Neural networks

Alexandre Boulch

www.boulch.eu

IOGS - ATSI

Outline

- Introduction
- The artificial neuron
- Stochastic gradient descent
- Multi-label classification

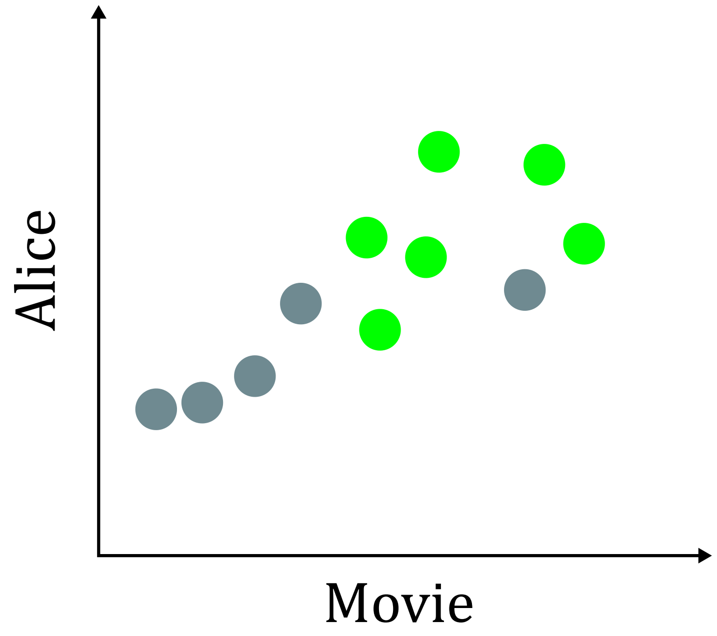
Motivation

Simple problem:

Predict if Bob will like a movie given Alice's grade

Hypothesis:

Linear problem



Motivation

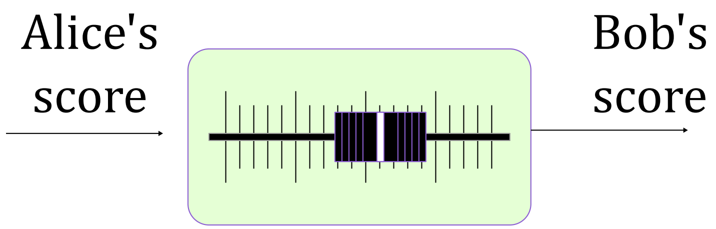
Simple problem:

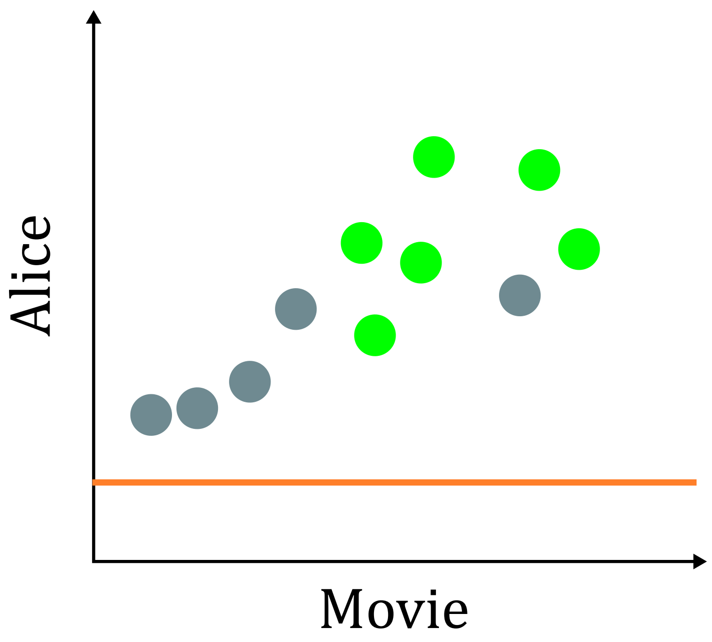
Predict if Bob will like a movie given Alice's grade

Hypothesis:

A simple threshold:

$$y = sign(x + b)$$





Motivation

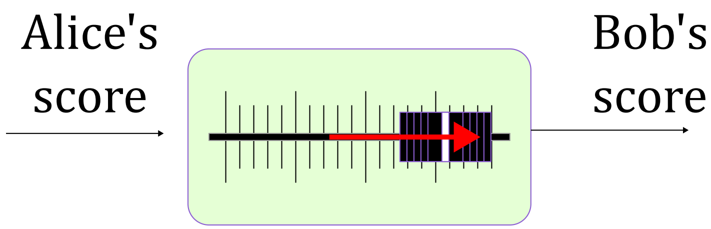
Simple problem:

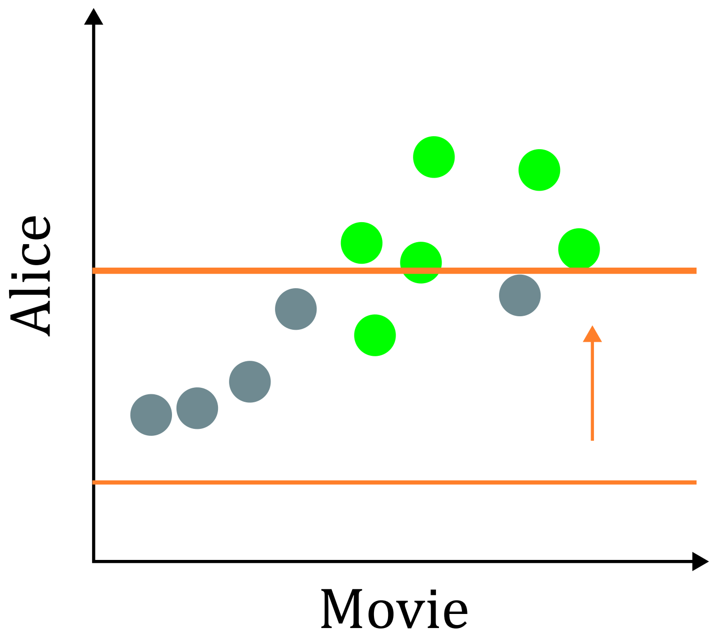
Predict if Bob will like a movie given Alice's grade

Hypothesis:

A simple threshold

$$y = sign(x + b)$$





Motivation

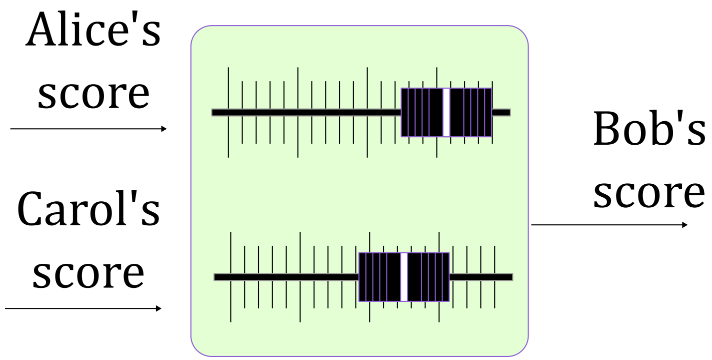
Simple problem:

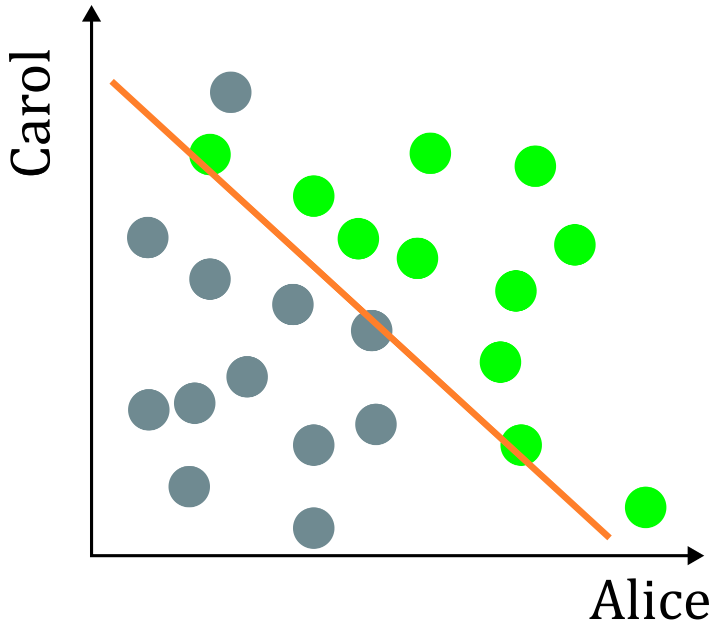
Predict if Bob will like a movie given Alice's and Carol's grades

