EE-559 – Deep learning

1.1. From neural networks to deep learning

François Fleuret

https://fleuret.org/ee559/

Tue Feb 19 14:17:30 UTC 2019

idiap RESEARCH INSTITUTE

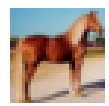
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

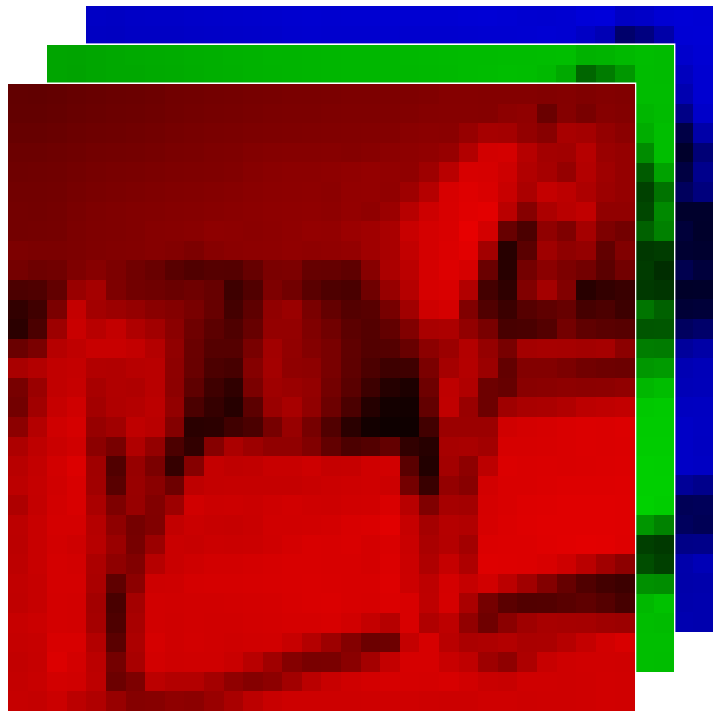
Many applications require the automatic extraction of "refined" information from raw signal (*e.g.* image recognition, automatic speech processing, natural language processing, robotic control, geometry reconstruction).

(ImageNet)

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

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.

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.

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.

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

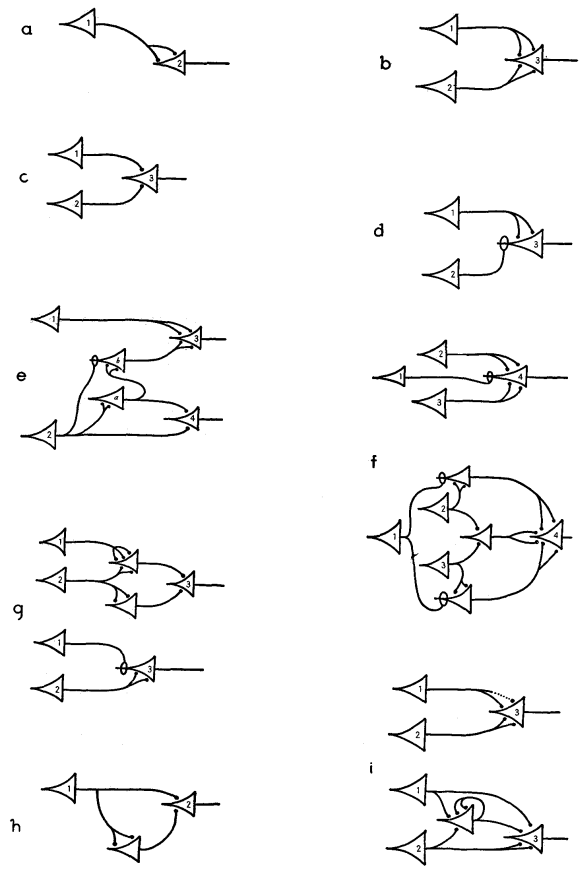
Classical ML methods combine a "learnable" model from statistics (*e.g.* "linear regression") with prior knowledge in pre-processing.

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).

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.

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.

1949 – Donald Hebb proposes the Hebbian Learning principle.

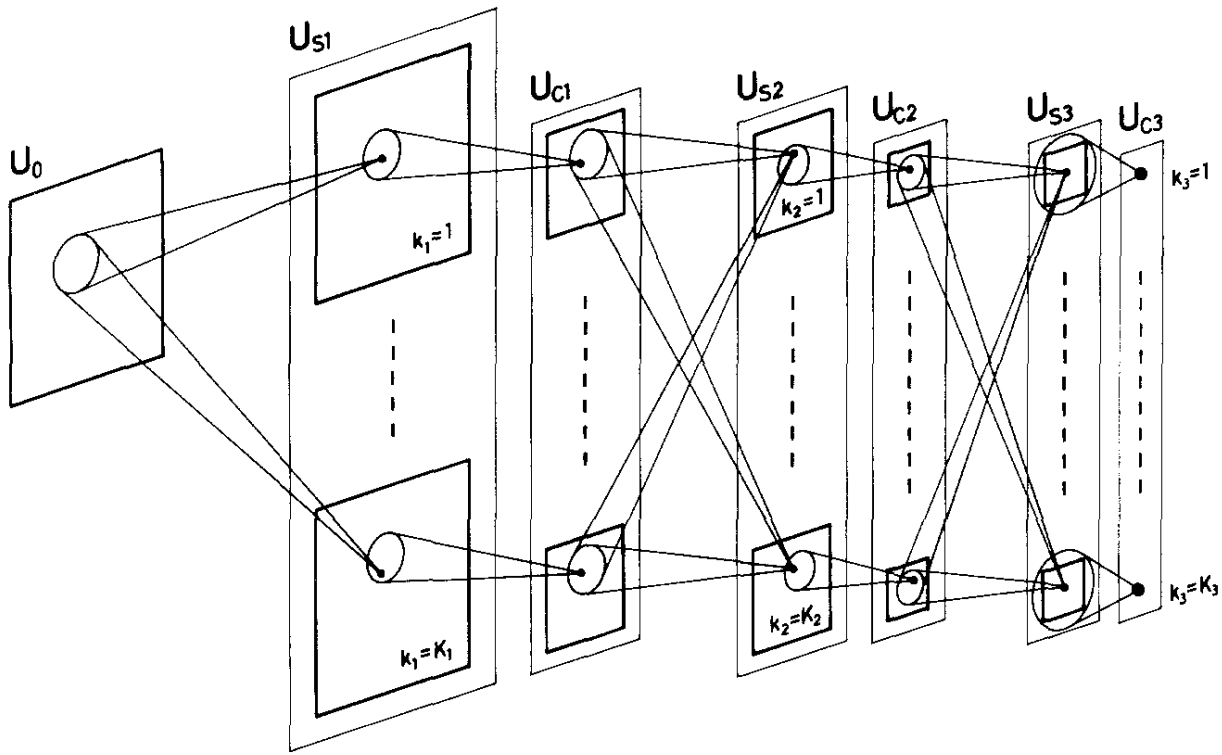
1951 – Marvin Minsky creates the first ANN (Hebbian learning, 40 neurons).

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.

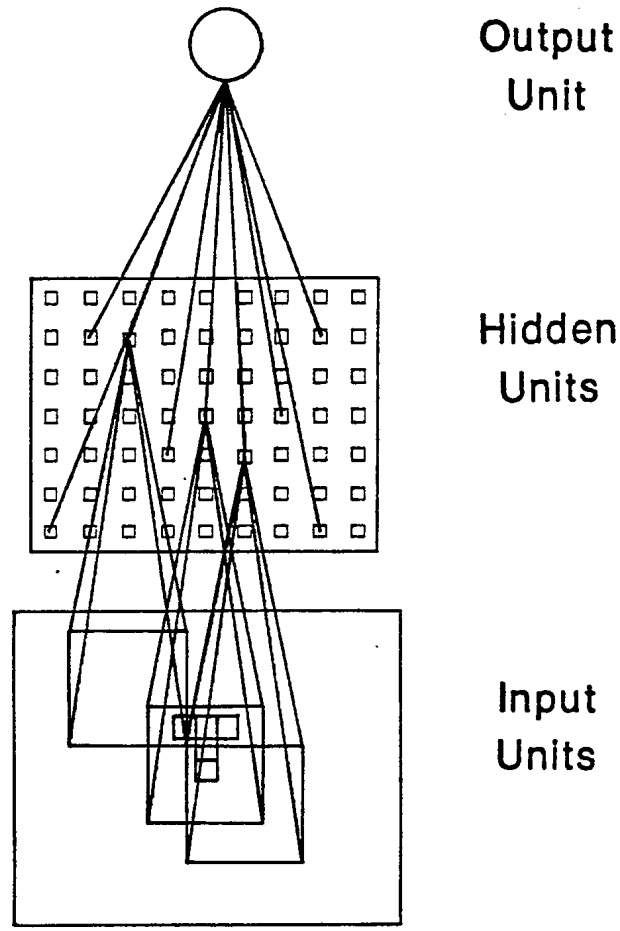
Neocognitron



Follows Hubel and Wiesel's results.

(Fukushima, 1980)

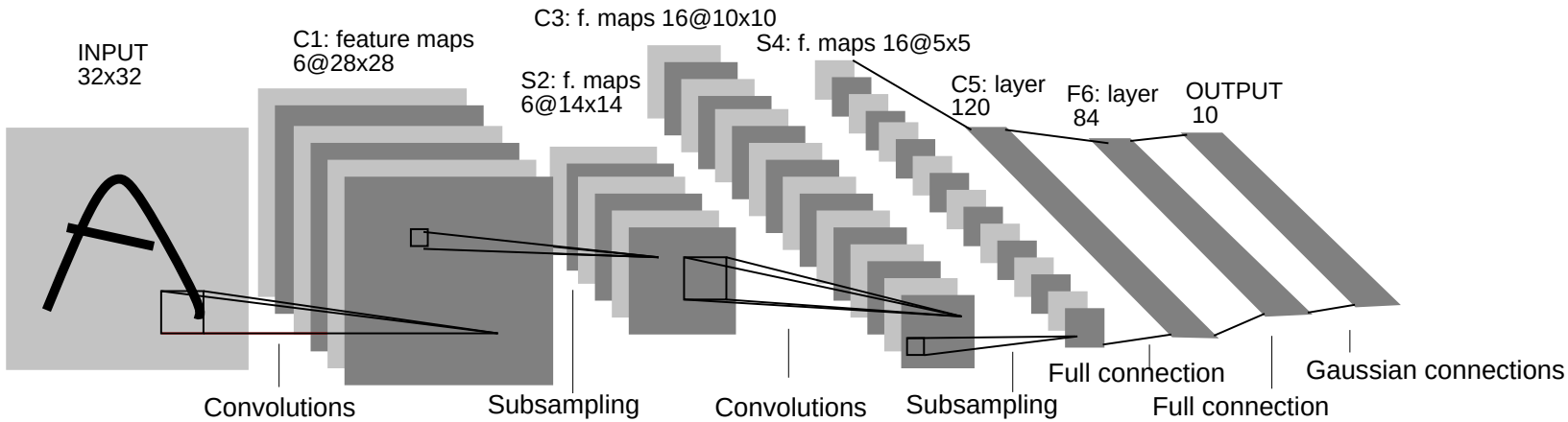
Network for the T-C problem



Trained with back-prop.

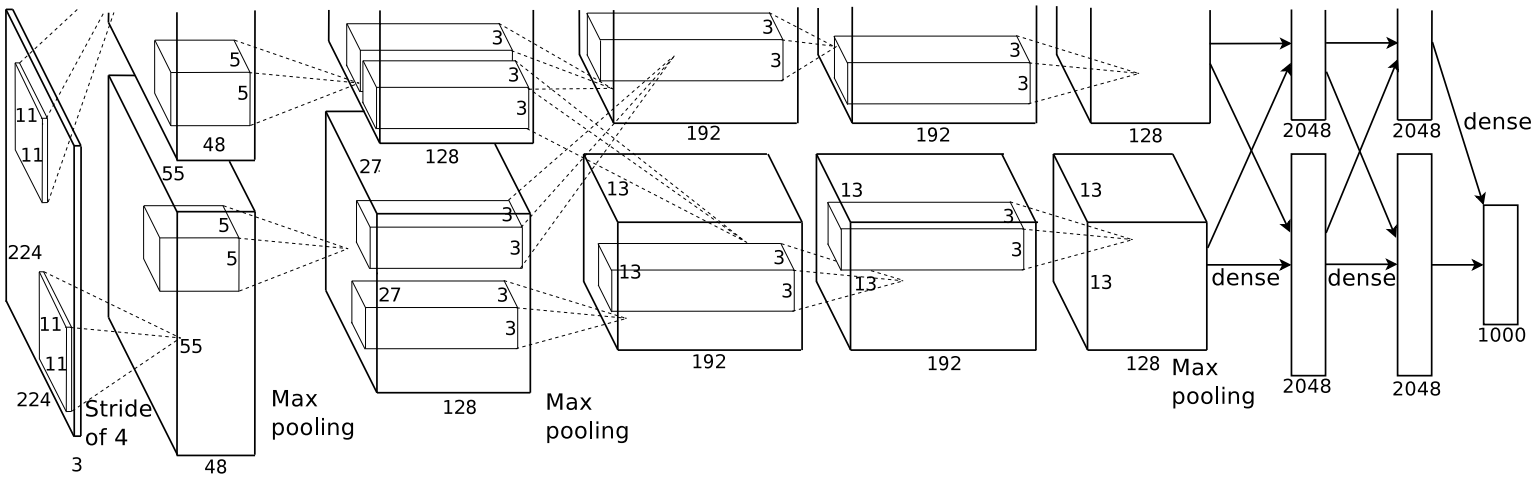
(Rumelhart et al., 1988)

LeNet-5



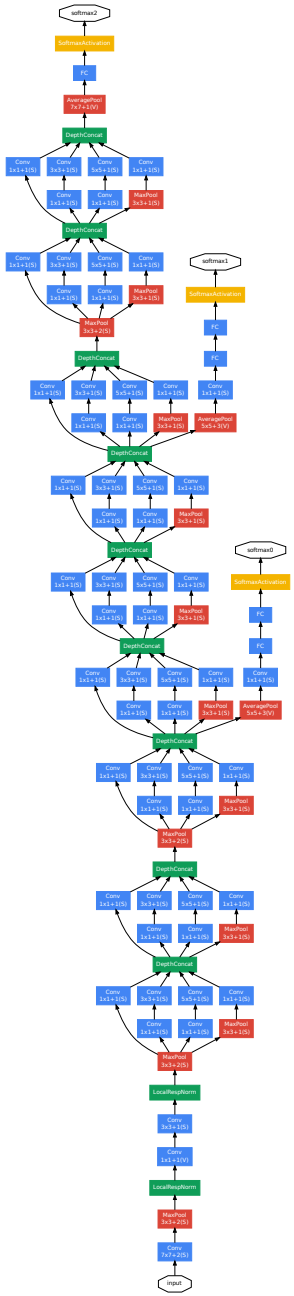
(leCun et al., 1998)

AlexNet



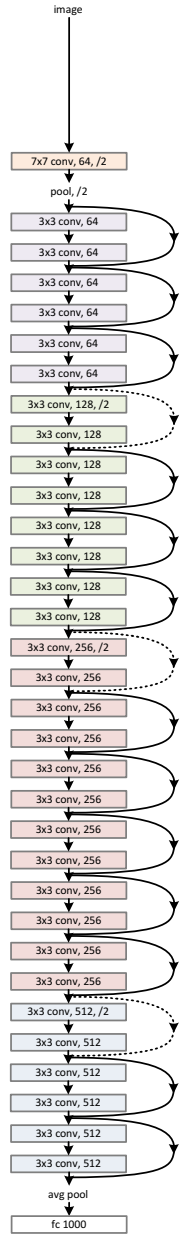
(Krizhevsky et al., 2012)

