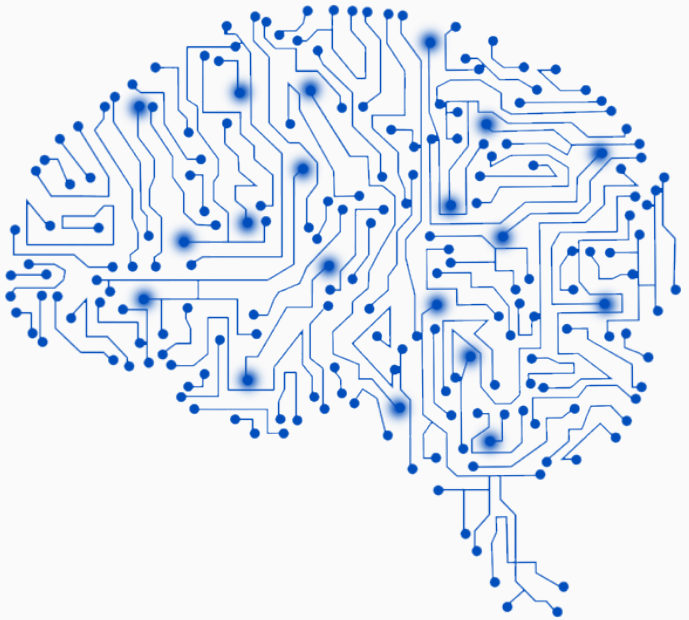
Course VI – Deep Neural Networks

Bruno Galerne

Friday January 26, 2024



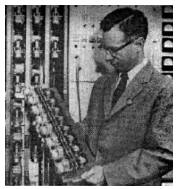
Credits

Most of the slides from **Charles Deledalle's** course "UCSD ECE285 Machine learning for image processing" (30 $\times$ 50 minutes course)

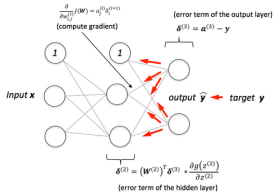
`www.charles-deledalle.fr/`

`https://www.charles-deledalle.fr/pages/teaching.php#learning`

Timeline of (deep) learning



1958 Perceptron



1974 Backpropagation

Convolution Neural Networks for Handwritten Recognition

1998

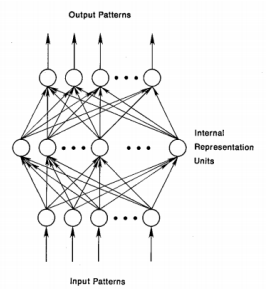
Google Brain Project on 16k Cores

2012

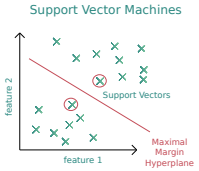
awkward silence (AI winter)

1969 Perceptrons book

~1980 Multilayer network



1995 SVM reigns

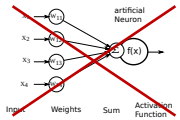


2006 Restricted Boltzmann Machine

2012 AlexNet wins ImageNet

IMAGENET

Perceptron criticized



(Source: Lucas Masuch & Vincent Lepetit)

Modern deep learning – Recipes

What has changed again?

Modern deep learning makes use of several tools and heuristics for training large architectures: type of units, normalization, optimization...

1. Fight vanishing/exploding gradients

   $\longrightarrow$ Rectifiers, Gradient clipping.

2. Improve accuracy by using tons of data thanks to
   - Stochastic gradient descent (and variants),
   - GPUs, parallelization, distributed computing, ...

3. Fight poor solutions (saddle points) and over-fitting

   $\longrightarrow$ Early stopping, Dropout, Batch normalization, Regularization, ...

4. Multitude of open source solutions.

Disclaimers

The field evolves so quickly that parts of what follows are already outdated...

Many recent tools, sometimes less than a year old, become the new standard just because they work better.

Why do they work better? The answer is usually based on intuition, and rigorous answers are often lacking.

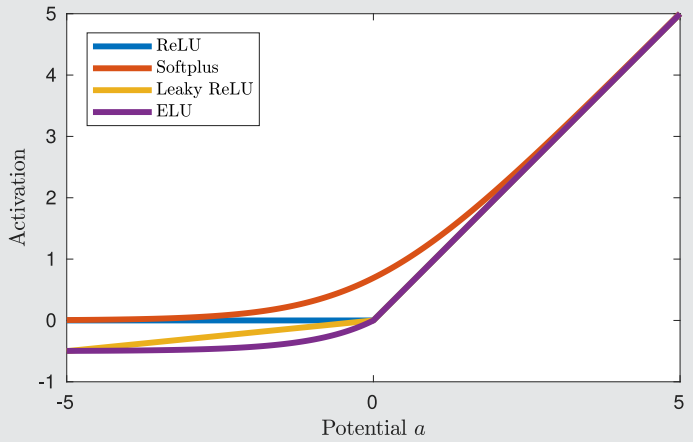
Rectifier (Glorot, Bordes and Bengio, 2011)

- **Goal:** Avoid the gradient vanishing problem.
- **How:** make sure $g'(a_j)$ is not close to zero for all training points, this was often arising with hyperbolic tangents or sigmoids.

- Rectifier linear unit (ReLU):

$$g(a) = \max(a, 0)$$

- Softplus: $g(a) = \log(1 + \exp a)$
- Leaky ReLU: $g(a) = \max(a, 0.1a)$
- Exponential LU (ELU): $g(a) = \begin{cases} a & \text{if} \quad a \geqslant 0 \\ \alpha(e^a - 1) & \text{otherwise} \end{cases}$



In a randomly initialized network, about 50% of hidden units are activated. It also promotes sparsity of hidden units (structured/organized features).

Gradient clipping (Mikolov, 2012; Pascanu et al., 2012)

- **Goal:** Avoid the gradient exploding problem.
- **How:** make sure $\nabla E(\boldsymbol{W})$ does not get too large.

- Cliffs cause the learning process to overshoot a desirable minimum.
- Preserve the direction but prevent extreme values:

$$\text{If } \|\nabla E(\boldsymbol{W}^t)\| \leqslant \tau\text{:} \qquad \boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma \nabla E(\boldsymbol{W}^t)$$

$$\text{Else:} \qquad \boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma\tau \frac{\nabla E(\boldsymbol{W}^t)}{\|\nabla E(\boldsymbol{W}^t)\|}$$

- Pick threshold τ by averaging the norm over a large number of updates.
- New solution $\boldsymbol{W}^{t+1}$ stays in a ball of radius τ around the current solution.

Stochastic gradient descent

Reminder: (Batch) gradient descent with backprop

1. For each training sample
   - Perform backprop,
   - Sum their gradients.
2. Update the weights (based on the gradient over the whole *batch*).

Go to Step 1

→ Convergence to a stationary point under some technical assumptions.

- Requires the loading of the whole dataset into memory (big data sets),
- Can't be efficiently parallelized or used online,
- Scanning the whole training set is slow (*epoch*),
- *Epoch:* a scan in which each training sample has been visited at least once.
- After 10 epochs, the weights have been updated only 10 times,
- Designed to fall into local minima.

Stochastic gradient descent

Stochastic gradient descent (SGD) **(Robbins and Monro, 1951)**

1. For each training sample (shuffled randomly – thus "stochastic")
   - Perform backprop,
   - Evaluate its gradient,
   - Update the weights (based only on that sample).

   Go to Step 1

→ Convergence under technical assumptions with high probability.

- After 10 epochs, the weights have been updated $10 \times N$ times (N size of the training set),
- Faster if the gradient of each sample is a good proxy for the total gradient,
- Can get out of local minima as it explores the space in a random fashion,
- It allows online learning,
- But it is noisy, unstable and can still get trapped into local minima.

- Origin: ADALINE (Widrow & Hoff, 1960).

Stochastic gradient descent

Mini-batch gradient descent

1. Shuffle: create a random partition of subsets of size b (*mini-batches*)
2. For each mini-batch
   - Perform backprop and accumulate the gradients,
   - Update the weights (based on the gradient of the *mini-batch*)

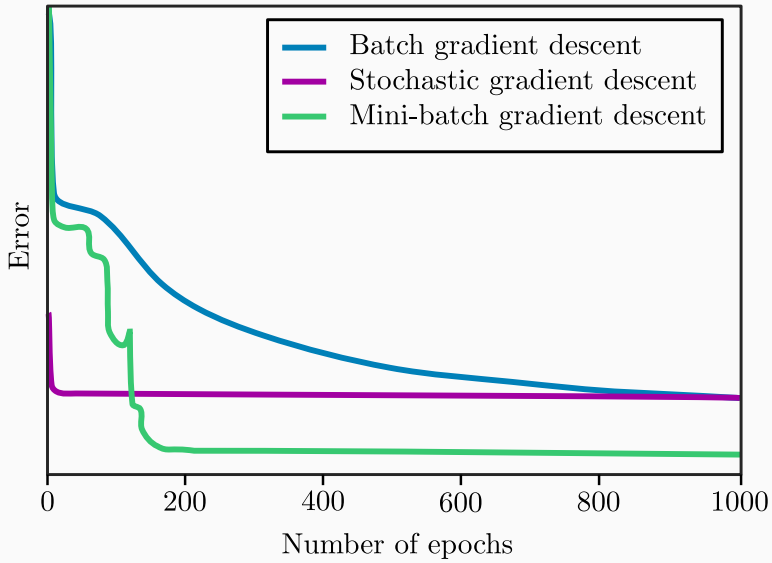
Go to Step 1

$\rightarrow$ Convergence under technical assumptions with high probability.

- Usually, a mini-batch contains around 50 to 256 data points,
- After 10 epochs, the weights have been updated $10 \times N/b$ times
- Smoothes the noise, then provides a better proxy for the total gradient,
- Can still get out of some local minima,
- Trade-off between batch GD and SGD (stability vs. speed)
- Enjoys highly optimized matrix manipulation tools.

Remark: Nowadays, SGD often referred to as mini-batch gradient descent.

Stochastic gradient descent



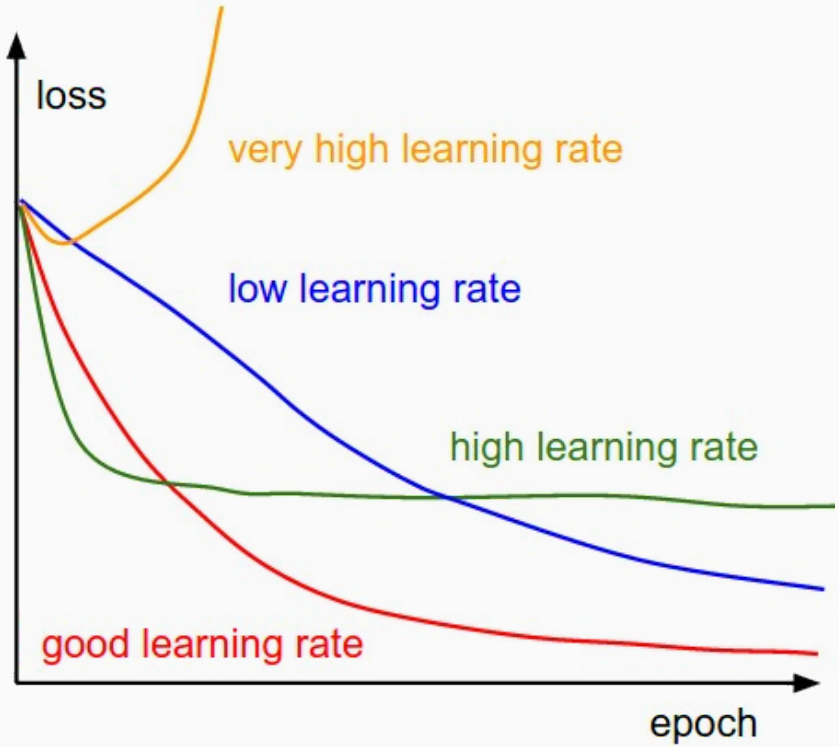
Mini-batch gradient descent:

- Not only faster,
- Usually finds better solutions.

Because randomness helps explore new configurations,
and mini-batches favor more relevant configurations.

But make sure to adjust the learning rates.

Learning rate (reminder)



$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^{t} - \gamma \nabla E(\boldsymbol{W}^{t})$$

Learning rate schedules

Learning rate schedules: reduce γ at each epoch t

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma^t \nabla E(\boldsymbol{W}^t) \quad \text{with} \quad \lim_{t \to \infty} \gamma^t = 0$$

- Time-based decay: $\gamma^t = \frac{1}{1+\kappa t}\gamma^0, \kappa > 0$ (*aka*, search-then-converge)
- Step decay: $\gamma^t = \kappa^t \gamma^0, 0 < \kappa < 1$ (*aka*, drop decay)
- Exponential decay: $\gamma^t = \exp(-\kappa t)\gamma^0, \kappa > 0$

- Guaranteed to converge (avoid SGD from oscillating forever),
- But may stop too early (too fast),
- Connections with simulated annealing (γ^t seen as temperature),
  - $\rightarrow$ Hence, often referred to as annealing.
    - Large γ^t: explore the space (allows uphill jump),
    - Small γ^t: move downhill towards the best local solution.
- Variant: restart schedule from the best iterate (achieving smaller loss).

Adaptive learning rates

Adaptive learning rates: adapt γ during the iteration t independently for each connection w_{ij}

$$w_{ij}^{t+1} \leftarrow w_{ij}^{t} - \gamma_{ij}^{t} \frac{\partial E(\boldsymbol{W}^t)}{\partial w_{i,j}^{t}}$$

Current standard: Adam (Kingma & Ba, 2015) (more than 60 K citations!)

Momentum and Nesterov accelerated gradient (NAG)

Idea: denoise gradients (in SGD) by using inertia, memory of velocity

(Stochastic) gradient descent (for each mini-batch)

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma \nabla E(\boldsymbol{W}^t)$$

Momentum – introduce velocity $\boldsymbol{V}^t$ (parameter $\rho \approx .9$)

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \underbrace{\left(\gamma \nabla E(\boldsymbol{W}^t) + \rho \boldsymbol{V}^t\right)}_{=\boldsymbol{V}^{t+1}}$$

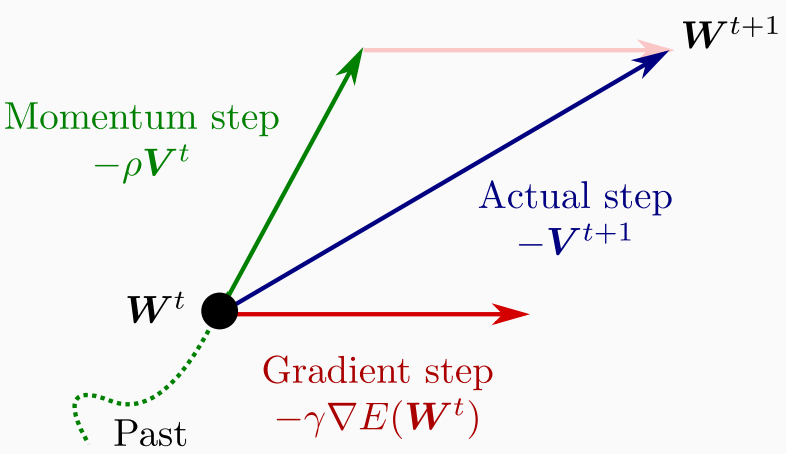
Nesterov acceleration (1983) (in deep community: Sutskever *et al.*, 2013)

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \underbrace{\left(\gamma \nabla E(\boldsymbol{W}^t - \rho \boldsymbol{V}^t) + \rho \boldsymbol{V}^t\right)}_{=\boldsymbol{V}^{t+1}}$$

Nice illustrations: `https://distill.pub/2017/momentum/`

Momentum and Nesterov accelerated gradient (NAG)

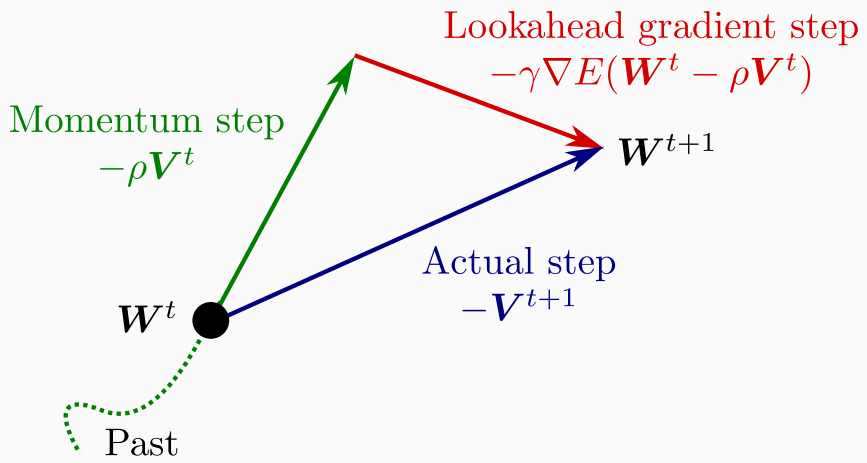
Momentum update



$$W^{t+1} \leftarrow W^t - \underbrace{\left(\gamma \nabla E(W^t) + \rho V^t\right)}_{=V^{t+1}}$$

(Correction of the gradient)

Nesterov acceleration update



$$W^{t+1} \leftarrow W^t - \underbrace{\left(\gamma \nabla E(W^t - \rho V^t) + \rho V^t\right)}_{=V^{t+1}}$$

(Correction of the momentum)

In convex and batch settings, NAG decays in $O(1/t^2)$ instead of $O(1/t)$.
For deep learning, dramatic acceleration observed in practice.
It also improves regularity of SGD.

