Deep Learning

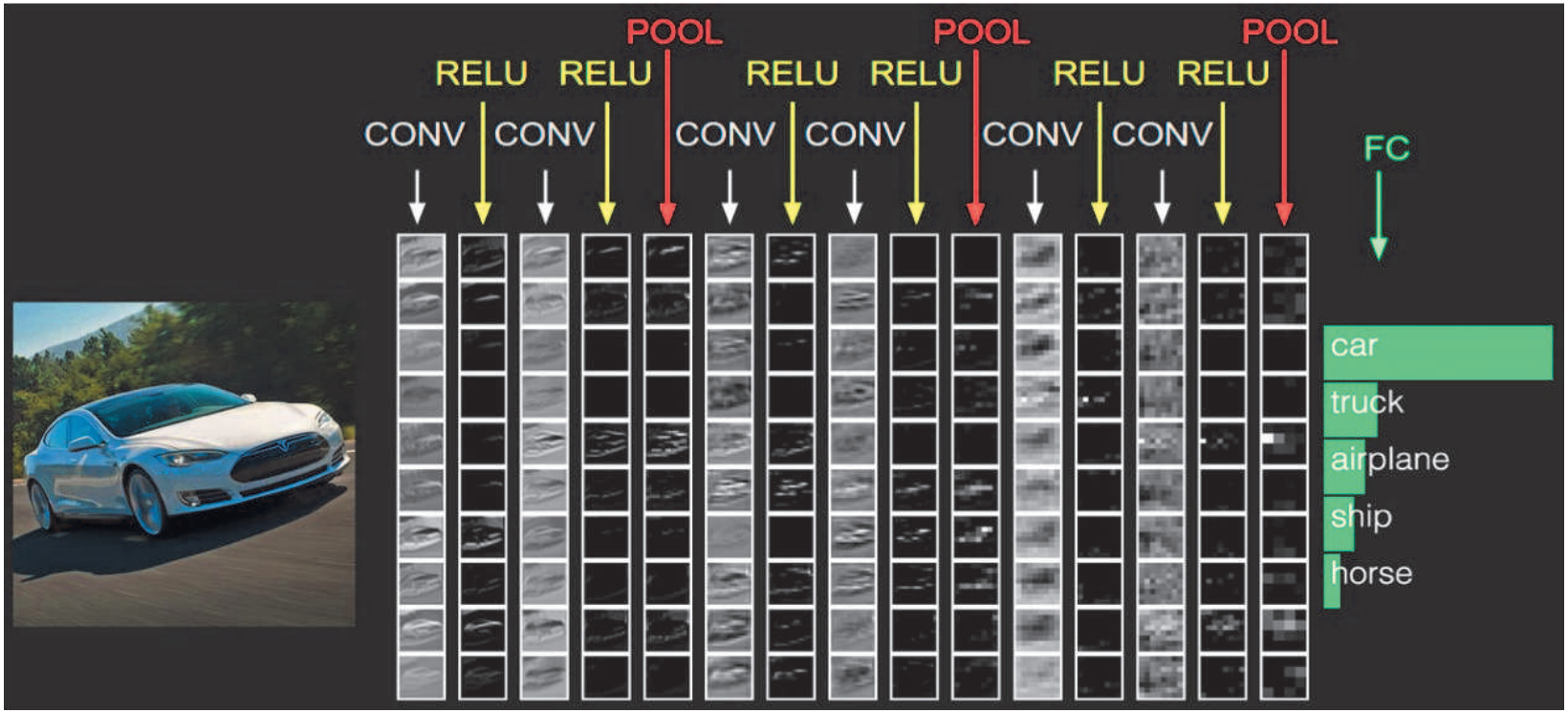
A journey from feature extraction and engineering to end-to-end pipelines

Part 4: Under the hood

Andrei Bursuc

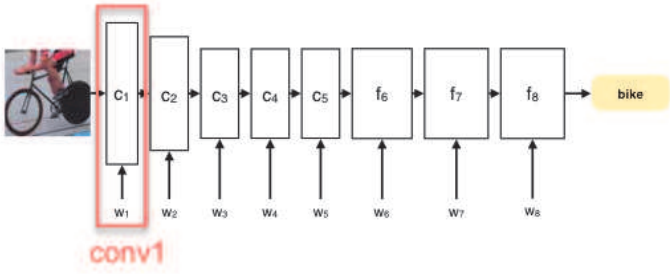
With slides from A. Karpathy, F. Fleuret, J. Johnson, S. Yeung, G. Louppe, Y. Avrithis ...

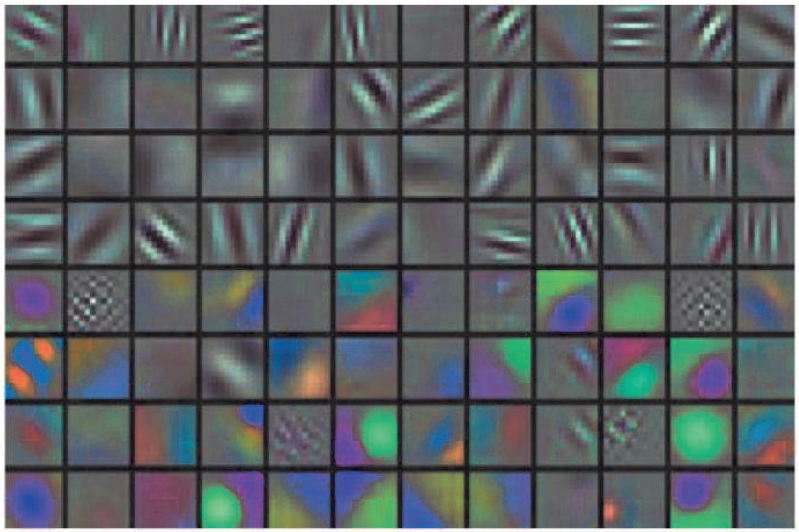
Understanding and visualizing CNNs

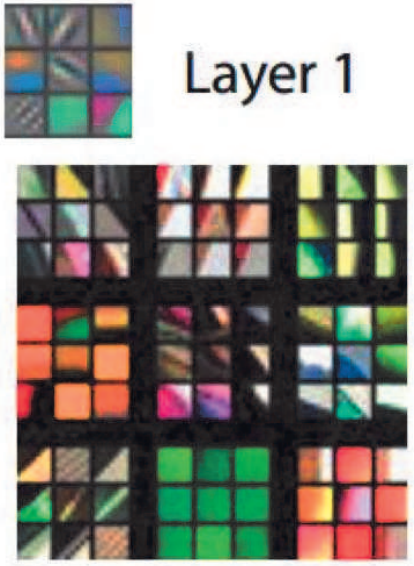


What happens inside a CNN?

Visualize first layers filters/weights



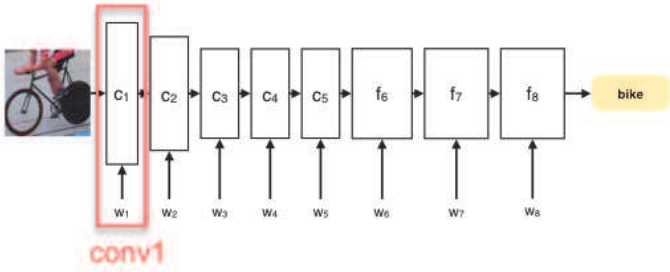


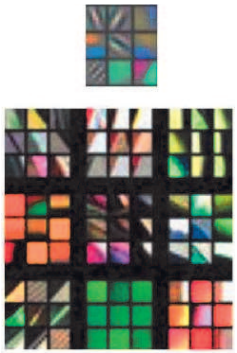


Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

What happens inside a CNN?

Visualize first layers filters/weights



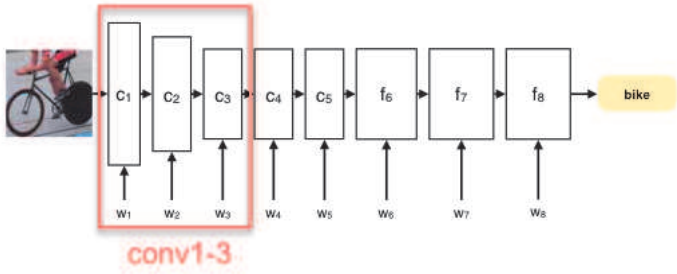


Layer 1

Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

What happens inside a CNN?

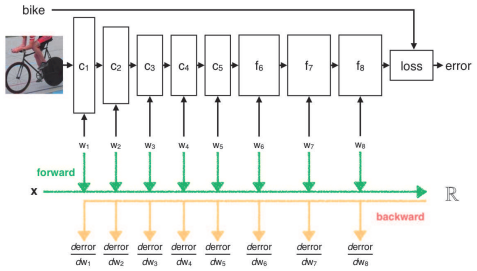
- Visualize behavior in higher layers
- We can visualize filters at higher layers, but they are less intuitive

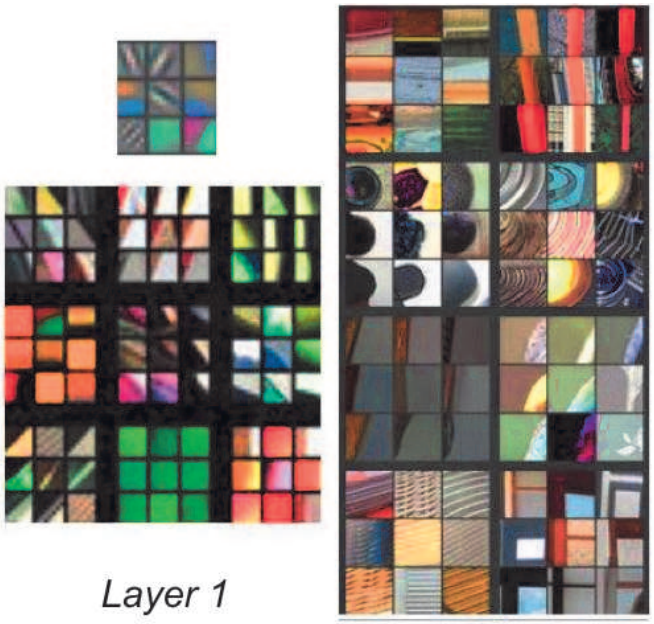


Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

What happens inside a CNN?

Visualize first layers filters/weights

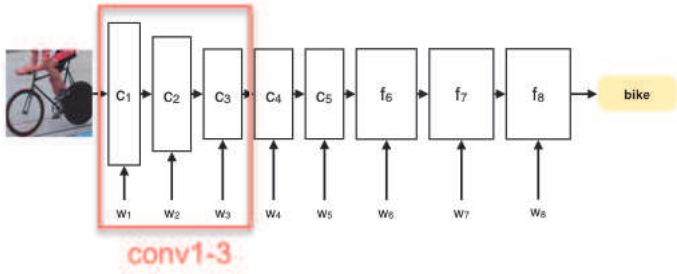




Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

What happens inside a CNN?

Visualize first layers
filters/weights



Layer 1

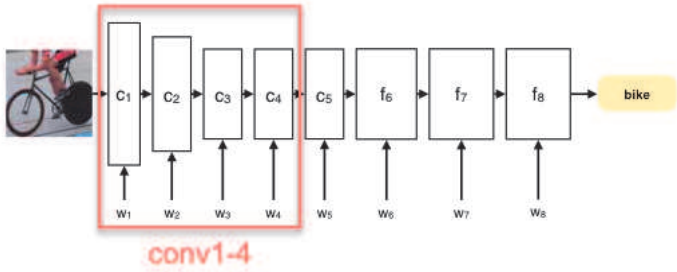
Layer 2

Layer 3

Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

What happens inside a CNN?

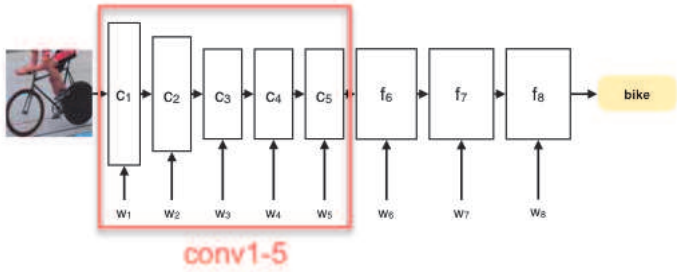
Visualize first layers
filters/weights



Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

What happens inside a CNN?

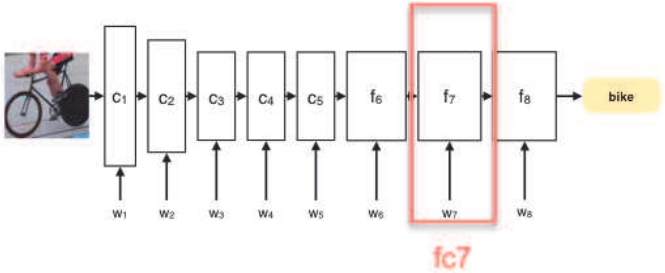
Visualize first layers filters/weights



Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

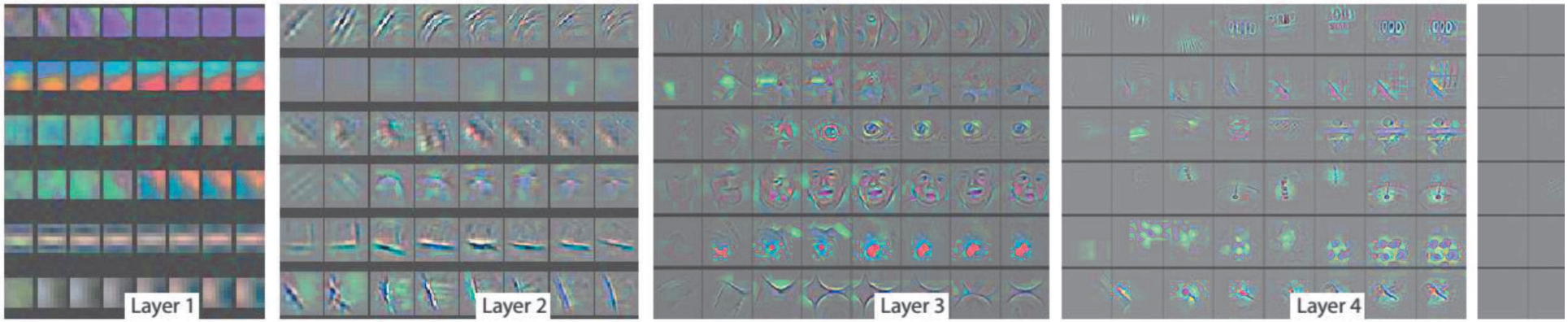
What happens inside a CNN?

- 4096d "signature" for an image (layer right before the classifier)
- Visualize with t-SNE: here



Feature evolution during training

- For a particular neuron (that generates a feature map)
- Pick the strongest activation during training
- For epochs 1, 2, 5, 10, 20, 30, 40, 64



Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

Visualize layer activations/feature maps

AlexNet

Figure credit: F. Fleuret

Visualize layer activations/feature maps

AlexNet

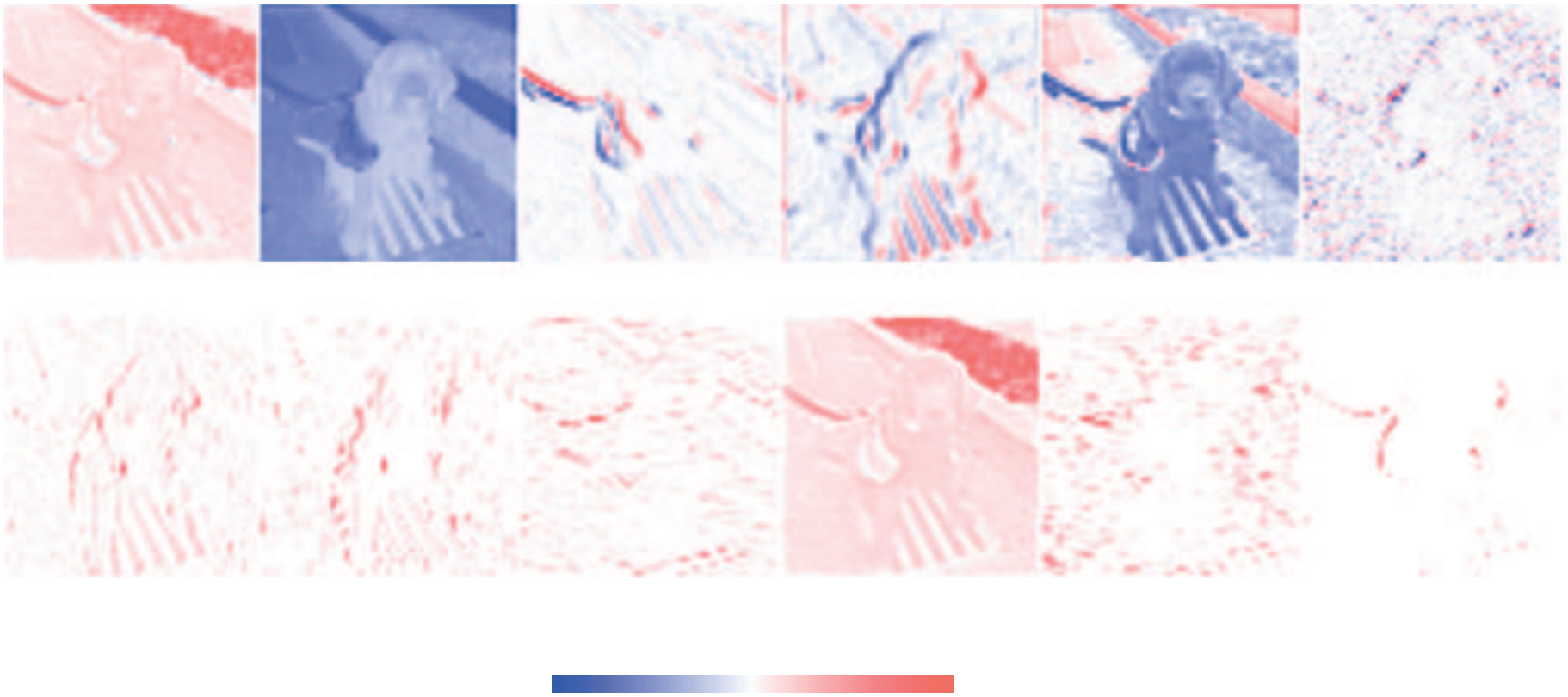


Figure credit: F. Fleuret

Visualize layer activations/feature maps

AlexNet

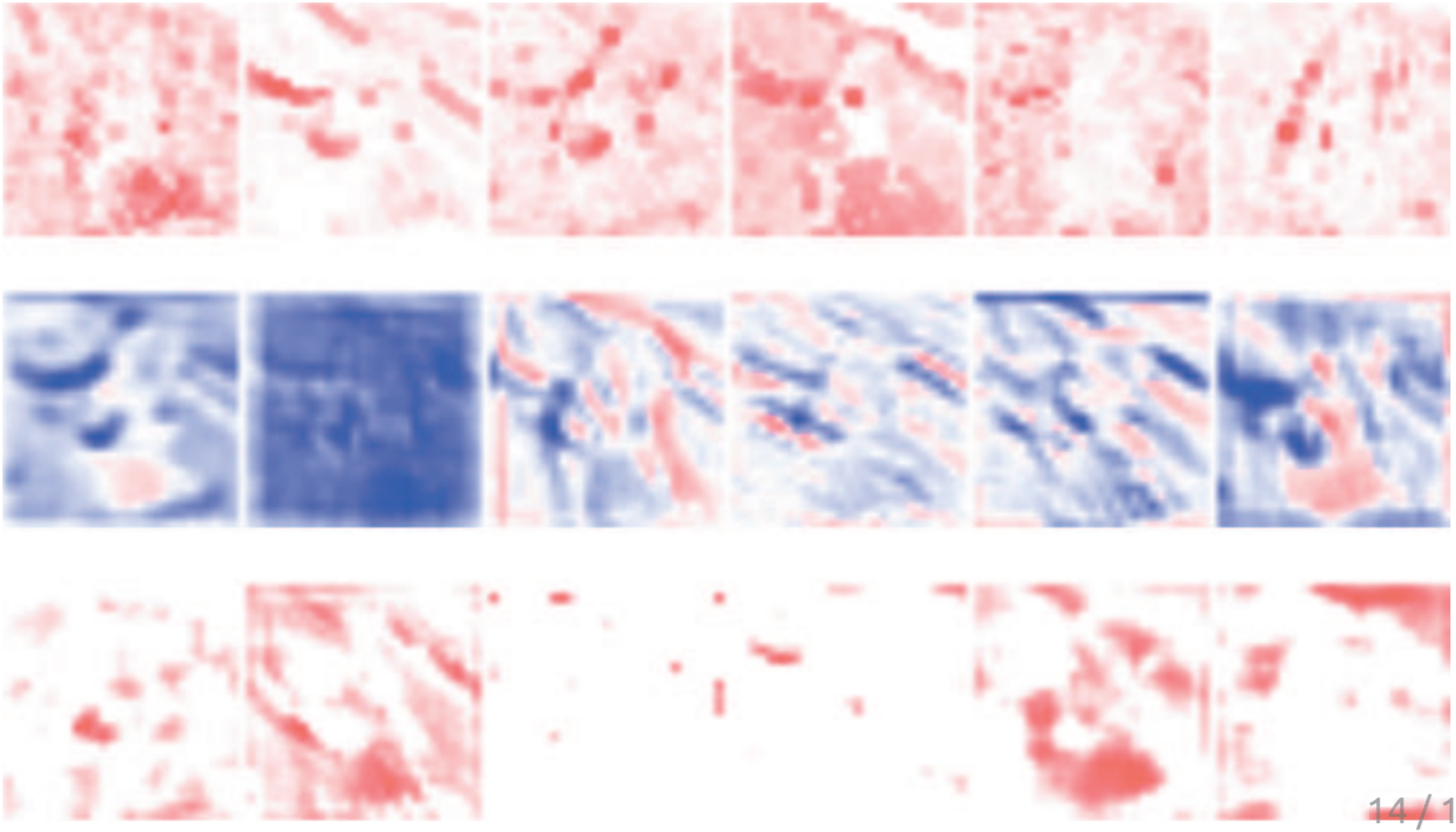


Figure credit: F. Fleuret

Visualize layer activations/feature maps

AlexNet

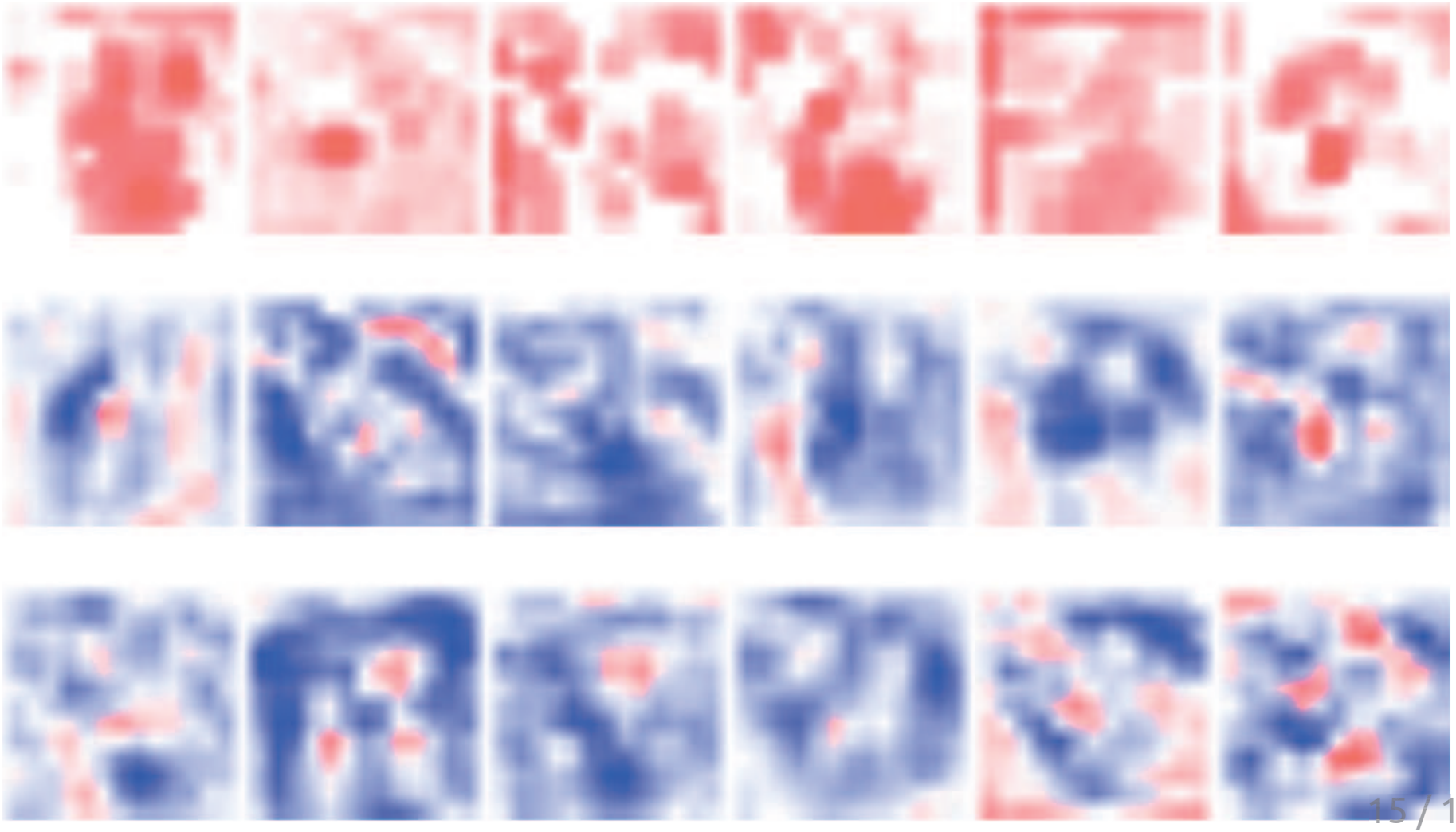


Figure credit: F. Fleuret

Visualize layer activations/feature maps

AlexNet

Figure credit: F. Fleuret

Visualize layer activations/feature maps

ResNet152

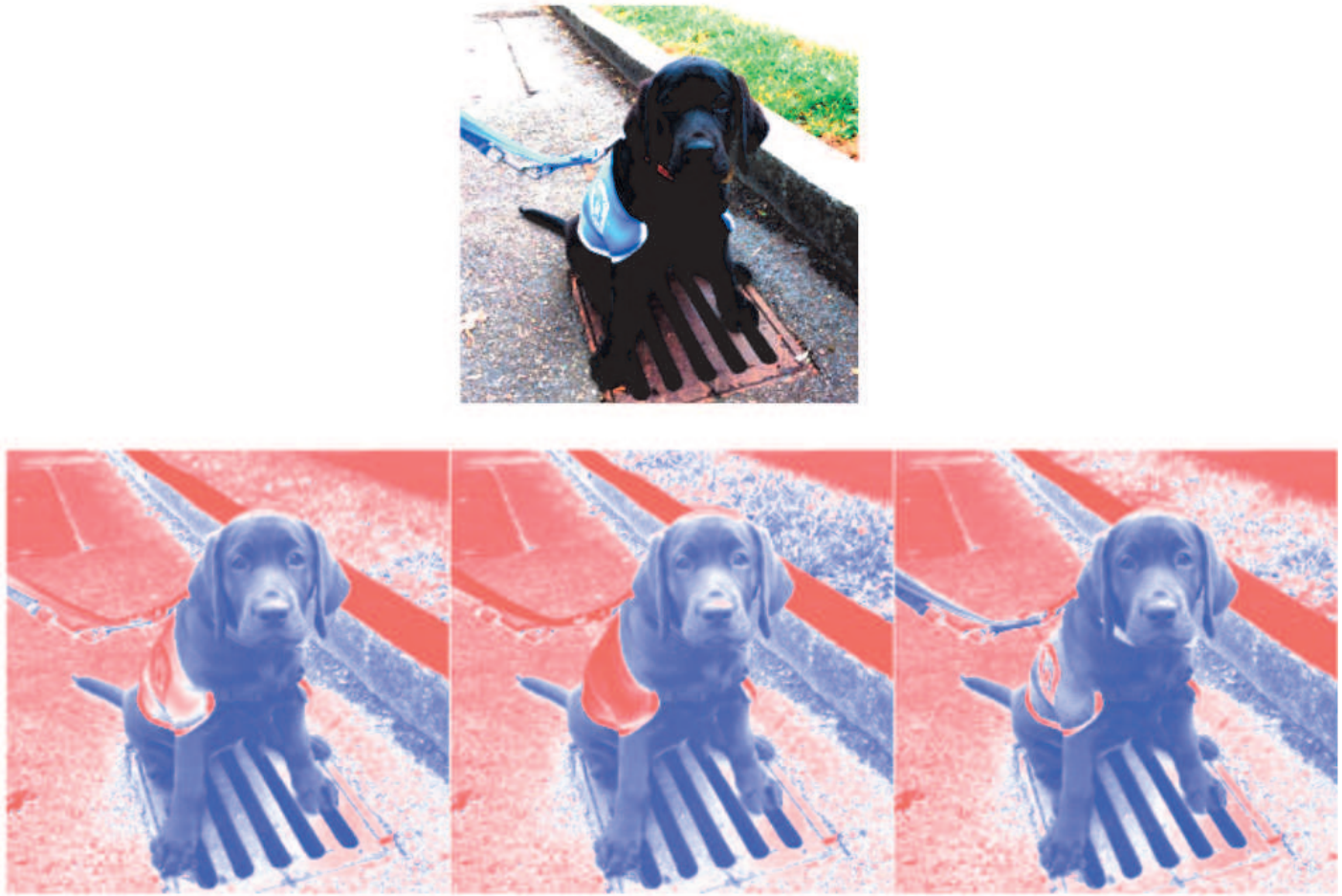


Figure credit: F. Fleuret

Visualize layer activations/feature maps

ResNet152

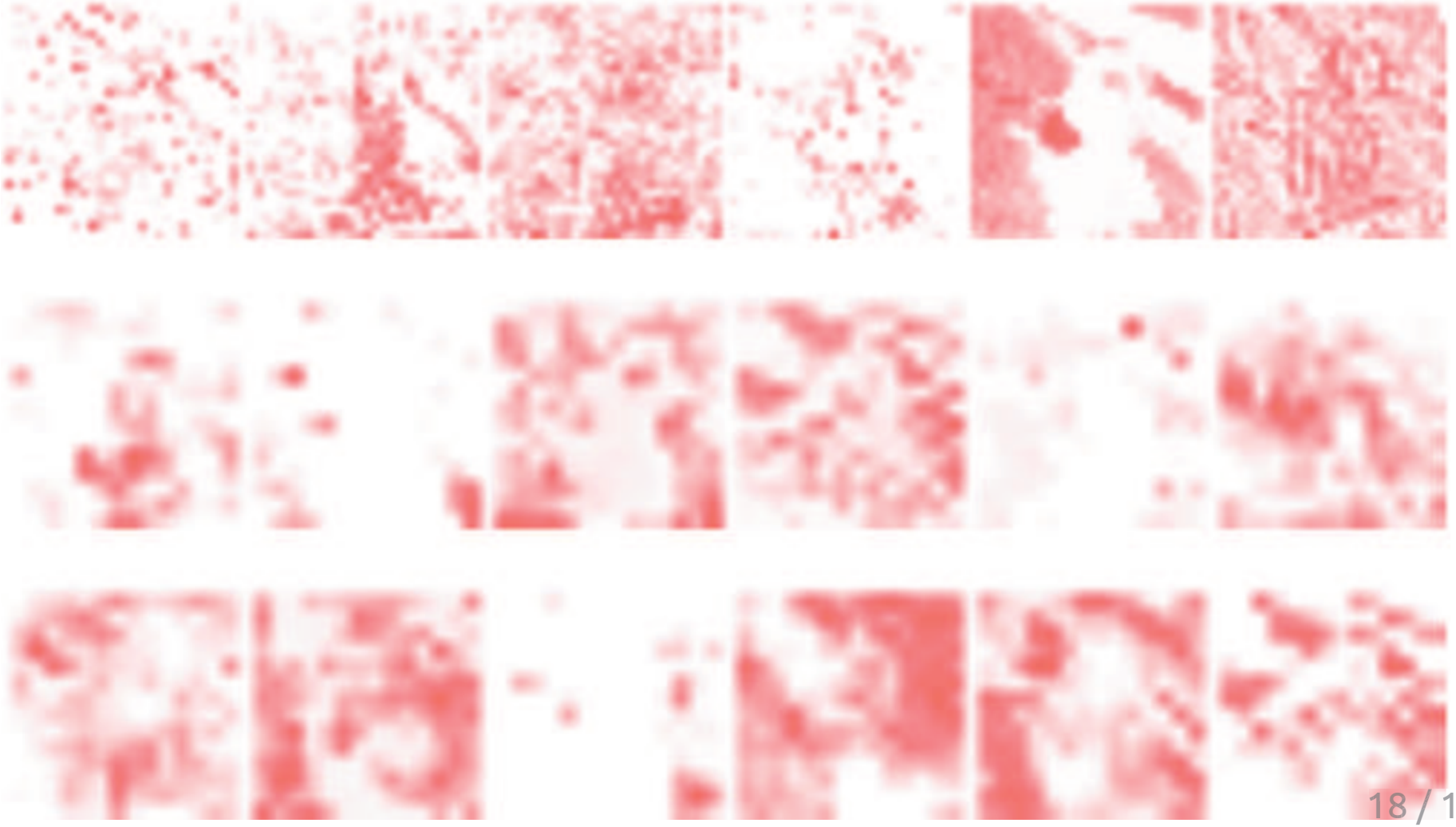


Figure credit: F. Fleuret

Visualize layer activations/feature maps

ResNet152

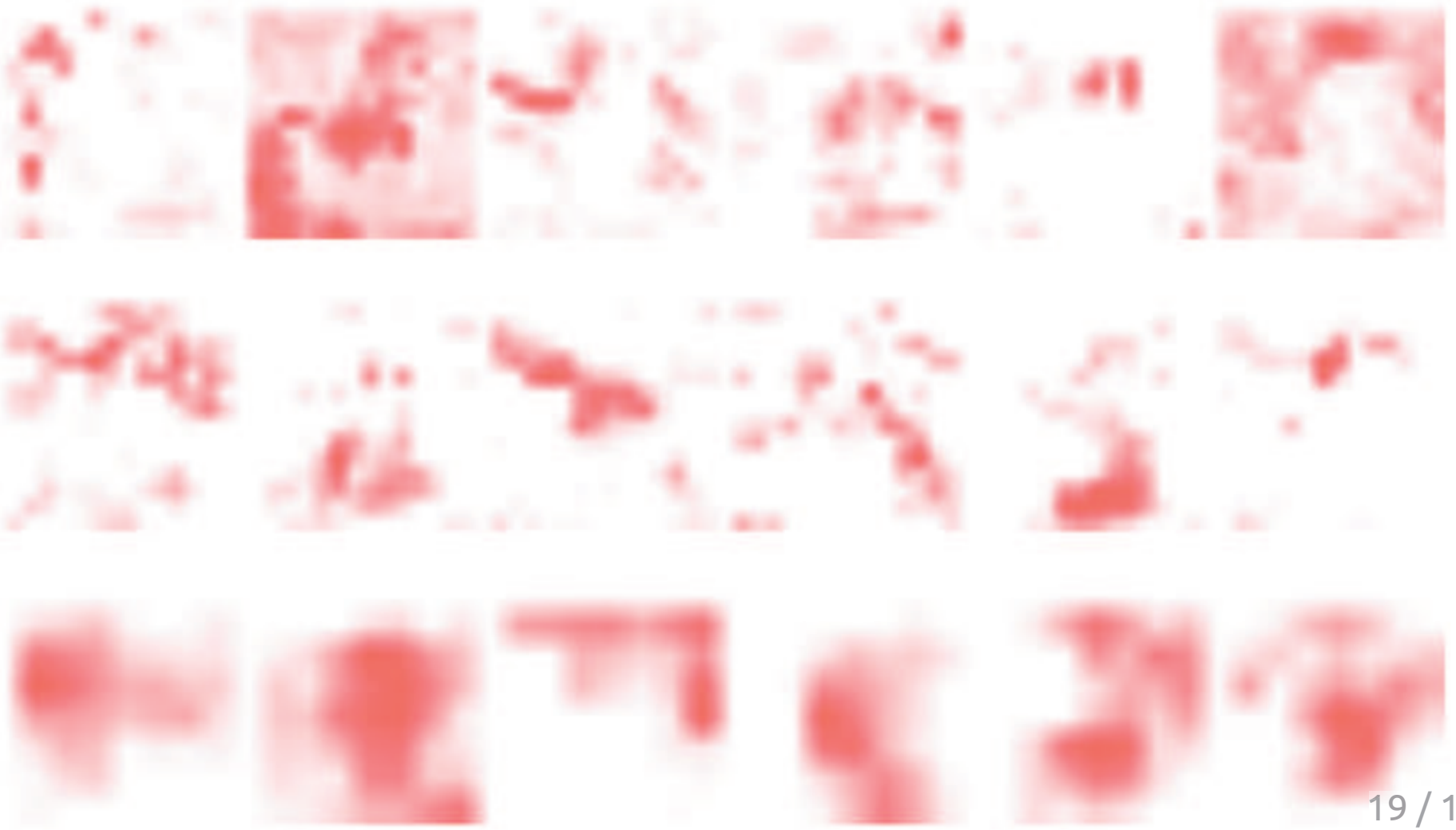
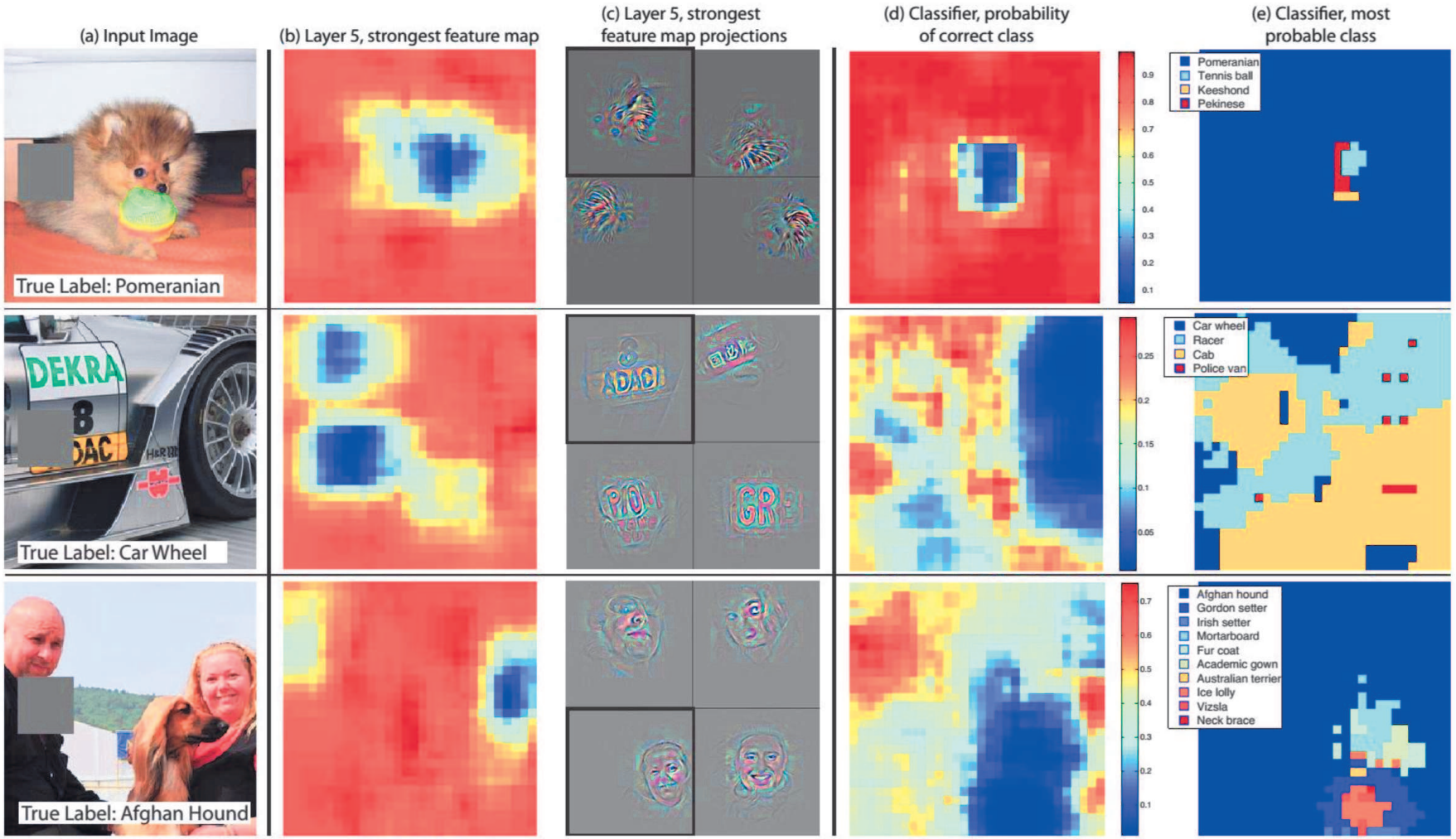


Figure credit: F. Fleuret

Occlusion sensitivity



Visualizing and Understanding Convolutional Networks, M. Zeiler & R. Fergus, ECCV 2014

Occlusion sensitivity

An approach to understand the behavior of a network is to look at the output of the network "around" an image.

We can get a simple estimate of the importance of a part of the input image by computing the difference between:

1. the value of the maximally responding output unit on the image, and
2. the value of the same unit with that part occluded.

Occlusion sensitivity

An approach to understand the behavior of a network is to look at the output of the network "around" an image.

We can get a simple estimate of the importance of a part of the input image by computing the difference between:

1. the value of the maximally responding output unit on the image, and
2. the value of the same unit with that part occluded.

This is computationally intensive since it requires as many forward passes as there are locations of the occlusion mask, ideally the number of pixels.

Occlusion sensitivity

Original images

Occlusion mask 32×32

Figure credit: F. Fleuret

Occlusion sensitivity

Original images

Occlusion sensitivity, mask 32×32, stride of 2, Alexnet

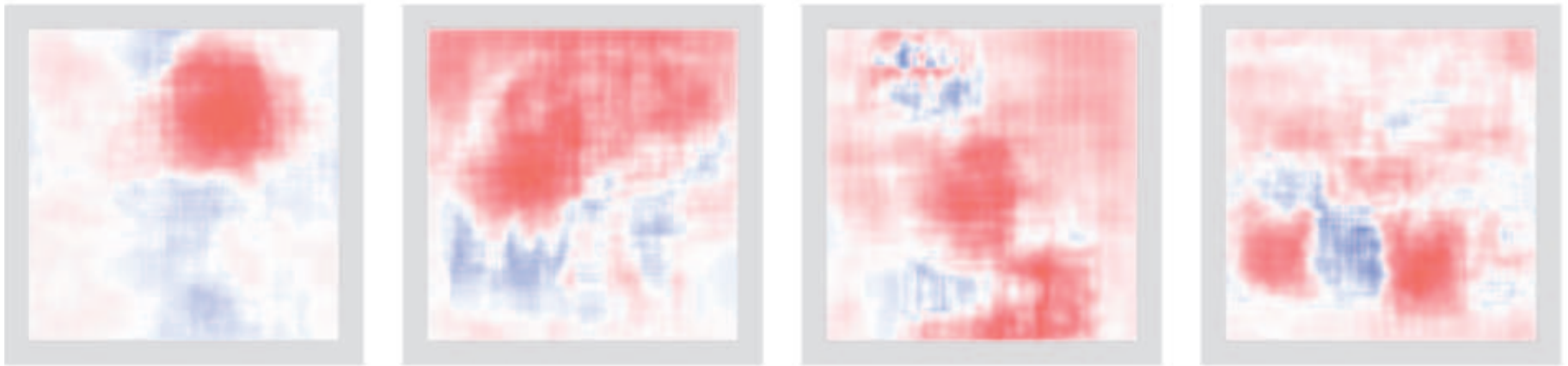


Figure credit: F. Fleuret

Occlusion sensitivity

Original images

Occlusion sensitivity, mask 32 × 32, stride of 2, VGG16

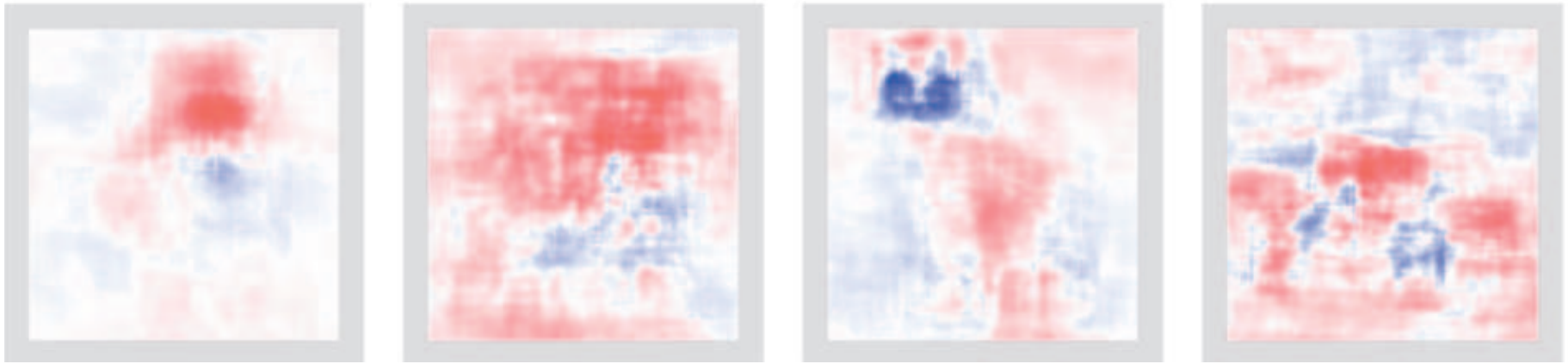


Figure credit: F. Fleuret

Occlusion sensitivity

Original images

Occlusion sensitivity, mask 32 × 32, stride of 2, VGG19

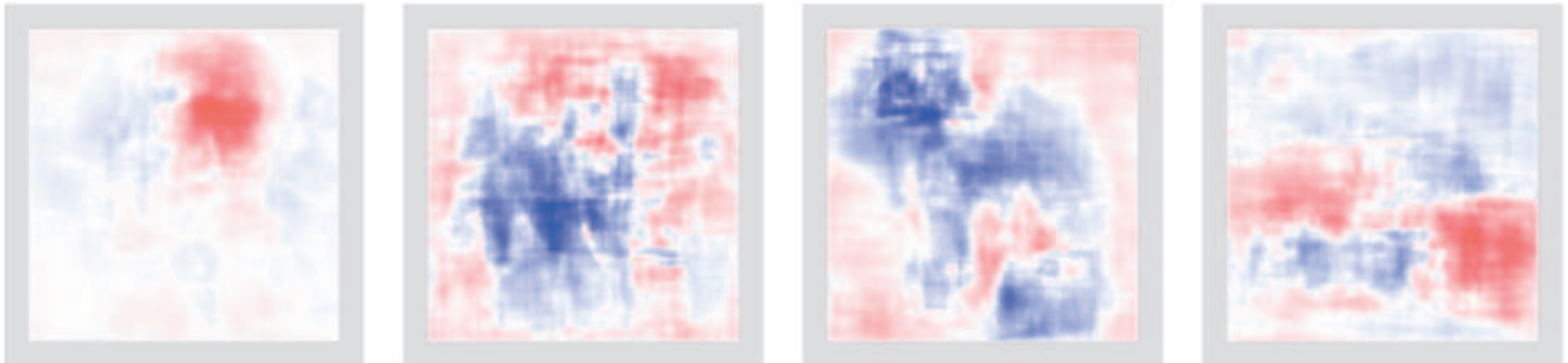
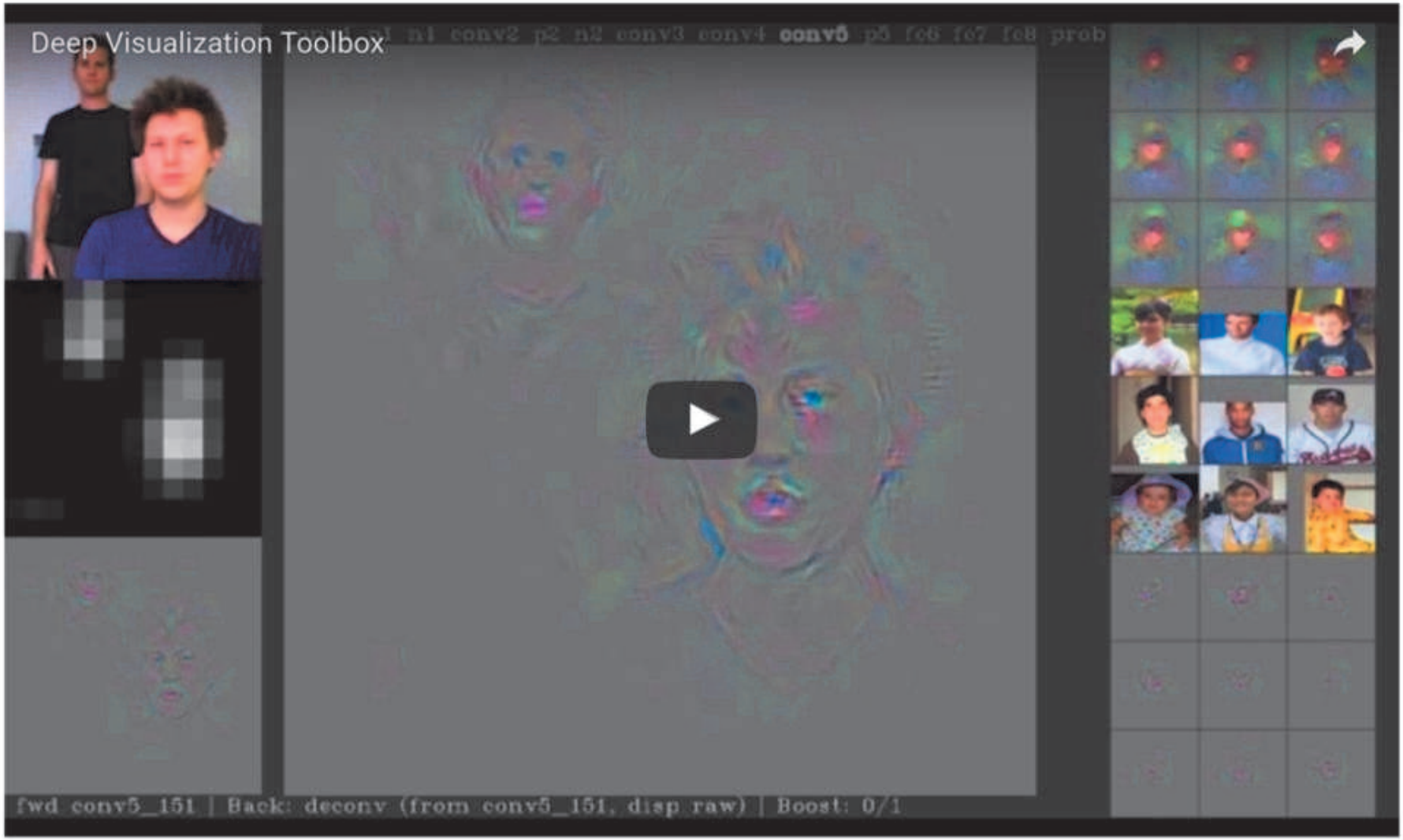


Figure credit: F. Fleuret

Visualize arbitrary neurons

DeepVis toolbox https://www.youtube.com/watch?v=AgkfIQ4IGaM



Maximum response samples

What does a convolutional network see?

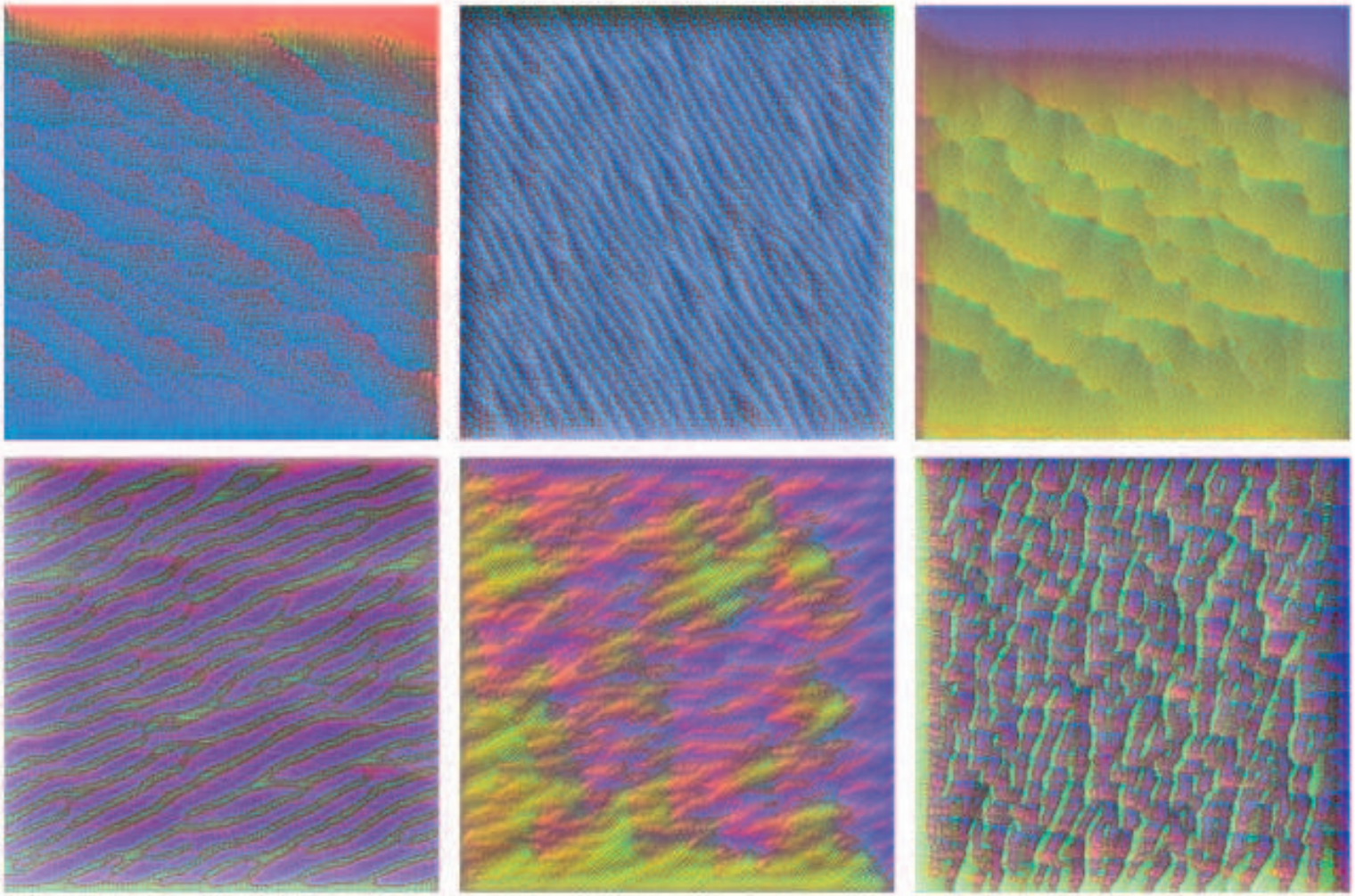
Convolutional networks can be inspected by looking for input images $\mathbf{x}$ that maximize the activation $\mathbf{h}_{\ell,d}(\mathbf{x})$ of a chosen convolutional kernel $\mathbf{u}$ at layer ℓ and index d in the layer filter bank.

