دانشگاه کردستان
University of Kurdistan
زانکۆی کوردستان

Department of Computer Engineering
University of Kurdistan

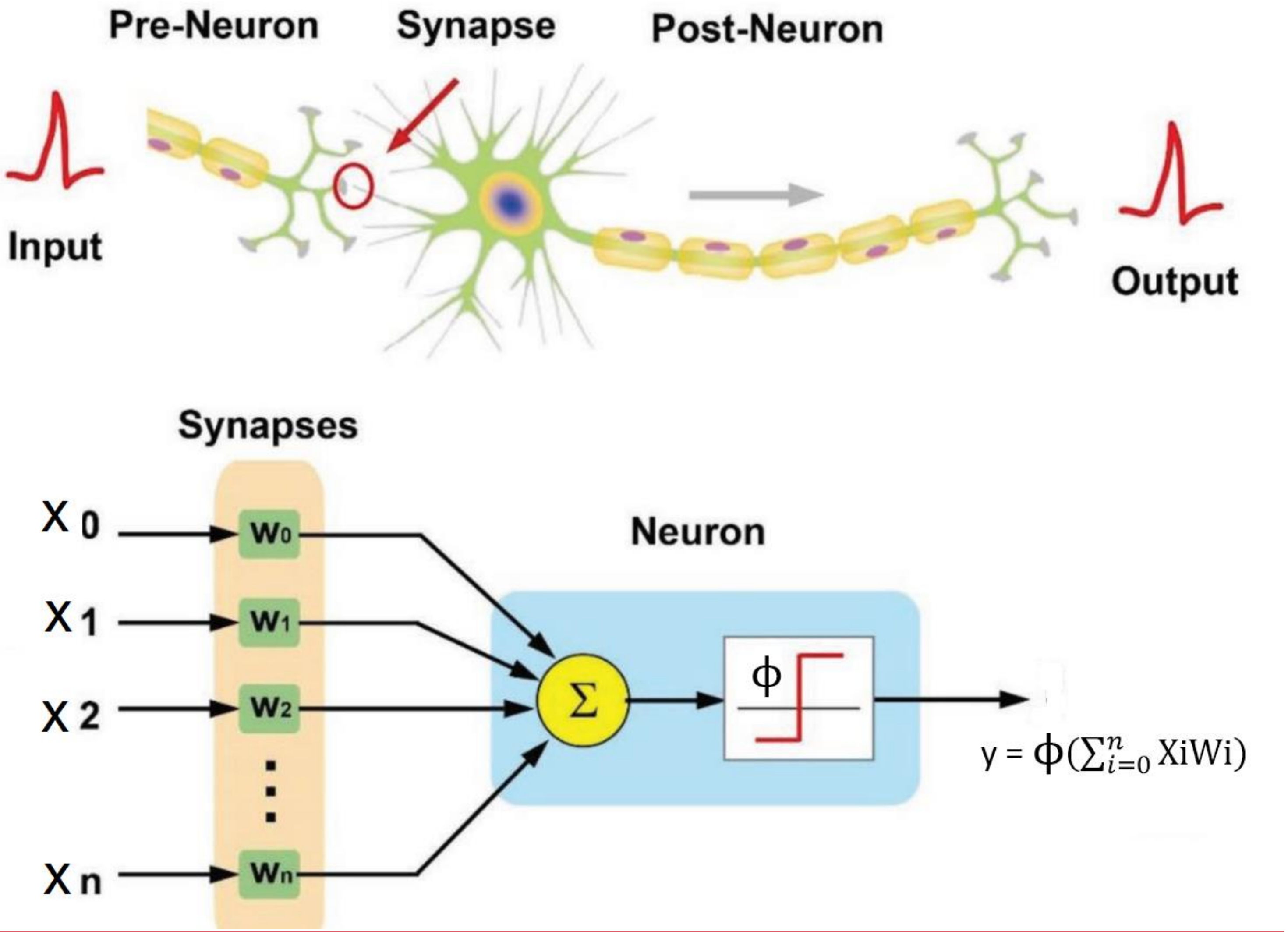
Deep Learning (Graduate level)

Feed Forward NNs & Multi-Layer Perceptron

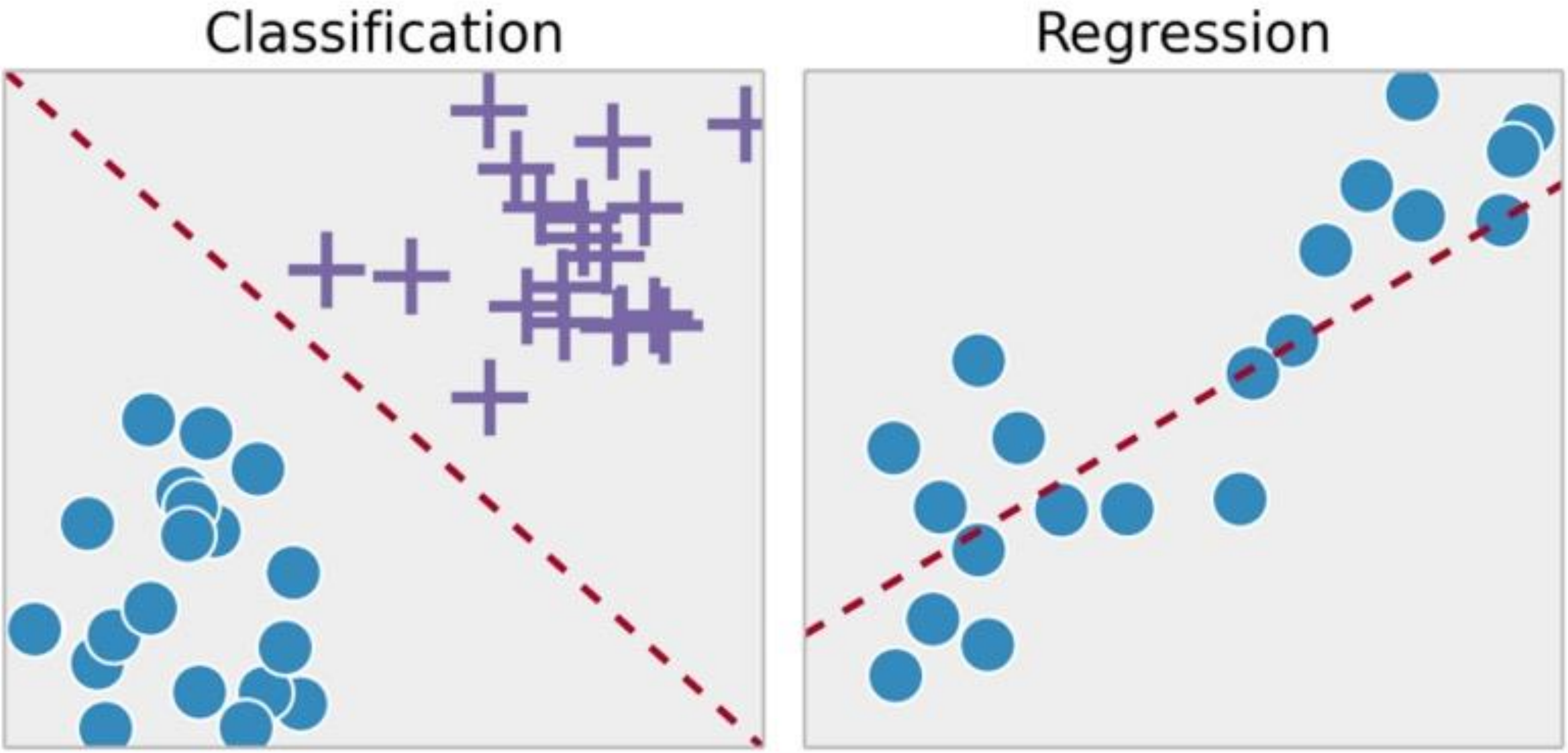
By: Dr. Alireza Abdollahpouri

Session 1: Foundations

Recap: Biological Inspiration



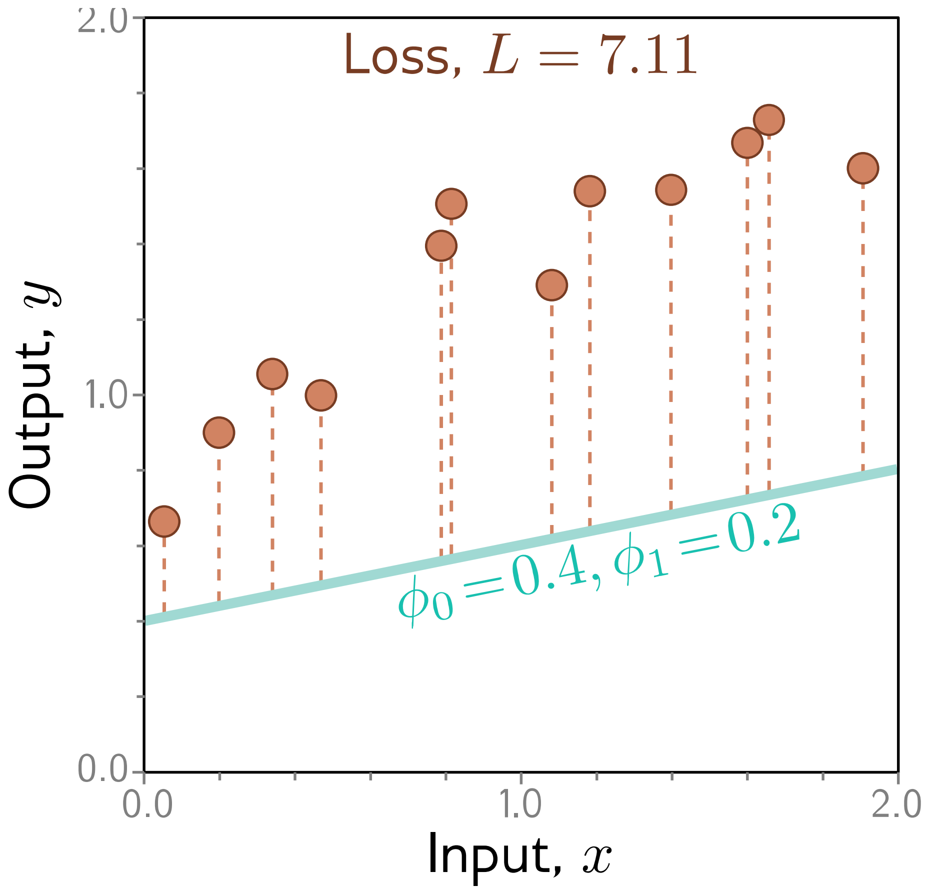
Supervised learning



Output has discrete value

Output has continuous value

Regression - Supervised learning



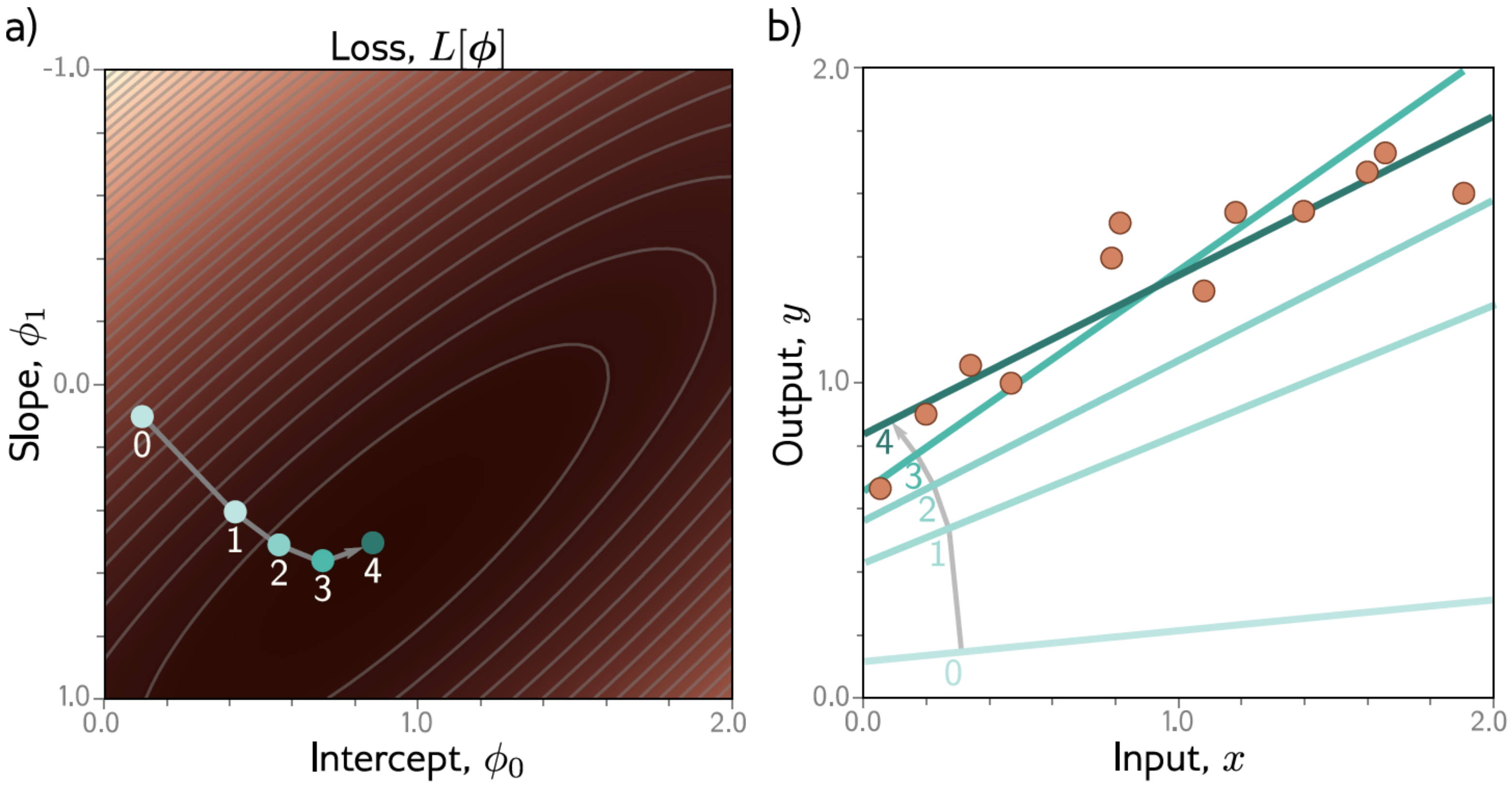
Loss function:

$$L[\phi] = \sum_{i=1}^{I} (\mathrm{f}[x_i, \phi] - y_i)^2$$

$$= \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2$$

"Least squares loss function"

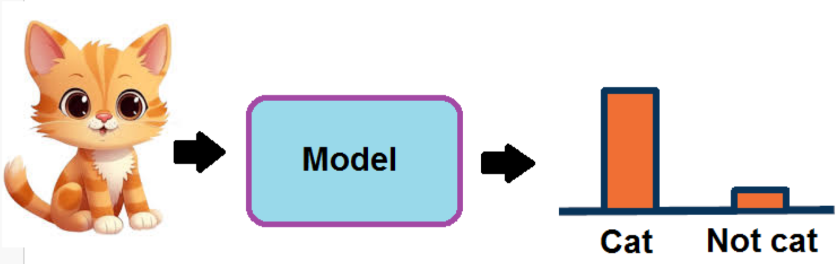
Regression - Supervised learning



This technique is known as
gradient descent

Classification- Supervised learning

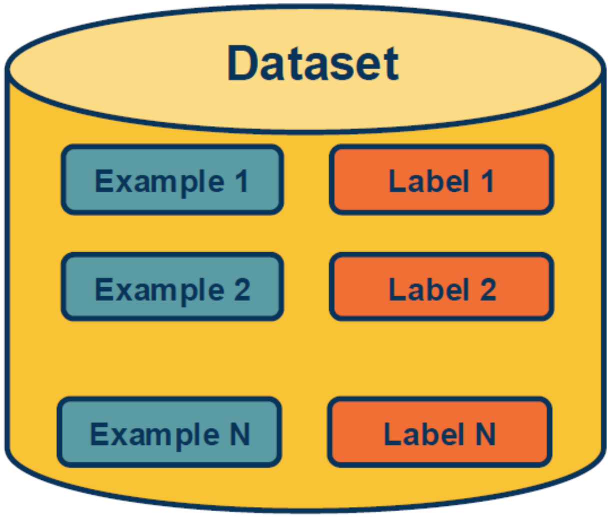
- Train Input: $\{X, Y\}$
- Learning output: $f : X \rightarrow Y$
- Usually f is a **distribution**, e.g. $P(y|x)$



Dataset

$X = \{x_1, x_2, \dots, x_N\}$ *where* $x \in \mathbb{R}^d$ Examples

$Y = \{y_1, y_2, \dots, y_N\}$ *where* $y \in \mathbb{R}^c$ Labels



Classification- Supervised learning

Given a **Training Set** of examples (pattern, label) the objective is to “learn” to classify a new pattern with the correct label.

	pattern	label
Example 1		cat
Example 2		not cat
Example 3		cat
Example 4		not cat
Example 5		not cat
Example 6		cat

→ cat

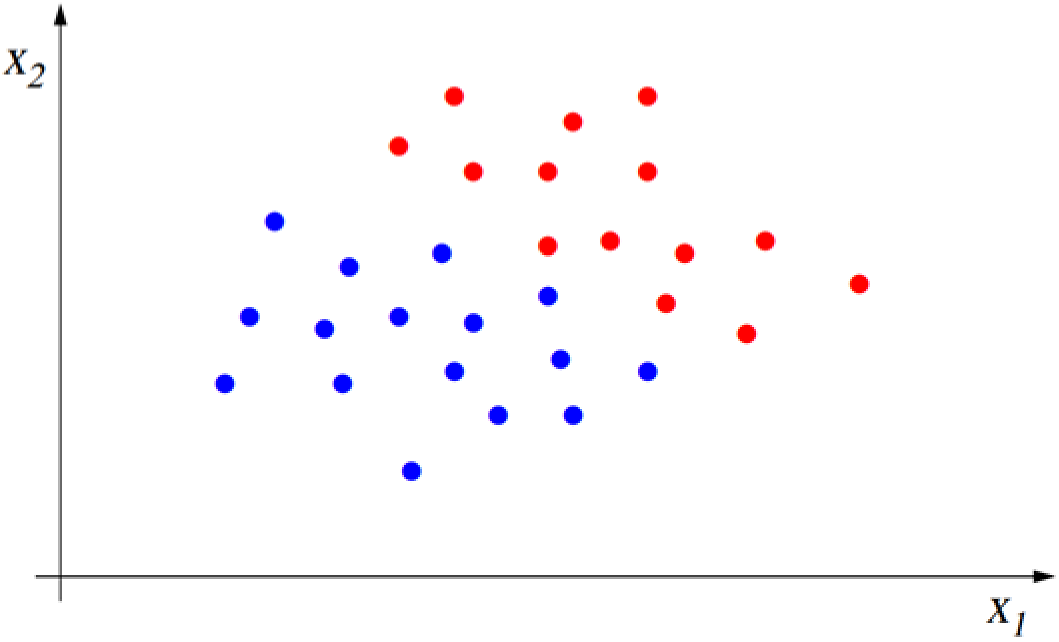
$$\mathrm{TS} = \{(\boldsymbol{x}_\mathrm{k}, t_k),\ k = 1, \ldots, M\}$$

$$\boldsymbol{x}_k = \begin{pmatrix} x_{k1} \\ x_{k2} \\ \vdots \\ x_{kn} \end{pmatrix} \qquad \boldsymbol{x}_k \in R^n \qquad t_k \in [0,1]$$

Classification

Basically we want our system to classify a set of patterns as belonging to a given class or not.

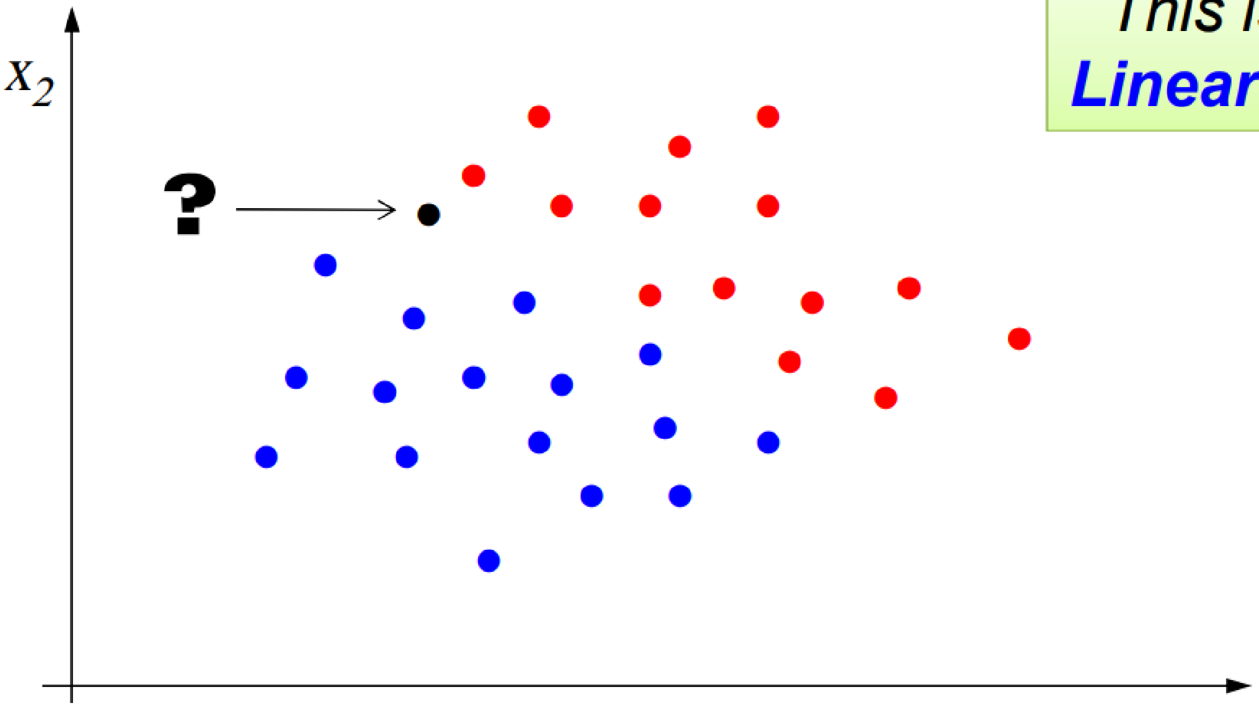
If $\mathbf{x} \in \mathbf{R}^2$ *we can represent the situation on a plane:*



Classification

A simple approach to classify a new pattern is to look at the closest neighbor and return its label.

A better way is to find a line that best separates the data set:



This is called a ***Linear Classifier***

Linear Classifier

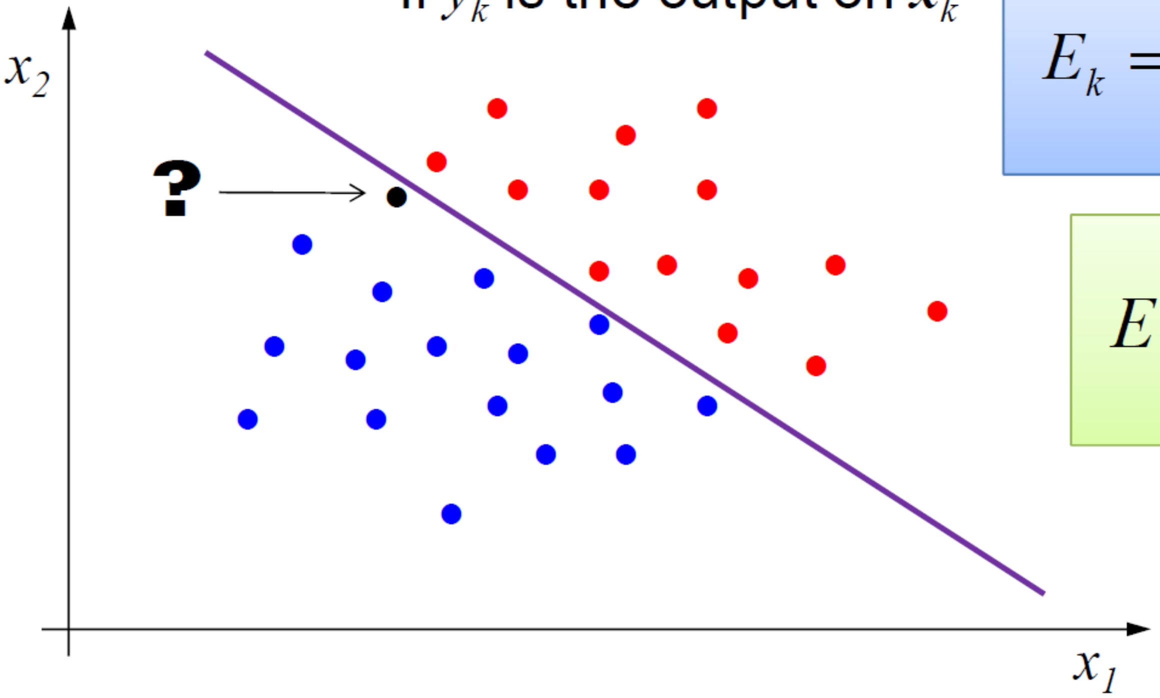
GOAL: identify the line that “best separates” the two data sets.

Error: the quality of separation is measured by a *loss function* that measures the error of the classifier:

If y_k is the output on $\boldsymbol{x}_k$

$$E_k = \frac{1}{2}(t_k - y_k)^2$$

$$E = \frac{1}{M}\sum_{k=1}^{M} E_k$$



Supervised learning

In a supervised learning paradigm, a neural network operates in two distinct phases:

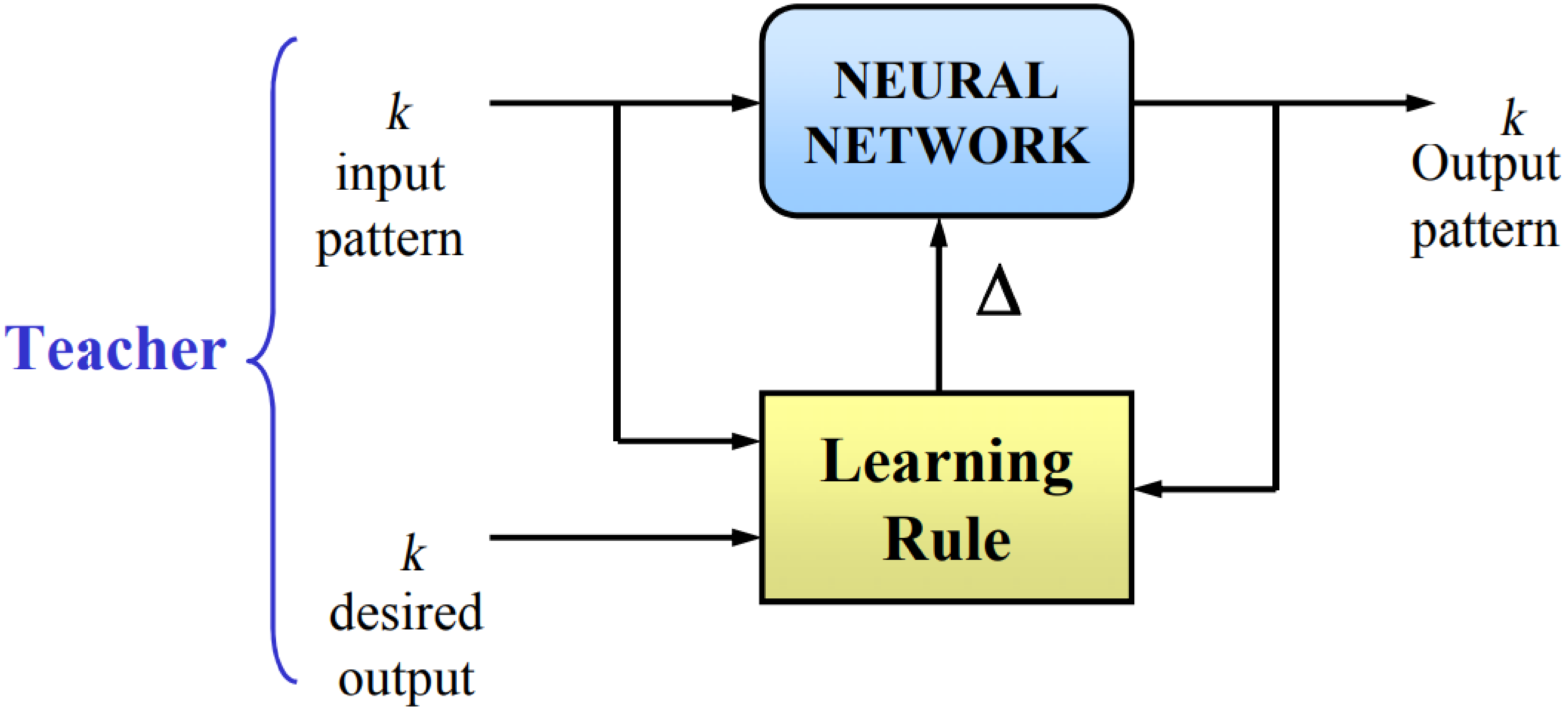
1. ***A learning phase***

   The network is trained to classify the examples in the Training Set (weights are modified based on errors).

2. ***An operating phase***

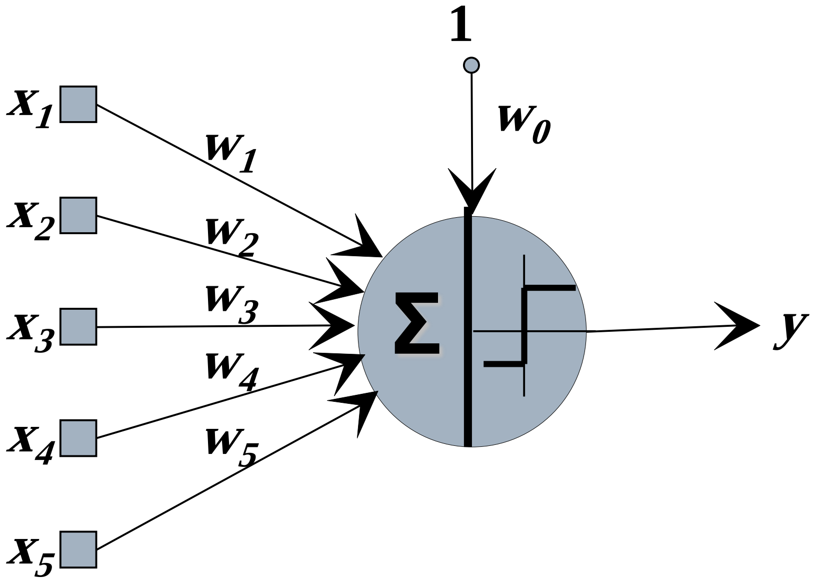
   The network is used on new data never seen before (weights are kept fixed)

Learning phase



The Perceptron

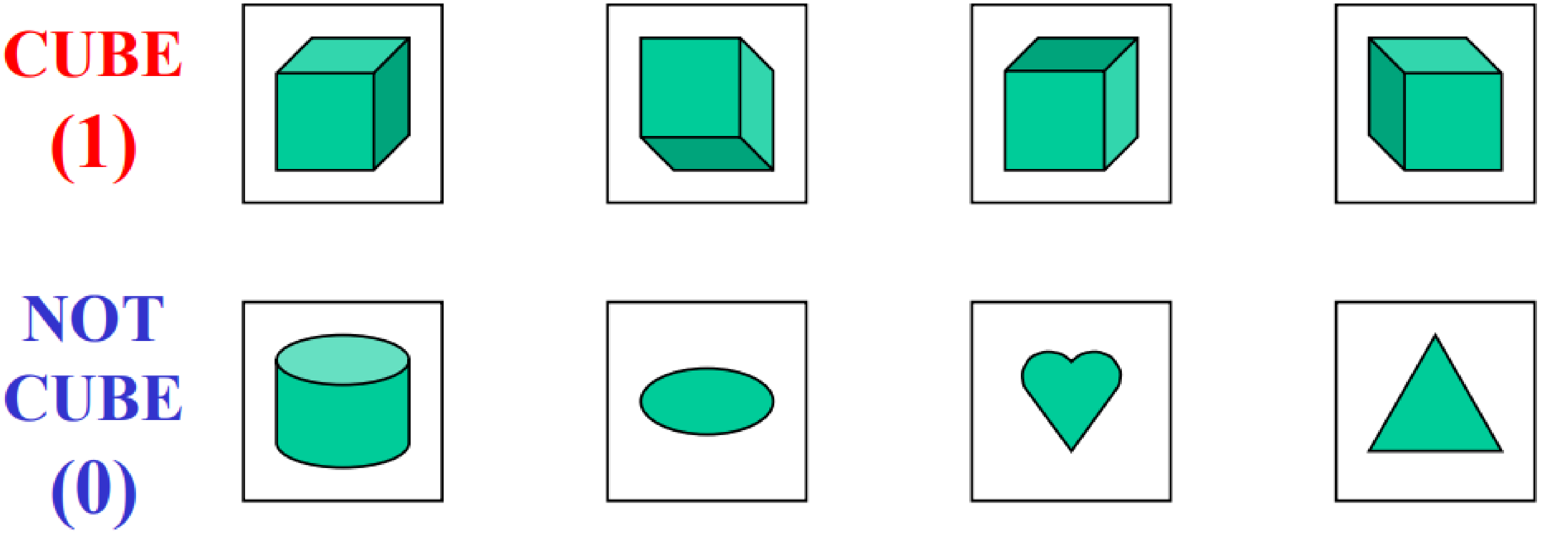
The first model of a biological neuron



$$y = \operatorname{sgn}\left(\sum_{i=1}^{n} w_i x_i + w_0 \right)$$

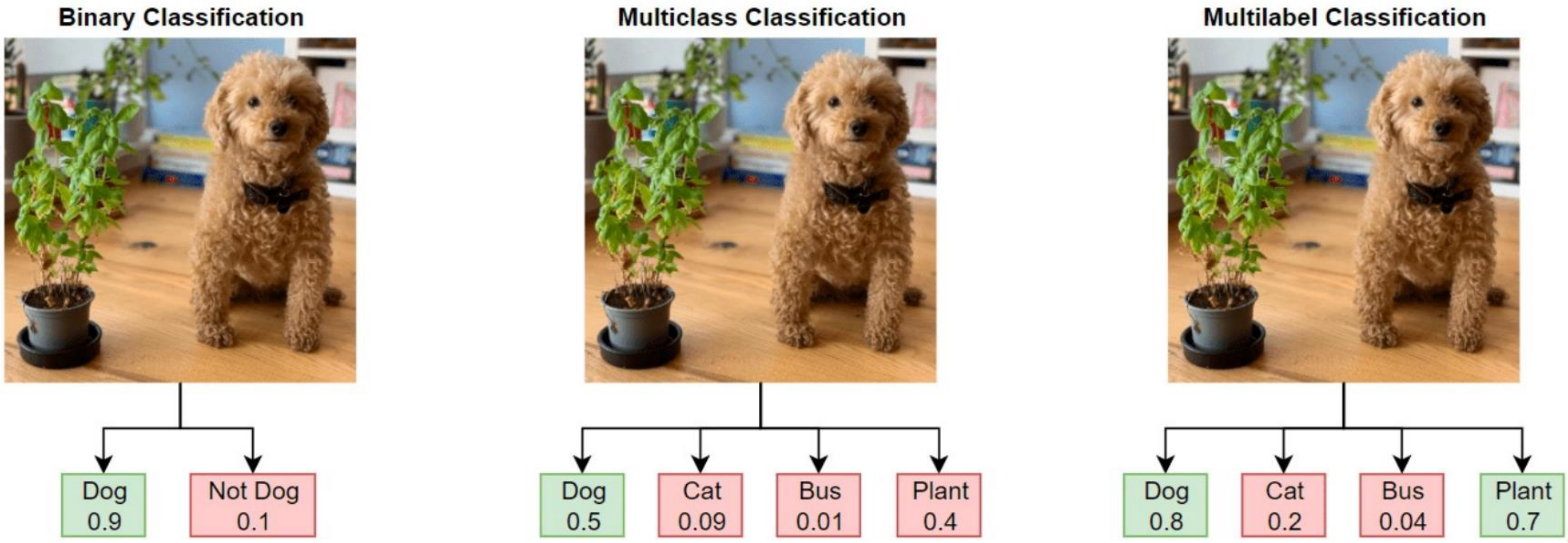
Perceptron for Classification

- A perceptron can be trained to recognize whether an input pattern X belongs or not to a class C:



3 Types of Classification

- Binary Classification
- Multi-class Classification
- Multi-label Classification



Binary Classification

Multiclass Classification

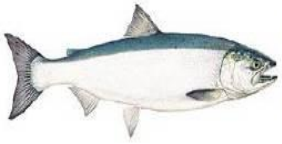
Multilabel Classification

Classification – An Example

Problem: Sorting incoming fish on a conveyor belt.

