

CAS – Ifl – Deep Learning

François Fleuret

Course content:

- Machine learning basics.
- Multi-layer perceptron, convolutions, gradient descent.
- Graphs of tensor operators, autograd.
- Deep-learning specific techniques.
- Computer-vision, generative models, a bit of NLP.

CAS – Deep learning

1. Tensors and multi-layer perceptrons

François Fleuret

<https://www.idiap.ch/~fleuret/>

Fri Feb 22 13:18:05 UTC 2019


1.1. From neural networks to deep learning

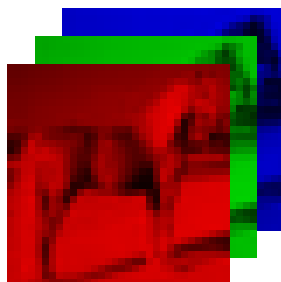
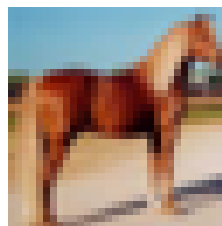
Many applications require the automatic extraction of “refined” information from raw signal (e.g. image recognition, automatic speech processing, natural language processing, robotic control, geometry reconstruction).



(ImageNet)

Our brain is so good at interpreting visual information that the “semantic gap” is hard to assess intuitively.

This  is a horse



```
>>> from torchvision.datasets import CIFAR10
>>> cifar = CIFAR10('./data/cifar10/', train=True, download=True)
Files already downloaded and verified
>>> x = torch.from_numpy(cifar.train_data)[43].transpose(2, 0).transpose(1, 2)
>>> x[:, :4, :8]
tensor([[ [ 99,  98, 100, 103, 105, 107, 108, 110],
         [100, 100, 102, 105, 107, 109, 110, 112],
         [104, 104, 106, 109, 111, 112, 114, 116],
         [109, 109, 111, 113, 116, 117, 118, 120]],

        [[166, 165, 167, 169, 171, 172, 173, 175],
         [166, 164, 167, 169, 169, 171, 172, 174],
         [169, 167, 170, 171, 171, 173, 174, 176],
         [170, 169, 172, 173, 175, 176, 177, 178]],

        [[198, 196, 199, 200, 200, 202, 203, 204],
         [195, 194, 197, 197, 197, 199, 200, 201],
         [197, 195, 198, 198, 198, 199, 201, 202],
         [197, 196, 199, 198, 198, 199, 200, 201]]], dtype=torch.uint8)
```

Extracting semantic automatically requires models of extreme complexity, which cannot be designed by hand.

Techniques used in practice consist of

1. defining a parametric model, and
2. optimizing its parameters by “making it work” on training data.

This is similar to biological systems for which the model (e.g. brain structure) is DNA-encoded, and parameters (e.g. synaptic weights) are tuned through experiences.

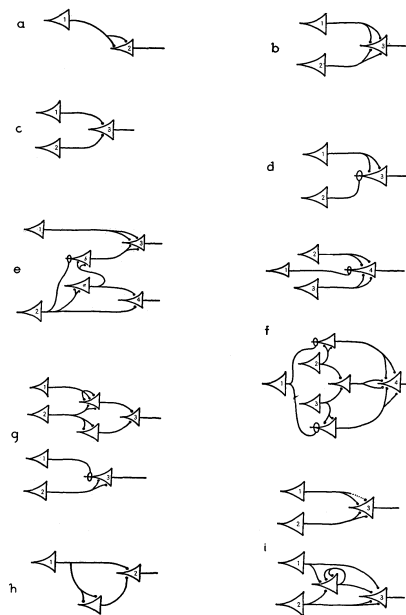
Deep learning encompasses software technologies to scale-up to billions of model parameters and as many training examples.

There are strong connections between standard statistical modeling and machine learning.

Classical ML methods combine a “learnable” model from statistics (e.g. “linear regression”) with prior knowledge in pre-processing.

“Artificial neural networks” pre-dated these approaches, and do not follow that dichotomy. They consist of “deep” stacks of parametrized processing.

From artificial neural networks to “Deep Learning”

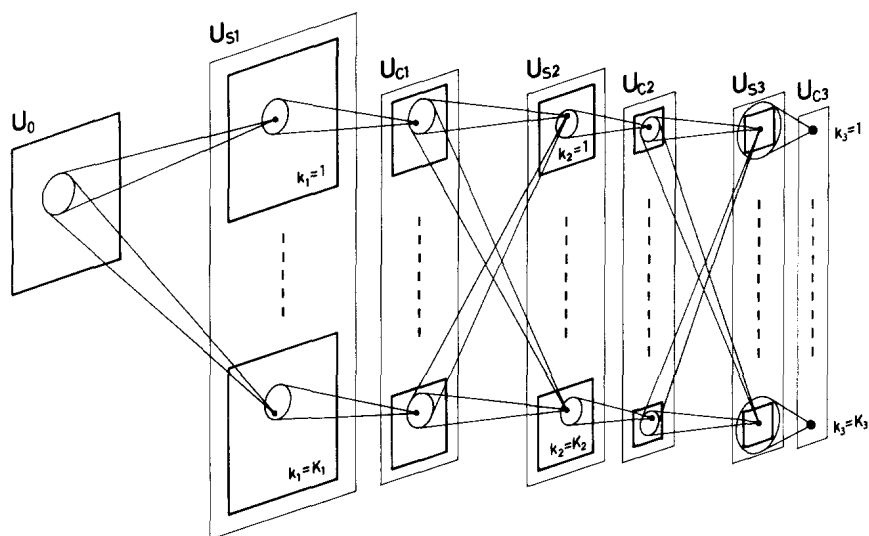


Networks of “Threshold Logic Unit”

(McCulloch and Pitts, 1943)

- 1949 – Donald Hebb proposes the Hebbian Learning principle.
- 1951 – Marvin Minsky creates the first ANN (Hebbian learning, 40 neurons).
- 1958 – Frank Rosenblatt creates a perceptron to classify 20×20 images.
- 1959 – David H. Hubel and Torsten Wiesel demonstrate orientation selectivity and columnar organization in the cat's visual cortex.
- 1982 – Paul Werbos proposes back-propagation for ANNs.

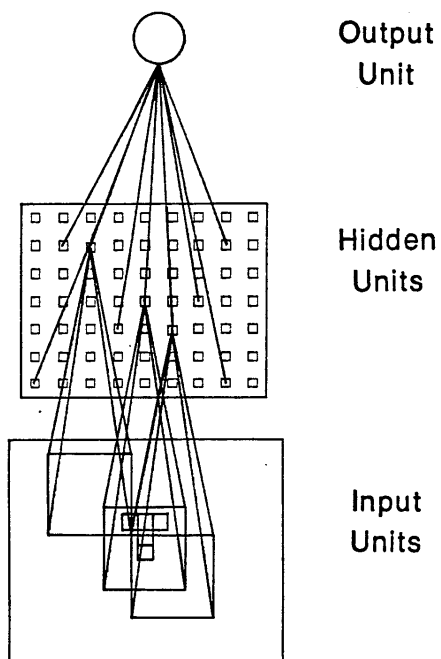
Neocognitron



Follows Hubel and Wiesel's results.

(Fukushima, 1980)

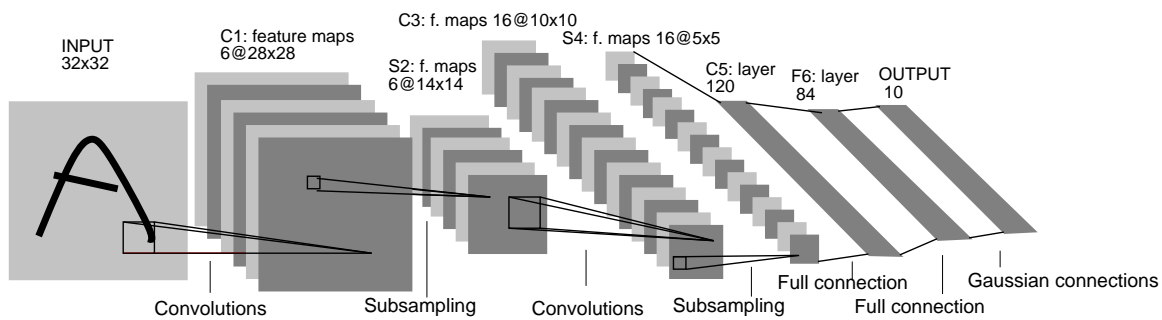
Network for the T-C problem



Trained with back-prop.

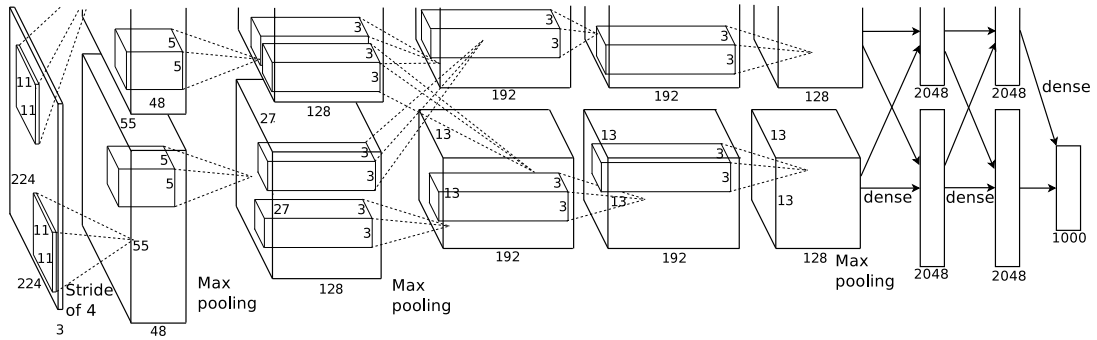
(Rumelhart et al., 1988)

LeNet-5



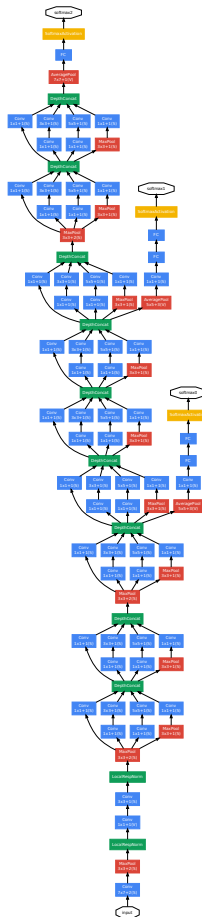
(leCun et al., 1998)

AlexNet

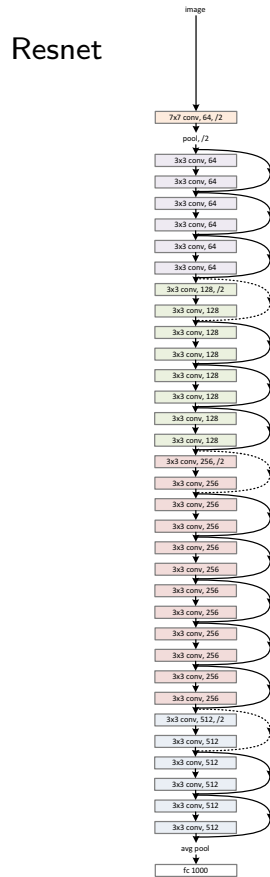


(Krizhevsky et al., 2012)

GoogLeNet



(Szegedy et al., 2015)



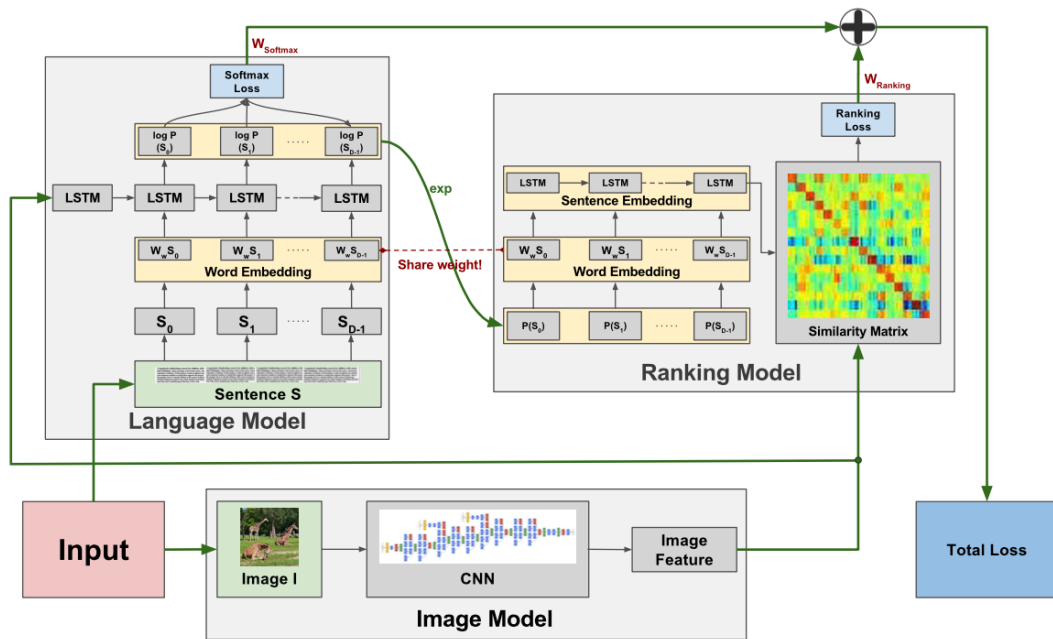
(He et al., 2015)

Deep learning is built on a natural generalization of a neural network: **a graph of tensor operators**, taking advantage of

- the chain rule (aka “back-propagation”),
- stochastic gradient decent,
- convolutions,
- parallel operations on GPUs.

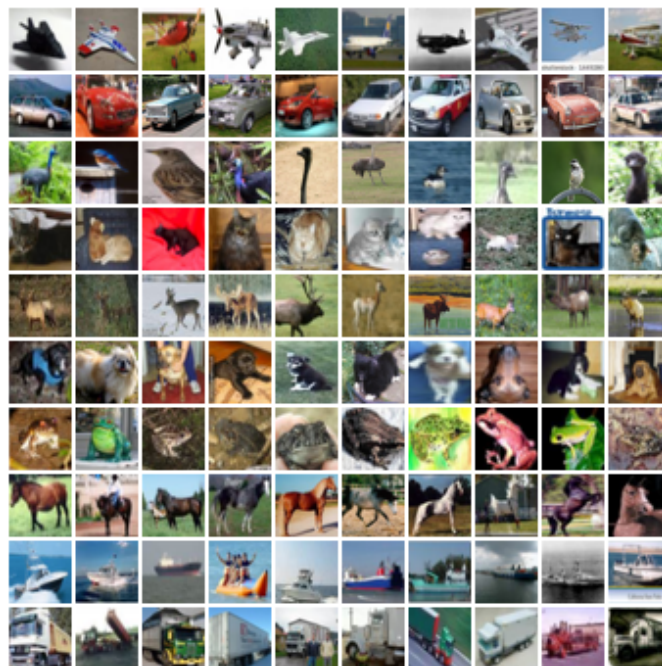
This does not differ much from networks from the 90s

This generalization allows to design complex networks of operators dealing with images, sound, text, sequences, etc. and to train them end-to-end.



(Yeung et al., 2015)

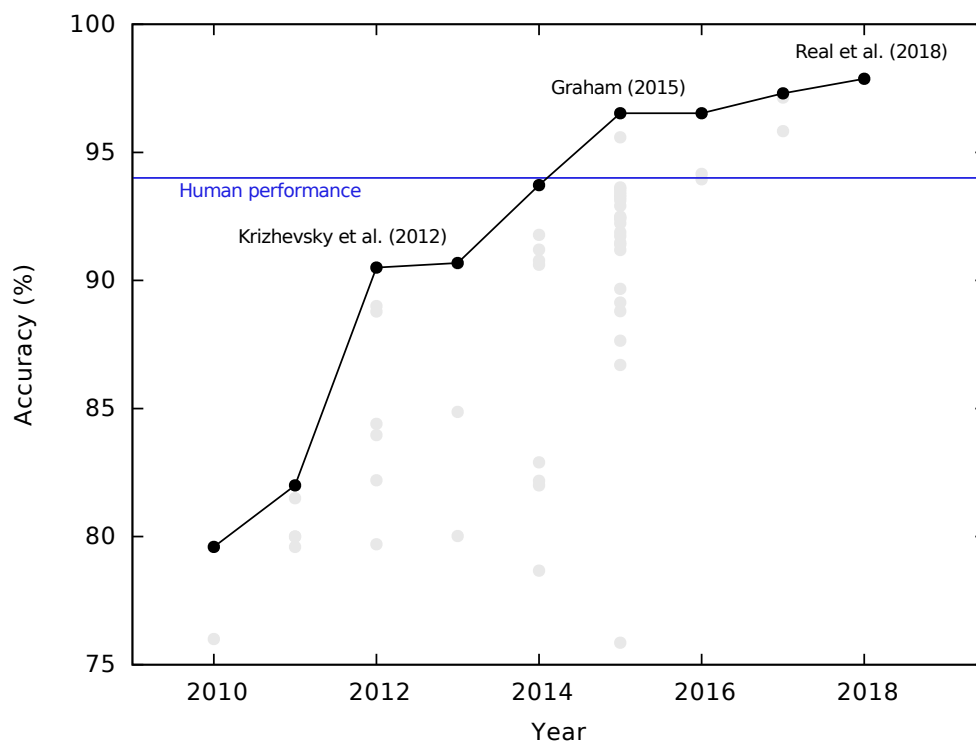
CIFAR10



32 × 32 color images, 50k train samples, 10k test samples.

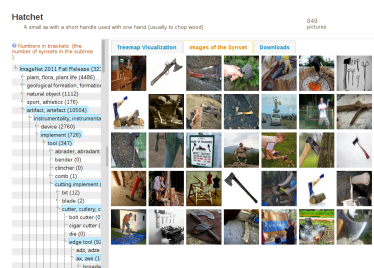
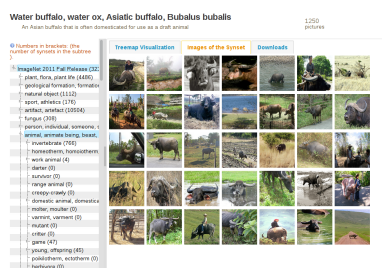
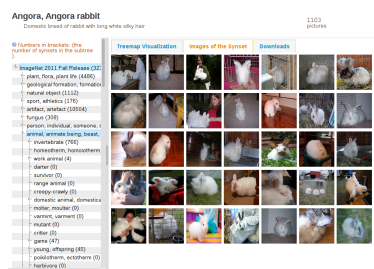
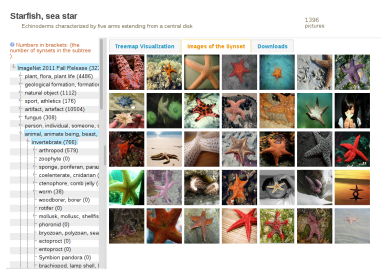
(Krizhevsky, 2009, chap. 3)

Performance on CIFAR10



ImageNet Large Scale Visual Recognition Challenge.

1000 categories, > 1M images



(<http://image-net.org/challenges/LSVRC/2014/browse-synsets>)

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43 [†]
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	19.38	4.49

Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except [†] reported on the test set).