Weight decay

Goal: prevent from exploding weights, limit the freedom, avoid over-fitting.

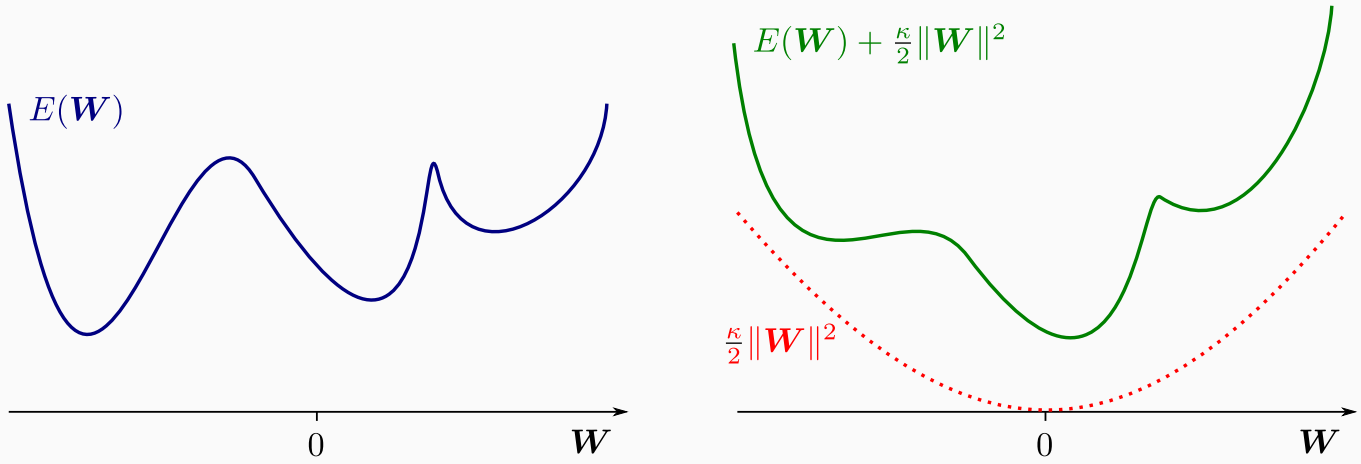
Weight decay:

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma(\nabla E(\boldsymbol{W}^t) + \kappa \boldsymbol{W}^t)$$

It corresponds to adding an ℓ_2 penalty to the loss

$$\min_{\boldsymbol{W}} E(\boldsymbol{W}) + \frac{\kappa}{2}\|\boldsymbol{W}\|_F^2$$

and has the effect of shrinking the weights towards zero.



Gradient noise (Neelakantan, 2015)

If a little bit of noise is good, what about adding noise to the gradient?

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma(\nabla E(\boldsymbol{W}^t) + \mathcal{N}(0, \sigma_t^2)) \quad \text{with} \quad \sigma_t = \frac{\sigma_0}{(1+t)^\kappa}$$

Works surprisingly great in many situations!

All together

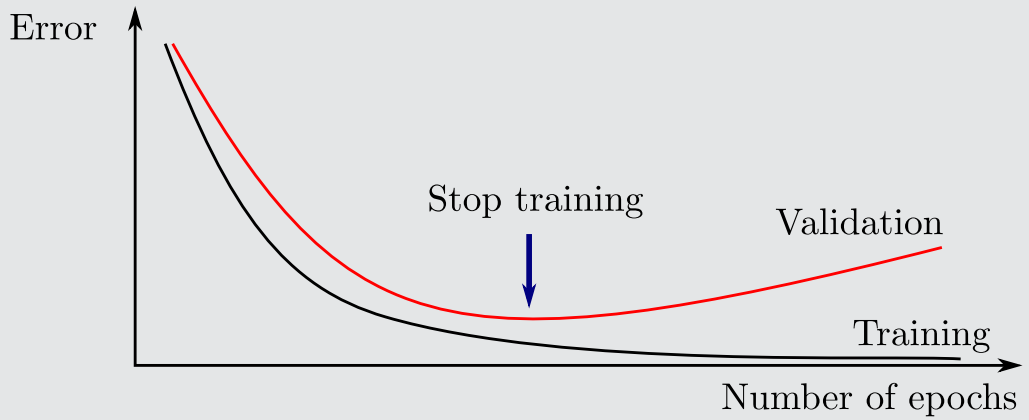
Nesterov + Adaptive learning rates + Weight decay + Gradient noise:

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \underbrace{(\gamma^t(\nabla E(\boldsymbol{W}^t - \rho\boldsymbol{V}^t) + \kappa(\boldsymbol{W}^t - \rho\boldsymbol{V}^t) + \mathcal{N}(0, {\sigma_t}^2)) + \rho\boldsymbol{V}^t)}_{=\boldsymbol{V}^{t+1}}$$

Good luck to tune all the parameters jointly!
Luckily, many toolboxes offer good default settings.

Early stopping (Girosi, Jones and Poggio, 1995)

- **Goal:** Reduce over-fitting.
- **How:** Stop the iterations of gradient descent before convergence.



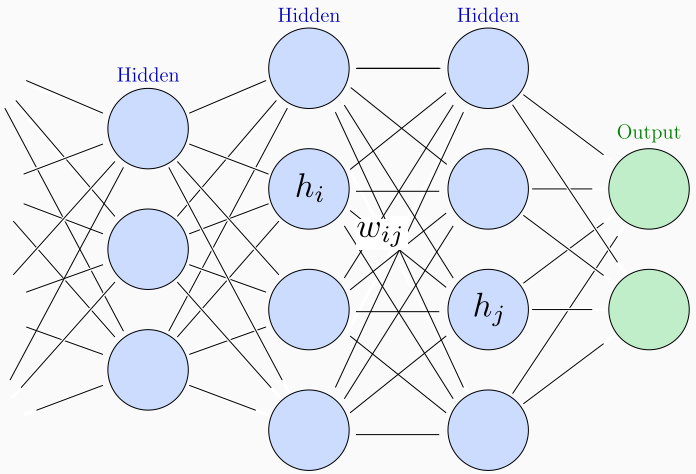
1. Split the training set into a training subset and a validation subset,
2. Train only on the training subset,
3. Evaluate the error on the validation subset every few epochs,
4. Stop training as soon as the error on the validation starts increasing.

Normalized initialization

Idea: A good weight initialization should be random

$$w_{ij} \sim \underbrace{\mathcal{N}(0, \sigma^2)}_{\text{Gaussian}} \quad \text{or} \quad w_{ij} \sim \underbrace{\mathcal{U}\left(-\sqrt{3}\sigma, \sqrt{3}\sigma\right)}_{\text{Uniform}}$$

with standard deviation σ chosen such that all (hidden) outputs h_j have unit variance (during the first step of backprop).



$$\mathrm{Var}[h_j] = \mathrm{Var}\left[g\left(\sum_i w_{ij} h_i\right)\right] = 1$$

Randomness:
prevents units from learning the same thing.

Normalization: preserves dynamic of the outputs and gradients at each layer:
$\rightarrow$ all units get updated with similar speed.

Normalized initialization

1. Normalize the inputs with about zero-mean and unit variance,
2. Initialize all the biases to zero,
3. For the weights w_{ij} of a unit j with N inputs and activation g:
   - **Xavier initialization** (Glorot & Bengio, 2010): for $g(a) = a$

$$\mathrm{Var}\left[g\left(\sum_i w_{ij} h_i\right)\right] = \sum_i \sigma^2 \underbrace{\mathrm{Var}\left[h_i\right]}_{=1} = 1 \quad \Rightarrow \quad \sigma = \frac{1}{\sqrt{N}}$$

   (Works also for $g(a) = \tanh(a)$.)

   - **He initialization** (He et al., 2015): for ReLU $\quad \rightarrow \quad \sigma = \sqrt{\frac{2}{N}}$

   (First time, a machine surpasses humans on ImageNet.)

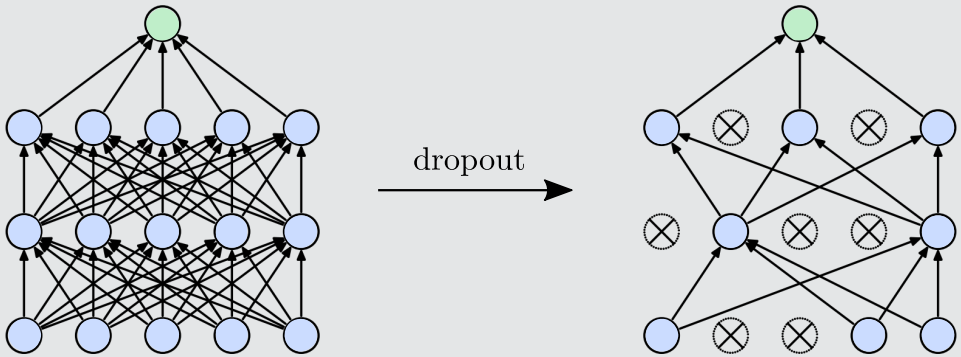
   - **Kumar initialization** (Kumar, 2017): for any g differentiable at 0:

$$\sigma^2 = \frac{1}{N g'(0)^2 (1 + g(0)^2)}$$

Dropout (Hinton *et al.*, 2012)

- **Goal:** Reduce over-fitting.
- **How:** Train smaller networks.

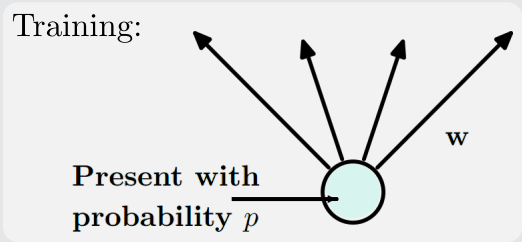
- Build a sub-network for each training sample,

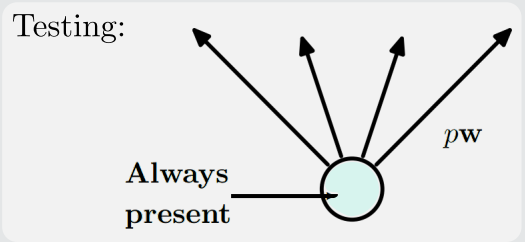


- Drop nodes with probability $1 - p$ or keep them with probability p,
  (in practice use a random binary mask to disable entries of $\boldsymbol{W}$).
- Usually, $p = .5$ for hidden layers, and $p \approx 1$ for the input.

Dropout (Hinton *et al.*, 2012)

- Use back-propagation for each data point and its sub-network,
- Note that each sub-network within a mini-batch shares common weights,
- At test time, dropout is disabled and unit outputs are multiplied by their corresponding p such that all units behave as if trained without dropout.





- Prevents units from co-adapting too much (depend on each other),
- Allows the network to learn more robust and specific features,
- Converges faster, and each epoch costs less.

Batch normalization (Ioffe and Szegedy, 2015)

- **Goal:** Learn faster, learn better.
- **How:** Normalize inputs of each hidden layer.

Internal Covariate Shift

- With backprop, the distribution of the inputs $\boldsymbol{h}$ of a (deep) hidden layer keeps changing during the iterations.
- This is painful as the layers need to continuously adapt to it, then
  - Requires low learning rates,
  - Careful parameter initialization,
  - Proper choices of activation functions,
  - Leads to slow convergence, and poor solutions.

Batch normalization (Ioffe and Szegedy, 2015)

Standardization

- Transform $\boldsymbol{a}$ (recall, $\boldsymbol{a} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$) to satisfy
  - **Zero-mean:** remove arbitrary shift of your data,
  - **Unit variance:** remove arbitrary spread of your data,

- Standardization dates back to classical ML with hand-crafted features,
- Classically applied to the input features,
- Here features at each (hidden) layer at each epoch are normalized.

Batch normalization (Ioffe and Szegedy, 2015)

Idea: perform for each unit i of a given hidden layer

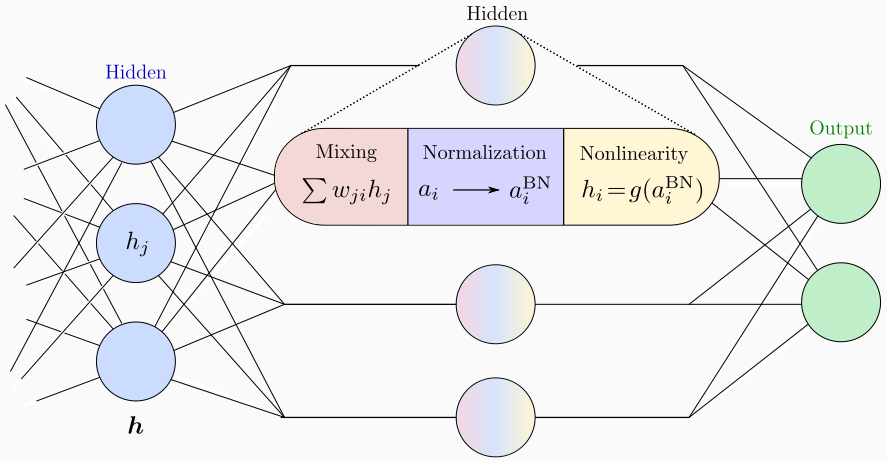
$$a_i^{\text{BN}} \leftarrow \beta_i + \alpha_i \underbrace{\frac{a_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}}_{\text{standardized input}}$$

- μ_i, σ_i: mean and standard deviation of a_i over the current mini-batch, (obtained during the forward step)
- α_i, β_i: scalar parameters learned by backprop, (updated during the backward step)
- $\varepsilon > 0$: regularization to avoid instabilities.

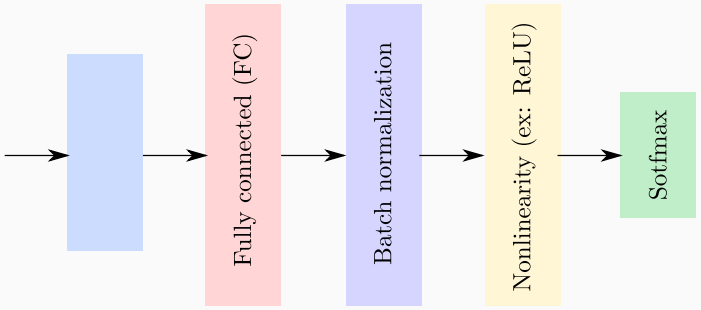
α_i and β_i introduced to preserve the network capacity.
(to make use of non-linearities, otherwise sigmoid $\approx$ linear)

During testing, replace μ_i and σ_i by an average of the last estimates obtained over mini-batches during training.

Batch normalization (Ioffe and Szegedy, 2015)



Old school representation



Modern representation

- α_i and β_i encode the range of inputs given to the activation function, (can be seen as parameters of the activation function)
- each layer learns a little bit more by itself independently from other layers,
- allows higher learning rates (larger gradient descent step size γ),
- reduce over-fitting – slight regularization effects.

Residual Networks (ResNets) (Microsoft – He *et al.*, 2015).

Up to 2015, deep was about ≈ 20 layers, and using more was harmful.

Goal: Get deeper and deeper while improving accuracy.

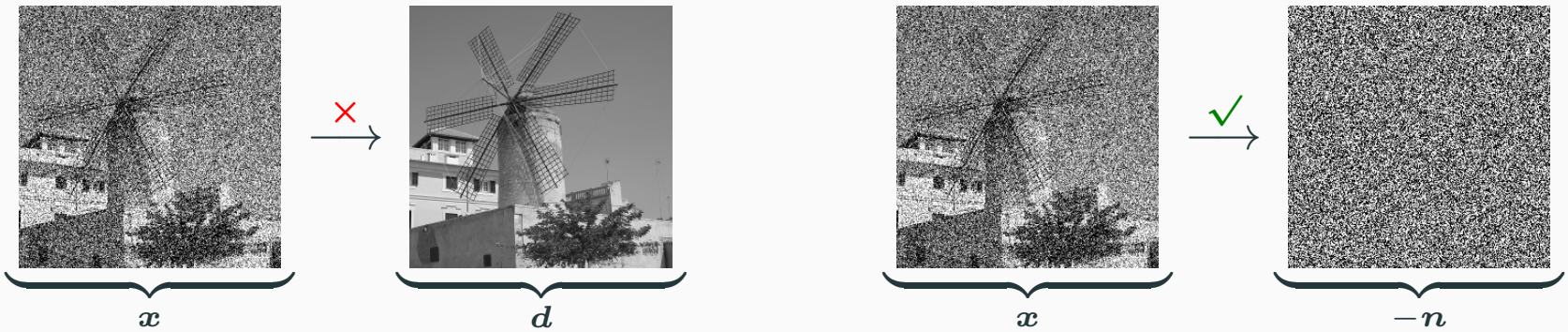
Idea: Learn $f : \boldsymbol{x} \to \underbrace{\boldsymbol{d} - \boldsymbol{x}}_{\text{residual}}$ instead of $h : \boldsymbol{x} \to \boldsymbol{d}$ (assuming same size), then give your prediction of $\boldsymbol{d}$ as $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{x}$

- Tough to learn a mapping close to the identity with non-linear layers:
  - Consider a 1 ReLU unit:
    ❶ $y = \underbrace{\max(0, wx + b)}_{h(x)}$
    ❷ $y = \underbrace{\max(0, wx + b)}_{f(x)} + x.$
  - Learn $y = x$: trivial for ❷, impossible for ❶.
  - Of course, possible with more units but less trivial.
- Simpler to learn what the fluctuations around x should be.