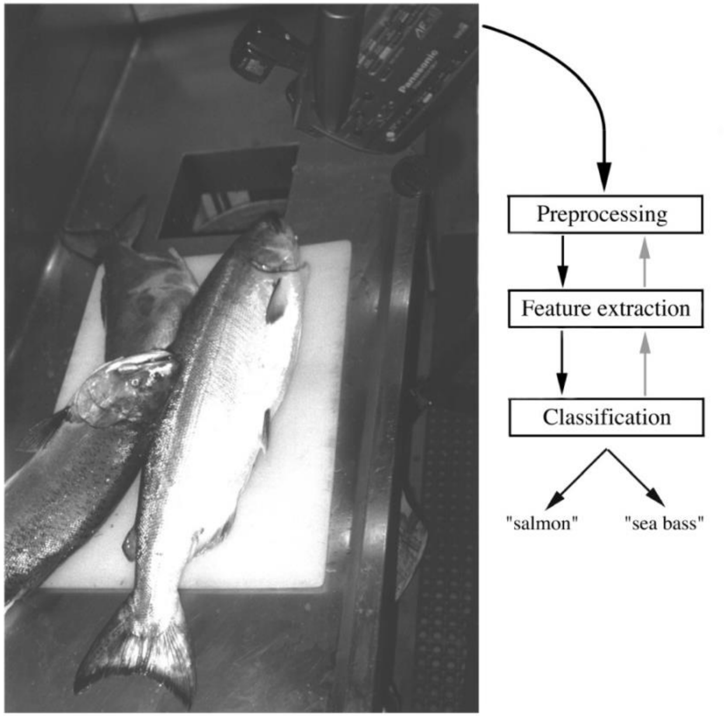
Assumption: Two kind of fish:

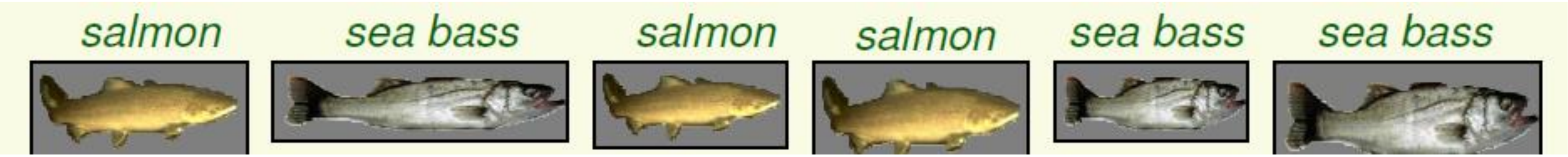
(1) sea bass
(2) salmon



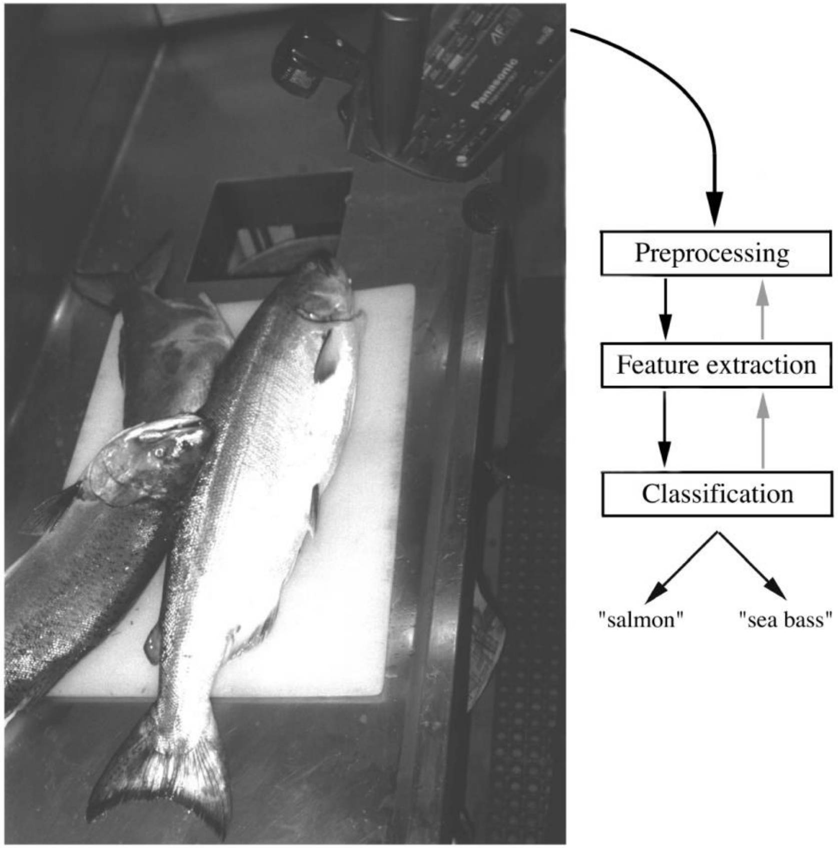
(d)

(e)





Pre-processing Step



Preprocessing

Feature extraction

Classification

"salmon"

"sea bass"

Example

(1) Image enhancement

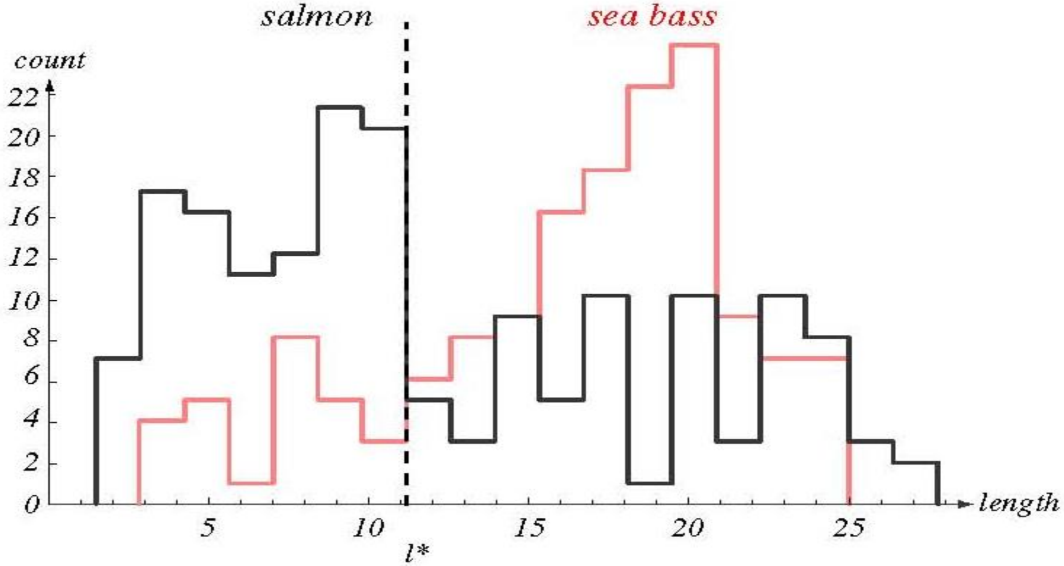
(2) Separate touching or occluding fish

(3) Find the boundary of each fish

Feature Extraction

- Assume a fisherman told us that a sea bass is generally longer than a salmon.
- We can use length as a feature and decide between sea bass and salmon according to a threshold on length.
- How should we choose the threshold?

“Length” Histograms

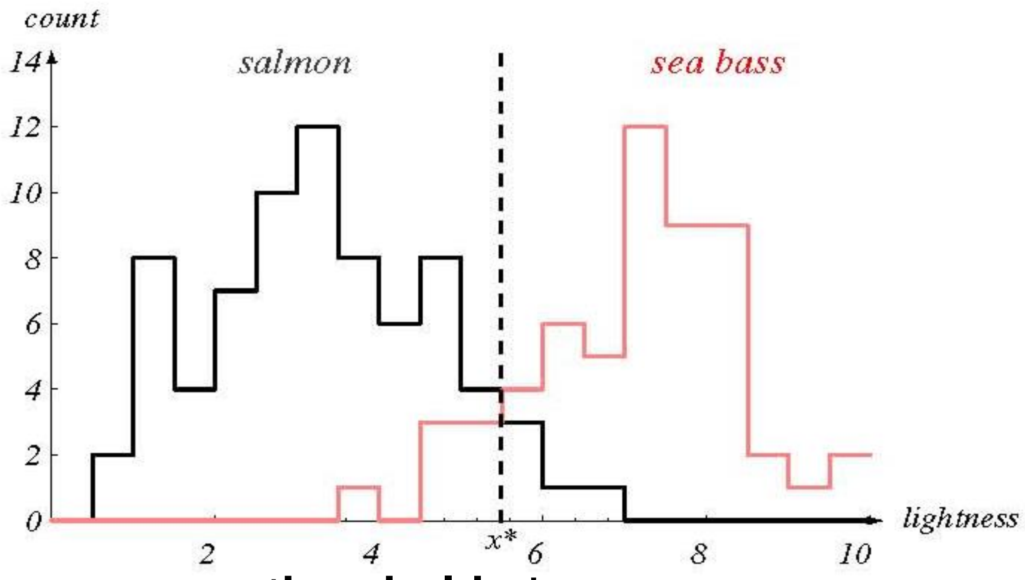


threshold l*

- Even though sea bass is longer than salmon on the average, there are many examples of fish where this observation does not hold.

“Average Lightness” Histograms

- Consider a different feature such as “average lightness”



threshold *x**

- It seems easier to choose the threshold x^* but we still cannot make a perfect decision.

Multiple Features

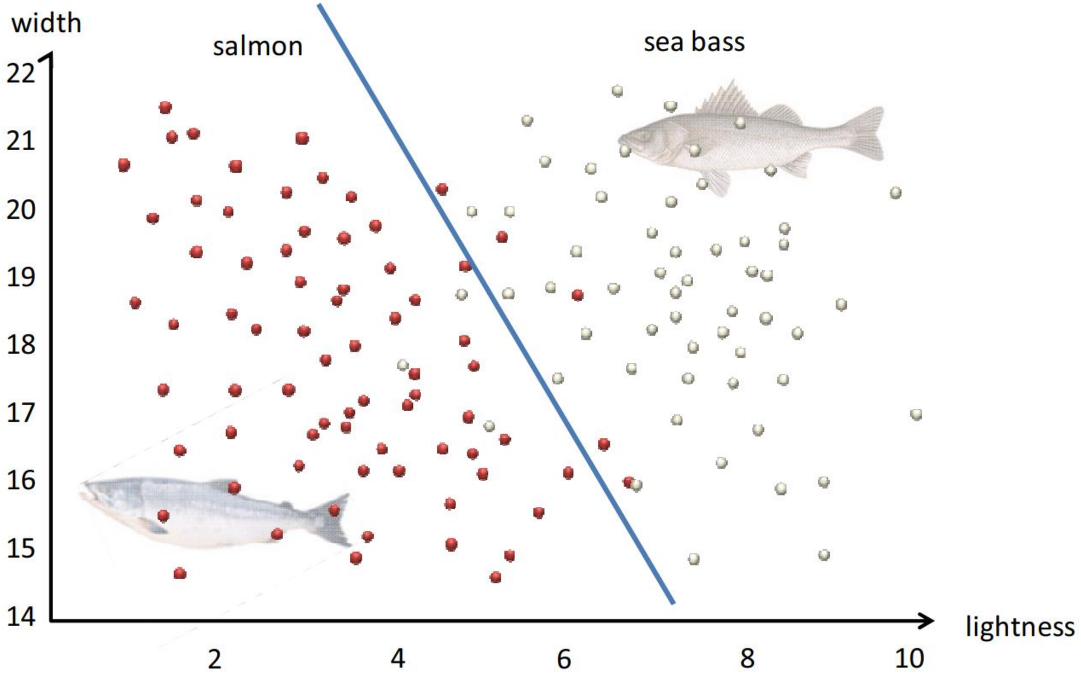
- To improve recognition accuracy, we might have to use more than one features at a time.
  - Single features might not yield the best performance.
  - Using combinations of features might yield better performance.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \begin{array}{l} x_1 : lightness \\ x_2 : width \end{array}$$

- How many features should we choose?

Classification

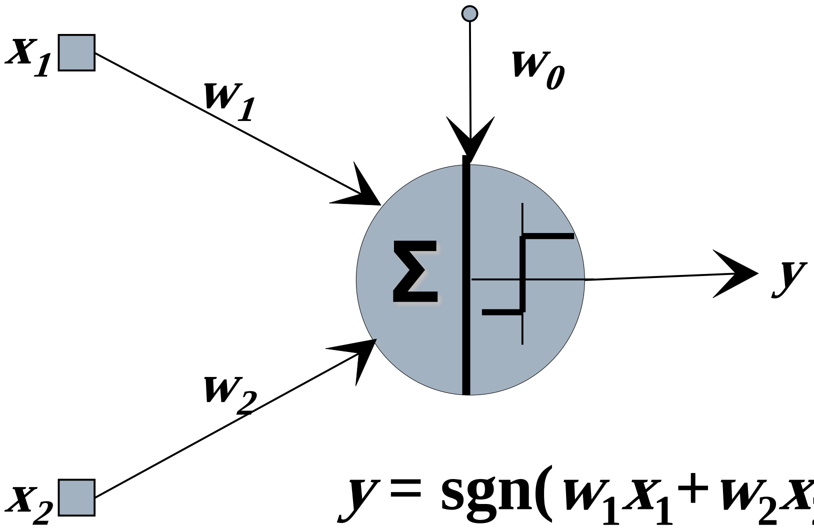
- Partition the *feature space* into two regions by finding the decision boundary that minimizes the error.



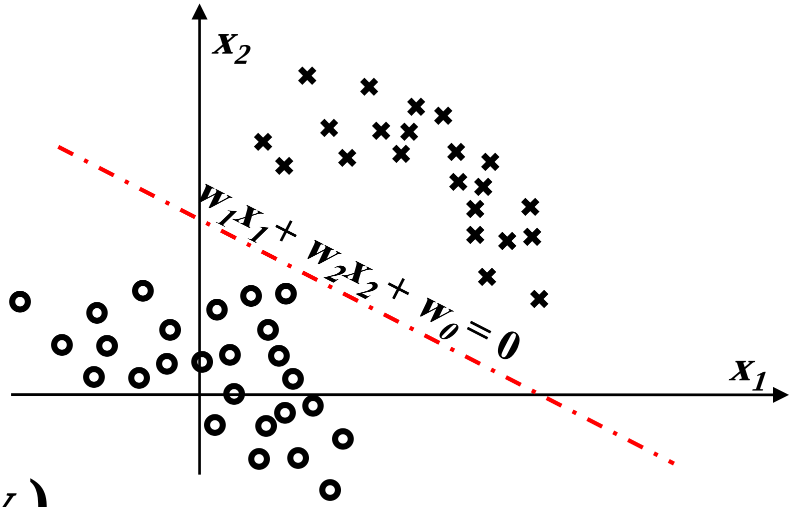
- How should we find the optimal decision boundary?

What a Perceptron does?

For a perceptron with 2 input variables namely x_1 and x_2 Equation $W^T X = 0$ determines a line **separating** positive from negative examples.

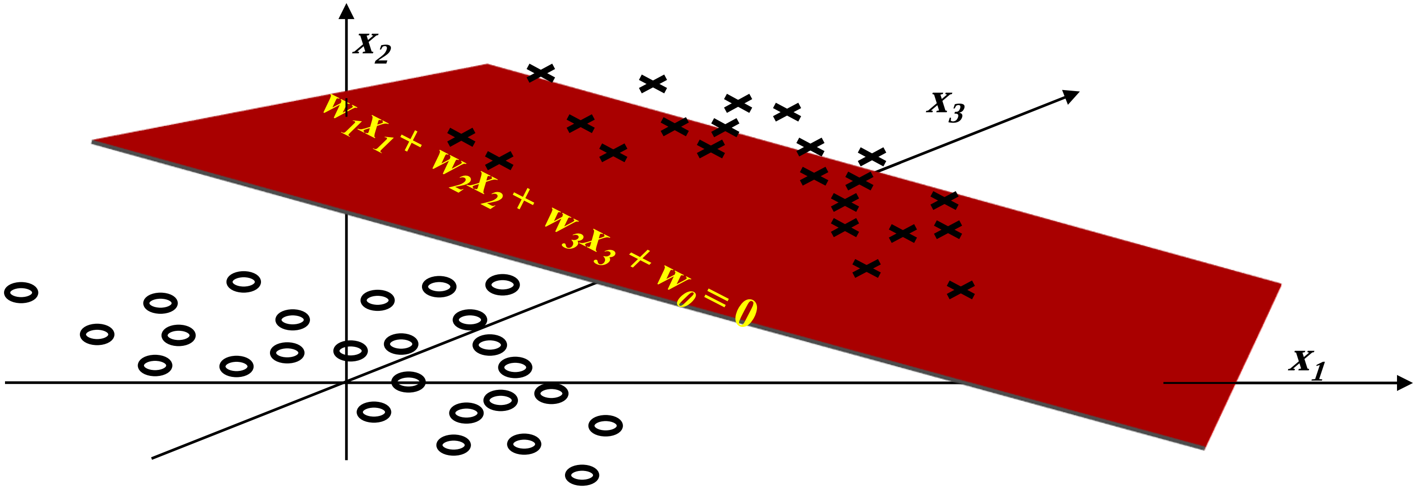


$$y = \text{sgn}(w_1 x_1 + w_2 x_2 + w_0)$$



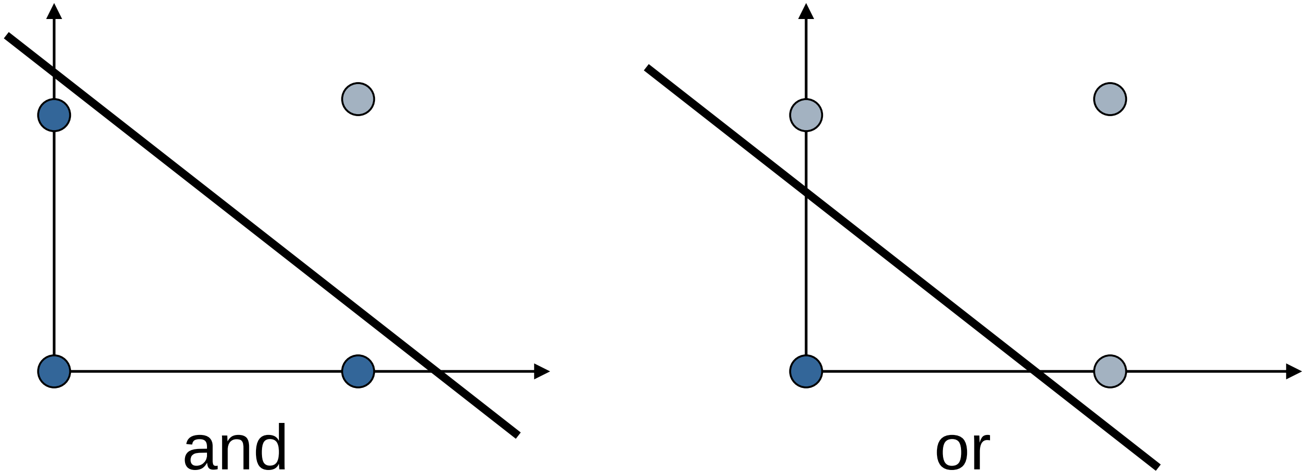
What a Perceptron does?

For a perceptron with n input variables, it draws a Hyper-plane as the decision boundary over the (n-dimensional) input space. It classifies input patterns into two classes.



The perceptron outputs 1 for instances lying on one side of the hyperplane and outputs –1 for instances on the other side.

What can be represented using Perceptrons?

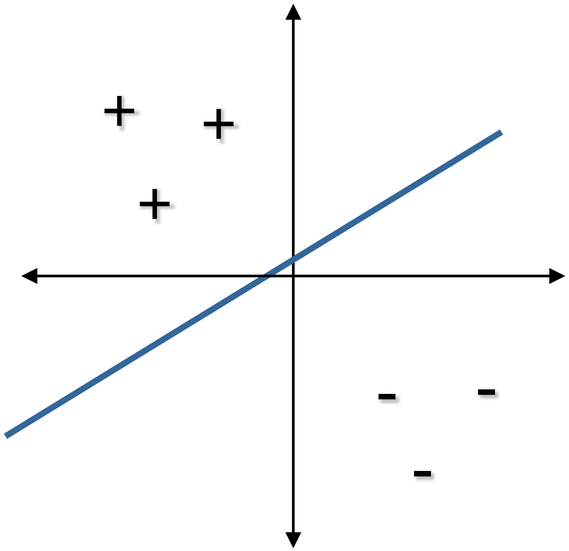


Representation Theorem: Perceptrons can only represent **linearly separable** functions.

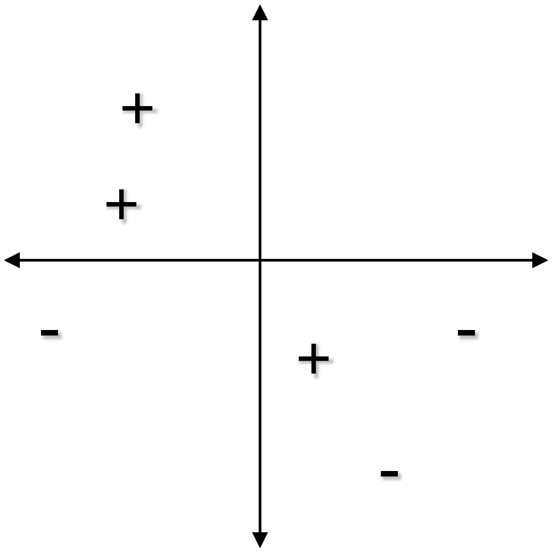
Examples: AND,OR, NOT.

Limits of the Perceptron

A perceptron can learn only examples that are called "linearly separable". These are examples that can be perfectly separated by a hyperplane.



Linearly separable



Non-linearly separable

Session 2:
The Learning Process

Learning Perceptrons

- Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameters changes take place.

- In the case of Perceptrons, we use a **supervised learning**.

- Learning a perceptron means finding the right values for W that satisfy the input examples {(input$_i$, target$_i$)*}

- The hypothesis space of a perceptron is the space of all weight vectors.

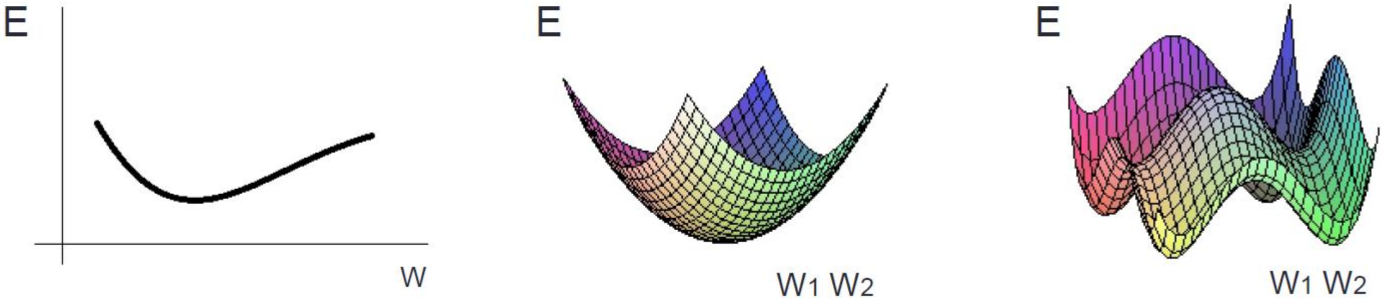
How to find the weights?

We want to find a set of weights that enable our perceptron to correctly classify our data.

We can define a cost function: $E = \frac{1}{2}(y - g(z))^2$

$\left\{ \begin{array}{l} \textbf{y}\text{: is the correct output} \\ \textbf{g(z)}\text{: is the actual output} \end{array} \right.$

- ➢ we know $\partial E/\partial wi$ then we can search for a minimum of E in **weight space**



Learning Perceptrons

Principle of learning using the perceptron rule:

1. A set of training examples is given: {(x, t)*} where x is the input and **t** the target output [**supervised learning**]

2. Examples are presented to the network.

3. For each example, the network gives an output o.

4. If there is an error, the hyperplane is moved in order to correct the output error.

5. When all training examples are correctly classified, Stop learning.

Learning Perceptrons

More formally, the algorithm for learning Perceptrons is as follows:

1. Assign random values to the weight vector

2. Apply the perceptron rule to every training example

3. Are all training examples correctly classified?

    Yes. Quit
    No. Go Back to Step 2.

Perceptron Training Rule

The perceptron training rule:

For a new training example [X = (x_1, x_2, …, x_n), **t**] update each weight according to this rule:

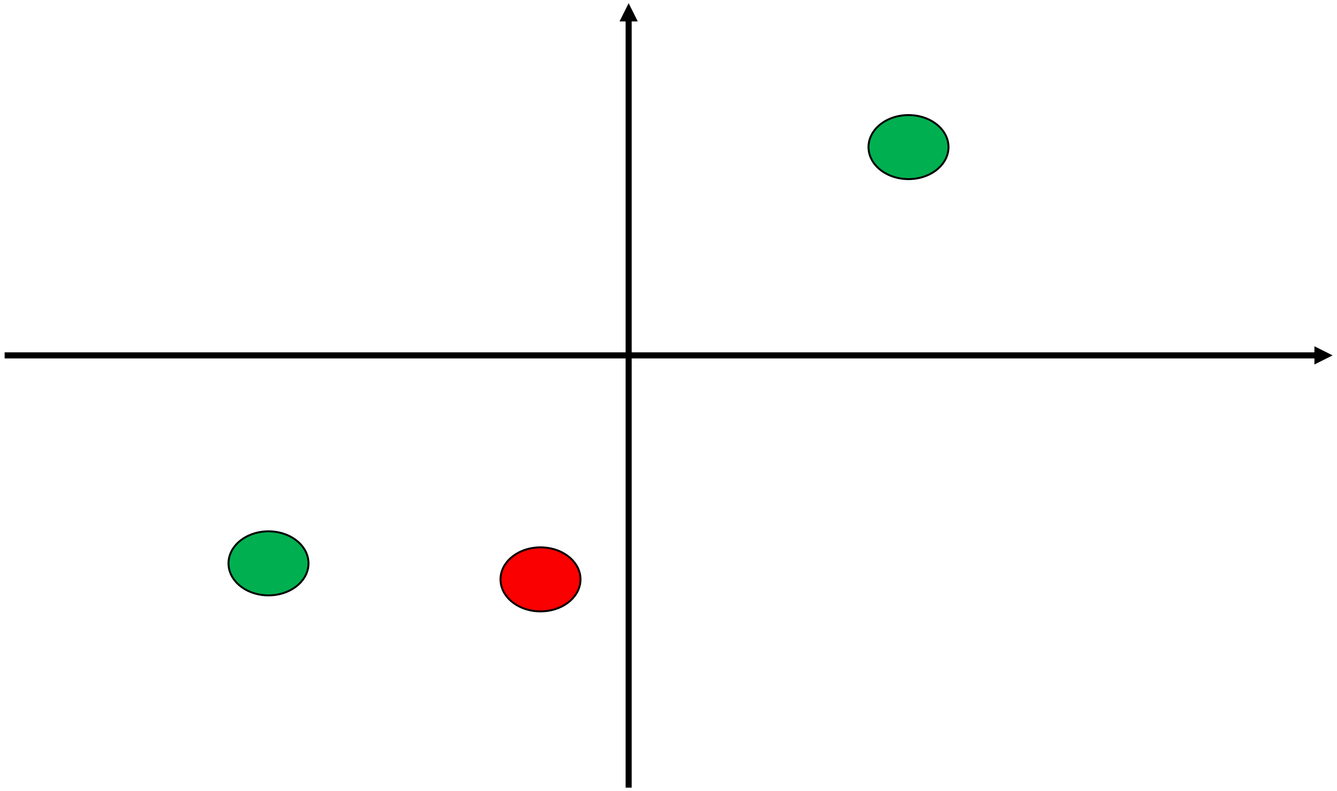
$$\mathbf{w_i = w_i + \Delta w_i}$$

Where $\mathbf{\Delta w_i = \eta.(t\text{-}o).x_i}$

t: target output
o: output generated by the perceptron
η: constant called the learning rate (e.g., 0.1)

