Deep learning

1.1. From neural networks to deep learning

François Fleuret

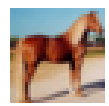
https://fleuret.org/dlc/

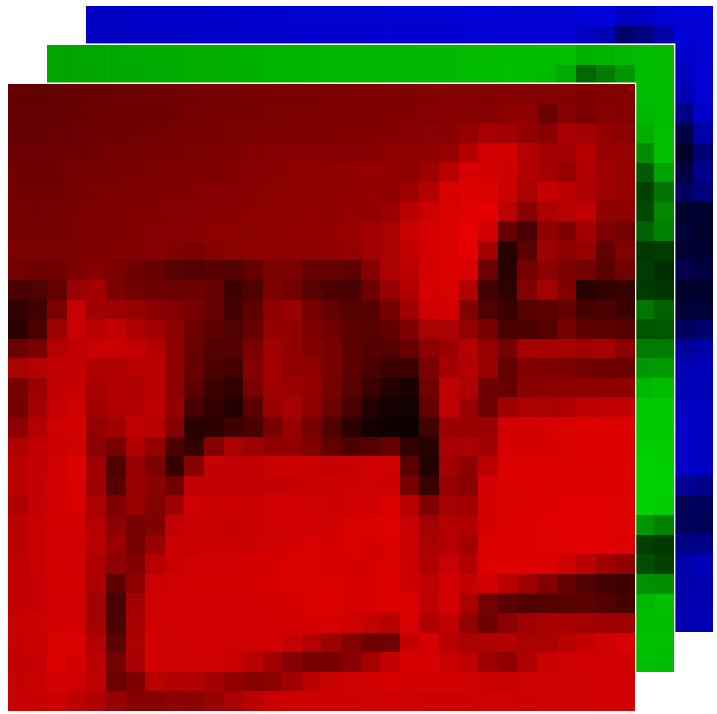
UNIVERSITÉ
DE GENÈVE

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.data)[43].permute(2, 0, 1)
>>> 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.

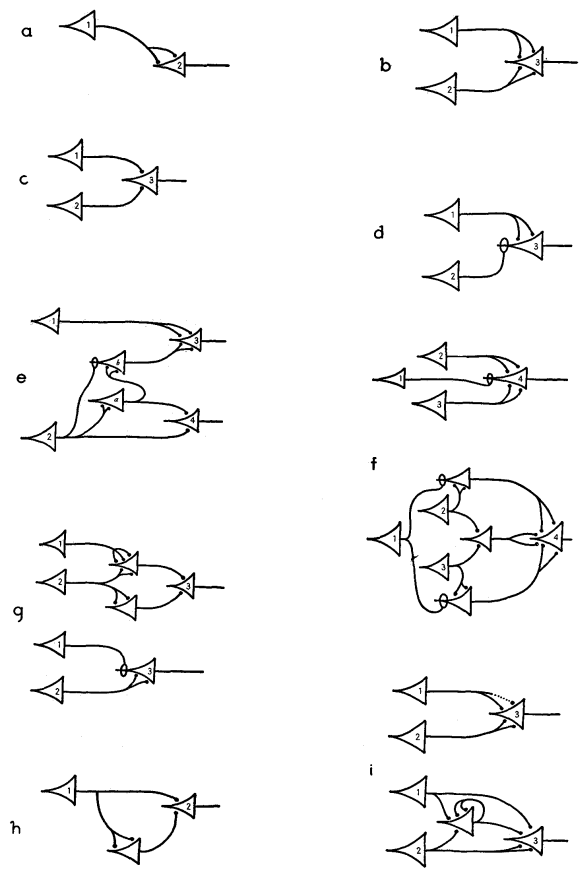
There are strong connections between standard statistical modeling and machine learning.

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

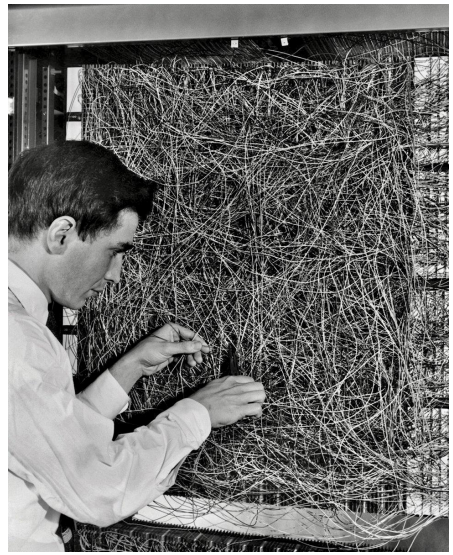
"Artificial neural networks" pre-dated these approaches, and do not follow this dichotomy. They consist of "deep" stacks of parametrized processing.

From artificial neural networks to "Deep Learning"

Networks of "Threshold Logic Unit"



(McCulloch and Pitts, 1943)



Frank Rosenblatt working on the Mark I perceptron (1956)

1949 – Donald Hebb proposes the Hebbian Learning principle (Hebb, 1949).

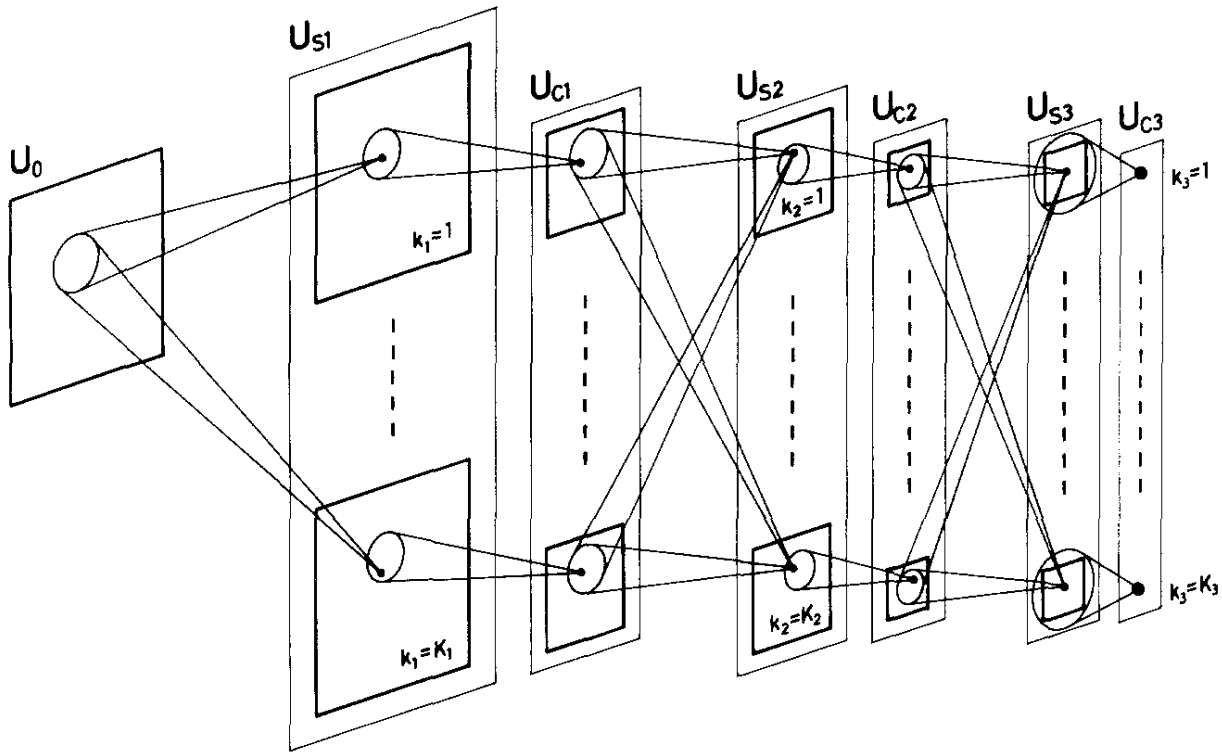
1951 – Marvin Minsky creates the first ANN (Hebbian learning, 40 neurons).

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 (Hubel and Wiesel, 1962).

1982 – Paul Werbos proposes back-propagation for ANNs (Werbos, 1981).

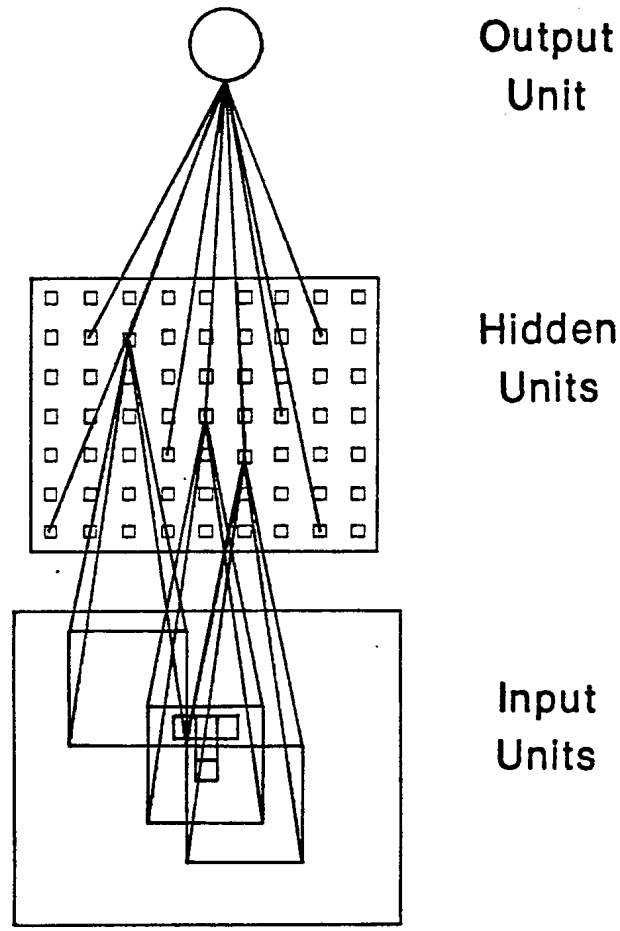
Neocognitron



(Fukushima, 1980)

This model follows Hubel and Wiesel's results.

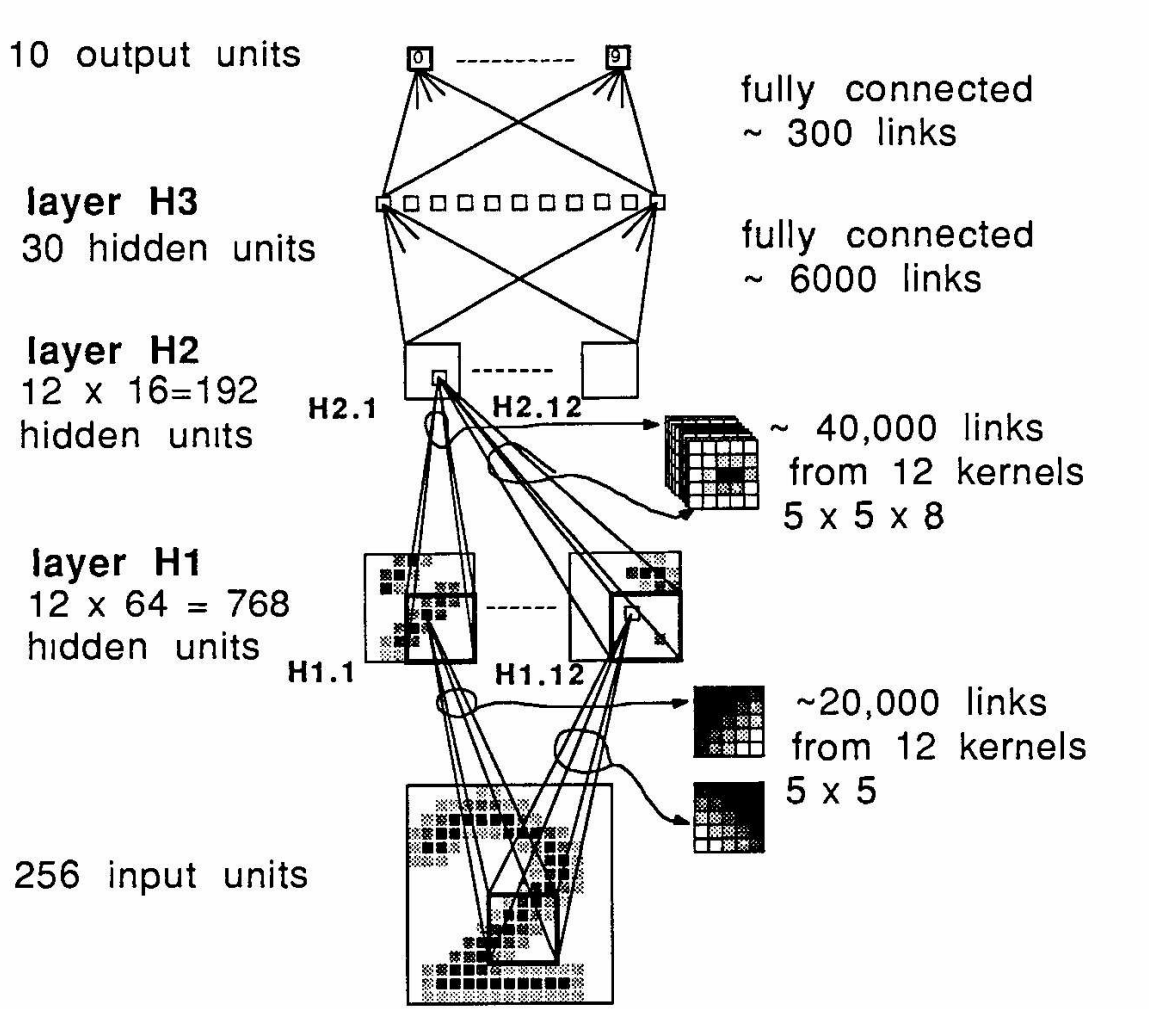
Network for the T-C problem



Trained with back-prop.

(Rumelhart et al., 1988)

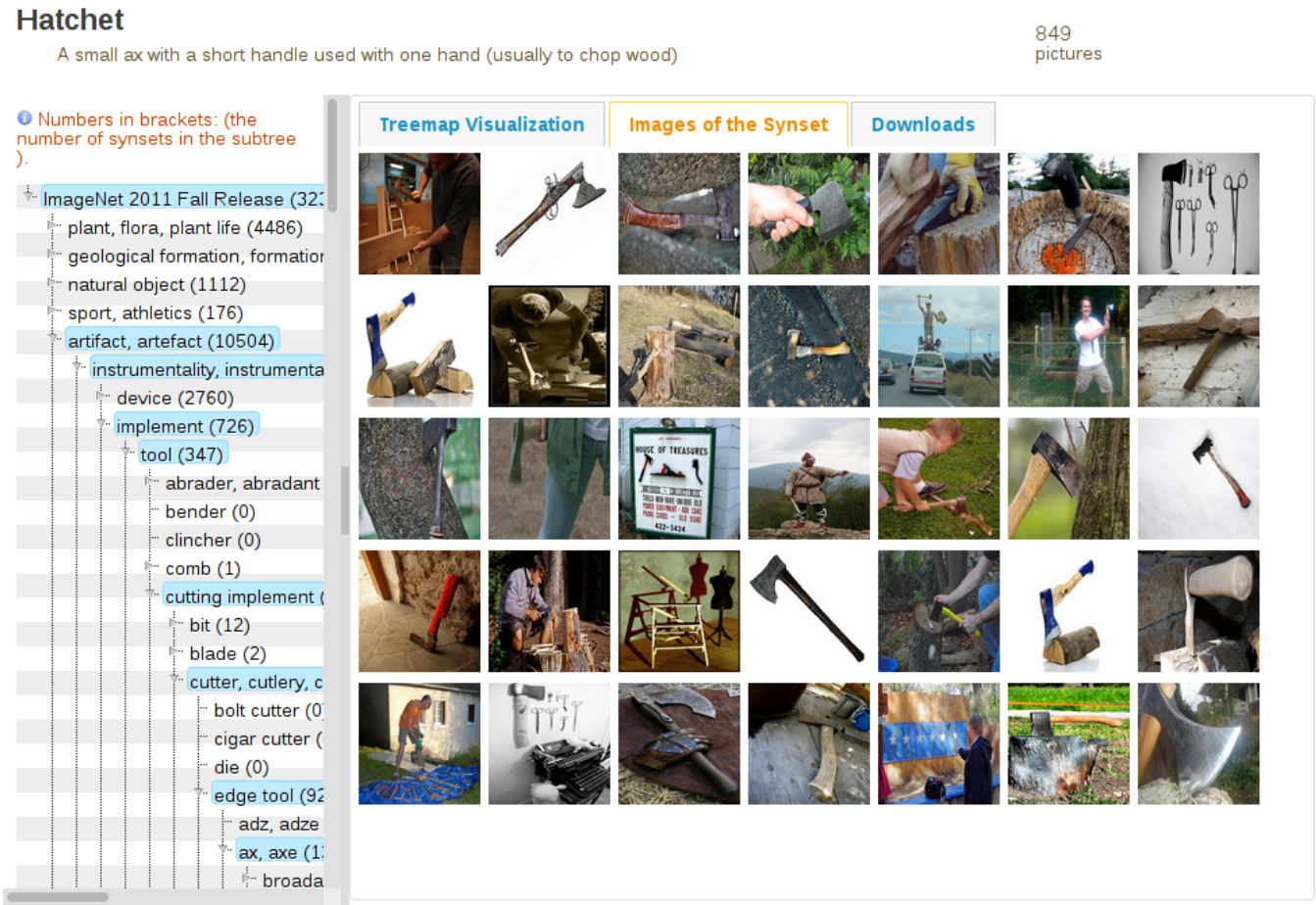
LeNet family



(LeCun et al., 1989)

