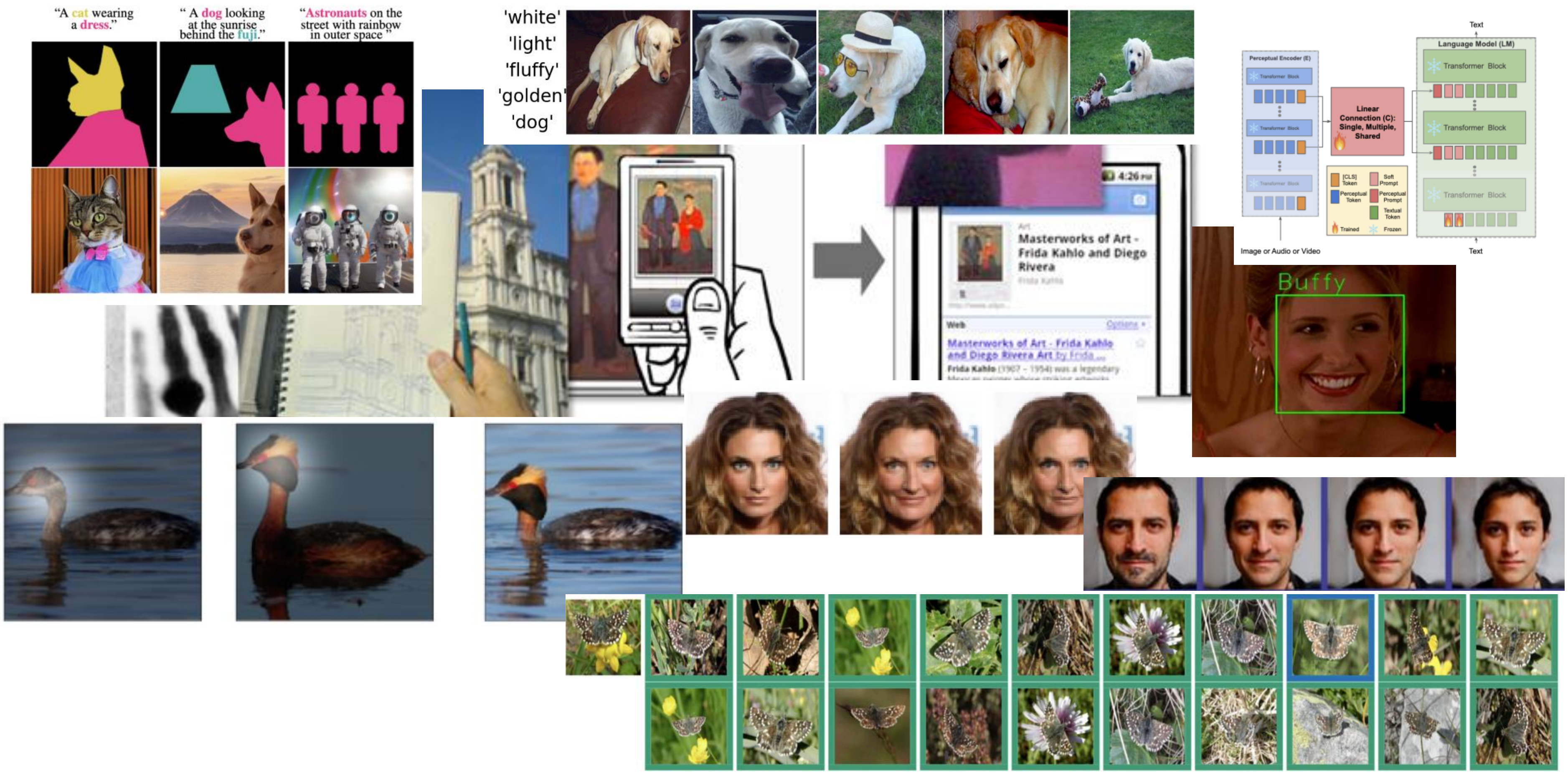


RDFIA: deep learning for Vision

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

Matthieu Cord
Sorbonne University

Course Outline

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

1. Intro to Computer Vision and Machine Learning
2. Intro to Neural Networks + Machine Learning theory
3. Neural Nets for Image Classification
4. Large ConvNets
5. Vision Transformers
6. Segmentation, Transfer learning and domain adaptation
7. Vision-Language models
8. Explaining VLMS
9. Self Supervised Learning in Vision
10. Generative models with GANs
11. Control Jan 07, 2026
12. Diffusion models
13. Bayesian deep learning

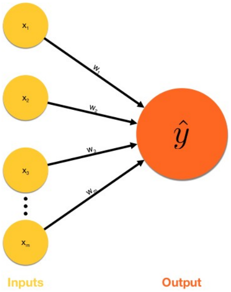
14 Uncertainty, Robustness

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

Outline

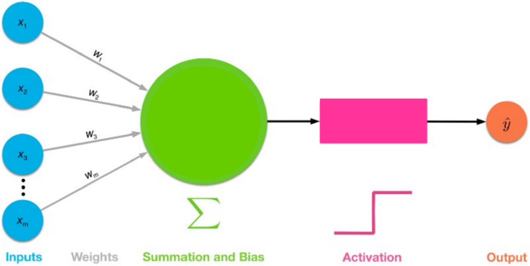
The Formal Neuron: 1943 [?]

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



The Formal Neuron: 1943 [?]