GoogLeNet



(Szegedy et al., 2015)

Resnet



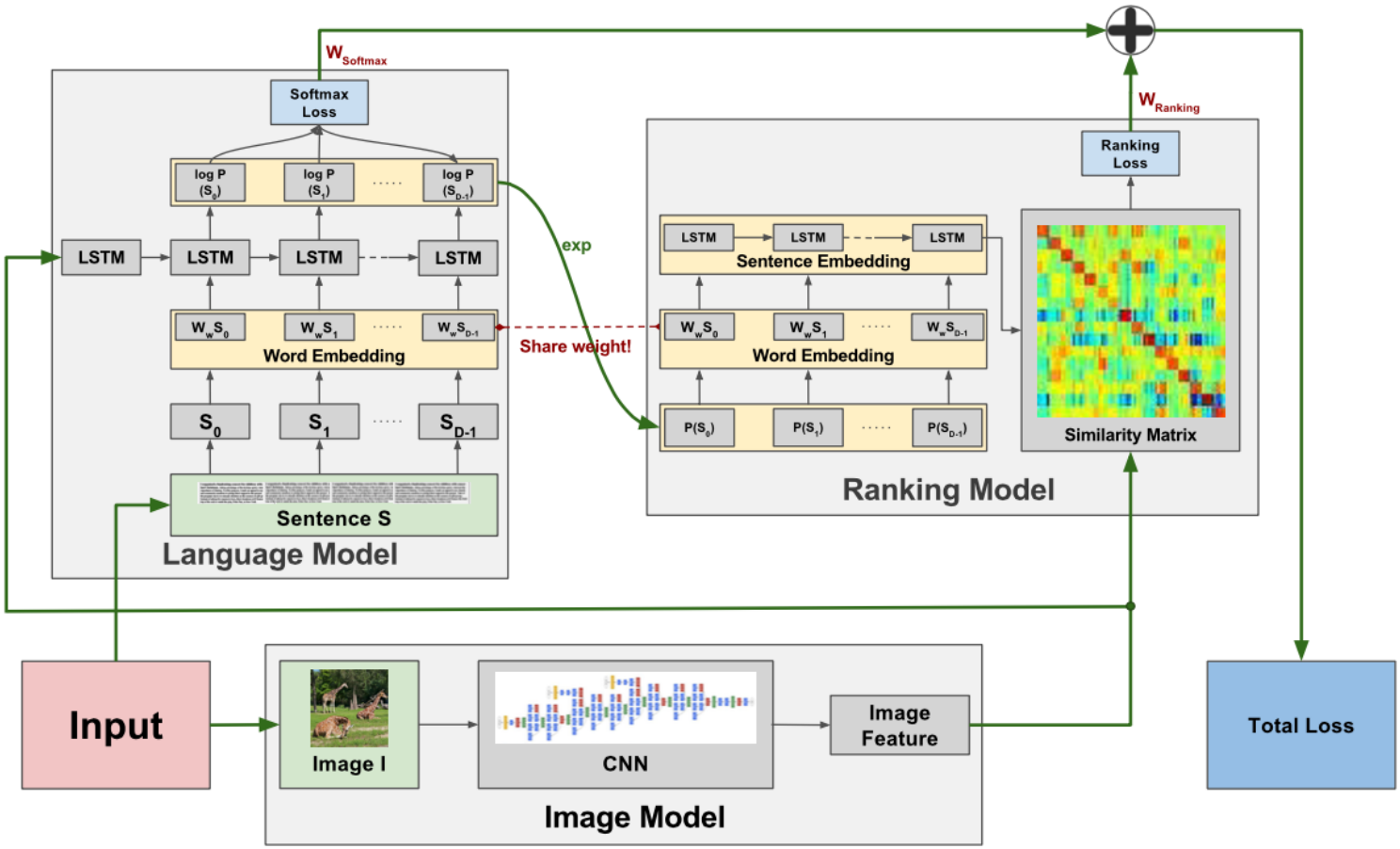
(He et al., 2015)

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.

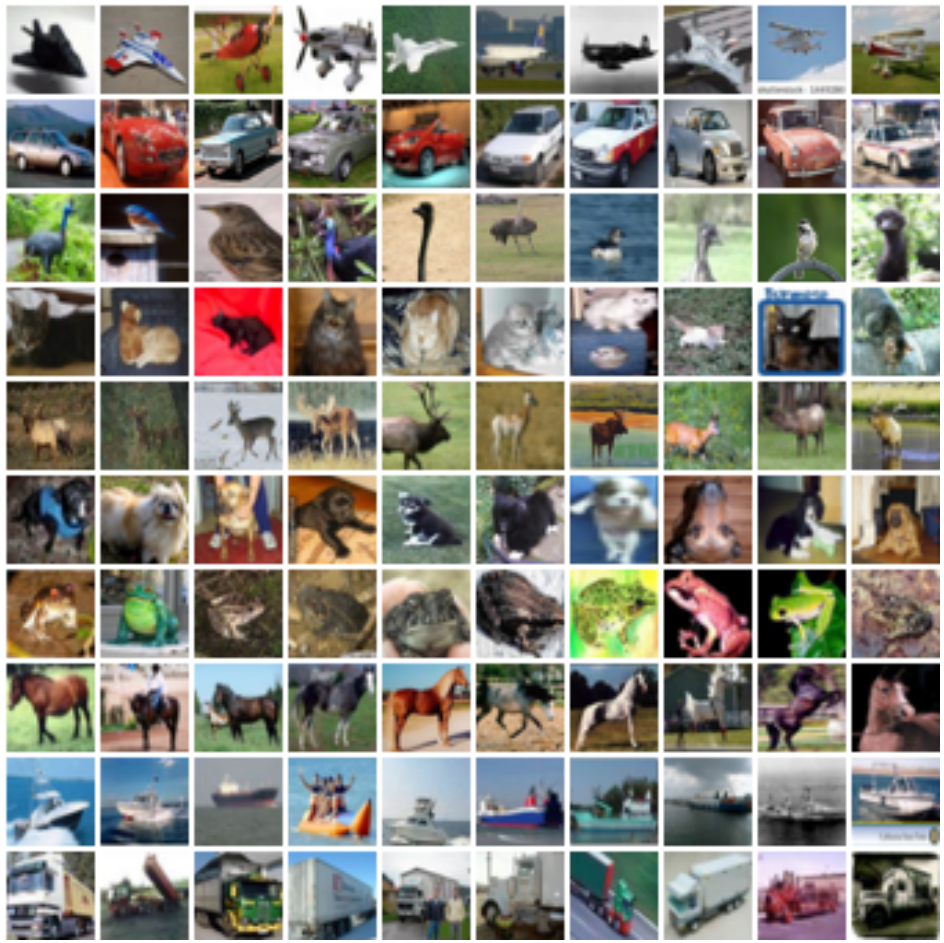
This does not differ much from networks from the 90s

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



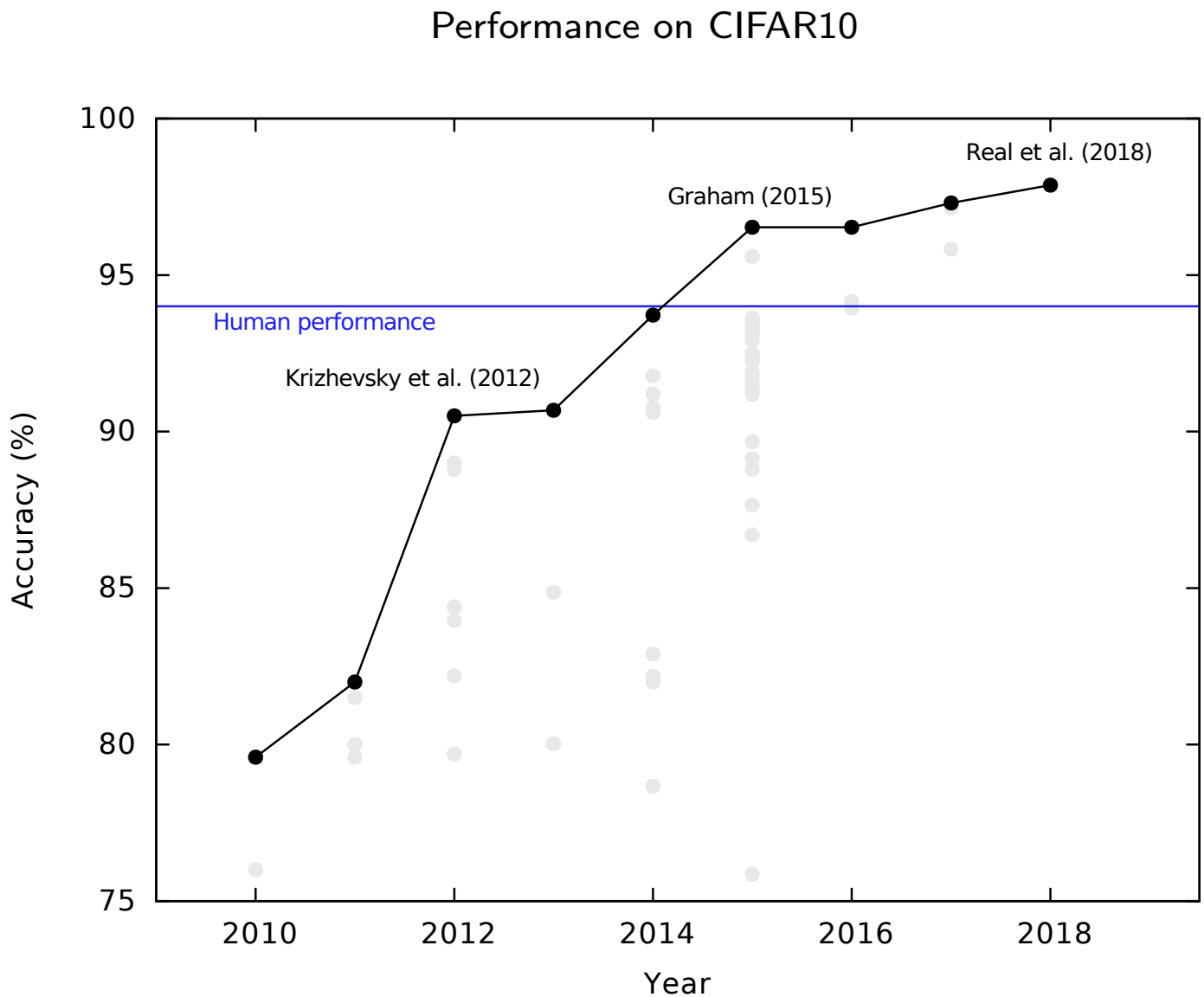
(Yeung et al., 2015)

CIFAR10



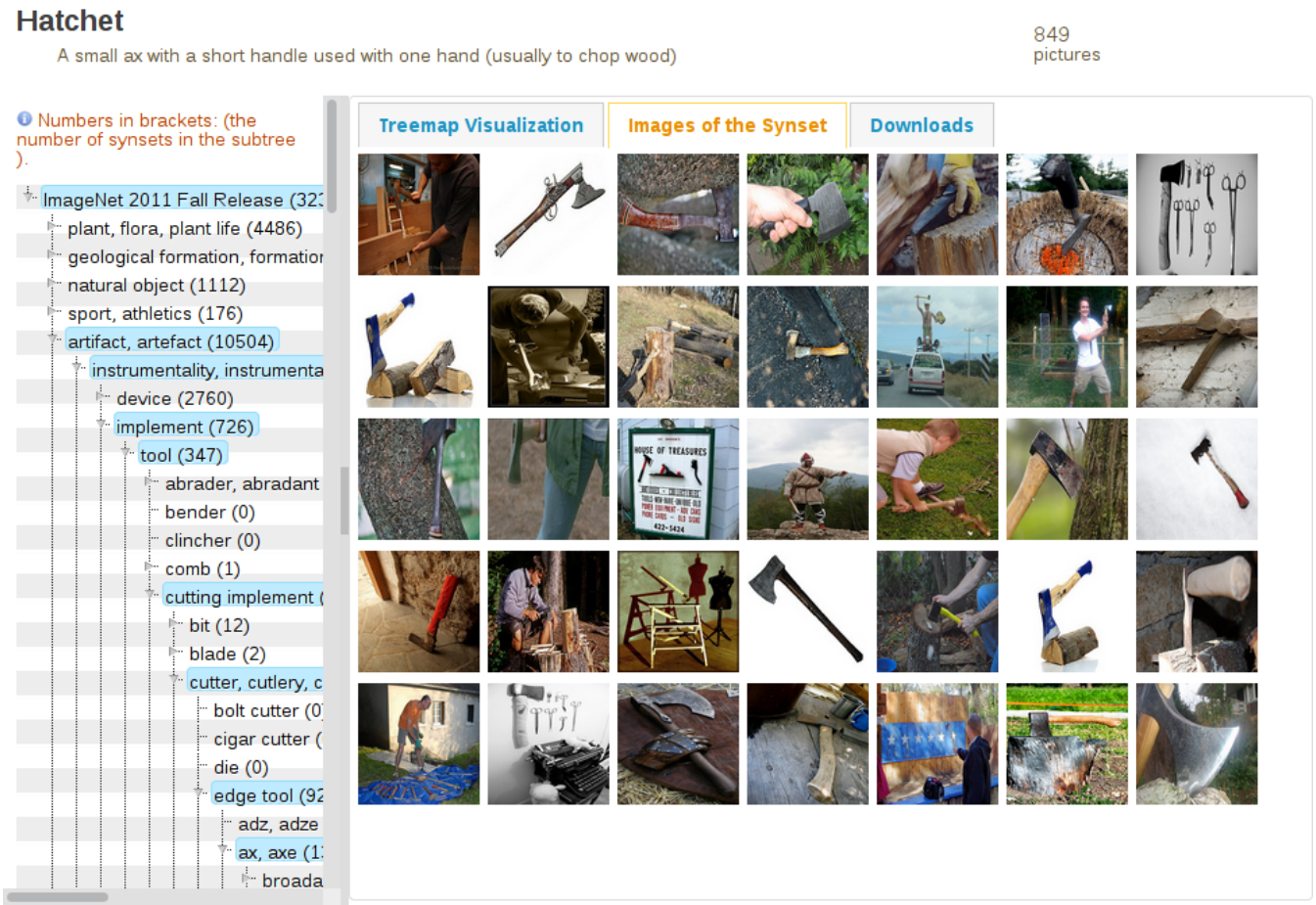
32×32 color images, 50k train samples, 10k test samples.

(Krizhevsky, 2009, chap. 3)


Performance on CIFAR10
100
95
90
85
80
75
Accuracy (%)
2010
2012
2014
2016
2018
Year
Human performance
Krizhevsky et al. (2012)
Graham (2015)
Real et al. (2018)

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)

The end

References

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.

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.

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.

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.

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.