Such images can be found by gradient ascent on the input space:

$$\begin{aligned} \mathcal{L}_{\ell,d}(\mathbf{x}) &= ||\mathbf{h}_{\ell,d}(\mathbf{x})||_2 \\ \mathbf{x}_0 &\sim U[0,1]^{C\times H\times W} \\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \gamma \nabla_{\mathbf{x}} \mathcal{L}_{\ell,d}(\mathbf{x}_t) \end{aligned}$$

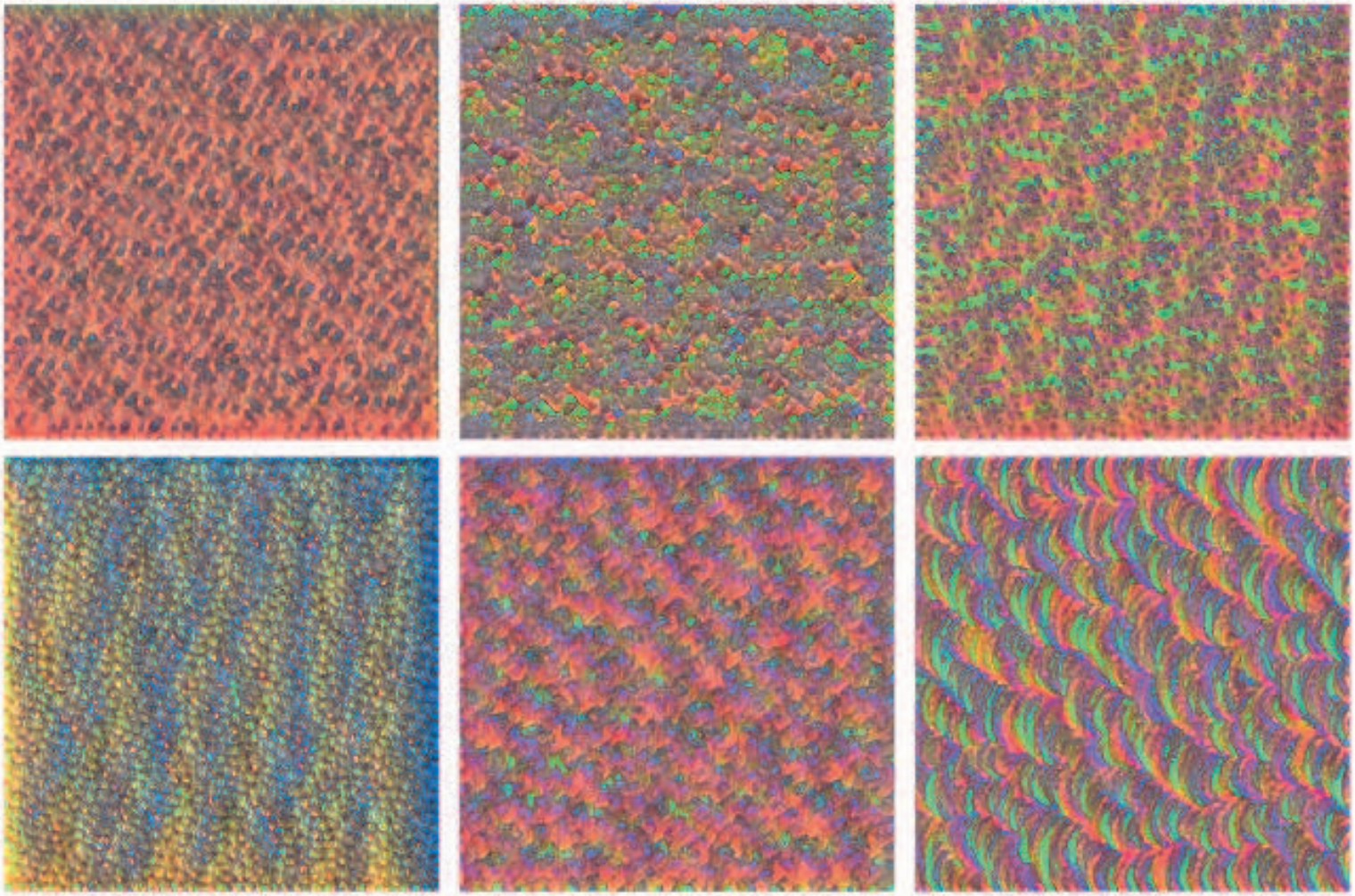
Maximum response samples

VGG16, maximizing a channel of the 4th convolution layer



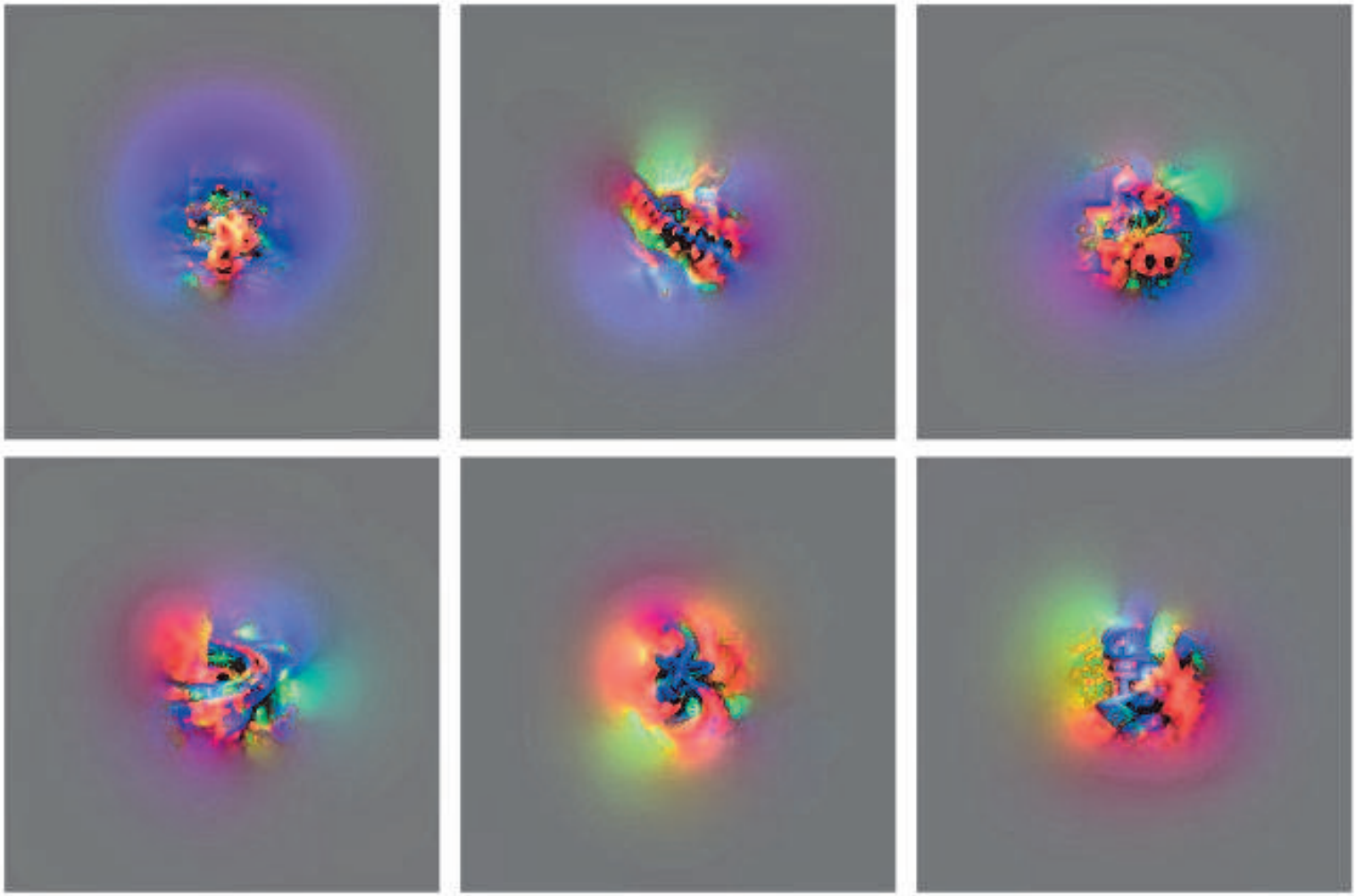
Maximum response samples

VGG16, maximizing a channel of the 7th convolution layer



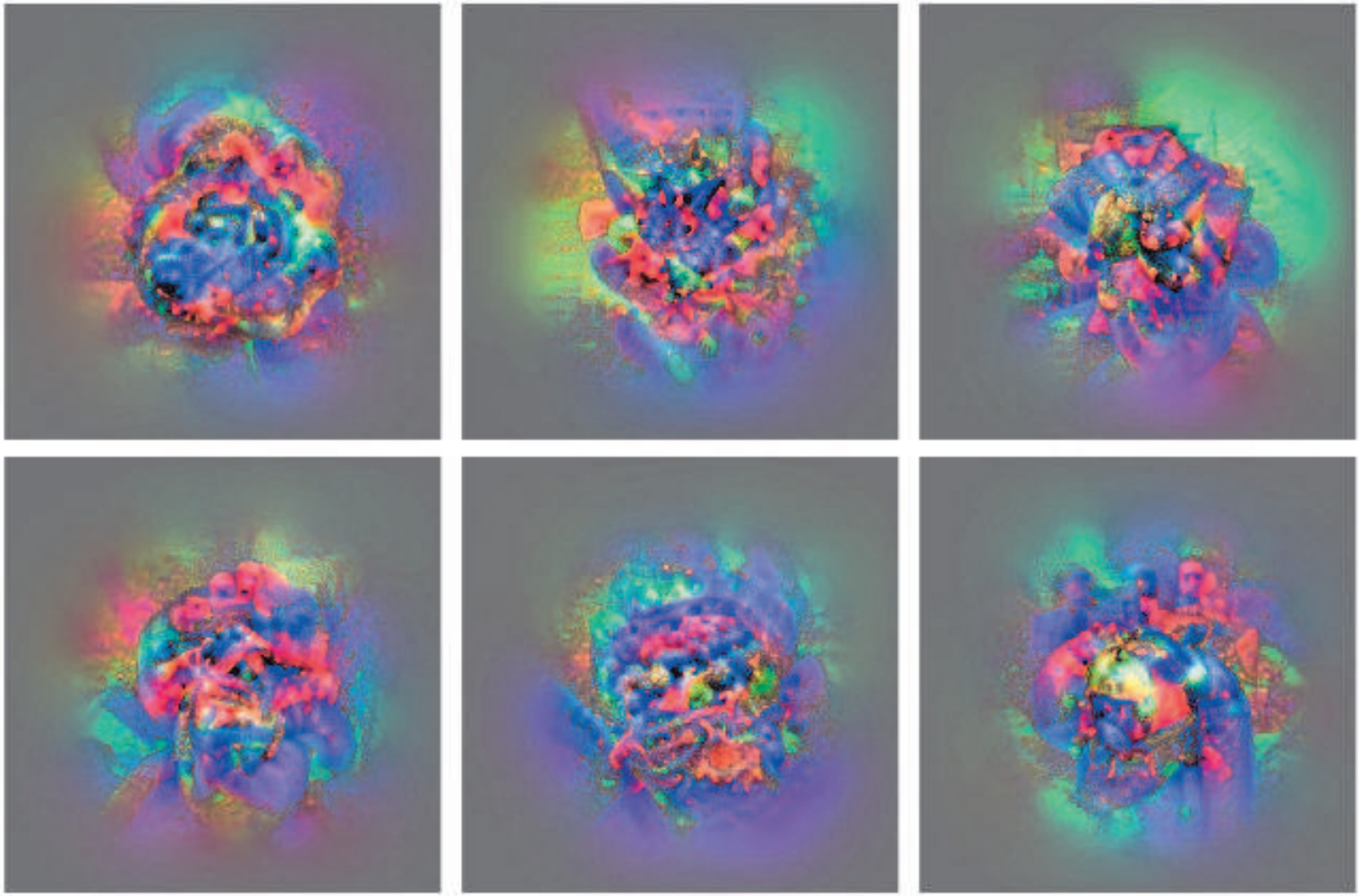
Maximum response samples

VGG16, maximizing a unit of the 10th convolution layer



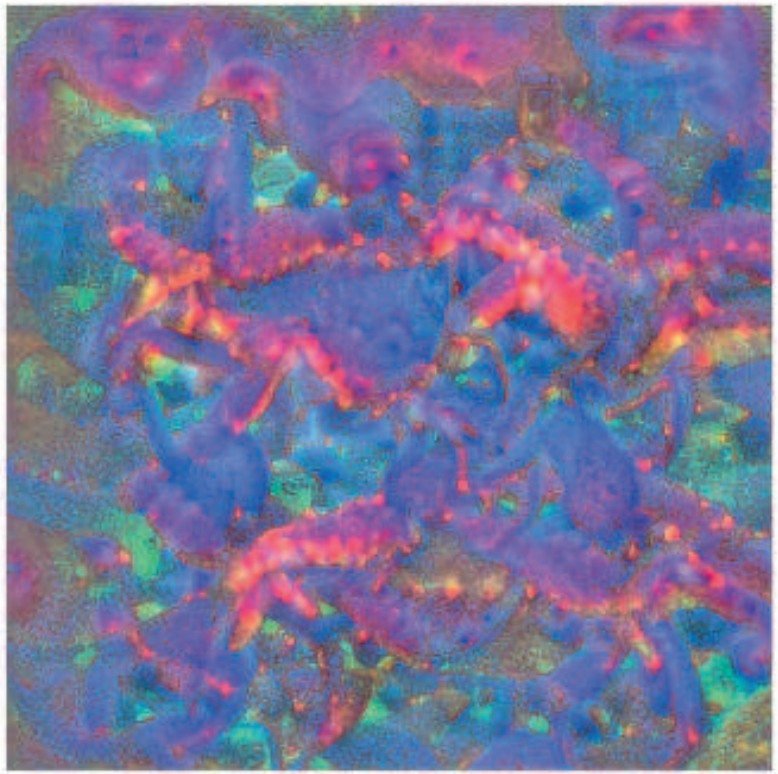
Maximum response samples

VGG16, maximizing a unit of the 13th (and last) convolution layer

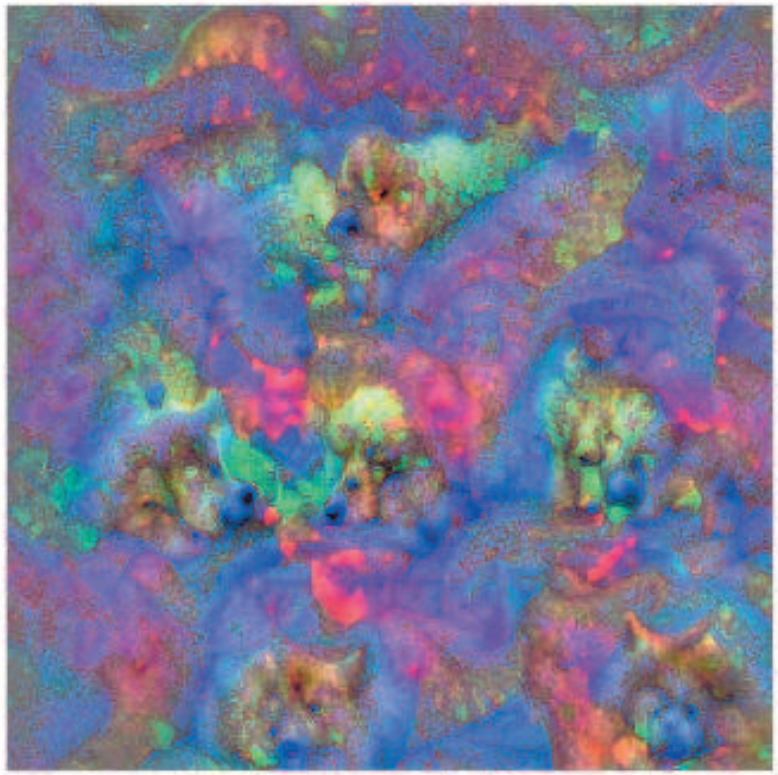


Maximum response samples

VGG16, maximizing a unit of the output layer



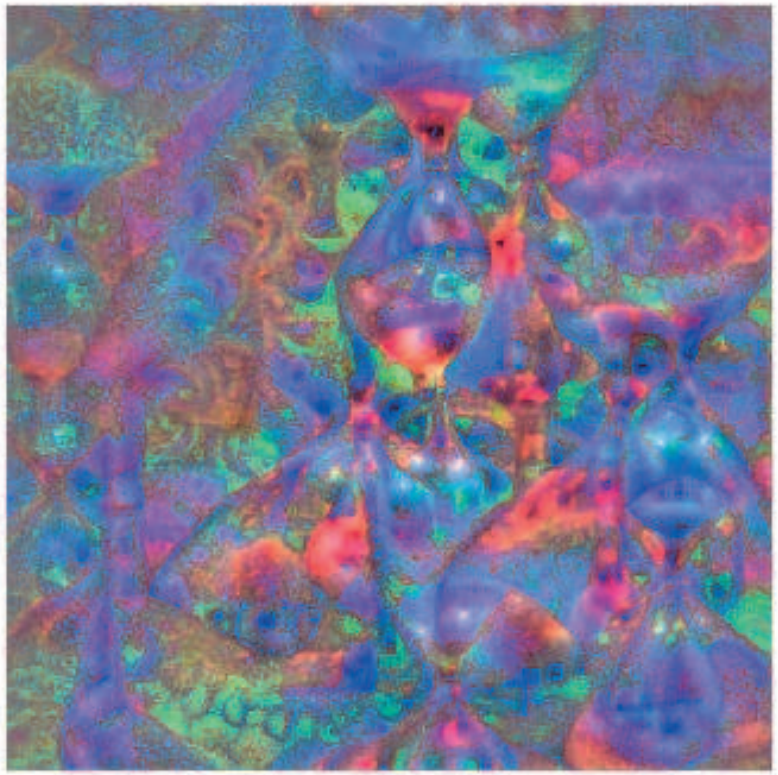
"King crab"



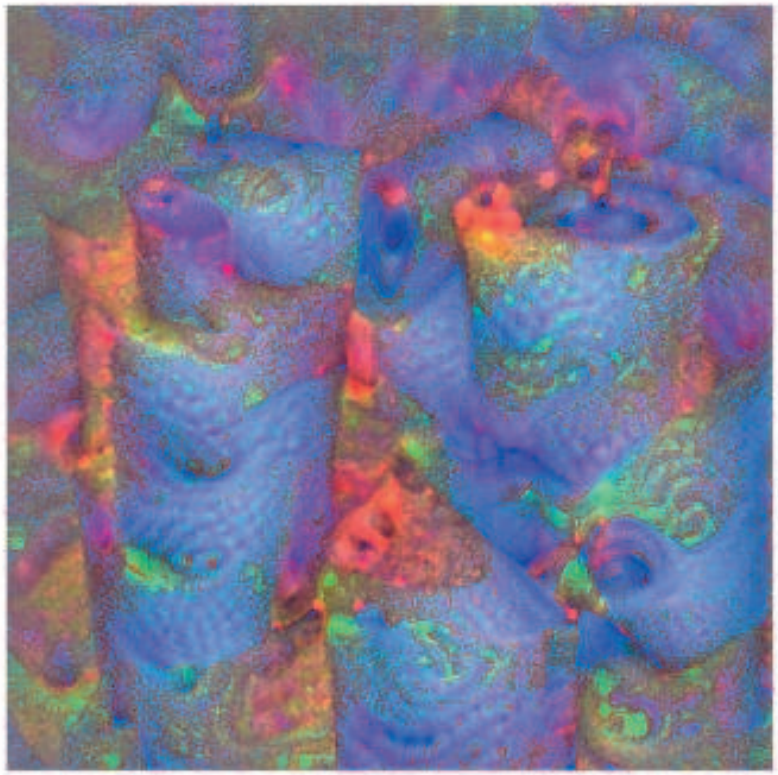
"Samoyed" (that's a fluffy dog)

Maximum response samples

VGG16, maximizing a unit of the output layer



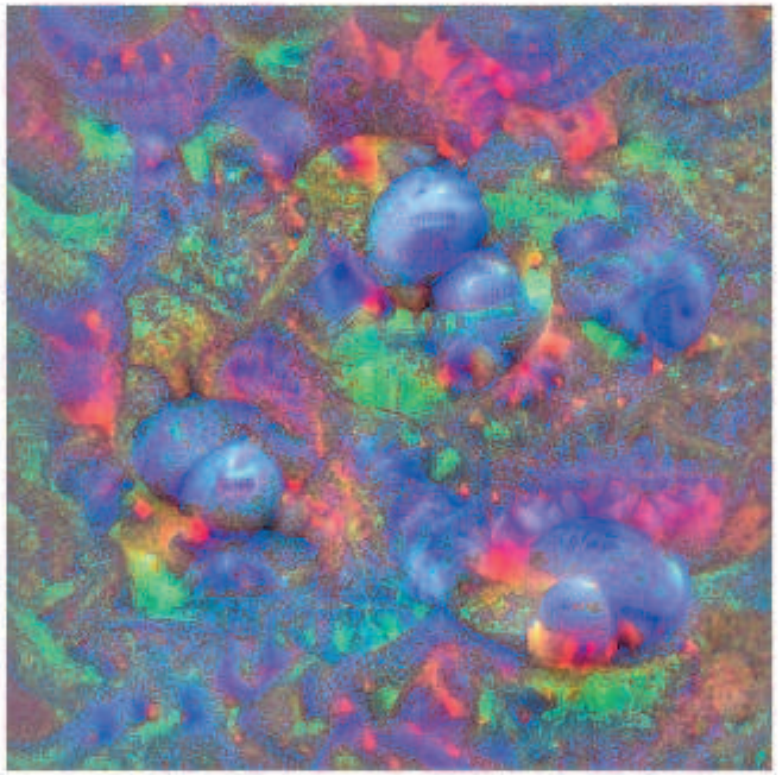
"Hourglass"



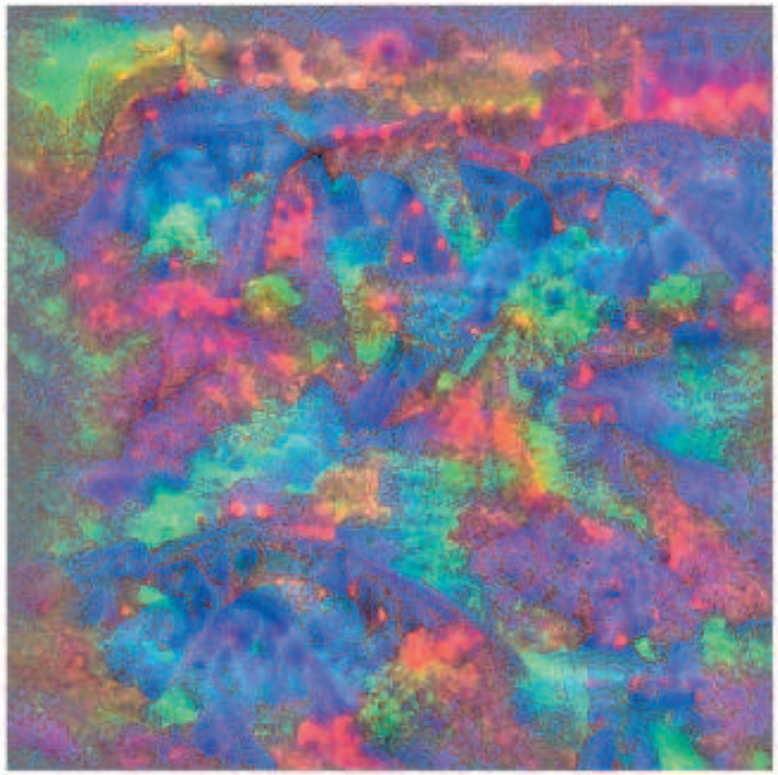
"Paper towel"

Maximum response samples

VGG16, maximizing a unit of the output layer



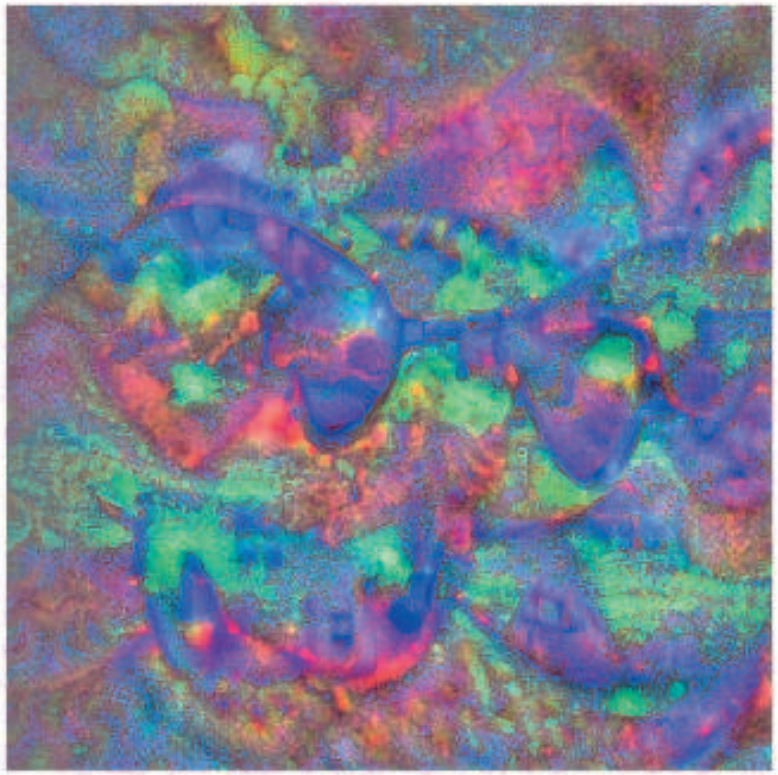
"Ping-pong ball"



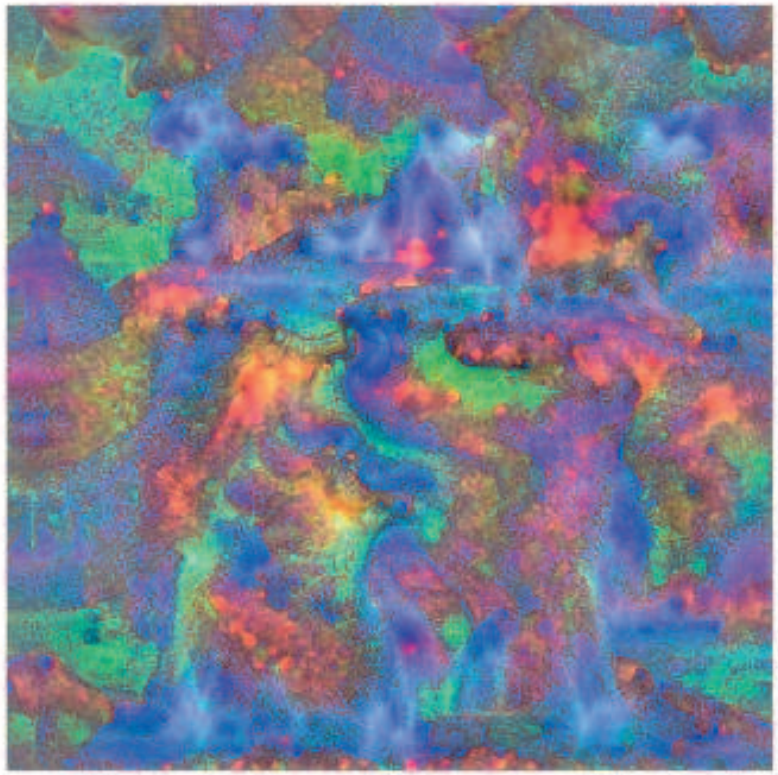
"Steel arch bridge"

Maximum response samples

VGG16, maximizing a unit of the output layer

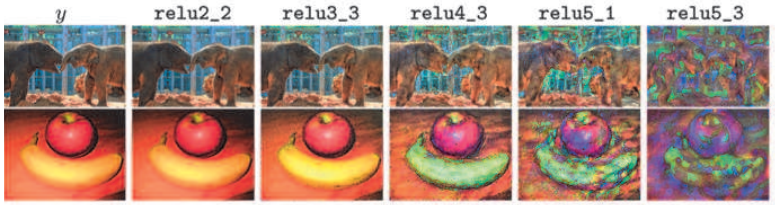


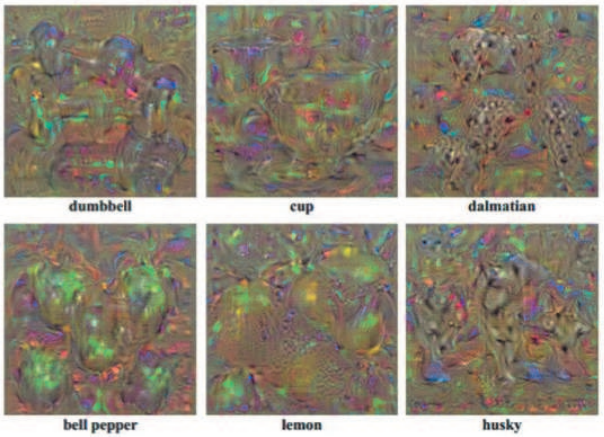
"Sunglass"

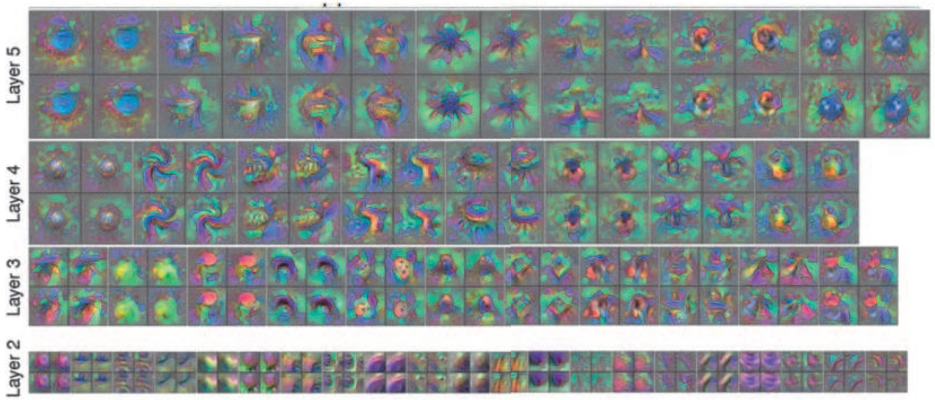


"Geyser"