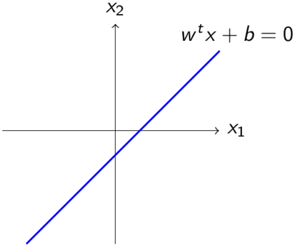
- Mapping from x to $\hat{y}$:
  1. Linear (affine) mapping: $s = \mathrm{w}^\top \mathrm{x} + b$
  2. Non-linear activation function: f: $\hat{y} = f(s)$



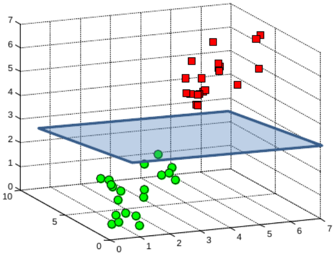
The Formal Neuron: Linear Mapping

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

2D hyperplane: line

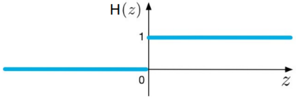


3D hyperplane: plane

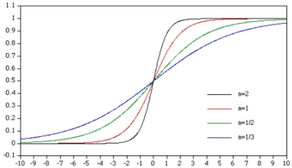


The Formal Neuron: Activation Function

- $\hat{y} = f(\mathrm{w}^\top \mathrm{x} + b)$,
- f: activation function
  - Bio-inspired choice: Step (Heaviside) function: $H(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$

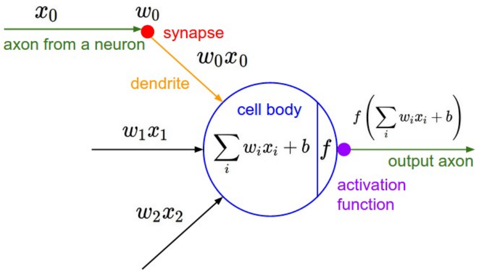


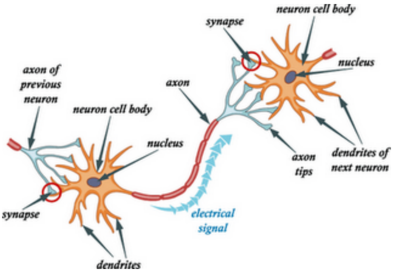
  - Popular f choices: sigmoid, tanh, ReLU, GELU, ...
  - Sigmoid: $\sigma(z) = (1 + e^{-az})^{-1}$



    - $a \uparrow$: 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: $\hat{y} = H(\mathrm{w}^\top \mathrm{x} + b)$
  - $\hat{y} = 1$ (activated) $\Leftrightarrow \mathrm{w}^\top \mathrm{x} \geq -b$
  - $\hat{y} = 0$ (unactivated) $\Leftrightarrow \mathrm{w}^\top \mathrm{x} < -b$
- **Biological Neurons:** output activated
  $\Leftrightarrow$ input weighted by synaptic weight $\geq$ threshold

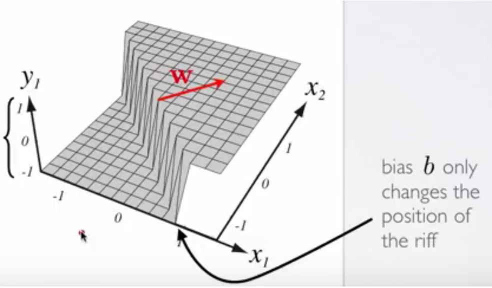
The Formal neuron: Application to Binary Classification

- Binary Classification: label input $\mathbf{x}$ as belonging to class 1 or 0
- Neuron output with sigmoid:

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

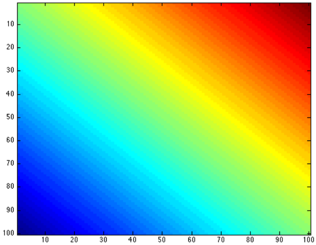
- Sigmoid: probabilistic interpretation $\Rightarrow \hat{y} \sim P(1|\mathbf{x})$
  - Input $\mathbf{x}$ classified as 1 if $P(1|\mathbf{x}) > 0.5 \Leftrightarrow \mathbf{w}^\top \mathbf{x} + b > 0$
  - Input $\mathbf{x}$ classified as 0 if $P(1|\mathbf{x}) < 0.5 \Leftrightarrow \mathbf{w}^\top \mathbf{x} + b < 0$

  $\Rightarrow$ sign($\mathbf{w}^\top \mathbf{x} + b$): linear boundary decision in input space !



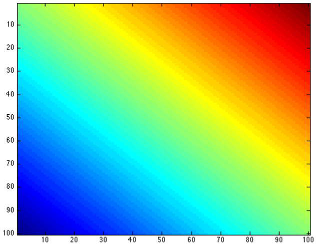
The Formal neuron: Toy Example for Binary Classification

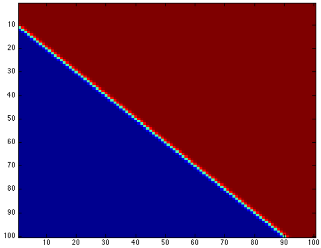
- 2d example: $m = 2$, $\mathrm{x} = \{x_1, x_2\} \in [-5; 5] \times [-5; 5]$
- Linear mapping: $\mathrm{w} = [1; 1]$ and $b = -2$
- Result of linear mapping : $s = \mathrm{w}^\top \mathrm{x} + b$