EE-559 – Deep learning

1.2. Current applications and success

François Fleuret

https://fleuret.org/ee559/

Tue Feb 19 14:18:27 UTC 2019

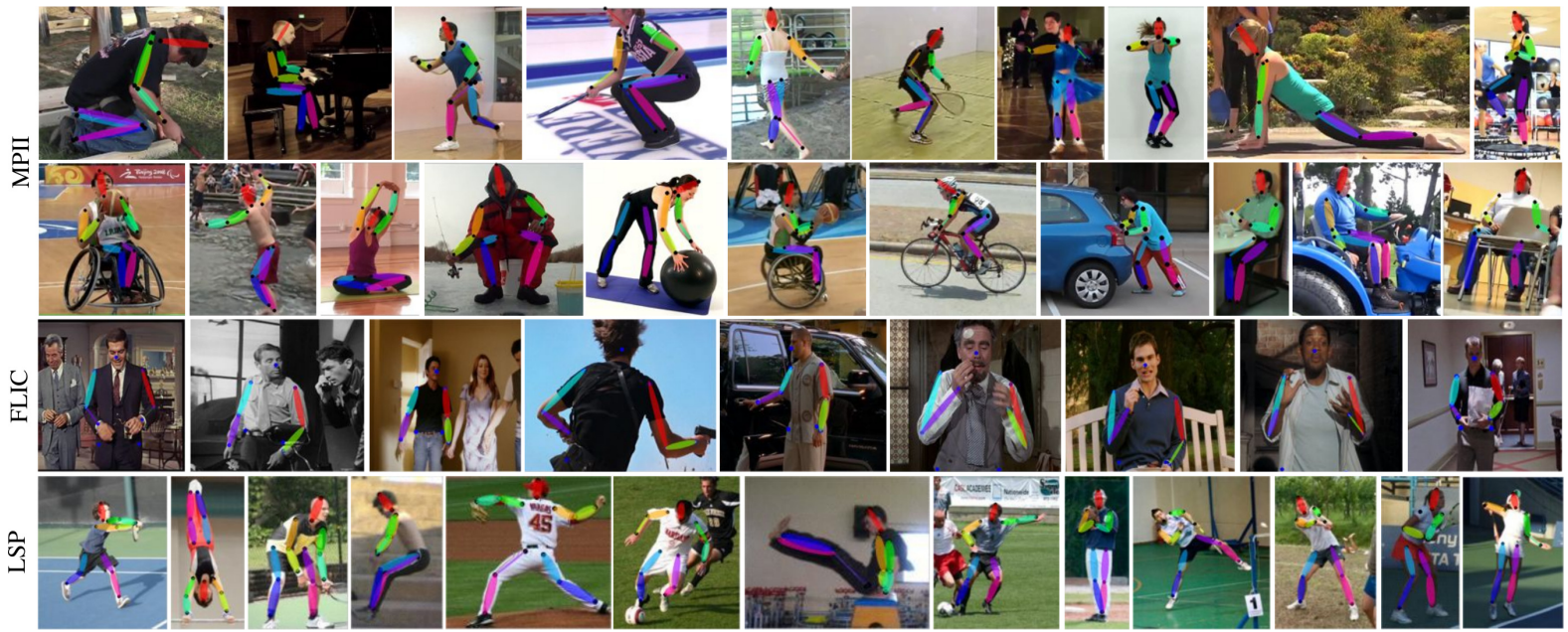
idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Object detection and segmentation

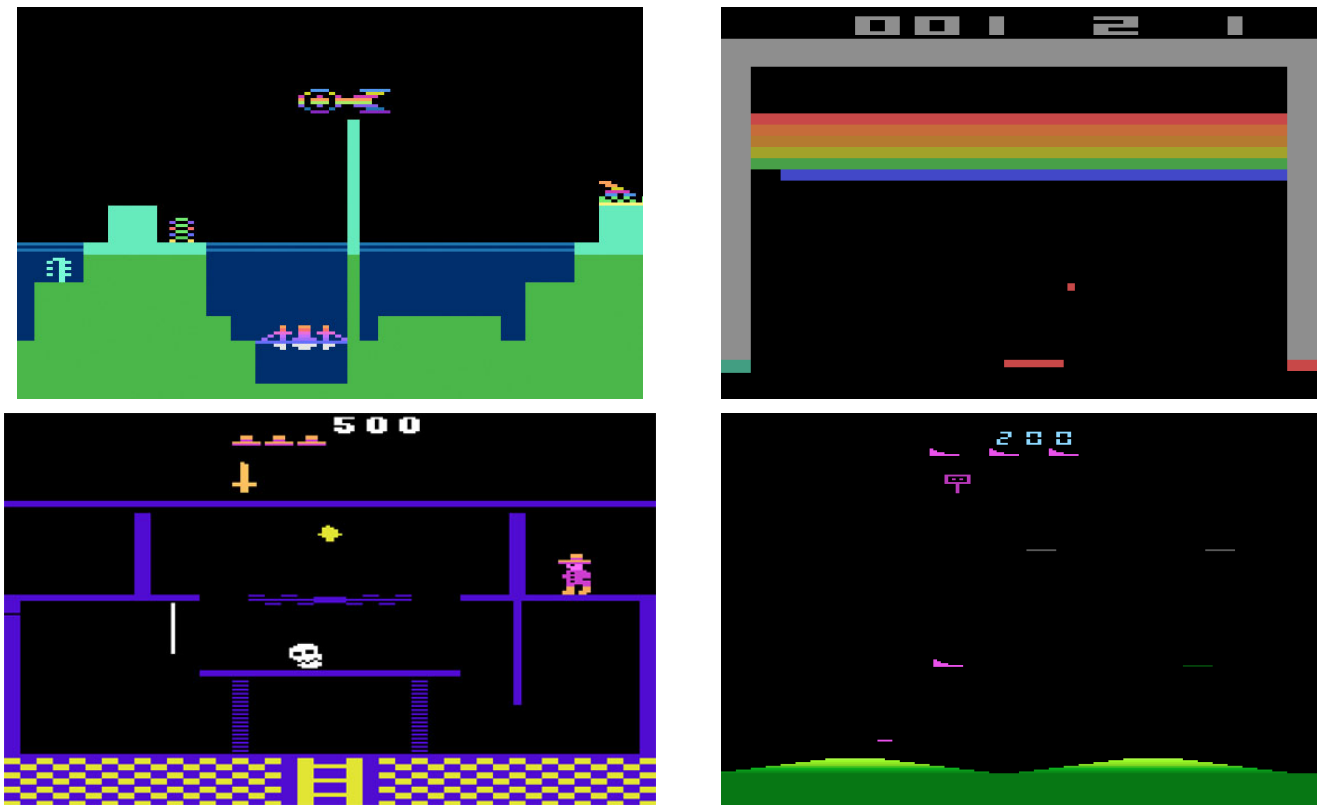
(Pinheiro et al., 2016)

Human pose estimation



(Wei et al., 2016)

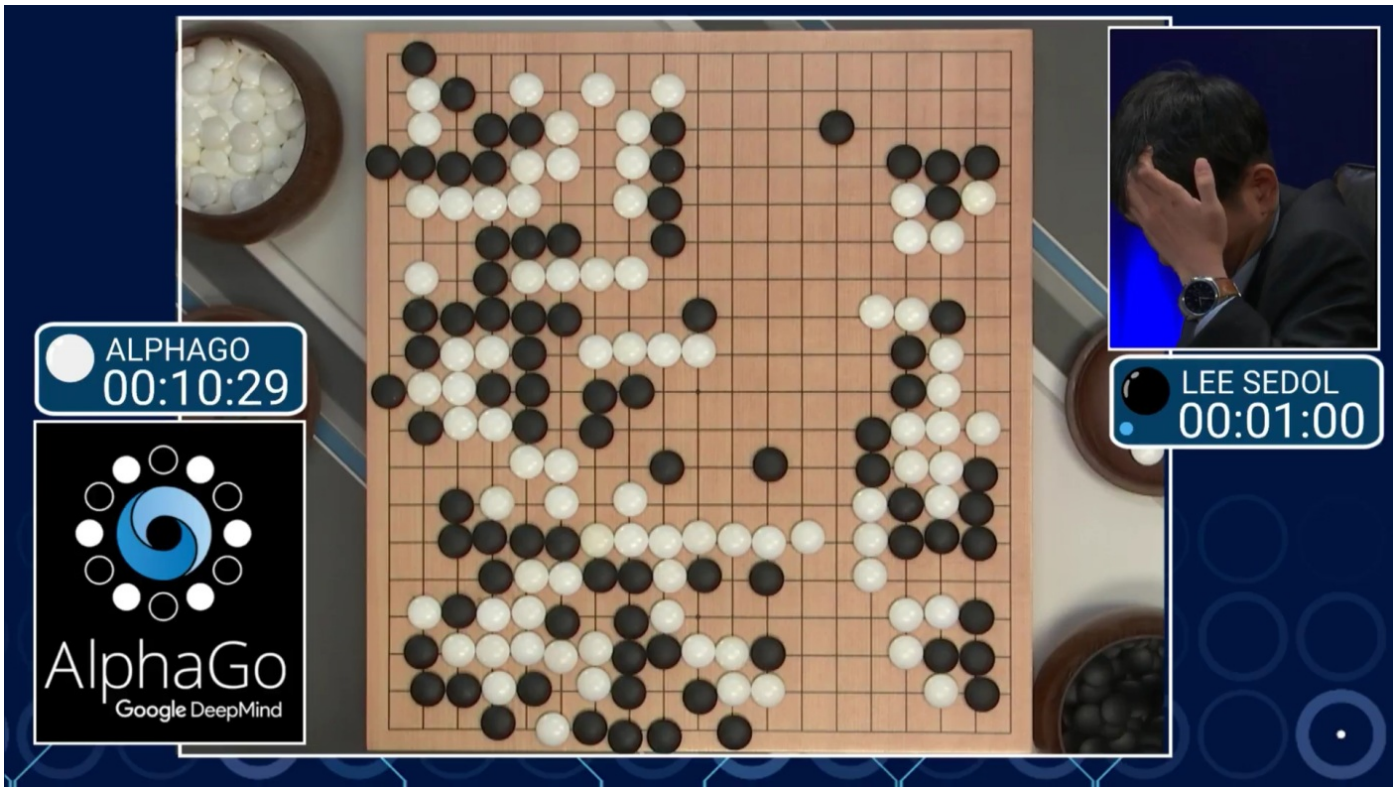
Reinforcement learning



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.

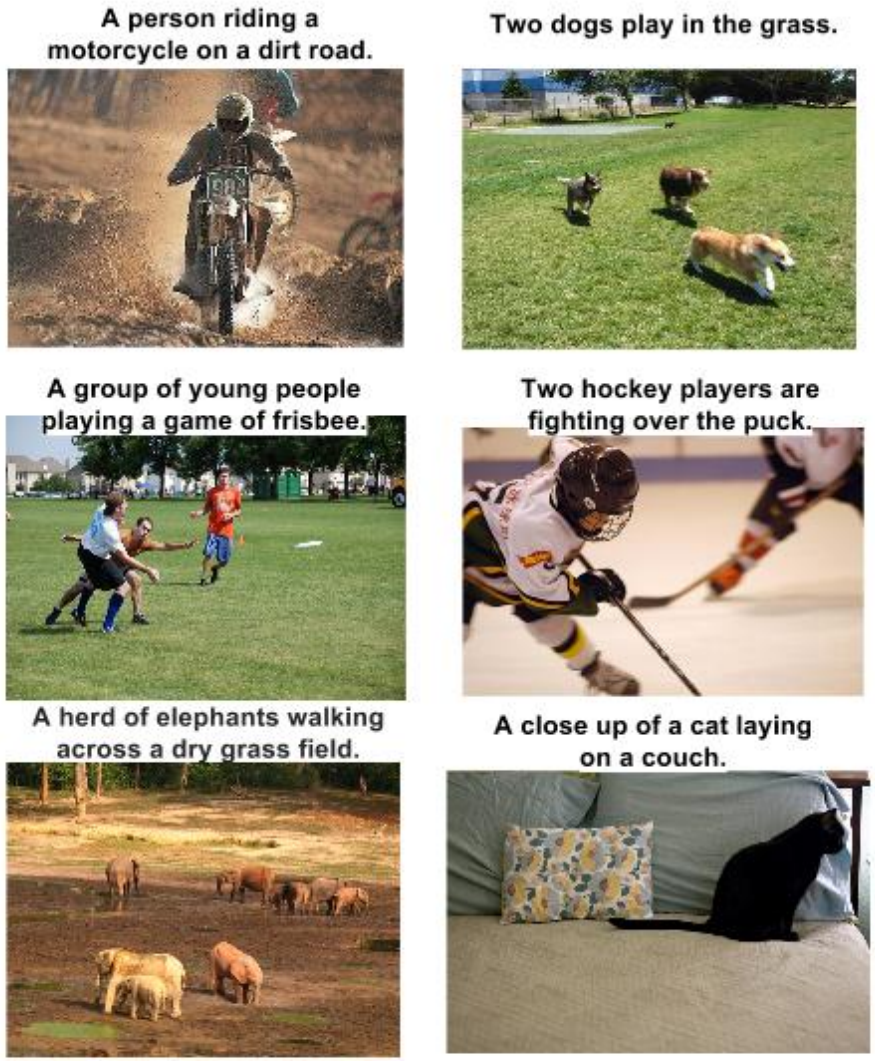
➙ "La raison pour laquelle Boeing fait cela est de créer plus de sièges pour rendre son avion plus compétitif avec nos produits", a déclaré Kevin Keniston, chef du confort des passagers chez Airbus.

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



(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:

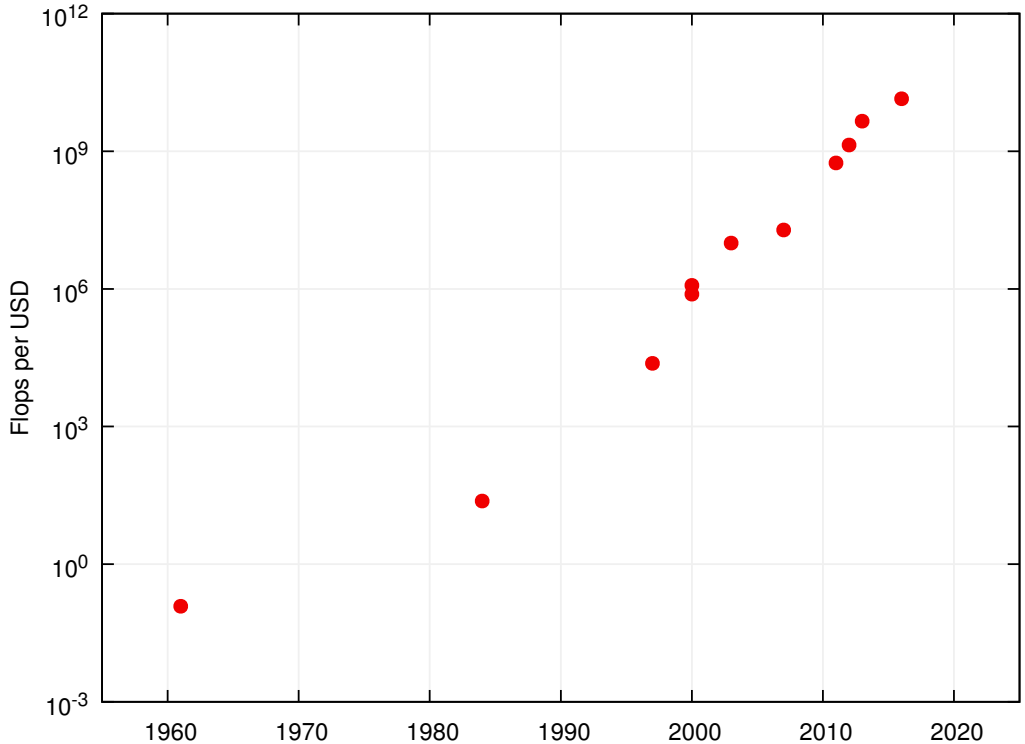
- Five decades of research in machine learning,
- CPUs/GPUs/storage developed for other purposes,
- lots of data from "the internet",
- tools and culture of collaborative and reproducible science,
- resources and efforts from large corporations.

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.

