

COURS RDFIA deep Image

<https://cord.isir.upmc.fr/teaching-rdfia/>

Matthieu Cord
Sorbonne University

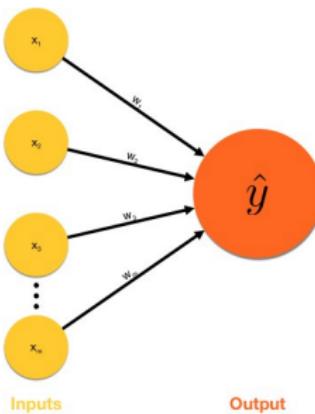
Course Outline

1. Computer Vision and ML basics: Visual (local)
feature detection and description, Bag of Word Image
representation, Linear classification (SVM)
2. Introduction to Neural Networks (NNs)
3. Machine Learning basics (2): Risk, Classification,
Datasets, benchmarks and evaluation
4. Neural Nets for Image Classification
5. Vision Transformers
6. Transfer learning and domain adaptation
7. Segmentation and Detection
8. Generative models with GANs
9. Generative models with diffusion
10. Large VL models: CLIP, StableDiffusion,
Flamingo
11. Control (to be checked) -- Explainable AI,
Fairness
- 12/13 Bayesian deep learning
- 14 Robustness

Evaluations: Control (30%) + Practicals (3 reports; 70%)
can be modified by 10% between the 2 evaluations

The Formal Neuron: 1943 [MP43]

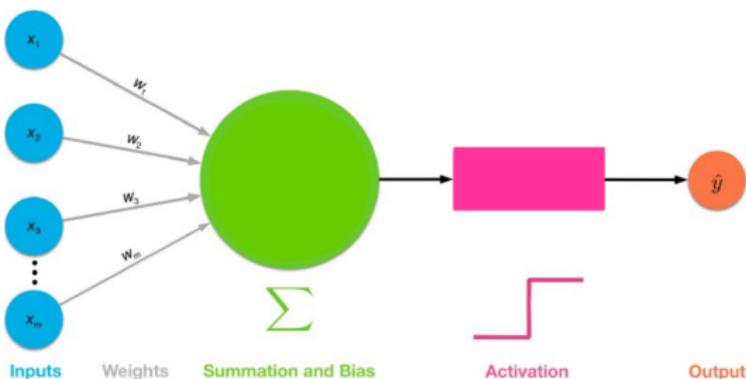
- ▶ Basis of Neural Networks
- ▶ Input: vector $x \in \mathbb{R}^m$, i.e. $x = \{x_i\}_{i \in \{1, 2, \dots, m\}}$
- ▶ Neuron output $\hat{y} \in \mathbb{R}$: scalar



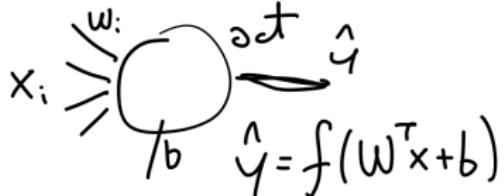
The Formal Neuron: 1943 [MP43]

► Mapping from x to \hat{y} :

1. Linear (affine) mapping: $s = w^T x + b$
2. Non-linear activation function: $f: \hat{y} = f(s)$

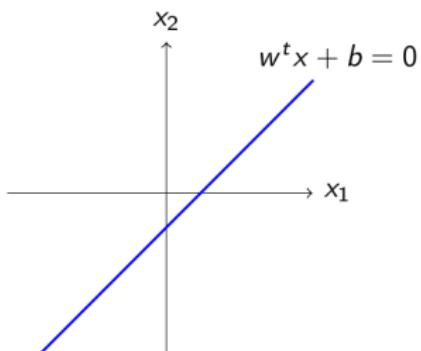


The Formal Neuron: Linear Mapping

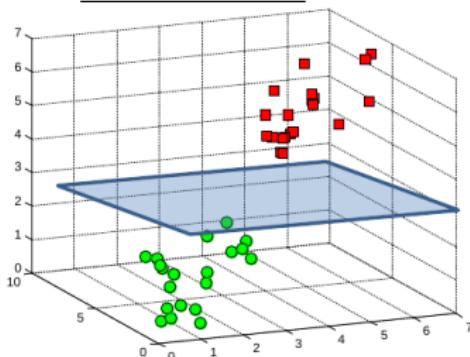


- ▶ Linear (affine) mapping: $s = w^T x + b = \sum_{i=1}^m w_i x_i + b$
 - ▶ w : normal vector to an hyperplane in $\mathbb{R}^m \Rightarrow$ **linear boundary**
 - ▶ b bias, shift the hyperplane position

2D hyperplane: line



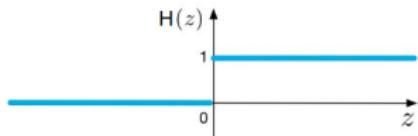
3D hyperplane: plane



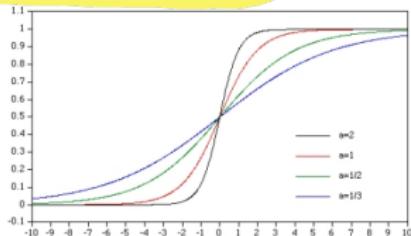
The Formal Neuron: Activation Function

- ▶ $\hat{y} = f(w^\top x + b)$,
- ▶ f : activation function

▶ Bio-inspired choice: Step (Heaviside) function: $H(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$

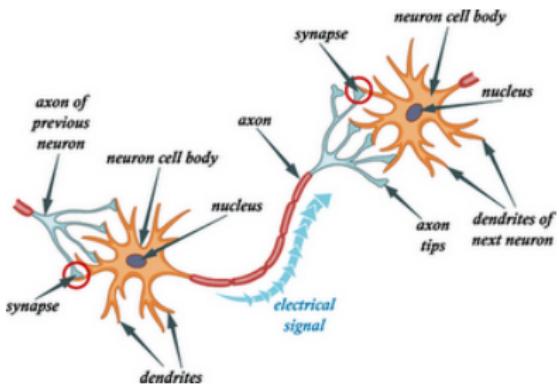
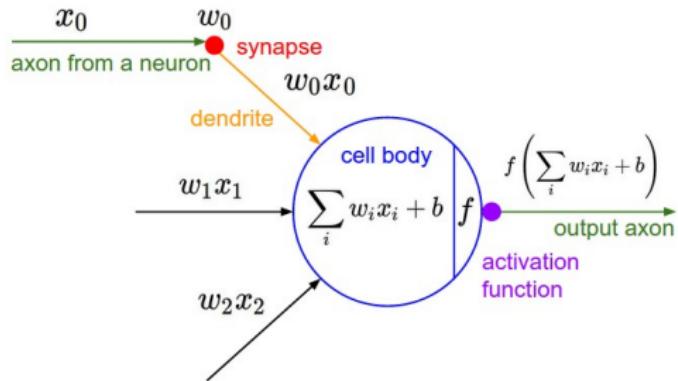


- ▶ Popular f choices: sigmoid, tanh, ReLU, GELU, ...
- ▶ Sigmoid: $\sigma(z) = (1 + e^{-az})^{-1}$



- ▶ $a \uparrow$: more similar to step function (step: $a \rightarrow \infty$)
- ▶ Sigmoid: linear and saturating regimes

Step function: Connection to Biological Neurons



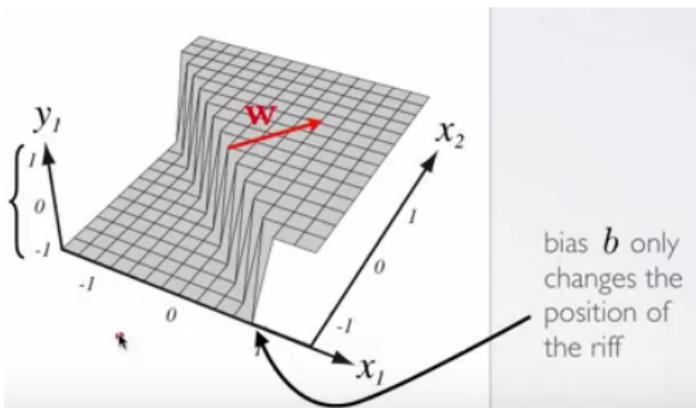
- ▶ Formal neuron, step activation H : $\hat{y} = H(w^T x + b)$
 - ▶ $\hat{y} = 1$ (activated) $\Leftrightarrow w^T x \geq -b$
 - ▶ $\hat{y} = 0$ (unactivated) $\Leftrightarrow w^T x < -b$
- ▶ **Biological Neurons:** output activated
 \Leftrightarrow input weighted by synaptic weight \geq threshold

The Formal neuron: Application to Binary Classification

- ▶ Binary Classification: label input x as belonging to class 1 or 0
- ▶ Neuron output with sigmoid:

$$\hat{y} = \frac{1}{1 + e^{-a(w^T x + b)}}$$

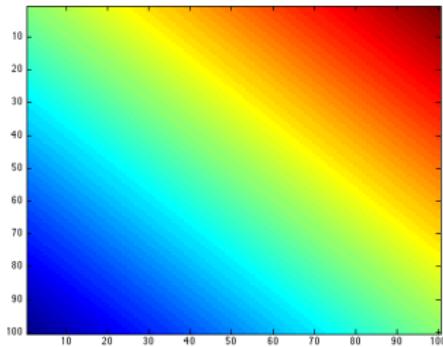
- ▶ Sigmoid: probabilistic interpretation $\Rightarrow \hat{y} \sim P(1|x)$
 - ▶ Input x classified as 1 if $P(1|x) > 0.5 \Leftrightarrow w^T x + b > 0$
 - ▶ Input x classified as 0 if $P(1|x) < 0.5 \Leftrightarrow w^T x + b < 0$
 $\Rightarrow \text{sign}(w^T x + b)$: linear boundary decision in input space !