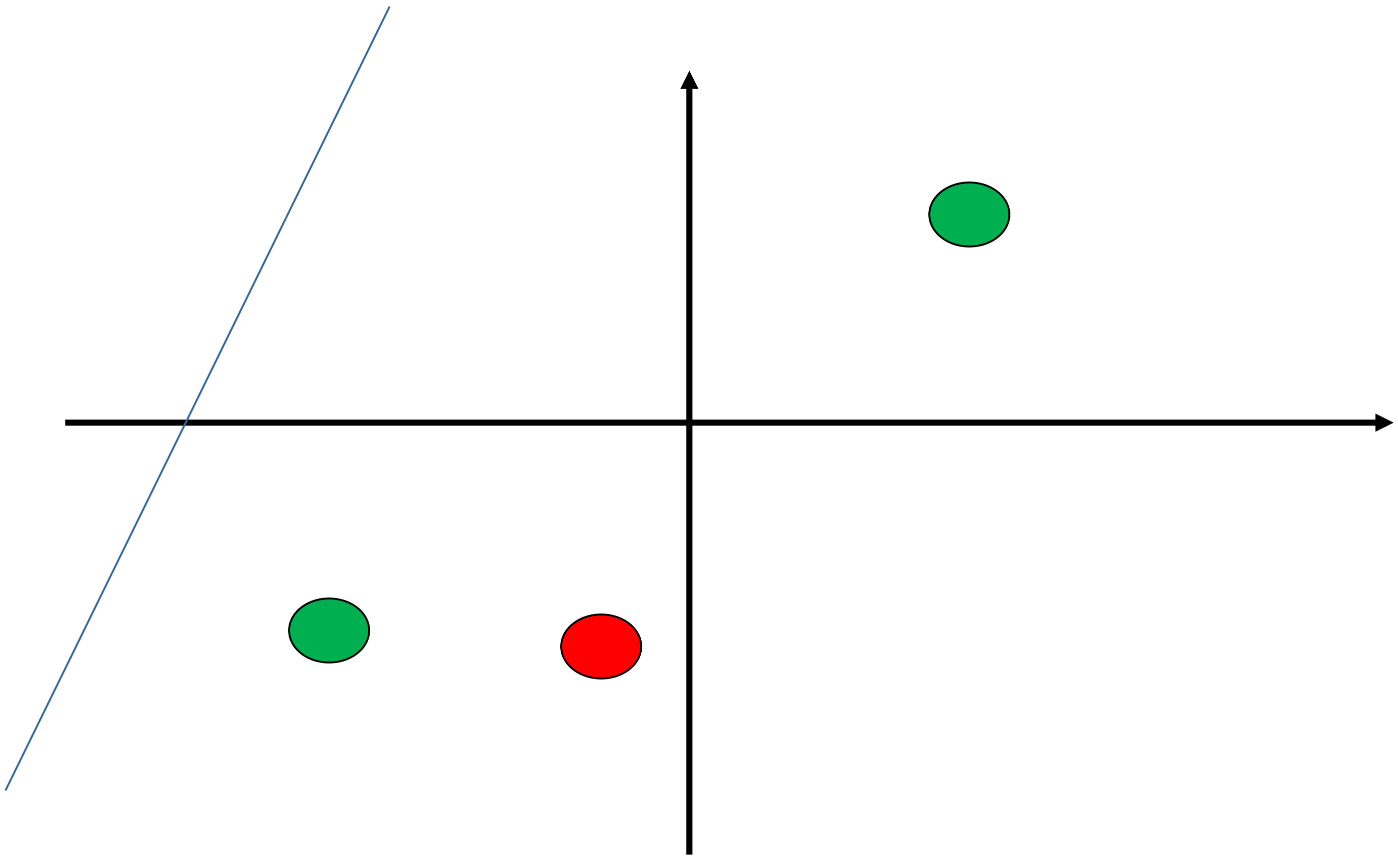
Perceptron Training Rule

Consider the following example: (two classes: Red and Green)



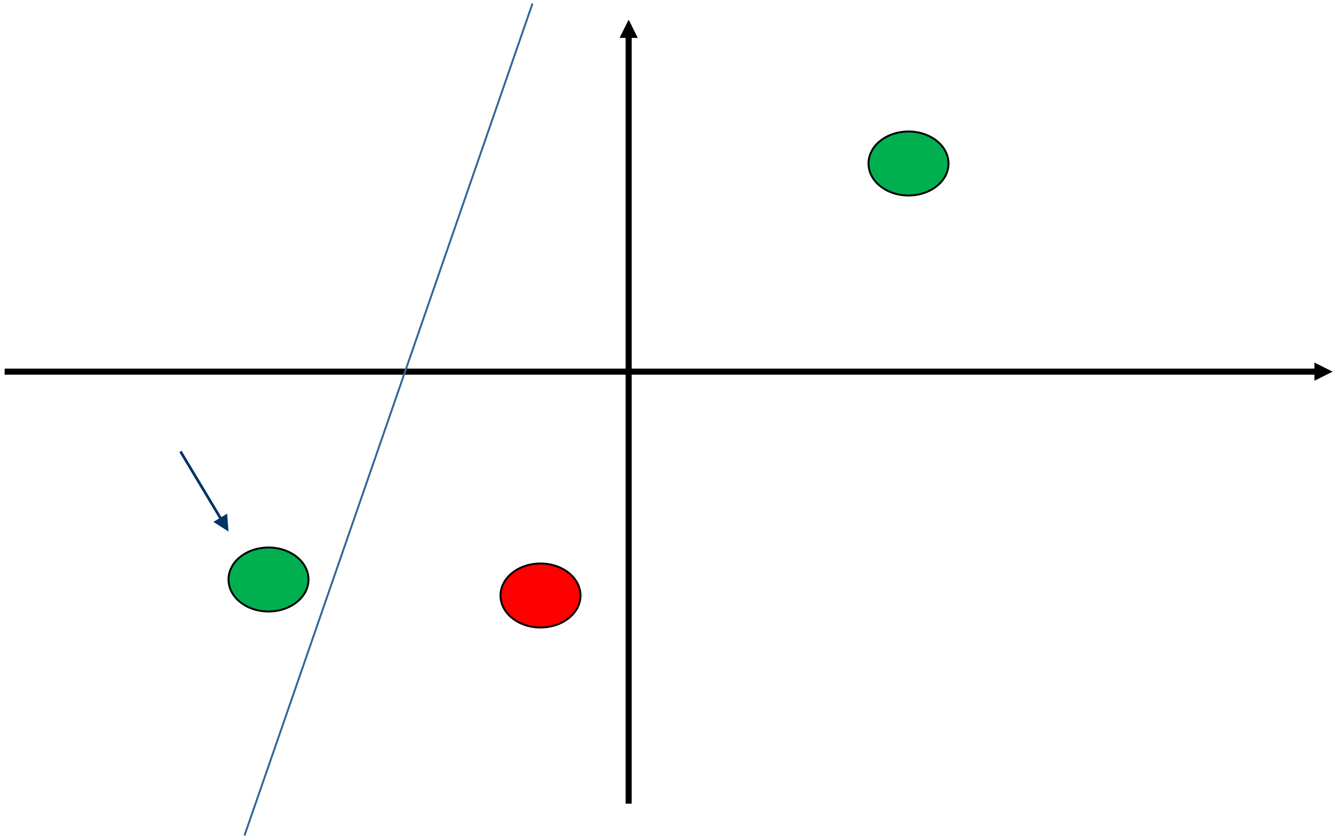
Perceptron Training Rule

Random Initialization of perceptron weights … **(A random line here)**



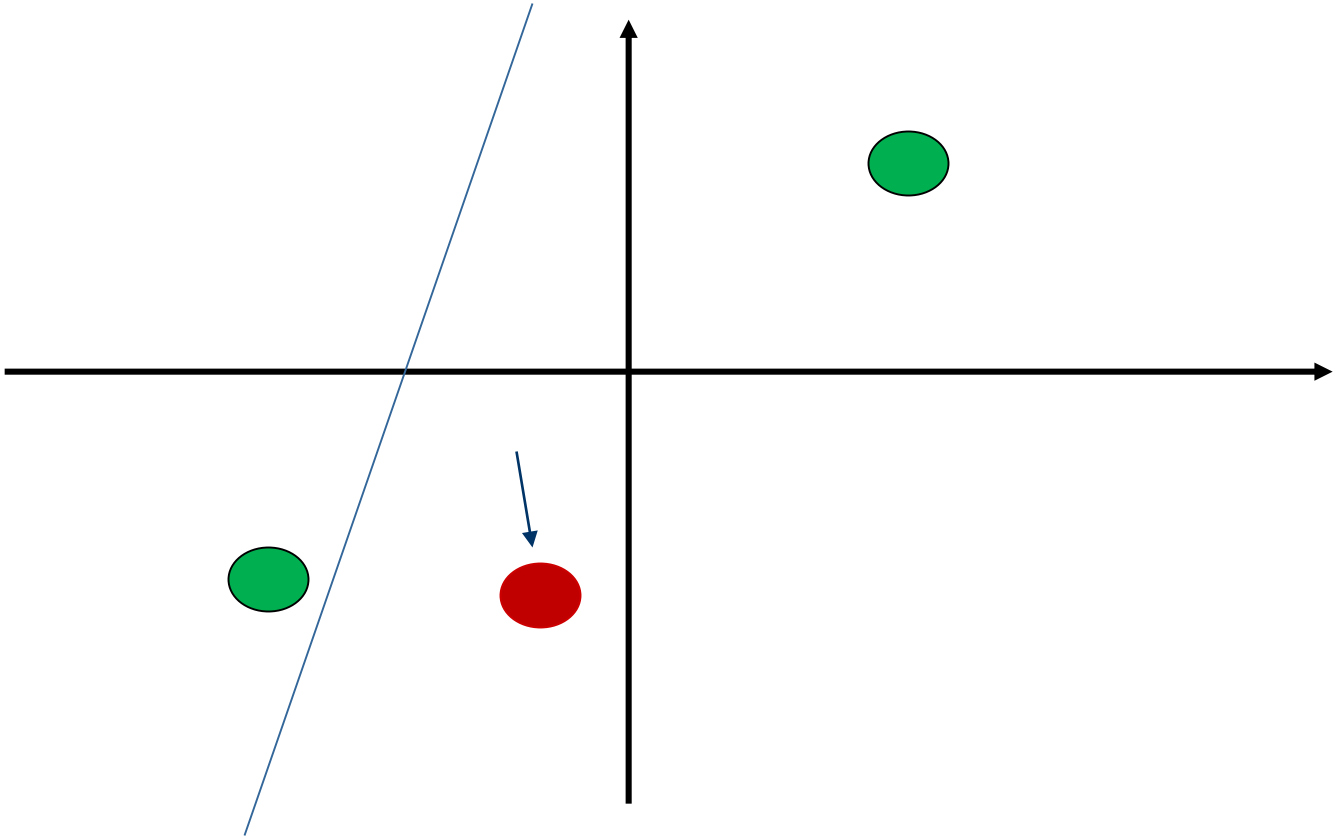
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



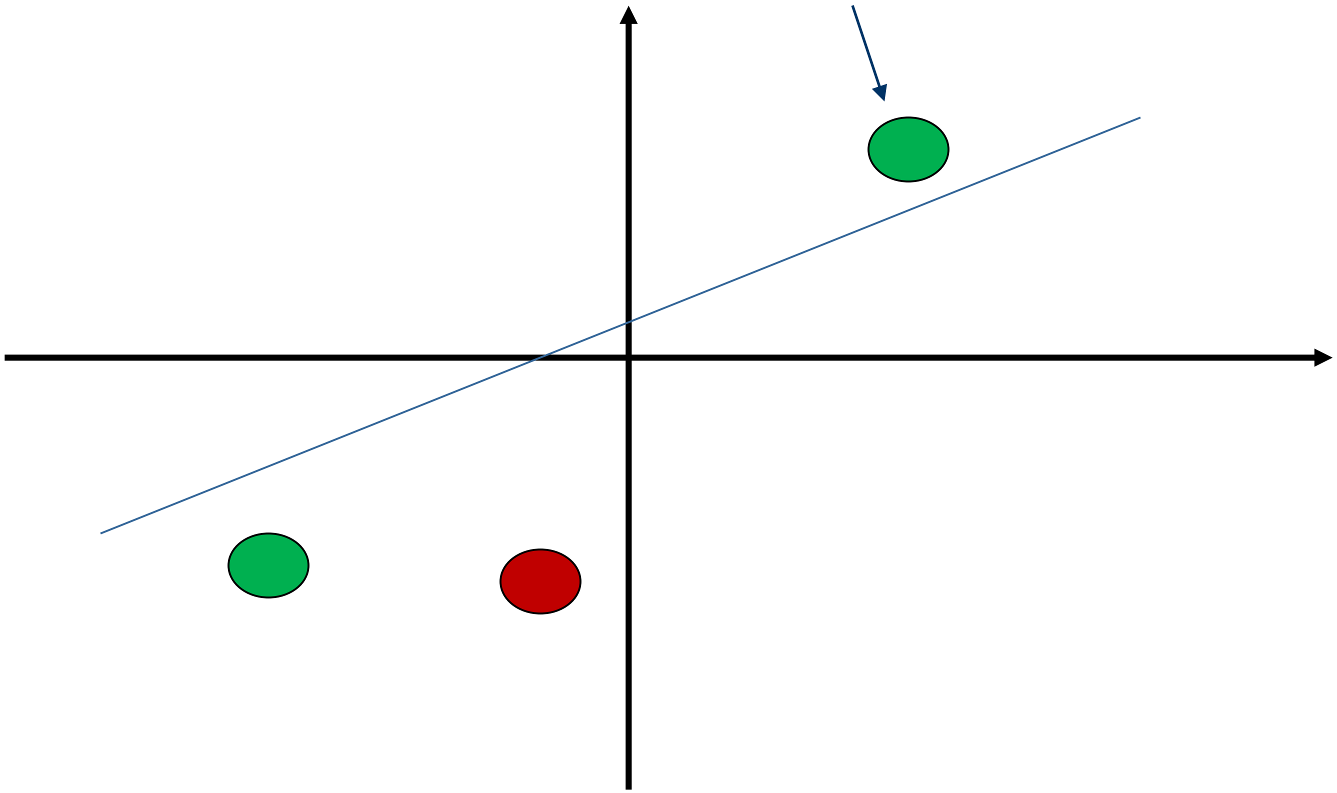
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



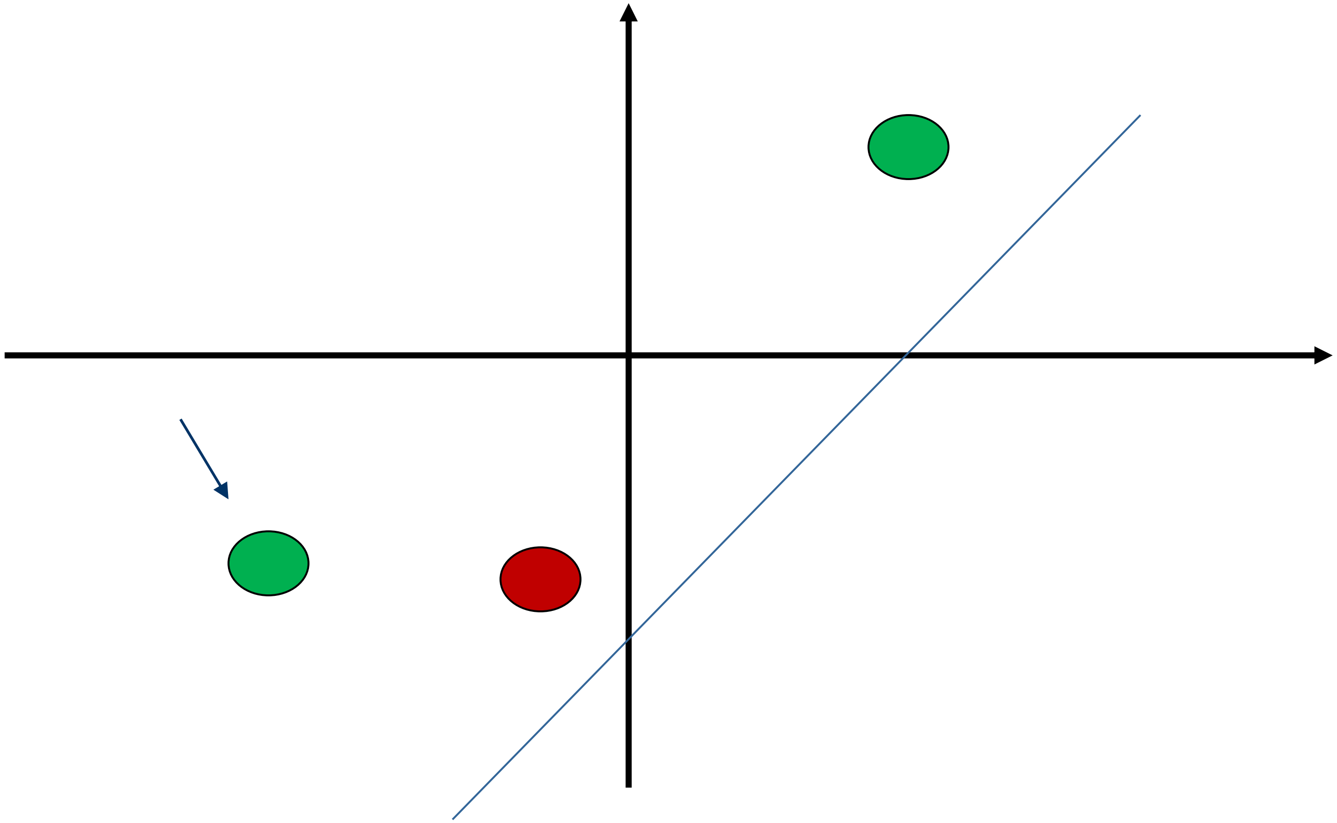
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



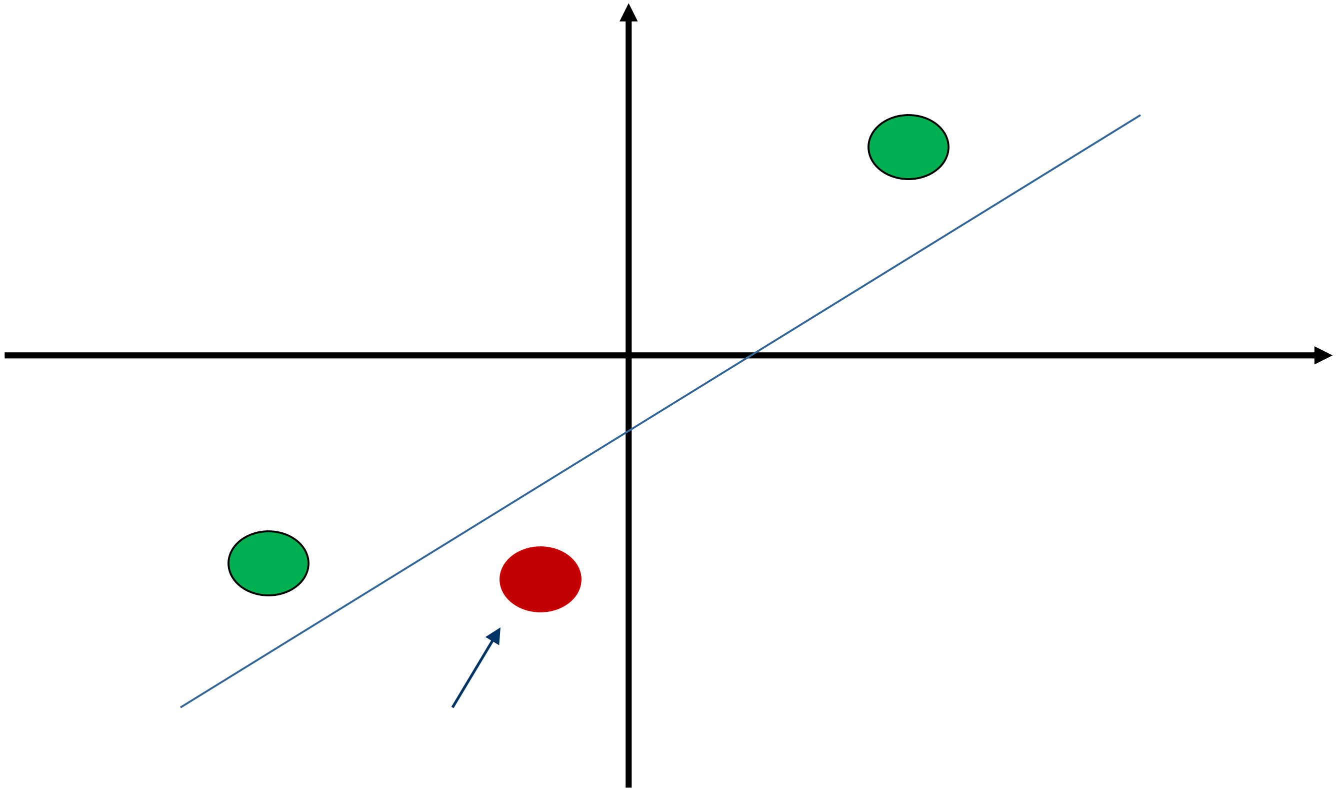
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



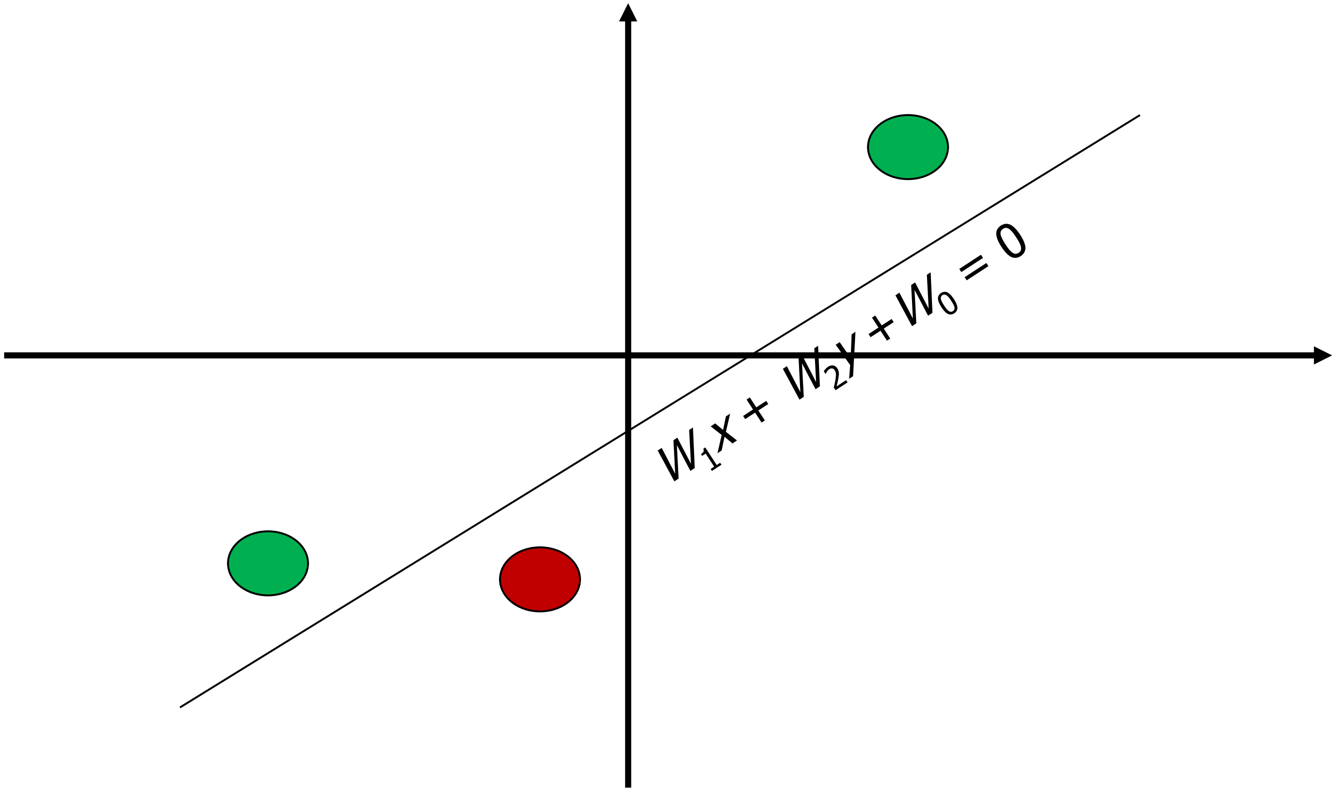
Perceptron Training Rule

All examples are correctly classified … stop Learning



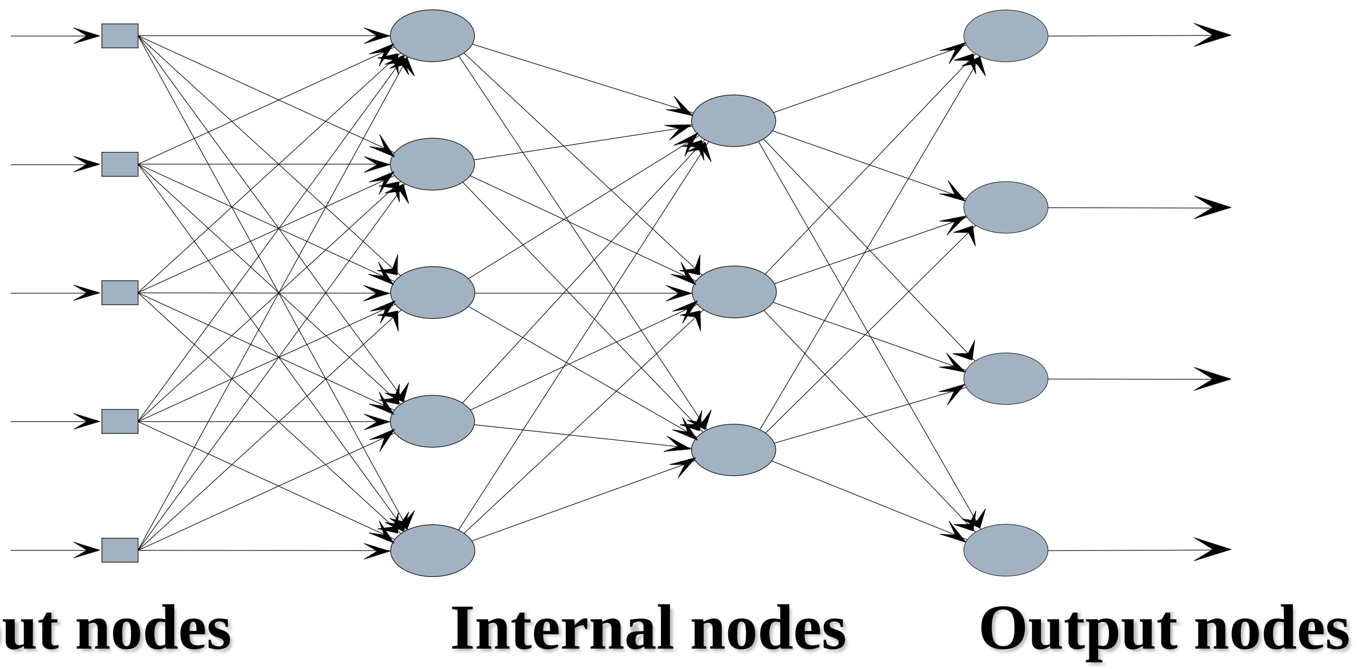
Perceptron Training Rule

The straight line $w_1 x + w_2 y + w_0 = 0$ separates the two classes (W0 , W1 and W2 are the parameters that have been learned)

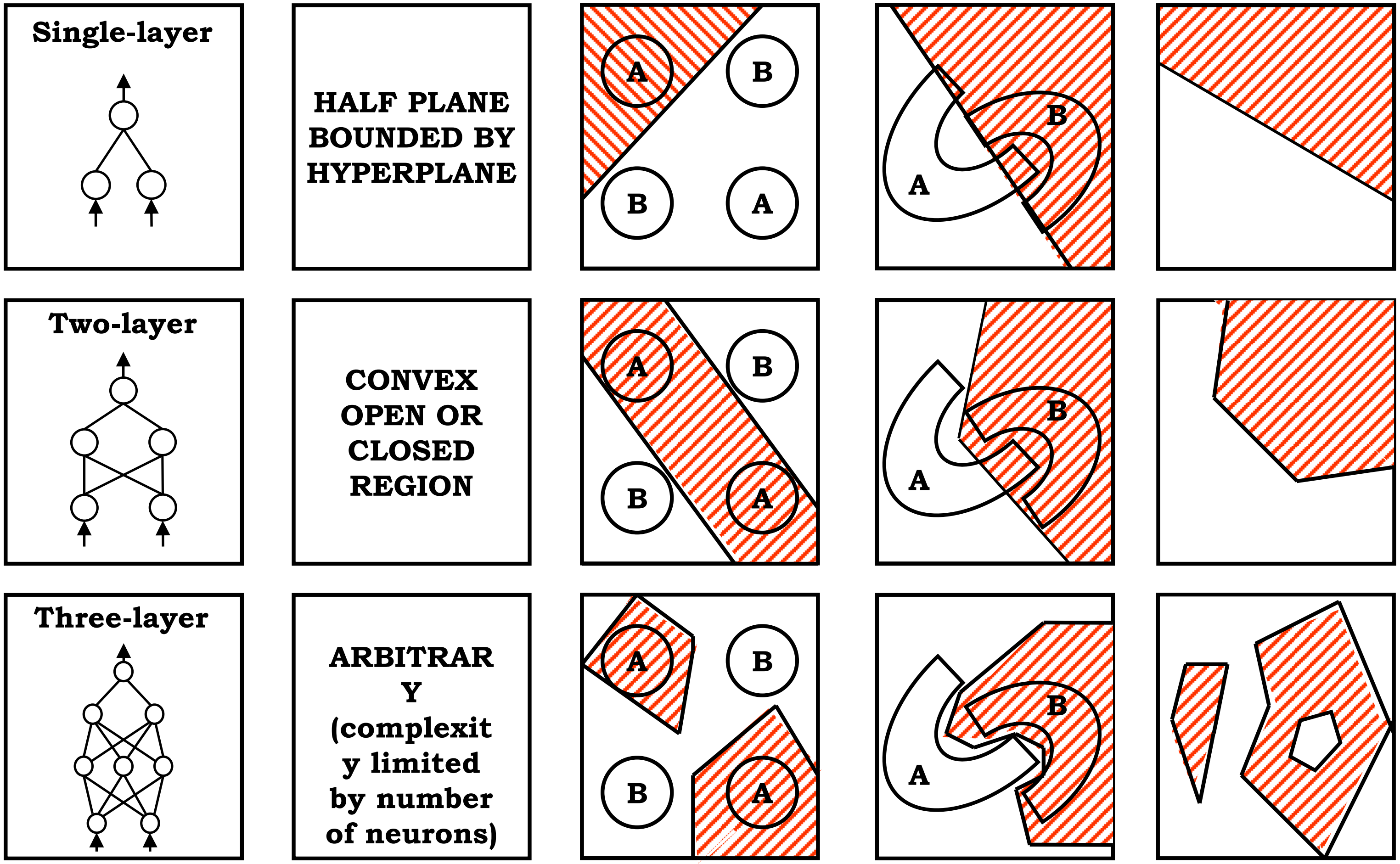


Multi-Layer Perceptron

In contrast to perceptrons, multilayer networks can learn not only multiple decision boundaries, but the boundaries may be nonlinear.



MLP- Complex Decision Boundaries



Single-layer

HALF PLANE BOUNDED BY HYPERPLANE

Two-layer

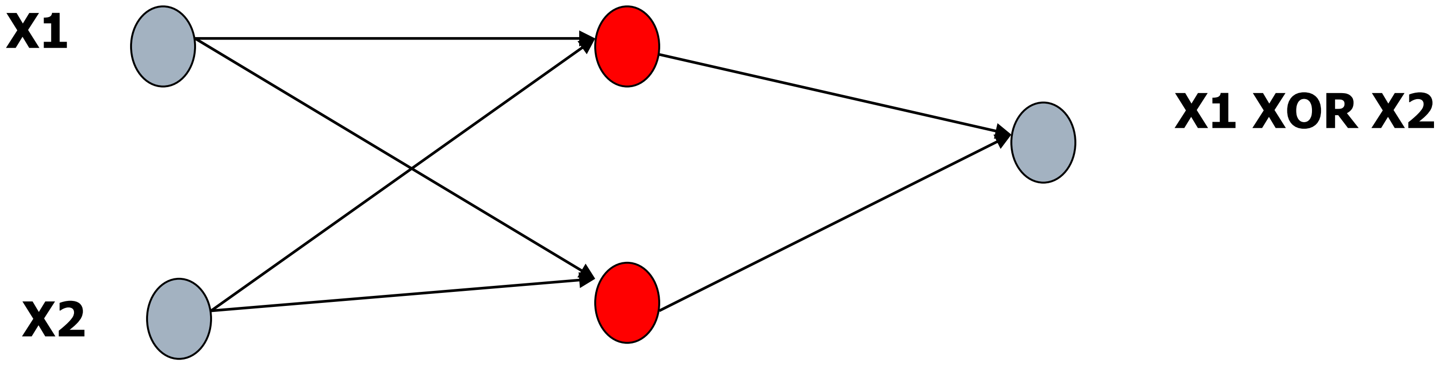
CONVEX OPEN OR CLOSED REGION

Three-layer

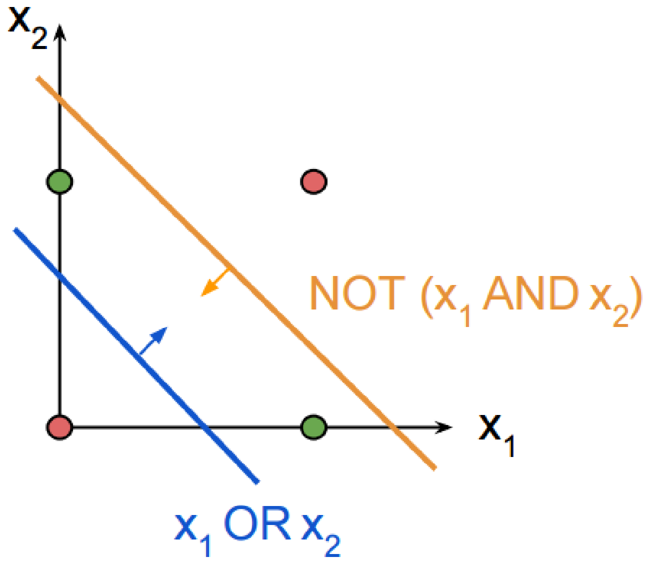
ARBITRAR Y (complexit y limited by number of neurons)

University of Kurdistan

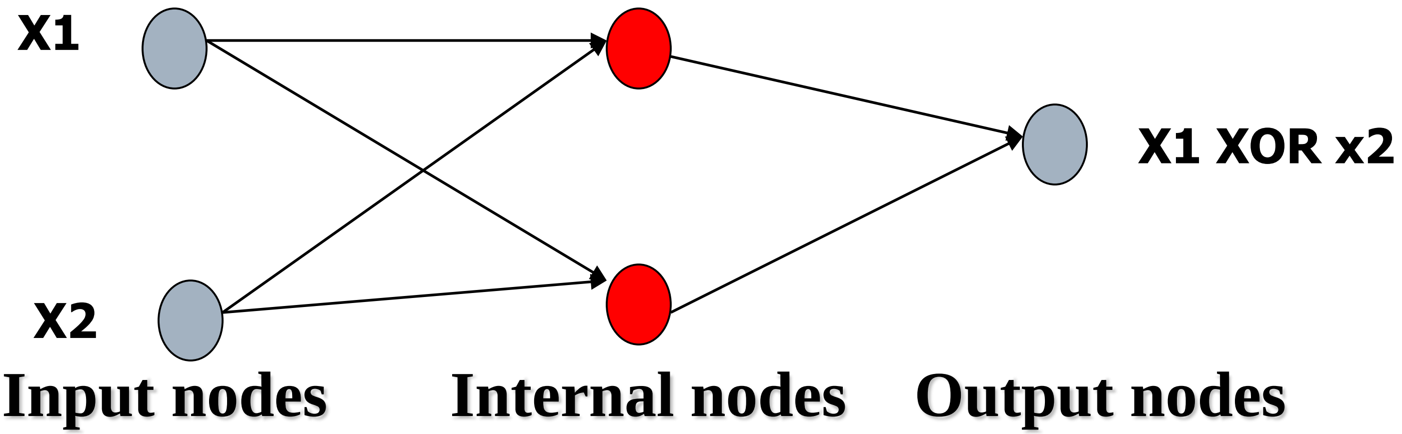
Solution for XOR : Add a hidden layer !!