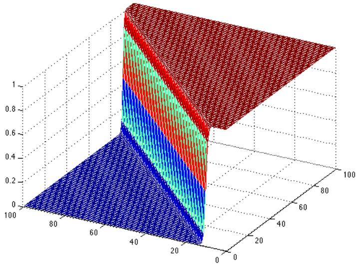


The Formal neuron: Toy Example for Binary Classification

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

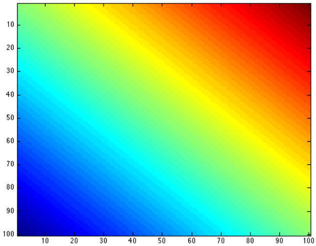


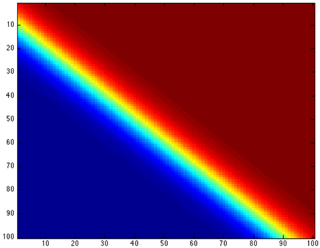


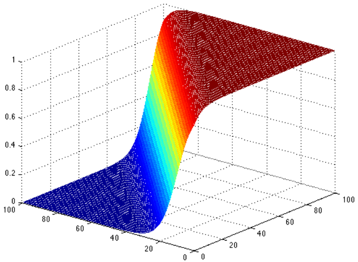


The Formal neuron: Toy Example for Binary Classification

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

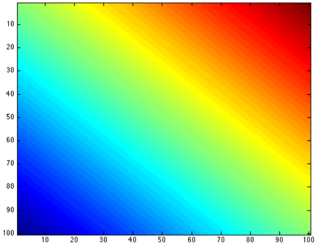


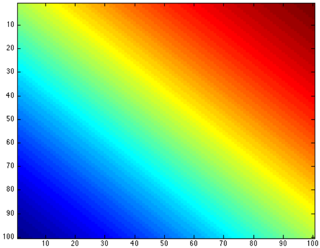


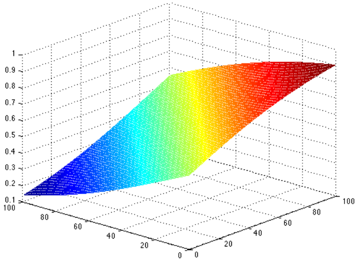


The Formal neuron: Toy Example for Binary Classification

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

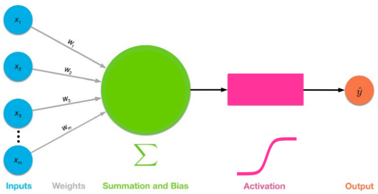


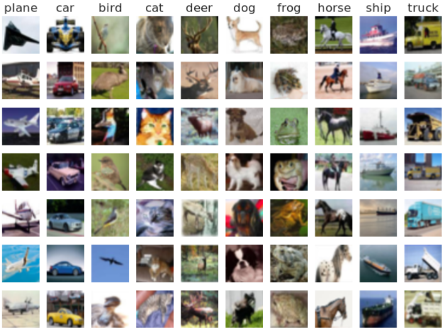




From Formal Neuron to Neural Networks

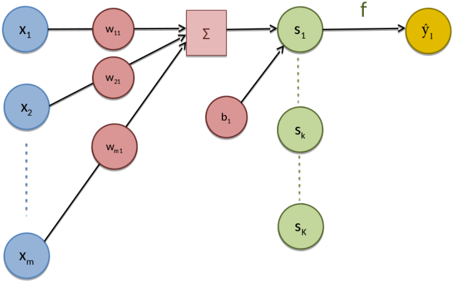
- Formal Neuron:
  1. A single scalar output
  2. Linear decision boundary for binary classification
- Single scalar output: limited for several tasks
  - Ex: multi-class classification, *e.g.* MNIST or CIFAR



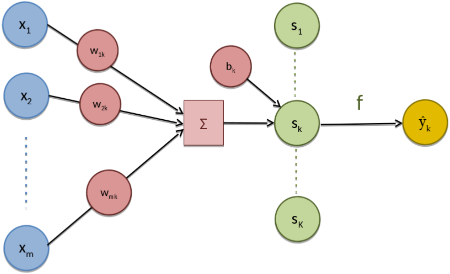


Perceptron and Multi-Class Classification

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



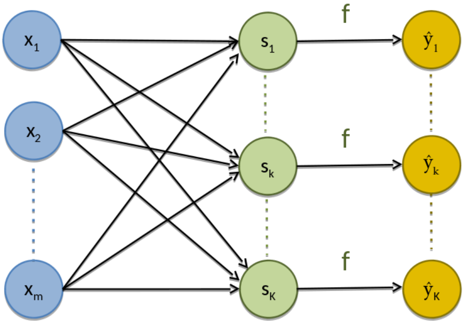
Perceptron and Multi-Class Classification



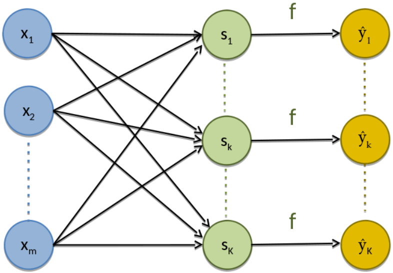
- Input x in $\mathbb{R}^m$
- Output neuron $\hat{y}_k$ is a formal neuron:
  - Linear (affine) mapping: $s_k = \mathrm{w_k}^\top \mathrm{x} + b_k$
  - Non-linear activation function: f: $\hat{y}_k = f(s_k)$
- Linear mapping parameters:
  - $\mathrm{w_k} = \{w_{1k}, ..., w_{mk}\} \in \mathbb{R}^m$
 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