Hypothesis:

An affine function

$$y = sign(w_a x_a + w_c x_c + b)$$





Motivation

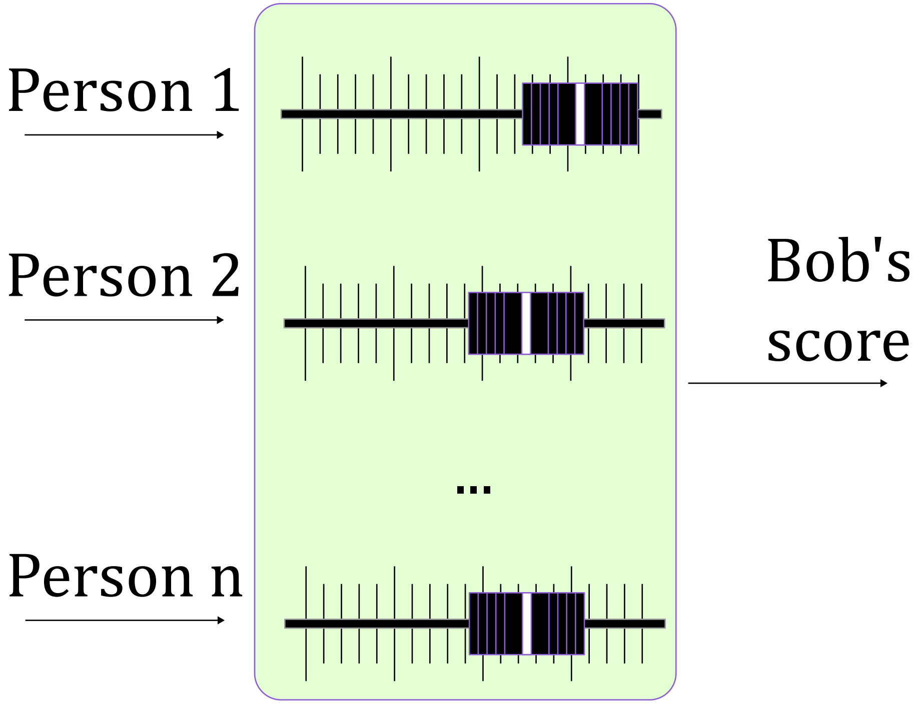
More complicated problem:

Predict if Bob will like a movie given a large user database

Hypothesis:

An affine function

$$y = sign(w_1 x_1 + w_2 x_2 + ... + b)$$



Motivation

More complicated problem:

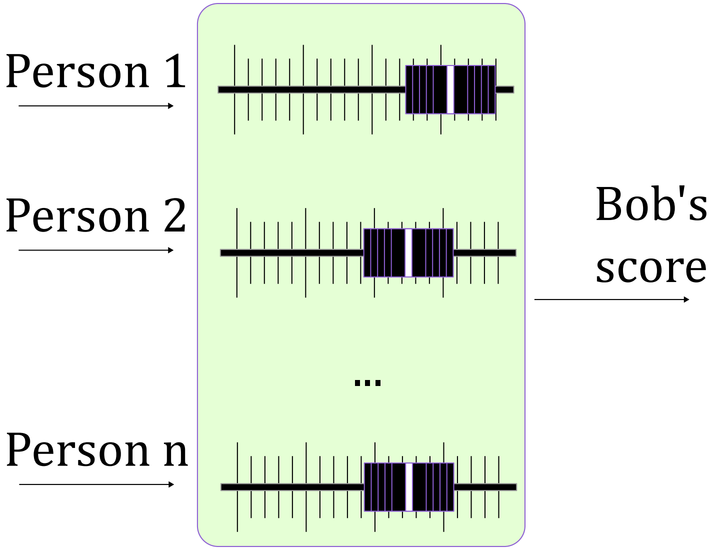
Predict if Bob will like a movie given a large user database

Hypothesis:

Non-affine function ?

$$y = \phi(\{x_i\}, \Theta)$$

x_i inputs, Θ parameters, ϕ function



Motivation

Ideally

- General machine learning architectures / bricks
  - use the same approach for various problems
- Learn the parameters
  - Use data to automatically extractknowledge

Neural networks

- Currently one of the most efficient approach for machine learning

Artificial neuron

Historical Background

- 1958: Rosenbalt, perceptron
- 1965: Ivakhenko and Lapa, neural networks with several layers
- 1975-1990: Backpropagation, Convolutional Neural Networks
- 2007+: Deep Learning era (see Deep Learning sesion)
  - Large convolutional neural networks
  - Transformers
  - Generative models
  - "Foundation models"
  - ...

Bio inspired model

- The brain is made of neurons.
- Receive, process and transmit action potential.
- Multiple recievers (dentrites), single transmitter (axon)

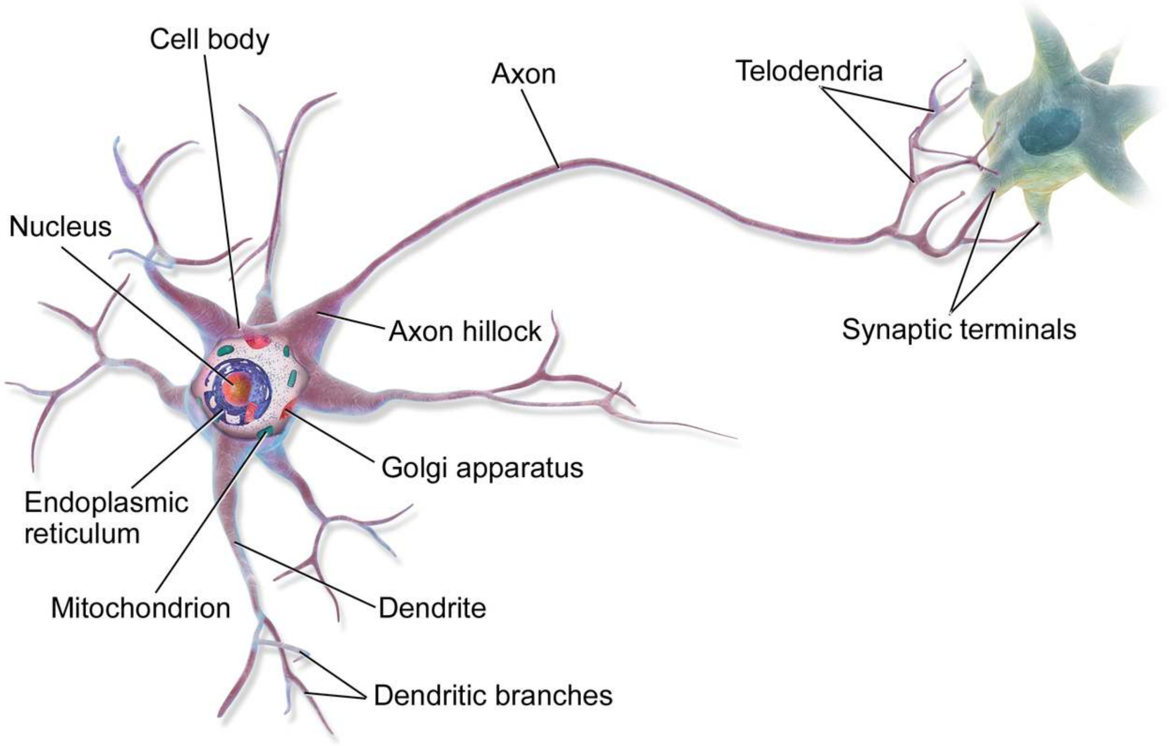


image wikipedia

Formulation

A simple model of the neuron: activation level is the weighted sum of the inputs

$$y = \sigma(\mathbf{w}\mathbf{x} + b)$$

- $\mathbf{x} \in \mathbb{R}^n$ the input vector
- $\mathbf{w} \in 1 \times \mathbb{R}^n$ the weight matrix
- $b \in \mathbb{R}$ the bias
- $y \in \mathbb{R}$ the output
- σ, $\mathbb{R} \rightarrow \mathbb{R}$ the activation function

