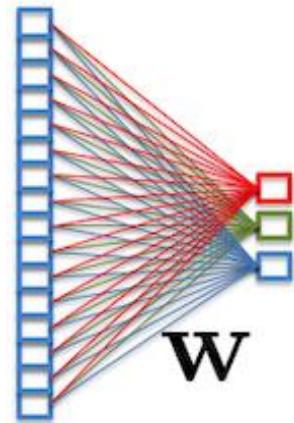


AI and Data Analysis

2.2 (Generalized) Linear models

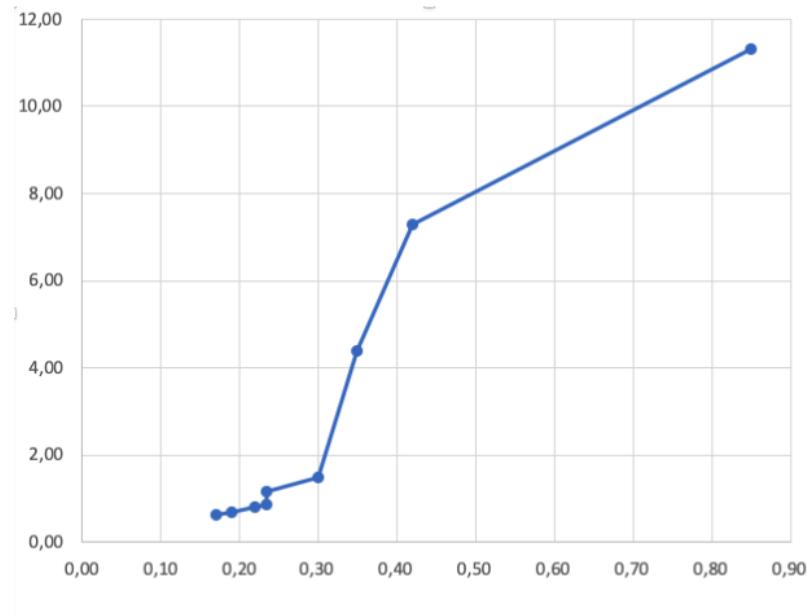


Example: Sand corn vs. Slope

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]

Sand_slope.csv
Can be downloaded on
the lecture website!

Data loading and conversion

```
1 import numpy as np
2 from numpy import genfromtxt
3 import torch
4
5 # Import the text file into a numpy array
6 n = genfromtxt('sand_slope.csv', delimiter=';',
7                 skip_header=1)
8
9 # Convert to torch tensor
10 D = torch.tensor(n, dtype=torch.float32)
11
12 # Separate into two different vectors
13 X = D[:,0].view(-1,1)
14 Y = D[:,1].view(-1,1)
```

Data loading and conversion

```
1 print (X,Y)
```

```
1 tensor ([[0.1700] ,  
2 [0.1900] ,  
3 [0.2200] ,  
4 [0.2350] ,  
5 [0.2350] ,  
6 [0.3000] ,  
7 [0.3500] ,  
8 [0.4200] ,  
9 [0.8500]]))  
10 tensor ([[ 0.6300] ,  
11 [ 0.7000] ,  
12 [ 0.8200] ,  
13 [ 0.8800] ,  
14 [ 1.1500] ,  
15 [ 1.5000] ,  
16 [ 4.4000] ,  
17 [ 7.3000] ,  
18 [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:

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

Precision of the solution (L_1 Norm)

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

```
1 tensor ([[11.6522]])  
2 tensor ([[1.9809] ,  
3          [2.2139] ,  
4          [2.5635] ,  
5          [2.7383] ,  
6          [2.7383] ,  
7          [3.4956] ,  
8          [4.0783] ,  
9          [4.8939] ,  
10         [9.9043]] )  
11 tensor ([[ 0.6300] ,  
12          [ 0.7000] ,  
13          [ 0.8200] ,  
14          [ 0.8800] ,  
15          [ 1.1500] ,  
16          [ 1.5000] ,  
17          [ 4.4000] ,  
18          [ 7.3000] ,  
19          [11.3000]] )  
20 tensor(14.1739)
```

Improvement: handle bias

The solution is bad! What happened? We forgot 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 :

$$Xc = [X \ 1]$$