Input nodes Internal nodes Output nodes



Solution for XOR : Add a hidden layer !!



The problem is: How to learn Multi Layer Perceptrons??

Solution: Backpropagation Algorithm invented by Rumelhart and colleagues in 1986

Problems

- How do we train a multi-layered network?
- What is the desired output of hidden neurons?

This problem remained unsolved for many years, causing the so called “AI winter”.

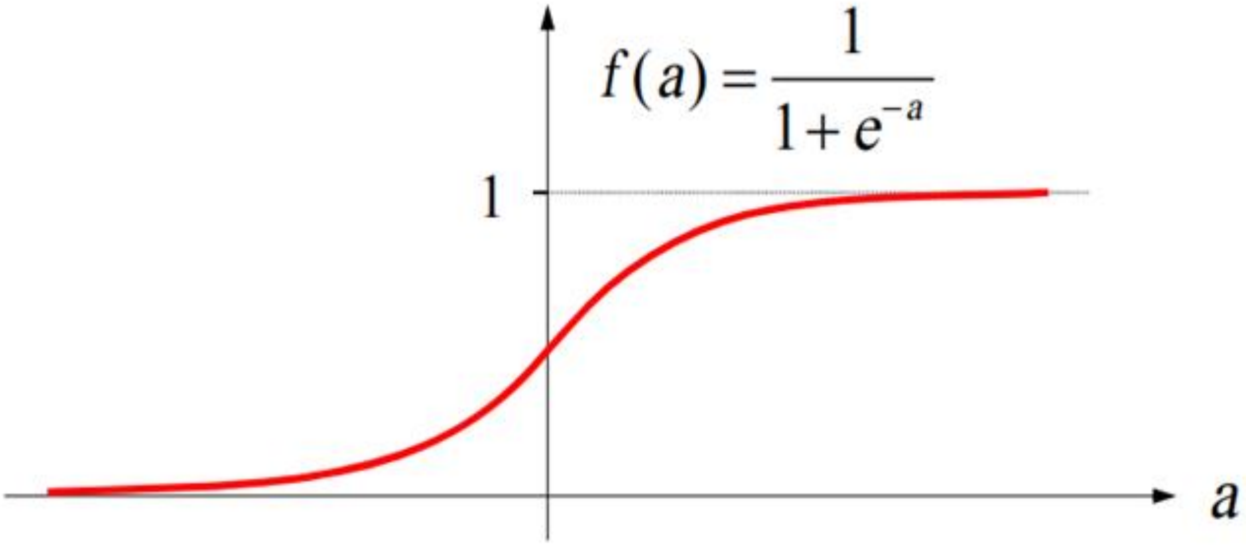
Rumelhart

Hinton

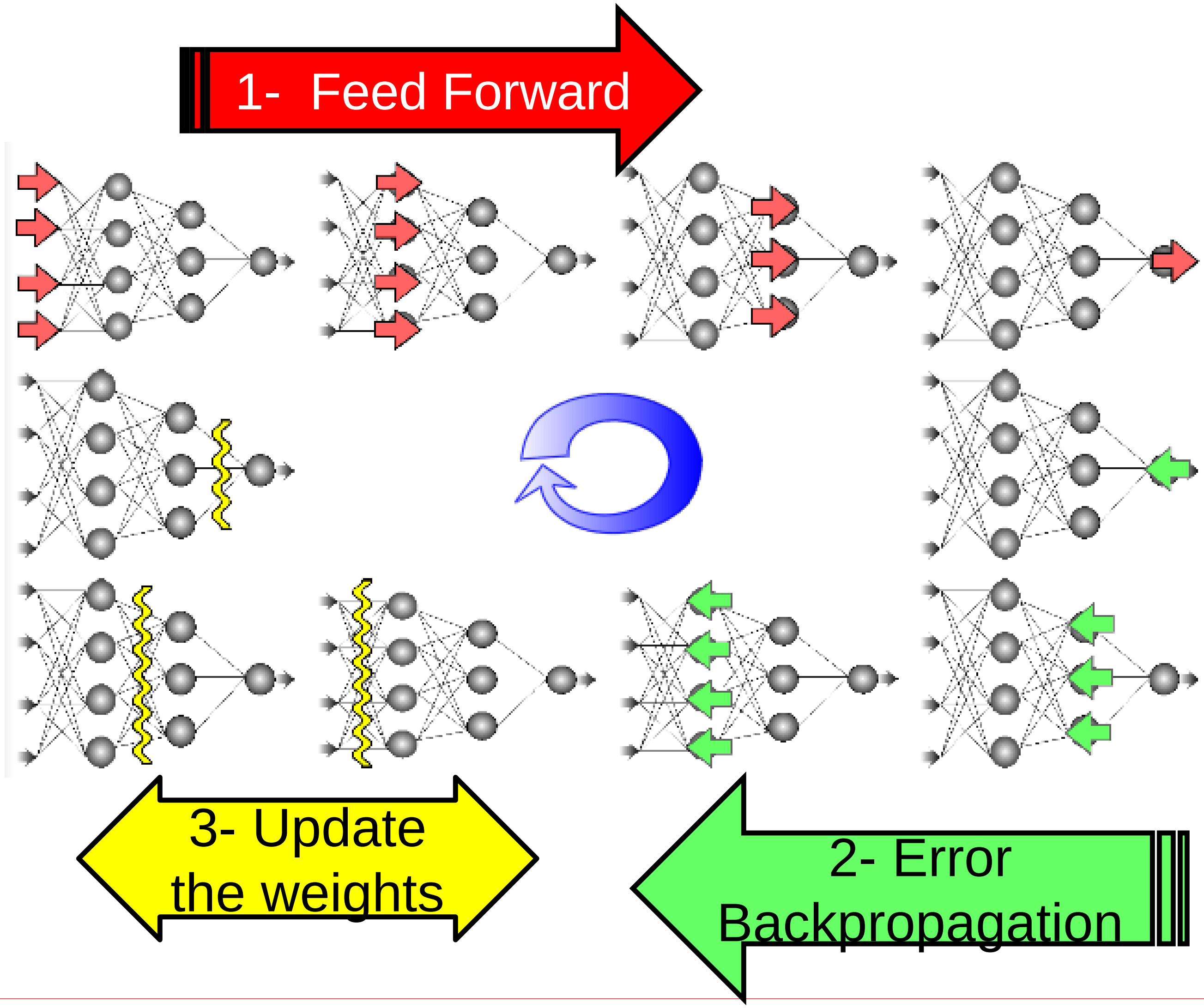
Backpropagation learning Algorithm

(Rumelhart-Hinton-Williams, 1986)

- *Multi-layer feedforward networks;*
- *Output is generated using a sigmoid function:* $y_j = f(a_j)$



Backpropagation- Algorithm



1- Feed Forward

2- Error Backpropagation

3- Update the weights

Backpropagation: Objectives

- **Learning**

  Teach the network a set of desired associations $(\boldsymbol{x}_k, \boldsymbol{t}_k)$ provided by the **Training Set**.

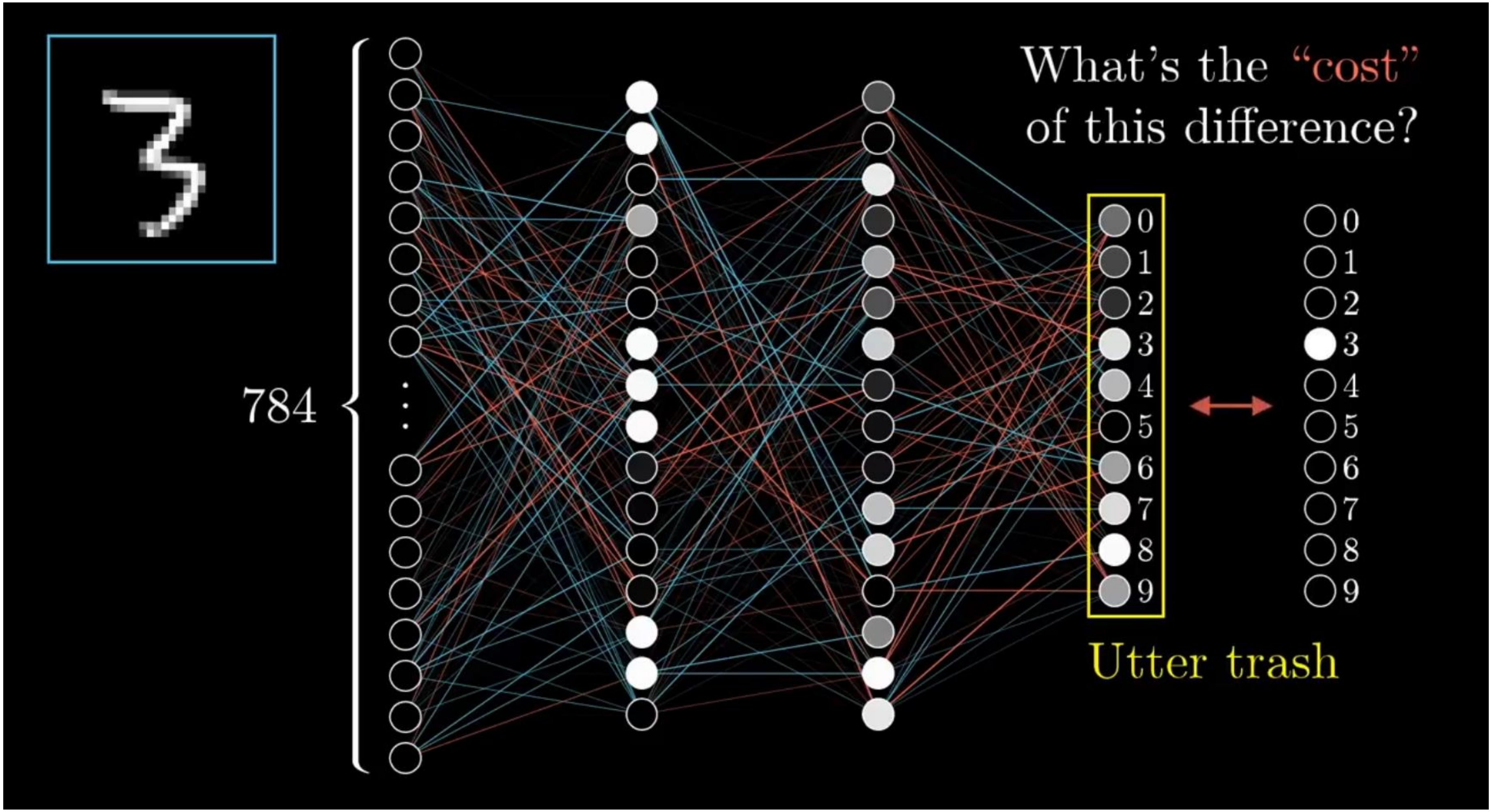
- **Convergence**

  Reduce the global error E by changing weights, such that $E < \varepsilon$, in a finite amount of time.

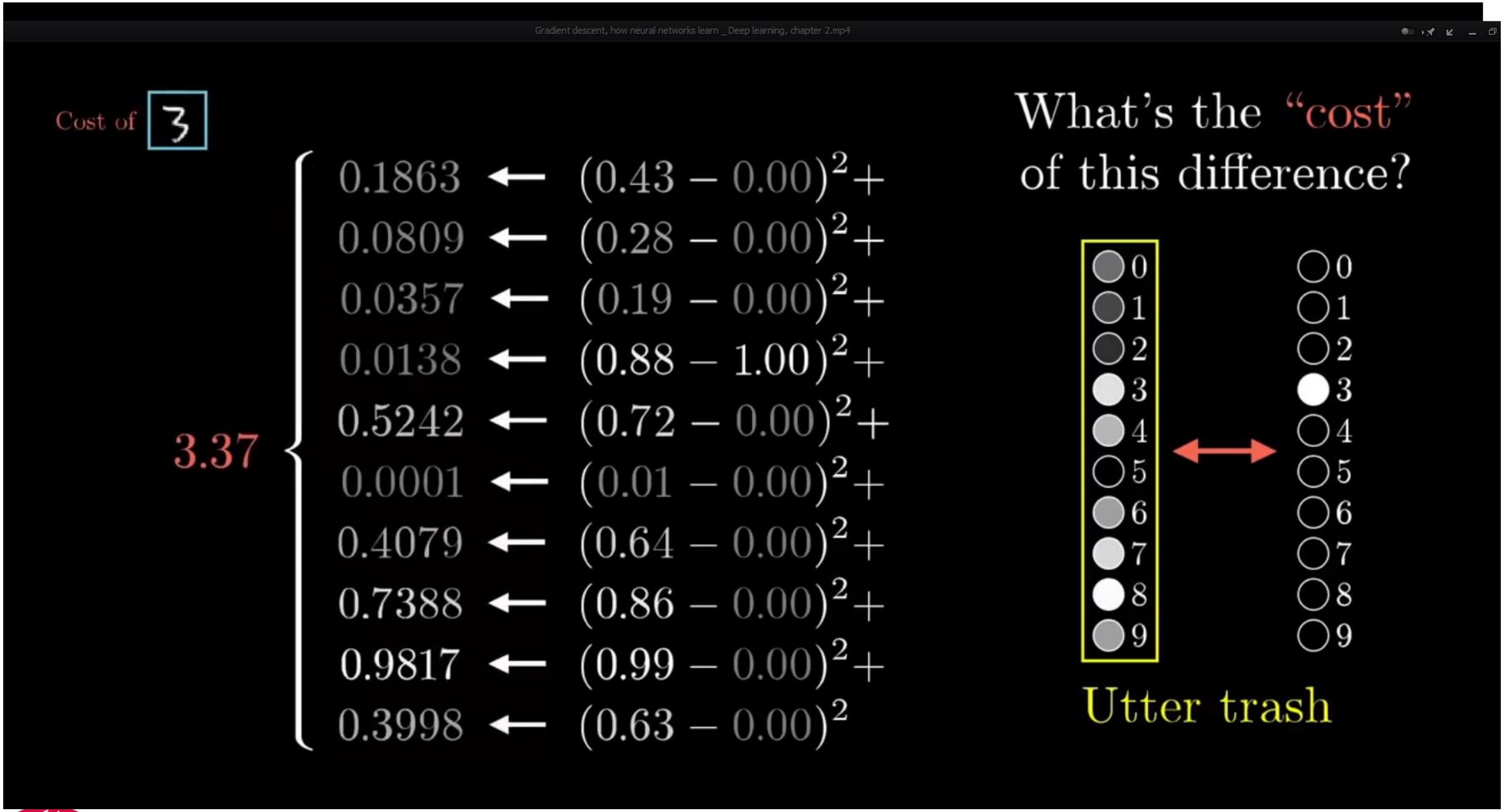
- **Generalization**

  Make the network to respond well on inputs that were never shown (i.e., not in the Training Set).

Backpropagation (Error or cost)



Backpropagation (Error or cost)



Gradient descent, how neural networks learn _ Deep learning, chapter 2.mp4

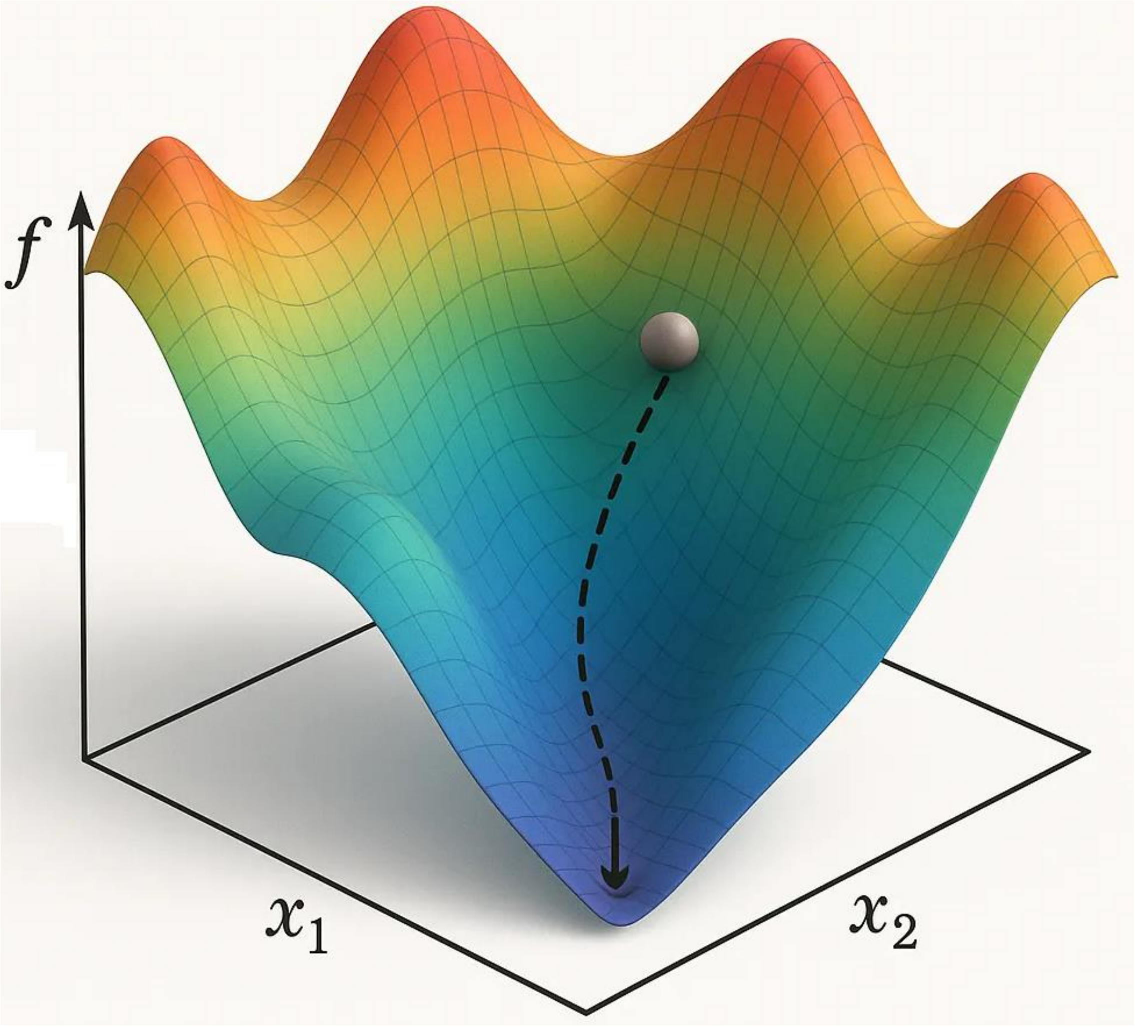
Cost of $\boxed{3}$

$$
3.37 \left\{
\begin{array}{lcl}
0.1863 & \leftarrow & (0.43-0.00)^2+ \\
0.0809 & \leftarrow & (0.28-0.00)^2+ \\
0.0357 & \leftarrow & (0.19-0.00)^2+ \\
0.0138 & \leftarrow & (0.88-1.00)^2+ \\
0.5242 & \leftarrow & (0.72-0.00)^2+ \\
0.0001 & \leftarrow & (0.01-0.00)^2+ \\
0.4079 & \leftarrow & (0.64-0.00)^2+ \\
0.7388 & \leftarrow & (0.86-0.00)^2+ \\
0.9817 & \leftarrow & (0.99-0.00)^2+ \\
0.3998 & \leftarrow & (0.63-0.00)^2
\end{array}
\right.
$$

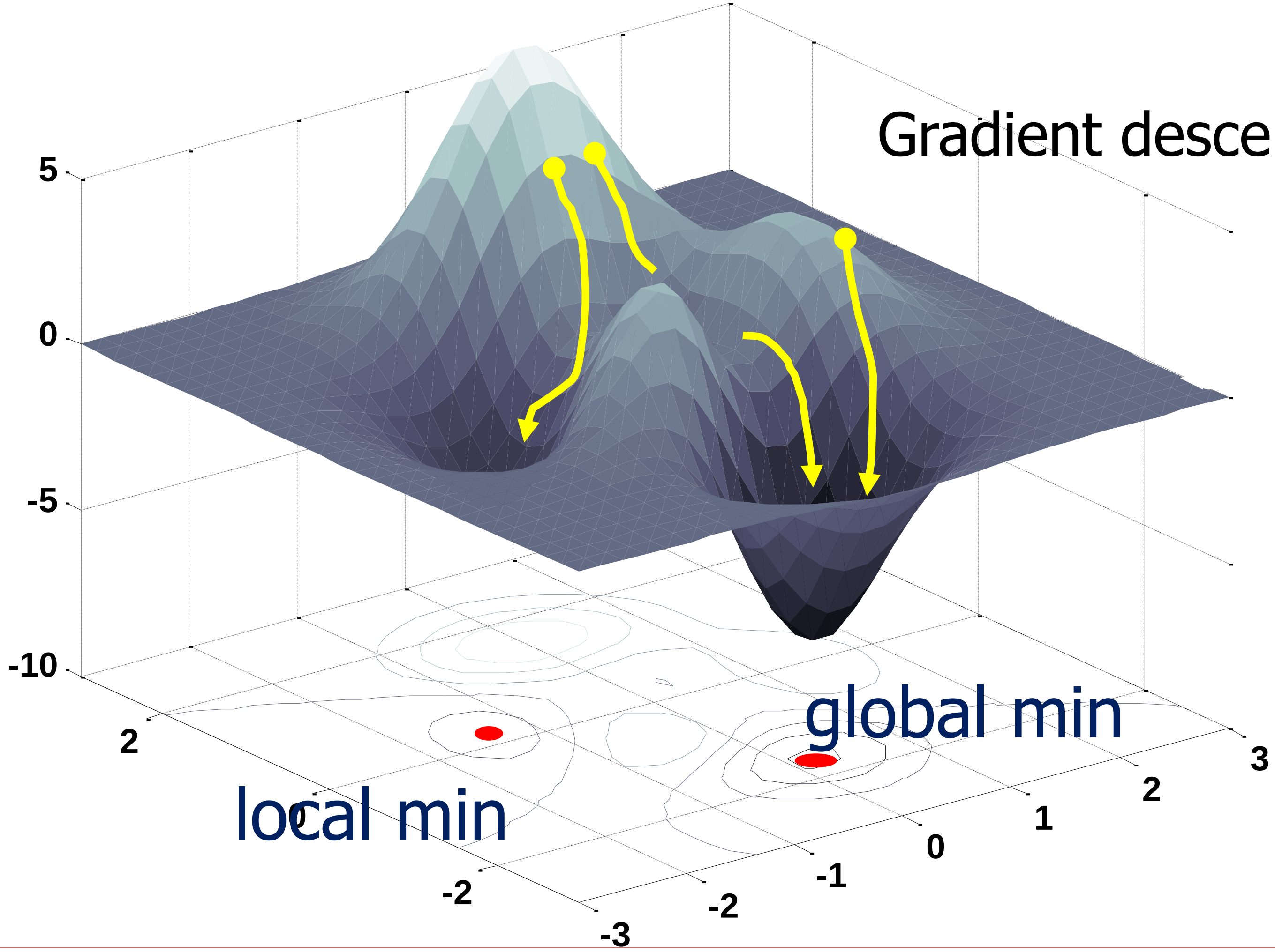
What's the "cost" of this difference?

Utter trash

Error space (Multi-Modal Cost Surface)



Error space (Multi-Modal Cost Surface)

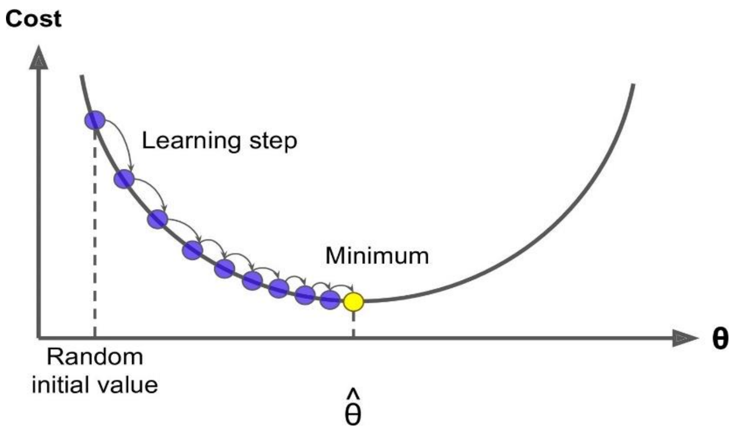


Gradient Descent

- We have a cost function $J(\theta)$ we want to minimize
  - We can use Gradient Descent algorithm!

- **Idea:** for current value of θ, calculate gradient of $J(\theta)$, then take small step in direction of negative gradient. Repeat.

- Note: Our objectives may not be convex like this. But life turns out to be okay!



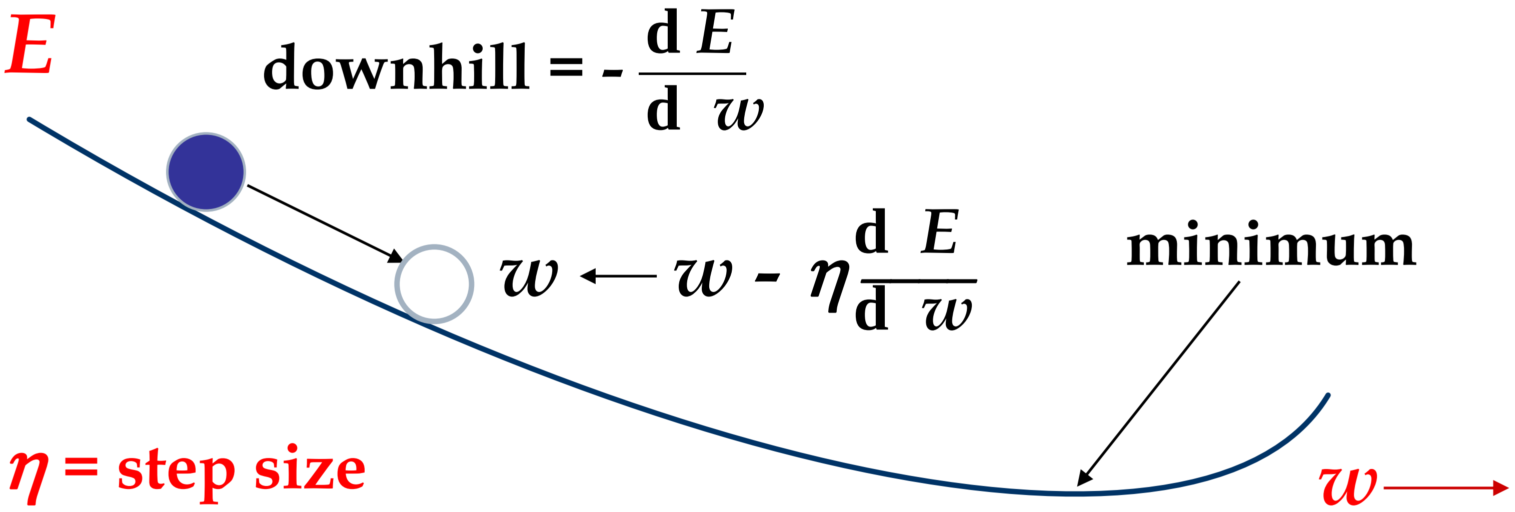
Gradient Descent: Intuition

- Imagine you’re blindfolded
- Need to walk down a hill
- You can use your hands to find the directions that may be downhill

[slide: Andrej Karpathy]

Minimizing Error Using Steepest Descent

- The main idea: Find the way downhill and take a step:



E

$$\text{downhill} = -\frac{dE}{dw}$$

$$w \leftarrow w - \eta \frac{dE}{dw}$$

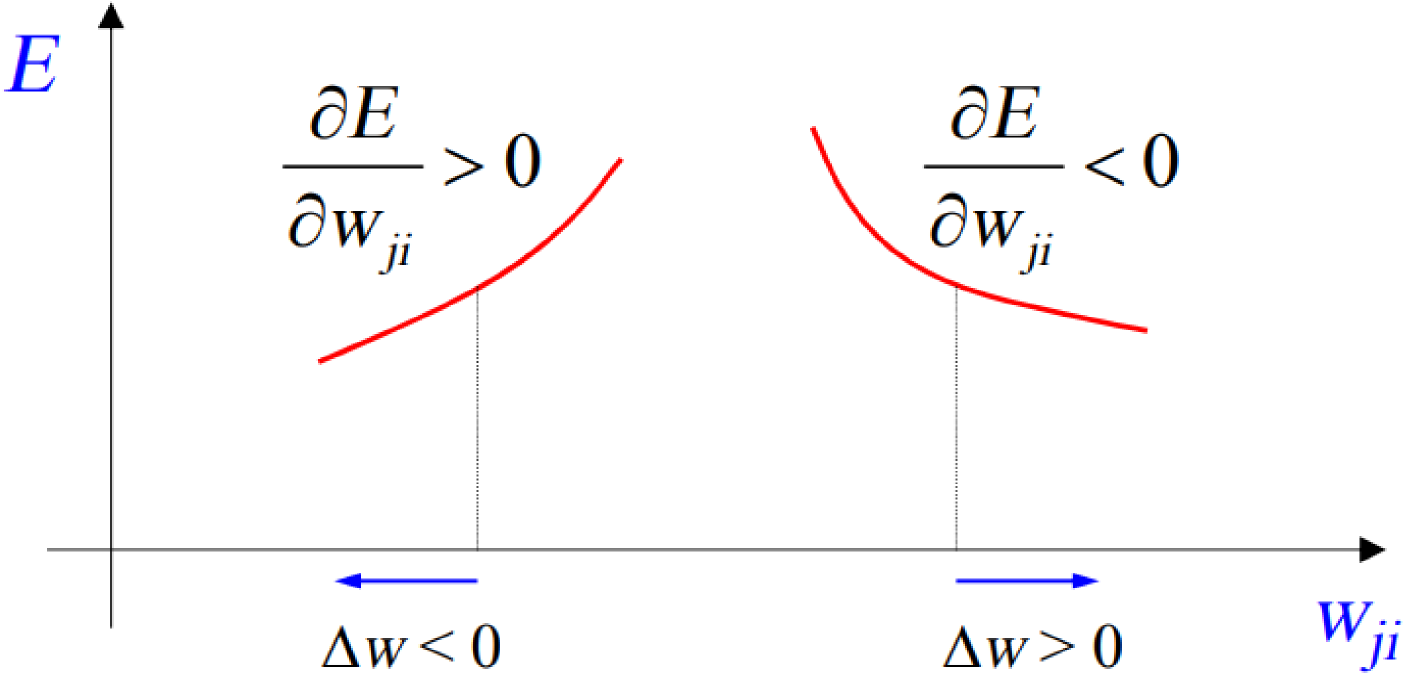
minimum

η = step size

$w \longrightarrow$

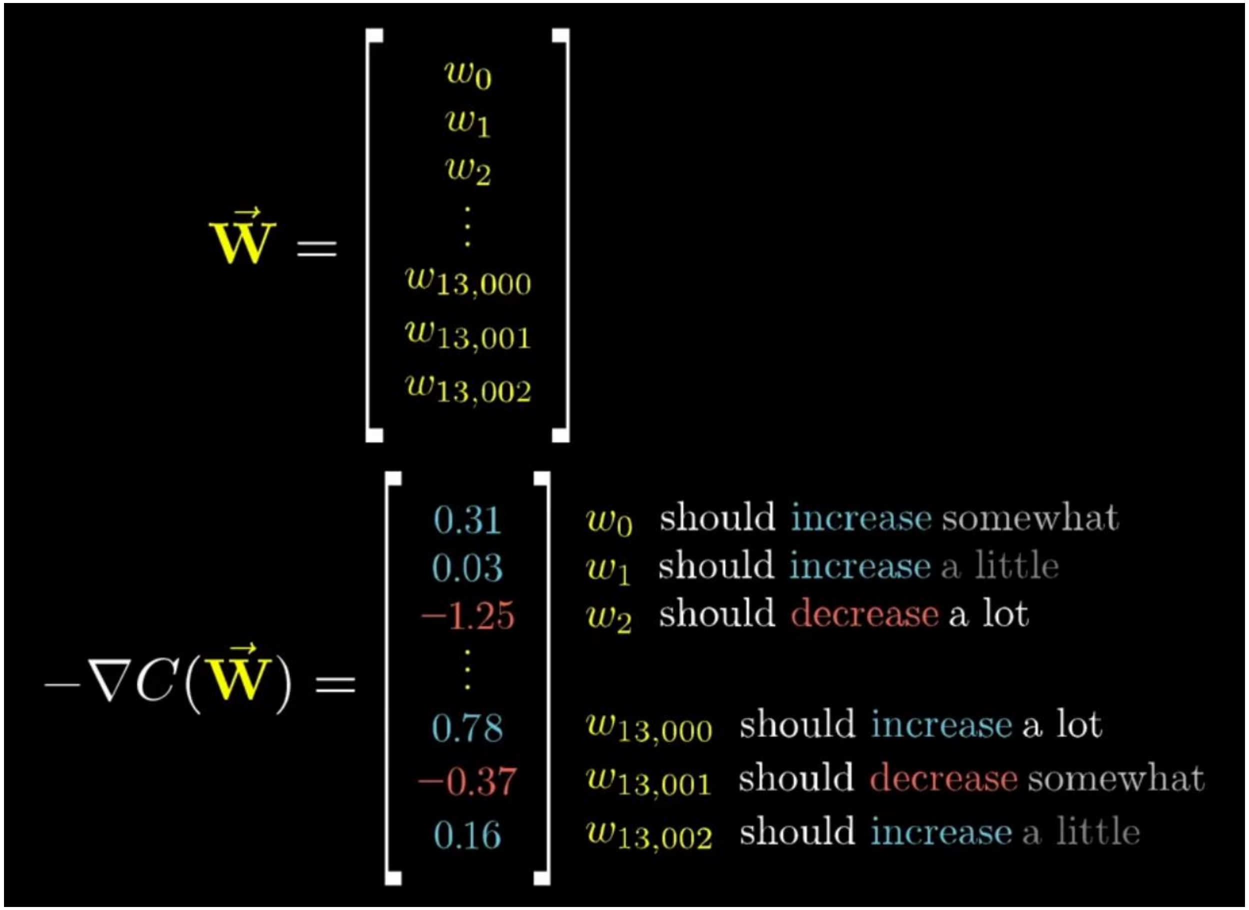
Convergence

To reduce the error by changing weights, the following strategy is adopted:



