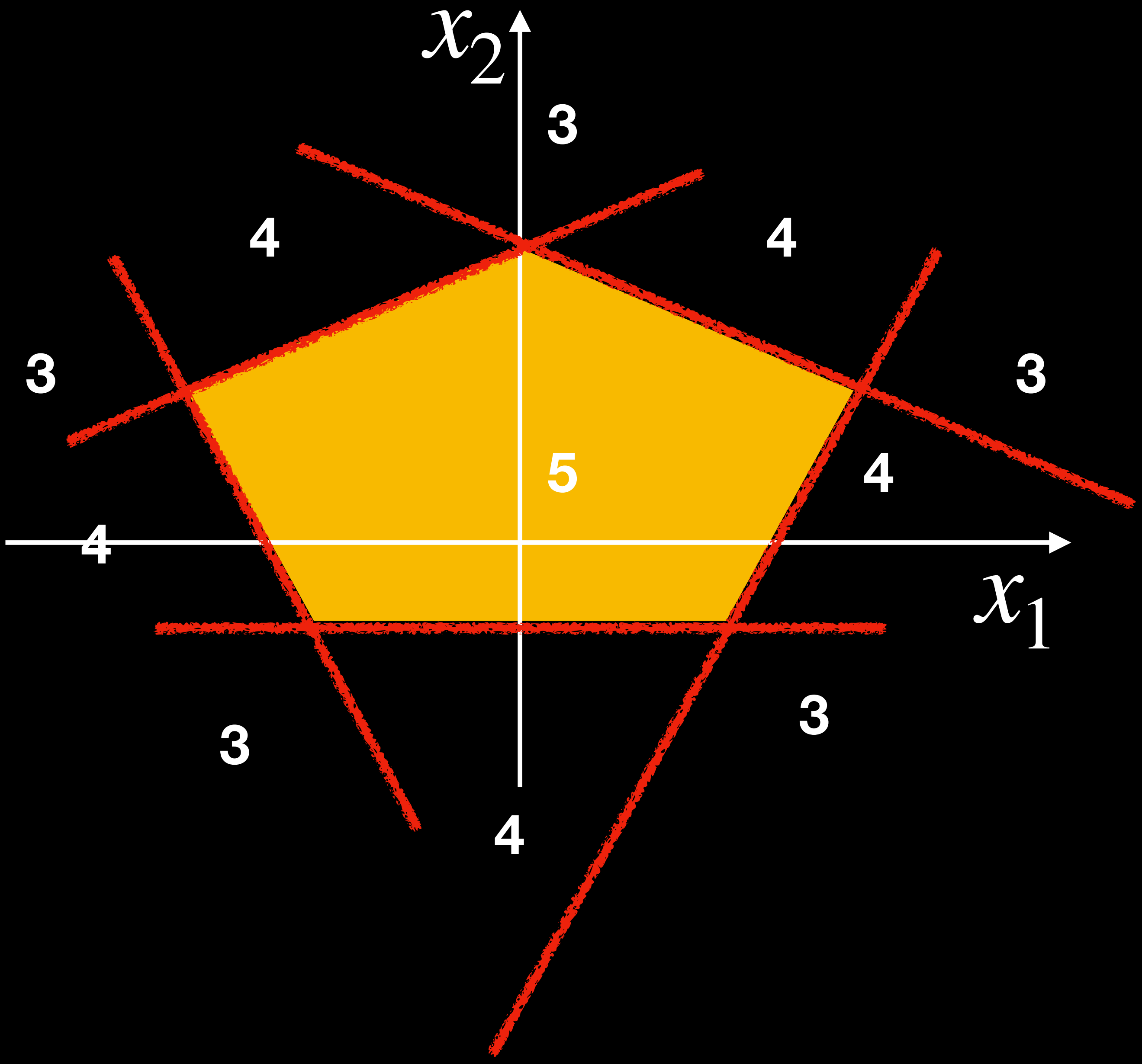


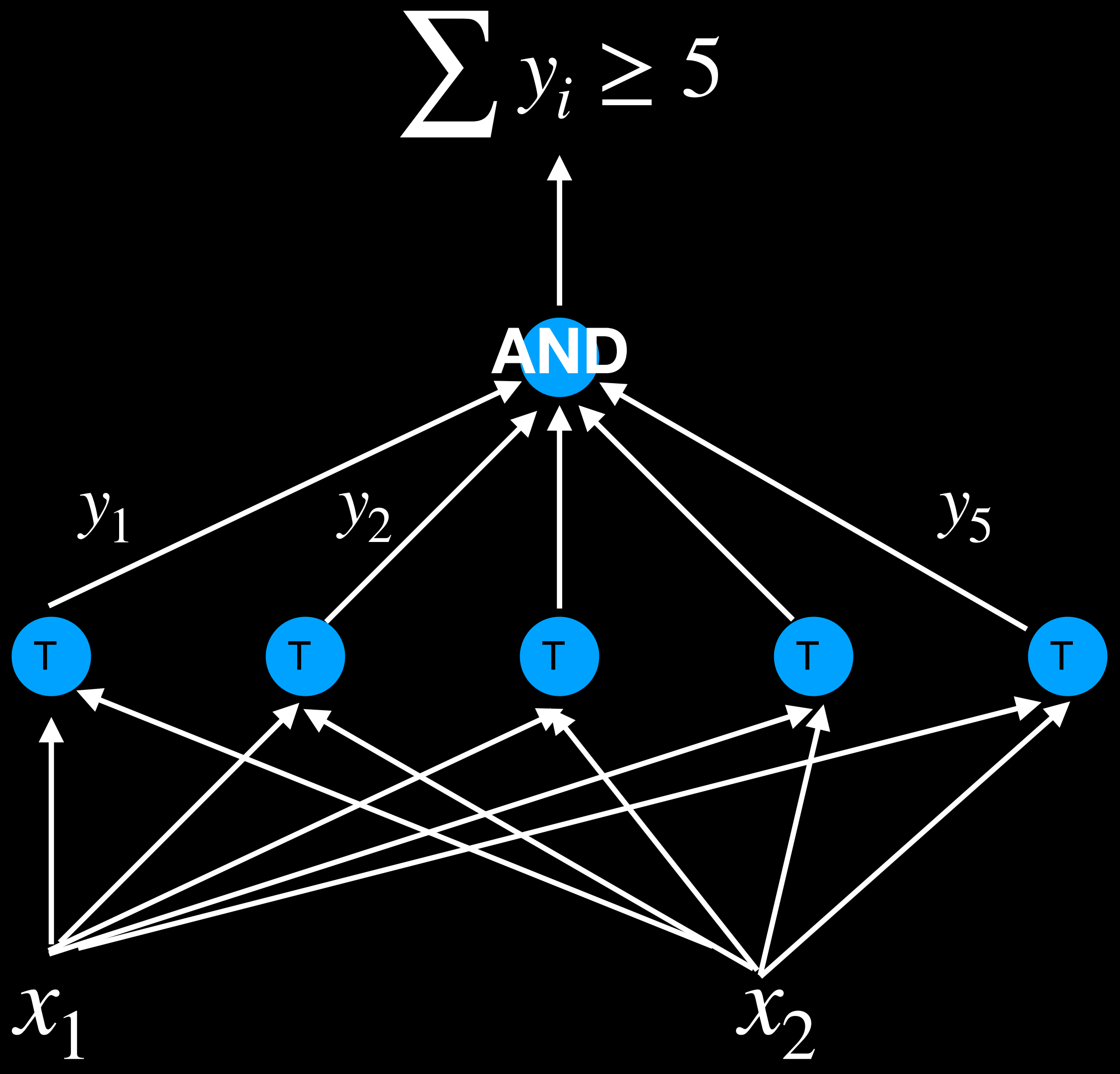
“Decision” Boundaries



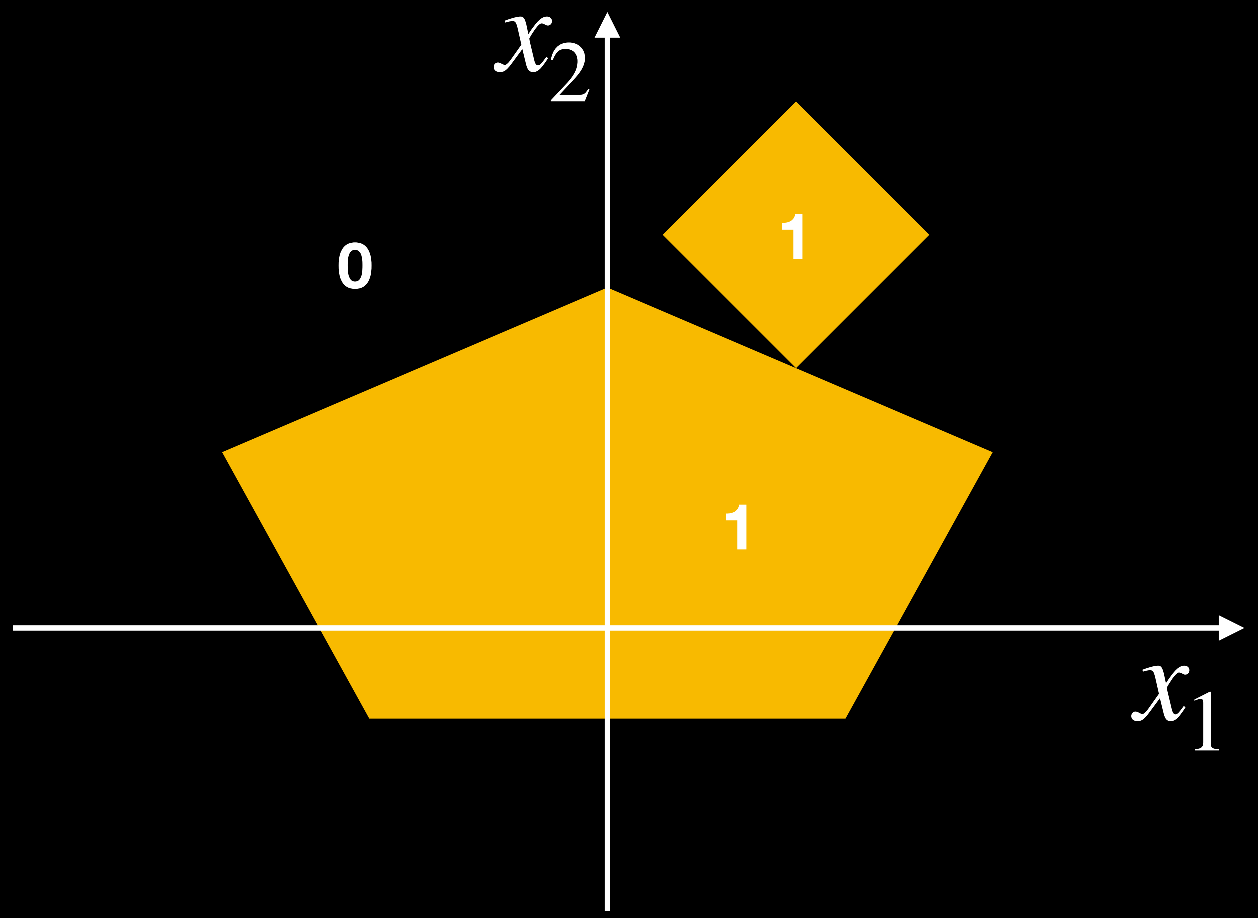


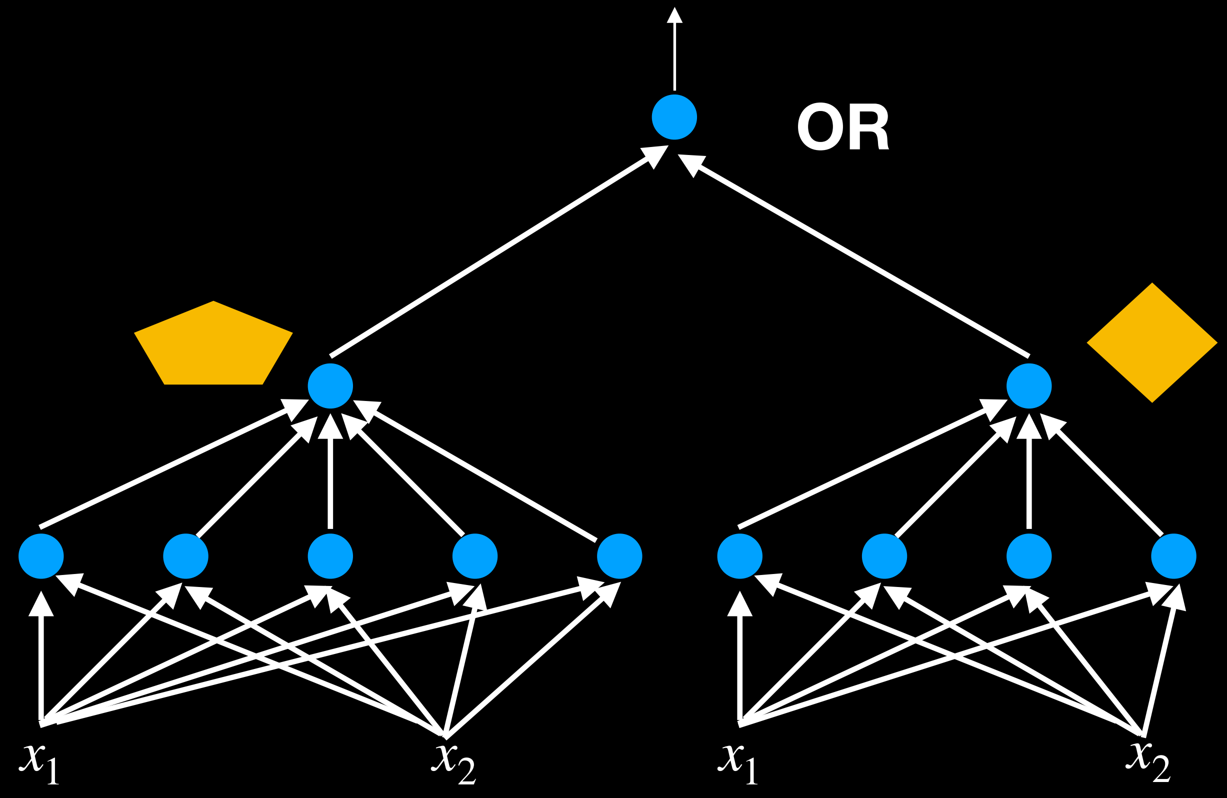
“Decision” Boundaries





“Decision” Boundaries





- Network to fire if the input is in the yellow area
  - – “OR” two polygons
  - – A third layer is required

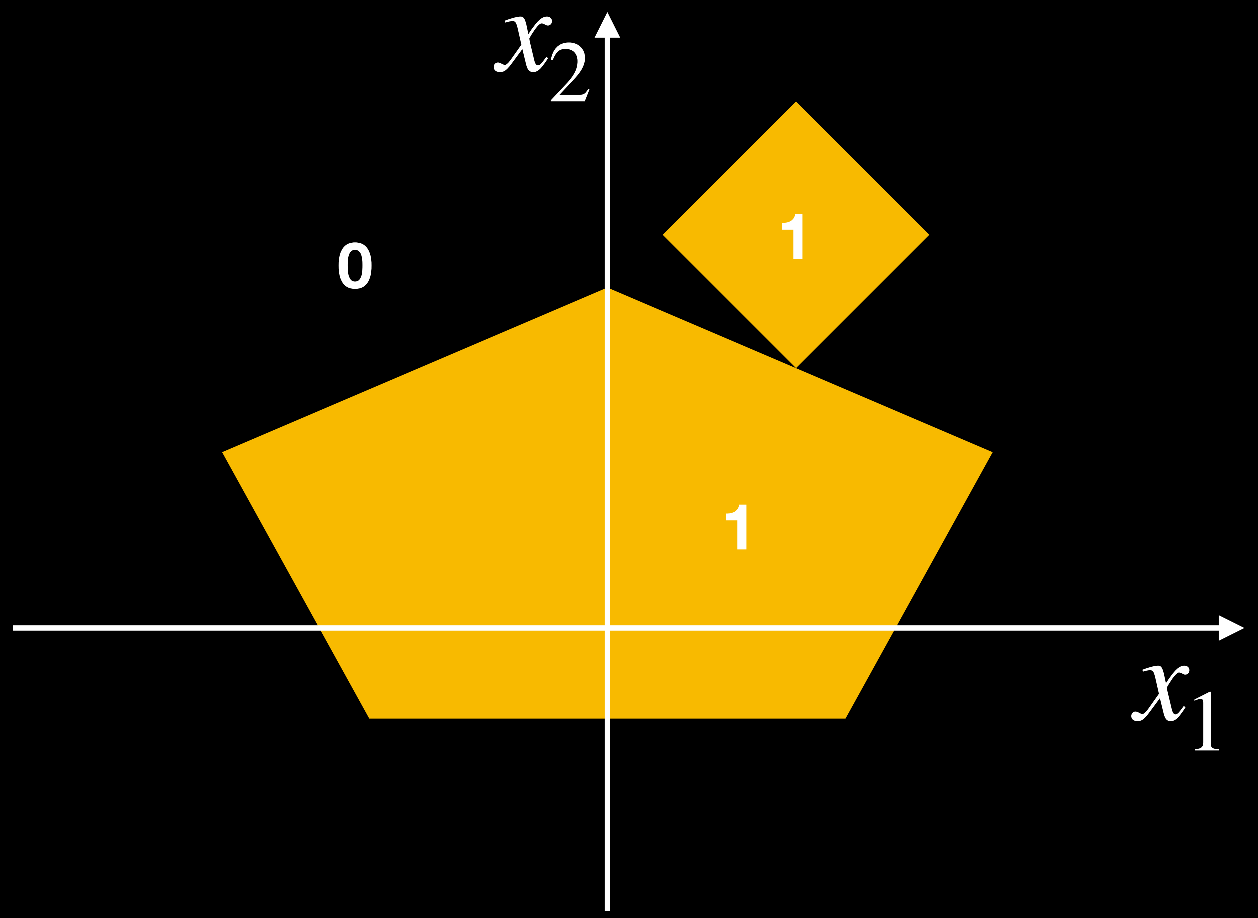
A summary

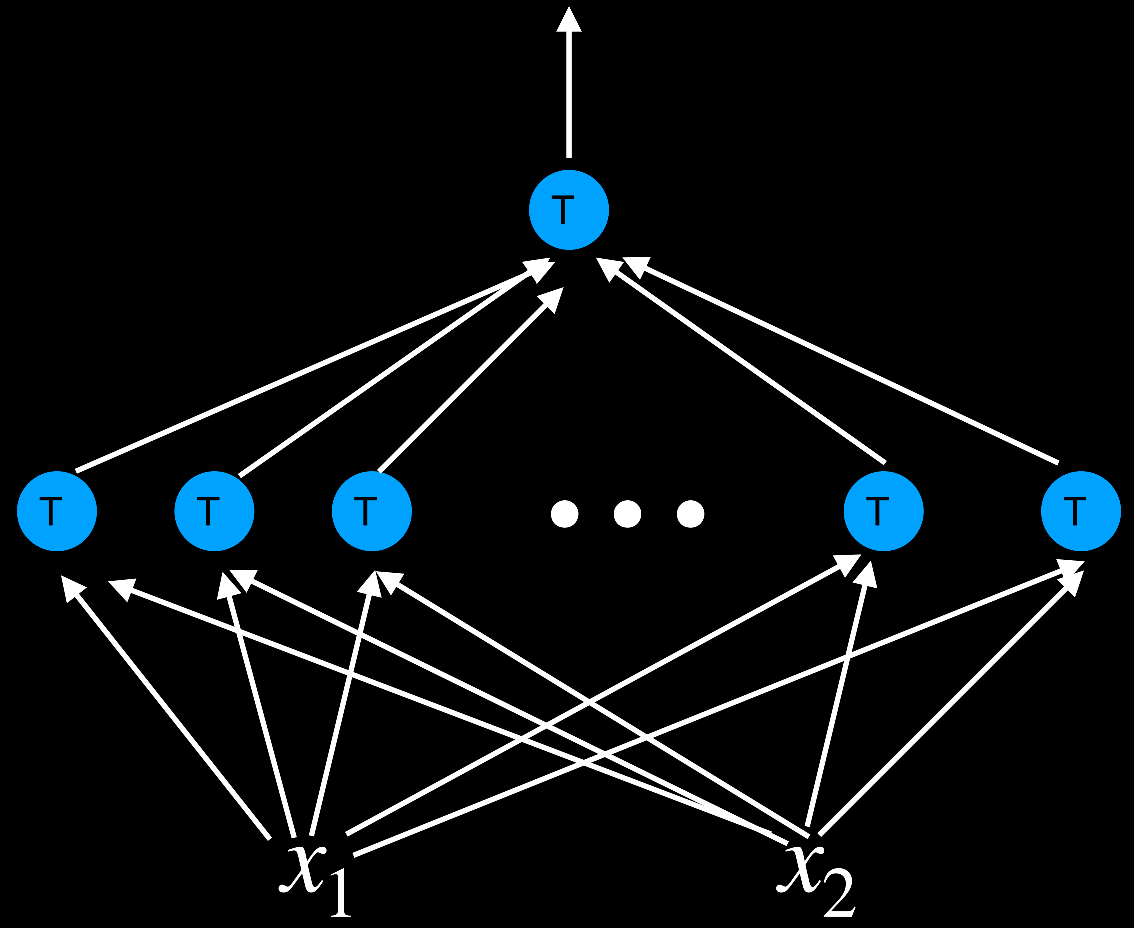
- MLPs are connectionist computational models
  - Individual perceptrons are computational equivalent of neurons
  - The MLP is a layered composition of many perceptrons

- MLPs can model Boolean functions
  - Individual perceptrons can act as Boolean gates
  - Networks of perceptrons are Boolean functions

- MLPs are Boolean machines
  - They represent Boolean functions over linear boundaries
  - They can represent arbitrary decision boundaries
  - They can be used to classify data

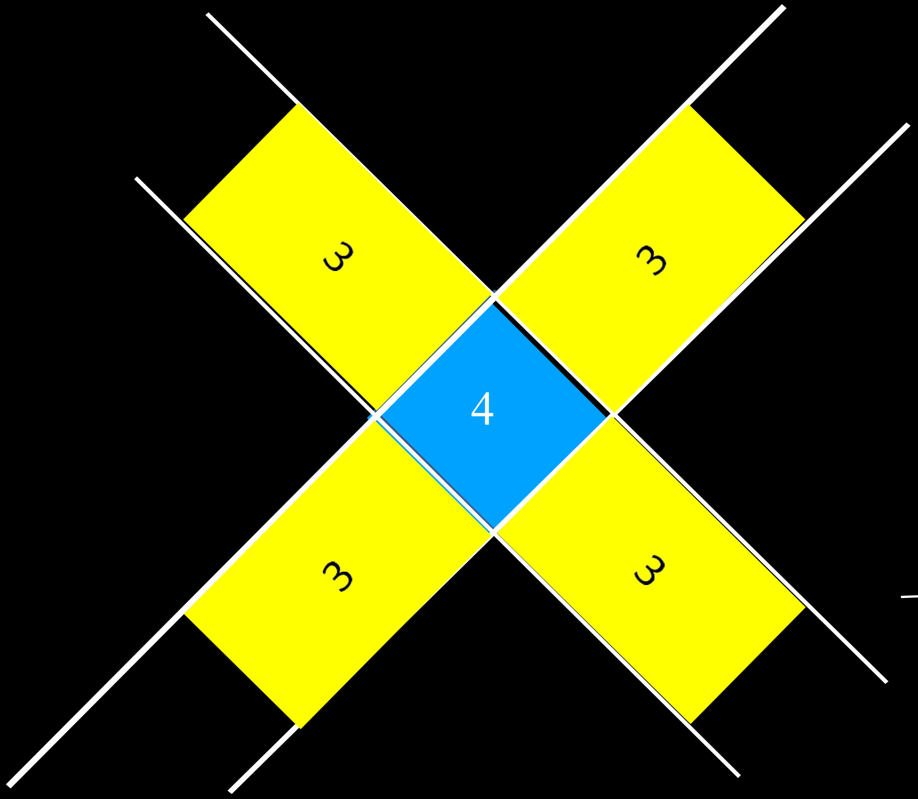
“Decision” Boundaries

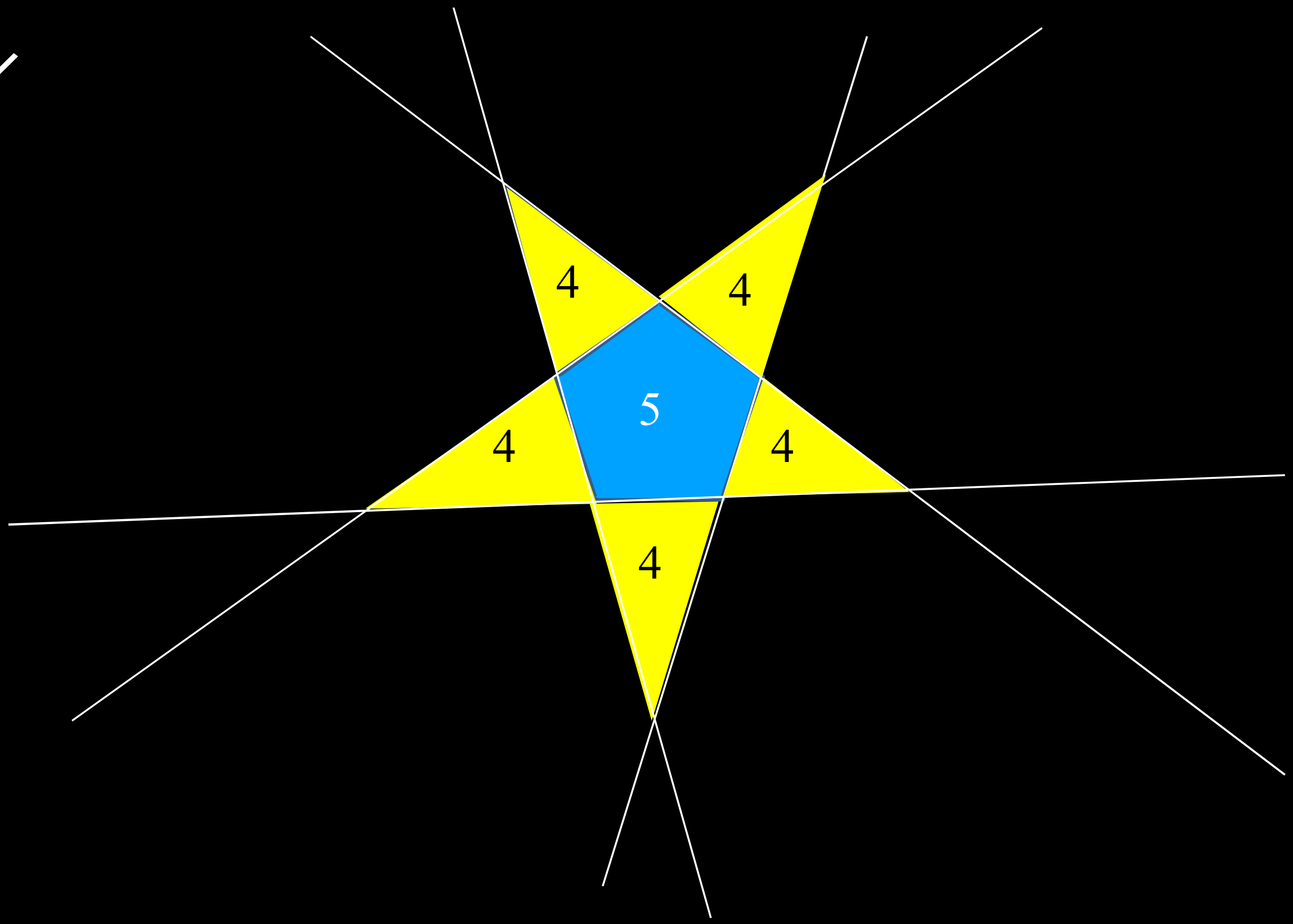


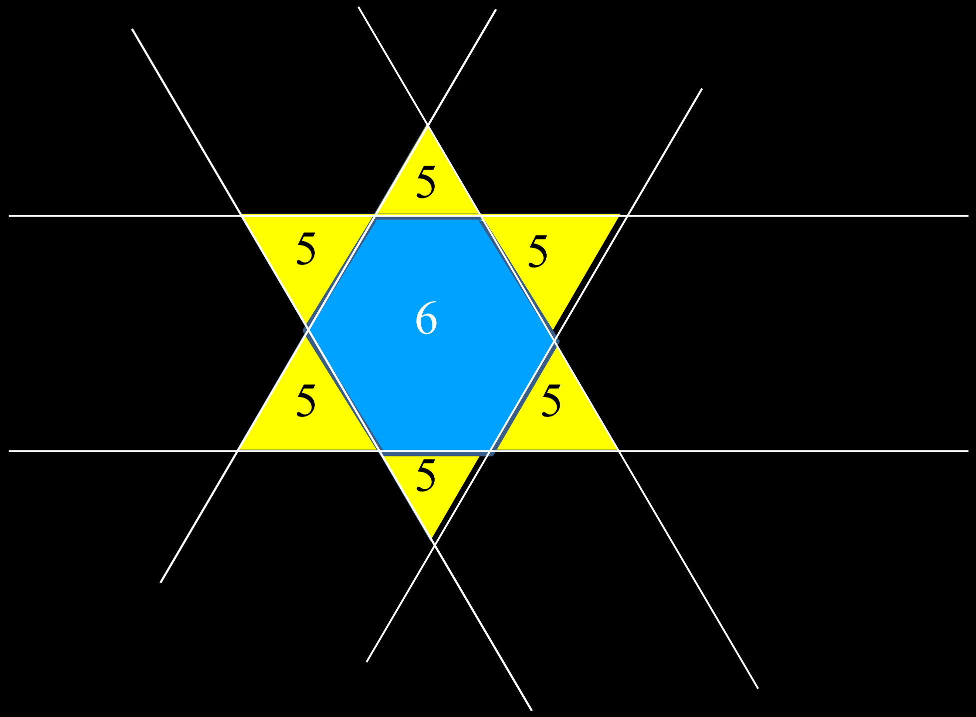


- **How would you compose the decision boundary to the left with only one hidden layer?**