1958 : Rosenbalt, perceptron

Back to example

Simple problem:

Predict if Bob will like a movie given Alice's and Carol's grades

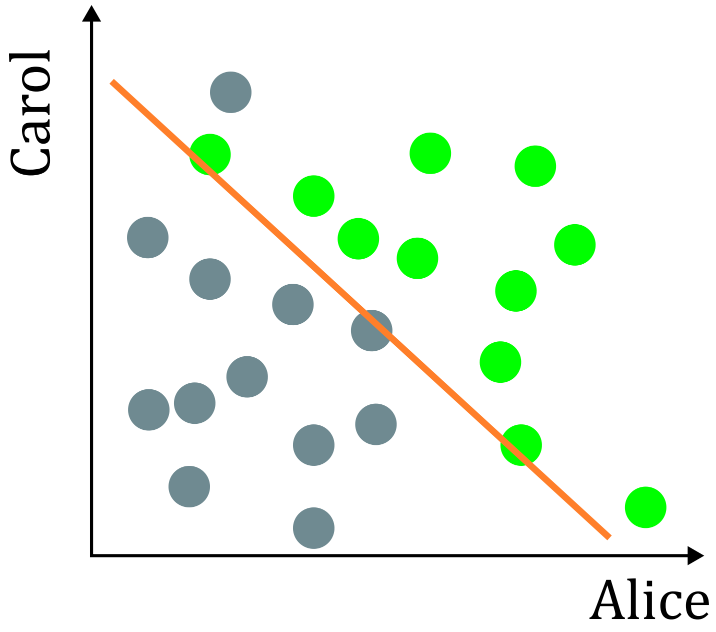
Hypothesis:

An affine function

$$y = sign(w_a x_a + w_c x_c + b)$$

It can be modeled with a single neuron with:

$$\sigma(.) = sign(.)$$



Geometric interpretation of the neuron

$$\mathcal{P} : \mathbf{wx} + b = 0$$

is an hyperplan of $\mathbb{R}^n$, with n the dimension of the space.

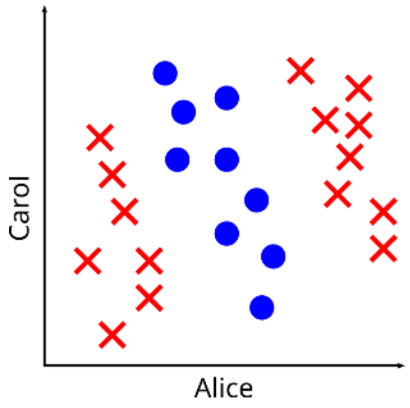
The sign of activation y defines on which side of $\mathcal{P}$ lies $\mathbf{x}$.

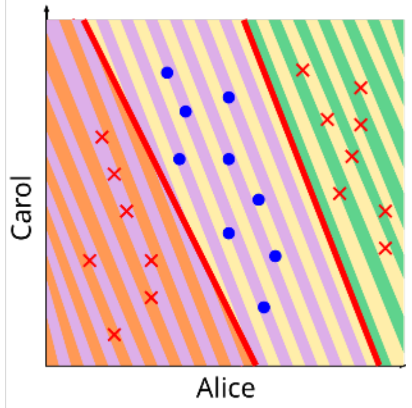
The artificial neuron $\rightarrow$ linear decision.

Previous course on SVM: the SVM was originally formulated using neurons.