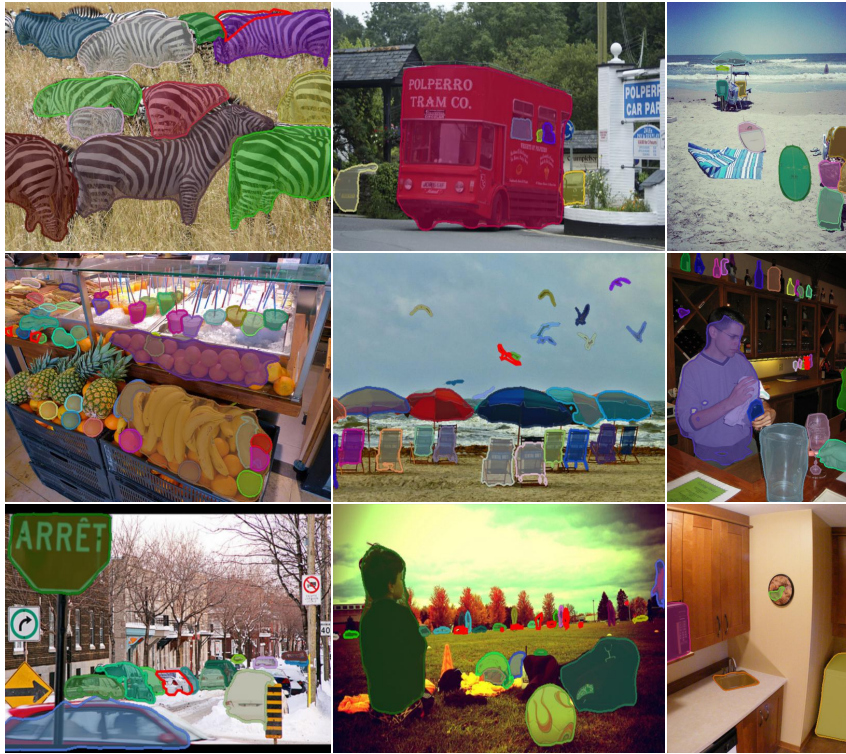
method	top-5 err. (test)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	3.57

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

(He et al., 2015)

1.2. Current applications and success

Object detection and segmentation



(Pinheiro et al., 2016)

Human pose estimation



(Wei et al., 2016)

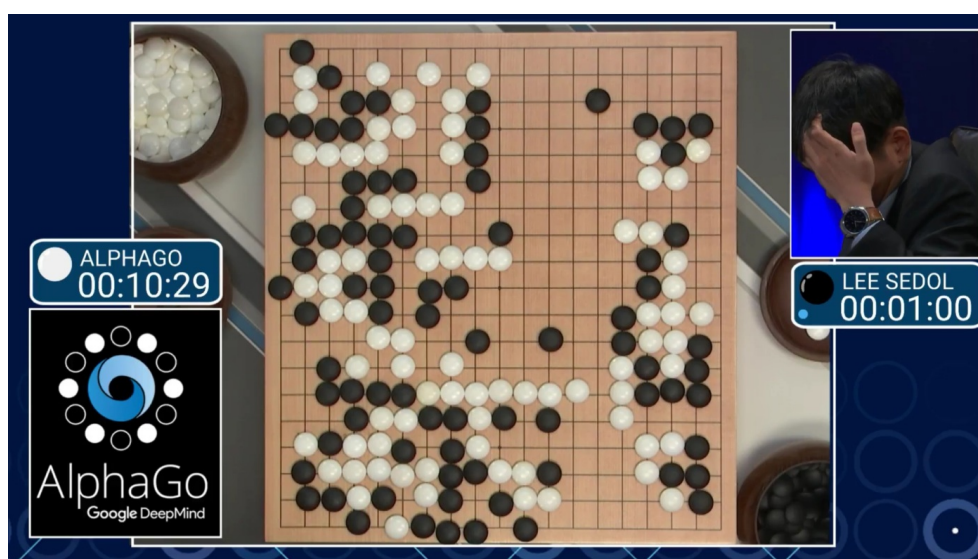
Reinforcement learning



Self-trained, plays 49 games at human level.

(Mnih et al., 2015)

Strategy games



March 2016, 4-1 against a 9-dan professional without handicap.

(Silver et al., 2016)

Translation

“The reason Boeing are doing this is to cram more seats in to make their plane more competitive with our products,” said Kevin Keniston, head of passenger comfort at Europe’s Airbus.

- “La raison pour laquelle Boeing fait cela est de créer plus de sièges pour rendre son avion plus compétitif avec nos produits”, a déclaré Kevin Keniston, chef du confort des passagers chez Airbus.

When asked about this, an official of the American administration replied: “The United States is not conducting electronic surveillance aimed at offices of the World Bank and IMF in Washington.”

- Interrogé à ce sujet, un fonctionnaire de l’administration américaine a répondu: “Les États-Unis n’effectuent pas de surveillance électronique à l’intention des bureaux de la Banque mondiale et du FMI à Washington”

(Wu et al., 2016)

Auto-captioning

A person riding a motorcycle on a dirt road.



Two dogs play in the grass.



A group of young people playing a game of frisbee.



Two hockey players are fighting over the puck.



A herd of elephants walking across a dry grass field.



A close up of a cat laying on a couch.



(Vinyals et al., 2015)

Question answering

I: Jane went to the hallway.
I: Mary walked to the bathroom.
I: Sandra went to the garden.
I: Daniel went back to the garden.
I: Sandra took the milk there.
Q: Where is the milk?
A: garden

I: It started boring, but then it got interesting.
Q: What's the sentiment?
A: positive

(Kumar et al., 2015)

Image generation



(Brock et al., 2018)

Text generation

System Prompt (human-written)

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Model Completion (machine-written, 10 tries)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

(Radford et al., 2019)

Why does it work now?

The success of deep learning is multi-factorial:

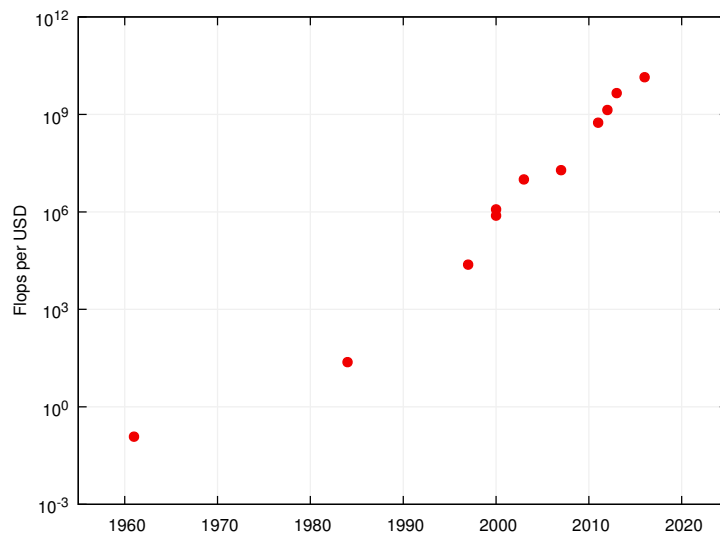
- Five decades of research in machine learning,
- CPUs/GPUs/storage developed for other purposes,
- lots of data from “the internet” ,
- tools and culture of collaborative and reproducible science,
- resources and efforts from large corporations.

Five decades of research in ML provided

- a taxonomy of ML concepts (classification, generative models, clustering, kernels, linear embeddings, etc.),
- a sound statistical formalization (Bayesian estimation, PAC),
- a clear picture of fundamental issues (bias/variance dilemma, VC dimension, generalization bounds, etc.),
- a good understanding of optimization issues,
- efficient large-scale algorithms.

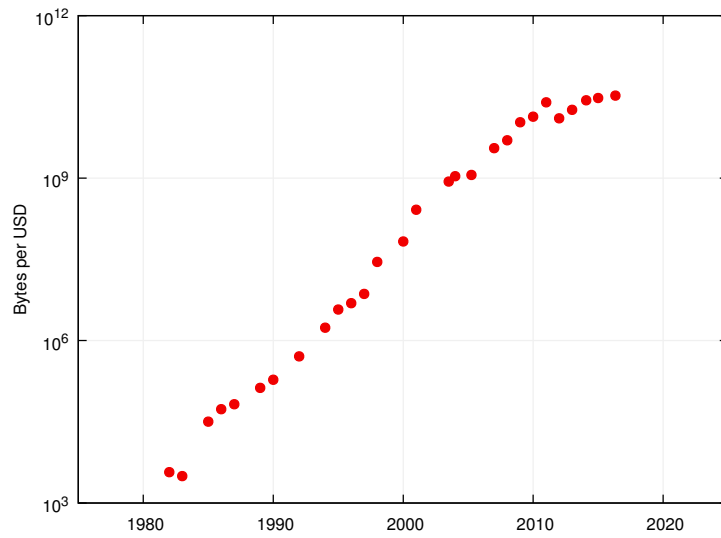
From a practical perspective, deep learning

- lessens the need for a deep mathematical grasp,
- makes the design of large learning architectures a system/software development task,
- allows to leverage modern hardware (clusters of GPUs),
- does not plateau when using more data,
- makes large trained networks a commodity.



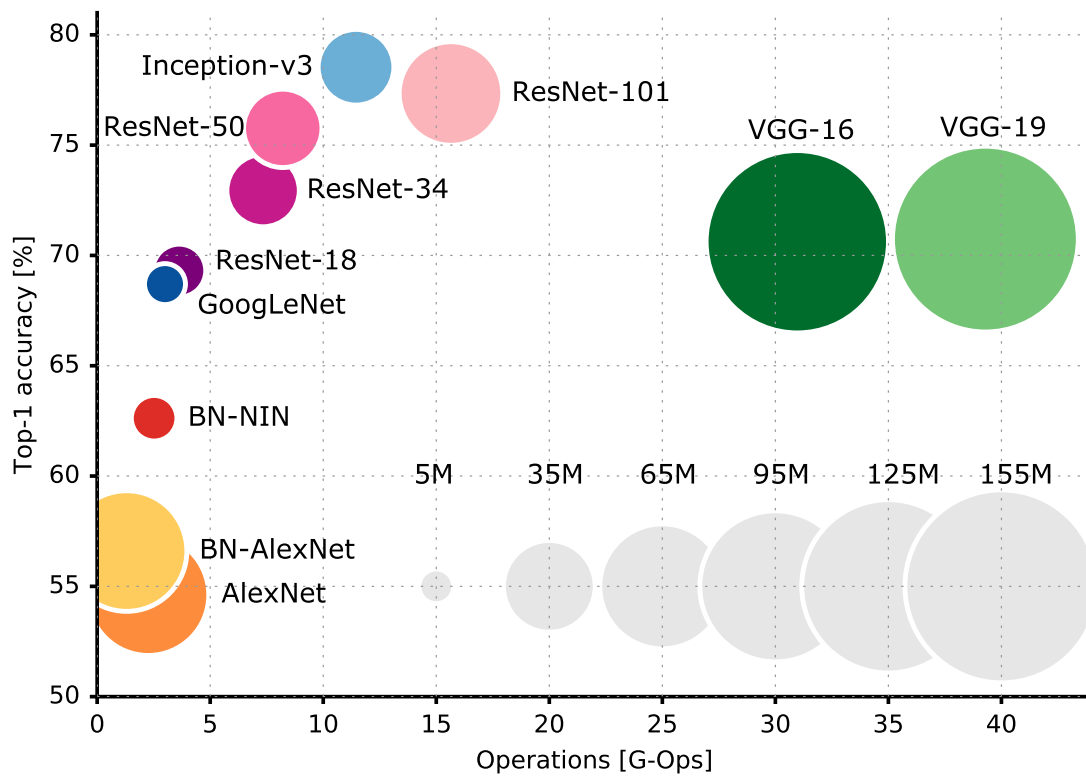
(Wikipedia “FLOPS”)

	TFlops (10^{12})	Price	GFlops per \$
Intel i7-6700K	0.2	\$344	0.6
AMD Radeon R-7 240	0.5	\$55	9.1
NVIDIA GTX 750 Ti	1.3	\$105	12.3
AMD RX 480	5.2	\$239	21.6
NVIDIA GTX 1080	8.9	\$699	12.7



(John C. McCallum)

The typical cost of a 4Tb hard disk is \$120 (Dec 2016).



(Canziani et al., 2016)

Data-set	Year	Nb. images	Resolution	Nb. classes
MNIST	1998	6.0×10^4	28×28	10
NORB	2004	4.8×10^4	96×96	5
Caltech 101	2003	9.1×10^3	$\simeq 300 \times 200$	101
Caltech 256	2007	3.0×10^4	$\simeq 640 \times 480$	256
LFW	2007	1.3×10^4	250×250	–
CIFAR10	2009	6.0×10^4	32×32	10
PASCAL VOC	2012	2.1×10^4	$\simeq 500 \times 400$	20
MS-COCO	2015	2.0×10^5	$\simeq 640 \times 480$	91
ImageNet	2016	14.2×10^6	$\simeq 500 \times 400$	21,841
Cityscape	2016	25×10^3	$2,000 \times 1000$	30

“Quantity has a Quality All Its Own.”

(Thomas A. Callaghan Jr.)

Implementing a deep network, PyTorch

Deep-learning development is usually done in a framework:

	Language(s)	License	Main backer
PyTorch	Python	BSD	Facebook
Caffe2	C++, Python	Apache	Facebook
TensorFlow	Python, C++	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon
CNTK	Python, C++	MIT	Microsoft
Torch	Lua	BSD	Facebook
Theano	Python	BSD	U. of Montreal
Caffe	C++	BSD 2 clauses	U. of CA, Berkeley

A fast, low-level, compiled backend to access computation devices, combined with a slow, high-level, interpreted language.

We will use the PyTorch framework for our experiments.



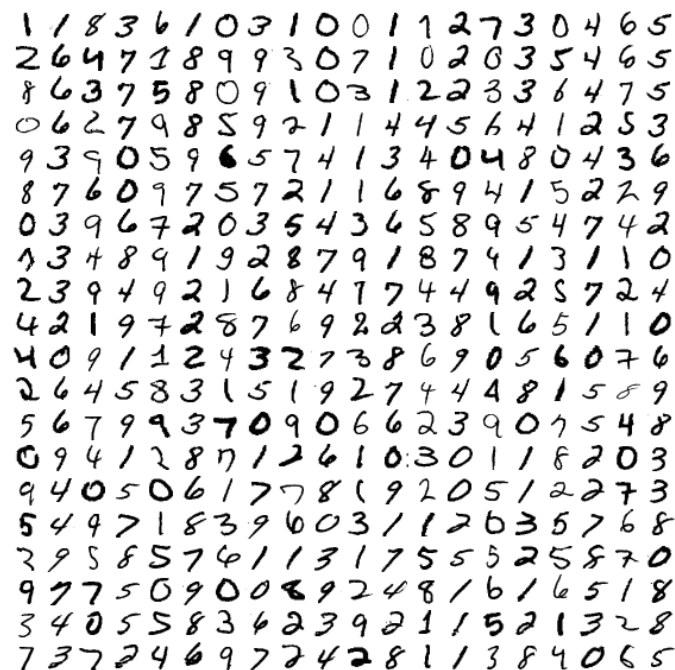
<http://pytorch.org>

"PyTorch is a python package that provides two high-level features:

- *Tensor computation (like numpy) with strong GPU acceleration*
- *Deep Neural Networks built on a tape-based autograd system*

You can reuse your favorite python packages such as numpy, scipy and Cython to extend PyTorch when needed."

MNIST data-set



28 × 28 grayscale images, 60k train samples, 10k test samples.

(LeCun et al., 1998)

```

model = nn.Sequential(
    nn.Conv2d( 1, 32, 5), nn.MaxPool2d(3), nn.ReLU(),
    nn.Conv2d(32, 64, 5), nn.MaxPool2d(2), nn.ReLU(),
    Flatten(),
    nn.Linear(256, 200), nn.ReLU(),
    nn.Linear(200, 10)
)

nb_epochs, batch_size = 10, 100
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1)

model.to(device)
criterion.to(device)
train_input, train_target = train_input.to(device), train_target.to(device)

mu, std = train_input.mean(), train_input.std()
train_input.sub_(mu).div_(std)

for e in range(nb_epochs):
    for input, target in zip(train_input.split(batch_size),
                             train_target.split(batch_size)):
        output = model(input)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

$\simeq 7s$ on a GTX1080, $\simeq 1\%$ test error

1.4. Tensor basics and linear regression

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- A 0d tensor is a scalar,
- A 1d tensor is a vector (e.g. a sound sample),
- A 2d tensor is a matrix (e.g. a grayscale image),
- A 3d tensor can be seen as a vector of identically sized matrix (e.g. a multi-channel image),
- A 4d tensor can be seen as a matrix of identically sized matrix, or a sequence of 3d tensors (e.g. a sequence of multi-channel images),
- etc.

Tensors are used to encode the signal to process, but also the internal states and parameters of models.

Manipulating data through this constrained structure allows to use CPUs and GPUs at peak performance.

Compounded data structures can represent more diverse data types.

PyTorch is a Python library built on top of Torch's THNN computational backend.

Its main features are:

- Efficient tensor operations on CPU/GPU,
- automatic on-the-fly differentiation (autograd),
- optimizers,
- data I/O.

“Efficient tensor operations” encompass both standard linear algebra and, as we will see later, deep-learning specific operations (convolution, pooling, etc.)

A key specificity of PyTorch is the central role of autograd to compute derivatives of *anything!* We will come back to this.

```

>>> x = torch.empty(2, 5)
>>> x.size()
torch.Size([2, 5])
>>> x.fill_(1.125)
tensor([[ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250],
        [ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250]])
>>> x.mean()
tensor(1.1250)
>>> x.std()
tensor(0.)
>>> x.sum()
tensor(11.2500)
>>> x.sum().item()
11.25

```

In-place operations are suffixed with an underscore, and a 0d tensor can be converted back to a Python scalar with `item()`.



Reading a coefficient also generates a 0d tensor.

```

>>> x = torch.tensor([[11., 12., 13.], [21., 22., 23.]])
>>> x[1, 2]
tensor(23.)

```

PyTorch provides operators for component-wise and vector/matrix operations.

```

>>> x = torch.tensor([ 10., 20., 30.])
>>> y = torch.tensor([ 11., 21., 31.])
>>> x + y
tensor([ 21., 41., 61.])
>>> x * y
tensor([ 110., 420., 930.])
>>> x**2
tensor([ 100., 400., 900.])
>>> m = torch.tensor([[ 0., 0., 3. ],
...                  [ 0., 2., 0. ],
...                  [ 1., 0., 0. ]])
>>> m.mv(x)
tensor([ 90., 40., 10.])
>>> m @ x
tensor([ 90., 40., 10.])

```

And as in numpy, the `:` symbol defines a range of values for an index and allows to slice tensors.

```
>>> import torch
>>> x = torch.empty(2, 4).random_(10)
>>> x
tensor([[8., 1., 1., 3.],
        [7., 0., 7., 5.]])
>>> x[0]
tensor([8., 1., 1., 3.])
>>> x[0, :]
tensor([8., 1., 1., 3.])
>>> x[:, 0]
tensor([8., 7.])
>>> x[:, 1:3] = -1
>>> x
tensor([[ 8., -1., -1.,  3.],
        [ 7., -1., -1.,  5.]])
```

PyTorch provides interfacing to standard linear operations, such as linear system solving or Eigen-decomposition.

```
>>> y = torch.empty(3).normal_()
>>> y
tensor([ 0.0477,  0.8834, -1.5996])
>>> m = torch.empty(3, 3).normal_()
>>> q, _ = torch.gels(y, m)
>>> torch.mm(m, q)
tensor([[ 0.0477],
        [ 0.8834],
        [-1.5996]])
```

Example: linear regression

Given a list of points

$$(x_n, y_n) \in \mathbb{R} \times \mathbb{R}, \quad n = 1, \dots, N,$$

can we find the “best line”

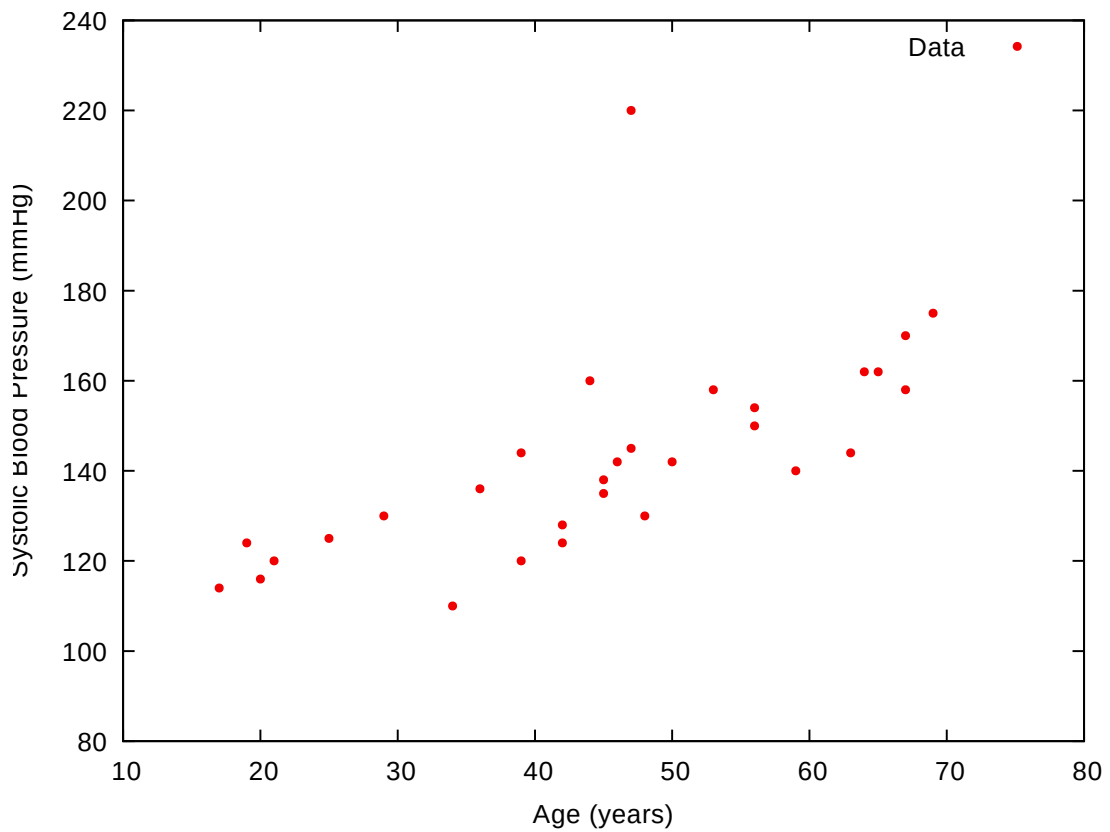
$$f(x; a, b) = ax + b$$

going “through the points”, e.g. minimizing the mean square error

$$\operatorname{argmin}_{a,b} \frac{1}{N} \sum_{n=1}^N \underbrace{(ax_n + b - y_n)}_{f(x_n; a, b)}^2.$$

Such a model would allow to predict the y associated to a new x , simply by calculating $f(x; a, b)$.