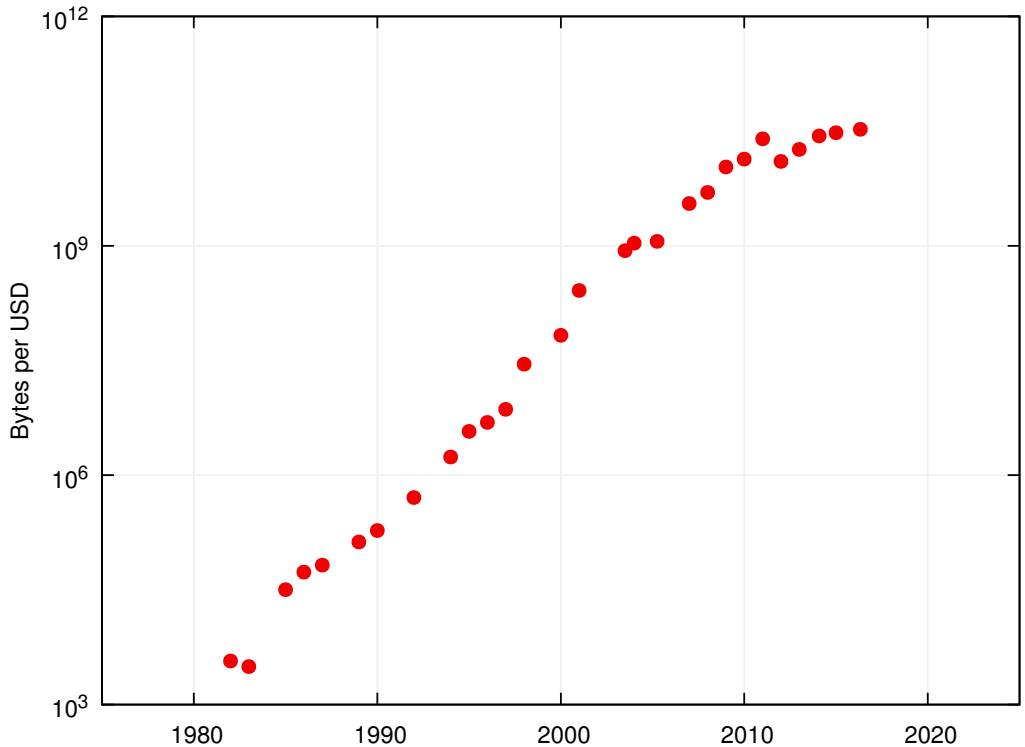
From a practical perspective, deep learning

- 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.



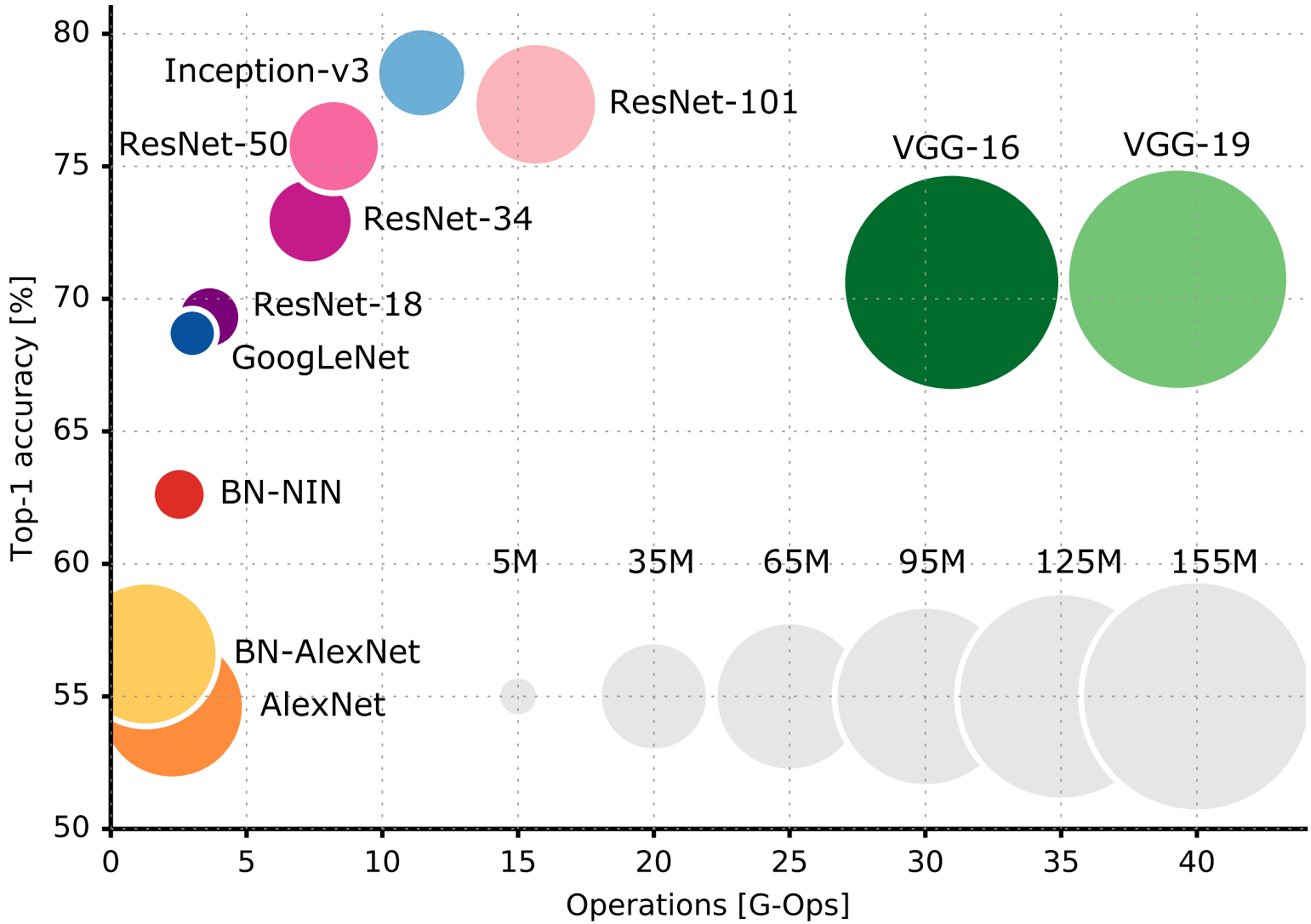
(Wikipedia "FLOPS")

	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.

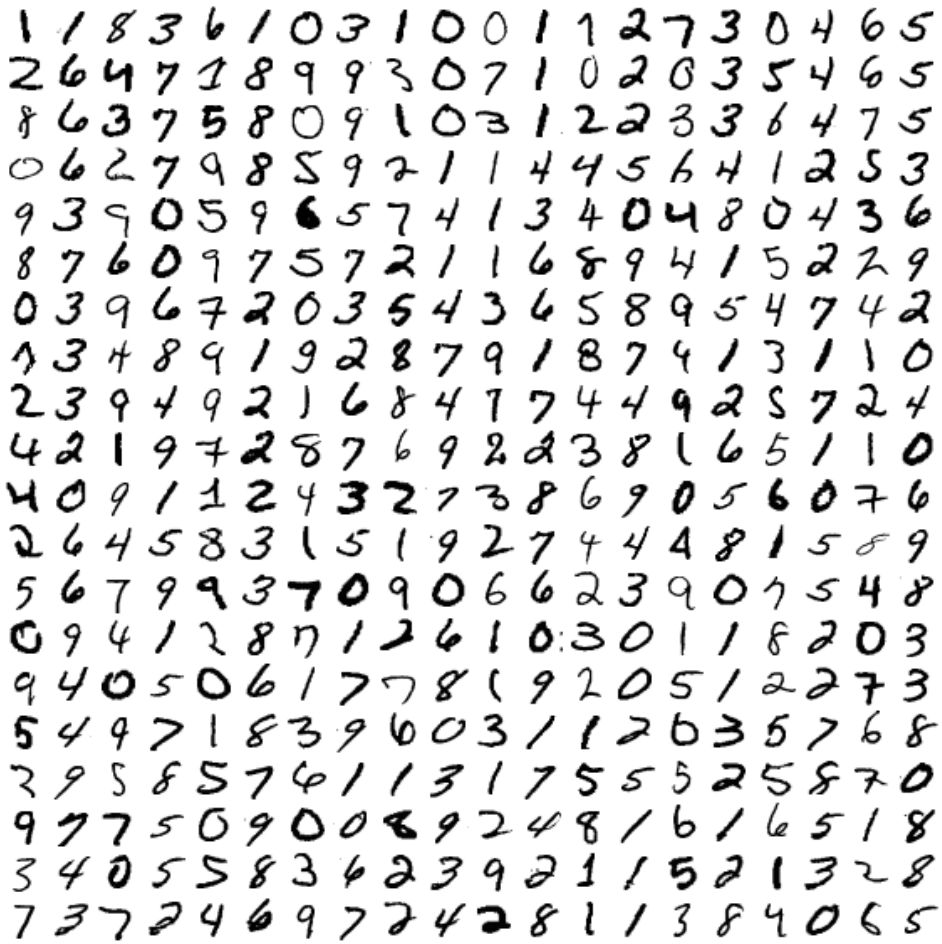
http://pytorch.org

"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(),
    Flattener(),
    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

The end

References

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.

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.

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/.

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.

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.