Many more visualization techniques

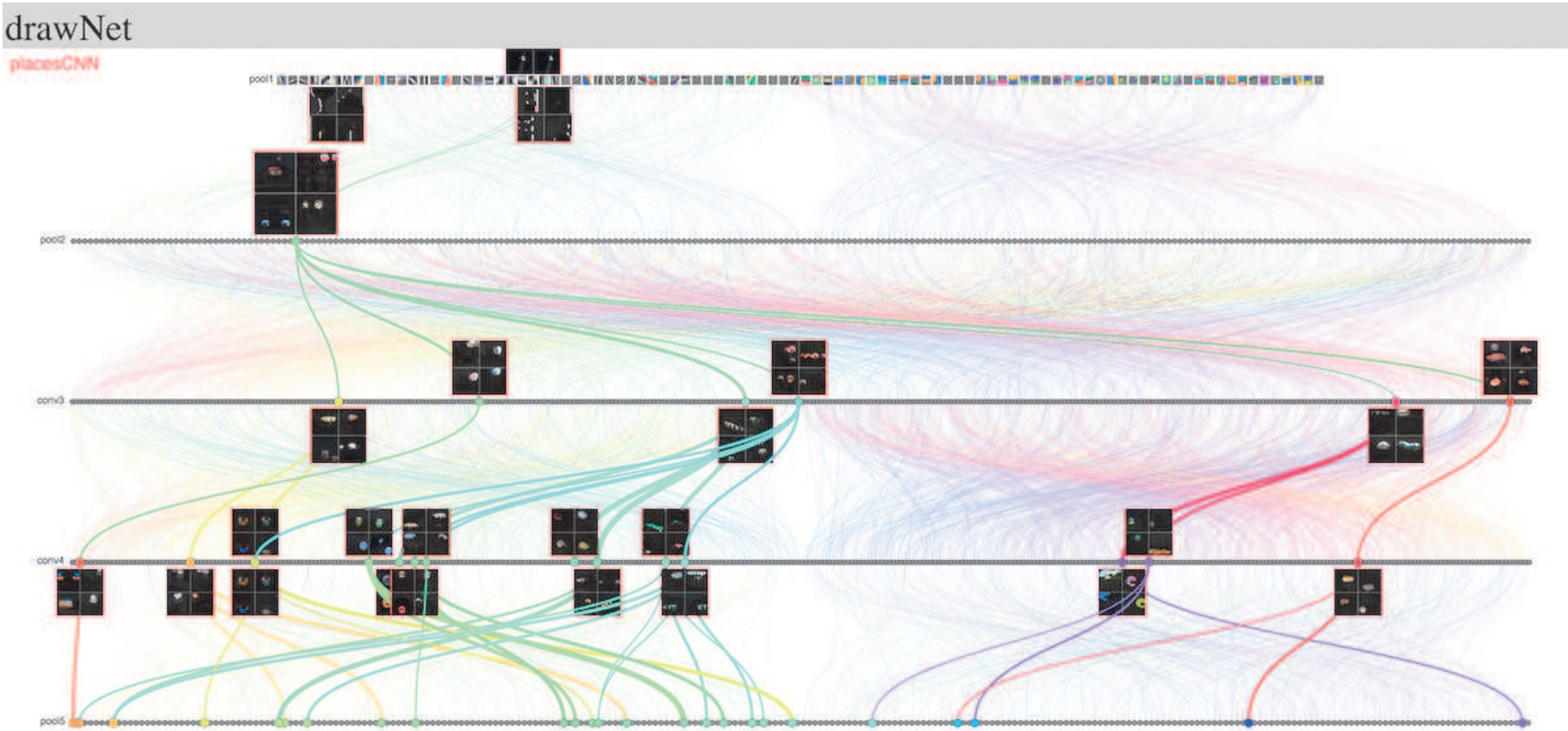






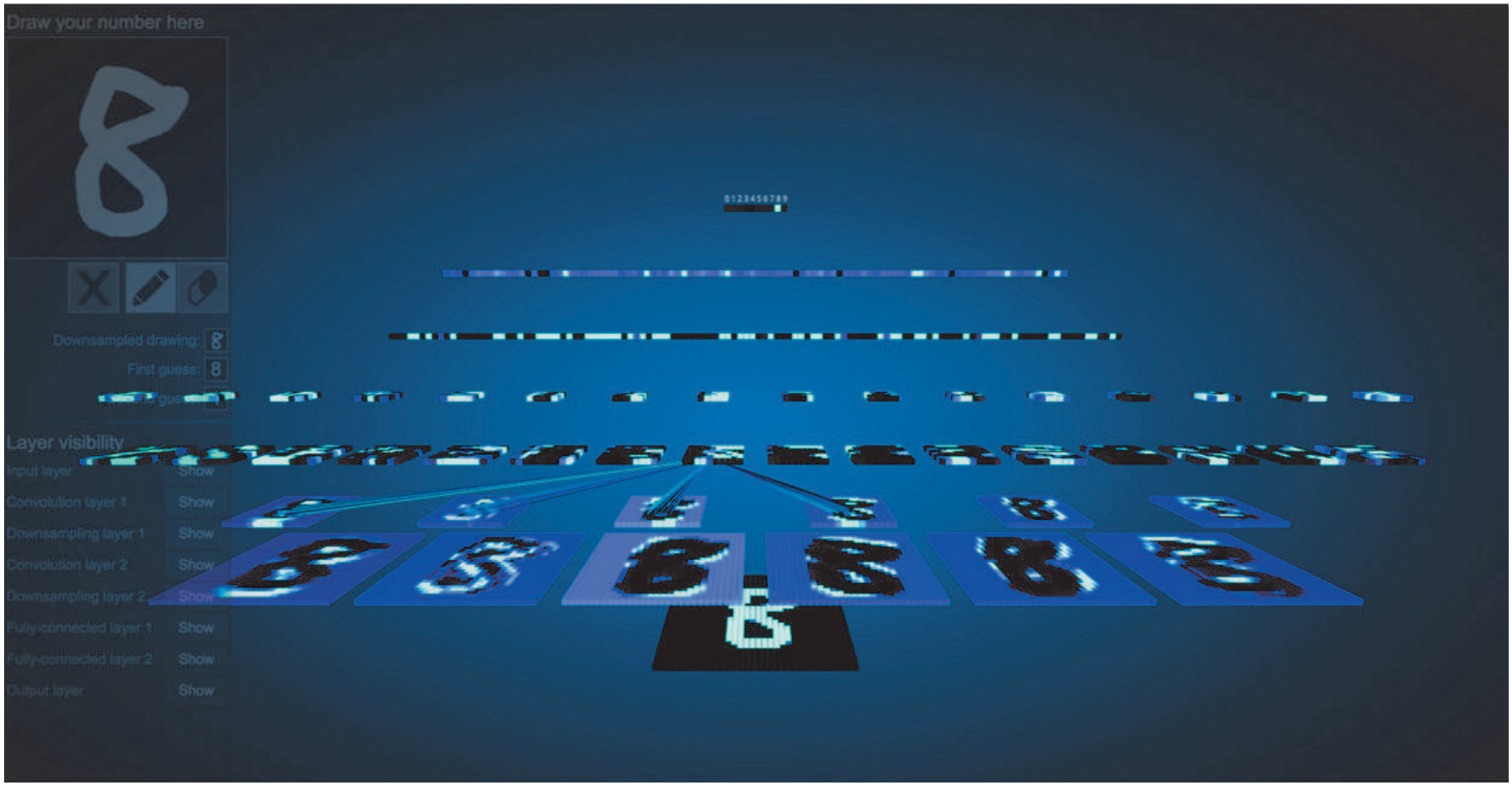
Other resources

DrawNet
http://people.csail.mit.edu/torralba/research/drawCNN/drawNet.html



Other resources

Basic CNNs http://scs.ryerson.ca/~aharley/vis/

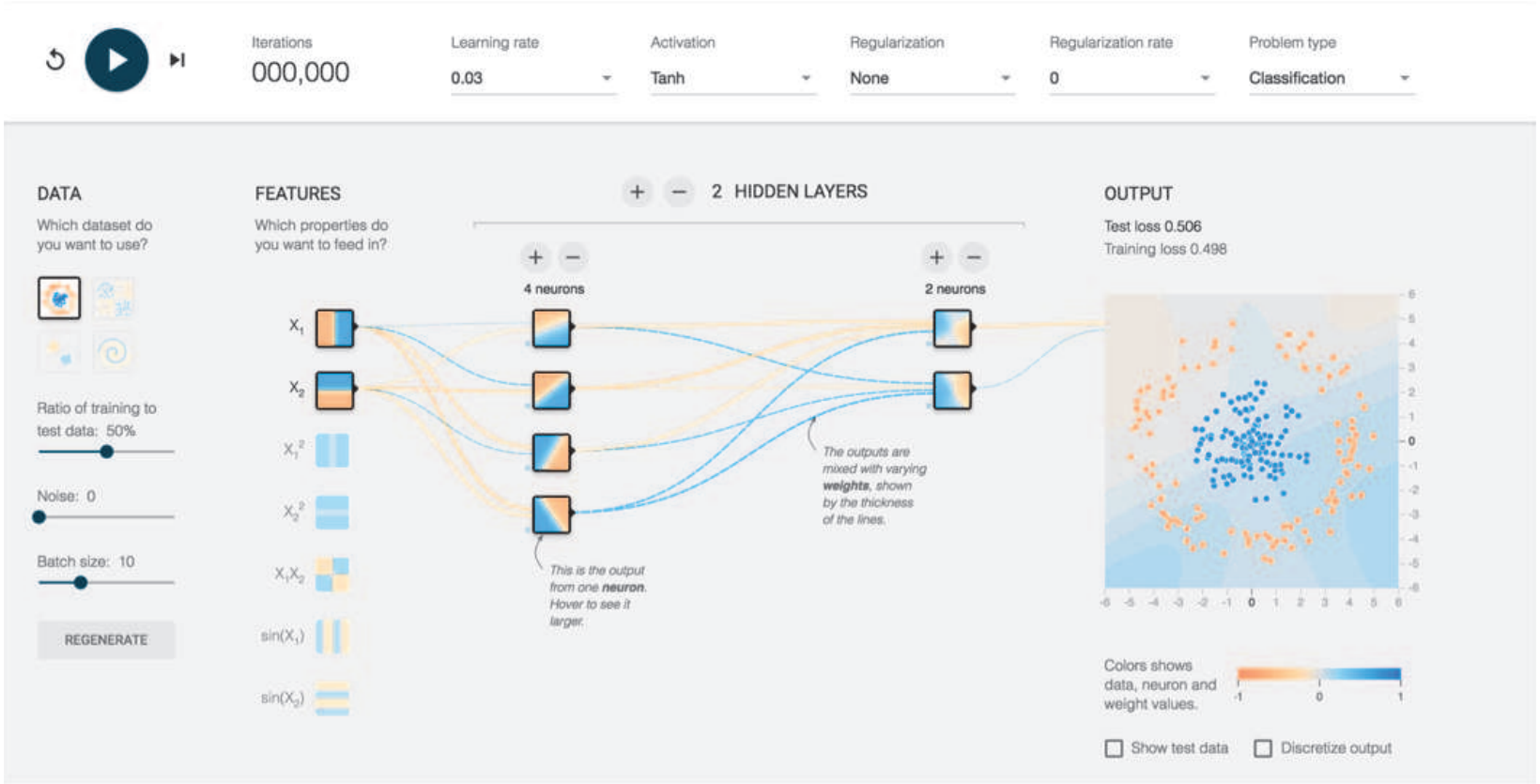


Other resources

Keras-JS https://transcranial.github.io/keras-js/

Other resources

TensorFlow playground http://playground.tensorflow.org



Adversarial attacks

Locality assumption

"The deep stack of non-linear layers are a way for the model to encode a non-local generalization prior over the input space. In other words, it is assumed that is possible for the output unit to assign probabilities to regions of the input space that contain no training examples in their vicinity.

It is implicit in such arguments that local generalization---in the very proximity of the training examples---works as expected. And that in particular, for a small enough radius $\epsilon > 0$ in the vicinity of a given training input $\mathbf{x}$, an $\mathbf{x} + \mathbf{r}$ satisfying $||\mathbf{r}|| < \epsilon$ will get assigned a high probability of the correct class by the model."

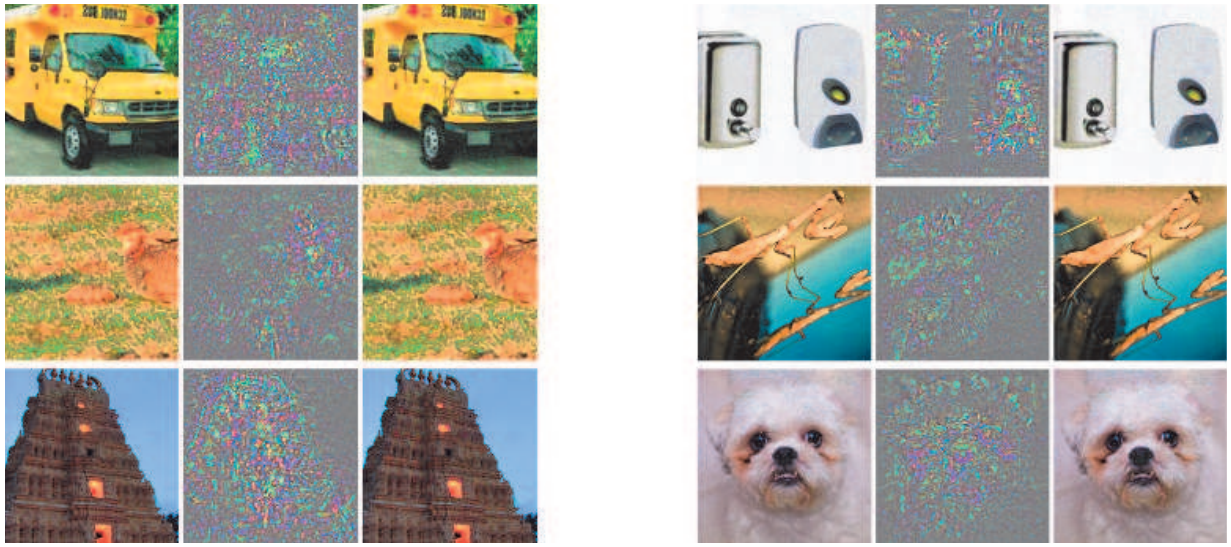
(Szegedy et al, 2013)

Adversarial examples

$$\begin{aligned} &\min ||\mathbf{r}||_2 \\ \text{s.t. } &f(\mathbf{x}+\mathbf{r}) = y' \\ &\mathbf{x}+\mathbf{r} \in [0,1]^p \end{aligned}$$

where

- y' is some target label, different from the original label y associated to $\mathbf{x}$,
- f is a trained neural network.



(Left) Original images $\mathbf{x}$. (Middle) Noise $\mathbf{r}$. (Right) Modified images $\mathbf{x} + \mathbf{r}$.
All are classified as 'Ostrich'. (Szegedy et al, 2013)

Even simpler, take a step along the direction of the sign of the gradient at each pixel:

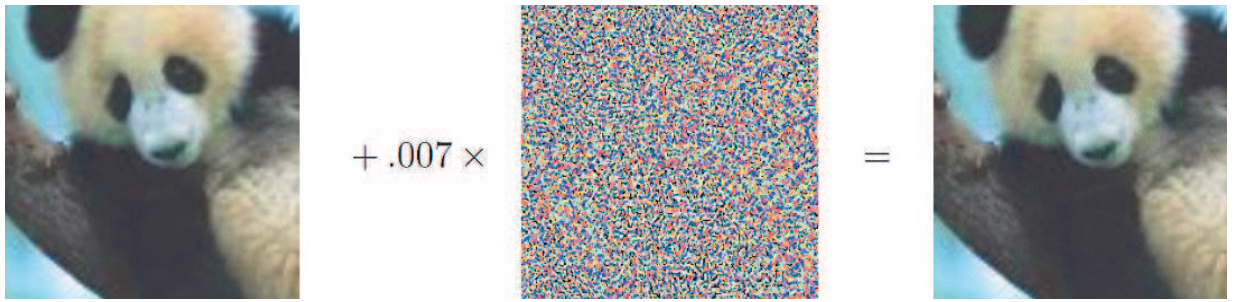
$$\mathbf{r} = \epsilon \, \text{sign}(\nabla_{\mathbf{x}} \ell(y', f(\mathbf{x})))$$

where ϵ is the magnitude of the perturbation.

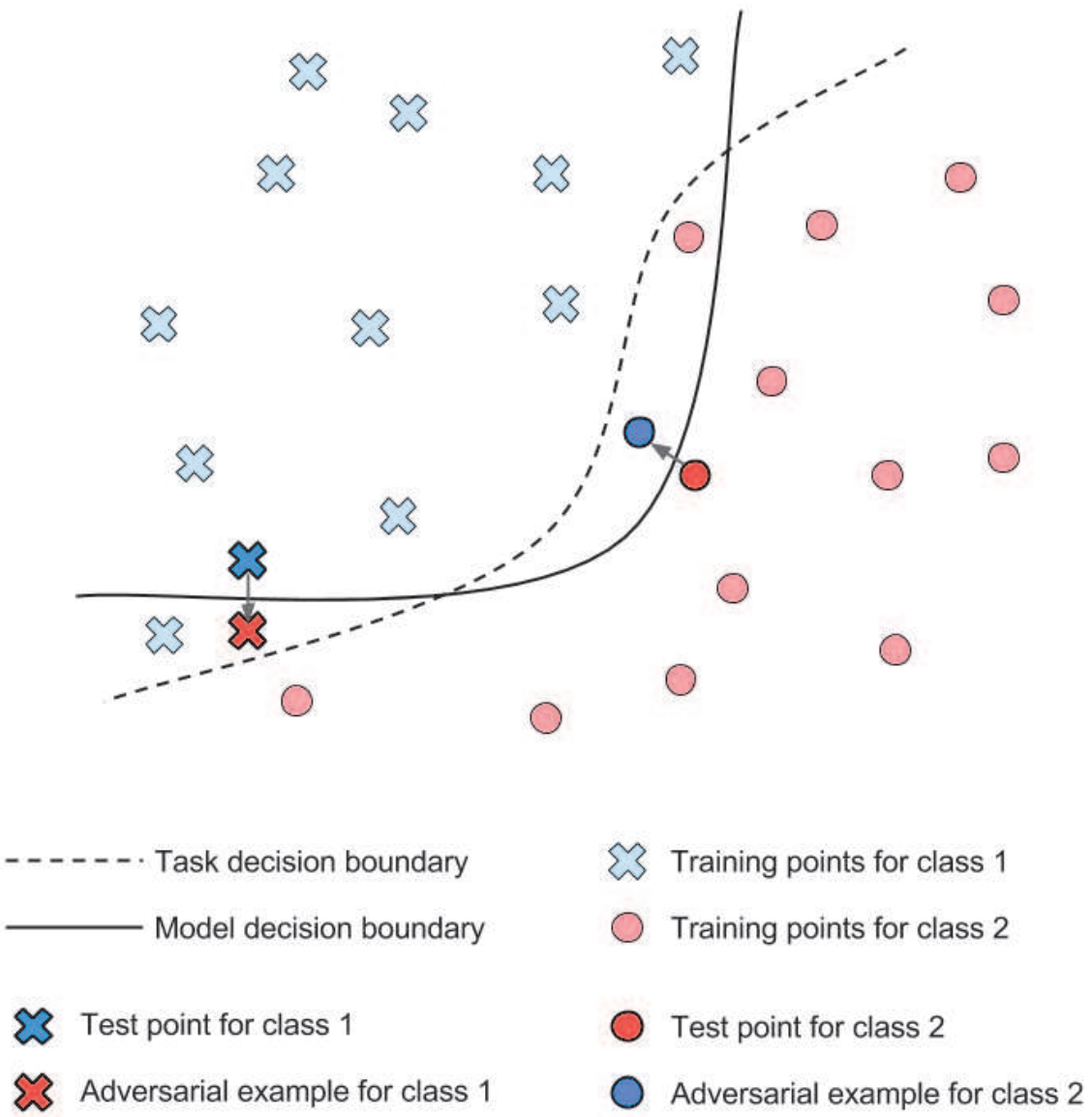
Even simpler, take a step along the direction of the sign of the gradient at each pixel:

$$\mathbf{r} = \epsilon \, \text{sign}(\nabla_{\mathbf{x}} \ell(y', f(\mathbf{x})))$$

where ϵ is the magnitude of the perturbation.



The panda on the right is classified as a 'Gibbon'. (Goodfellow et al, 2014)



Credits: Breaking things easy (Papernot and Goodfellow, 2016)

Not just for neural networks

Many other machine learning models are subject to adversarial examples, including:

- Linear models
  - Logistic regression
  - Softmax regression
  - Support vector machines
- Decision trees
- Nearest neighbors

Fooling neural networks

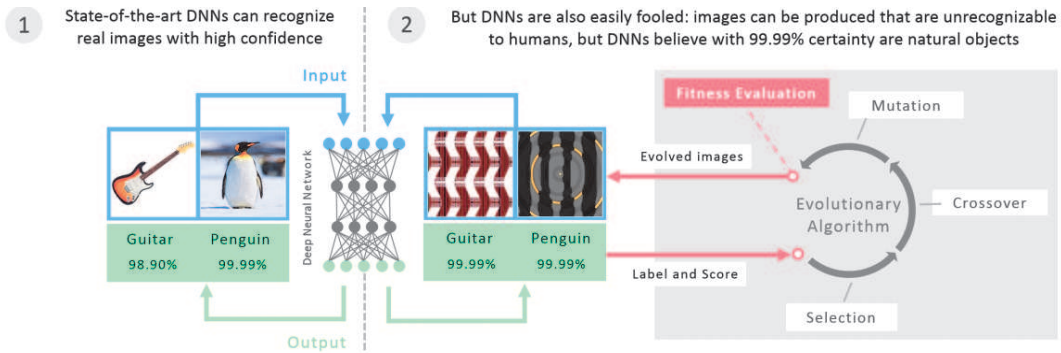


Figure 2. Although state-of-the-art deep neural networks can increasingly recognize natural images (*left panel*), they also are easily fooled into declaring with near-certainty that unrecognizable images are familiar objects (*center*). Images that fool DNNs are produced by evolutionary algorithms (*right panel*) that optimize images to generate high-confidence DNN predictions for each class in the dataset the DNN is trained on (here, ImageNet).

(Nguyen et al, 2014)

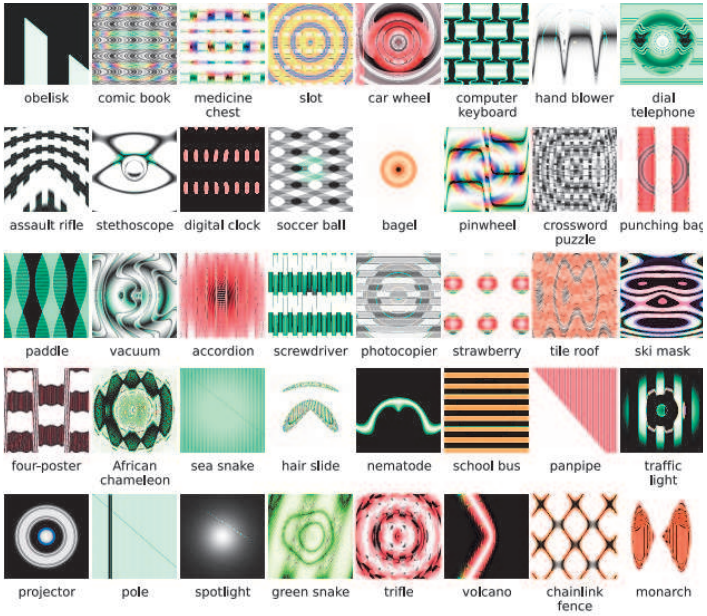
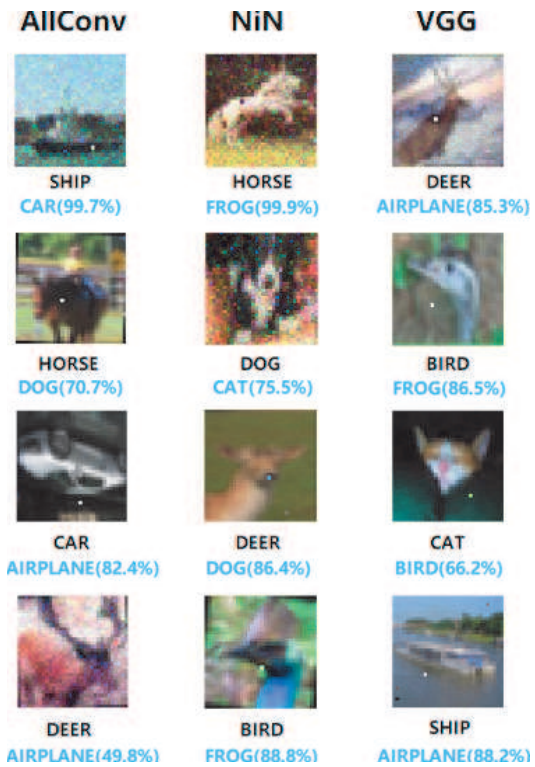


Figure 8. Evolving images to match DNN classes produces a tremendous diversity of images. Shown are images selected to showcase diversity from 5 evolutionary runs. The diversity suggests that the images are non-random, but that instead evolutions producing discriminative features of each target class. The mean DNN confidence scores for these images is 99.12%.

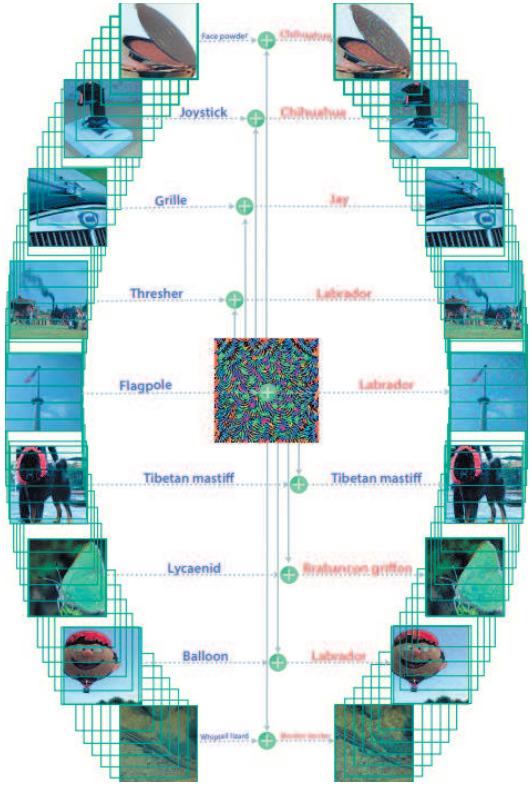
(Nguyen et al, 2014)

One pixel attacks



(Su et al, 2017)

Universal adversarial perturbations



(Moosavi-Dezfooli et al, 2016)

Fooling deep structured prediction models

Figure 1: We cause the network to generate a *minion* as segmentation for the adversarially perturbed version of the original image. Note that the original and the perturbed image are indistinguishable.

(Cisse et al, 2017)

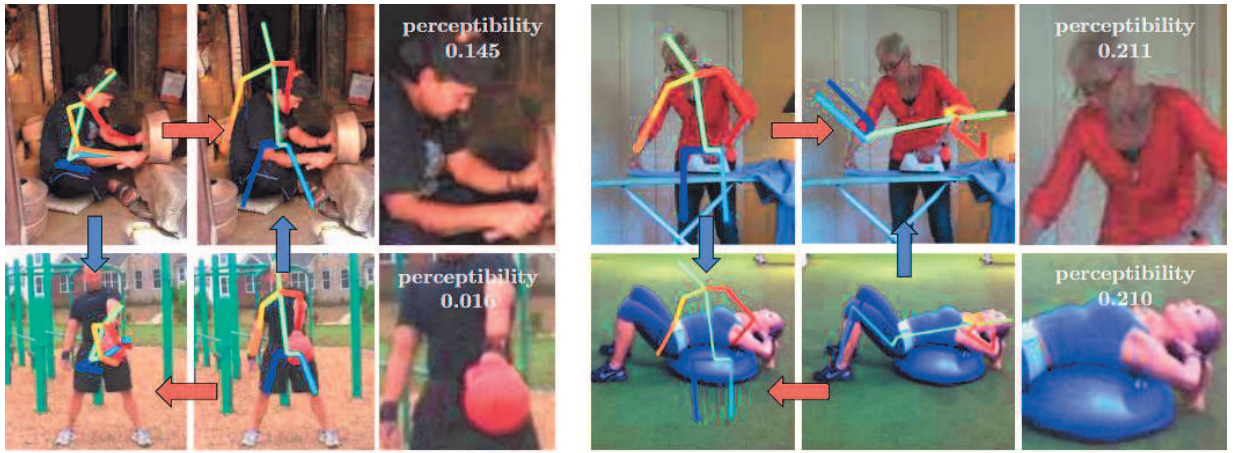
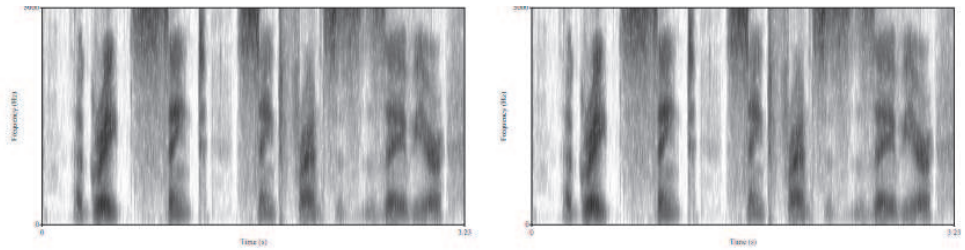


Figure 4: Examples of successful targetted attacks on a pose estimation system. Despite the important difference between the images selected, it is possible to make the network predict the wrong pose by adding an imperceptible perturbation.