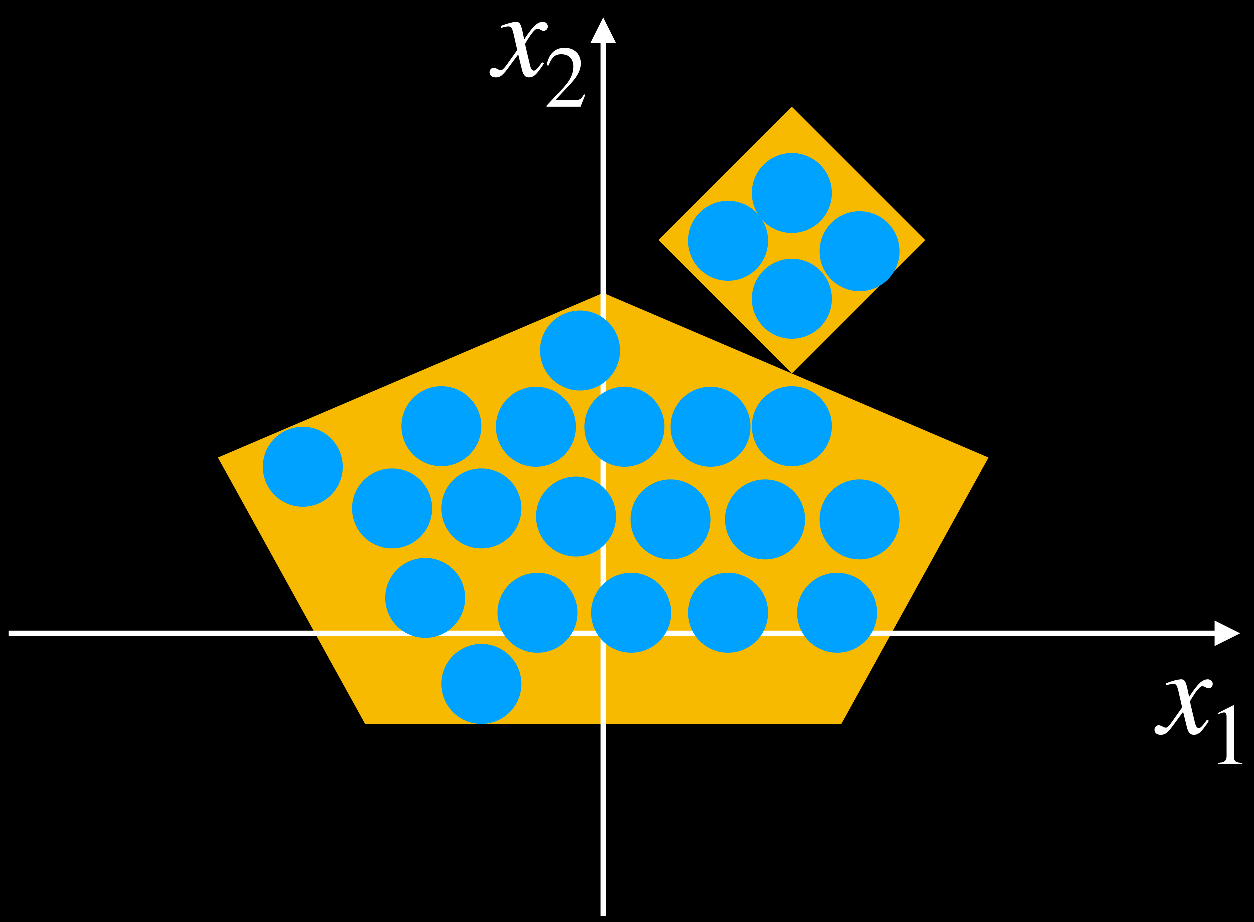
Composing decision boundaries

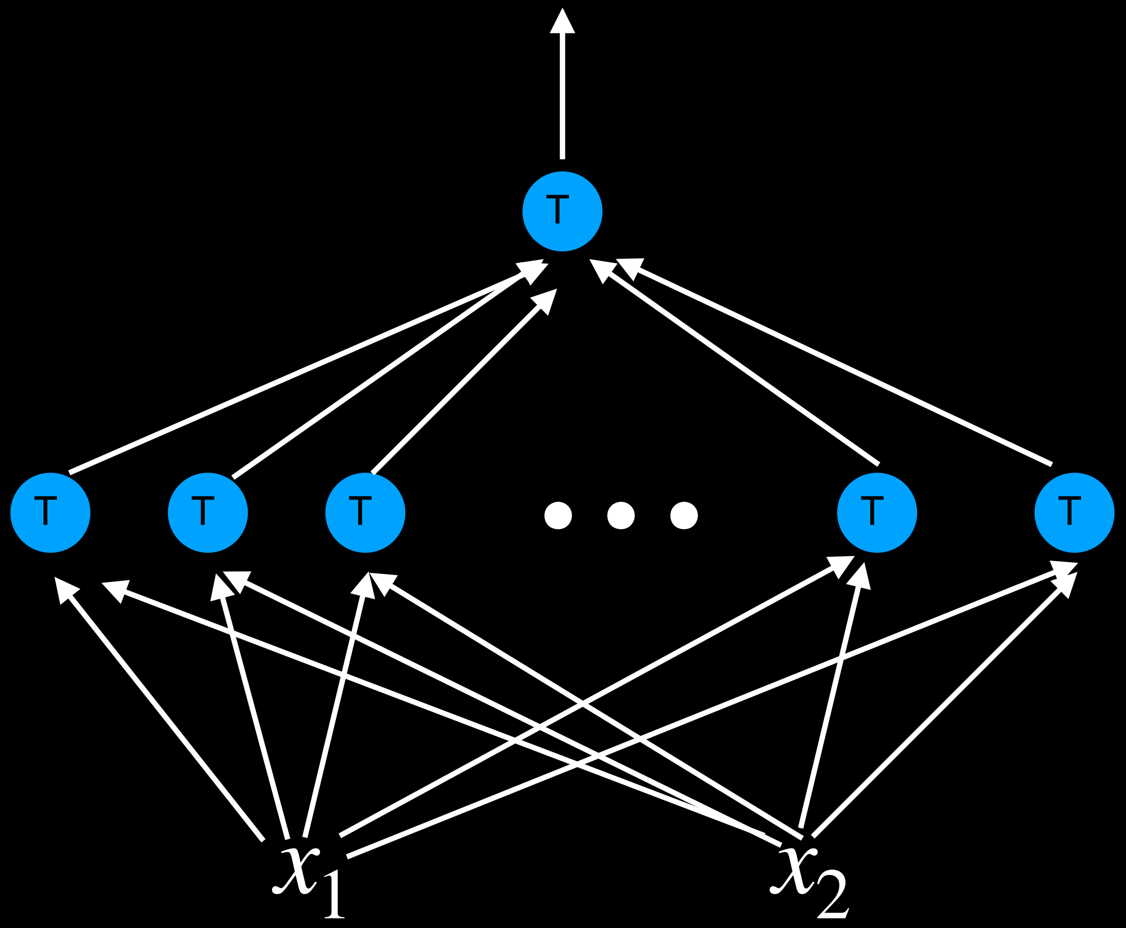






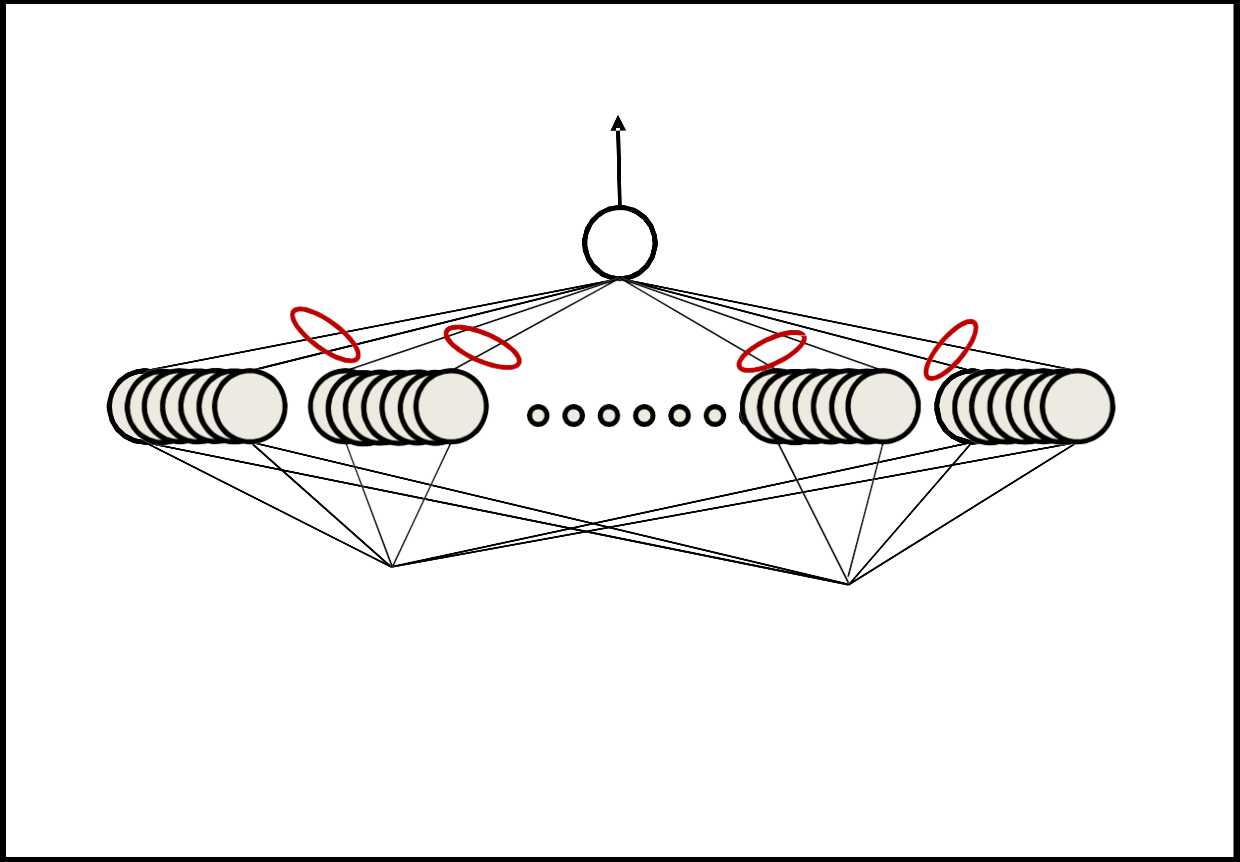
“Decision” Boundaries

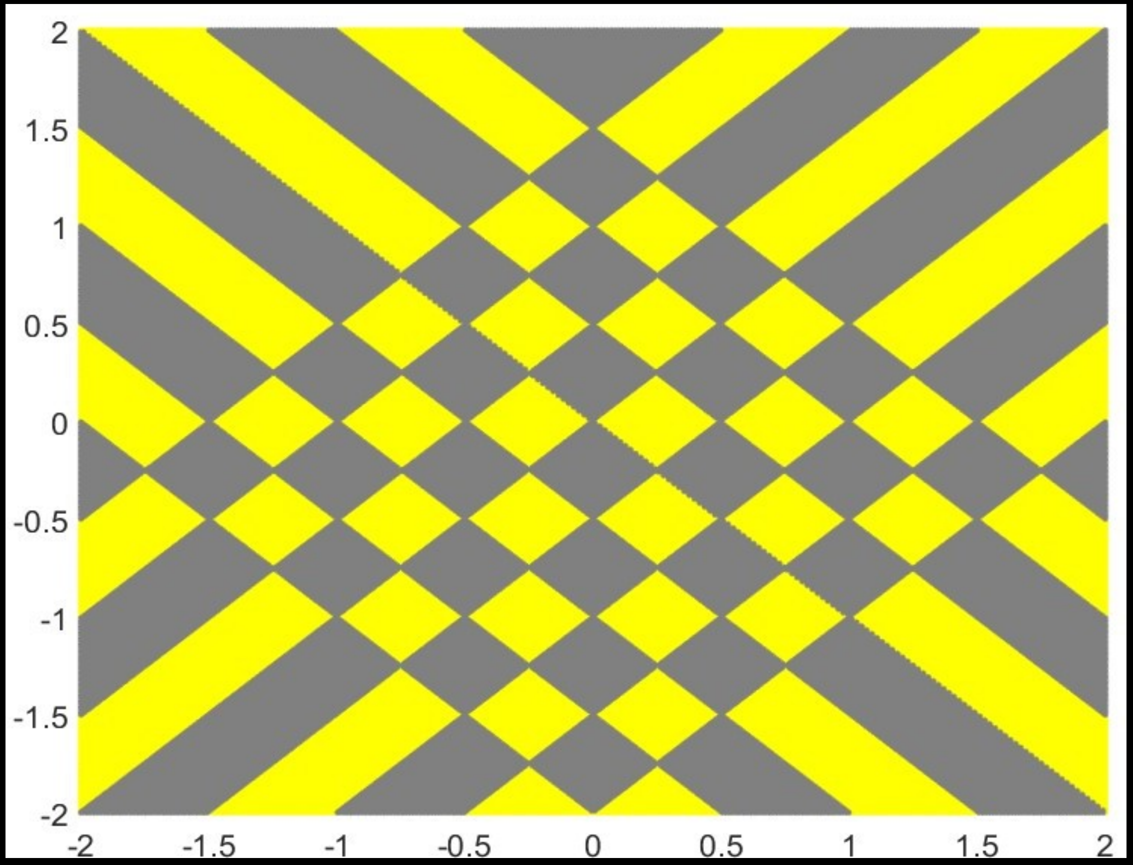




- **MLPs can capture any classification boundary**
- **A one-layer MLP can model any classification boundary**
- **MLPs are universal classifiers**

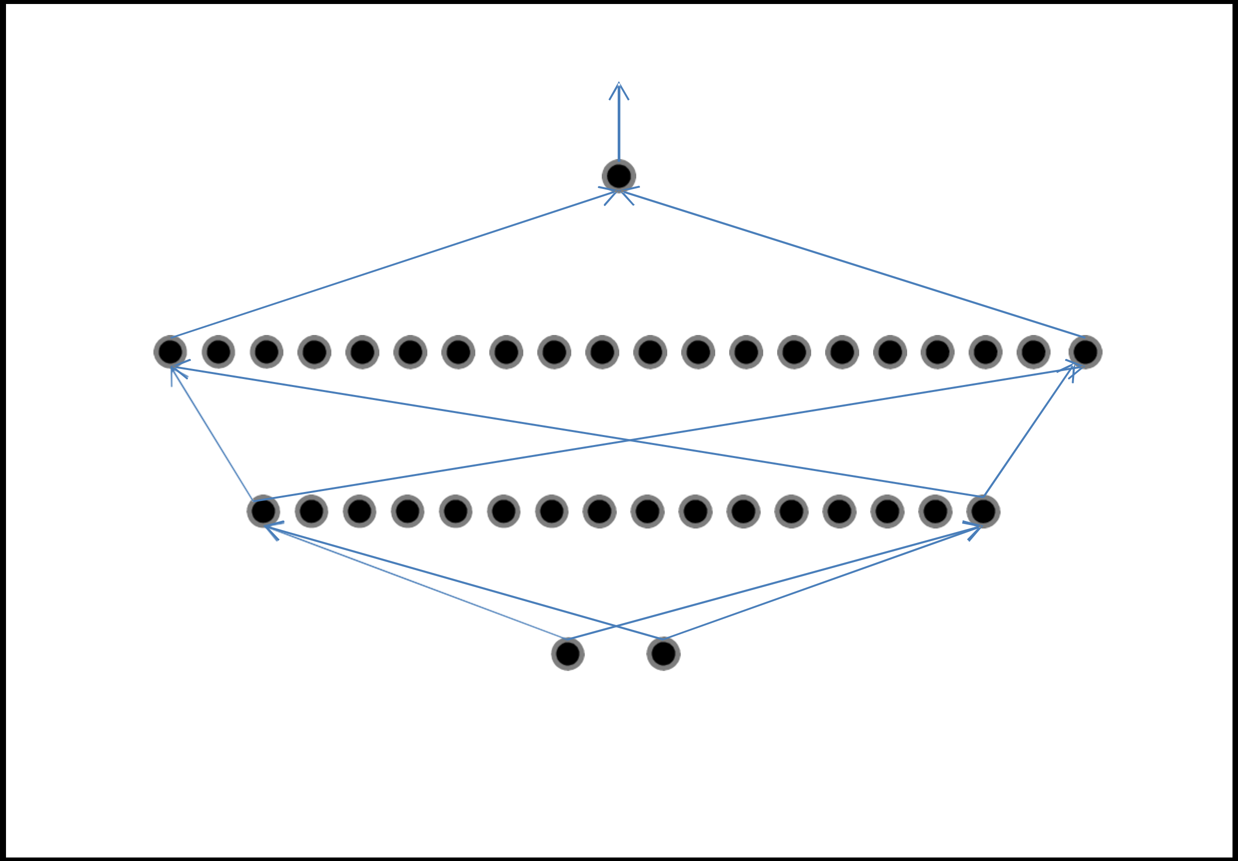
However…

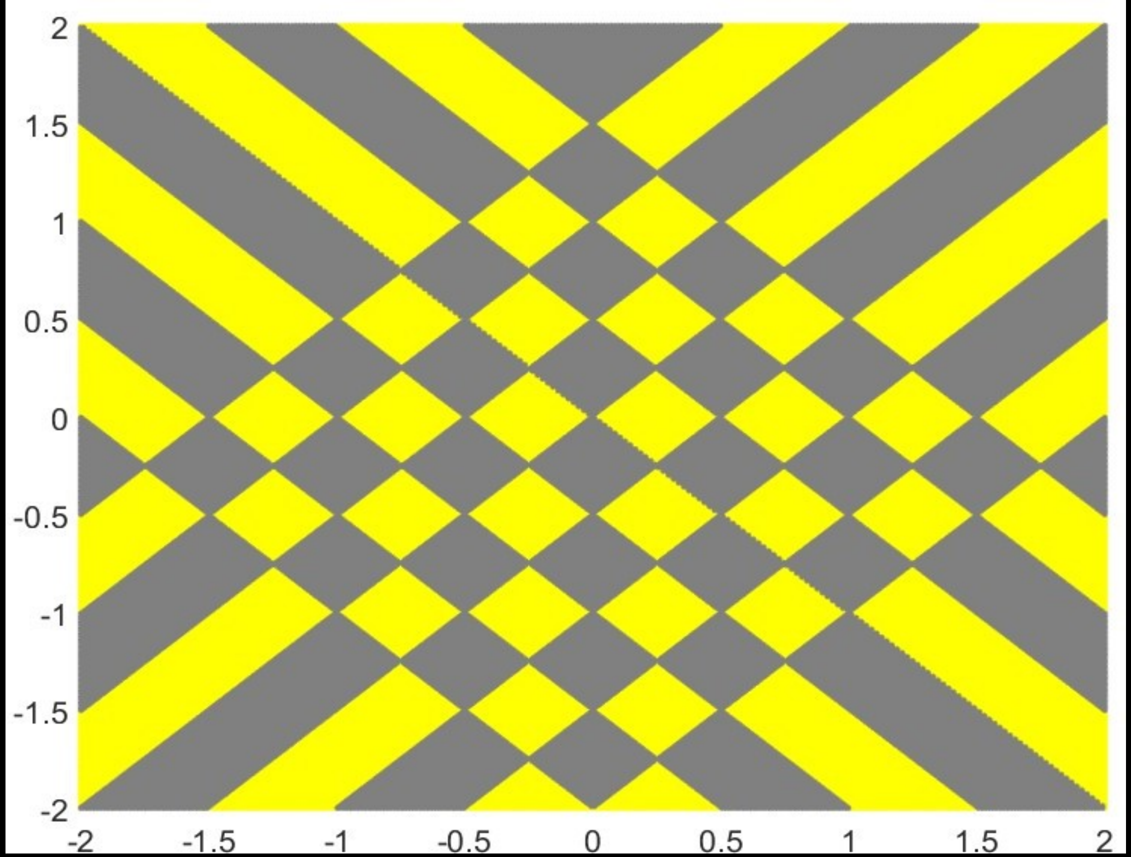




- Anaïve one-hidden-layer neural network will required infinite hidden neurons

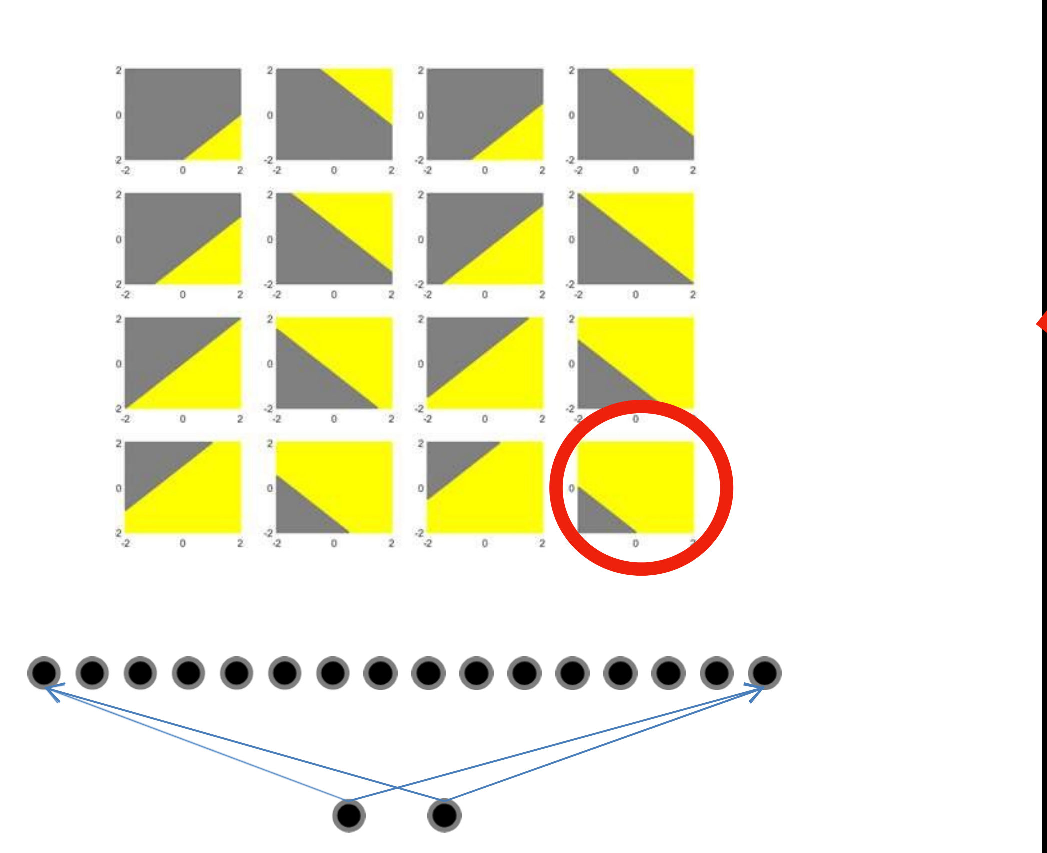
How to improve

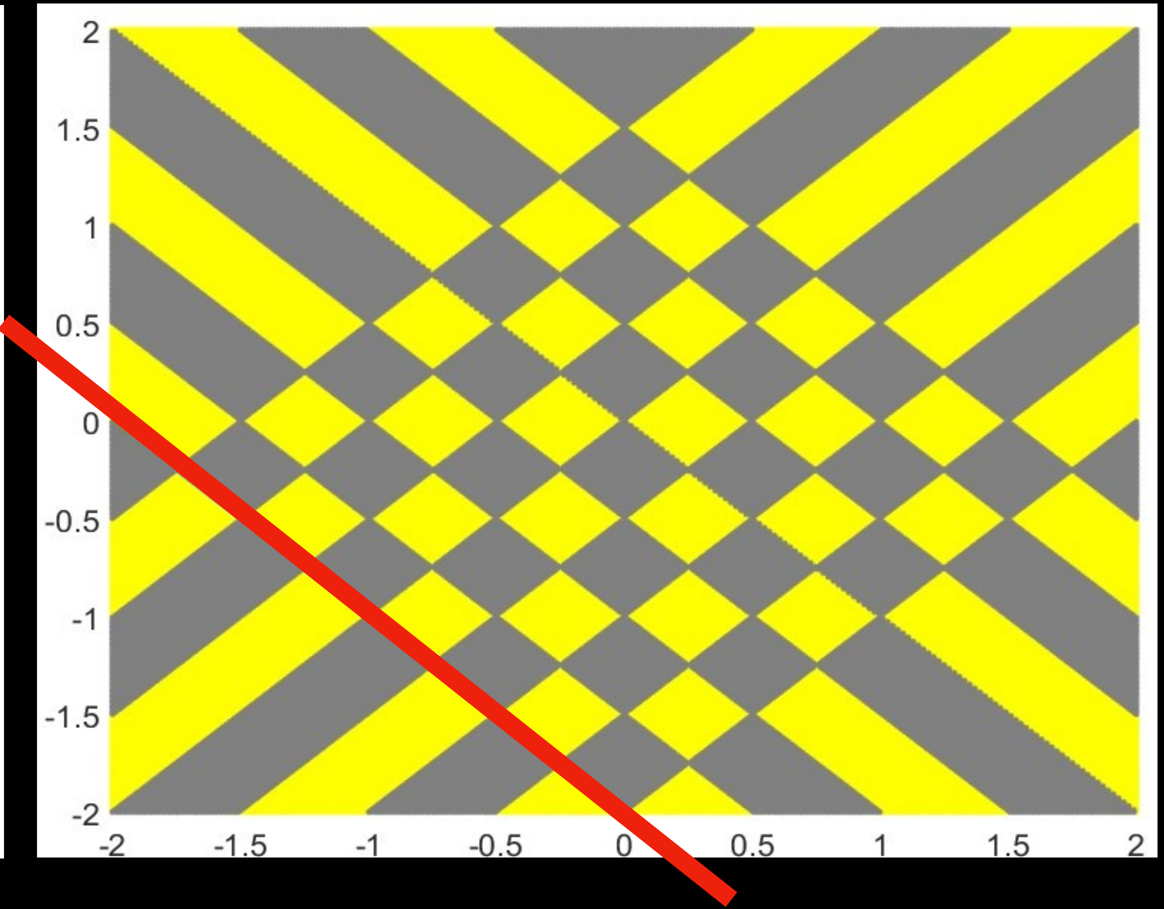




- Two hidden-layer network: 56 hidden neurons

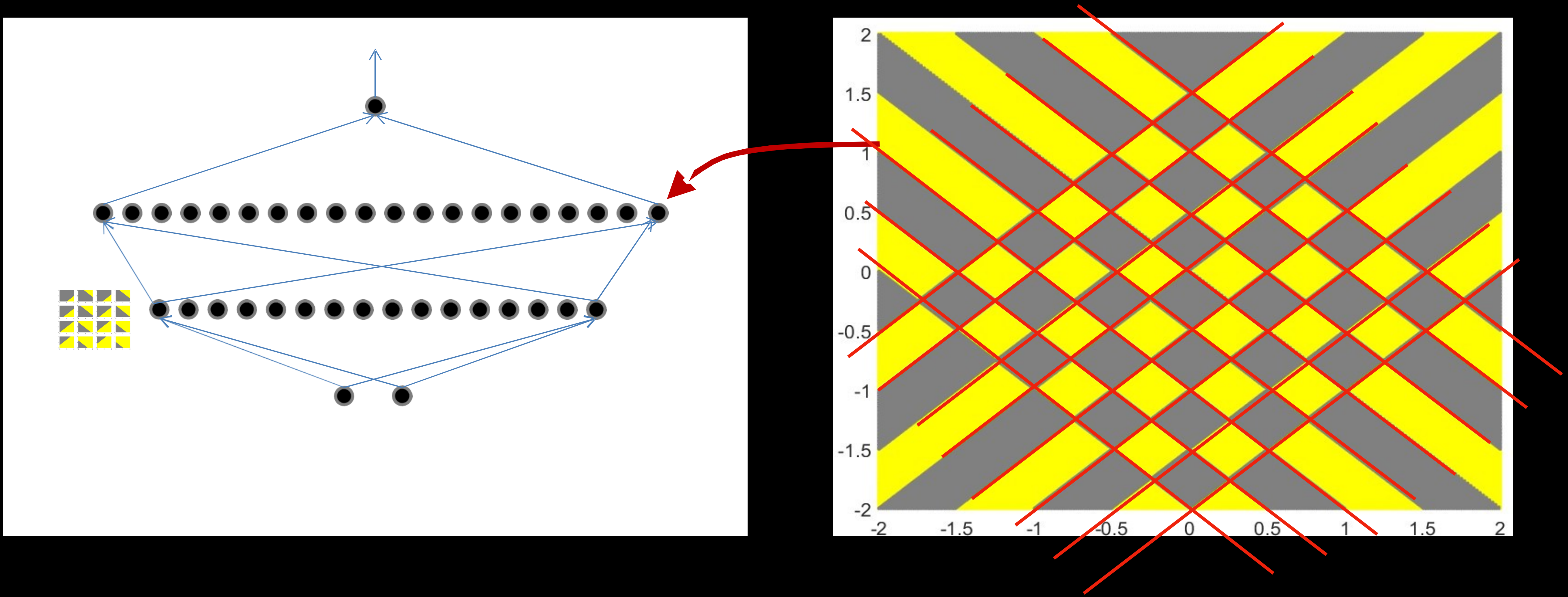
How to improve





- Two layer network: 56 hidden neurons
  - 16 neurons in hidden layer 1

How to improve



- **Two-layer network: 56 hidden neurons**
  - **16 in hidden layer 1**
  - **40 in hidden layer 2 ($\lfloor (n+2)^2/8 \rfloor$)**
  - **57 total neurons, including output neuron**

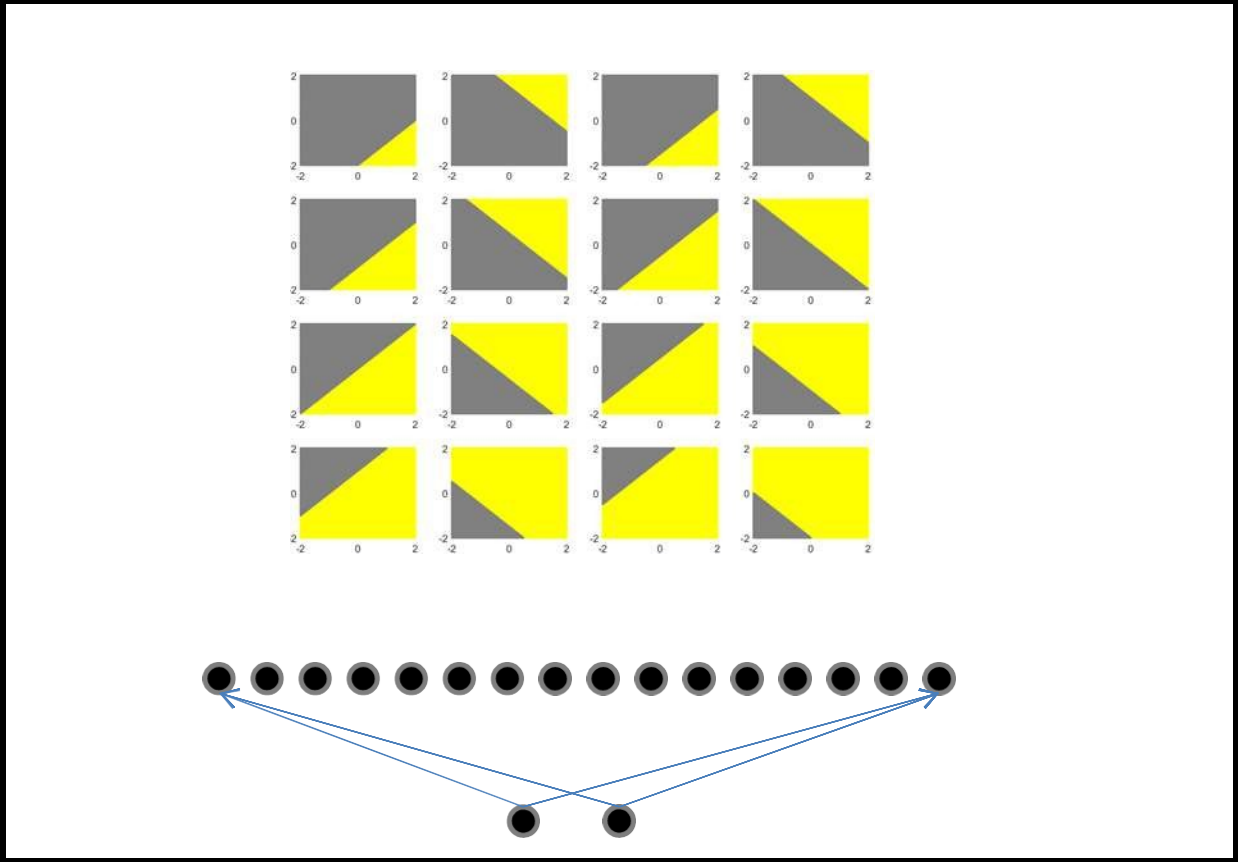
Depth

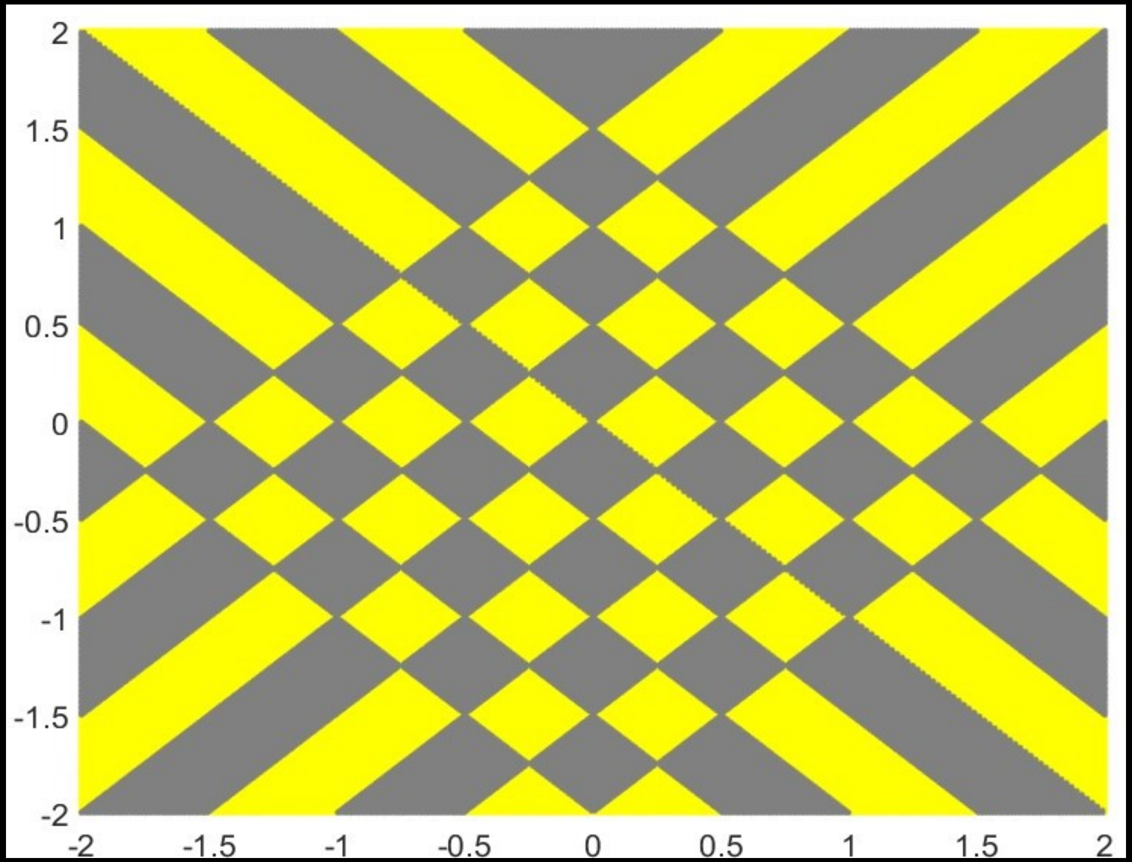
- "Shallow vs deep sum-product networks,"
  Oliver Dellaleau and Yoshua Bengio
  - For networks where layers alternately perform either sums or products, a deep network may require an exponentially fewer number of layers than a shallow one.