```
bash> cat systolic-blood-pressure-vs-age.dat
39 144
47 220
45 138
47 145
65 162
46 142
67 170
42 124
67 158
56 154
64 162
56 150
59 140
34 110
42 128
/.../
```



$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}} \quad \underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\alpha \in \mathbb{R}^{2 \times 1}} \simeq \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

```
import torch, numpy

data = torch.tensor(numpy.loadtxt('systolic-blood-pressure-vs-age.dat'))
nb_samples = data.size(0)

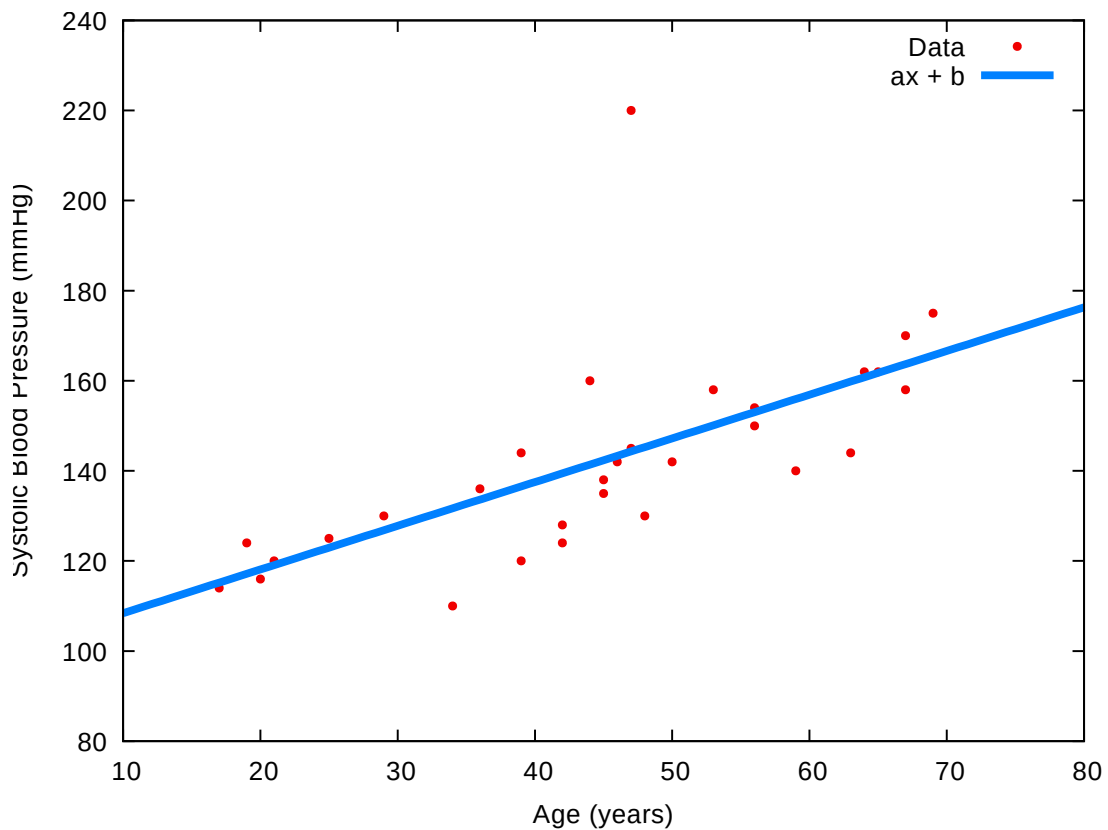
x, y = torch.empty(nb_samples, 2), torch.empty(nb_samples, 1)

x[:, 0] = data[:, 0]
x[:, 1] = 1

y[:, 0] = data[:, 1]

alpha, _ = torch.gels(y, x)

a, b = alpha[0, 0].item(), alpha[1, 0].item()
```



1.5. High dimension tensors

A tensor can be of several types:

- `torch.float16`, `torch.float32`, `torch.float64`,
- `torch.uint8`,
- `torch.int8`, `torch.int16`, `torch.int32`, `torch.int64`

and can be located either in the CPU's or in a GPU's memory.

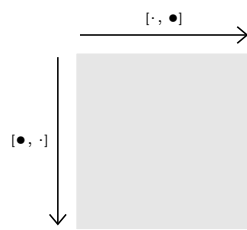
Operations with tensors stored in a certain device's memory are done by that device. We will come back to that later.

```

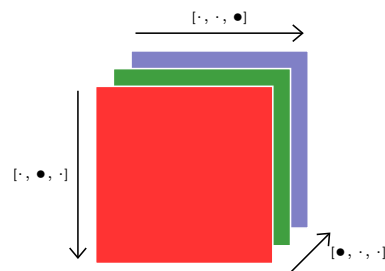
>>> x = torch.zeros(1, 3)
>>> x.dtype, x.device
(torch.float32, device(type='cpu'))
>>> x = x.long()
>>> x.dtype, x.device
(torch.int64, device(type='cpu'))
>>> x = x.to('cuda')
>>> x.dtype, x.device
(torch.int64, device(type='cuda', index=0))

```

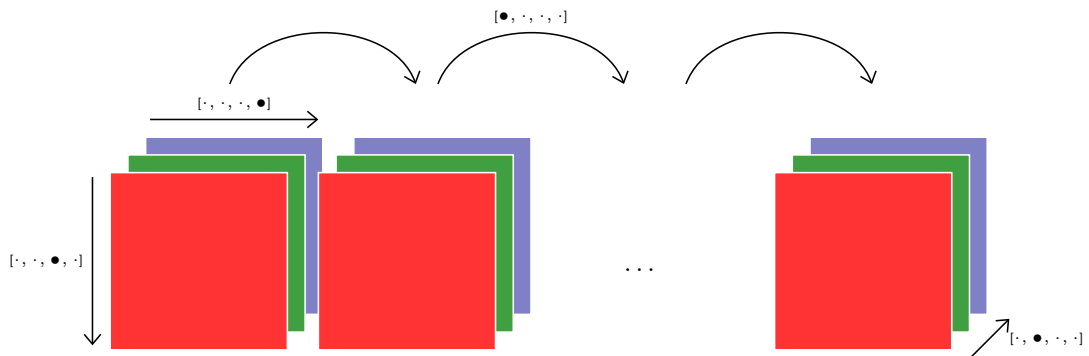
2d tensor (e.g. grayscale image)



3d tensor (e.g. rgb image)



4d tensor (e.g. sequence of rgb images)



Here are some examples from the vast library of tensor operations:

Creation

- `torch.empty(*size, ...)`
- `torch.zeros(*size, ...)`
- `torch.full(size, value, ...)`
- `torch.tensor(sequence, ...)`
- `torch.eye(n, ...)`
- `torch.from_numpy(ndarray)`

Indexing, Slicing, Joining, Mutating

- `torch.Tensor.view(*size)`
- `torch.cat(inputs, dimension=0)`
- `torch.chunk(tensor, chunks, dim=0)[source]`
- `torch.split(tensor, split_size, dim=0)[source]`
- `torch.index_select(input, dim, index, out=None)`
- `torch.t(input, out=None)`
- `torch.transpose(input, dim0, dim1, out=None)`

Filling

- `Tensor.fill_(value)`
- `torch.bernoulli_(proba)`
- `torch.normal_([mu, [std]])`

Pointwise math

- `torch.abs(input, out=None)`
- `torch.add()`
- `torch.cos(input, out=None)`
- `torch.sigmoid(input, out=None)`
- (+ many operators)

Math reduction

- `torch.dist(input, other, p=2, out=None)`
- `torch.mean()`
- `torch.norm()`
- `torch.std()`
- `torch.sum()`

BLAS and LAPACK Operations

- `torch.eig(a, eigenvectors=False, out=None)`
- `torch.gels(B, A, out=None)`
- `torch.inverse(input, out=None)`
- `torch.mm(mat1, mat2, out=None)`
- `torch.mv(mat, vec, out=None)`



```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



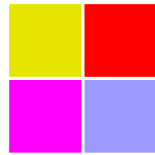
x.t()



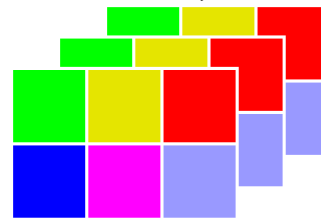
x.view(-1)



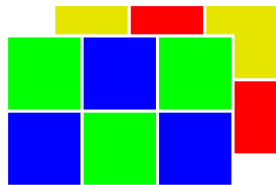
x.view(3, -1)



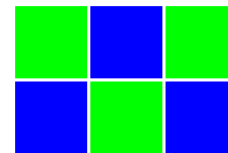
x.narrow(1, 1, 2)



x.view(1, 2, 3).expand(3, 2, 3)



```
x = torch.tensor([ [ [ 1, 2, 1 ],
                    [ 2, 1, 2 ] ],
                  [ [ 3, 0, 3 ],
                    [ 0, 3, 0 ] ] ])
```



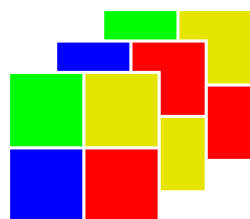
x.narrow(0, 0, 1)



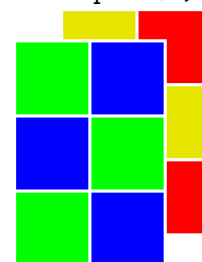
x.narrow(2, 0, 2)



x.transpose(0, 1)



x.transpose(0, 2)



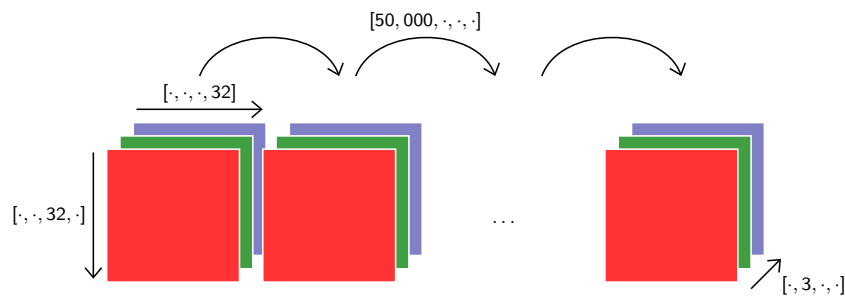
x.transpose(1, 2)

PyTorch offers simple interfaces to standard image data-bases.

```
import torch, torchvision
cifar = torchvision.datasets.CIFAR10('./cifar10/', train = True, download = True)
x = torch.from_numpy(cifar.train_data).transpose(1, 3).transpose(2, 3).float()
x = x / 255
print(x.type(), x.size(), x.min().item(), x.max().item())
```

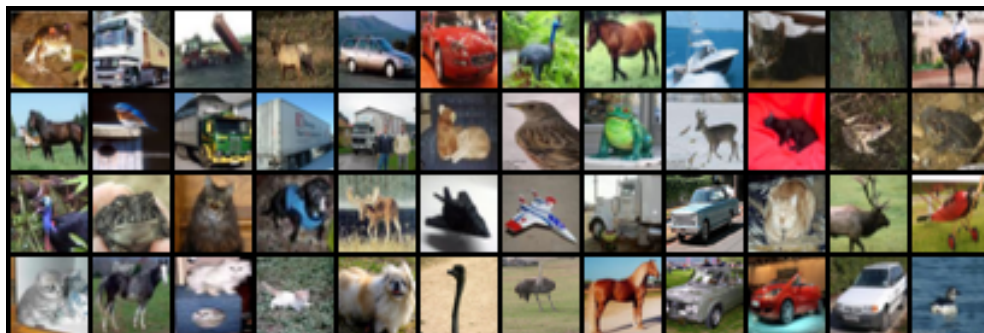
prints

```
Files already downloaded and verified
torch.FloatTensor torch.Size([50000, 3, 32, 32]) 0.0 1.0
```

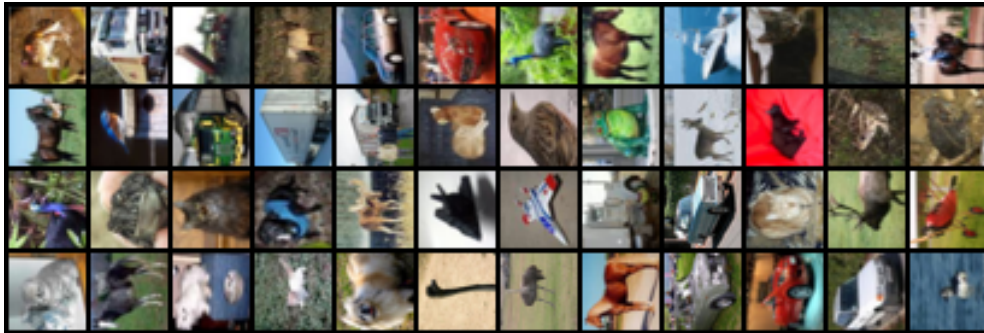


```
# Narrows to the first images, converts to float
x = x.narrow(0, 0, 48).float()

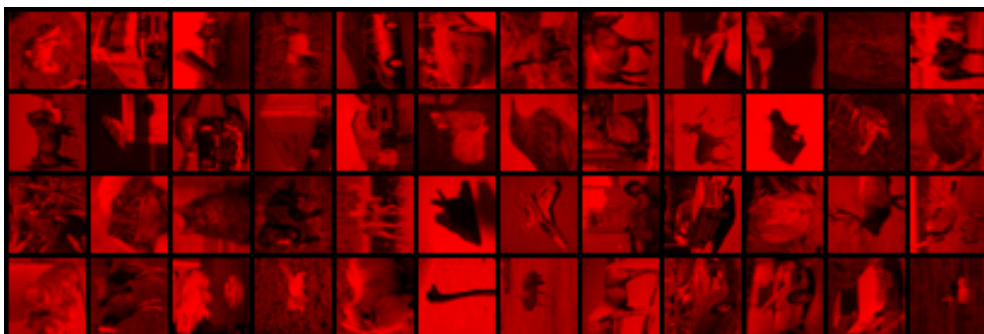
# Saves these samples as a single image
torchvision.utils.save_image(x, 'cifar-4x12.png', nrow = 12)
```



```
# Switches the row and column indexes
x.transpose_(2, 3)
torchvision.utils.save_image(x, 'cifar-4x12-rotated.png', nrow = 12)
```



```
# Kills the green and blue channels
x.narrow(1, 1, 2).fill_(0)
torchvision.utils.save_image(x, 'cifar-4x12-rotated-and-red.png', nrow = 12)
```



Broadcasting

Broadcasting automatically expands dimensions by replicating coefficients, when it is necessary to perform operations that are “intuitively reasonable”.

For instance:

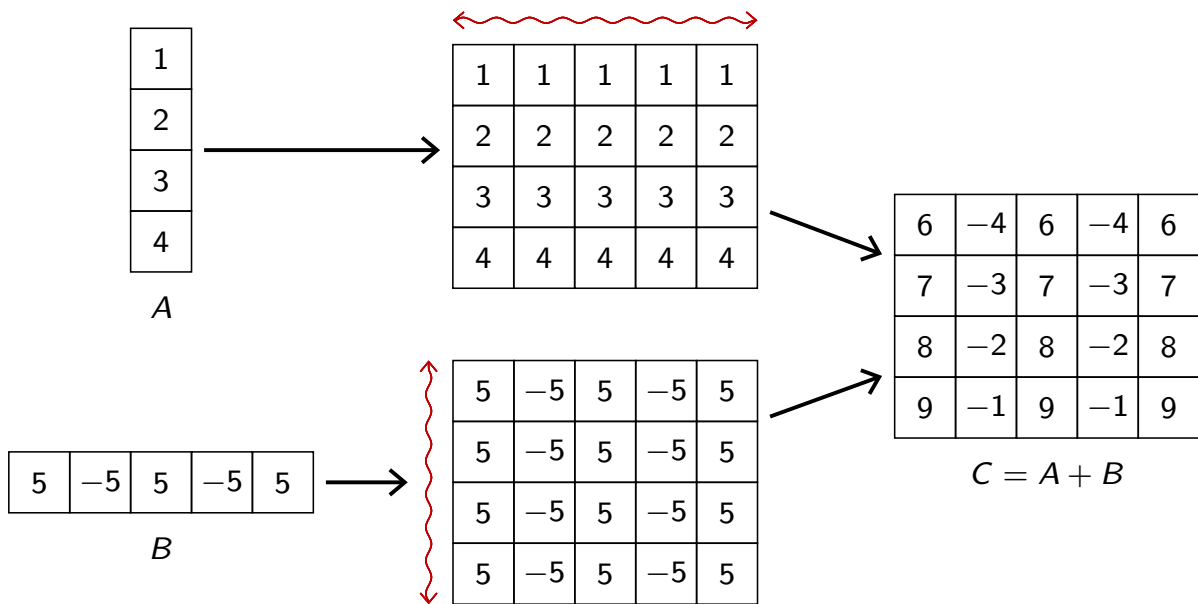
```
>>> x = torch.empty(100, 4).normal_(2)
>>> x.mean(0)
tensor([2.0476, 2.0133, 1.9109, 1.8588])
>>> x -= x.mean(0) # This should not work!
>>> x.mean(0)
tensor([-4.0531e-08, -4.4703e-07, -1.3471e-07,  3.5763e-09])
```

Precisely, broadcasting proceeds as follows:

1. If one of the tensors has fewer dimensions than the other, it is reshaped by adding as many dimensions of size 1 as necessary in the front; then
2. for every dimension mismatch, **if one of the two tensors is of size one**, it is expanded along this axis by replicating coefficients.

If there is a tensor size mismatch for one of the dimension and neither of them is one, the operation fails.

```
A = torch.tensor([[1.], [2.], [3.], [4.]])
B = torch.tensor([[5., -5., 5., -5., 5.]])
C = A + B
```



Broadcasted

2.1. Loss and risk

The general objective of machine learning is to capture regularity in data to make predictions.

In our regression example, we modeled age and blood pressure as being linearly related, to predict the latter from the former.

There are multiple types of inference that we can roughly split into three categories:

- Classification (e.g. object recognition, cancer detection, speech processing),
- regression (e.g. customer satisfaction, stock prediction, epidemiology), and
- density estimation (e.g. outlier detection, data visualization, sampling/synthesis).

The standard formalization considers a measure of probability

$$\mu_{X,Y}$$

over the observation/value of interest, and i.i.d. training samples

$$(x_n, y_n), \quad n = 1, \dots, N.$$

Intuitively, for classification it can often be interpreted as

$$\mu_{X,Y}(x, y) = \mu_{X|Y=y}(x) P(Y = y)$$

that is, draw Y first, and given its value, generate X .

Here

$$\mu_{X|Y=y}$$

stands for the population of the observable signal for class y (e.g. “sound of an /ē/”, “image of a cat”).

For regression, one would interpret the joint law more naturally as

$$\mu_{X,Y}(x, y) = \mu_{Y|X=x}(y) \mu_X(x)$$

which would be: first, generate X , and given its value, generate Y .

In the simple cases

$$Y = f(X) + \epsilon$$

where f is the deterministic dependency between x and y , and ϵ is a random noise, independent of X .

With such a model, we can more precisely define the three types of inferences we introduced before:

Classification,

- (X, Y) random variables on $\mathcal{X} = \mathbb{R}^D \times \{1, \dots, C\}$,
- we want to estimate $\operatorname{argmax}_y P(Y = y | X = x)$.

Regression,

- (X, Y) random variables on $\mathcal{X} = \mathbb{R}^D \times \mathbb{R}$,
- we want to estimate $\mathbb{E}(Y | X = x)$.

