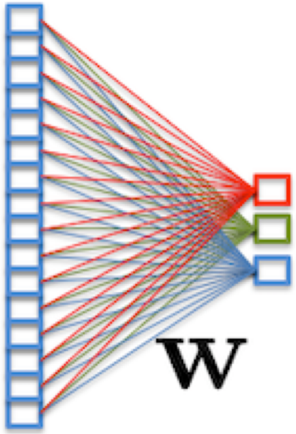
5IF - Deep Learning and Differentiable Programming

2.2 (Generalized) Linear models

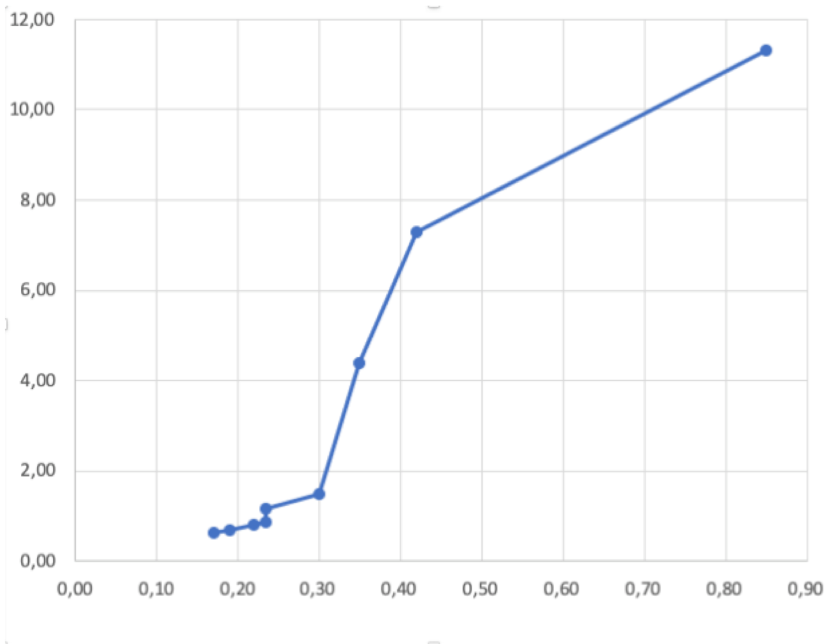


INSA LYON Christian Wolf

Example: Sand corn vs. Slope

Variable 1 : median diameter (mm) of granules of sand
Variable 2 : gradient of beach slope in degrees

Diameter	Slope
0,17	0,63
0,19	0,70
0,22	0,82
0,23	0,88
0,23	1,15
0,30	1,50
0,35	4,40
0,42	7,30
0,85	11,30



https://college.cengage.com/mathematics/brase/understandable_statistics/7e/students/datasets/slr/frames/frame.html

[Physical geography by A.M King, Oxford Press, England]

Data loading and conversion

```
import numpy as np
from numpy import genfromtxt
import torch

# Import the text file into a numpy array
n = genfromtxt('sand_slope.csv', delimiter=';')

# Convert to torch tensor
D = torch.tensor(n, dtype=torch.float32)

# Separate into two different vectors
X = D[:,0].view(-1,1)
Y = D[:,1].view(-1,1)
```

Data loading and conversion

```python
print (X,Y)
```

```
tensor([[0.1700],
        [0.1900],
        [0.2200],
        [0.2350],
        [0.2350],
        [0.3000],
        [0.3500],
        [0.4200],
        [0.8500]])
tensor([[ 0.6300],
        [ 0.7000],
        [ 0.8200],
        [ 0.8800],
        [ 1.1500],
        [ 1.5000],
        [ 4.4000],
        [ 7.3000],
        [11.3000]])
```

The perfect regression?

We suppose the following relationship for a single data pair (x, y):

$$y = x * w$$

with w a correlation coefficient.

For the full data:

$$Y = XW$$

If we the number of data points matches the problem, we can solve the linear problem perfectly with

$$W = X^{-1}Y$$

What happens if we have many more points (typical case?)

Solving the least squares problem

We want to solve the regression problem

$$\min_{W} ||XW - Y||_2$$

Solving the least squares problem

We want to solve the regression problem

$$\min_{W} ||XW - Y||_2$$

A solution with the Moore-Penrose can be given as:

$$X^{+} = (X^T X)^{-1} X^T$$

The regression coefficients are given by $W = X^{+}Y$
In PyTorch:

```
PI = torch.mm(torch.inverse(torch.mm(torch.transpose(X,0,1),X)), torch.transpose(X,0,1))
W = torch.mm(PI,Y)
```

Precision of the solution (L_1 Norm)

```
print (torch.mm(X,W), Y, torch.dist(torch.mm(X,W), Y,
    1))
```

```
tensor([[11.6522]])
tensor([[1.9809],
        [2.2139],
        [2.5635],
        [2.7383],
        [2.7383],
        [3.4956],
        [4.0783],
        [4.8939],
        [9.9043]])
tensor([[ 0.6300],
        [ 0.7000],
        [ 0.8200],
        [ 0.8800],
        [ 1.1500],
        [ 1.5000],
        [ 4.4000],
        [ 7.3000],
        [11.3000]])
tensor(14.1739)
```

Improvement: handle bias

The solution is bad! What happened? We forget the bias term. A single data point is regressed as $y = x * W$ without constant bias.