E
$\frac{\partial E}{\partial w_{ji}} > 0$
$\frac{\partial E}{\partial w_{ji}} < 0$
$\Delta w < 0$
$\Delta w > 0$
w_{ji}

Weight updates



$$\vec{\mathbf{W}} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{13,000} \\ w_{13,001} \\ w_{13,002} \end{bmatrix}$$

$$-\nabla C(\vec{\mathbf{W}}) = \begin{bmatrix} 0.31 \\ 0.03 \\ -1.25 \\ \vdots \\ 0.78 \\ -0.37 \\ 0.16 \end{bmatrix}$$

w_0 should increase somewhat

w_1 should increase a little

w_2 should decrease a lot

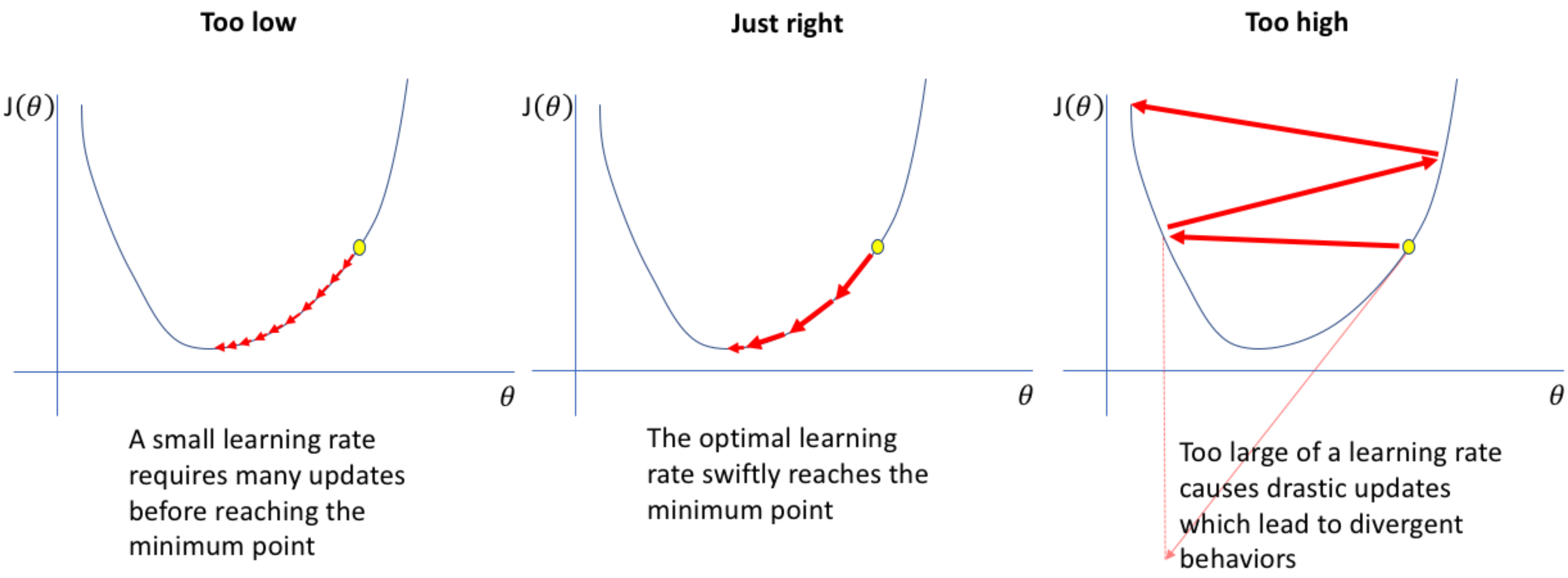
$w_{13,000}$ should increase a lot

$w_{13,001}$ should decrease somewhat

$w_{13,002}$ should increase a little

Gradient Descent: Setting the Step Size

- What is a good value for step size η?



Too low

A small learning rate requires many updates before reaching the minimum point

Just right

The optimal learning rate swiftly reaches the minimum point

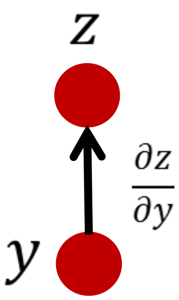
Too high

Too large of a learning rate causes drastic updates which lead to divergent behaviors

Derivatives

- First let’s get the notation right:

- The arrow shows functional dependence of z on y, i.e. given y, we can calculate z.
  - For example: $z(y) = 2y^2$

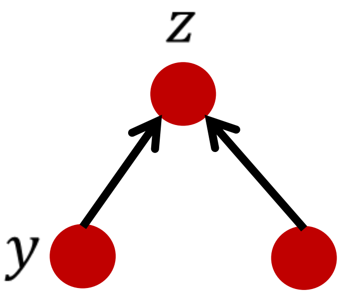


- The derivative of z, with respect to y: $\frac{\partial z}{\partial y}$

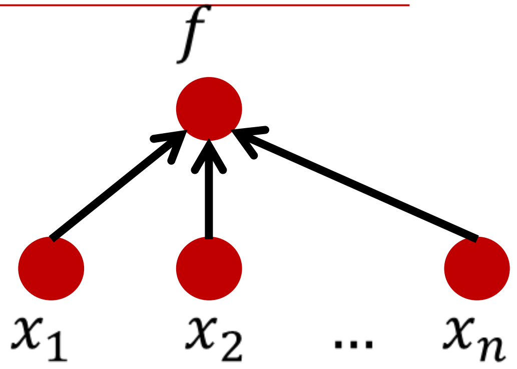
Quiz time!

- If $z(x, y) = y^4 x^5$ what is the following derivative $\frac{\partial z}{\partial y}$?

  1. $\frac{\partial \mathrm{z}}{\partial \mathrm{y}} = 4\mathrm{y}^3\mathrm{x}^5$
  2. $\frac{\partial \mathrm{z}}{\partial \mathrm{y}} = 5\mathrm{y}^4\mathrm{x}^4$
  3. $\frac{\partial \mathrm{z}}{\partial \mathrm{y}} = 20\mathrm{y}^3\mathrm{x}^4$
  4. None of the above



Gradient



- Given a function with 1 output and n inputs

$$f(\mathbf{x}) = f(x_1, x_2, \dots, x_n) \in \mathbb{R}$$

- Its gradient is a vector of partial derivatives with respect to each input

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n$$

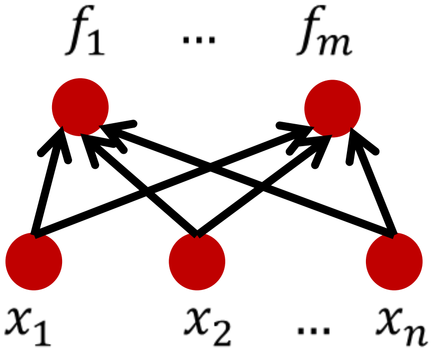
(always assume vectors are **column vectors**, i.e., they're in $\mathbb{R}^{n \times 1}$)

$$\nabla f(x, y) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 4x \\ 2y \end{bmatrix}$$

$$z = f(x, y) = 2x^2 + y^2$$

$$\nabla f(1, 4) = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

Jacobian Matrix: Generalization of the Gradient



- Given a function with ***m* outputs** and ***n* inputs**

$$\mathbf{f}(\mathbf{x}) = [f_1(x_1, x_2, \dots, x_n), \dots, f_m(x_1, x_2, \dots, x_n)] \in \mathbb{R}^m$$

- It's Jacobian is an ***m* x *n* matrix** of partial derivatives: $\left(\mathbf{J_f}(\mathbf{x})\right)_{ij} = \frac{\partial f_i}{\partial x_j}$

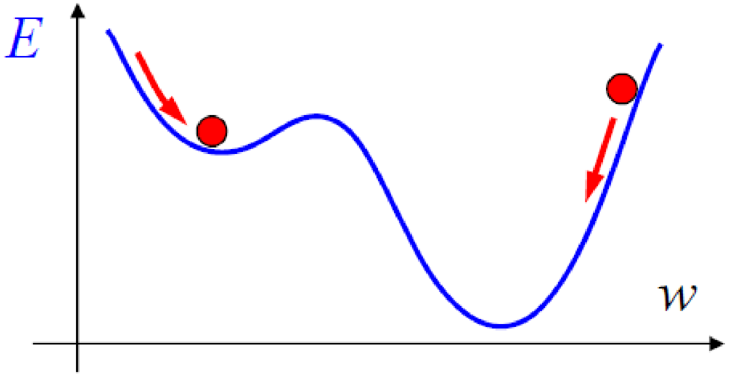
$$\mathbf{J_f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Local vs. global minimum

There are two approaches to escape local minima:

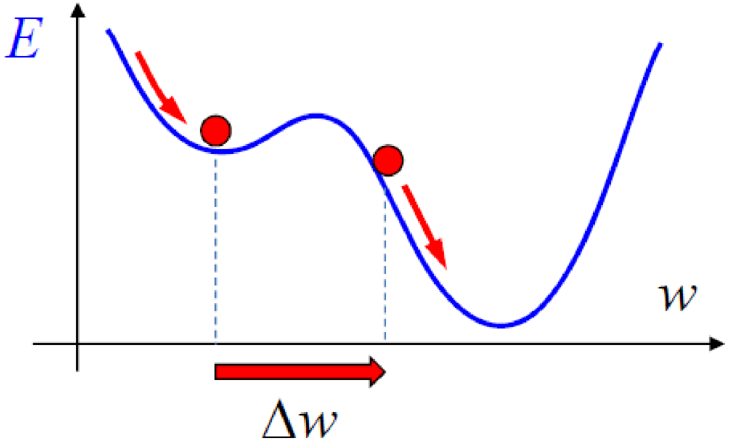
1. Reset the network

Restart training with new random weights. You may be luckier!

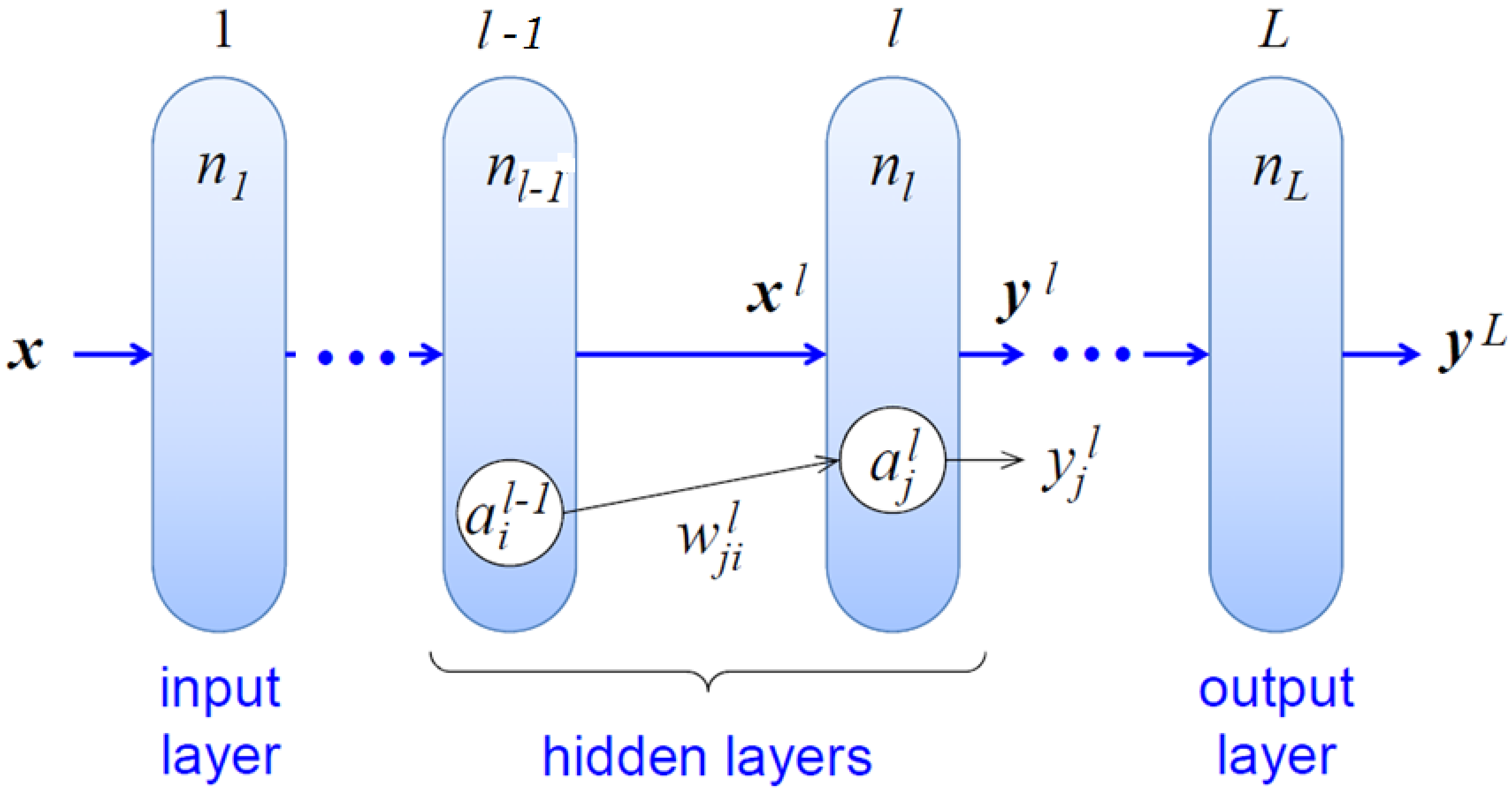


2. Make a random jump

Add a random value to the weights. It may be enough to escape.



Backpropagation- Algorithm



Updating weights

Therefore, the weights are changed according to the following law:

$$\boxed{\Delta \boldsymbol{w}_{ji} = -\eta \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w}_{ji}}}$$

Gradient rule

η = Learning coefficient (learning rate)

Back-propagating error

$$\Delta w_{ji}(k) = -\eta \frac{\partial E_k}{\partial w_{ji}} = -\eta \frac{\partial E_k}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

being $a_j = \sum_{i=1}^{n} w_{ji} x_i + b_j$ we have: $\boxed{\frac{\partial a_j}{\partial w_{ji}} = x_i}$

and by defining $\boxed{\delta_j \overset{\Delta}{=} -\frac{\partial E_k}{\partial a_j}}$ then: $\boxed{\Delta w_{ji}(k) = \eta \, \delta_j \, x_i}$

So, now the problem is to compute δ_j for each neuron.

Computing δ$_j$ (for output layer)

$$\delta_j^L = -\frac{\partial E_k}{\partial a_j} = -\frac{\partial E_k}{\partial y_j}\frac{\partial y_j}{\partial a_j}$$

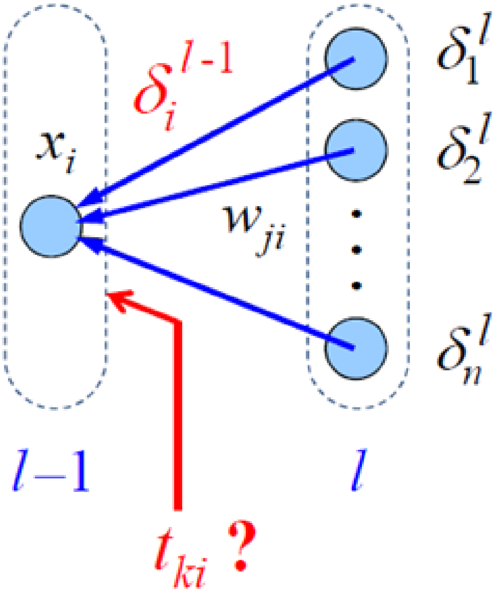
being $E_k = \frac{1}{2}\sum_{j=1}^{n_L}(t_{kj} - y_j^L)^2$ we have: $\boxed{\frac{\partial E_k}{\partial y_j} = -(t_{kj} - y_j^L)}$

and since $\frac{\partial y_j}{\partial a_j} = f'(a_j)$ we obtain:

$$\boxed{\delta_j^L = (t_{kj} - y_j^L)f'(a_j^L)}$$

Computing δ_j (for hidden layer)

Note that the same formula $\delta_i^{l-1} = (t_{ki} - y_i^{l-1}) f'(a_i^{l-1})$ cannot be used for hidden neurons, since t_{ki} is unknown!



Hence the idea is to backpropagate the errors back through the weights to assign a blame to hidden neurons.

$$\boxed{\delta_i^{l-1} = f'(a_i^{l-1}) \sum_{j=1}^{n_l} w_{ji}^l \delta_j^l}$$

Updating weights

Generalized Delta Rule:

$$\Delta w_{ji}(k) = \eta \delta_j x_i$$

For the output neurons:

$$\delta_j^L = (t_{kj} - y_j^L) f'(a_j^L)$$

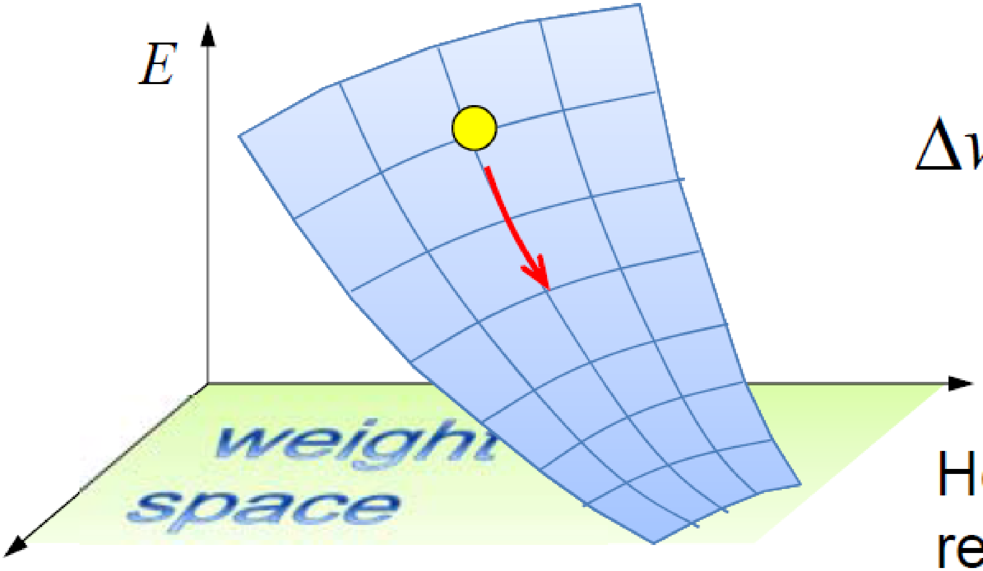
For the hidden neurons

$$\delta_i^{l-1} = f'(a_i^{l-1}) \sum_{j=1}^{n_l} w_{ji}^l \delta_j^l$$

Back Propagation: Algorithm

```
randomly initialize the weights;
do {
    initializes the global error $E = 0$;
    for each $(X_k, t_k) \in TS$ {
        compute $y_k$ and error $E_k$ ;
        compute $\delta_j$ on the output layer;
        compute $\delta_{the}$ on the hidden layer;
        update weights of the network: $\Delta w = \eta\delta x$ ;
        updates the global error: $E = E + E_k$ ; }
} while $(E > \varepsilon)$;
```

Minimizing the global error

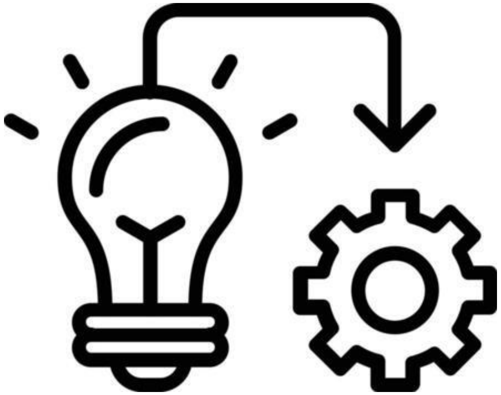


$$\Delta w_{ji} = \frac{1}{M}\sum_{k=1}^{M} \Delta w_{ji}(k)$$

Hence a single learning step requires:

- show all the M examples;
- store all the weights variations for each example;
- update all the weights after completing the training set.

A pass of the entire training set is called an **epoch**.



Session 3: Practical Implementation

NN DESIGN ISSUES

1. Architectural Design Issues

Length and depth of layers, type of layers, activation functions

2. Data-Related Issues

Quantity & Quality of Data, Feature Selection & Representation, Data Imbalance

3.Optimization & Training Issues

Choice of Loss Function, Choice of Optimizer: Learning Rate, Batch Size

4. Generalization & Regularization Issues

Overfitting, underfitting, Regularization Techniques

5. Computational and Hardware Issues

Training Time, Hardware Selection, Inference Speed

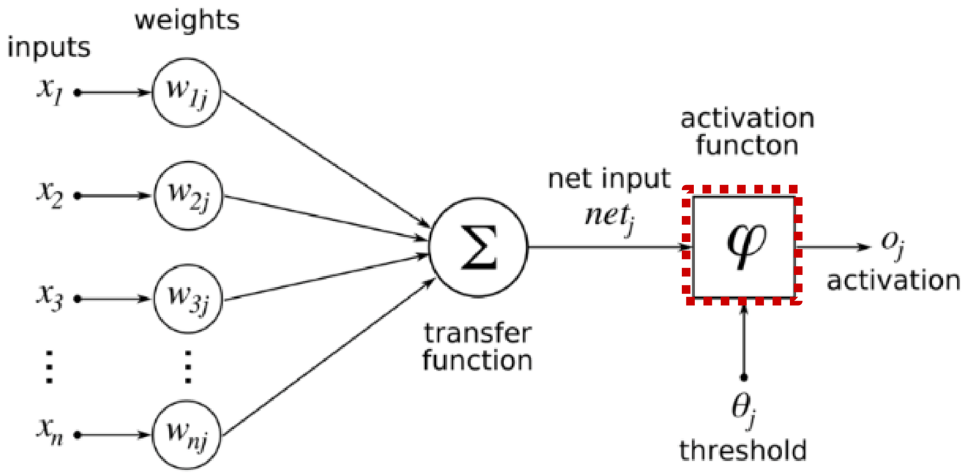
6. Interpretability and Explainability SHAP, LIME, and attention visualization

Architectural Design Issues

- The number of layers and neurons depend on the specific task.
- In practice this issue is solved by trial and error.
- Two types of adaptive algorithms can be used:
  - start from a large network and successively remove some neurons and links until network performance degrades.
  - begin with a small network and introduce new neurons until performance is satisfactory.

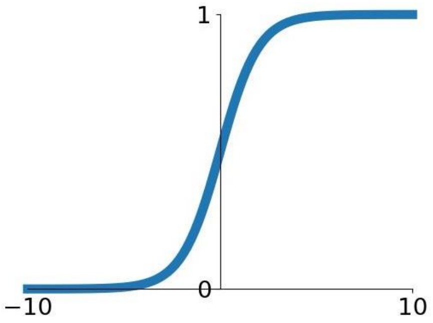
Activation Functions

- How do you choose what activation function to use?
- In general, it is problem-specific and might require trial-and-error.
- Here are some tips about popular action functions.



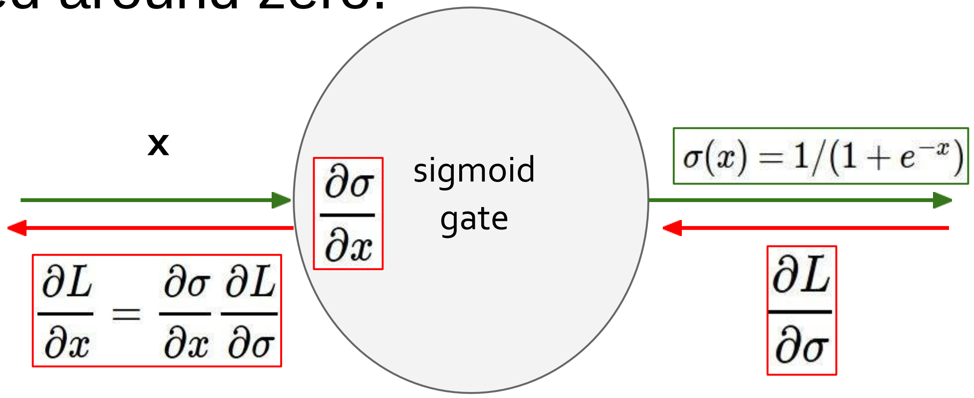
Activation Functions: Sigmoid

- Squashes numbers to range [0,1]
- Historically popular, interpretation as “firing rate” of a neuron



- Limitation 1: Saturated neurons “kill” the gradients
  - Whenever |x| > 5, the gradients are basically zero.
- Limitation 2: Not centered around zero.

If all the gradients flowing back will be zero and weights will never change.



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

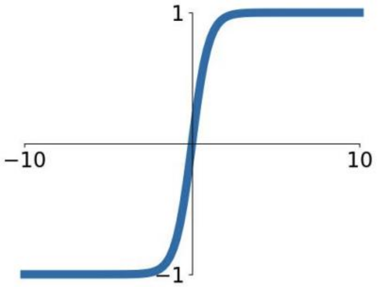
$$\frac{\partial \sigma}{\partial x}$$

$$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x}\frac{\partial L}{\partial \sigma}$$

$$\frac{\partial L}{\partial \sigma}$$

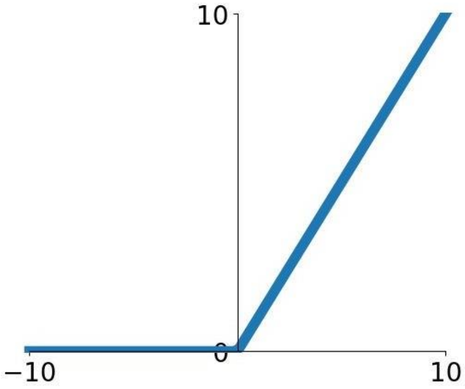
Activation Functions: Tanh & ReLU

- Symmetric around [-1, 1]
- Still saturates |x| > 3 and “kill” the gradients
- Zero-centered — faster optimization (why?)



tanh(x)

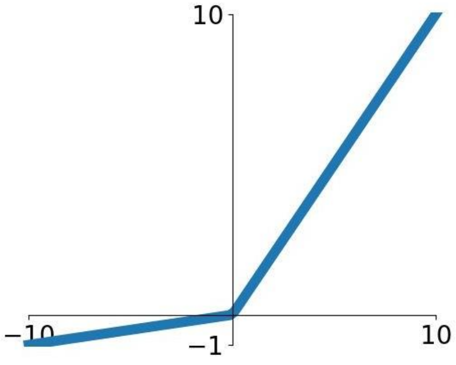
- Computationally efficient
- In practice, converges faster than sigmoid/tanh in practice
- Does not saturate (in +region) — will die less!



ReLU
(Rectified Linear Unit)

Activation Functions: Leaky ReLU

- Does not saturate — will not die.
- Computationally efficient
- In practice it converges faster than
- sigmoid/tanh in practice
- Other parametrized variants:
  - Parametric Rectifier (PReLU):
  - Maxout: $\max(w_1^T x + b_1, w_2^T x + b_2)$



$$f(x) = \max(\alpha x, x)$$

- Provide more flexibility, though at the cost of more learnable parameters.

Choose Activations: In Practice

- Other activations: ELU, Swish, GELU, Softplus, ….

- In general, it is problem-specific and might require trial-and-error.

- A useful recipe:
  1. Generally, ReLU is a good activation to start with.
  2. Time/compute permitting, you can try other activations to squeeze out more performance.

Data Representation

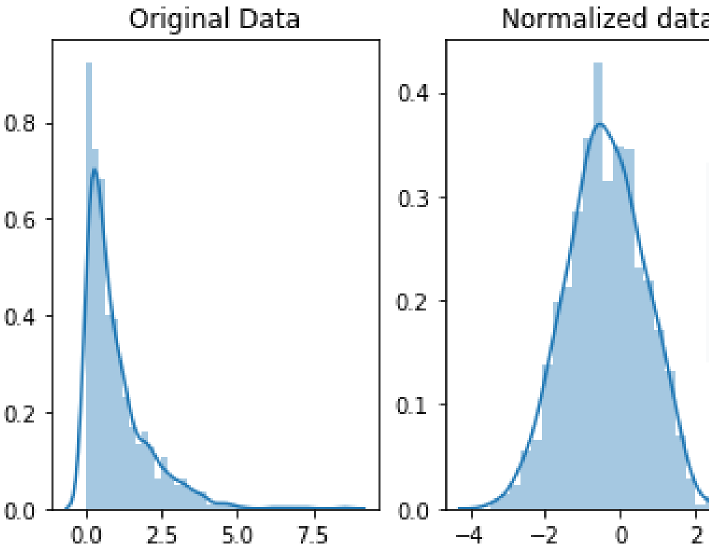
- Data representation **depends on the problem**.
- In general ANNs work on continuous (real valued) attributes. Therefore symbolic attributes are encoded into continuous ones.
- Attributes of different types may have different ranges of values which affect the training process.
- Normalization may be used, like the following one which scales each attribute to assume values between 0 and 1.

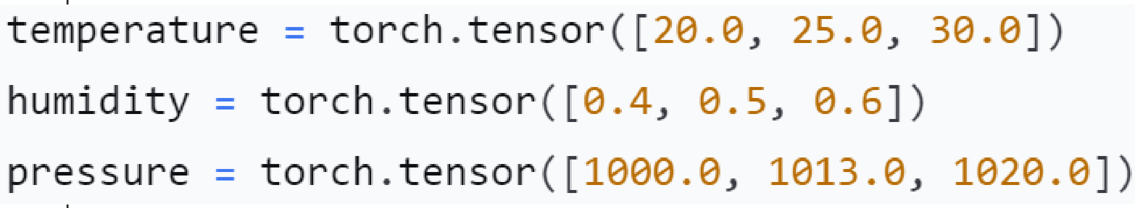
$$x_i = \frac{x_i - \min_{\mathrm{i}}}{\max_i - \min_{\mathrm{i}}}$$

for each value x_i of i^{th} attribute, $\min_i$ and $\max_i$ are the minimum and maximum value of that attribute over the training set.

Normalize Your Data!

