NEURAL NETWORK & BACK-PROPAGATION LEARNING

Lecture-2

STTP on “Deep Learning, Computer Vision and Speech Processing”

By: Suprava, Patnaik, Professor, ExTC, XIE, Mumbai

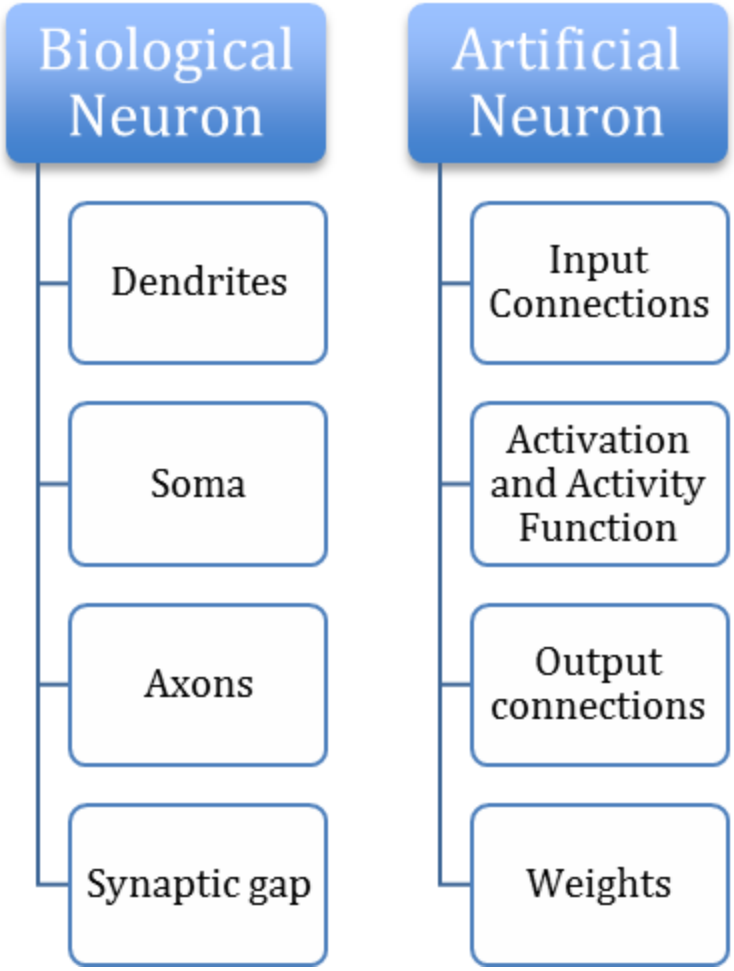
Outline

- Introduction
- Softmax
- Backpropagation

Neural Network Model

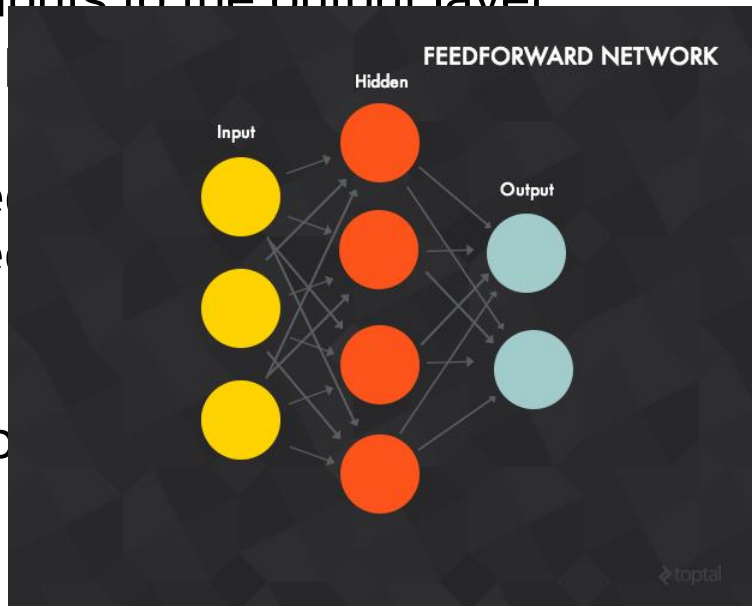
- A system that processes information in somewhat bland analogy to how information is thought to be processed by biological entities.

- Three characteristics:
  - **NN architecture:** feed-forward, recurrent, multi-single layer, so on..
  - **The learning algorithm:** Hebbian, Back-propagation, etc.
  - **Activity functions:** monitors what internal-state will cause firing (i,.e output will be passed on to neighboring state)



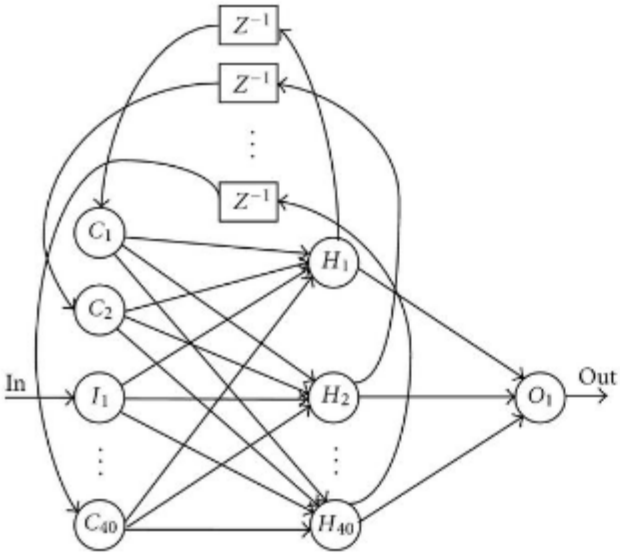
Environment

- A neural network is really just a composition of perceptrons, connected in different ways and operating on different activation functions.
- An input, output, and one or more *hidden* layers. The figure here shows a network with a 3-unit input layer, 4-unit hidden layer and an output layer with 2 units
- Each unit is a single perceptron.
- The units of the input layer serve as inputs for the units of the hidden layer, while the hidden layer units are inputs to the output layer
- Each connection between two neurons
perceptron weights).
- Each unit of layer *t* is typically connecte
layer *t - 1* (although you could disconne
to 0).
- To process input data, you “clamp” the
setting the values of the vector as “c
units.

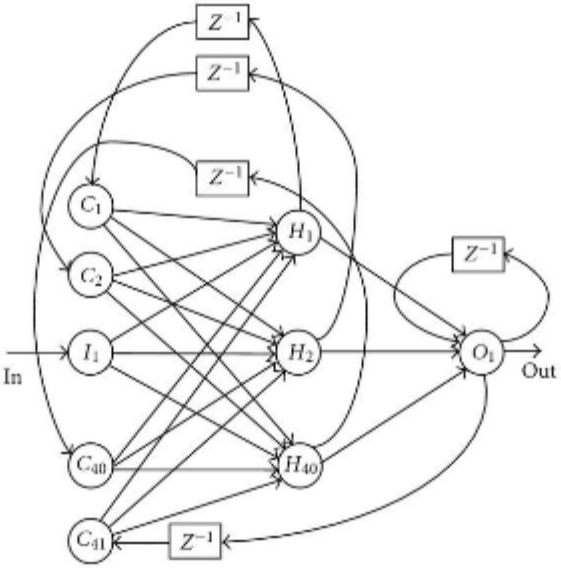


Other Common Architectures

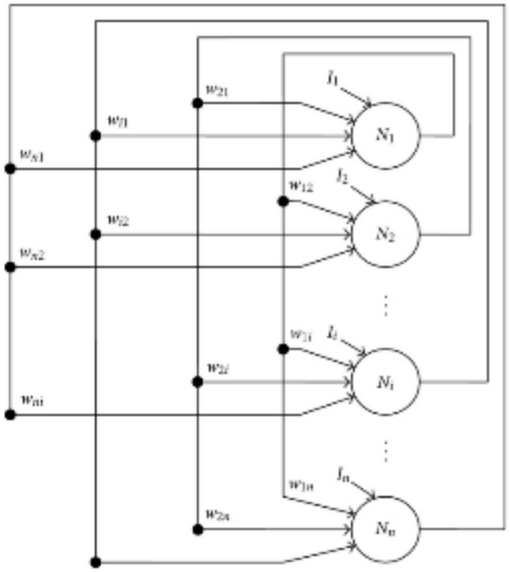
- RNN



Elman Neural Network



Jordan Neural Network

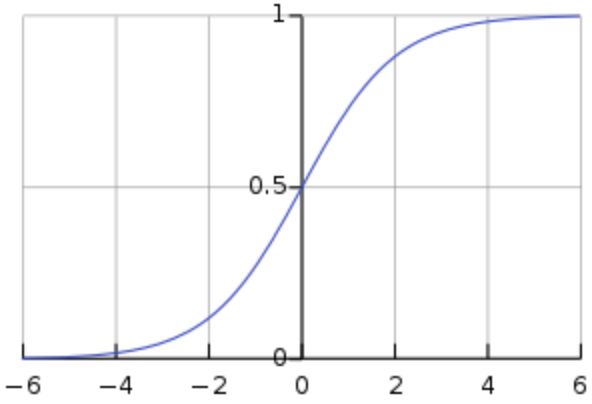


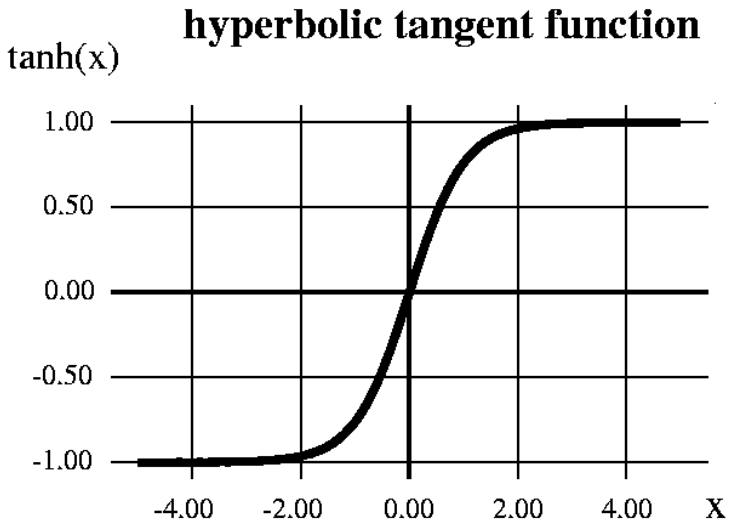
Hopfield Neural Network

Beyond linearity

- A linear composition of a bunch of linear functions (LTU) is still just a linear function, so most neural networks use non-linear activation functions.
- **logistic, tanh, binary or rectifier.**

Logistic function

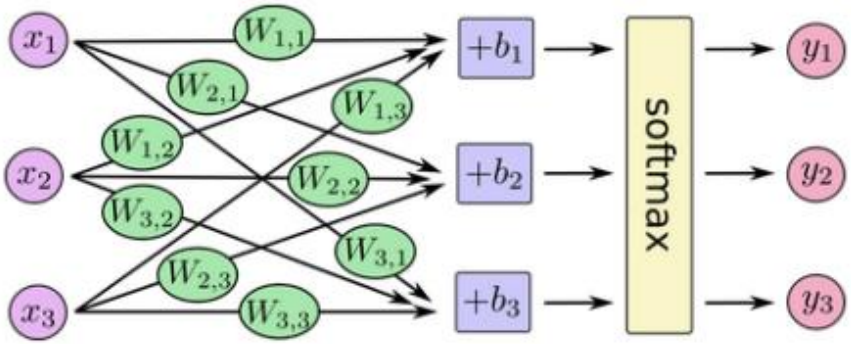




Network training

- Supervised training of the multilayer perceptrons is known as backpropagation or generalised delta rule.
- A training sample is presented and propagated forward through the network.
- The output error is calculated, typically the mean squared error.
- Where *t* is the target value and *y* is the actual network output. Other error calculations are also acceptable, but the MSE is a good choice.
- Network error is minimized using a methods like stochastic gradient descent, Hebbian etc.

3-class Logistic Regression with 3 inputs



$$\boldsymbol{a} = W^T\boldsymbol{x} + \boldsymbol{b}$$

$$W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \end{bmatrix} = \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix}$$

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{a})$$

$$y_i = \frac{\exp(a_i)}{\sum_{j=1}^{3} \exp(a_j)}$$

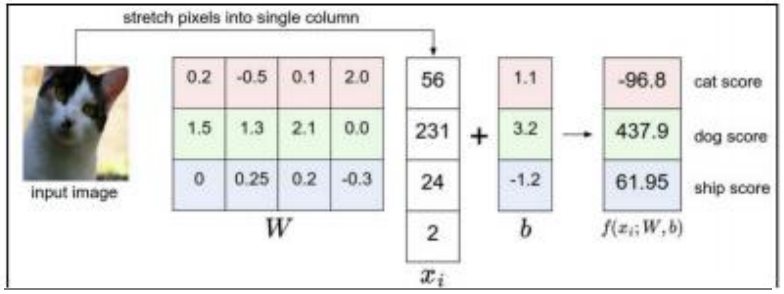
Network Computes

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix} \right)$$

In matrix multiplication notation

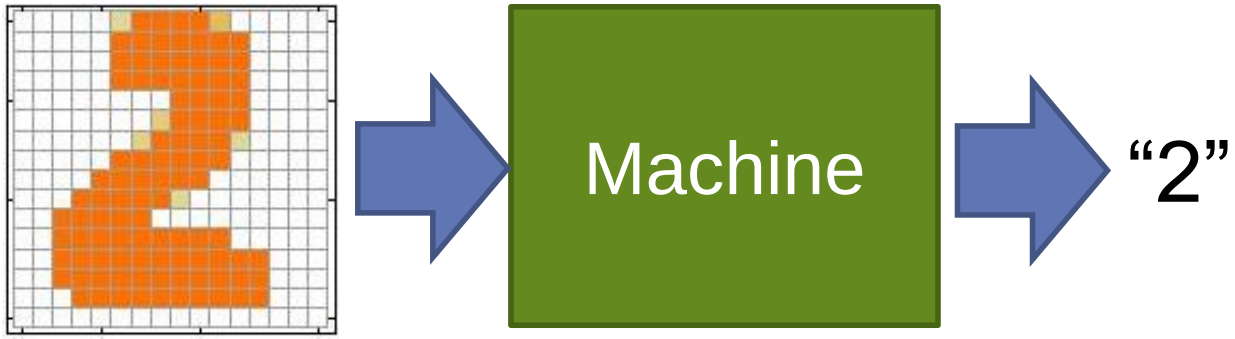
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

An example

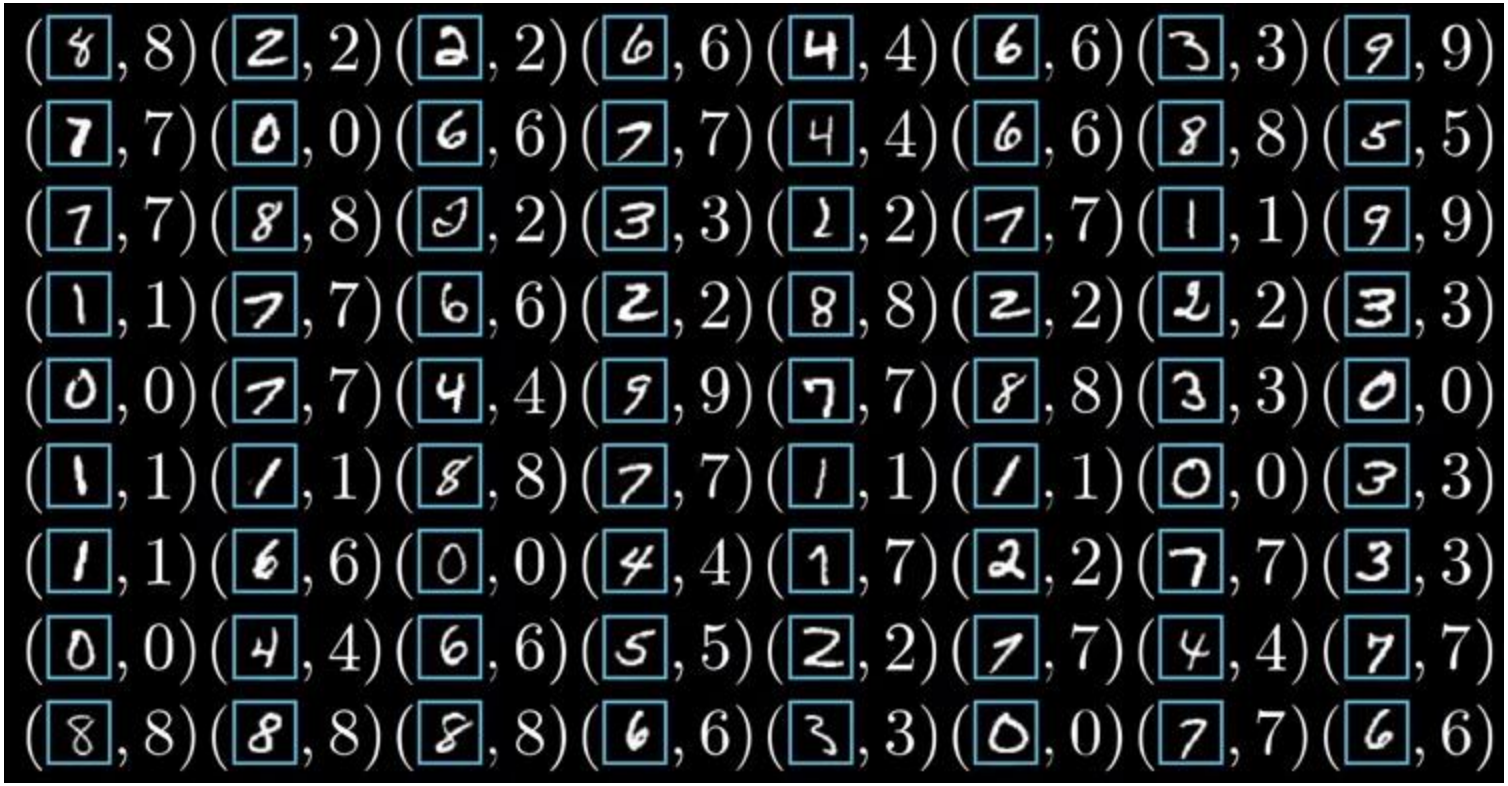


Example of an application

- Handwriting Digit Recognition

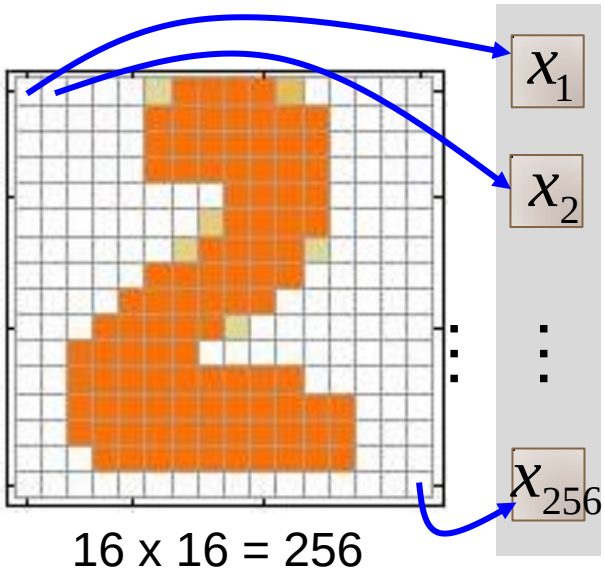


MNSIT


(8, 8) (2, 2) (2, 2) (6, 6) (4, 4) (6, 6) (3, 3) (9, 9)
(7, 7) (0, 0) (6, 6) (7, 7) (4, 4) (6, 6) (8, 8) (5, 5)
(7, 7) (8, 8) (9, 2) (3, 3) (2, 2) (7, 7) (1, 1) (9, 9)
(1, 1) (7, 7) (6, 6) (2, 2) (8, 8) (2, 2) (2, 2) (3, 3)
(0, 0) (7, 7) (4, 4) (9, 9) (7, 7) (8, 8) (3, 3) (0, 0)
(1, 1) (1, 1) (8, 8) (7, 7) (1, 1) (1, 1) (0, 0) (3, 3)
(1, 1) (6, 6) (0, 0) (4, 4) (1, 7) (2, 2) (7, 7) (3, 3)
(0, 0) (4, 4) (6, 6) (5, 5) (2, 2) (1, 7) (4, 4) (7, 7)
(8, 8) (8, 8) (8, 8) (6, 6) (3, 3) (0, 0) (7, 7) (6, 6)