Density estimation,

- X random variable on $\mathcal{X} = \mathbb{R}^D$,
- we want to estimate μ_X .

The boundaries between these categories are fuzzy:

- Regression allows to do classification through class scores.
- Density models allow to do classification thanks to Bayes' law.

etc.

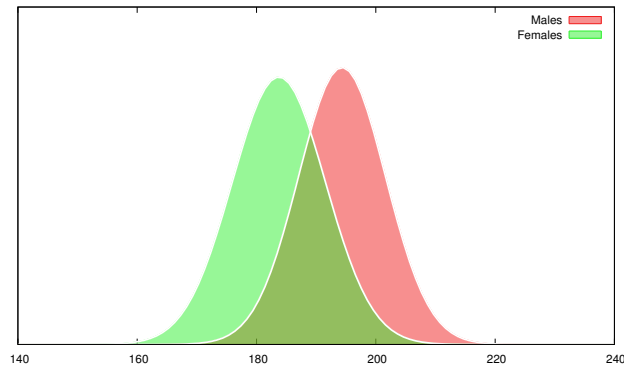
We call **generative** classification methods with an explicit data model, and **discriminative** the ones bypassing such a modeling .

Example: Can we predict a Brazilian basketball player's gender G from his/her height H ?

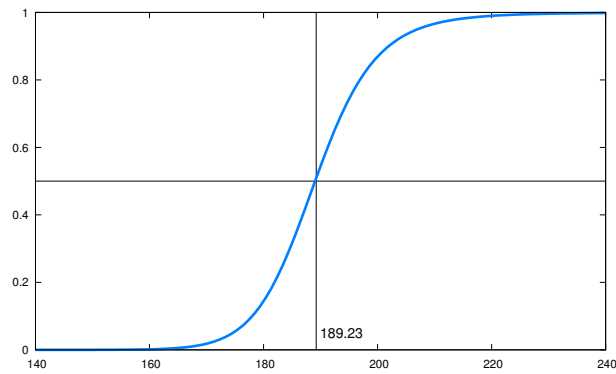
Females: 190 182 188 184 196 173 180 193 179 186 185 169

Males: 192 190 183 199 200 190 195 184 190 203 205 201

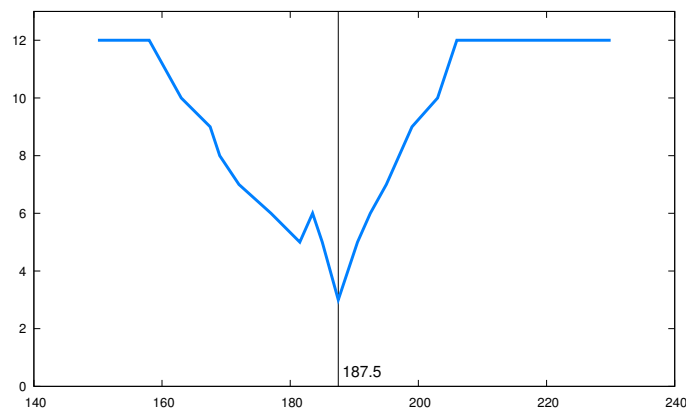
In the **generative** approach, we model $\mu_{H|G=g}(h)$



and use Bayes's law $P(G = g | H = h) = \frac{\mu_{H|G=g}(h)P(G=g)}{\mu_H(h)}$



In the **discriminative** approach we directly pick the threshold that works the best on the data:



Note that it is harder to design a confidence indicator.

Risk, empirical risk

Learning consists of finding in a set \mathcal{F} of functionals a “good” f^* (or its parameters’ values) usually defined through a loss

$$\ell : \mathcal{F} \times \mathcal{Z} \rightarrow \mathbb{R}$$

such that $\ell(f, z)$ increases with how wrong f is on z . For instance

- for classification:

$$\ell(f, (x, y)) = \mathbf{1}_{\{f(x) \neq y\}},$$

- for regression:

$$\ell(f, (x, y)) = (f(x) - y)^2,$$

- for density estimation:

$$\ell(q, z) = -\log q(z).$$

The loss may include additional terms related to f itself.

We are looking for an f with a small **expected risk**

$$R(f) = \mathbb{E}_Z (\ell(f, Z)),$$

which means that our learning procedure would ideally choose

$$f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} R(f).$$

Although this quantity is unknown, if we have i.i.d. training samples

$$\mathcal{D} = \{Z_1, \dots, Z_N\},$$

we can compute an estimate, the **empirical risk**:

$$\hat{R}(f; \mathcal{D}) = \hat{\mathbb{E}}_{\mathcal{D}}(\ell(f, Z)) = \frac{1}{N} \sum_{n=1}^N \ell(f, Z_n).$$

We have

$$\begin{aligned} \mathbb{E}_{Z_1, \dots, Z_N} (\hat{R}(f; \mathcal{D})) &= \mathbb{E}_{Z_1, \dots, Z_N} \left(\frac{1}{N} \sum_{n=1}^N \ell(f, Z_n) \right) \\ &= \frac{1}{N} \sum_{n=1}^N \mathbb{E}_{Z_n} (\ell(f, Z_n)) \\ &= \frac{1}{N} \sum_{n=1}^N \mathbb{E}_Z (\ell(f, Z)) \\ &= \mathbb{E}_Z (\ell(f, Z)) \\ &= R(f). \end{aligned}$$

The empirical risk is an **unbiased estimator** of the expected risk.

Finally, given \mathcal{D} , \mathcal{F} , and ℓ , “learning” aims at computing

$$f^* = \operatorname{argmin}_{f \in \mathcal{F}} \hat{R}(f; \mathcal{D}).$$

- Can we bound $R(f)$ with $\hat{R}(f; \mathcal{D})$?

Yes if f is not chosen using \mathcal{D} . Since the Z_n are independent, we just need to take into account the variance of $\hat{R}(f; \mathcal{D})$.

- Can we bound $R(f^*)$ with $\hat{R}(f^*; \mathcal{D})$?



Unfortunately not simply, and not without additional constraints on \mathcal{F} .

For instance if $|\mathcal{F}| = 1$, we can!

Note that in practice, we call “loss” both the functional

$$\ell : \mathcal{F} \times \mathcal{Z} \rightarrow \mathbb{R}$$

and the empirical risk minimized during training

$$\mathcal{L}(f) = \frac{1}{N} \sum_{n=1}^N \ell(f, z_n).$$

2.2. Over and under fitting

You want to hire someone, and you evaluate candidates by asking them ten technical yes/no questions.

Would you feel confident if you interviewed one candidate and he makes a perfect score?

What about interviewing ten candidates and picking the best? What about interviewing one thousand?

With

$$Q_k^n \sim \mathcal{B}(0.5), \quad n = 1, \dots, 1000, \quad k = 1, \dots, 10,$$

independent standing for “candidate n answers question k correctly”, we have

$$\forall n, P(\forall k, Q_k^n = 1) = \frac{1}{1024}$$

and

$$P(\exists n, \forall k, Q_k^n = 1) \simeq 0.62.$$

So there is 62% chance that among 1,000 candidates answering completely at random, one will score perfectly.

Selecting a candidate based on a statistical estimator biases the said estimator for that candidate. And you need a greater number of “competence checks” if you have a larger pool of candidates.

Over and under-fitting, capacity. K -nearest-neighbors

A simple classification procedure is the “ K -nearest neighbors.”

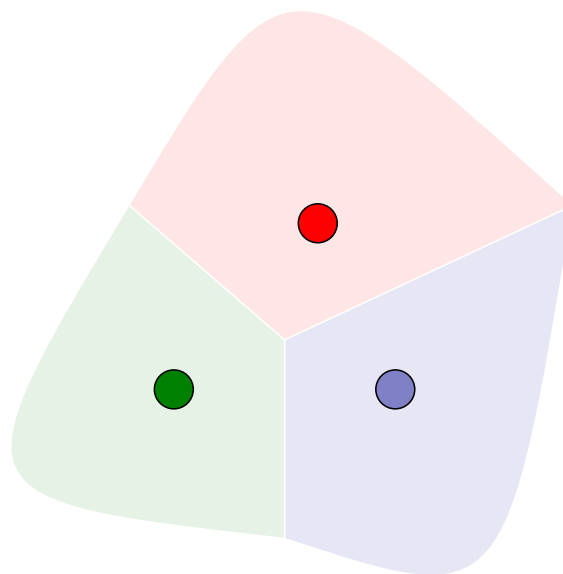
Given

$$(x_n, y_n) \in \mathbb{R}^D \times \{1, \dots, C\}, \quad n = 1, \dots, N$$

to predict the y associated to a new x , take the y_n of the closest x_n :

$$\begin{aligned} n^*(x) &= \operatorname{argmin}_n \|x_n - x\| \\ f^*(x) &= y_{n^*(x)}. \end{aligned}$$

This recipe corresponds to $K = 1$, and makes the empirical training error zero.

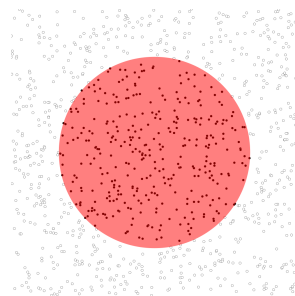


$K = 1$

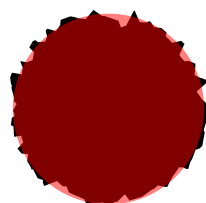
Under mild assumptions of regularities of $\mu_{X,Y}$, for $N \rightarrow \infty$ the asymptotic error rate of the 1-NN is less than twice the (optimal!) Bayes' Error rate.

It can be made more stable by looking at the $K > 1$ closest training points, and taking the majority vote.

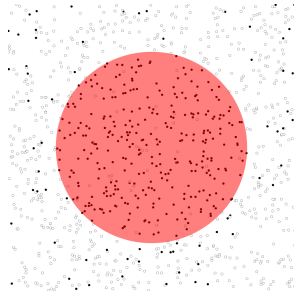
If we let also $K \rightarrow \infty$ "not too fast", the error rate is the (optimal!) Bayes' Error rate.



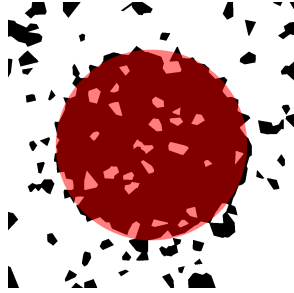
Training set



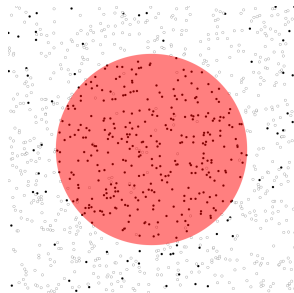
Prediction (K=1)



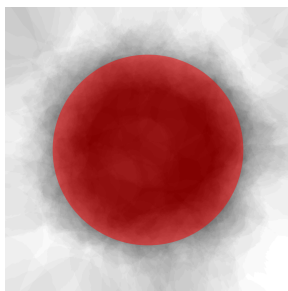
Training set



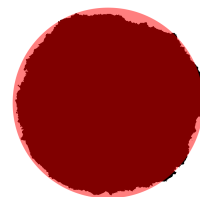
Prediction (K=1)



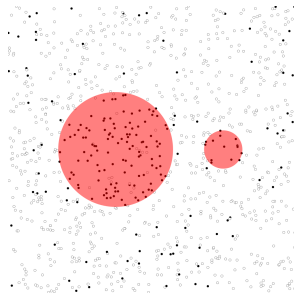
Training set



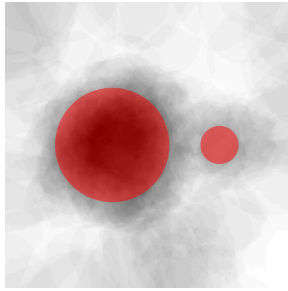
Votes (K=51)



Prediction (K=51)



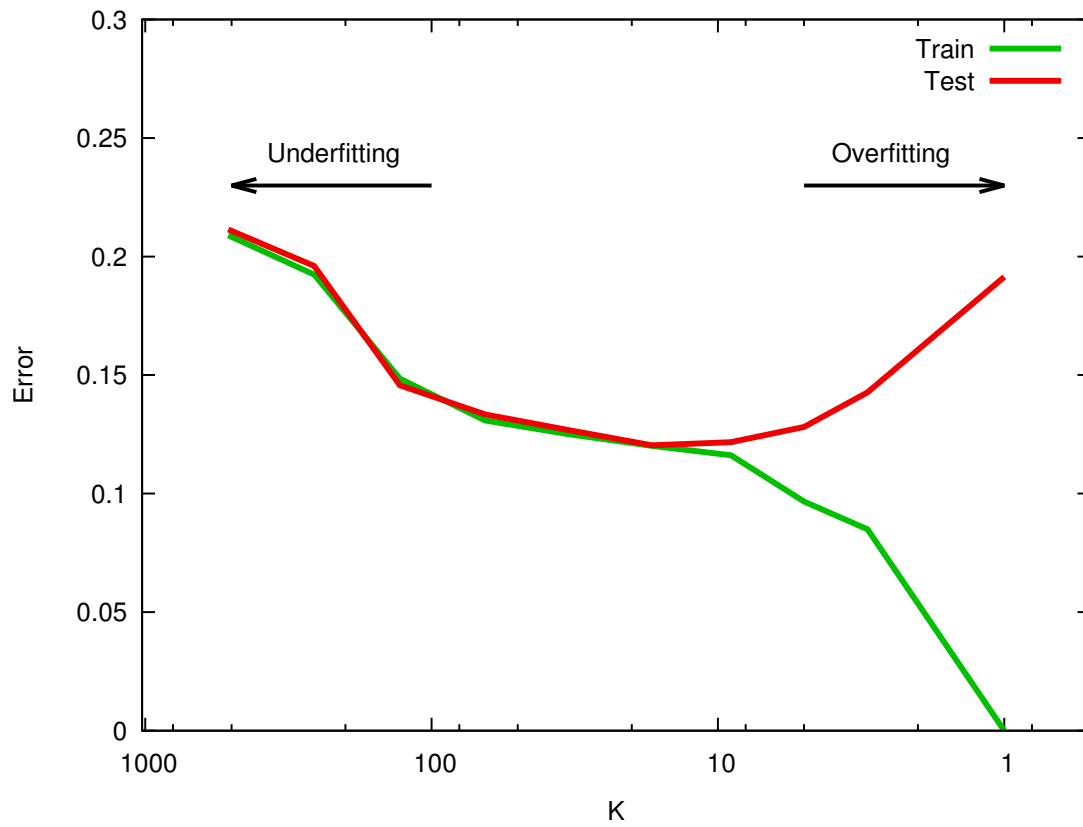
Training set



Votes (K=51)



Prediction (K=51)



Over and under-fitting, capacity, polynomials

Given a polynomial model

$$\forall x, \alpha_0, \dots, \alpha_D \in \mathbb{R}, f(x; \alpha) = \sum_{d=0}^D \alpha_d x^d.$$

and training points $(x_n, y_n) \in \mathbb{R}^2, n = 1, \dots, N$, the quadratic loss is

$$\begin{aligned} \mathcal{L}(\alpha) &= \sum_n (f(x_n; \alpha) - y_n)^2 \\ &= \sum_n \left(\sum_{d=0}^D \alpha_d x_n^d - y_n \right)^2 \\ &= \left\| \begin{pmatrix} x_1^0 & \dots & x_1^D \\ \vdots & & \vdots \\ x_N^0 & \dots & x_N^D \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_D \end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \right\|^2. \end{aligned}$$

Hence, minimizing this loss is a standard quadratic problem, for which we have efficient algorithms.

$$\operatorname{argmin}_{\alpha} \left\| \begin{pmatrix} x_1^0 & \dots & x_1^D \\ \vdots & & \vdots \\ x_N^0 & \dots & x_N^D \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_D \end{pmatrix} - \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \right\|^2$$

```
def fit_polynomial(D, x, y):
    X = torch.empty(x.size(0), D + 1)
    for d in range(D + 1):
        X[:, d] = x.pow(d)

    # gels expects a matrix for target
    Y = y.view(-1, 1)

    # LAPACK's GEneralized Least-Square
    alpha, _ = torch.gels(Y, X)

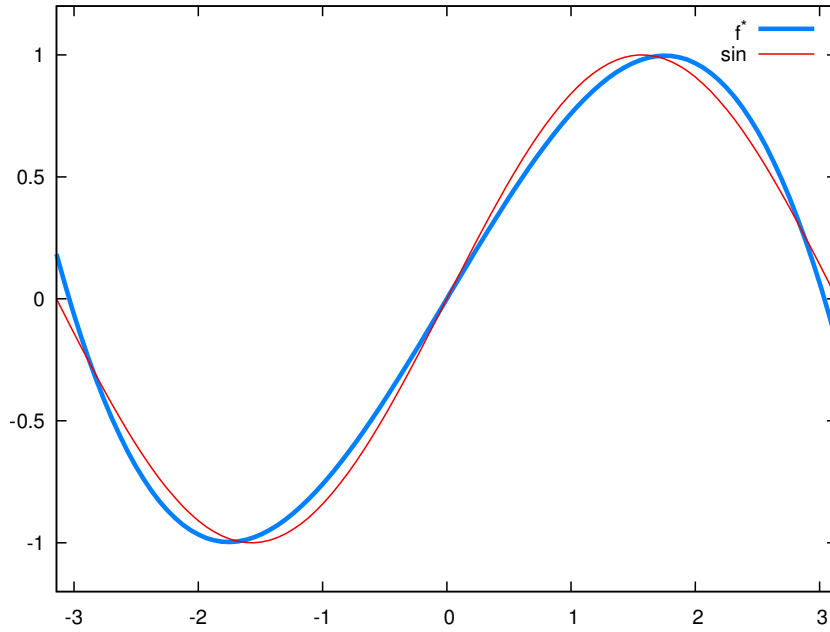
    return alpha[:D+1, 0]
```

```
D, N = 4, 100
x = torch.linspace(-math.pi, math.pi, N)
y = x.sin()
alpha = fit_polynomial(D, x, y)

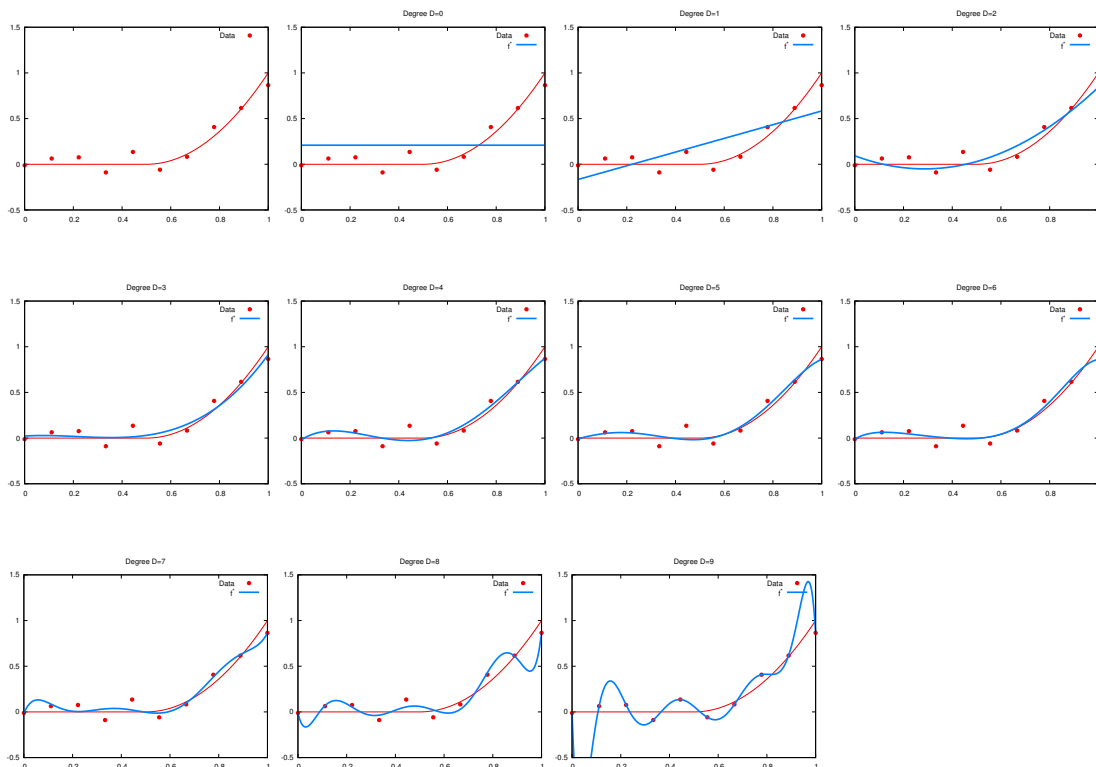
X = torch.empty(N, D + 1)
for d in range(D + 1):
    X[:, d] = x.pow(d)

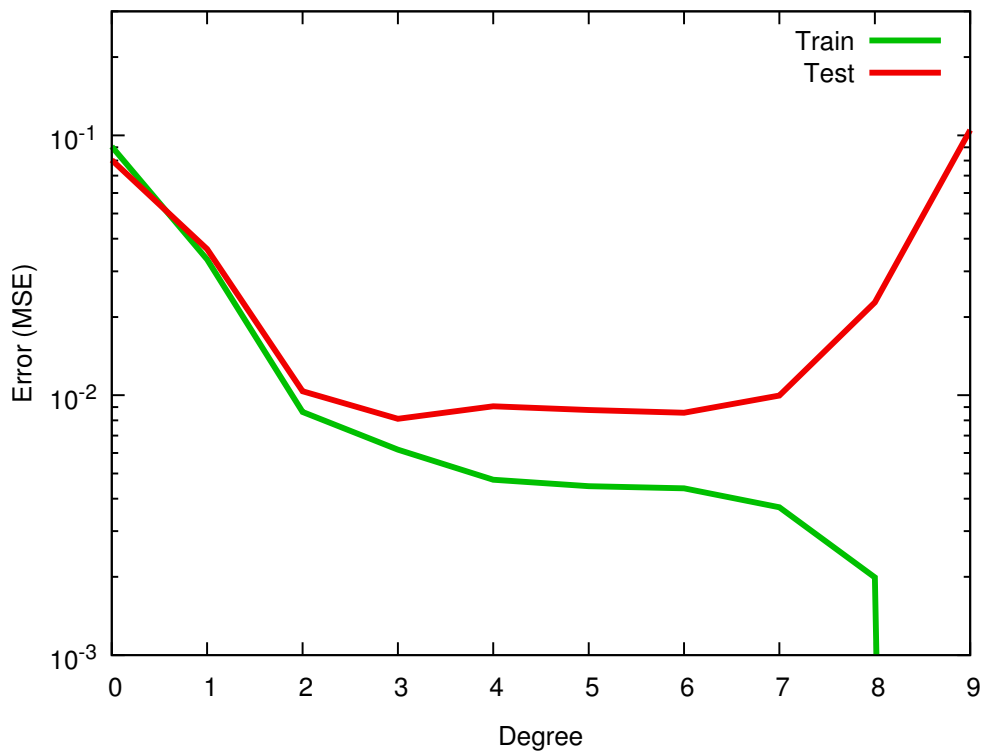
yhat = X.mv(alpha)

for k in range(N):
    print(x[k].item(), y[k].item(), yhat[k].item())
```