Solution: Add bias term $y = x * w + b$.

This can be achieved by adding a constant row of "1" to the matrix X:

$$Xc = [\; X \quad \mathbf{1} \;]$$

```
Xc = torch.cat((X, torch.ones((X.size(0),1))), 1)

# Same code as before:
PIc = torch.mm(torch.inverse(torch.mm(torch.transpose(
    Xc,0,1), Xc)), torch.transpose(Xc,0,1))
Wc = torch.mm(PIc,Y)
```

Precision of the solution (L_1 Norm)

```
print (torch.mm(Xc,Wc), Y, torch.dist(torch.mm(Xc,Wc),
     Y, 1))
```

```
tensor([[17.1594],
        [-2.4759]])
tensor([[ 0.4412],
        [ 0.7844],
        [ 1.2991],
        [ 1.5565],
        [ 1.5565],
        [ 2.6719],
        [ 3.5299],
        [ 4.7310],
        [12.1095]])
tensor([[ 0.6300],
        [ 0.7000],
        [ 0.8200],
        [ 0.8800],
        [ 1.1500],
        [ 1.5000],
        [ 4.4000],
        [ 7.3000],
        [11.3000]])
tensor(7.2559)
```

Inhouse solution

PyTorch has a solution ready-to go:

```
G,_ = torch.gels (Y,X)
# The solution is in the first row
print ("By gesls(): W=",G[0])

G,_ = torch.gels (Y,Xc)
# The solution is in the first two rows
print ("By gesls(): W=",G[0:2])
```

```
By gesls(): W= tensor([11.6522])
By gesls(): W= tensor([[17.1594],
        [-2.4759]])
```

Linear classification (2 classes)

A decision function ist modeled through a linear relationship

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0$$

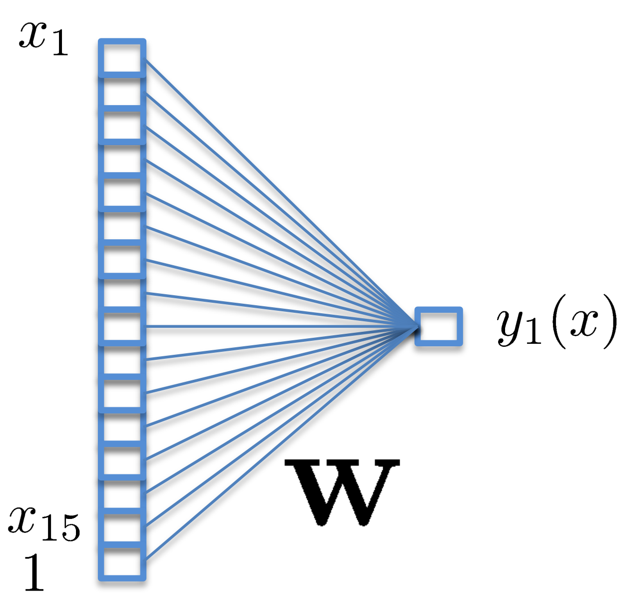
w_0 is the bias term which can be integrated by adding « 1 » to the input vector:

$$y(\mathbf{x}) = \widetilde{\mathbf{w}}^{\mathrm{T}}\widetilde{\mathbf{x}}.$$

Interpretation :

$y(x) \geq 0 \longrightarrow$ Classe 1

$y(x) < 0 \longrightarrow$ Classe 2



The constant « 1 » adds a « biais » to the model

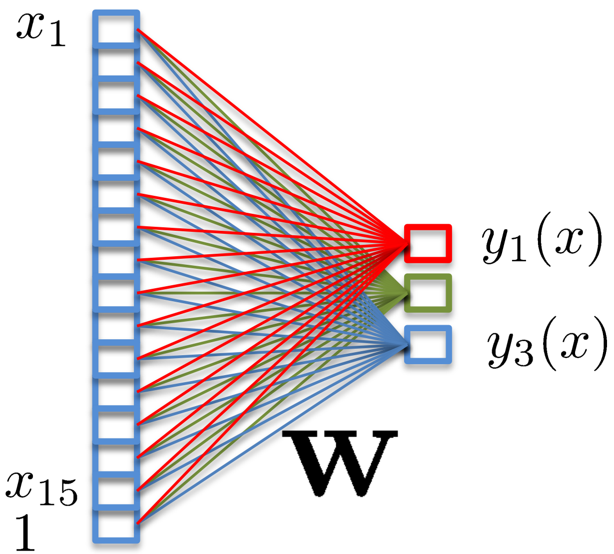
Linear classification(K classes)

Multiple parametric functions

$$y_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0}$$

In vectorial notation with integrated bias:

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}}$$



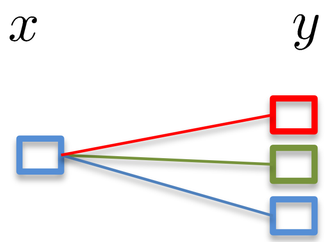
Interpretation :

Class k if $\quad y_k(x) > y_j(x) \quad \forall j \neq k$

« *The winner takes it all* »

Visualization of a simple problem

Linear classifier: 1D input, 3 classes



Input :

$$x = \begin{bmatrix} \cdot \\ 1 \end{bmatrix}$$

Parameters :

$$\mathbf{w}_k = \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$$

Output of one class:

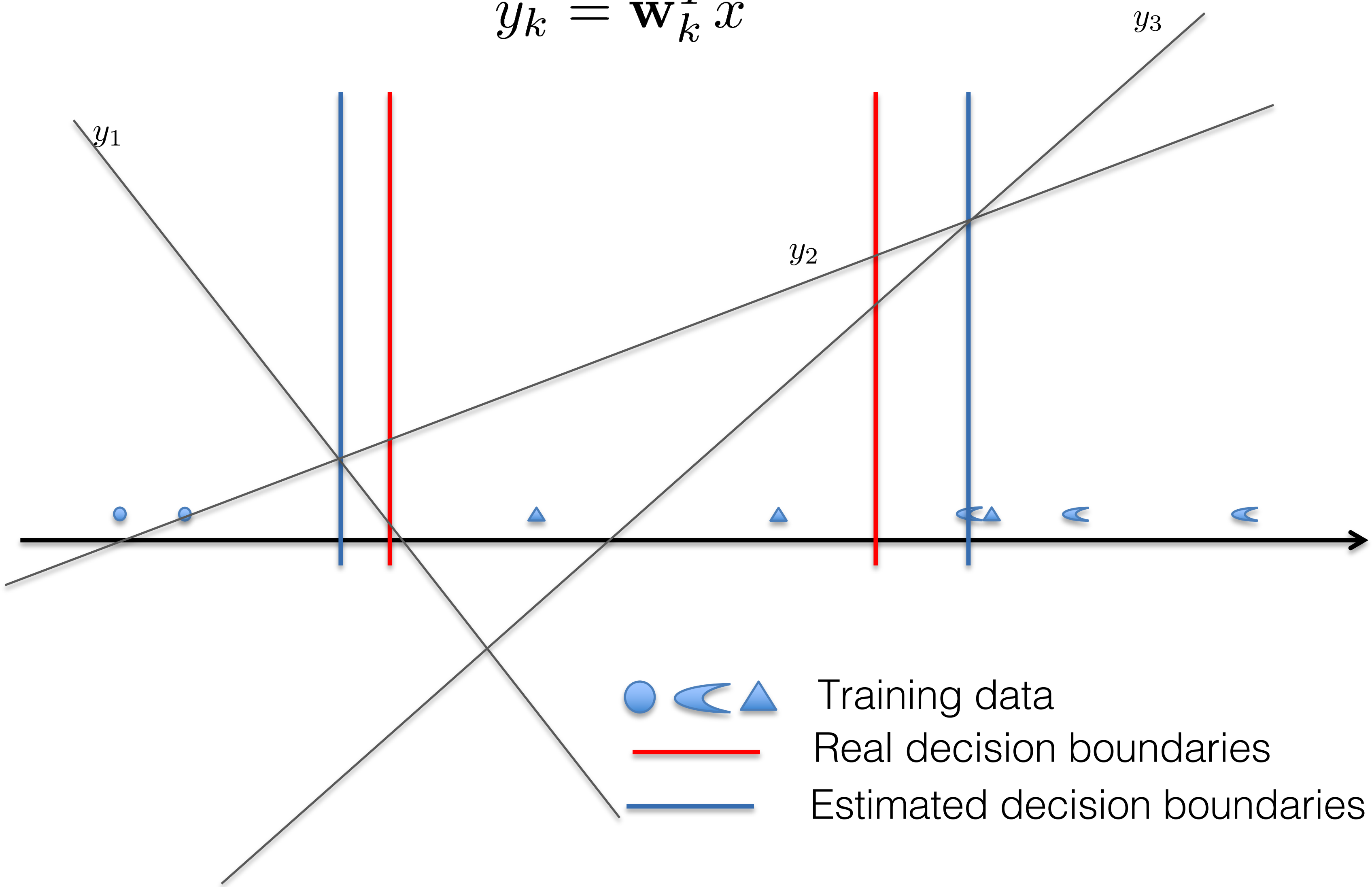
$$y_k = \mathbf{w}_k^T x$$

Output of all classes:

$$y = Wx$$

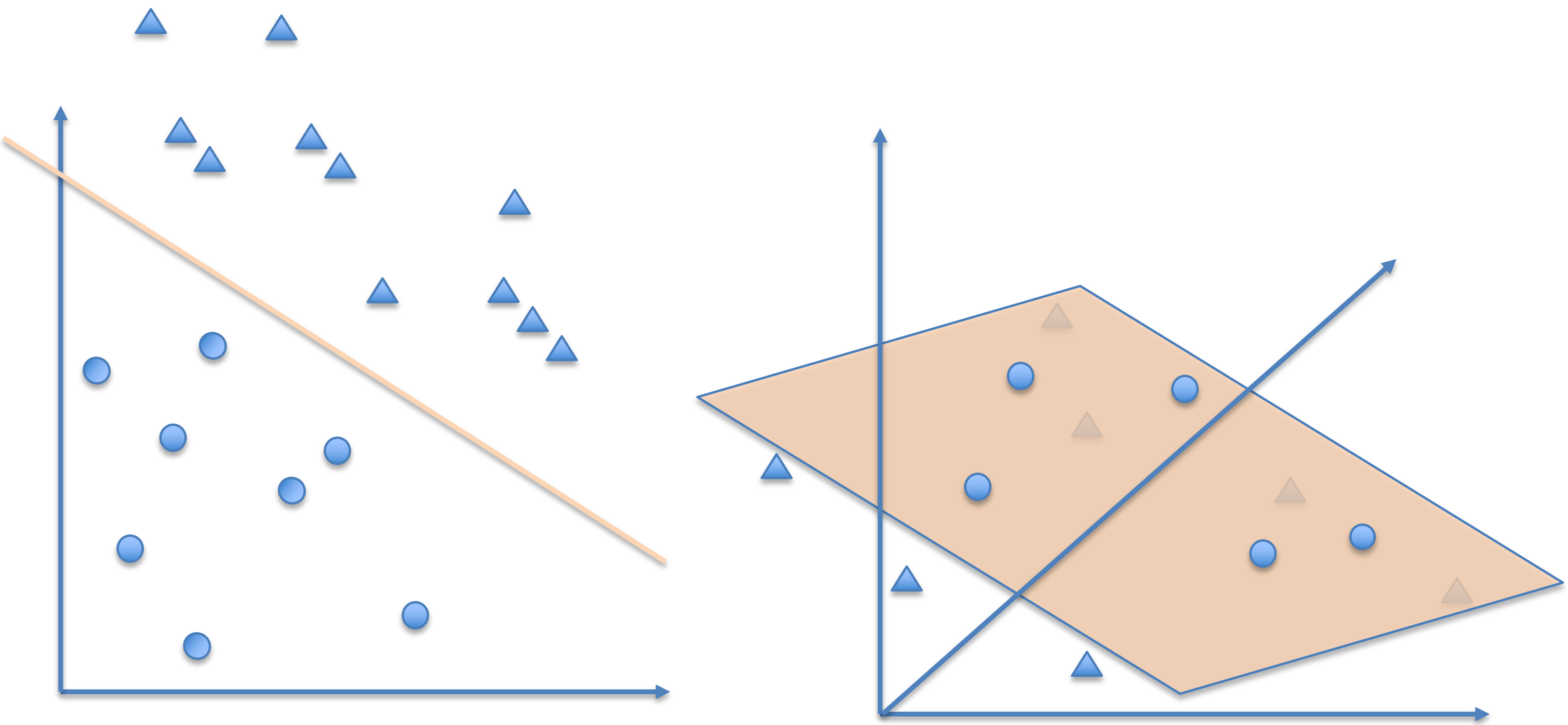
Visualization of a simple problem

$$y_k = \mathbf{w}_k^T x$$



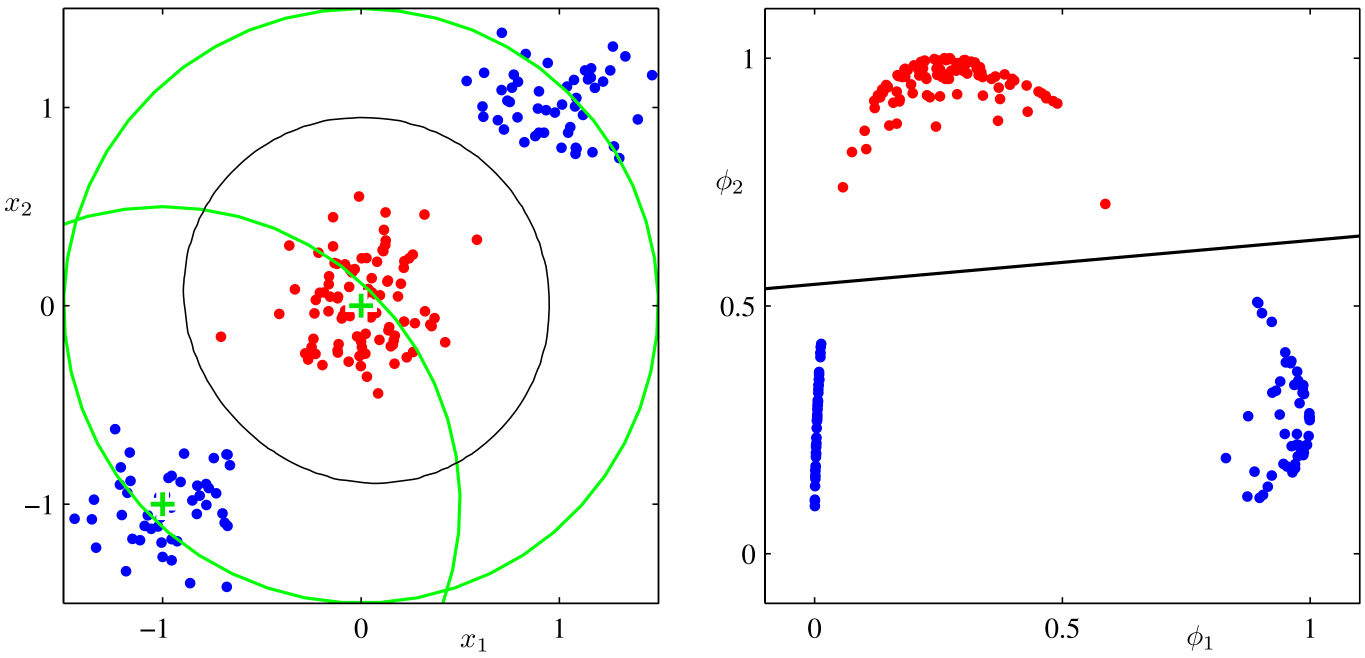
Decision functions

Decision functions of linear classifiers are linear, i.e. d-dimensional hyper-planes in input space.



The non-linear case

Pre-processing : Non-linear transformation of the data, according to the application



Gaussian basis functions

[C. Bishop, Pattern recognition and Machine learning, 2006]

(Generalized) linear models

How do we train a linear model for classification?

What is the loss function?

(A simple L_1 or L_2 norm is not optimal / justified on categorical data like class labels)

How do we train it?

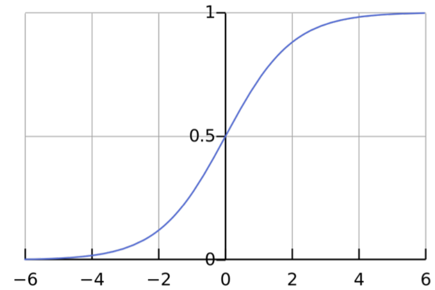
Logistic regression (2 classes)

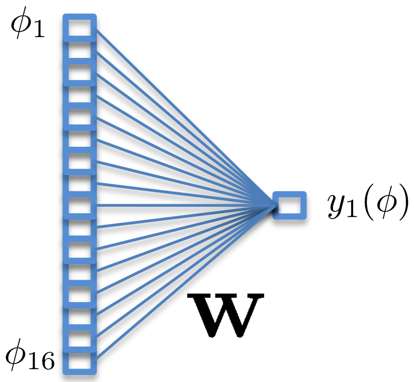
A linear model (eventually on transformed input) + a non-linearity at the output