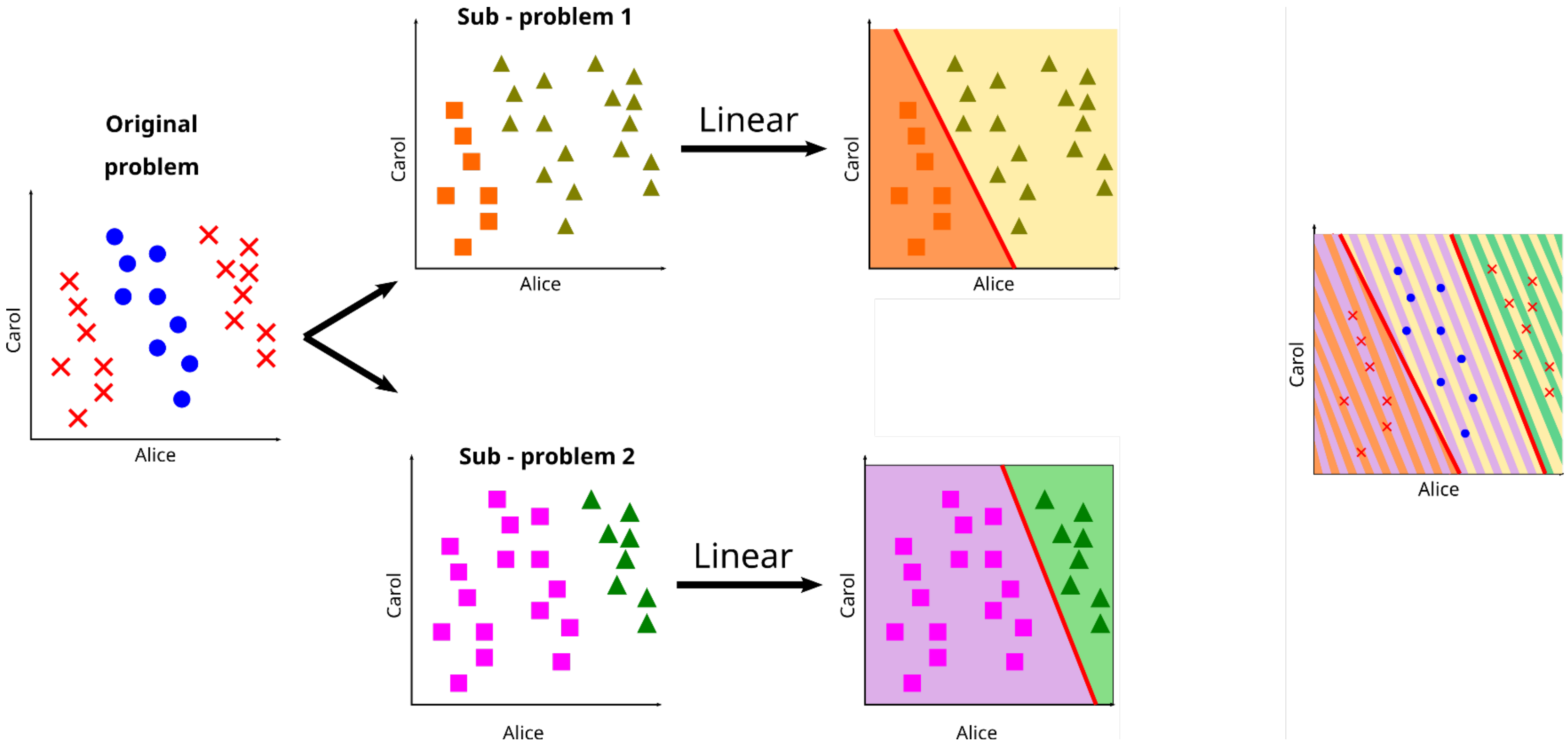
Limitations

Original problem

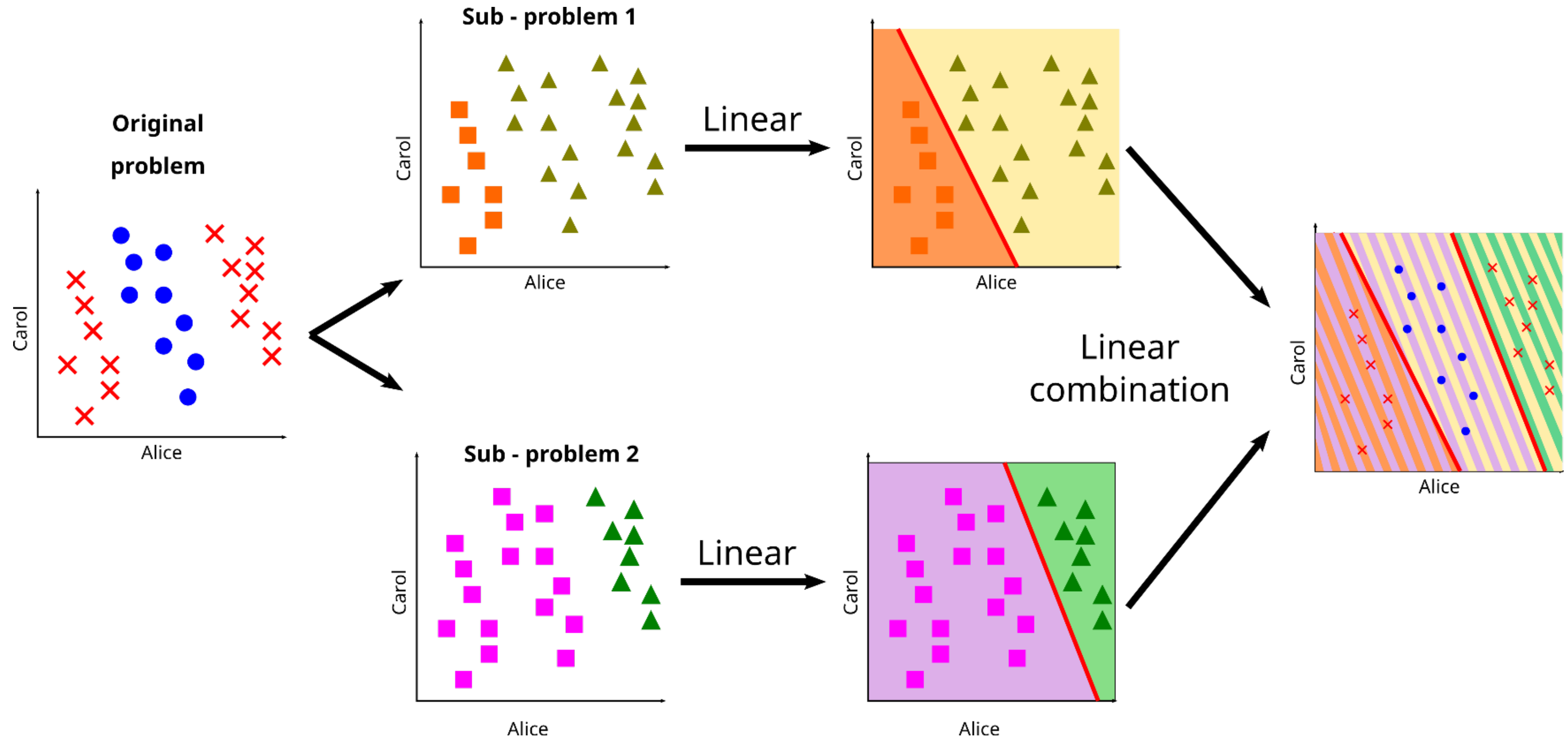




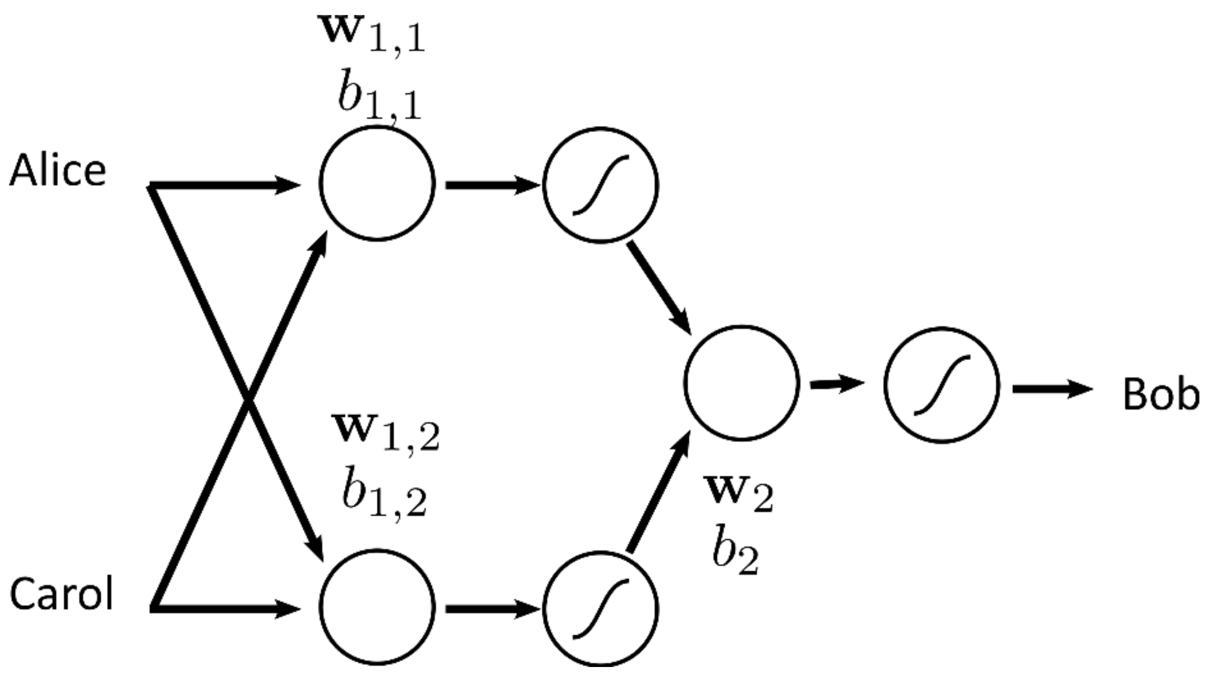
Limitations

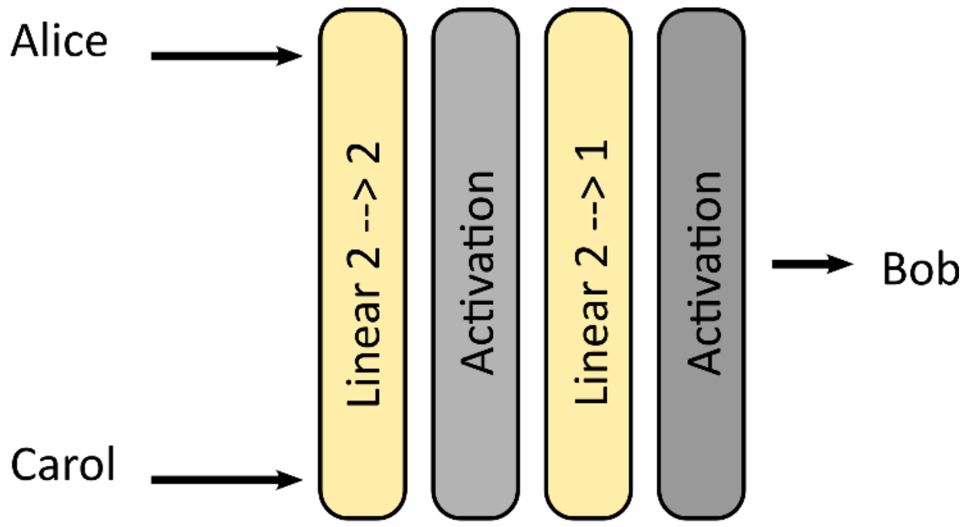


Limitations



Possible solution: using multiple stacked neurons





Universal approximation theorem

Arbitrary width

Multilayer feed-forward networks with as few as one hidden layer are universal approximators

- Cybenko (1989) for sigmoid activation functions
- Hornik et al. (1989) for 1 hidden layer
- Hornik (1991) any choice of the activation function

Arbitrary depth

- Gripenberg (2003)
- Yarotsky (2017), Lu et al (2017)
- Hanin (2018)
- Kidger (2020)

Neural networks in practice

Network design

- high neuron number: very computationally expensive
- prefer stacking more layers

Optimization

- several parameters (probably many)
- automatic optimization
- **gradient descent**