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 ~ matrix multiplication: $\mathrm{s} = \mathrm{xW} + \mathrm{b}$
  - W matrix of size $m \times K$ - columns are $\mathrm{w_k}$
  - b: bias vector - size $1 \times K$
- Element-wise non-linear activation: $\hat{\mathrm{y}} = f(\mathrm{s})$



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
    - $\hat{y_k} \sim P(k|\mathrm{x}, \mathrm{W})$

$\Rightarrow$ **Logistic Regression (LR) Model!**

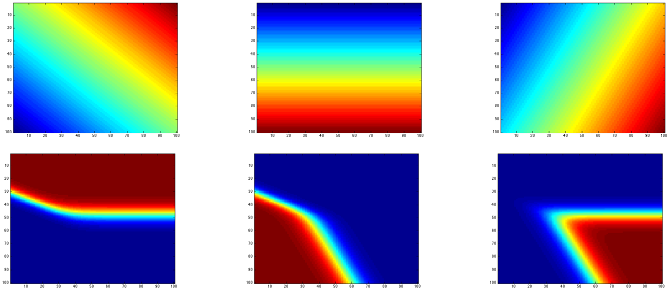
2d Toy Example for Multi-Class Classification

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

$\mathbf{w_1} = [\mathbf{1}; \mathbf{1}]$, $b_1 = -2$ | $\mathbf{w_2} = [\mathbf{0}; -\mathbf{1}]$, $b_2 = 1$ | $\mathbf{w_3} = [\mathbf{1}; -\mathbf{0.5}]$, $b_3 = 10$

Linear mapping for each class:
$s_k = \mathbf{w_k}^\top \mathbf{x} + b_k$

Soft-max output:
$P(k|\mathbf{x}, \mathbf{W})$

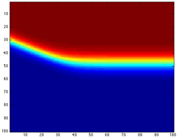


2d Toy Example for Multi-Class Classification

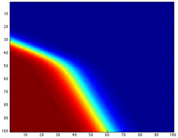
- $\mathrm{x} = \{x_1, x_2\} \in [-5; 5] \times [-5; 5]$, $\hat{y}$: 3 outputs (classes)

Soft-max output:
$P(k|\mathrm{x}, \mathrm{W})$

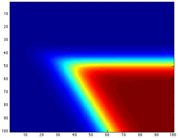
$\mathbf{w_1} = [\mathbf{1}; \mathbf{1}]$, $b_{\mathbf{1}} = \mathbf{-2}$



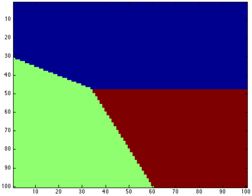
$\mathbf{w_2} = [\mathbf{0}; \mathbf{-1}]$, $b_{\mathbf{2}} = \mathbf{1}$



$\mathbf{w_3} = [\mathbf{1}; \mathbf{-0.5}]$, $b_{\mathbf{3}} = \mathbf{10}$

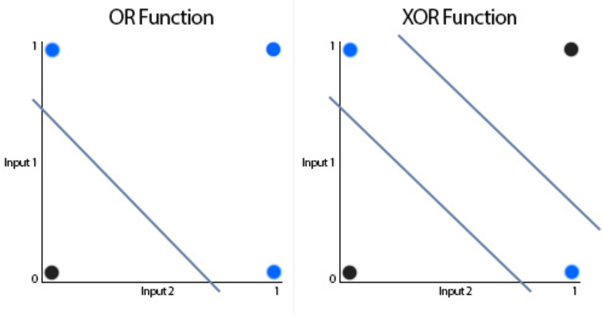


Class Prediction:
$k^* = \arg\max_k P(k|\mathrm{x}, \mathrm{W})$



X-OR Problem

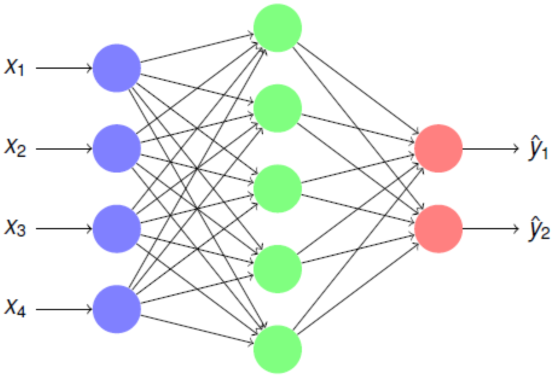
- Logistic Regression (LR): NN with 1 input layer & 1 output layer
- LR: limited to linear decision boundaries
- **X-OR:** NOT 1 and 2 OR NOT 2 AND 1
  - **X-OR: Non linear decision function**



Beyond Linear Classification

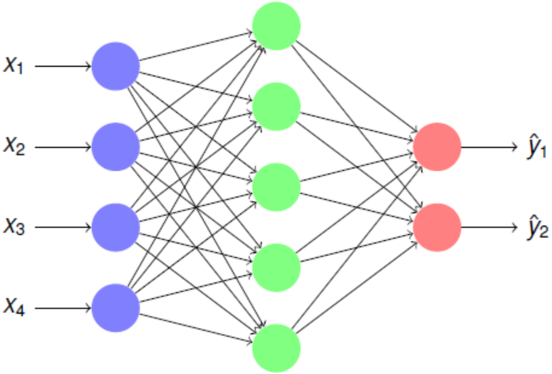
- LR: limited to linear boundaries
- **Solution:** add a layer!