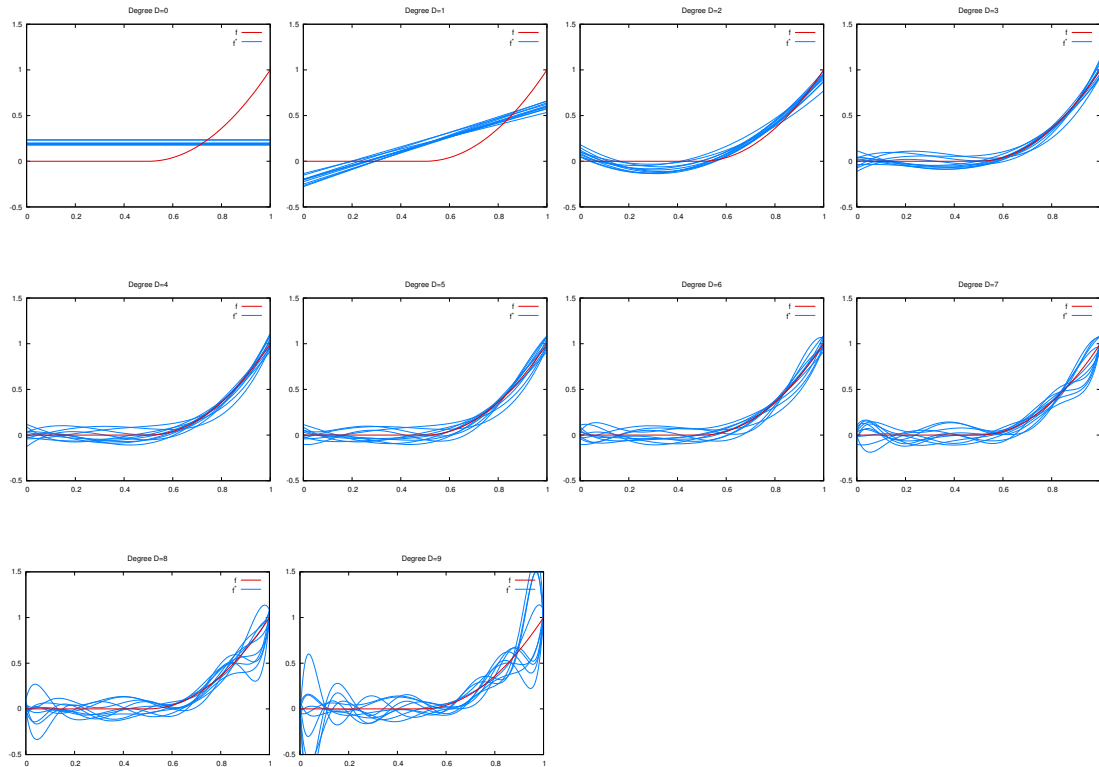


We can use that model to illustrate how the prediction changes when we increase the degree or the regularization.





We can visualize the influence of the noise by generating multiple training sets $\mathcal{D}_1, \dots, \mathcal{D}_M$ with different noise, and training one model on each.



We can reformulate this control of the degree with a penalty

$$\mathcal{L}(\alpha) = \sum_n (f(x_n; \alpha) - y_n)^2 + \sum_d I_d(\alpha_d)$$

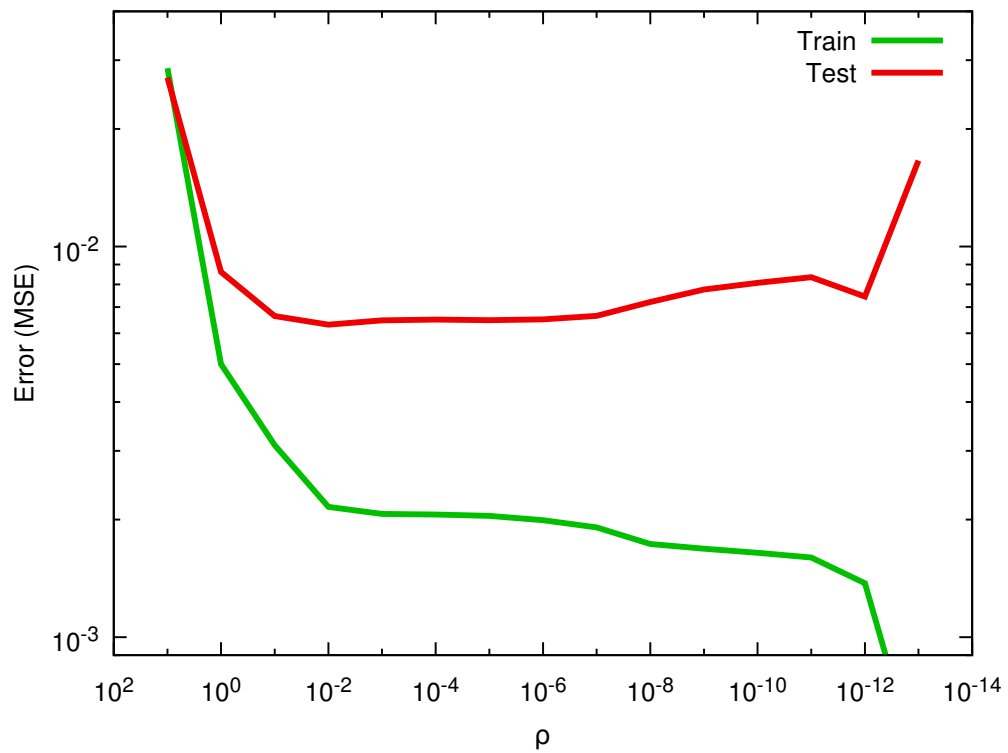
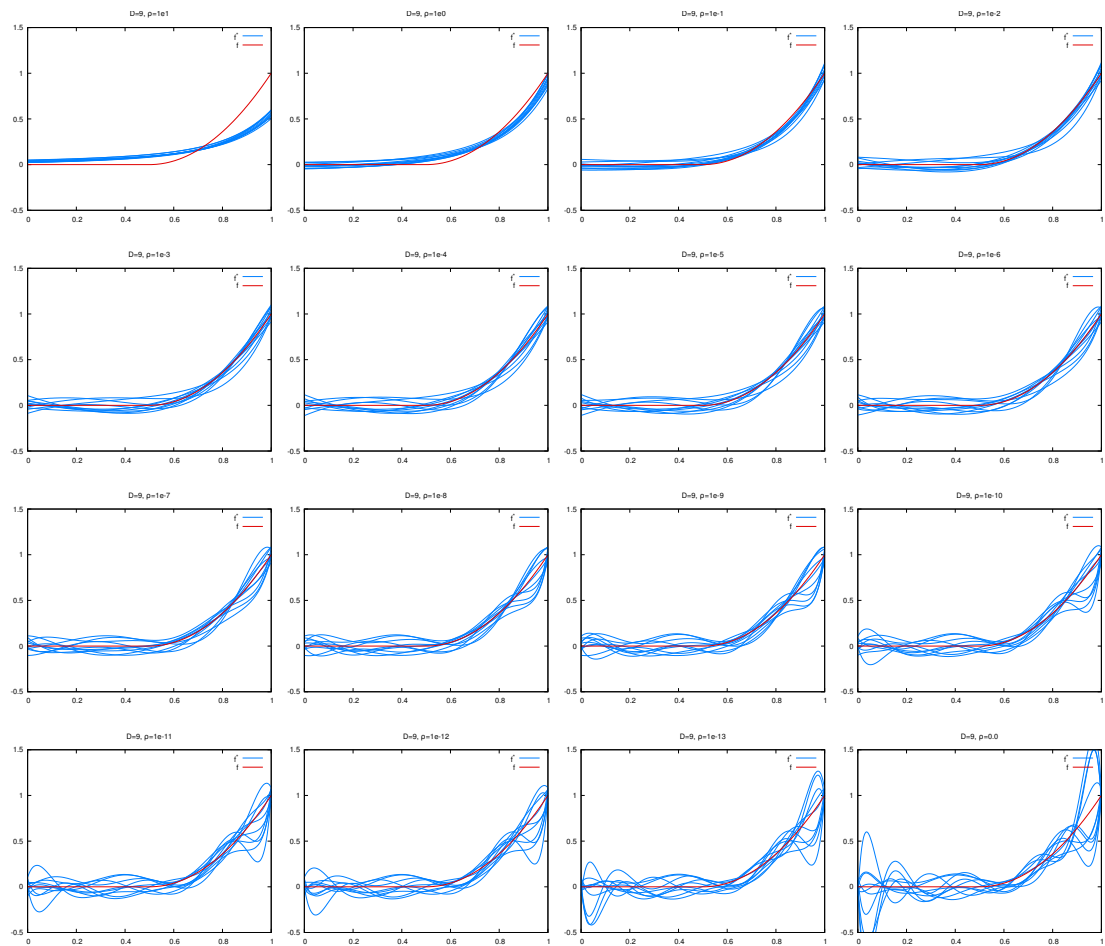
where

$$I_d(\alpha) = \begin{cases} 0 & \text{if } d \leq D \text{ or } \alpha = 0 \\ +\infty & \text{otherwise.} \end{cases}$$

Such a penalty kills any term of degree $> D$.

This motivates the use of more subtle variants. For instance, to keep all this quadratic

$$\mathcal{L}(\alpha) = \sum_n (f(x_n; \alpha) - y_n)^2 + \rho \sum_d \alpha_d^2.$$



We define the **capacity** of a set of predictors as its ability to model an arbitrary functional. This is a vague definition, difficult to make formal.

A mathematically precise notion is the Vapnik–Chervonenkis dimension of a set of functions, which, in the Binary classification case, is the cardinality of the largest set that can be labeled arbitrarily (Vapnik, 1995).

It is a very powerful concept, but is poorly adapted to neural networks. We will not say more about it in this course.

Although the capacity is hard to define precisely, it is quite clear in practice how to modulate it for a given class of models.

In particular one can control over-fitting either by

- Reducing the space \mathcal{F} (less functionals, constrained or degraded optimization), or
- Making the choice of f^* less dependent on data (penalty on coefficients, margin maximization, ensemble methods).

2.4. Proper evaluation protocols

Learning algorithms, in particular deep-learning ones, require the tuning of many meta-parameters.

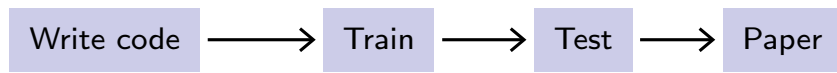
These parameters have a strong impact on the performance, resulting in a “meta” over-fitting through experiments.

We must be extra careful with performance estimation.

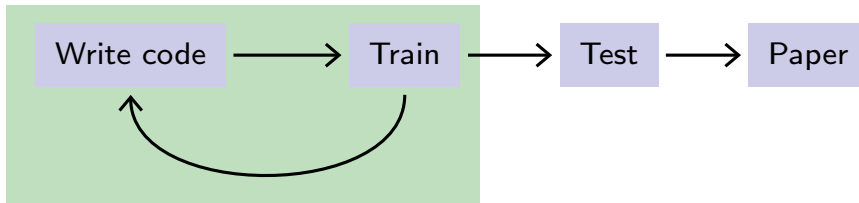
Running 100 times the same experiment on MNIST, with randomized weights, we get:

Worst	Median	Best
1.3%	1.0%	0.82%

The ideal development cycle is

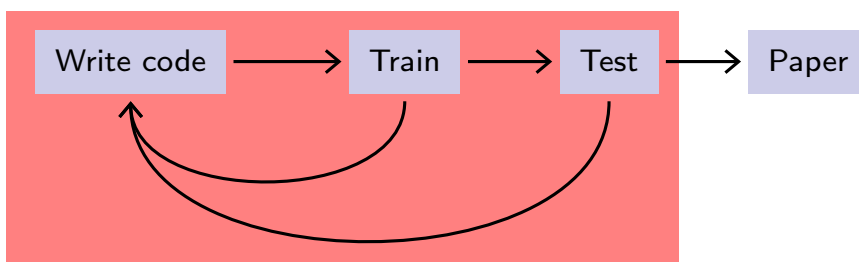



or in practice something like

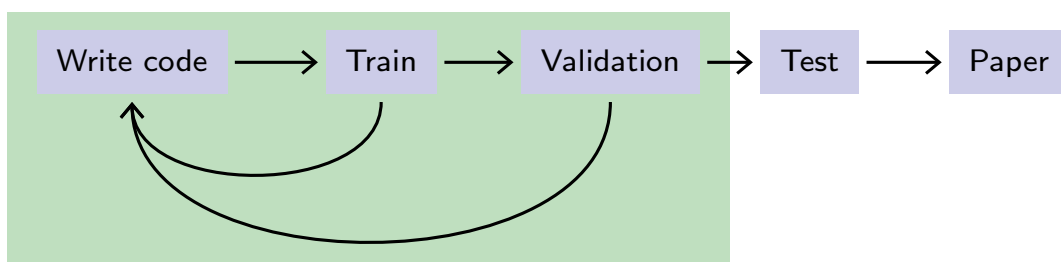


There may be over-fitting, but it does not bias the final performance evaluation.

Unfortunately, it often looks like



 This should be avoided at all costs. The standard strategy is to have a separate validation set for the tuning.

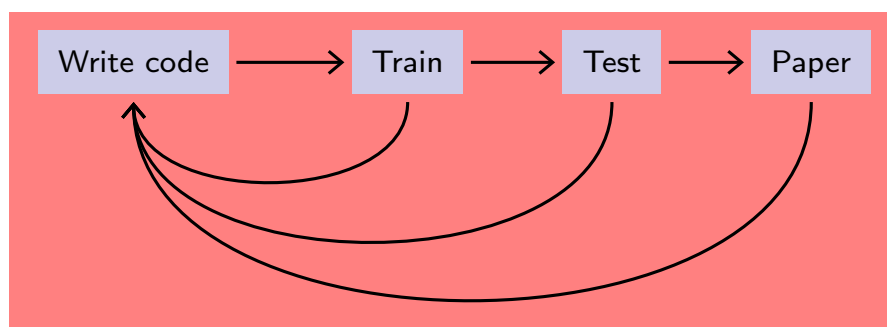


When data is scarce, one can use cross-validation: average through multiple random splits of the data in a train and a validation sets.

There is no unbiased estimator of the variance of cross-validation valid under all distributions (Bengio and Grandvalet, 2004).

Some data-sets (MNIST!) have been used by thousands of researchers, over millions of experiments, in hundreds of papers.

The global overall process looks more like



“Cheating” in machine learning, from bad to “are you kidding?”:

- “Early evaluation stopping”,
- meta-parameter (over-)tuning,
- data-set selection,
- algorithm data-set specific clauses,
- seed selection.

Top-tier conferences are demanding regarding experiments, and are biased against “complicated” pipelines.

The community pushes toward accessible implementations, reference data-sets, leader boards, and constant upgrades of benchmarks.

3.1. The perceptron

The first mathematical model for a neuron was the Threshold Logic Unit, with Boolean inputs and outputs:

$$f(x) = \mathbf{1}_{\{w \sum_i x_i + b \geq 0\}}.$$

It can in particular implement

$$\begin{aligned} \text{or}(u, v) &= \mathbf{1}_{\{u+v-0.5 \geq 0\}} && (w = 1, b = -0.5) \\ \text{and}(u, v) &= \mathbf{1}_{\{u+v-1.5 \geq 0\}} && (w = 1, b = -1.5) \\ \text{not}(u) &= \mathbf{1}_{\{-u+0.5 \geq 0\}} && (w = -1, b = 0.5) \end{aligned}$$

Hence, **any Boolean function can be build with such units.**

(McCulloch and Pitts, 1943)

The perceptron is very similar

$$f(x) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

but the inputs are real values and the weights can be different.

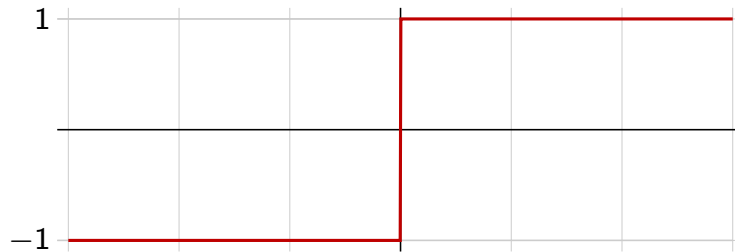
This model was originally motivated by biology, with w_i being the *synaptic weights*, and x_i and f firing rates.

It is a (very) crude biological model.

(Rosenblatt, 1957)

To make things simpler we take responses ± 1 . Let

$$\sigma(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

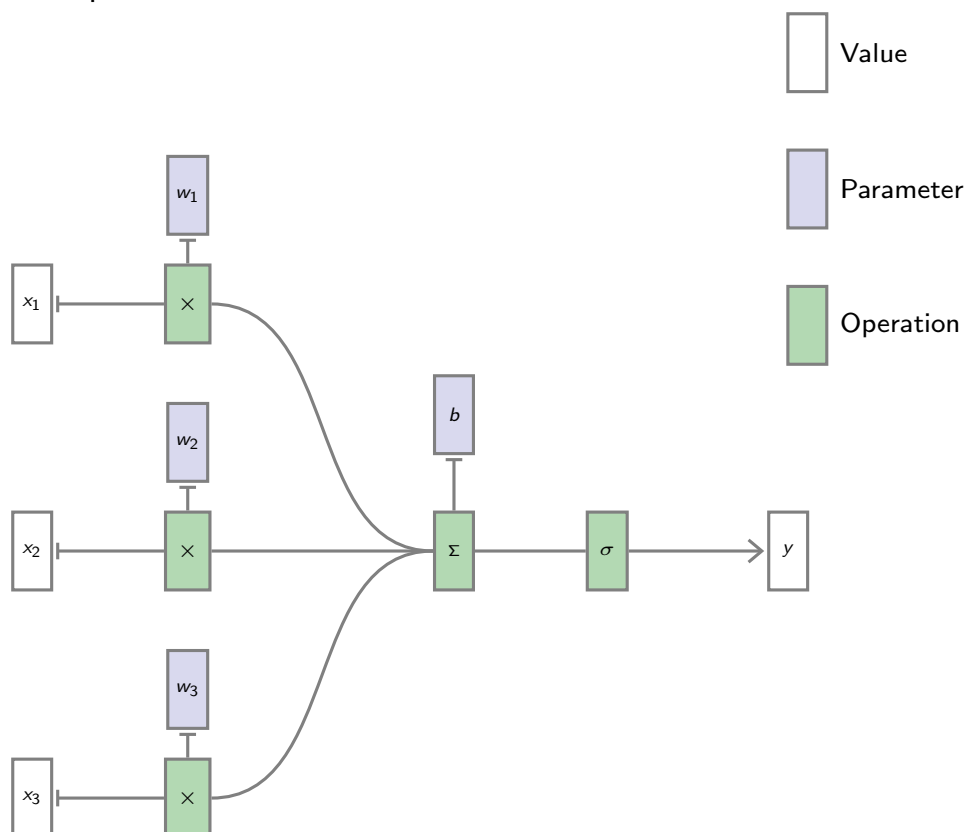


The perceptron classification rule boils down to

$$f(x) = \sigma(w \cdot x + b).$$

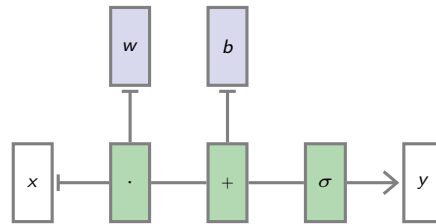
For neural networks, the function σ that follows a linear operator is called the **activation function**.