Handwriting Digit Recognition (softmax)

Input

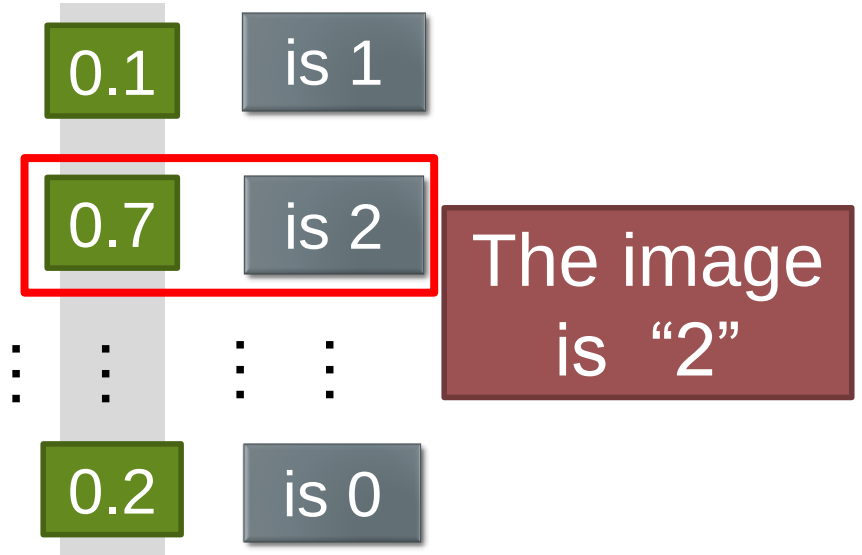


16 x 16 = 256

Ink → 1

No ink → 0

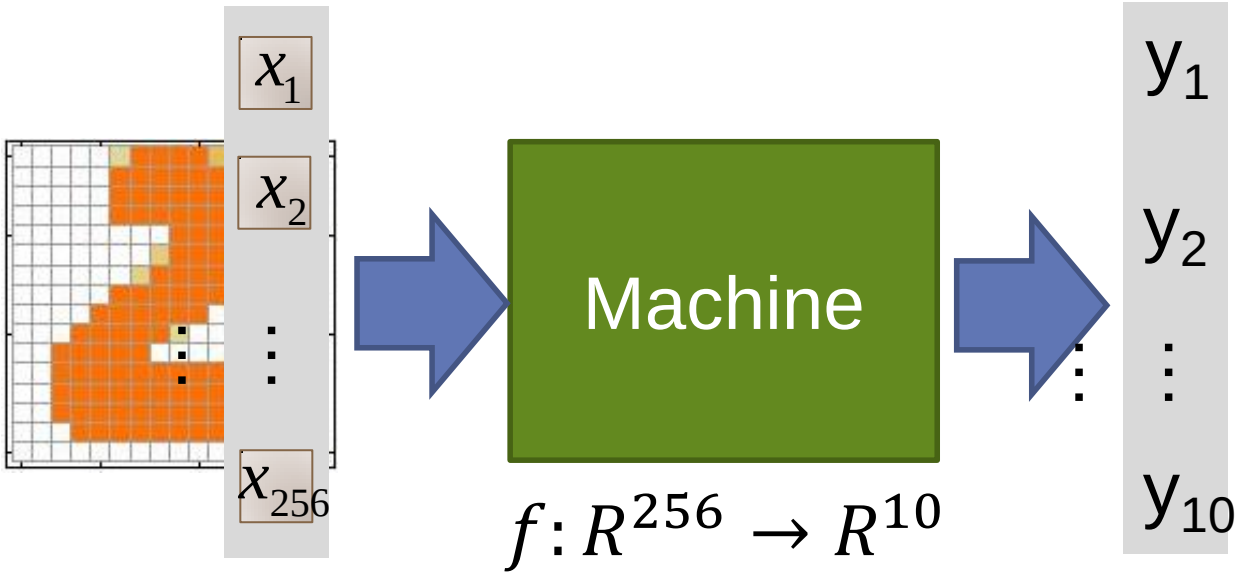
Output



Each dimension represents the confidence of a digit.

Example Application

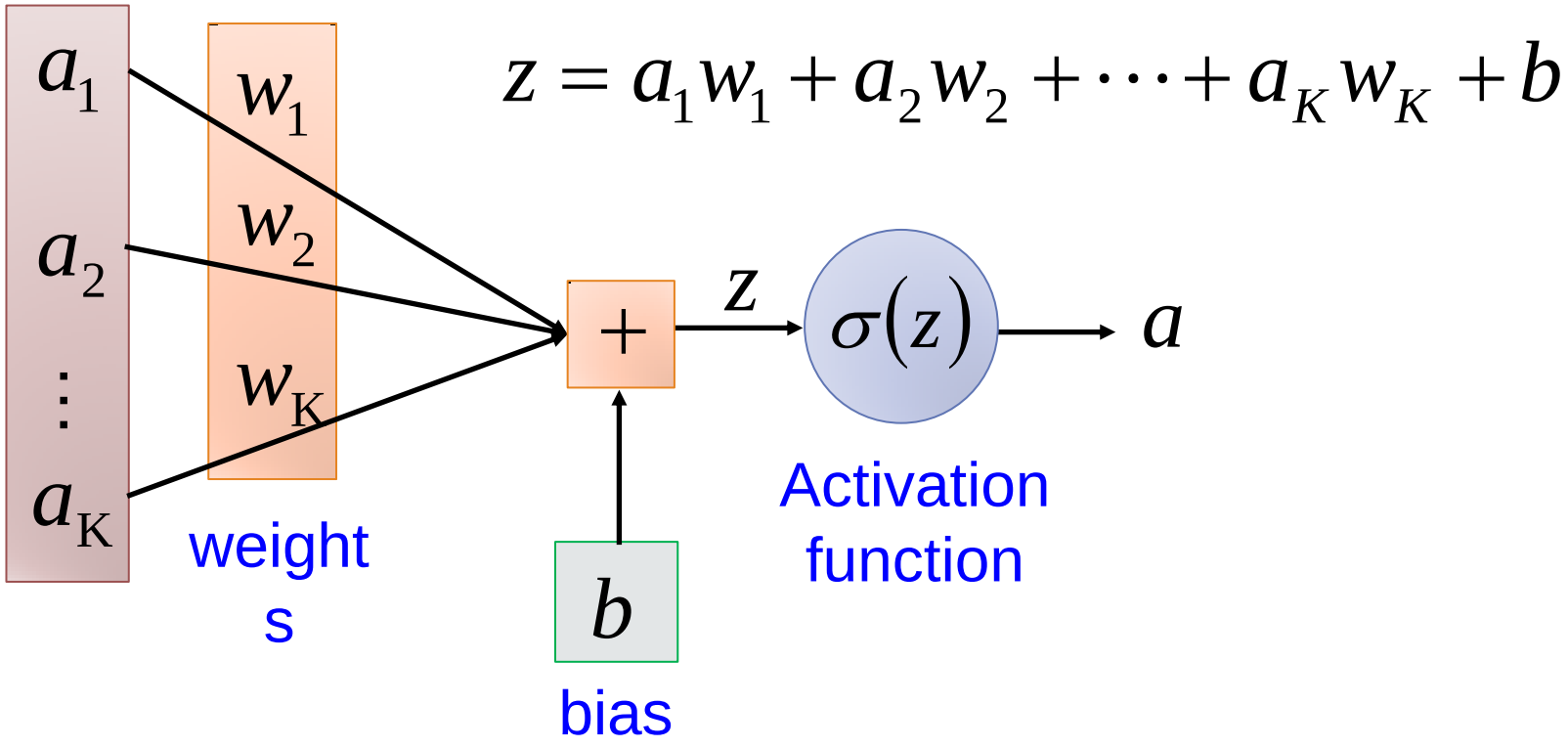
- Handwriting Digit Recognition



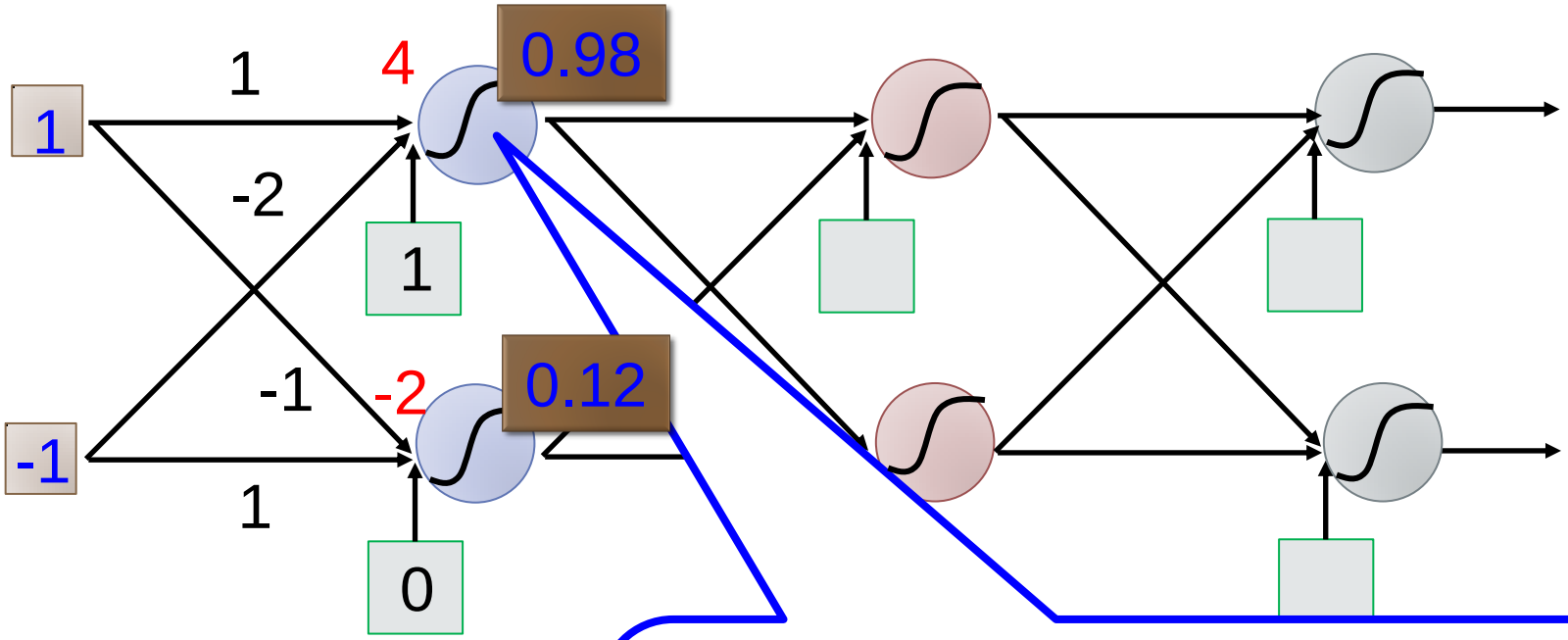
In deep learning, the function f is represented by neural network

Element of Neural Network

Neuron $f: R^K \rightarrow R$

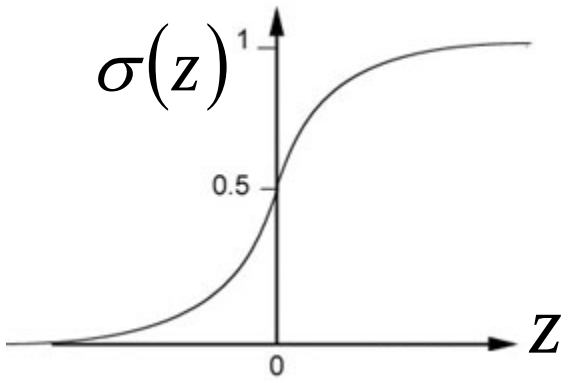


Example of Neural Network

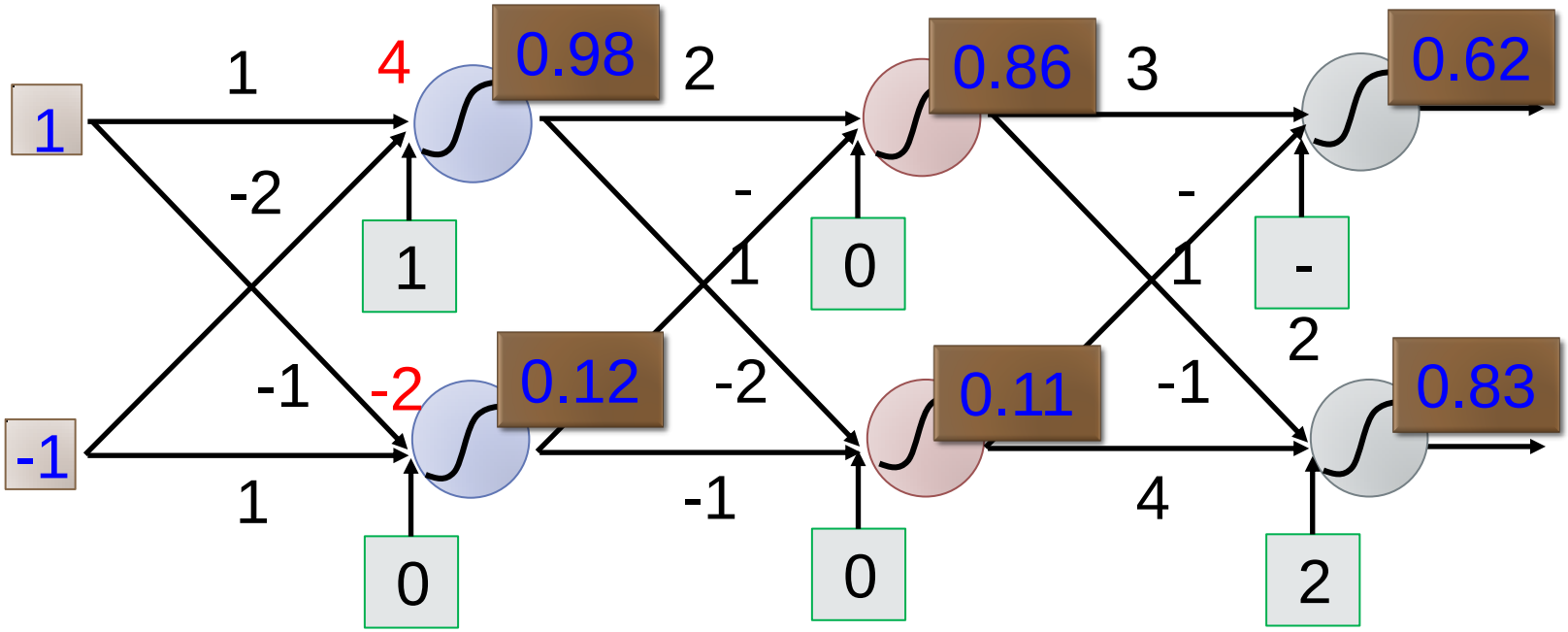


Sigmoid Function

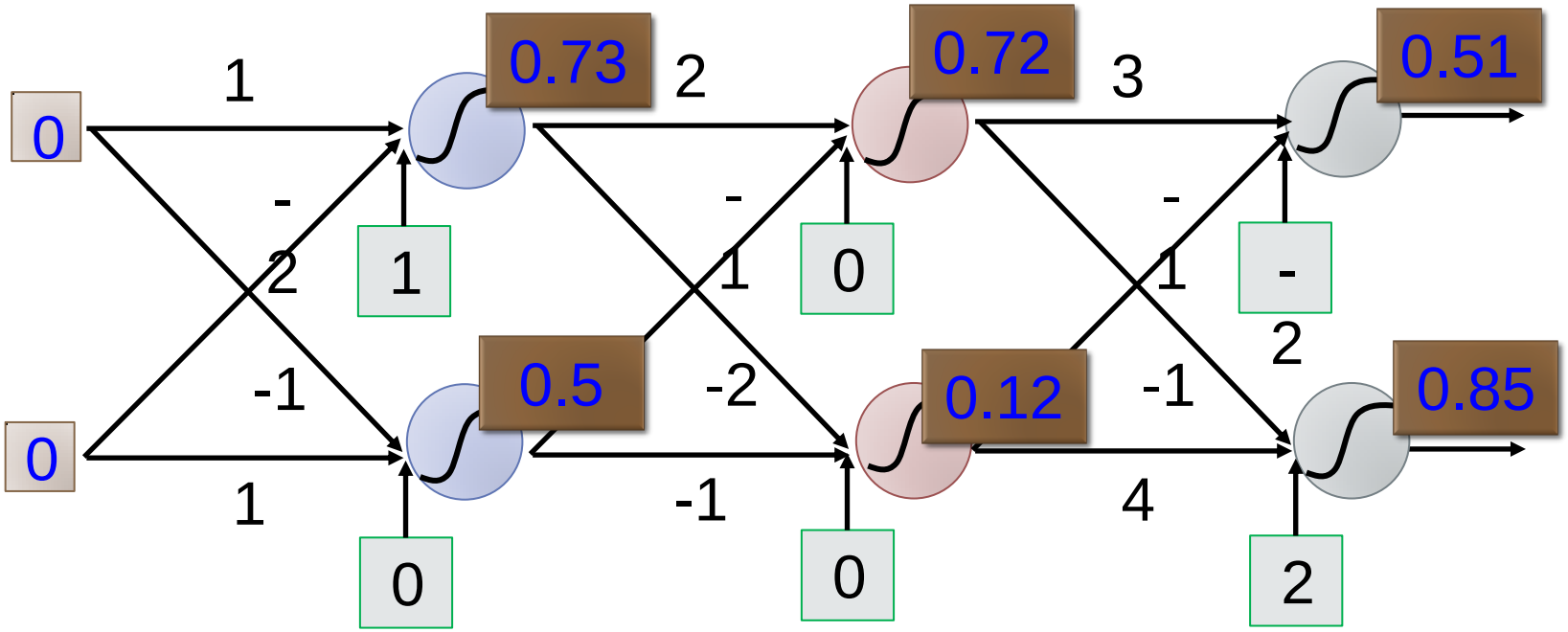
$$\sigma(z) = \frac{1}{1+e^{-z}}$$



Example of Neural Network



Example of Neural Network

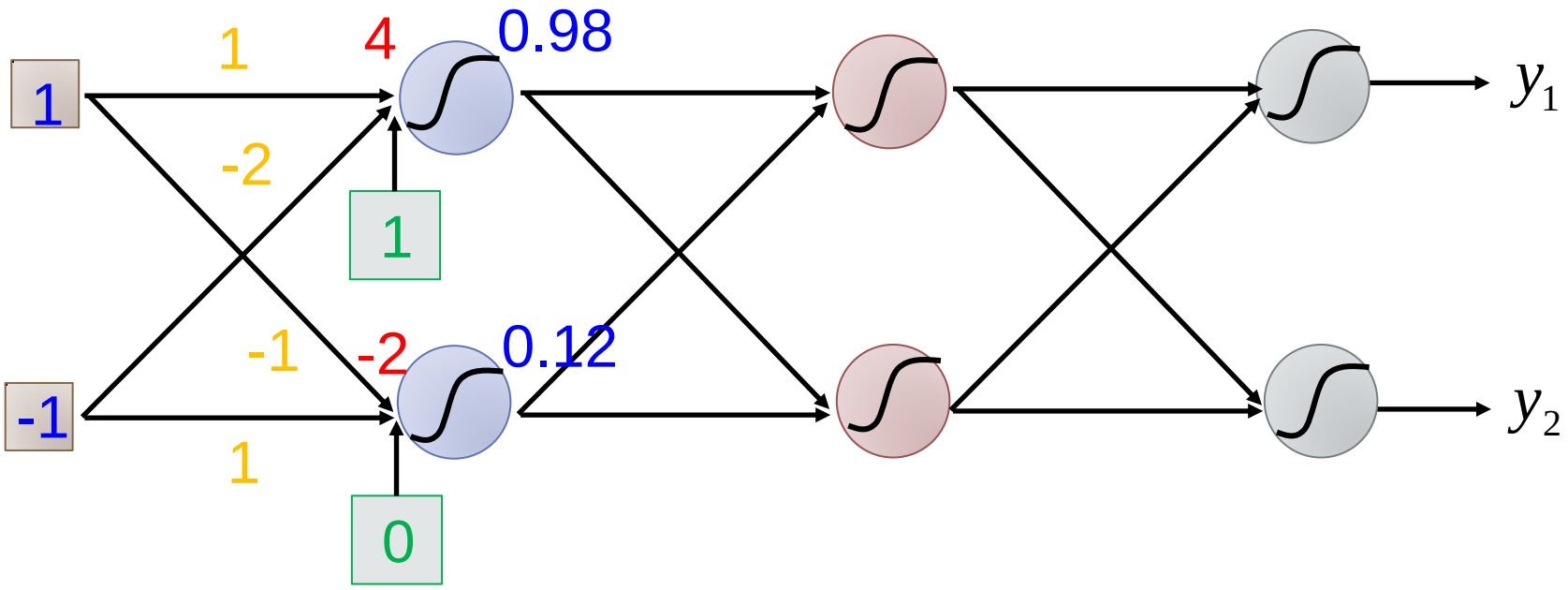


$f: R^2 \rightarrow R^2$

$$f\left(\begin{bmatrix}1\\-1\end{bmatrix}\right) = \begin{bmatrix}0.62\\0.83\end{bmatrix} \quad f\left(\begin{bmatrix}0\\0\end{bmatrix}\right) = \begin{bmatrix}0.51\\0.85\end{bmatrix}$$