Residual Networks (ResNets)

Example (Denoising $x = d + n$ with d the desired clean image)

- The residual is (minus) the noise component: $d - x = -n$,
- subtle difference between the input $x = d + n$ and output d
  $\rightarrow$ harder to learn.
- large difference between the input $x = d + n$ and the residual $-n$
  $\rightarrow$ easier to learn.



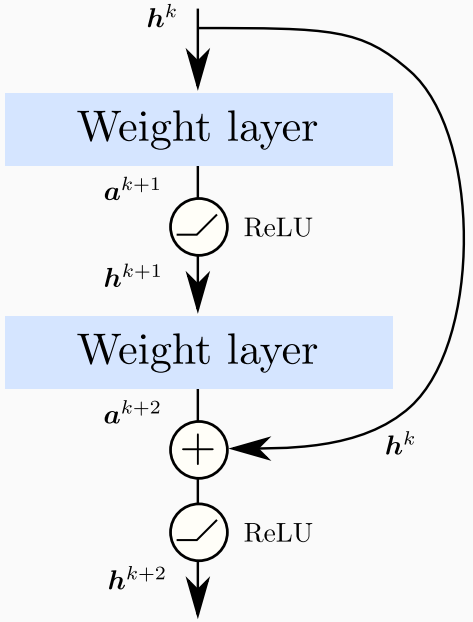
But why does it help me to be deeper?

Residual Networks (ResNets)

- Don't necessarily use this concept only for the input and output layers,
- Instead, use successions of hidden layers with same size,
- Use it every two hidden layers:

$$\begin{aligned} \boldsymbol{a}^{k+1} &= \boldsymbol{W}^k \boldsymbol{h}^k \\ \boldsymbol{h}^{k+1} &= g(\boldsymbol{a}^{k+1}) \\ \boldsymbol{a}^{k+2} &= \boldsymbol{W}^{k+1} \boldsymbol{h}^{k+1} \\ \boldsymbol{h}^{k+2} &= g(\boldsymbol{a}^{k+2} + \boldsymbol{h}^k) \end{aligned}$$

- Use shortcut connections:



- It does not require extra parameters,
- Does not add computational overhead,
- Acts as a kind of preconditioning
  $\longrightarrow$ allowing for huge accelerations,
- Led to a 152 deep NN that won the 2015 ImageNet competition,
  $\longrightarrow$ we will return to this next class.

Regularization

- **Goal:** Avoid over-fitting.
- **How:** Use prior knowledge.

Early stopping, dropout, batch normalization, *etc.*, implicitly promote some regularity. But we can also introduce regularity explicitly given prior knowledge on the problem/application context.

Remember: autoencoders extract relevant features by making use of

- Undercomplete representation (remove nodes):

  information can be compressed/encoded with a small dictionary,

- Overcomplete/sparse representation (penalize node's outputs):

  information can be encoded with a few atoms of a large dictionary.

In both cases, we have used prior knowledge that information must be structured/organized.

Regularization

In general, there exists a lot more of intrinsic regularity in the data that should be injected in the model (all the more for images).

For instance, as for the nodes, depending on the application, some connections can be removed (non-fully connected layer), or their weights can be penalized by adding a regularization term Ω in the loss function:

- $\Omega = \ell_1$ to promote sparsity (sparse layer),
- $\Omega = \ell_2$ to promote small coefficients (weight decay),
- ...

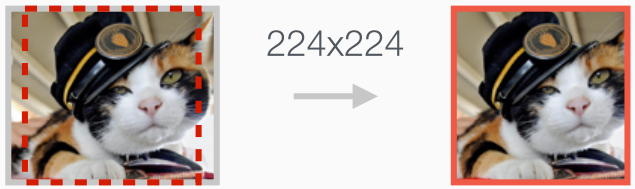
As we will see next class, Convolution Neural Networks (CNN) are a kind of highly constrained architecture very relevant for images.

Regularity is probably the most important ingredient allowing for deep learning.

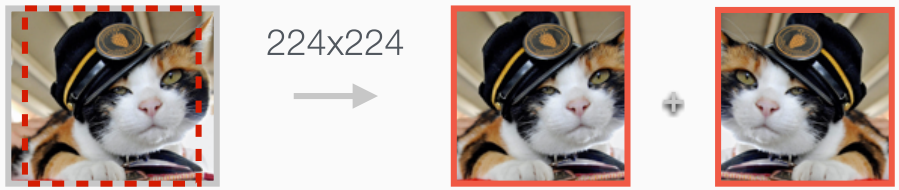
Data augmentation

Idea: generate artificially more data points from the training set itself.

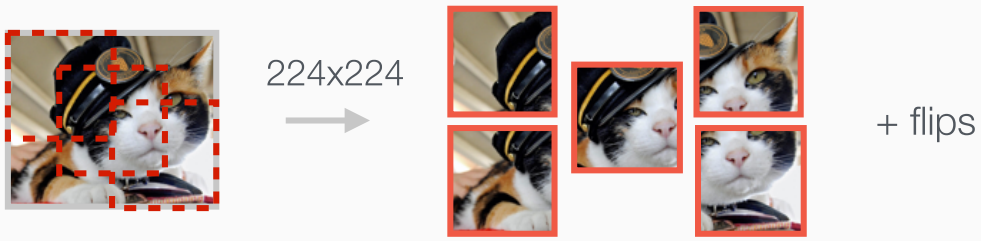
No augmentation (=1 image)



Flip augmentation (=2 images)



Crop + flip augmentation (=10 images)



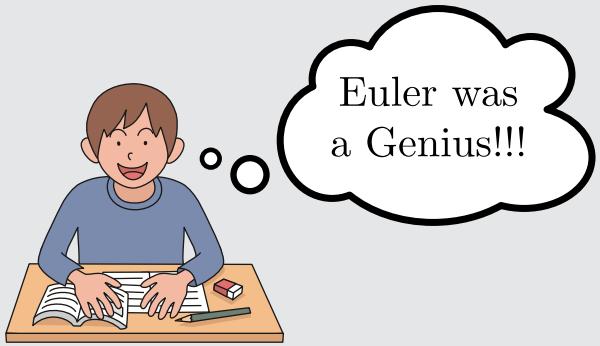
Others: rotations, zooms (rescalings), brightness, noise, ...

(Source: Ken Chatfield)

Curriculum learning (Bengio et al, 2009)

- **Convex criteria:** the order of presentation of samples should not matter to the convergence point, but could influence convergence speed.
- **Non-convex criteria:** the order and selection of samples could yield a better solution.

1. Integers
2. Real numbers
3. Complex numbers
4. $e^{i\pi} = -1$

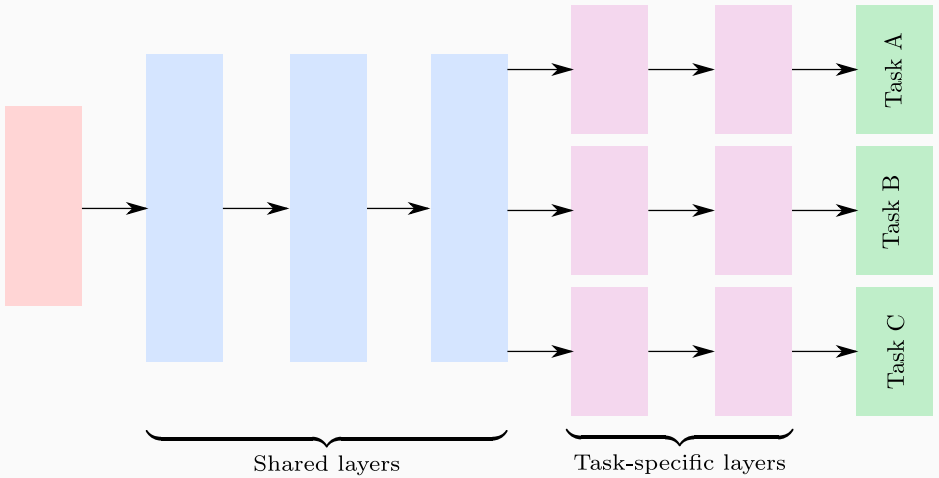


(Inspiration: Pawan Kumar)

- Start learning on easy samples, and increase the difficulty step by step.
- Improves but requires to sort the training samples by difficulty.
- When you cannot or do not know how, shuffling randomly is safer.

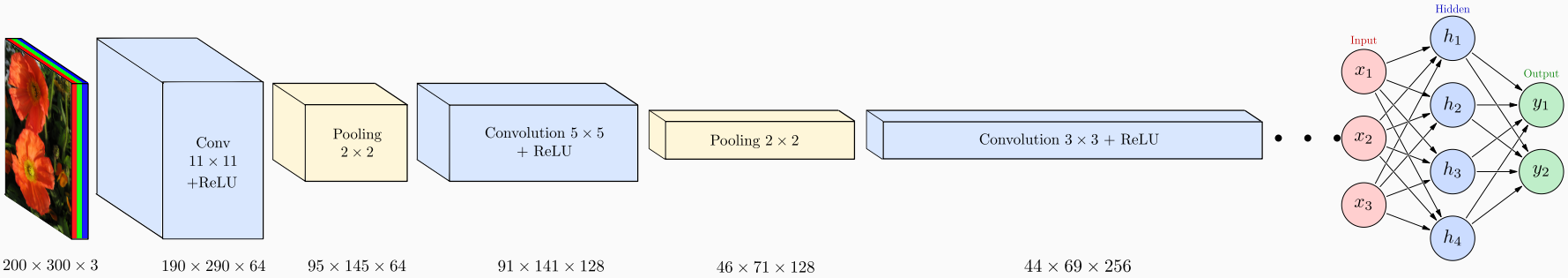
Multitask learning (Caruana, 1997)

- Learning jointly how to solve several tasks improves performance,
- Effect of regularization (avoid over-fitting),
- Useful in context of insufficiently labeled data.



Share first hidden layers, use specific ones last.

Transfer learning (Pratt, 1993, 1997)



1. Train first on a given large dataset (eg detect red cars), or use a publicly available model.
2. Fine-tune using specific small dataset (eg detect tumors):
   - Redefine classifier part.
   - Train the new classifier.
   - Fine-tune all other parameters or keep the one before.

Open source solutions

Most popular

- **TensorFlow** – Google – Python, C/C++
- **Keras** high level interface for TensorFlow
- **PyTorch** – Facebook – Python

Main features

- Implementations of most classical DL models, tools, recipes...
- Pretrained models,
- GPU/CUDA support,
- Flexible interfaces,
- Automatic differentiation.

(Multi) GPUs architectures

- Neural Networks are inherently parallel,
- GPUs are good at matrix/tensor operations,
- Much faster than CPU, lower power, lower cost,
- New architectures are optimized for deep learning:

NVIDIA Tesla V100: World's first GPU to break the 100 teraflops barrier of deep learning performance. New feature: 640 tensor cores. Relatively cheap ($\approx$ \$8,000). Released June 2017.

- Relatively easy to manipulate thanks to CUDA,
- Many deep learning toolboxes have GPU support,
- Can be integrated in clusters/datacenters for distributed computing.

Best practices

- Check/clean your data: remove corrupted ones, perform normalization, ...
- Shuffle the training samples,
- Split your data into training (70%) and testing (30%),
- Use early stopping: take another 30% of training for validation,
- Never train on test data,
- Start with an existing network and adapt it to your problem,
- Start smallish, keep adding layers and nodes until you overfit too much,
- Check that you can achieve zero loss on a tiny subset (20 examples),
- Track the loss during training,
- Track the first-layer features,
- Try the successive rates: $\gamma = 0.3, 0.1, 0.03, 0.01, 0.003, 0.001$.

Recommended reading:

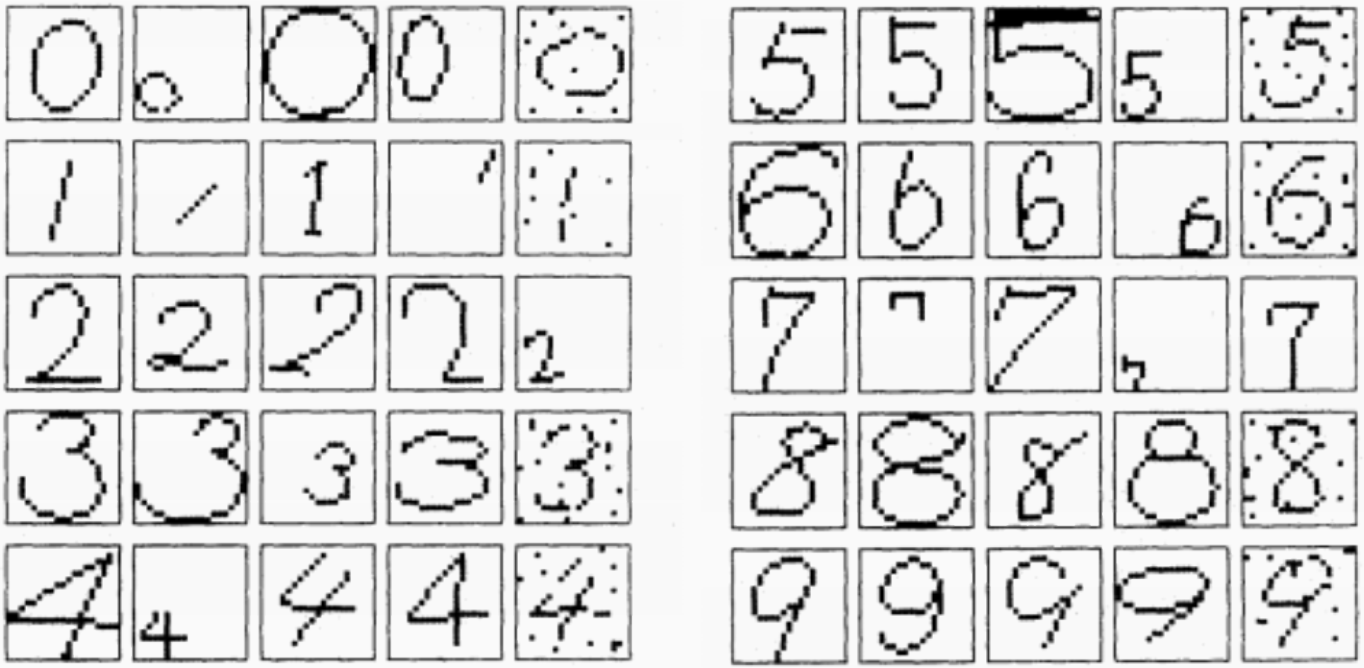
`https://karpathy.github.io/2019/04/25/recipe/`

Successful CNNs architectures

Neocognitron (Fukushima, 1979, 1980, 1987)

In the history of Optical Character Recognition (OCR), first system working

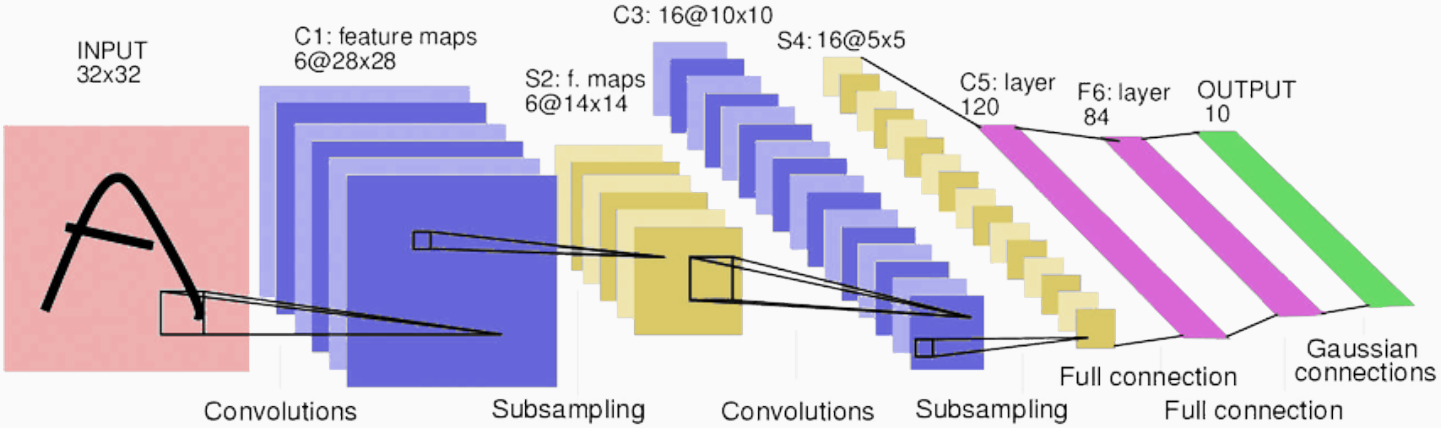
- regardless of where the digits were placed in the field of view,
- high degree of tolerance to distortion of the character,
- fairly insensitive to the size of the character.