- We do not like very large numbers.
  - Large numbers lead to numerical problems (e.g., overflow) and lead to NaNs 😭
- We prefer if our data is distributed around zero.





```
temperature = torch.tensor([20.0, 25.0, 30.0])
humidity = torch.tensor([0.4, 0.5, 0.6])
pressure = torch.tensor([1000.0, 1013.0, 1020.0])
```

Normalization

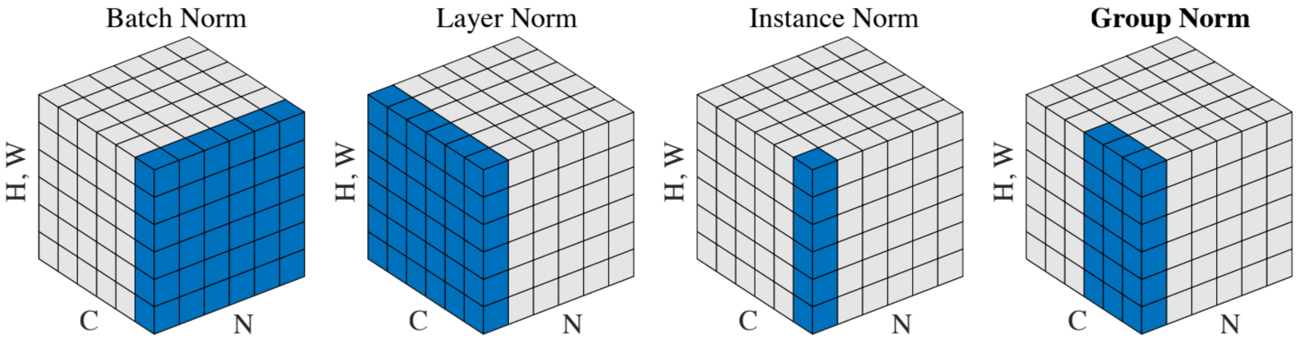
- Normalization of values standardizes the ranges of values
- Prevents value disparities
- Stabilizes + speeds up training

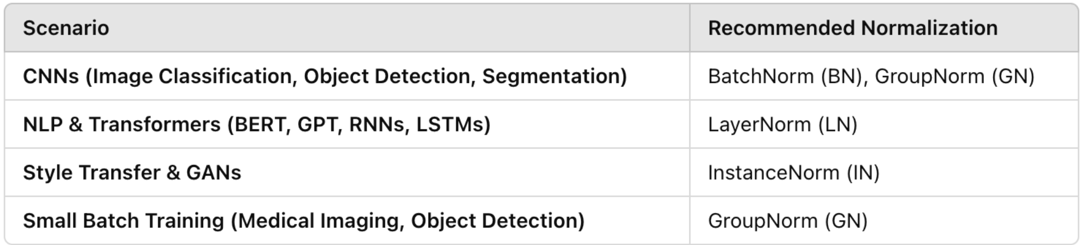
$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

H, W: height and width of images; in language, different words in your sentence.

C: channels of each RGB pixel; in language, dimensions of each word vector

N: batch size





Scenario	Recommended Normalization
CNNs (Image Classification, Object Detection, Segmentation)	BatchNorm (BN), GroupNorm (GN)
NLP & Transformers (BERT, GPT, RNNs, LSTMs)	LayerNorm (LN)
Style Transfer & GANs	InstanceNorm (IN)
Small Batch Training (Medical Imaging, Object Detection)	GroupNorm (GN)

Benefits of Normalization

Faster Training: By normalizing the inputs, we can use higher learning rates and the network converges faster.

Regularization Effect: BatchNorm, in particular, adds some noise to the network (due to mini-batch statistics) which can have a regularizing effect and reduce overfitting.

Gradient Flow: It helps in mitigating the vanishing/exploding gradient problems by keeping the activations in a stable range.

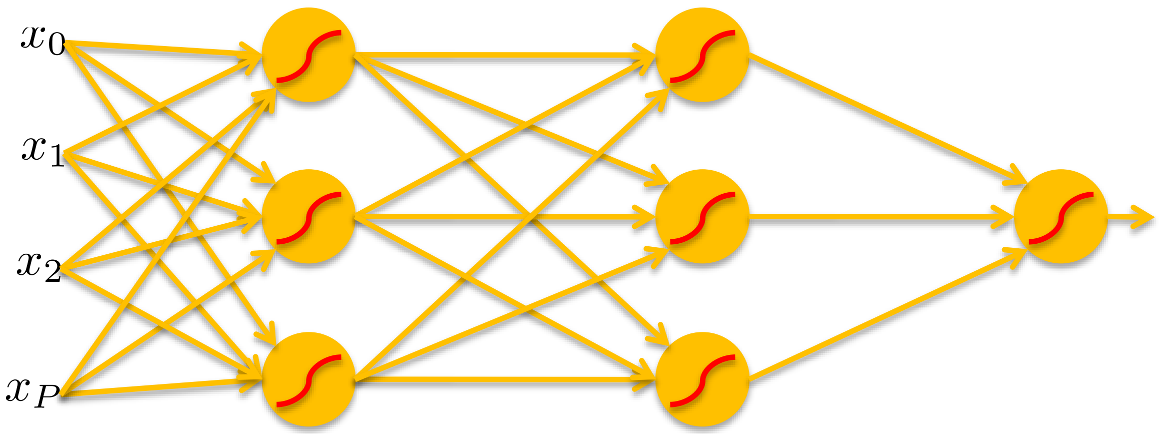
Less Sensitive to Initialization: Normalization makes the network less sensitive to the initial weight values.

Network parameters

- How are the weights initialized?
- How is the learning rate chosen?
- How many hidden layers and how many neurons?
- How many examples in the training set?

Weight Initialization

- Initializing all weights with a fixed constant (e.g., 0’s) is a very bad idea! (why?)



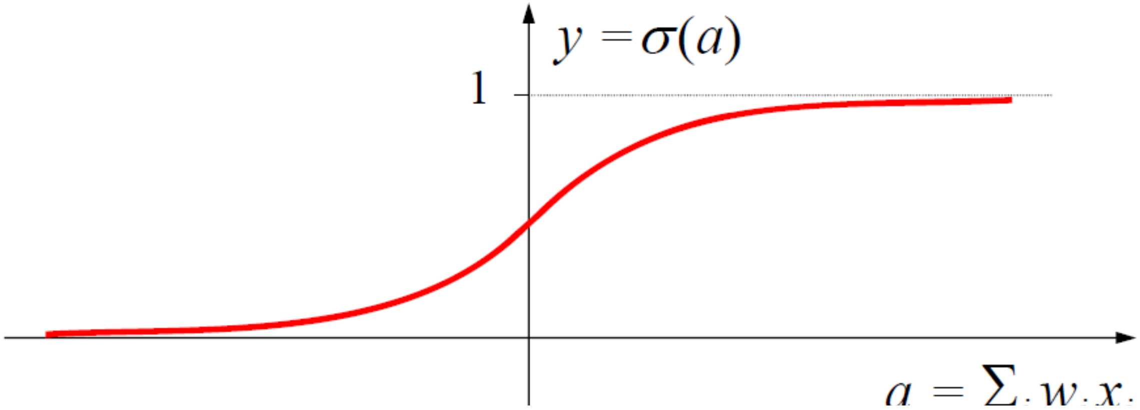
- If the neurons start with the same weights, then all the neurons will follow the same gradient, and will always end up doing the same thing as one another.
- Effective initialization is one that breaks such ”symmetries” in the weight space.

Weight Initialization

To favor the learning phase, weights have to be initialized with random small values. In fact:

$$\Delta \boldsymbol{w} = \eta \delta \boldsymbol{x} \propto f'(a)$$

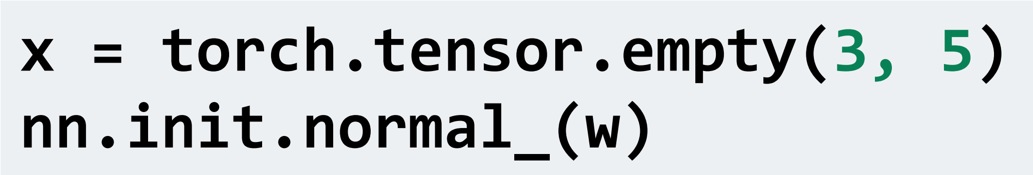
small $|w|$ $\Rightarrow$ small $|a|$ $\Rightarrow$ big $f'(a)$ $\Rightarrow$ big Δw



Hence, weights are initialized as random variables with normal distribution with mean 0 and standard deviation $\sqrt{1/n_{in}}$ where n_{in} is the number of inputs of the neuron.

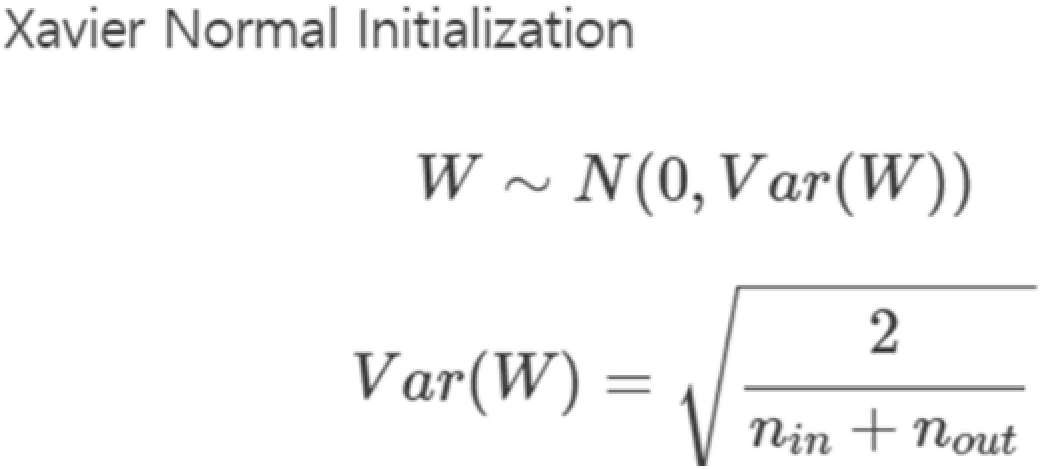
Weight Initialization

- Better idea: initialize weights with random Gaussian noise.



```
x = torch.tensor.empty(3, 5)
nn.init.normal_(w)
```

- There are fancier initializations (Xavier, Kaiming, etc.) that we won’t get into.



Xavier Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

Choice of learning rate

- The right value of η depends on the application.
- Values between 0.1 and 0.9 have been used in many applications.
- Other heuristics is that adapt η during the training as described in previous slides.
- It is common to start with large values and decrease monotonically.

  - Start with 0.9 and decrease every 5 epochs

  - Use a Gaussian function

  - $\eta = 1/k$

  - …

Size of Training set

- For traditional ML: **At least 10 samples per feature/parameter**
- In neural networks, this becomes challenging since modern models have millions of parameters, but it's a good starting point for simpler models.

- Other rule:

$$N > \frac{|W|}{(1-a)}$$

|W|= number of weights
a=expected accuracy on test set

Size of Training set

Real-World Examples

Problem Domain	Typical Dataset Size	Model Type
MNIST Digit Recognition	60,000 images	Simple CNN
CIFAR-10	50,000 images	Medium CNN
ImageNet	1.2M images	Deep CNN (ResNet)
IMDB Sentiment	25,000 reviews	LSTM/Transformer
Wikipedia Training	~3B tokens	GPT-style LLM
Common Crawl	Hundreds of billions of tokens	Modern LLMs

Hyper parameters

People distinguish two types of parameters in a neural network:

Model parameters

They are those that are found by learning:

- Weights
- Biases

Network hyper-parameters

They are those that have to be tuned to optimize learning:

- Number of hidden layers
- Number of hidden neurons
- Weight initialization range
- Training set size
- Mini-batch size
- …

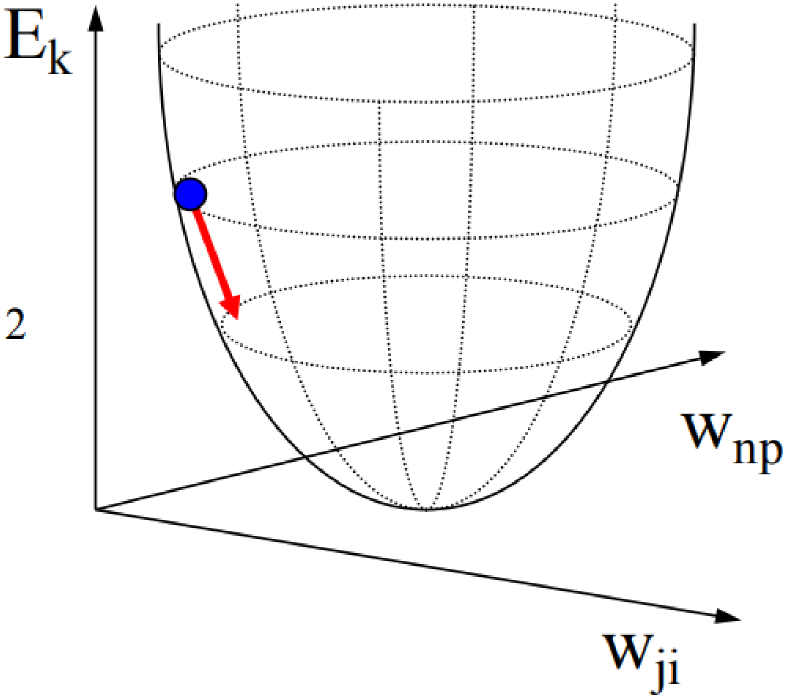
- Loss function (error)
- Activation function
- Number of epochs
- Learning rate
- Momentum
- …

Remarks: Learning Rate

- The error has a quadratic form in the space of weights:

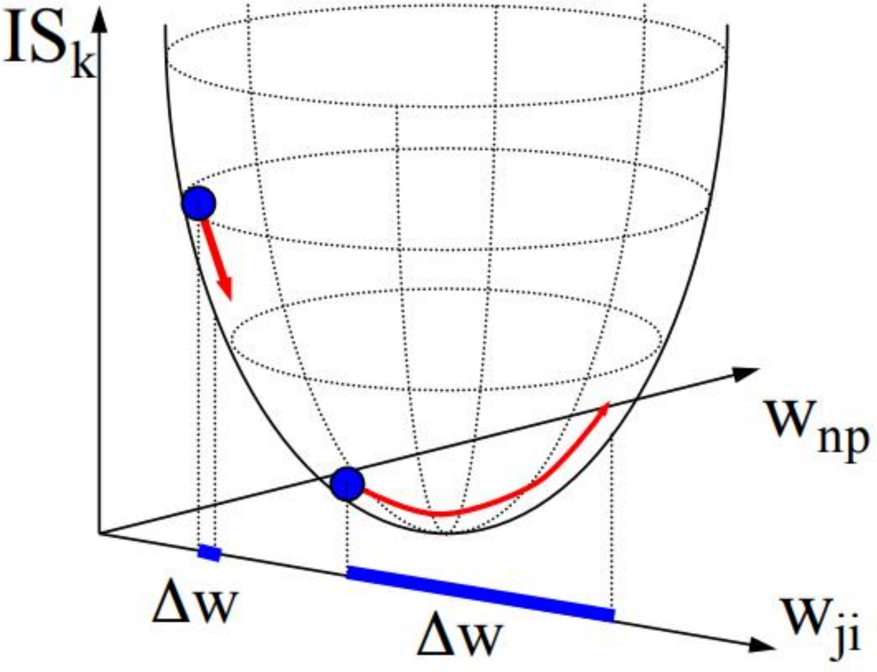
$$E_k = \sum_{j=1}^{n} (t_{kj} - y_{kj})^2$$

$$= \sum_{j=1}^{n} \left(t_{kj} - f\left(\sum_{i=1}^{p} w_{ji} x_{ki}\right) \right)^2$$



Remarks: Learning Rate

$$\boxed{\Delta w_{ji} = \eta \delta_j x_i}$$



η too small ⇒ slow learning

η too big ⇒ fluctuations

Remarks: Learning Rate

Possible solutions

- Vary η as a function of the error, to speed up convergence at the beginning and reduce oscillations at the end.
- Attenuate oscillations with a low-pass filter on the weights:

$$\boxed{\Delta w_{ji}(t) = \eta \delta_j x_i + \mu \, \Delta w_{ji}(t-1)}$$

μ is called **momentum**

Splitting data sets

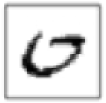
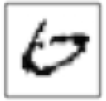
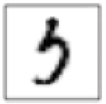
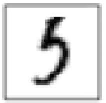
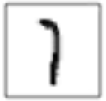
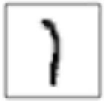
Training Set	Used for training the model parameters
Validation Set	Used for tuning the hyper-parameters (Used to decide when to stop training only by monitoring the error.)
Test Set	Used for assessing the performance of a fully-trained network

Consistency of the Train Set

If some examples are inconsistent, convergence of learning is not guaranteed:

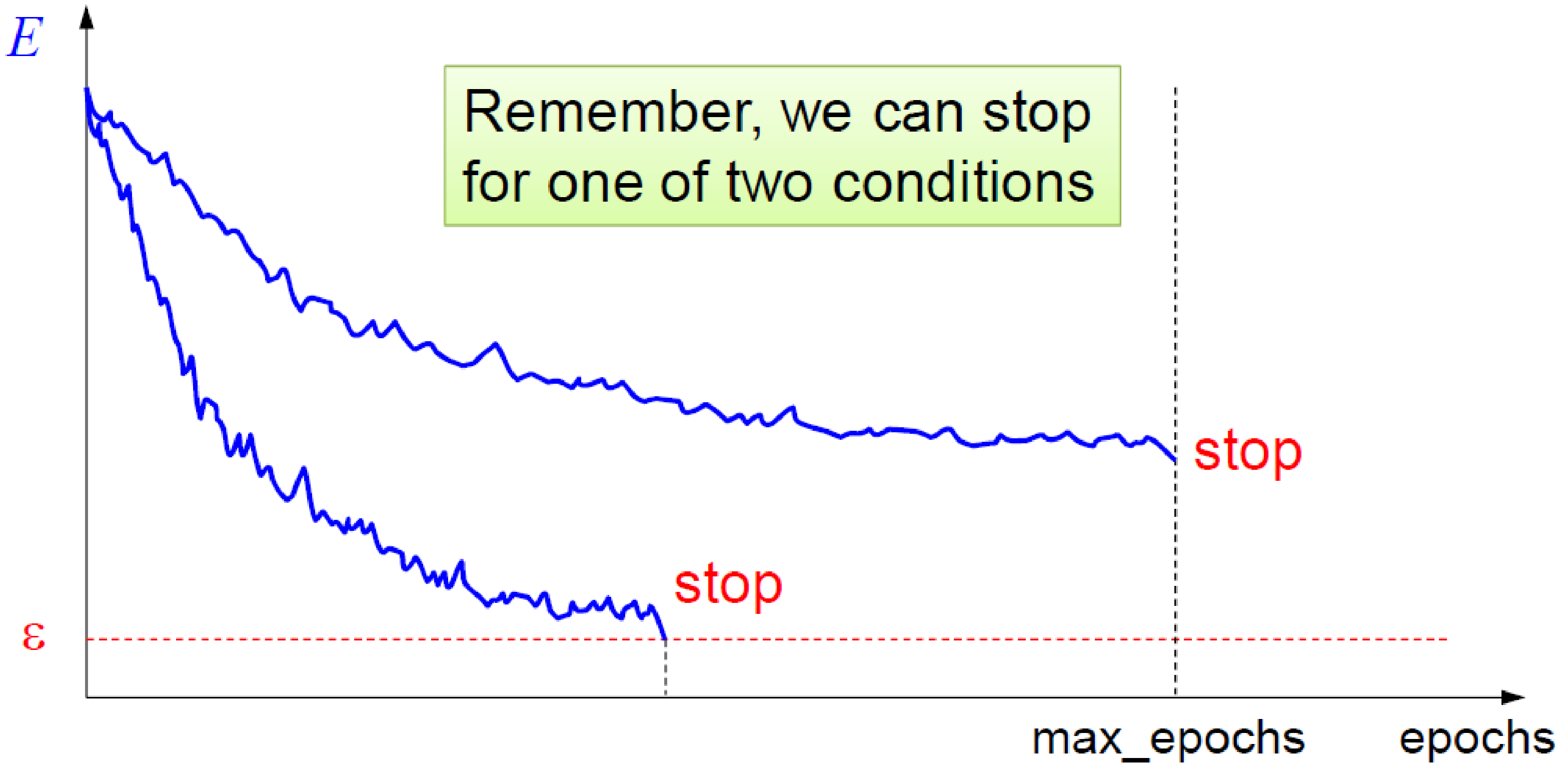
In real cases, inconsistencies can be introduced by similar noisy patterns belonging to different classes

Examples of problematic training patterns taken from images of handwritten characters:

sample	target
	0
	6
	3
	5
	7
	1

Stopping criteria

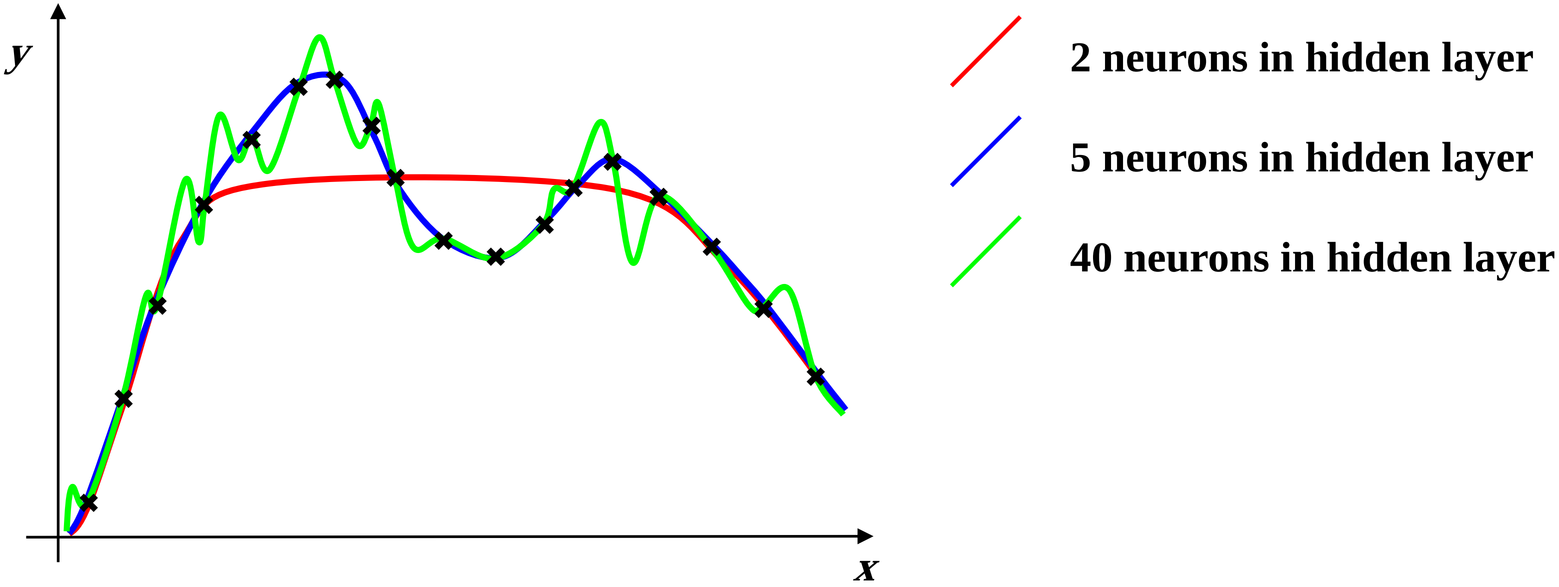
Once we have trained our network, how can we evaluate its performance?



Remember, we can stop for one of two conditions

The problem of overfitting …

- Approximation of the function $y = f(x)$:



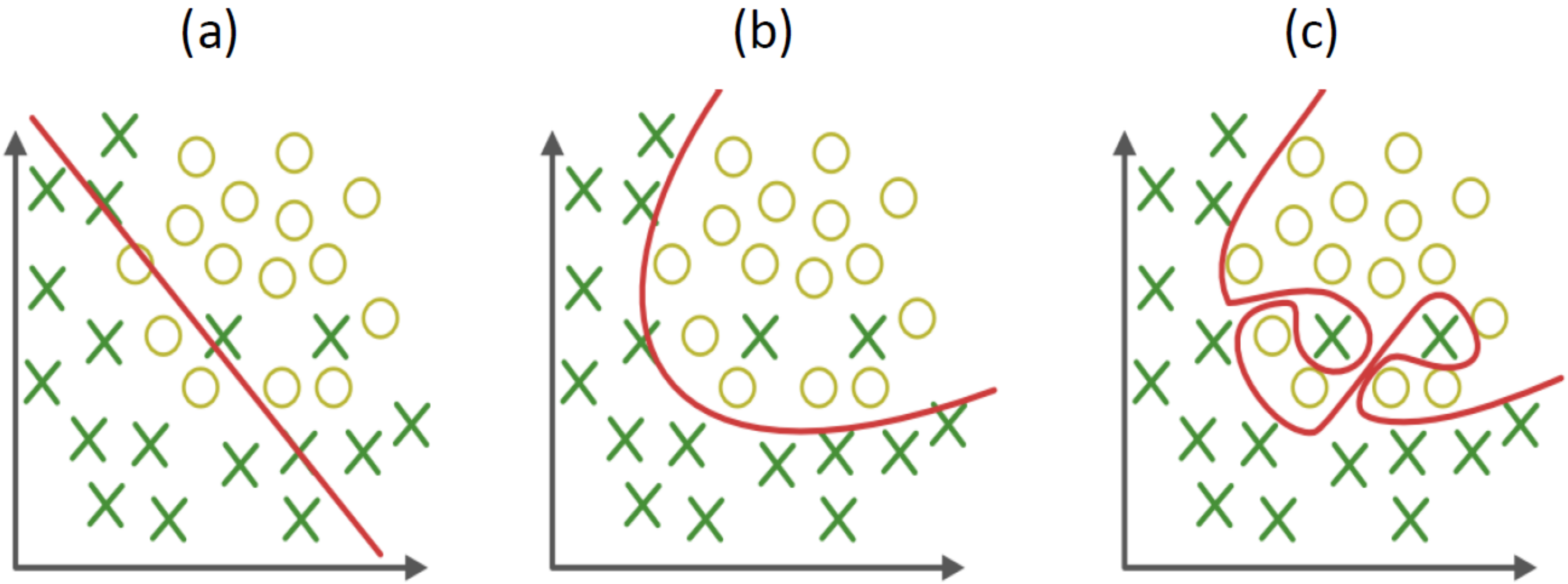
- The overfitting is not detectable in the learning phase …

Overfitting and underfitting

Which model would you choose to separate x from o?

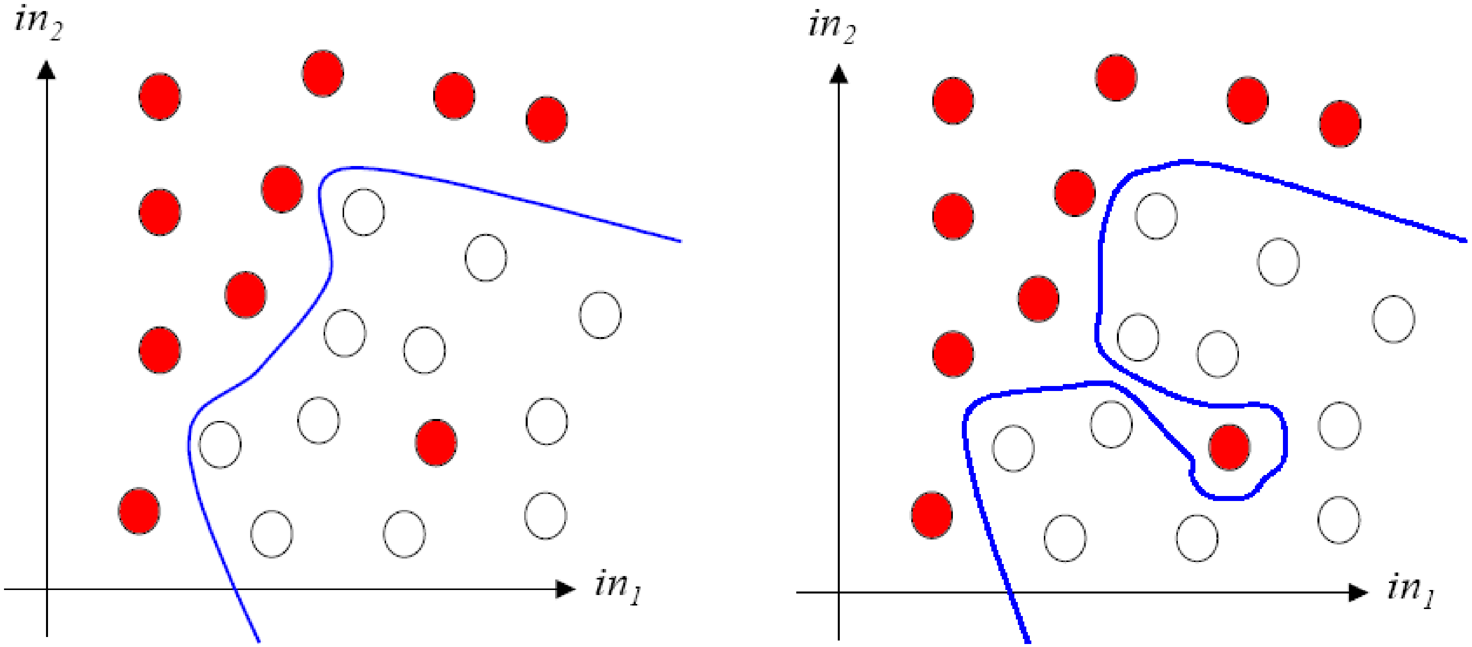
Underfits: too simple to explain the data

Overfits: too complex to generalize to a test set



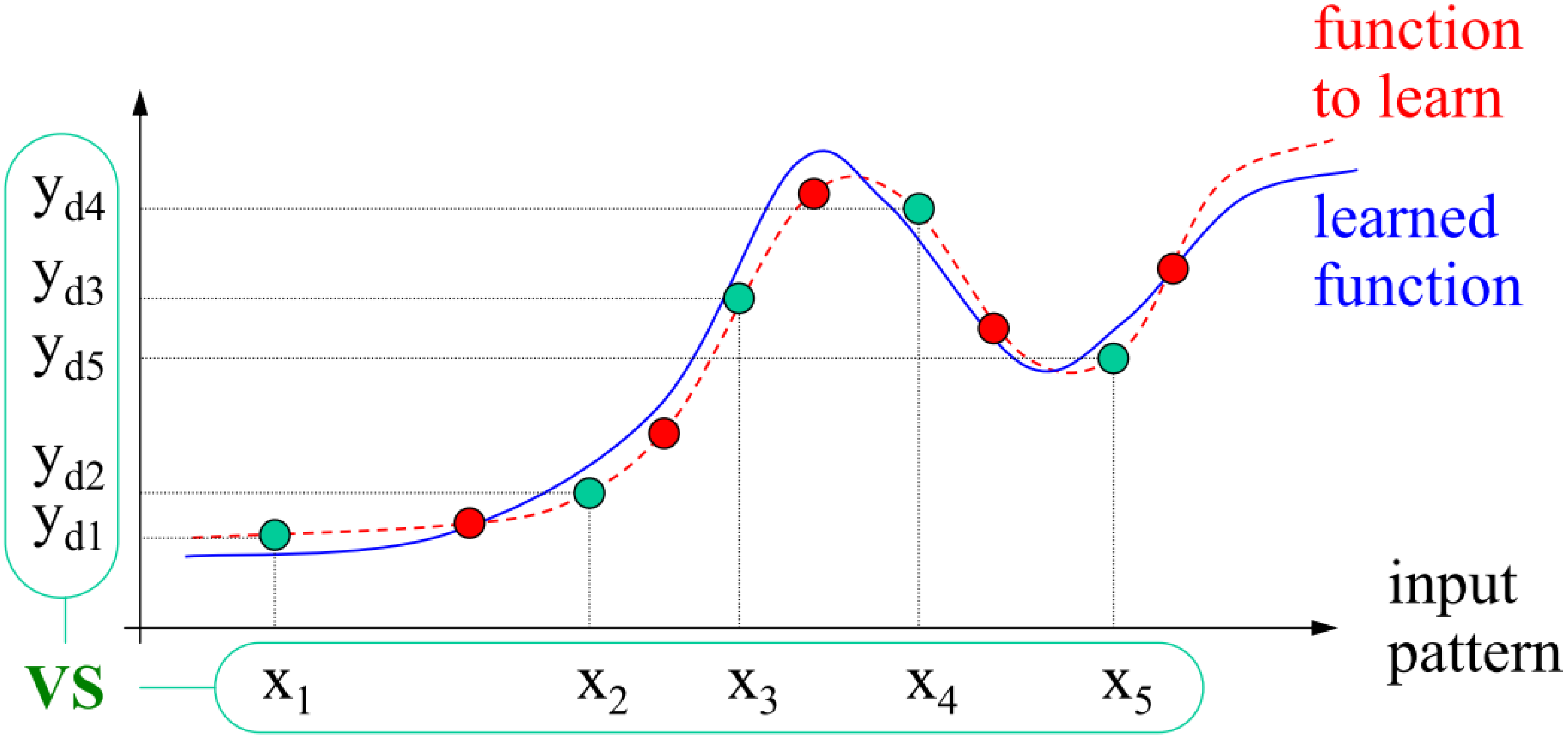
Generalization in Classification

- Suppose the task of our network is to learn a classification decision boundary
- Our aim is for the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately, as that is likely to reduce the generalization ability.