We can represent this “neuron” as follows:



We can also use tensor operations, as in

$$f(x) = \sigma(w \cdot x + b).$$



Given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{-1, 1\}, \quad n = 1, \dots, N,$$

a very simple scheme to train such a linear operator for classification is the **perceptron algorithm**:

1. Start with $w^0 = 0$,
2. while $\exists n_k$ s.t. $y_{n_k} (w^k \cdot x_{n_k}) \leq 0$, update $w^{k+1} = w^k + y_{n_k} x_{n_k}$.

The bias b can be introduced as one of the w s by adding a constant component to x equal to 1.

(Rosenblatt, 1957)

```

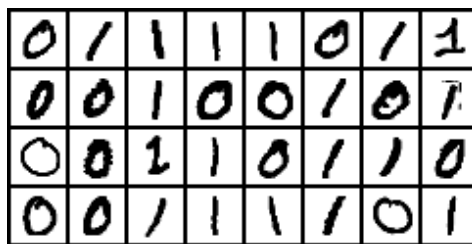
def train_perceptron(x, y, nb_epochs_max):
    w = torch.zeros(x.size(1))

    for e in range(nb_epochs_max):
        nb_changes = 0
        for i in range(x.size(0)):
            if x[i].dot(w) * y[i] <= 0:
                w = w + y[i] * x[i]
                nb_changes = nb_changes + 1
        if nb_changes == 0: break;

    return w

```

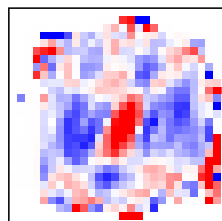
This crude algorithm works often surprisingly well. With MNIST’s “0”s as negative class, and “1”s as positive one.



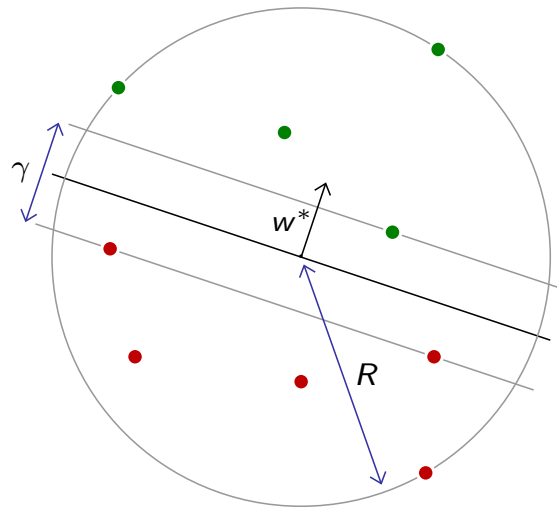
```

epoch 0 nb_changes 64 train_error 0.23% test_error 0.19%
epoch 1 nb_changes 24 train_error 0.07% test_error 0.00%
epoch 2 nb_changes 10 train_error 0.06% test_error 0.05%
epoch 3 nb_changes 6 train_error 0.03% test_error 0.14%
epoch 4 nb_changes 5 train_error 0.03% test_error 0.09%
epoch 5 nb_changes 4 train_error 0.02% test_error 0.14%
epoch 6 nb_changes 3 train_error 0.01% test_error 0.14%
epoch 7 nb_changes 2 train_error 0.00% test_error 0.14%
epoch 8 nb_changes 0 train_error 0.00% test_error 0.14%

```



We can get a convergence result under two assumptions:



1. The x_n are in a sphere of radius R :
 $\exists R > 0, \forall n, \|x_n\| \leq R.$
2. The two populations can be separated with a margin $\gamma > 0$.
 $\exists w^*, \|w^*\| = 1, \exists \gamma > 0, \forall n, y_n (x_n \cdot w^*) \geq \gamma/2.$

To prove the convergence, let us make the assumption that there still is a misclassified sample at iteration k , and w^{k+1} is the weight vector updated with it. We have

$$\begin{aligned}
 w^{k+1} \cdot w^* &= (w^k + y_{n_k} x_{n_k}) \cdot w^* \\
 &= w^k \cdot w^* + y_{n_k} (x_{n_k} \cdot w^*) \\
 &\geq w^k \cdot w^* + \gamma/2 \\
 &\geq (k+1) \gamma/2.
 \end{aligned}$$

Since

$$\|w^k\| \|w^*\| \geq w^k \cdot w^*,$$

we get

$$\begin{aligned}
 \|w^k\|^2 &\geq (w^k \cdot w^*)^2 / \|w^*\|^2 \\
 &\geq k^2 \gamma^2 / 4.
 \end{aligned}$$

And

$$\begin{aligned}\|w^{k+1}\|^2 &= w^{k+1} \cdot w^{k+1} \\ &= (w^k + y_{n_k} x_{n_k}) \cdot (w^k + y_{n_k} x_{n_k}) \\ &= w^k \cdot w^k + 2 \underbrace{y_{n_k} w^k \cdot x_{n_k}}_{\leq 0} + \underbrace{\|x_{n_k}\|^2}_{\leq R^2} \\ &\leq \|w^k\|^2 + R^2 \\ &\leq (k+1) R^2.\end{aligned}$$

Putting these two results together, we get

$$k^2 \gamma^2 / 4 \leq \|w^k\|^2 \leq k R^2$$

hence

$$k \leq 4R^2/\gamma^2,$$

hence no misclassified sample can remain after $\lfloor 4R^2/\gamma^2 \rfloor$ iterations.

This result makes sense:

- The bound does not change if the population is scaled, and
- the larger the margin, the more quickly the algorithm classifies all the samples correctly.

The perceptron stops as soon as it finds a separating boundary.

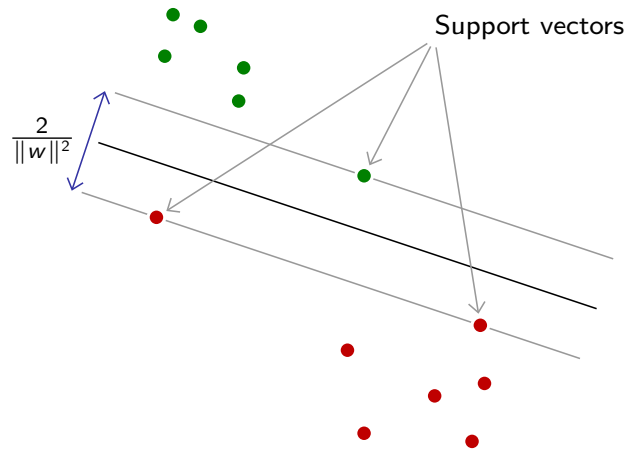
Other algorithms maximize the distance of samples to the decision boundary, which improves robustness to noise.

Support Vector Machines (SVM) achieve this by minimizing

$$\mathcal{L}(w, b) = \lambda \|w\|^2 + \frac{1}{N} \sum_n \max(0, 1 - y_n(w \cdot x_n + b)),$$

which is convex and has a global optimum.

$$\mathcal{L}(w, b) = \lambda \|w\|^2 + \frac{1}{N} \sum_n \max(0, 1 - y_n(w \cdot x_n + b))$$



Minimizing $\max(0, 1 - y_n(w \cdot x_n + b))$ pushes the n th sample beyond the plane $w \cdot x + b = y_n$, and minimizing $\|w\|^2$ increases the distance between the $w \cdot x + b = \pm 1$.

At convergence, only a small number of samples matter, the “support vectors”.

The term

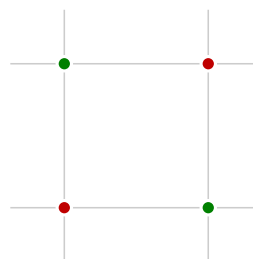
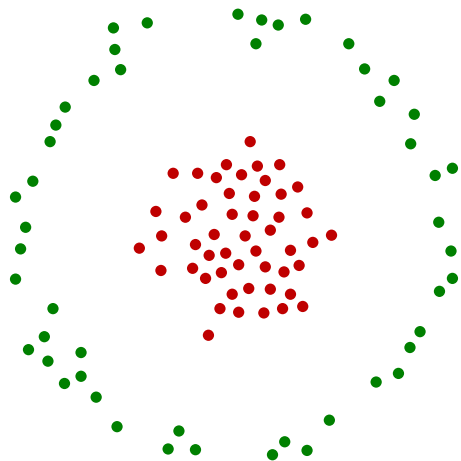
$$\max(0, 1 - \alpha)$$

is the so called “hinge loss”



3.3. Linear separability and feature design

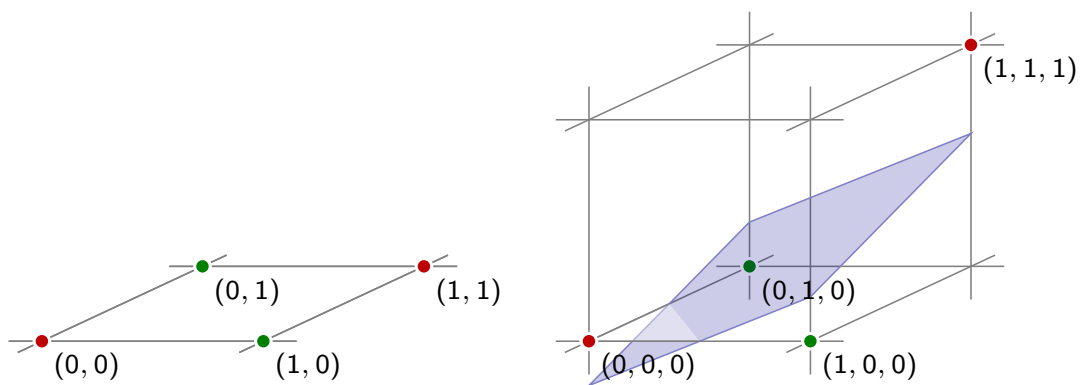
The main weakness of linear predictors is their lack of capacity. For classification, the populations have to be **linearly separable**.

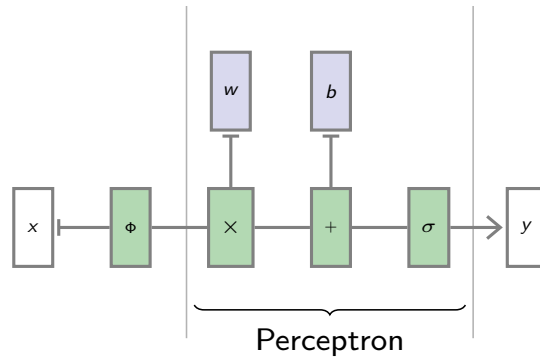


“xor”

The xor example can be solved by pre-processing the data to make the two populations linearly separable.

$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$





This is similar to the polynomial regression. If we have

$$\Phi : x \mapsto (1, x, x^2, \dots, x^D)$$

and

$$\alpha = (\alpha_0, \dots, \alpha_D)$$

then

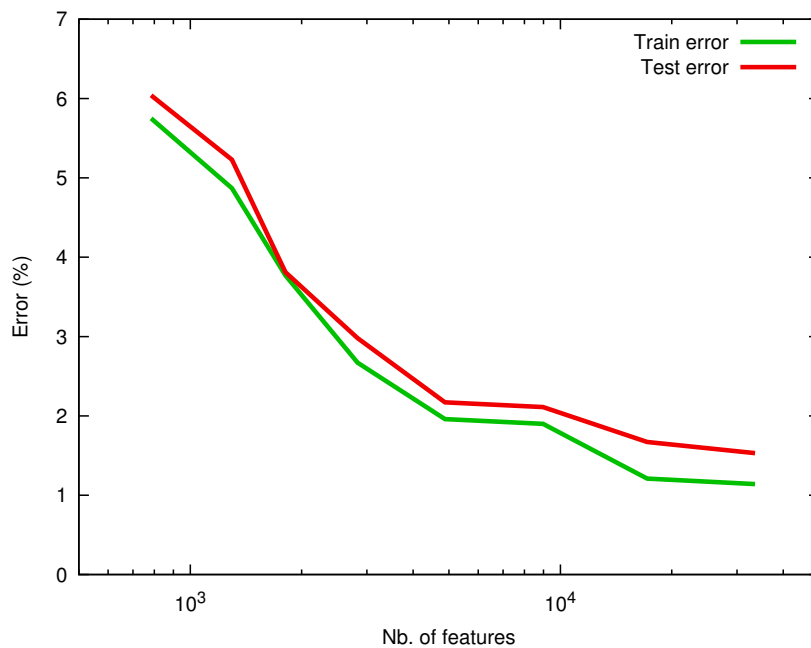
$$\sum_{d=0}^D \alpha_d x^d = \alpha \cdot \Phi(x).$$

By increasing D , we can approximate any continuous real function on a compact space (Stone-Weierstrass theorem).

It means that we can make the capacity as high as we want.

We can apply the same to a more realistic binary classification problem: MNIST's "8" vs. the other classes with a perceptron.

The original 28×28 features are supplemented with the products of pairs of features taken at random.



Remember the bias-variance tradeoff:

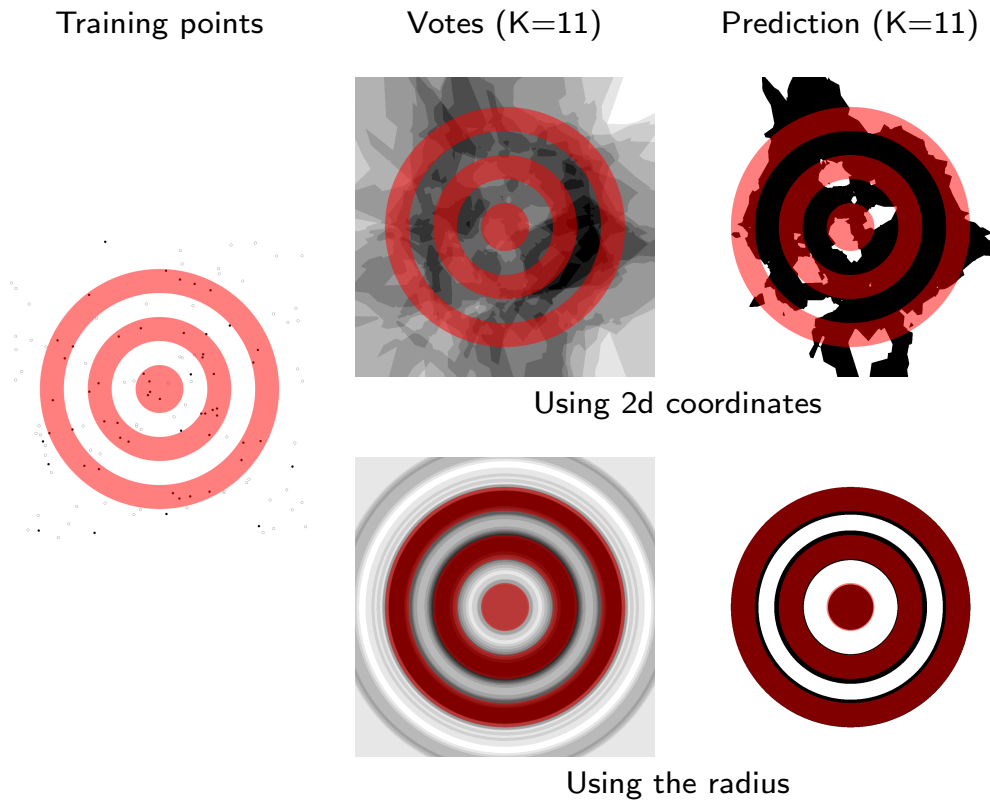
$$\mathbb{E}((Y - y)^2) = \underbrace{(\mathbb{E}(Y) - y)^2}_{\text{Bias}} + \underbrace{\mathbb{V}(Y)}_{\text{Variance}}.$$

The right class of models reduces the bias more and increases the variance less.

Beside increasing capacity to reduce the bias, “feature design” may also be a way of reducing capacity without hurting the bias, or with improving it.

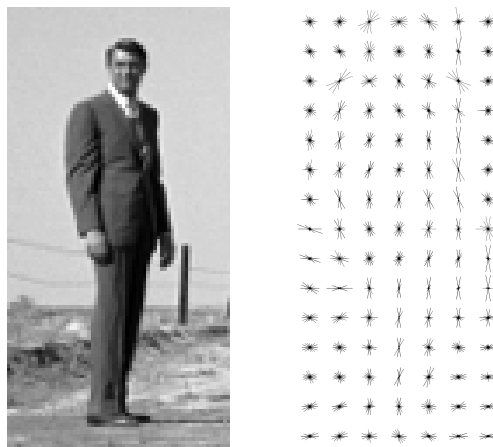
In particular, good features should be invariant to perturbations of the signal known to keep the value to predict unchanged.

We can illustrate the use of features with k -NN on a task with radial symmetry. Using the radius instead of 2d coordinates allows to cope with label noise.



A classical example is the “Histogram of Oriented Gradient” descriptors (HOG), initially designed for person detection.

Roughly: divide the image in 8×8 blocks, compute in each the distribution of edge orientations over 9 bins.



Dalal and Triggs (2005) combined them with a SVM, and Dollár et al. (2009) extended them with other modalities into the “channel features”.

Many methods (perceptron, SVM, k -means, PCA, etc.) only require to compute $\kappa(x, x') = \Phi(x) \cdot \Phi(x')$ for any (x, x') .

So one needs to specify κ alone, and may keep Φ undefined.

This is the **kernel trick**, which we will not talk about in this course.

Training a model composed of manually engineered features and a parametric model such as logistic regression is now referred to as “**shallow learning**”.

The signal goes through a single processing trained from data.

3.4. Multi-Layer Perceptrons

So far we have seen linear classifiers of the form

$$\begin{aligned}\mathbb{R}^D &\rightarrow \mathbb{R} \\ x &\mapsto \sigma(w \cdot x + b),\end{aligned}$$

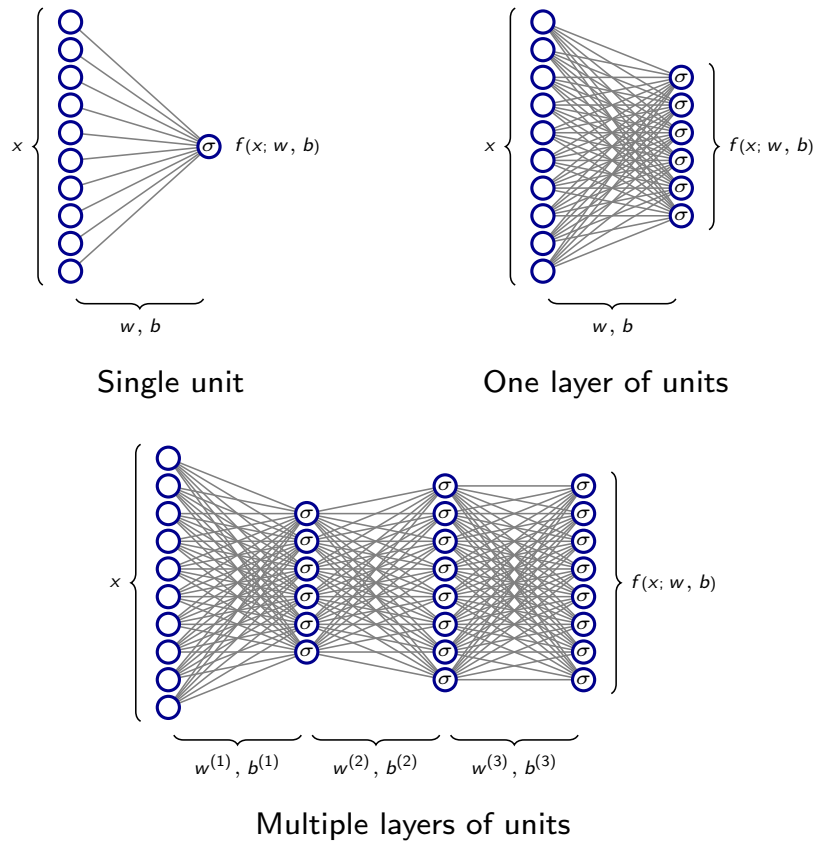
with $w \in \mathbb{R}^D$, $b \in \mathbb{R}$, and $\sigma : \mathbb{R} \rightarrow \mathbb{R}$.