The Formal neuron: Toy Example for Binary Classification

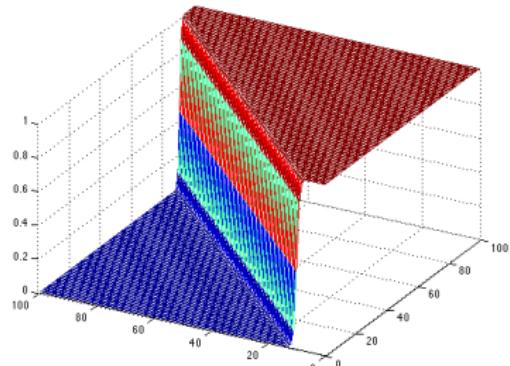
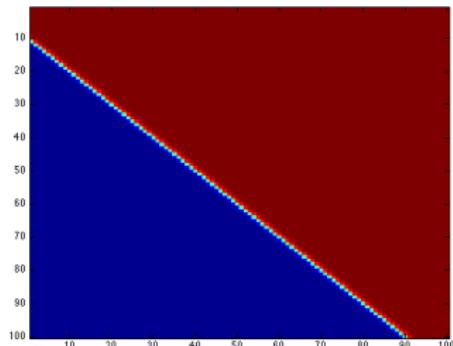
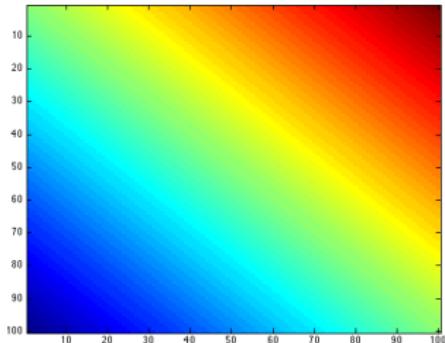
- ▶ 2d example: $m = 2$, $x = \{x_1, x_2\} \in [-5; 5] \times [-5; 5]$
- ▶ Linear mapping: $w = [1; 1]$ and $b = -2$
- ▶ Result of linear mapping : $s = w^T x + b$

$$\hat{y} = \frac{1}{1 + e^{-as}}$$



The Formal neuron: Toy Example for Binary Classification

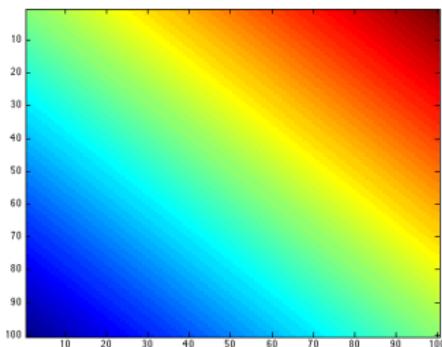
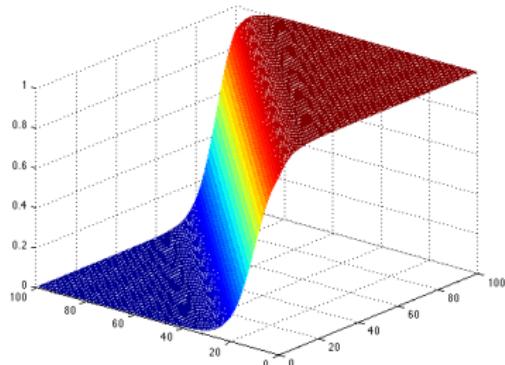
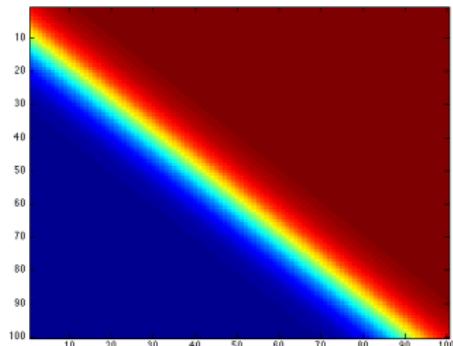
- ▶ 2d example: $m = 2$, $x = \{x_1, x_2\} \in [-5; 5] \times [-5; 5]$
- ▶ Linear mapping: $w = [1; 1]$ and $b = -2$
- ▶ Result of linear mapping : $s = w^T x + b$
- ▶ Sigmoid activation function: $\hat{y} = \left(1 + e^{-a(w^T x + b)}\right)^{-1}$,
 $a = 10$



The Formal neuron: Toy Example for Binary Classification

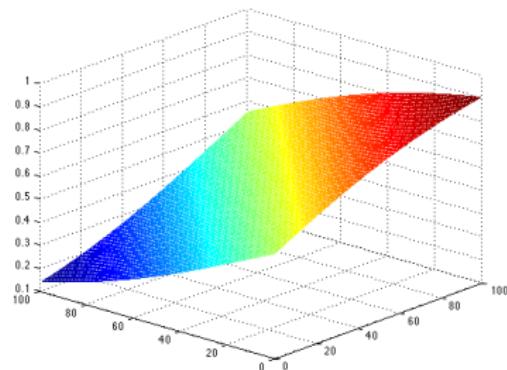
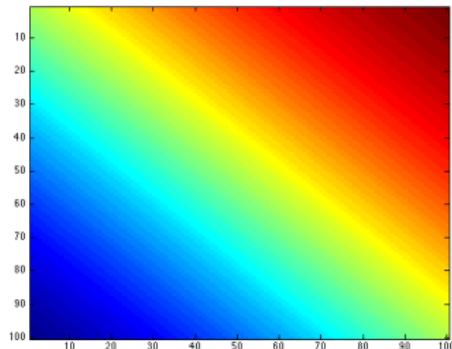
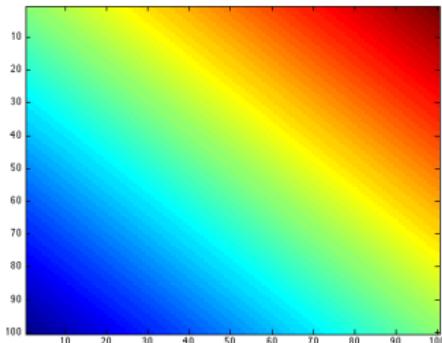
- ▶ 2d example: $m = 2$, $\mathbf{x} = \{x_1, x_2\} \in [-5; 5] \times [-5; 5]$
- ▶ Linear mapping: $\mathbf{w} = [1; 1]$ and $b = -2$
- ▶ Result of linear mapping : $s = \mathbf{w}^T \mathbf{x} + b$
- ▶ Sigmoid activation function: $\hat{y} = \left(1 + e^{-a(\mathbf{w}^T \mathbf{x} + b)}\right)^{-1}$,
 $a = 1$

smoother with smaller value of a



The Formal neuron: Toy Example for Binary Classification

- ▶ 2d example: $m = 2$, $x = \{x_1, x_2\} \in [-5; 5] \times [-5; 5]$
- ▶ Linear mapping: $w = [1; 1]$ and $b = -2$
- ▶ Result of linear mapping : $s = w^T x + b$
- ▶ Sigmoid activation function: $\hat{y} = \left(1 + e^{-a(w^T x + b)}\right)^{-1}$,
 $a = 0.1$



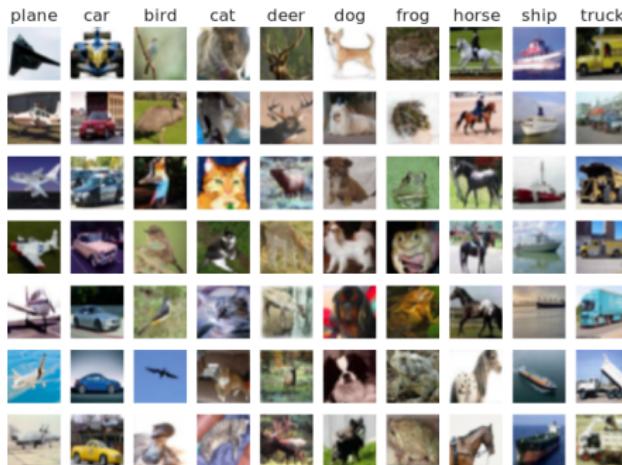
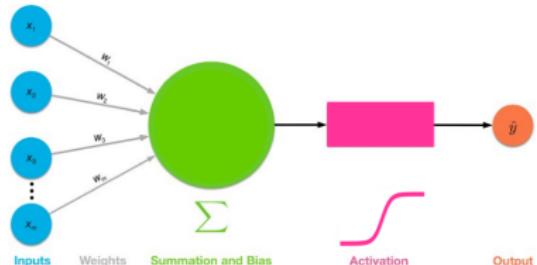
From Formal Neuron to Neural Networks

► Formal Neuron:

1. A single scalar output
2. Linear decision boundary for binary classification

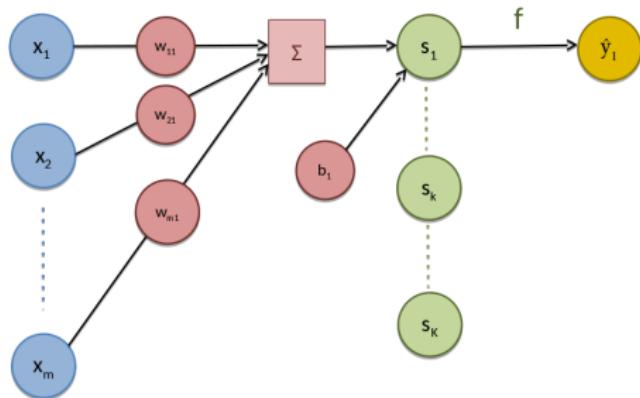
► Single scalar output: limited for several tasks

- Ex: multi-class classification, e.g. MNIST or CIFAR



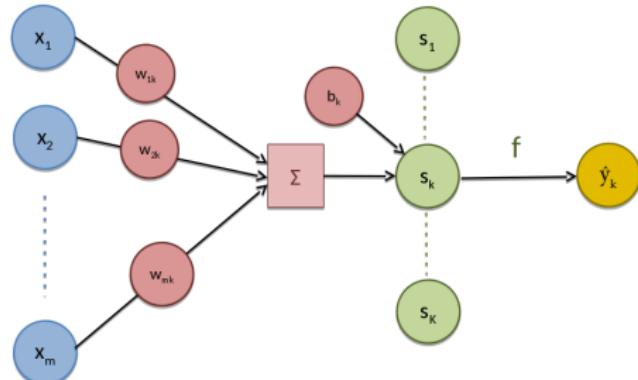
Perceptron and Multi-Class Classification

- ▶ Formal Neuron: limited to binary classification
- ▶ Multi-Class Classification: use several output neurons instead of a single one!
⇒ Perceptron
- ▶ Input x in \mathbb{R}^m
- ▶ Output neuron \hat{y}_1 is a formal neuron:
 - ▶ Linear (affine) mapping: $s_1 = w_1^\top x + b_1$
 - ▶ Non-linear activation function: f :
 $\hat{y}_1 = f(s_1)$
- ▶ Linear mapping parameters:
 - ▶ $w_1 = \{w_{11}, \dots, w_{m1}\} \in \mathbb{R}^m$
 - ▶ $b_1 \in \mathbb{R}$



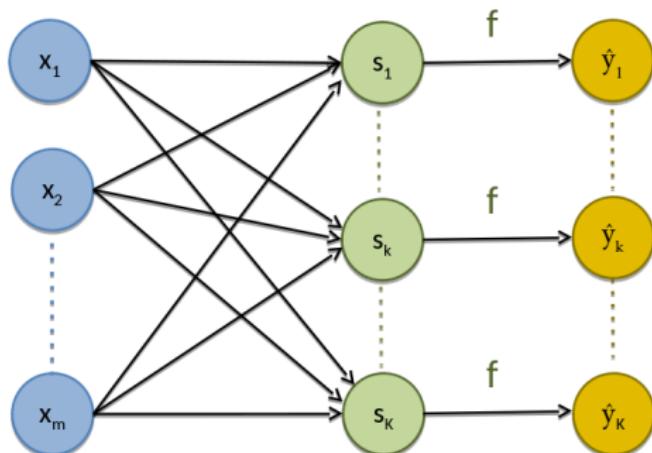
Perceptron and Multi-Class Classification

- ▶ Input x in \mathbb{R}^m
- ▶ Output neuron \hat{y}_k is a formal neuron:
 - ▶ Linear (affine) mapping: $s_k = w_k^\top x + b_k$
 - ▶ Non-linear activation function: f :
 $\hat{y}_k = f(s_k)$
- ▶ Linear mapping parameters:
 - ▶ $w_k = \{w_{1k}, \dots, w_{mk}\} \in \mathbb{R}^m$
 - ▶ $b_k \in \mathbb{R}$



Perceptron and Multi-Class Classification

- ▶ Input x in \mathbb{R}^m ($1 \times m$), output \hat{y} : concatenation of K formal neurons
- ▶ Linear (affine) mapping \sim matrix multiplication: $s = xW + b$
 - ▶ W matrix of size $m \times K$ - columns are w_k
 - ▶ b : bias vector - size $1 \times K$
- ▶ Element-wise non-linear activation: $\hat{y} = f(s)$



$$\hat{y} = f(xW + b)$$

$1 \times m \quad m \times K \quad 1 \times K$

$1 \times K$

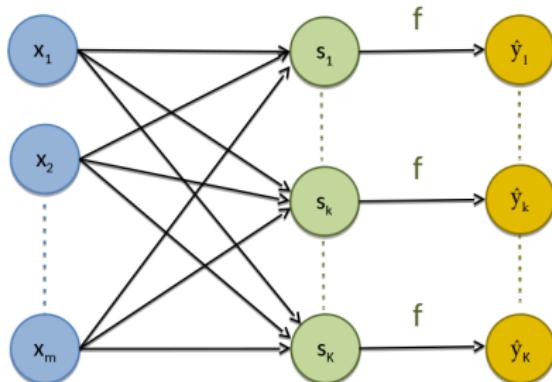
Perceptron and Multi-Class Classification

- ▶ **Soft-max Activation:**

$$\hat{y}_k = f(s_k) = \frac{e^{s_k}}{\sum_{k'=1}^K e^{s_{k'}}}$$

- ▶ Note that $f(s_k)$ depends on the other $s_{k'}$,
the arrow is a functional link
- ▶ **Probabilistic interpretation for multi-class classification:**

- ▶ Each output neuron \Leftrightarrow class
- ▶ $\hat{y}_k \sim P(k|x, W)$



⇒ **Logistic Regression (LR) Model!**

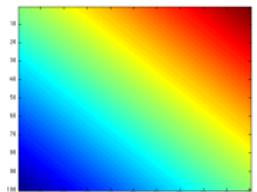
2d Toy Example for Multi-Class Classification

- $x = \{x_1, x_2\} \in [-5; 5] \times [-5; 5]$, \hat{y} : 3 outputs (classes)

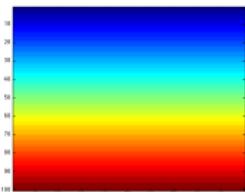
$$\mathbf{w}_1 = [1; 1], b_1 = -2$$

Linear mapping for each class:

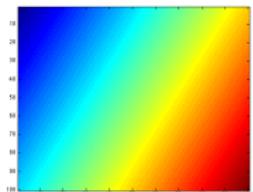
$$s_k = \mathbf{w}_k^\top x + b_k$$



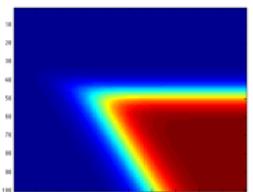
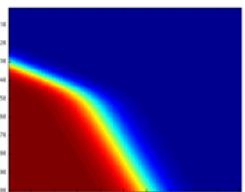
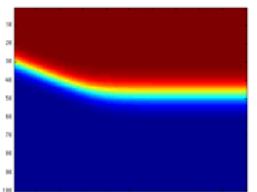
$$\mathbf{w}_2 = [0; -1], b_2 = 1$$



$$\mathbf{w}_3 = [1; -0.5], b_3 = 10$$



Soft-max output:
 $P(k|x, W)$

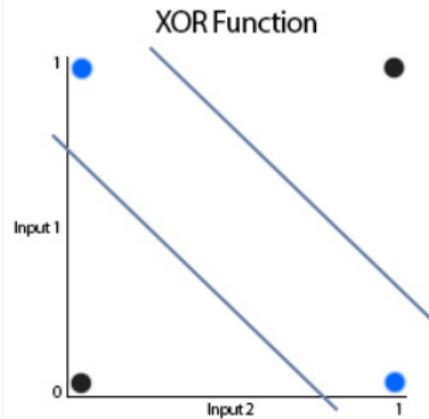
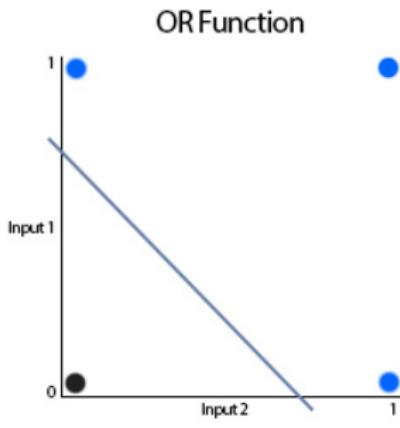


$$k^* = \arg\max_k P(k|x, W)$$

Beyond Linear Classification

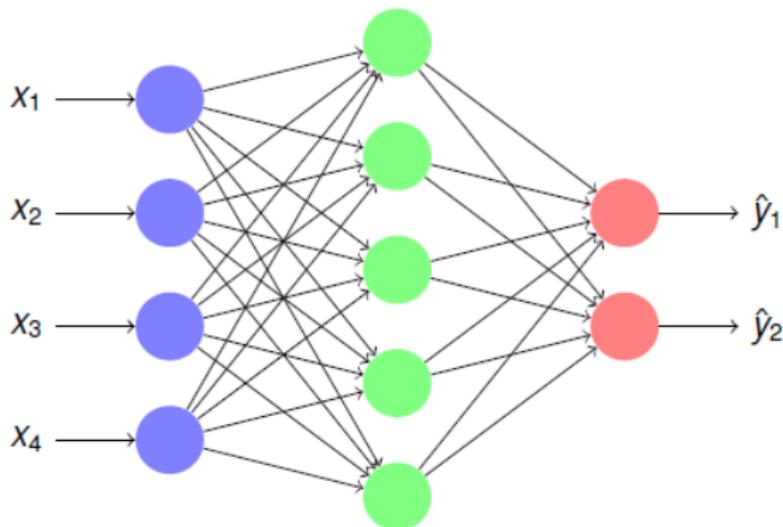
X-OR Problem

- ▶ Logistic Regression (LR): NN with 1 input layer & 1 output layer
- ▶ LR: limited to linear decision boundaries
- ▶ **X-OR: NOT 1 and 2 OR NOT 2 AND 1**
 - ▶ **X-OR: Non linear decision function**