Different parameters define different function

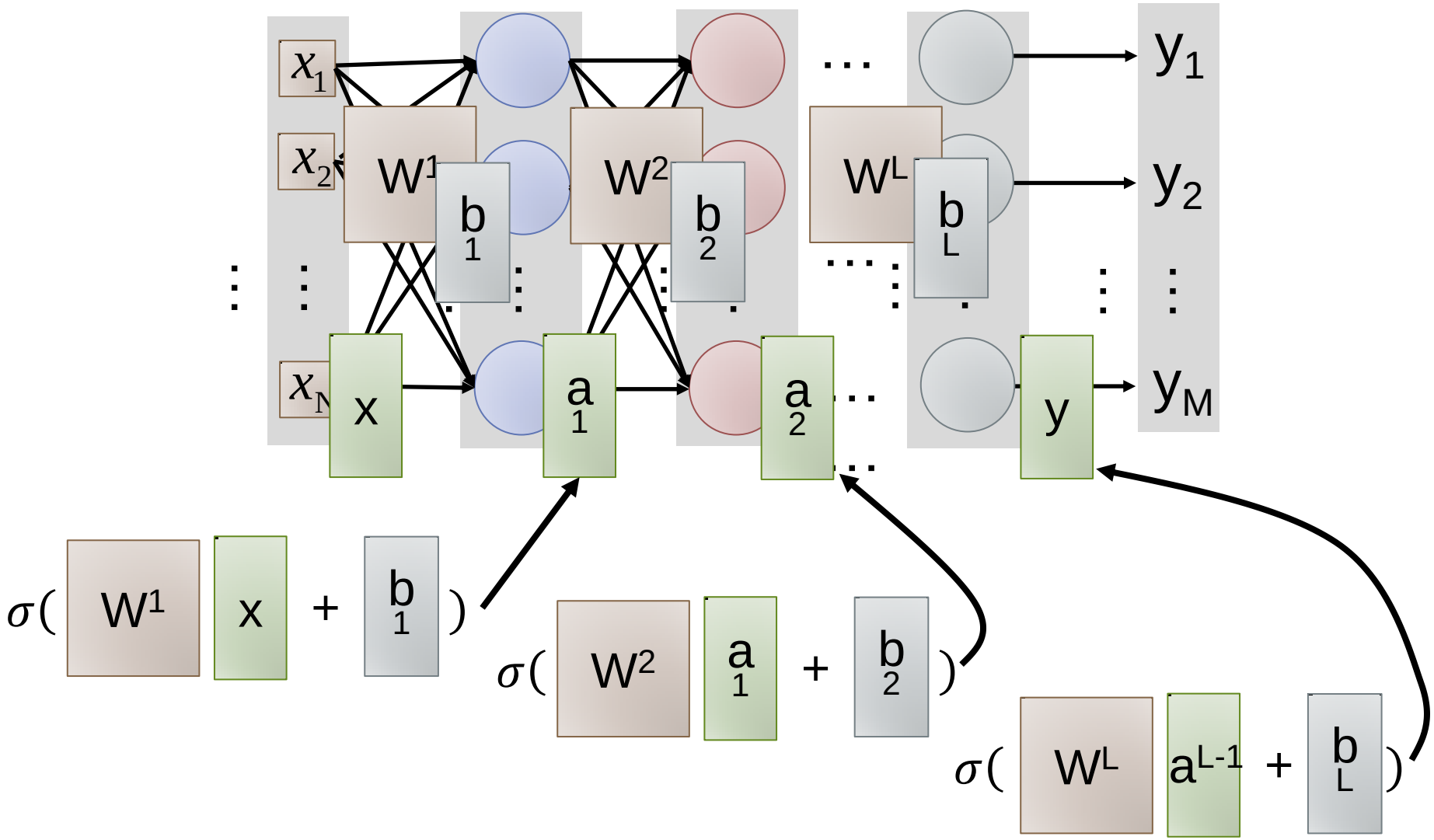
Matrix Operation



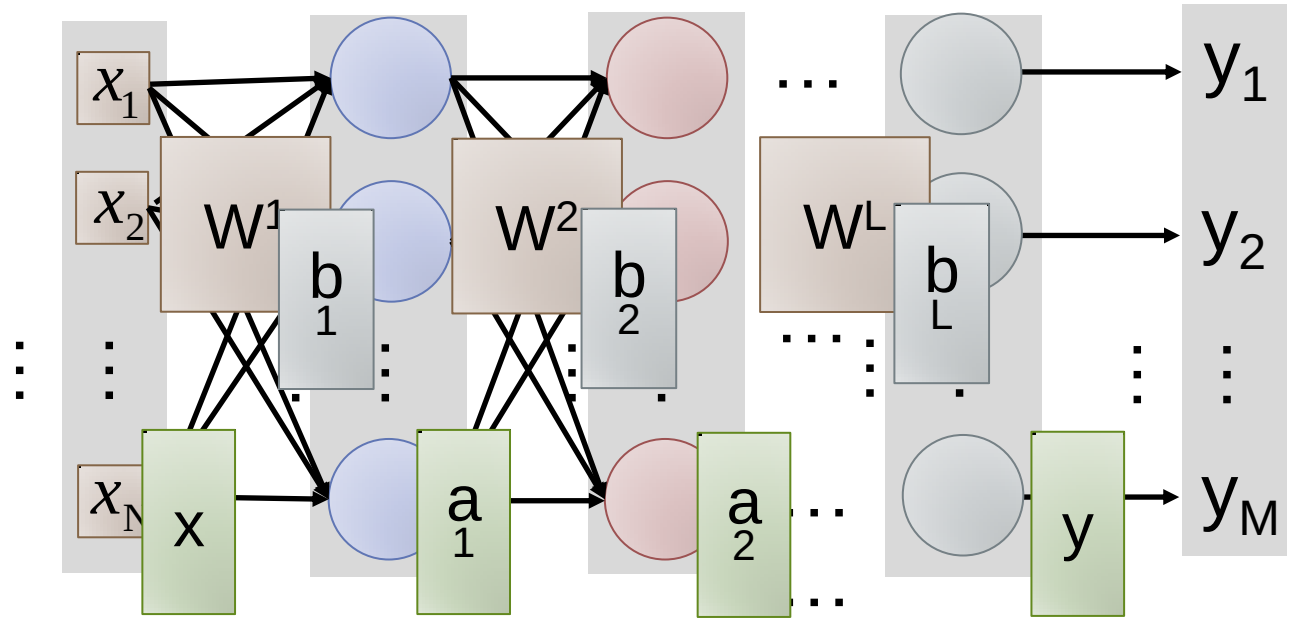
$$\sigma\left(\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

$$\begin{bmatrix} 4 \\ -2 \end{bmatrix}$$

Neural Network



Neural Network



$$y = f(x)$$

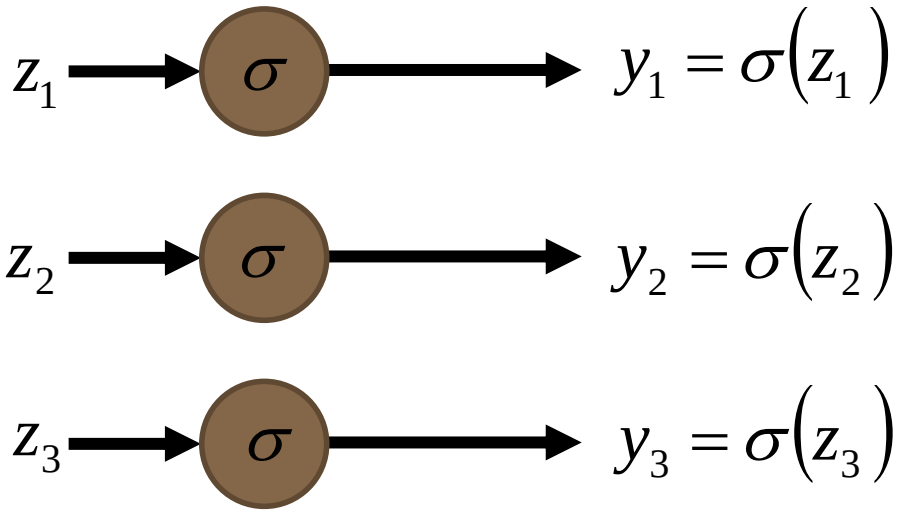
Using parallel computing techniques to speed up matrix operation

$$= \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b_1) + b_2) \cdots + b_L)$$

Softmax

- Softmax layer as the output layer

Ordinary Layer



In general, the output of network can be any value.

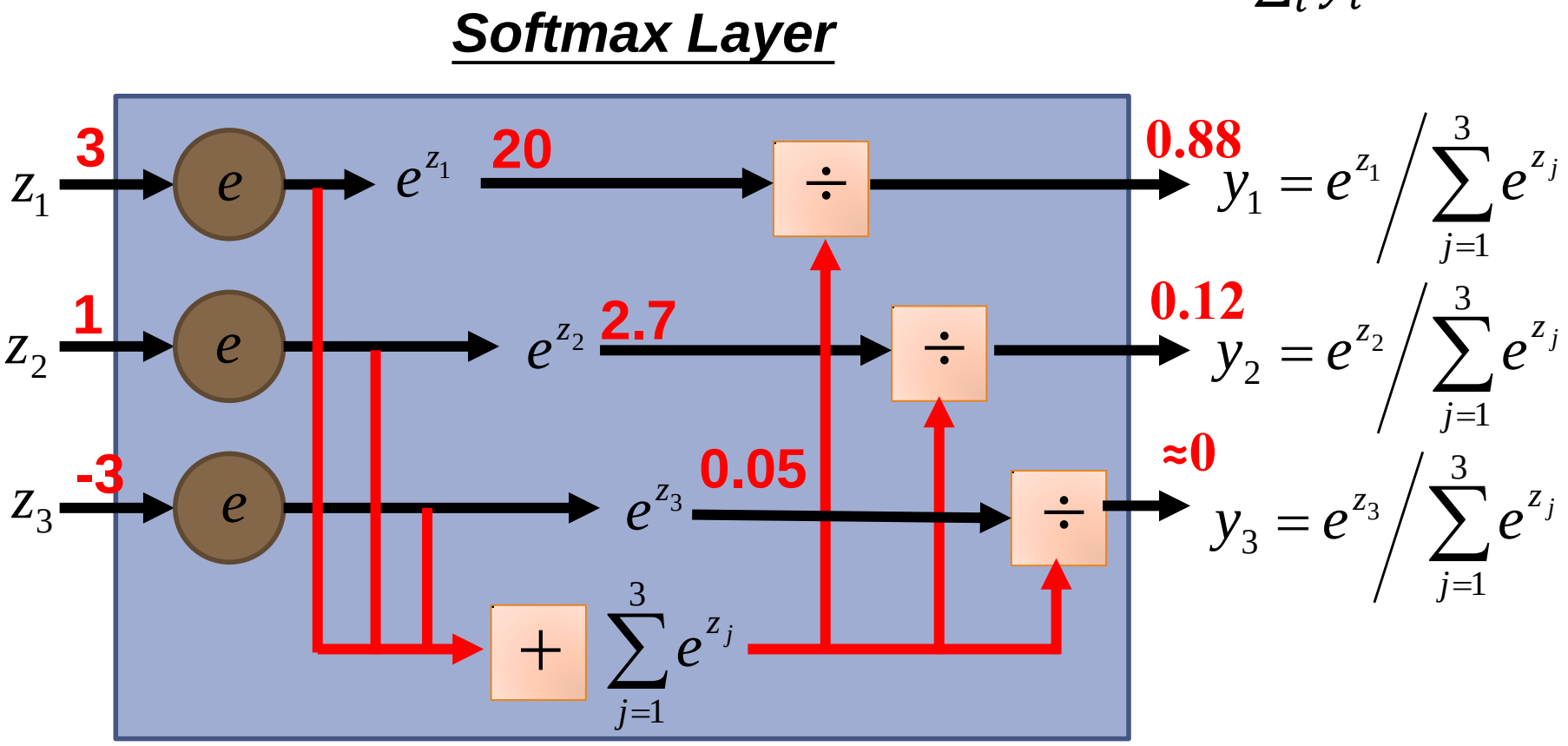
May not be easy to interpret

Softmax

- Softmax layer as the output layer

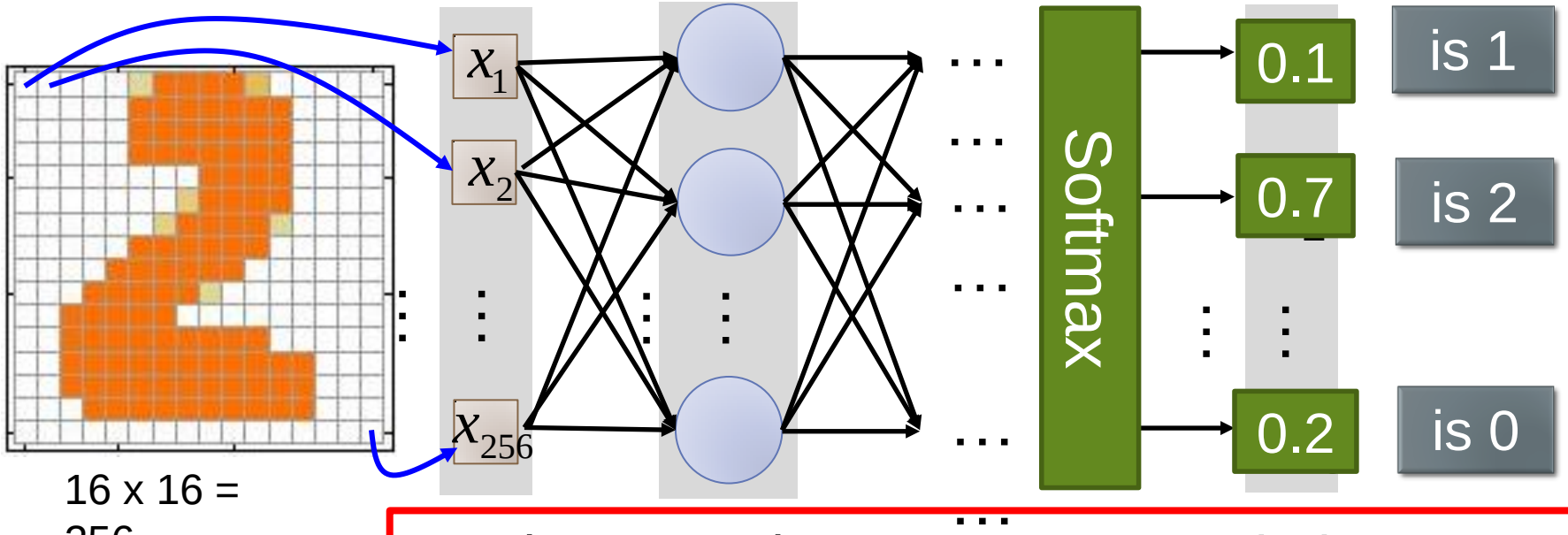
Probability:

- $1 > y_i > 0$
- $\sum_i y_i = 1$



How to set network parameters

$$\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$$



16 x 16 = 256

Ink → 1

No ink → 0

Set the network parameters θ such that ……

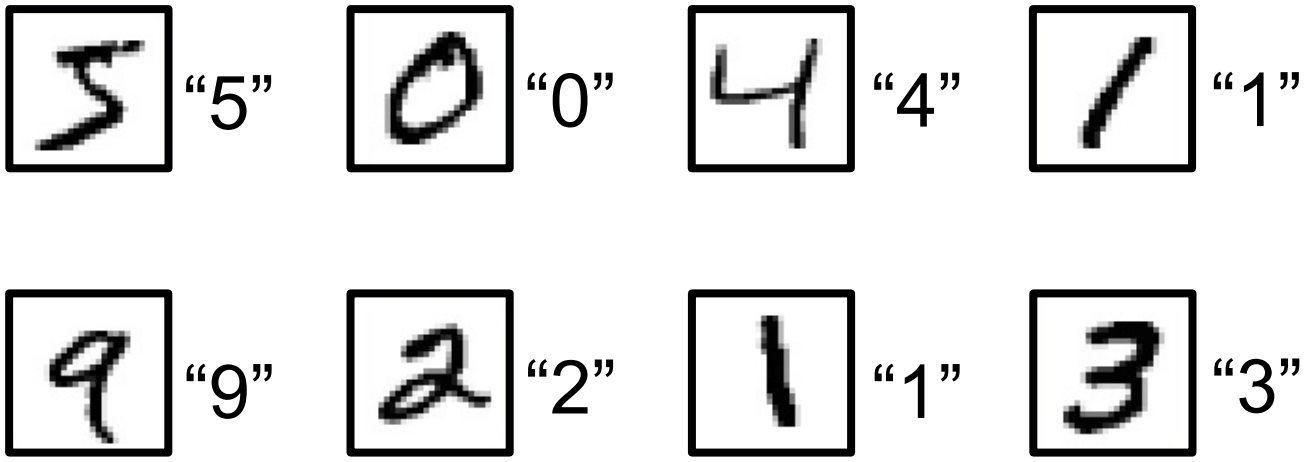
Inpu … um value

Input: … y_2 has the maximum value

How to let the neural network achieve this

Training Data

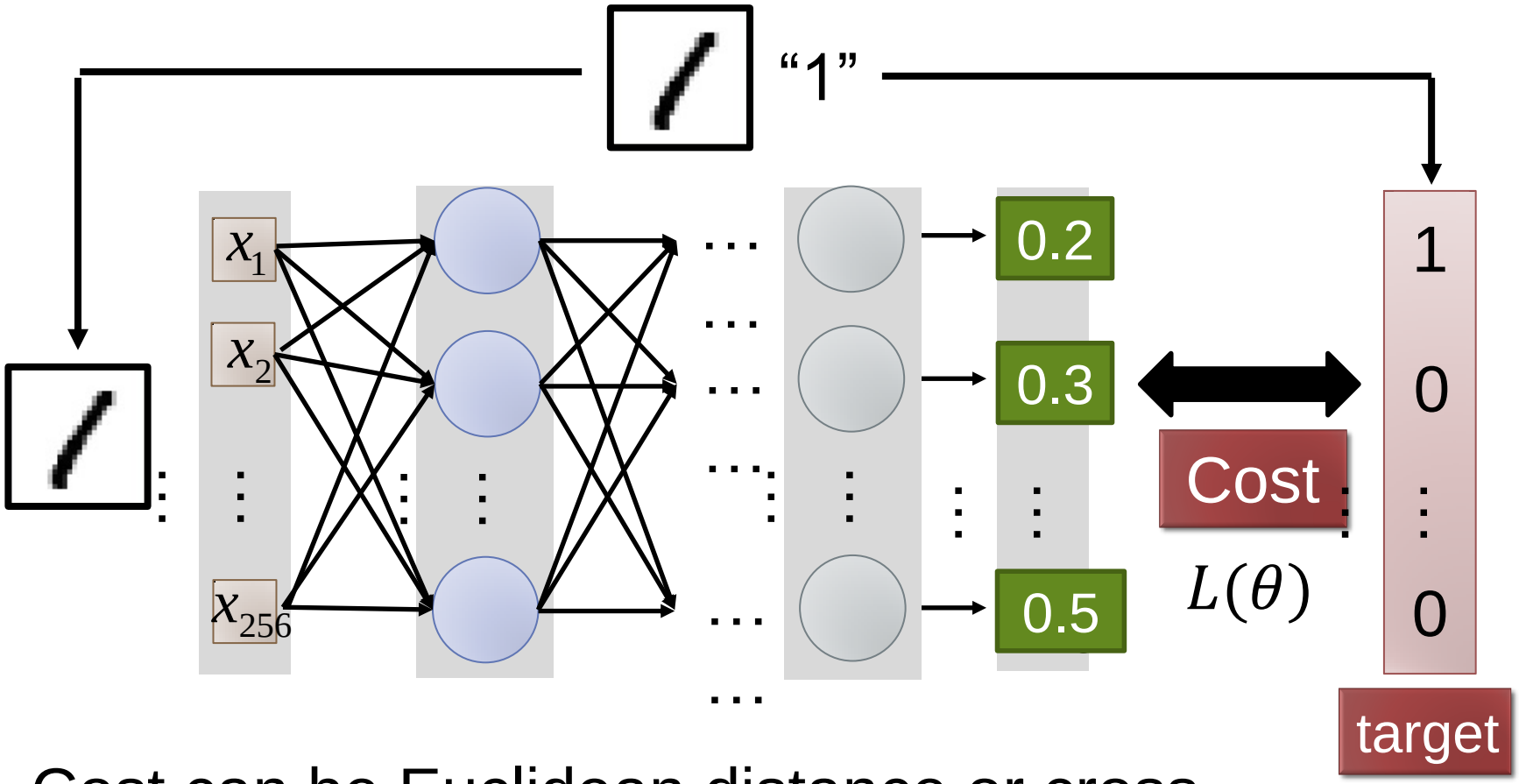
- Preparing training data: images and their labels



Using the training data to find the network parameters.

Cost

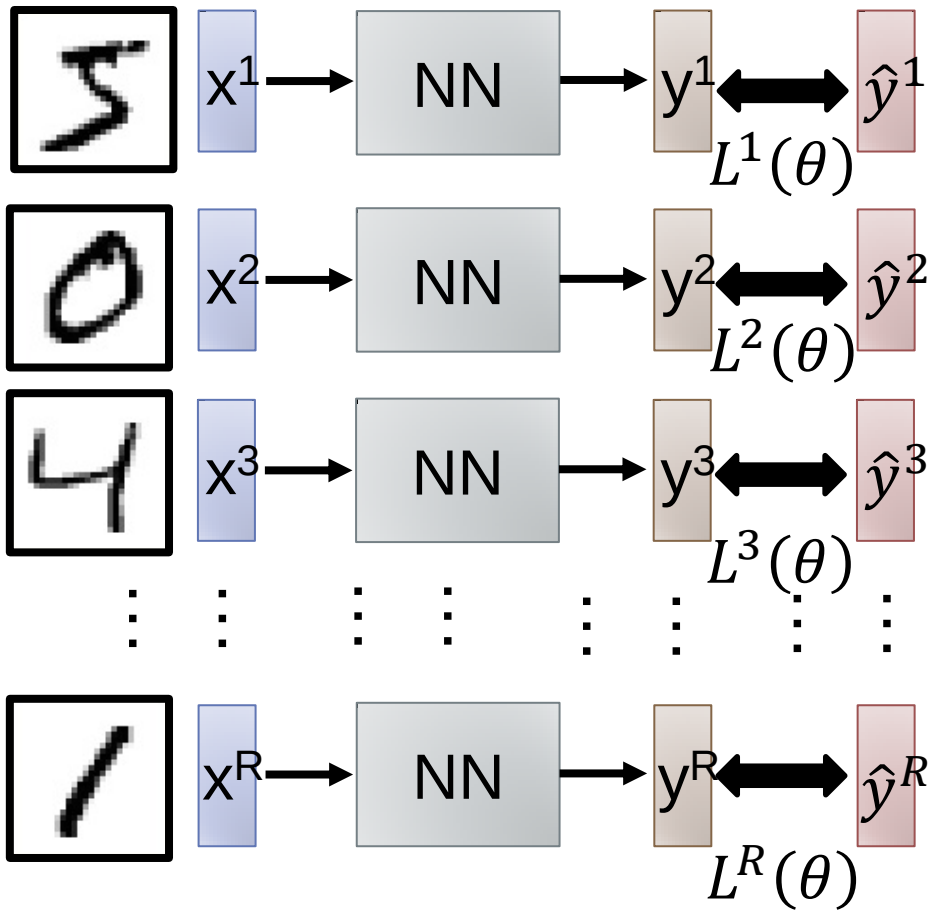
Given a set of network parameters θ, each example has a cost value.



Cost can be Euclidean distance or cross entropy of the network output and target

Total Cost

For all training data …



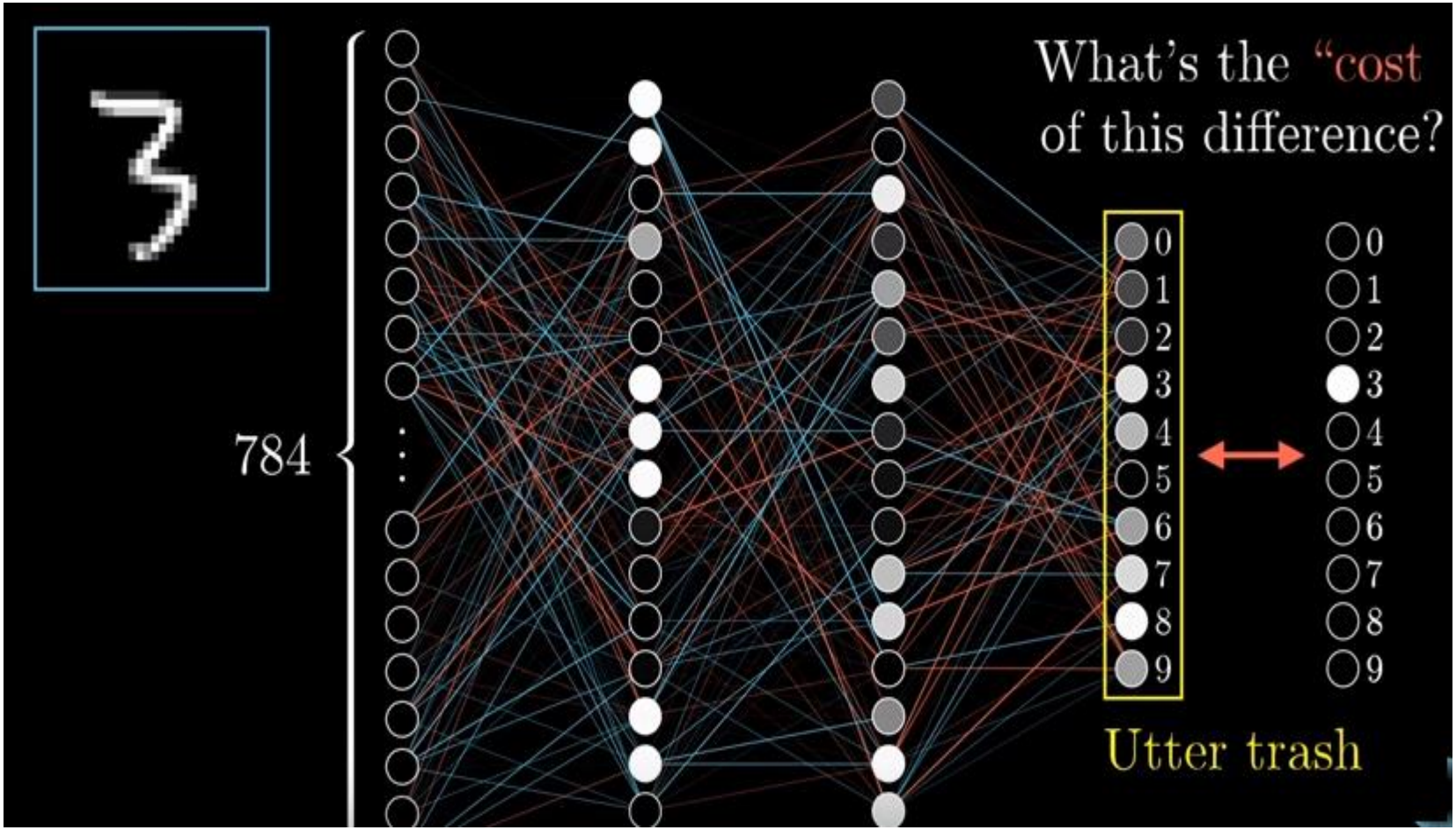
Total Cost:

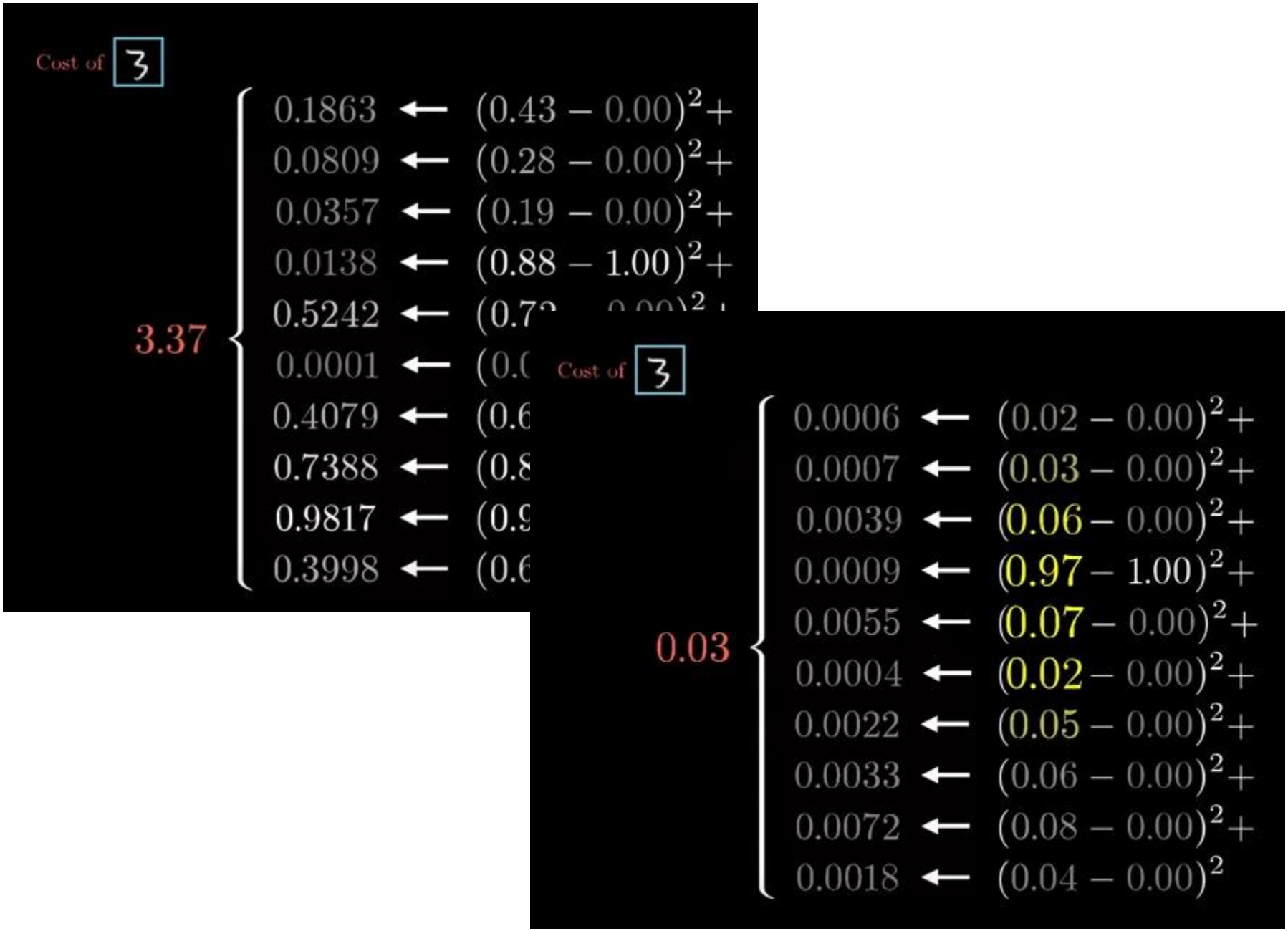
$$C(\theta) = \sum_{r=1}^{R} L^r(\theta)$$

How bad the network parameters θ is on this task

Find the network parameters θ^* that minimize this value

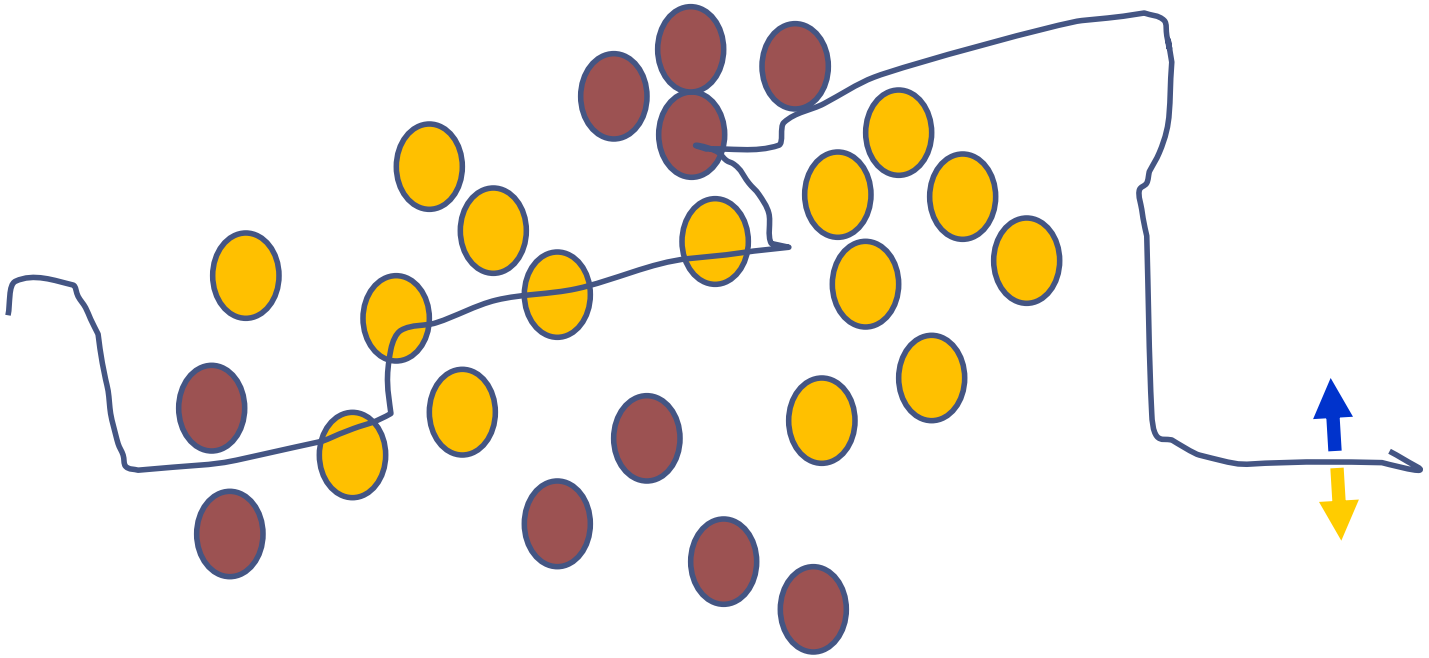
A layered structure?