```
1 Xc = torch.cat((X, torch.ones((X.size(0),1))), 1)
2
3 # Same code as before:
4 PItc = torch.mm(torch.inverse(torch.mm(torch.transpose(
    Xc,0,1), Xc)), torch.transpose(Xc,0,1))
5 Wc = torch.mm(PItc, Y)
```

Precision of the solution (L_1 Norm)

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

```
1 tensor([[17.1594],  
2      [-2.4759]])  
3 tensor([[ 0.4412],  
4      [ 0.7844],  
5      [ 1.2991],  
6      [ 1.5565],  
7      [ 1.5565],  
8      [ 2.6719],  
9      [ 3.5299],  
10     [ 4.7310],  
11     [12.1095]])  
12 tensor([[ 0.6300],  
13     [ 0.7000],  
14     [ 0.8200],  
15     [ 0.8800],  
16     [ 1.1500],  
17     [ 1.5000],  
18     [ 4.4000],  
19     [ 7.3000],  
20     [11.3000]])  
21 tensor(7.2559)
```

Inhouse solution

PyTorch has a solution ready-to go:

```
1 G,_ = torch.gels (Y,X)
2 # The solution is in the first row
3 print ("By gesls(): W=",G[0])
4
5 G,_ = torch.gels (Y,Xc)
6 # The solution is in the first two rows
7 print ("By gesls(): W=",G[0:2])
```