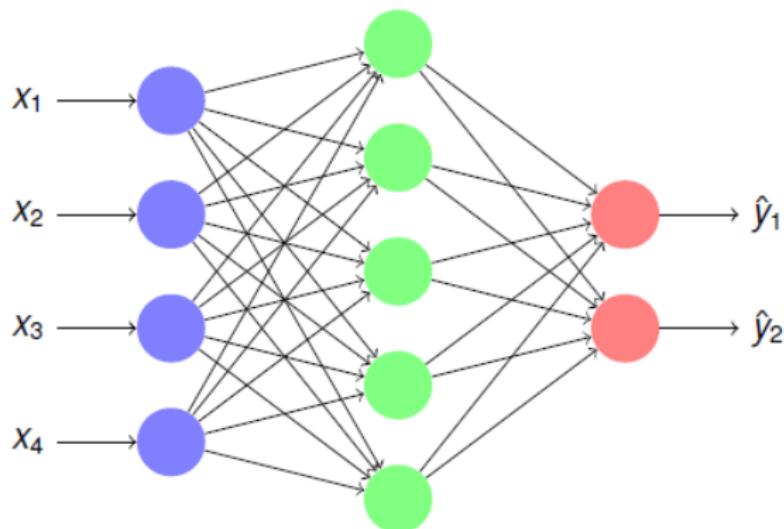
Beyond Linear Classification

- ▶ LR: limited to linear boundaries
- ▶ Solution: add a layer!
- ▶ Input x in \mathbb{R}^m , e.g. $m = 4$
- ▶ Output \hat{y} in \mathbb{R}^K (K # classes), e.g. $K = 2$
- ▶ **Hidden layer h in \mathbb{R}^L**



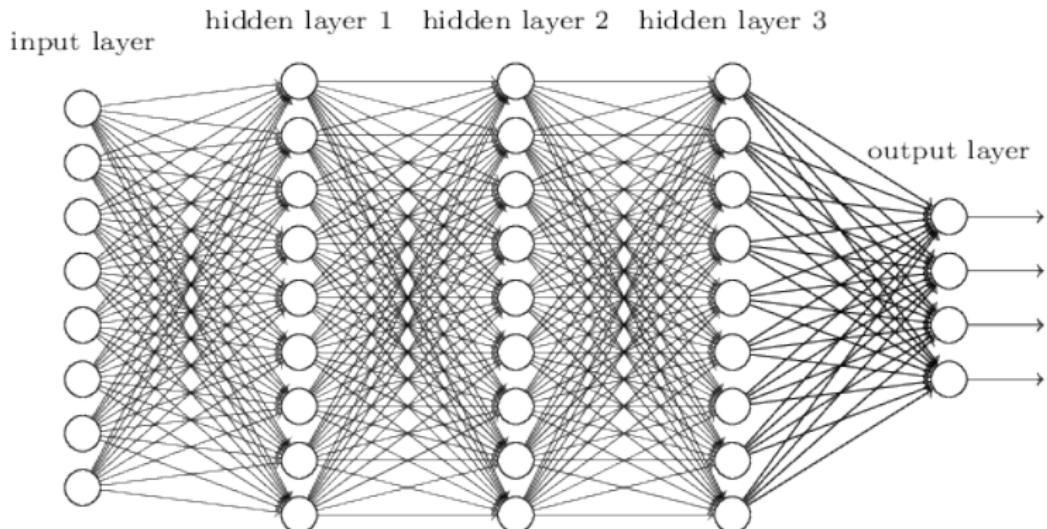
Multi-Layer Perceptron

- ▶ **Hidden layer h :** x projection to a new space \mathbb{R}^L
 - ▶ Neural Net with ≥ 1 hidden layer:
Multi-Layer Perceptron (MLP)
 - ▶ h : intermediate representations of x for classification \hat{y} :
- ▶ $h = f(xW_1 + b_1)$
 f non-linear activation,
 $s = hW_2 + b_2$
 $\hat{y} = \text{SoftMax}(s)$
 - ▶ **Mapping from x to \hat{y} : non-linear boundary!**
⇒ **non-linear activation f crucial!**



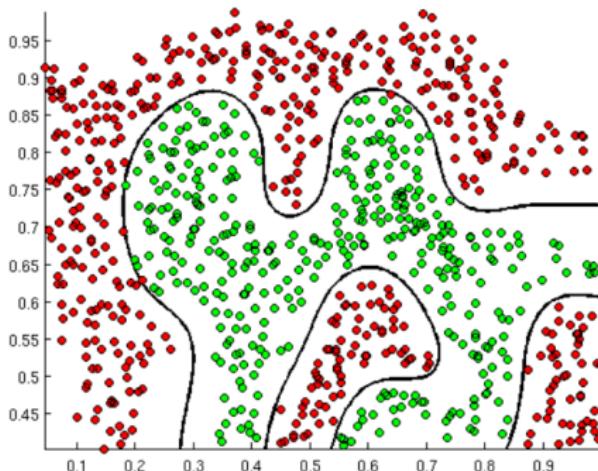
Deep Neural Networks

- ▶ Adding more hidden layers: Deep Neural Networks (DNN) \Rightarrow **Basis of Deep Learning**
- ▶ Each layer h^l projects layer h^{l-1} into a new space
- ▶ Gradually learning intermediate representations useful for the task



Conclusion

- ▶ Deep Neural Networks: applicable to classification problems with non-linear decision boundaries



- ▶ Visualize prediction from fixed model parameters
- ▶ Reverse problem: **Supervised Learning**

Outline

Training Deep Neural Networks

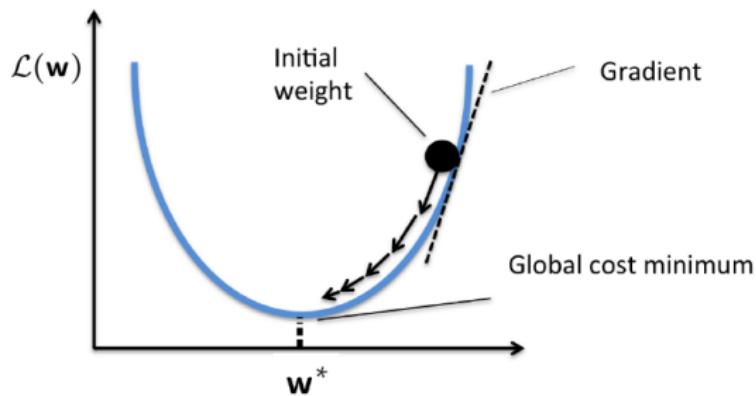
Training Multi-Layer Perceptron (MLP)

- ▶ Input x , output y
- ▶ A parametrized (w) model $x \Rightarrow y$: $f_w(x_i) = \hat{y}_i$
- ▶ Supervised context:
 - ▶ Training set $\mathcal{A} = \{(x_i, y_i^*)\}_{i \in \{1, 2, \dots, N\}}$
 - ▶ Loss function $\ell(\hat{y}_i, y_i^*)$ for each annotated pair (x_i, y_i^*)
 - ▶ Goal: Minimizing average loss \mathcal{L} over training set: $\mathcal{L}(w) = \frac{1}{N} \sum_{i=1}^N \ell(\hat{y}_i, y_i^*)$
- ▶ Assumptions: parameters $w \in \mathbb{R}^d$ continuous, \mathcal{L} differentiable
- ▶ Gradient $\nabla_w = \frac{\partial \mathcal{L}}{\partial w}$: steepest direction to decrease loss $\mathcal{L}(w)$



MLP Training

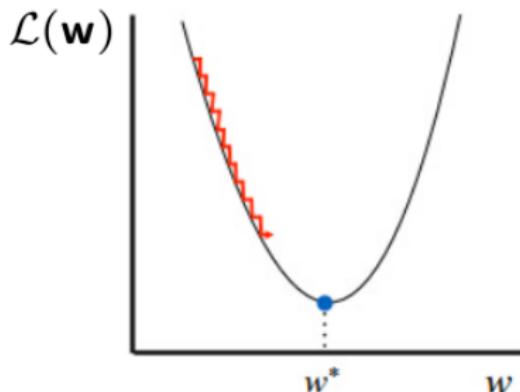
- ▶ Gradient descent algorithm:
 - ▶ Initialize parameters w
 - ▶ Update: $w^{(t+1)} = w^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial w}$
 - ▶ Until convergence, e.g. $\|\nabla_w\|^2 \approx 0$