$$\underbrace{p(\mathcal{C}_1|\boldsymbol{\phi})}_{\text{Direct model of the posterior probability}} = y(\boldsymbol{\phi}) = \sigma\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}\right)$$

σ is the logistic function (« sigmoid ») ensuring that the output is between 0 and 1:

$$\sigma(\eta) = \frac{1}{1+\exp(-\eta)}$$





Logistic regression (K classes)

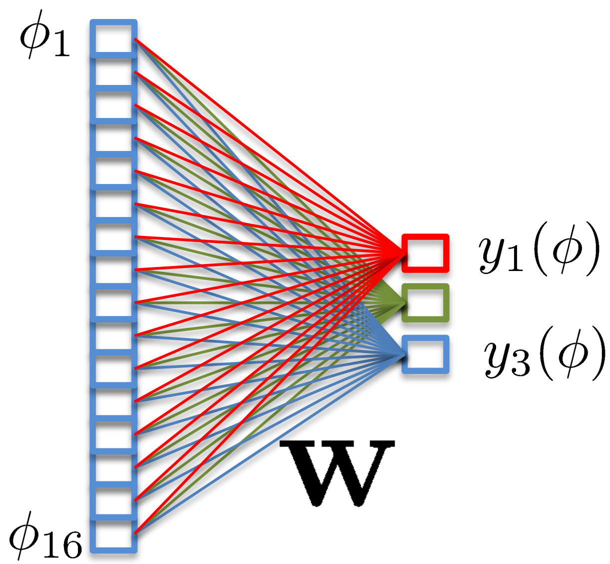
The extension is similar to the linear case

$$p(\mathcal{C}_k|\boldsymbol{\phi}) = y_k(\boldsymbol{\phi}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

« Softmax » to ensure that the output sums to 1

$$a_k = \mathbf{w}_k^{\mathrm{T}}\boldsymbol{\phi}.$$

Linearities



Logistic regression: motivation

Allows a probabilistic view of classification and training.

The objective (loss) is convex: the global minimum can be attained.

The decision functions are still linear
(« Generalized linear model »).