Some digits correctly recognized

Successful CNNs architectures

LeNet-5 (LeCun, Bottou, Bengio and P. Haffner, 1998)



- Classifier for 32×32 hand-written digits,
- Was used for reading $\approx 10\%$ of the checks in North America,
- Demo: `http://yann.lecun.com/exdb/lenet/`,
- Used hyperbolic tangent connection after convolutions,
- Plugged a linear classifier using Gaussian connections (RBF kernels):

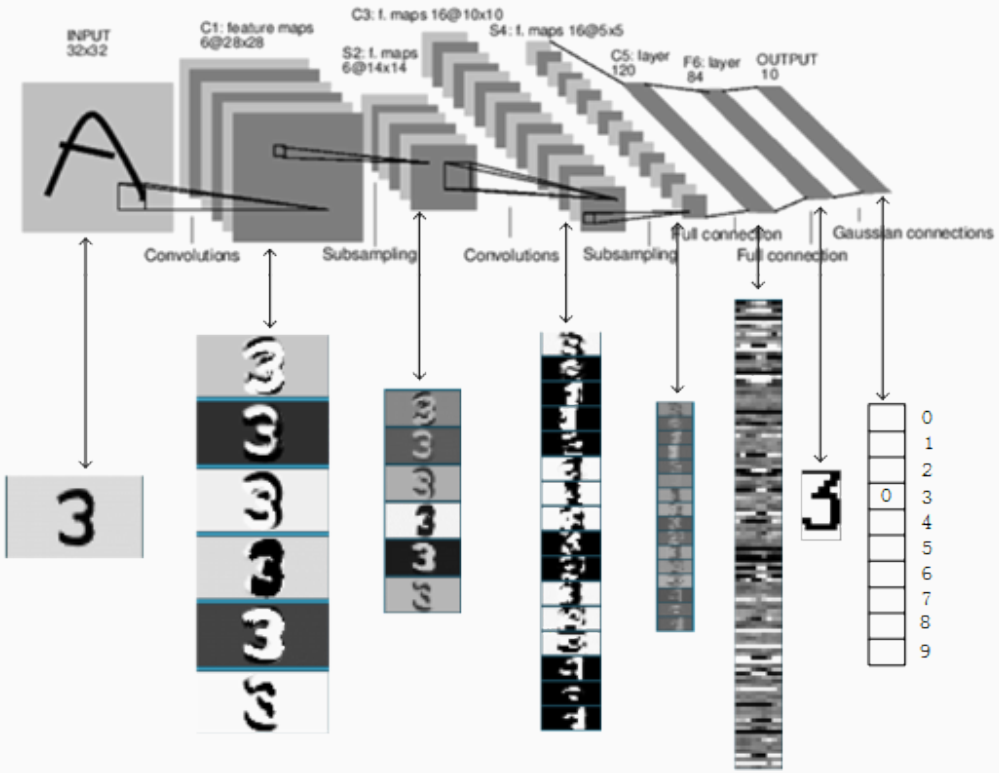
$$y_j = \sum_i (h_i - w_{ij})^2 \text{ instead of classical dot-products } y_j = \sum_i w_{ij} h_i.$$

First deep architecture (5 layers) that was successfully trained.

LeNet-5 (LeCun, Bottou, Bengio and P. Haffner, 1998)

- Trained on MNIST:
  - 28×28 images,
  - 60,000 for training,
  - 10,000 for testing.
- Used data augmentation:
  - generated random distortions,
  - extended to 32×32px.
  - 600,000 training samples,
- Backprop with SGD:
  - only 20 epochs,
  - no mini-batches,
  - 60,000 parameters,
  - about 3 days (at that time).

LeNet-5 (LeCun, Bottou, Bengio and P. Haffner, 1998)



What did it learn? Look at the evolution of the features in the network.

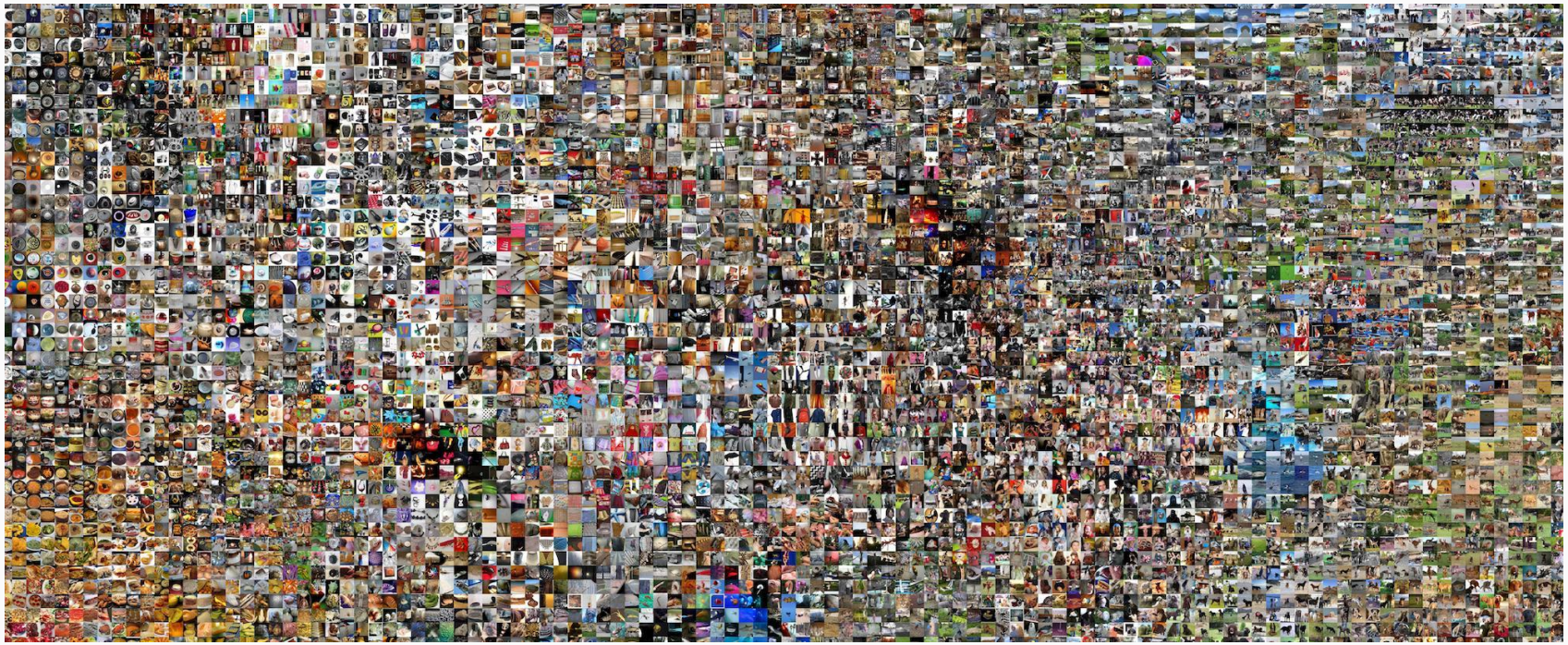
Successful CNNs architectures

LeNet-5 (LeCun, Bottou, Bengio and P. Haffner, 1998)

Made only 82 errors among the 10,0000 samples of the testing dataset.
Nowadays, the best results on MNIST are of 23 errors (Cireșan *et. al*, 2012).

But with the proliferation of multimedia, MNIST challenge became outdated.

ImageNet



ImageNet:

- 12 million labeled images,
- 22,000 classes,
- Labeled by crowd-sourcing (Amazon Mechanical Turk).

ImageNet challenge (ILSVRC)

ILSVRC:

- Annual challenges since 2010,
- Limited to 1,000 classes,
- 1.2 million of 256×256 images for training,
- 50,000 images for validation and 100,000 for testing.

ImageNet classification challenge (ILSVRC)

Ranking: top-5 error

Make five guesses – Correct if the desired label is within these guesses.

top-5 error rate: percentage of incorrect such answers.

why? desired labels can be ambiguous

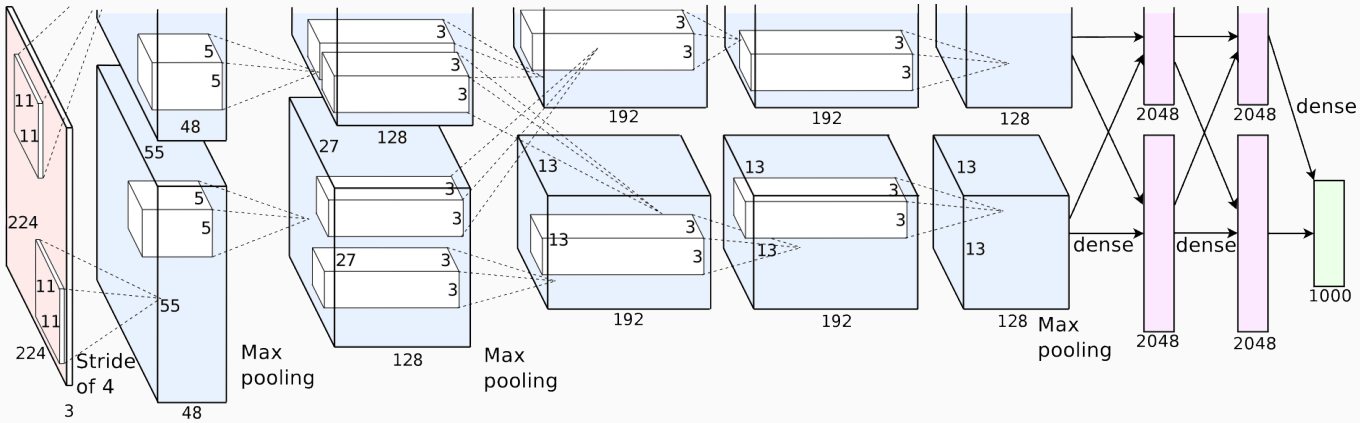
(a) Siberian husky

(b) Eskimo dog

And the 2012 winner was...

Successful CNNs architectures

AlexNet (Krizhevsky, Sutskever, Hinton, 2012)



- Similar to LeNet-5, a bit deeper (8 layers), but much larger.
- Use ReLU after convolutions, use softmax for the output layer.

Introduced concepts:

- **Specific GPU architecture:** split the channels in two stages in order to be trained simultaneously on two Nvidia Geforce GTX 580 GPUs.
- **Local Response Normalization:** after applying the first 2 filter banks, they normalize each feature map with respect to their adjacent ones.
- **Overlapping pooling:** use striding also after pooling with $z = 3$ and $s = 2$.

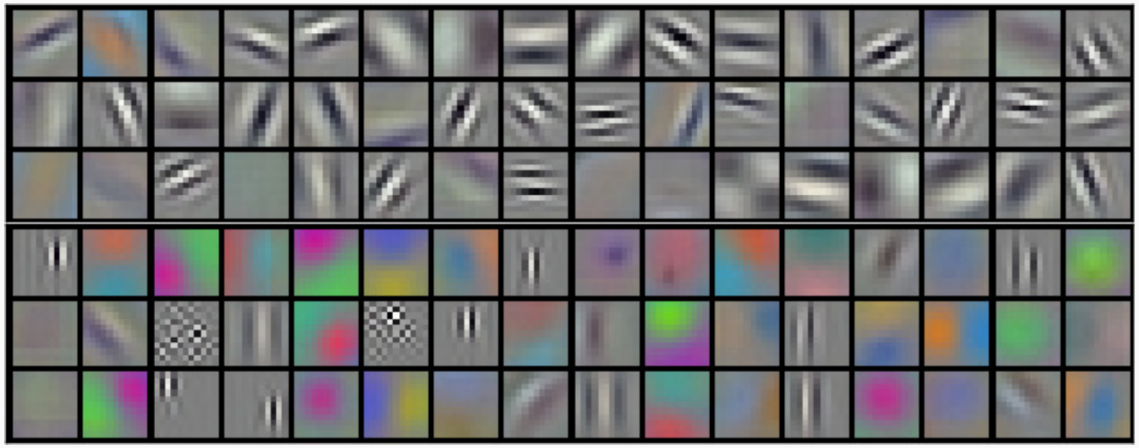
AlexNet (Krizhevsky, Sutskever, Hinton, 2012)

Other settings:

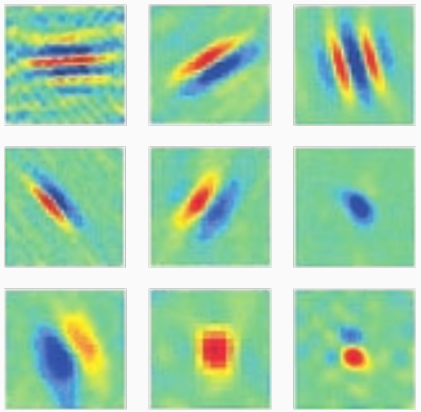
- **Number of parameters**: 60 million,
- **Data augmentation:** horizontal reflection, 224×224 subimages from 256×256 images, altering RGB channels. At test time, average the prediction obtained on 10 subimages extracted from the original one.
- **Dropout:** used for hidden neurons with probability $.5$.
- **Optimization:** SGD with mini-batch size 128 + weight decay + momentum $\rightarrow$ Took 6 days for 90 epochs.
- **Combination:** Trained 7 such networks and average their predictions (some of them were pretrained on other datasets and fine-tuned.)

AlexNet (Krizhevsky, Sutskever, Hinton, 2012)

What did it learn? Look at the filters that have been learned.



receptive fields estimated by reverse correlation:

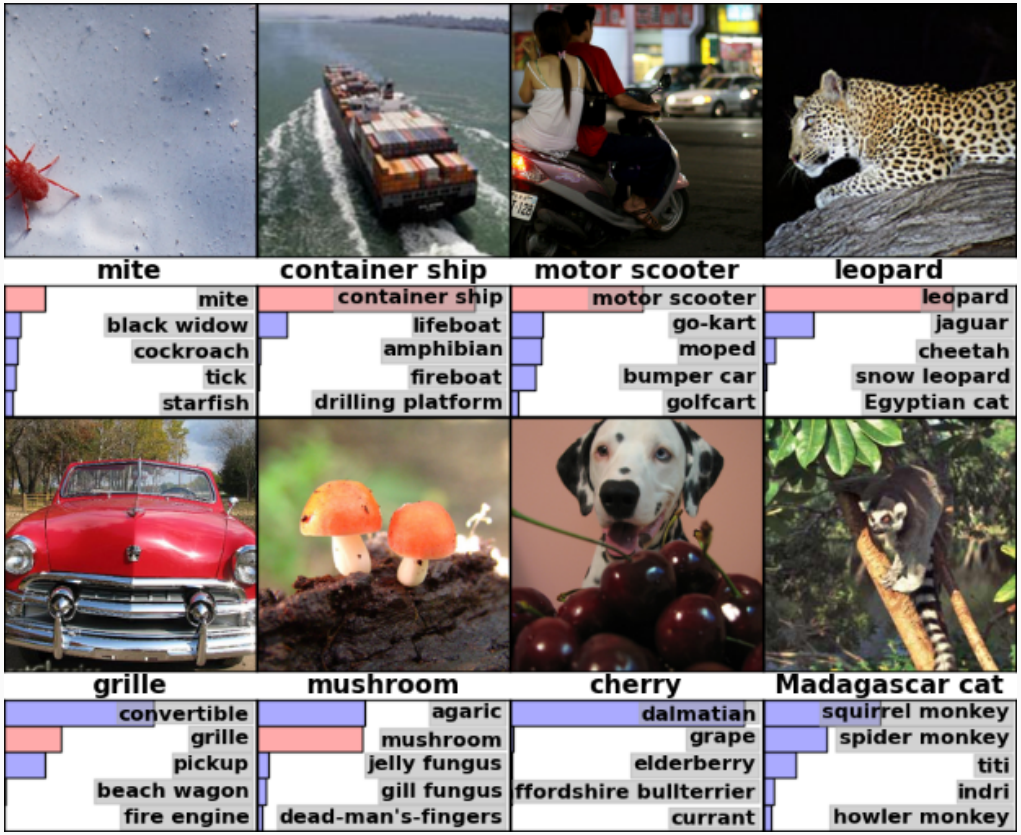


(left) 96 filters learned in the first convolution layer in AlexNet.