- The number of neurons required in a shallow network is potentially exponential in the dimensionality of the input
  - Alternately, exponential in the number of statistically independent features

The features

Not independent features





- **Deep neural network can extract the features**

A summary

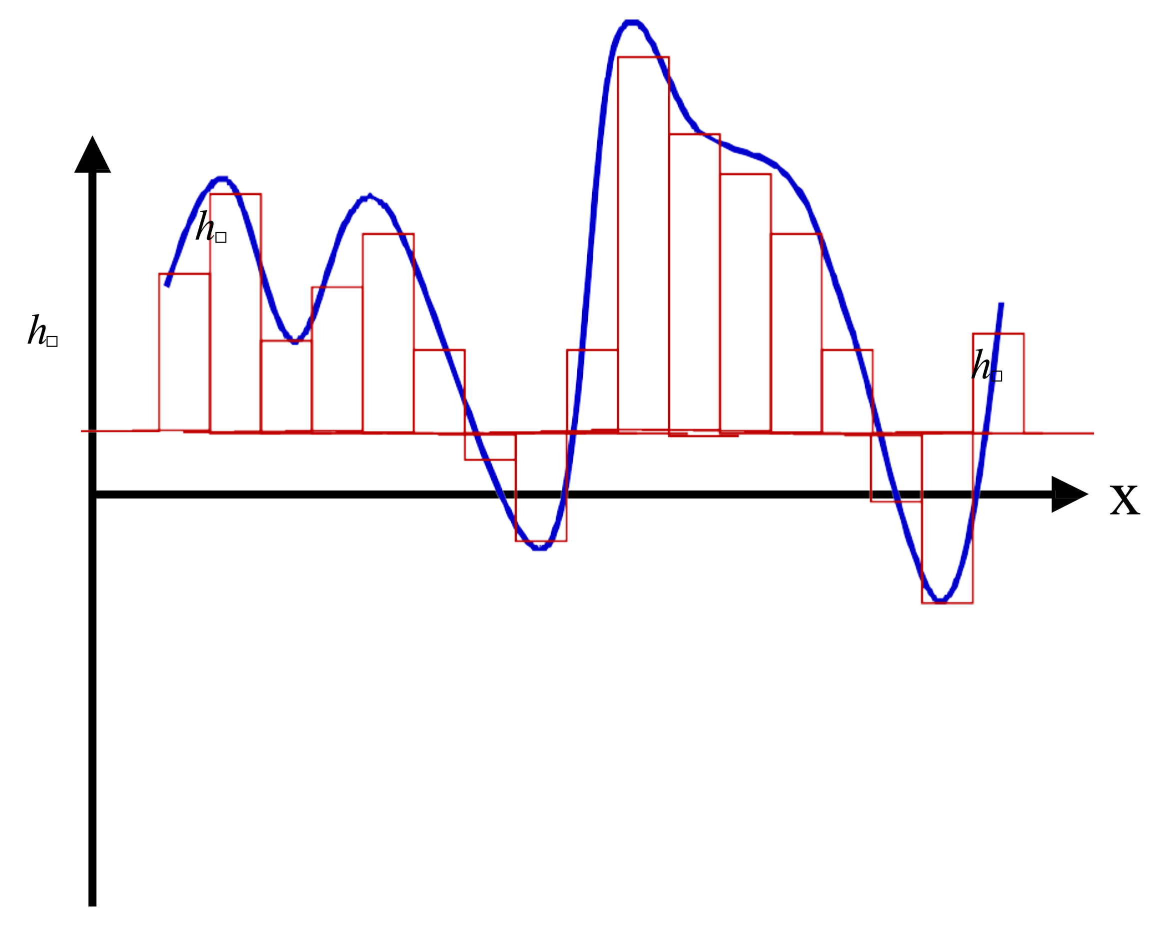
- Multi-layer perceptrons are Universal Boolean Machines
  - Even a network with a single hidden layer is a universal Boolean machine

- Multi-layer perceptrons are Universal Classification Functions
  - Even a network with a single hidden layer is a universal classifier

- But a single-layer network may require an exponentially large number of perceptrons than a deep one

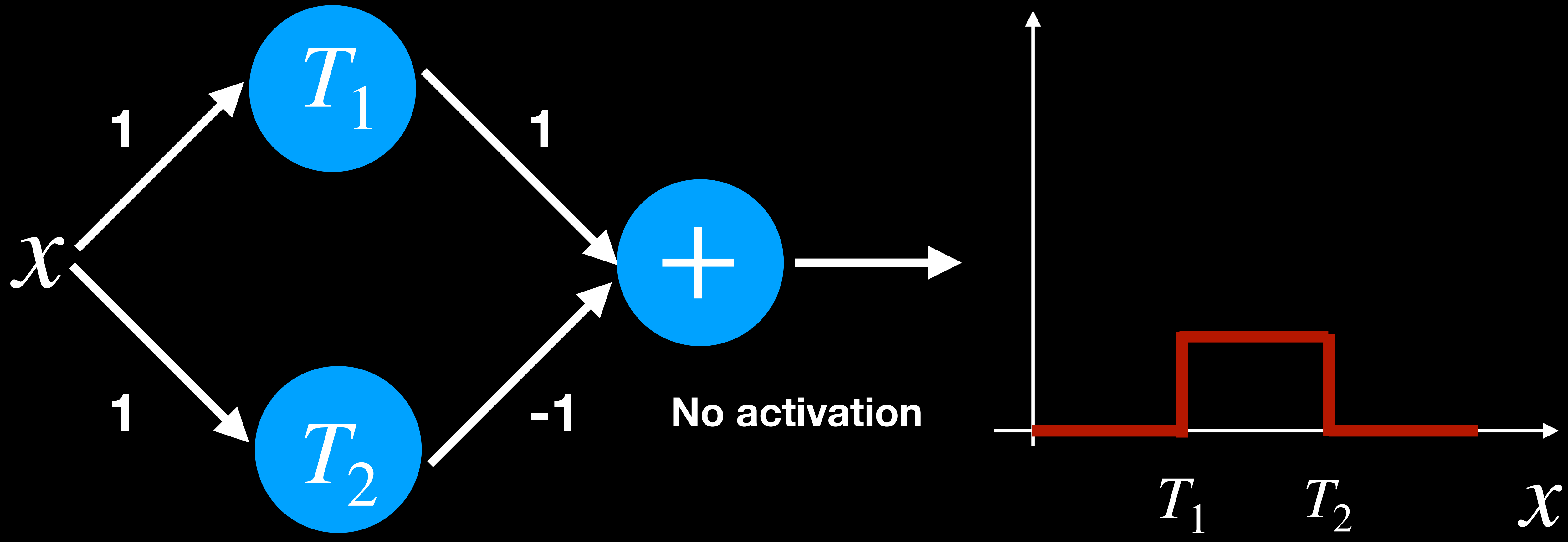
- Deeper networks may require far fewer neurons than shallower networks to express the same function
  - Could be exponentially smaller
  - Deeper networks are more expressive

Function Approximation (single input)



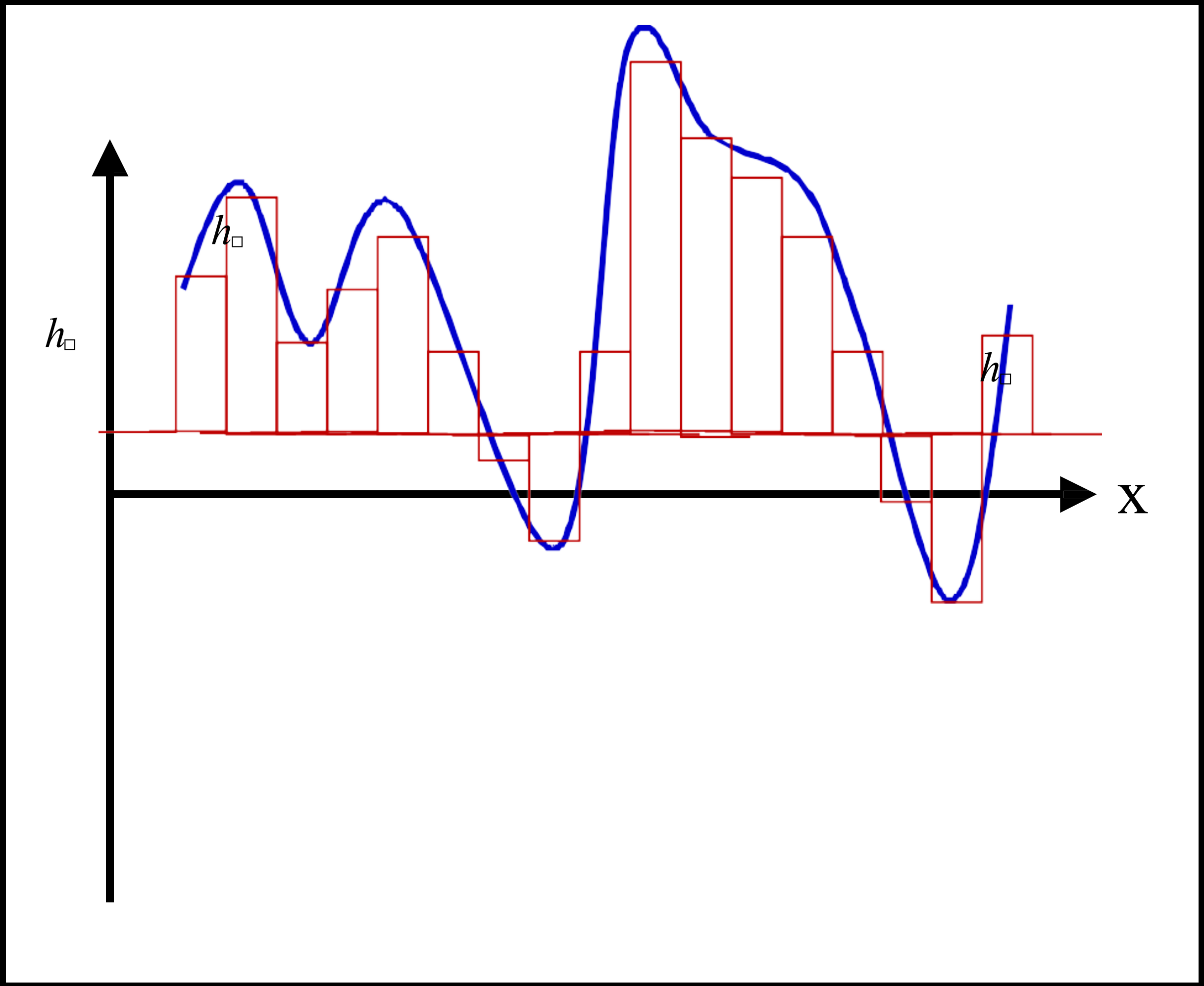
How to approximate function above by using threshold MLPs?

Function Approximation (single input)



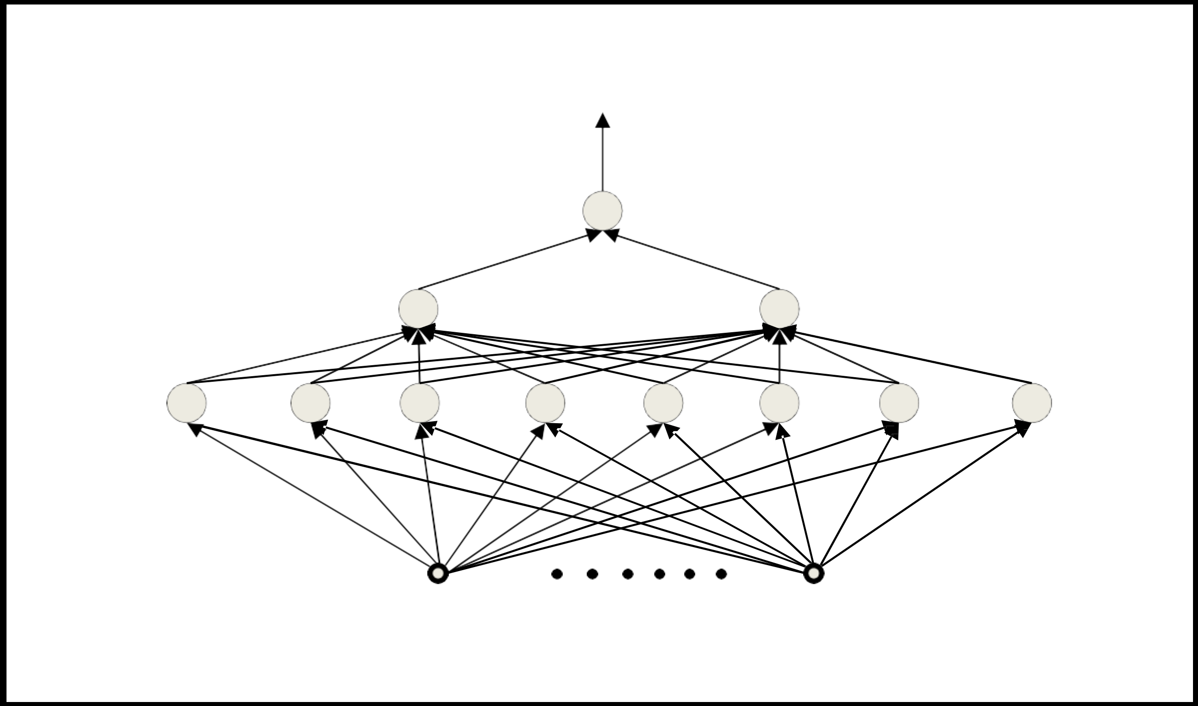
- A simple 3-unit MLP with a "summing" output unit can generate a "square pulse" over an input
  - Output is 1 only if the input lies between T_1 and T_2
  - T_1 and T_2 can be arbitrarily specified

Function Approximation(single input)



- A simple 3-unit MLP can generate a "square pulse" over an input
- A MLP with many units can model an arbitrary function over an input
  - – To arbitrary precision
    - Simply make the individual pulses narrower
- This generalizes to functions of any number of inputs

Think the network as a function



$$f : \{0,1\} \rightarrow \{0,1\}$$

$$f : R^n \rightarrow \{0,1\}$$

$$f : R^n \rightarrow (0,1)$$

$$f : R^n \rightarrow (-1,1)$$

$$f : R^n \rightarrow [0,\infty)$$

- Output unit with activation function
  - Threshold or Sigmoid, ReLU or any other

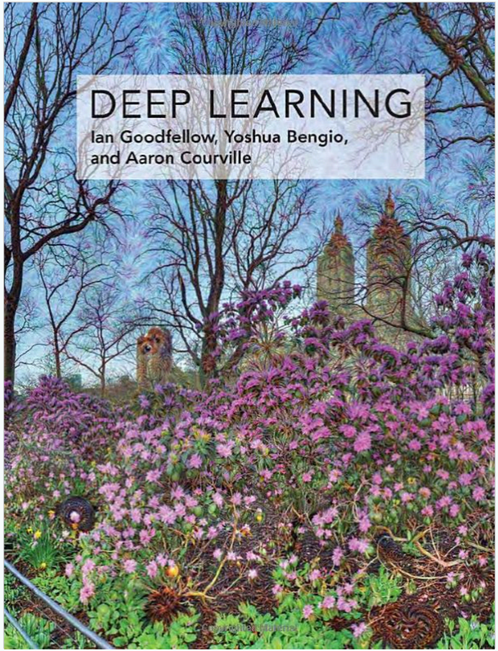
- The network is actually a universal map from the entire domain of input values to the entire range of the output activation
  - All values the activation function of the output neuron

A summary

- **Multi-layer perceptrons are Universal Boolean Machines**
  - **Even a network with a single hidden layer is a universal Boolean machine**

- **Multi-layer perceptrons are Universal Classification Functions**
  - **Even a network with a single hidden layer is a universal classifier**

- **Multi-layer perceptrons are Universal Function approximate for entires class of functions (maps) it represents**



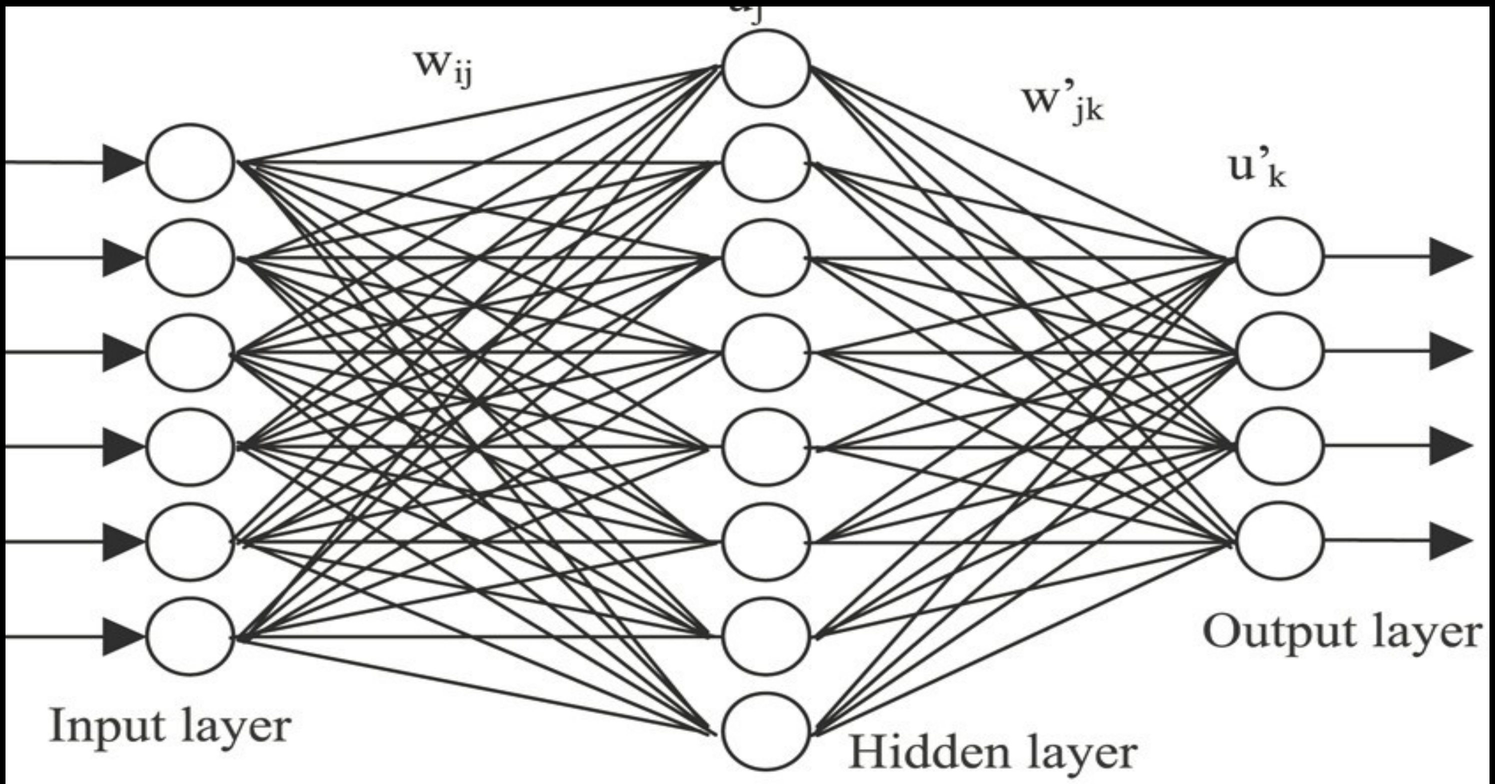
In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.

-P.193

Think the network as a function

See Code

A summary



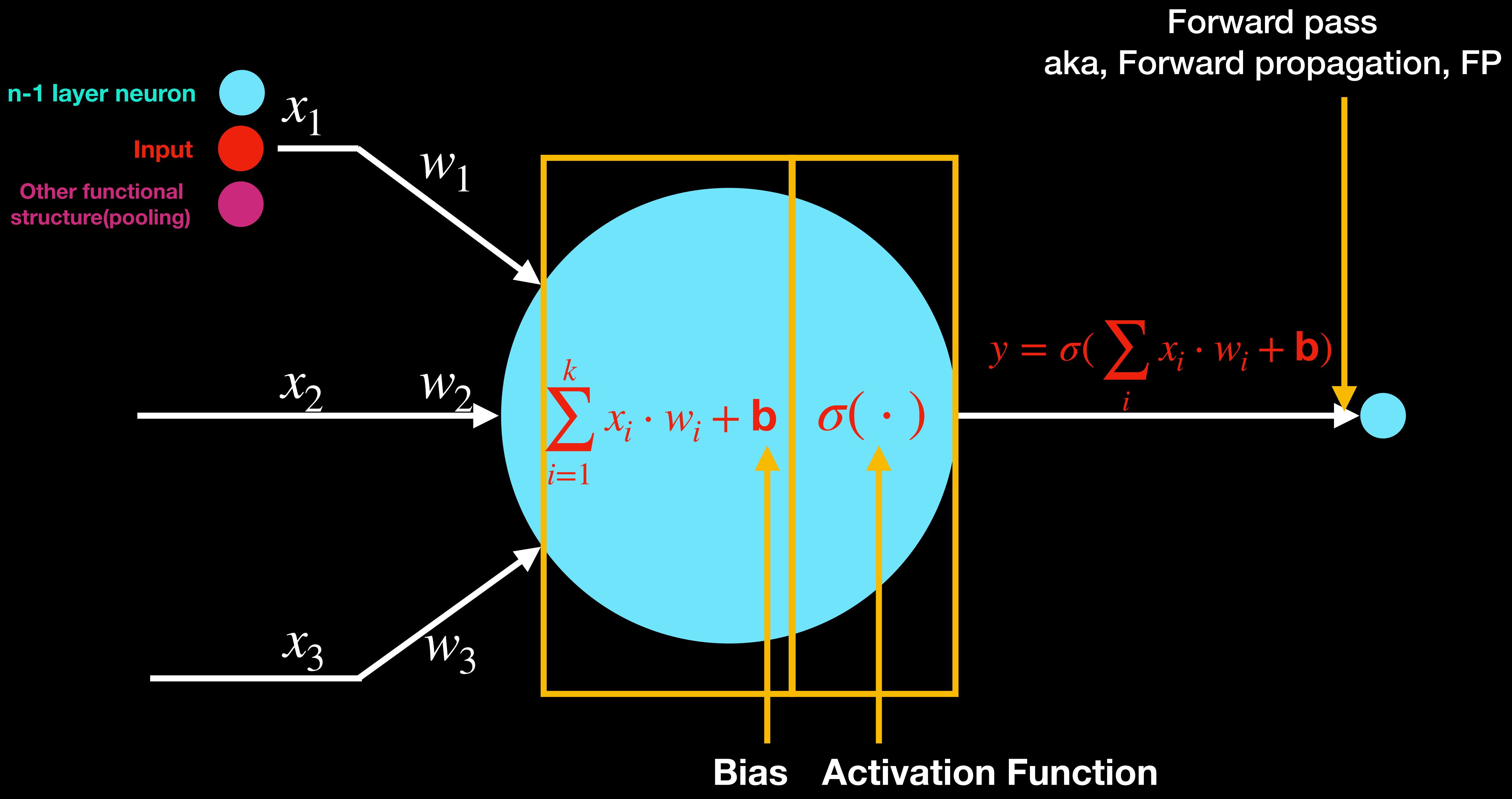
- Neural networks are universal function approximators
  - Can model any Boolean function
  - Can model any classification boundary
  - Can model any continuous valued function

The “capacity” of a network

- VC dimension
- Some Papers
  - Koiran and Sontag (1998): For “linear” or threshold units, VC dimension is proportional to the number of weights
    - For units with piecewise linear activation it is proportional to the square of the number of weights
  - Batlett, Harvey, Liaw, Mehrabian “Nearly-tight VC-dimension bounds for piecewise linear neural networks” (2017):
    - For any W, L s.t W> CL > C^2, there exists a ReLU network with less Players, less W weights with VC dimension > $\frac{WL}{C}\log_2(\frac{W}{L})$
- Network capacity, generalization ability, etc

The Perceptron

The structural building block of deep learning



- 🔵 n-1 layer neuron
- 🔴 Input
- 🟣 Other functional structure(pooling)

Forward pass
aka, Forward propagation, FP

$$y = \sigma(\sum_{i} x_i \cdot w_i + \mathbf{b})$$

$$\sum_{i=1}^{k} x_i \cdot w_i + \mathbf{b}$$

$$\sigma(\cdot)$$

Bias

Activation Function

The Perceptron: Bias

The structural building block of deep learning

Are you going to have lunch in Sadler Center?