Cost of 3
0.1863 ← (0.43 − 0.00)² +
0.0809 ← (0.28 − 0.00)² +
0.0357 ← (0.19 − 0.00)² +
0.0138 ← (0.88 − 1.00)² +
0.5242 ← (0.7
3.37
0.0001 ← (0.(
0.4079 ← (0.6
0.7388 ← (0.8
0.9817 ← (0.9
0.3998 ← (0.6
Cost of 3
0.0006 ← (0.02 − 0.00)² +
0.0007 ← (0.03 − 0.00)² +
0.0039 ← (0.06 − 0.00)² +
0.0009 ← (0.97 − 1.00)² +
0.0055 ← (0.07 − 0.00)² +
0.03
0.0004 ← (0.02 − 0.00)² +
0.0022 ← (0.05 − 0.00)² +
0.0033 ← (0.06 − 0.00)² +
0.0072 ← (0.08 − 0.00)² +
0.0018 ← (0.04 − 0.00)²

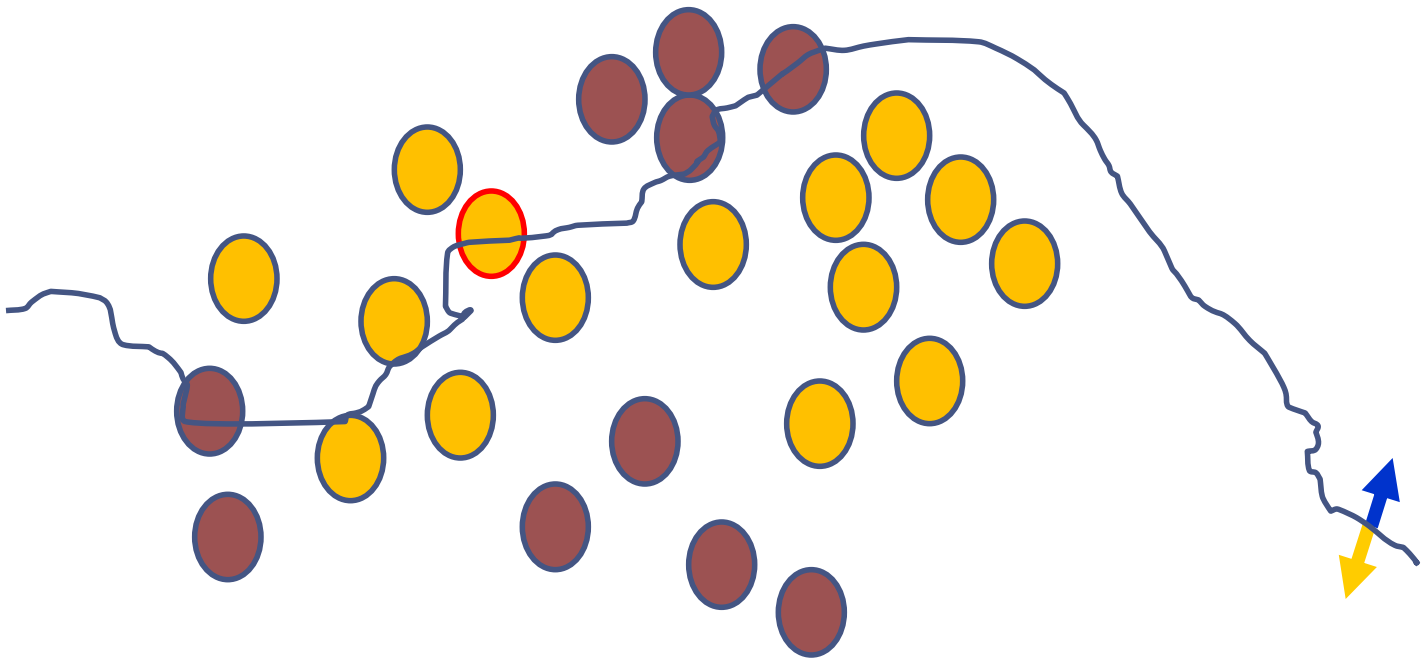
The decision boundary perspective…

Initial random weights



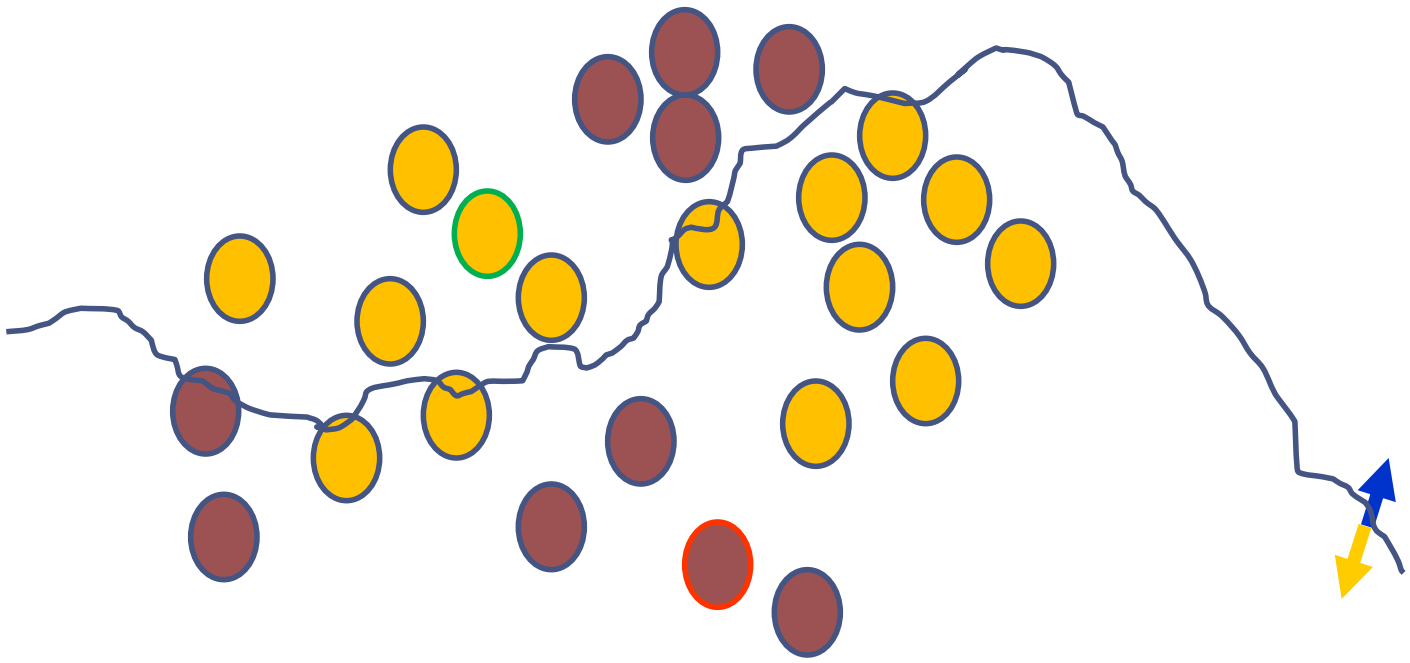
The decision boundary perspective…

Present a training instance / adjust the weights



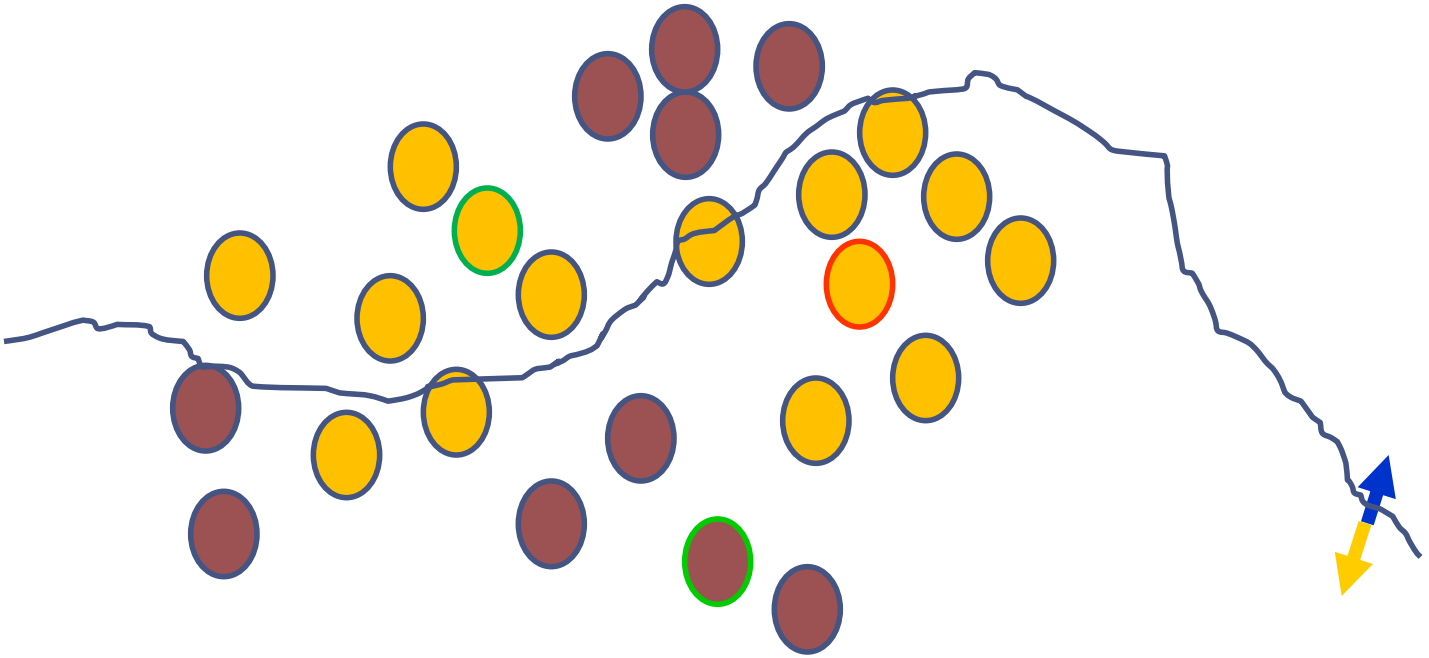
The decision boundary perspective…

Present a training instance / adjust the weights



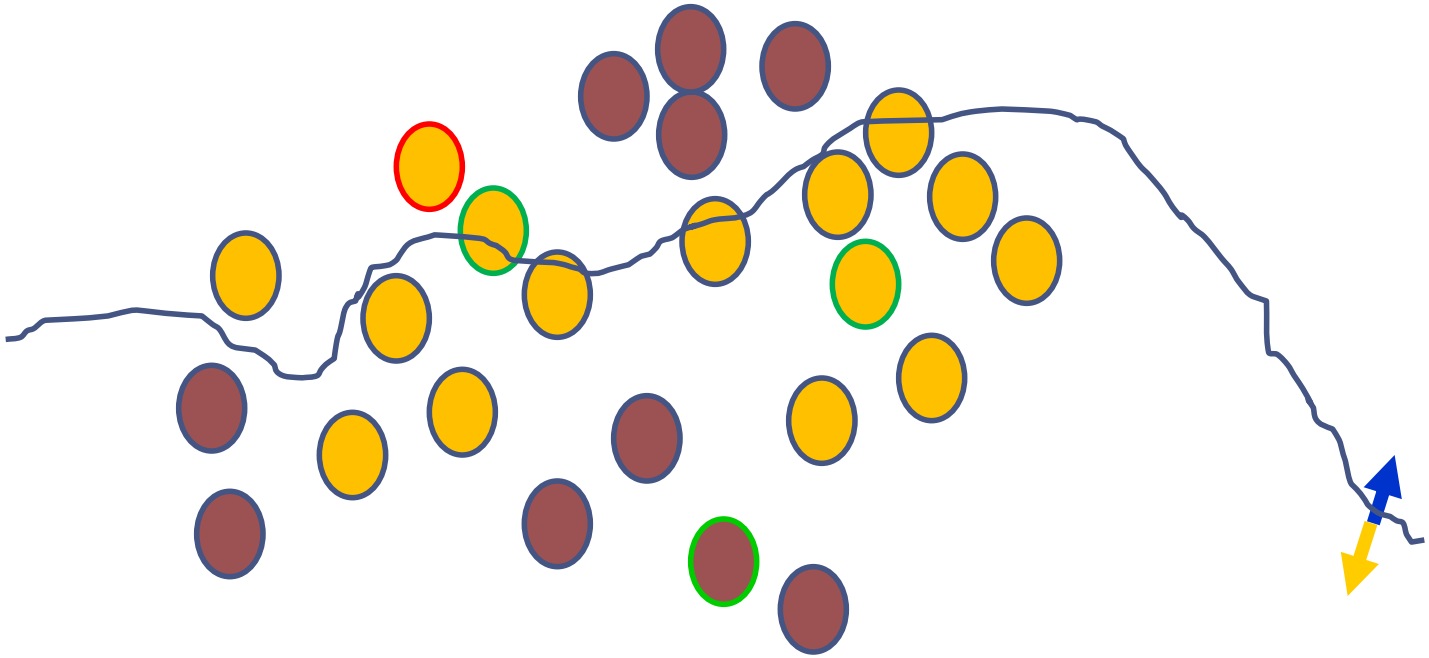
The decision boundary perspective…

Present a training instance / adjust the weights



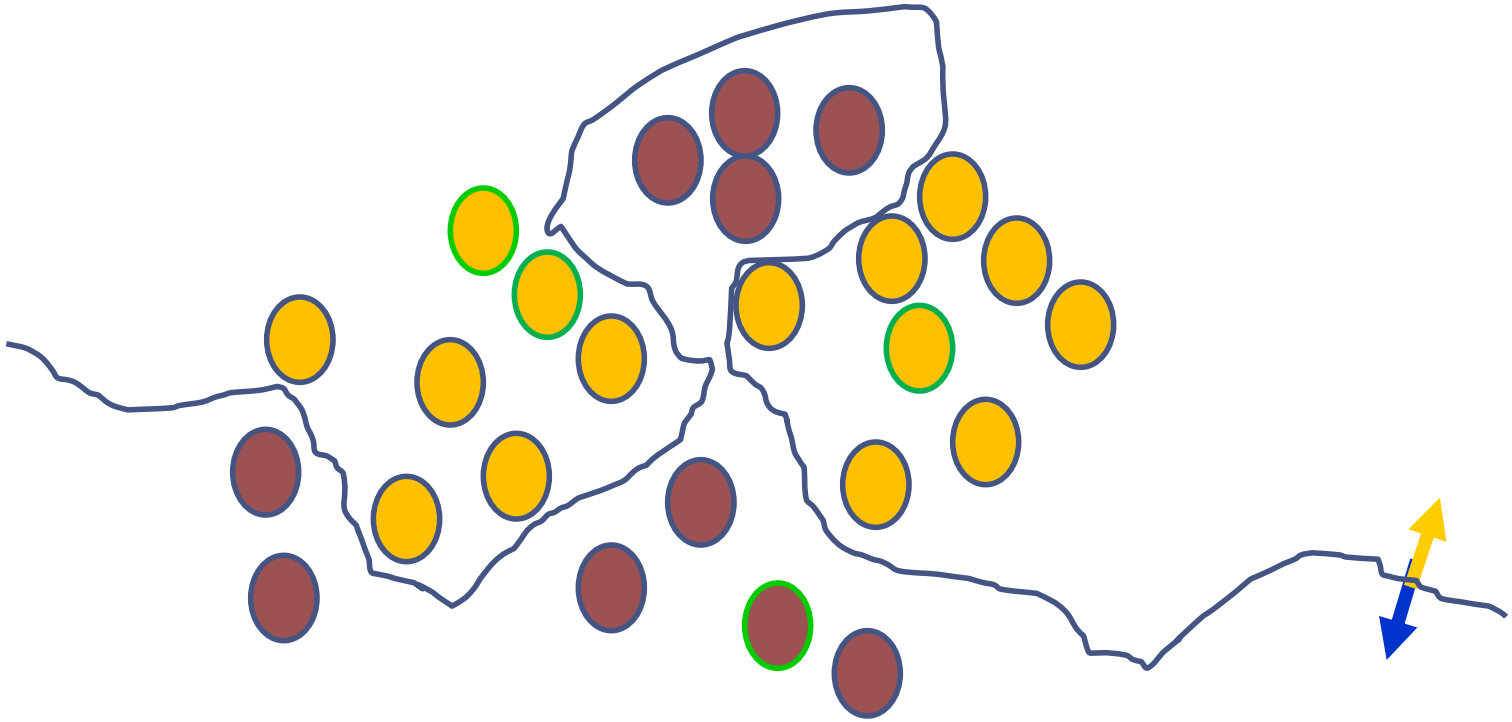
The decision boundary perspective…

Present a training instance / adjust the weights



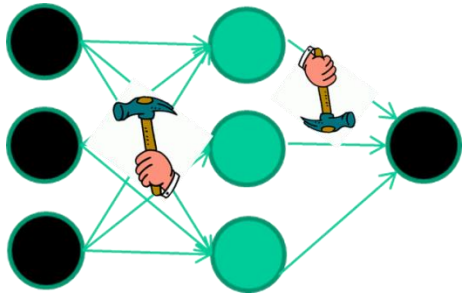
The decision boundary perspective…

Eventually ….



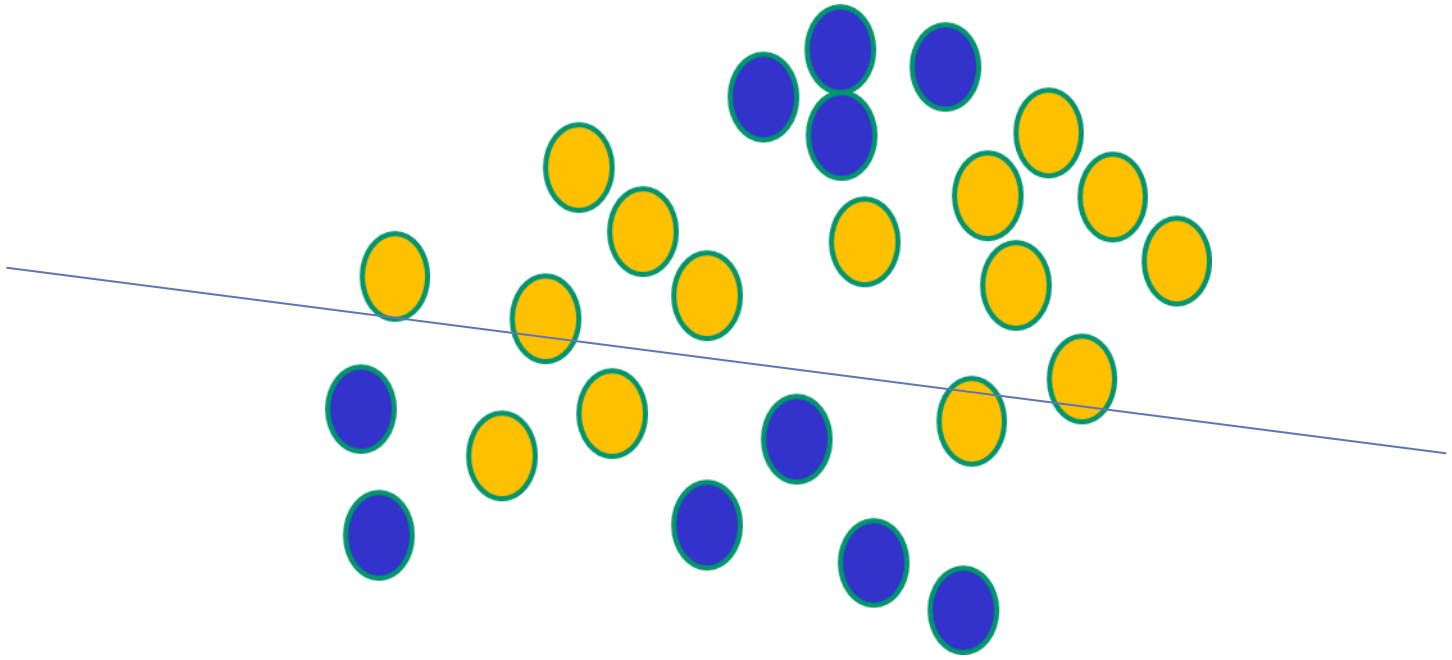
The points to be noted:

- weight-learning algorithms for NNs are dumb
- they work by making thousands and thousands of tiny adjustments, each making the network do better at the most recent pattern, but perhaps a little worse on many others
- but, by dumb luck, eventually this tends to be good enough to learn effective classifiers for many real applications