(Cisse et al, 2017)



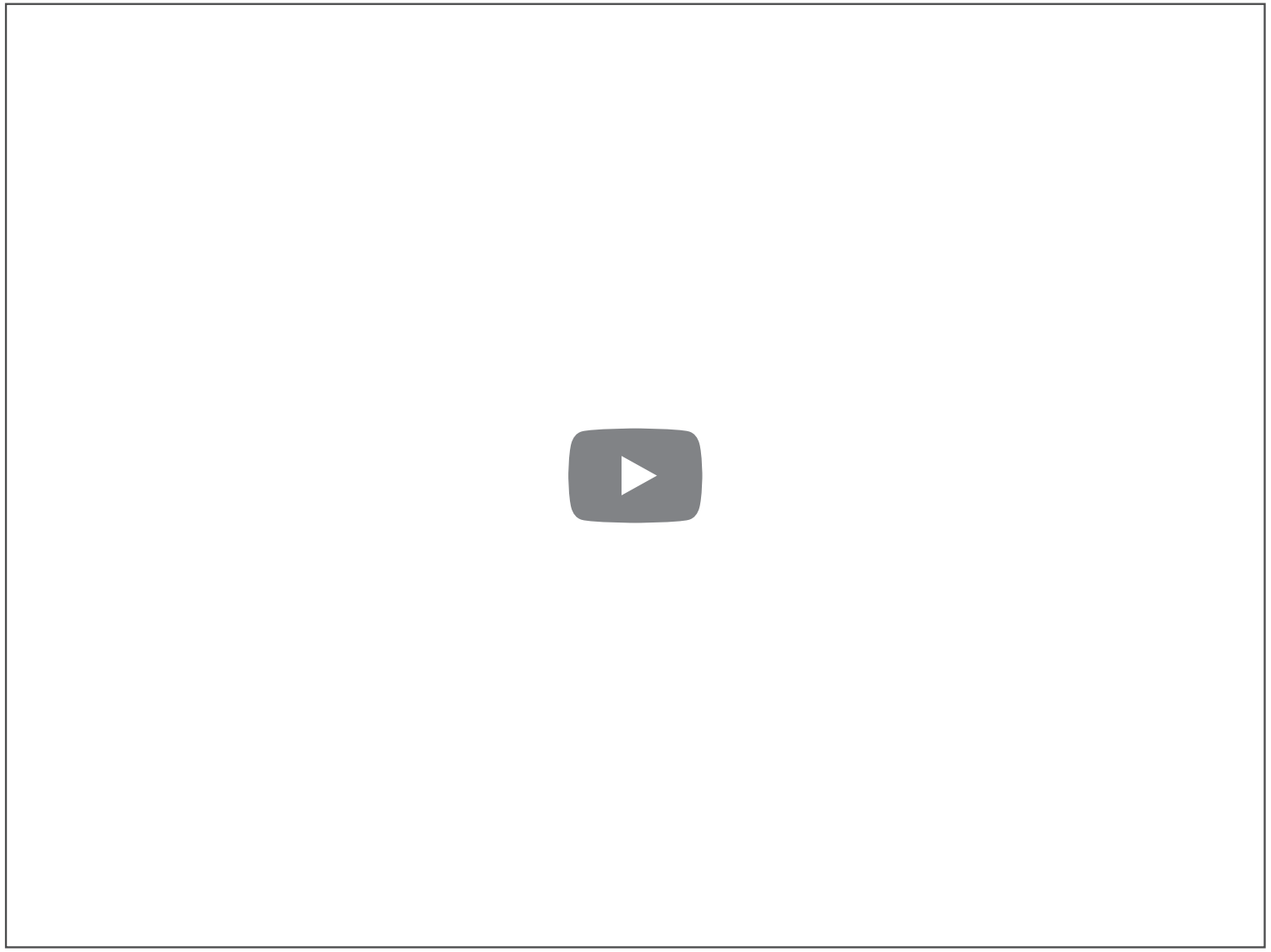
(a) a great saint saint francis zaviour (b) i great sinkt shink t frimsuss avir

Figure 7: The model models' output for each of the spectrograms is located at the bottom of each spectrogram. The target transcription is: `A Great Saint Saint Francis Xavier.`

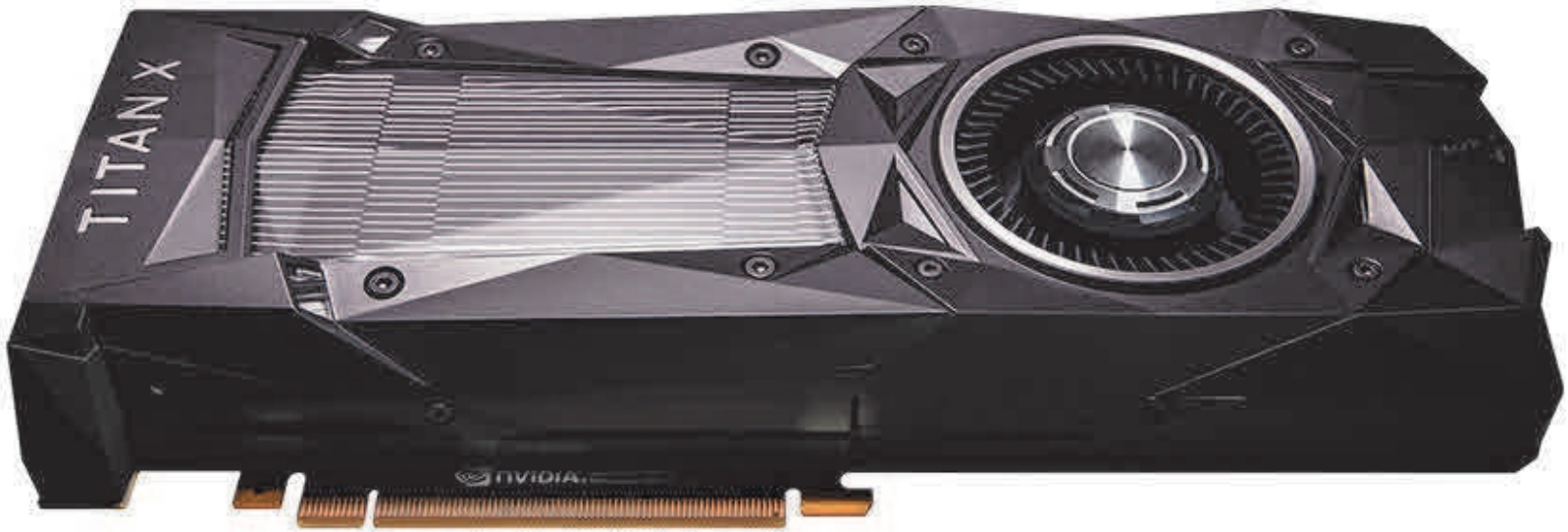
(Cisse et al, 2017)

Attacks in the real world

Attacks in the real world



GPUs



CPU vs GPU

CPU

GPU

CPU vs GPU

- CPU:
  - fewer cores; each core is faster and more powerful
  - useful for sequential tasks
- GPU:
  - more cores; each core is slower and weaker
  - great for parallel tasks

CPU vs GPU

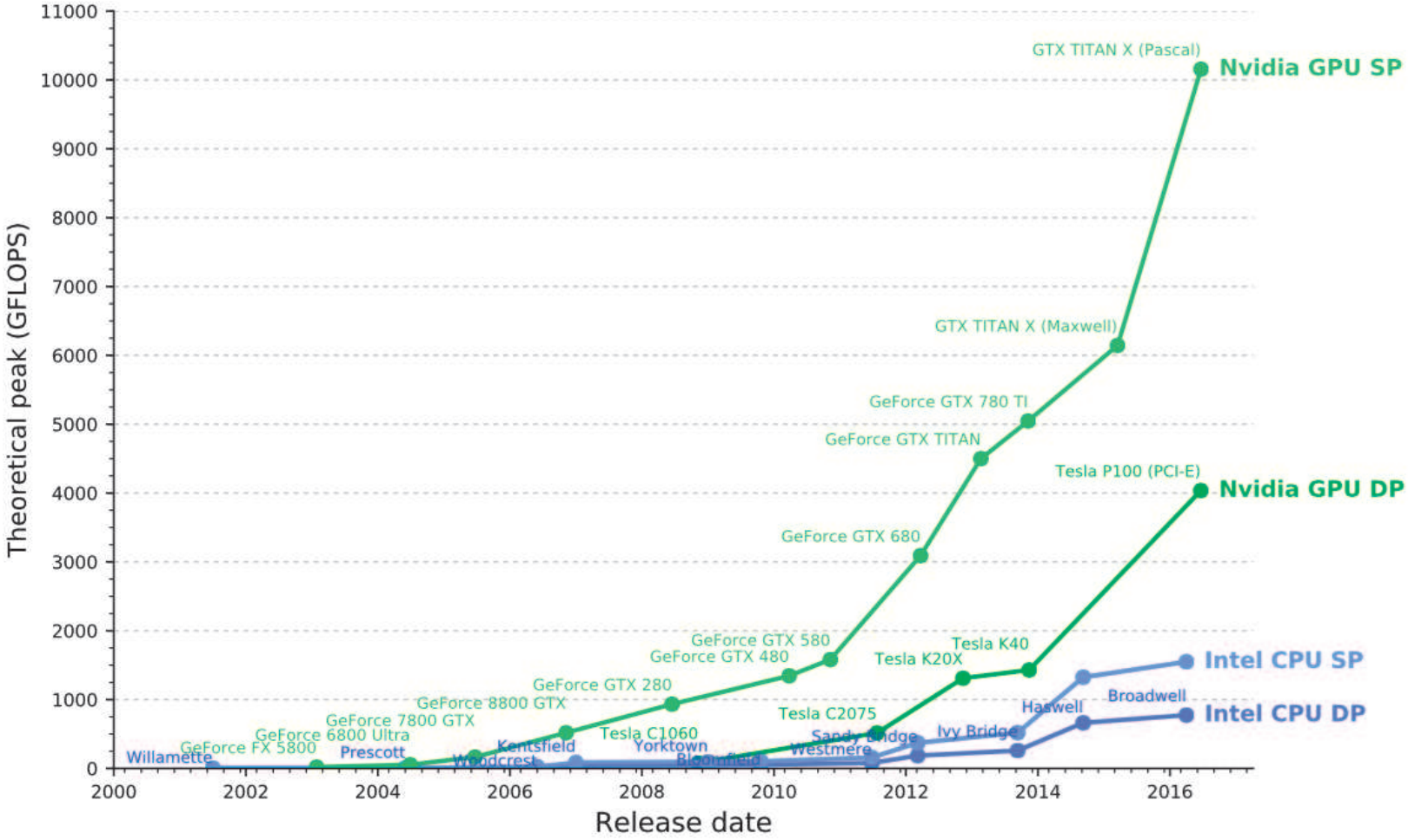
- CPU:
  - fewer cores; each core is faster and more powerful
  - useful for sequential tasks
- GPU:
  - more cores; each core is slower and weaker
  - great for parallel tasks

	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$399

Figure credit: J. Johnson

CPU vs GPU

- SP = single precision, 32 bits / 4 bytes
- DP = double precision, 64 bits / 8 bytes



CPU vs GPU

TABLE 7. COMPARATIVE EXPERIMENT RESULTS (TIME PER MINI-BATCH IN SECOND)

		Desktop CPU (Threads used)				Server CPU (Threads used)						Single GPU		
		1	2	4	8	1	2	4	8	16	32	G980	G1080	K80
FCN-S	Caffe	1.324	0.790	**0.578**	15.444	1.355	0.997	0.745	**0.573**	0.608	1.130	0.041	**0.030**	0.071
	CNTK	1.227	0.660	**0.435**	-	1.340	0.909	0.634	0.488	**0.441**	1.000	0.045	**0.033**	0.074
	TF	7.062	4.789	2.648	**1.938**	9.571	6.569	3.399	1.710	0.946	**0.630**	0.060	**0.048**	0.109
	MXNet	4.621	2.607	2.162	**1.831**	5.824	3.356	2.395	2.040	**1.945**	2.670	-	**0.106**	0.216
	Torch	1.329	0.710	**0.423**	-	1.279	1.131	0.595	0.433	**0.382**	1.034	0.040	**0.031**	0.070
AlexNet-S	Caffe	1.606	0.999	**0.719**	-	1.533	1.045	**0.797**	0.850	0.903	1.124	0.034	**0.021**	0.073
	CNTK	3.761	1.974	**1.276**	-	3.852	2.600	1.567	1.347	**1.168**	1.579	0.045	**0.032**	0.091
	TF	6.525	2.936	1.749	**1.535**	5.741	4.216	2.202	1.160	**0.701**	0.962	0.059	**0.042**	0.130
	MXNet	2.977	2.340	2.250	**2.163**	3.518	3.203	2.926	2.828	**2.827**	2.887	0.020	**0.014**	0.042
	Torch	4.645	2.429	**1.424**	-	4.336	2.468	1.543	1.248	**1.090**	1.214	0.033	**0.023**	0.070
RenNet-50	Caffe	11.554	7.671	**5.652**	-	10.643	8.600	6.723	**6.019**	6.654	8.220	-	**0.254**	0.766
	CNTK	-	-	-	-	-	-	-	-	-	-	0.240	**0.168**	0.638
	TF	23.905	16.435	10.206	**7.816**	29.960	21.846	11.512	6.294	**4.130**	4.351	0.327	**0.227**	0.702
	MXNet	48.000	46.154	44.444	**43.243**	57.831	57.143	54.545	54.545	**53.333**	55.172	0.207	**0.136**	0.449
	Torch	13.178	7.500	**4.736**	4.948	12.807	8.391	5.471	4.164	**3.683**	4.422	0.208	**0.144**	0.523
FCN-R	Caffe	2.476	1.499	**1.149**	-	2.282	1.748	1.403	1.211	1.127	**1.127**	0.025	**0.017**	0.055
	CNTK	1.845	0.970	0.661	**0.571**	1.592	0.857	0.501	0.323	**0.252**	0.280	0.025	**0.017**	0.053
	TF	2.647	1.913	1.157	**0.919**	3.410	2.541	1.297	0.661	0.361	**0.325**	0.033	**0.020**	0.063
	MXNet	1.914	1.072	0.719	**0.702**	1.609	1.065	0.731	0.534	0.451	**0.447**	0.029	**0.019**	0.060
	Torch	1.670	0.926	**0.565**	0.611	1.379	0.915	0.662	0.440	0.402	**0.366**	0.025	**0.016**	0.051
AlexNet-R	Caffe	3.558	2.587	**2.157**	2.963	4.270	3.514	3.381	**3.364**	4.139	4.930	0.041	**0.027**	0.137
	CNTK	9.956	7.263	**5.519**	6.015	9.381	6.078	4.984	**4.765**	6.256	6.199	0.045	**0.031**	0.108
	TF	4.535	3.225	1.911	**1.565**	6.124	4.229	2.200	1.396	1.036	**0.971**	**0.227**	0.317	0.385
	MXNet	13.401	12.305	12.278	**11.950**	17.994	17.128	16.764	**16.471**	17.471	17.770	0.060	**0.032**	0.122
	Torch	5.352	3.866	**3.162**	3.259	6.554	5.288	4.365	**3.940**	4.157	4.165	0.069	**0.043**	0.141
RenNet-56	Caffe	6.741	5.451	**4.989**	6.691	7.513	**6.119**	6.232	6.689	7.313	9.302	-	**0.116**	0.378
	CNTK	-	-	-	-	-	-	-	-	-	-	0.206	**0.138**	0.562
	TF	-	-	-	-	-	-	-	-	-	-	0.225	**0.152**	0.523
	MXNet	34.409	31.255	**30.069**	31.388	44.878	43.775	**42.299**	42.965	43.854	44.367	0.105	**0.074**	0.270
	Torch	5.758	3.222	**2.368**	2.475	8.691	4.965	3.040	**2.560**	2.575	2.811	0.150	**0.101**	0.301
LSTM	Caffe	-	-	-	-	-	-	-	-	-	-	-	-	-
	CNTK	0.186	0.120	**0.090**	0.118	0.211	0.139	0.117	0.114	**0.114**	0.198	0.018	**0.017**	0.043
	TF	4.662	3.385	1.935	**1.532**	6.449	4.351	2.238	1.183	0.702	**0.598**	0.133	**0.065**	0.140
	MXNet	-	-	-	-	-	-	-	-	-	-	0.089	**0.079**	0.149
	Torch	6.921	3.831	**2.682**	3.127	7.471	4.641	3.580	**3.260**	5.148	5.851	0.399	**0.324**	0.560

Note: The mini-batch sizes for FCN-S, AlexNet-S, ResNet-50, FCN-R, AlexNet-R, ResNet-56 and LSTM are 64, 16, 16, 1024, 1024, 128 and 128 respectively.

Benchmarking State-of-the-Art Deep Learning Software Tools, Shi et al., 2016

CPU vs GPU

- more benchmarks available at https://github.com/jcjohnson/cnn-benchmarks

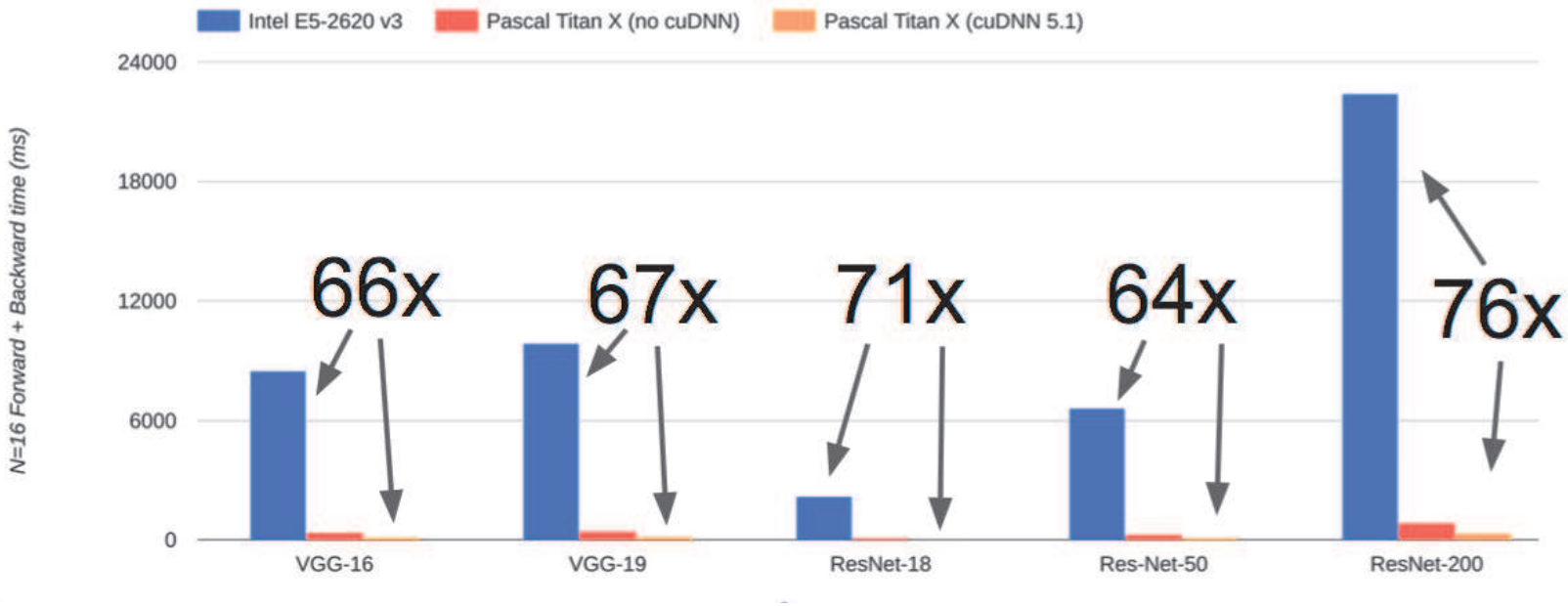


Figure credit: J. Johnson

CPU vs GPU

- more benchmarks available at https://github.com/jcjohnson/cnn-benchmarks

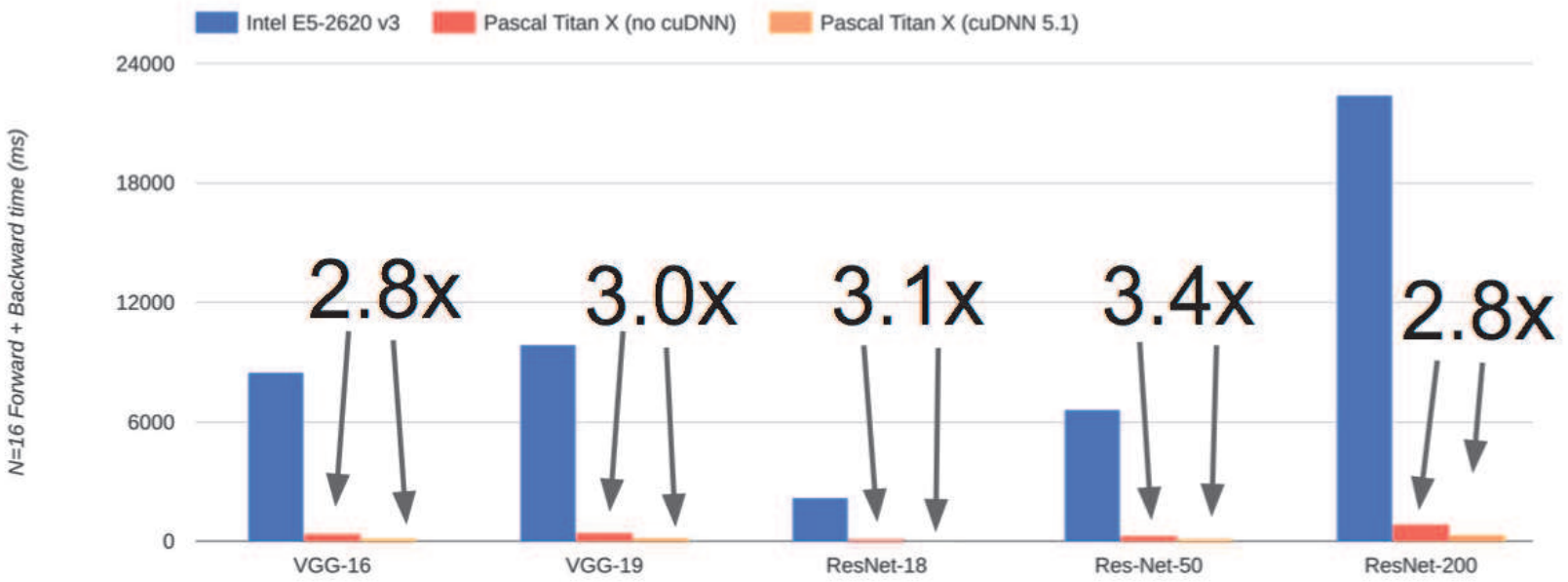
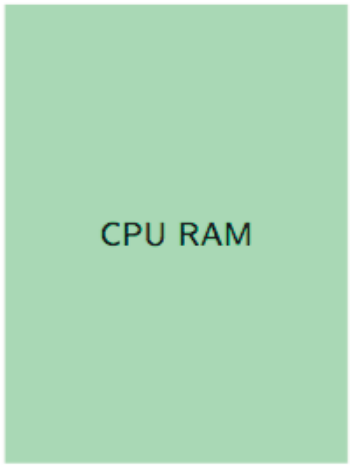