Stochactic gradient descent

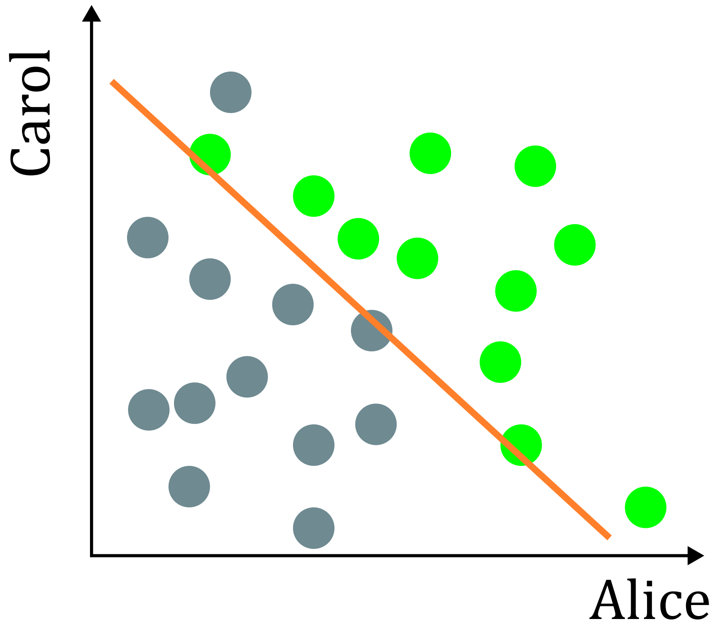
Optimizing the parameters

Objective

Find $\mathbf{w}$ and b such that:

$$\hat{y} = \sigma(\mathbf{w}x + b) \approx y$$

for $x \in M$, the set of movies.



Gradient descent

Objective

Reach the bottom of the valley

Gradient descent

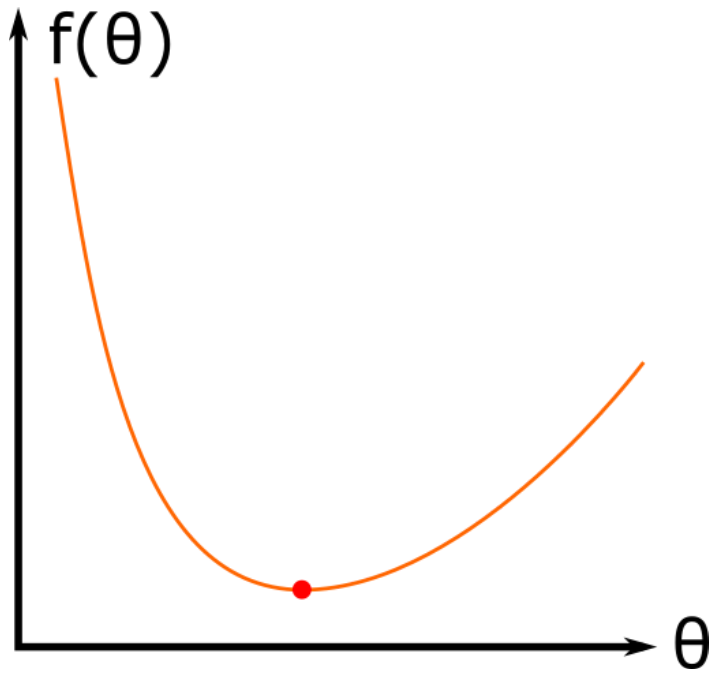
Objective

Reach the bottom of the valley

Gradient descent

Minimizing an objective function $f : \theta \to f(\theta)$

$$\underset{\theta}{\operatorname{argmin}} f(\theta)$$

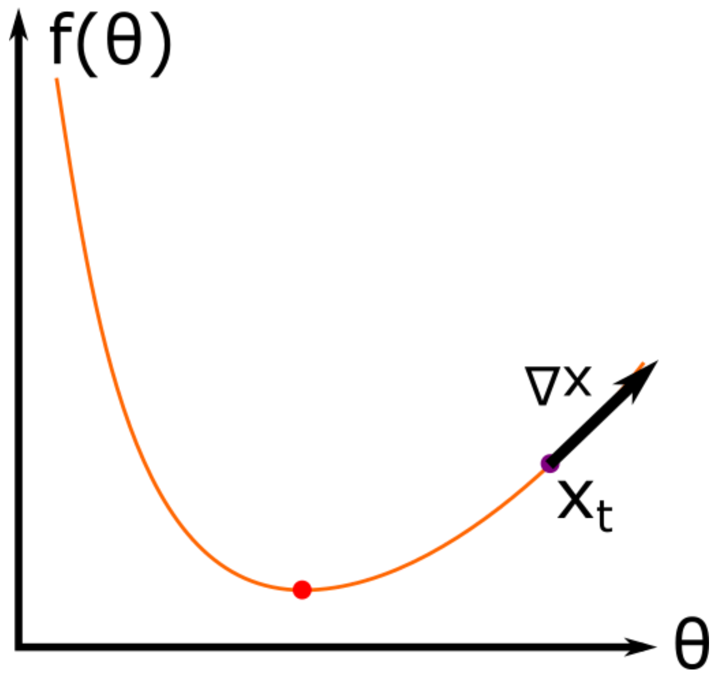


Gradient descent

For every smooth function f

$$\forall x, \nabla f \neq 0 \Rightarrow \exists \epsilon, f(x) > f(x - \epsilon \nabla f_x)$$

i.e., following the opopsite direction of the gradient leands to a local minimum of the function.



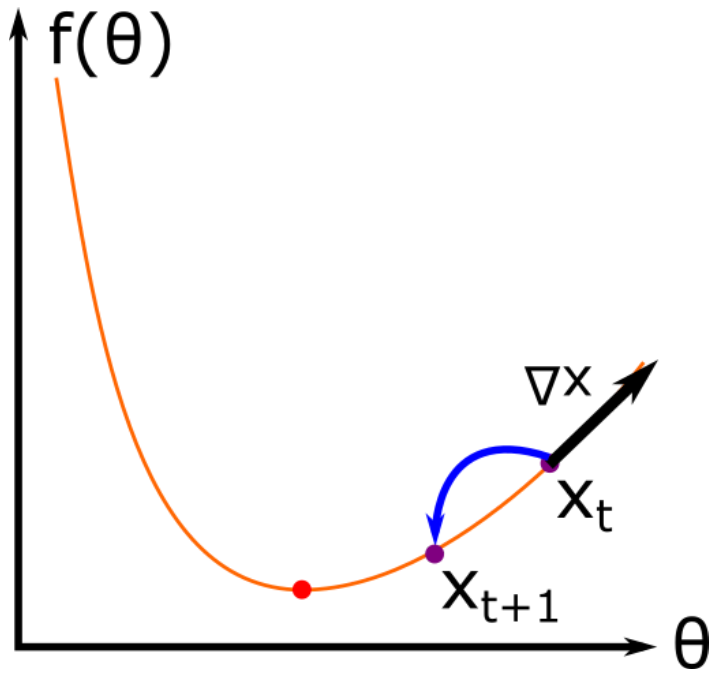
Gradient descent

An iteratieve agorithm:

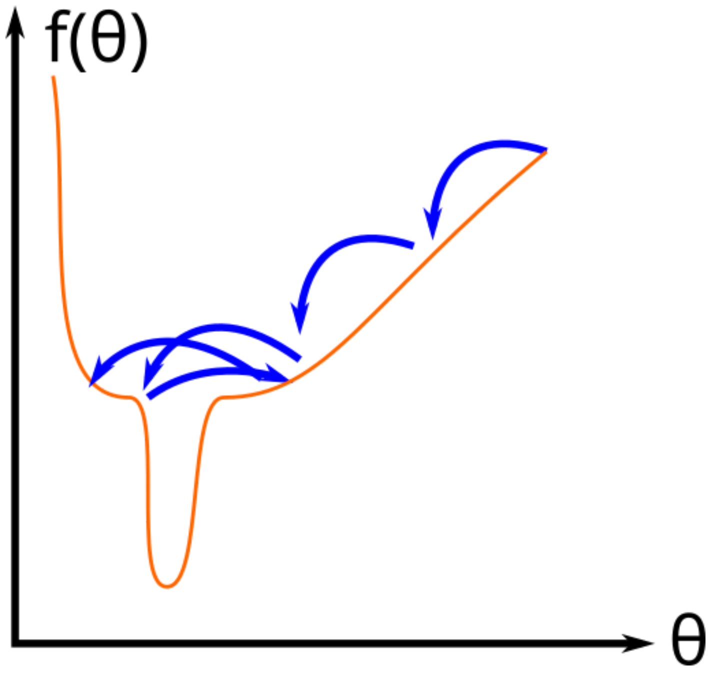
$$\theta_{i+1} \longleftarrow \theta_i - r\nabla\theta_i$$

r is the learning rate.