This can naturally be extended to a multi-dimension output by applying a similar transformation to every output, which leads to

$$\begin{aligned}\mathbb{R}^D &\rightarrow \mathbb{R}^C \\ x &\mapsto \sigma(wx + b),\end{aligned}$$

with $w \in \mathbb{R}^{C \times D}$, $b \in \mathbb{R}^C$, and σ is applied component-wise.

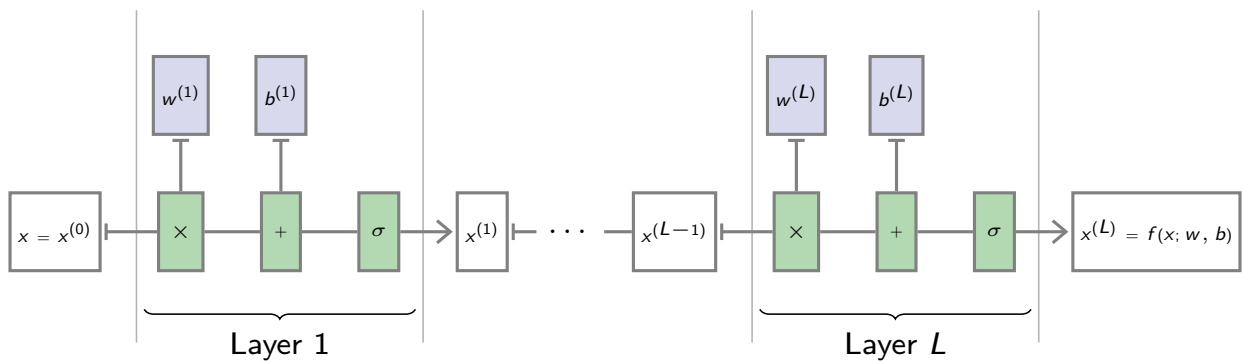
Even though it has no practical value implementation-wise, we can represent such a model as a combination of units, and extend it.



This latter structure can be formally defined, with $x^{(0)} = x$,

$$\forall l = 1, \dots, L, x^{(l)} = \sigma \left(w^{(l)} x^{(l-1)} + b^{(l)} \right)$$

and $f(x; w, b) = x^{(L)}$.



Such a model is a **Multi-Layer Perceptron (MLP)**.

Note that if σ is an affine transformation, the full MLP is a composition of affine mappings, and itself an affine mapping.

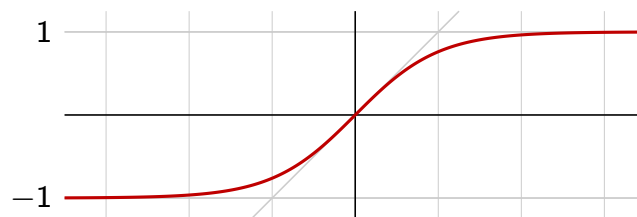
Consequently:



The activation function σ should be non-linear, or the resulting MLP is an affine mapping with a peculiar parametrization.

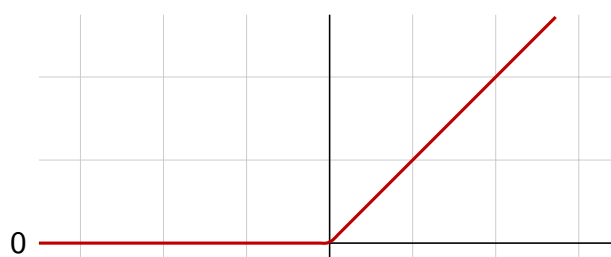
The two classical activation functions are the hyperbolic tangent

$$x \mapsto \frac{2}{1 + e^{-2x}} - 1$$



and the rectified linear unit (ReLU)

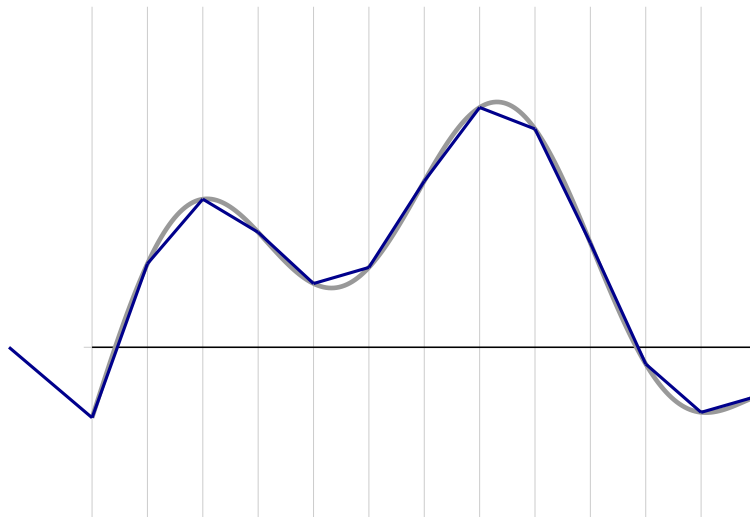
$$x \mapsto \max(0, x)$$



Universal approximation

We can approximate any $\psi \in \mathcal{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1x + b_1) + \sigma(w_2x + b_2) + \sigma(w_3x + b_3) + \dots$$



This is true for other activation functions under mild assumptions.

Extending this result to any $\psi \in \mathcal{C}([0, 1]^D, \mathbb{R})$ requires a bit of work.

First, we can use the previous result for the sin function

$$\forall A > 0, \epsilon > 0, \exists N, (\alpha_n, a_n) \in \mathbb{R} \times \mathbb{R}, n = 1, \dots, N,$$

$$\text{s.t. } \max_{x \in [-A, A]} \left| \sin(x) - \sum_{n=1}^N \alpha_n \sigma(x - a_n) \right| \leq \epsilon.$$

And the density of Fourier series provides

$$\forall \psi \in \mathcal{C}([0, 1]^D, \mathbb{R}), \delta > 0, \exists M, (v_m, \gamma_m, c_m) \in \mathbb{R}^D \times \mathbb{R} \times \mathbb{R}, m = 1, \dots, M,$$

$$\text{s.t. } \max_{x \in [0, 1]^D} \left| \psi(x) - \sum_{m=1}^M \gamma_m \sin(v_m \cdot x + c_m) \right| \leq \delta.$$

So, $\forall \xi > 0$, with

$$\delta = \frac{\xi}{2}, A = \max_{1 \leq m \leq M} \max_{x \in [0, 1]^D} |v_m \cdot x + c_m|, \text{ and } \epsilon = \frac{\xi}{2 \sum_m |\gamma_m|}$$

we get, $\forall x \in [0, 1]^D$,

$$\begin{aligned} & \left| \psi(x) - \sum_{m=1}^M \gamma_m \left(\sum_{n=1}^N \alpha_n \sigma(v_m \cdot x + c_m - a_n) \right) \right| \\ & \leq \underbrace{\left| \psi(x) - \sum_{m=1}^M \gamma_m \sin(v_m \cdot x + c_m) \right|}_{\leq \frac{\xi}{2}} \\ & \quad + \underbrace{\sum_{m=1}^M |\gamma_m| \left| \sin(v_m \cdot x + c_m) - \sum_{n=1}^N \alpha_n \sigma(v_m \cdot x + c_m - a_n) \right|}_{\leq \frac{\xi}{2 \sum_m |\gamma_m|}} \\ & \leq \frac{\xi}{2} \end{aligned}$$

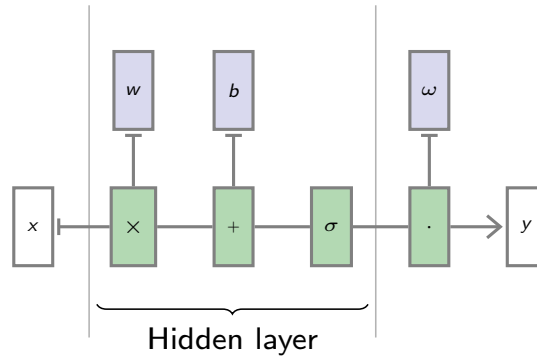
So we can approximate any continuous function

$$\psi : [0, 1]^D \rightarrow \mathbb{R}$$

with a one hidden layer perceptron

$$x \mapsto \omega \cdot \sigma(wx + b),$$

where $b \in \mathbb{R}^K$, $w \in \mathbb{R}^{K \times D}$, and $\omega \in \mathbb{R}^K$.



This is the **universal approximation theorem**.



A better approximation requires a larger hidden layer (larger K), and this theorem says nothing about the relation between the two.

3.5. Gradient descent

We saw that training consists of finding the model parameters minimizing an empirical risk or loss, for instance the mean-squared error (MSE)

$$\mathcal{L}(w, b) = \frac{1}{N} \sum_n (f(x_n; w, b) - y_n)^2.$$

Other losses are more fitting for classification, certain regression problems, or density estimation. We will come back to this.

So far we minimized the loss either with an analytic solution for the MSE, or with *ad hoc* recipes for the empirical error rate (*k*-NN and perceptron).

There is generally no *ad hoc* method. The logistic regression for instance

$$P_w(Y = 1 | X = x) = \sigma(w \cdot x + b), \text{ with } \sigma(x) = \frac{1}{1 + e^{-x}}$$

leads to the loss

$$\mathcal{L}(w, b) = - \sum_n \log \sigma(y_n(w \cdot x_n + b))$$

which cannot be minimized analytically.

The general minimization method used in such a case is the **gradient descent**.

Given a functional

$$f : \mathbb{R}^D \rightarrow \mathbb{R}$$
$$x \mapsto f(x_1, \dots, x_D),$$

its gradient is the mapping

$$\nabla f : \mathbb{R}^D \rightarrow \mathbb{R}^D$$
$$x \mapsto \left(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_D}(x) \right).$$

To minimize a functional

$$\mathcal{L} : \mathbb{R}^D \rightarrow \mathbb{R}$$

the gradient descent uses local linear information to iteratively move toward a (local) minimum.

For $w_0 \in \mathbb{R}^D$, consider an approximation of \mathcal{L} around w_0

$$\tilde{\mathcal{L}}_{w_0}(w) = \mathcal{L}(w_0) + \nabla \mathcal{L}(w_0)^T (w - w_0) + \frac{1}{2\eta} \|w - w_0\|^2.$$

Note that the chosen quadratic term does not depend on \mathcal{L} .

We have

$$\nabla \tilde{\mathcal{L}}_{w_0}(w) = \nabla \mathcal{L}(w_0) + \frac{1}{\eta} (w - w_0),$$

which leads to

$$\operatorname{argmin}_w \tilde{\mathcal{L}}_{w_0}(w) = w_0 - \eta \nabla \mathcal{L}(w_0).$$

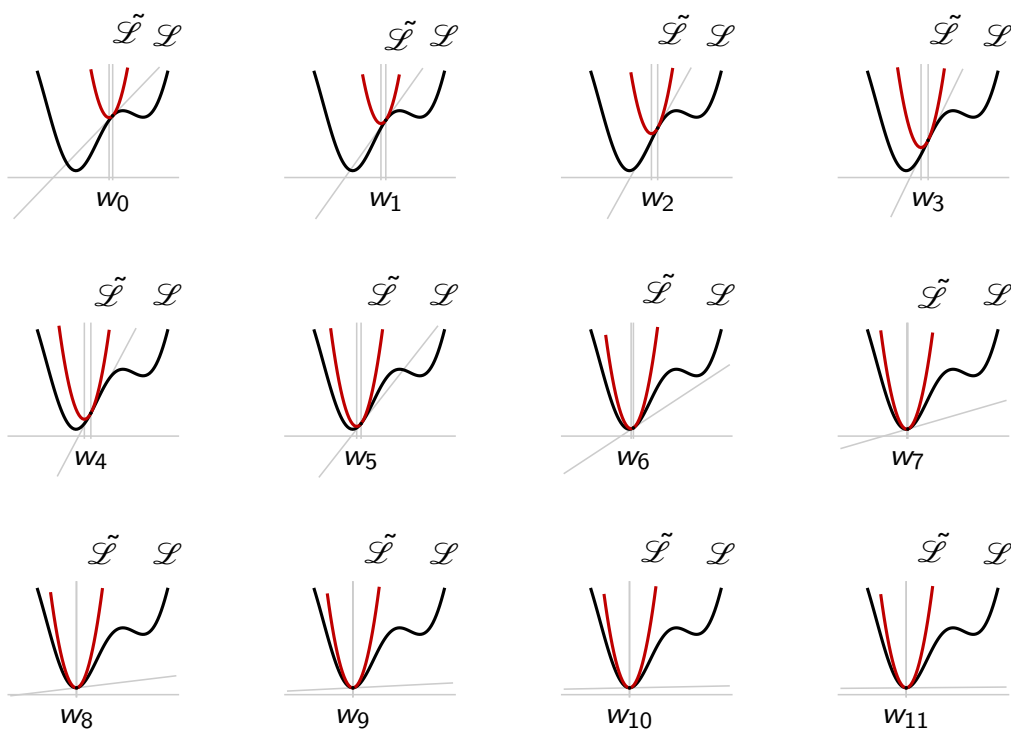
The resulting iterative rule, which goes to the minimum of the approximation at the current location, takes the form:

$$w_{t+1} = w_t - \eta \nabla \tilde{\mathcal{L}}(w_t),$$

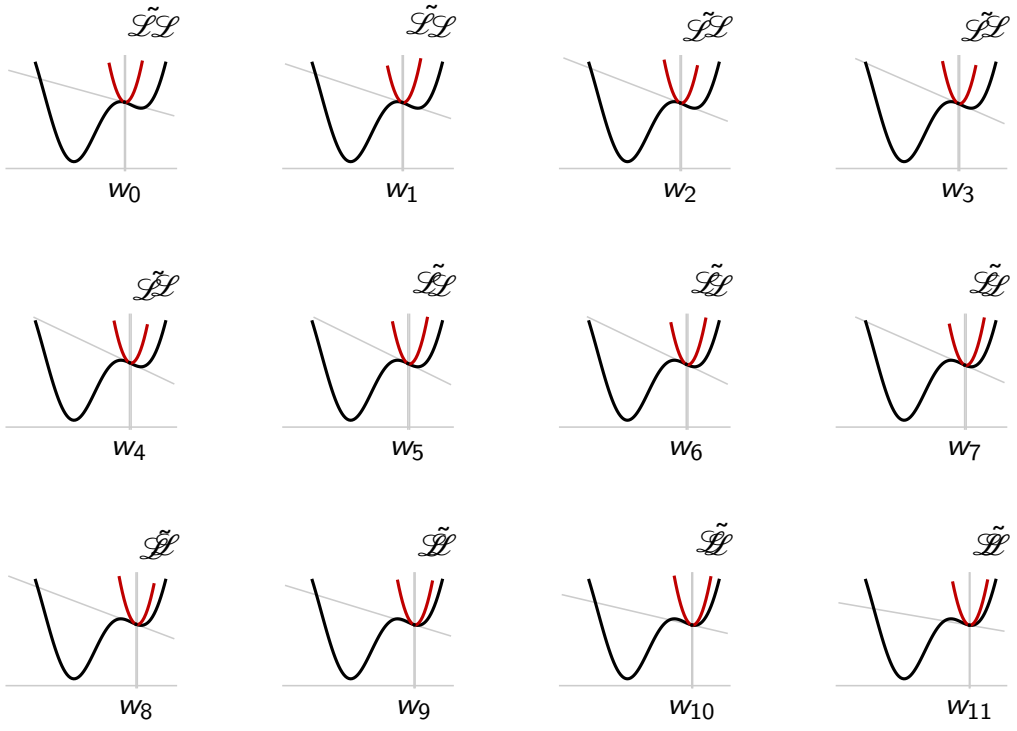
which corresponds intuitively to “following the steepest descent”.

This [most of the time] eventually ends up in a **local** minimum, and the choices of w_0 and η are important.

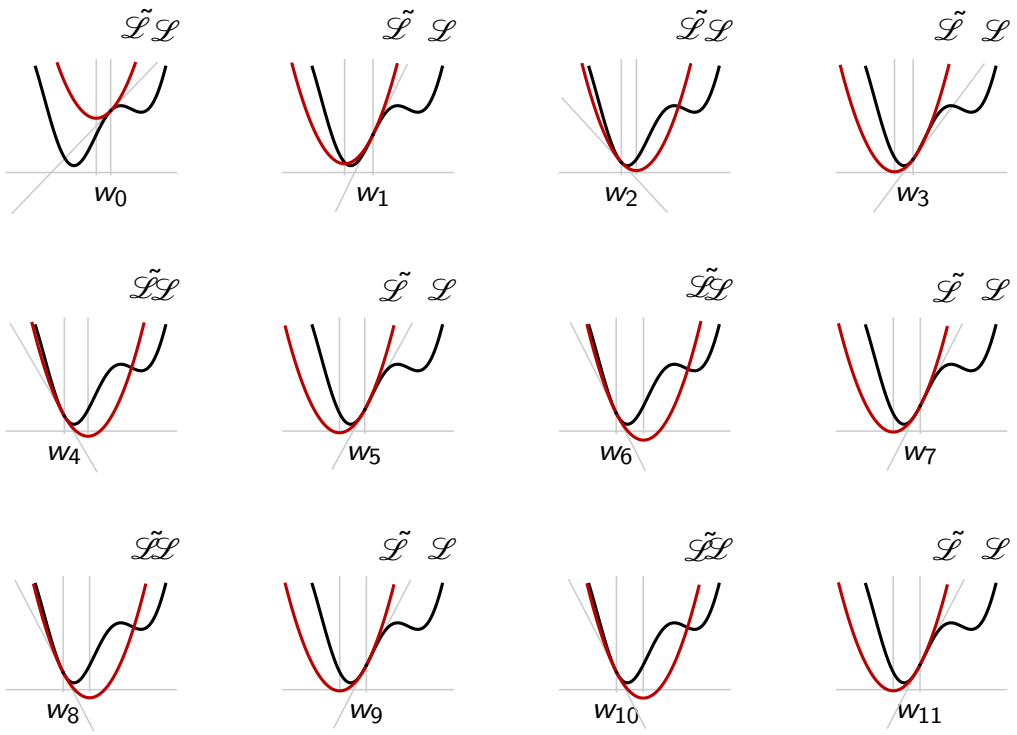
$$\eta = 0.125$$

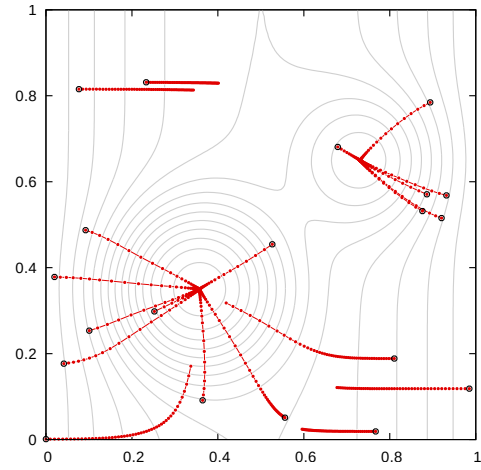
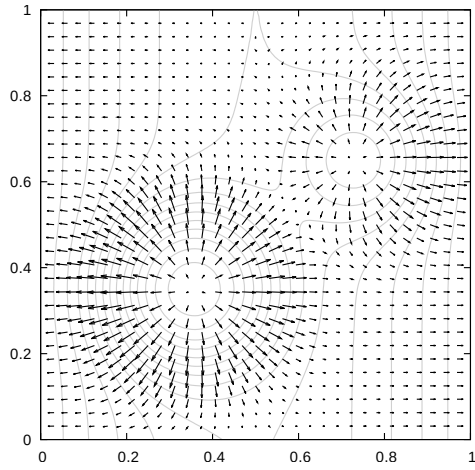
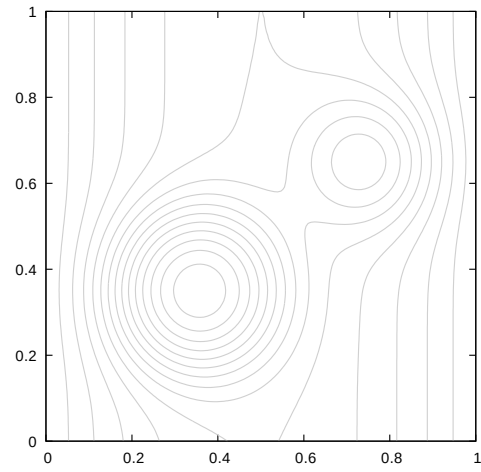
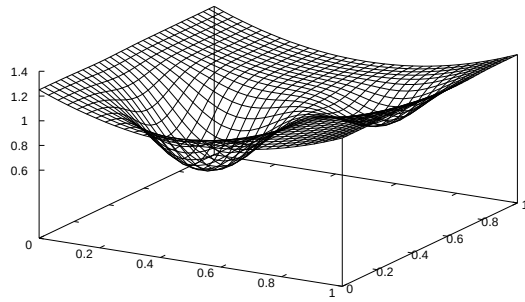


$$\eta = 0.125$$



$$\eta = 0.5$$





We saw that the minimum of the logistic regression loss

$$\mathcal{L}(w, b) = - \sum_n \log \sigma(y_n(w \cdot x_n + b))$$

does not have an analytic form.