Weather: 0 or 1 $w_1 = 1$

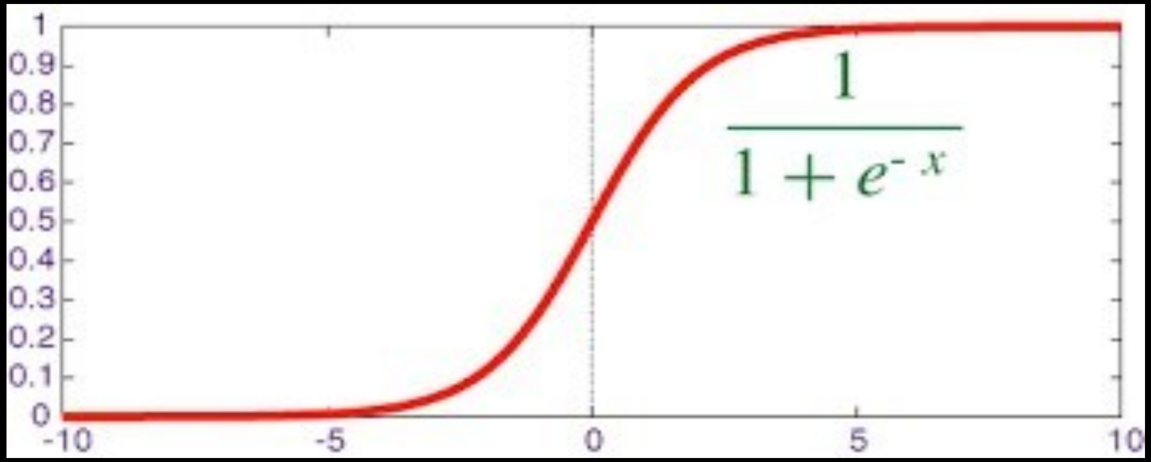
Foods: 0 or 1 $w_2 = 1$

Dinning Dollar: 0 or 1 $w_3 = 1$

If b = 0

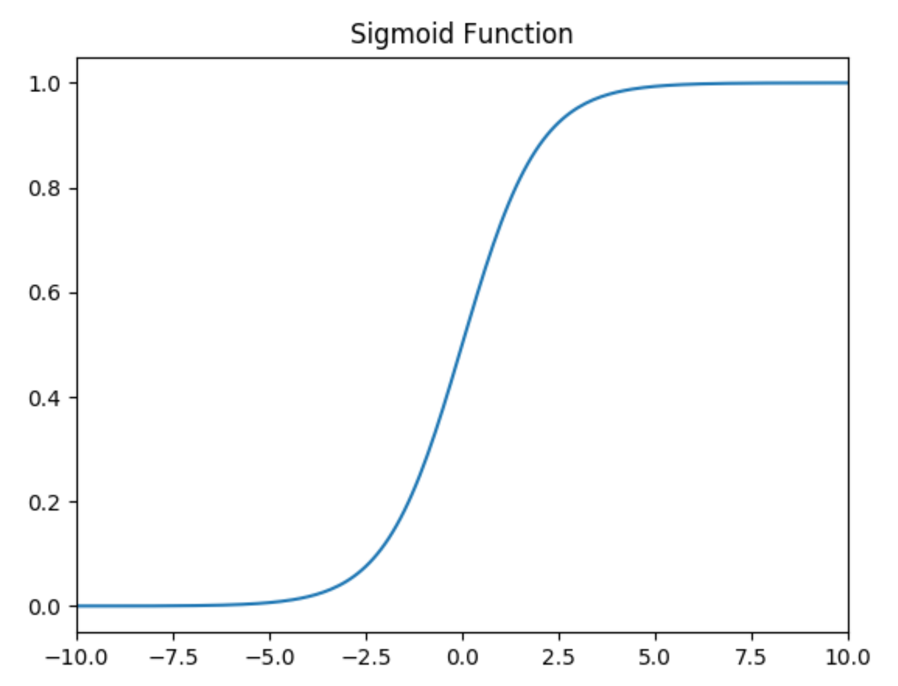
If b = 1

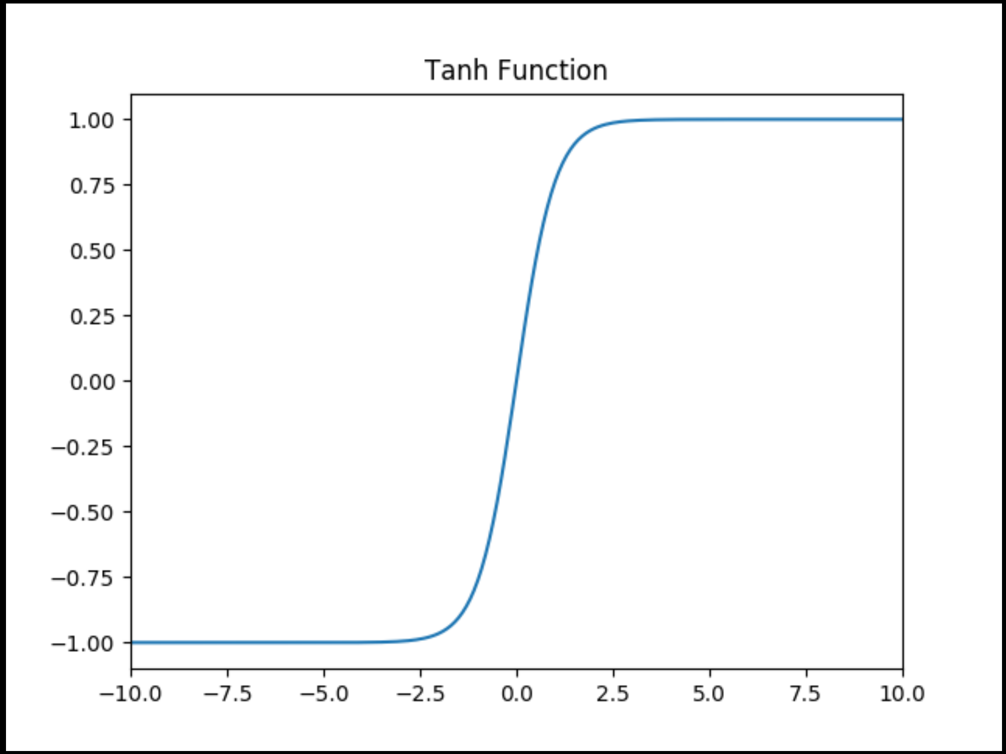
If b > 4

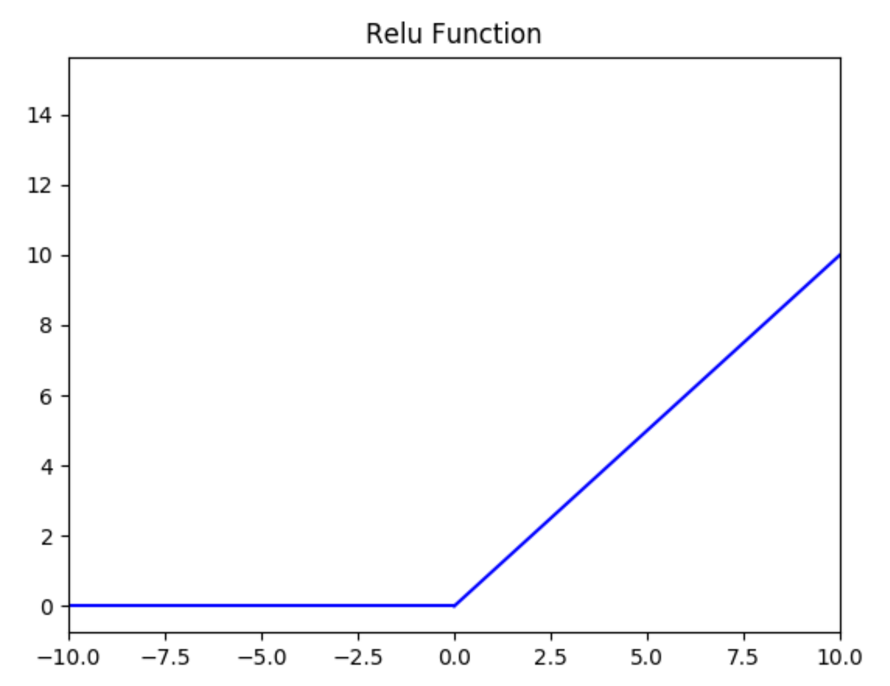


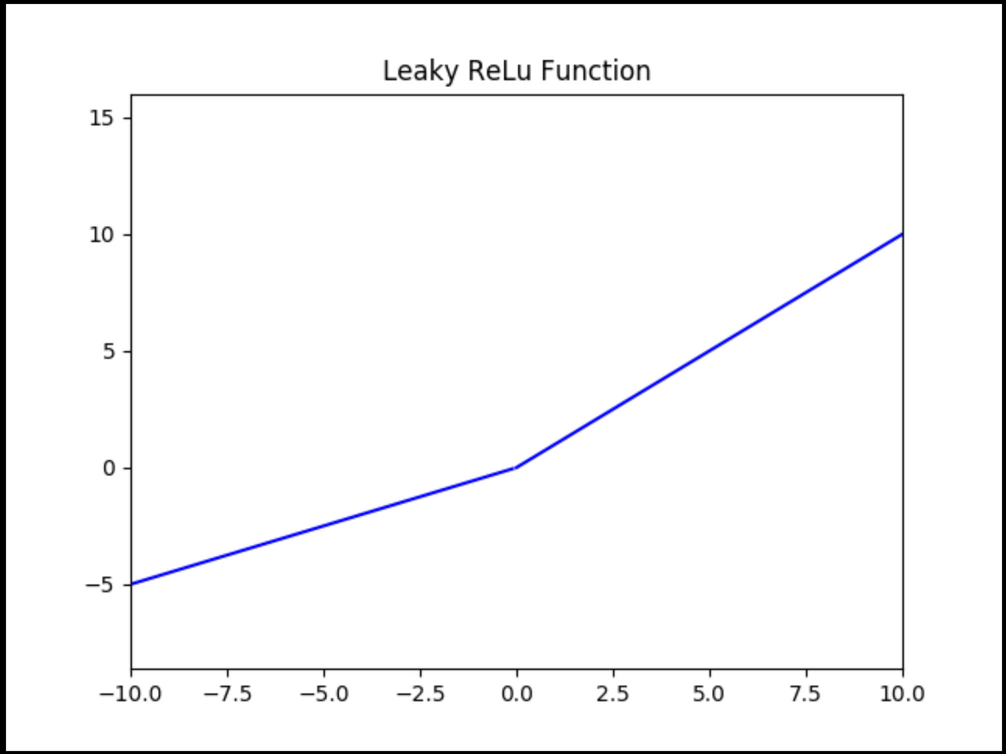
The Perceptron: Activations

The structural building block of deep learning



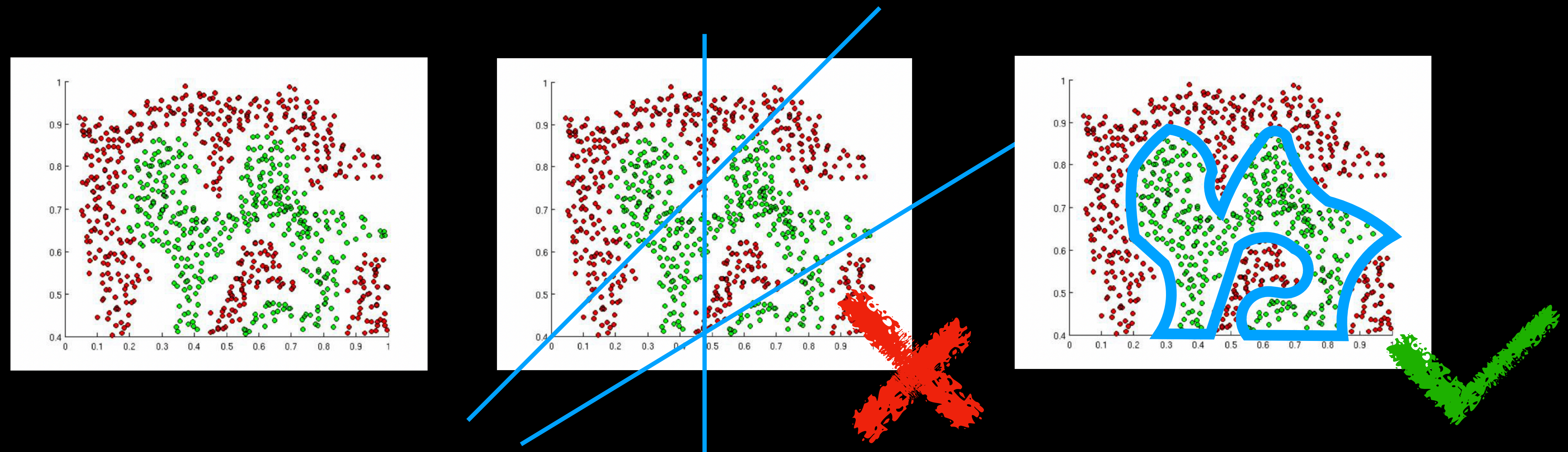




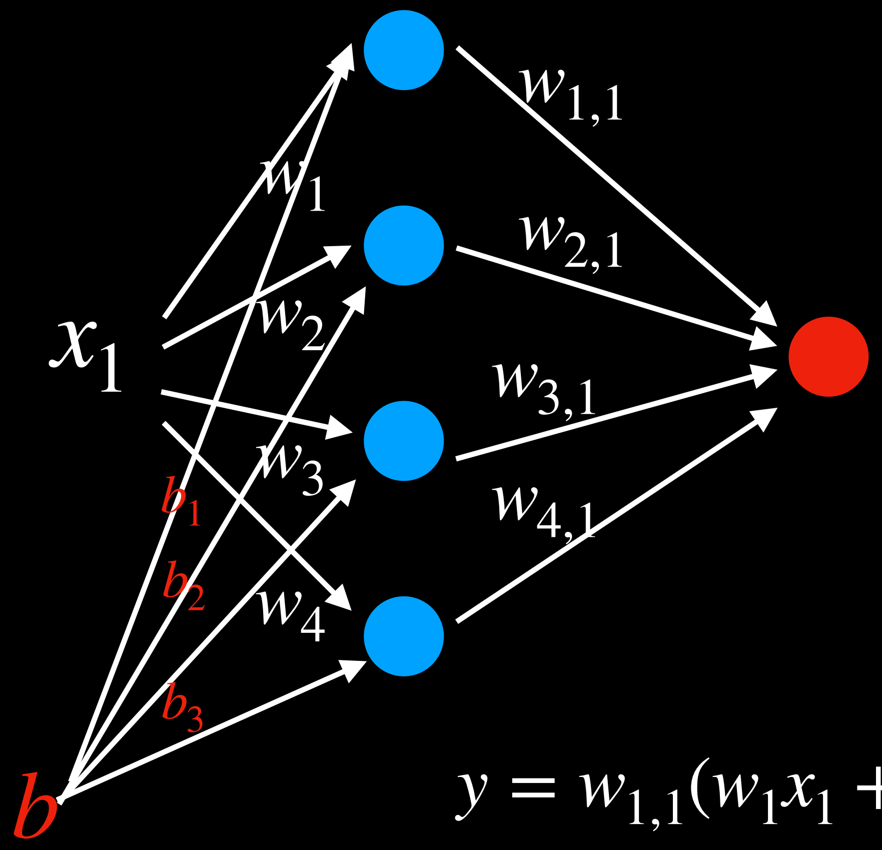


The Perceptron: Activations

The structural building block of deep learning



The purpose of activation functions is to introduce non-linearities into the network



$$w_1x_1 + b_1$$

$$w_2x_1 + b_2$$

$$w_3x_1 + b_3$$

$$w_4x_1 + b_4$$

$$y = w_{1,1}(w_1x_1 + b_1) + w_{2,1}(w_2x_1 + b_2) + w_{3,1}(w_3x_1 + b_3)w_{14,1}(w_4x_1 + b_4)$$

The Perceptron: Activations

The structural building block of deep learning

See Code

Activation Functions: Properties

Nonlinearity

Differentiability

Easiness

Monotonicity

Non-saturation

Identity(near the origin)

Ranging

Less coefficients

Zero-centered or not

Activation Functions: Properties

Non-saturation:
simply understand as some interval where the gradient equals to 0

$$(|\lim_{x\to-\infty}\sigma(x)\to+\infty|)\vee(|\lim_{x\to+\infty}\sigma(x)\to+\infty|)$$

Identity(near the origin): $\sigma(x) \approx x$

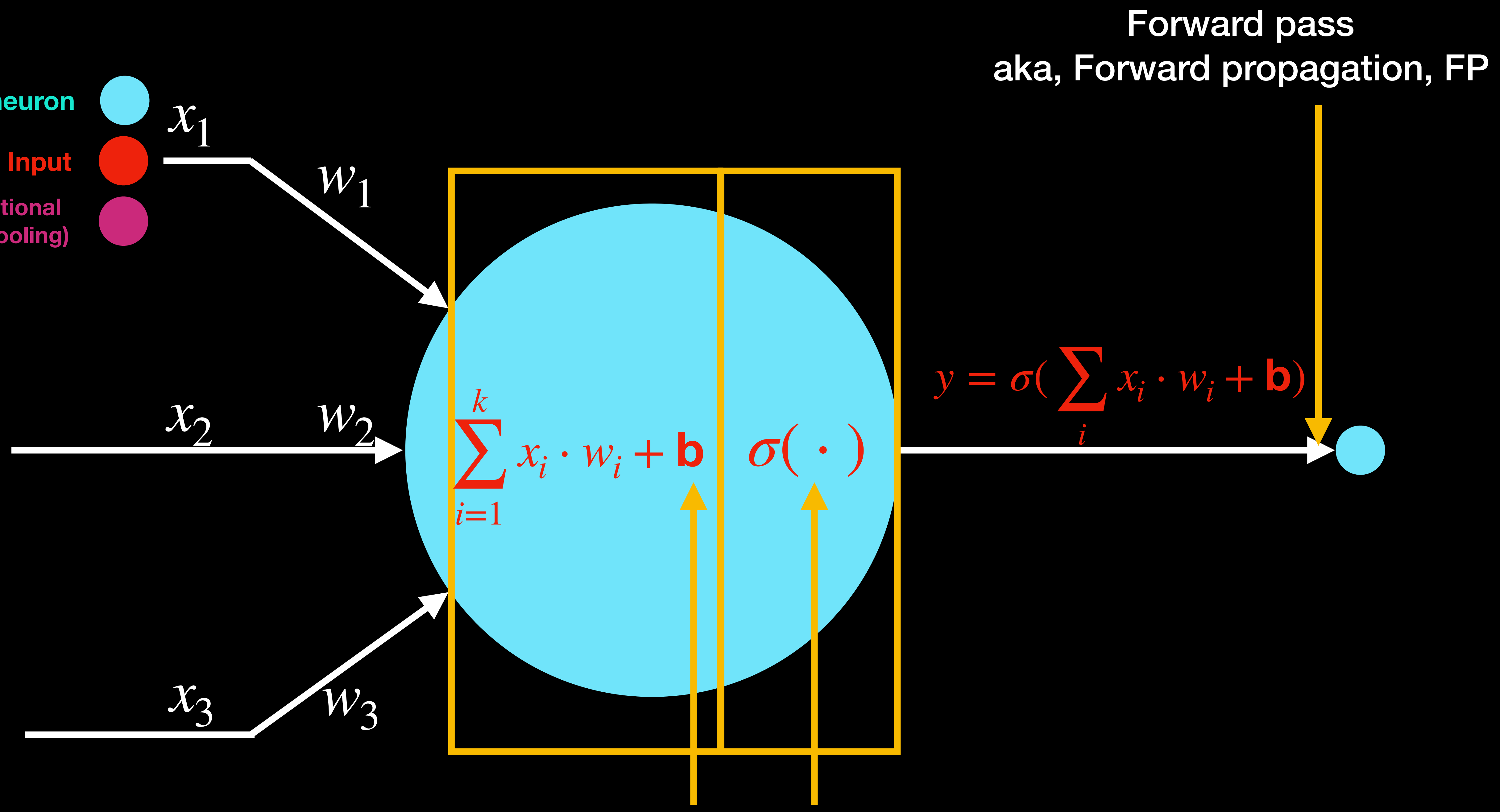
Ranging

Less coefficients

zero-centered: ensure the mean activation value is around zero

The Perceptron

The structural building block of deep learning



● n-1 layer neuron

● Input

● Other functional structure(pooling)

Forward pass
aka, Forward propagation, FP

x_1 w_1

x_2 w_2

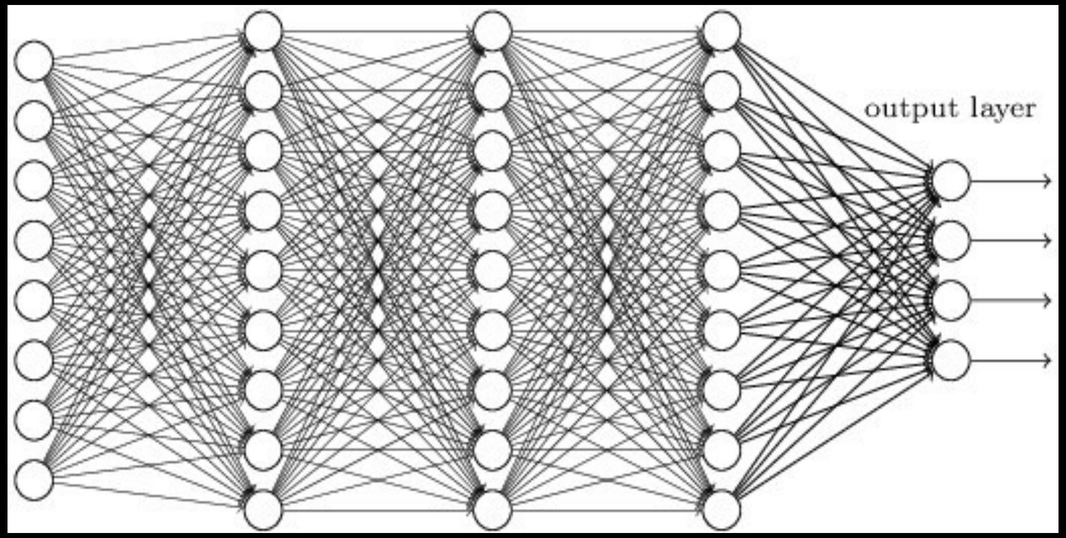
x_3 w_3

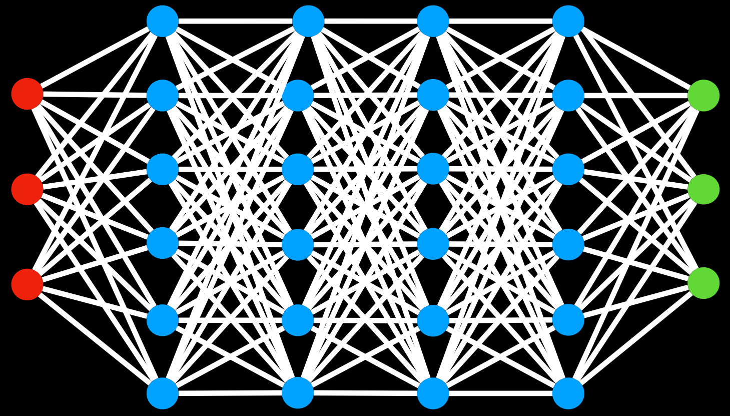
$$\sum_{i=1}^{k} x_i \cdot w_i + \mathbf{b} \qquad \sigma(\,\cdot\,)$$

$$y = \sigma\left(\sum_i x_i \cdot w_i + \mathbf{b}\right)$$

Bias **Activation Function**

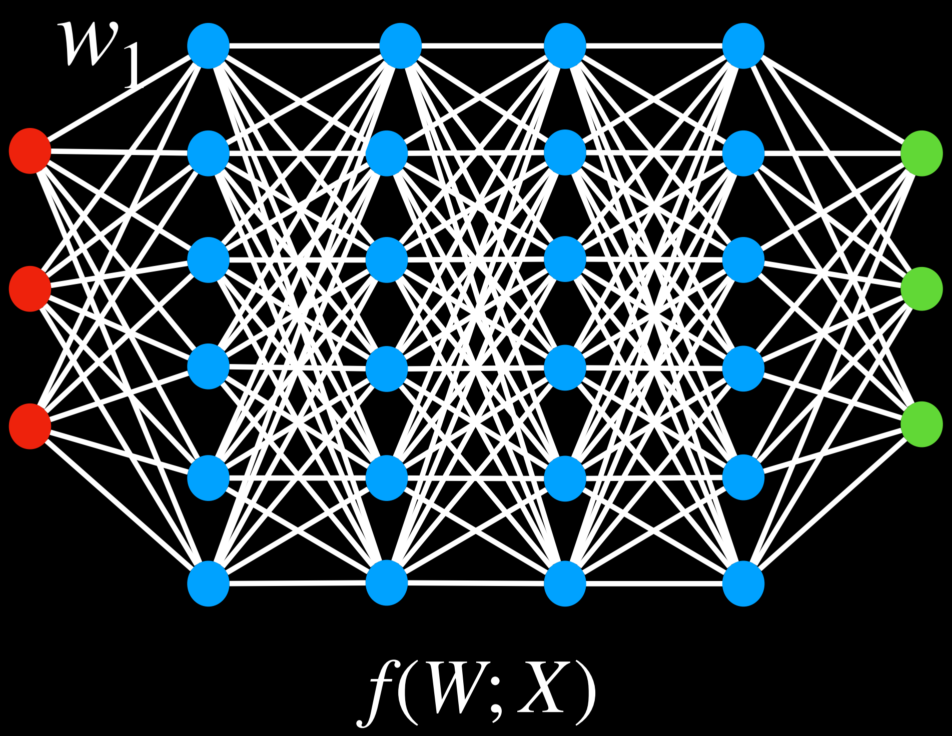
Think the network as a function





- We will assume a feed-forward network
  - No loops: Neuron outputs do not feed back to their inputs directly or indirectly
- Part of the design of a network: The architecture
  - How many layers/neurons, which neuron connects to which and how, etc.
- For now, assume the architecture of the network is capable of representing the needed function

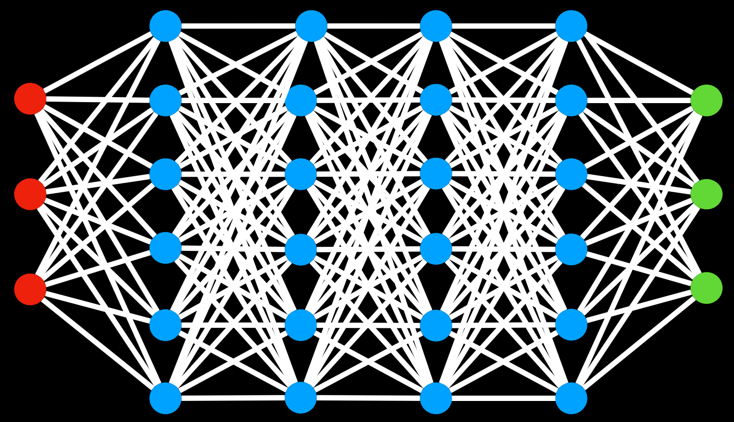
What we learn: The parameters of the network



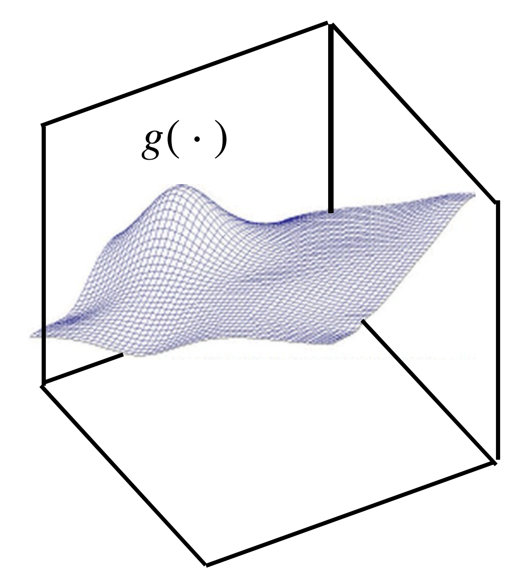
The network is a function f() with parameters W which must be set to the appropriate values to get the desired behavior from the net

- Given: the architecture of the network
- The parameters of the network: The weights and biases
- Learning the network : Determining the values of these parameters such that the network computes the desired function

How to learn a network



$$f(\cdot) : \mathscr{R}^m \rightarrow \mathscr{R}^c$$



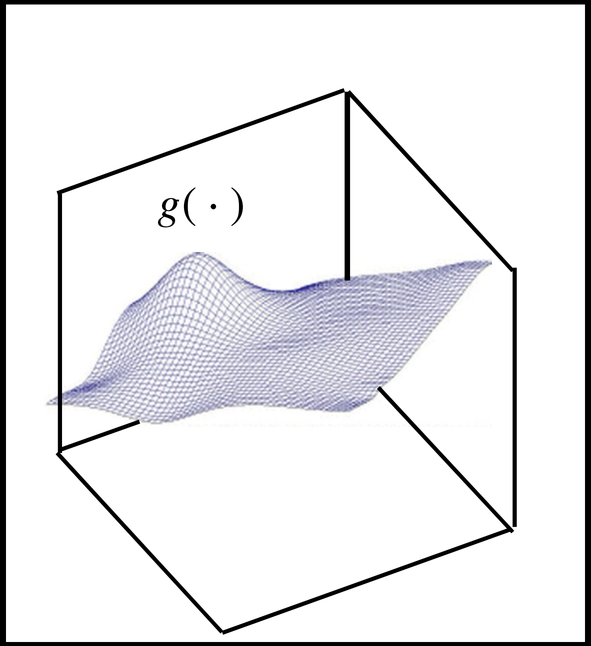
Suppose $g(\cdot)$ is given

When $f(W; X)$ has the capacity to exactly represent $g(\cdot)$

$$\hat{W} = argmin_w \int_X div(f(W; X), g(\cdot))dX$$

div() is a divergence function that goes to 0 when $f(W; X) = g(X)$

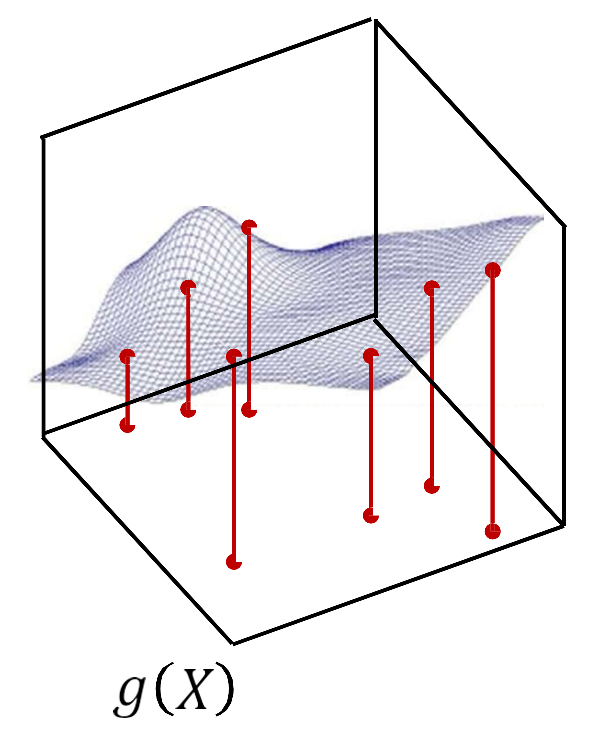
However…

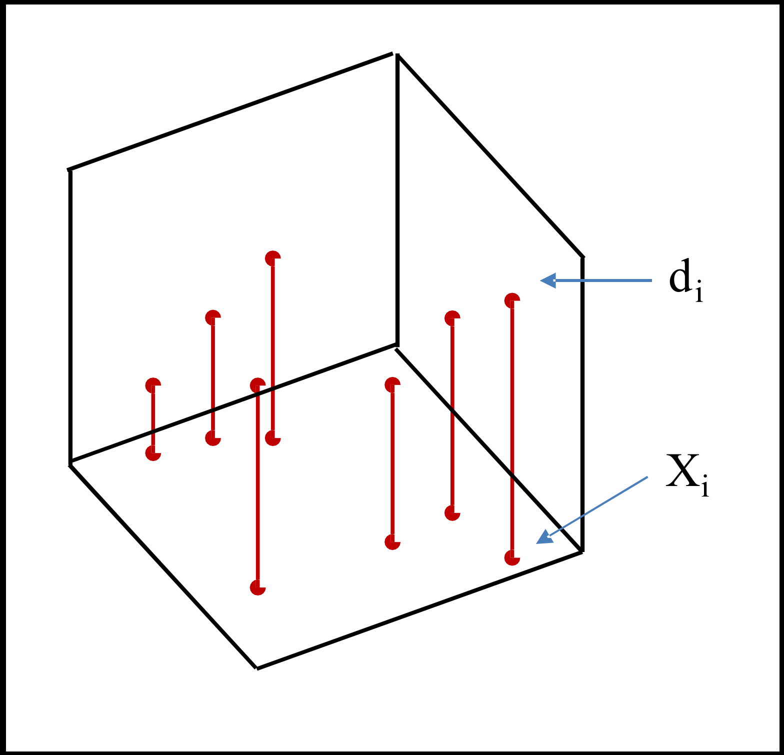


- **Function $g(\cdot)$ must be fully specified**
  - **Known everywhere, i.e. for every input**

- **In practice we will not have such specification**

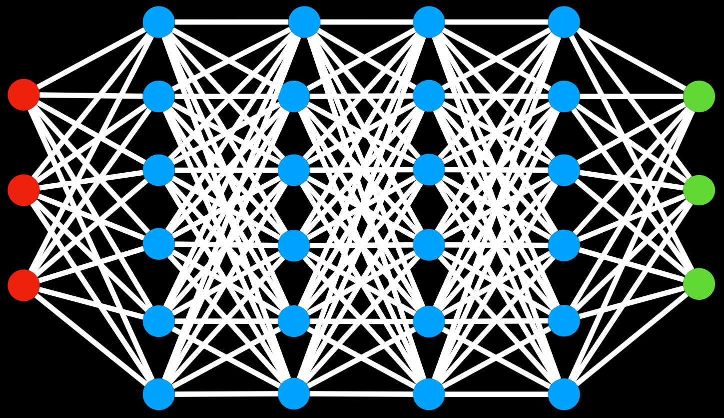
Sampling



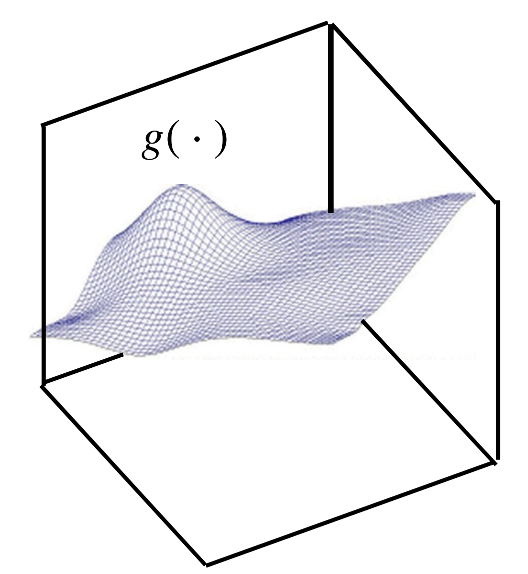


- **Sample** $g(\cdot)$
  - **Basically, get input-output pairs for a number of samples of input**
    - **Many samples** (X_i, d_i) **where** $d_i = g(X_i) + \epsilon$
  - **Good sampling: the samples of** X **will be drawn from** $P(X)$
- **Very easy to do in most problems: just gather training data**
  - **E.g. set of images and their class labels**
  - **E.g. speech recordings and their transcription**