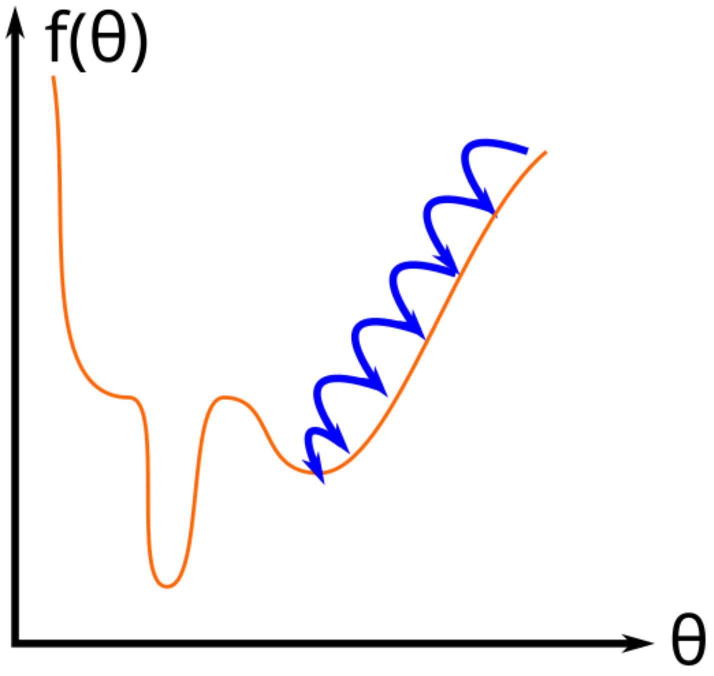
f must be **continuous** and **differentiable** almost everywhere.



Gradient descent



Too high learning rate.



Too low learning rate.

Back to example

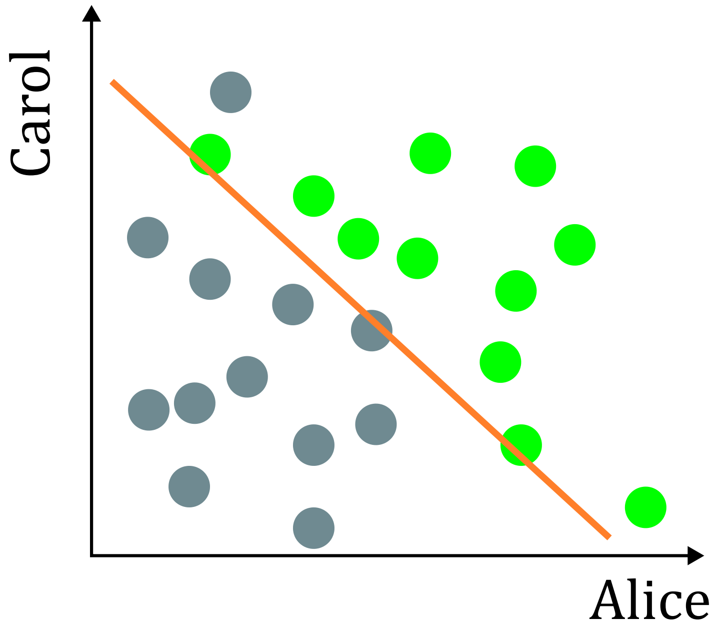
Simple problem:

Predict if Bob will like a movie given Alice's and Carol's grades

Hypothesis:

An affine function

$$y = sign(w_a x_a + w_c x_c + b)$$



Problem of sign function

Need a loss function

Sigmoid function

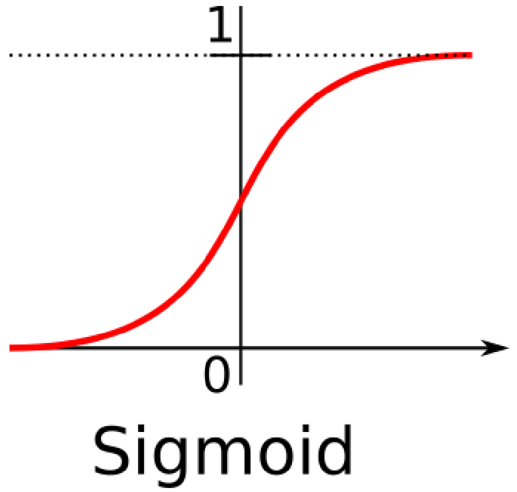
Problem of sign function

- Zero-gradient everywhere
- Not differentiable at 0

Solution

The sigmoid function is an approximation of the sign function

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



Objective / loss function

We need a function to compare predictions and ground truth.

A function such that:

- it takes 0 values if $\hat{y} = y$
- it is differentiable
- it increases along with the "difference" between the $\hat{y}$ and y

Possibilities:

- Squared differences: $||\hat{y} - y||_2^2$
- Cross entropy (for categorical loss)
- ...

Optimization

Forward

Iteratively compute the output of the network

Backward

Itertatively compute the derivatives starting from the output

Weight update

Update weights according to learning rate.

$$\theta_{i+1} \leftarrow \theta_i - r\nabla\theta_i$$

Mean Squared Differences

Loss function

$$\mathcal{L} = \frac{1}{N}\sum_{1}^{N}||\hat{y}_n - y_n||_2^2$$

Single neuron model

$$\mathcal{L} = \frac{1}{N}\sum_{1}^{N}||\sigma(\mathbf{w}x_n + b) - y_n||_2^2$$

Derivatives:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i} = \frac{2}{N}\sum_{1}^{N} x_i \sigma(\mathbf{w}x_n + b)(1 - \sigma(\mathbf{w}x_n + b))(\sigma(\mathbf{w}x_n + b) - y_n)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{N}\sum_{1}^{N} \sigma(\mathbf{w}x_n + b)(1 - \sigma(\mathbf{w}x_n + b))(\sigma(\mathbf{w}x_n + b) - y_n)$$

Mean Squared Differences

Loss function

$$\mathcal{L} = \frac{1}{N} \sum_{1}^{N} ||\hat{y}_n - y_n||_2^2$$

Single neuron model

$$\mathcal{L} = \frac{1}{N} \sum_{1}^{N} ||\sigma(\mathbf{w}x_n + b) - y_n||_2^2$$

Derivatives $\hat{y}_n$ **is computed at prediction time**:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i} = \frac{2}{N} \sum_{1}^{N} x_i \hat{y}_n (1 - \hat{y}_n)(\hat{y}_n - y_n)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{N} \sum_{1}^{N} \hat{y}_n (1 - \hat{y}_n)(\hat{y}_n - y_n)$$

Mean Squared Differences

Loss function

$$\mathcal{L} = \frac{1}{N}\sum_{1}^{N} ||\hat{y}_n - y_n||_2^2$$

Two neurons model

$$\mathcal{L} = \frac{1}{N}\sum_{1}^{N} ||\sigma(\mathbf{w_2}(\sigma(\mathbf{w_1}x_n + b_1)) + b_2) - y_n||_2^2$$

Derivatives:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_{1,i}} = \dots \quad \frac{\partial \mathcal{L}}{\partial b_1} = \dots \quad \frac{\partial \mathcal{L}}{\partial \mathbf{w}_{2,i}} = \dots \quad \frac{\partial \mathcal{L}}{\partial b_2} = \dots$$

We do not want to explicitely compute the loss function.