We can derive

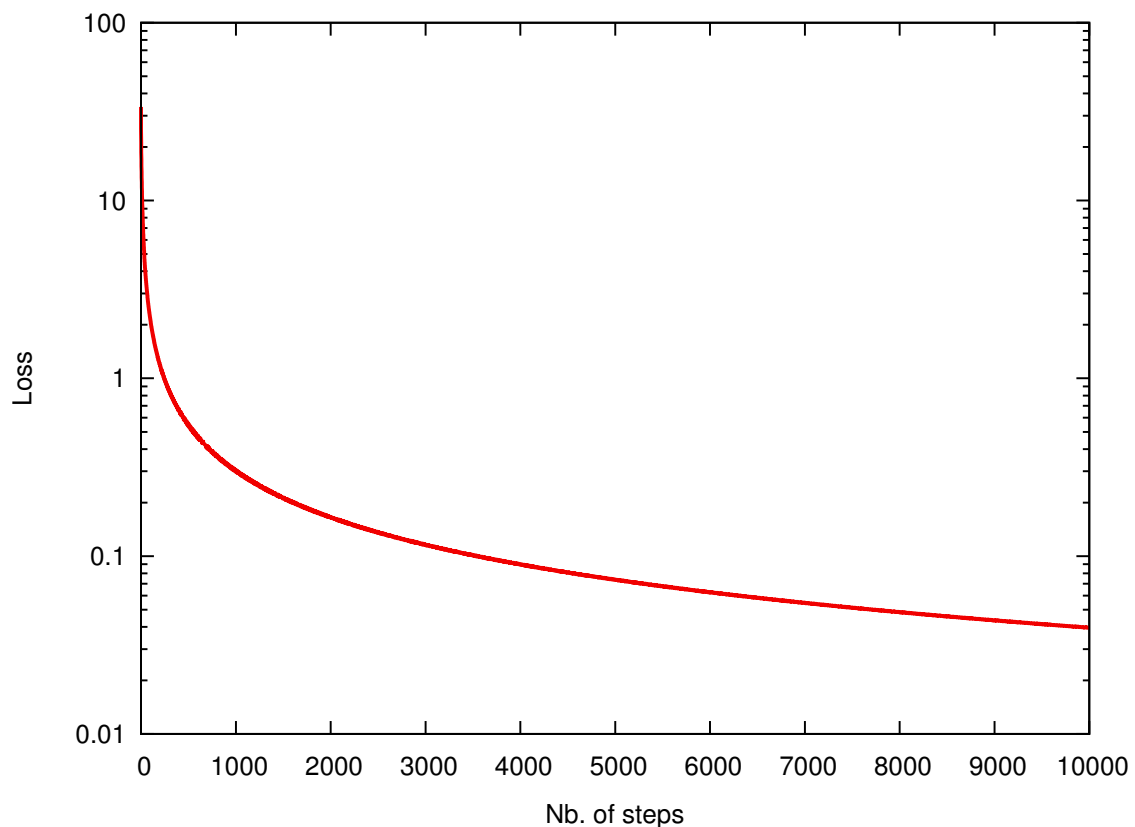
$$\frac{\partial \mathcal{L}}{\partial b} = - \sum_n \underbrace{y_n \sigma(-y_n(w \cdot x_n + b))}_{u_n},$$
$$\forall d, \frac{\partial \mathcal{L}}{\partial w_d} = - \sum_n \underbrace{x_{n,d} y_n \sigma(-y_n(w \cdot x_n + b))}_{v_{n,d}},$$

which can be implemented as

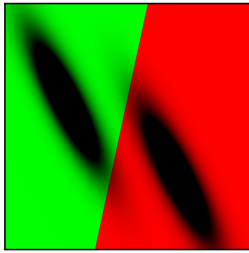
```
def gradient(x, y, w, b):  
    u = y * (- y * (x.mv(w) + b)).sigmoid_()  
    v = x * u.view(-1, 1) # Broadcasting  
    return - v.sum(0), - u.sum()
```

and the gradient descent as

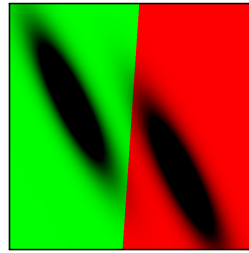
```
w, b = torch.empty(x.size(1)).normal_(), 0  
eta = 1e-1  
  
for k in range(nb_iterations):  
    dw, db = gradient(x, y, w, b)  
    w -= eta * dw  
    b -= eta * db
```



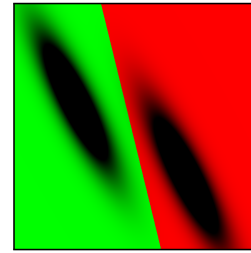
With 100 training points and $\eta = 10^{-1}$.



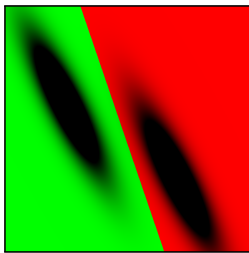
$n = 0$



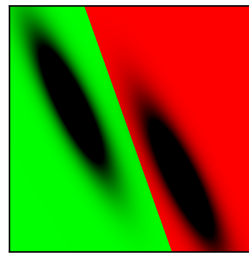
$n = 10$



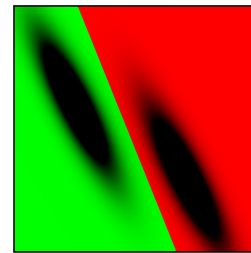
$n = 10^2$



$n = 10^3$



$n = 10^4$



LDA

3.6. Back-propagation

We want to train an MLP by minimizing a loss over the training set

$$\mathcal{L}(w, b) = \sum_n \ell(f(x_n; w, b), y_n).$$

To use gradient descent, we need the expression of the gradient of the loss with respect to the parameters:

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b_i^{(l)}}.$$

So, if we define $\ell_n = \ell(f(x_n; w, b), y_n)$, what we need is

$$\frac{\partial \ell_n}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \ell_n}{\partial b_i^{(l)}}.$$

For clarity, we consider a single training sample x , and introduce $s^{(1)}, \dots, s^{(L)}$ as the summations before activation functions.

$$x^{(0)} = x \xrightarrow{w^{(1)}, b^{(1)}} s^{(1)} \xrightarrow{\sigma} x^{(1)} \xrightarrow{w^{(2)}, b^{(2)}} s^{(2)} \xrightarrow{\sigma} \dots \xrightarrow{w^{(L)}, b^{(L)}} s^{(L)} \xrightarrow{\sigma} x^{(L)} = f(x; w, b).$$

Formally we set $x^{(0)} = x$,

$$\forall l = 1, \dots, L, \quad \begin{cases} s^{(l)} = w^{(l)} x^{(l-1)} + b^{(l)} \\ x^{(l)} = \sigma(s^{(l)}), \end{cases}$$

and we set the output of the network as $f(x; w, b) = x^{(L)}$.

This is the forward pass.

The core principle of the back-propagation algorithm is the “chain rule” from differential calculus:

$$(g \circ f)' = (g' \circ f)f'$$

which generalizes to longer compositions and higher dimensions

$$J_{f_N \circ f_{N-1} \circ \dots \circ f_1}(x) = \prod_{n=1}^N J_{f_n}(f_{n-1} \circ \dots \circ f_1(x)),$$

where $J_f(x)$ is the Jacobian of f at x , that is the matrix of the linear approximation of f in the neighborhood of x .

The linear approximation of a composition of mappings is the product of their individual linear approximations.

What follows is exactly this principle applied to a MLP.

$$\dots \xrightarrow{\sigma} \underbrace{x^{(l-1)}} \xrightarrow{w^{(l)}, b^{(l)}} s^{(l)} \xrightarrow{\sigma} x^{(l)} \xrightarrow{w^{(l+1)}, b^{(l+1)}} s^{(l+1)} \xrightarrow{\sigma} \dots x^{(L)} \rightarrow \ell$$

We have

$$s_i^{(l)} = \sum_j w_{i,j}^{(l)} x_j^{(l-1)} + b_i^{(l)},$$

so $w_{i,j}^{(l)}$ influences ℓ only through $s_i^{(l)}$, and we get

$$\frac{\partial \ell}{\partial w_{i,j}^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}} \frac{\partial s_i^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}} x_j^{(l-1)},$$

and similarly

$$\frac{\partial \ell}{\partial b_i^{(l)}} = \frac{\partial \ell}{\partial s_i^{(l)}}.$$

Since we know $x_j^{(l-1)}$ from the forward pass, we only need $\frac{\partial \ell}{\partial s_i^{(l)}}$.

$$\dots \xrightarrow{\sigma} x^{(l-1)} \xrightarrow{w^{(l)}, b^{(l)}} \underbrace{s^{(l)}} \xrightarrow{\sigma} x^{(l)} \xrightarrow{w^{(l+1)}, b^{(l+1)}} s^{(l+1)} \xrightarrow{\sigma} \dots \rightarrow x^{(L)} \rightarrow \ell$$

We have

$$x_i^{(l)} = \sigma(s_i^{(l)}),$$

and since $s_i^{(l)}$ influences ℓ only through $x_i^{(l)}$, the chain rule gives

$$\frac{\partial \ell}{\partial s_i^{(l)}} = \frac{\partial \ell}{\partial x_i^{(l)}} \frac{\partial x_i^{(l)}}{\partial s_i^{(l)}} = \frac{\partial \ell}{\partial x_i^{(l)}} \sigma'(s_i^{(l)}),$$

Since we know $s_i^{(l)}$ from the forward pass, we only need $\frac{\partial \ell}{\partial x_i^{(l)}}$.

$$\dots \xrightarrow{\sigma} x^{(l-1)} \xrightarrow{w^{(l)}, b^{(l)}} s^{(l)} \xrightarrow{\sigma} x^{(l)} \xrightarrow{w^{(l+1)}, b^{(l+1)}} s^{(l+1)} \xrightarrow{\sigma} \dots \rightarrow \underbrace{x^{(L)}} \rightarrow \ell$$

We know

$$\frac{\partial \ell}{\partial x_i^{(L)}}$$

from the definition of ℓ , and $\forall l = 1, \dots, L - 1$, since

$$s_h^{(l+1)} = \sum_i w_{h,i}^{l+1} x_i^{(l)} + b_h^{l+1},$$

and $x_i^{(l)}$ influences ℓ only through the $s_h^{(l+1)}$, we have

$$\frac{\partial \ell}{\partial x_i^{(l)}} = \sum_h \frac{\partial \ell}{\partial s_h^{(l+1)}} \frac{\partial s_h^{(l+1)}}{\partial x_i^{(l)}} = \sum_h \frac{\partial \ell}{\partial s_h^{(l+1)}} w_{h,i}^{l+1}.$$

To write all this in tensorial form, if $\psi : \mathbb{R}^N \rightarrow \mathbb{R}^M$, we will use the standard Jacobian notation

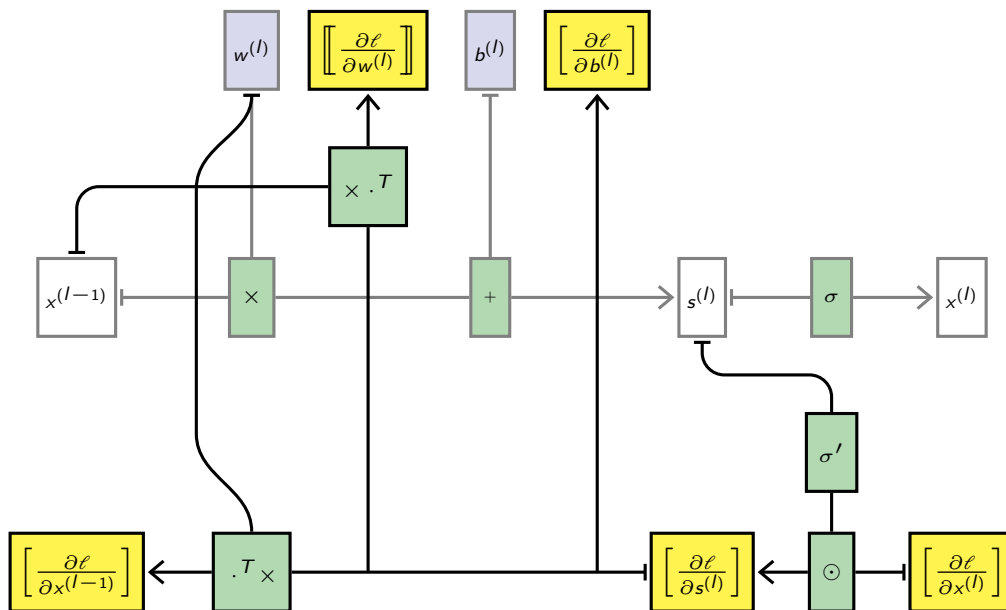
$$\left[\frac{\partial \psi}{\partial \mathbf{x}} \right] = \begin{pmatrix} \frac{\partial \psi_1}{\partial x_1} & \cdots & \frac{\partial \psi_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial \psi_M}{\partial x_1} & \cdots & \frac{\partial \psi_M}{\partial x_N} \end{pmatrix},$$

and if $\psi : \mathbb{R}^{N \times M} \rightarrow \mathbb{R}$, we will use the compact notation, also tensorial

$$\left[\left[\frac{\partial \psi}{\partial \mathbf{w}} \right] \right] = \begin{pmatrix} \frac{\partial \psi}{\partial w_{1,1}} & \cdots & \frac{\partial \psi}{\partial w_{1,M}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \psi}{\partial w_{N,1}} & \cdots & \frac{\partial \psi}{\partial w_{N,M}} \end{pmatrix}.$$

A standard notation (that we do not use here) is

$$\left[\frac{\partial \ell}{\partial \mathbf{x}^{(l)}} \right] = \nabla_{\mathbf{x}^{(l)}} \ell \quad \left[\frac{\partial \ell}{\partial \mathbf{s}^{(l)}} \right] = \nabla_{\mathbf{s}^{(l)}} \ell \quad \left[\frac{\partial \ell}{\partial \mathbf{b}^{(l)}} \right] = \nabla_{\mathbf{b}^{(l)}} \ell \quad \left[\left[\frac{\partial \ell}{\partial \mathbf{w}^{(l)}} \right] \right] = \nabla_{\mathbf{w}^{(l)}} \ell.$$



Forward pass

Compute the activations.

$$x^{(0)} = x, \quad \forall l = 1, \dots, L, \quad \begin{cases} s^{(l)} = w^{(l)}x^{(l-1)} + b^{(l)} \\ x^{(l)} = \sigma(s^{(l)}) \end{cases}$$

Backward pass

Compute the derivatives of the loss wrt the activations.

$$\begin{cases} \left[\frac{\partial \ell}{\partial x^{(L)}} \right] \text{ from the definition of } \ell & \left[\frac{\partial \ell}{\partial s^{(l)}} \right] = \left[\frac{\partial \ell}{\partial x^{(l)}} \right] \odot \sigma'(s^{(l)}) \\ \text{if } l < L, \left[\frac{\partial \ell}{\partial x^{(l)}} \right] = (w^{(l+1)})^T \left[\frac{\partial \ell}{\partial s^{(l+1)}} \right] & \end{cases}$$

Compute the derivatives of the loss wrt the parameters.

$$\left[\left[\frac{\partial \ell}{\partial w^{(l)}} \right] \right] = \left[\frac{\partial \ell}{\partial s^{(l)}} \right] (x^{(l-1)})^T \quad \left[\frac{\partial \ell}{\partial b^{(l)}} \right] = \left[\frac{\partial \ell}{\partial s^{(l)}} \right].$$

Gradient step

Update the parameters.

$$w^{(l)} \leftarrow w^{(l)} - \eta \left[\left[\frac{\partial \ell}{\partial w^{(l)}} \right] \right] \quad b^{(l)} \leftarrow b^{(l)} - \eta \left[\frac{\partial \ell}{\partial b^{(l)}} \right]$$

In spite of its hairy formalization, the backward pass is a simple algorithm: apply the chain rule again and again.

As for the forward pass, it can be expressed in tensorial form. Heavy computation is concentrated in linear operations, and all the non-linearities go into component-wise operations.

Regarding computation, since the costly operation for the forward pass is

$$s^{(l)} = w^{(l)}x^{(l-1)} + b^{(l)}$$

and for the backward

$$\left[\frac{\partial \ell}{\partial x^{(l)}} \right] = \left(w^{(l+1)} \right)^T \left[\frac{\partial \ell}{\partial s^{(l+1)}} \right]$$

and

$$\left[\frac{\partial \ell}{\partial w^{(l)}} \right] = \left[\frac{\partial \ell}{\partial s^{(l)}} \right] \left(x^{(l-1)} \right)^T,$$

the rule of thumb is that the backward pass is twice more expensive than the forward one.

References

- Y. Bengio and Y. Grandvalet. No unbiased estimator of the variance of k-fold cross-validation. Journal of Machine Learning Research (JMLR), 5:1089–1105, 2004.
- A. Brock, J. Donahue, and K. Simonyan. Large scale gan training for high fidelity natural image synthesis. CoRR, abs/1809.11096, 2018.
- A. Canziani, A. Paszke, and E. Culurciello. An analysis of deep neural network models for practical applications. CoRR, abs/1605.07678, 2016.
- N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Conference on Computer Vision and Pattern Recognition (CVPR), pages 886–893, 2005.
- P. Dollár, Z. Tu, P. Perona, and S. Belongie. Integral channel features. In British Machine Vision Conference, pages 91.1–91.11, 2009.
- K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36(4): 193–202, April 1980.
- K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. CoRR, abs/1512.03385, 2015.
- A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto, 2009.
- A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In Neural Information Processing Systems (NIPS), 2012.

- A. Kumar, O. Irsoy, J. Su, J. Bradbury, R. English, B. Pierce, P. Ondruska, I. Gulrajani, and R. Socher. Ask me anything: Dynamic memory networks for natural language processing. CoRR, abs/1506.07285, 2015.
- Y. leCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.
- W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.
- V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, Feb. 2015.
- P. O. Pinheiro, T.-Y. Lin, R. Collobert, and P. Dollár. Learning to refine object segments. In European Conference on Computer Vision (ECCV), pages 75–91, 2016.
- A. Radford, J. Wu, D. Amodei, D. Amodei, J. Clark, M. Brundage, and I. Sutskever. Better language models and their implications. web, February 2019. <https://blog.openai.com/better-language-models/>.
- F. Rosenblatt. The perceptron—A perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.
- D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of Research, chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, 1988.
-
- D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. Nature, 529:484–503, 2016.
- C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Conference on Computer Vision and Pattern Recognition (CVPR), 2015.
- V. N. Vapnik. The Nature of Statistical Learning Theory. Springer-Verlag, New York, 1995.
- O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In Conference on Computer Vision and Pattern Recognition (CVPR), 2015.
- S. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh. Convolutional pose machines. CoRR, abs/1602.00134, 2016.
- Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google’s neural machine translation system: Bridging the gap between human and machine translation. CoRR, abs/1609.08144, 2016.
- S. Yeung, O. Russakovsky, G. Mori, and L. Fei-Fei. End-to-end learning of action detection from frame glimpses in videos. CoRR, abs/1511.06984, 2015.