Minimizing expected error



$$f(\cdot) : \mathscr{R}^m \rightarrow \mathscr{R}^c$$

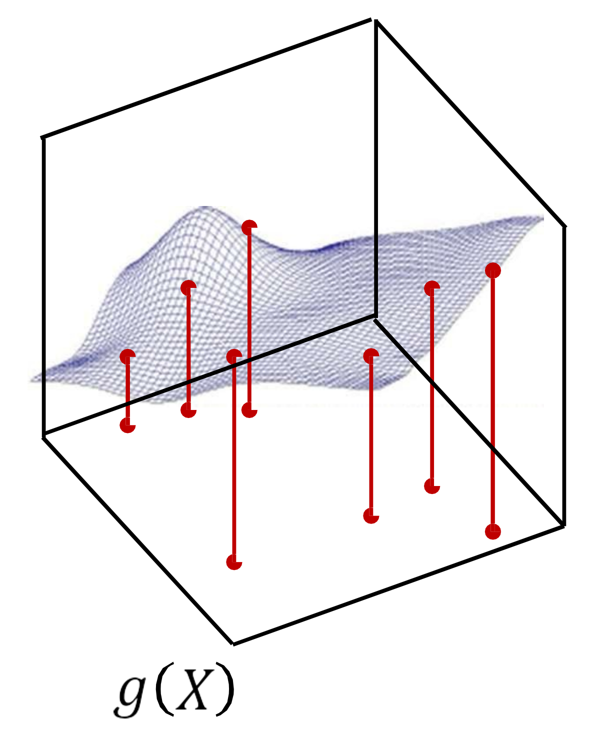


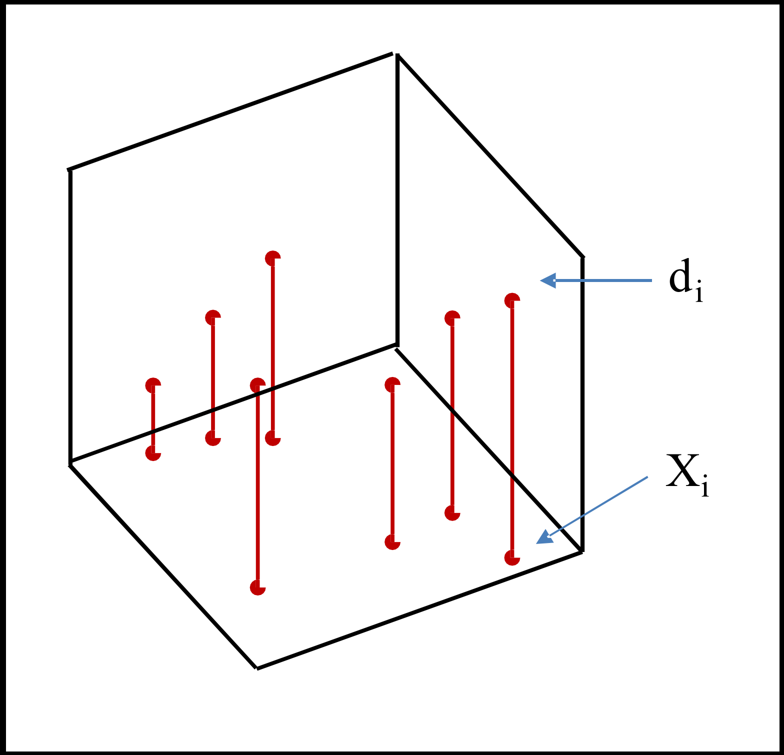
More generally, assuming X is a random variable

$$W = argmin_w \int_X div(f(W;X), g(\cdot))P(X)dX$$

$$= argmin_W E[div(f(W;X), g(\cdot))]$$

The Empirical Risk





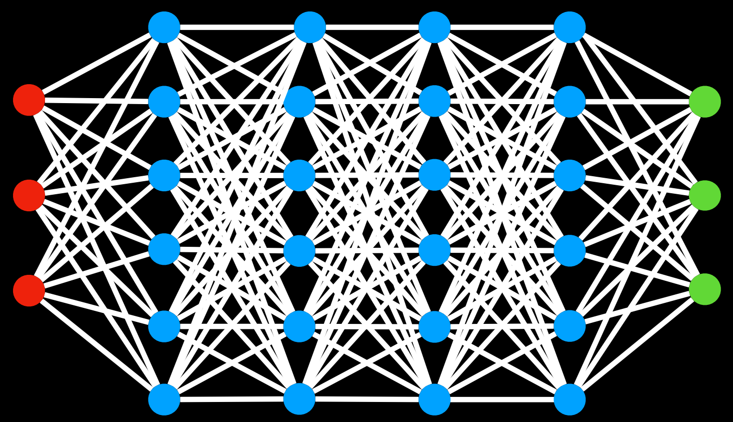
The expected error (or risk) is the average error over the entire input space

$$E[div(f(W;X),g(X))] = \int_X div(f(W;X),g(X))P(X)dX$$

The empirical estimate of the expected error is the average error over the samples

$$E[div(f(W;X),g(X))] \approx \frac{1}{N}\sum_{i=1}^{N} div(f(W_i;X),d_i)$$

The Empirical Risk Minimization problem



$f(W; X)$

- Given a training set of input-output pairs $(X_1, d_1), (x_2, d_2), \cdots, (x_N, d_N)$
  - Error on the ith instance: $div(f(W; x_i), d_i)$
  - Empirical average error (Empirical Risk) on all training data:

$$Loss(W) = \frac{1}{N} \sum_{i=1}^{N} div(f(W_i; X), d_i)$$

- Estimate the parameters to minimize the empirical estimate of expected error

$$\hat{W} = argmin_W Loss(W)$$

I.e. minimize the empirical risk over the drawn samples

ERM problem Statement

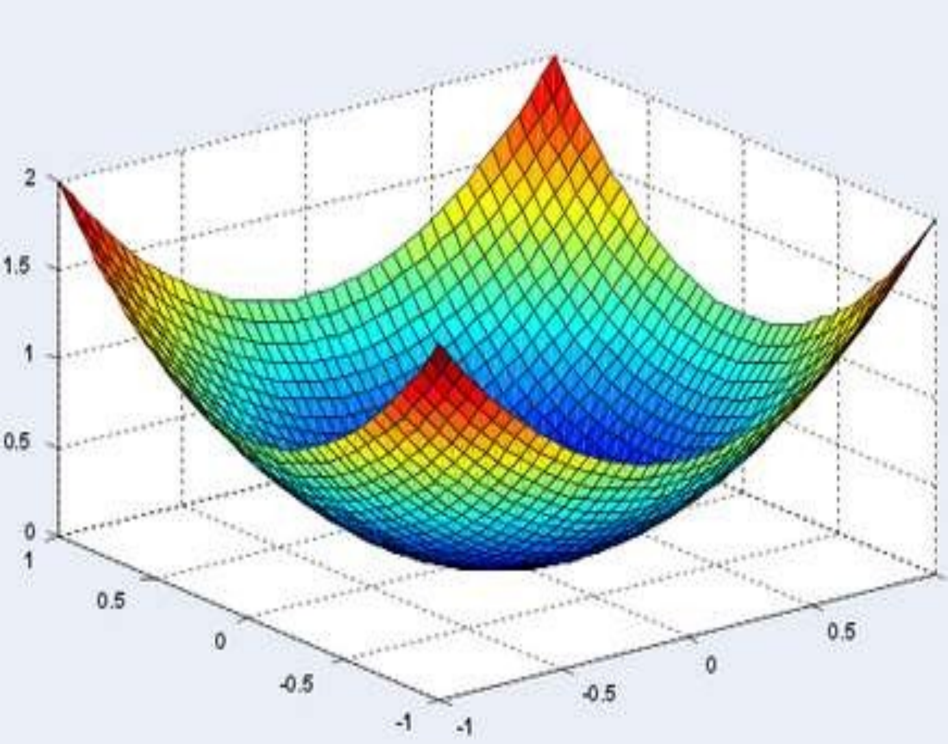
- **Given a training set of input-output pairs** $(X_1, d_1), (X_2, d_2), \cdots, (X_N, d_N)$

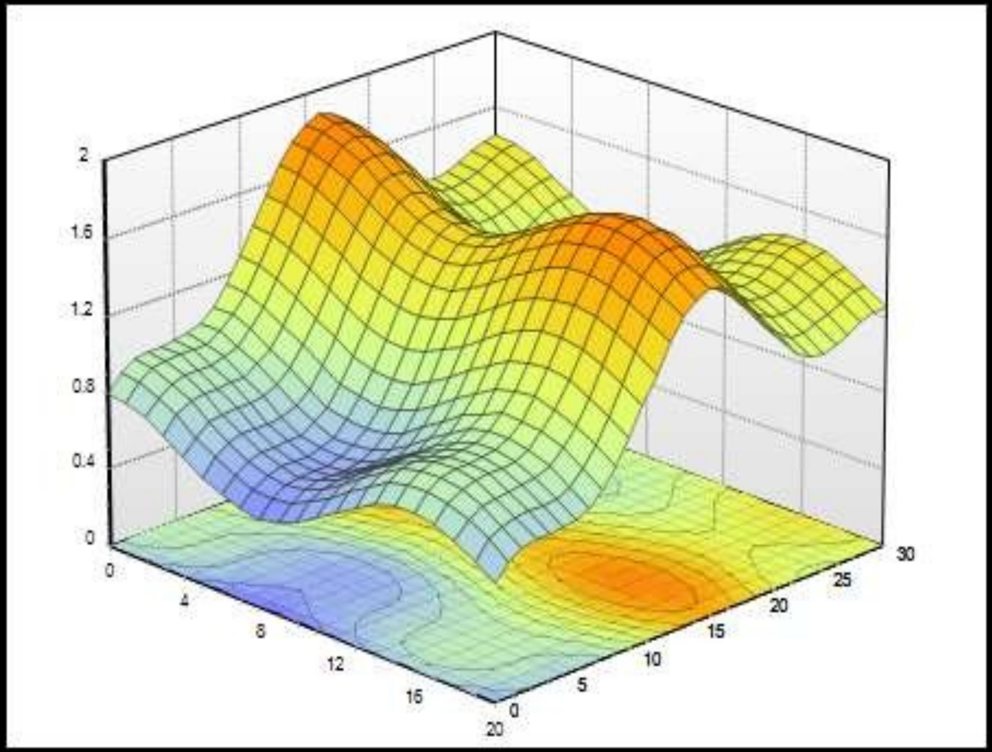
- **Minimize the following function (w.r.t W)**

$$Loss(W) = \frac{1}{N} \sum_{i=1}^{N} div(f(W_i; X), d_i) + \gamma(w)$$

- **This is problem of function minimization**

How to solve ERM problem?





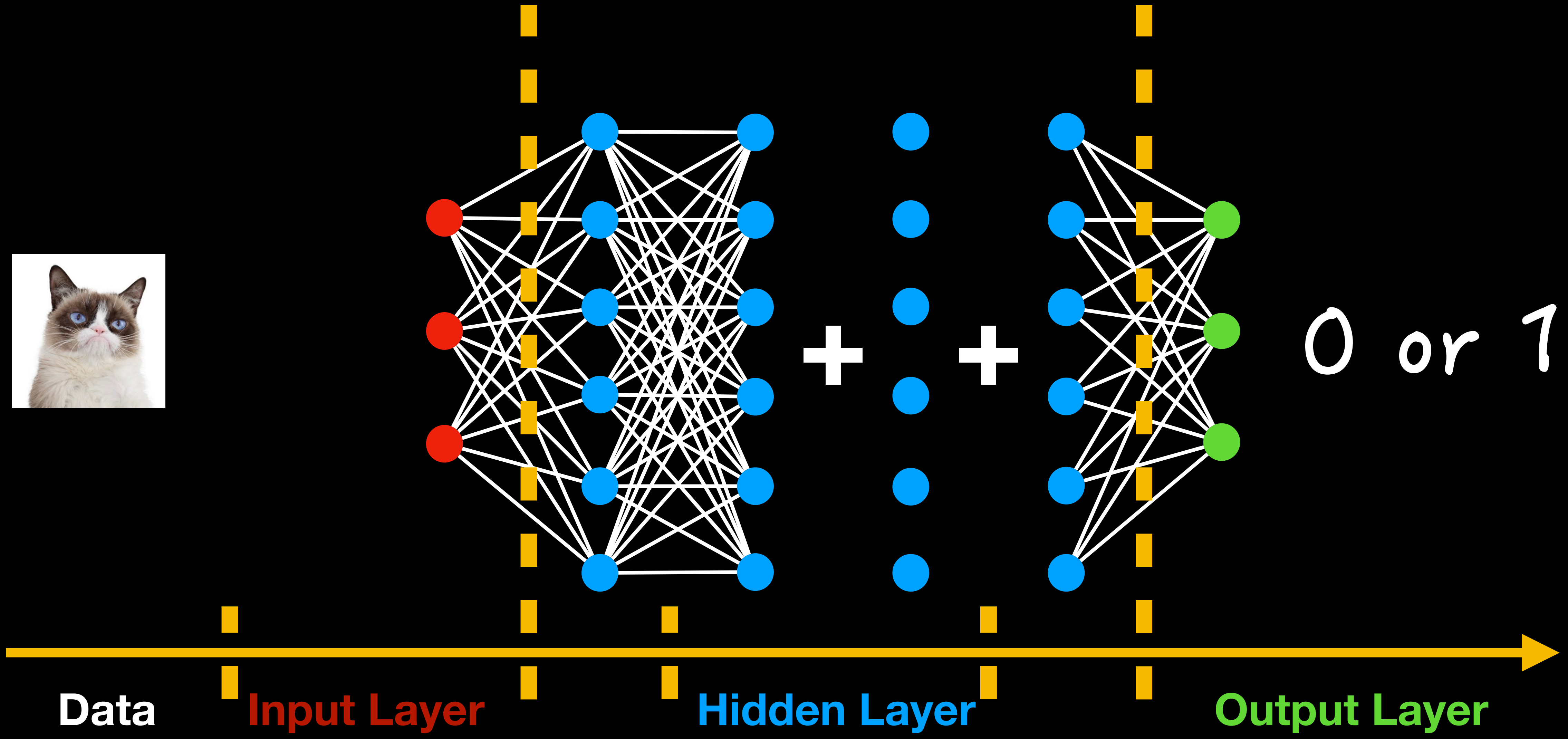
Gradient Descent Algorithm (GD)

- In order to minimize any function f(w) w.r.t w

```
Do
    For every component $i$
        $w_{i,t} = w_{i,t-1} - \eta^t \frac{df}{dw_{i,t-1}}$
        $t \rightarrow t+1$
While $|f(w_t) - f(w_{t-1})| > \epsilon$
```

- See later lecture

What is f(): Typical network



0 or 1

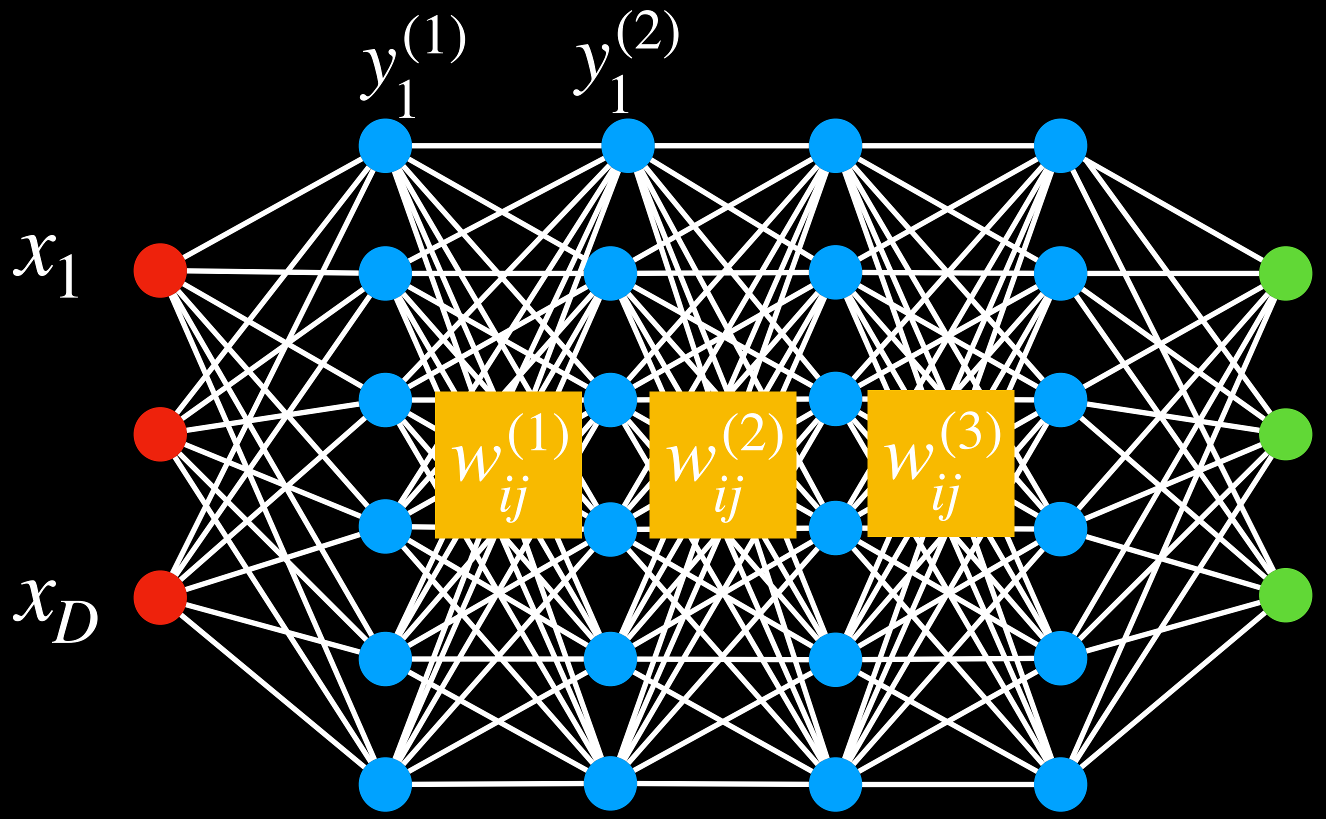
Data

Input Layer

Hidden Layer

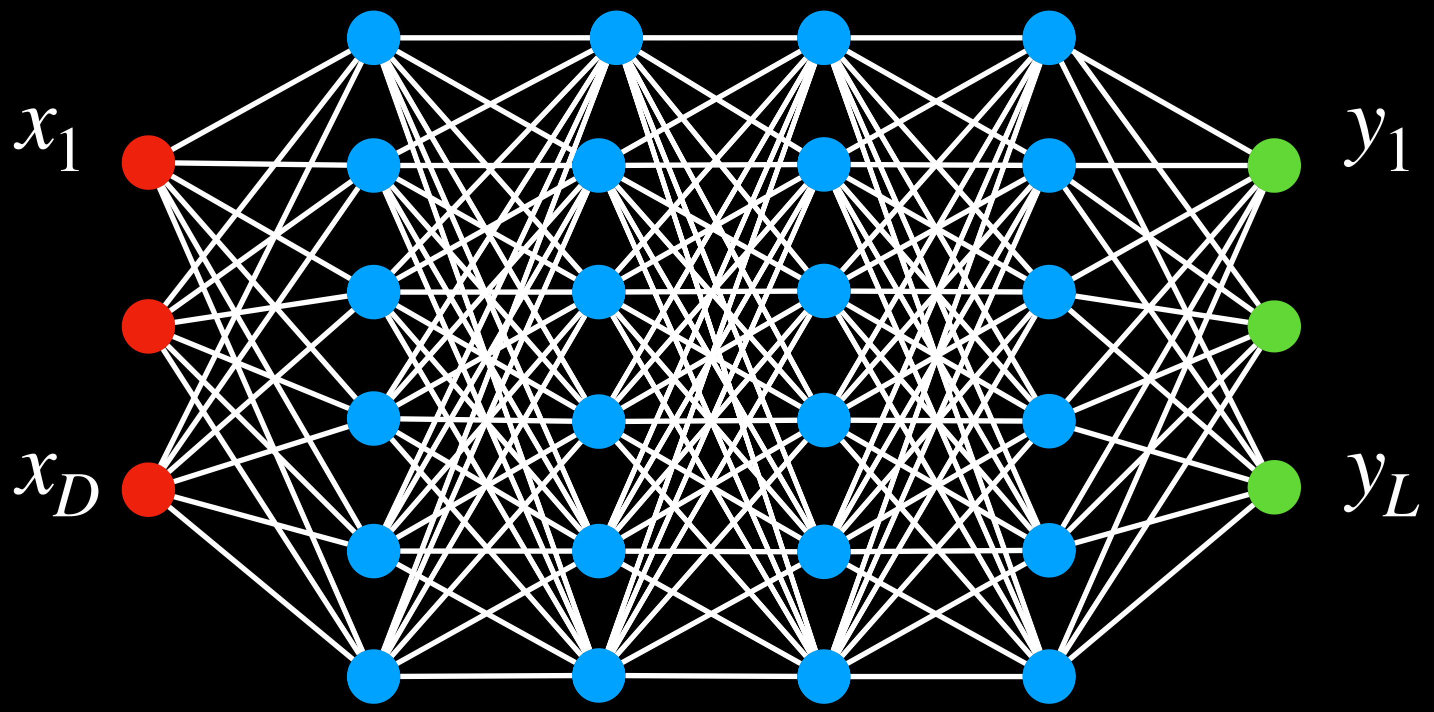
Output Layer

Notation



- The input layer is the 0^{th} layer
- We will represent the output of the i^{th} perceptron of the k^{th} layer as y_i^k
- Input to network: $y_i^{(0)} = x_i$
- Output to network: $y_i = y_i^N$
- We will represent the weight of the connection between the i^{th} unit of the $(k-1)^{th}$ layer and the j^{th} unit of the k^{th} layer as $w_{ij}^{(k)}$
  - The bias to the j^{th} unit of the k^{th} layer is $b_j^{(k)}$

Notation

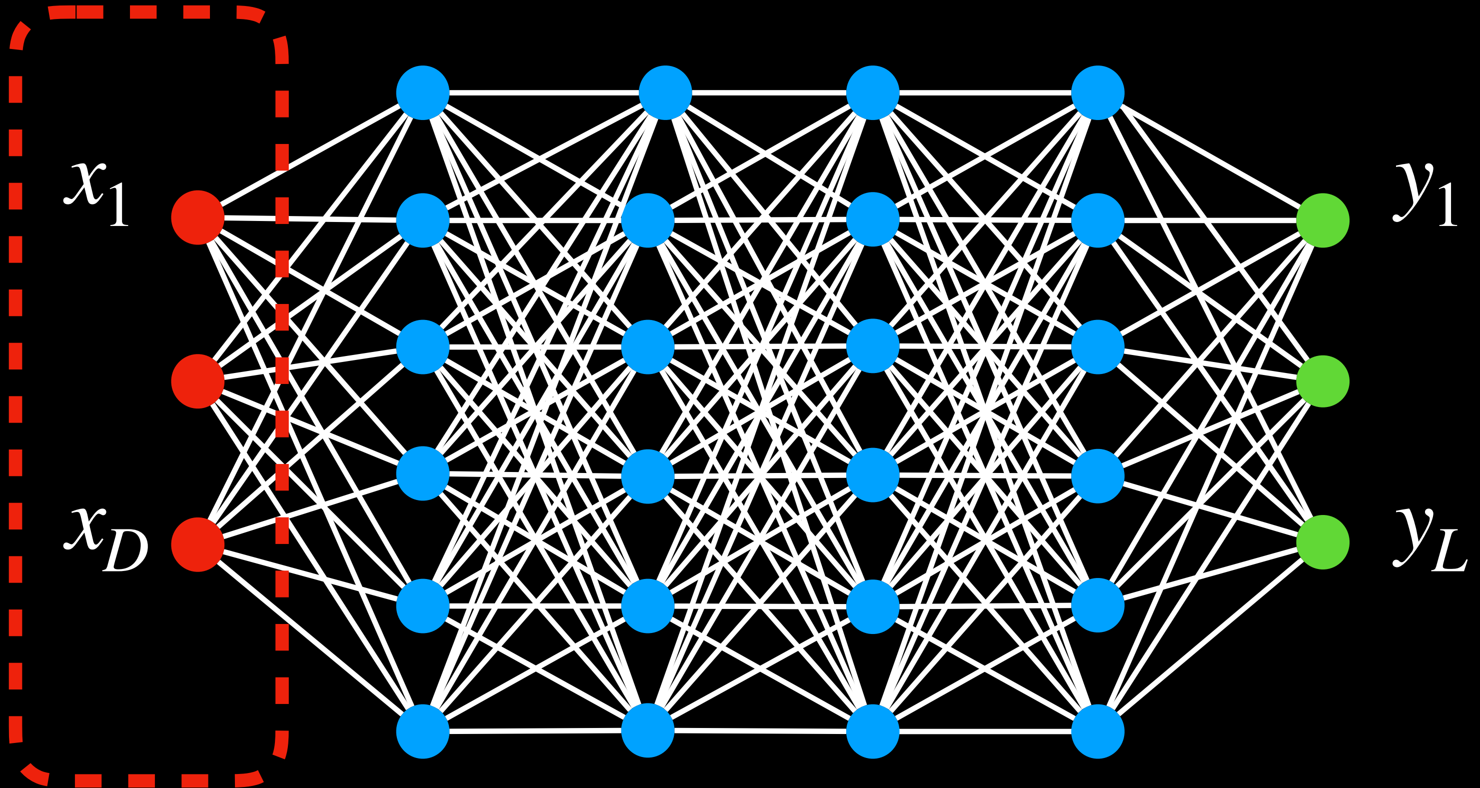


- Given a training set of input-output pairs $(X_1, d_1), (x_2, d_2), \cdots, (x_N, d_N)$

- $X_n = [x_{n1}, x_{n2}, \cdots, x_{nD}]$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \cdots, d_{nL}]$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \cdots, y_{nL}]$ is the nth vector of *actual* outputs of the network

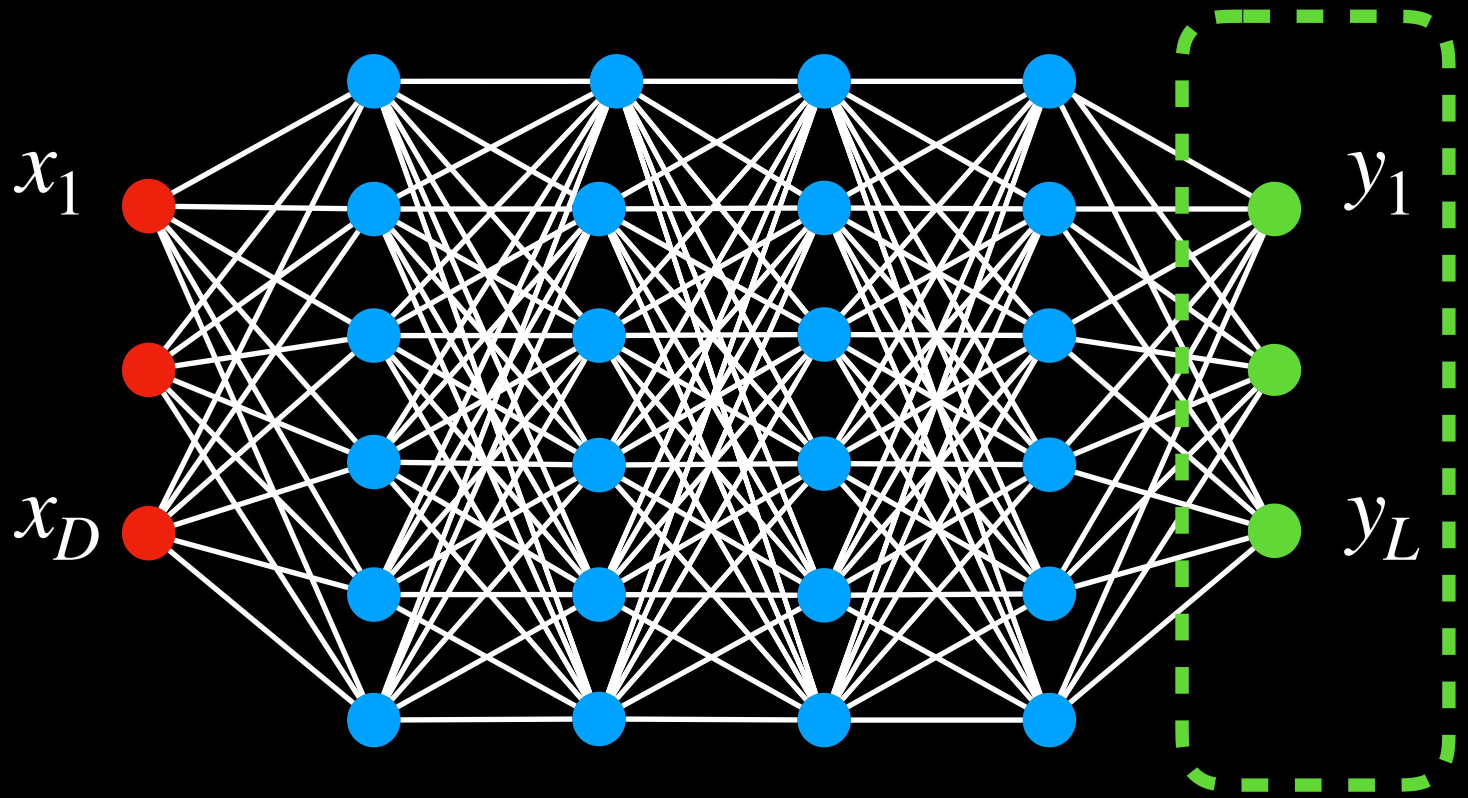
- We will sometimes drop the first subscript when referring to a *specific* instance

Representing the input



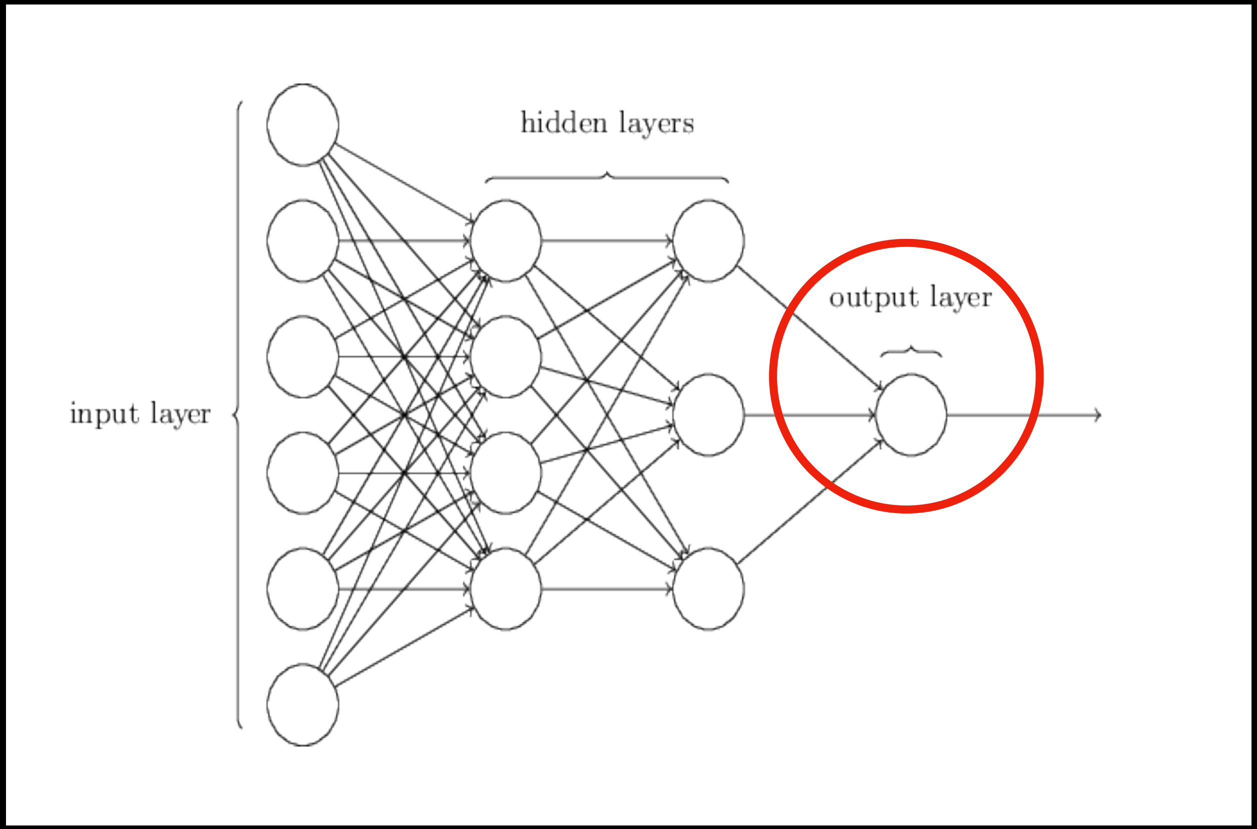
- Vectors of numbers
  - – (or may even be just a scalar, if input layer is of size 1)
  - – E.g. vector of pixel values
    - We will see how this happens later in the course (CNN)
  - – E.g. vector of speech features
  - – E.g. real-valued vector representing text
  - – Other real valued vectors

Representing the output



- If the desired *output* is real-valued, no special tricks are necessary
  - – Scalar Output : single output neuron
    - d = scalar (real value)
  - – Vector Output : as many output neurons as the dimension of the desired output
    - d = [d_1 d_2 .. d_L] (vector of real values)

Representing the output



- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 = Yes it's acat
  - 0 = No it's not a cat.