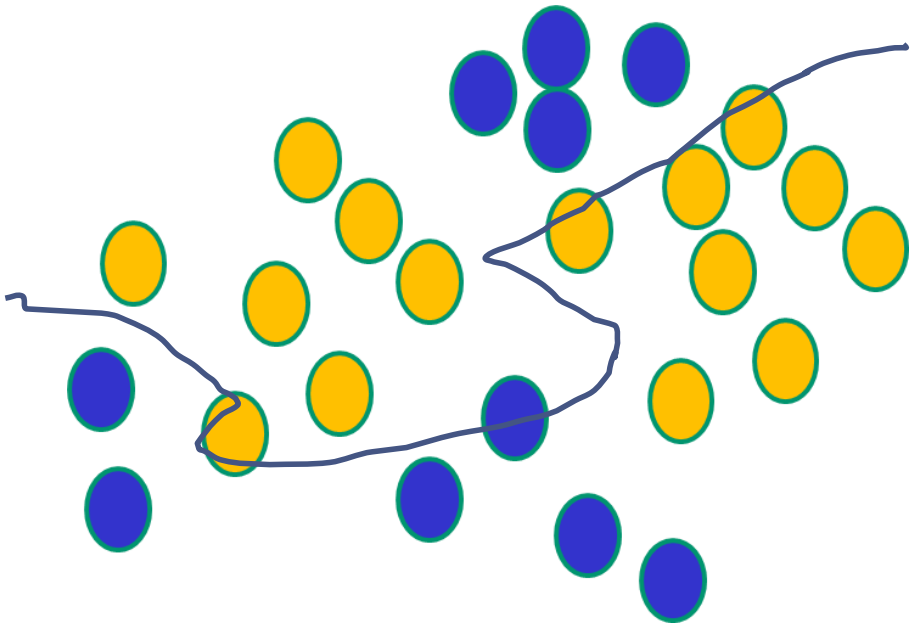
Some other points

If $f(x)$ is linear, the NN can **only** draw straight decision boundaries (even if there are many layers of units)



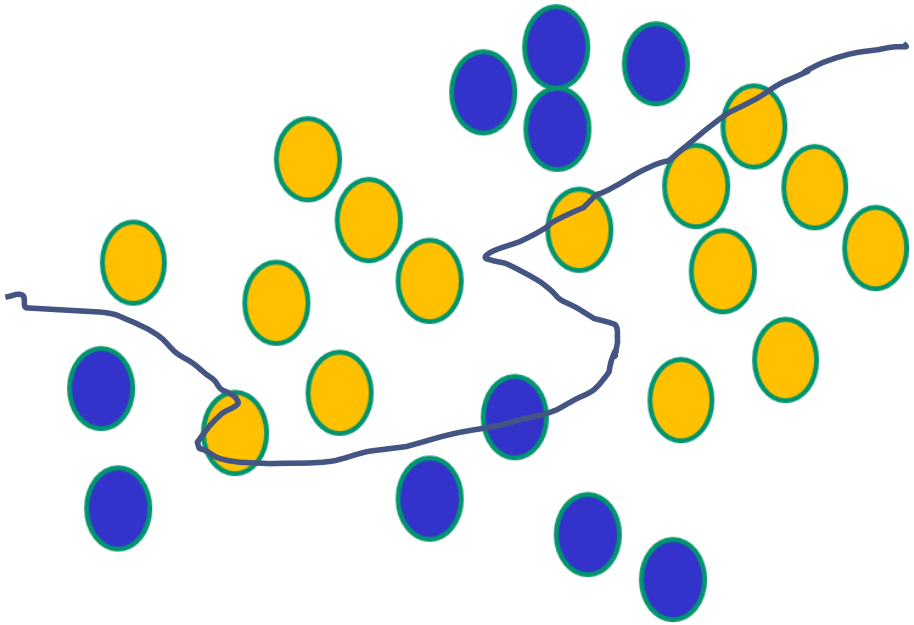
Some other points

NNs use nonlinear *f*(x) so they can draw complex boundaries,but keep the data unchanged

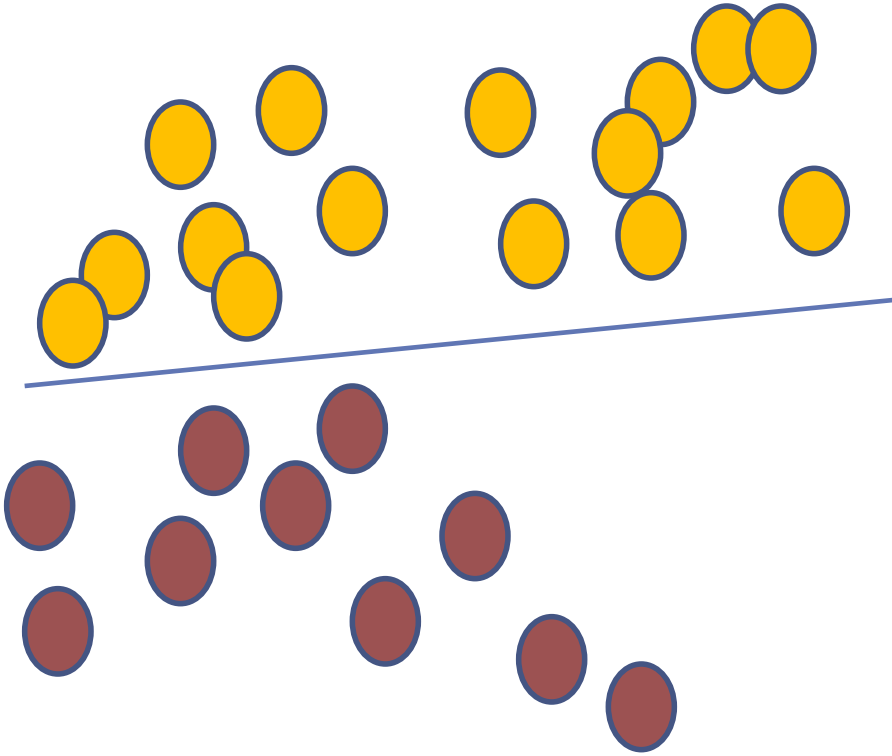


Some other points

NNs use nonlinear *f*(x) so they can draw complex boundaries, first but keep the data unchanged



SVMs only draw straight lines, but they transform the data in a way that makes that OK



Generalised Gradient Descent Learning

Delta learning

- Learning algorithm: same as Perceptron learning except in error calculation.
- Learning takes **gradient descent** approach to reduce E by modify W

$$E = \frac{1}{P}\sum_{p=1}^{P}\frac{1}{2}(t(p) - y\_in(p))^2 \qquad \nabla \boldsymbol{E} = (\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}_1}, ... \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}_n})$$

$$\frac{\partial E}{\partial w_i} = [\frac{1}{P}\sum_{p=1}^{P}(t(p) - y\_in(p))]\frac{\partial}{\partial w_i}(t(p) - y\_in(p)$$

$$= -[\frac{1}{P}\sum_{p=1}^{P}(t(p) - y\_in(p))]x_i$$

$$\Delta \boldsymbol{w_i} \propto -\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w_i}} \quad = [\frac{2}{\boldsymbol{P}}\sum_{1}^{\boldsymbol{P}}(\boldsymbol{t(p)} - \boldsymbol{y\_in(p)})]\boldsymbol{x_i}$$

- **How to apply the delta rule**
  - **Method 1 (sequential mode):** change wi after each training pattern by $\alpha(t(p) - y\_in(p))x_i$
  - **Method 2 (batch mode):** change w_i at the end of each epoch. Within an epoch, cumulate $\alpha(t(p) - y\_in(p))x_i$ for every pattern **(x(p), t(p))**
  - Method 2 is slower but may provide slightly better results (because Method 1 may be sensitive to the sample ordering, and noise)

Backpropagation Algorithm – Main Idea – error in hidden layers

The ideas of the algorithm can be summarized as follows :

1. Computes the **error term for the output units** using the observed error.

2. From output layer, repeat
   - propagating the error term back to the previous layer and
   - updating the weights between the two layers

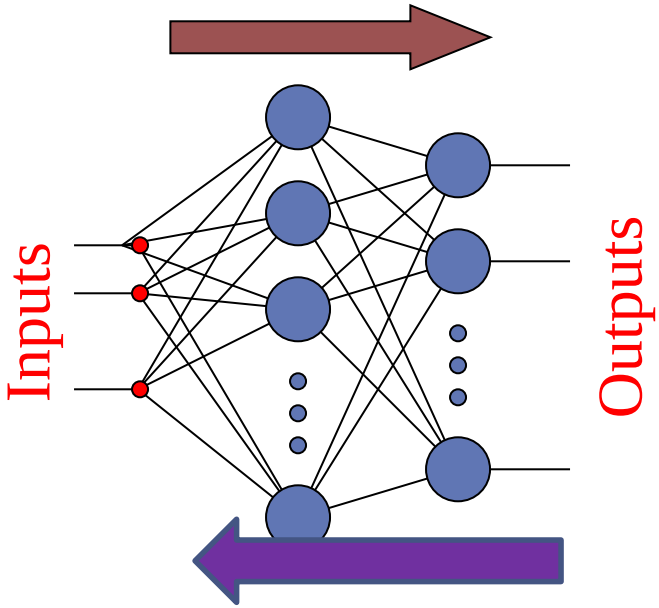
   until the earliest hidden layer is reached.

Backpropagation Learning

- The **Backpropagation** algorithm is an approach for dividing the contribution of each weight while optimizing the goal (produce correct output).
- Basically a modified perceptron rule
- Know as **generalized delta learning**

What we want???

Feedforward Pass (Activation)



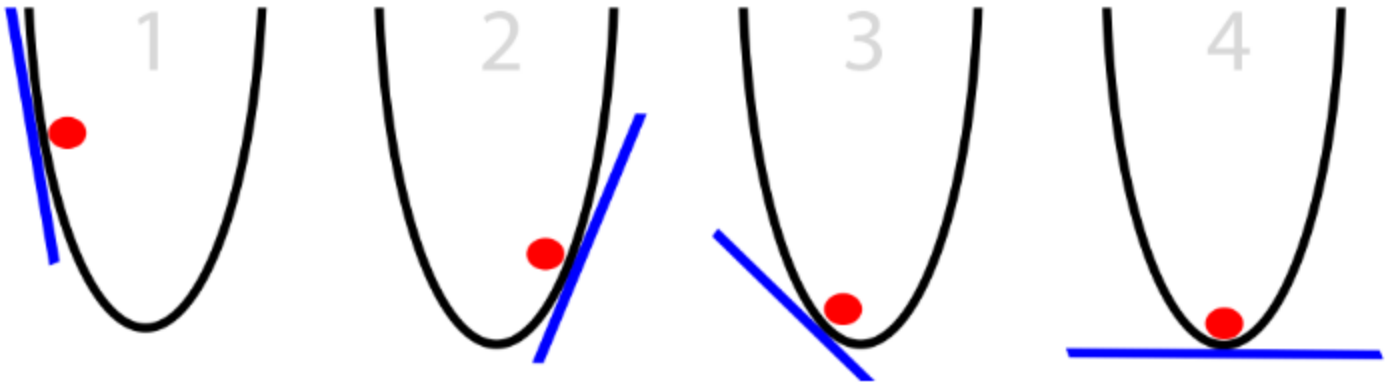
Backward Pass (Error)

Backpropagation Algorithm

- Initialize weights (typically random!)
- Keep doing epochs
  - For each example **e** in training set do
    - **forward pass** to compute
      - Output = *neural-net-output*(network,e)
      - error = (Target-Output) at each output unit
    - **backward pass** to calculate deltas for all weights
    - **update all weights**
  - end
- until tuning set error stops improving

Gradient descent

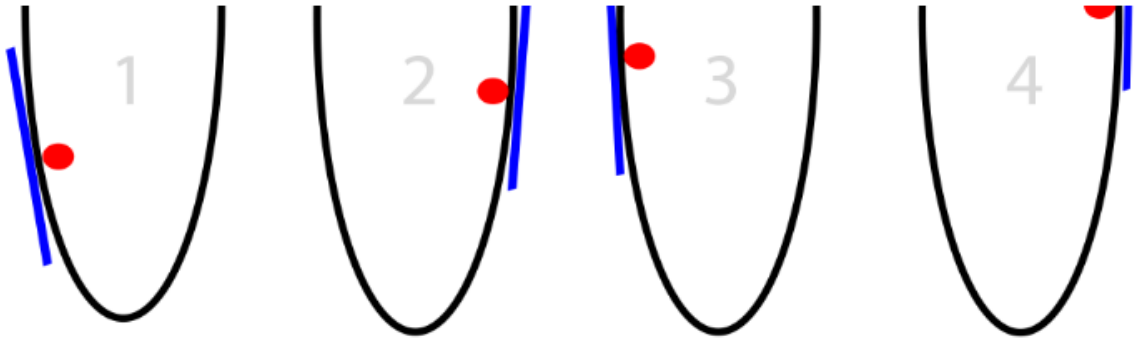
- Optimization: find the lowest point in the bucket



- Calculate slope at current position
- If slope is negative, move right
  (**gradient descent: opposite of gradient**)
- If slope is positive, move left
- (Repeat until slope == 0)

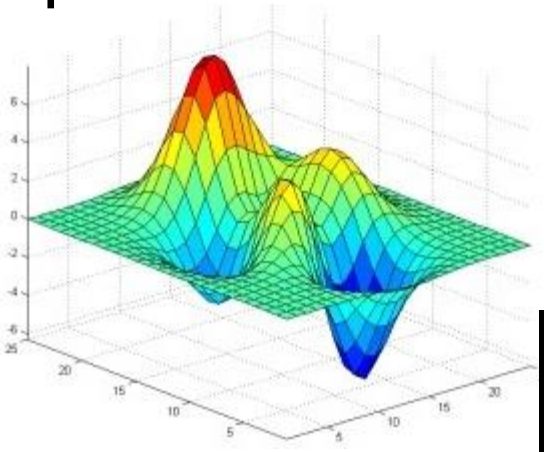
How to move?

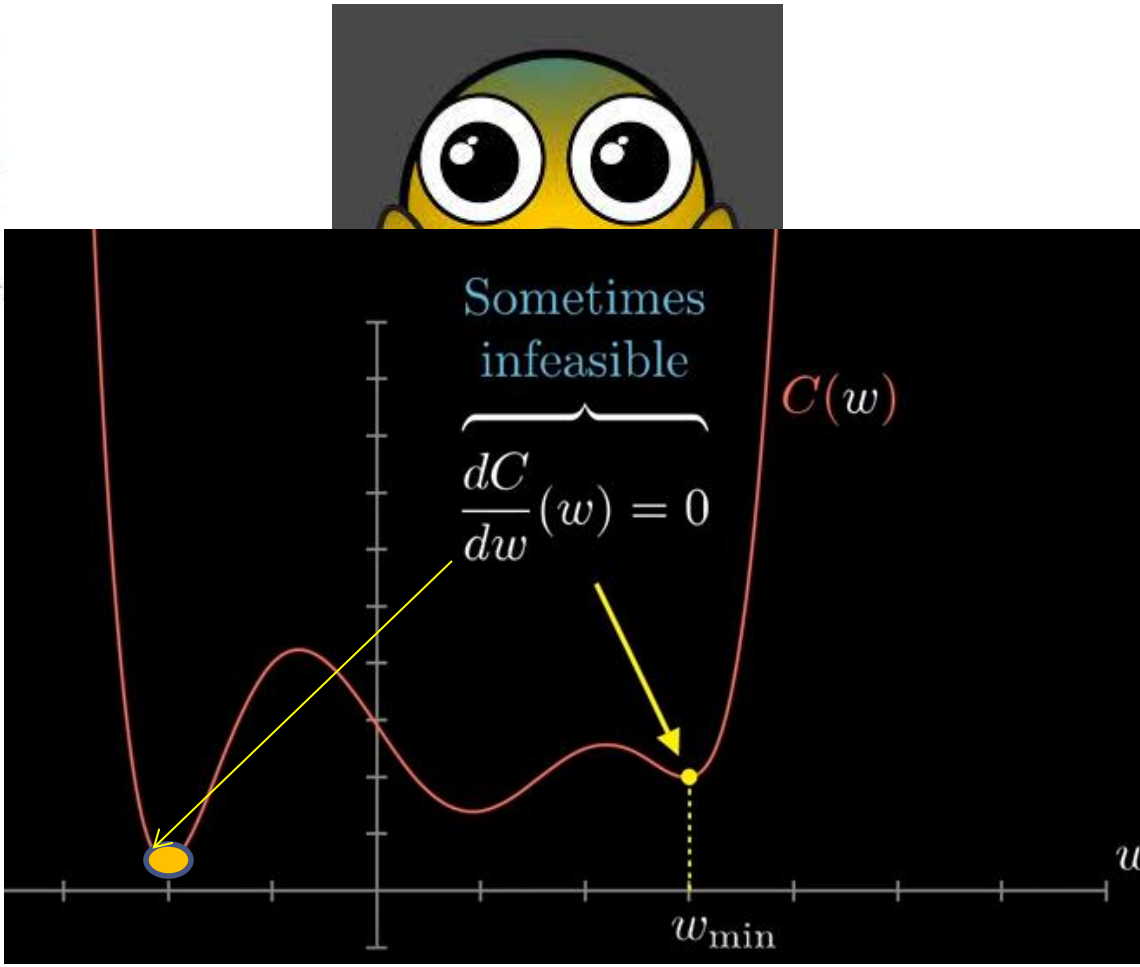
- **How much should the ball move** at each time step?
- The steeper the slope, the farther the ball is from the bottom. That's helpful!
- **Sometimes the slope is so steep that we overshoot by a lot, so that landing is even farther away than the error we started!**



- If our gradients are too big, we make them smaller! We do this by multiplying them (all of them) by a single number between 0 and 1 (such as 0.01). This fraction is typically a single float called **alpha or eta or learning rate**.

Error bucket may be of a funny shape, and following the slope doesn't take you to the absolute lowest point.