Figure credit: J. Johnson

System



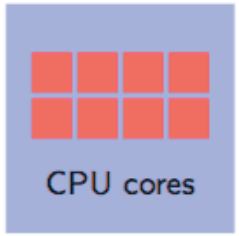


Figure credit: F. Fleuret

System

Figure credit: F. Fleuret

System

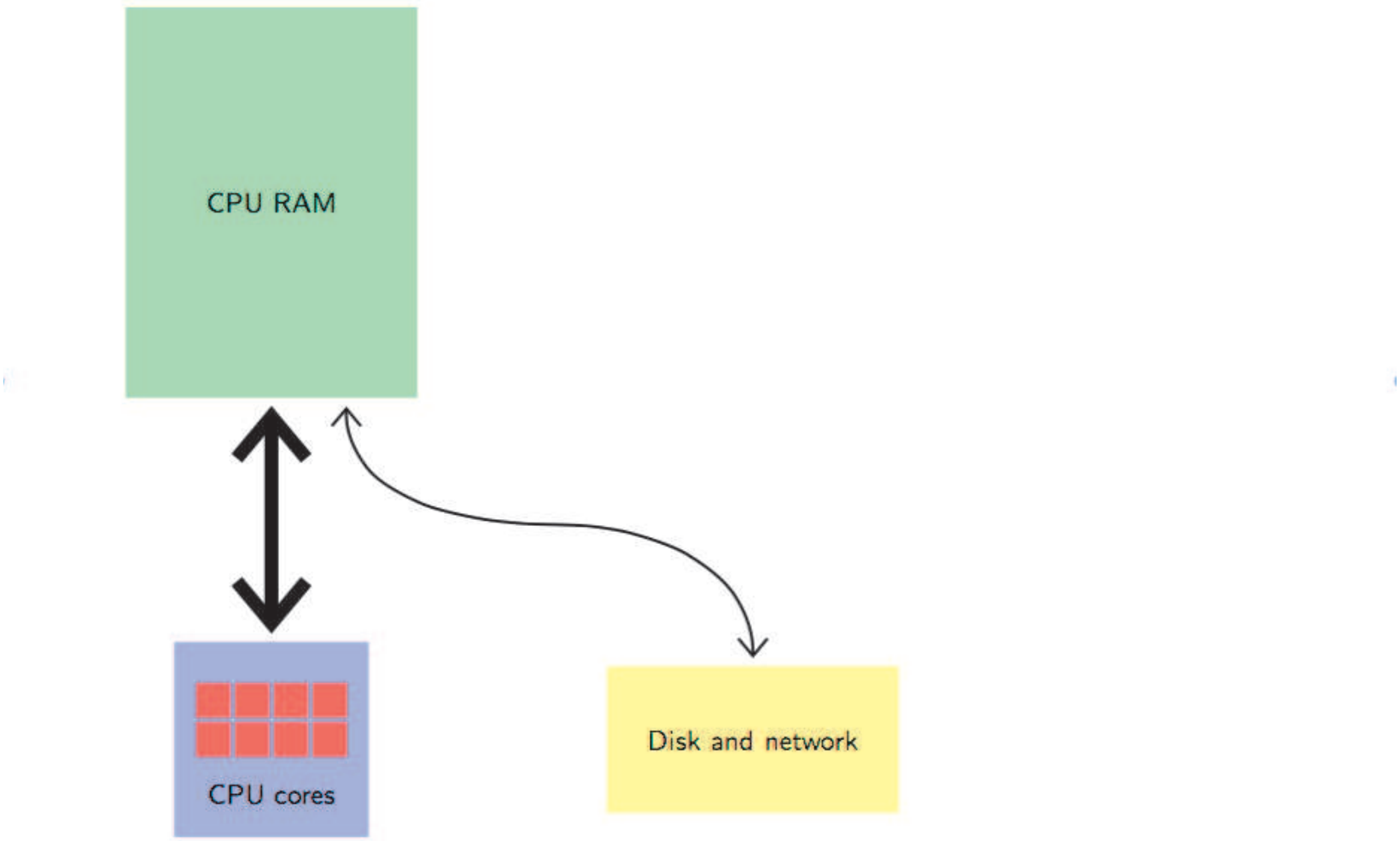


Figure credit: F. Fleuret

System

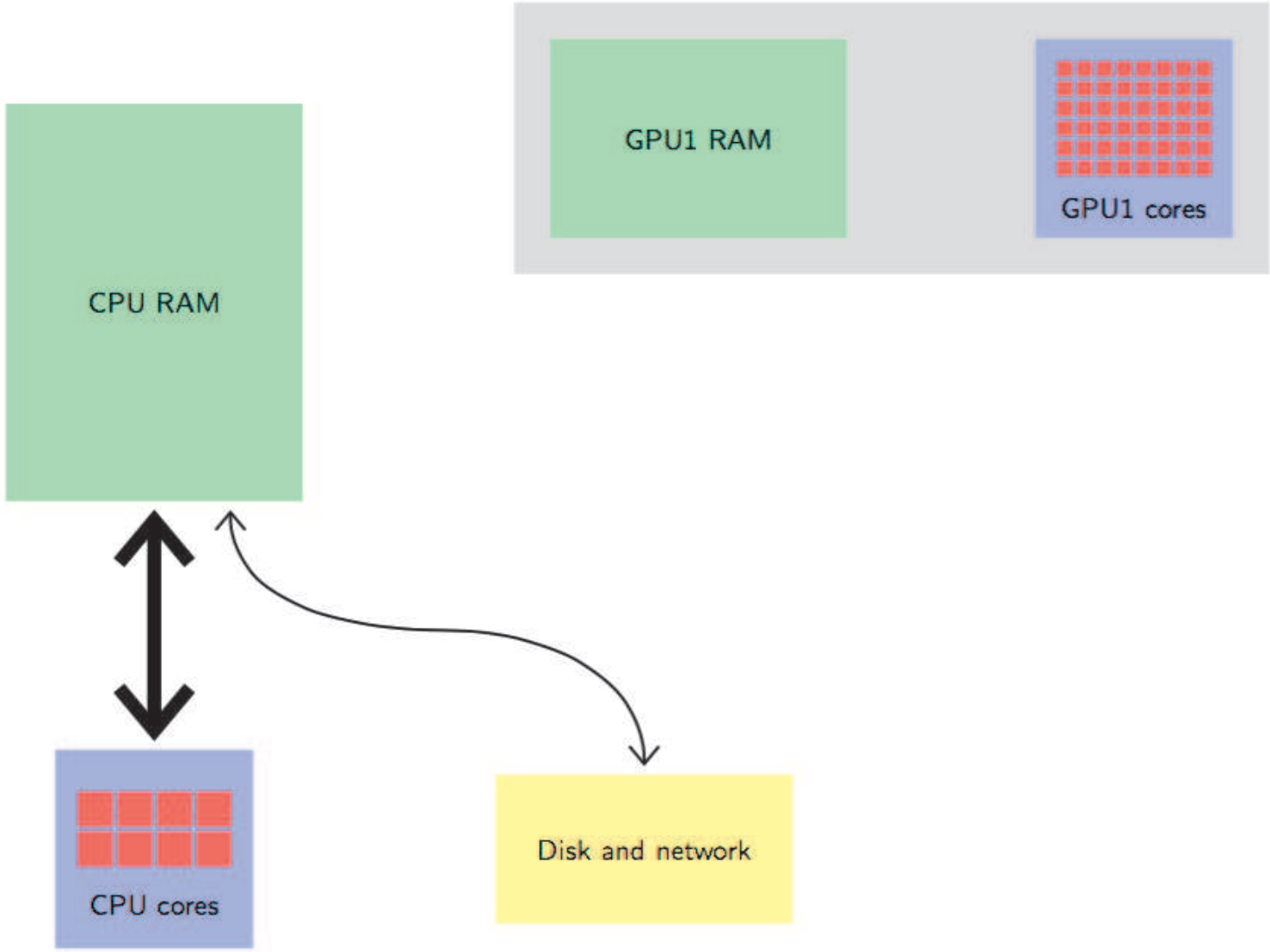


Figure credit: F. Fleuret

System

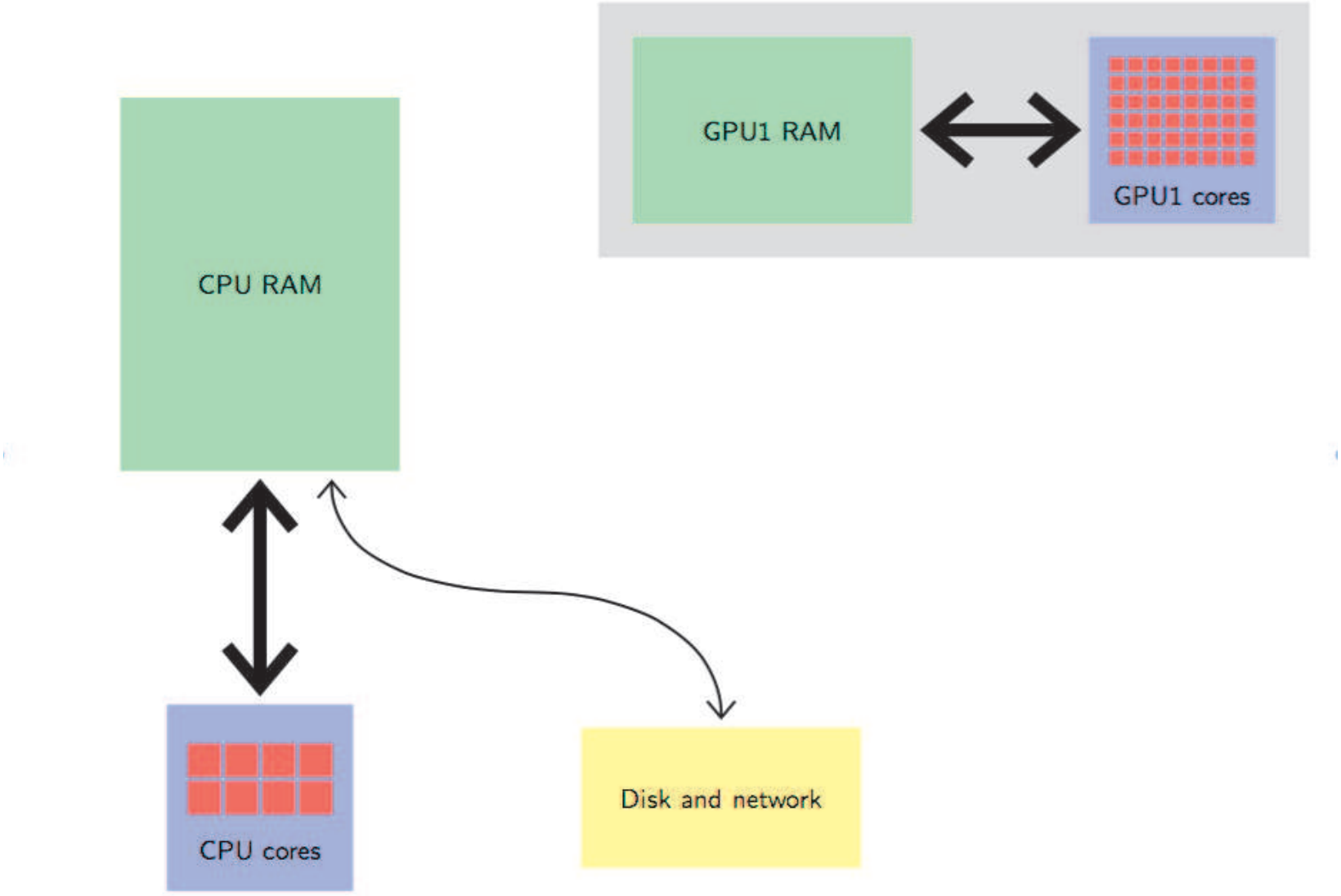


Figure credit: F. Fleuret

System

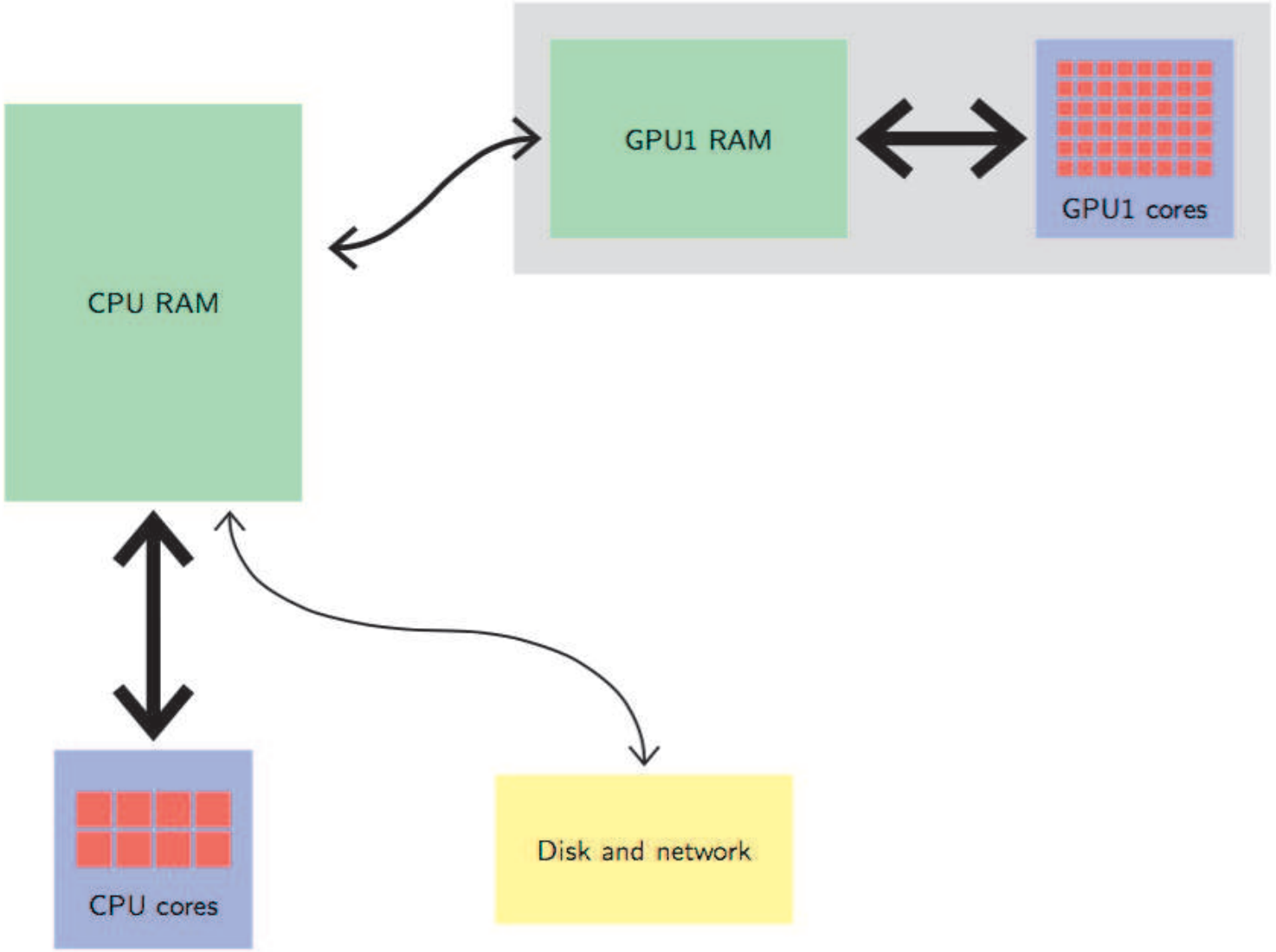


Figure credit: F. Fleuret

System

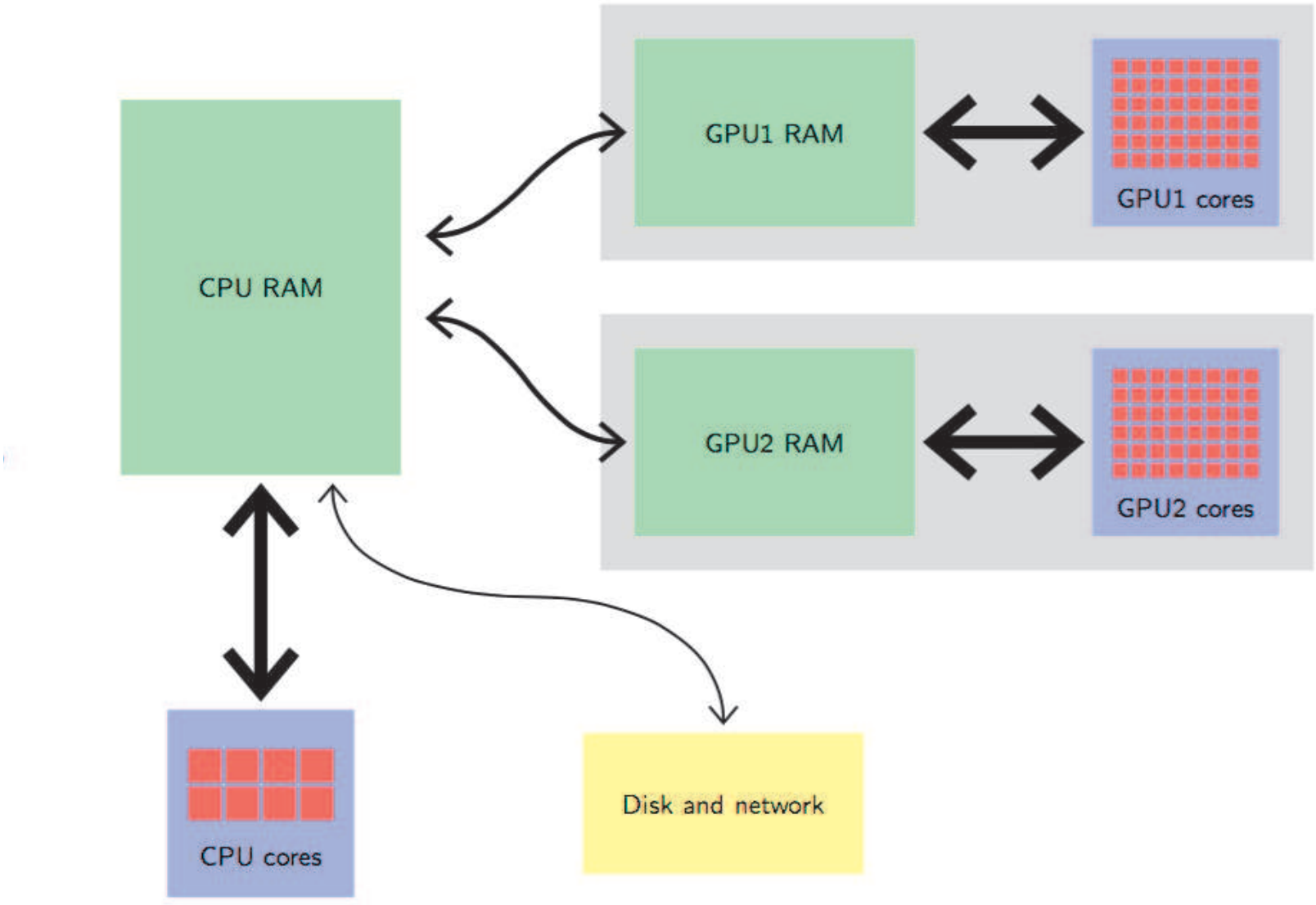


Figure credit: F. Fleuret

GPU

- NVIDIA GPUs are programmed through CUDA (Compute Unified Device Architecture)
- The alternative is OpenCL, supported by several manufacturers but with significant less investments than Nvidia
- Nvidia and CUDA are dominating the field by far, though some alternatives start emerging: Google TPUs, embedded devices.

Libraries

- BLAS (Basic Linear Algebra Subprograms): vector/matrix products, and the cuBLAS implementation for NVIDIA GPUs
- LAPACK (Linear Algebra Package): linear system solving, Eigen-decomposition, etc.
- cuDNN (NVIDIA CUDA Deep Neural Network library) computations specific to deep-learning on NVIDIA GPUs.

GPU usage in pytorch

- Tensors of torch.cuda types are in the GPU memory. Operations on them are done by the GPU and resulting tensors are stored in its memory.
- Operations cannot mix different tensor types (CPU vs. GPU, or different numerical types); except `copy_()`
- Moving data between the CPU and the GPU memories is far slower than moving it inside the GPU memory.

GPU usage in pytorch

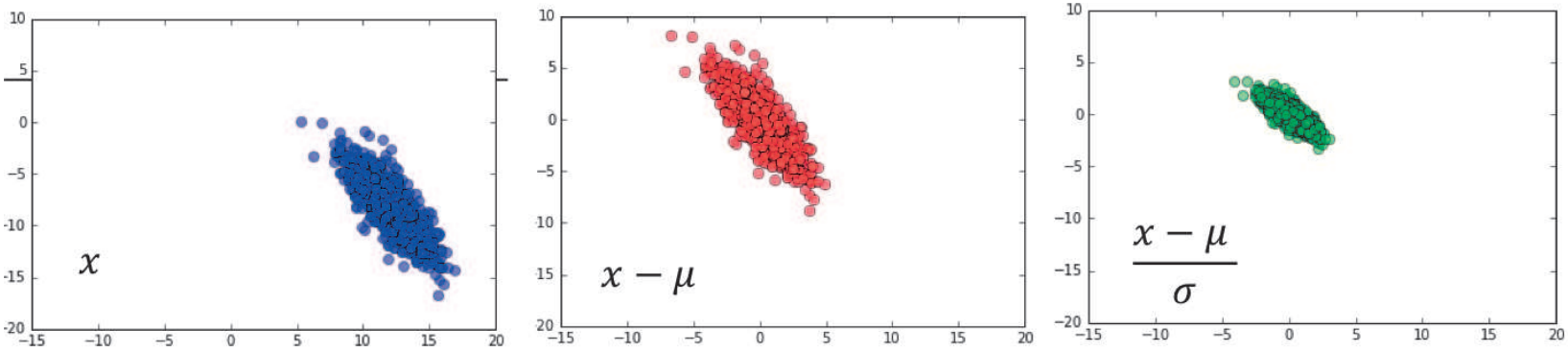
- The `Tensor` method `cuda()` returns a clone on the GPU if the tensor is not already there or returns the tensor itself if it was already there, keeping the bit precision.
- The method `cpu()` makes a clone on the CPU if needed.
- They both keep the original tensor unchanged

Training deep networks

Tricks of the trade

Data pre-processing

- Input variables should be as decorrelated as possible
  - Input variables are "more independent"
  - Network is forced to find non-trivial correlations between inputs
  - Decorrelated inputs $\rightarrow$ better optimization
- Input variables follow a more of less Gaussian distribution
- In practice:
  - compute mean and standard deviation
    - per pixel: (μ, σ^2)
    - per color channel:



Data pre-processing

Code from `torchvision/transforms/functional.py`

```
def normalize(tensor, mean, std):
...
    for t, m, s in zip(tensor, mean, std):
        t.sub_(m).div_(s)

    return tensor
```

Data augmentation

- Changing the pixels without changing the label
- Train on transformed data
- Widely used

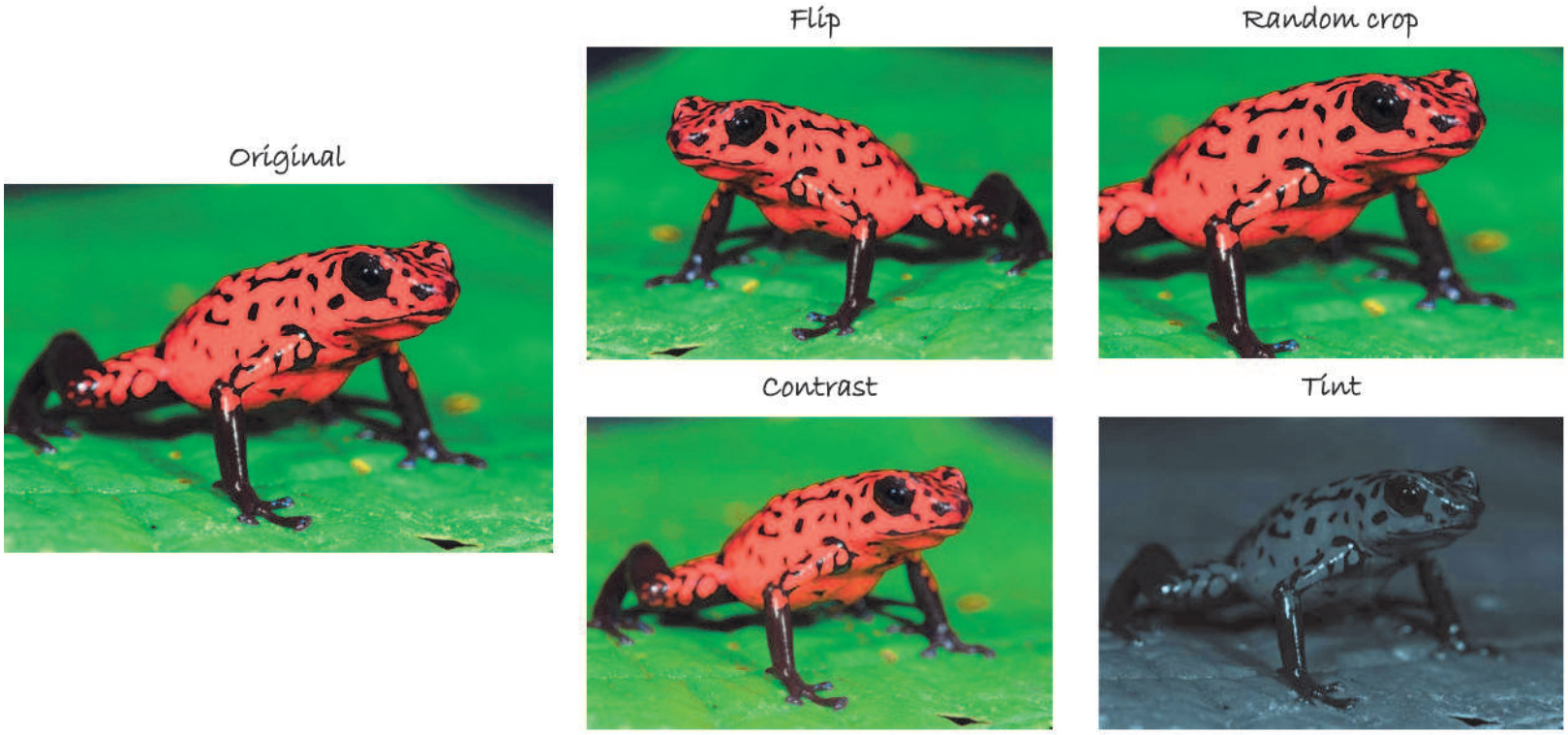


Figure credit: E. Gavves

Data augmentation

Horizontal flips

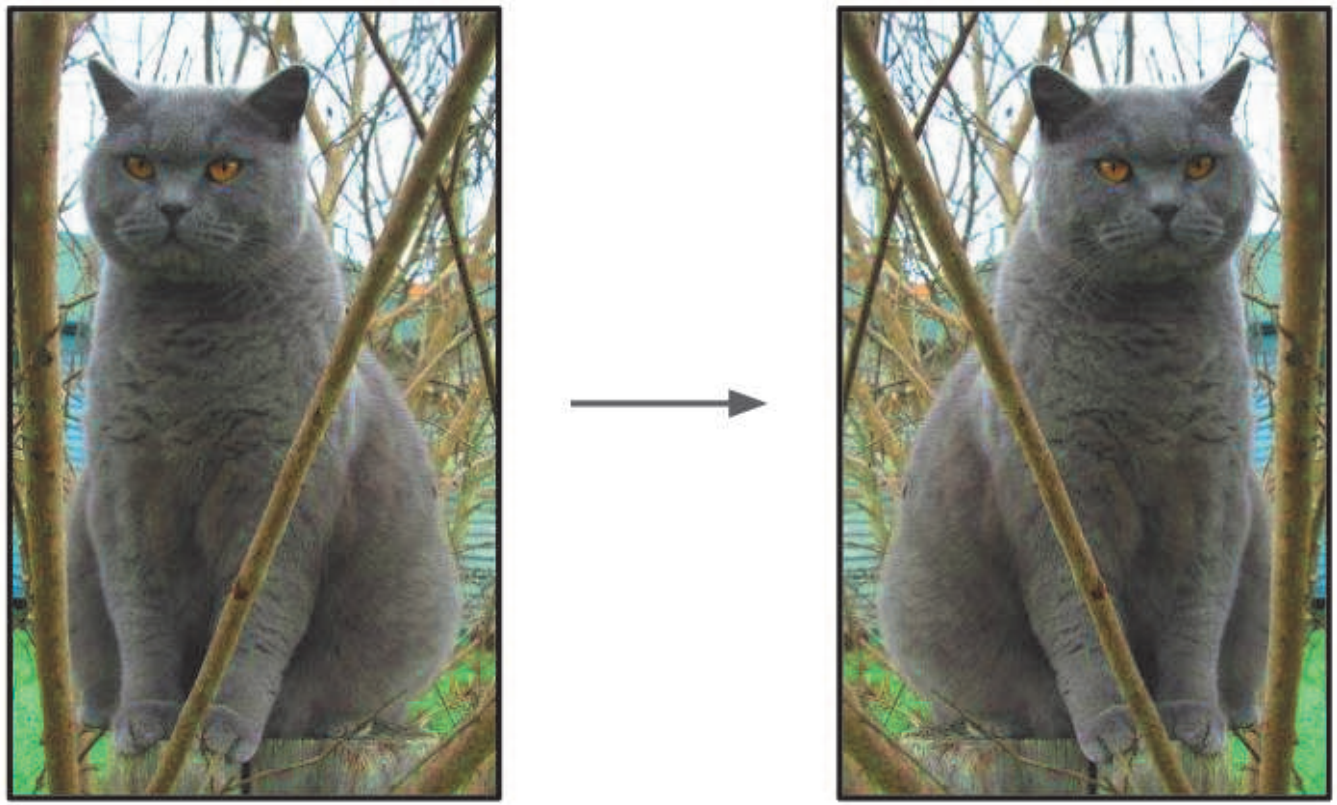


Figure credit: A. Karpathy

Data augmentation

Random crops/scales

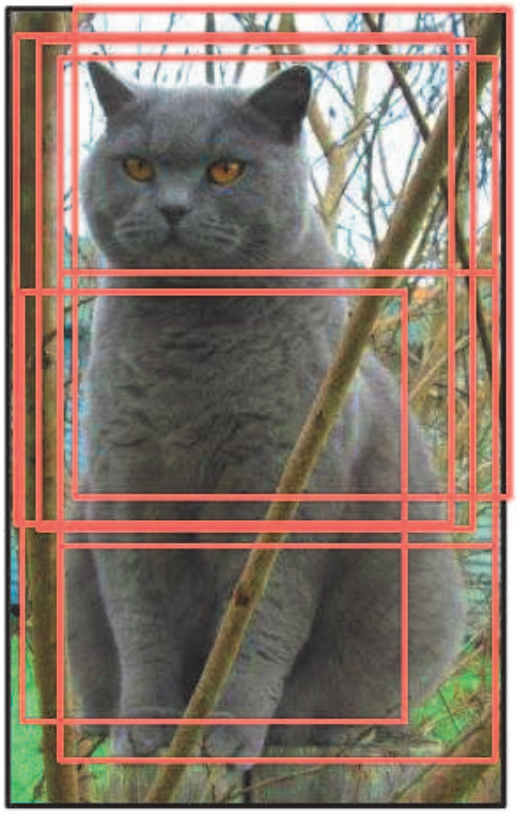
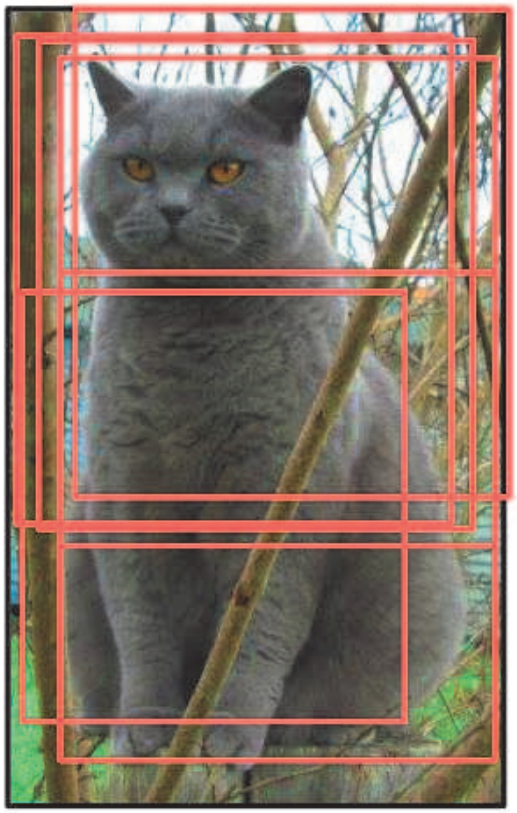