```
1 By gesls(): W= tensor([11.6522])
2 By gesls(): W= tensor([[17.1594],
3                      [-2.4759]])
```

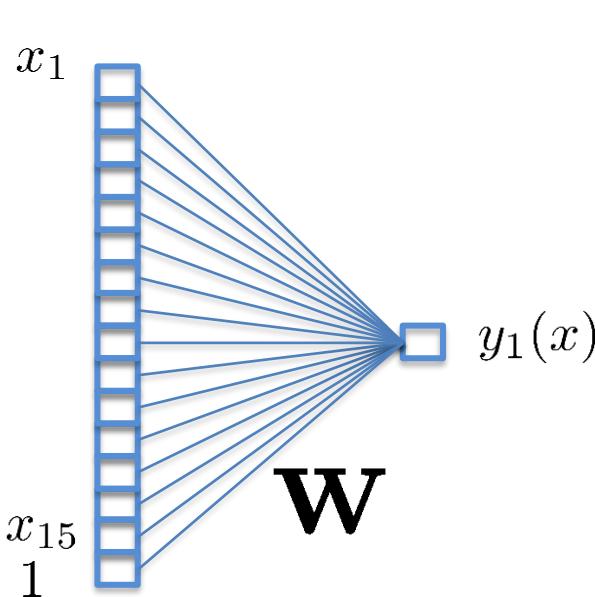
Linear classification (2 classes)

A decision function is modeled through a linear relationship

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

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

$$y(\mathbf{x}) = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$$



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

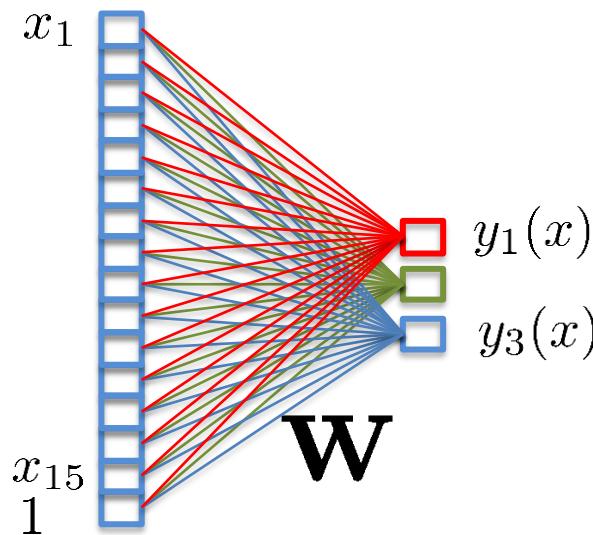
Linear classification(K classes)

Multiple parametric functions

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

In vectorial notation with integrated bias:

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



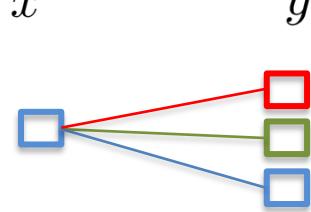
Interpretation :

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

« The winner takes it all »

Visualization of a simple problem

Linear classifier: 1D input, 3 classes



Input :

$$x = \begin{bmatrix} \cdot \\ 1 \end{bmatrix} \quad \mathbf{w}_k = \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$$

Parameters :

Output of one class:

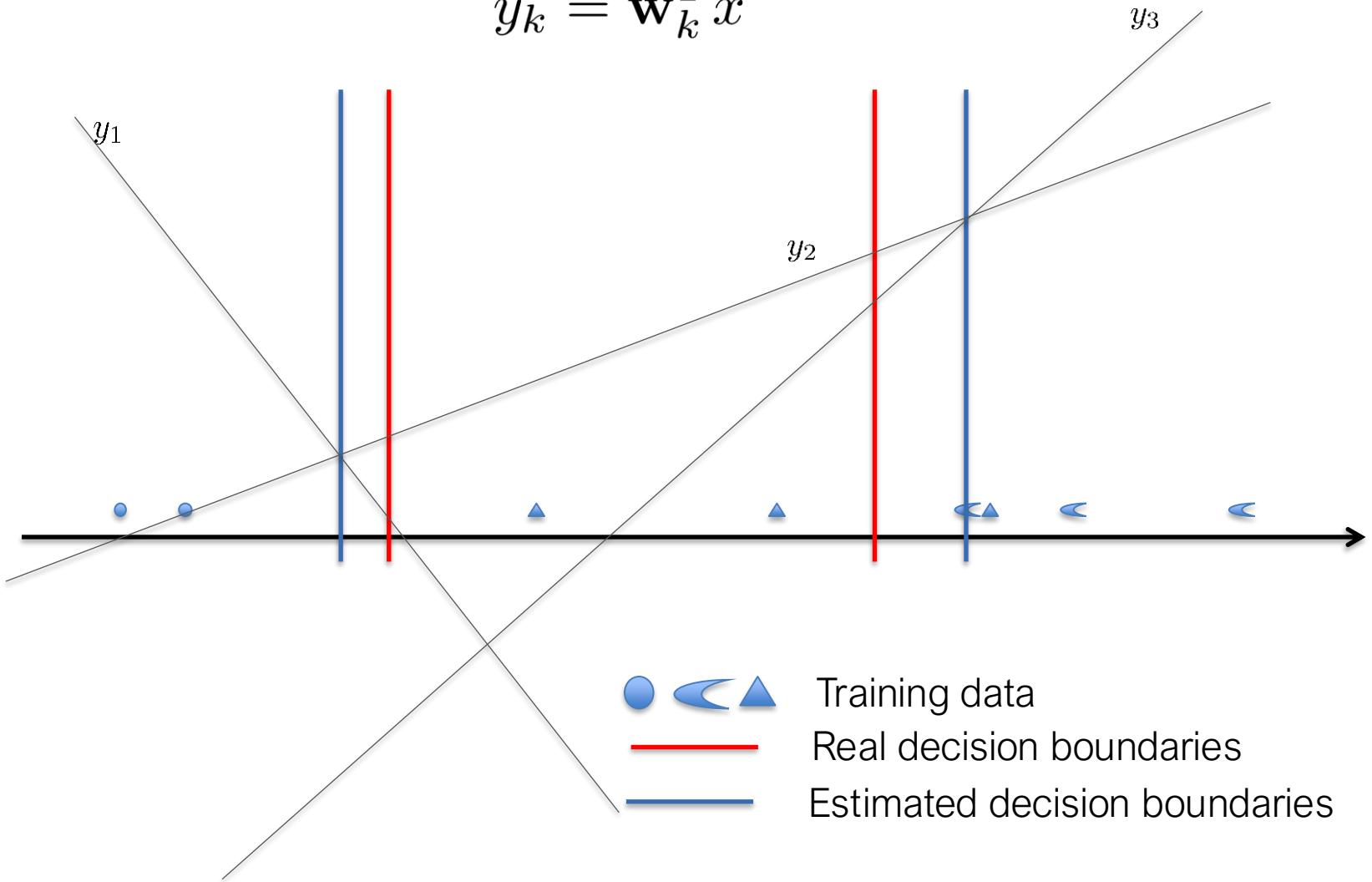
$$y_k = \mathbf{w}_k^T x$$

Output of all classes:

$$y = Wx$$

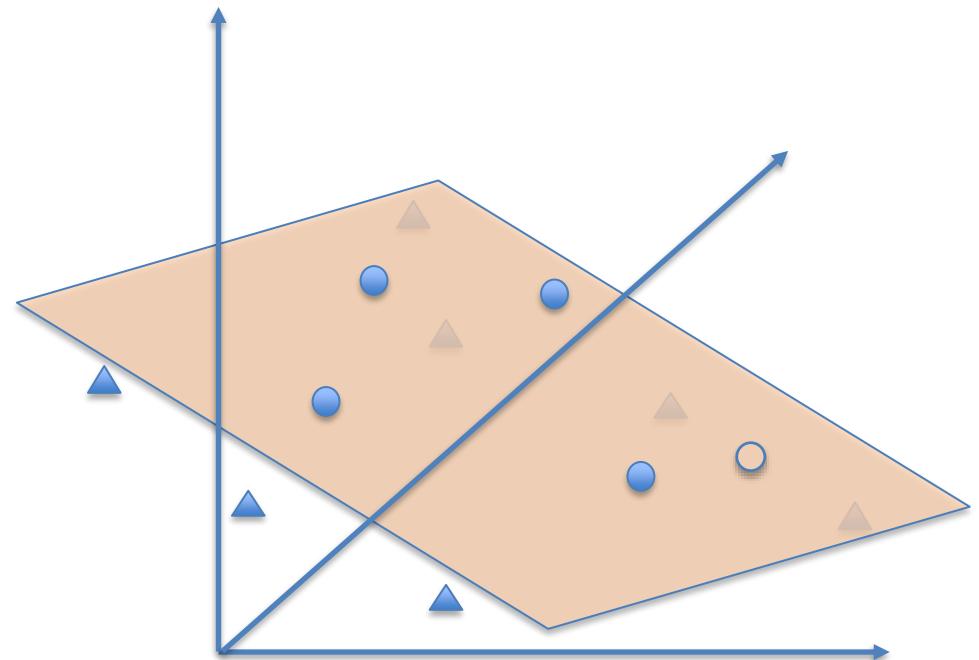
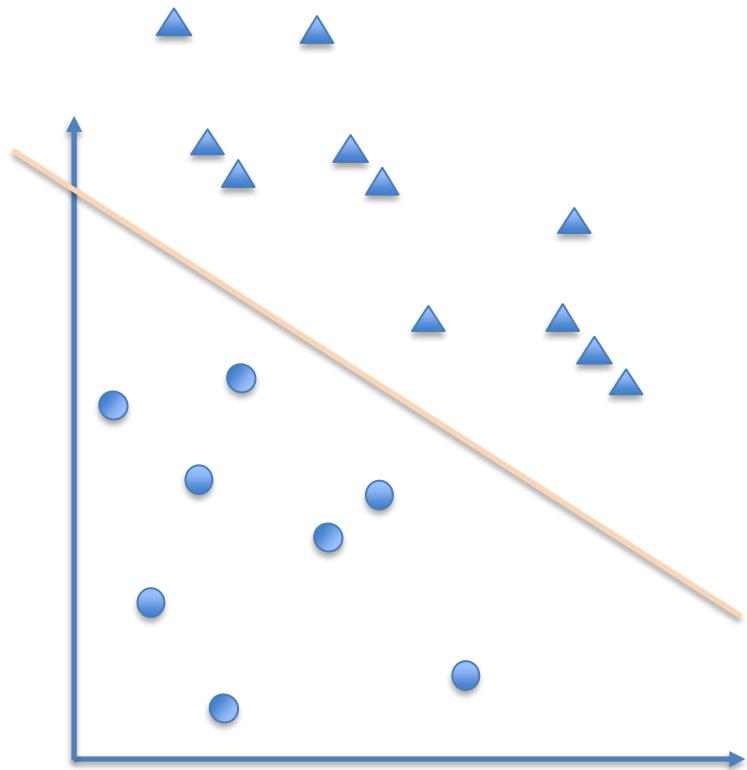
Visualization of a simple problem