Multi-class output: one-hot representations

- Consider a network that must distinguish if an input is a cat, a dog, a camel, a hat, or a flower
- We can represent this set as the following vector:

$$[\text{cat dog camel hat flower}]^T$$

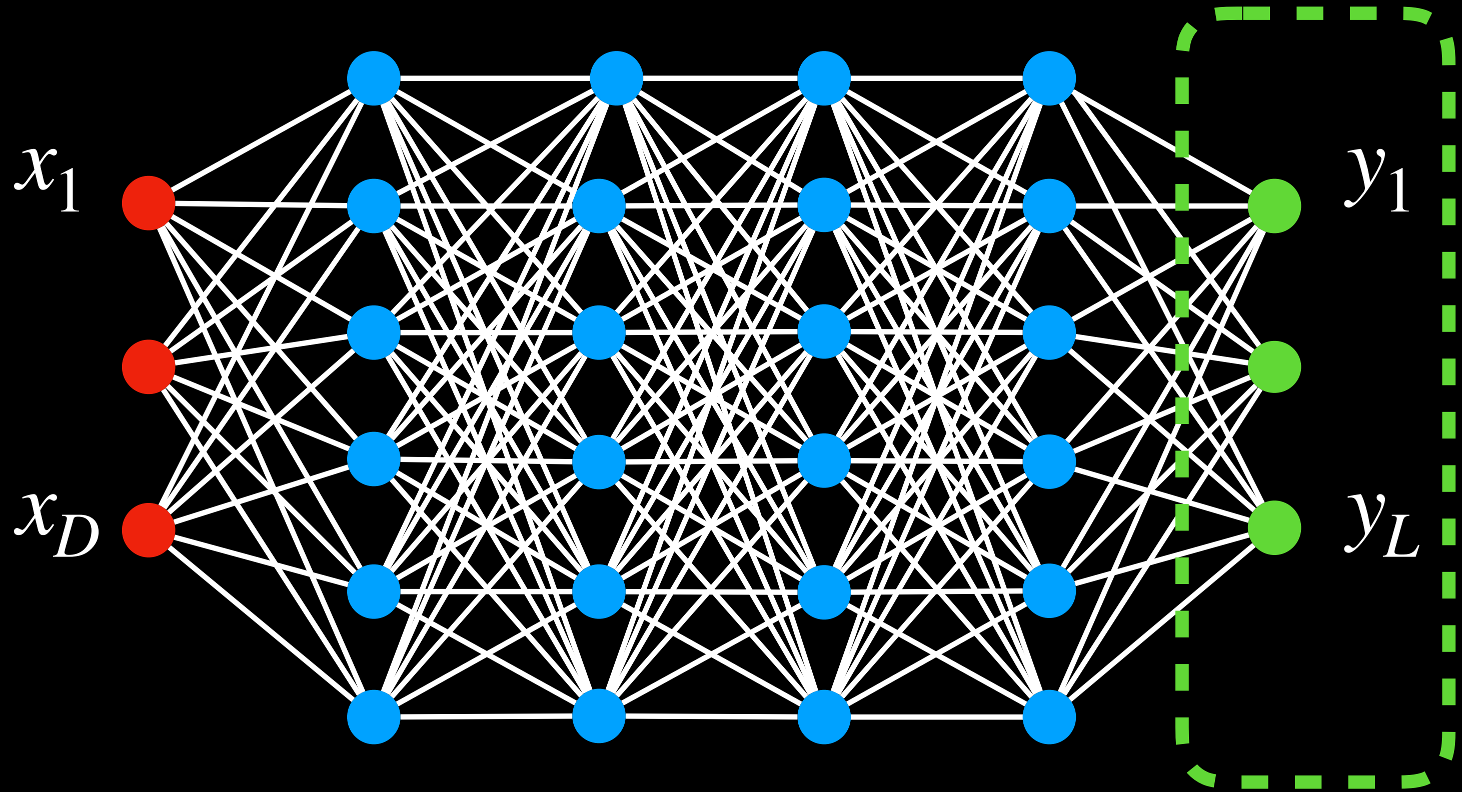
- For inputs of each of the five classes the desired output is:

cat: $[1\ 0\ 0\ 0\ 0]^T$
dog: $[0\ 1\ 0\ 0\ 0]^T$
camel: $[0\ 0\ 1\ 0\ 0]^T$
hat: $[0\ 0\ 0\ 1\ 0]^T$
flower: $[0\ 0\ 0\ 0\ 1]^T$

- For an input of any class, we will have a five-dimensional vector output with four zeros and a single 1 at the position of that class

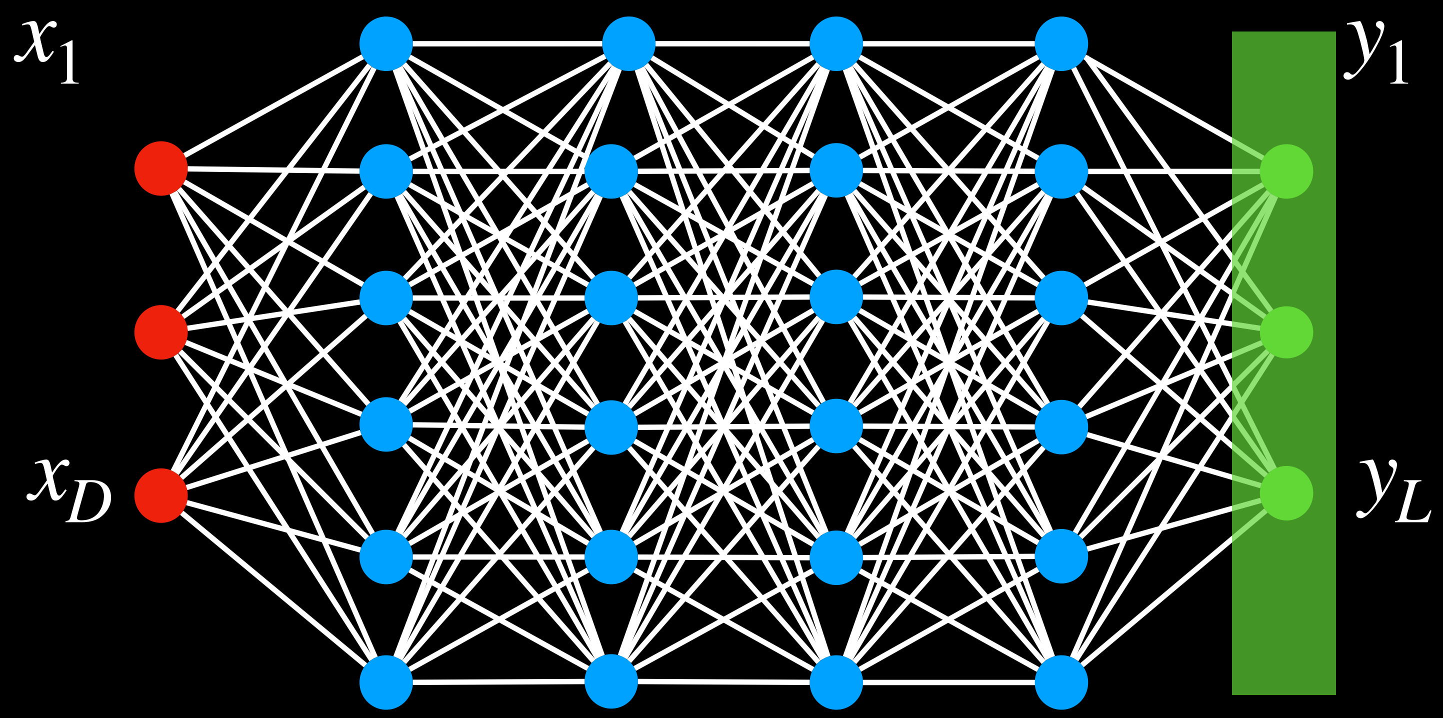
- This is a *one hot vector*

Multi-class networks



- For a multi-class classifier with N classes, the one-hot representation will have N binary outputs
  - – An N-dimensional binary vector
- The neural network's output too must ideally be binary (N-1 zeros and a single 1 in the right place)
- More realistically, it will be a probability vector
  - – N probability values that sum to 1

Multi-class classification: Output

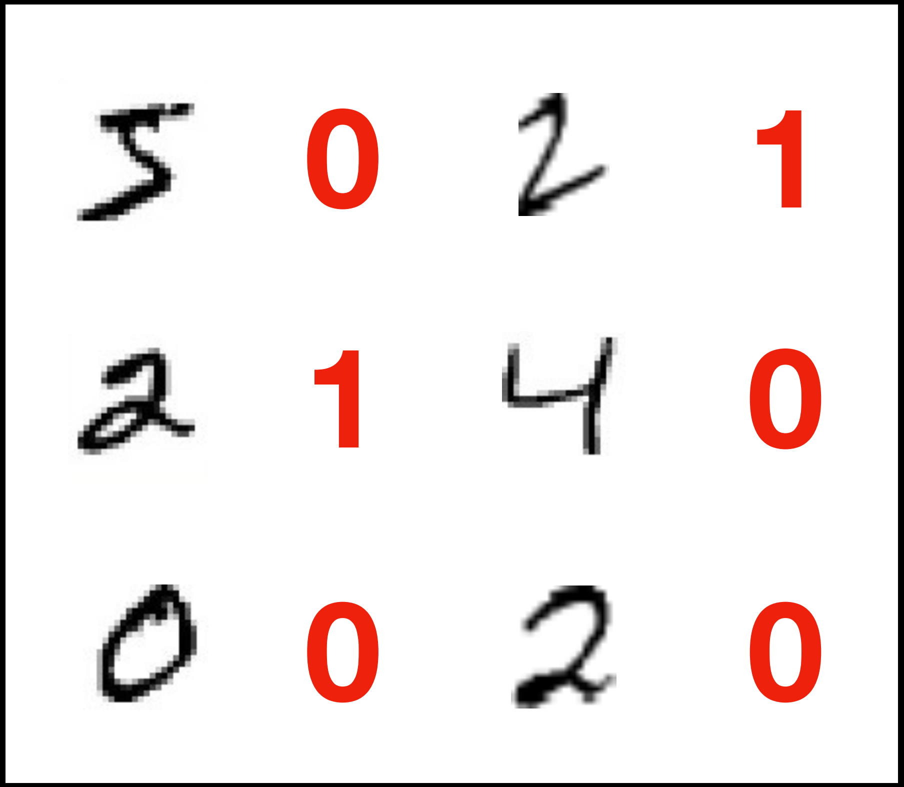


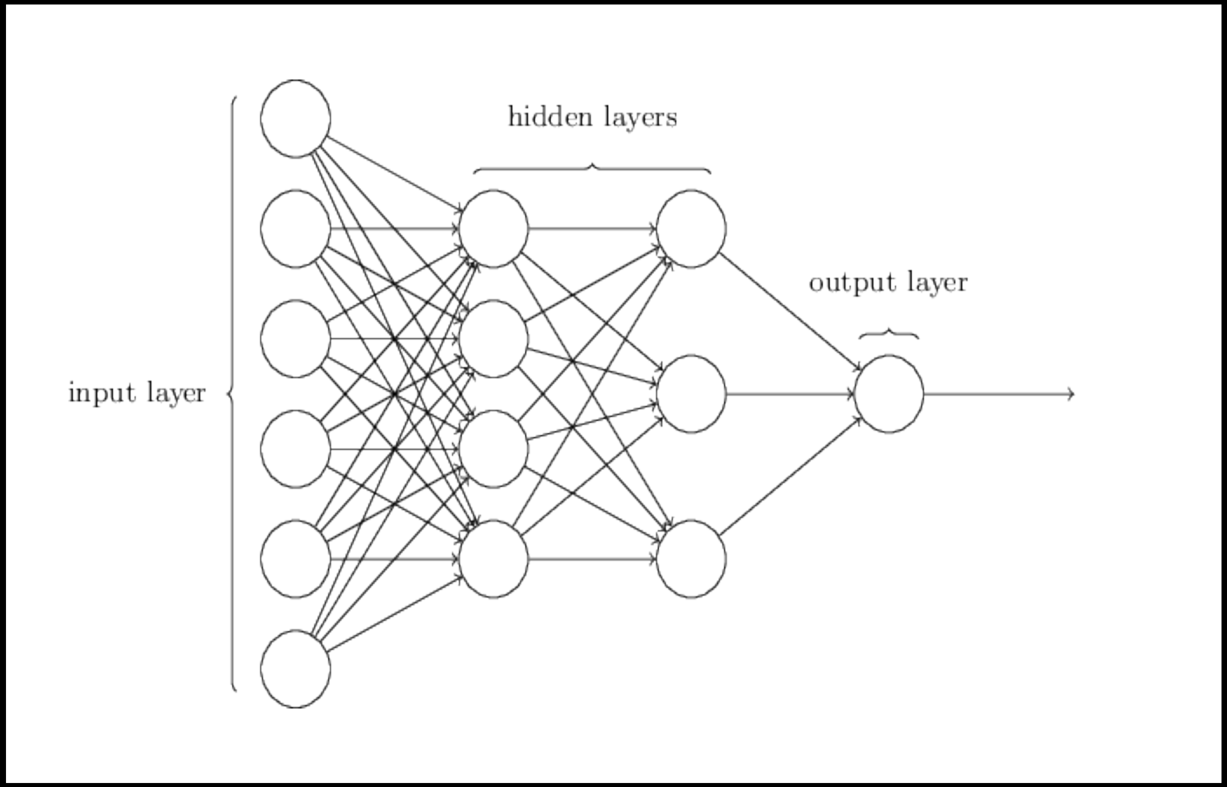
- Softmax *vector* activation is often used at the output of multi-class classifier nets

$$z_i = \sum_j w_{ji}^{(n)} y_j^{(n-1)} \qquad y_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- This can be viewed as the probability $y_i = P(class = i \mid X)$

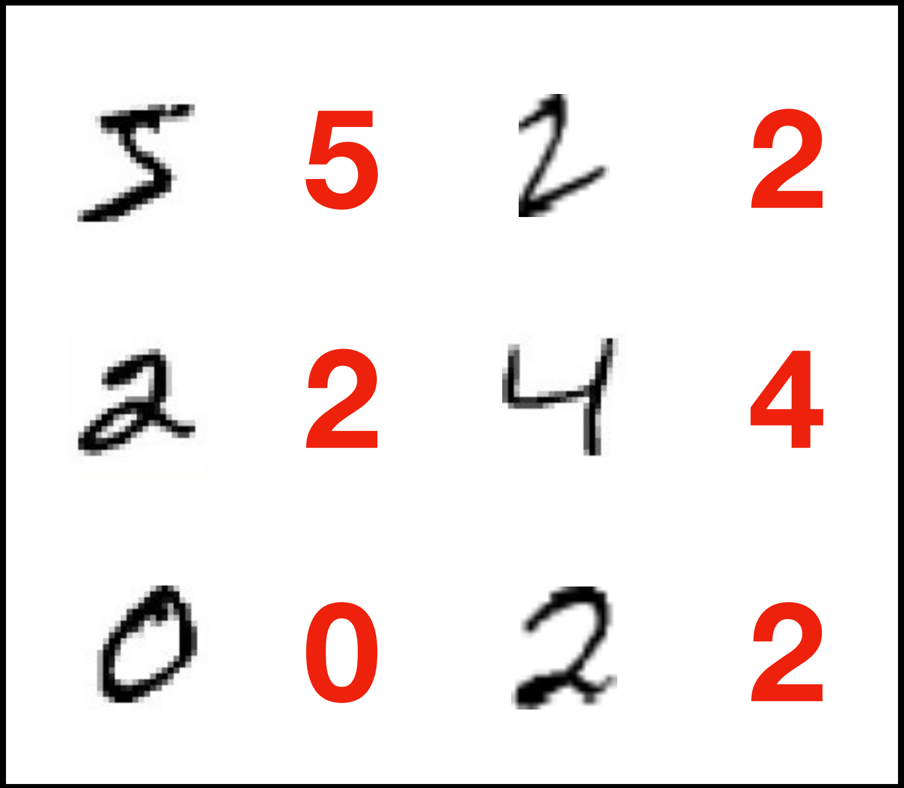
Typical Problem: binary classification

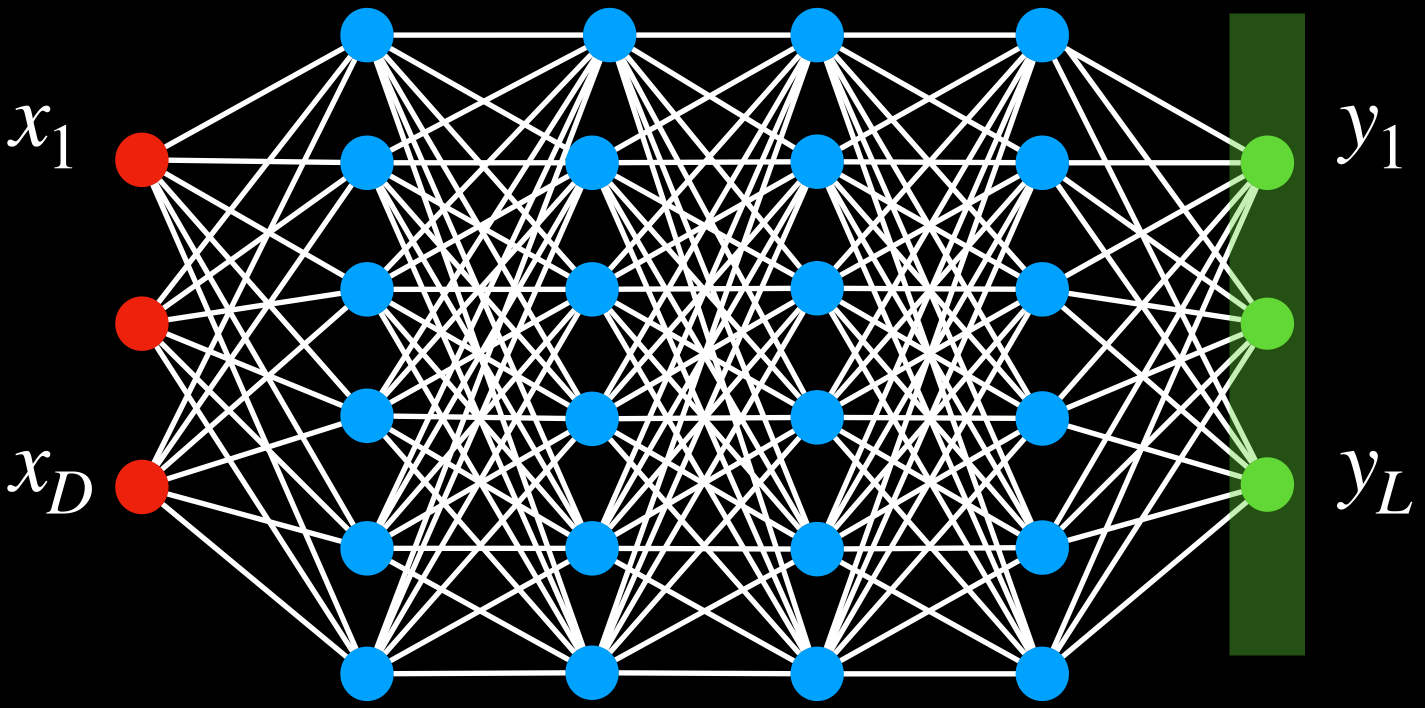




- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job

Typical Problem:
multi-class classification





ERM problem Statement

- **Given a training set of input-output pairs** $(X_1, d_1), (X_2, d_2), \cdots, (X_N, d_N)$ ✓
- **Minimize the following function (w.r.t W)**

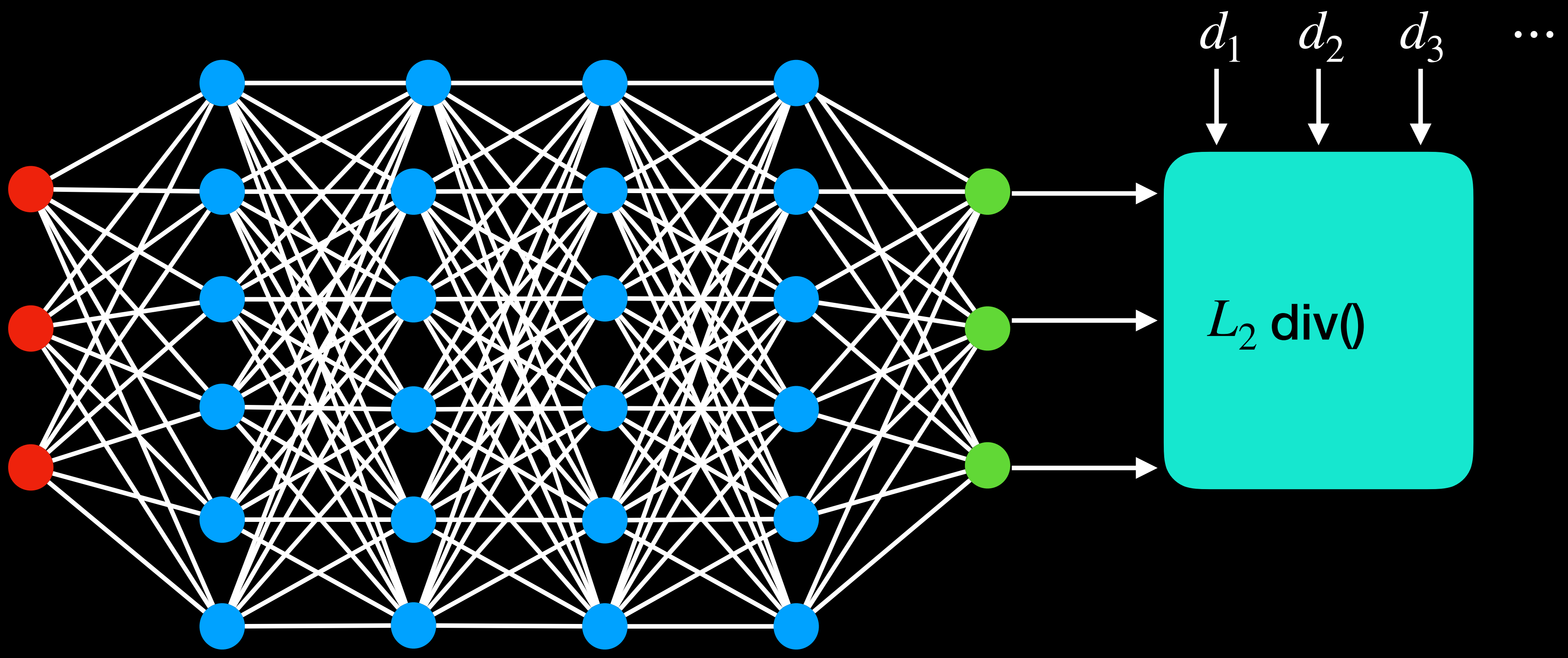
$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N} div(f(W_i; X), d_i) + \gamma(w)$$ ✓

- **This is problem of function minimization**

What is the divergence function: div() ?

Note: For Loss(W) to be differentiable w.r.t W,
div() must be differentiable.

Examples of divergence functions



- For real-valued output vectors, the (scaled) L_2 divergence is popular

$$Div(Y,d) = \frac{1}{2}||Y-d||^2 = \frac{1}{2}\sum_i (y_i - d_i)^2$$

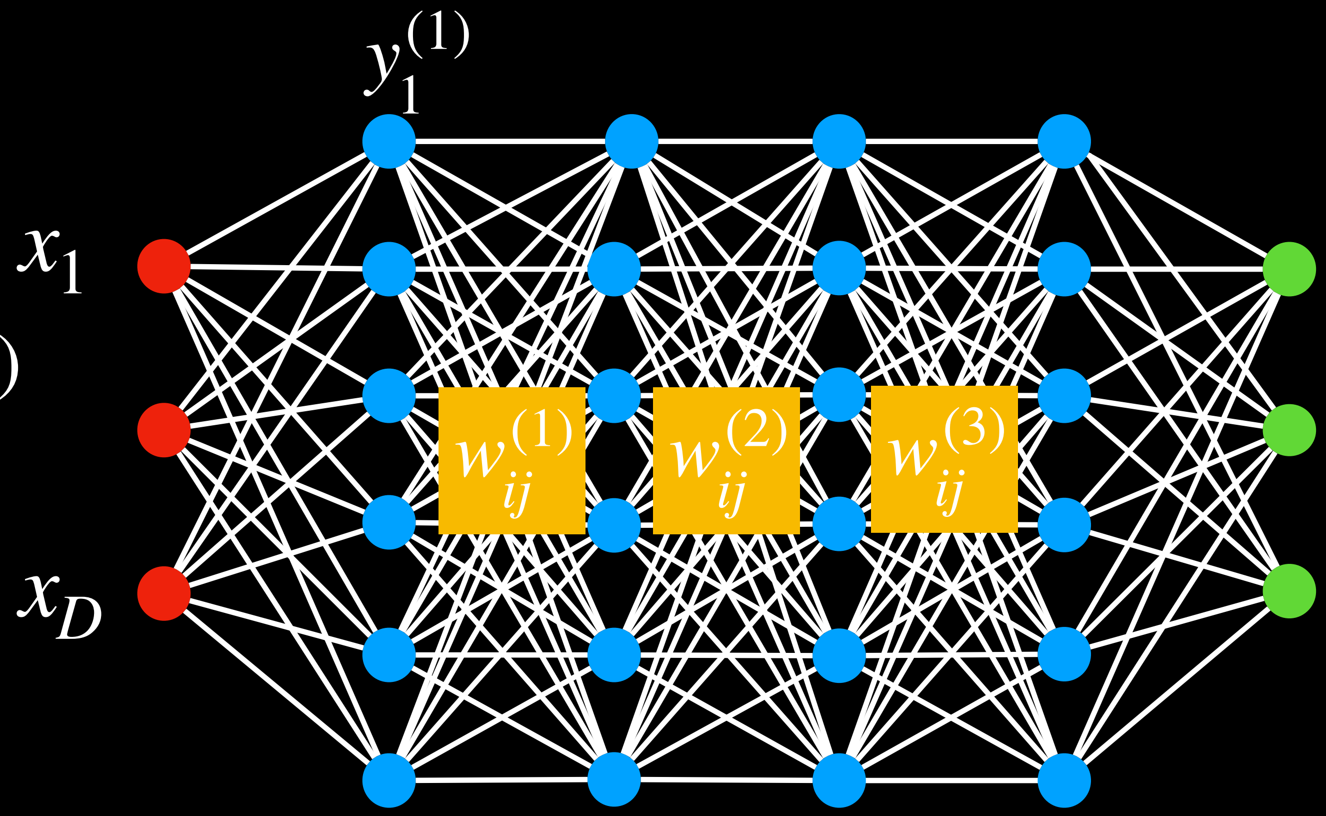
  - Squared Euclidean distance between true and desired output
  - Note: this is differentiable

$$\frac{Div(Y,d)}{dy_i} = (y_i - d_i)$$

Training Neural Network with GD

Loss:

$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N} div(f(W_i; X), d_i)$$



Algorithm:

$$w_{i,t} = w_{i,t-1} - \eta^k \frac{df}{dw_{i,t-1}}$$

For every layer k, for all i, j, update:

$$w_{ij,t}^{(k)} = w_{ij,t-1}^{(k)} - \eta \frac{dLoss}{dw_{ij,t-1}^{(k)}}$$

Chain rule

- For any nested function $y = f(g(w))$

$$\frac{dy}{dw} = \frac{\partial f}{\partial g(w)} \frac{dg(w)}{dw}$$

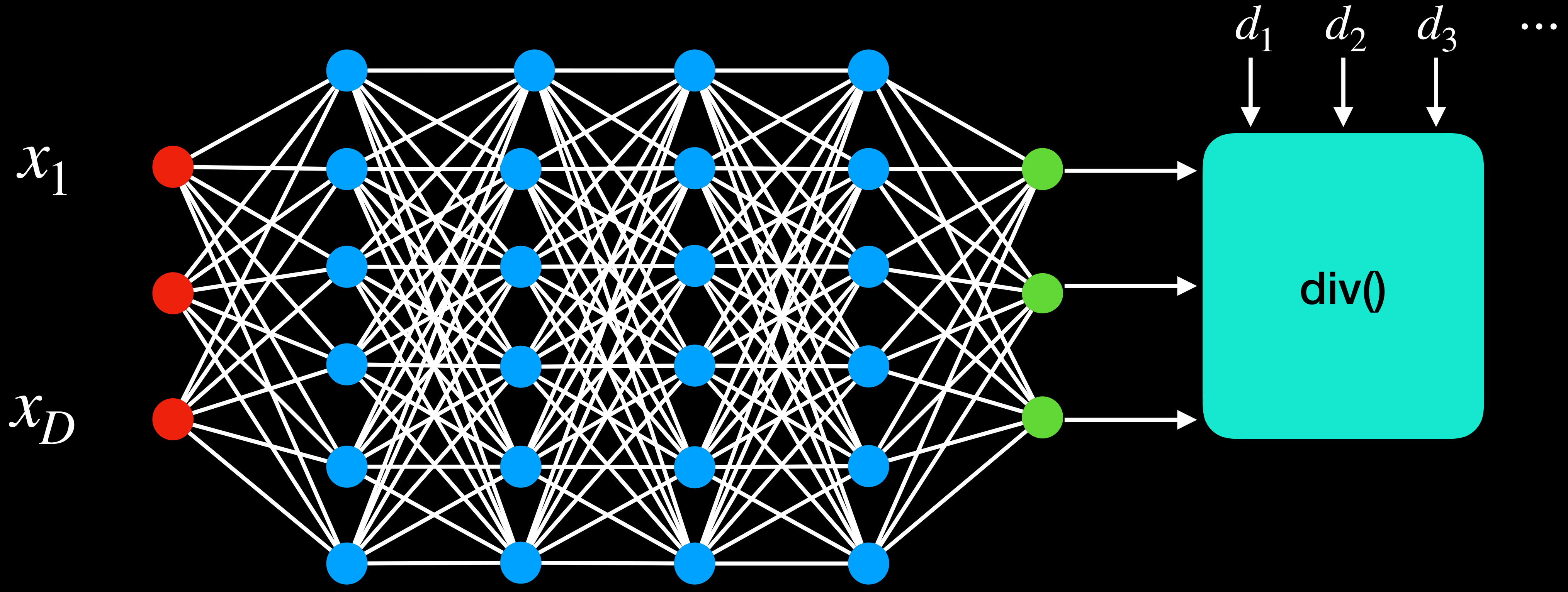
- Check

$$\triangle\, y = \frac{dy}{dw} \triangle\, w$$

$$z = g(w) \rightarrow \triangle\, z = \frac{dg(w)}{dw} \triangle\, w$$

$$y = f(z) \rightarrow \triangle\, y = \frac{df}{dz} \triangle\, z = \frac{df}{dx} \frac{dg(w)}{dw} \triangle\, w$$