Figure credit: A. Karpathy

Data augmentation

Random crops/scales

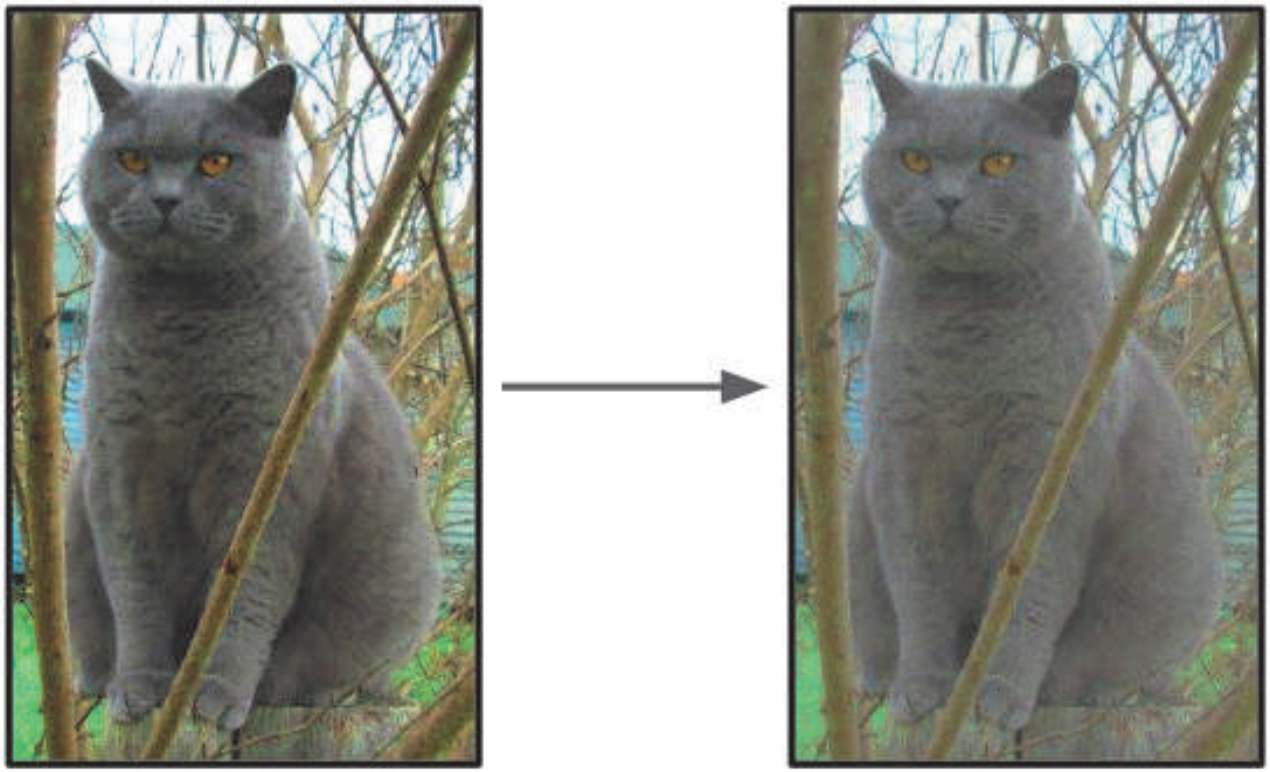


- Training: sample random crops/scales
- Testing: average a fixed set of crops

Figure credit: A. Karpathy

Data augmentation

Color jitter

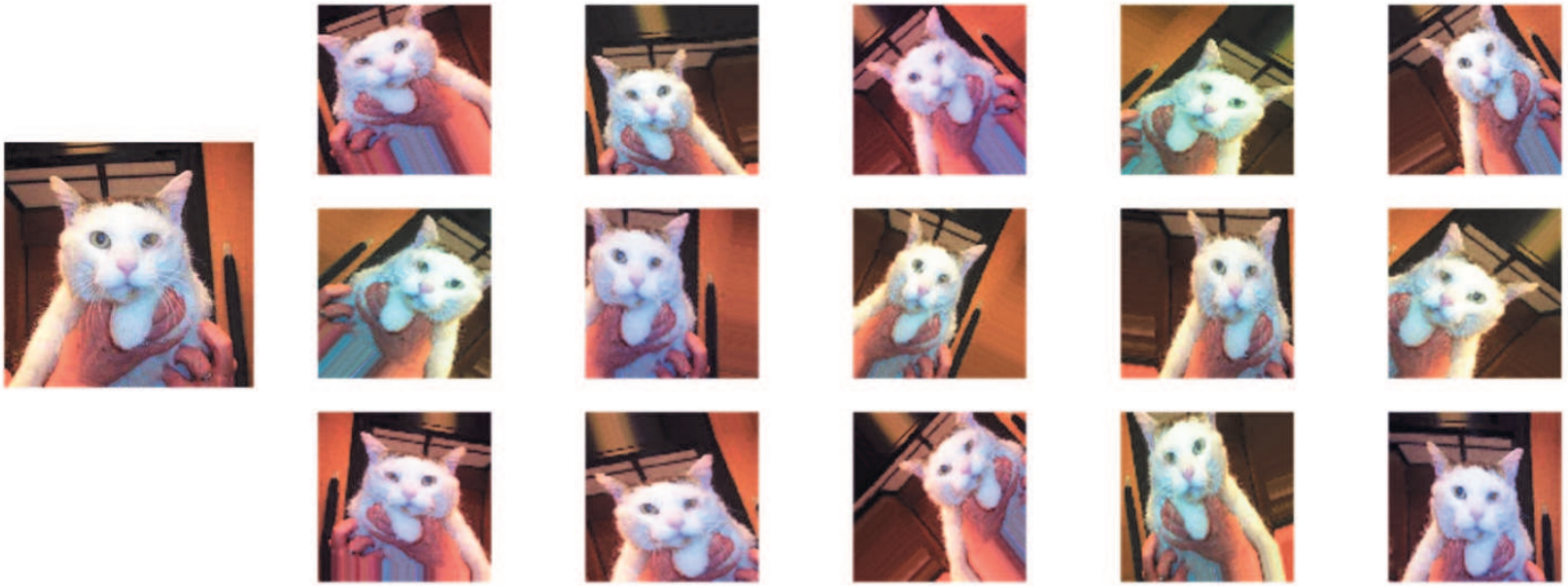


- randomly jitter color, brightness, contrast, etc.
- other more complex alternatives exist (PCA-jittering)

Figure credit: A. Karpathy

Data augmentation

- Various techniques can be mixed
- Domain knowledge helps in finding new data augmentation techniques
- Very useful for small datasets



Data augmentation

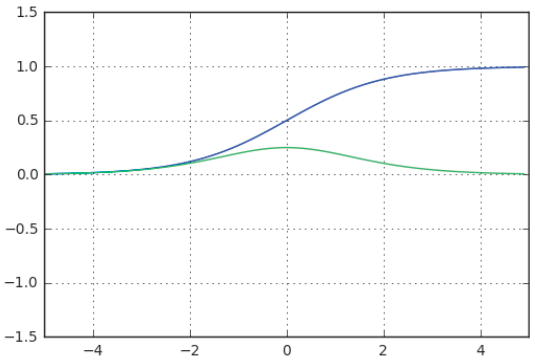
```
from torchvision import transforms

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, stauration=0.2, hue=0
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Scale(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}
```

no need for data augmentation on validation set

Weight initialization

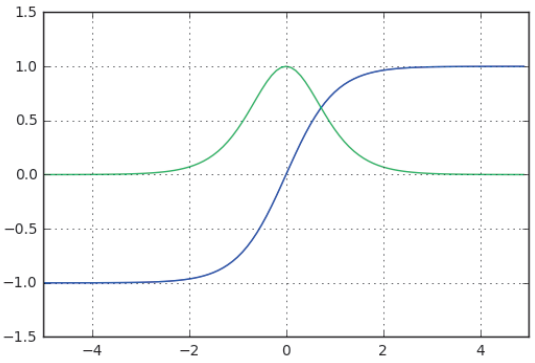
There are a few contradictory requirements:

- Weights need to be small enough
  - around origin for symmetric activation functions (tanh, sigmoid) → stimulate activation functions near their linear regime
  - larger gradients → faster training
- Weights need to be large enough
  - otherwise signal is too weak for any serious learning



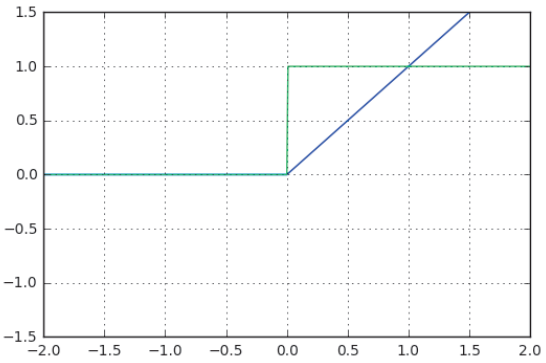
$$\text{sigm}(x) = \frac{1}{1+e^{-x}}$$

$$\text{sigm}'(x) = \text{sigm}(x)(1-\text{sigm}(x))$$



$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\tanh'(x) = 1-\tanh(x)^2$$



$$\text{relu}(x) = \max(0,x)$$

$$\text{relu}'(x) = 1_{x>0}$$

Weight initialization

- Weights should evolve at the same rate across layers during training, and no layer should reach a saturation behavior before others.
- Weights must be initialized to preserve the variance of the activations during the forward and backward computations
  - neurons will operate in their full capacity
- Initialize weights to be asymmetric
  - if all weights are 0, neurons generate same gradient
- Initialization depends on non-linearities and data normalization

Weight initialization

From `torch/nn/modules/linear.py`

```
def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
    self.weight.data.uniform_(-stdv, stdv)
    if self.bias is not None:
        self.bias.data.uniform_(-stdv, stdv)
```

Weight initialization

From `torch/nn/modules/linear.py`

```
def reset_parameters(self):
    stdv = 1. / math.sqrt(self.weight.size(1))
    self.weight.data.uniform_(-stdv, stdv)
    if self.bias is not None:
        self.bias.data.uniform_(-stdv, stdv)
```

When used with tanh almost all neurons get completely either -1 and 1. Gradients will be zero

Xavier initialization

- We get a better compromise with "Xavier initialization"
- From `torch/nn/init.py`:

```
def xavier_normal(tensor, gain=1):
    if isinstance(tensor, Variable):
            xavier_normal(tensor.data, gain=gain)
            return tensor

    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    std = gain * math.sqrt(2.0 / (fan_in + fan_out))
    return tensor.normal_(0, std)
```

`fan_in` = num neurons in the input

`fan_out` = num neurons at the output

Understanding the difficulty of training deep feedforward neural networks, Glorot & Bengio, 2010

Xavier initialization

- We get a better compromise with "Xavier initialization"
- From `torch/nn/init.py`:

```
def xavier_normal(tensor, gain=1):
    if isinstance(tensor, Variable):
            xavier_normal(tensor.data, gain=gain)
            return tensor

    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    std = gain * math.sqrt(2.0 / (fan_in + fan_out))
    return tensor.normal_(0, std)
```

Unlike sigmoids, ReLUs ground to 0 the linear activation about half the time

Understanding the difficulty of training deep feedforward neural networks, Glorot & Bengio, 2010

Kaiming He initialization

- Double weight variance (i.e. multiply with $\sqrt{2}$) in order to:
  - compensate for the zero flat area → input and output maintain same variance
  - very similar to Xavier initialization
- From `torch/nn/init.py`:

```
def kaiming_normal(tensor, a=0, mode='fan_in'):
  if isinstance(tensor, Variable):
      kaiming_normal(tensor.data, a=a, mode=mode)
      return tensor

  fan = _calculate_correct_fan(tensor, mode)
  gain = calculate_gain('leaky_relu', a)
  std = gain / math.sqrt(fan)
  return tensor.normal_(0, std)
```

$$gain = \sqrt{2}$$

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification, He et al., 2015

Kaiming He initialization

The same type of reasoning can be applied to other activation functions

From `torch/nn/init.py`:

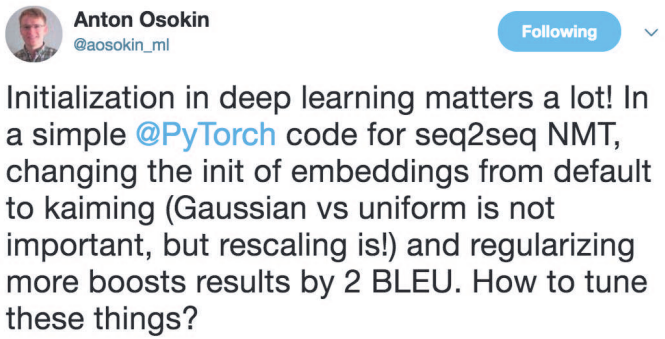
```
def calculate_gain(nonlinearity, param=None):
  linear_fns = ['linear', 'conv1d', 'conv2d', 'conv3d', 'conv_transpose1d', 'conv_
      if nonlinearity in linear_fns or nonlinearity == 'sigmoid':
          return 1
      elif nonlinearity == 'tanh':
          return 5.0 / 3
      elif nonlinearity == 'relu':
          return math.sqrt(2.0)
      elif nonlinearity == 'leaky_relu':
          if param is None:
              negative_slope = 0.01
          elif not isinstance(param, bool) and isinstance(param, int) or isinstanc
              # True/False are instances of int, hence check above
              negative_slope = param
          else:
              raise ValueError("negative_slope {} not a valid number".format(param
          return math.sqrt(2.0 / (1 + negative_slope ** 2))
      else:
          raise ValueError("Unsupported nonlinearity {}".format(nonlinearity))
```

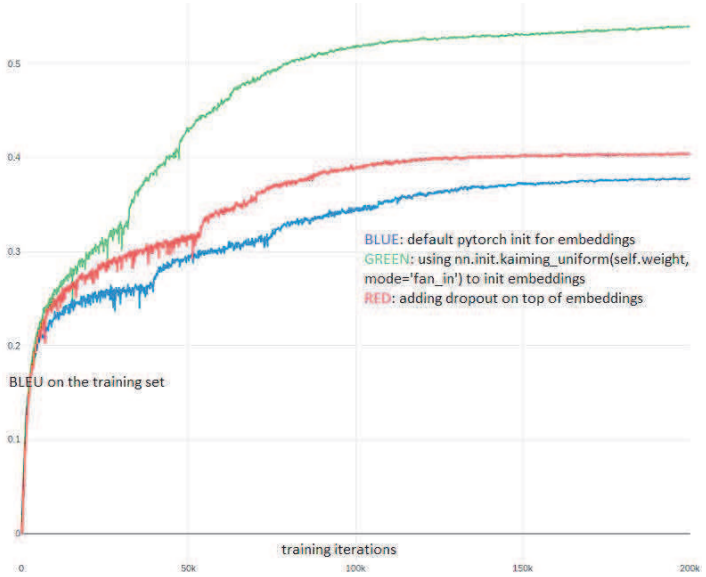
Weight initialization

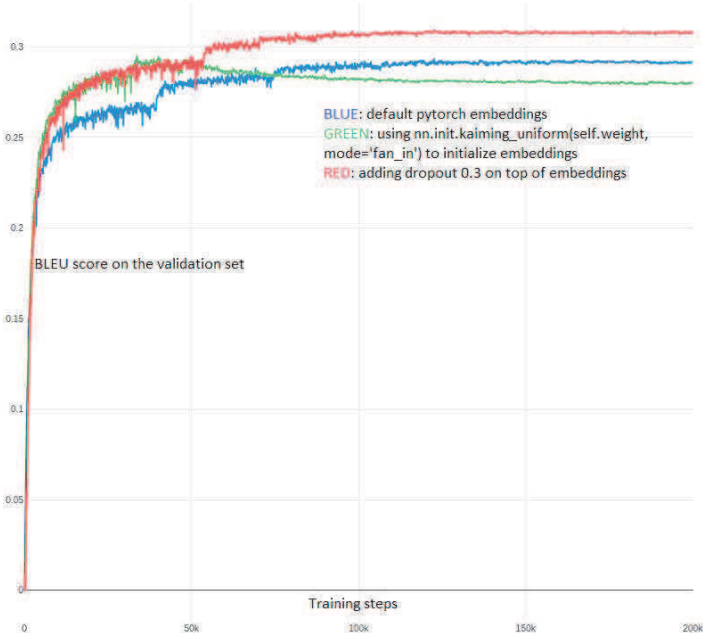
Does it actually matter that much?

Weight initialization

Does it actually matter that much?







Hyper-parameter search

- Coarse → fine cross validation stage
- First stage: only a few epochs to get rough idea of what params work
- Second stage: longer running time, finer search

- Usually there are some typical values for:

  - Learning rate: [1e-1,1e-5] (log space steps)
  - weight-decay: 0.0005
  - momentum: 0.5, 0.9, 0.99

- Learning rate:

  - For learning rate use log scale when checking values
  - If loss == NaN , learning rate is too big
  - If loss stagnates, learning rate is too small

Architecture hyperparamenters

There is no silver bullet.

- Re-use something well known that works and start from there
- Modulate the capacity until it overfits a small subset, but does not overfit / underfit the full set
- Capacity increases with more layers, more channels, larger receptive fields, or more units
- Regularization to reduce the capacity or induce sparsity
- Use prior knowledge about the "scale of meaningful context" to size filters / combinations of filters (e.g. knowing the size of objects in a scene, the max duration of a sound snippet that matters)
- Grid-search all the variations that come to mind (if you can afford to)

Slide credit: F. Fleuret

Architecture hyperparamenters

- Number of hidden layers
  - start small (a few layers) and increase complexity gradually
  - add more layers → check if performance (on validation set) improves
  - add more neurons → check if performance (on validation set) improves

Architecture hyperparamenters

- Number of hidden layers
  - start small (a few layers) and increase complexity gradually
  - add more layers → check if performance (on validation set) improves
  - add more neurons → check if performance (on validation set) improves
- Activation function
  - start with ReLU then check out others: LeakyReLU, PReLU, etc.

Architecture hyperparamenters

- Number of hidden layers
  - start small (a few layers) and increase complexity gradually
  - add more layers → check if performance (on validation set) improves
  - add more neurons → check if performance (on validation set) improves
- Activation function
  - start with ReLU then check out others: LeakyReLU, PReLU, etc.
- Type and amount of regularization
  - use L_2 even if network is deep or wide
  - weight decay = $5e-5$
  - you can set weight decay to 0 is learning rate is very small.

Learning rate

The most tweaked hyperparameter

Ben Recht

Learning rate

The most tweaked hyperparameter

Ben Recht

Very active area of research!

Learning rate

The appropriate learning rate will lead to faster convergence by:

- reducing the loss quickly → large learning rate
- not be trapped in bad minimum → large learning rate
- not bounce around in narrow valleys → small learning rate
- not oscillate around a minimum → small learning rate

Slide credit: F. Fleuret

Learning rate

The appropriate learning rate will lead to faster convergence by:

- reducing the loss quickly → large learning rate
- not be trapped in bad minimum → large learning rate
- not bounce around in narrow valleys → small learning rate
- not oscillate around a minimum → small learning rate

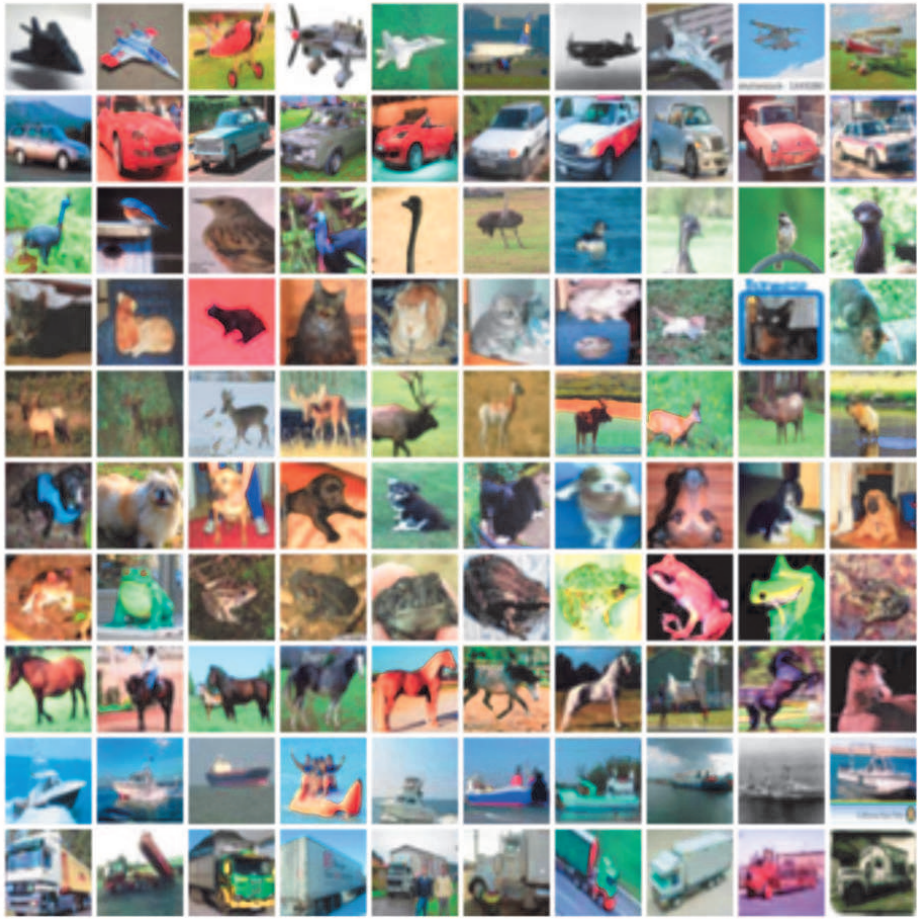
So learning rate should be larger at the beginning and smaller in the end.

The practical strategy is to look at the losses and error rates across epochs and pick a learning rate and learning rate adaptation.

Slide credit: F. Fleuret

Learning rate

CIFAR10 dataset



32 x 32 color images, 50k train samples, 10k test samples, 10 classes

Learning rate

Small CNN on CIFAR10, cross-entropy, batch size 100, η = 1e-1

Figure credit: F. Fleuret

Learning rate

Small CNN on CIFAR10, cross-entropy, batch size 100

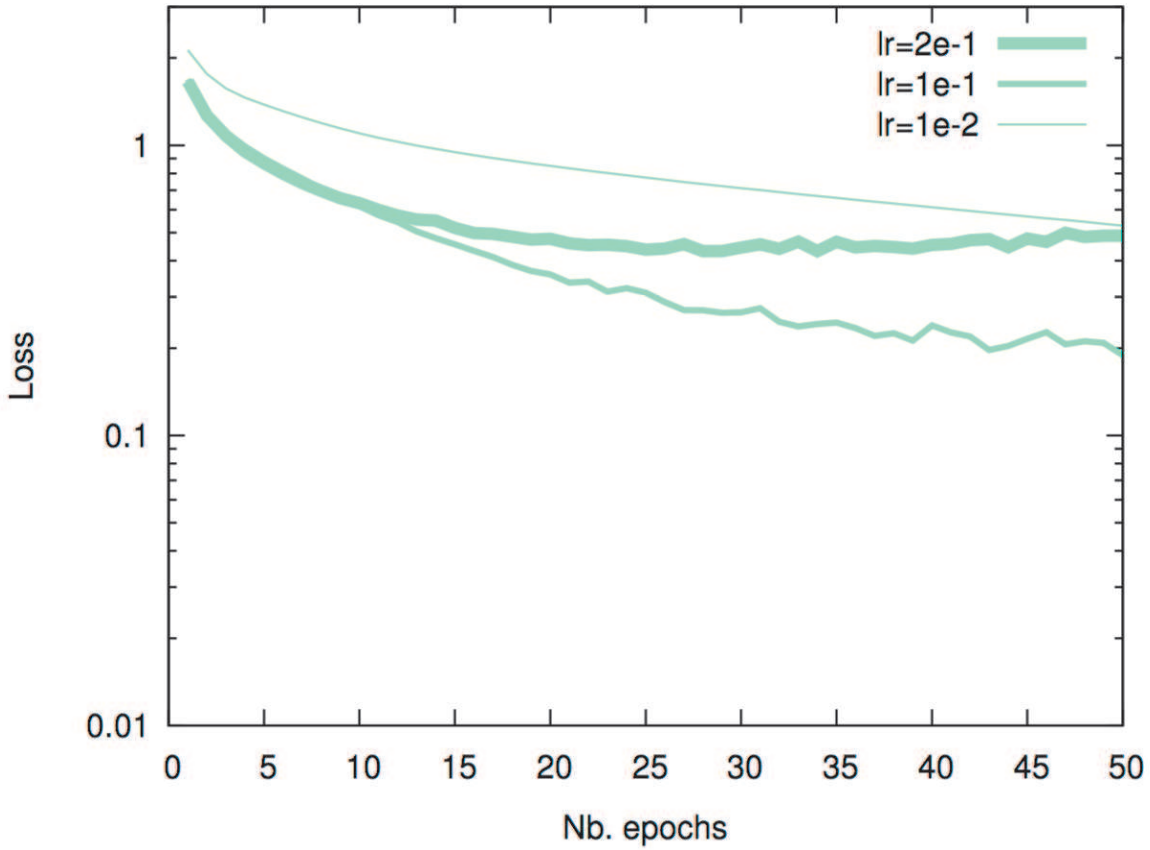


Figure credit: F. Fleuret

Learning rate

Using η=1e-1 for 25 epochs, then reducing it.

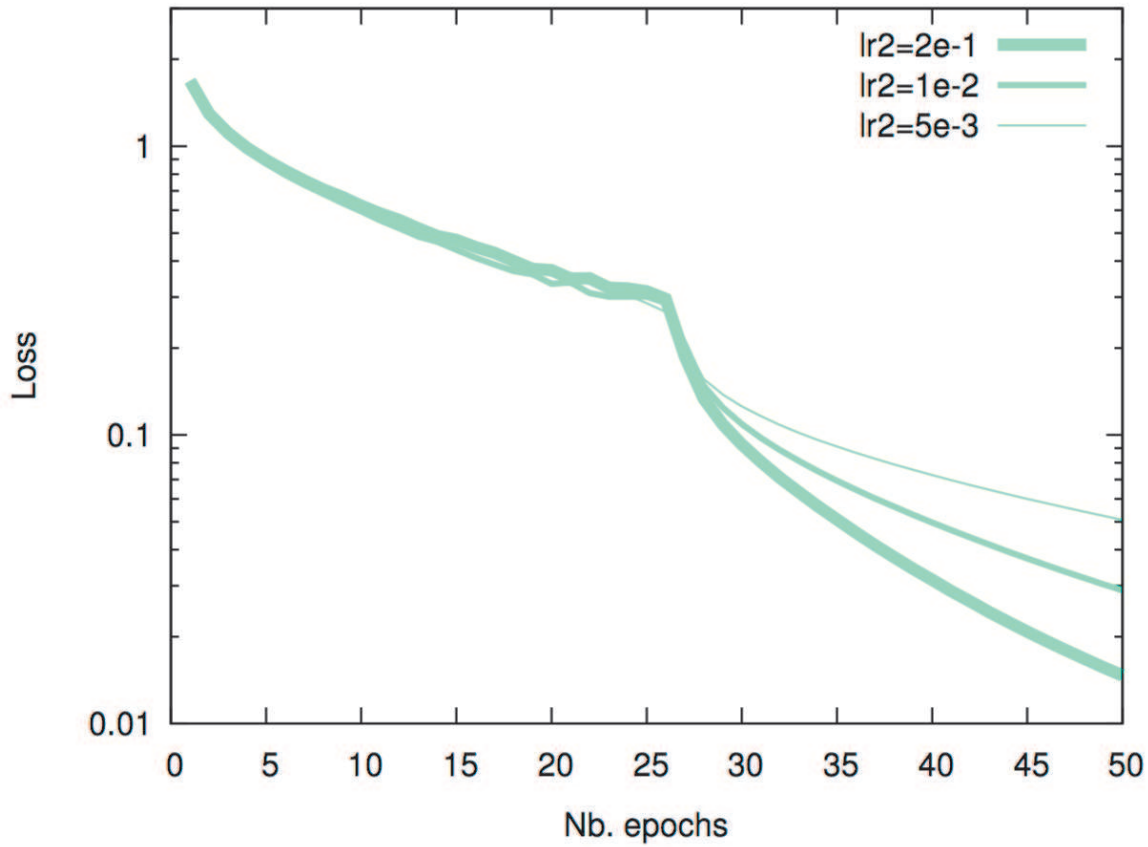


Figure credit: F. Fleuret

Learning rate

Using η=1e-1 for 25 epochs, then reducing reducing it to 1e-2

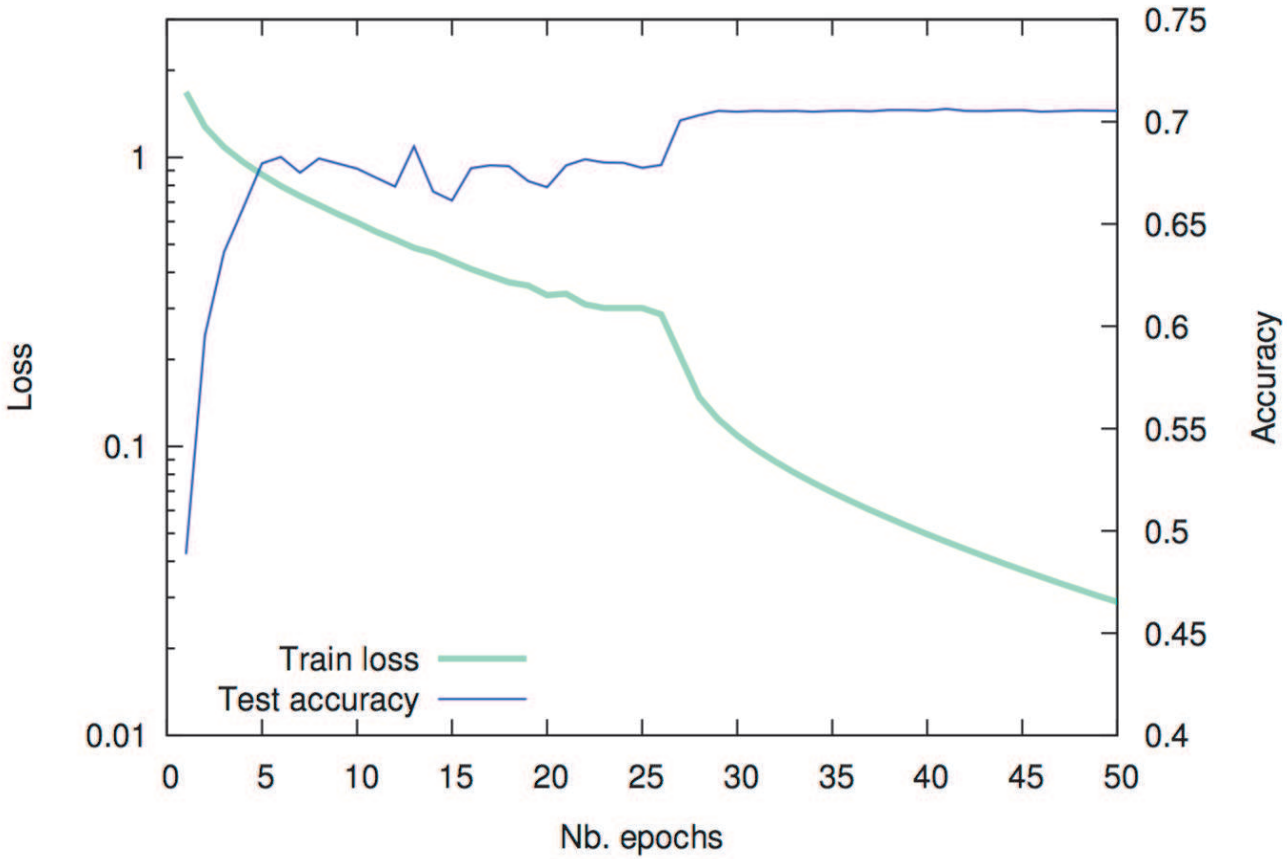


Figure credit: F. Fleuret

Learning rate

The test loss is a poor performance indicator, as it may increase even more on misclassified examples, and decrease less on the ones getting fixed.