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



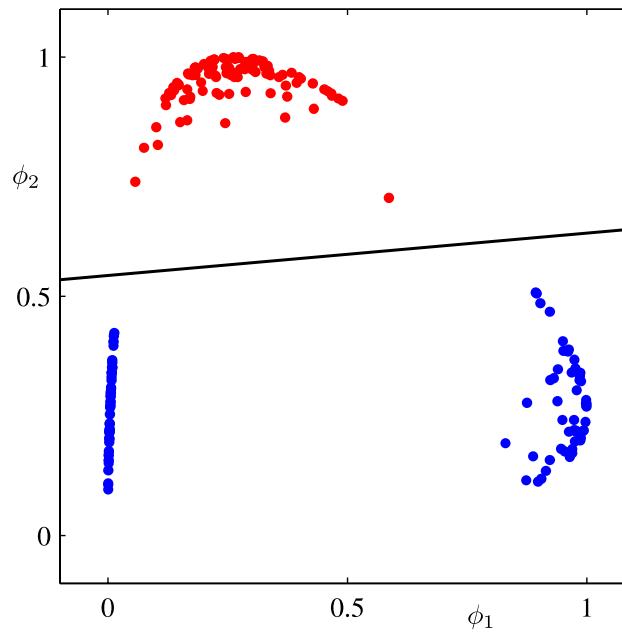
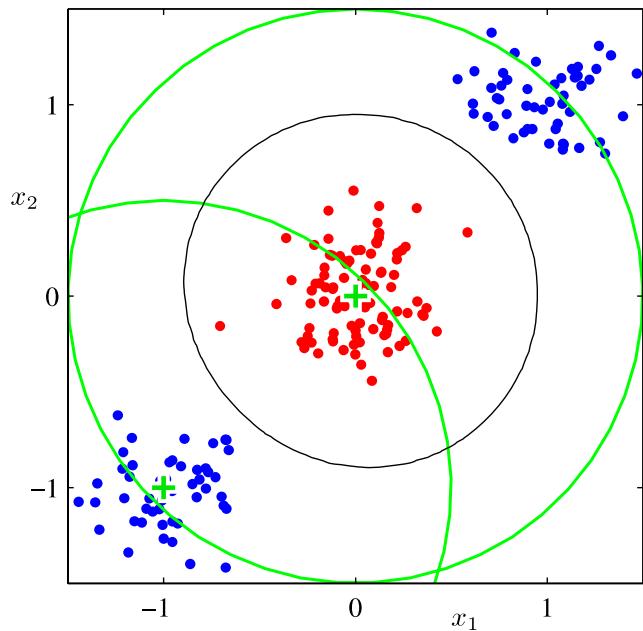
Decision functions

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



The non-linear case

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



Gaussian basis functions

(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)

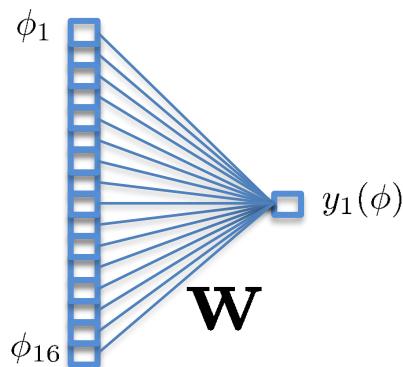
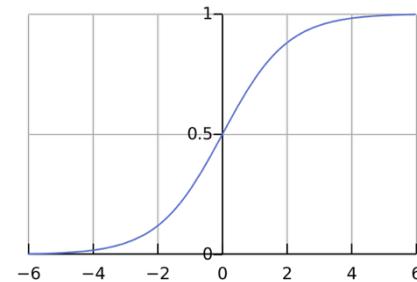
A linear model (eventually on transformed input) + a non-linearity at the output

$$p(\mathcal{C}_1 | \phi) = y(\phi) = \sigma(\mathbf{w}^T \phi)$$

Direct model of the posterior probability