Logistic regression: training

Training dataset:

Inputs x_n transformed by basis functions $\phi(x_n)$

Categorical outputs t_n (« targets »), 1-à-K encoded (« *hot-one-encoded »):*

$$t_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Real (ground-truth) class of sample n

Objective : learn parameters $\mathbf{w}$ according to a criterion

Training

To estimate the parameters $\mathbf{w}$ we minimise the following error function (the negative log likelihood of the data):

$$E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\ln p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}$$

« Cross-entropy loss »

It can be minimized by gradient descent:

$$\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{n=1}^{N} (y_{nj} - t_{nj})\, \boldsymbol{\phi}_n$$

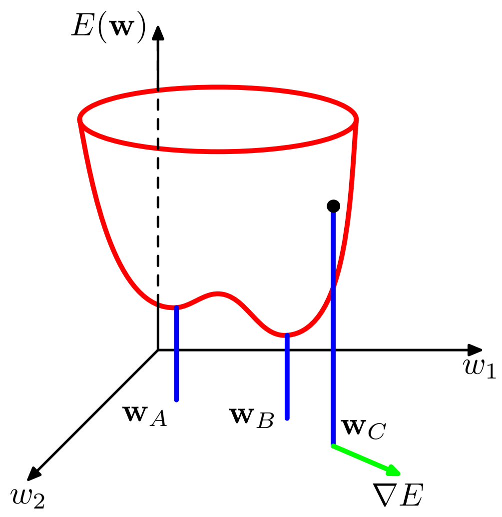
[C. Bishop, Pattern recognition and Machine learning, 2006]

Learning by gradient descent

Iterative minimisation through gradient descent:

$$\theta^{[t+1]} = \theta^{[t]} + \boxed{\nu} \nabla \mathcal{L}(h(x, \theta), y^*)$$

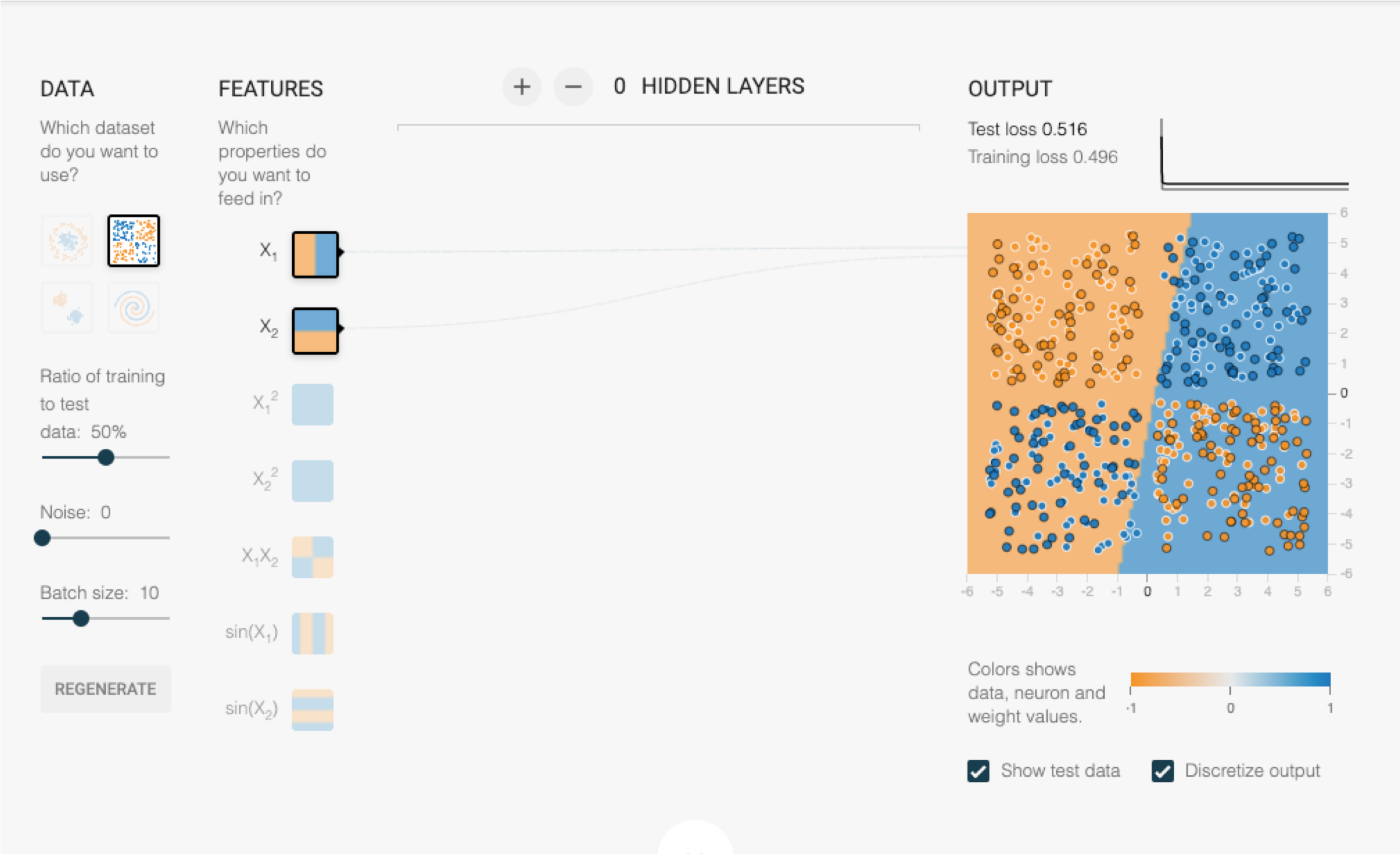
Learning rate



[Figure: C. Bishop, 2006]