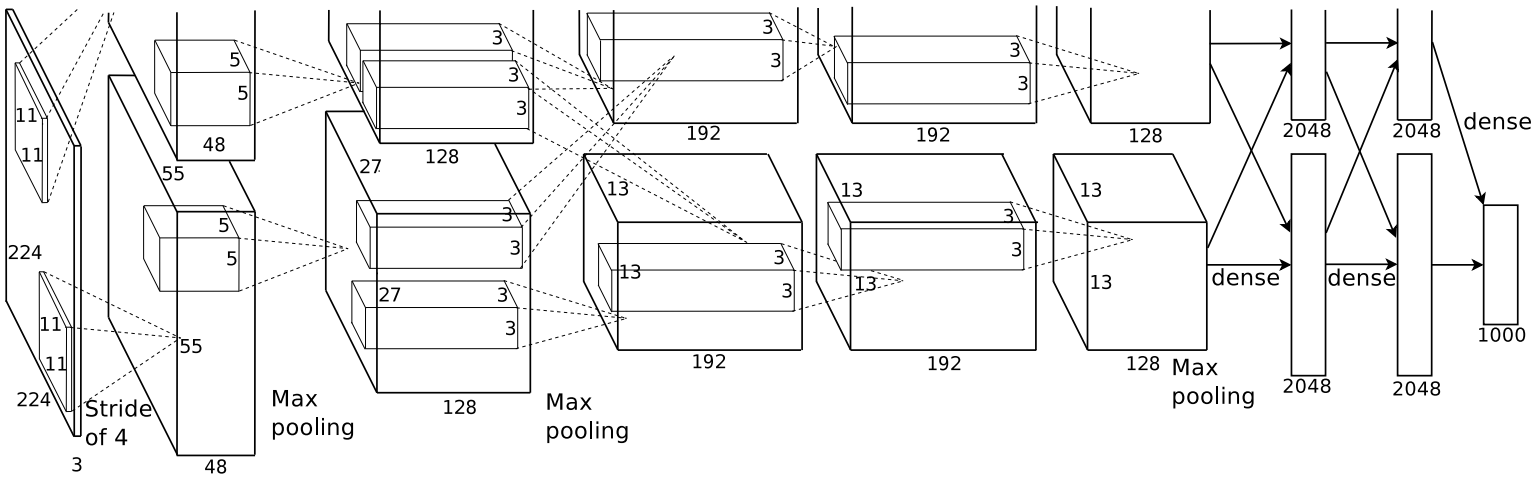
ImageNet Large Scale Visual Recognition Challenge.

Started 2010, 1 million images, 1000 categories



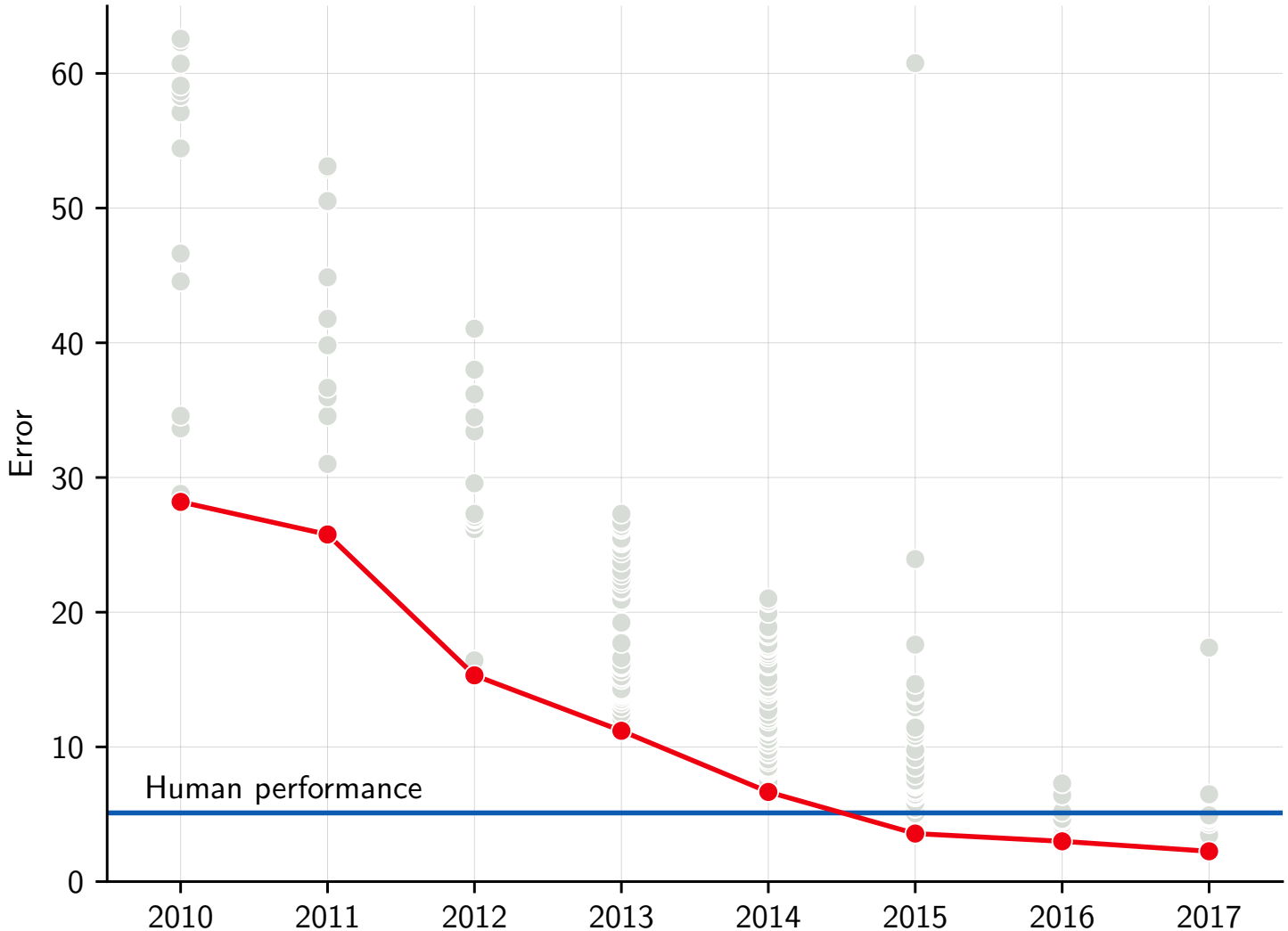
(http://image-net.org/challenges/LSVRC/2014/browse-synsets)

AlexNet

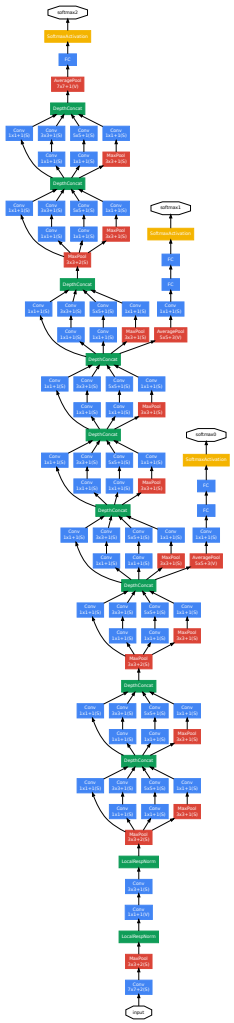


(Krizhevsky et al., 2012)

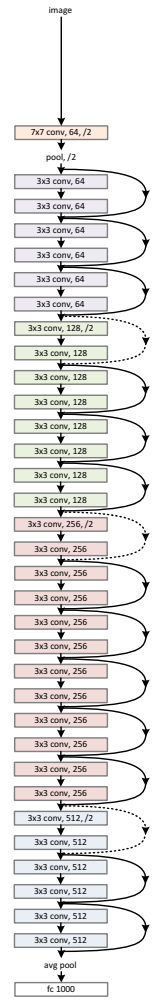
Top-5 error rate on ImageNet



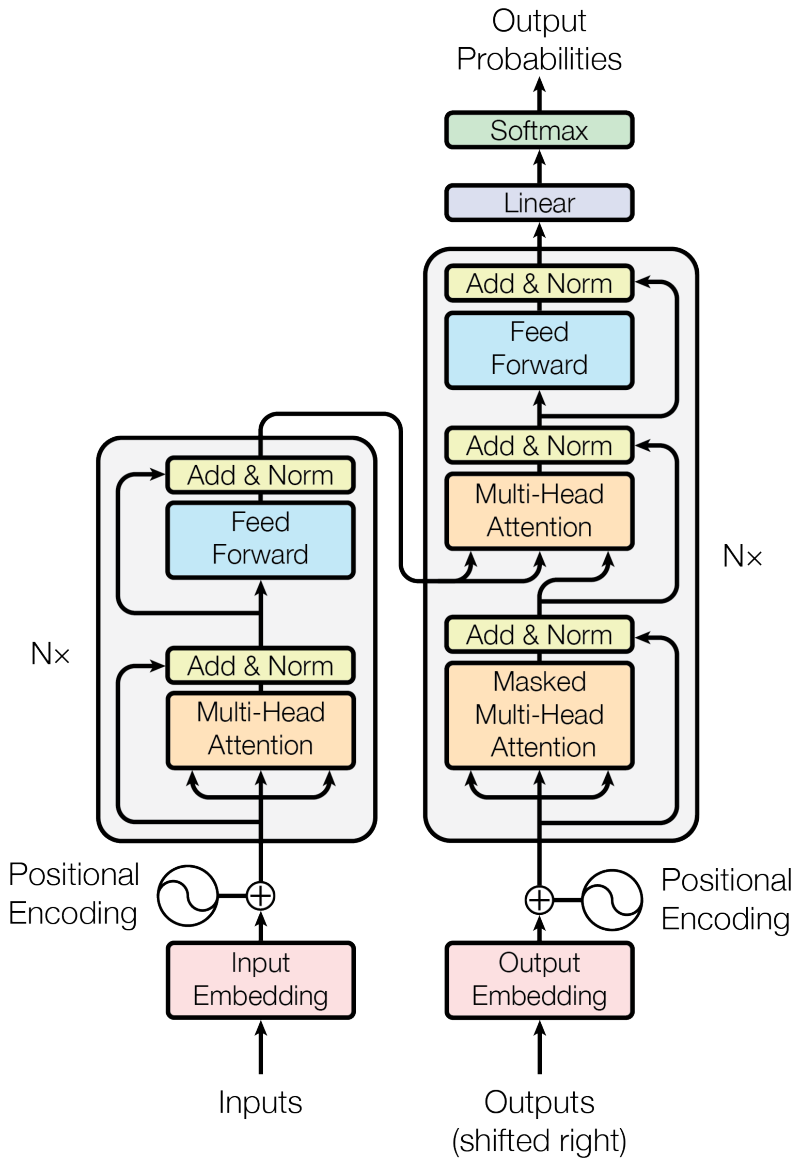
(Gershgorn, 2017)



GoogleNet (Szegedy et al., 2015)



ResNet (He et al., 2015)



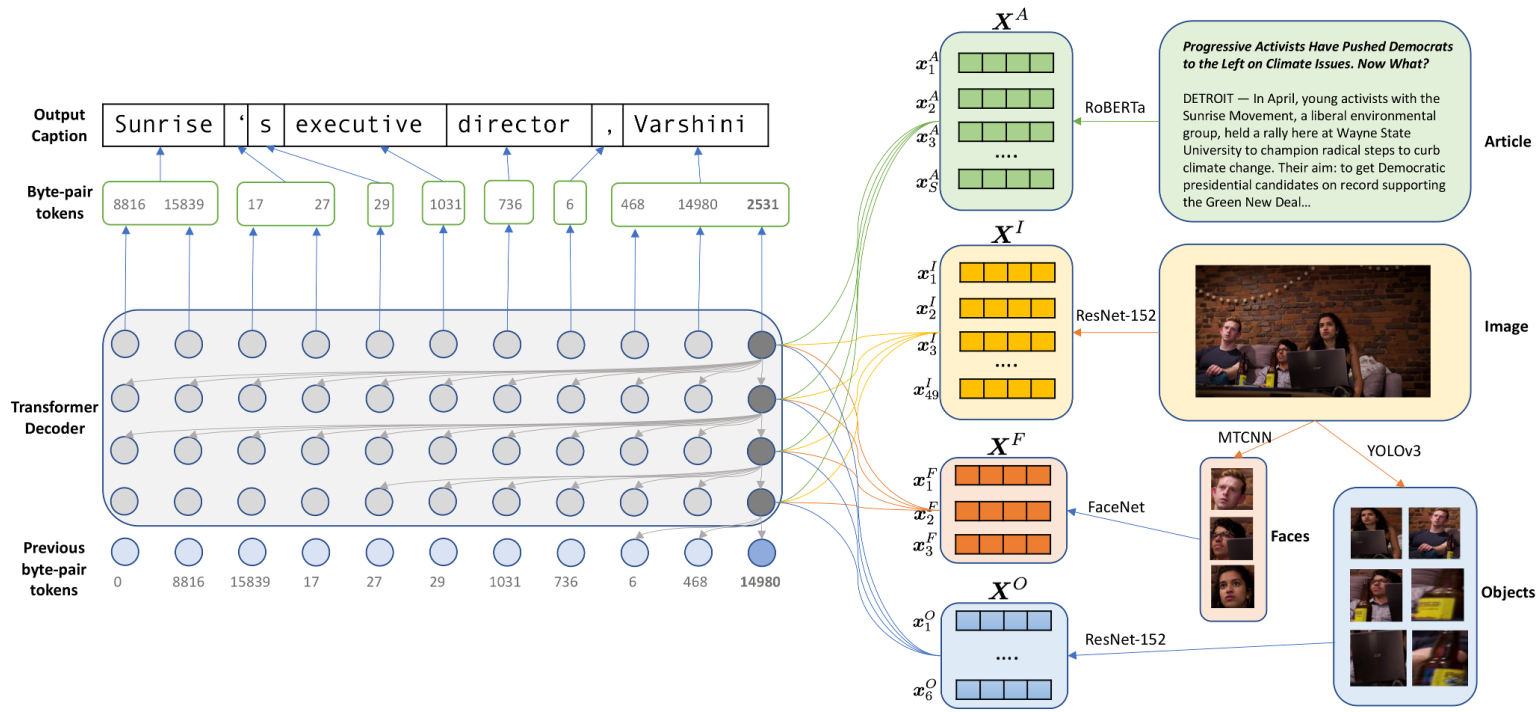
(Vaswani et al., 2017)

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.