Many filters turn out to be edge detectors, similar to Gabor filters, common to human visual systems (right).

Successful CNNs architectures

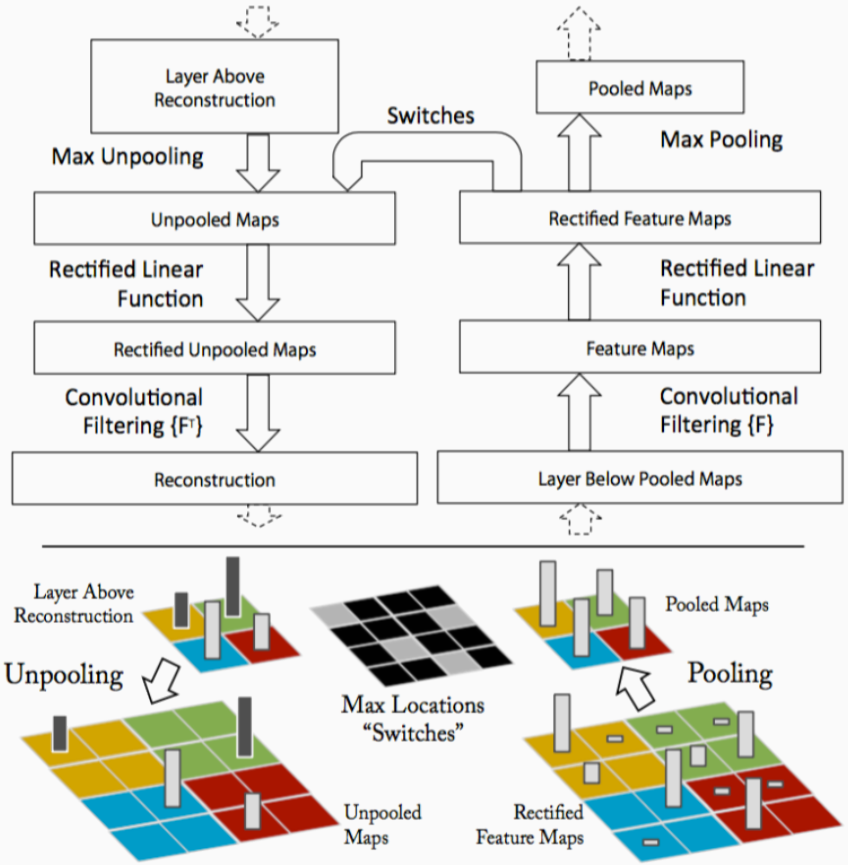
AlexNet (Krizhevsky, Sutskever, Hinton, 2012)



- Won ILSVRC 2012 by reducing the top-5 error to 16.4%.
- The second place, which was not a CNN, was around 26.2%.

ZFNet (Zeiler & Fergus, 2013)

Visualizing the evolution of features, or the learned filters is not informative enough to understand what has been learned.

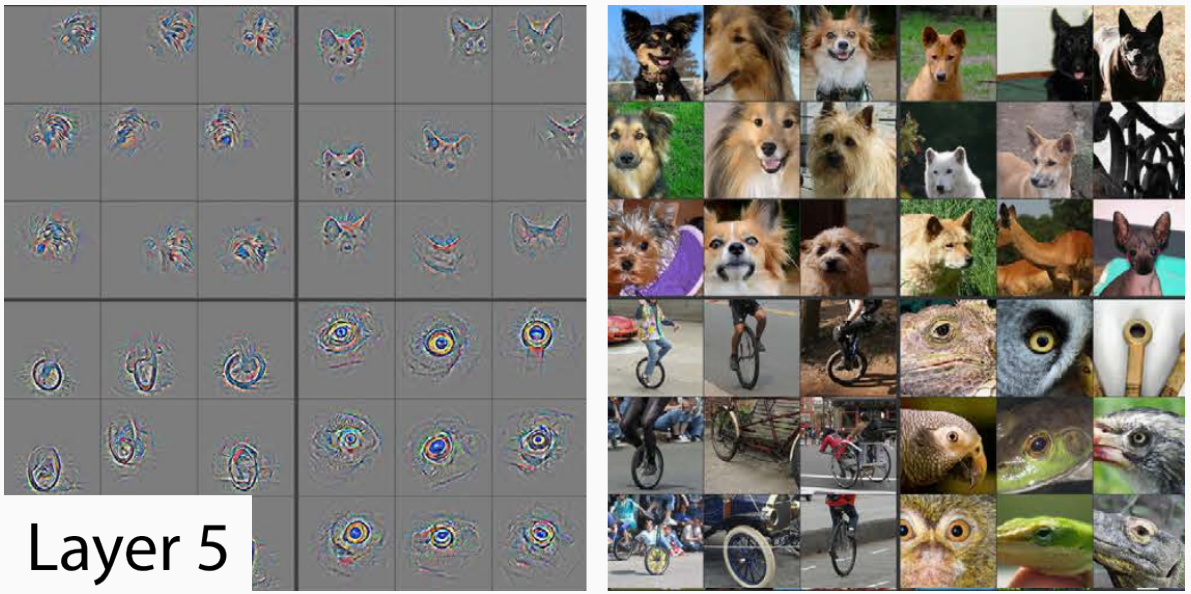


Introduced concept

Visualization with DeconvNet:
Offered a way to project an intermediate feature back to the image: starting from this feature, browse the network backward, replace pooling by unpooling, and convolutions by transposed convolutions (*i.e.*, flip the filters, aka, deconvolution layer).

We will go back to this later.

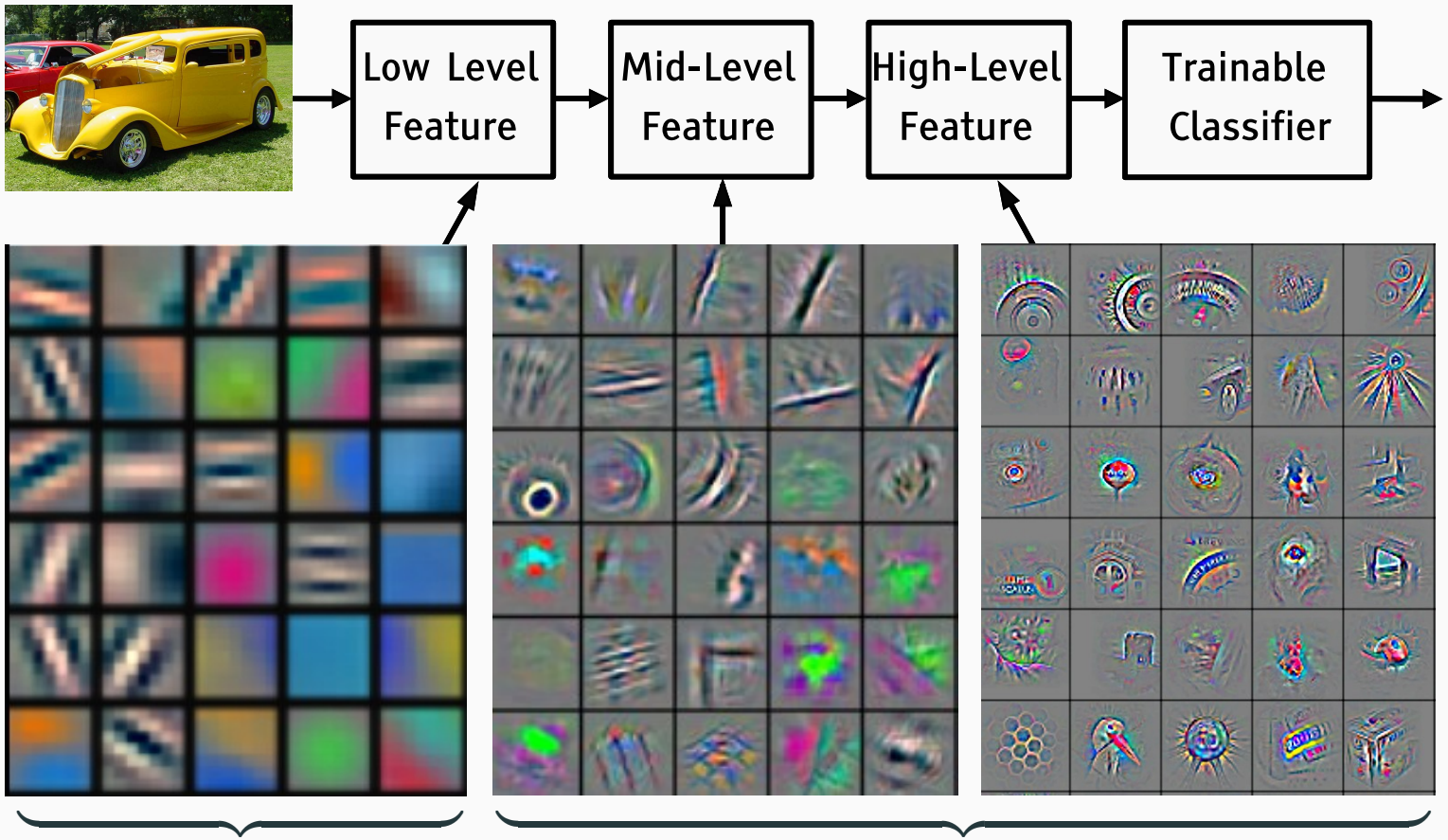
ZFNet (Zeiler & Fergus, 2013)



- Look at the intermediate features of AlexNet at each layer,
- Inspect the features and make some changes accordingly:
  - Layer 1: Conv 11×11 s4 $\rightarrow$ Conv 7×7 s2,
  - Layer 2: Conv 5×5 s1 $\rightarrow$ Conv 5×5 s2,
  - Use also Local Response Normalization,
  - Do not use a split architecture in two stages for GPU.

Successful CNNs architectures

ZFNet (Zeiler & Fergus, 2013)



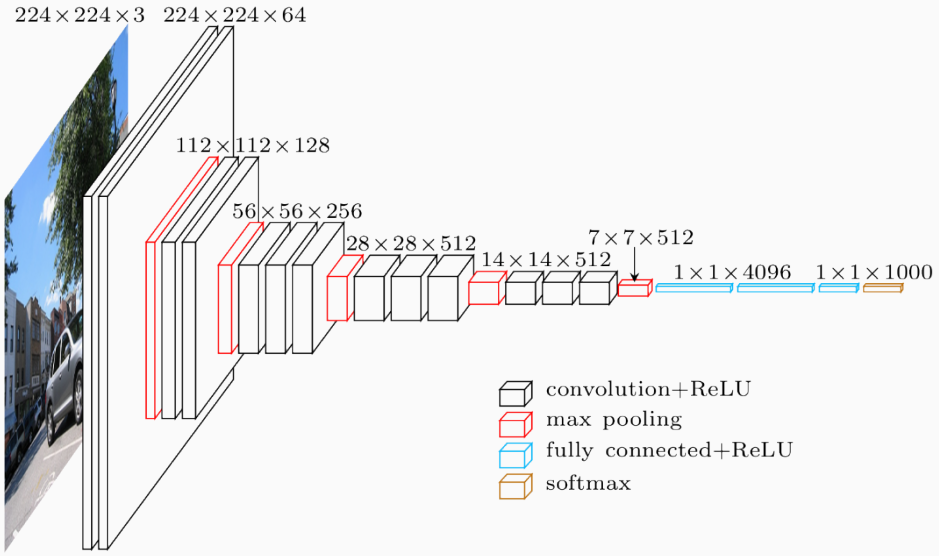
That's how we can produce such beautiful illustrations!

ZFNet (Zeiler & Fergus, 2013)

- Other settings:
  - **Number of parameters**: < 60 million,
  - **Data-augmentation**: crops and flips,
  - **Dropout**: used for FC layers with probability $.5$,
  - **Filter normalization**: when their norm exceed a threshold,
  - **Optimization**: SGD with mini-batch size 128 + momentum
    $\rightarrow$ Took 12 days for 70 epochs on a single Nvidia GTX 580 GPU.
  - **Combination:** Average the prediction of 5 such CNNs and another one with a slightly different architecture:
    - Layer 3,4,5: #Channels 384, 384, 256 $\rightarrow$ 512, 1024, 512.
- Won ILSVRC 2013 with 11.7% top-5 error (AlexNet: 16.4%).

Successful CNNs architectures

VGG (Simonyan & Zisserman, 2014)



Introduced concept

Deep and simple:

- 16 conv filters, 3×3 s1,
- 5 max pool, 2×2 s2,
- 3 FC layers,
- No need of local response normalization.

Why does it work?

- Two first 3×3 conv layers: effective receptive field is 5×5,
- Three first 3×3 conv layers: effective receptive field is 7×7,
- Why is it better than ZFNet which uses 7×7?
  - More discriminant: 3 ReLUs instead of 1 ReLU,
  - Less parameters: $3 \times (3 \times 3) = 27$ vs $1 \times (7 \times 7) = 49$.
- Next, apply max-pooling and the effective receptive field double!

VGG (Simonyan & Zisserman, 2014)

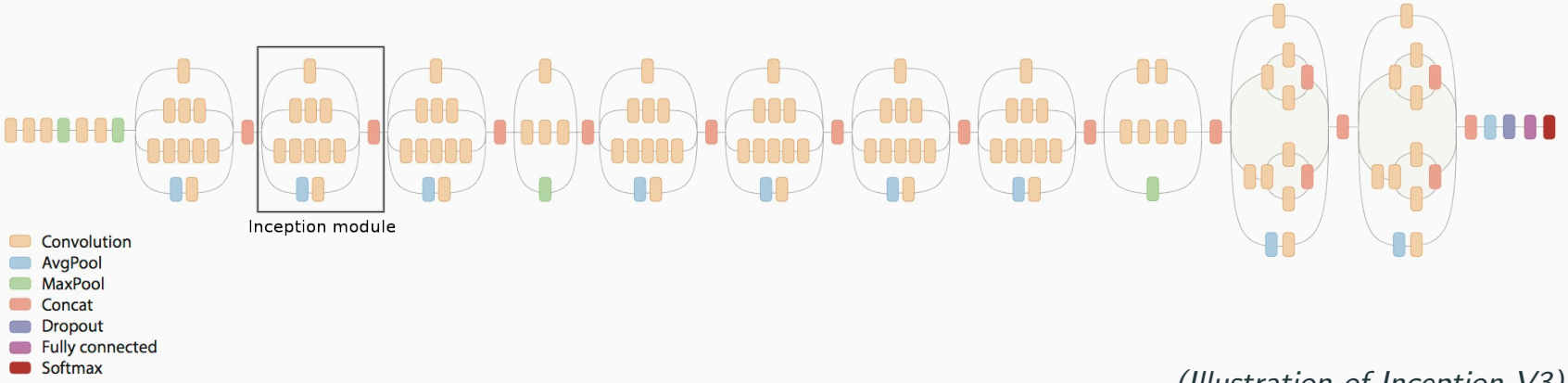
Other settings:

- **Number of parameters**: 140 million,
- **Data-augmentation**: crops, flips, RGB color shift and scaling jittering.
- **Dropout**: used for the first two FC layers with probability $.5$,
- **Optim**: SGD with mini-batch size 256 + momentum + weight decay
  $\rightarrow$ Took about 3 weeks for 74 epochs on 4 Nvidia Titan Black GPUs.
- **Combination:** Combined 7 such CNNs.

Runner-up of ILSVRC 2014 with 7.3% top-5 error (ZFNet: 11.7%).
Not winner but very influential: "Make it deep. Keep it simple".

Successful CNNs architectures

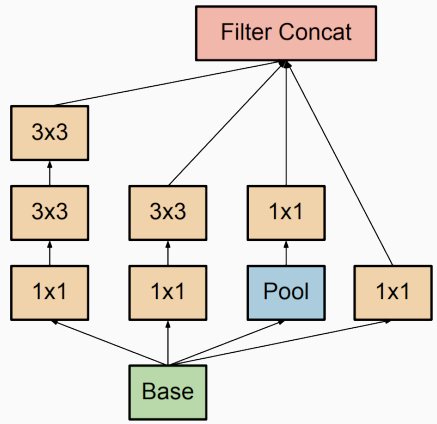
Inception/GoogLeNet (Szegedy *et al.*, 2014), V2 (2015), V3 (2016), V4 (2017)



(Illustration of Inception V3)

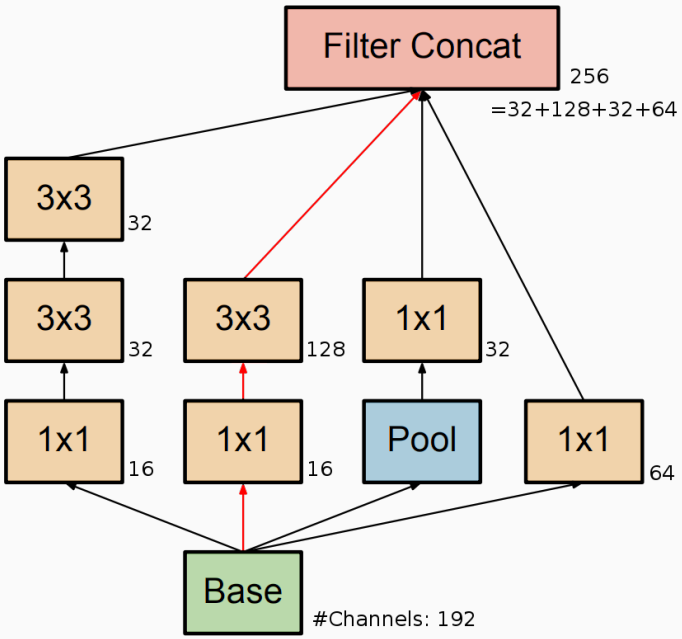
Introduced concept:

- **Inception module:** apply filters of different sizes in parallel and concatenate their answers. Each module captures in parallel details at different scales and has thus different effective receptive fields.
- **Deeper with fewer parameters:** introduced "1×1" convolutions for dimensionality reduction. The first version had 22 layers and 4 million parameters! (AlexNet has 8 layers and 60 million parameters).