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)

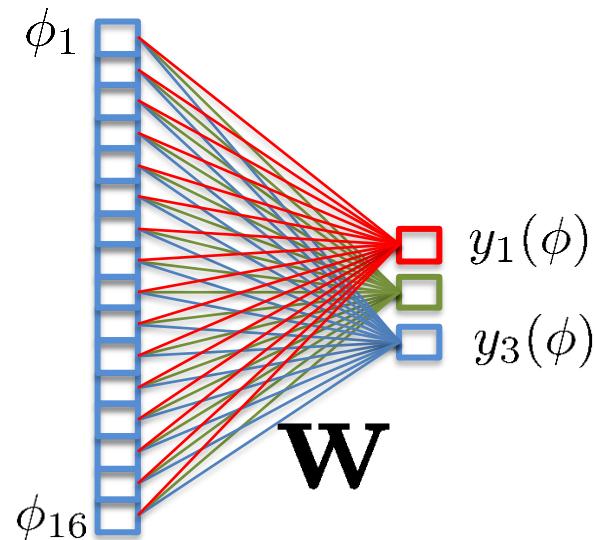
The extension is similar to the linear case

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

« Softmax » to ensure that the output sums to 1

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

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* »):

$$t_n = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad \text{Real (ground-truth) class of sample n}$$

Objective : learn parameters w according to a criterion

Training

To estimate the parameters we minimise the following error function (the negative log likelihood of the data):

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