(Tran et al., 2020)

Deep learning

1.2. Current applications and success

François Fleuret

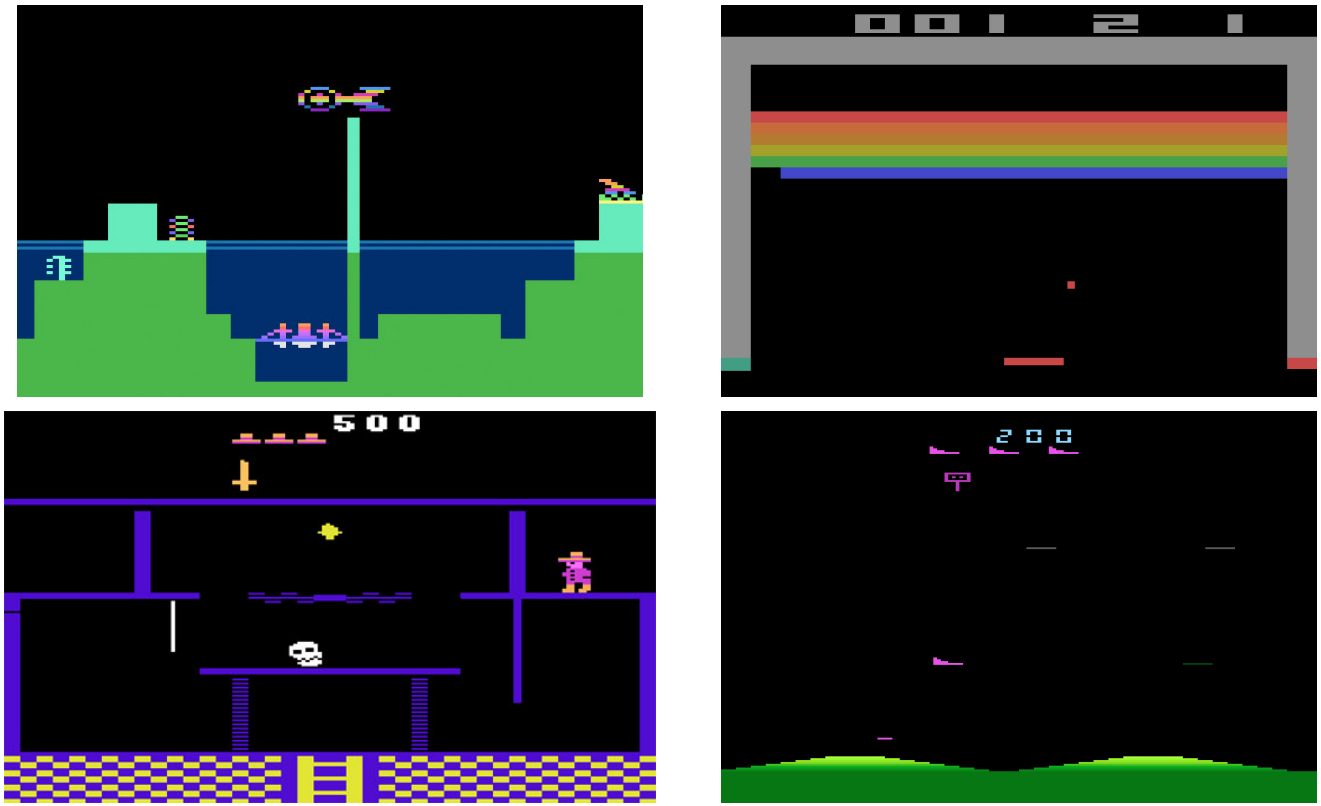
https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Object detection and segmentation

(Pinheiro et al., 2016)

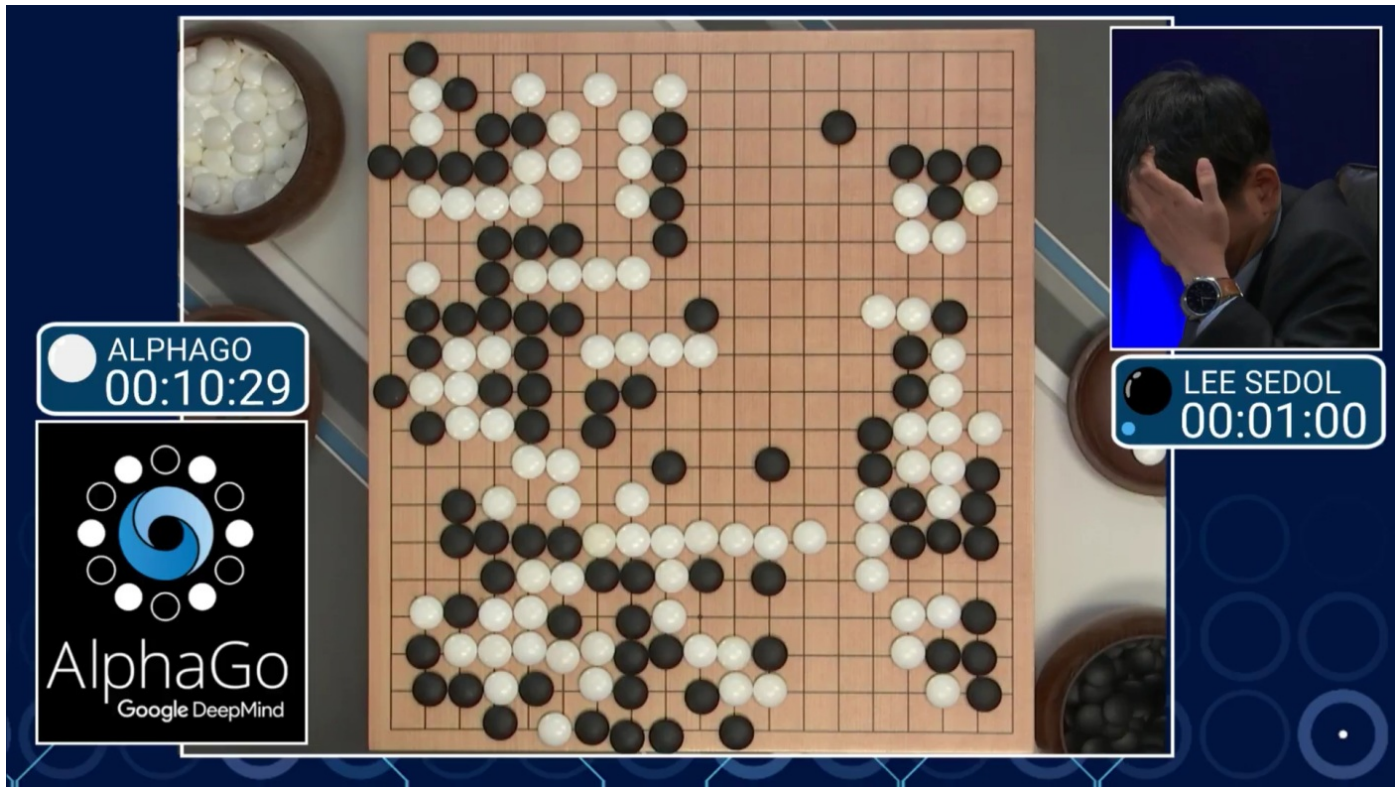
Reinforcement learning



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"

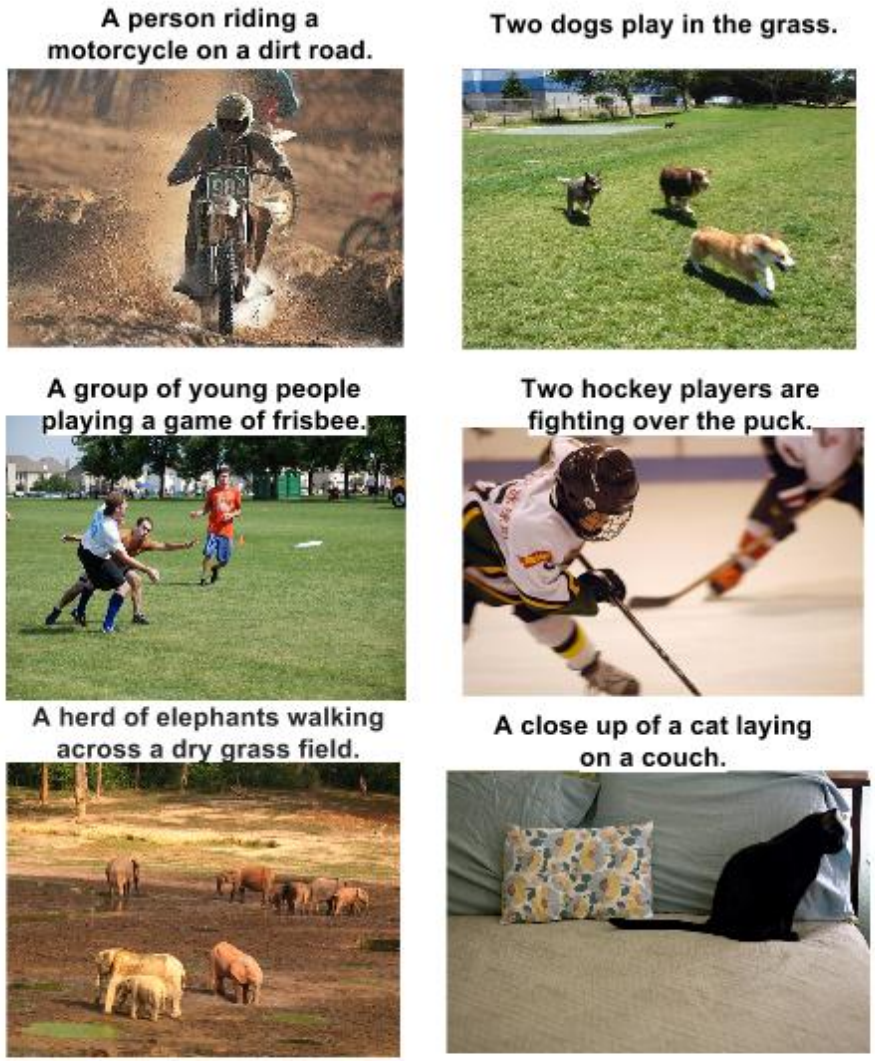
(Wu et al., 2016)

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)

Auto-captioning



(Vinyals et al., 2015)

Image generation

(Brock et al., 2018)

Text generation

System Prompt (human-written)

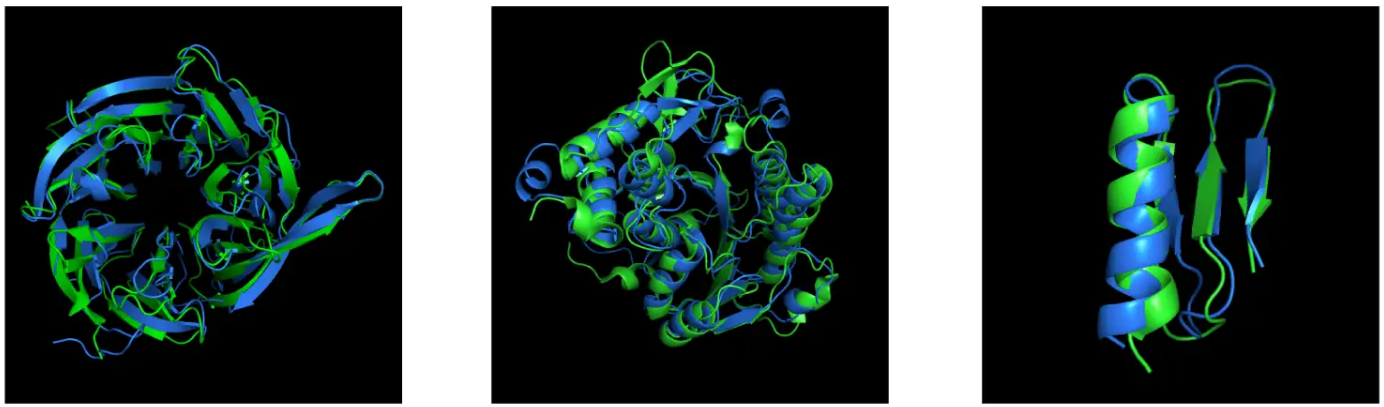
```
In a shocking finding, scientist discovered a herd of unicorns living
in a remote, previously unexplored valley, in the Andes
Mountains. Even more surprising to the researchers was the fact that
the unicorns spoke perfect English.
```