EE-559 – Deep learning

1.3. What is really happening?

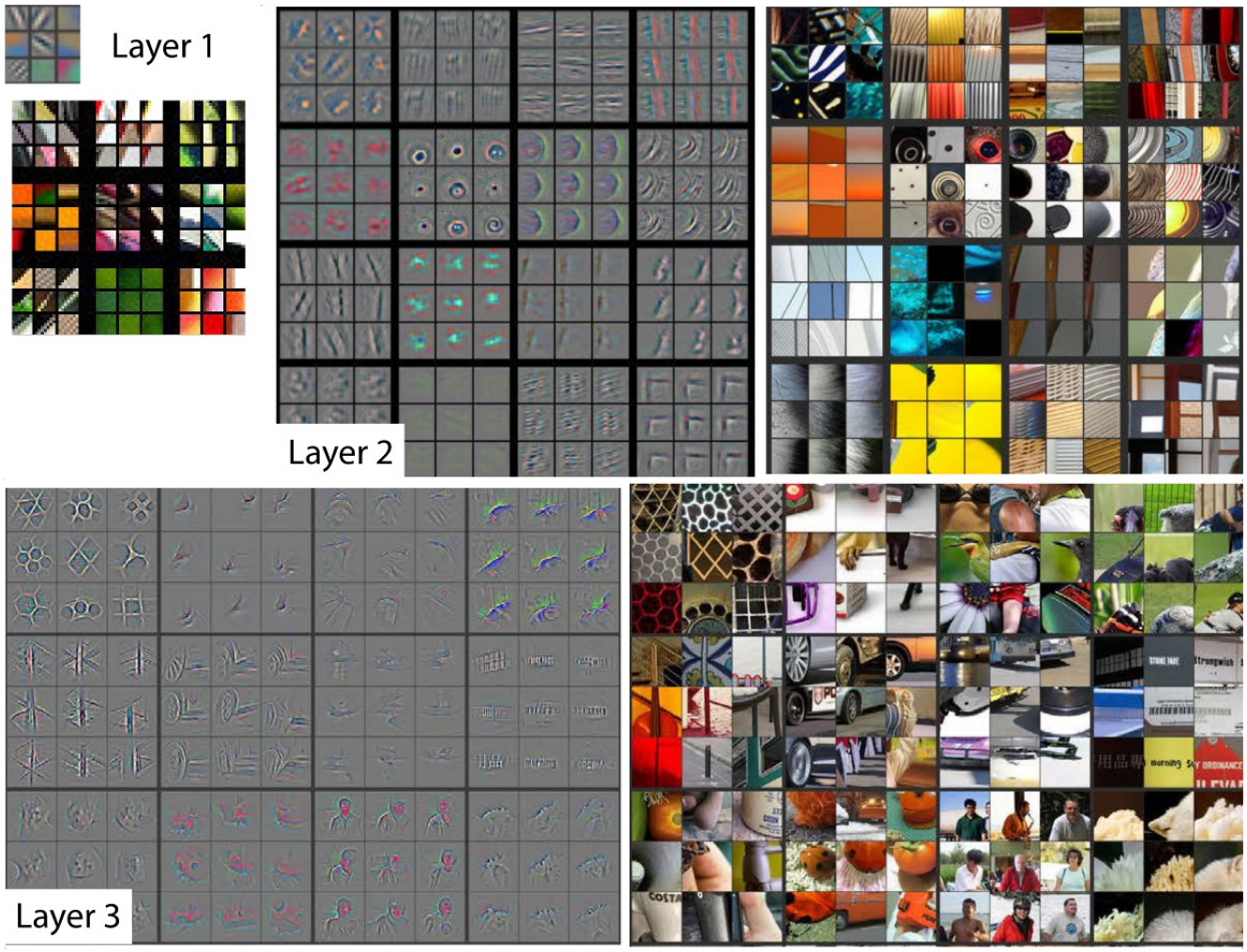
François Fleuret

https://fleuret.org/ee559/

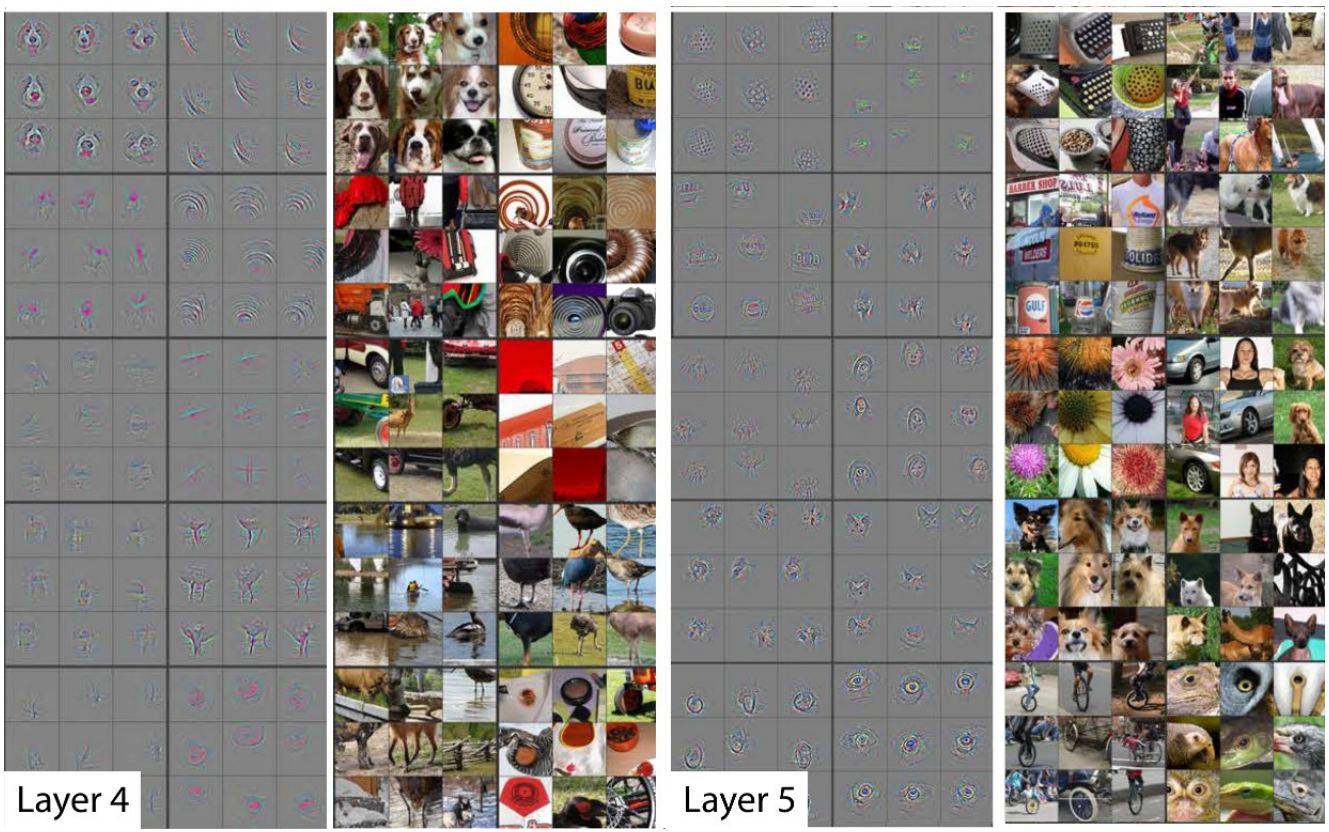
Tue Feb 19 14:19:41 UTC 2019

idiap RESEARCH INSTITUTE

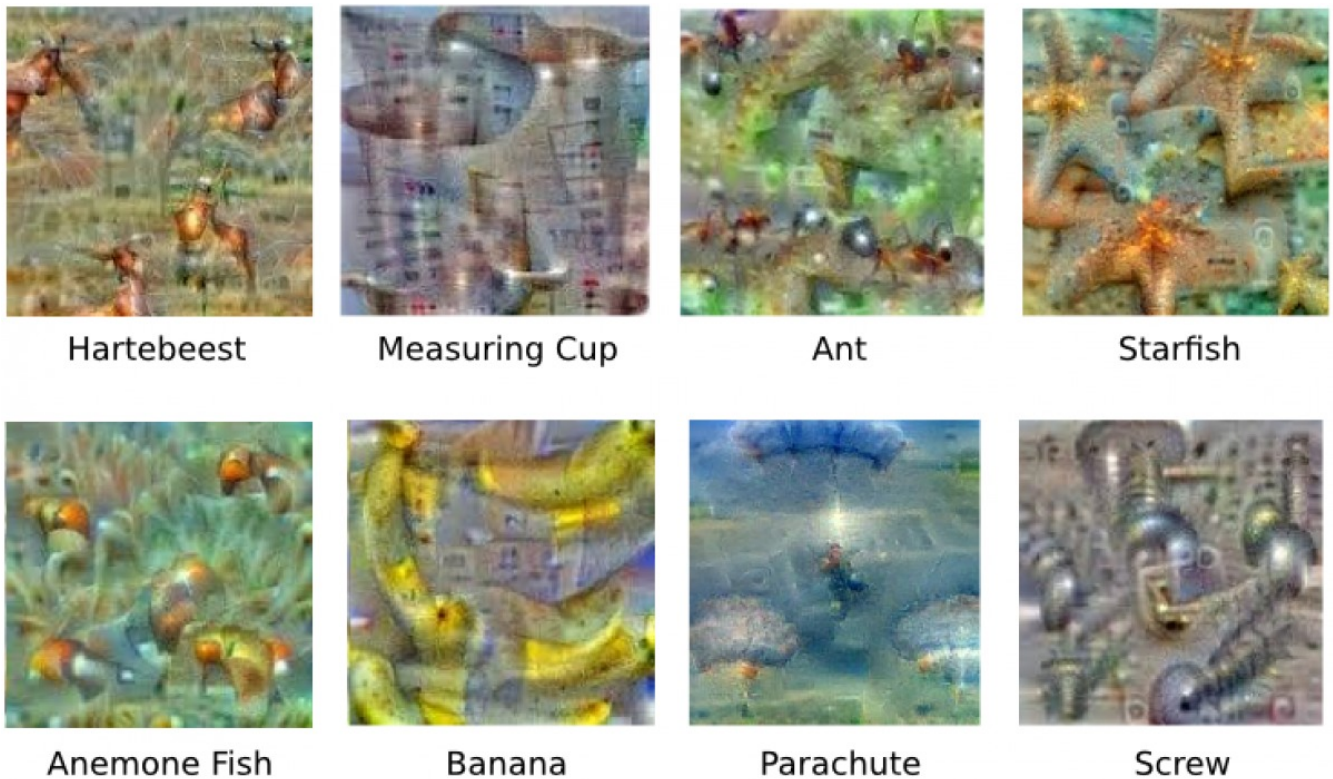
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE



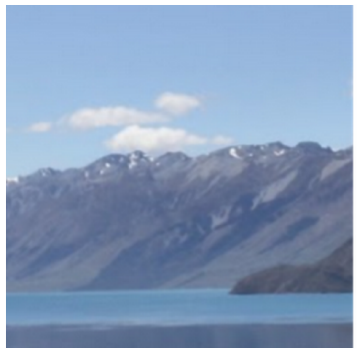
(Zeiler and Fergus, 2014)



(Zeiler and Fergus, 2014)



(Google's Deep Dreams)

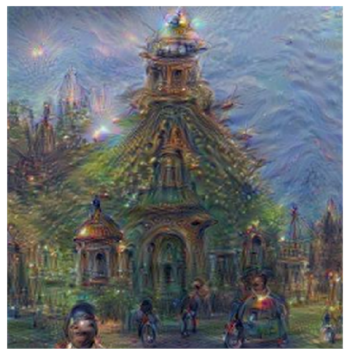


Horizon

Trees

Leaves

Towers & Pagodas

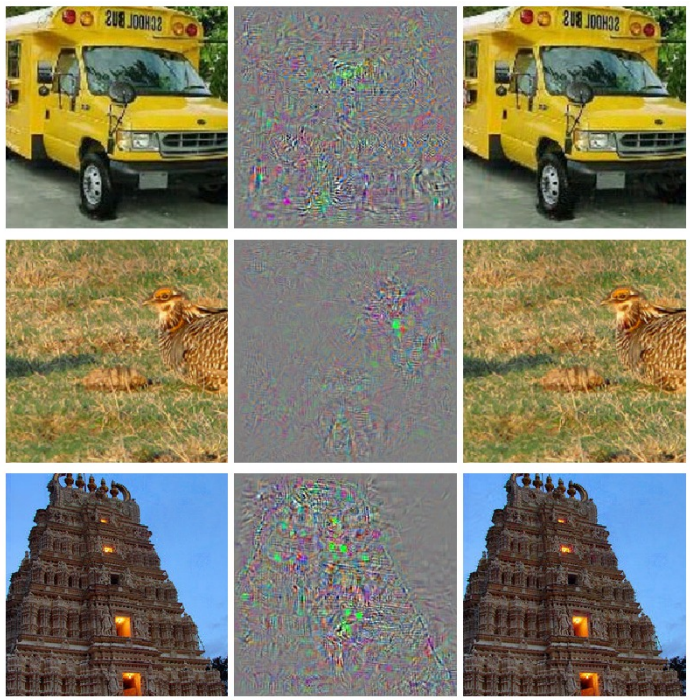


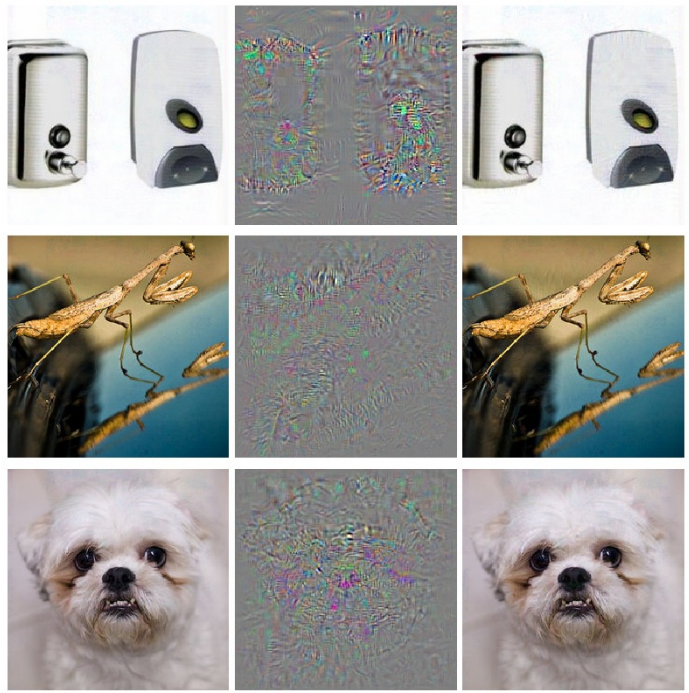
Buildings

Birds & Insects

(Google's Deep Dreams)

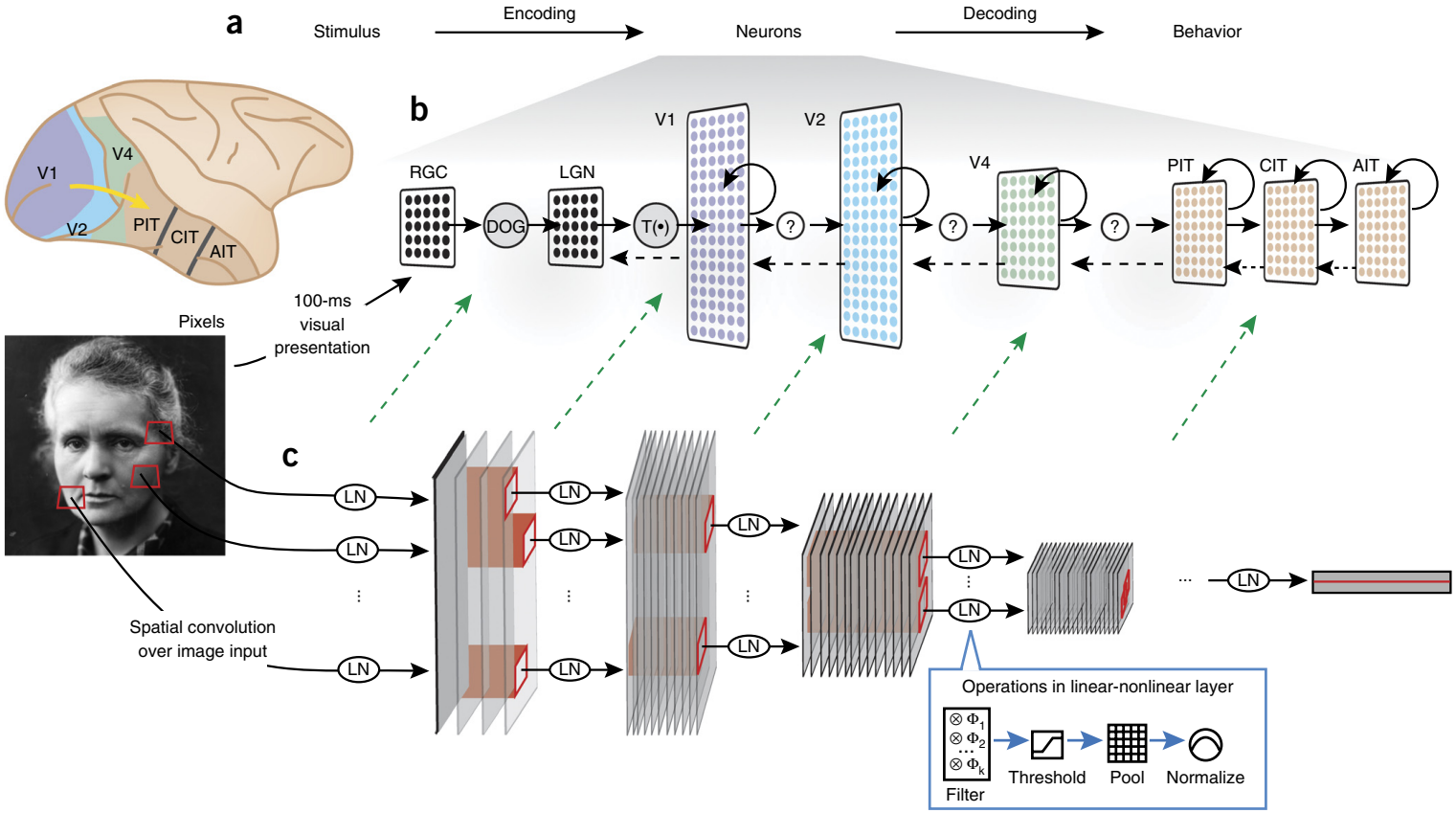
(Thorne Brandt)



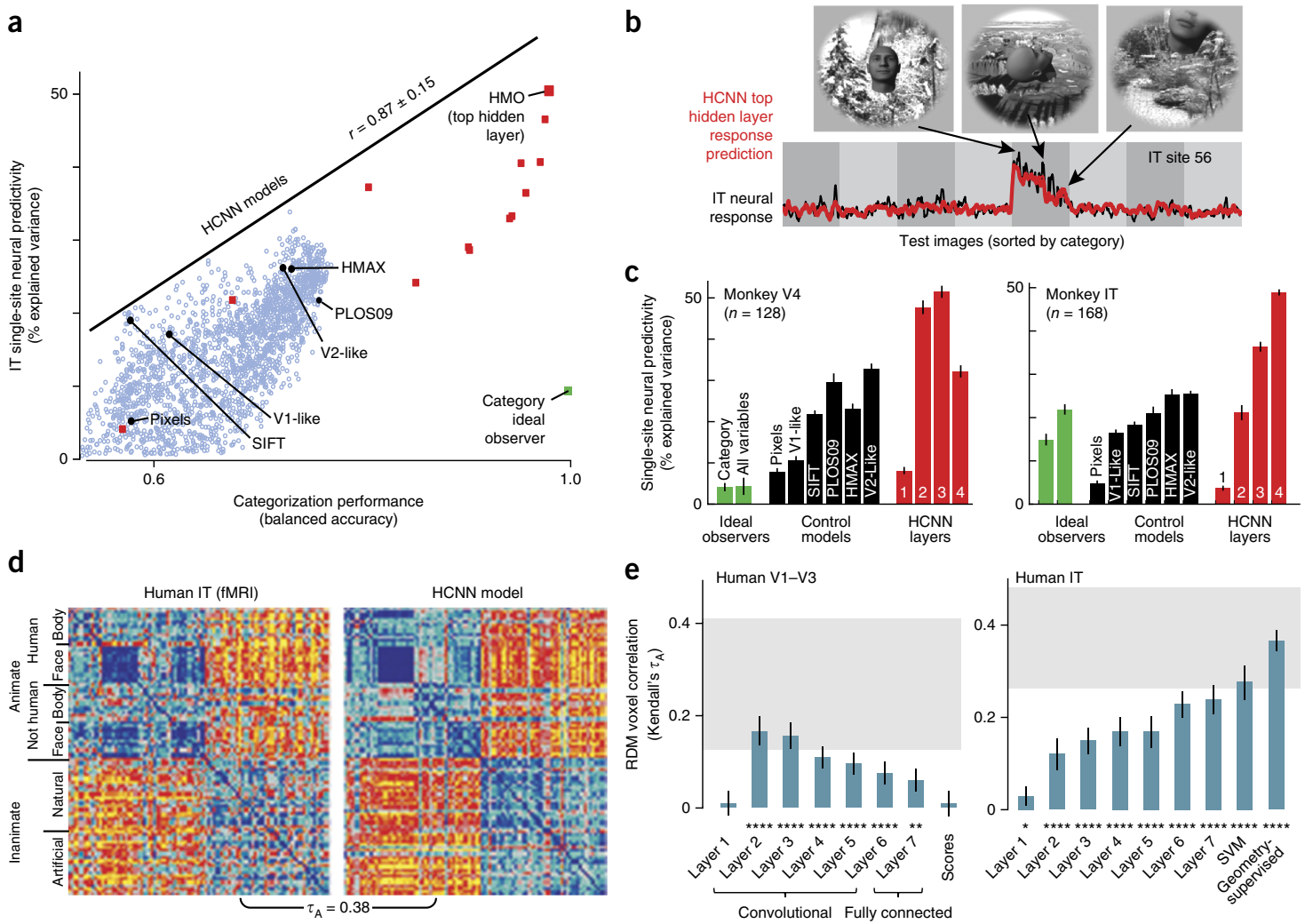


(Szegedy et al., 2014)