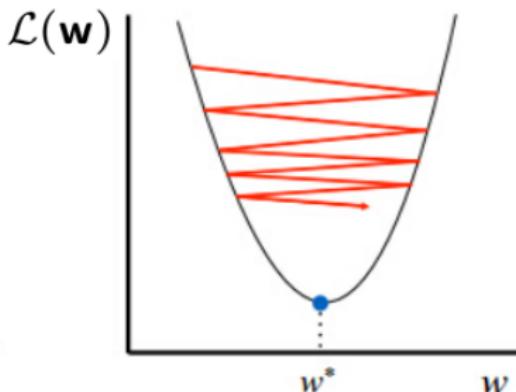
Gradient Descent

Update rule: $\boxed{\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}}}$ η learning rate

- ▶ Convergence ensured ? \Rightarrow provided a "well chosen" learning rate η



Too small: converge
very slowly



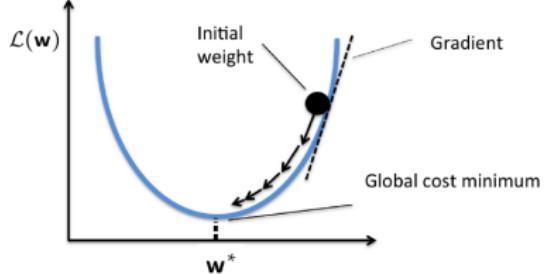
Too big: overshoot and
even diverge

Gradient Descent

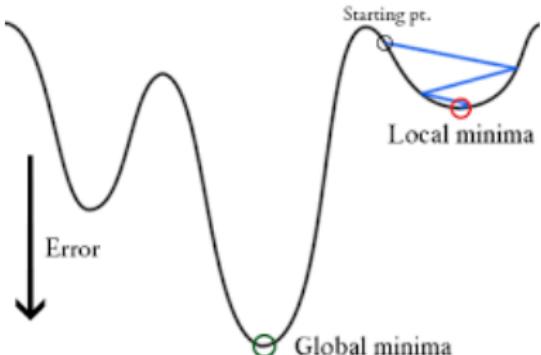
Update rule: $w^{(t+1)} = w^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial w}$

- ▶ **Global minimum ?**

⇒ convex a) vs non convex b) loss $\mathcal{L}(w)$



a) Convex function



a) Non convex function

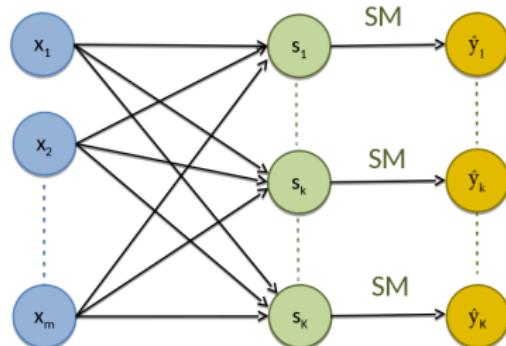
Supervised Learning: Multi-Class Classification

- ▶ Logistic Regression for multi-class classification
- ▶ $s_i = x_i W + b$
- ▶ Soft-Max (SM): $\hat{y}_k \sim P(k/x_i, W, b) = \frac{e^{s_k}}{\sum_{k'=1}^K e^{s_{k'}}}$
- ▶ Supervised loss function: $\mathcal{L}(W, b) = \frac{1}{N} \sum_{i=1}^N \ell(\hat{y}_i, y_i^*)$

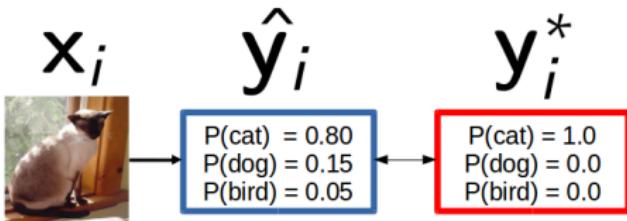
1. $y \in \{1; 2; \dots; K\}$

2. $\hat{y}_i = \arg \max_k P(k/x_i, W, b)$

3. $\ell_{0/1}(\hat{y}_i, y_i^*) = \begin{cases} 1 & \text{if } \hat{y}_i \neq y_i^* \\ 0 & \text{otherwise} \end{cases} : \text{0/1 loss}$



Logistic Regression Training Formulation



- ▶ **Input x_i , ground truth output supervision y_i^***
- ▶ One hot-encoding for y_i^* :

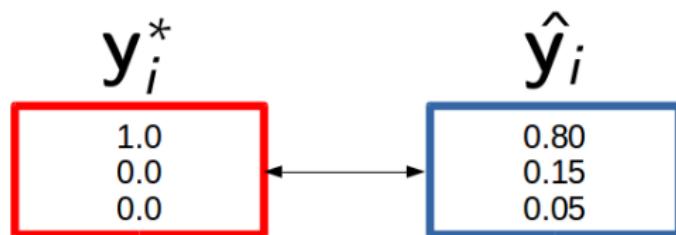
$$y_{c,i}^* = \begin{cases} 1 & \text{if } c \text{ is the ground truth class for } x_i \\ 0 & \text{otherwise} \end{cases}$$

Logistic Regression Training Formulation

- ▶ Loss function: multi-class Cross-Entropy (CE) ℓ_{CE}
- ▶ ℓ_{CE} : Kullback-Leiber divergence between y_i^* and \hat{y}_i

$$\ell_{CE}(\hat{y}_i, y_i^*) = KL(y_i^*, \hat{y}_i) = - \sum_{c=1}^K y_{c,i}^* \log(\hat{y}_{c,i}) = -\log(\hat{y}_{c^*,i})$$

- ▶ \triangle KL asymmetric: $KL(\hat{y}_i, y_i^*) \neq KL(y_i^*, \hat{y}_i)$ \triangle

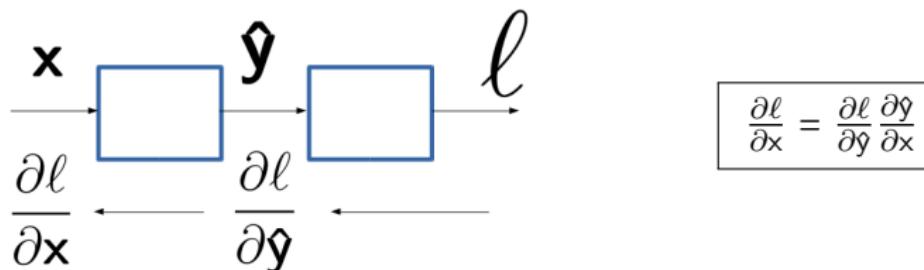


$$KL(y_i^*, \hat{y}_i) = -\log(\hat{y}_{c^*,i}) = -\log(0.8) \approx 0.22$$

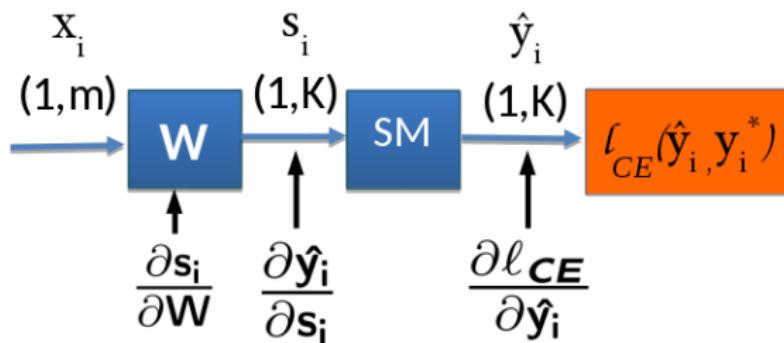
Logistic Regression Training

- ▶ $\mathcal{L}_{CE}(W, b) = \frac{1}{N} \sum_{i=1}^N \ell_{CE}(\hat{y}_i, y_i^*) = -\frac{1}{N} \sum_{i=1}^N \log(\hat{y}_{c^*, i})$
- ▶ ℓ_{CE} smooth convex upper bound of $\ell_0/1$
⇒ **gradient descent optimization**
- ▶ Gradient descent: $W^{(t+1)} = W^{(t)} - \eta \frac{\partial \mathcal{L}_{CE}}{\partial W}$ ($b^{(t+1)} = b^{(t)} - \eta \frac{\partial \mathcal{L}_{CE}}{\partial b}$)
- ▶ **MAIN CHALLENGE:** computing $\frac{\partial \mathcal{L}_{CE}}{\partial W} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \ell_{CE}}{\partial W}$?
⇒ Key Property: chain rule $\frac{\partial x}{\partial z} = \frac{\partial x}{\partial y} \frac{\partial y}{\partial z}$
⇒ **Backpropagation of gradient error!**

Chain Rule



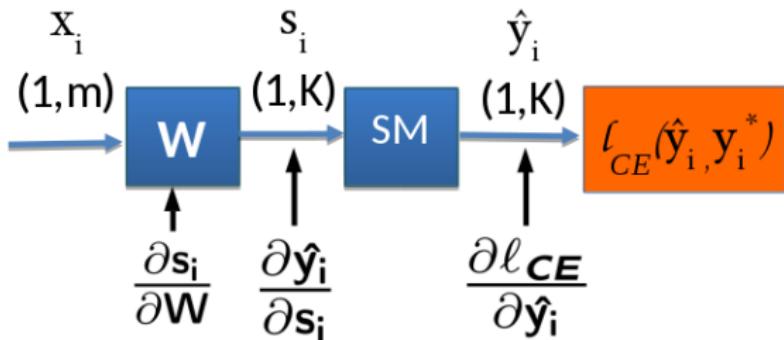
- ▶ Logistic regression: $\frac{\partial \ell_{CE}}{\partial W} = \frac{\partial \ell_{CE}}{\partial y_i} \frac{\partial y_i}{\partial s_i} \frac{\partial s_i}{\partial W}$