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.
```

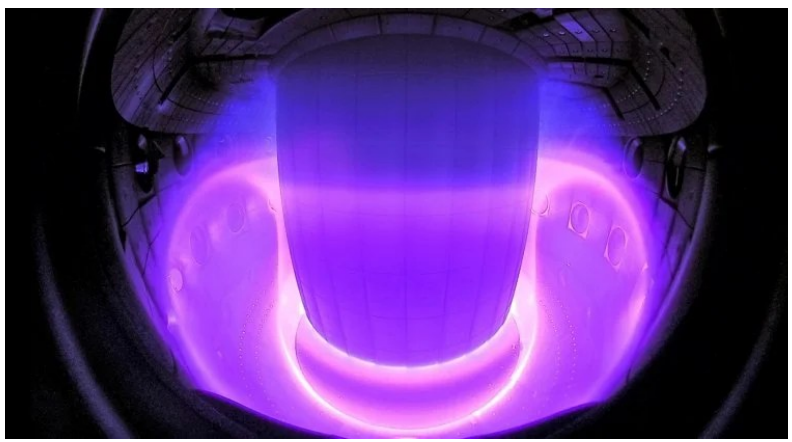
(Radford et al., 2019)

Protein folding prediction



(Jumper et al., 2021)

Plasma confinement



(Degrave et al., 2022)

Why does it work now?

The success of deep learning is multi-factorial:

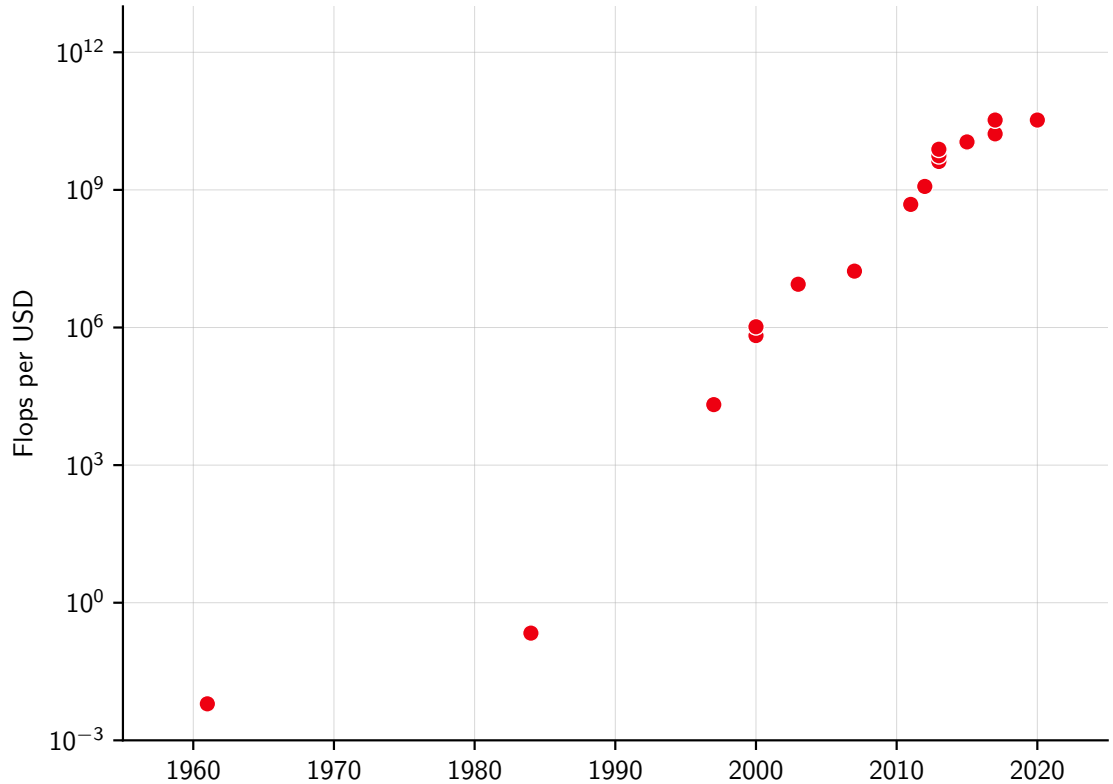
- Five decades of research in machine learning,
- CPUs/GPUs/storage developed for other purposes,
- lots of data from "the internet",
- tools and culture of collaborative and reproducible science,
- resources and efforts from large corporations.

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

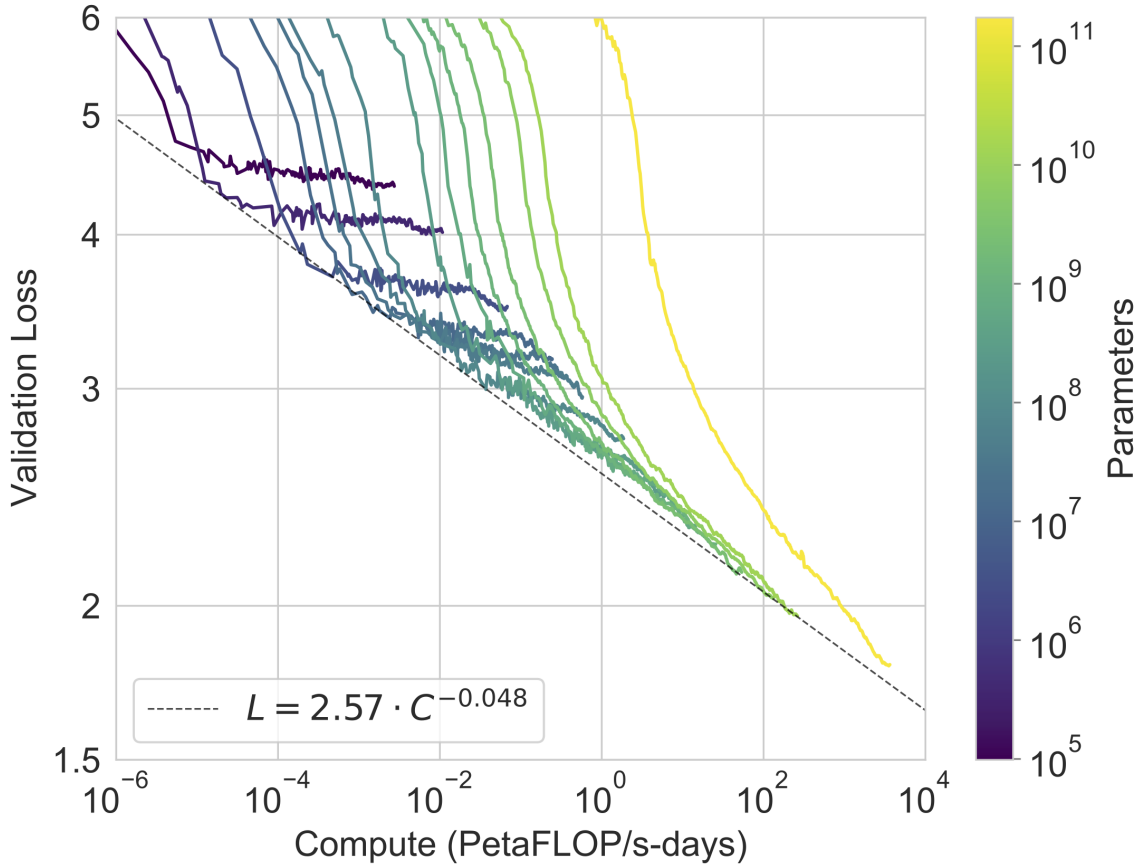
- lessens the need for a deep mathematical grasp,
- makes the design of large learning architectures a system/software development task,
- allows to leverage modern hardware (clusters of GPUs),
- does not plateau when using more data,
- makes large trained networks a commodity.



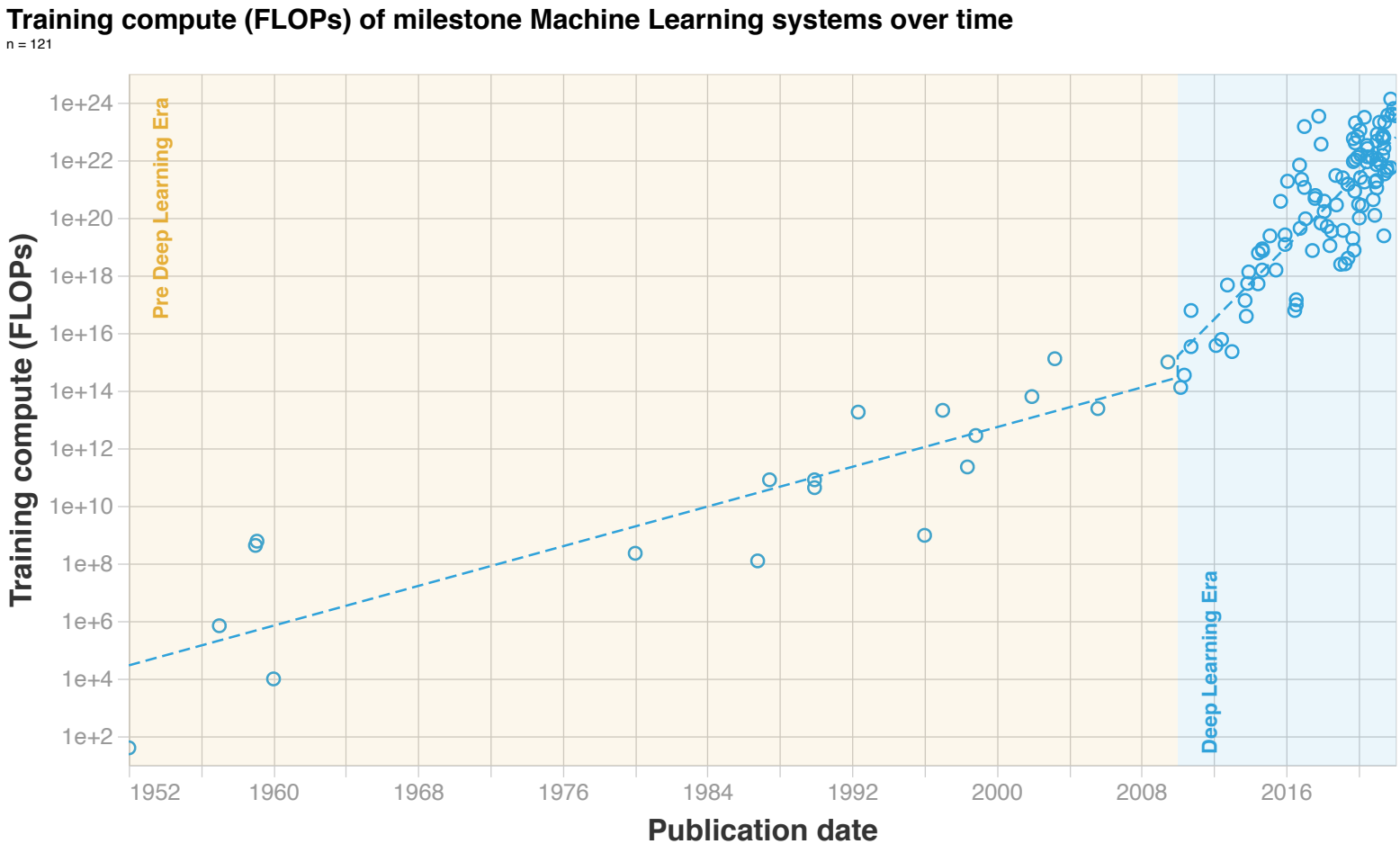
(Wikipedia "FLOPS")

	TFlops (10^{12})	**Price**	**GFlops per $**
Intel Core i7-6700K	0.2	$275	0.7
Intel Core i9-7980XE	0.9	$1'999	0.5
AMD Ryzen 7 PRO 4750G	1.1	$640	1.7
NVIDIA GTX 2080 Ti	14.2	$999	14.2
NVIDIA GTX 3090	35.5	$1'500	23.7
AMD Radeon RX 6900 XT	23.0	$999	23.0

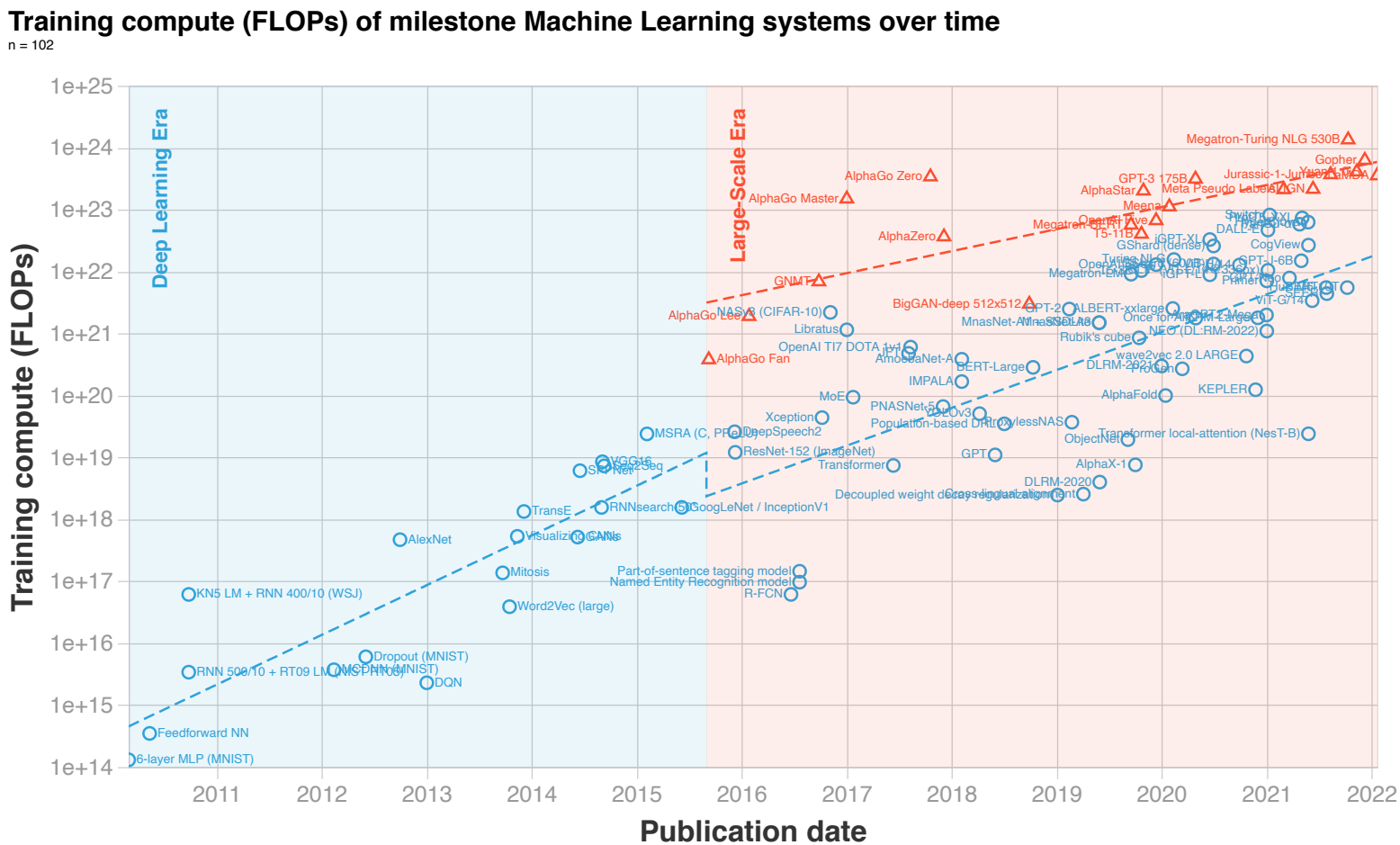
Validation loss for language models vs. training compute.



(Brown et al., 2020)



(Sevilla et al., 2022)



(Sevilla et al., 2022)

Computer vision

Data-set		**Year**	**Nb. images**	**Size**
MNIST	(classification)	1998	60K	12Mb
Caltech 101	(classification)	2003	9.1K	130Mb
Caltech 256	(classification)	2007	30K	1.2Gb
CIFAR10	(classification)	2009	60K	160Mb
ImageNet	(classification)	2012	1.2M	150Gb
MS-COCO	(segmentation)	2015	200K	32Gb
Cityscape	(segmentation)	2016	25K	60Gb
LAION-5B	(multi-modal)	2022	5.85B	240Tb

Natural Language Processing

Data-set		**Year**	**Size**
SST2	(sentiment analysis)	2013	20Mb
WMT-18	(translation)	2018	7Gb
OSCAR	(language model)	2020	6Tb

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.

(Richard Sutton, 2019)

Quantity has a Quality All Its Own.

(Thomas A. Callaghan Jr., 1979)

Implementing a deep network, PyTorch

Deep-learning development is usually done in a framework:

	Language(s)	**License**	**Main backer**
PyTorch	**Python**, C++	BSD	Facebook
TensorFlow	Python, C++	Apache	Google
JAX	Python	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. Python has an incredible ecosystem and is used across fields.

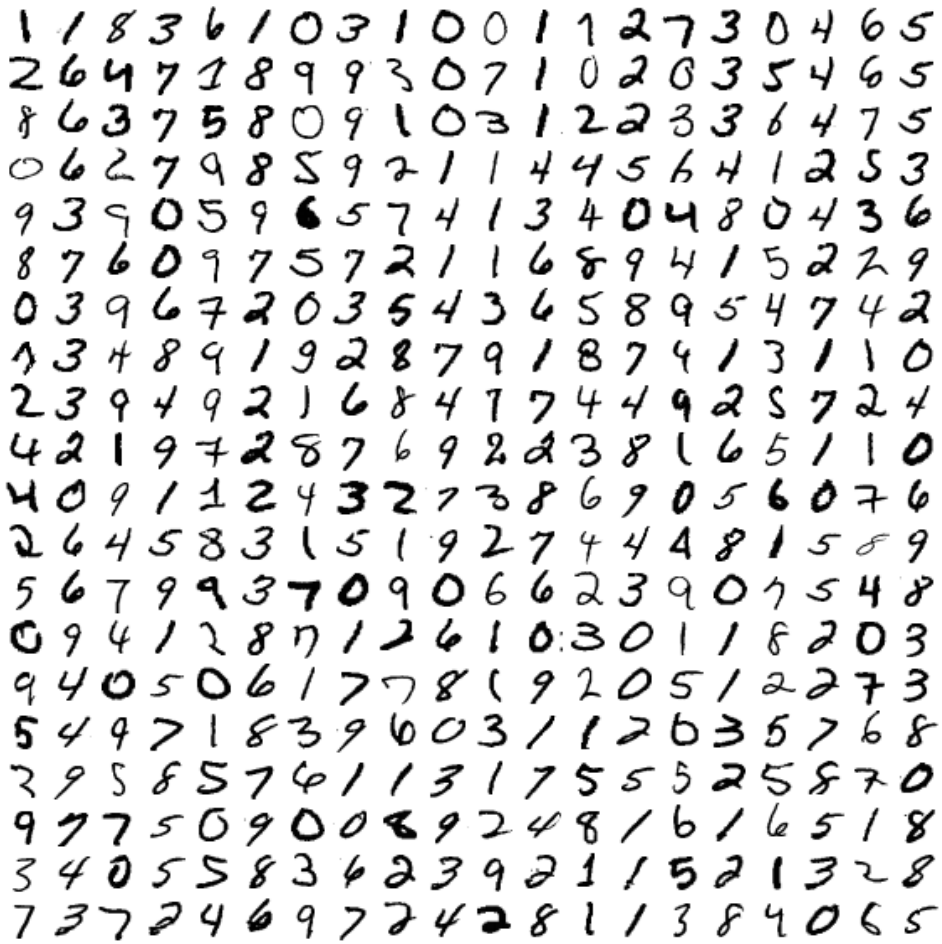
We will use the PyTorch framework for our experiments (Paszke et al., 2019).

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"*

MNIST data-set

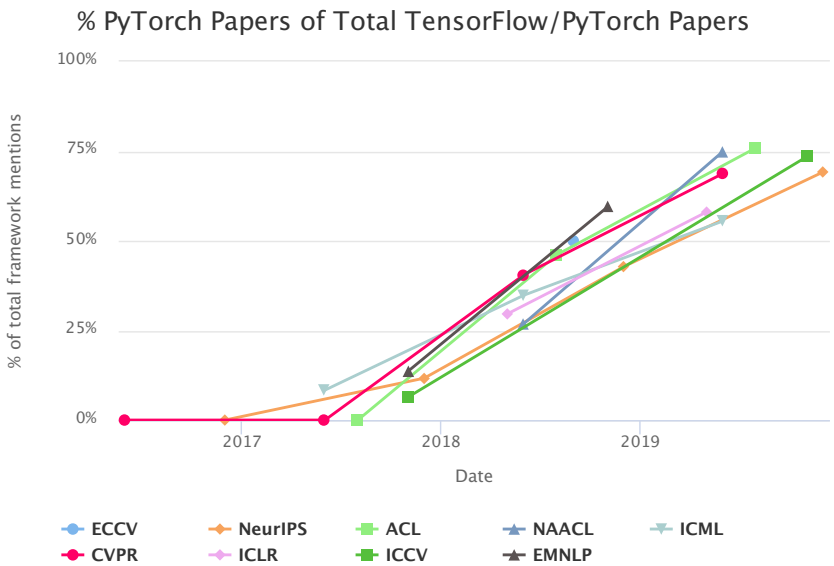


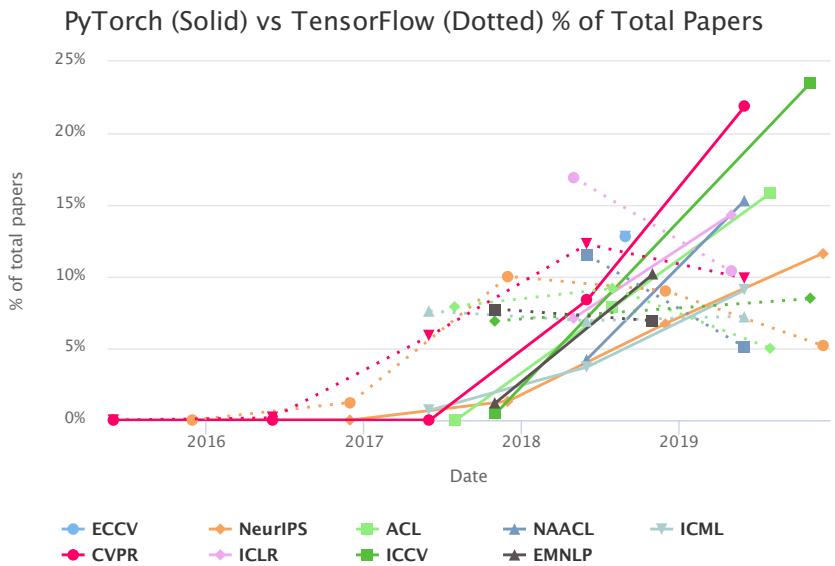
28×28 grayscale images, 60K train samples, 10K test samples.