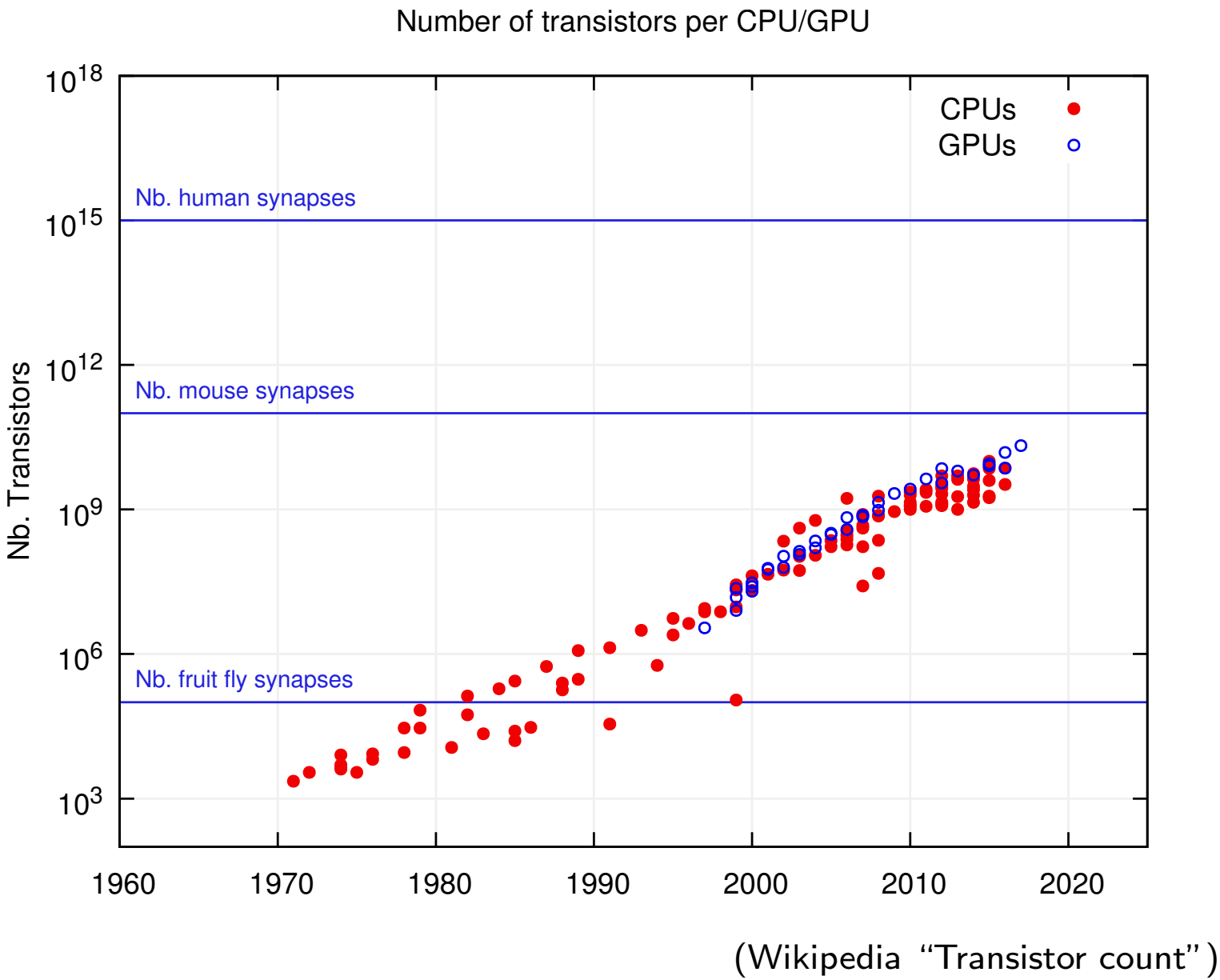
Relations with the biology



(Yamins and DiCarlo, 2016)



(Yamins and DiCarlo, 2016)


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

The end

References

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*, 2014.

D. L. K. Yamins and J. J. DiCarlo. Using goal-driven deep learning models to understand sensory cortex. *Nature neuroscience*, 19:356–65, Feb 2016.

M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision (ECCV)*, 2014.

EE-559 – Deep learning

1.4. Tensor basics and linear regression

François Fleuret

https://fleuret.org/ee559/

Mon Feb 18 13:32:52 UTC 2019

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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

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.

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.

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.

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.

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.)

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
```

```
>>> 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()`.

```
>>> 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

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

can we find the "best line"

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

going "through the points"

Given a list of points

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

can we find the "best line"

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

going "through the points", *e.g.* minimizing the mean square error

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

Given a list of points

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

can we find the "best line"

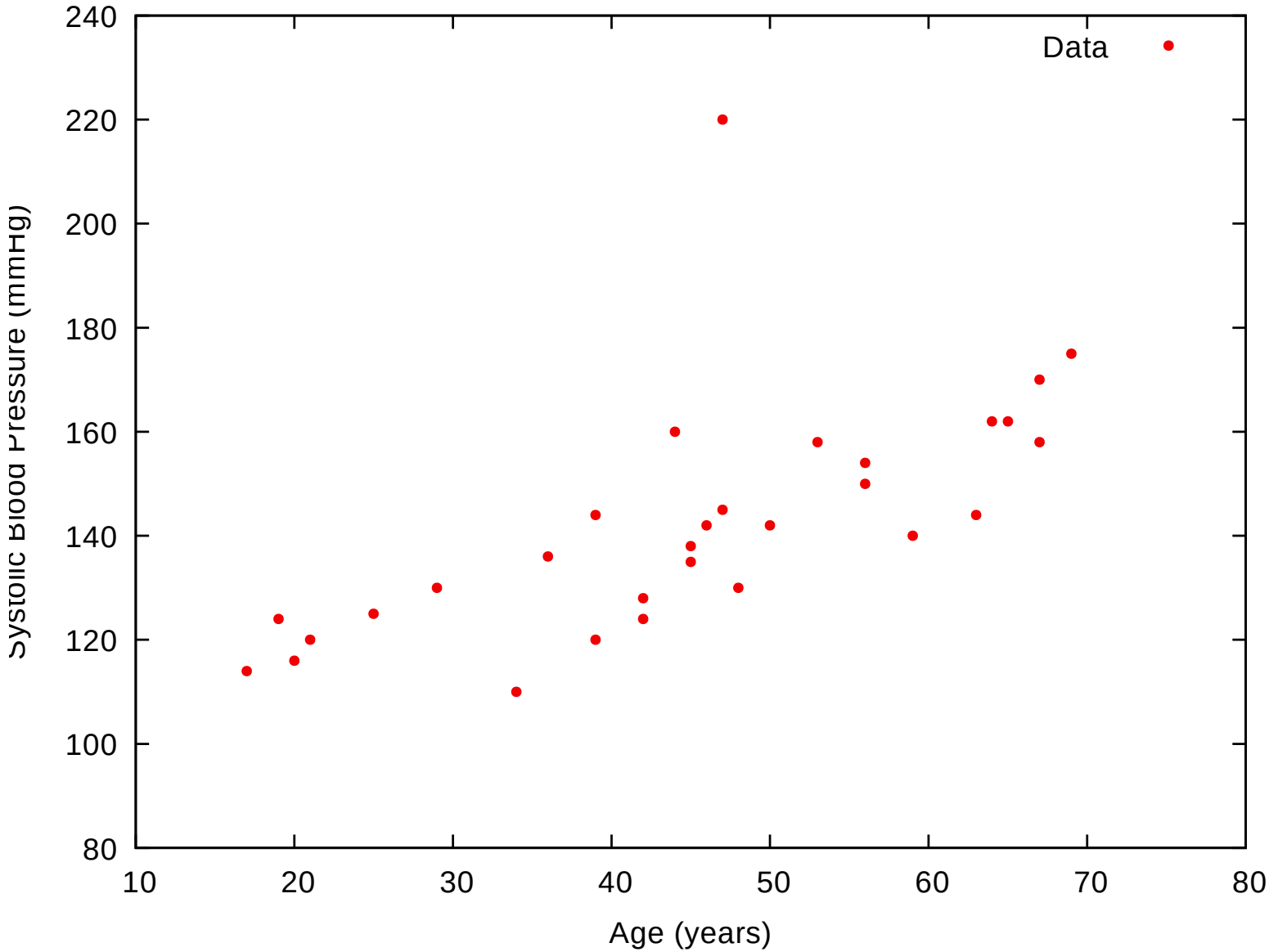
$$f(x; a, b) = ax + b$$

going "through the points", *e.g.* minimizing the mean square error

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

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

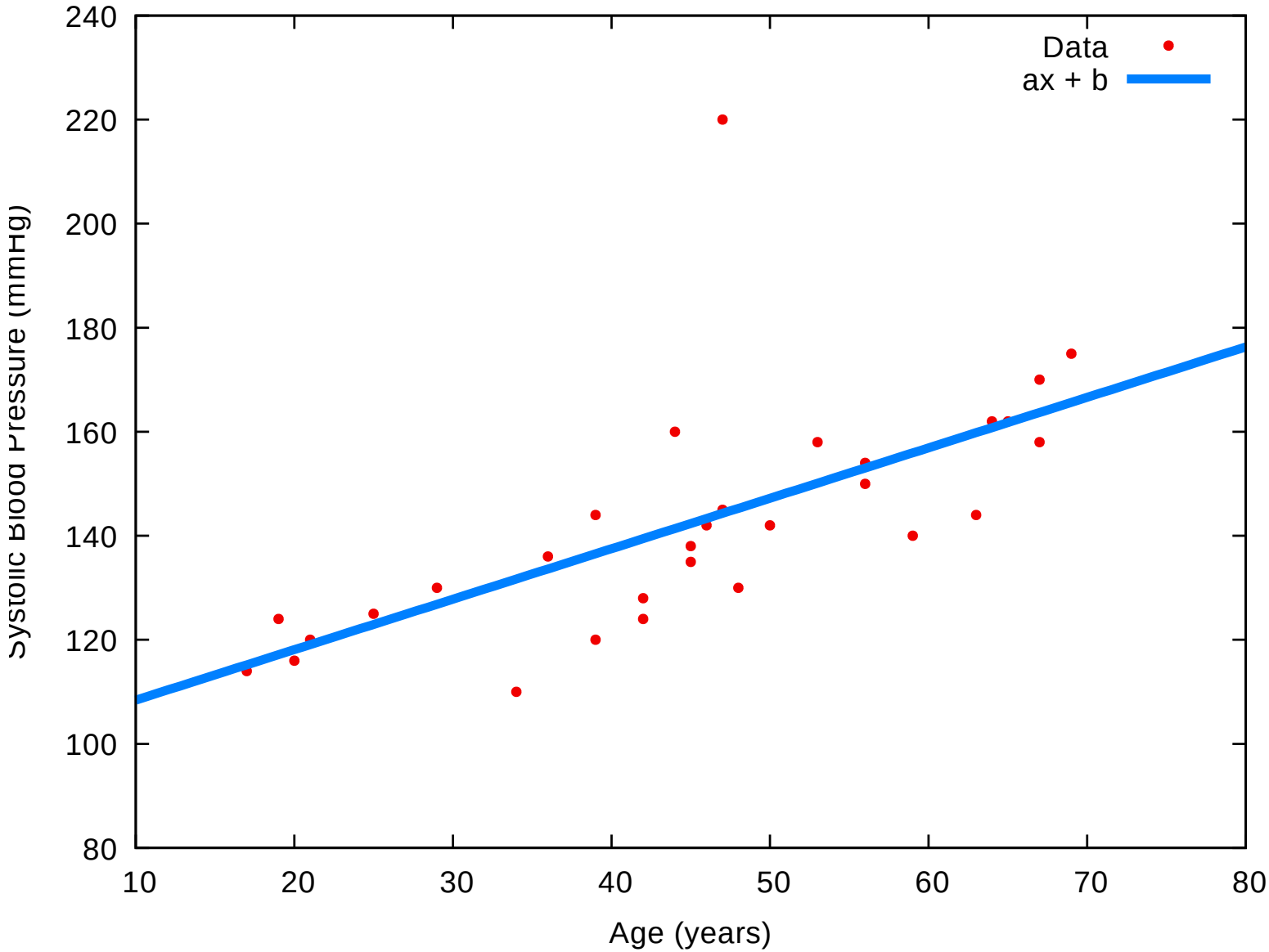
$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}}$$

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

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

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

```
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.gels(y, x)

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


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

The end

EE-559 – Deep learning

1.5. High dimension tensors

François Fleuret

https://fleuret.org/ee559/

Mon Feb 18 15:52:38 UTC 2019

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

A tensor can be of several types:

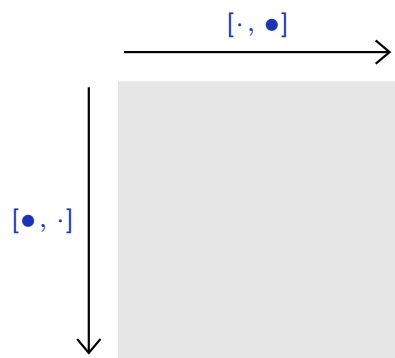
- `torch.float16`, `torch.float32`, `torch.float64`,
- `torch.uint8`,
- `torch.int8`, `torch.int16`, `torch.int32`, `torch.int64`

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

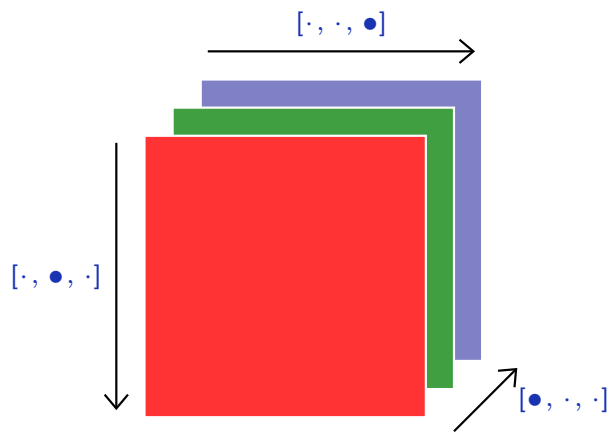
Operations with tensors stored in a certain device's memory are done by that device. We will come back to that later.

```
>>> 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))
```

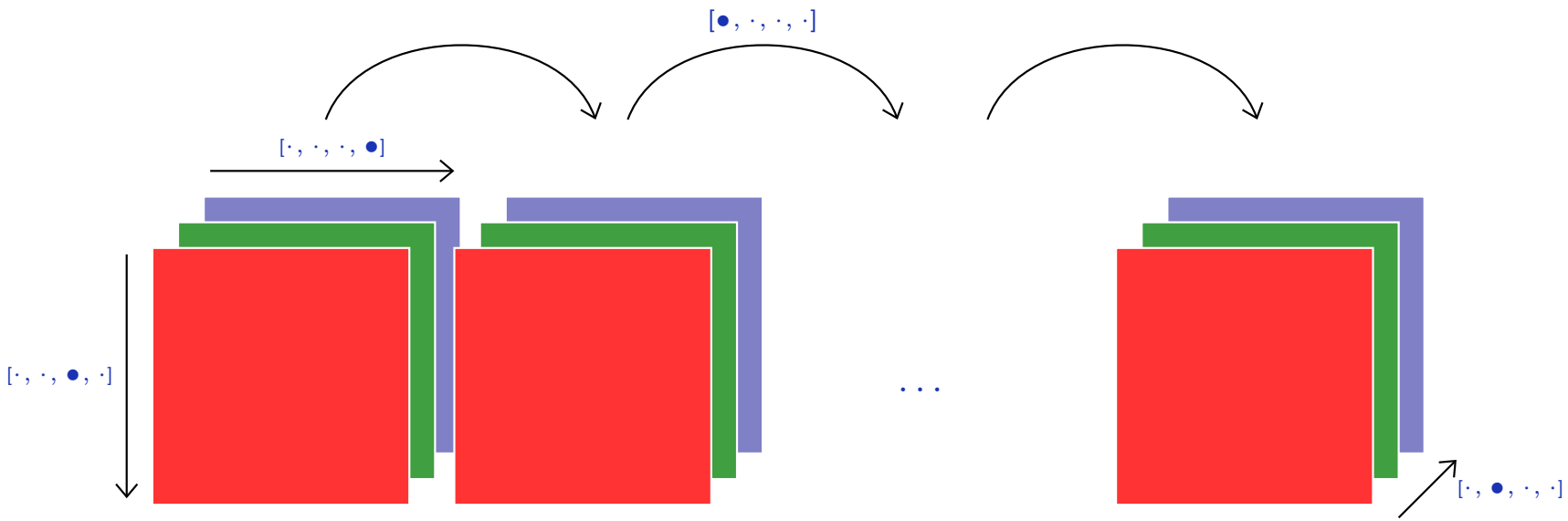
2d tensor (*e.g.* grayscale image)