- 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** $\mathbb{R}^L$



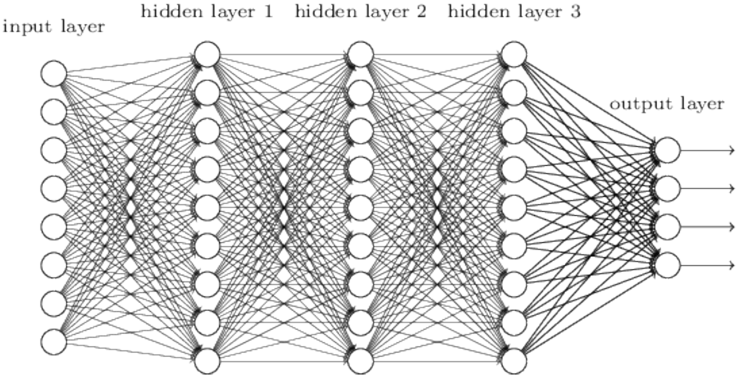
Multi-Layer Perceptron

- **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,
  $s = hW_2 + b_2$
  $\hat{y} = SoftMax(s)$
- **Mapping from x to $\hat{y}$: non-linear boundary!**
  $\Rightarrow$ non-linear activation f crucial!



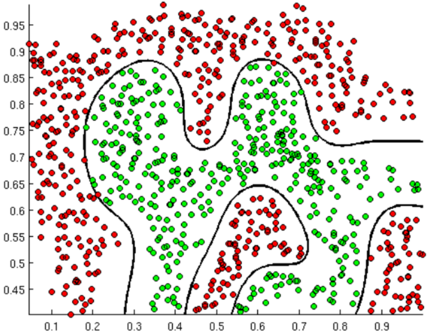
Deep Neural Networks

- 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**

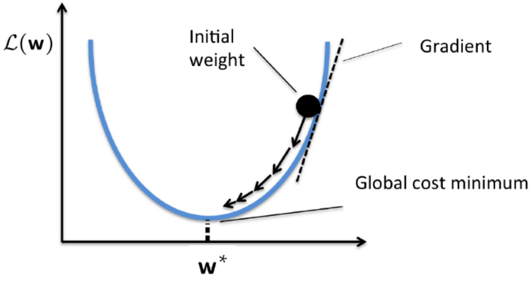
Outline

Training Multi-Layer Perceptron (MLP)

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

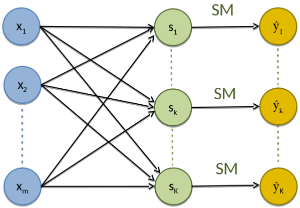
MLP Training

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



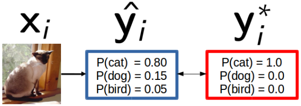
Supervised Learning: Multi-Class Classification

- 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: $\mathcal{L}(W, b) = \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{y}_i, y_i^*)$



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

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

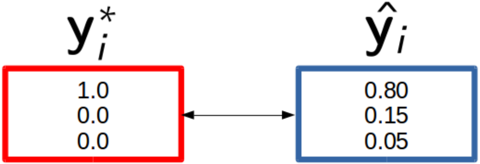


Logistic Regression Training Formulation

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

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

- ⚠ KL asymmetric: $KL(\hat{\mathrm{y}}_\mathrm{i}, \mathrm{y}_\mathrm{i}^*) \neq KL(\mathrm{y}_\mathrm{i}^*, \hat{\mathrm{y}}_\mathrm{i})$ ⚠



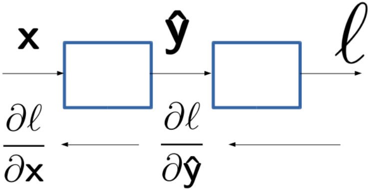
$$KL(\mathbf{y}_\mathbf{i}^*, \hat{\mathbf{y}}_\mathbf{i}) = -log(\hat{y}_{c^*,i}) = -log(0.8) \approx 0.22$$

Logistic Regression Training

- $\mathcal{L}_{CE}(\mathrm{W}, \mathrm{b}) = \frac{1}{N} \sum_{i=1}^{N} \ell_{CE}(\hat{\mathrm{y}}_i, \mathrm{y}_i^*) = -\frac{1}{N} \sum_{i=1}^{N} log(\hat{y}_{c^*,i})$
- ℓ_{CE} smooth convex upper bound of $\ell_{0/1}$
  $\Rightarrow$ **gradient descent optimization**
- Gradient descent: $\mathrm{W}^{(t+1)} = \mathrm{W}^{(t)} - \eta \frac{\partial \mathcal{L}_{CE}}{\partial \mathrm{W}}$ $\quad (\mathrm{b}^{(t+1)} = \mathrm{b}^{(t)} - \eta \frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{b}})$
- **MAIN CHALLENGE: computing** $\frac{\partial \mathcal{L}_{CE}}{\partial \mathrm{W}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \ell_{CE}}{\partial \mathrm{W}}$?