(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(),
    nn.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_targets = train_input.to(device), train_targets.to(device)

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

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

$\simeq$8s on a GTX1080, $\simeq$1% test error


% PyTorch Papers of Total TensorFlow/PyTorch Papers
% of total framework mentions
100%
75%
50%
25%
0%
2017
2018
2019
Date
ECCV
CVPR
NeurIPS
ICLR
ACL
ICCV
NAACL
EMNLP
ICML


PyTorch (Solid) vs TensorFlow (Dotted) % of Total Papers
% of total papers
25%
20%
15%
10%
5%
0%
2016
2017
2018
2019
Date
ECCV
CVPR
NeurIPS
ICLR
ACL
ICCV
NAACL
EMNLP
ICML

(He, 2019)

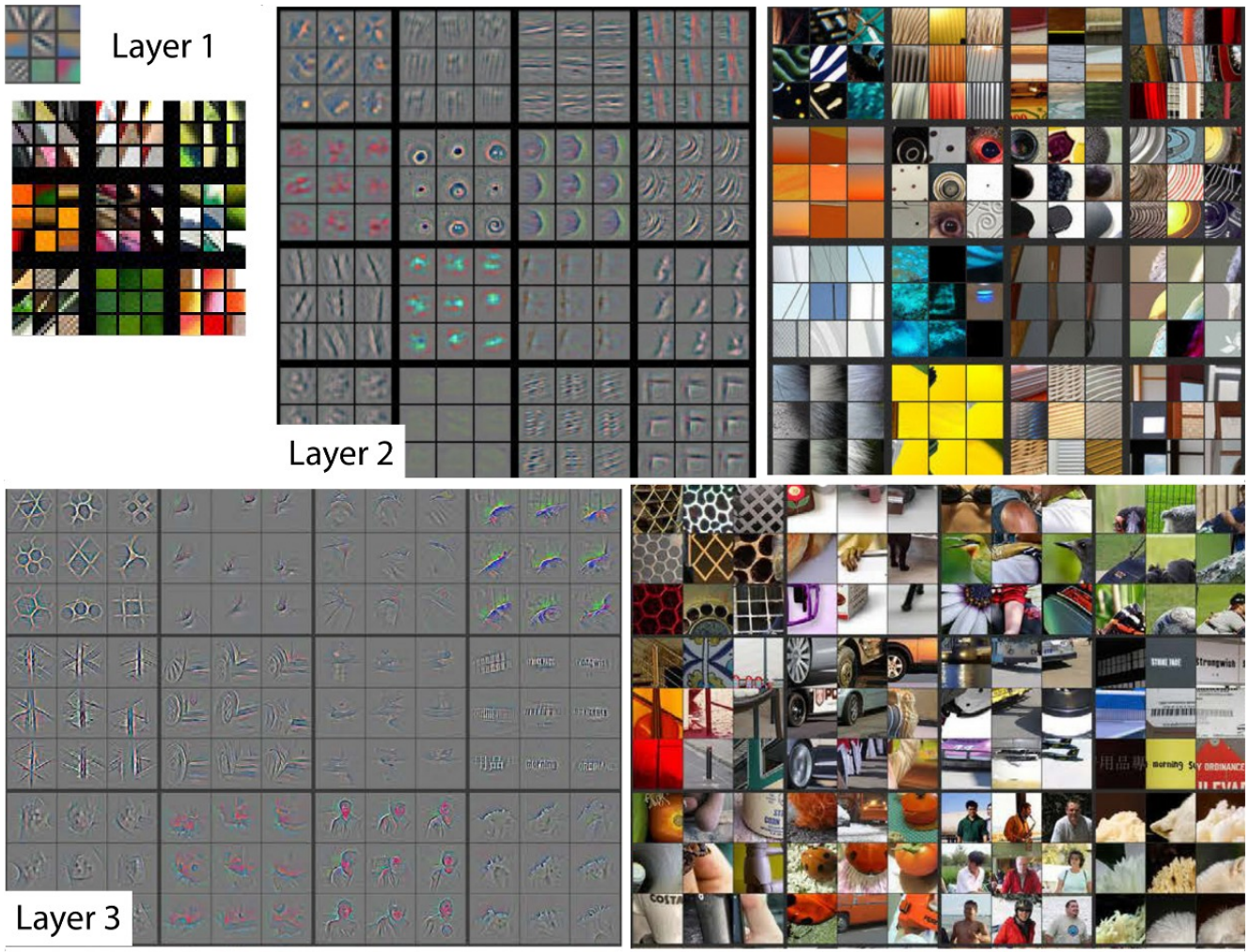
Deep learning

1.3. What is really happening?

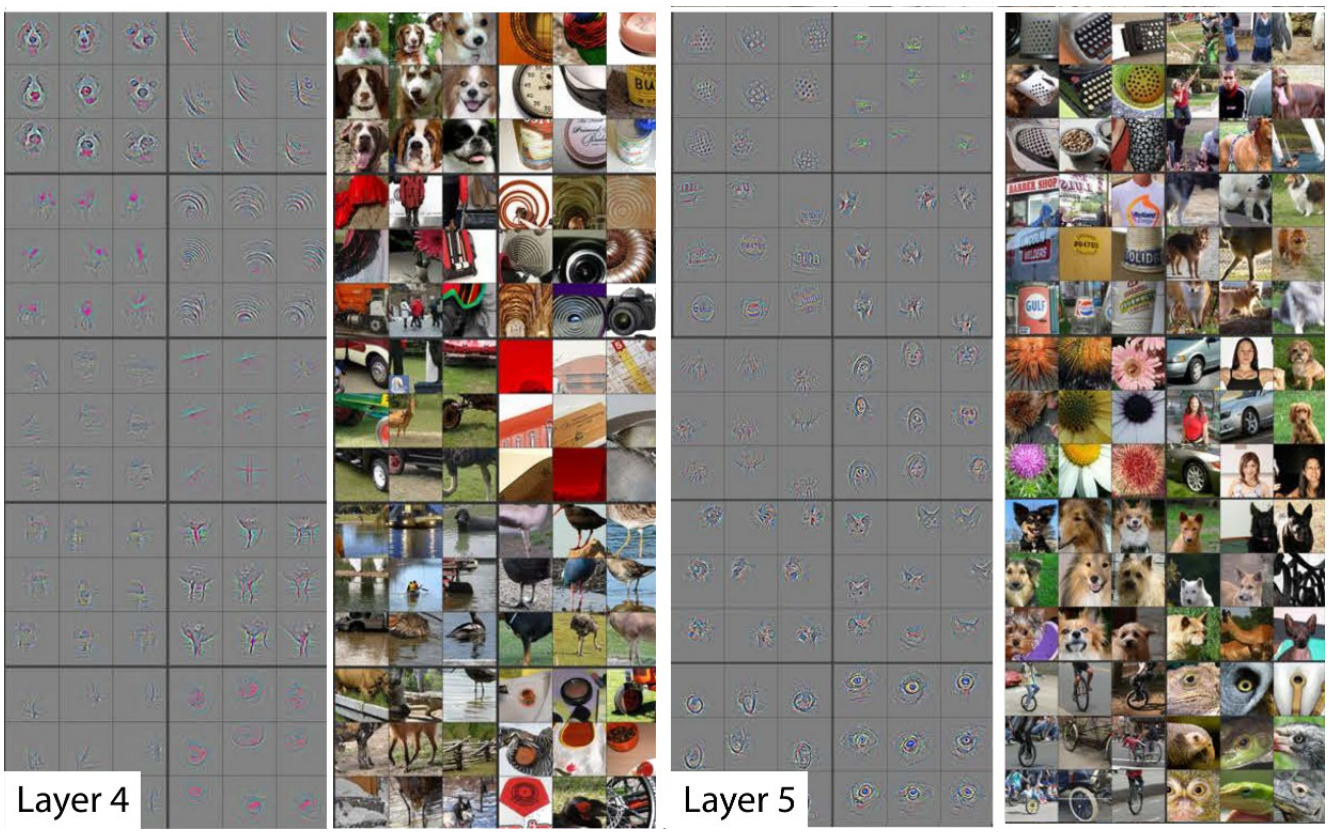
François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

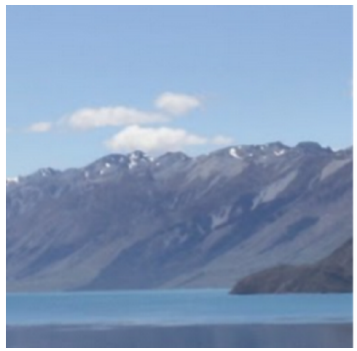


(Zeiler and Fergus, 2014)

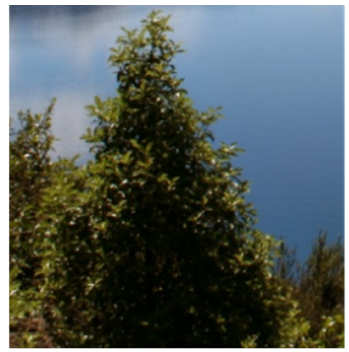


(Zeiler and Fergus, 2014)

(Google's Deep Dreams)



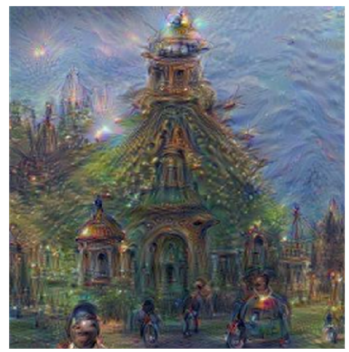
Horizon



Trees

Leaves

Towers & Pagodas

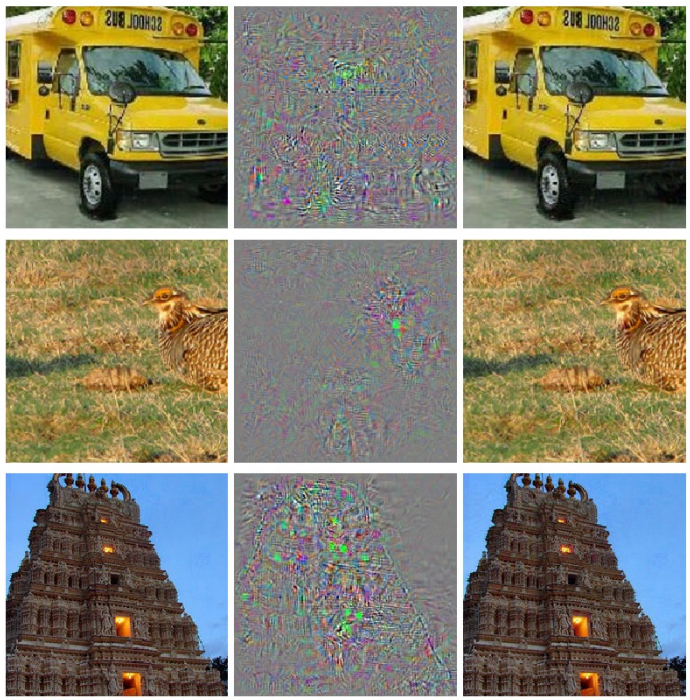


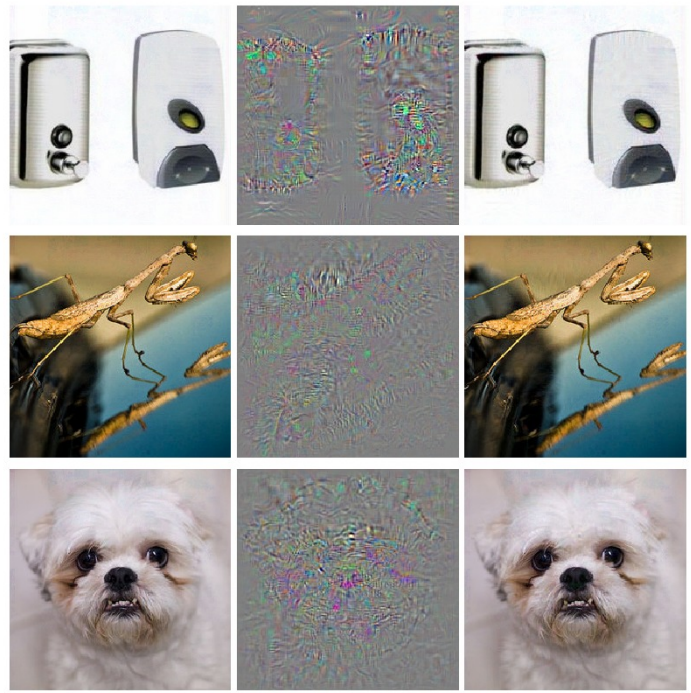
Buildings

Birds & Insects

(Google's Deep Dreams)

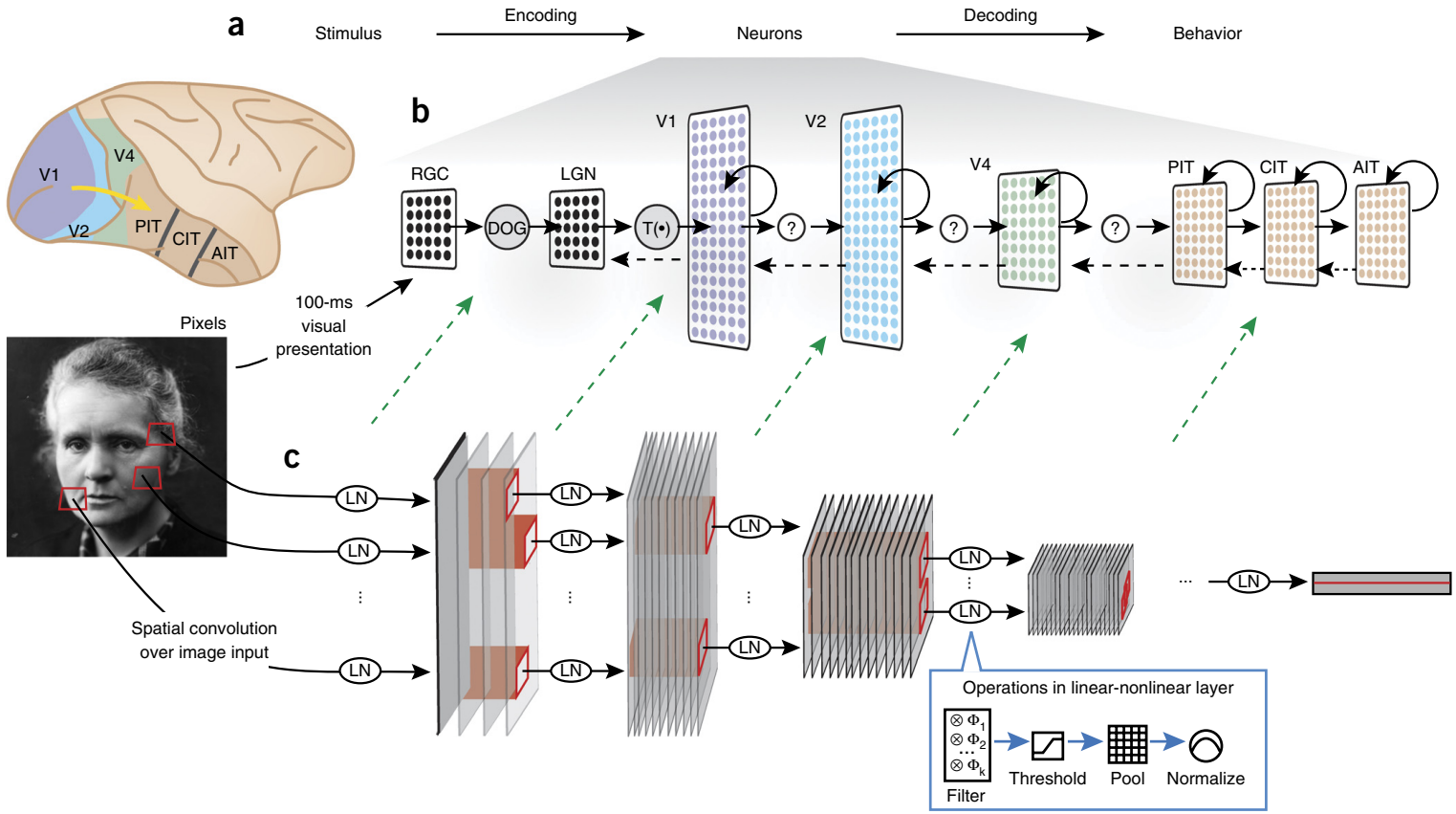
(Thorne Brandt)



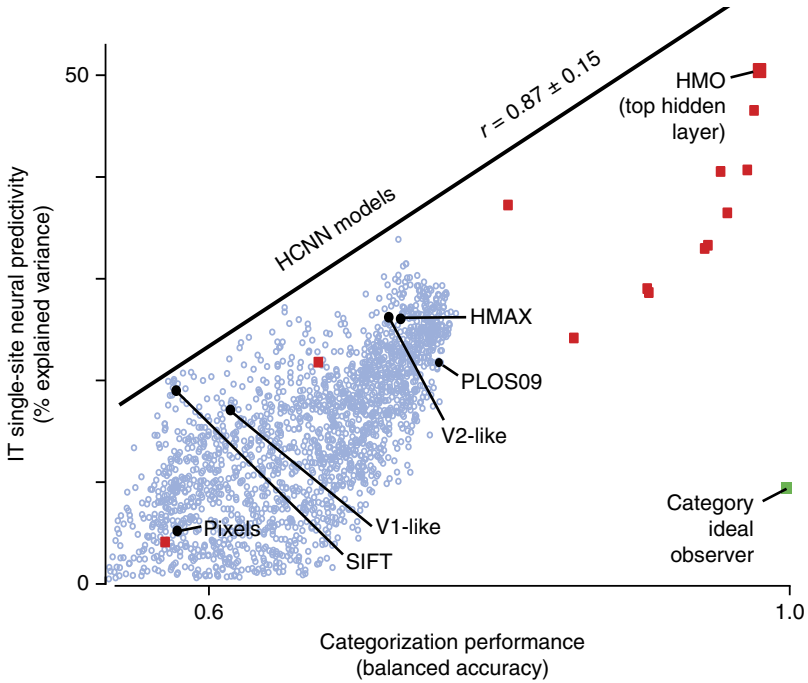


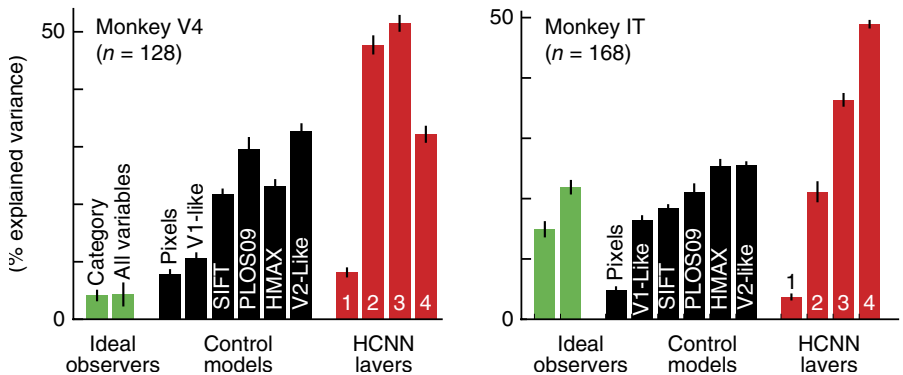
(Szegedy et al., 2014)

Relations with the biology

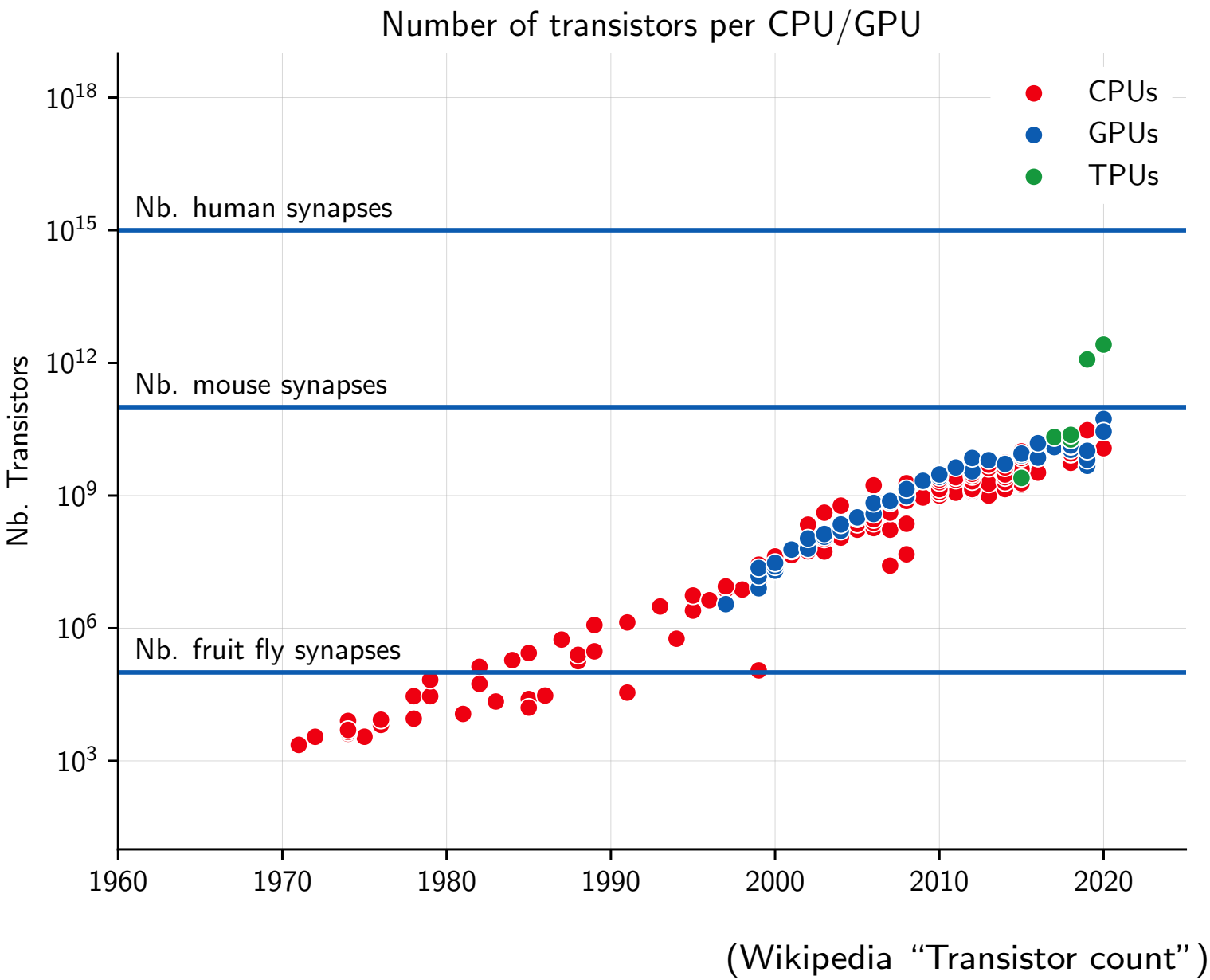


(Yamins and DiCarlo, 2016)





(Yamins and DiCarlo, 2016)


Number of transistors per CPU/GPU
CPUs
GPUs
TPUs
Nb. human synapses
Nb. mouse synapses
Nb. fruit fly synapses
Nb. Transistors
10^{18}
10^{15}
10^{12}
10^{9}
10^{6}
10^{3}
1960
1970
1980
1990
2000
2010
2020
(Wikipedia "Transistor count")

Deep learning

1.4. Tensor basics and linear regression

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

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 matrices, 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. Compounded data structures can represent more diverse data types.

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

The "dimension" of a vector in linear algebra is its number of coefficients, while the "dimension" of a tensor is the number of indices to specify one of its coefficients.

E.g. an element of $\mathbb{R}^3$ is a three-dimension vector, but a one-dimension tensor.

PyTorch's 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 returns 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.randint(10, (2, 4))
>>> x
tensor([[8, 7, 6, 6],
        [5, 0, 4, 8]])
>>> x[0]
tensor([8, 7, 6, 6])
>>> x[0, :]
tensor([8, 7, 6, 6])
>>> x[:, 0]
tensor([8, 5])
>>> x[:, 1:3] = -1
>>> x
tensor([[ 8, -1, -1,  6],
        [ 5, -1, -1,  8]])
```