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

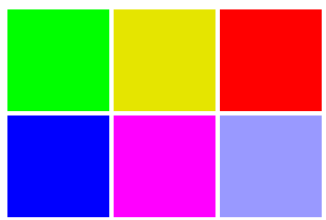
- `torch.abs(input, out=None)`
- `torch.add()`
- `torch.cos(input, out=None)`
- `torch.sigmoid(input, out=None)`
- `(+ many operators)`

Math reduction

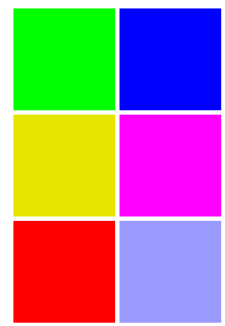
- `torch.dist(input, other, p=2, out=None)`
- `torch.mean()`
- `torch.norm()`
- `torch.std()`
- `torch.sum()`

BLAS and LAPACK Operations

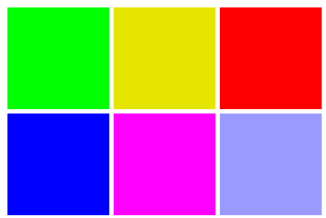
- `torch.eig(a, eigenvectors=False, out=None)`
- `torch.gels(B, A, out=None)`
- `torch.inverse(input, out=None)`
- `torch.mm(mat1, mat2, out=None)`
- `torch.mv(mat, vec, out=None)`



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

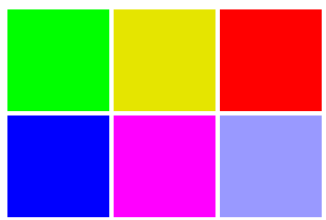


```
x.t()
```

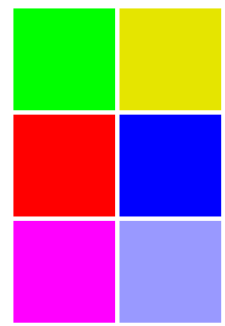


```
x.view(-1)
```

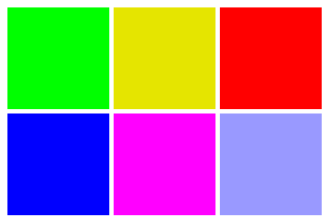
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



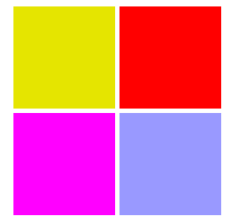
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



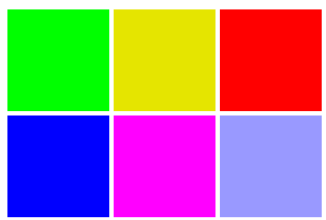
```
x.view(3, -1)
```



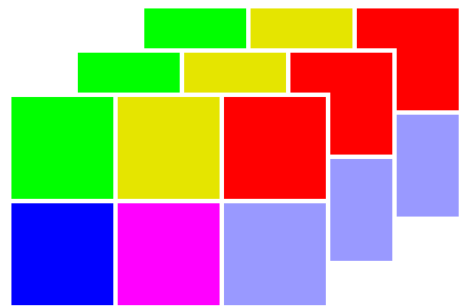
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



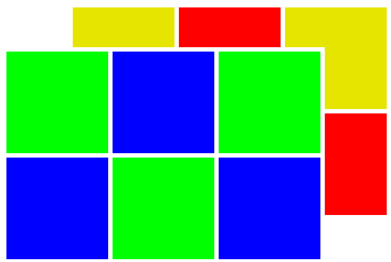
```
x.narrow(1, 1, 2)
```



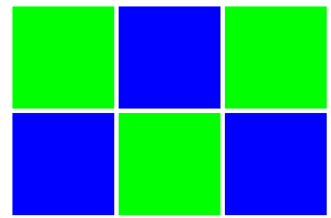
```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



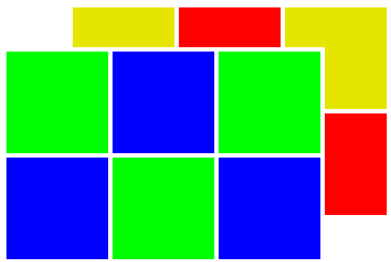
```
x.view(1, 2, 3).expand(3, 2, 3)
```



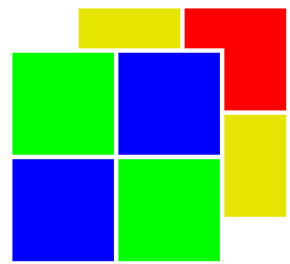
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



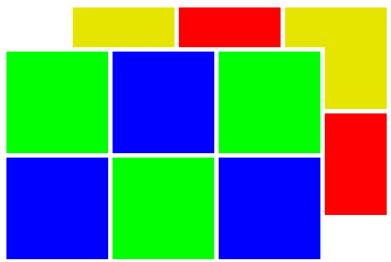
```
x.narrow(0, 0, 1)
```



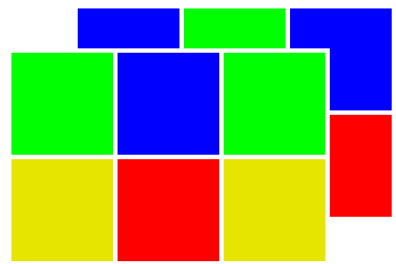
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



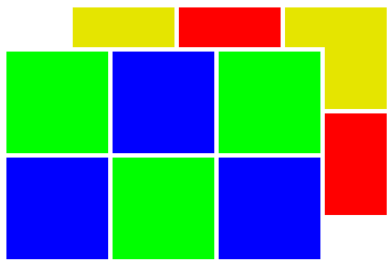
```
x.narrow(2, 0, 2)
```



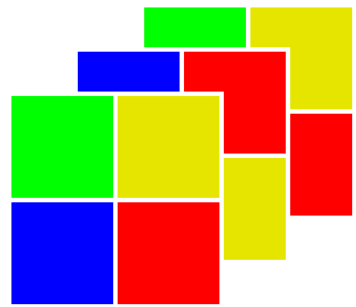
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



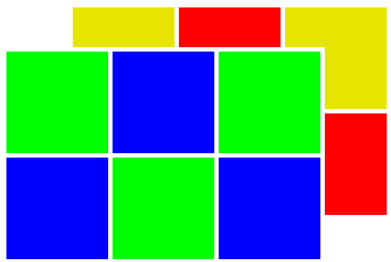
```
x.transpose(0, 1)
```



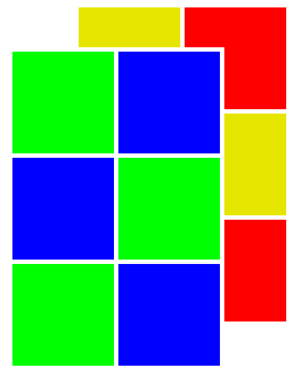
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



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



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



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

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())
```

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

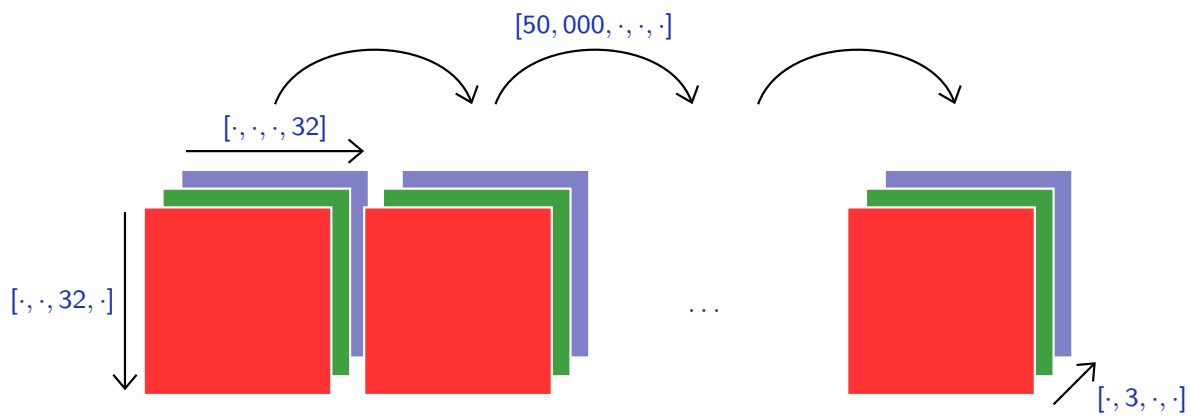
```
Files already downloaded and verified
torch.FloatTensor torch.Size([50000, 3, 32, 32]) 0.0 1.0
```

PyTorch offers simple interfaces to standard image data-bases.

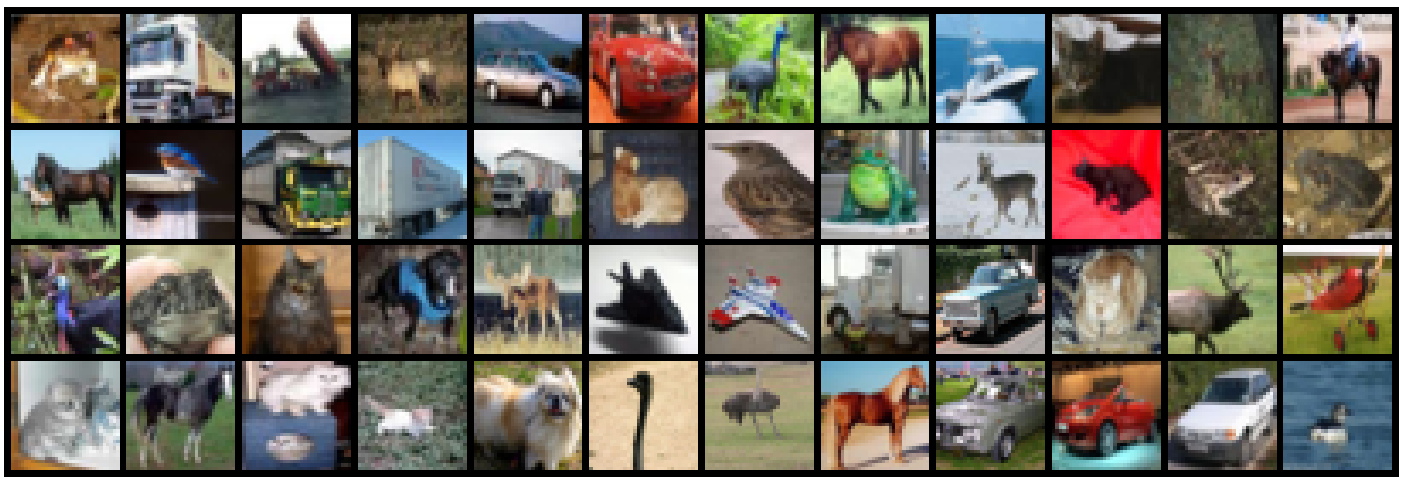
```
import torch, torchvision
cifar = torchvision.datasets.CIFAR10('./cifar10/', train = True, download = True)
x = torch.from_numpy(cifar.train_data).transpose(1, 3).transpose(2, 3).float()
x = x / 255
print(x.type(), x.size(), x.min().item(), x.max().item())
```

prints

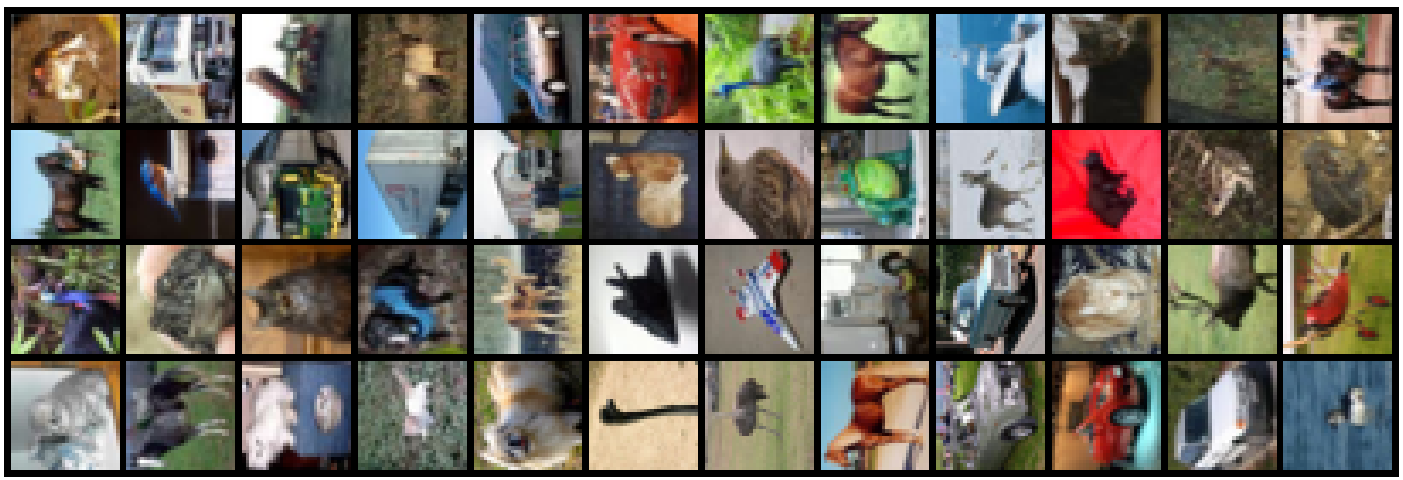
```
Files already downloaded and verified
torch.FloatTensor torch.Size([50000, 3, 32, 32]) 0.0 1.0
```