Chain rule

In practice, we make an intensive use of the chain rule:

$$\frac{\partial g \circ f(a,b)}{\partial a} = \frac{\partial f(a,b)}{\partial a} g'(f(a,b))$$

or for three functions:

$$\frac{\partial h \circ g \circ f(a,b)}{\partial a} = \frac{\partial g \circ f(a,b)}{\partial a} h'(g(f(a,b)))$$

$$\frac{\partial h \circ g \circ f(a,b)}{\partial a} = \frac{\partial f(a,b)}{\partial a} g'(f(a,b)) h'(g(f(a,b)))$$

Chain rule applied to neural networks

Forward

$$\boldsymbol{y}_1 = f_1(\boldsymbol{x}_n, \theta_1) \qquad \boldsymbol{y}_2 = f_2(\boldsymbol{y}_1) \qquad y_3 = f_3(\boldsymbol{y}_2, \theta_2) \qquad \hat{y} = L(y_3, y_n)$$

valeo.ai

Chain rule applied to neural networks

Forward

$$\boldsymbol{y}_1 = f_1(\boldsymbol{x}_n, \theta_1) \qquad \boldsymbol{y}_2 = f_2(\boldsymbol{y}_1) \qquad y_3 = f_3(\boldsymbol{y}_2, \theta_2) \qquad \hat{y} = L(y_3, y_n)$$

Backward

$$\frac{\partial L}{\partial y_3} = \dots$$

$$\frac{\partial L}{\partial y_2} = \boxed{\frac{\partial f_3}{\partial y_2}} \boxed{\frac{\partial L}{\partial y_3}}$$

$$\frac{\partial L}{\partial \theta_2} = \boxed{\frac{\partial f_3}{\partial \theta_2}} \boxed{\frac{\partial L}{\partial y_3}}$$

$$\frac{\partial L}{\partial y_1} = \boxed{\frac{\partial f_2}{\partial y_1}} \boxed{\frac{\partial L}{\partial y_2}}$$

$$\frac{\partial L}{\partial \theta_1} = \boxed{\frac{\partial f_1}{\partial \theta_1}} \boxed{\frac{\partial L}{\partial y_1}}$$

Chain rule applied to neural networks

Forward

$$\boldsymbol{y}_1 = f_1(\boldsymbol{x}_n, \theta_1) \qquad \boldsymbol{y}_2 = f_2(\boldsymbol{y}_1) \qquad y_3 = f_3(\boldsymbol{y}_2, \theta_2) \qquad \hat{y} = L(y_3, y_n)$$

Backward

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial f_1}{\partial \theta_1} \frac{\partial L}{\partial y_1} \qquad \frac{\partial L}{\partial y_1} = \frac{\partial f_2}{\partial y_1} \frac{\partial L}{\partial y_2} \qquad \frac{\partial L}{\partial y_2} = \frac{\partial f_3}{\partial y_2} \frac{\partial L}{\partial y_3} \qquad \frac{\partial L}{\partial y_3} = \dots$$

$$\frac{\partial L}{\partial \theta_2} = \frac{\partial f_3}{\partial \theta_2} \frac{\partial L}{\partial y_3}$$

$$\theta_1 \leftarrow \theta_1 - r \frac{\partial L}{\partial \theta_1}$$

Weight update

$$\theta_2 \leftarrow \theta_2 - r \frac{\partial L}{\partial \theta_2}$$

Code architecture

```
class Module:
  def __init__(self, …):
    self.weights = ...

  def forward(self, x): y = function(f)                  # compute output
    self.ctx =                                            # save the stuff for backward
                                                          #     (save computation time)
    return y

  def backward(self, grad_output):
    self.grad_weights = … * grad_output                   # compute gradient w.r.t. parameters
    grad_input = … * grad_output                          # compute gradient w.r.t. input
    return grad_input                                     # return gradient w.r.t. input for use
                                                          #      in previous layer

  def update_weights(self, lr):
    self.weights = self.weights - lr * self.grad_weights  # apply gradient descent
```

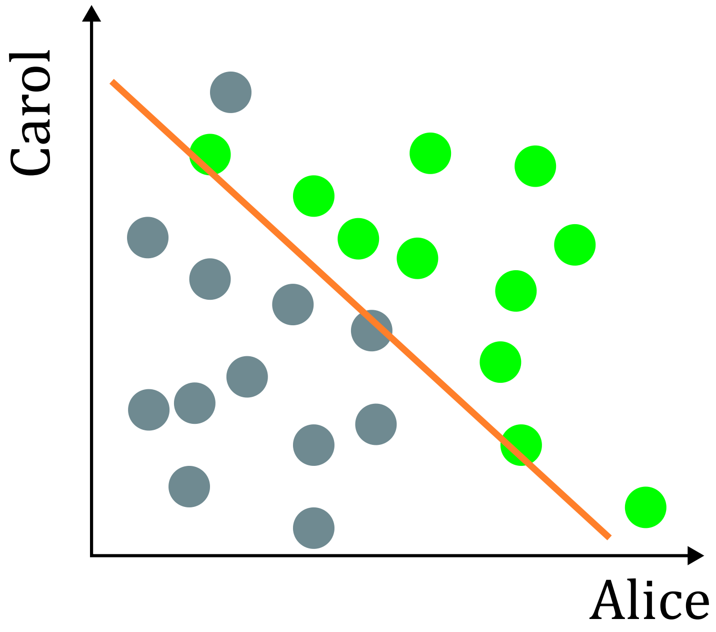
Optimizing the parameters

Objective

Find $\mathbf{w}$ and b such that:

$$\hat{y} = \sigma(\mathbf{w}x + b) \approx y$$

for all $x \in \mathcal{M}$, the set of movies.



In practice

No access to the whole set of movies, only a training subset:

$$(x_n, y_n) \in \mathbb{R} \times \{0,1\} \in X_{train}$$

Limits of gradient descent

Objectives

$$\underset{\Theta}{\operatorname{argmin}} \quad \mathcal{L}(X_{train}, \Theta)$$

with $\mathcal{L} = \frac{1}{N}\sum_N \mathcal{L}_n$ and $\nabla\mathcal{L}\frac{1}{N}\sum_N \nabla\mathcal{L}_n$

Minimizing over X_{train}:

- requires computing $\mathcal{L}_n$ for all elements of X_{train}
- is time consuming for one iteration
- can be untracktable for large X_{train}

Stochastic gradient Descent (SGD)

Idea

Approximate the training set by picking only one sample at each iteration

$$\mathcal{L} = \mathcal{L}_n \qquad \nabla\mathcal{L} = \nabla\mathcal{L}_n$$

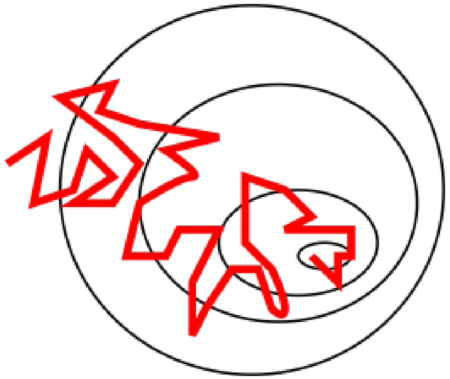
Is it the same as gradient descent?

$$\begin{aligned}
\mathbb{E}_{n\sim U}[\frac{\partial}{\partial w}\mathcal{L}_n(w)] &= \frac{\partial}{\partial w}\mathbb{E}_{n\sim U}[\mathcal{L}_n(w)] \quad \textbf{(Fubini)} \\
&= \frac{\partial}{\partial w}\sum_{i=1}^{N}\mathbb{P}(n=i)\mathcal{L}_i(w) \\
&= \frac{\partial}{\partial w}\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(w) = \frac{\partial}{\partial w}\mathcal{L}(w)
\end{aligned}$$

Stochastic gradient Descent (SGD)

Problem

Very slow convergence.

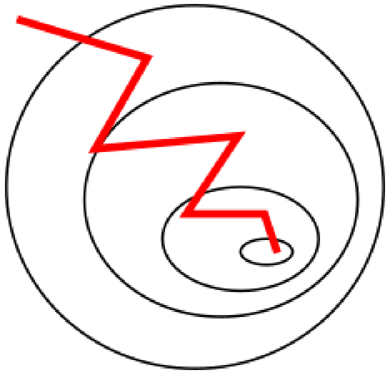


Solution

Average gradient over **batches**.