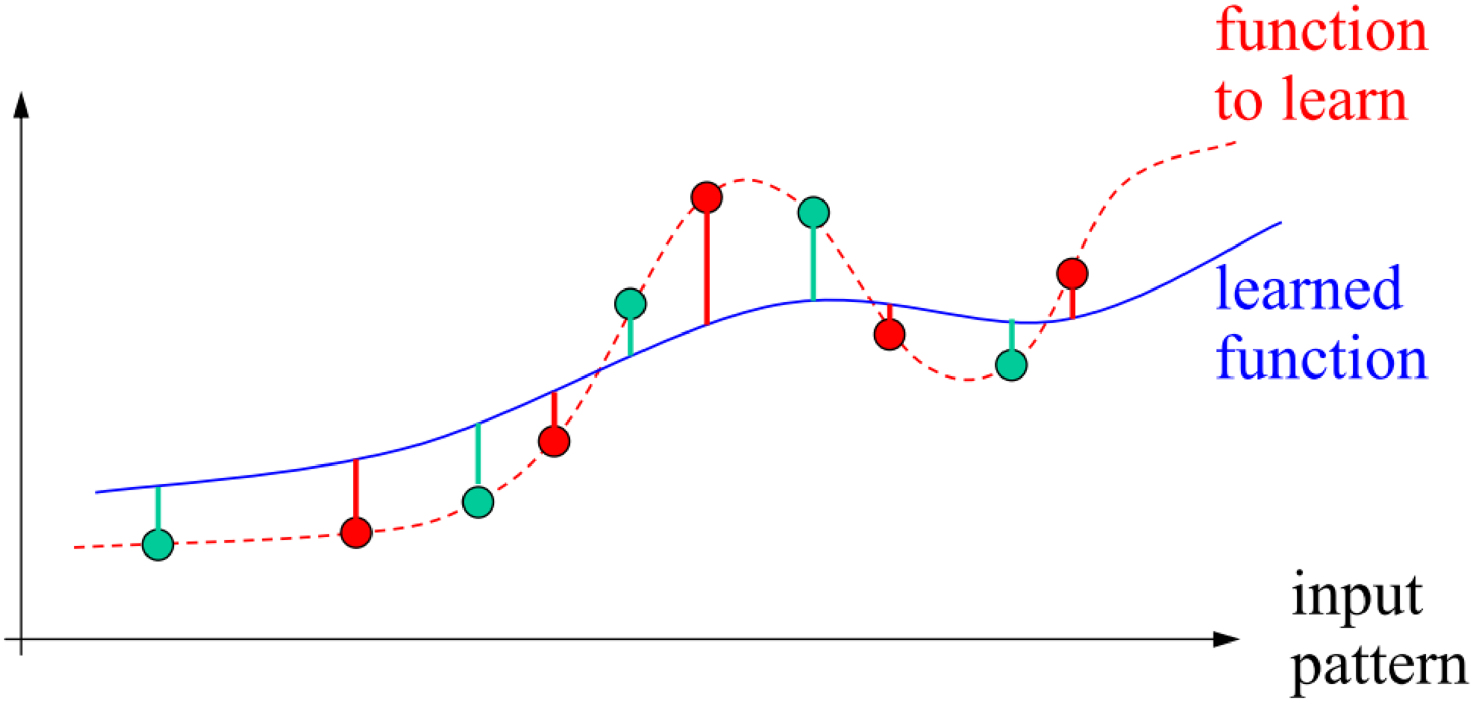
Generalization in Function Approximation

If the network is well dimensioned, both the errors E_{TS} and E_{VS} are small



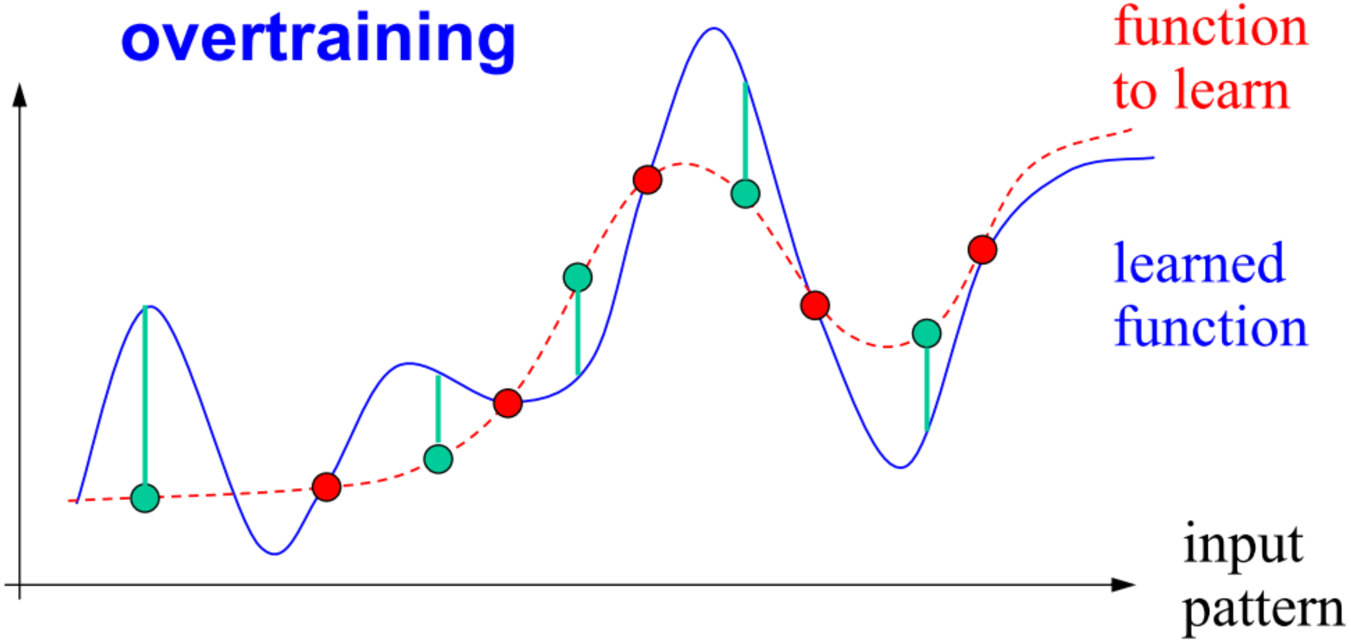
Generalization in Function Approximation

If the network does not have enough hidden neurons, it is not able to approximate the function and both errors are large:



Generalization in Function Approximation

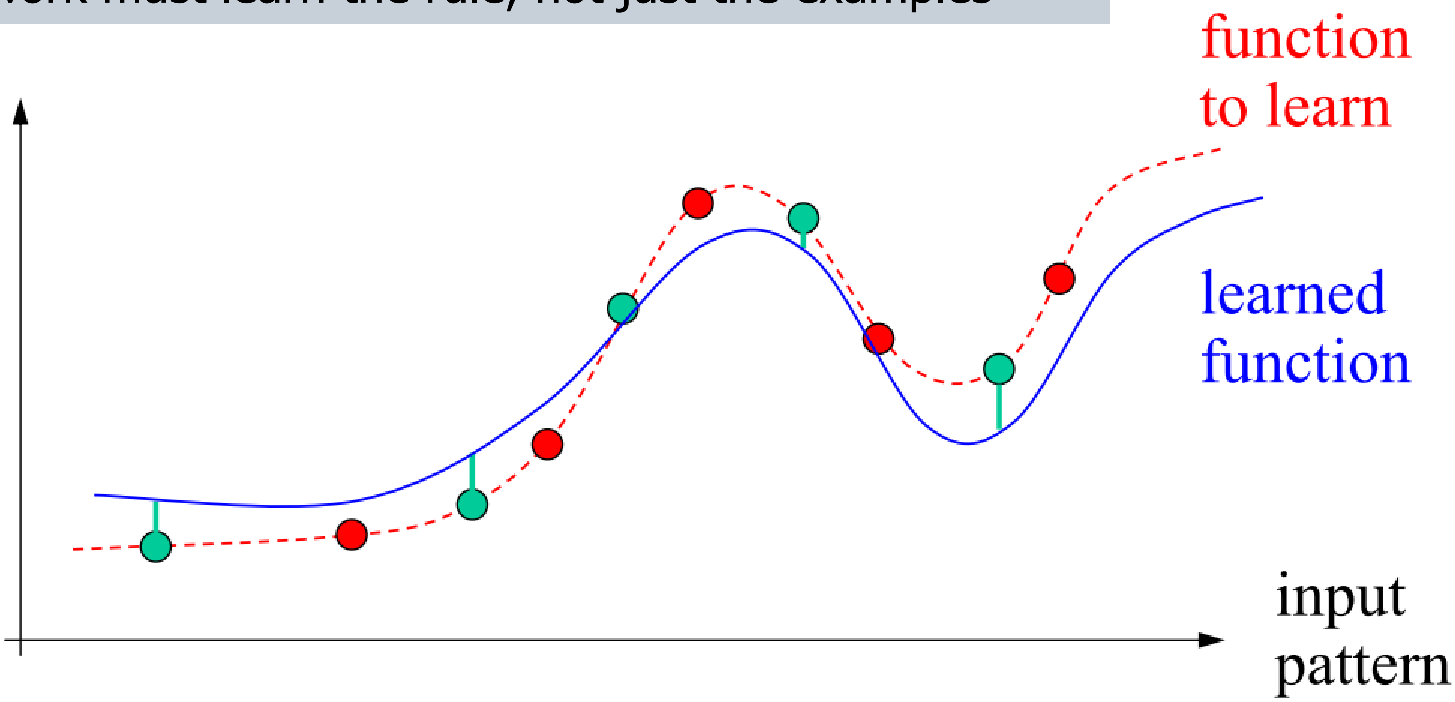
If the network has too many hidden neurons, it could respond correctly to the TS ($E_{TS} < \varepsilon$), but could not generalize well (E_{VS} too large):



Generalization in Function Approximation

To avoid the overtraining effect, we can train the network using the TS, but monitor E_{VS} and stop the training when ($E_{VS} < \varepsilon$):

(network must learn the rule, not just the examples



Tackling overfitting

1. Data-Level Methods (The Best Medicine)

2. Model Architecture & Complexity

3. Training Process Techniques

4. Advanced & Modern Methods

1. Data-Level Methods

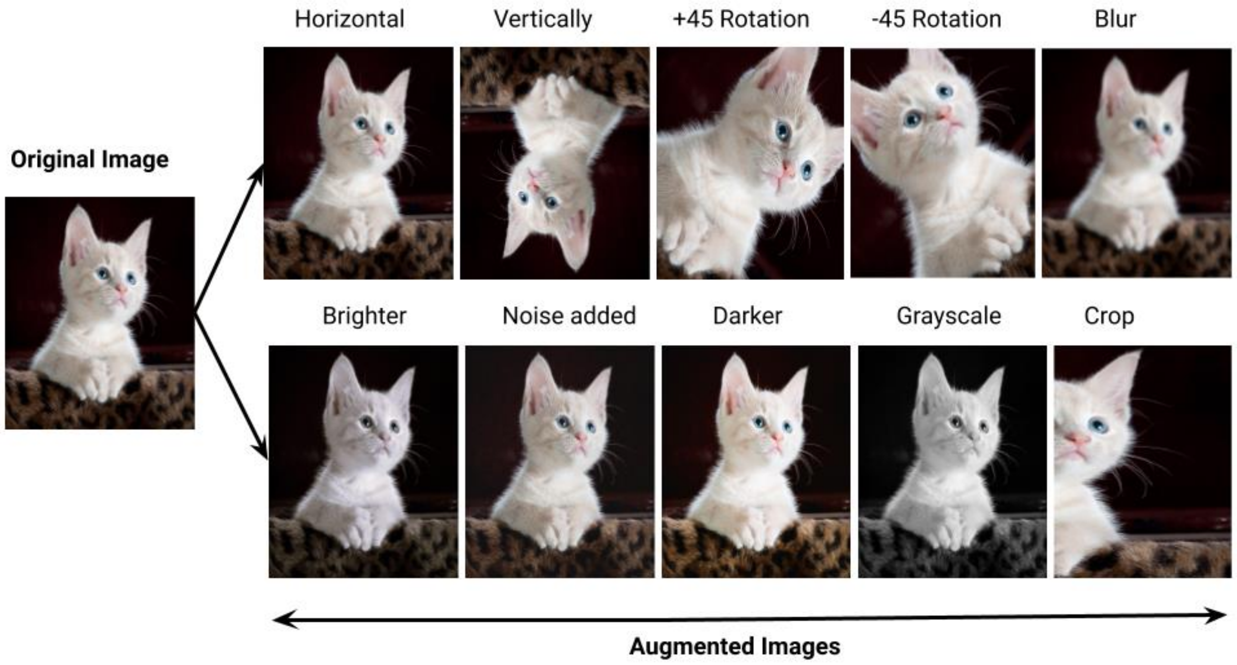
Get More Data: This is the most straightforward and powerful method. A larger dataset inherently provides more varied examples, making it harder for the network to memorize and forcing it to learn general features.

- **Data Augmentation:** Artificially increase the size and diversity of your training data by applying realistic transformations. This is highly domain-specific.

- **Data Cleaning:** Fix incorrect labels, remove outliers, and handle missing values. Clean data helps the model learn the true signal instead of fitting to noise.

Data augmentation

- Frequent operation in image data analysis
- Slightly transform images to generate unseen examples.
- The model will be forced to learn the general properties of objects
- Common transformations: rotation, width/height shift, shear, zoom, flip



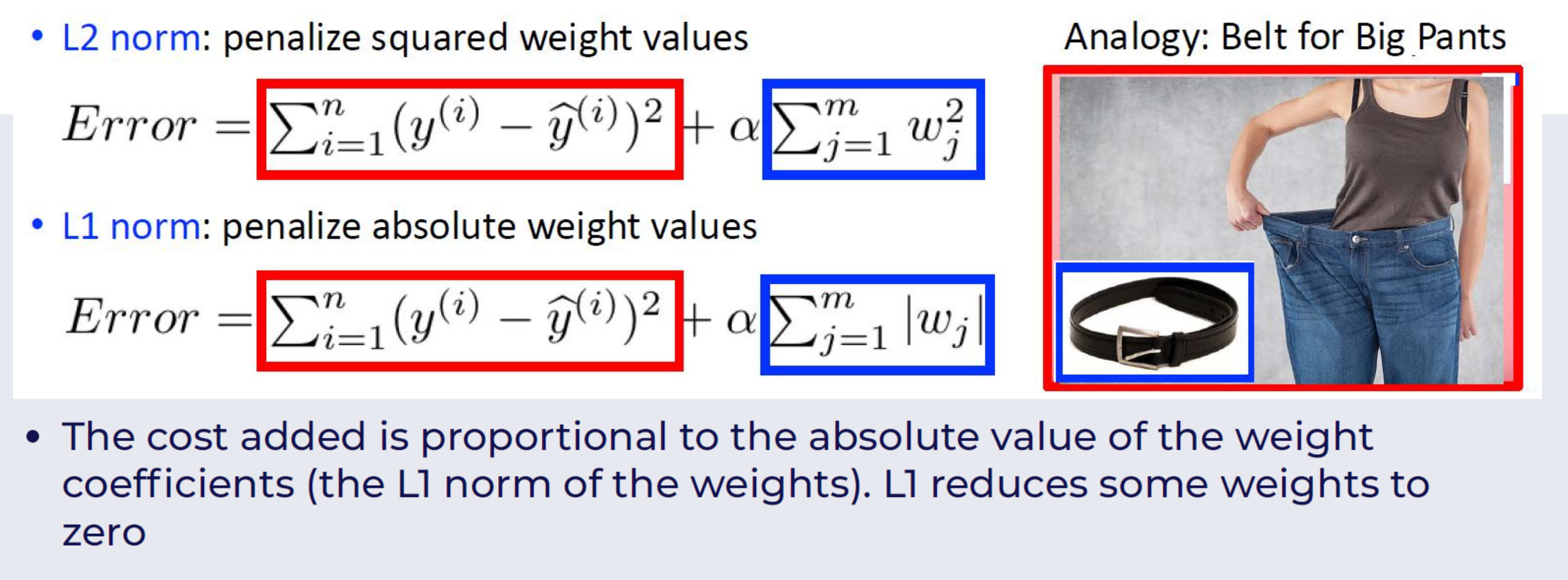
2. Model Architecture & Complexity

Simplify the Model: Use a model with fewer parameters (e.g., fewer layers, fewer neurons per layer). A model that is too powerful will easily overfit. Start with a simpler model and increase complexity only if necessary.

Apply Regularization: These techniques explicitly penalize model complexity by adding a term to the loss function (L1 , L2)

Use Dropout: During training, dropout randomly "drops out" (sets to zero) a proportion of the neurons in a layer for each training sample. This prevents neurons from co-adapting too much and forces the network to learn redundant, robust representations.

Regularization



- L2 norm: penalize squared weight values

$$Error = \boxed{\sum_{i=1}^{n} (y^{(i)} - \widehat{y}^{(i)})^2} + \alpha \boxed{\sum_{j=1}^{m} w_j^2}$$

- L1 norm: penalize absolute weight values

$$Error = \boxed{\sum_{i=1}^{n} (y^{(i)} - \widehat{y}^{(i)})^2} + \alpha \boxed{\sum_{j=1}^{m} |w_j|}$$

Analogy: Belt for Big Pants

- The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights). L1 reduces some weights to zero

For general-purpose prevention of overfitting, especially in deep learning and with correlated features, L2 regularization (which is also known as **Weight Decay**) is the most common and default choice

Why regularization helps to prevent overfitting

Without Regularization

The model tries to fit the training data perfectly, which often means:

- Assigning very large weights to certain features
- Learning to track every little fluctuation in the data (including random noise)
- Becoming extremely sensitive to small changes in input

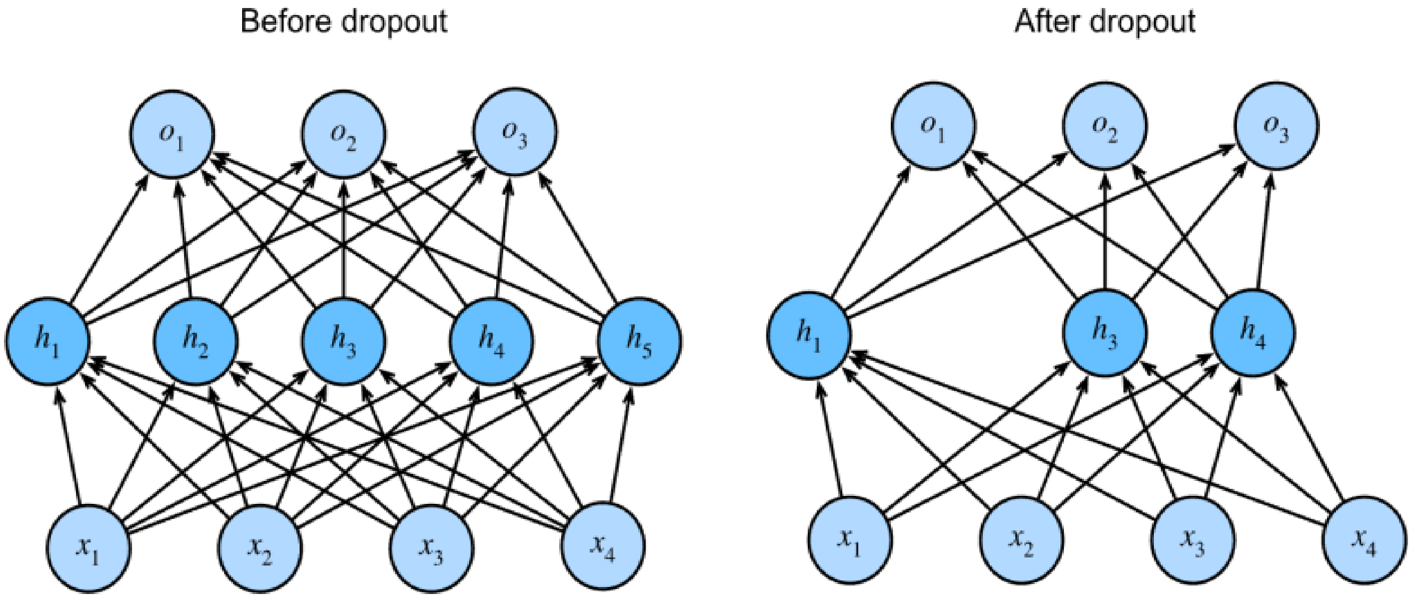
With Regularization

L2 Regularization (Ridge) - The "Gentle Guide"

- What it does: Penalizes large weights by adding their squared values to the loss function
- Effect: Prevents any single feature from having too much influence
- Analogy: Like a teacher saying "Don't focus too much on any one topic - understand everything moderately well"

Dropout

- Dropout regularizes the network by randomly dropping neurons from the neural network during training



self.dropout = nn.Dropout(0.4)

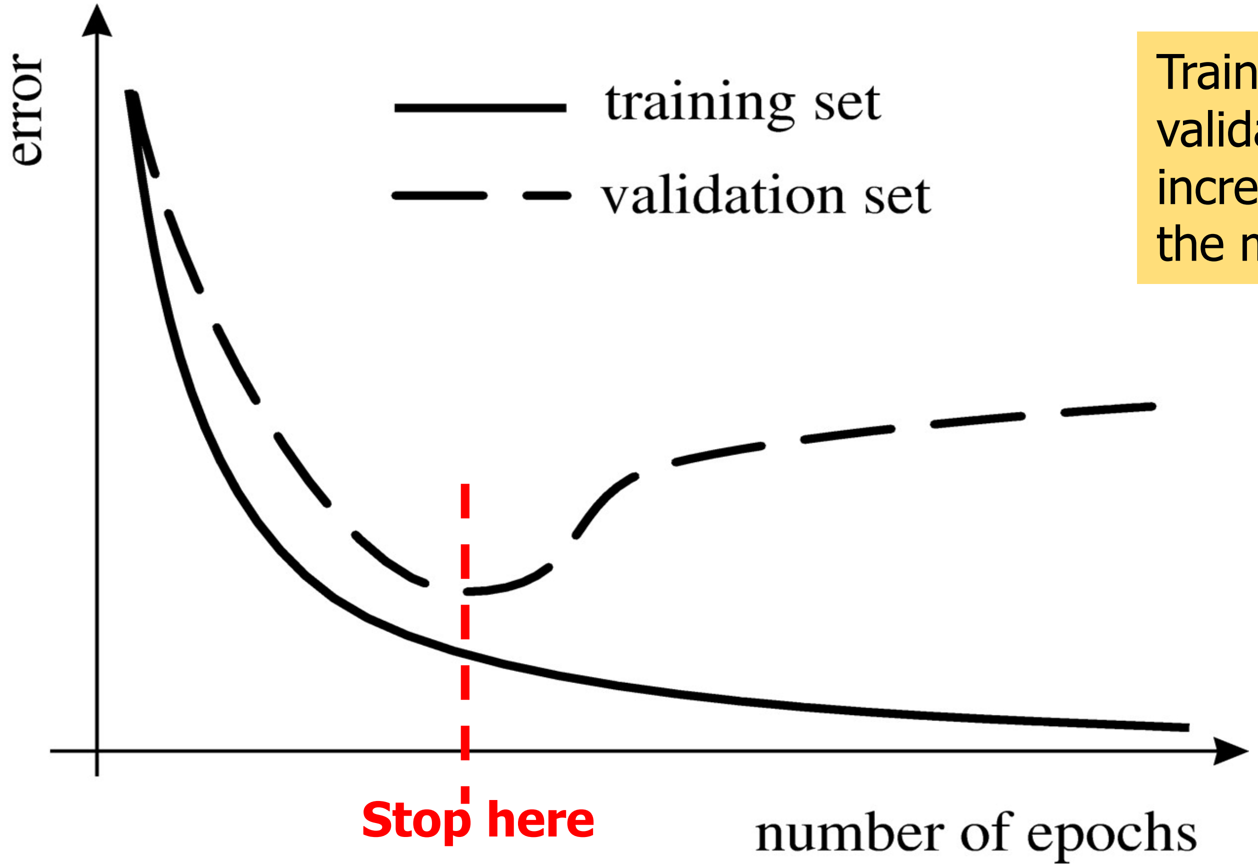
3. Training Process Techniques

Early Stopping: A simple and highly effective method.

Reduce Model Complexity via Training: Instead of changing the architecture, you can control complexity during training.

Earlier Stopping - Good Generalization

Use of a validation set allows periodic testing to see whether the model has overfitted



Training stops when the validation loss starts to increase, indicating that the model is overfitting.

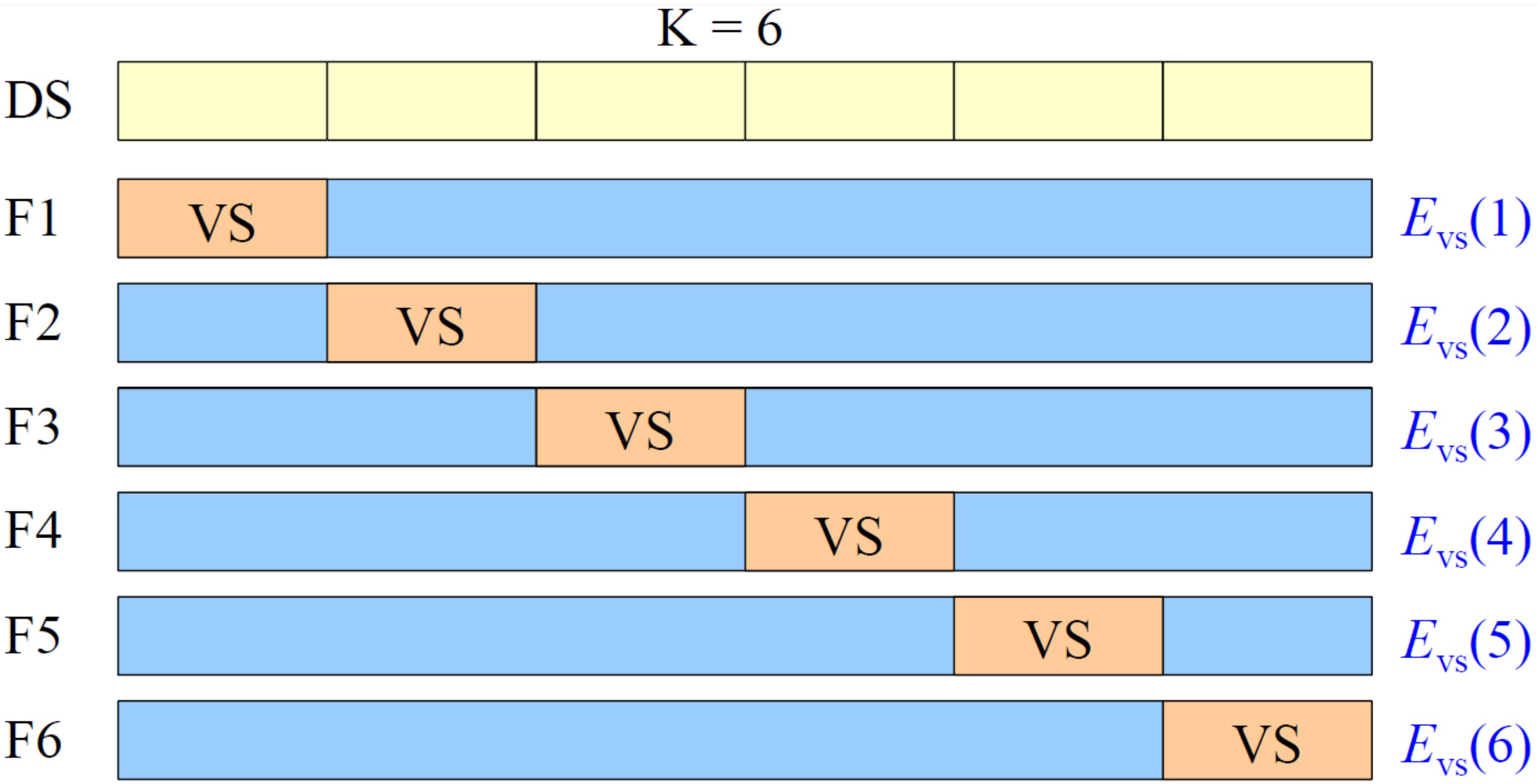
4. Advanced & Modern Methods

Batch Normalization: While its primary goal is to stabilize and accelerate training by reducing internal covariate shift, it also has a slight regularization effect. The noise introduced by calculating mean and variance on mini-batches adds a beneficial stochasticity, similar to dropout.

Transfer Learning: Instead of training a massive network from scratch on a small dataset, start with a model pre-trained on a very large dataset (e.g., ImageNet for vision, BERT for NLP). Then, fine-tune the last few layers on your specific task. This leverages generalized features learned from big data.

Ensemble Methods: Train multiple different models independently and combine their predictions (e.g., by averaging or voting). Because different models will overfit in different ways, their combination tends to be more robust and generalizable. This is computationally expensive but very powerful.

K-Fold Cross Validation

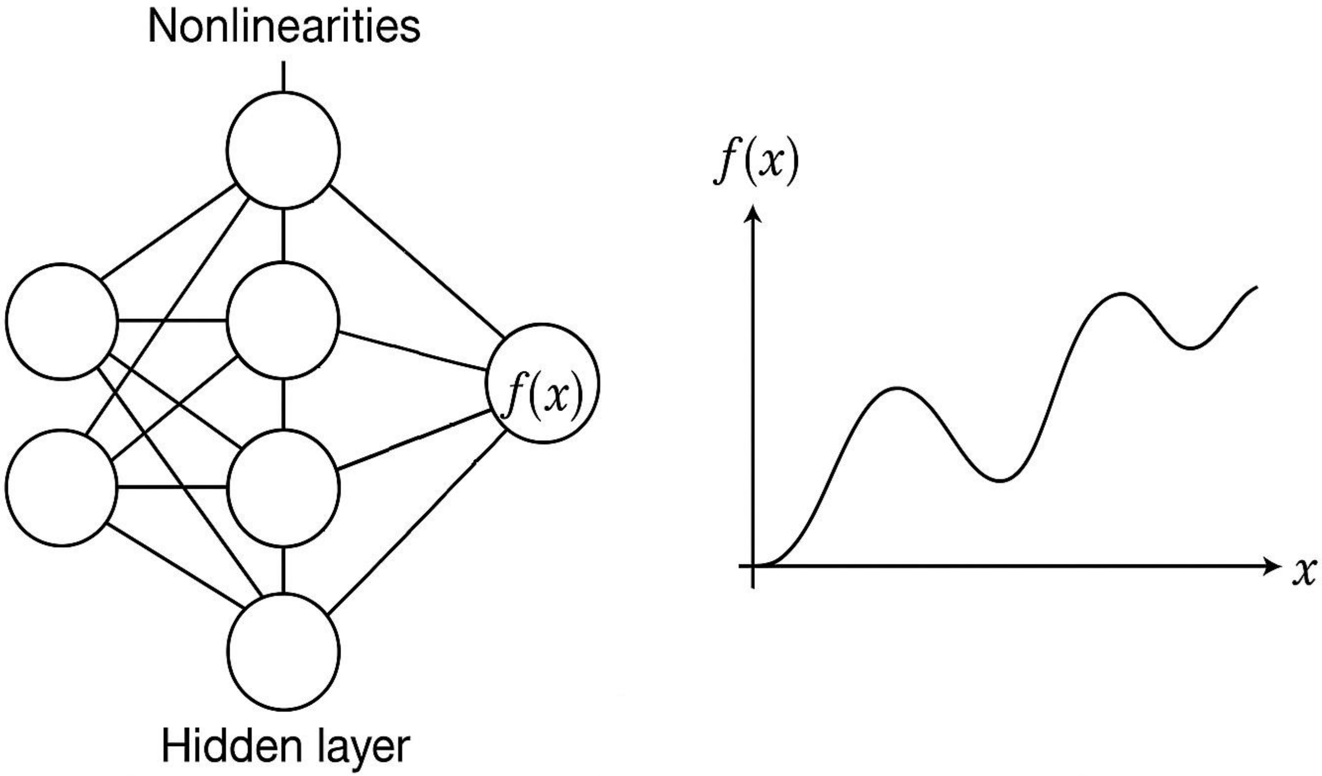


$$E_{vs} = \frac{1}{K}\sum_{i=1}^{K} E_{vs}(i)$$

$$A_{vs} = \frac{1}{K}\sum_{i=1}^{K} A_{vs}(i)$$

NN: Universal Approximator

- Any desired continuous function can be implemented by a three-layer network given sufficient number of hidden units, proper non-linearitiers and weights (Kolmogorov)

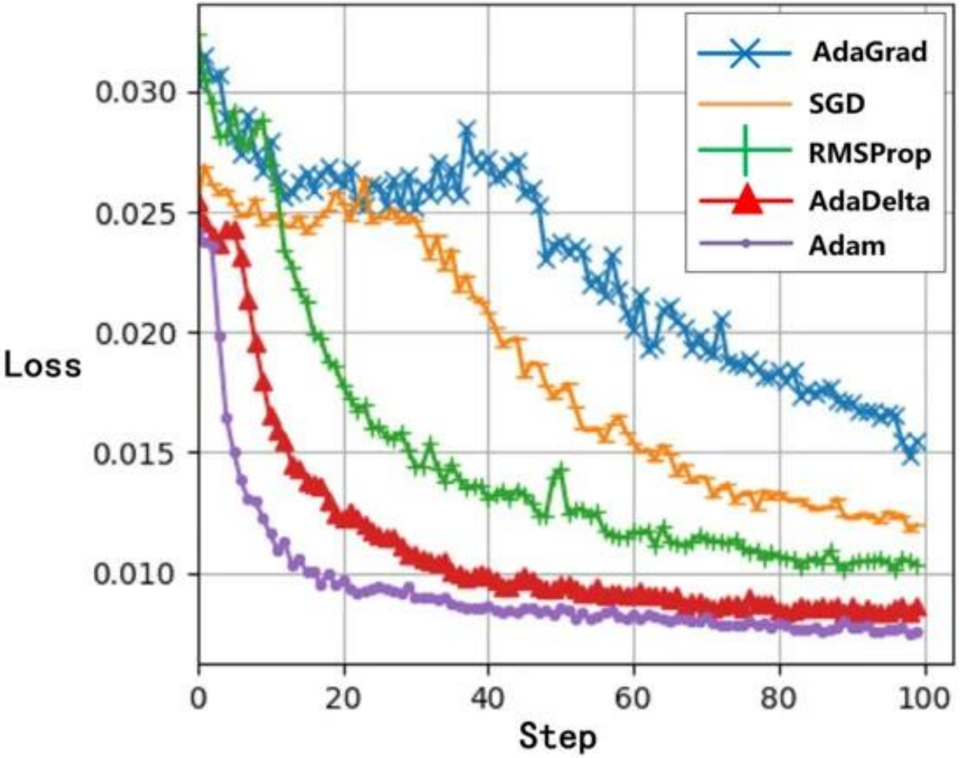


Optimizers

- **Momentum:** Accelerates SGD by using a moving average of past gradients.
- **SGD:** Updates weights using the gradient from a single random data batch.
- **AdaGrad:** Adapts each parameter's learning rate based on its past squared gradients.
- **AdaDelta:** Improves AdaGrad by using a fixed window of past gradients.
- **RMSprop:** Uses a moving average of squared gradients to adapt the learning rate.
- **Adam:** Combines Momentum and RMSprop, using both gradient means and variances.