(An inception module)

Inception/GoogLeNet (Szegedy *et al.*, 2014), V2 (2015), V3 (2016), V4 (2017)



What are 1×1 convolutions?

- 1×1 (multi) convolutions: outputs are simply element-wise weighted sums of the inputs. Producing less outputs than inputs allows for dimensionality reduction.

Why do they perform dimensionality reduction?

- They can take 192 feature maps and reduce the information to 16 feature maps (with same width and height).

Why do they allow for reducing the number of parameters?

- Consider the branch in red:
  - Without: $192 \times 3 \times 3 \times 128 \approx 221,000$ parameters,
  - With: $192 \times 16 + 16 \times 3 \times 3 \times 128 \approx 21,500$ parameters.

They also decrease by the same factor the computation cost!

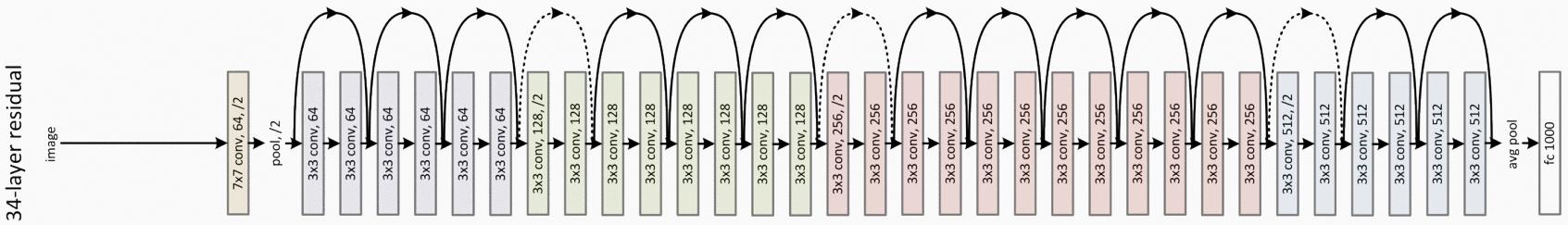
Inception/GoogLeNet (Szegedy *et al.*, 2014), V2 (2015), V3 (2016), V4 (2017)

Other settings:

- **Number of parameters**: 6.8 million (VGG: 140M),
- **Data-augmentation**: crops of different sizes, flips, random interpolations.
- **Dropout**: used for one FC layer with probability $.4$,
- **Optimization**: SGD + momentum + learning rate schedule
  $\rightarrow$ Authors suggest it may take about a week with few GPUs.
- **Combination:** Average the prediction of 7 CNNs over 144 crops per images (shifted, rescaled and flipped).

Won ILSVRC 2014 with 6.7% top-5 error (VGG: 7.3%).

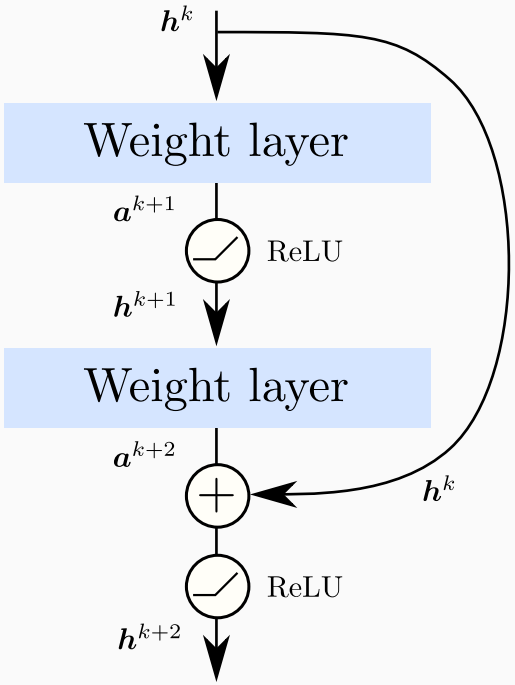
ResNet (Microsoft – He *et al.*, 2016)



(short version only 34 layers here!)

Introduced concept:

- **Be ultra deep:** 152 layers! First layer uses 7×7 convolutions and everything else is 3×3 with only two pooling layers.
- **Residual layer:** learn how to change the input instead of learning what the output should be (using shortcut connections). Refer to the previous class.



ResNet (Microsoft – He *et al.*, 2015)

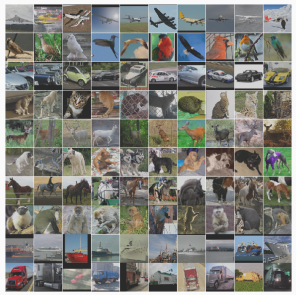
Other settings:

- **Number of parameters**: 60 million,
- **Data-augmentation**: resized, cropped, flipped.
- **Batch-normalization**: between convolutions and ReLU (no dropout).
- **Optimization**: SGD with mini-batch size 256 + momentum
  + weight decay + learning rate schedule
  + He initialization (He *et al.*, 2015. Same guy)
  → Trained for about two weeks using 8 GPUs.
- **Combination:** average predictions of 6 CNNs (only two with 152 layers) over ten crops and multiple scales.

Won ILSVRC 2015 with 3.57% top-5 error (GoogLeNet, 6.7%).
First time a system beats human level prediction on ImageNet (about 5.1%).

Other standard datasets

CIFAR-10



STL-10

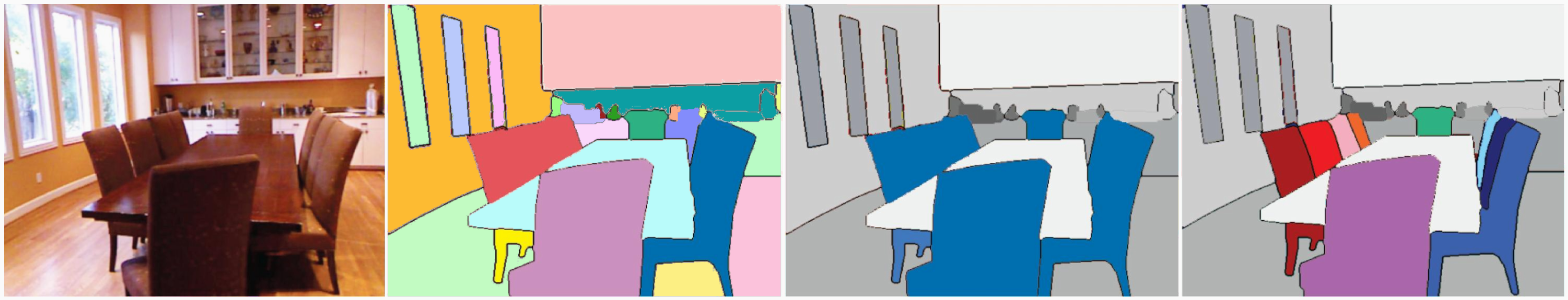
SVHN

- CIFAR-10 datasets
  - 60000 32x32 color images,
  - 10 classes, with 6000 images per class,
  - 50000 training images and 10000 testing images.
- CIFAR-100 datasets
  - Same but with 100 classes,
  - Each image as a fine and coarse label.
- STL-10
  - Similar as CIFAR-10 but images are 96x96.
- SVHN (Street View House Numbers)
  - Similar to MNIST but with 600,000 color digit images.

Evaluation on these datasets usually comes with other metrics.

Segmentation

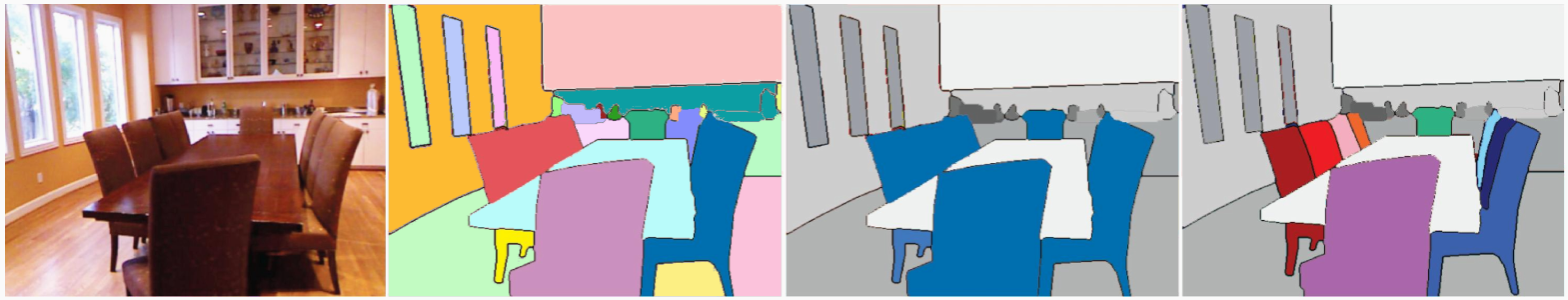
Segmentation – Terminology



Input Image | Segmentation | Semantic Segmentation | Semantic Instance Segmentation

- **Segmentation:**
  - Partition of an image into several "coherent" parts/segments,
  - Without any attempt at understanding what these parts represent,
  - Typically based on color, textures, smoothness of boundaries,
  - Also referred to as super-pixel segmentation.

Segmentation – Terminology

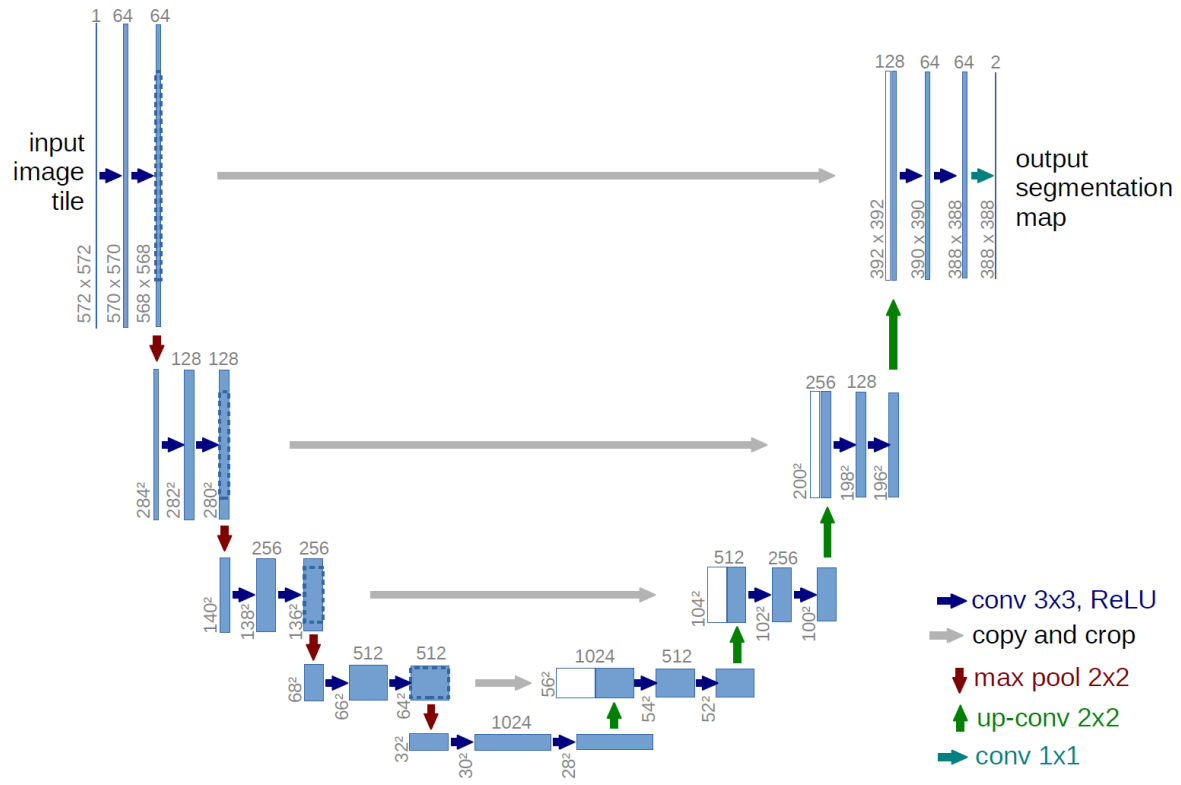


Input Image | Segmentation | Semantic Segmentation | Semantic Instance Segmentation

- **Semantic segmentation:**
  - Each segment corresponds to a class label (objects + background),
  - Also referred to as scene parsing or scene labeling.
- **Instance segmentation:**
  - Find object boundaries between objects, including delineations between instances of the same object.
- **Semantic instance segmentation:** find object boundaries + labels.

Network for segmentation

U-net architecture (O. Ronneberger et al, 2015)



- Idea: Classify each pixel
- Condense spatial information as for image classification.
- Re-affine spatially the classification step by step with mirror upsampling steps (transpose of conv2D with padding) and concatenation.

Network for segmentation

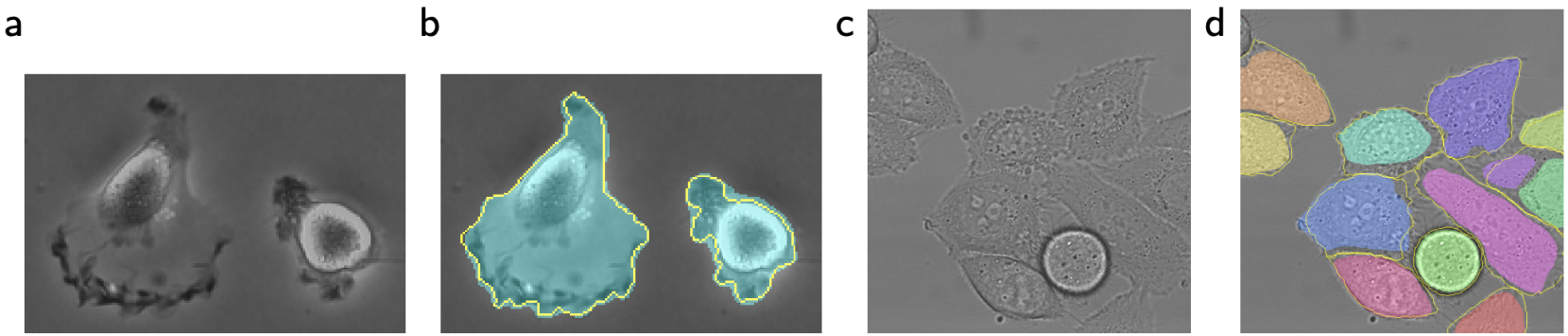
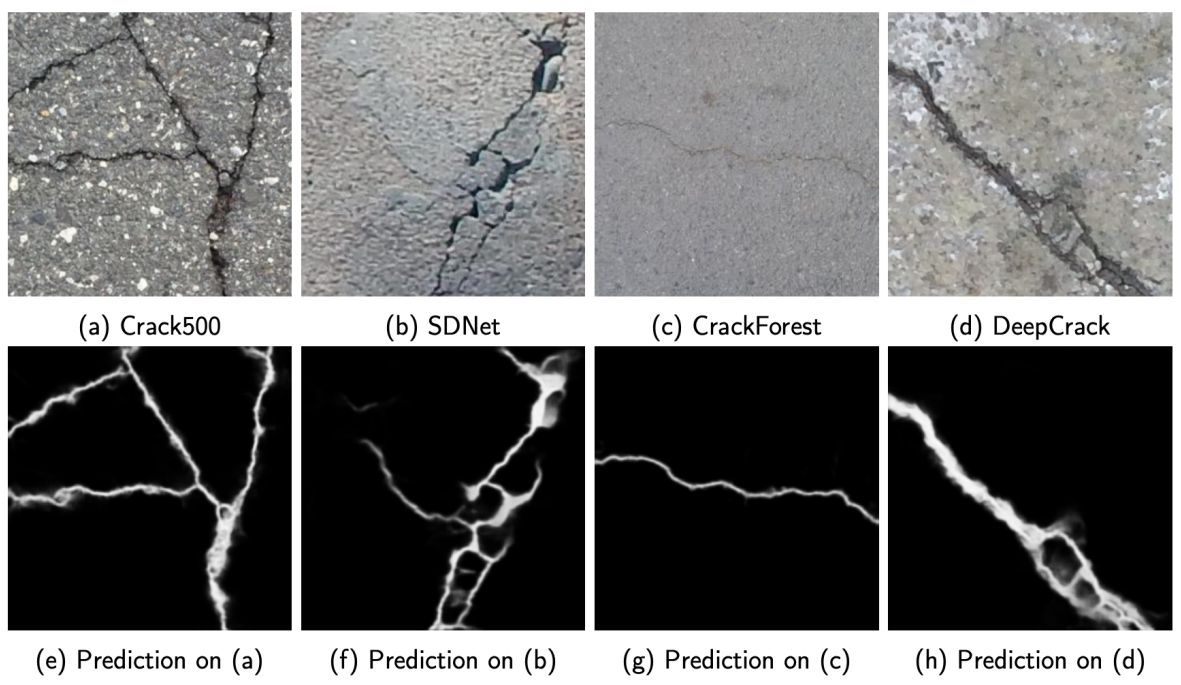


Fig. 4. Result on the ISBI cell tracking challenge. (**a**) part of an input image of the "PhC-U373" data set. (**b**) Segmentation result (cyan mask) with manual ground truth (yellow border) (**c**) input image of the "DIC-HeLa" data set. (**d**) Segmentation result (random colored masks) with manual ground truth (yellow border).