« Cross-entropy loss »

It can be minimized by gradient descent:

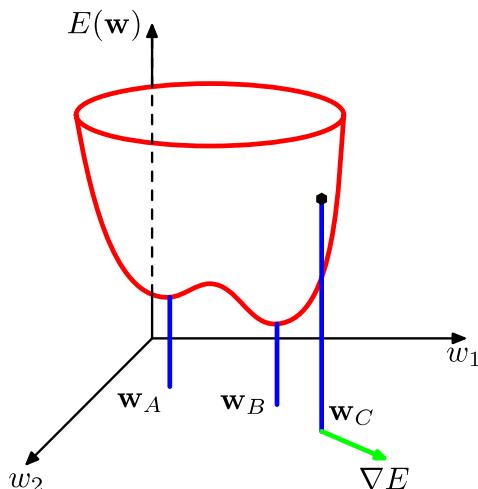
$$\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K) = \sum_{n=1}^N (y_{nj} - t_{nj}) \boldsymbol{\phi}_n$$

Learning by gradient descent

Iterative minimisation through **gradient descent**:

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

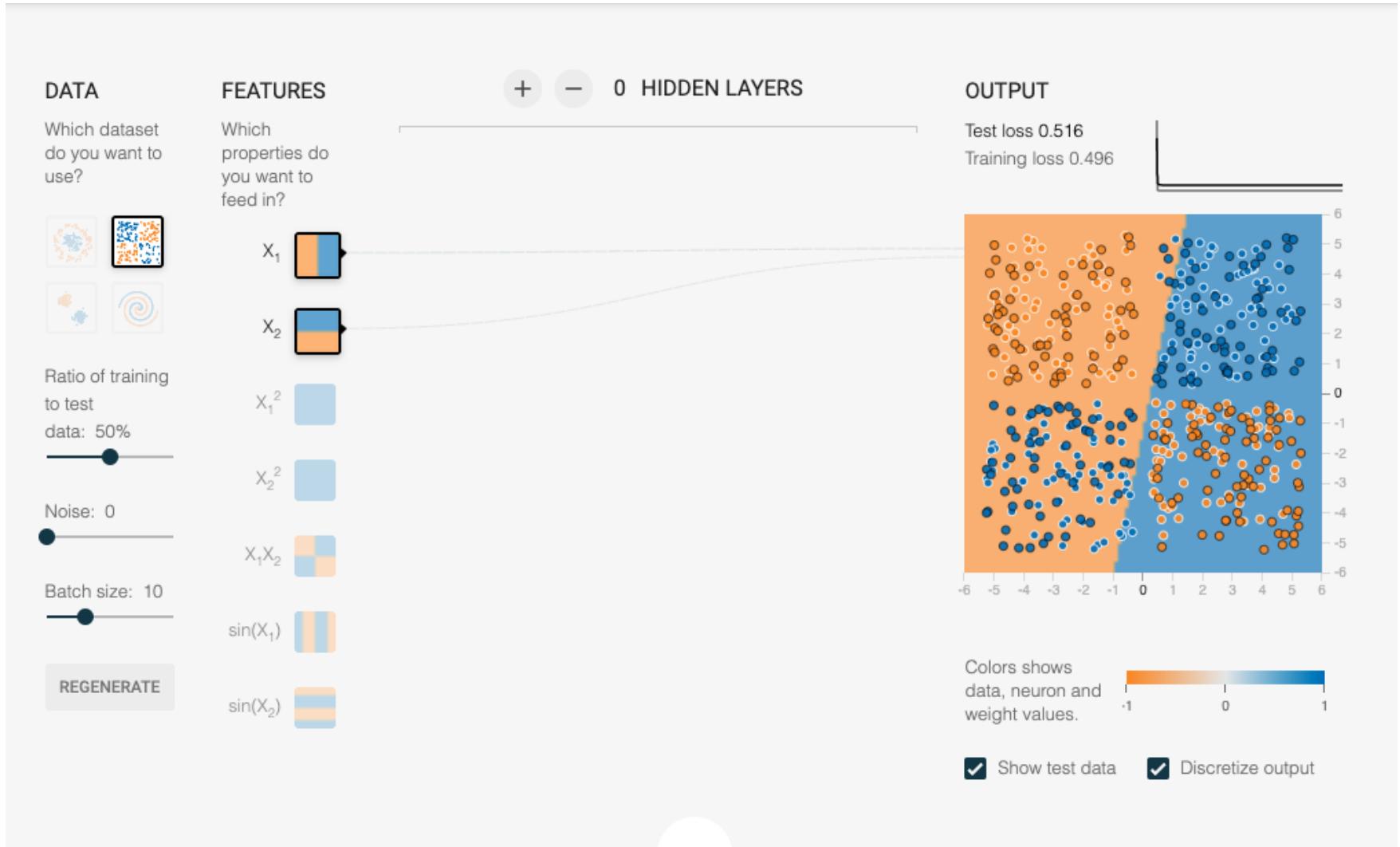
 *Learning rate*



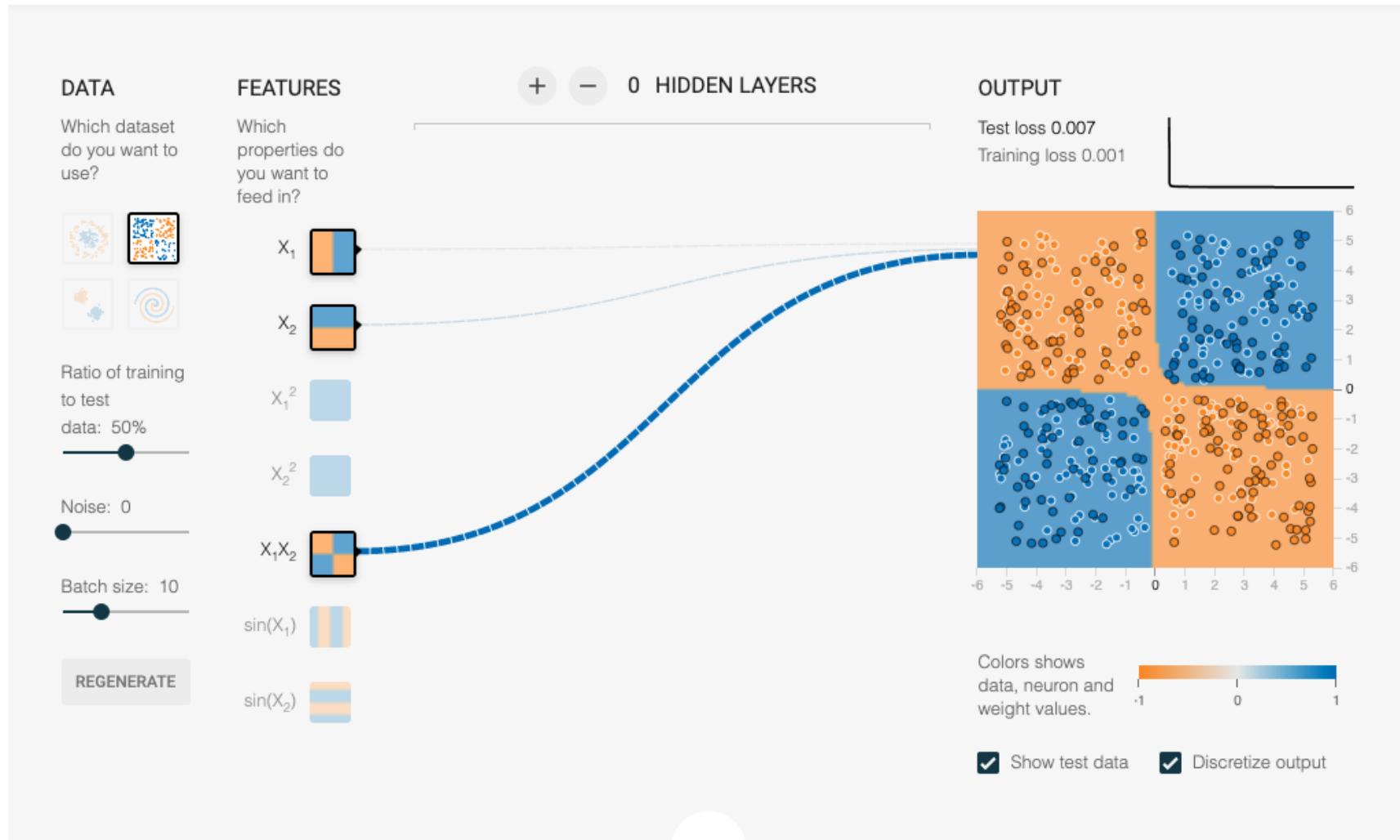
Can be blocked in a local minimum (not that it matters much ...)

[Figure: C. Bishop, 2006]

Linear separation fails on XOR



Unless we calculate 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
```