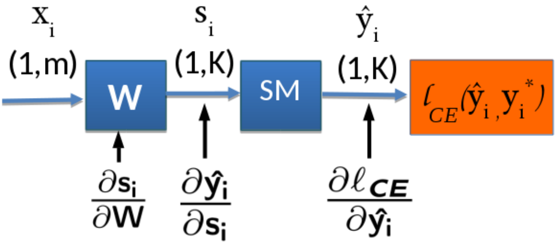
  $\Rightarrow$ **Key Property:** chain rule $\frac{\partial \mathrm{x}}{\partial z} = \frac{\partial \mathrm{x}}{\partial \mathrm{y}} \frac{\partial \mathrm{y}}{\partial z}$
  $\Rightarrow$ **Backpropagation of gradient error!**

Chain Rule



$$\frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x}$$

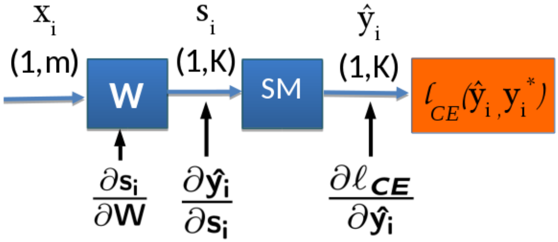
- Logistic regression: $\frac{\partial \ell_{CE}}{\partial W} = \frac{\partial \ell_{CE}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_i} \frac{\partial s_i}{\partial W}$



Logistic Regression Training: Backpropagation

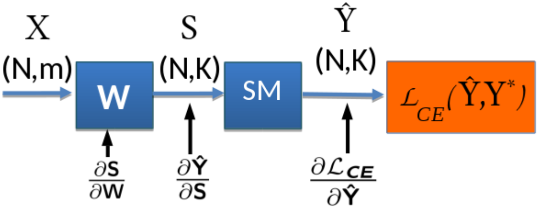
$$\frac{\partial \ell_{CE}}{\partial \mathrm{W}} = \frac{\partial \ell_{CE}}{\partial \hat{\mathrm{y}}_\mathrm{i}} \frac{\partial \hat{\mathrm{y}}_\mathrm{i}}{\partial \mathrm{s}_\mathrm{i}} \frac{\partial \mathrm{s}_\mathrm{i}}{\partial \mathrm{W}}, \; \ell_{CE}(\hat{\mathrm{y}}_i, \mathrm{y}_i^*) =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

- $\frac{\partial \ell_{CE}}{\partial \hat{\mathrm{y}}_\mathrm{i}} = \frac{-1}{\hat{y}_{c^*,i}} = \frac{-1}{\hat{\mathrm{y}}_\mathrm{i}} \odot \delta_{\mathbf{c},\mathbf{c}^*}$
- $\frac{\partial \ell_{CE}}{\partial \mathrm{s}_\mathrm{i}} = \hat{\mathrm{y}}_\mathrm{i} - \mathrm{y}_\mathrm{i}^* = \delta_\mathrm{i}^\mathrm{y}$
- $\frac{\partial \ell_{CE}}{\partial \mathrm{W}} = \mathrm{x}_\mathrm{i}{}^T \delta_\mathrm{i}^\mathrm{y}$



Logistic Regression Training: Backpropagation

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

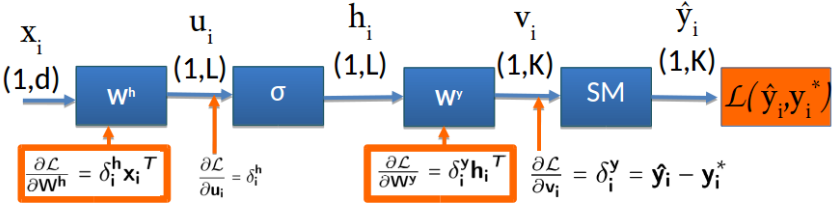


- $\frac{\partial \mathcal{L}_{CE}}{\partial \mathbf{s}} = \hat{Y} - Y^* = \Delta^y$
- $\frac{\partial \mathcal{L}_{CE}}{\partial W} = X^T \Delta^y$

Perceptron Training: Backpropagation

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

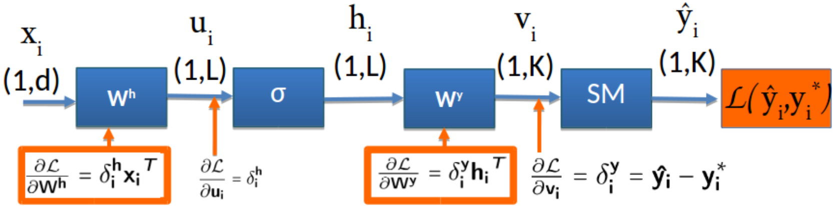
$\Rightarrow$ computing $\boxed{\frac{\partial \mathcal{L}_{CE}}{\partial W^y} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \ell_{CE}}{\partial W^y}}$ and $\boxed{\frac{\partial \mathcal{L}_{CE}}{\partial W^h} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \ell_{CE}}{\partial W^h}}$