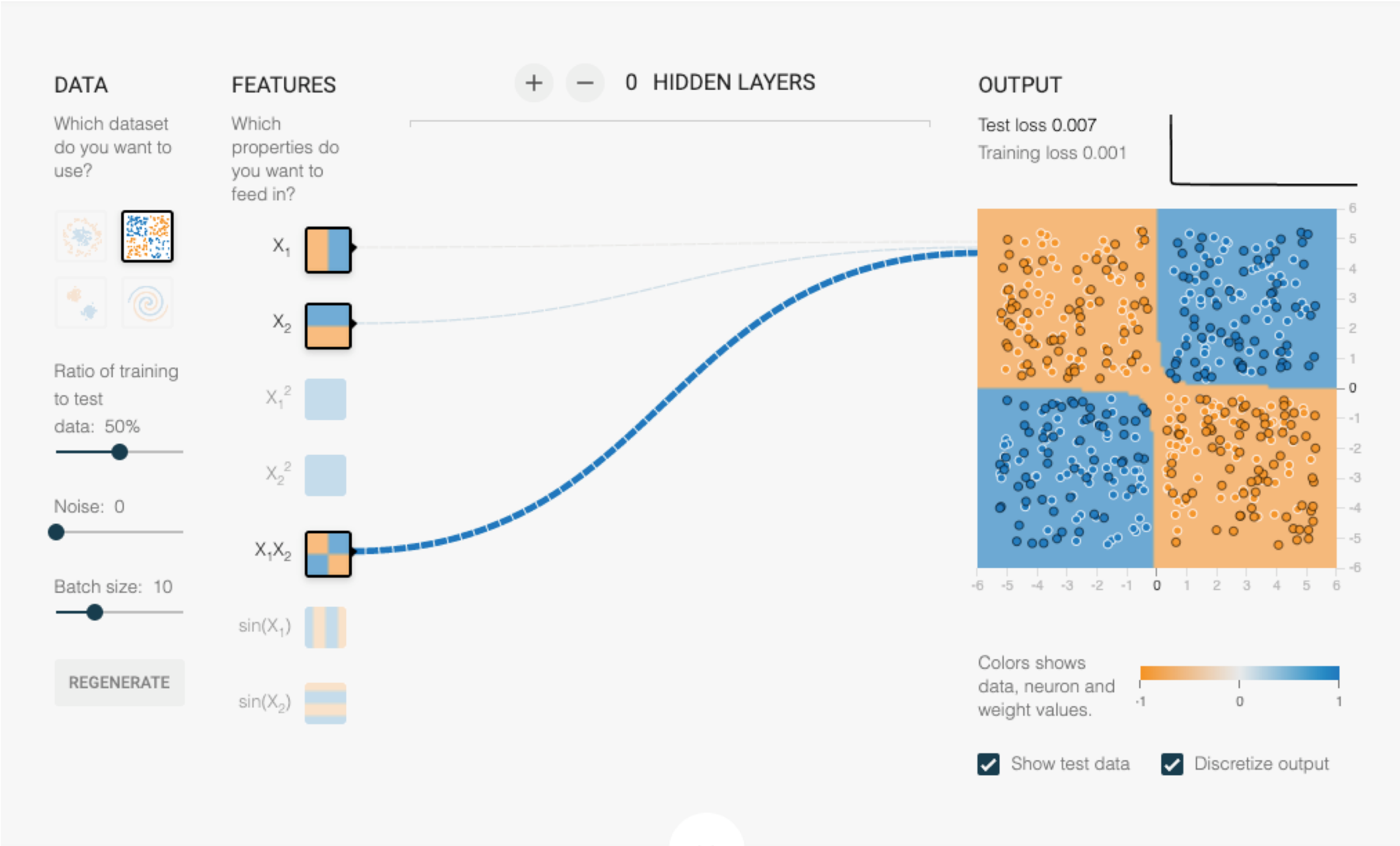
Can be blocked in a local minimum (not that it matters much …)