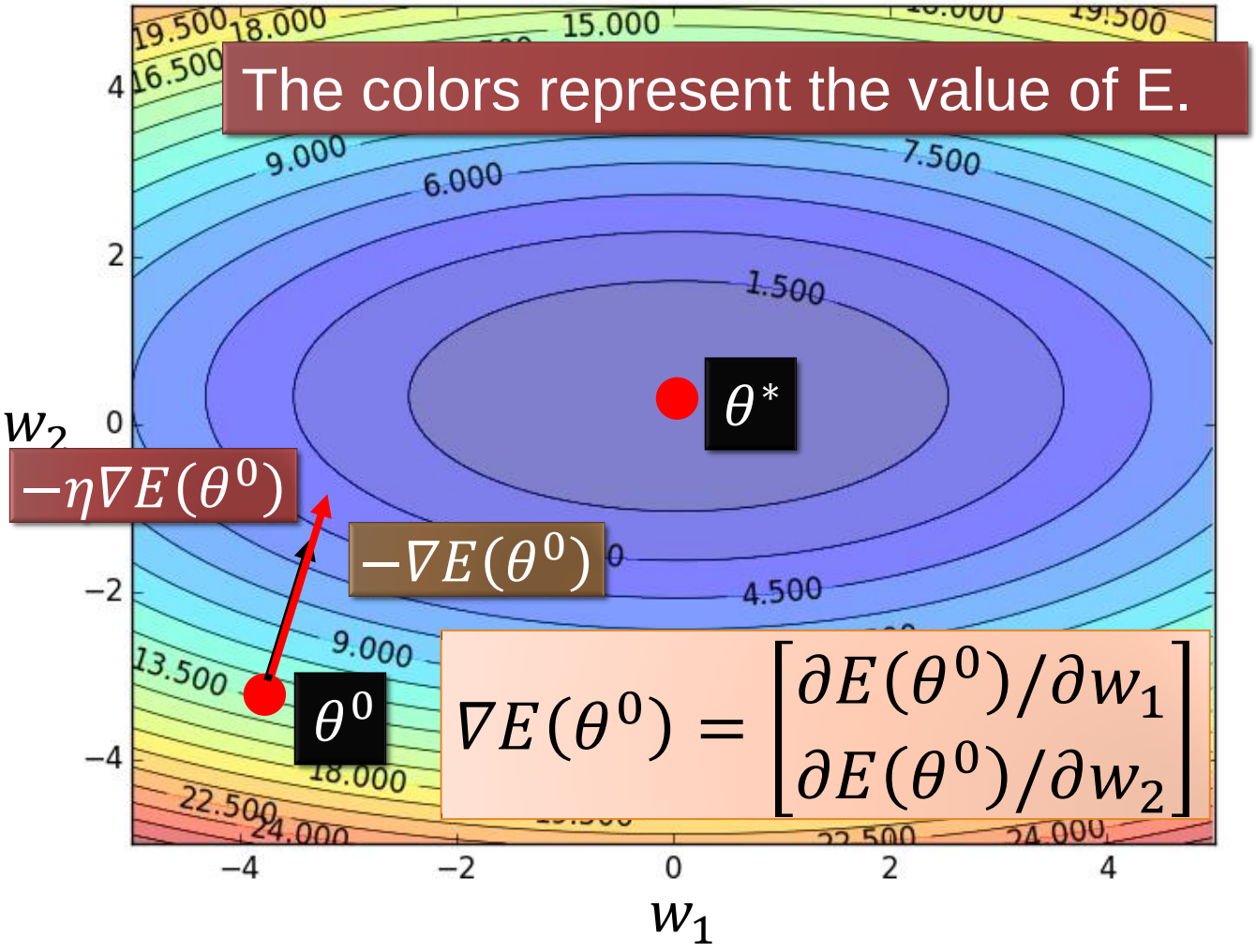
Solutions:

1. Multiple initialization
2. Large # of parameters
3. Momentum

Gradient Descent

Assume there are only two parameters w_1 and w_2 in a network. $\theta = \{w_1, w_2\}$

Error Surface



Randomly pick a starting point θ^0

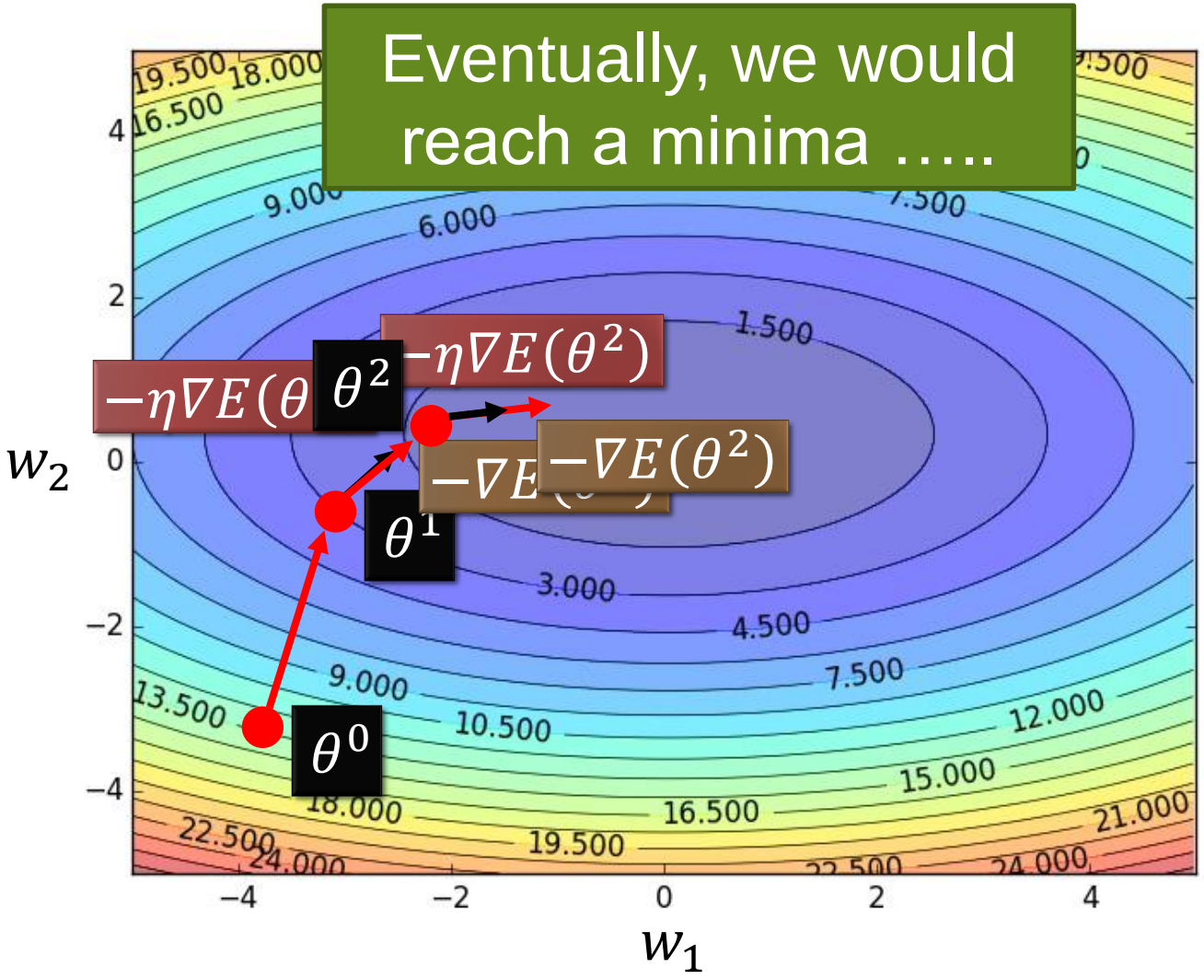
Compute the negative gradient at θ^0

$\Rightarrow -\nabla E(\theta^0)$

Times the learning rate η

$\Rightarrow -\eta\nabla E(\theta^0)$

Gradient Descent



Randomly pick a starting point θ^0

Compute the negative gradient at θ^0

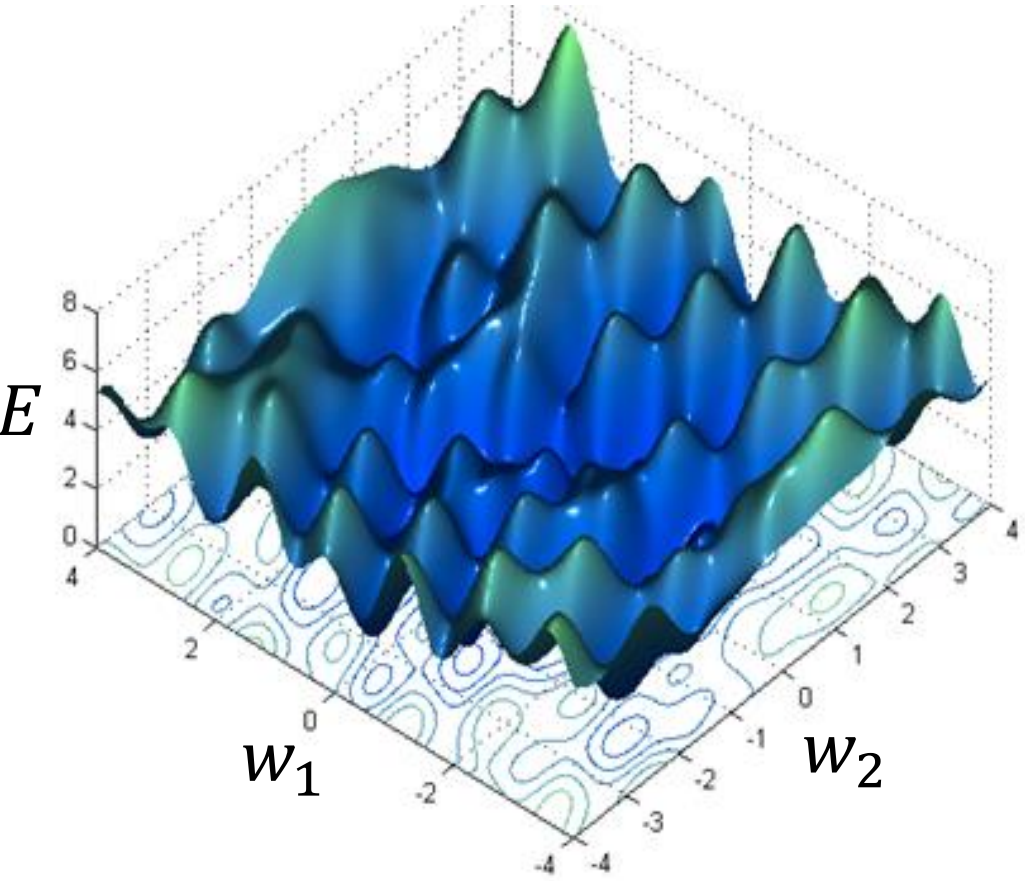
$-\nabla E(\theta^0)$

Times the learning rate η

$-\eta\nabla E(\theta^0)$

Local Minima

- Gradient descent never guarantee global minima

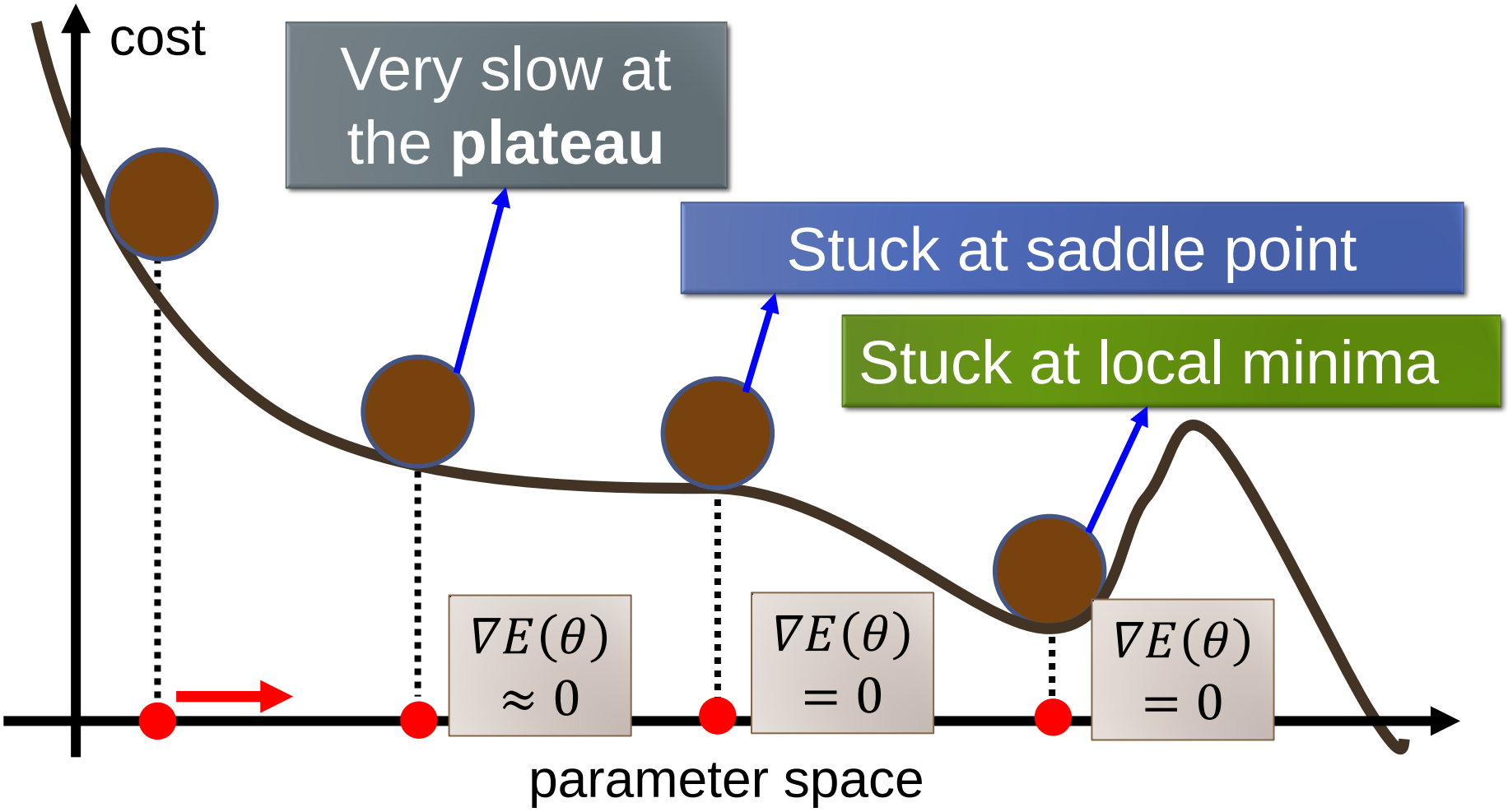


Different initial point θ^0

Reach different minima, so different results

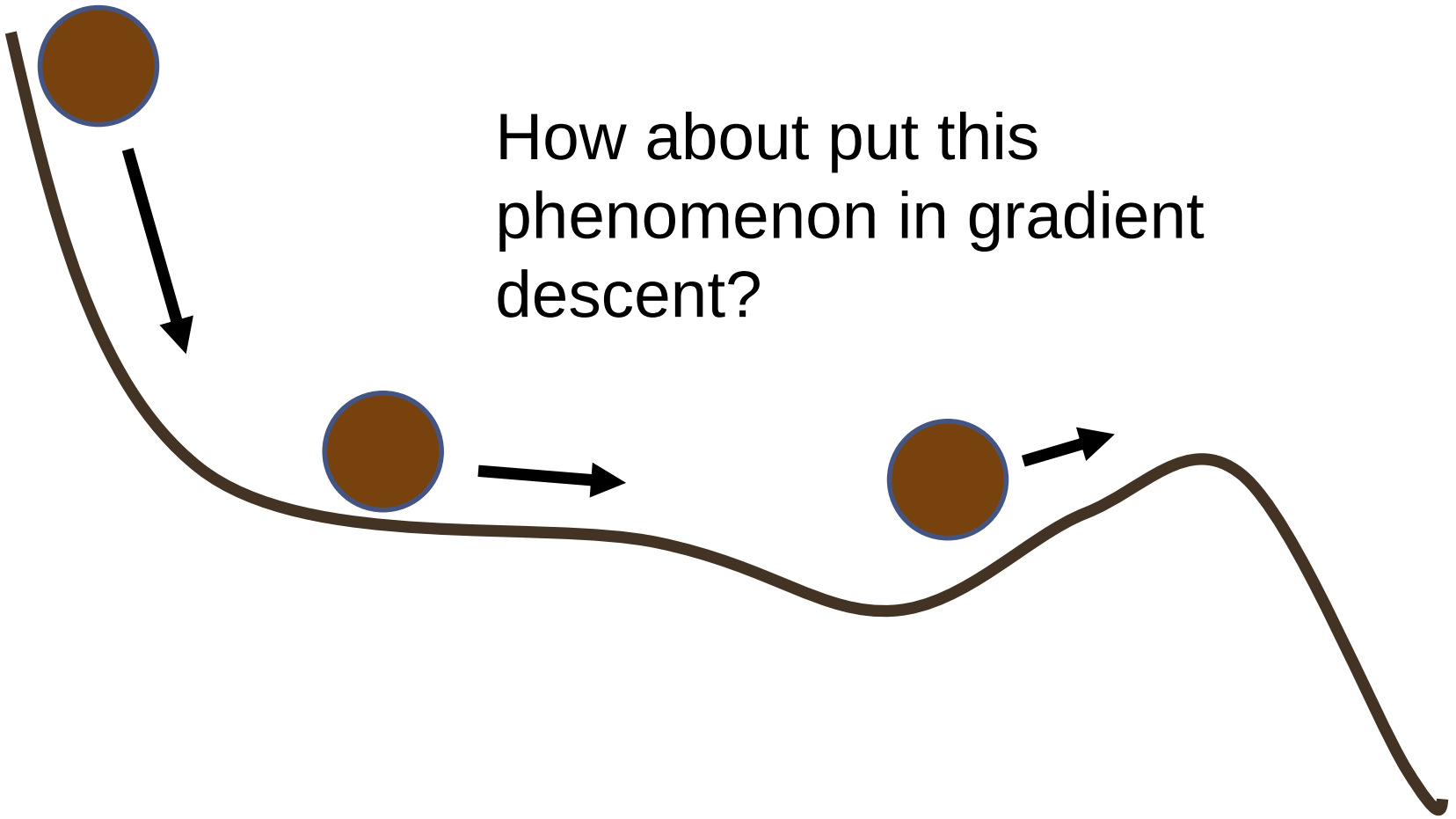
Who is Afraid of Non-Convex Loss Functions?
http://videolectures.net/eml07_lecun_wia/