Logistic Regression Training: Backpropagation

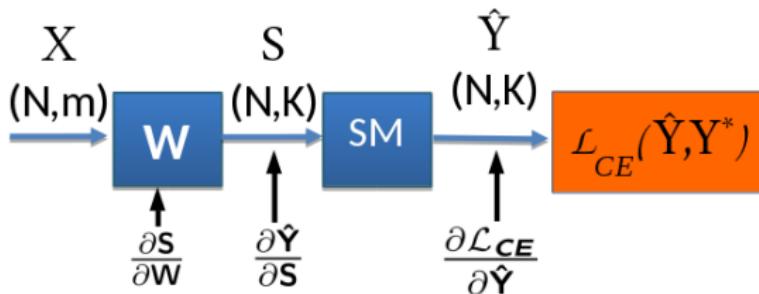
$$\frac{\partial \ell_{CE}}{\partial W} = \frac{\partial \ell_{CE}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_i} \frac{\partial s_i}{\partial W}, \ell_{CE}(\hat{y}_i, y_i^*) = -\log(\hat{y}_{c^*, i}) \Rightarrow \text{Update for 1 example:}$$

- ▶ $\frac{\partial \ell_{CE}}{\partial \hat{y}_i} = \frac{-1}{\hat{y}_{c^*, i}} = \frac{-1}{\hat{y}_i} \odot \delta_{c, c^*}$
- ▶ $\frac{\partial \ell_{CE}}{\partial s_i} = \hat{y}_i - y_i^* = \delta_i^y$
- ▶ $\frac{\partial \ell_{CE}}{\partial W} = x_i^T \delta_i^y$



Logistic Regression Training: Backpropagation

- ▶ Whole dataset: data matrix X ($N \times m$), label matrix \hat{Y} , Y^* ($N \times K$)
- ▶ $\mathcal{L}_{CE}(W, b) = -\frac{1}{N} \sum_{i=1}^N \log(\hat{y}_{c^*, i})$, $\frac{\partial \mathcal{L}_{CE}}{\partial W} = \frac{\partial \mathcal{L}_{CE}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial S} \frac{\partial S}{\partial W}$



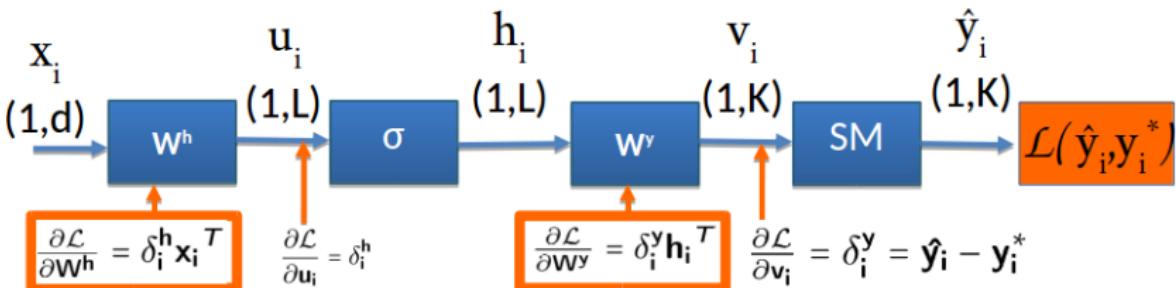
- ▶
$$\frac{\partial \mathcal{L}_{CE}}{\partial s} = \hat{Y} - Y^* = \Delta^y$$
- ▶
$$\frac{\partial \mathcal{L}_{CE}}{\partial W} = X^T \Delta^y$$

Perceptron Training: Backpropagation

$$\text{ReLU} \rightarrow \frac{d\text{ReLU}}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \quad \text{easy backprop}$$

- ▶ Perceptron vs Logistic Regression: adding hidden layer (sigmoid)
- ▶ **Goal:** Train parameters W^y and W^h (+bias) with Backpropagation

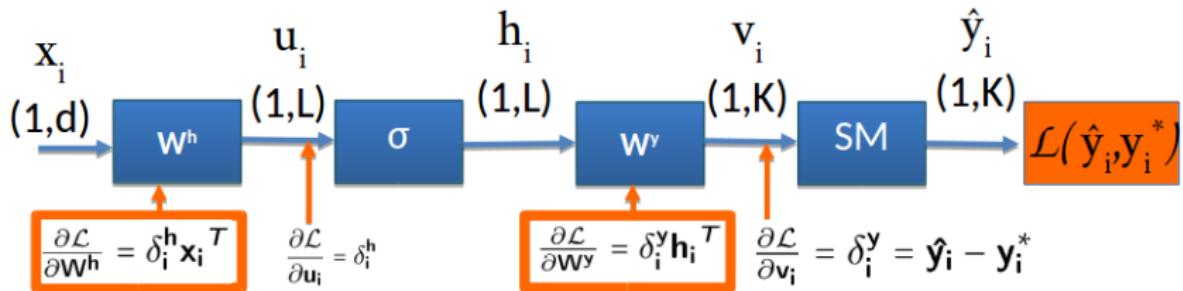
⇒ computing $\frac{\partial \mathcal{L}_{CE}}{\partial W^y} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \ell_{CE}}{\partial W^y}$ and $\frac{\partial \mathcal{L}_{CE}}{\partial W^h} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \ell_{CE}}{\partial W^h}$



- ▶ Last hidden layer ~ Logistic Regression
- ▶ First hidden layer: $\frac{\partial \ell_{CE}}{\partial W^h} = x_i^T \frac{\partial \ell_{CE}}{\partial u_i} \Rightarrow$ **computing** $\frac{\partial \ell_{CE}}{\partial u_i} = \delta_i^h$

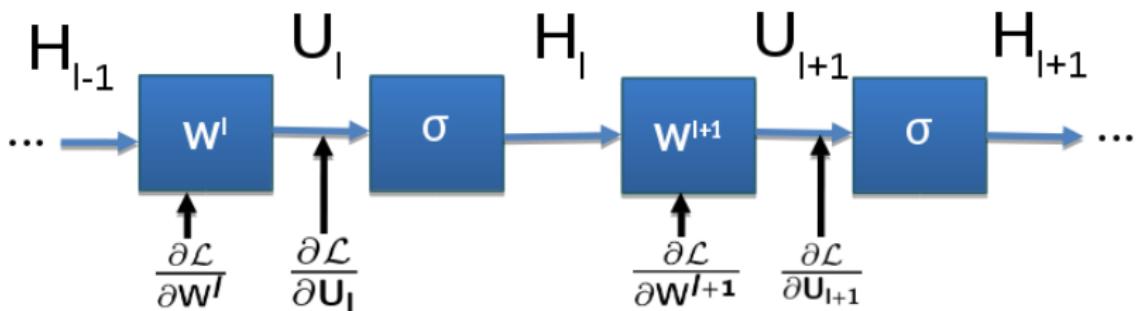
Perceptron Training: Backpropagation

- ▶ Computing $\frac{\partial \ell_{CE}}{\partial u_i} = \delta_i^h \Rightarrow$ use chain rule: $\frac{\partial \ell_{CE}}{\partial u_i} = \frac{\partial \ell_{CE}}{\partial v_i} \frac{\partial v_i}{\partial h_i} \frac{\partial h_i}{\partial u_i}$
- ▶ ... Leading to: $\frac{\partial \ell_{CE}}{\partial u_i} = \delta_i^h = \delta_i^y {}^T W^y \odot \sigma'(h_i) = \delta_i^y {}^T W^y \odot (h_i \odot (1 - h_i))$



Deep Neural Network Training: Backpropagation

- ▶ Multi-Layer Perceptron (MLP): adding more hidden layers
- ▶ Backpropagation update ~ Perceptron: **assuming** $\frac{\partial \mathcal{L}}{\partial U_{l+1}} = \Delta^{l+1}$ known
 - ▶ $\boxed{\frac{\partial \mathcal{L}}{\partial W^{l+1}} = H_l^T \Delta^{l+1}}$
 - ▶ Computing $\frac{\partial \mathcal{L}}{\partial U_l} = \Delta^l$ ($= \Delta^{l+1 T} W^{l+1} \odot H_l \odot (1 - H_l)$ sigmoid)
 - ▶ $\boxed{\frac{\partial \mathcal{L}}{\partial W^l} = H_{l-1}^T \Delta^{h_l}}$



Outline

Introduction to Neural nets

Training Deep Neural Networks

Introduction to Statistical Decision Theory

Statistical Decision Theory

Let $X \in \mathbb{R}^P$ be a real-valued random input vector,
 $Y \in \mathbb{R}$ be a real-valued random output variable,
 f be a function from \mathbb{R}^P to \mathbb{R} (e.g. f_w with parameters w or a deep neural network). Here, \hat{Y} is the prediction for Y :

$$X \xrightarrow{f()} \hat{Y} = f(X)$$

How to find the best function f ?