- Improved state-of-the-art in cell-tracking.
- Can be extended to very different contexts provided enough labeled data.

Network for segmentation

U-net architecture (O. Ronneberger et al, 2015)



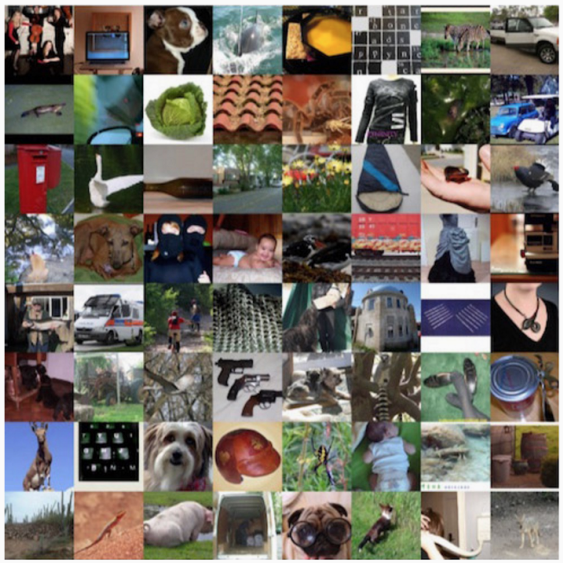
(S. Drouillet, 2020)

- Example usage: Crack detection
- The network outputs the probability that each pixel belongs to a crack.

Image generation

Motivations – Image generation

- **Goal:** Generate images that look like the ones of your training set.



Real images (ImageNet)

(Training set)

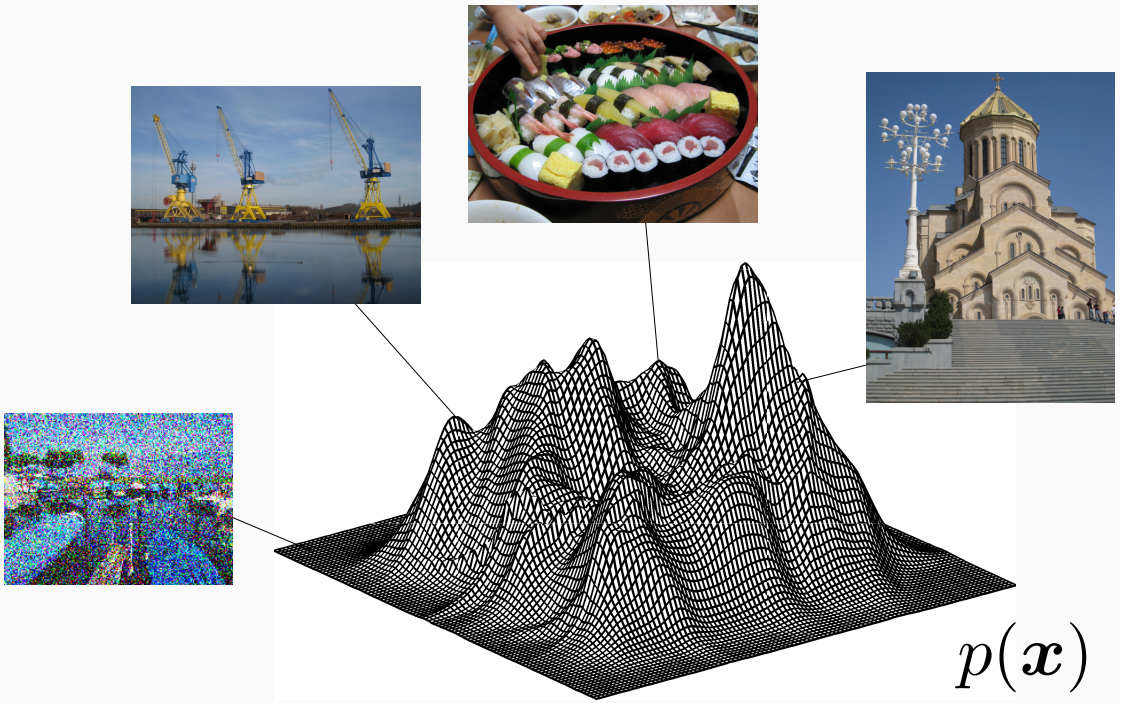
Generated images

(Results)

- **What?** Unsupervised learning.
- **Why?** Different reasons and applications:
  - Can be used for simulation, *e.g.*, to generate labeled datasets,
  - Must capture all subtle patterns $\rightarrow$ provide good features,
  - Can be used for other tasks: super-resolution, style transfer, ...

Image generation – Explicit density

1. Learn the distribution of images $p(\boldsymbol{x})$ on a training set.



2. Generate samples from this distribution.

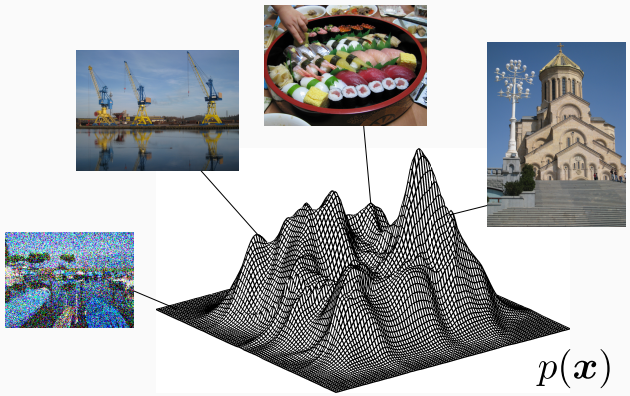
Image generation – Gaussian model

- Consider a Gaussian model for the distribution of images $\boldsymbol{x}$ with n pixels:

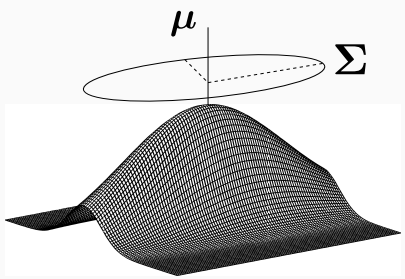
$$\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{2\pi}^n |\boldsymbol{\Sigma}|^{1/2}} \exp\left[(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}-\boldsymbol{\mu})\right]$$

  - $\boldsymbol{\mu}$: mean image,
  - $\boldsymbol{\Sigma}$: covariance matrix of images.



$\approx$



Gaussian prior $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

Image generation – Gaussian model

- Take a training dataset $\mathcal{T}$ of images:

$$\mathcal{T} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$$

$$= \left\{ \text{  }, \text{  }, \text{  }, \text{  }, \text{  }, \text{  }, \ldots \right\}_{\times N}$$

- Estimate the mean

$$\hat{\boldsymbol{\mu}} = \frac{1}{N} \sum_i \boldsymbol{x}_i = \text{  }$$

- Estimate the covariance matrix: $\hat{\boldsymbol{\Sigma}} = \frac{1}{N} \sum_i (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})^T = \hat{\boldsymbol{E}} \hat{\boldsymbol{\Lambda}} \hat{\boldsymbol{E}}^T$

$$\hat{\boldsymbol{E}} = \underbrace{\left\{ \text{  }, \text{  }, \text{  }, \text{  }, \text{  }, \text{  }, \ldots \right\}_{\times N}}_{\text{eigenvectors of } \hat{\boldsymbol{\Sigma}}, \textit{i.e.}, \text{ main variation axis}}$$

Image generation – Gaussian model

You now have learned a generative model:

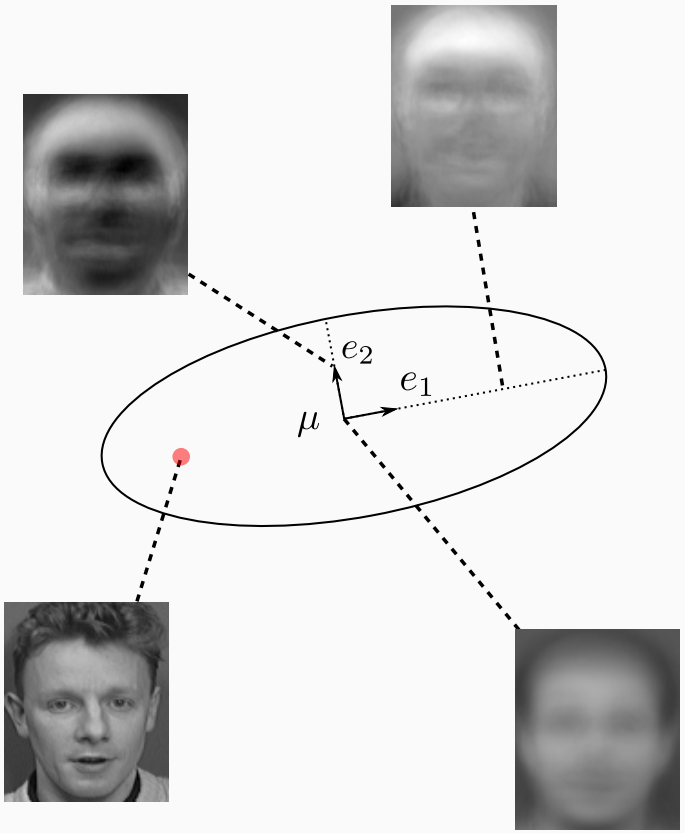


Image generation – Gaussian model

How to generate samples from $\mathcal{N}(\hat{\mu}, \hat{\Sigma})$?

$$\begin{cases} \boldsymbol{z} & \sim \mathcal{N}(0, \mathrm{Id}_n) \quad \leftarrow \text{Generate random latent variable} \\ \boldsymbol{x} & = \hat{\boldsymbol{\mu}} + \hat{\boldsymbol{E}}\hat{\boldsymbol{\Lambda}}^{1/2}\boldsymbol{z} \end{cases}$$

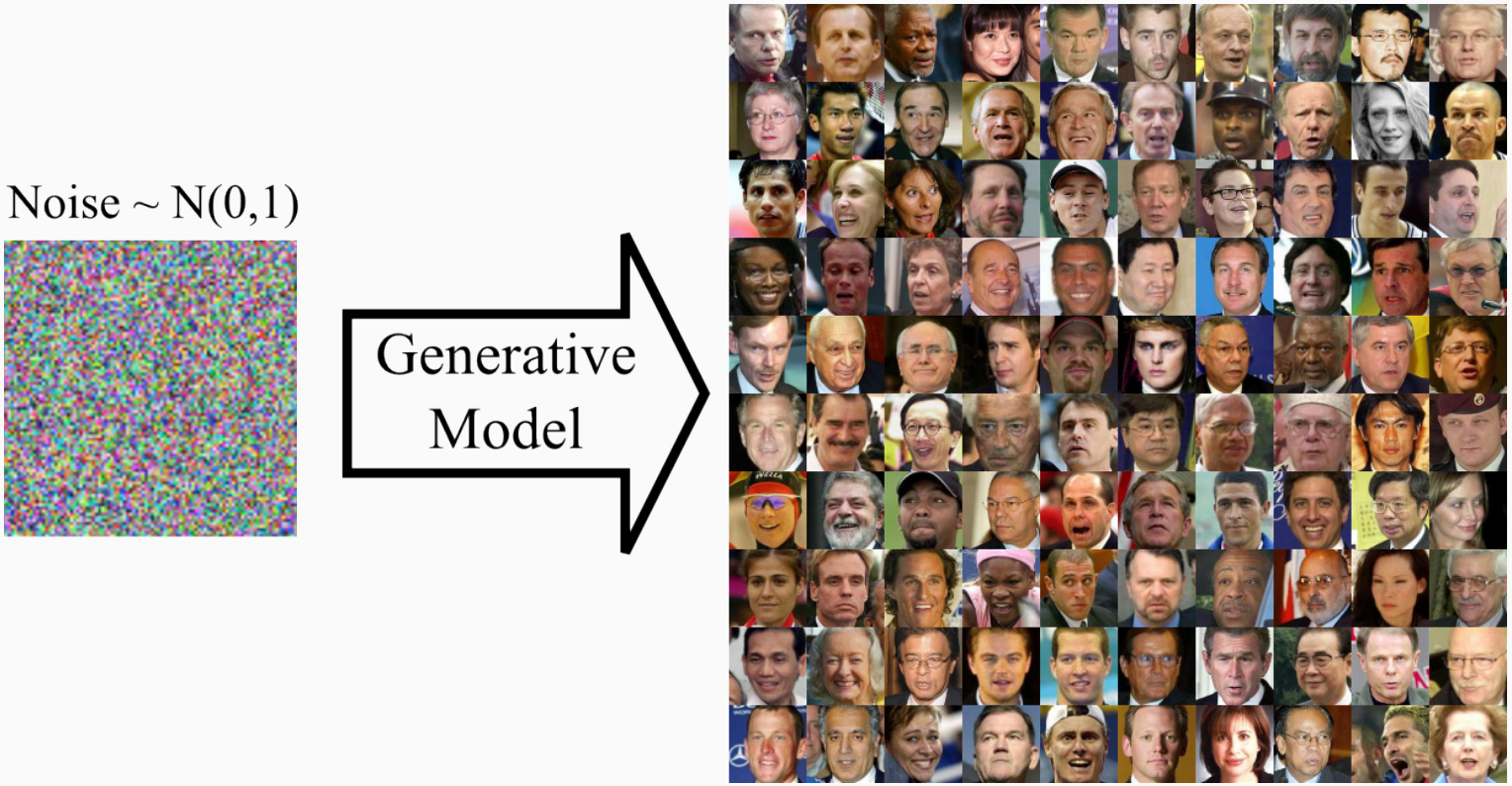
The model does not generate realistic faces.

The Gaussian distribution assumption is too simplistic.

Each generated image is just a linear random combination of the eigenvectors.

The generator corresponds to a linear neural network (without non-linearities).

Image generation – Beyond Gaussian models



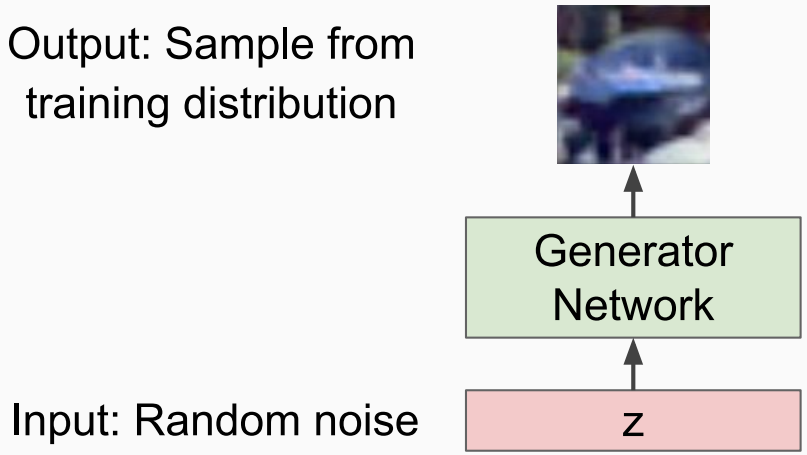
But the concept is interesting: can we find a transformation such that each random code can be mapped to a photo-realistic image?

We need to find a way to assess if an image is photo-realistic.