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

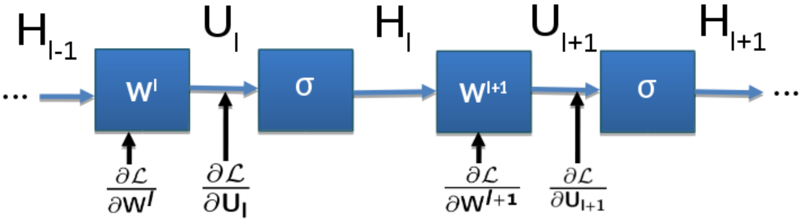
Perceptron Training: Backpropagation

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

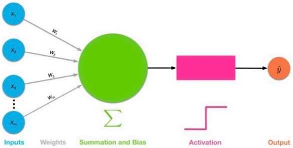


Deep Neural Network Training: Backpropagation

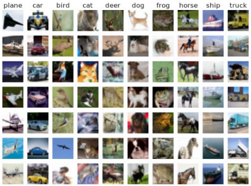
- Multi-Layer Perceptron (MLP): adding more hidden layers
- Backpropagation update ~ Perceptron: **assuming** $\frac{\partial \mathcal{L}}{\partial \mathrm{U_{l+1}}} = \Delta^{l+1}$ **known**
  - $\frac{\partial \mathcal{L}}{\partial \mathrm{W}^{l+1}} = \mathrm{H_l}^T \Delta^{l+1}$
  - Computing $\frac{\partial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



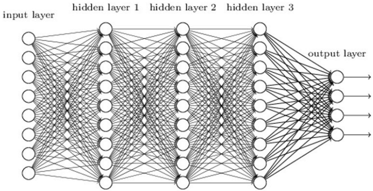
Recap MLP



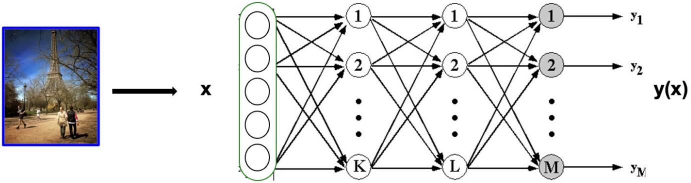
Inputs
Weights
Summation and Bias
Activation
Output



plane
car
bird
cat
deer
dog
frog
horse
ship
truck



input layer
hidden layer 1
hidden layer 2
hidden layer 3
output layer



x
1
2
K
1
2
L
1
2
M
y_1
y_2
y_M
y(x)

Outline

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:**
     $L(Y, f(X)) = (Y - f(X))^2$
   - **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)

How to find the best function f? (continued)

2. 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:

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

   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:

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

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 $R(f_w)$

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 $\mathcal{A}_N$:

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

ERM: Empirical Risk Minimization

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

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

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:

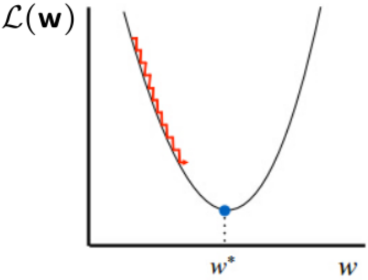
$$
\begin{aligned}
0 &\leq \mathcal{L}(w^{(t+1)}) \\
&= \mathcal{L}\left(w^{(t)} - \eta \nabla \mathcal{L}(w^{(t)})\right) \\
&\approx \mathcal{L}\left(w^{(t)}\right) - \eta \nabla \mathcal{L}(w^{(t)})^T \cdot \nabla \mathcal{L}(w^{(t)}) \text{(first order approximation)} \\
&= \mathcal{L}\left(w^{(t)}\right) - \eta \|\nabla \mathcal{L}(w^{(t)})\|^2 \\
&\leq \mathcal{L}\left(w^{(t)}\right)
\end{aligned}
$$

$\Rightarrow$ local Convergence

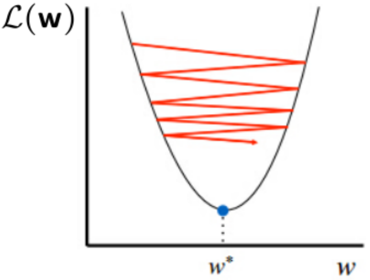
Gradient Descent

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

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



Too small: converge very slowly

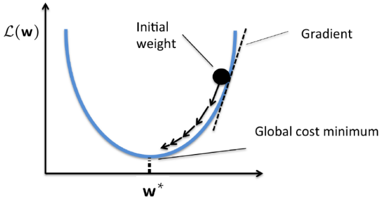


Too big: overshoot and even diverge

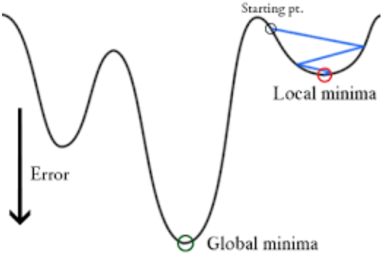
Gradient Descent

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

- **Global minimum ?**
  $\Rightarrow$ **convex** a) *vs* **non convex** b) loss $\mathcal{L}(w)$



a) Convex function



a) Non convex function

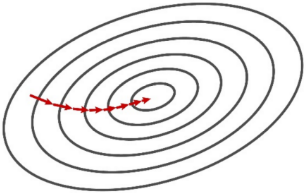
Neural Network Training: Optimization Issues

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

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

- Gradient descent optimization:

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



- Gradient $\nabla_{\mathrm{w}}^{(t)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \ell_{CE}(\hat{\mathrm{y}}_i, \mathrm{y}_i^*)}{\partial \mathrm{w}}\left(\mathrm{w}^{(t)}\right)$ linearly scales wrt:
  - w dimension
  - Training set size

⇒ Too slow even for moderate dimensionality & dataset size!