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

```
>>> y = torch.randn(3)
>>> y
tensor([ 1.3663, -0.5444, -1.7488])
>>> m = torch.randn(3, 3)
>>> q = torch.linalg.lstsq(m, y).solution
>>> m@q
tensor([ 1.3663, -0.5444, -1.7488])
```

Example: linear regression

Given a list of points

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

can we find the affine function

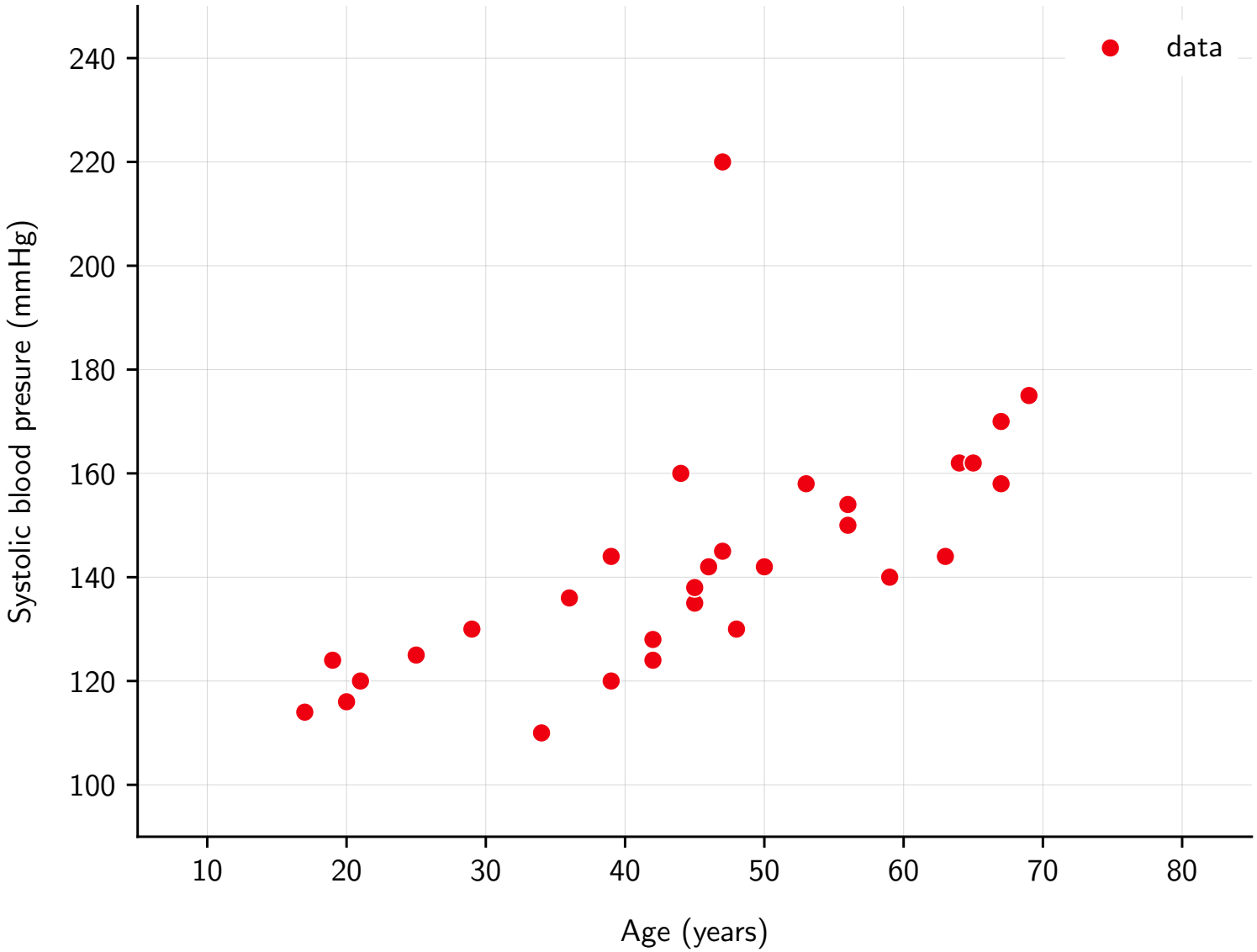
$$f(x; a, b) = ax + b$$

that "goes best through the points", e.g. minimizes the mean square error

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

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

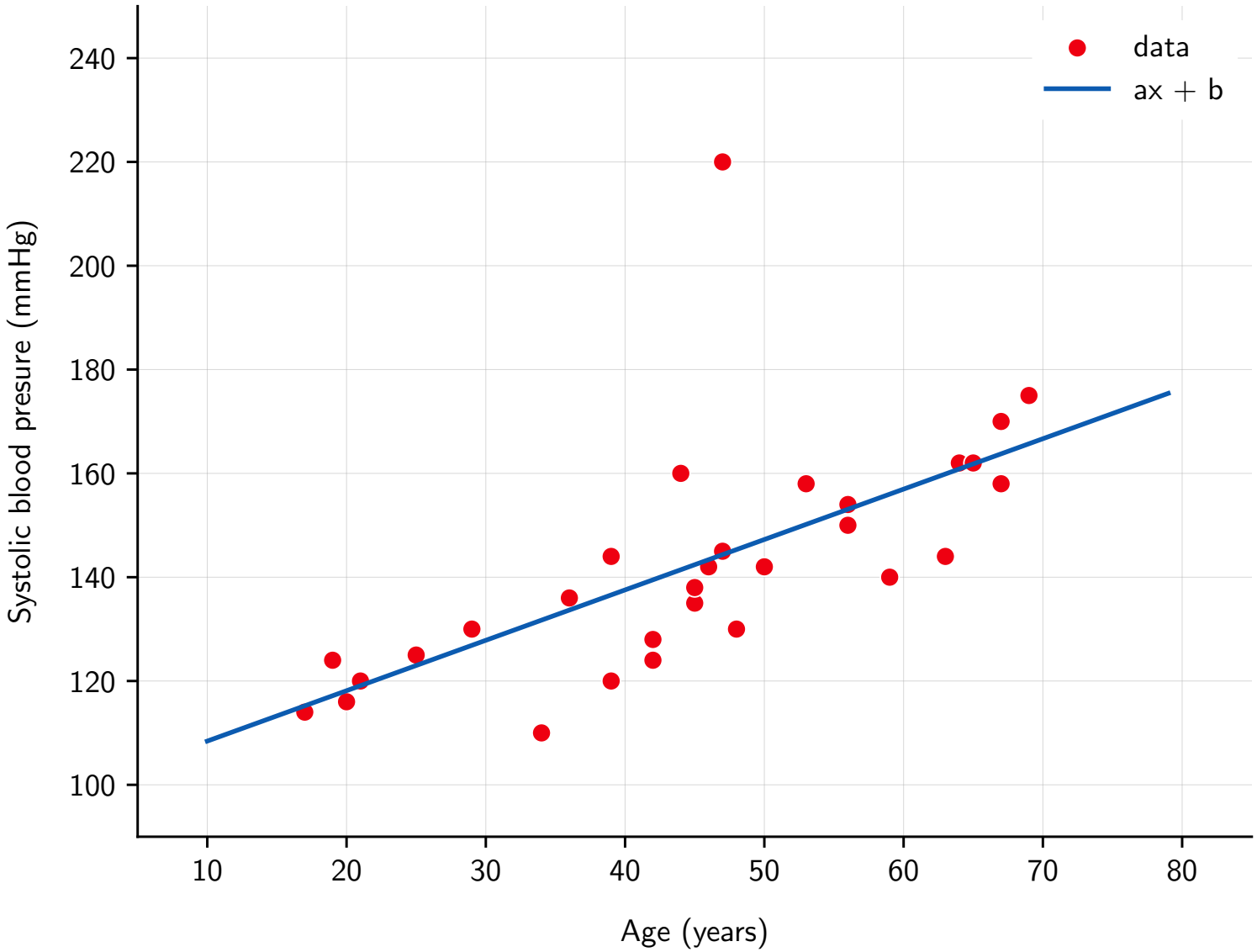
```
bash> cat systolic-blood-pressure-vs-age.dat
39  144
47  220
45  138
47  145
65  162
46  142
67  170
42  124
67  158
56  154
64  162
56  150
59  140
34  110
42  128
/.../
```


data
Systolic blood presure (mmHg)
240
220
200
180
160
140
120
100
10
20
30
40
50
60
70
80
Age (years)