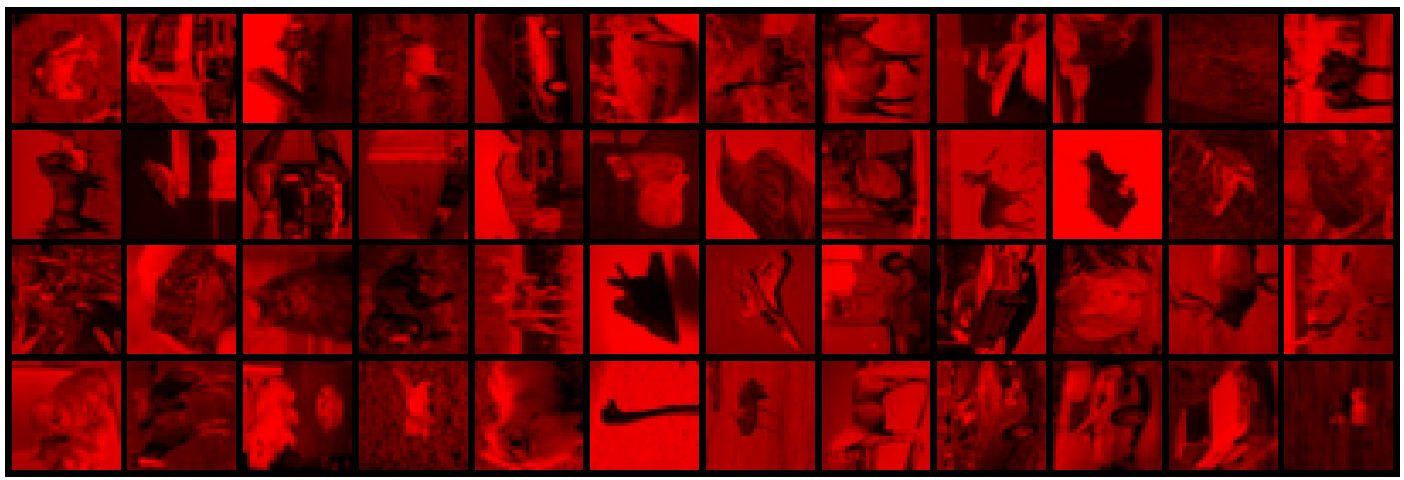
```
# Narrows to the first images, converts to float
x = x.narrow(0, 0, 48).float()

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

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

For instance:

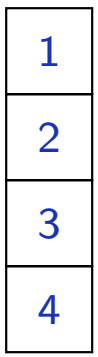
```
>>> x = torch.empty(100, 4).normal_(2)
>>> x.mean(0)
tensor([2.0476, 2.0133, 1.9109, 1.8588])
>>> x -= x.mean(0) # This should not work!
>>> x.mean(0)
tensor([-4.0531e-08, -4.4703e-07, -1.3471e-07,  3.5763e-09])
```

Precisely, broadcasting proceeds as follows:

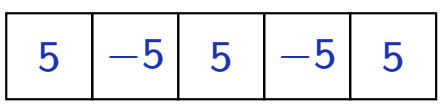
1. If one of the tensors has fewer dimensions than the other, it is reshaped by adding as many dimensions of size 1 as necessary in the front; then
2. for every dimension mismatch, **if one of the two tensors is of size one,** it is expanded along this axis by replicating coefficients.

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
```

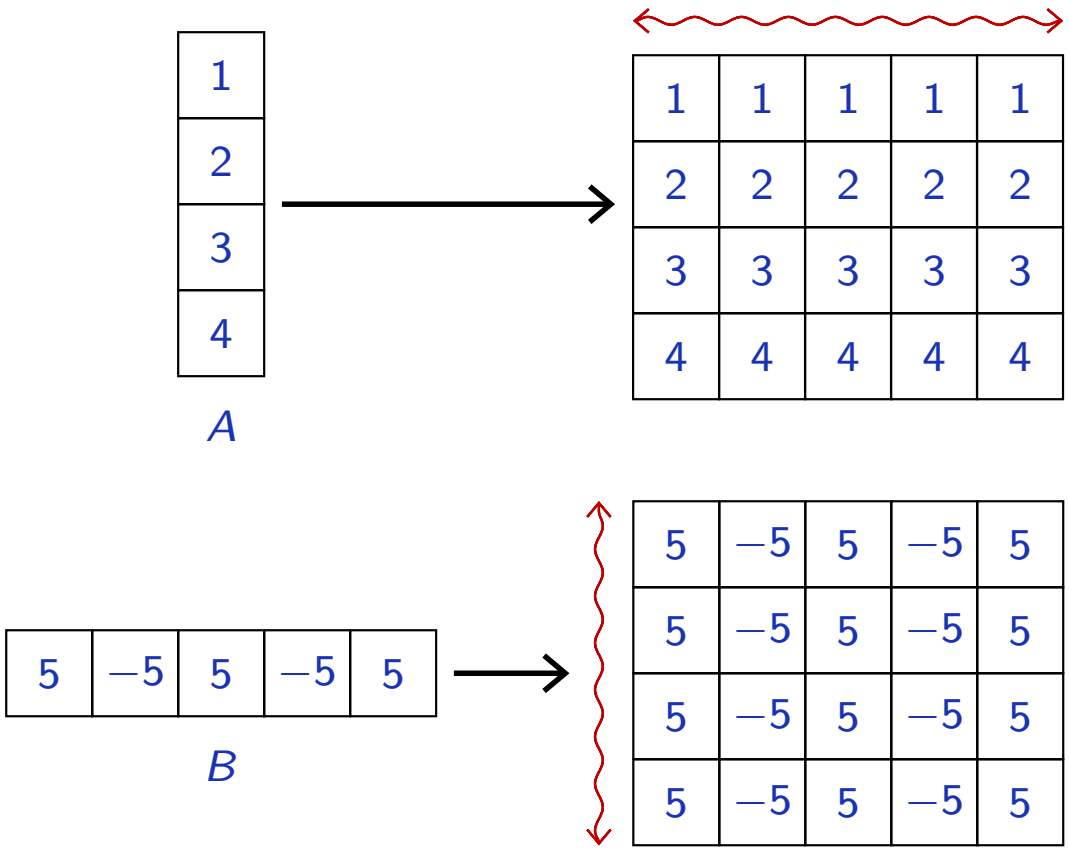


A



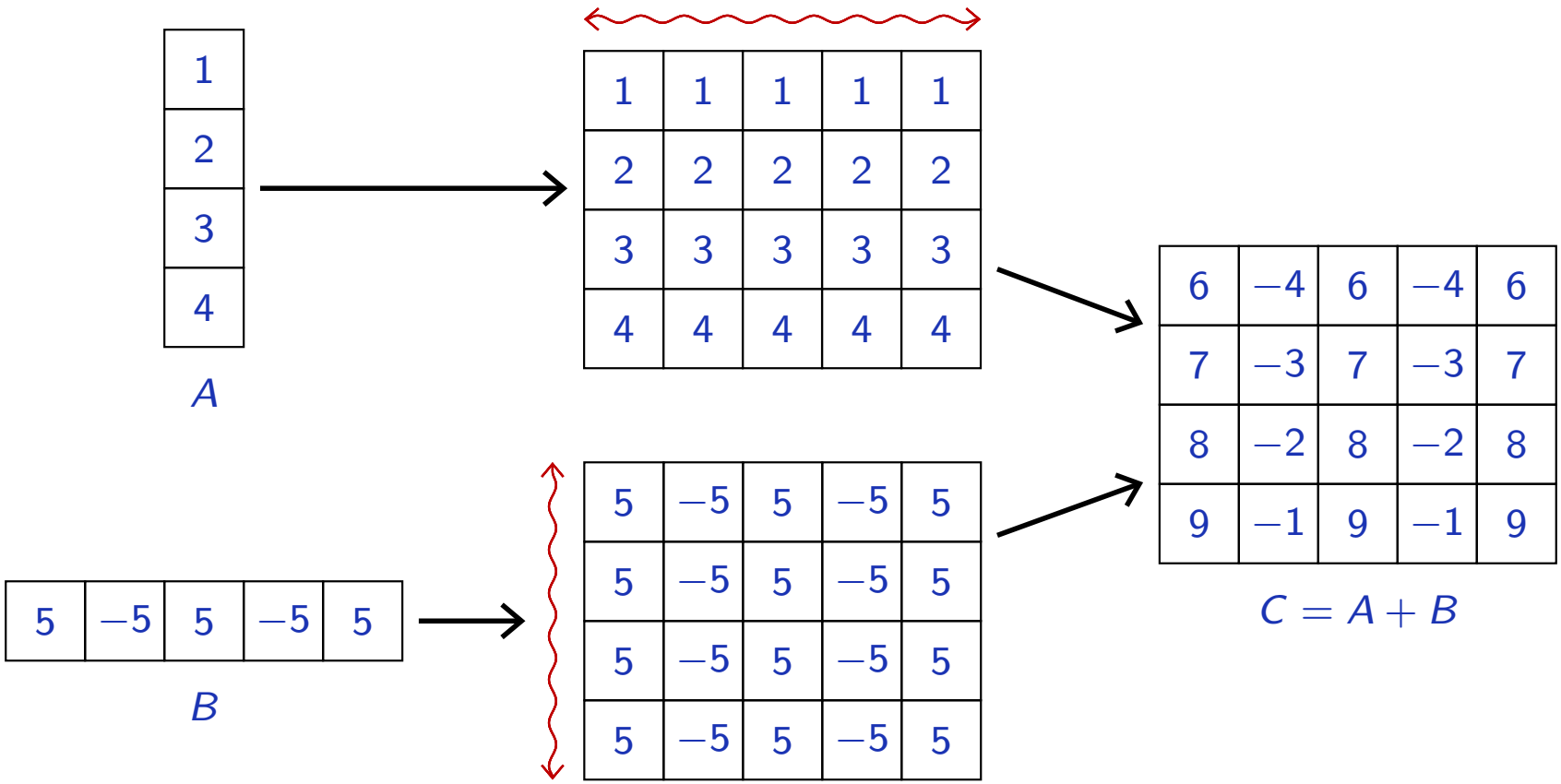
B

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



Broadcasted

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



Broadcasted

The end

EE-559 – Deep learning

1.6. Tensor internals

François Fleuret

https://fleuret.org/ee559/

Mon Feb 18 13:32:55 UTC 2019

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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.]])
```

Multiple tensors can share the same storage. It happens when using operations such as `view()`, `expand()` or `transpose()`.

```
>>> y = x.view(2, 2, 2)
>>> y
tensor([[[ 0.,  0.],
         [ 0.,  0.]],

        [[ 1.,  0.],
         [ 0.,  0.]]])
>>> y[1, 1, 0] = 7.0
>>> x
tensor([[ 0.,  0.,  0.,  0.],
        [ 1.,  0.,  7.,  0.]])
>>> y.narrow(0, 1, 1).fill_(3.0)
tensor([[[ 3.,  3.],
         [ 3.,  3.]]])
>>> x
tensor([[ 0.,  0.,  0.,  0.],
        [ 3.,  3.,  3.,  3.]])
```

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

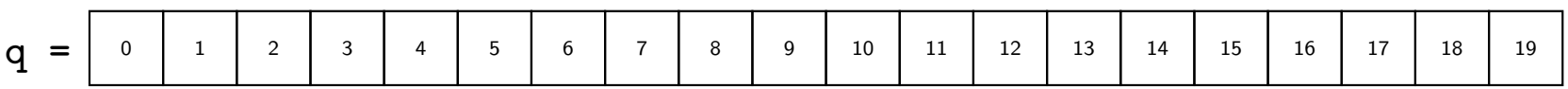
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by `1`, you have to move by `stride(k)` elements in the storage.

The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> 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.]])
```

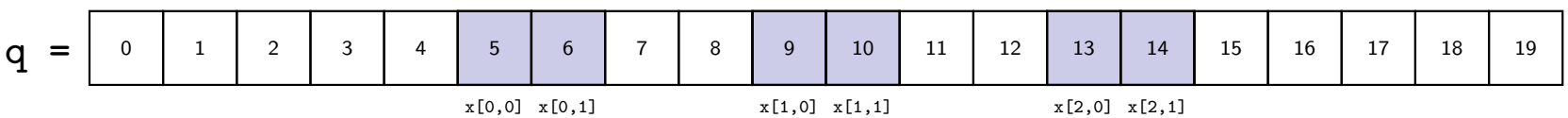
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> 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.]])
```



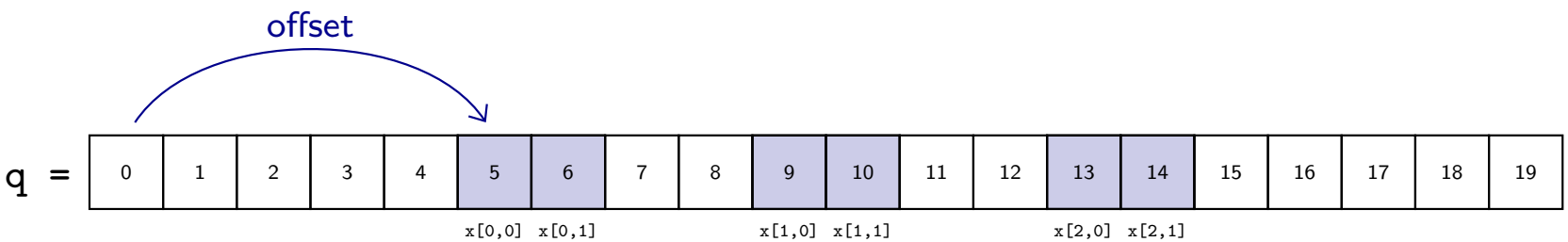
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> 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.]])
```



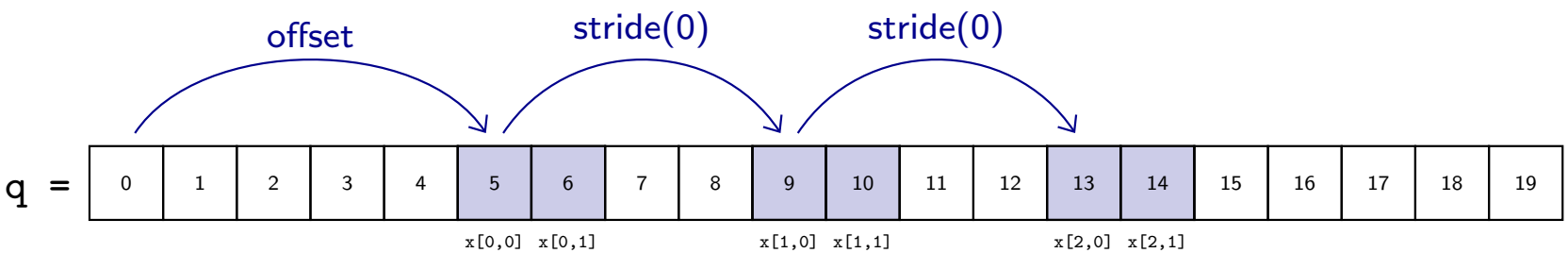
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> 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.]])
```



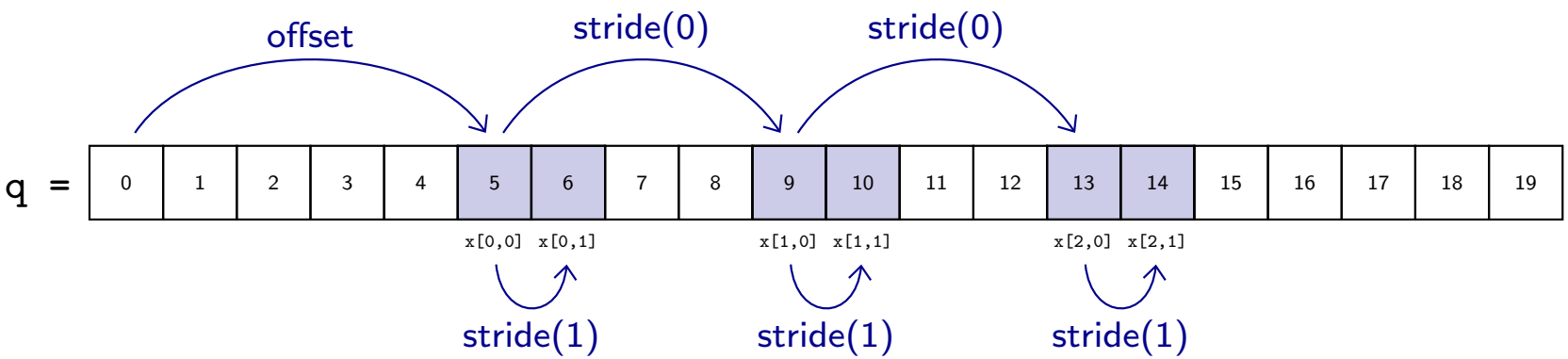
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> 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.]])
```



The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by `1`, you have to move by `stride(k)` elements in the storage.

```
>>> 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.]])
```

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.

This organization explains the following (maybe surprising) error

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

`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