Stochastic Gradient Descent

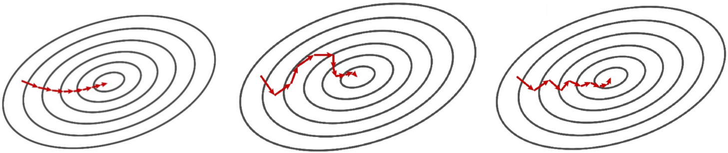
- **Solution:** approximate $\nabla_{\mathrm{w}}^{(t)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \ell_{CE}(\hat{\mathrm{y}}_i, \mathrm{y}_i^*)}{\partial \mathrm{w}}\left(\mathrm{w}^{(t)}\right)$ with subset of examples

  $\Rightarrow$ **Stochastic Gradient Descent (SGD)**
  - Use a single example (online):

$$\nabla_{\mathrm{w}}^{(t)} \approx \frac{\partial \ell_{CE}(\hat{\mathrm{y}}_i, \mathrm{y}_i^*)}{\partial \mathrm{w}}\left(\mathrm{w}^{(t)}\right)$$

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

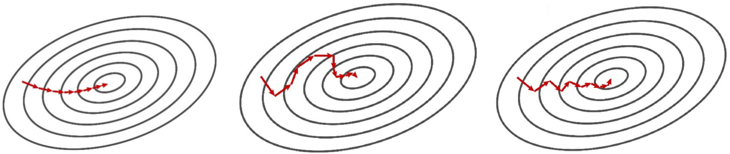
$$\nabla_{\mathrm{w}}^{(t)} \approx \frac{1}{B}\sum_{i=1}^{B} \frac{\partial \ell_{CE}(\hat{\mathrm{y}}_i, \mathrm{y}_i^*)}{\partial \mathrm{w}}\left(\mathrm{w}^{(t)}\right)$$



Full gradient SGD (online) SGD (mini-batch)

Stochastic Gradient Descent

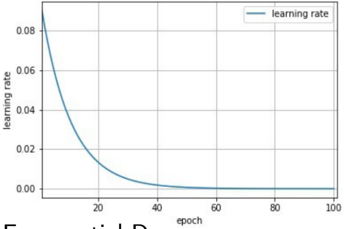
- **SGD: approximation of the true Gradient ∇_w !**
  - Noisy gradient can lead to bad direction, increase loss
  - **BUT:** much more parameter updates: online $\times N$, mini-batch $\times \frac{N}{B}$
  - **Faster convergence**, at the core of Deep Learning for large scale datasets



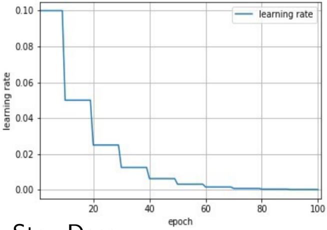
Full gradient SGD (online) SGD (mini-batch)

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 $\eta_t = \eta_0 \cdot r^{\frac{t}{t_u}}$ ...



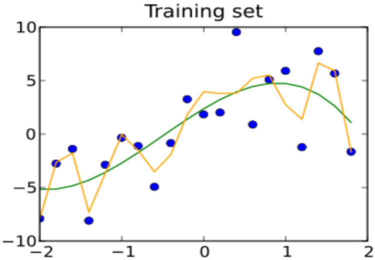
Exponential Decay ($\eta_0 = 0.1$, $\lambda = 0.1s$)



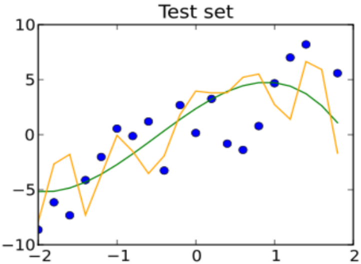
Step Decay ($\eta_0 = 0.1$, $r = 0.5$, $t_u = 10$)

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**



$\mathcal{L}_{train} = 4$, $\mathcal{L}_{train} = 9$



$\mathcal{L}_{test} = 15$, $\mathcal{L}_{test} = 13$

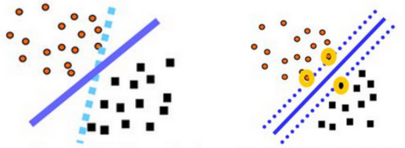
$\Rightarrow$ **Optimization $\neq$ Machine Learning!**
$\Rightarrow$ **Generalization / Overfitting!**

Regularization

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

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

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



L^2 **regularization**: interpretation

- "Smooth" interpretation of L^2 **regularization**, Cauchy-Schwarz:
$$(\langle \mathrm{w}, (\mathrm{x} - \mathrm{x}') \rangle)^2 \leq ||\mathrm{w}||^2 ||\mathrm{x} - \mathrm{x}'||^2$$
- Controlling L^2 norm $||\mathrm{w}||^2$: "small" variation between inputs x and x′
$\Rightarrow$ small variation in neuron prediction $\langle \mathrm{w}, \mathrm{x} \rangle$ and $\langle \mathrm{w}, \mathrm{x}' \rangle$

$\Rightarrow$ **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