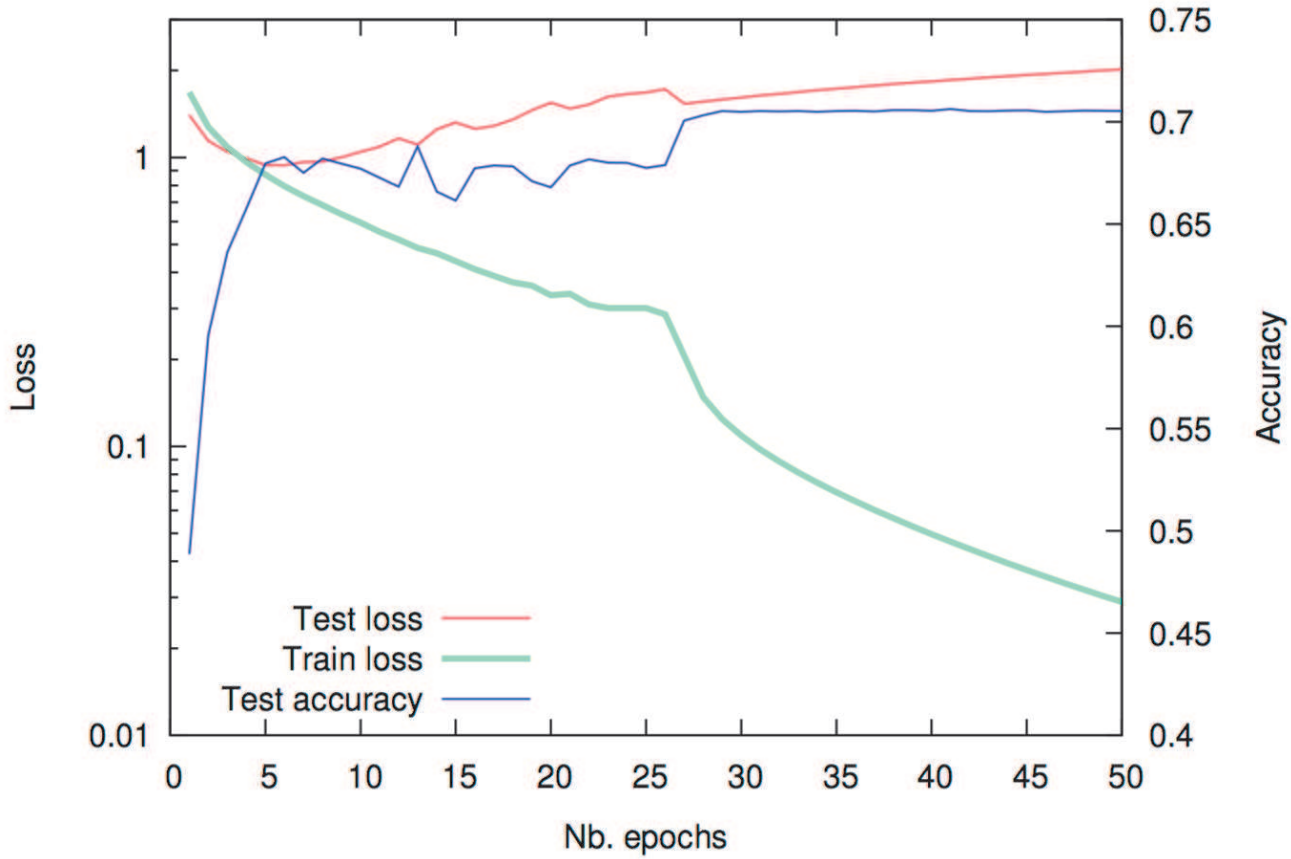


Figure credit: F. Fleuret

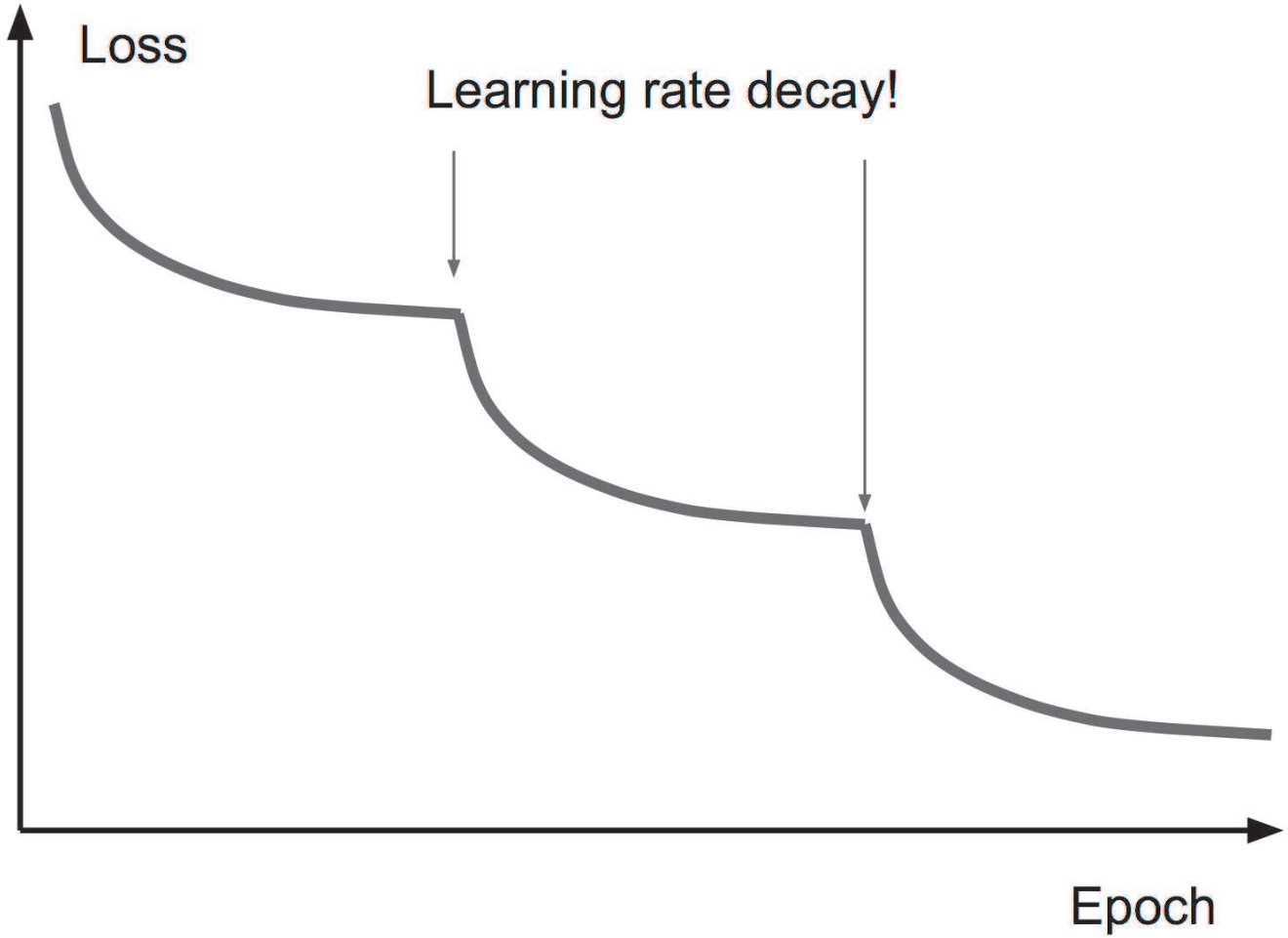
Learning rate schedules

Decay learning rate over time:

- constant: learning rate remains constant for all epochs (not a good idea)
- step decay: decay learning by fixed amount (e.g. half) every few epochs
- exponential decay: $\eta = \eta_0 e^{-kt}$
- inverse decay: $\eta = \frac{\eta_0}{1+kt}$

In many cases, step decay is preferred.

Learning rate schedules

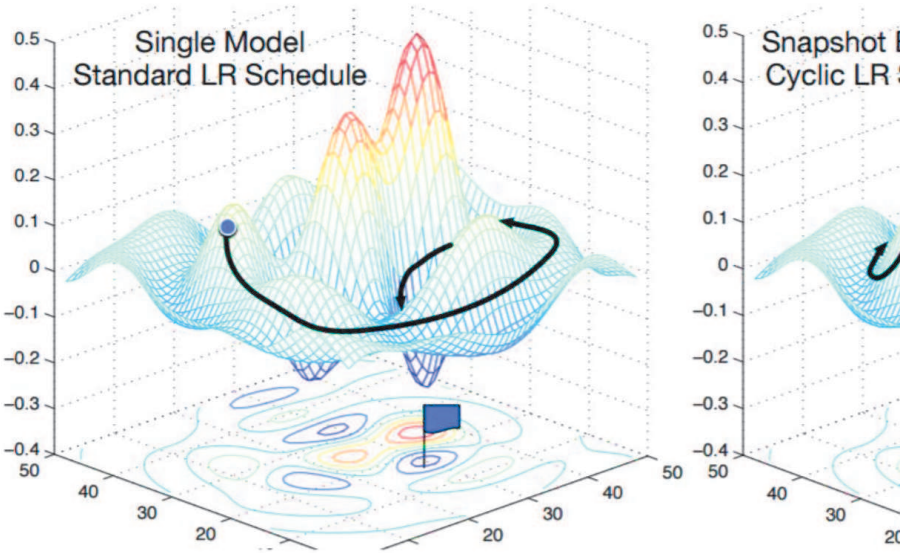


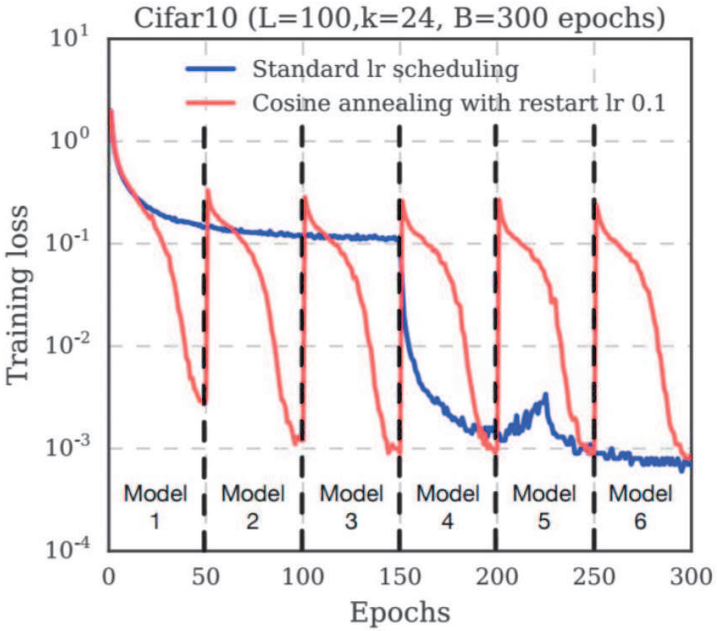
Decay is more common for SGD+momentum and less for Adam.

Learning rate schedules

Cyclic learning rates

Use multiple snapshots of a single model.





Snapshoht ensembles: train 1, get M free, Huang et al., ICLR 2017

Learning rate schedules

Using `torch.optim.lr_scheduler`:

Vanilla variants: `StepLR`, `MultiStepLR`, `ExponentialLR`

```
# Assuming optimizer uses lr = 0.5 for all groups
# lr = 0.05     if epoch < 30
# lr = 0.005    if 30 <= epoch < 60
# lr = 0.0005   if 60 <= epoch < 90
# ...
scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
for epoch in range(100):
    scheduler.step()
    train(...)
    validate(...)
```

```
# Assuming optimizer uses lr = 0.5 for all groups
# lr = 0.05     if epoch < 30
# lr = 0.005    if 30 <= epoch < 80
# lr = 0.0005   if epoch >= 80
scheduler = MultiStepLR(optimizer, milestones=[30,80], gamma=0.1)
for epoch in range(100):
    scheduler.step()
    train(...)
    validate(...)
```

Learning rate schedules

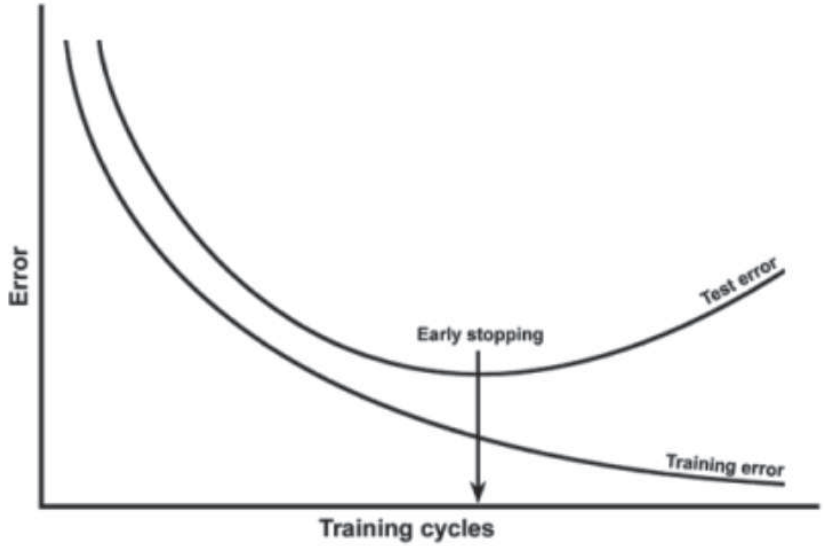
Using `torch.optim.lr_scheduler`:

Novel variants: `ReduceLROnPlateau`

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
scheduler = ReduceLROnPlateau(optimizer, 'min')
for epoch in range(10):
    train(...)
    val_loss = validate(...)
    # Note that step should be called after validate()
    scheduler.step(val_loss)
```

Early stopping

- To avoid overfitting another popular technique is early stopping
- Monitor performance on validation set
- Training the network will decrease training error, as well validation error (although with a slower rate usually)
- Stop when validation error starts increasing
  - most likely the network starts to overfit
  - use a patience term to let it degrade for a while and then stop



Loss functions

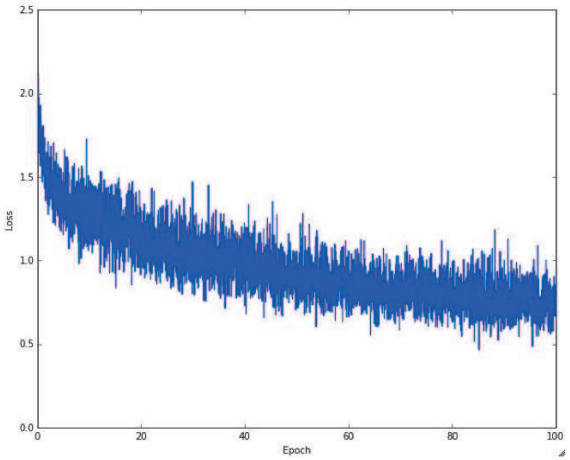
- Typically training is easier for classification than for regression to a scalar
- However many Computer Vision papers rely on regression losses (`MSE`, `L1`, `Huber`,etc.) with good results
- Multiple losses can be considered:
  - on the same output
  - by adding multiple heads to the network (e.g. classification + localization)
- pytorch has already many loss functions/criterions readily available

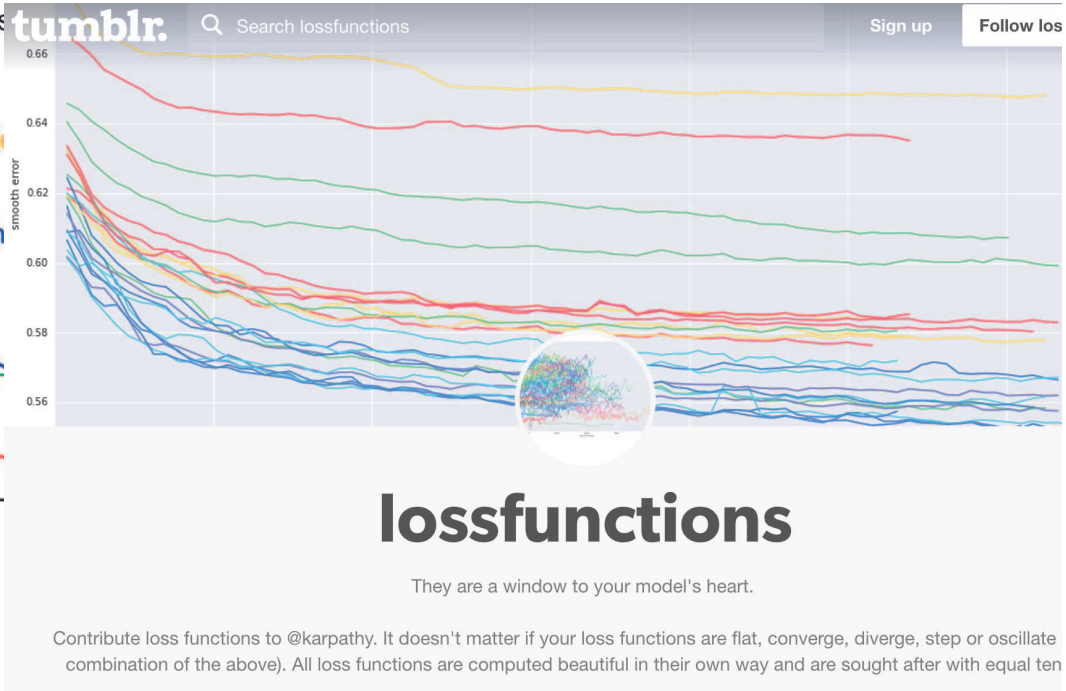
Summary

- Preprocess data to be centered on zero
- Initialize weights based on activation functions
- Always use L_2 regularization and dropout
- Use batch normalization generously
- Start with Adam, but switch to SGD once more familiar with the data and the problem

Babysitting your network

Lots of curve monitoring





Discover more bizarre looking curves

https://lossfunctions.tumblr.com/

Babysitting your network

- Always check gradients if not computed automatically
- Check that in the first steps you get a random loss
- Check network with few samples
  - turn off regularization. You should predictably overfit and have a 0 loss ○
  - turn or regularization. The loss should increase
- Have a separate validation set
  - Compare the curve between training and validation sets
  - There should be a gap, but not too large

Other common pitfalls

- inputs in range $[0, 255]$ instead of $[0, 1]$
- different pre-processing between train, valid, test
- non-shuffled dataset
- class imbalance
- too much data augmentation
- too much regularization

Other common pitfalls

- too much/too little capacity
- bugs in the loss function: wrong input, wrong gradients
- wrong dimensions of the layers
- exploding/vanishing gradients
- given too little time for training
- forgot in-appropriate `.train()`/`.eval()` flag on

Transfer learning

- Assume two datasets S and T
- Dataset S is fully annotated, plenty of images and we can train a model CNN_S on it
- Dataset T is not as much annotated and/or with fewer images
  - annotations of T do not necessarily overlap with S
- We can use the model CNN_S to learn a better CNN_T
- This is transfer learning

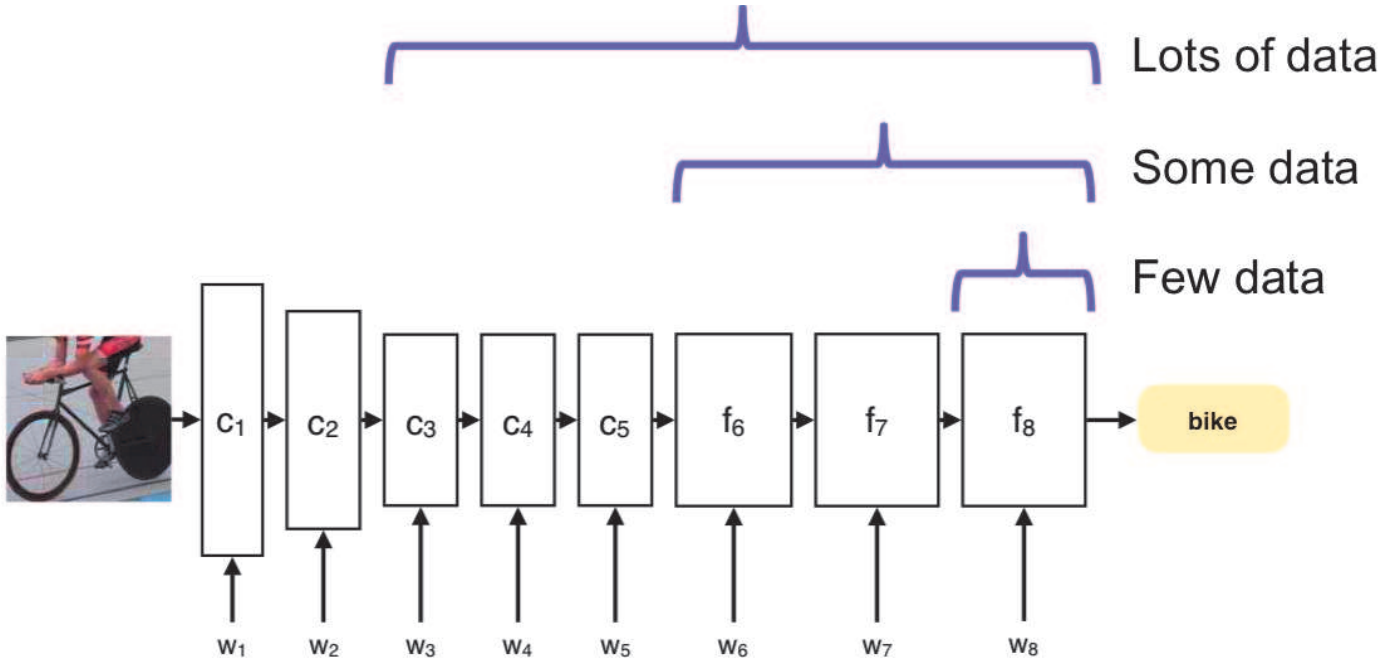
Transfer learning

- Even if our dataset T is not large, we can train a CNN for it
- Pre-train a CNN on the dataset S
- The we can do:
  - fine-tuning
  - use CNN as feature extractor

Fine-tuning

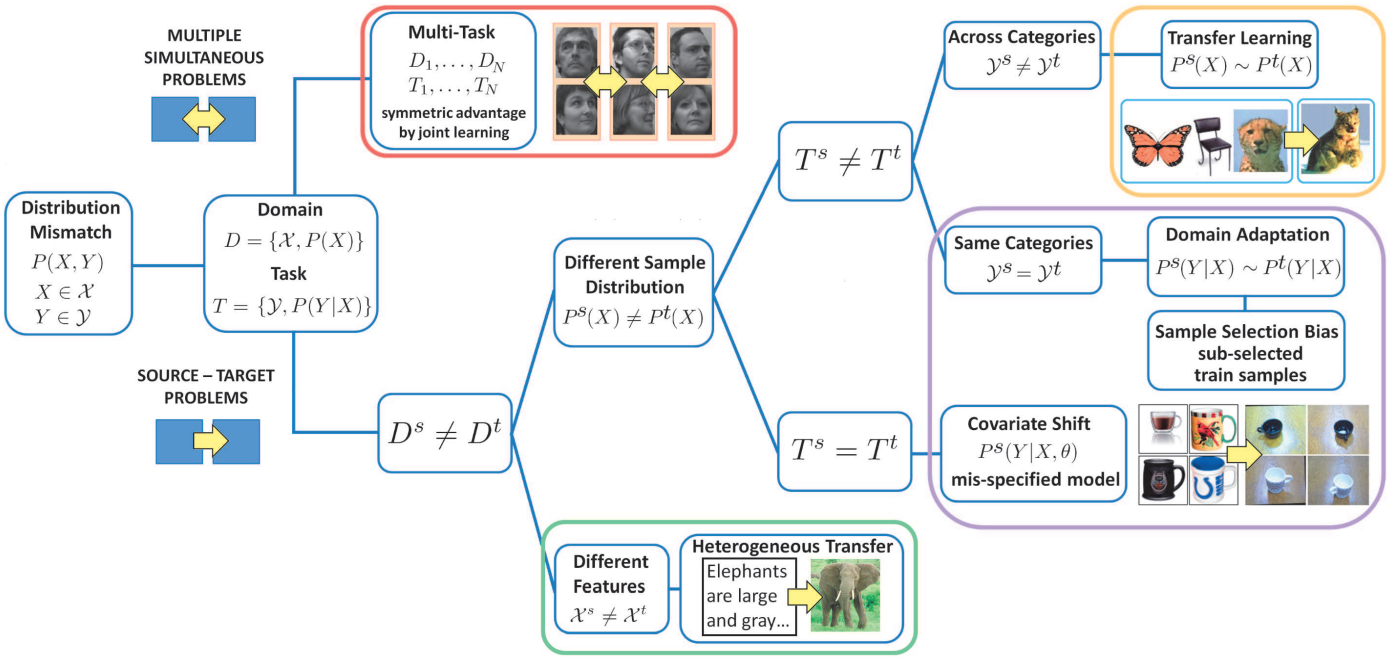
- Assume the parameters of CNN_S are already a good start near our final local optimum
- Use them as the initial parameters for our new CNN for the target dataset
- This is a good solution when the dataset T is relatively big
    - e.g. for Imagenet S with 1M images, T with a few thousands images

Fine-tuning



- Depending on the size of T decide which layer to freeze and which to finetune/replace
- Use lower learning rate when fine-tuning: about $\frac{1}{10}$ of original learning rate
  - for new layers use agressive learning rate
- If S and T are very similar,fine-tune only fully-connected layers

Transfer learning



(Tommasi, PhD thesis, 2012)