Besides local minima ……



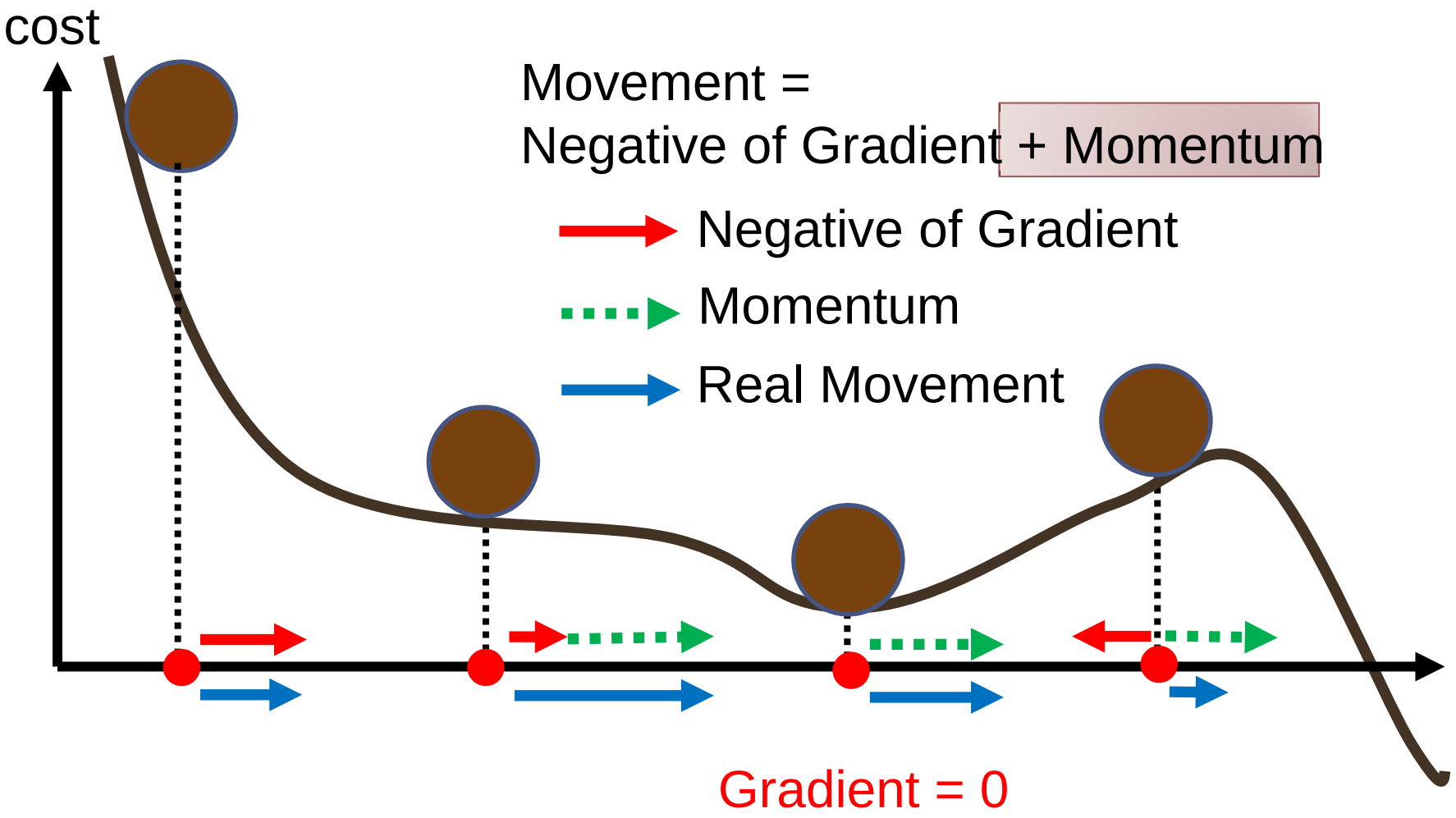
In physical world ……

- Momentum

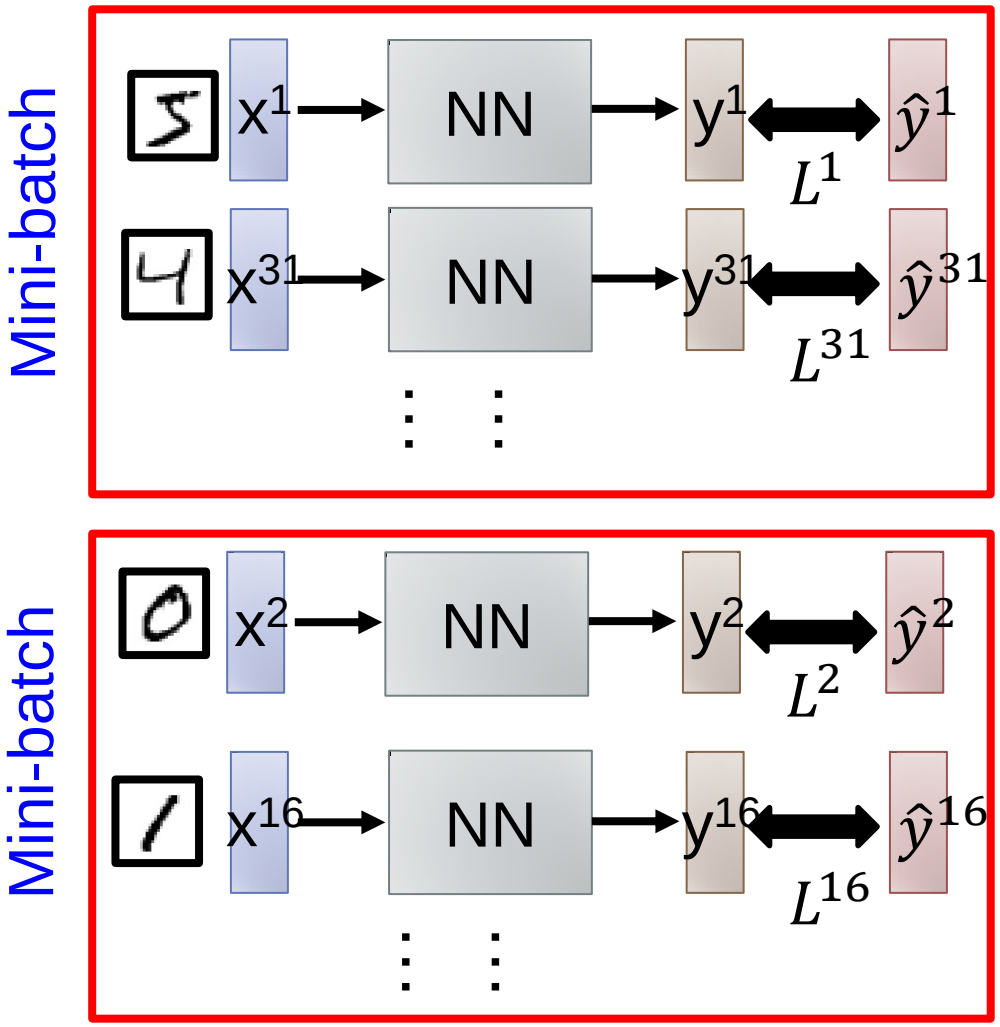


Momentum

Still not guarantee reaching global minima, but give some hope ……



Mini-batch



- Randomly initialize θ^0
- Pick the 1st batch

$E = L^1 + L^{31} + \cdots$

$\theta^1 \leftarrow \theta^0 - \eta \nabla E(\theta^0)$

- Pick the 2nd batch

$E = L^2 + L^{16} + \cdots$

$\theta^2 \leftarrow \theta^1 - \eta \nabla E(\theta^1)$

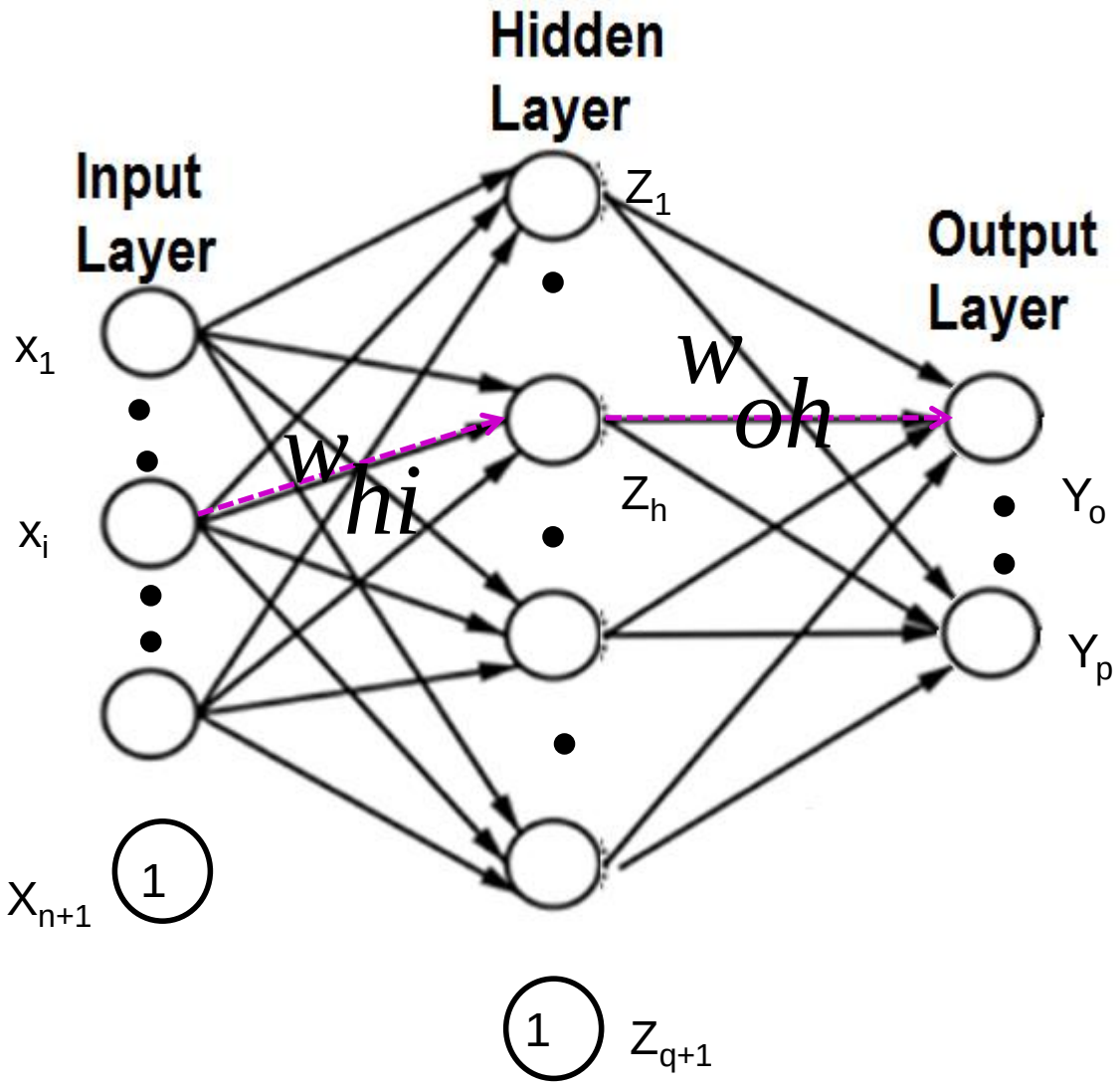
⋮

one epoch

Repeat the above process

BP Learning Equations

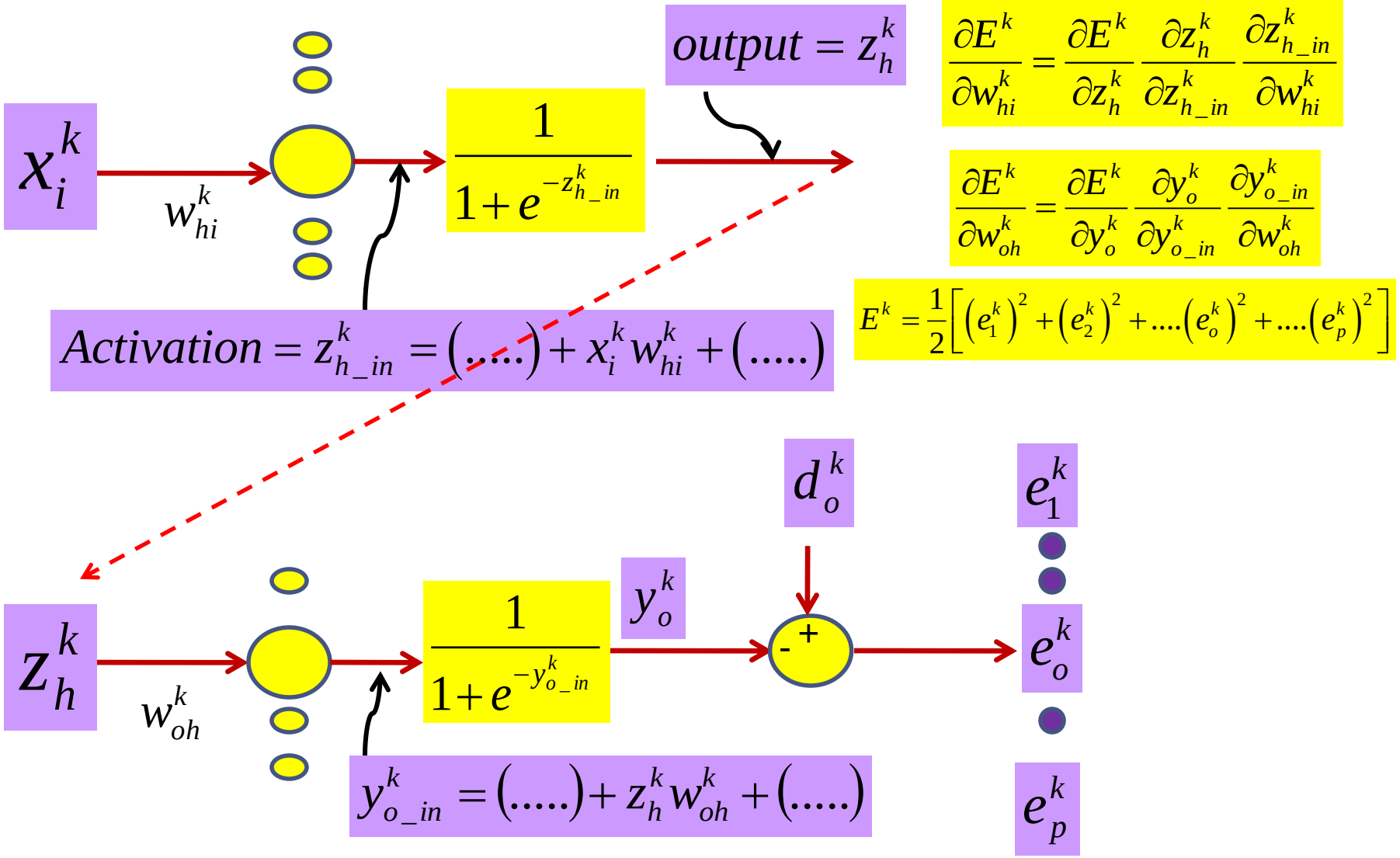
Backpropagation learning equations:



BP algorithm

- **Step 1:** Initialize weights at random, choose a learning rate η.
- **Step 2:** Do forward pass through net (with fixed weights) to produce output(s)
  - Inputs applied
  - Multiplied by weights
  - Summed
  - 'Squashed' by sigmoid activation function
  - Output passed to each neuron in next layer
- **Step 3:**Compute activation signals of input, hidden and output neurons by using procedure of step-2.
- **Step 4:** Compute error over the output neurons by comparing the generated outputs with the desired output.
- **Step 5:** Use the error to compute the change in the **hidden to output layer weights** as well as **input to hidden layer weights**. Update the weights.
- Repeat step-3 to step-5 until network is not trained or stopping condition is met.

Computation of feed forward activations



Hidden to output training

$$E = \frac{1}{2}\left[\left(e_1^k\right)^2 + \left(e_2^k\right)^2 + \ldots\left(e_o^k\right)^2 \ldots + \left(e_p^k\right)^2\right]$$

$$\frac{\partial E^K}{\partial y_o^k} = \frac{\partial\left(t_o^k - y_o^k\right)^2}{\partial y_o^k} = -e_o^k$$

$$\frac{\partial y_o^k}{\partial y_{o\_in}^k} = f'(y_{o\_in}^k) = \frac{\partial\left(\frac{1}{1\_e^{-y_{o\_in}^k}}\right)}{\partial y_{o\_in}^k} = y_o^k(1-y_o^k)$$

$$\frac{\partial E^k}{\partial w_{oh}^k} = \frac{\partial E^k}{\partial y_o^k}\frac{\partial y_o^k}{\partial y_{o\_in}^k}\frac{\partial y_{o\_in}^k}{\partial w_{oh}^k}$$

$$\frac{\partial y_{o\_in}^k}{\partial w_{oh}^k} = \frac{\partial \sum_{o=0}^{q} w_{oh}^k z_h^k}{\partial w_{oh}^k} = z_h^k$$

$$\frac{\partial E^k}{\partial w_{oh}^k} = -e_o^k y_o^k(1-y_o^k).z_h^k = \delta_o^k z_h^k$$

Derivative of sigmoid

$$\delta_o^k = -e_o^k.y_o^k\left(1-y_o^k\right)$$

Delta for output layer neuron

Training of Input ~Hidden

$$\frac{\partial E^k}{\partial w_{hi}^k} = \frac{\partial E^k}{\partial z_h^k}\frac{\partial z_h^k}{\partial z_{h\_in}^k}\frac{\partial z_{h\_in}^k}{\partial w_{hi}^k}$$

$$\frac{\partial E^k}{\partial z_h^k} = \sum_{o=1}^{p}\frac{\partial E^k}{\partial y_o^k}\frac{\partial y_o^k}{\partial z_h^k} = \sum_{o=1}^{p}\frac{\partial E^k}{\partial y_o^k}\frac{\partial y_o^k}{\partial y_{o\_in}^k}\frac{\partial y_{o\_in}^k}{\partial z_h^k} = \sum_{o=1}^{p} -e_o y_o(1-y_o) w_{oh}^k$$

$$= \sum_{o=1}^{p} \delta_o^k w_{oh}^k$$

$$\frac{\partial E^k}{\partial w_{hi}^k} = \sum_{o=1}^{p} \delta_o^k w_{oh}^k \; z_h^k\left(1-z_h^k\right) \; x_i^k = \sum_{o=1}^{p} e_h^k \; z_h^k\left(1-z_h^k\right) x_i^k = \sum_{o=1}^{p} \delta_h^k \; x_i^k$$

$$\sum_{o=1}^{p} \delta_o^k w_{oh}^k = e_h^k$$

Back-propagation

Updating the weights

- For connections between hidden ~ output

$$\frac{\partial E^k}{\partial w_{oh}^k} = \delta_o^k \quad z_h^k$$

$$w_{oh}^{k+1} = w_{oh}^k + \Delta w_{oh}^k = w_{oh}^k + \eta\left(\frac{\partial E^k}{\partial w_{oh}^k}\right) = w_{oh}^k + \eta\delta_o^k z_h^k$$

- For connections between input ~ hidden layer

$$\frac{\partial E^k}{\partial w_{hi}^k} = \sum_{o=1}^{p} \delta_h^k \quad x_i^k$$

$$w_{hi}^{k+1} = w_{hi}^k + \Delta w_{hi}^k = w_{hi}^k + \eta\left(\frac{\partial E^k}{\partial w_{hi}^k}\right) = w_{hi}^k + \eta\delta_h^k x_i^k$$

Adaptivity and generalization of NN

- NNs are used for classification and function approximation or mapping problems which are:

- Tolerant of some imprecision.

- Have lots of training data available.

- Hard and fast rules cannot easily be applied.

Improvement in Back-propagation

- Impact of initial weight and bias:(common practice: select random values in range -1 to +1)
- Improper weights may drive the learning to stuck up at local minimum
- Weight updating is a function of presynaptic and post synaptic activation(derivative). It is important to avoid choice if weights which is likely to make the activation zero
- Inputs or weights should not be too large or too small. This may result activations falling in saturation zone of sigmoid (resulting in meaning less derivative) , not zero.
- How long to learn?(training – validation-testing)-(60%-20%-20%)
- Normally, you are training the network with the training set to adjust the weights. To make sure you don't over fit the network and also fine-tune models you need to input the validation set to the network and check if the error is within some range
- Training set is used to fit the parameters
- Validation set is used for tuning the parameters of a model.
- Test set is used for performance evaluation.
- If the accuracy over the training data set increases, but the accuracy over then validation data set stays the same or decreases, indicates over fitting of NN and we should stop training.

1. How many hidden layers ? How many neurons?

- data that is linearly separable don't need any hidden layers.
- One or two hidden layer is sufficient for the large majority of problems.
- There are some empirically-derived rules-of-thumb, the most commonly relied on is '*the optimal size of the hidden layer is usually close to mean of number of nodes in input and output layers*'.
- Additional rule of thumb:

$$N_{hidden} = \frac{No\ of\ training\ samples}{2 \times \left(N_{input} + N_{output} \right)}$$

2. How many training patterns?: If there are enough training patterns net will be able to generalize.

$$P = \frac{\text{no of weights to be trained}}{\text{expected minimum error}}$$

Impact of learning rate

Error Signal Attenuation:

- The error signal δ gets attenuated as it moves backward through multiple layers.

- So different layers learn at different rates. Input-to-hidden weights learn more slowly than hidden-to-output weights

- Solution: have different learning rates δ for different layers.

High Learning (large η): Weights can oscillate

Slow Learning: if the learning rate is set too low

-Solution: add a momentum term α.

$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}(t)} + \alpha \cdot \Delta w_{ij}(t-1)$$

If the direction of the gradient remains constant, the algorithm will take increasingly large steps, without incorporating oscillation. Alternatively one can consider an adaptive learning rate.

The Problem with Large Networks

- Vanishing gradients: as we add more and more hidden layers, backpropagation becomes less and less useful in passing information to the lower layers. In effect, as information is passed back, the gradients begin to vanish and become small relative to the weights of the networks.
- Overfitting: perhaps the central problem in machine learning. Briefly, overfitting describes the phenomenon of fitting the training data *too*closely, maybe with hypotheses that are *too* complex. In such a case, your learner ends up fitting the training data really well, but will perform much, much more poorly on real examples.

NN in practice

- **NETalk** : to pronounce each letter in a word in a sentence
- Speech Recognition (Waibel, 1989)
- Character Recognition (LeCun et al., 1989)
- Face Recognition (Mitchell)
- **ALVINN:** Steer a van down the road

Questions..??