How

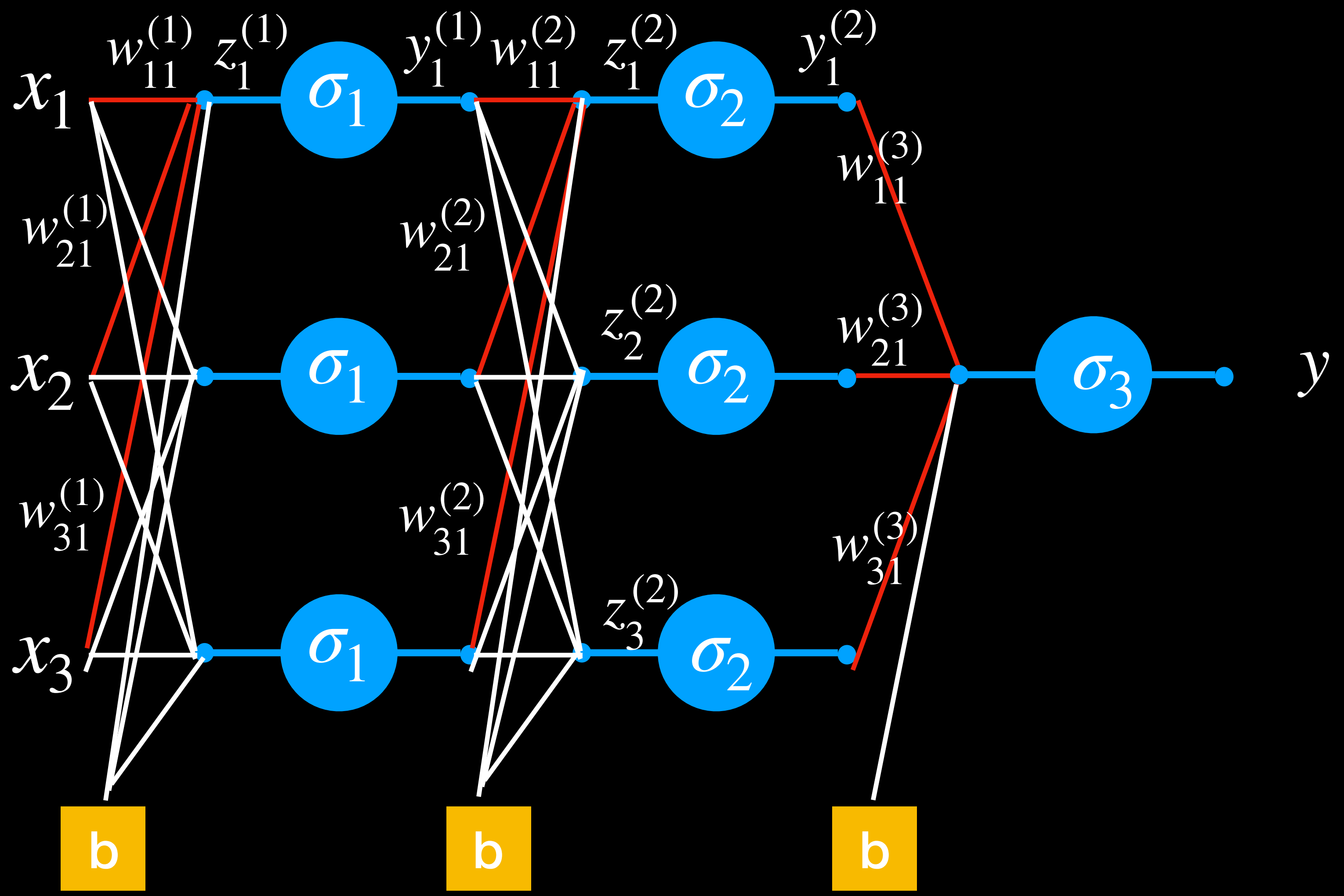


Chain rule

$$w_{ij,t}^{(k)} = w_{ij,t-1}^{(k)} - \eta \frac{dLoss}{dw_{ij,t-1}^{(k)}}$$

Example

$y^{(0)} = x$

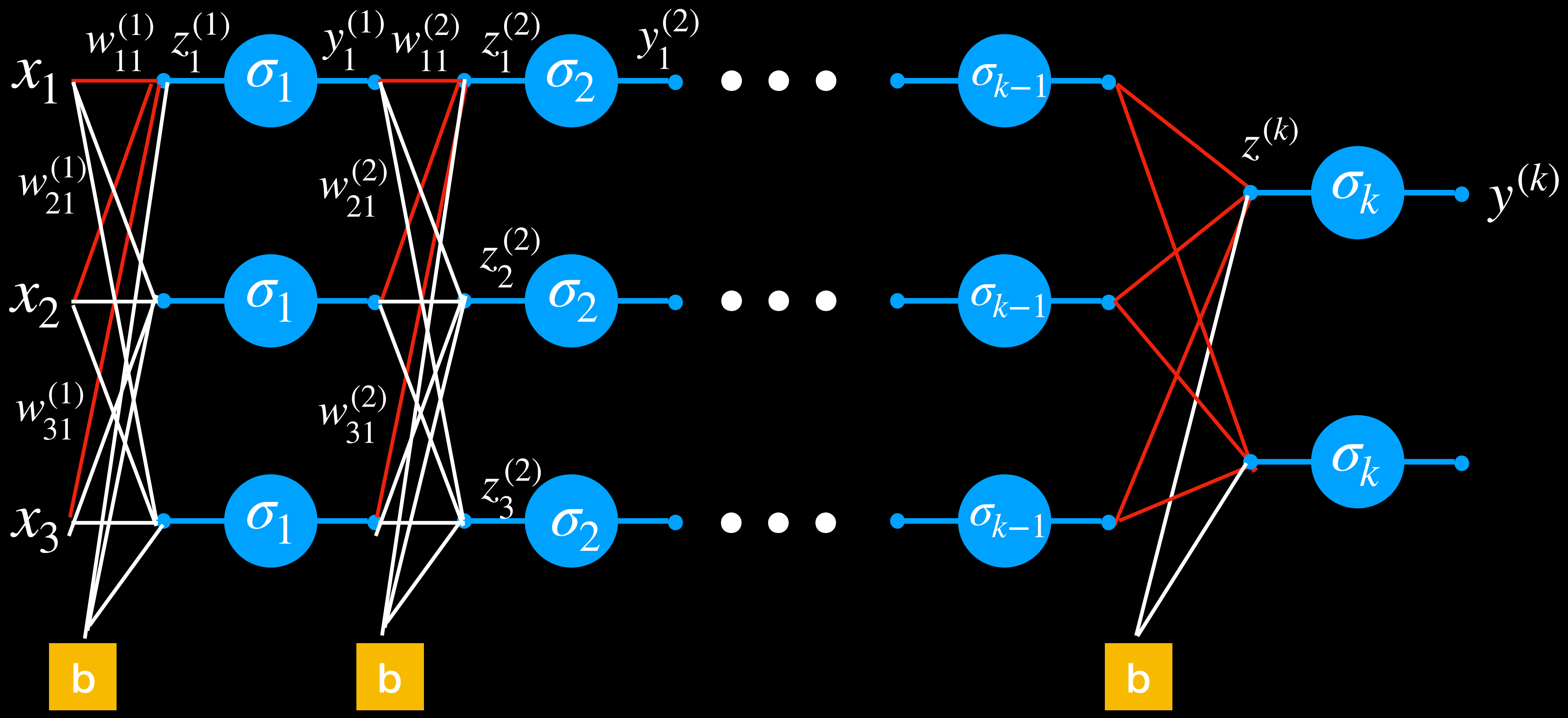


$$z_1^1 = \sum_i w_{i1}^{(1)} y_i^{(0)} \quad y_1^1 = \sigma_1(z_1^1) \rightarrow z_j^2 = \sum_i w_{ij}^{(2)} y_i^{(1)} \quad y_1^2 = \sigma(z_1^2) \rightarrow$$

$$\rightarrow \quad \rightarrow \quad \rightarrow \quad y$$

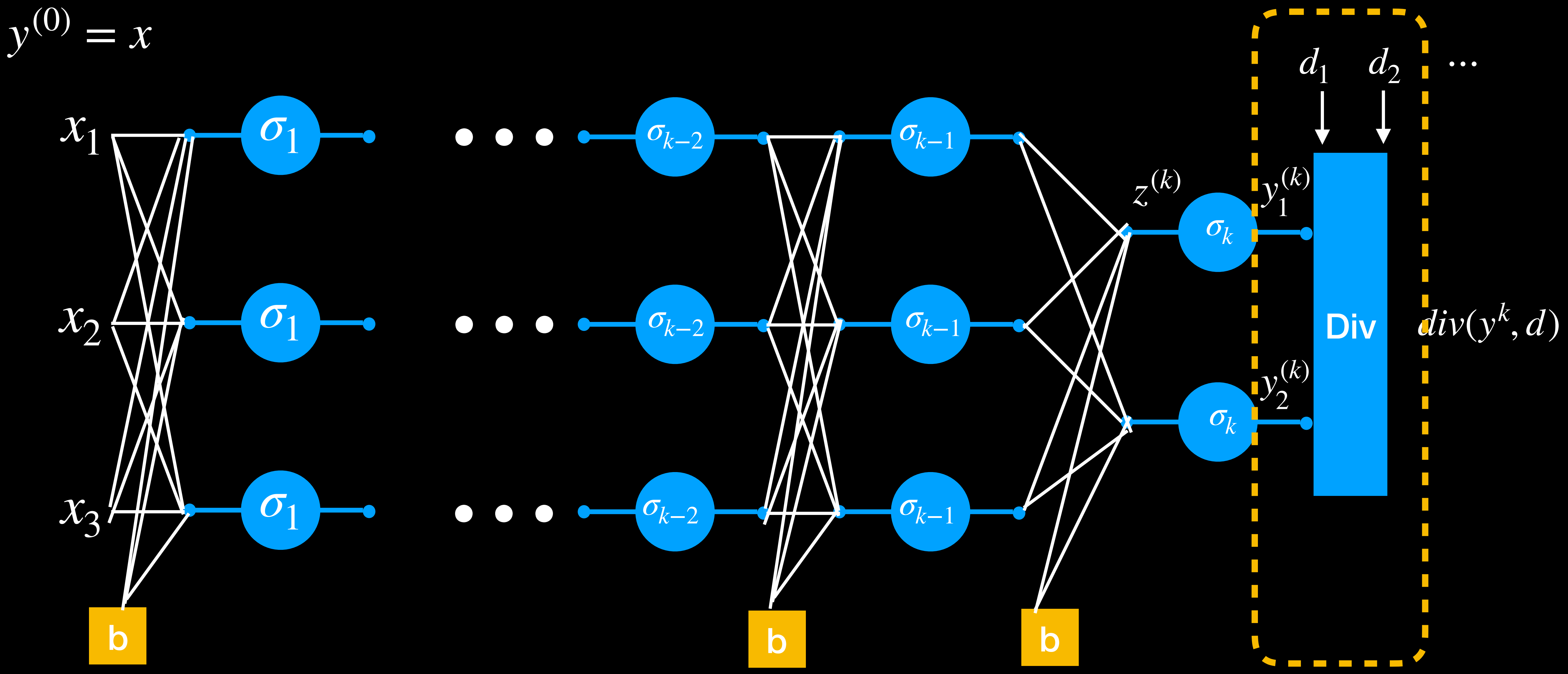
Forward Computation

$y^{(0)} = x$



$$Z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)} \qquad y_j^{(k)} = \sigma(z_j^{(k)})$$

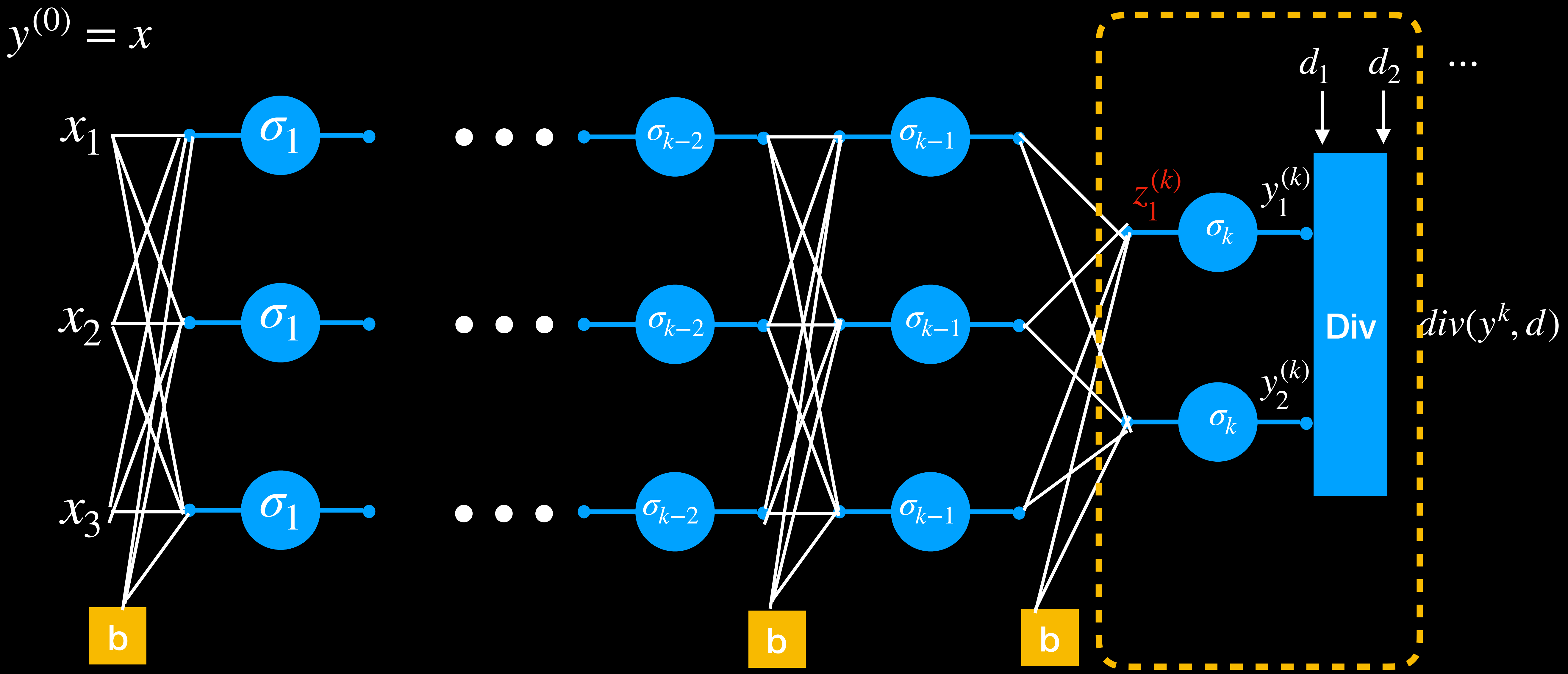
Backward Computation: derivatives



The derivative w.r.t the actual output of the network is simply the derivative w.r.t to the output of the final layer of the network

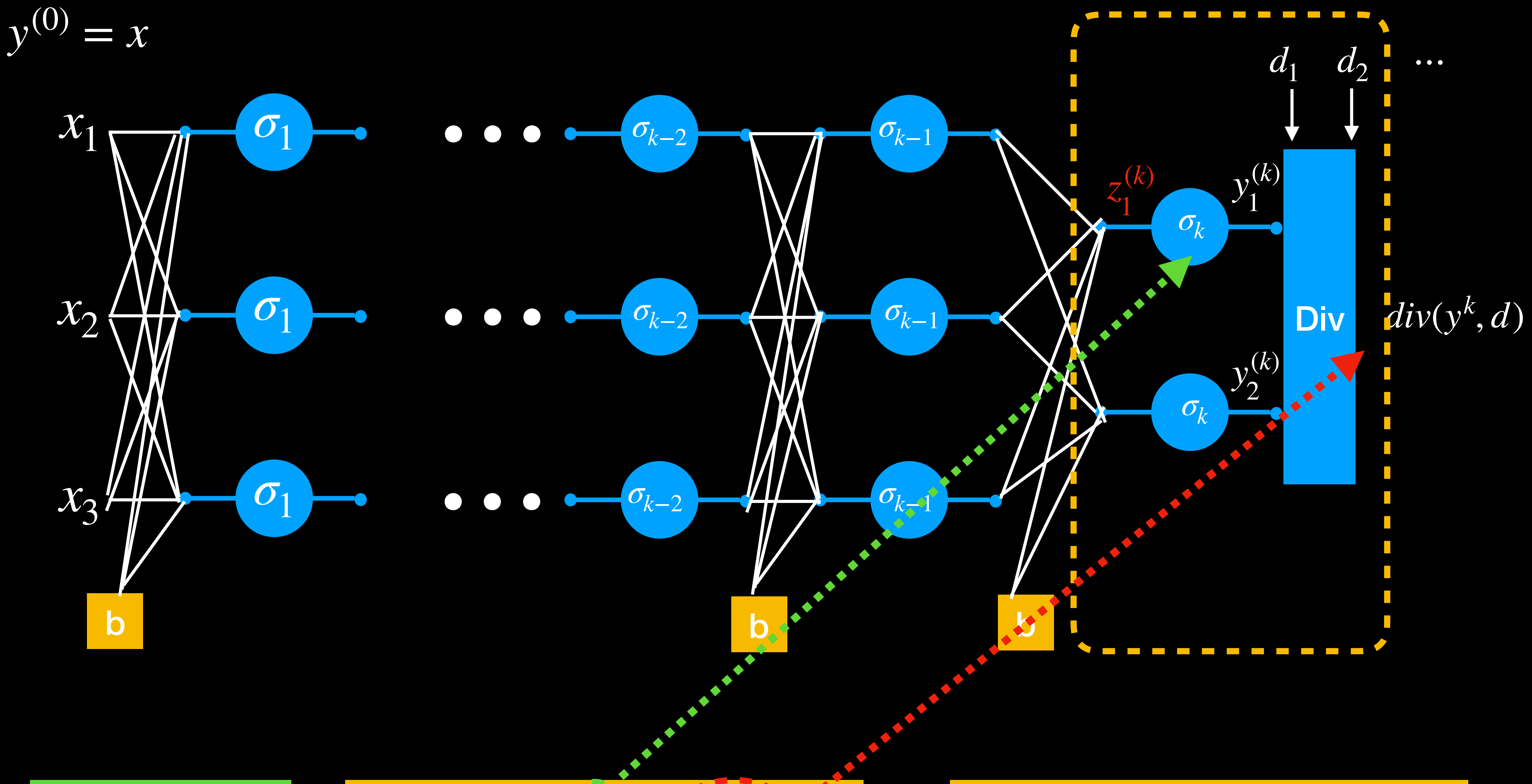
$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(k)}}$$

Backward Computation: derivatives



$$\frac{\partial Div(Y,d)}{\partial z_1^{(k)}} = \frac{\partial y_1^{(k)}}{\partial z_1^{(k)}} \frac{\partial Div(Y,d)}{\partial y_1^{(k)}} \longleftarrow \frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(k)}}$$

Backward Computation: derivatives



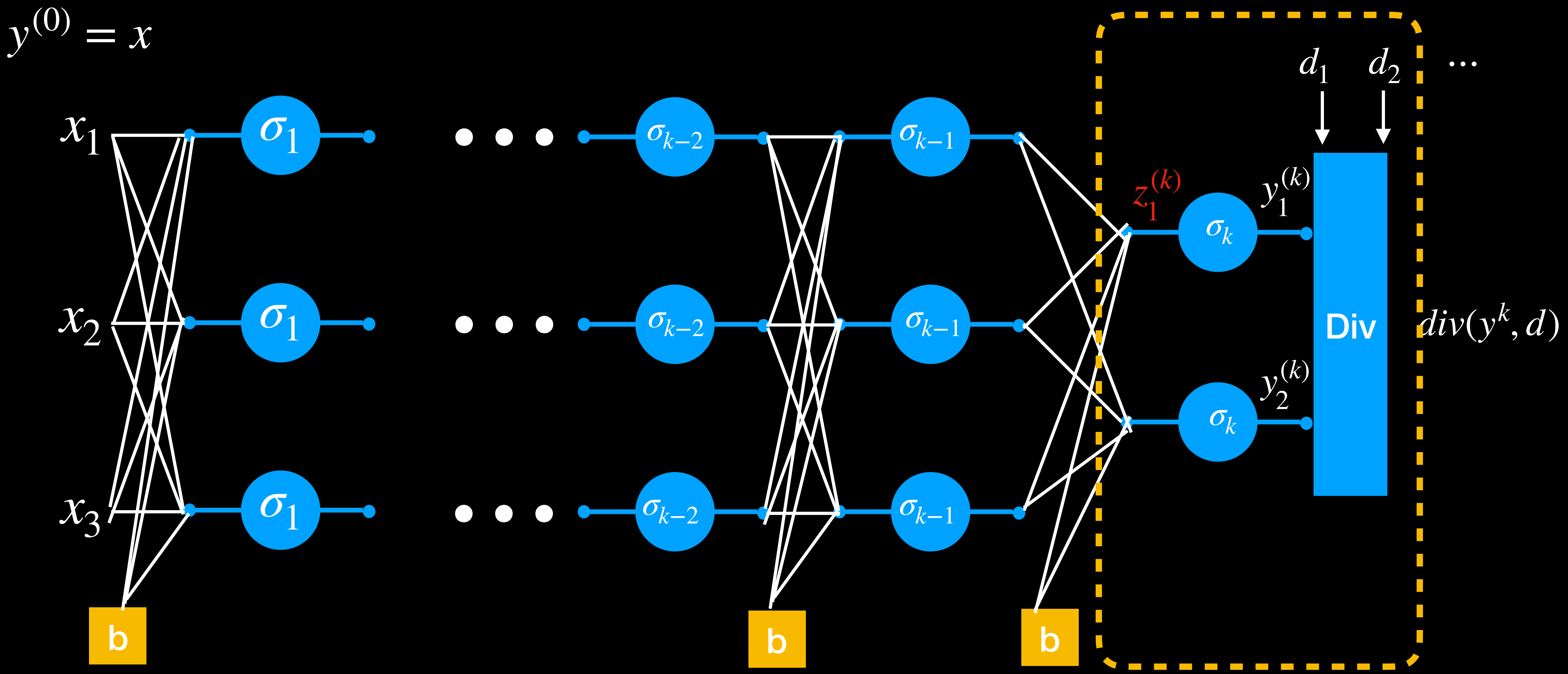
$$y_j^{(k)} = \sigma_k(z_j^{(k)})$$

$$\sigma_k'(z_j^{(k)})$$

$$\frac{\partial Div(Y,d)}{\partial z_1^{(k)}} = \frac{\partial y_1^{(k)}}{\partial z_1^{(k)}} \frac{\partial Div(Y,d)}{\partial y_1^{(k)}}$$

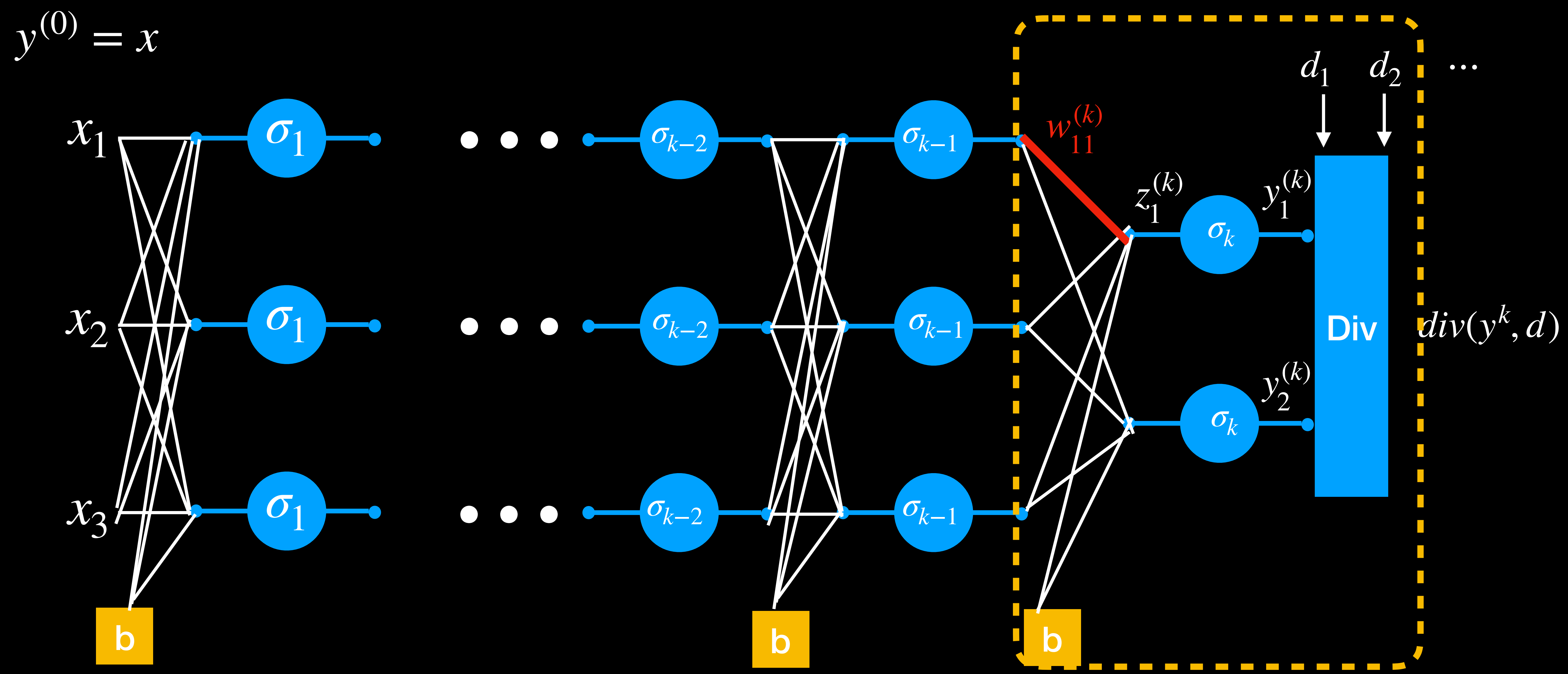
$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(k)}}$$

Backward Computation: derivatives



$$\frac{\partial Div(Y,d)}{\partial z_1^{(k)}} = \frac{\partial y_1^{(k)}}{\partial z_1^{(k)}} \frac{\partial Div(Y,d)}{\partial y_1^{(k)}} = \sigma_k^{'}(z_1^{(k)}) \frac{\partial Div}{\partial y_1^{(k)}}$$

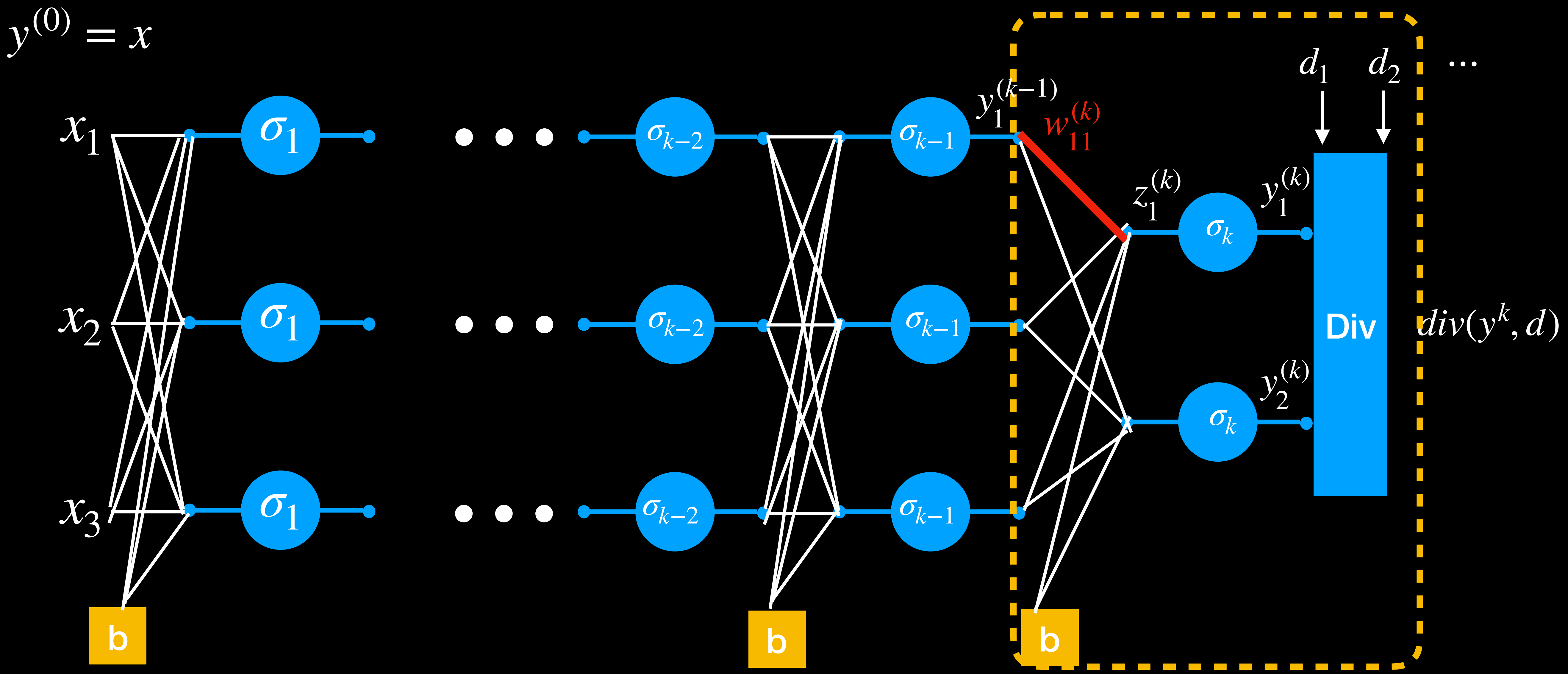
Backward Computation: derivatives



$$\frac{\partial Div(Y,d)}{\partial w_{11}^{(k)}} = \frac{\partial z_1^{(k)}}{\partial w_{11}^{(k)}} \frac{\partial Div}{\partial z_1^{(k)}}$$

$$\frac{\partial Div(Y,d)}{\partial z_1^{(k)}} = \frac{\partial y_1^{(k)}}{\partial z_1^{(k)}} \frac{\partial Div(Y,d)}{\partial y_1^{(k)}} = \sigma_k'(z_1^{(k)}) \frac{\partial Div}{\partial y_1^{(k)}}$$