Optimizers

- **Gradient Descent**: most fundamental technique to train Neural Networks
- Variants:
  - - Momentum
  - - SGD
  - - AdaGrad
  - - AdaDelta
  - - RMSprop
  - - Adam

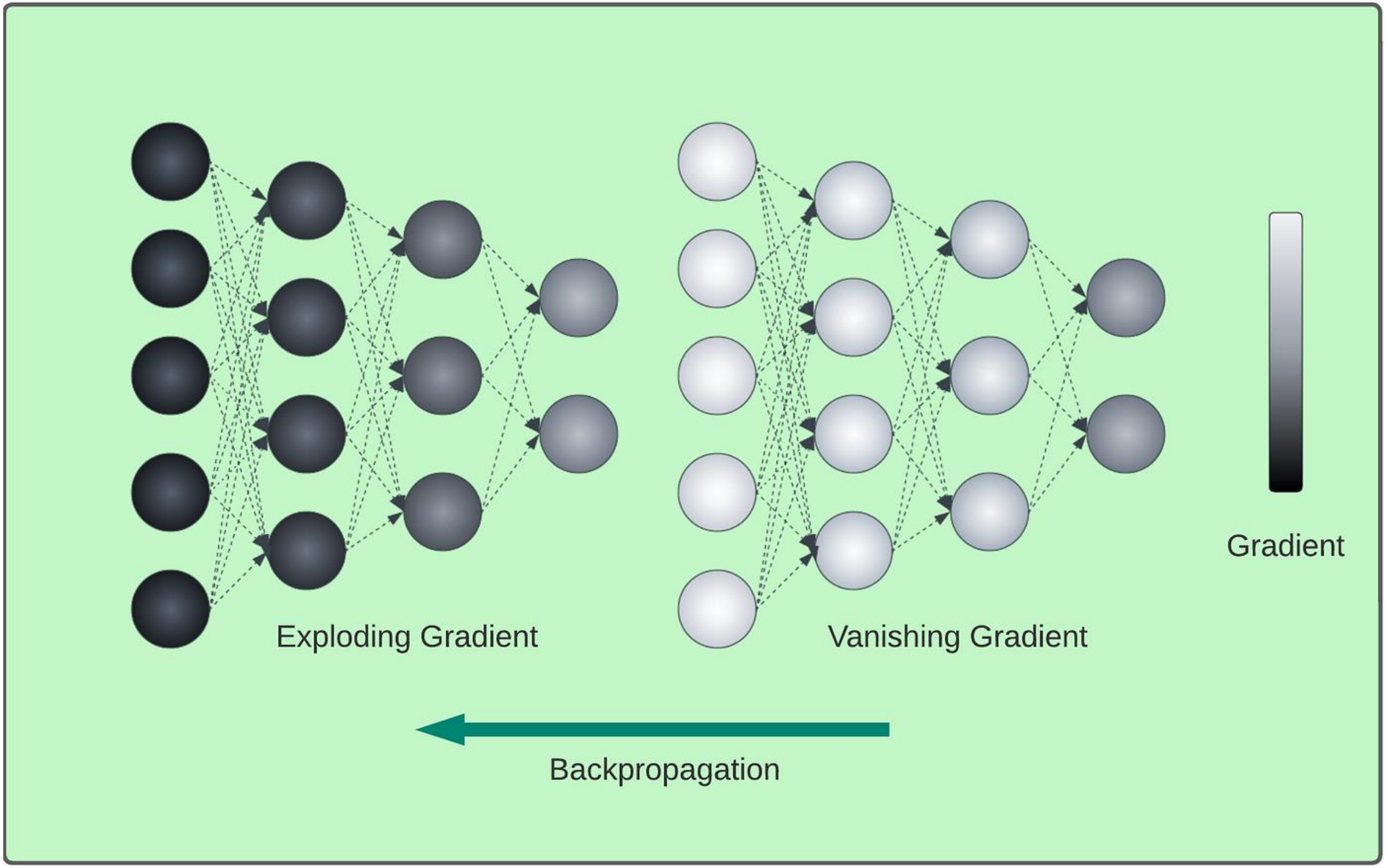


Exploding/Vanishing Gradients

- If many numbers |x| > 1 get multiplied, the result will become too big.
- NaN gradients --> no learning!

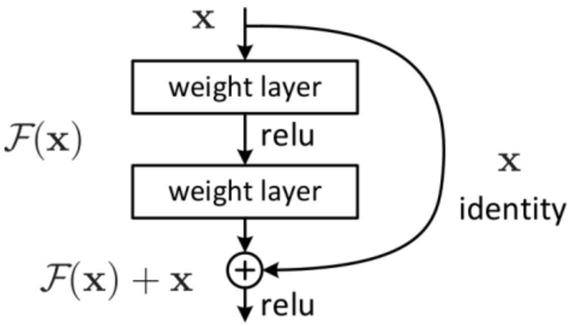
- If many numbers |x| < 1 get multiplied, the result will become too small.
- Zero gradients -> no learning!

Exploding/Vanishing Gradients



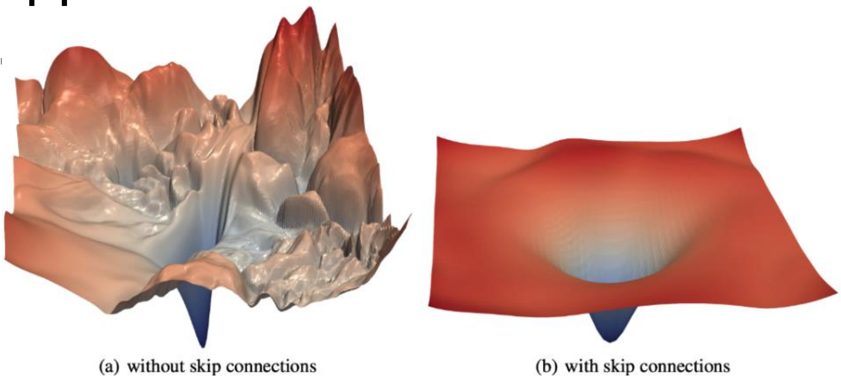
Residual Connections/Blocks

- Create direct “information highways” between layers.

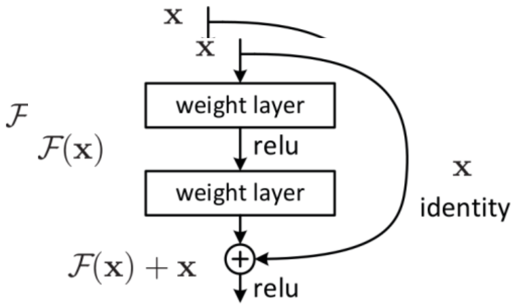


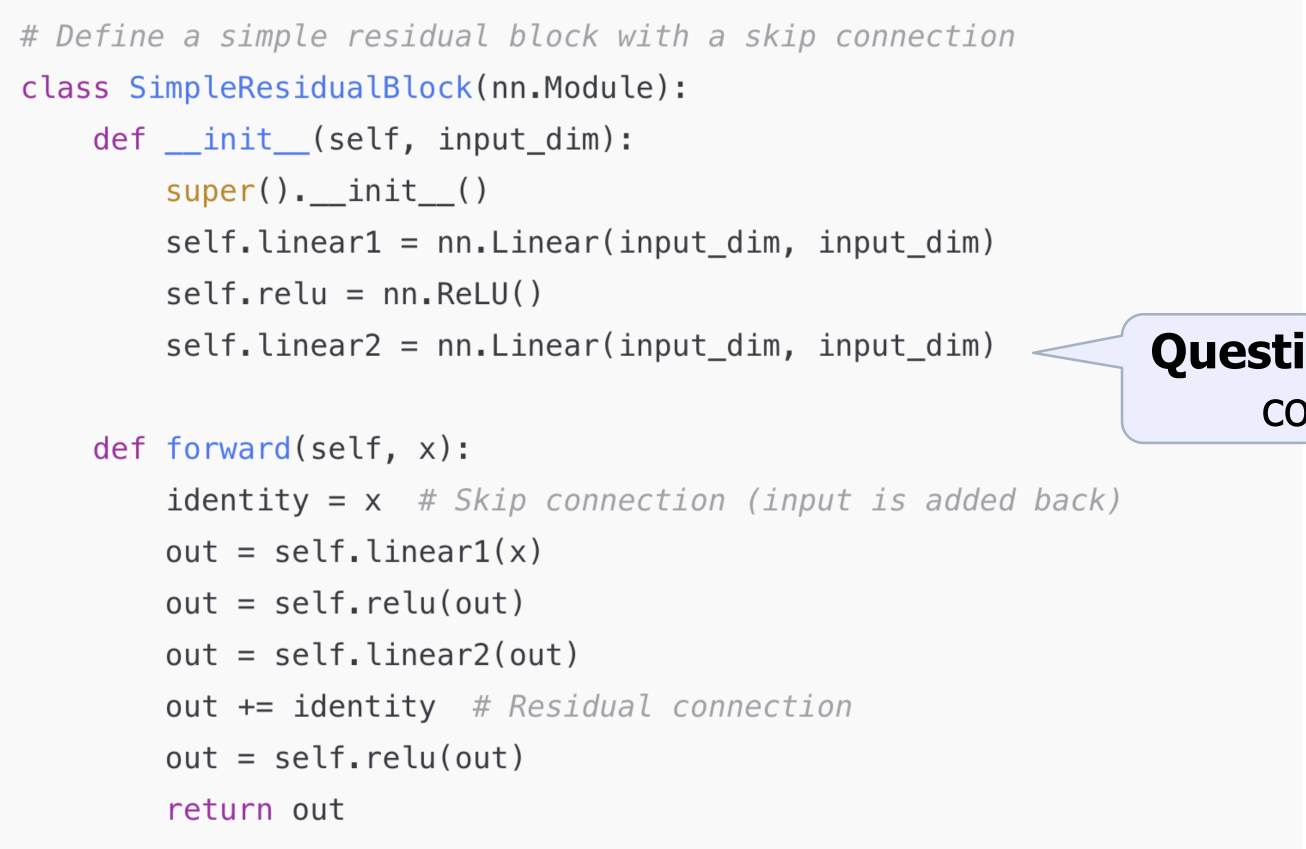
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
x
identity
weight layer
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

- Shown to diminish vanishing/exploding gradients
- Early in the training, there are fewer layers to propagate through.
  - The network would restore the skipped layers, as it learns richer features.
  - It is also shown to make the optimization objective smoother.



Residual Connections/Blocks


x
x
$\mathcal{F}$
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

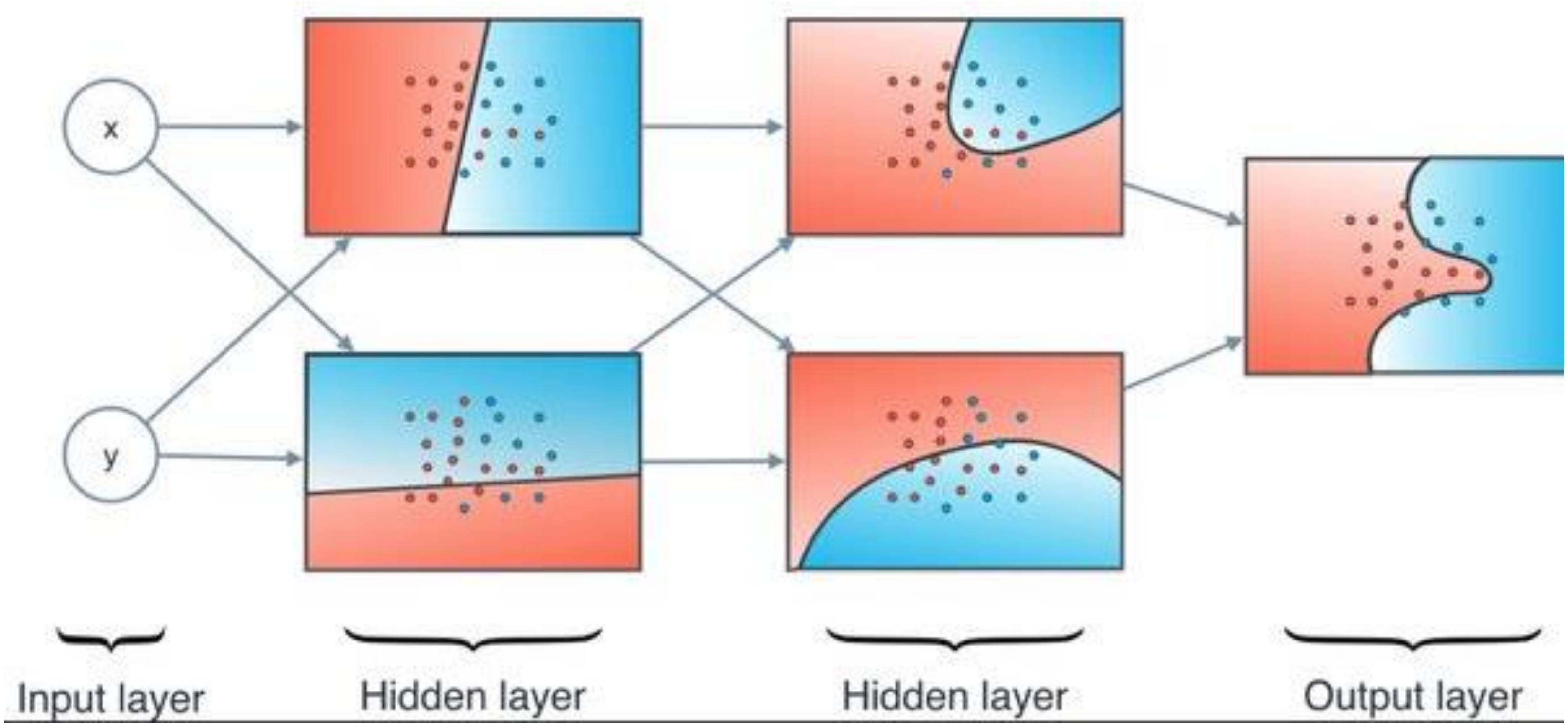


```python
# Define a simple residual block with a skip connection
class SimpleResidualBlock(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, input_dim)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(input_dim, input_dim)

    def forward(self, x):
        identity = x  # Skip connection (input is added back)
        out = self.linear1(x)
        out = self.relu(out)
        out = self.linear2(out)
        out += identity  # Residual connection
        out = self.relu(out)
        return out
```

Question: identify residual connections here

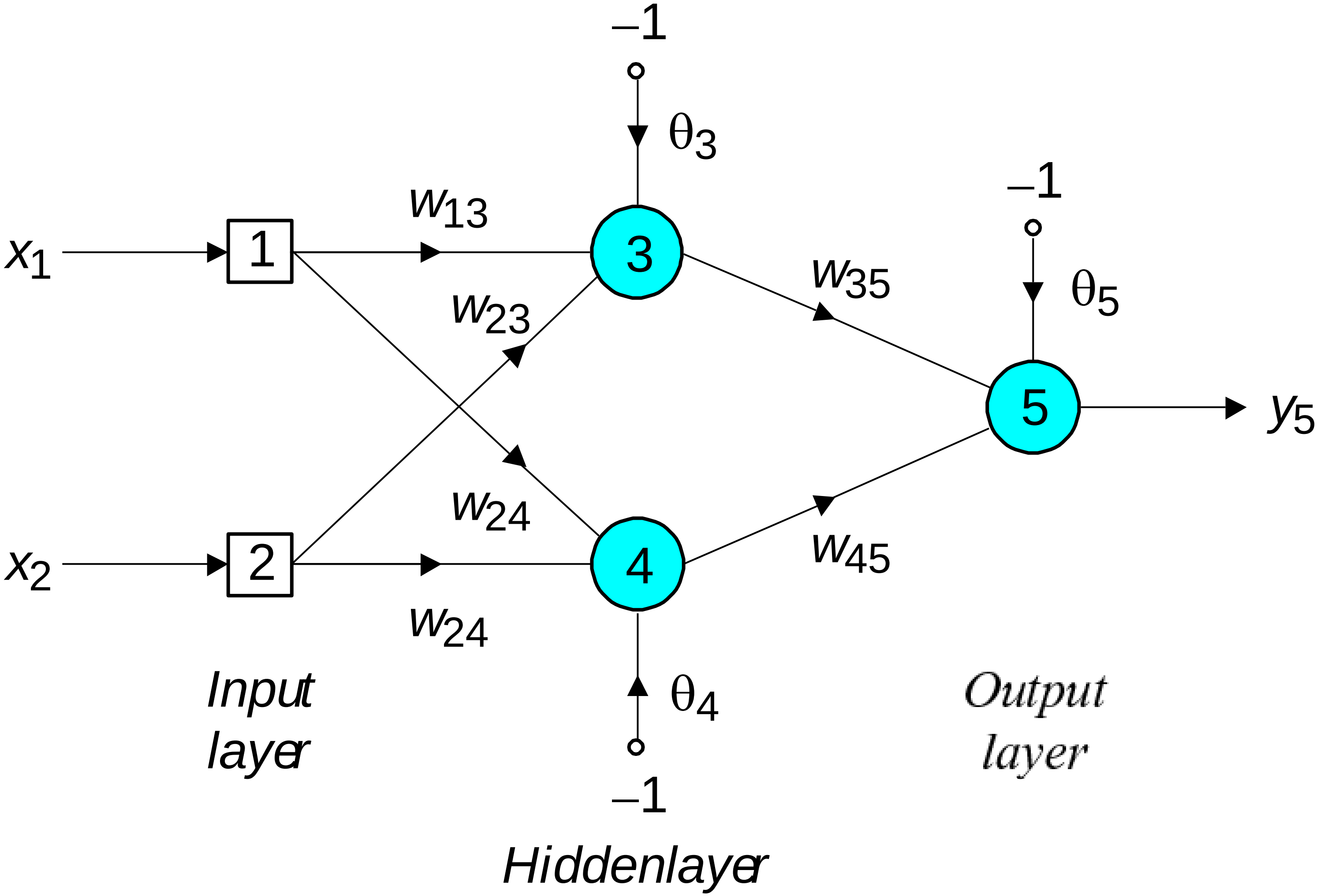
Intuition about Neural Net Representations



MLP- Summary

- Design the architecture, choose activation functions (e.g. sigmoids)
- Choose a way to initialize the weights (e.g. random initialization)
- Choose a loss function (e.g. log loss) to measure how well the model fits training data
- Choose an optimizer (typically an SGD variant) to update the weights

Solving the XOR operation



Solving the XOR operation

- The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to -1.

- The initial weights and threshold levels are set randomly as follows:
  $w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$, $\theta_3 = 0.8$, $\theta_4 = -0.1$ and $\theta_5 = 0.3$.

Solving the XOR operation

- We consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/\left[1 + e^{-(1\cdot 0.5 + 1\cdot 0.4 - 1\cdot 0.8)}\right] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/\left[1 + e^{-(1\cdot 0.9 + 1\cdot 1.0 + 1\cdot 0.1)}\right] = 0.8808$$

- Now the actual output of neuron 5 in the output layer is determined as:

$$y_5 = sigmoid(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/\left[1 + e^{-(-0.5250\cdot 1.2 + 0.8808\cdot 1.1 - 1\cdot 0.3)}\right] = 0.5097$$

- Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

Solving the XOR operation

- The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, *e*, from the output layer backward to the input layer.
- First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5\,(1 - y_5)\,e = 0.5097 \cdot (1 - 0.5097) \cdot (-0.5097) = -0.1274$$

- Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \cdot y_3 \cdot \delta_5 = 0.1 \cdot 0.5250 \cdot (-0.1274) = -0.0067$$
$$\Delta w_{45} = \alpha \cdot y_4 \cdot \delta_5 = 0.1 \cdot 0.8808 \cdot (-0.1274) = -0.0112$$
$$\Delta \theta_5 = \alpha \cdot (-1) \cdot \delta_5 = 0.1 \cdot (-1) \cdot (-0.1274) = -0.0127$$

Solving the XOR operation

- **Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:**

$$\delta_3 = y_3(1 - y_3) \cdot \delta_5 \cdot w_{35} = 0.5250 \cdot (1 - 0.5250) \cdot (-0.1274) \cdot (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \cdot \delta_5 \cdot w_{45} = 0.8808 \, (1 - 0.8808) \cdot (-0.1274) \cdot 1.1 = -0.0147$$

- **We then determine the weight corrections:**

$$\Delta w_{13} = \alpha \cdot x_1 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$
$$\Delta w_{23} = \alpha \cdot x_2 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$
$$\Delta \theta_3 = \alpha \cdot (-1) \cdot \delta_3 = 0.1 \cdot (-1) \cdot 0.0381 = -0.0038$$
$$\Delta w_{14} = \alpha \cdot x_1 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$
$$\Delta w_{24} = \alpha \cdot x_2 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$
$$\Delta \theta_4 = \alpha \cdot (-1) \cdot \delta_4 = 0.1 \cdot (-1) \cdot (-0.0147) = 0.0015$$

Solving the XOR operation

- At last, we update all weights and threshold:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

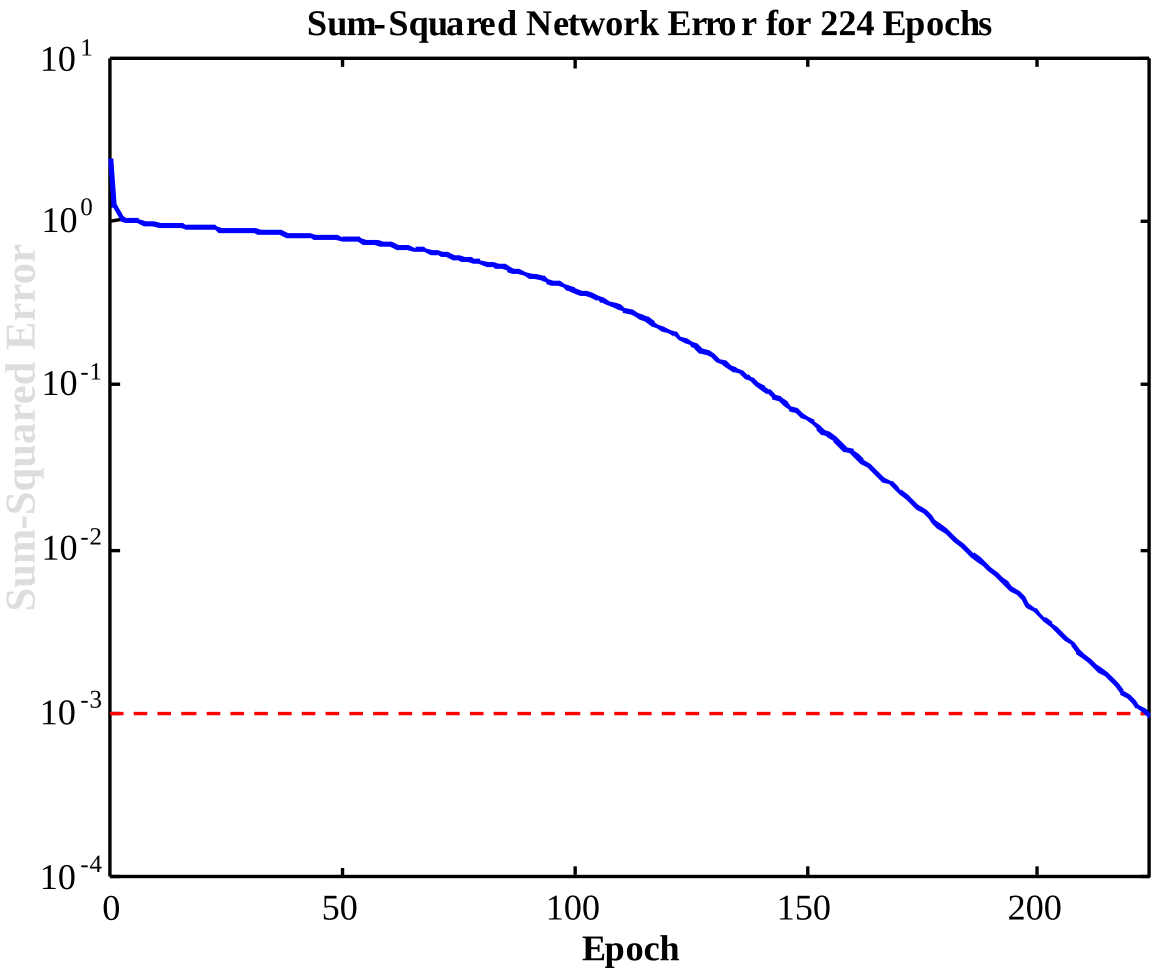
$$\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$$

- The training process is repeated until the sum of squared errors is less than 0.001.

Learning curve for operation XOR

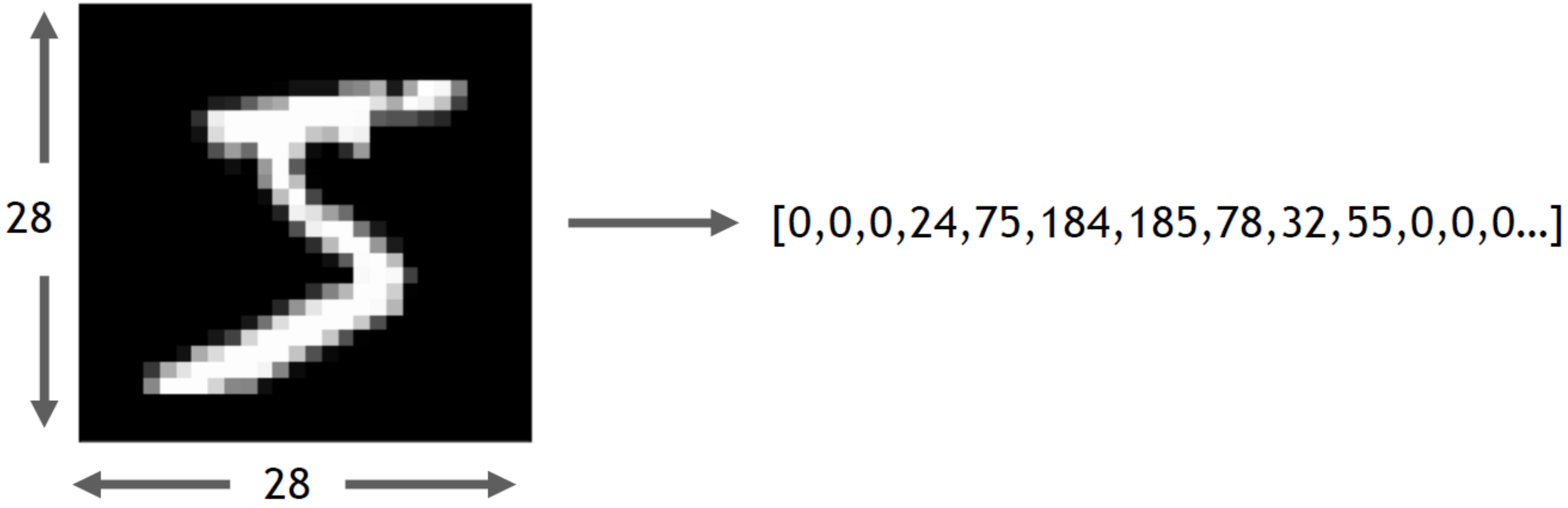


Final results of three-layer network learning

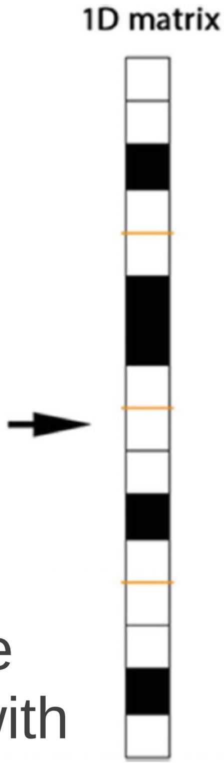
Inputs		Desired output	Actual output	Error	Sum of squared errors
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	–0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	–0.0175	

MLP using NMIST Dataset

- Each MNIST image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers:



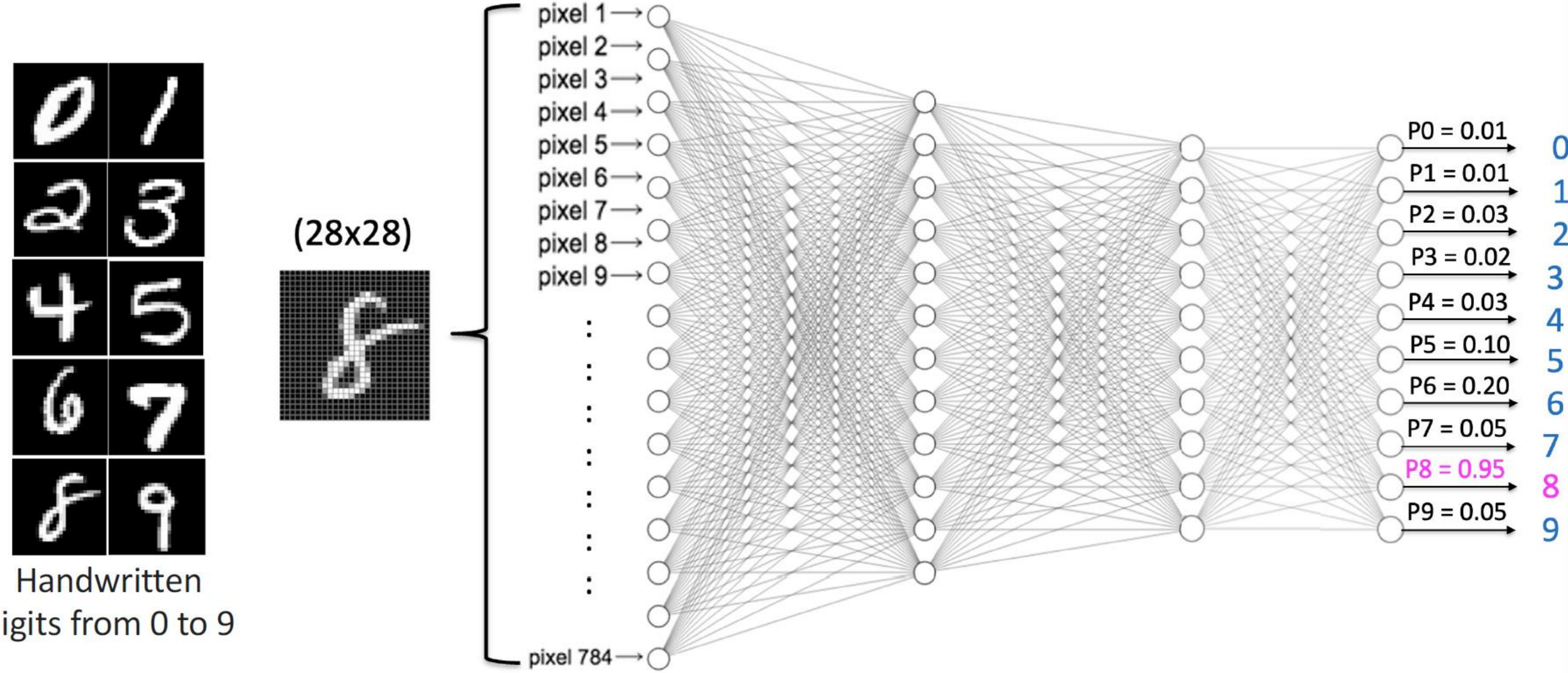
[0,0,0,24,75,184,185,78,32,55,0,0,0...]



- Thus, after flattening the image into vectors of 28*28=784, we obtain as mnist.train.images a tensor (an n-dimensional array) with a shape of [55000, 784]

MLP using NMIST Dataset

- In this example, we will use the PyTorch deep learning framework to create a MLP model that can recognize handwritten digits. We will train this model using a dataset called MNIST, which has 70,000 images of handwritten digits from 0 to 9.


Handwritten
igits from 0 to 9
(28x28)
pixel 1
pixel 2
pixel 3
pixel 4
pixel 5
pixel 6
pixel 7
pixel 8
pixel 9
pixel 784
P0 = 0.01
P1 = 0.01
P2 = 0.03
P3 = 0.02
P4 = 0.03
P5 = 0.10
P6 = 0.20
P7 = 0.05
P8 = 0.95
P9 = 0.05
0
1
2
3
4
5
6
7
8
9