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}}$$

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

```
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.linalg.lstsq(x, y).solution

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


data
ax + b
Systolic blood presure (mmHg)
240
220
200
180
160
140
120
100
10
20
30
40
50
60
70
80
Age (years)

Deep learning

1.5. High dimension tensors

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

A tensor can be of several types:

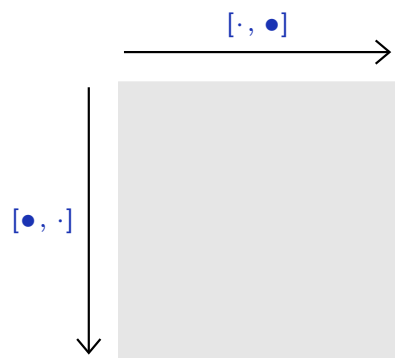
- `torch.float16`, `torch.float32`, `torch.float64`,
- `torch.uint8`,
- `torch.int8`, `torch.int16`, `torch.int32`, `torch.int64`

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

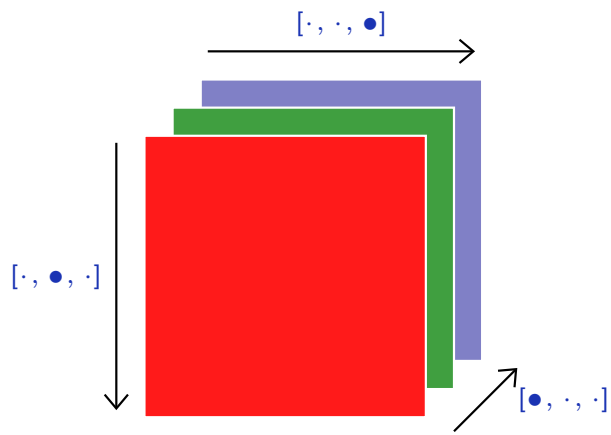
Operations with tensors stored in a certain device's memory are done by that device. We will come back to that later.

```
>>> 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))
```

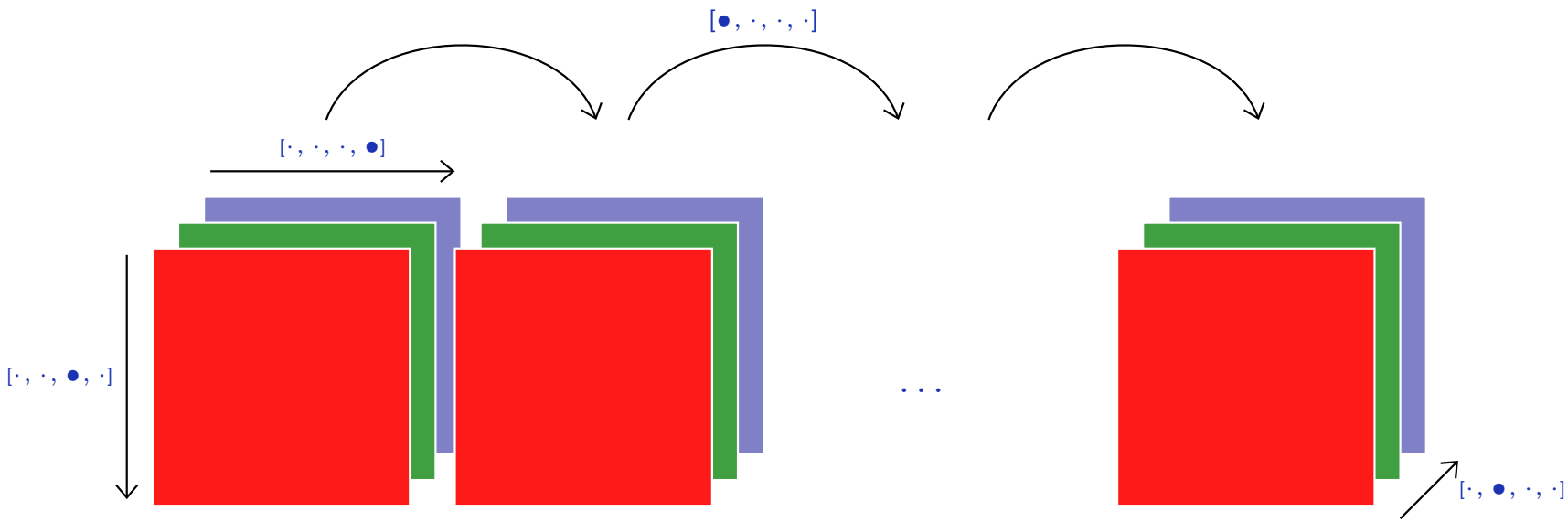
2d tensor (e.g. grayscale image)



3d tensor (e.g. rgb image)



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



Here are a few examples from the immense 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, nb_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

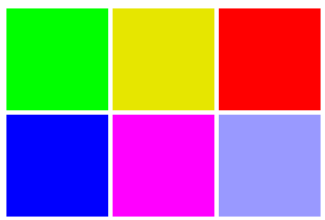
- `torch.abs(input, out=None)`
- `torch.add()`
- `torch.cos(input, out=None)`
- `torch.sigmoid(input, out=None)`

Math reduction

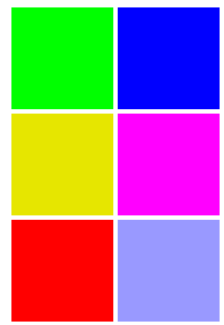
- `torch.dist(input, other, p=2, out=None)`
- `torch.mean()`
- `torch.norm()`
- `torch.std()`
- `torch.sum()`

BLAS and LAPACK Operations

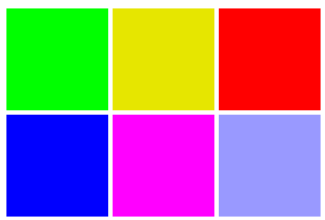
- `torch.linalg(a)`
- `torch.linalg(A, B)`
- `torch.inverse(input, out=None)`
- `torch.mm(mat1, mat2, out=None)`
- `torch.mv(mat, vec, out=None)`



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

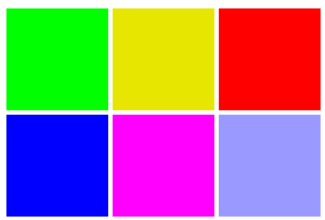


```
x.t()
```

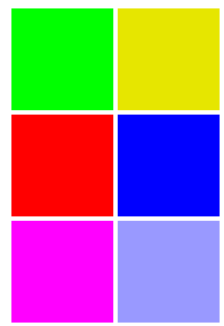


```
x.view(-1)
```

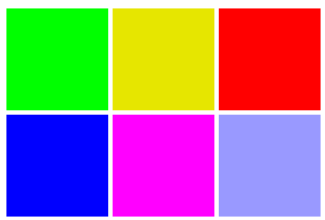
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



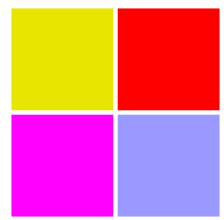
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



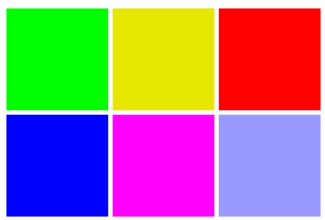
```
x.view(3, -1)
```



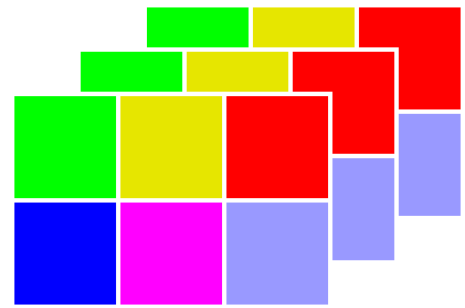
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



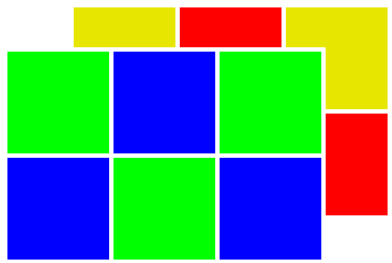
```
x[:, 1:3]
```



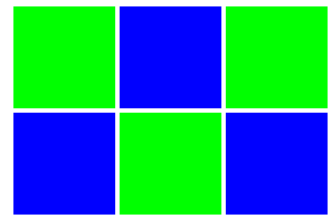
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



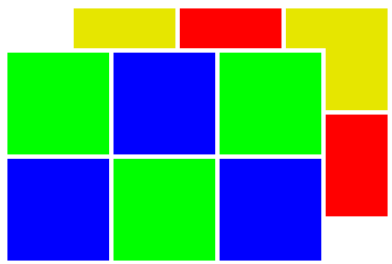
```
x.view(1, 2, 3).expand(3, 2, 3)
```



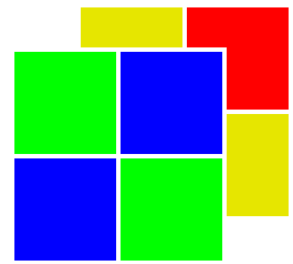
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



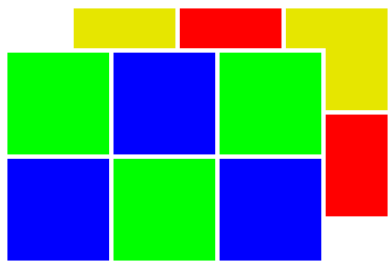
```
x[0:1, :, :]
```



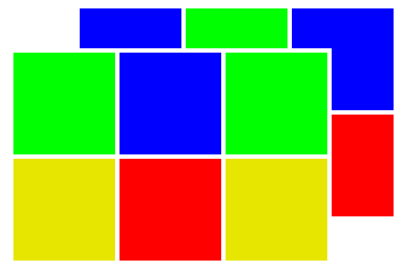
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



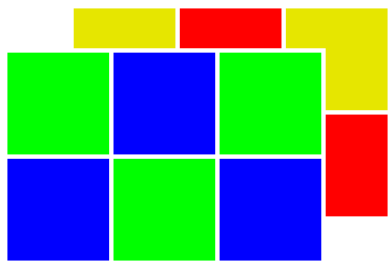
```
x[:, :, 0:2]
```



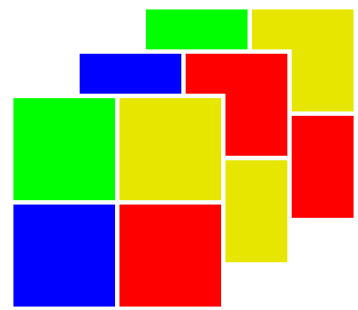
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



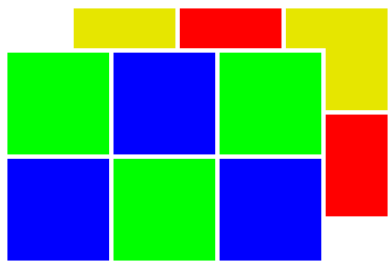
```
x.transpose(0, 1)
```



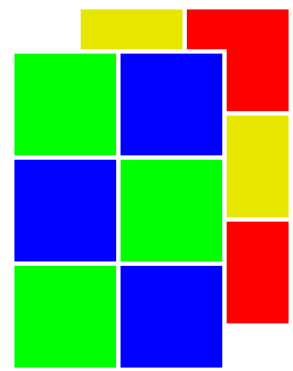
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



```
x.transpose(0, 2)
```



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



```
x.transpose(1, 2)
```

For efficiency reasons, different tensors can share the same data and **modifying one will modify the others**. By default do not make the assumption that two tensors refer to different data in memory.

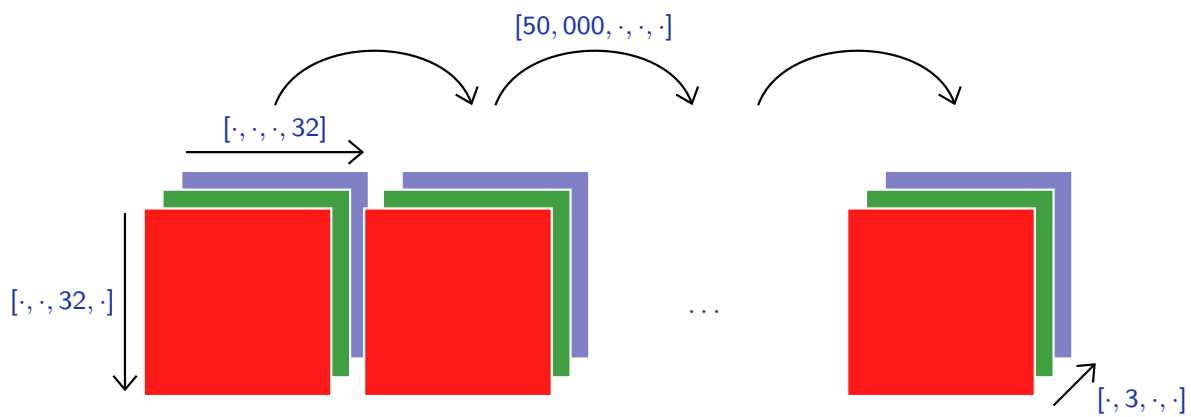
```
>>> a = torch.full((2, 3), 1)
>>> a
tensor([[1, 1, 1],
        [1, 1, 1]])
>>> b = a.view(-1)
>>> b
tensor([1, 1, 1, 1, 1, 1])
>>> a[1, 1] = 2
>>> a
tensor([[1, 1, 1],
        [1, 2, 1]])
>>> b
tensor([1, 1, 1, 1, 2, 1])
>>> b[0] = 9
>>> a
tensor([[9, 1, 1],
        [1, 2, 1]])
>>> b
tensor([9, 1, 1, 1, 2, 1])
```

PyTorch offers simple interfaces to standard image databases.

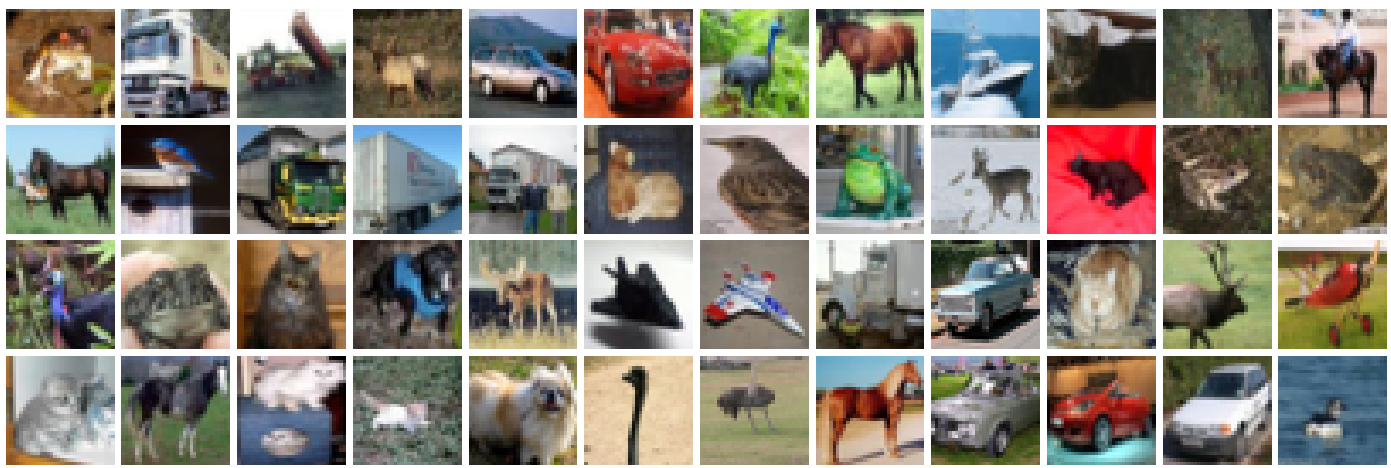
```
import torch, torchvision
cifar = torchvision.datasets.CIFAR10('./cifar10/', train = True, download = True)
x = torch.from_numpy(cifar.data).permute(0, 3, 1, 2).float() / 255
print(x.dtype, x.size(), x.min().item(), x.max().item())
```

prints

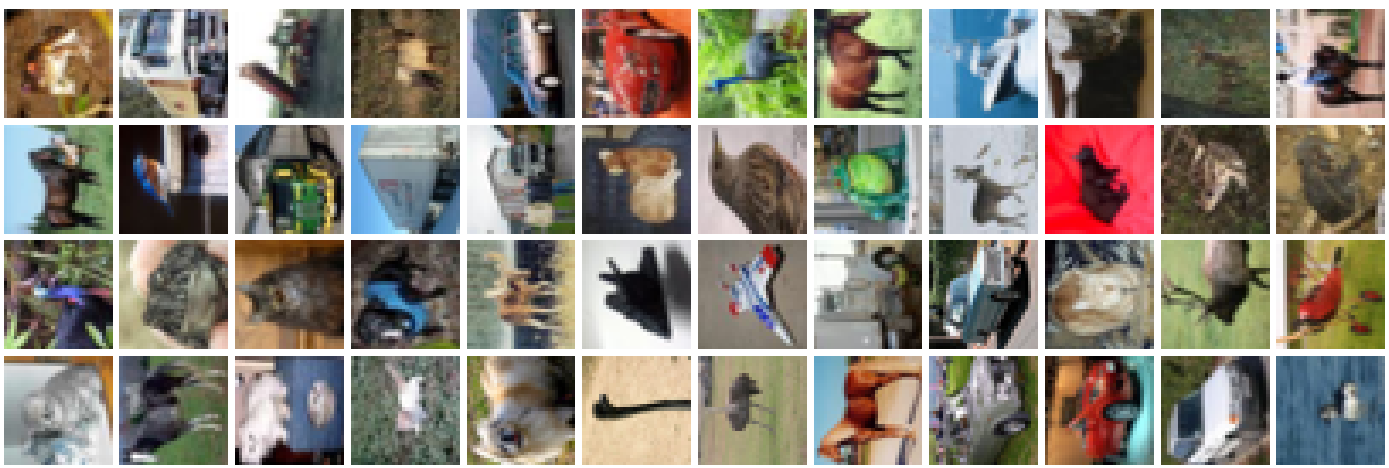
```
Files already downloaded and verified
torch.float32 torch.Size([50000, 3, 32, 32]) 0.0 1.0
```



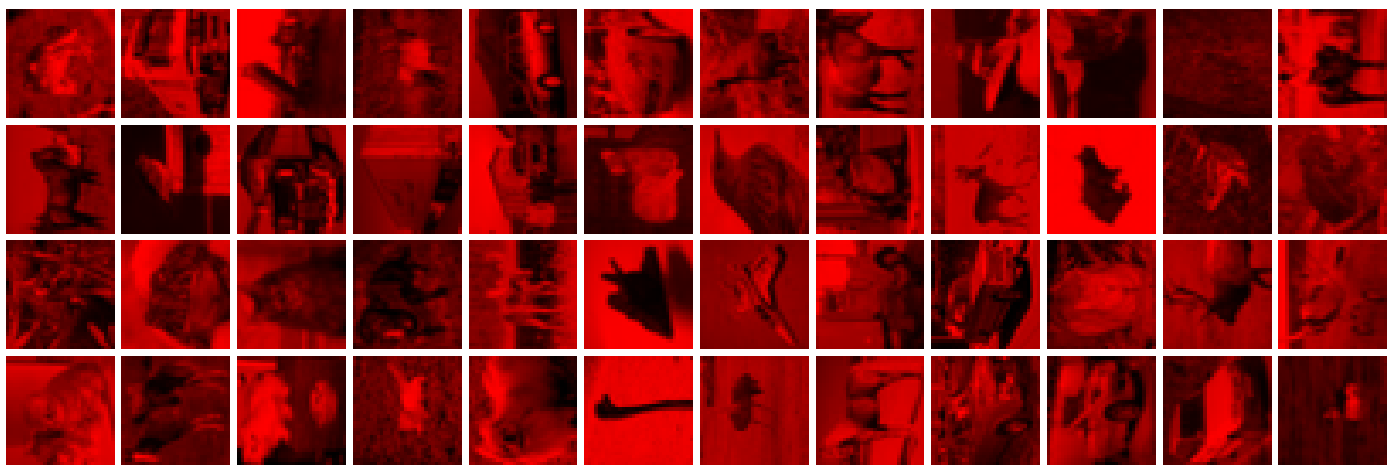
```
# Narrows to the first images, converts to float
x = x[:48]