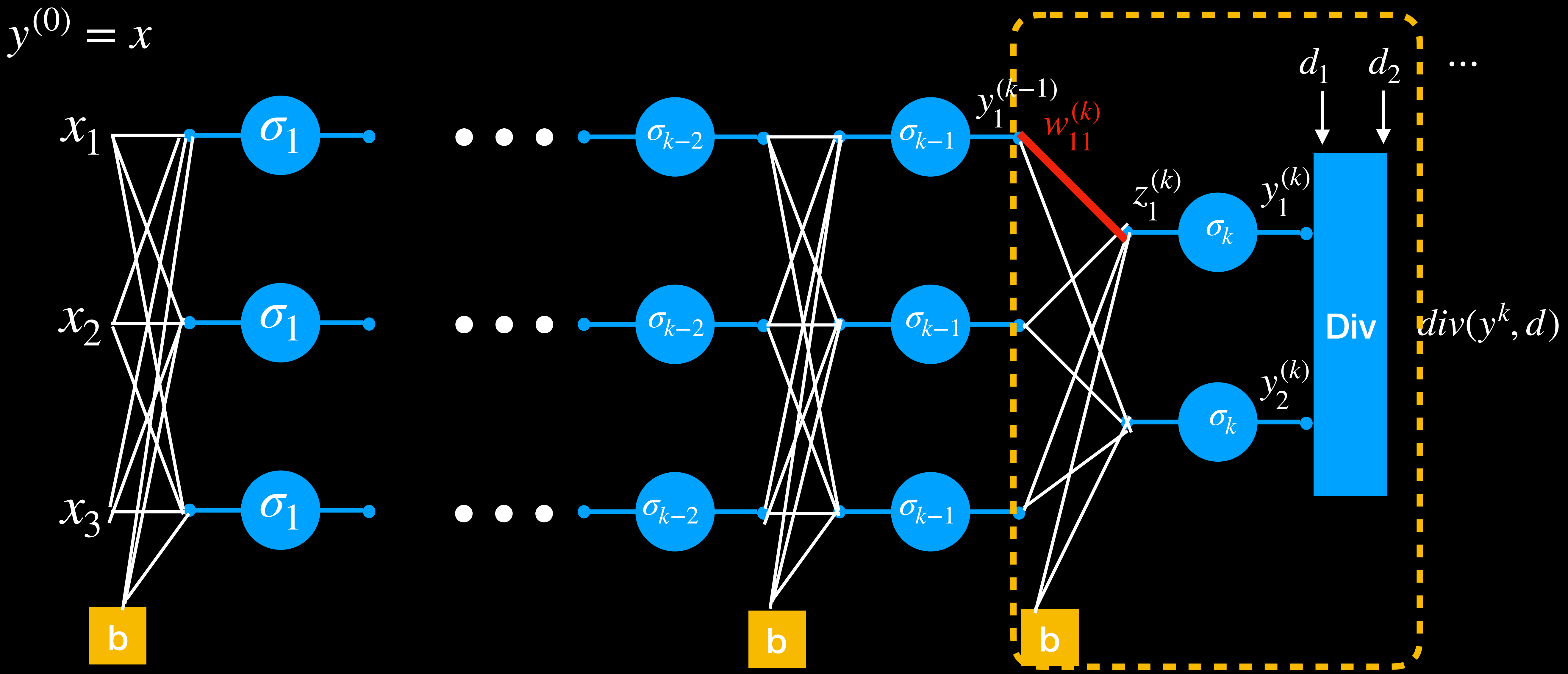
Backward Computation: derivatives



$$\frac{\partial Div(Y,d)}{\partial w_{11}^{(k)}} = \frac{\partial z_1^{(k)}}{\partial w_{11}^{(k)}} \frac{\partial Div}{\partial z_1^{(k)}}$$

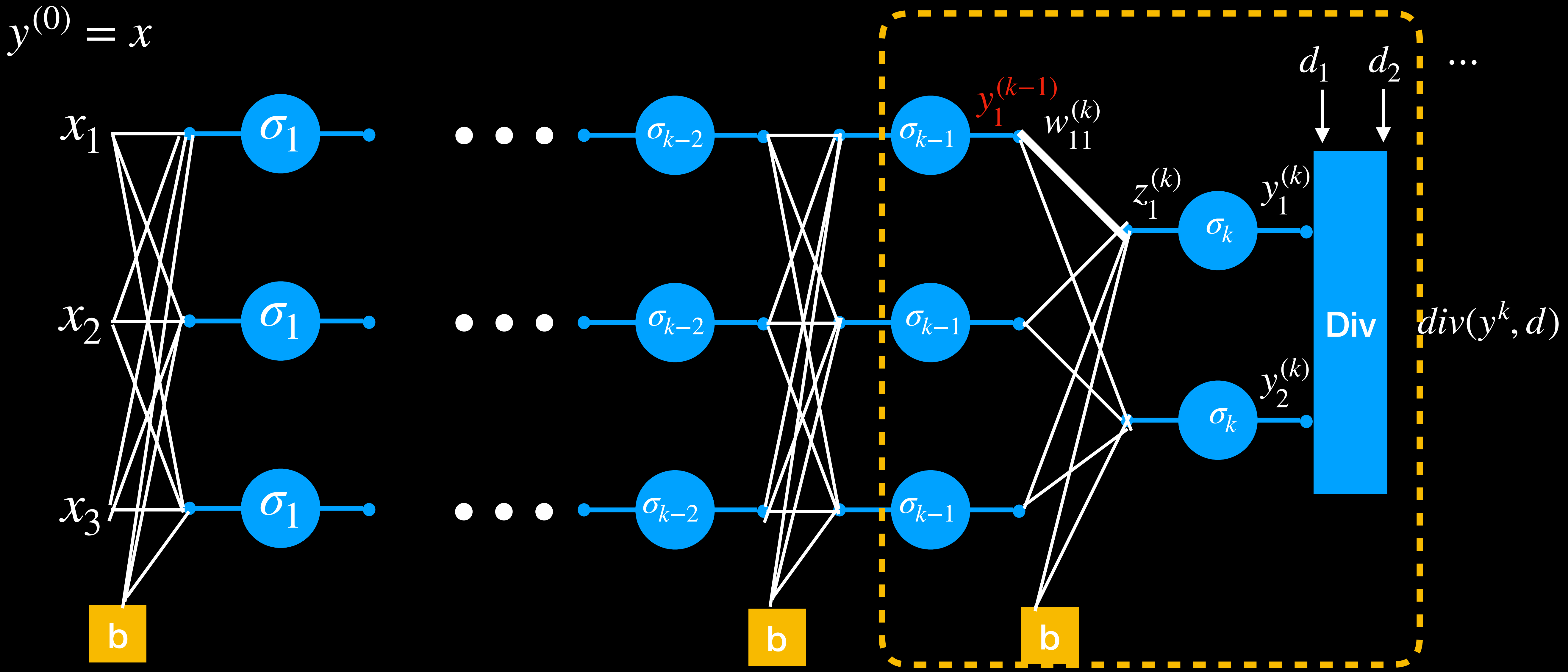
$$z_{11}^{(k)} = w_{11}^{k} y_1^{(k-1)} + b$$

Backward Computation: derivatives



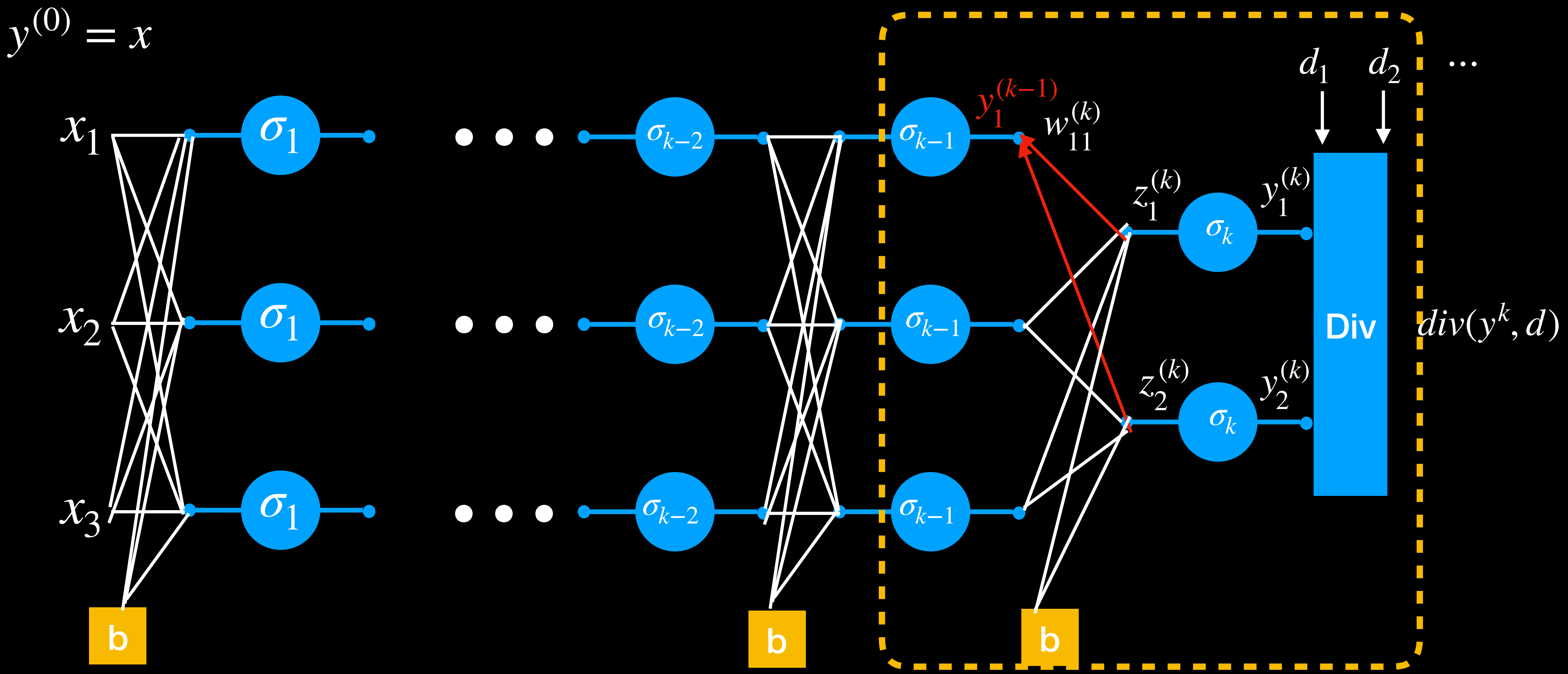
$$\frac{\partial Div(Y,d)}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}$$

Backward Computation: derivatives



$$\frac{\partial Div(Y, d)}{\partial y_1^{(k-1)}} = ?$$

Backward Computation: derivatives

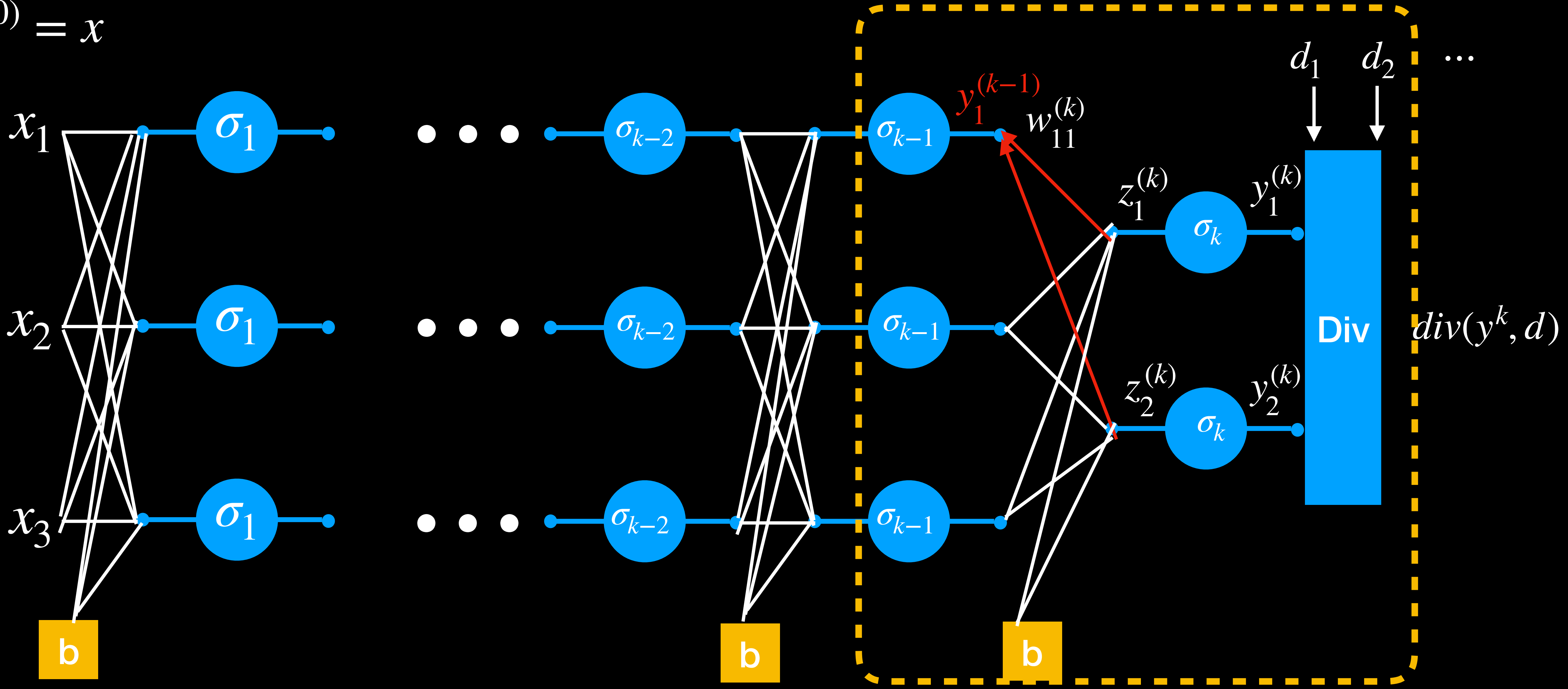


$$\frac{\partial Div(Y,d)}{\partial y_1^{(k-1)}} = \sum_j \frac{\partial z_j(k)}{\partial y_1^{(k-1)}} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$\frac{\partial Div(Y,d)}{\partial y_1^{(k-1)}} = ?$$

Backward Computation: derivatives

$y^{(0)} = x$

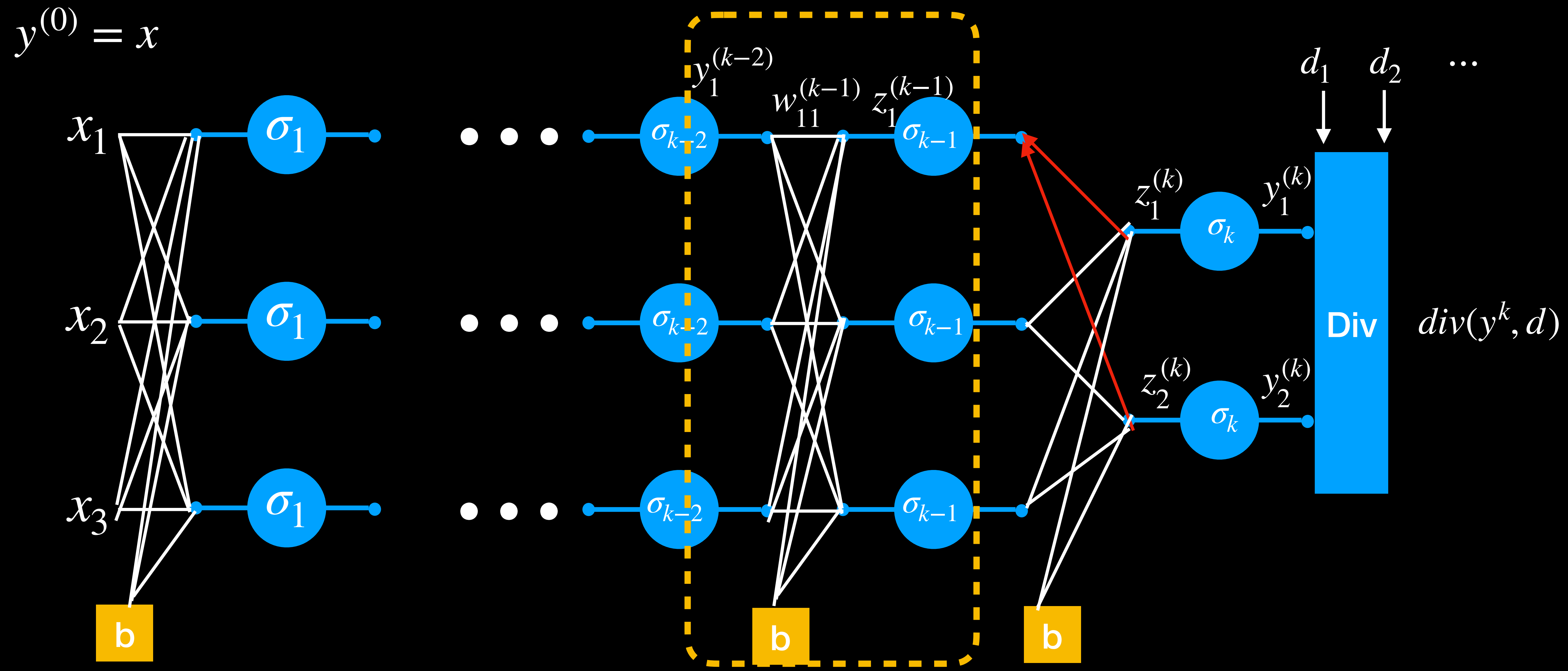


$$\frac{\partial Div(Y,d)}{\partial y_1^{(k-1)}} = \sum_j w_{1j}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$z_j^{(k)} = w_{1j}^k y_1^{(k-1)} + b$$

$$\frac{\partial Div(Y,d)}{\partial y_1^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_1^{(k-1)}} \frac{\partial Div}{\partial z_j^{(k)}}$$

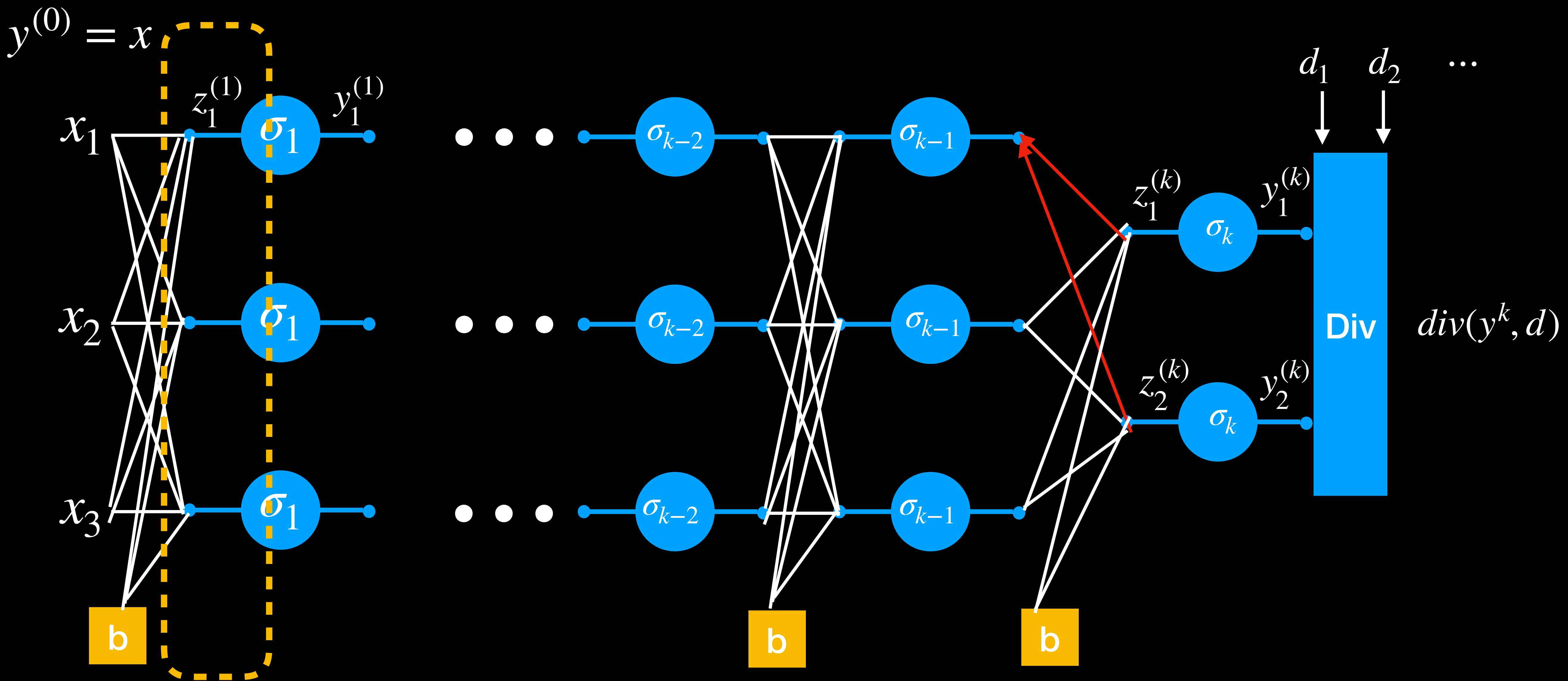
Backward Computation: derivatives



$$\frac{\partial Div(Y,d)}{\partial y_i^{(k-2)}} = \sum_j w_{ij}^{(k-1)} \frac{\partial Div}{\partial z_j^{(k-1)}}$$

$$\frac{\partial Div(Y,d)}{\partial w_{ij}^{(k-1)}} = y_i^{(k-2)} \frac{\partial Div}{\partial z_j^{(k-1)}}$$

Backward Computation: derivatives



$$\frac{\partial Div(Y,d)}{\partial w_{ij}^{(1)}} = y_i^{(1)} \frac{\partial Div}{\partial z_j^{(1)}}$$

$$\frac{\partial Div(Y,d)}{\partial z_i^{(1)}} = \sigma_1'(z_i^{(1)}) \frac{\partial Div}{\partial y_i^{(1)}}$$

$$\frac{\partial Div(Y,d)}{\partial y_i^{(1)}} = \sum_j w_{ij}^{(2)} \frac{\partial Div}{\partial z_j^{(2)}}$$

Backward Computation: derivatives

For Output layer (k):

$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(k)}}$$

$$\frac{\partial Div(Y,d)}{\partial z_i^{(k)}} = \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}} \frac{\partial Div(Y,d)}{\partial y_i^{(k)}}$$

Called "Backpropagation" because the derivative of the loss is propagated "backwards" through the network

For layer k -1 to 1:

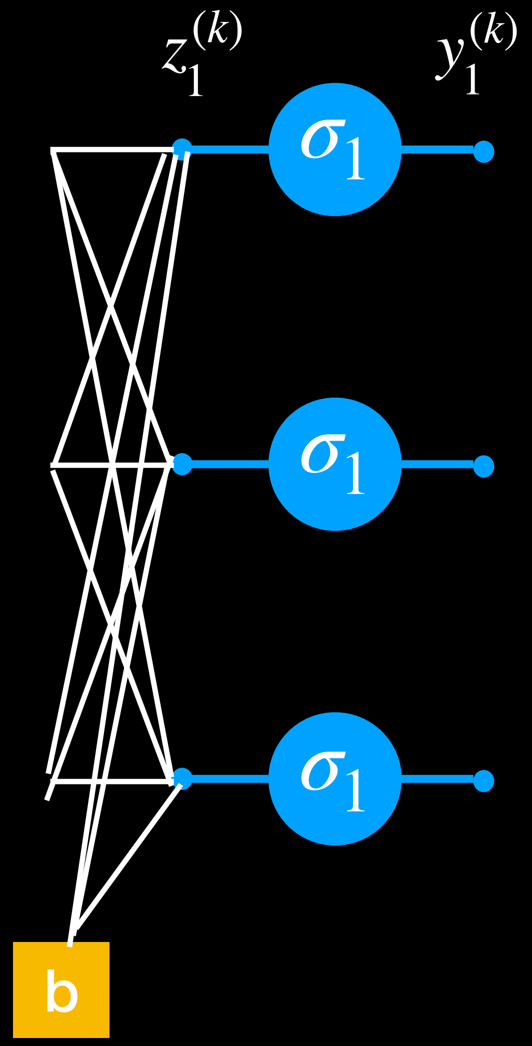
$$\frac{\partial Div(Y,d)}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}}$$

Backward weighted combination of next layer

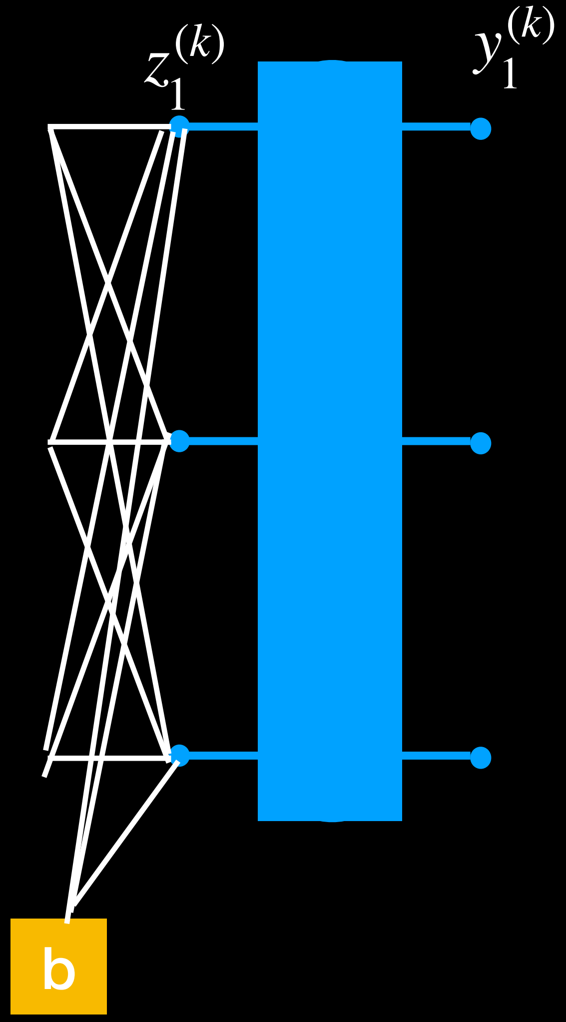
$$\frac{\partial Div(Y,d)}{\partial z_i^{(k)}} = \sigma_k^{'}(z_i^{(k)}) \frac{\partial Div}{\partial y_i^{(k)}}$$

$$\frac{\partial Div(Y,d)}{\partial w_{ij}^{(k+1)}} = y_i^{(k)} \frac{\partial Div}{\partial z_j^{(k+1)}}$$

Scalar activation VS vector activation

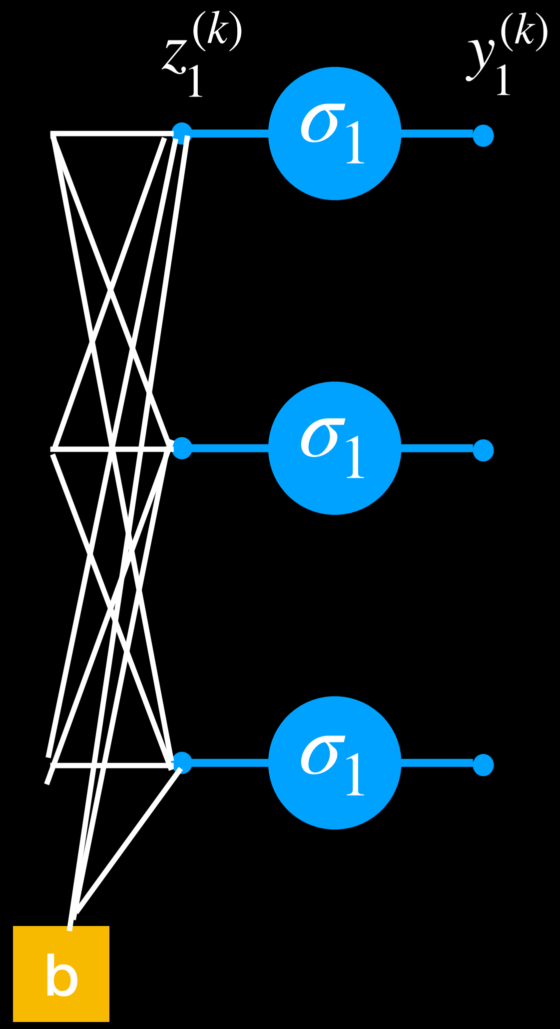


Scalar activation: Modifying z only changes corresponding y

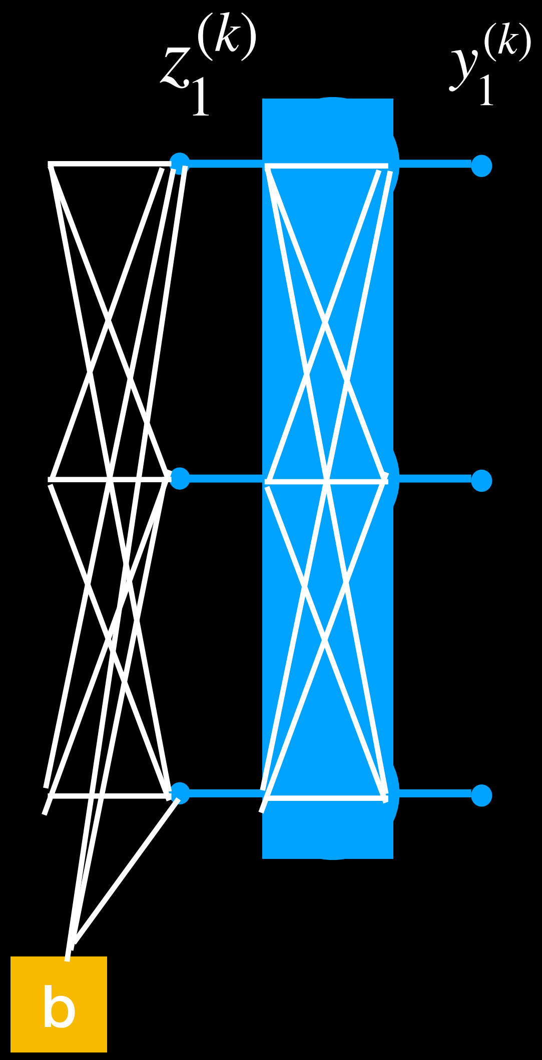


Vector activation: Modifying z potentially changes all y

Scalar activation VS vector activation

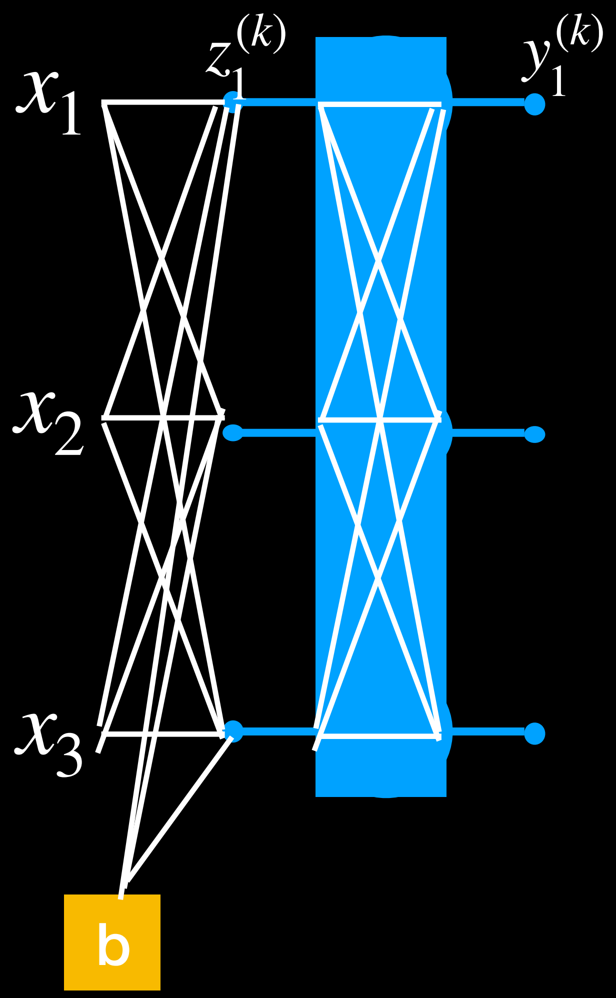


$$\frac{\partial Div(Y,d)}{\partial z_i^{(k)}} = \frac{dy_i^{(k)}}{dz_i^{(k)}} \frac{\partial Div}{\partial y_i^{(k)}}$$



$$\frac{\partial Div(Y,d)}{z_i^{(k)}} = \sum_j \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \frac{\partial Div}{\partial y_j^{(k)}}$$

vector activation example: Softmax



$$y_i^{(k)} = \frac{exp(z_i^{(k)})}{\sum_j exp(z_j^{(k)})}$$

Forward Pass

$$\frac{\partial Div(Y,d)}{z_i^{(k)}} = \sum_j \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \frac{\partial Div}{\partial y_j^{(k)}}$$

Backward Pass

$$\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = y_i^k(1 - y_i^k) \quad if \quad i = j$$

$$\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = -y_i^k y_j^k \quad if \quad i \neq j$$