Data loading and conversion

```
1 import numpy as np
2 from numpy import genfromtxt
3 import torch
4 from torch.nn import functional as F
5
6 # Import the text file into a numpy array
7 n = genfromtxt('breast-cancer-wisconsin-cleaned.csv',
8     delimiter=',')
9 D = torch.tensor(n, dtype=torch.float32)
10 N_samples = D.size(0)
11
12 # The input is the full matrix without first and
13 # last column, plus the 1 column for the bias
14 X=D[:,1:-1]
15 X = torch.cat ((X, torch.ones((X.size(0),1))),1)
16
17 # The targets. Change all 2->0 and 4->1
18 T=D[:, -1:]
19 T [T==2]=0
20 T [T==4]=1
```

The model

```
1 class LogisticRegression(torch.nn.Module):
2     def __init__(self):
3         super(LogisticRegression, self).__init__()
4
5         # The linear layer (input dim, output dim)
6         # It also contains a weight matrix
7         # (here single output-> vector)
8         self.fc1 = torch.nn.Linear(10, 1)
9
10    # The forward pass of the network. x is the input
11    def forward(self, x):
12        return F.sigmoid(self.fc1(x))
```

Set up the environment

```
1 # Instantiate the model
2 model = LogisticRegression()
3
4 # The loss function: binary cross-entropy
5 criterion = torch.nn.BCELoss()
6
7 # Set up the optimizer: stochastic gradient descent
8 # with a learning rate of 0.01
9 optimizer = torch.optim.SGD(model.parameters(), lr
=0.01)
```

Iterative training

```
1 # 1 epoch = 1 pass over the full dataset
2 for epoch in range(200):
3     print ("Starting epoch", epoch, " ", end=' ')
4     calcAccuracy()
5
6     for sample in range(N_samples):
7         # model -> train mode, clear gradients
8         model.train()
9         optimizer.zero_grad()
10
11        # Forward pass (stimulate model with inputs)
12        y = model(X[sample,:])
13
14        # Compute Loss
15        loss = criterion(y, T[sample])
16
17        # Backward pass: calculate the gradients
18        loss.backward()
19
20        # One step of stochastic gradient descent
21        optimizer.step()
```

Evaluation

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

```
1 def calcAccuracy():
2
3     # model -> eval mode
4     model.eval()
5     correct = 0.0
6     for sample in range(N_samples):
7
8         # threshold the output probability
9         y = 1 if model(X[sample,:]) > 0.5 else 0
10        correct += (y == T[sample]).numpy()
11
12        print ("Accuracy = ", 100.0*correct/N_samples)
```

Results

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

(...)

```
1 Starting epoch 197 Accuracy = [97.07174231]
2 Starting epoch 198 Accuracy = [97.07174231]
3 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.