A batch = random subset of training set

(All neural network librairies handle batches)



Vectorization trick

Numpy style

```
batch # size (B, in_size)
w # size (out_size, in_size)
B # size (out_size)

output = []
for i in range(batch.shape[0]):
  temp = w @ batch[i] + b
  output.append(temp)
output = np.stack(axis=0)

output # size (B, out_size)
```

With batch operations

```
batch # size (B, in_size)
w # size (in_size, out_size)
B # size (out_size)


output = batch @ w + B


output # size (B, out_size)
```

Multi-label classification

Information

Information estimate the number of bits required to encode/transmit an event:

- Always the same: less information
- Very various: more information

Information $h(j)$ for an event j, given $P(j)$, the probability of j:

$$h(j) = -log(P(j))$$

Entropy

Entropy is the number of bits to encode/transmit a **random** event:

- A skewed (biased) distribution, e.g., always same value: low entropy
- A uniform distribution: high entropy

Entropy $H(j)$, for a random variable with a set of j in C discrete states discrete states and their probability $P(j)$:

$$H(P) = -\sum_{j \in C} P(j) log(P(j))$$

Cross-Entropy

Cross entropy estimate the number of bits to transmit from one distribution Q to a second distribution P. P is the target, Q is the source.

$$H(P, Q) = -\sum_{j \in C} P(j) log(Q(j))$$

H estimates the additional number of bits to represent an event using P instead of Q.

Cross-entropy loss

For one sample x:

$$\mathcal{L}_{ce}(x) = H(P_x, Q_x) = -\sum_{j \in C} P_x(j) log(Q_x(j))$$

For a dataset:

$$\mathcal{L}_{ce} = H(P, Q) = -\frac{1}{N}\sum_{1}^{N}\sum_{j \in C} P_{x_n}(j) log(Q_{x_n}(j))$$

(averaged for insensibility to dataset size)

Cross-entropy loss - Classification

$$\mathcal{L}_{ce} = H(P,Q) = -\frac{1}{N}\sum_{1}^{N}\sum_{j\in C} P_{x_n}(j) log(Q_{x_n}(j))$$

For classification, let x_n be a sample of class $c_n \in C$.

$$P_{x_n}(i) = \begin{cases} 1, & \text{if } i = c_n. \\ 0, & \text{otherwise.} \end{cases}$$

Then:

$$\mathcal{L}_{ce} = -\frac{1}{N}\sum_{1}^{N} log(Q_{x_n}(c_n))$$

valeo.ai

Cross-entropy loss - Binary classification

$$\mathcal{L}_{ce}(x_n) = -log(Q_{x_n}(c_n)), \quad c_n \in \{c_0, c_1\}$$

- Let $y = \mathbb{P}(c_n = c_1)$.
- Let $\hat{y}_n$ be the **estimated probability** of class c_1 for x_n.
  (e.g., $\sigma(\phi(x_n))$, with σ a sigmoid and $\phi(x_n)$ be the output of the network)

$$\mathcal{L}_{ce}(x_n) = -log(\hat{y}_n), \quad \text{if } c_n = c_1, \text{ i.e., } y_n = 1$$
$$\mathcal{L}_{ce}(x_n) = -log(1 - \hat{y}_n), \quad \text{if } c_n = c_0, \text{ i.e., } y_n = 0$$

- Then the **Binary cross entropy** is:

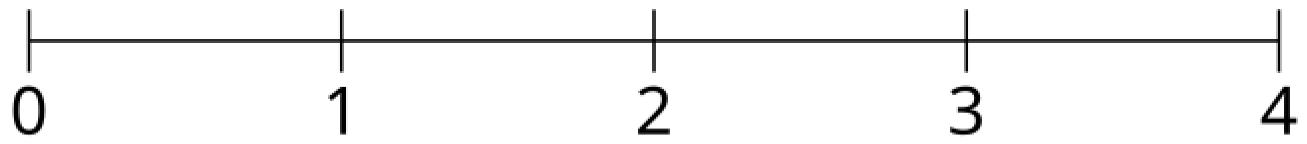
$$\mathcal{L}_{bce}(x_n) = -y_n log(\hat{y_n}) - (1 - y_n) log(1 - \hat{y}_n)$$

Multi-label classification

Can we use a single output for multi-label classification?

Example with 5 classes

$$\phi(x_n) = \hat{y}_n \in [0, 4]$$

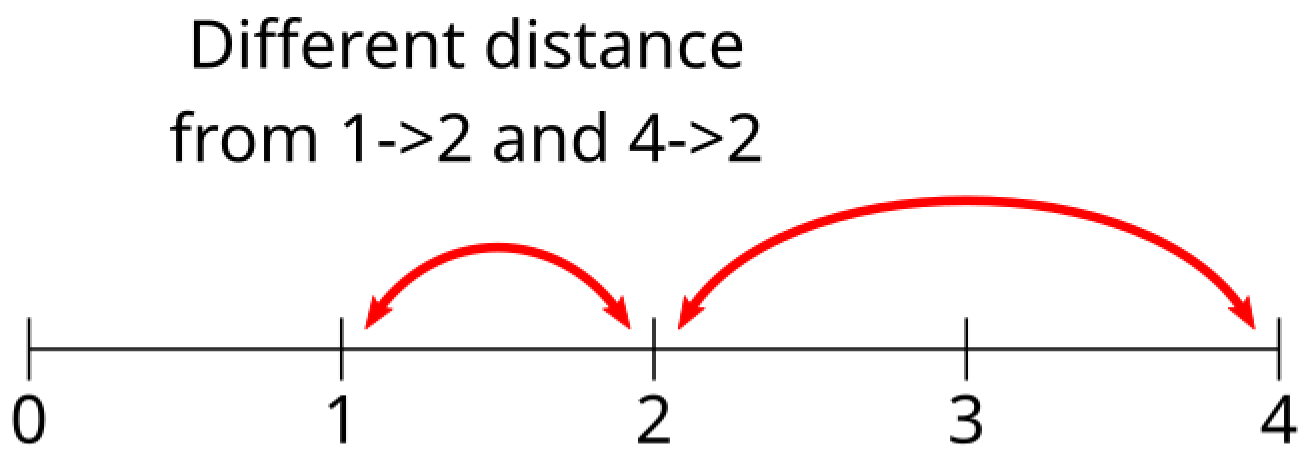


Cross-entropy loss - Multi-label classification

Can we use a single output for multi-label classification?

Example with 5 classes

$$\phi(x_n) = \hat{y}_n \in [0, 4]$$



Multi-label classification

Solution

Predict a vector, one value per class:

$$\hat{y}_n \in \mathbb{R}^C$$

Highest value is the selected class:

$$\hat{c}_n = \operatorname*{argmax}_i \hat{y}_n^i$$

What loss can we use?

Cross-entropy loss - Multi-label classification

argmax is not differentiable

Seeing the output as a distribution probability allows to use **cross-entropy**

Let $p_n^i = s(y_n)^i$ be a normalization layer, then:

$$\mathcal{L}_{ce}(x_n) = -log(s(\hat{y}_n)^{c_n}) = -log(p_n^{c_n})$$

What s can we use?

- euclidean normalization $y/||y||$
- Soft-Max

Cross-entropy loss - Multi-label classification

Soft-Max

$$p^i(x) = \frac{e^{x^i}}{\sum_{j \in C} e^{x^j}}$$

Good properties associated with cross entropy:

$$\mathcal{L}_{ce}(\hat{y}, c) = -\hat{y}^c + log(\sum_{j \in C} e^{\hat{y}^j})$$

And derivative:

$$\frac{\mathcal{L}_{ce}(\hat{y}, c)}{\partial \hat{y}^c} = -1 + s(\hat{y}) \qquad \frac{\mathcal{L}_{ce}(\hat{y}, c)}{\partial \hat{y}^i} = s(\hat{y}), \text{ if } i \neq c$$

Practical session

Practical session

Implement a simple neural network

- Define the number of layers / neurons
- Setup a stochastic gradient descent procedure
- Plot the results
- Explore several losses
- Go multi-labels

Tools

- Google Colab
- Pytorch
- Matplotlib / pyplot for visualization