1. Measure the difference between $f(X)$ and Y .

Define an error/loss function $L(Y, f(X))$, which penalizes errors in prediction. The loss function L is a non-negative, real-valued function.
Examples of common loss functions:

- ▶ **Squared Error Loss: (regression)**
$$L(Y, f(X)) = (Y - f(X))^2$$
- ▶ **0-1 Loss Function:**

$$L(Y, f(X)) = \begin{cases} 1 & \text{if } Y \neq f(X), \\ 0 & \text{if } Y = f(X) \end{cases}$$

- ▶ **Cross-Entropy Loss ($L = L_{CE}$) for classification:**

- ▶ Assume here that Y is a one-hot vector.
- ▶ Let c^* be the index of the correct class.

$$L(Y, f(X)) = -\log(\hat{Y}_{c^*})$$

Statistical Decision Theory (cont'd)

images as stochastic variables
 $P(X, Y)$

How to find the best function f ? (continued)

- As we consider random variables and probability spaces, assume a joint distribution $P(X, Y)$ exists

The criterion to minimize in choosing f is the Expected Prediction Error, also known as the Risk:

$$R(f) = EPE(f) = \mathbb{E}_{P(X, Y)}[L(Y, f(X))]$$

The risk can be expressed as an integral: since we suppose to be in a continuous domain

$$R(f) = \iint L(Y, f(X)) dP(X, Y)$$

Example: For $L = (Y - f(X))^2$: $R(f) = \iint (y - f(x))^2 p(x, y) dx dy$

(Final) Goal: Find a hypothesis f^* among a fixed class of functions \mathcal{F} for which the risk is minimal:

$$f^* = \operatorname{Argmin}_{f \in \mathcal{F}} R(f) \quad (1)$$

In-depth Problem and Machine Learning solution

Cannot solve (1)?!

Problem: $P(X, Y)$ unknown $\Rightarrow R(f)$ cannot be computed.

Solution:

- ▶ Fix \mathcal{F} to a parameterized family f_w where $w \in \mathbb{R}^d$
- ▶ Learn from examples! to approximate $R(f_w)$

*fix the architecture,
optimize $w \rightarrow f_w \rightarrow \min R(f_w)$*

Supervised Learning:

- ▶ $\mathcal{A}_N = \{x_i, y_i\}_{i=1..N}$ Training set implicit use of $P(X, Y)$ by *iid sampling*
- ▶ $x_i \longrightarrow f_w(x_i) = \hat{y}_i \longleftrightarrow y_i$

Empirical Risk Minimization

We can approximate $R(f)$ by averaging the loss function on \mathcal{A}_N :

$$R(f) = \iint L(Y, f(X)) dP(X, Y) \Rightarrow \text{ERM}(f_w) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}(y_i, f_w(x_i))$$

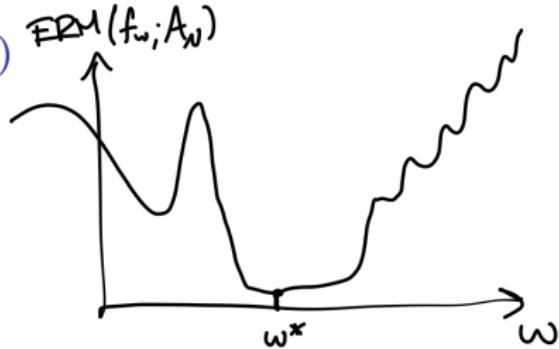
ERM: Empirical Risk Minimization

Requirement: $\{x_i, y_i\} \sim P(x, y)$ and N large \Rightarrow "good" approximation

New Objective: $w^* = \underset{w \in \mathbb{R}^d}{\operatorname{Argmin}} \text{ERM}(f_w; \mathcal{A}_N) = \underset{w \in \mathbb{R}^d}{\operatorname{Argmin}} \frac{1}{N} \sum_{i=1}^N \mathcal{L}(y_i, f_w(x_i))$

Supervised learning algorithm to solve this **optimization problem**.

Optimization Depending on $\mathcal{L}(w)$



- ▶ Depending on $\mathcal{L}(w)$, optimization is not always easy.
- ▶ In this course, $\mathcal{L}(w)$ is (supposed) differentiable for $w \in \mathbb{R}^d$.
- ▶ **Definition: Gradient** $\nabla \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial w} \right] \in \mathbb{R}^d$.

Gradient Descent Algorithm

Gradient Descent Algorithm:

- ▶ **Initialize:** $w^{(0)}$
- ▶ **Repeat:** $w^{(t+1)} = w^{(t)} - \eta \nabla \mathcal{L}(w^{(t)})$
- ▶ **Until:** Convergence $\|\nabla \mathcal{L}(w^{(t+1)})\|^2 \approx 0$

Remark about convergence:

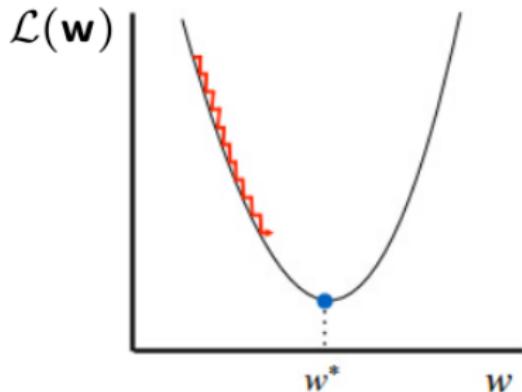
$$\begin{aligned} 0 &\leq \mathcal{L}(w^{(t+1)}) & f(x - x_0) &\approx f(x) - x_0 \cdot f'(x) \\ &= \mathcal{L}\left(w^{(t)} - \eta \nabla \mathcal{L}(w^{(t)})\right) \\ &\approx \mathcal{L}\left(w^{(t)}\right) - \eta \nabla \mathcal{L}(w^{(t)})^T \cdot \nabla \mathcal{L}(w^{(t)}) \quad (\text{first order approximation}) \\ &= \mathcal{L}\left(w^{(t)}\right) - \eta \|\nabla \mathcal{L}(w^{(t)})\|^2 \\ &\leq \mathcal{L}\left(w^{(t)}\right) \end{aligned}$$

⇒ local Convergence

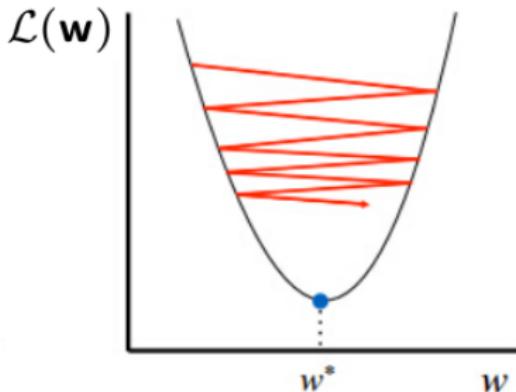
Gradient Descent

Update rule: $w^{(t+1)} = w^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial w}$ η learning rate

- ▶ Convergence ensured ? \Rightarrow provided a "well chosen" learning rate η



Too small: converge
very slowly



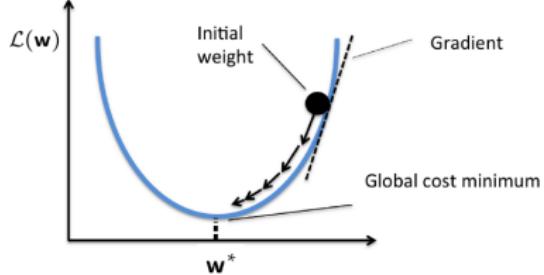
Too big: overshoot and
even diverge

Gradient Descent

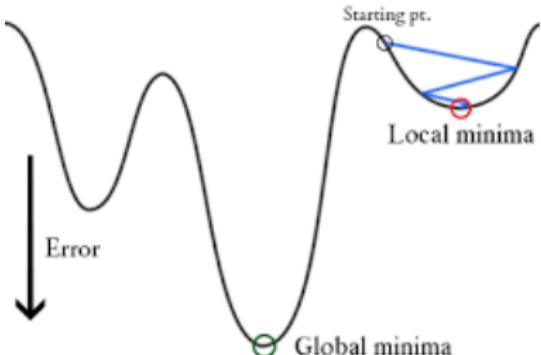
Update rule: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}}$

► Global minimum ?