Linear separation fails on XOR



https://playground.tensorflow.org

Unless we calcuate features



https://playground.tensorflow.org

Example: Wisconsin breast-cancer

Id	Cl.thickness	Cell.size	Cell.shape	Marg.adhesion	Epith.c.size	Bare.nuclei	Bl.cromatin	Normal.nucleoli	Mitoses	Class
1000025	5	1	1	1	2	1	3	1	1	benign
1002945	5	4	4	5	7	10	3	2	1	benign
1015425	3	1	1	1	2	2	3	1	1	benign
1016277	6	8	8	1	3	4	3	7	1	benign
1017023	4	1	1	3	2	1	3	1	1	benign
1017122	8	10	10	8	7	10	9	7	1	malignant
1018099	1	1	1	1	2	10	3	1	1	benign
1018561	2	1	2	1	2	1	3	1	1	benign
1033078	2	1	1	1	2	1	1	1	5	benign
1033078	4	2	1	1	2	1	2	1	1	benign
1035283	1	1	1	1	1	1	3	1	1	benign
1036172	2	1	1	1	2	1	2	1	1	benign
1041801	5	3	3	3	2	3	4	4	1	malignant

```
chris pytorch $ head breast-cancer-wisconsin.csv
1000025,5,1,1,1,2,1,3,1,1,2
1002945,5,4,4,5,7,10,3,2,1,2
1015425,3,1,1,1,2,2,3,1,1,2
1016277,6,8,8,1,3,4,3,7,1,2
1017023,4,1,1,3,2,1,3,1,1,2
1017122,8,10,10,8,7,10,9,7,1,4
1018099,1,1,1,1,2,10,3,1,1,2
1018561,2,1,2,1,2,1,3,1,1,2
1033078,2,1,1,1,2,1,1,1,5,2
1033078,4,2,1,1,2,1,2,1,1,2
```

https://www.machinelearningplus.com/machine-learning/logistic-regression-tutorial-examples-r/

Data loading and conversion

```
import numpy as np
from numpy import genfromtxt
import torch
from torch.nn import functional as F

# Import the text file into a numpy array
n = genfromtxt('breast-cancer-wisconsin-cleaned.csv',
    delimiter=',')
D = torch.tensor(n, dtype=torch.float32)
N_samples = D.size(0)

# The input is the full matrix without first and
# last column, plus the 1 column for the bias
X=D[:,1:-1]
X = torch.cat ((X, torch.ones((X.size(0),1))),1)

# The targets. Change all 2->0 and 4->1
T=D[:,-1:]
T[T==2]=0
T[T==4]=1
```

The model

```
class LogisticRegression(torch.nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()

        # The linear layer (input dim, output dim)
        # It also contains a weight matrix
        # (here single output-> vector)
        self.fc1 = torch.nn.Linear(10, 1)

    # The forward pass of the network. x is the input
    def forward(self, x):
        return F.sigmoid(self.fc1(x))
```

Set up the environment

```python
# Instantiate the model
model = LogisticRegression()

# The loss function: binary cross-entropy
criterion = torch.nn.BCELoss()

# Set up the optimizer: stochastic gradient descent
# with a learning rate of 0.01
optimizer = torch.optim.SGD(model.parameters(), lr
    =0.01)
```

Iterative training

```
# 1 epoch = 1 pass over the full dataset
for epoch in range(200):
    print ("Starting epoch", epoch, " ",end='')
    calcAccuracy()

    for sample in range(N_samples):
        # model -> train mode, clear gradients
        model.train()
        optimizer.zero_grad()

        # Forward pass (stimulate model with inputs)
        y = model(X[sample,:])

        # Compute Loss
        loss = criterion(y, T[sample])

        # Backward pass: calculate the gradients
        loss.backward()

        # One step of stochastic gradient descent
        optimizer.step()
```

Evaluation

Calculate the accuracy (in percent) at each epoch:
Proportion of correctly classified samples.
Random performance $= 50\%$ on a binary task.

```
def calcAccuracy():

    # model -> eval mode
    model.eval()
    correct = 0.0
    for sample in range(N_samples):

      # threshold the output probability
      y = 1 if model(X[sample,:]) > 0.5 else 0
      correct += (y == T[sample]).numpy()

    print ("Accuracy = ", 100.0*correct/N_samples)
```

Results

```
Starting epoch 0  Accuracy =  [34.69985359]
Starting epoch 1  Accuracy =  [91.21522694]
Starting epoch 2  Accuracy =  [93.99707174]
Starting epoch 3  Accuracy =  [95.16837482]
Starting epoch 4  Accuracy =  [95.75402635]
Starting epoch 5  Accuracy =  [95.60761347]
Starting epoch 6  Accuracy =  [95.75402635]
Starting epoch 7  Accuracy =  [95.75402635]
Starting epoch 8  Accuracy =  [96.33967789]
Starting epoch 9  Accuracy =  [96.48609078]
Starting epoch 10  Accuracy =  [96.63250366]
Starting epoch 11  Accuracy =  [96.63250366]
Starting epoch 12  Accuracy =  [96.63250366]
Starting epoch 13  Accuracy =  [96.63250366]
Starting epoch 14  Accuracy =  [96.63250366]
```

(...)

```
Starting epoch 197  Accuracy =  [97.07174231]
Starting epoch 198  Accuracy =  [97.07174231]
Starting epoch 199  Accuracy =  [97.07174231]
```

What is missing?

- The model is simpler than deep neural networks, but sufficient for the task.
- We did not use batch processing, i.e. using more than one sample for a given gradient update
- We calculated performance on the training set. We might overfit.