# Saves these samples as a single image
torchvision.utils.save_image(x, 'cifar-4x12.png',
                             nrow = 12, pad_value = 1.0)
```



```
# Switches the row and column indexes
x.transpose_(2, 3)
torchvision.utils.save_image(x, 'cifar-4x12-rotated.png',
                             nrow = 12, pad_value = 1.0)
```



```
# Kills the green and blue channels
x[:, 1:3].fill_(0)
torchvision.utils.save_image(x, 'cifar-4x12-rotated-and-red.png',
                             nrow = 12, pad_value = 1.0)
```



Broadcasting and Einstein summations

Broadcasting automagically expands dimensions by replicating coefficients, when it is necessary to perform operations that are "intuitively reasonable".

For instance:

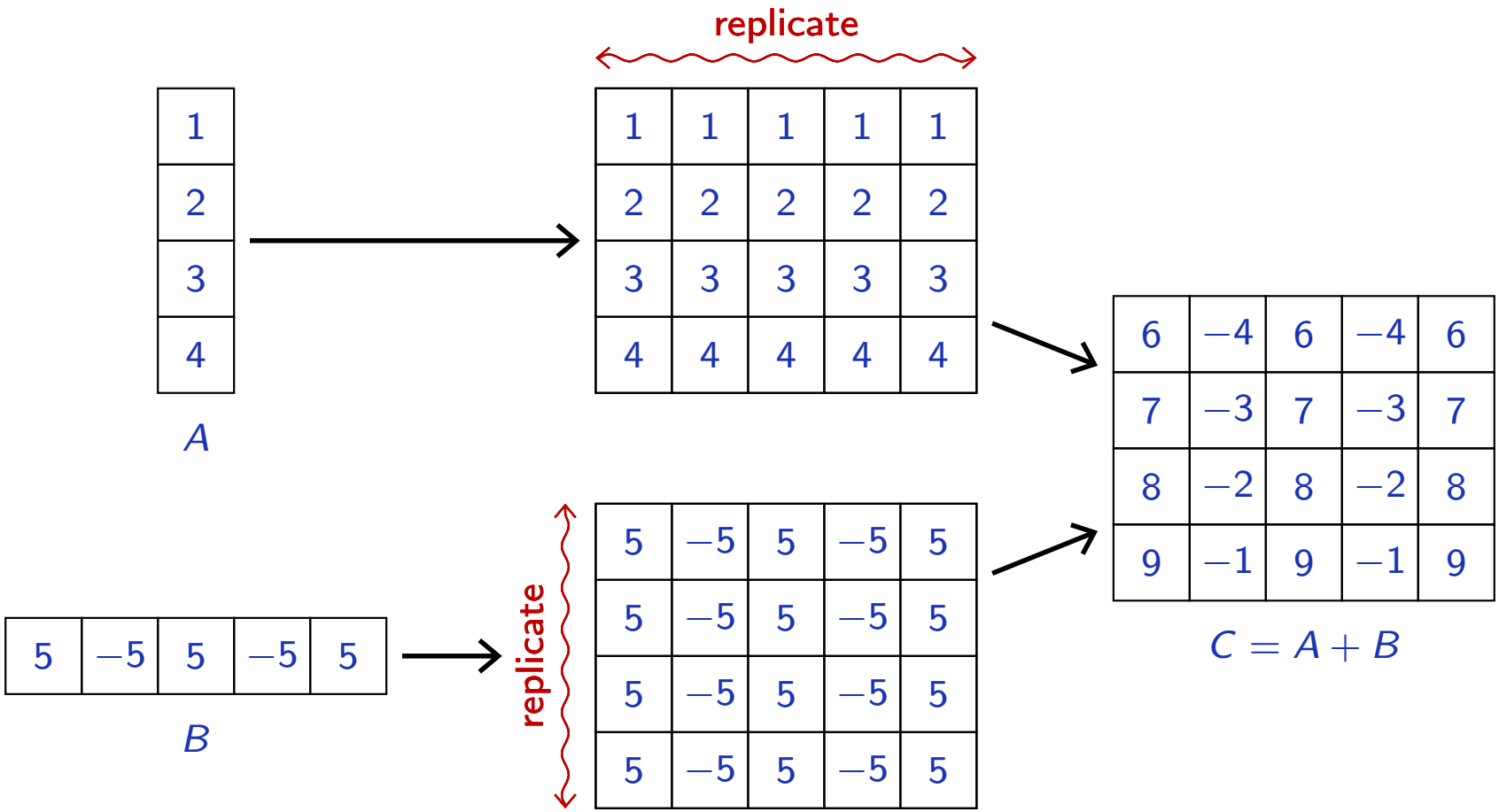
```
>>> x = torch.empty(100, 4).normal_(2)
>>> x.mean(0)
tensor([2.0476, 2.0133, 1.9109, 1.8588])
>>> x -= x.mean(0) # This should not work, but it does!
>>> x.mean(0)
tensor([-4.0531e-08, -4.4703e-07, -1.3471e-07,  3.5763e-09])
```

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

A powerful generic tool for complex tensorial operations is the **Einstein summation convention**. It provides a concise way of describing dimension re-ordering and summing of component-wise products along some of them.

`torch.einsum` takes as argument a string describing the operation, the tensors to operate on, and returns a tensor.

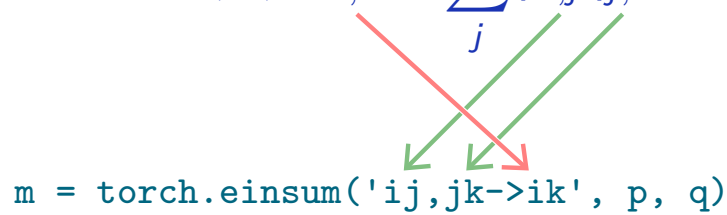
The operation string is a comma-separated list of indexing, followed by the indexing for the result.

Summations are executed on all indexes not appearing in the result indexing.

For instance, we can formulate that way the standard matrix product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{B\times C} \to \mathbb{R}^{A\times C}$$

$$\forall i,k,\ m_{i,k} = \sum_j p_{i,j} q_{j,k}$$



The summation is done along `j` since it does not appear after the `->`.

```
>>> p = torch.rand(2, 5)
>>> q = torch.rand(5, 4)
>>> torch.einsum('ij,jk->ik', p, q)
tensor([[2.0833, 1.1046, 1.5220, 0.4405],
        [2.1338, 1.2601, 1.4226, 0.8641]])
>>> p@q
tensor([[2.0833, 1.1046, 1.5220, 0.4405],
        [2.1338, 1.2601, 1.4226, 0.8641]])
```

Matrix-vector product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{B} \to \mathbb{R}^{A}$$

$$\forall i, k,\ \ w_i = \sum_j m_{i,j} v_j$$

```
w = torch.einsum('ij,j->i', m, v)
```

Hadamard (component-wise) product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{A\times B} \to \mathbb{R}^{A\times B}$$

$$\forall i, j,\ \ m_{i,j} = p_{i,j} q_{i,j}$$

```
m = torch.einsum('ij,ij->ij', p, q)
```

Extracting the diagonal:

$$\mathbb{R}^{D\times D} \to \mathbb{R}^{D}$$

$$\forall i, k,\ \ v_i = m_{i,i}$$

```
v = torch.einsum('ii->i', m)
```

Batch matrix product:

$$\mathbb{R}^{N\times A\times B} \times \mathbb{R}^{N\times B\times C} \to \mathbb{R}^{N\times A\times C}$$

$$\forall n, i, k,\ \ m_{n,i,k} = \sum_j p_{n,i,j} q_{n,j,k}$$

```
m = torch.einsum('nij,njk->nik', p, q)
```

Batch trace:

$$\mathbb{R}^{N\times D\times D} \to \mathbb{R}^{N}$$

$$\forall n,\ \ t_n = \sum_i m_{n,i,i}$$

```
t = torch.einsum('nii->n', m)
```

Tri-linear product along a channel:

$$\mathbb{R}^{N\times C\times T} \times \mathbb{R}^{N\times C\times T} \times \mathbb{R}^{N\times C\times T} \to \mathbb{R}^{N\times T}$$

$$\forall n, t,\ \ m_{n,t} = \sum_c p_{n,c,t} q_{n,c,t} r_{n,c,t}$$

```
m = torch.einsum('nct,nct,nct->nt', p, q, r)
```

Deep learning

1.6. Tensor internals

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

A tensor is a view of a [part of a] **storage**, which is a low-level 1d vector.

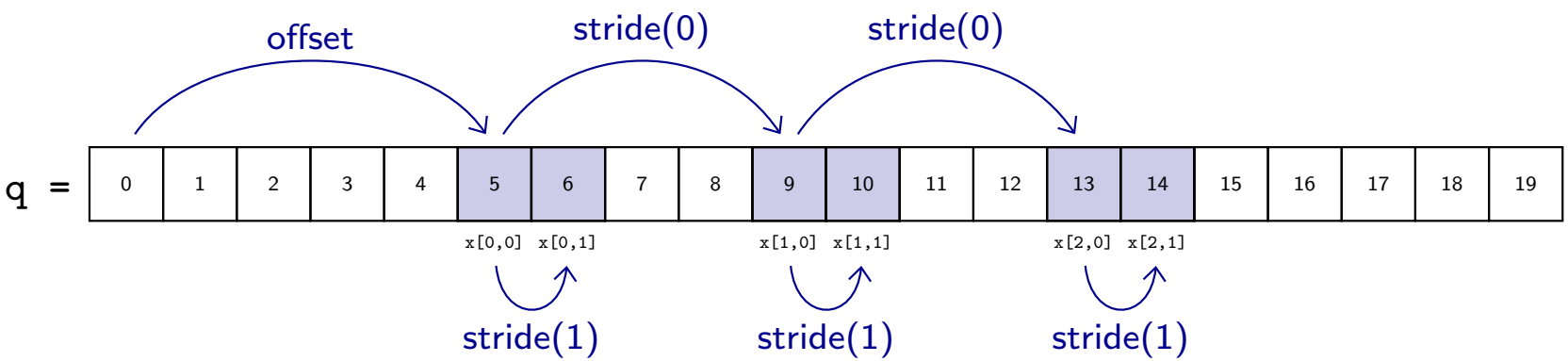
```
>>> x = torch.zeros(2, 4)
>>> x.storage()
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> q = x.storage()
>>> q[4] = 1.0
>>> x
tensor([[ 0.,  0.,  0.,  0.],
        [ 1.,  0.,  0.,  0.]])
```

The first coefficient of a tensor is the one at `storage_offset()` in `storage()`.

Incrementing index `k` by `1` move by `stride(k)` elements in the storage.

E.g. in a 2d tensor, incrementing the row index moves by `stride(0)` in the storage, and incrementing the column index moves by `stride(1)`.

```
>>> q = torch.arange(0., 20.).storage()
>>> x = torch.empty(0).set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))
>>> x
tensor([[ 5.,  6.],
        [ 9., 10.],
        [13., 14.]])
```



We can explicitly create different "views" of the same storage

```
>>> n = torch.linspace(1, 4, 4)
>>> n
tensor([ 1.,  2.,  3.,  4.])
>>> torch.tensor(0.).set_(n.storage(), 1, (3, 3), (0, 1))
tensor([[ 2.,  3.,  4.],
        [ 2.,  3.,  4.],
        [ 2.,  3.,  4.]])
>>> torch.tensor(0.).set_(n.storage(), 1, (2, 4), (1, 0))
tensor([[ 2.,  2.,  2.,  2.],
        [ 3.,  3.,  3.,  3.]])
```

This is in particular how transpositions and broadcasting are implemented.

```
>>> x = torch.empty(100, 100)
>>> x.stride()
(100, 1)
>>> y = x.t()
>>> y.stride()
(1, 100)
```

This organization explains the following (maybe surprising) error

```
>>> x = torch.empty(100, 100)
>>> x.t().view(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: invalid argument 2: view size is not compatible with
input tensor's size and stride (at least one dimension spans across
two contiguous subspaces). Call .contiguous() before .view()
```

`x.t()` shares `x`'s storage and cannot be "flattened" to 1d.

This can be fixed with `contiguous()`, which returns a contiguous version of the tensor, **making a copy if needed.**

The function `reshape()` combines `view()` and `contiguous()`.

The end