⇒ convex a) vs non convex b) loss $\mathcal{L}(\mathbf{w})$



a) Convex function



a) Non convex function

Neural Network Training: Optimization Issues

- ▶ Classification loss over training set (vectorized w , b ignored):

$$\mathcal{L}_{CE}(w) = \frac{1}{N} \sum_{i=1}^N \ell_{CE}(\hat{y}_i, y_i^*) = -\frac{1}{N} \sum_{i=1}^N \log(\hat{y}_{c^*, i})$$

- ▶ Gradient descent optimization:

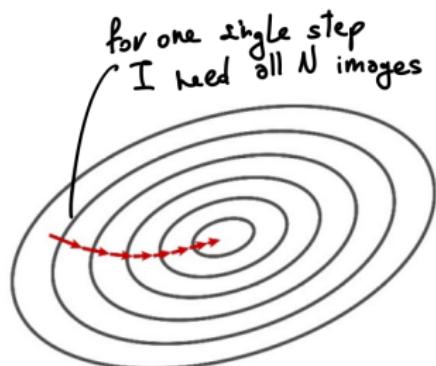
$$w^{(t+1)} = w^{(t)} - \eta \frac{\partial \mathcal{L}_{CE}}{\partial w} (w^{(t)}) = w^{(t)} - \eta \nabla_w^{(t)}$$

- ▶ Gradient $\nabla_w^{(t)} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \ell_{CE}(\hat{y}_i, y_i^*)}{\partial w} (w^{(t)})$ **linearly scales**

wrt:

- ▶ w dimension
- ▶ **Training set size**

⇒ **Too slow even for moderate dimensionality & dataset size!**



Stochastic Gradient Descent

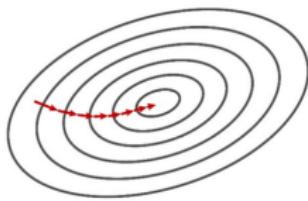
- ▶ **Solution:** approximate $\nabla_w^{(t)} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \ell_{CE}(\hat{y}_i, y_i^*)}{\partial w} (w^{(t)})$ with subset of examples
⇒ **Stochastic Gradient Descent (SGD)**

- ▶ **Use a single example (online):** very coarse approximation of $\nabla_w^{(t)}$

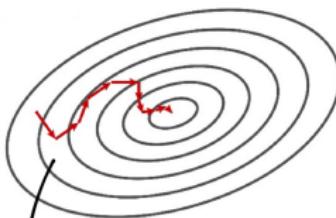
$$\nabla_w^{(t)} \approx \frac{\partial \ell_{CE}(\hat{y}_i, y_i^*)}{\partial w} (w^{(t)})$$

- ▶ **Mini-batch:** use $B < N$ examples:

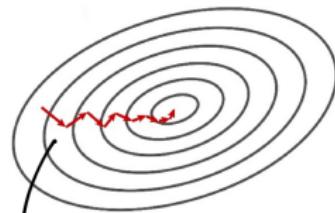
$$\nabla_w^{(t)} \approx \frac{1}{B} \sum_{i=1}^B \frac{\partial \ell_{CE}(\hat{y}_i, y_i^*)}{\partial w} (w^{(t)})$$



Full gradient



SGD (online)



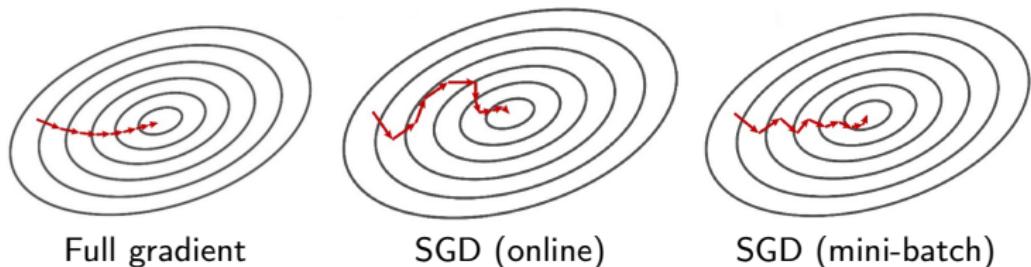
SGD (mini-batch)

depends strongly on
the single example $\frac{\partial \ell}{\partial w}$

needs only B examples

Stochastic Gradient Descent

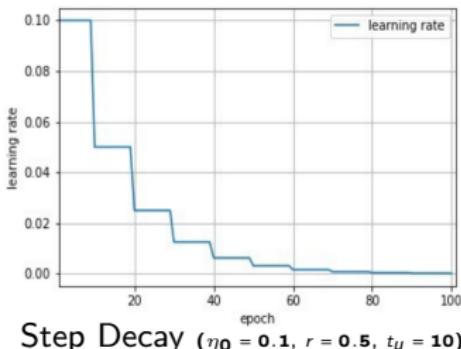
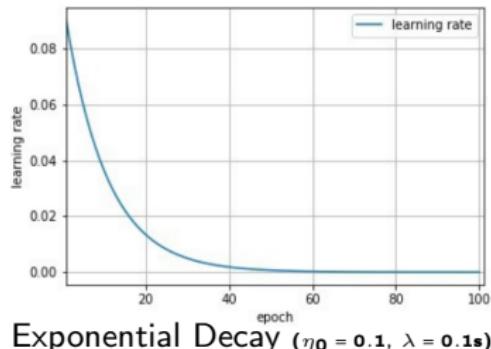
- ▶ SGD: approximation of the true Gradient ∇_w !
 - ▶ Noisy gradient can lead to bad direction, increase loss
 - ▶ **BUT:** much more parameter updates: online $\times N$, mini-batch $\times \frac{N}{B}$
 - ▶ **Faster convergence**, at the core of Deep Learning for large scale datasets



B depends on the
GPU RAM size

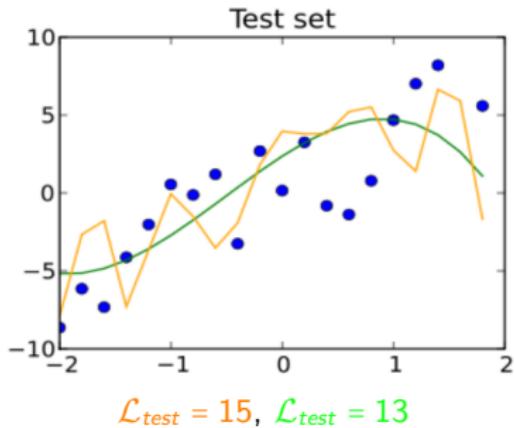
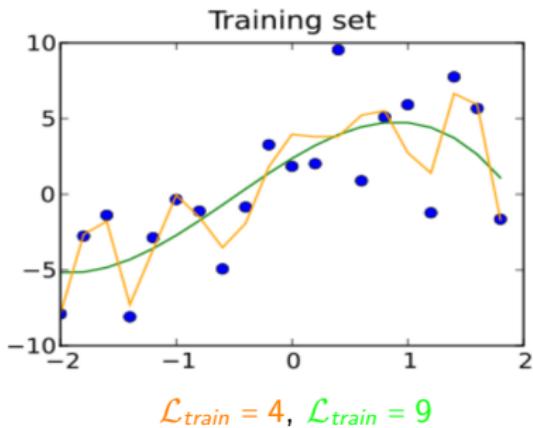
Optimization: Learning Rate Decay

- ▶ Gradient descent optimization: $w^{(t+1)} = w^{(t)} - \eta \nabla_w^{(t)}$
- ▶ η setup? \Rightarrow open question
- ▶ Learning Rate Decay: decrease η during training progress
 - ▶ Inverse (time-based) decay: $\eta_t = \frac{\eta_0}{1+r \cdot t}$, r decay rate
 - ▶ Exponential decay: $\eta_t = \eta_0 \cdot e^{-\lambda t}$
 - ▶ Step Decay $\eta_t = \eta_0 \cdot r^{\frac{t}{t_u}} \dots$



Generalization and Overfitting

- ▶ **Learning:** minimizing classification loss \mathcal{L}_{CE} over training set
 - ▶ Training set: sample representing data vs labels distributions
 - ▶ Ultimate goal: train a prediction function with low prediction error on the **true (unknown) data distribution**



⇒ Optimization ≠ Machine Learning!
⇒ Generalization / Overfitting!

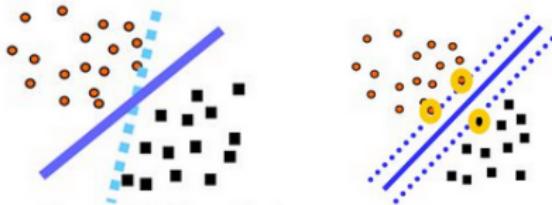
Regularization

- ▶ **Regularization:** improving generalization, i.e. test (\neq train) performances
- ▶ Structural regularization: add $Prior R(w)$ in training objective:

$$\mathcal{L}(w) = \mathcal{L}_{CE}(w) + \alpha R(w)$$

- ▶ L^2 regularization: weight decay, $R(w) = ||w||^2$
 - ▶ Commonly used in neural networks
 - ▶ Theoretical justifications, generalization bounds (SVM)
- ▶ Other possible $R(w)$: L^1 regularization, dropout, etc

take the NN
with the smaller
weights
↓
Simpler



L^2 regularization: interpretation

if $x - x'$ is small, i.e. the 2 imgs are close,
then also the predicted label should be close.
(\hat{y}, \hat{y}')

- "Smooth" interpretation of L^2 regularization, Cauchy-Schwarz:

$$(\langle w, (x - x') \rangle)^2 \leq \|w\|^2 \|x - x'\|^2$$

Limit the divergence

- Controlling L^2 norm $\|w\|^2$: "small" variation between inputs x and x'
⇒ small variation in neuron prediction $\langle w, x \rangle$ and $\langle w, x' \rangle$
⇒ Supports simple, i.e. smoothly varying prediction models

Regularization and hyper-parameters

- ▶ **Neural networks:** hyper-parameters to tune:
 - ▶ **Training parameters:** learning rate, weight decay, learning rate decay, # epochs, etc
 - ▶ **Architectural parameters:** number of layers, number neurones, non-linearity type, etc
- ▶ **Hyper-parameters tuning:** ⇒ improve generalization: estimate performances on a validation set