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

- **Goal:** design a complex model with high capacity able to map latent random noise vectors z to a realistic image x.
- **Idea:** Take a deep neural network

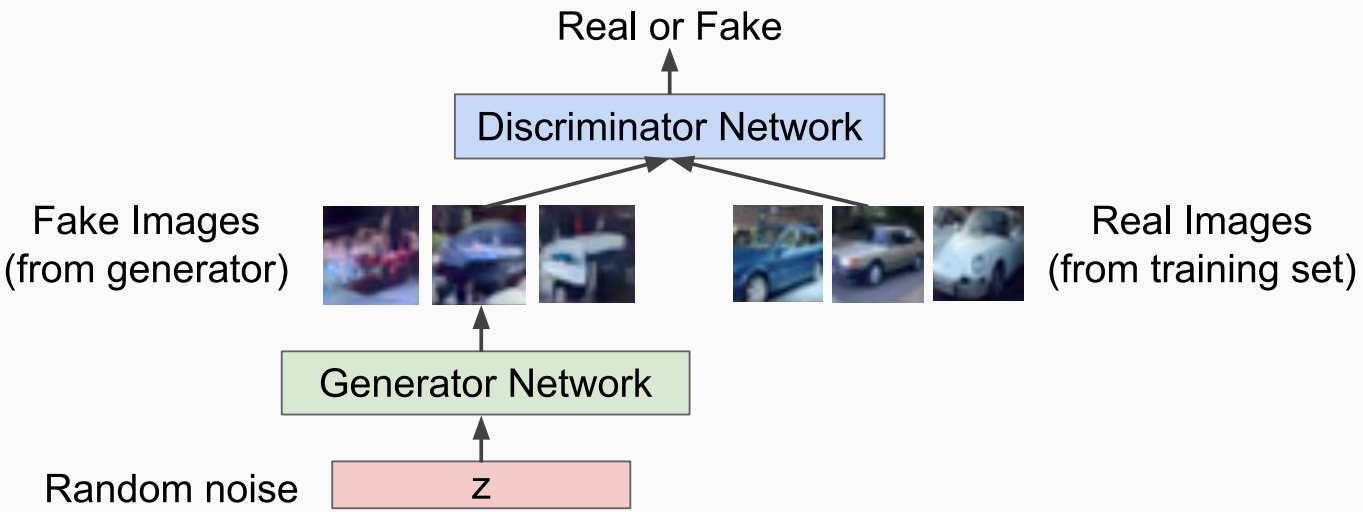


- **What about the loss?** Measure if the generated image is photo-realistic.

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

Define a loss measuring how much you can fool a classifier that has learned to distinguish between real and fake images.



- **Discriminator network:** try to distinguish between real and fake images.
- **Generator network:** fool the discriminator by generating realistic images.

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

- **Discriminator network:** Consider two sets
  - $\mathcal{T}_{\text{real}}$: a dataset of n real images,
  - $\mathcal{T}_{\text{fake}}$: a dataset of m fake images.

- **Goal:** find the parameters θ_d of a binary classification network $\boldsymbol{x} \mapsto D_{\theta_d}(\boldsymbol{x})$ meant to classify real and fake images.
  Minimize the cross entropy, or maximize its negation

$$\max_{\theta_d} \underbrace{\frac{1}{n} \sum_{\boldsymbol{x} \in \mathcal{T}_{\text{real}}} \log D_{\theta_d}(\boldsymbol{x})}_{\substack{\text{force predicted labels to be 1} \\ \text{for real images}}} + \underbrace{\frac{1}{m} \sum_{\boldsymbol{x} \in \mathcal{T}_{\text{fake}}} \log(1 - D_{\theta_d}(\boldsymbol{x}))}_{\substack{\text{force predicted labels to be 0} \\ \text{for fake images}}}$$

- **How:** use gradient ascent with backprop (+SGD, batch-normalization. . .).

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

- **Generator network:** Consider a given discriminative model $\boldsymbol{x} \mapsto D_{\theta_d}(\boldsymbol{x})$ and consider $\mathcal{T}_{\text{rand}}$ a set of m random latent vectors.
- **Goal:** find the parameters θ_g of a network $\boldsymbol{x} \mapsto G_{\theta_g}(\boldsymbol{z})$ generating images from random vectors $\boldsymbol{z}$ such that it fools the discriminator

$$\min_{\theta_g} \underbrace{\frac{1}{m} \sum_{\boldsymbol{z} \in \mathcal{T}_{\text{rand}}} \log(1 - D_{\theta_d}(G_{\theta_g}(\boldsymbol{z})))}_{\substack{\text{force the discriminator to think that} \\ \text{our generated fake images are not fake (away from 0)}}} \tag{1}$$

or alternatively (works better in practice)

$$\max_{\theta_g} \underbrace{\frac{1}{m} \sum_{\boldsymbol{z} \in \mathcal{T}_{\text{rand}}} \log D_{\theta_d}(G_{\theta_g}(\boldsymbol{z}))}_{\substack{\text{force the discriminator to think that} \\ \text{our generated fake images are real (close to 1)}}} \tag{2}$$

- **How:** gradient descent for (1) or gradient ascent for (2) with backprop...

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

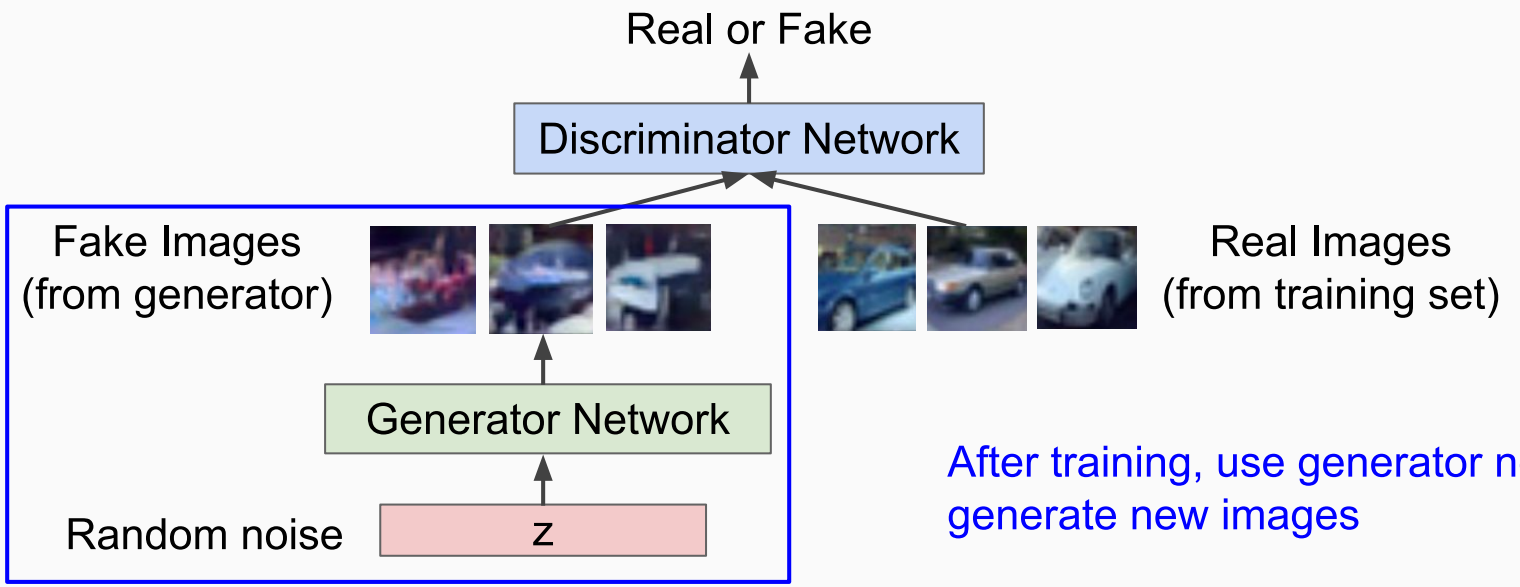
- **Train both networks jointly**.
- Minimax loss in a two player game (each player is a network):

$$\min_{\theta_g} \max_{\theta_d} \frac{1}{n} \sum_{\boldsymbol{x} \in \mathcal{T}_{\text{real}}} \log D_{\theta_d}(\boldsymbol{x}) + \frac{1}{m} \sum_{\boldsymbol{z} \in \mathcal{T}_{\text{rand}}} \log(1 - D_{\theta_d}(\underbrace{G_{\theta_g}(\boldsymbol{z})}_{\text{fake}}))$$

- **Algorithm:** repeat until convergence
  1. Fix θ_g, update θ_d with one step of gradient ascent,
  2. Fix θ_d, update θ_g with one step of gradient descent for (1), (or one step of gradient ascent for (2).)

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

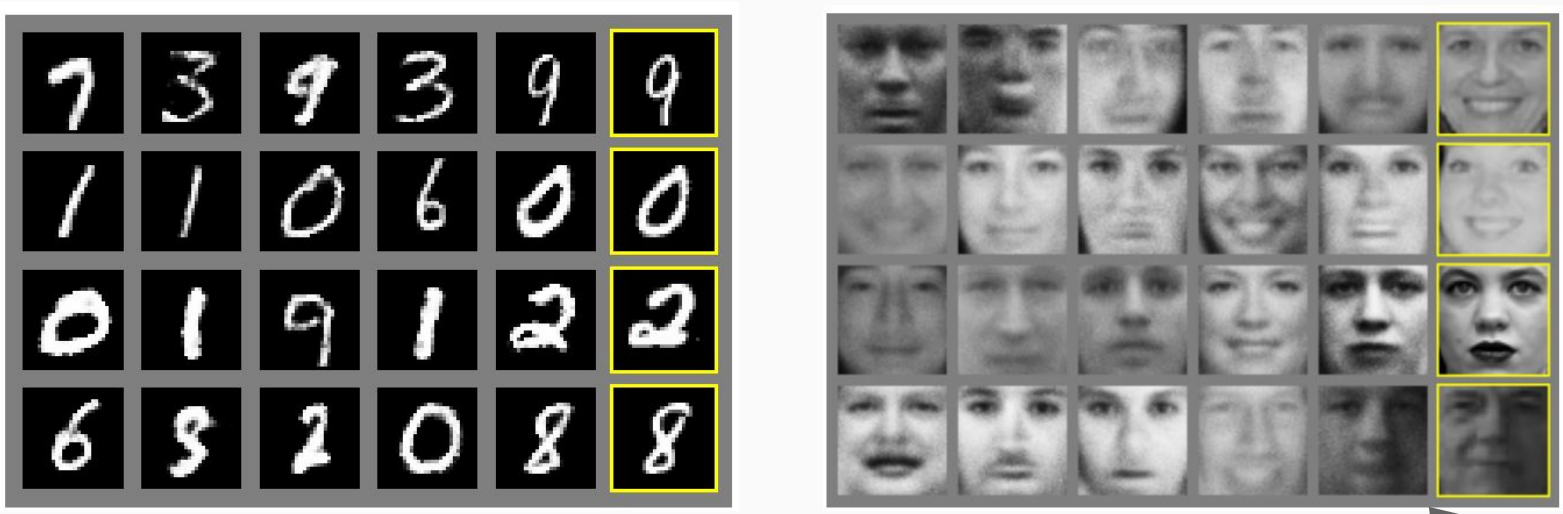


After training, use generator network to generate new images

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

Generated samples

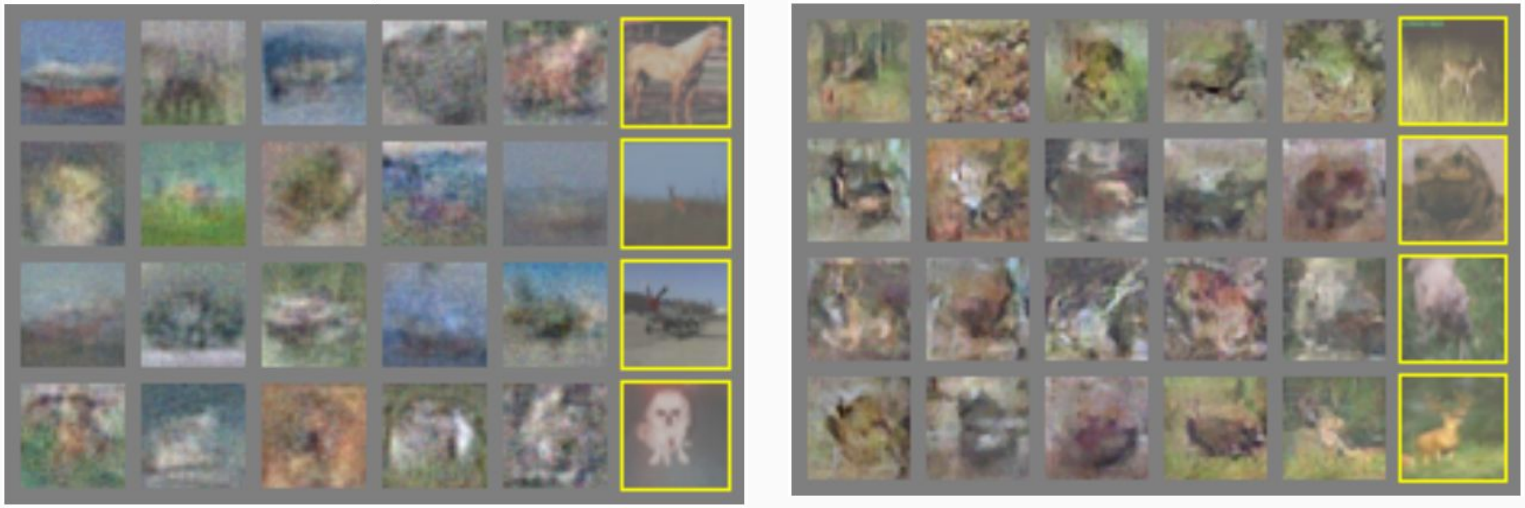


Nearest neighbor from training set

Generative Adversarial Networks

(Goodfellow et al., NIPS 2014)

Generated samples (CIFAR-10)

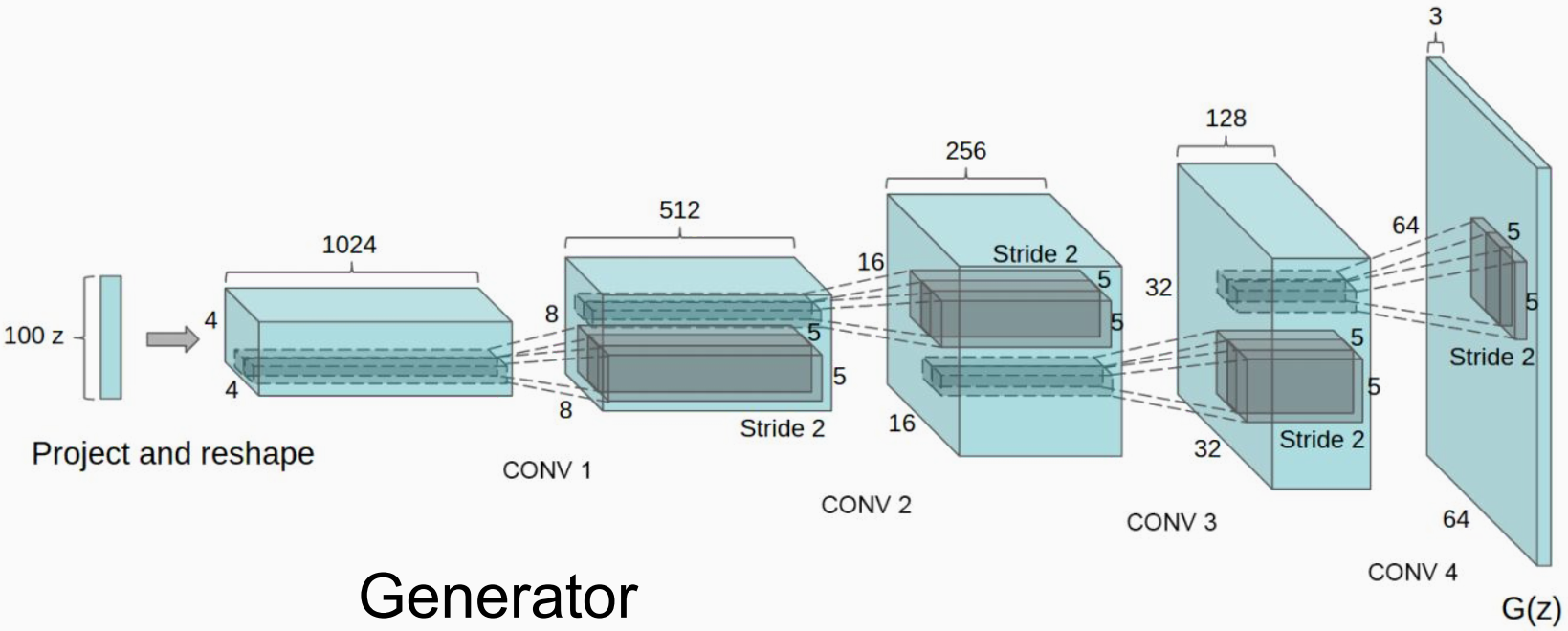


Nearest neighbor from training set

Convolutional GAN

(Radford et al., 2016)

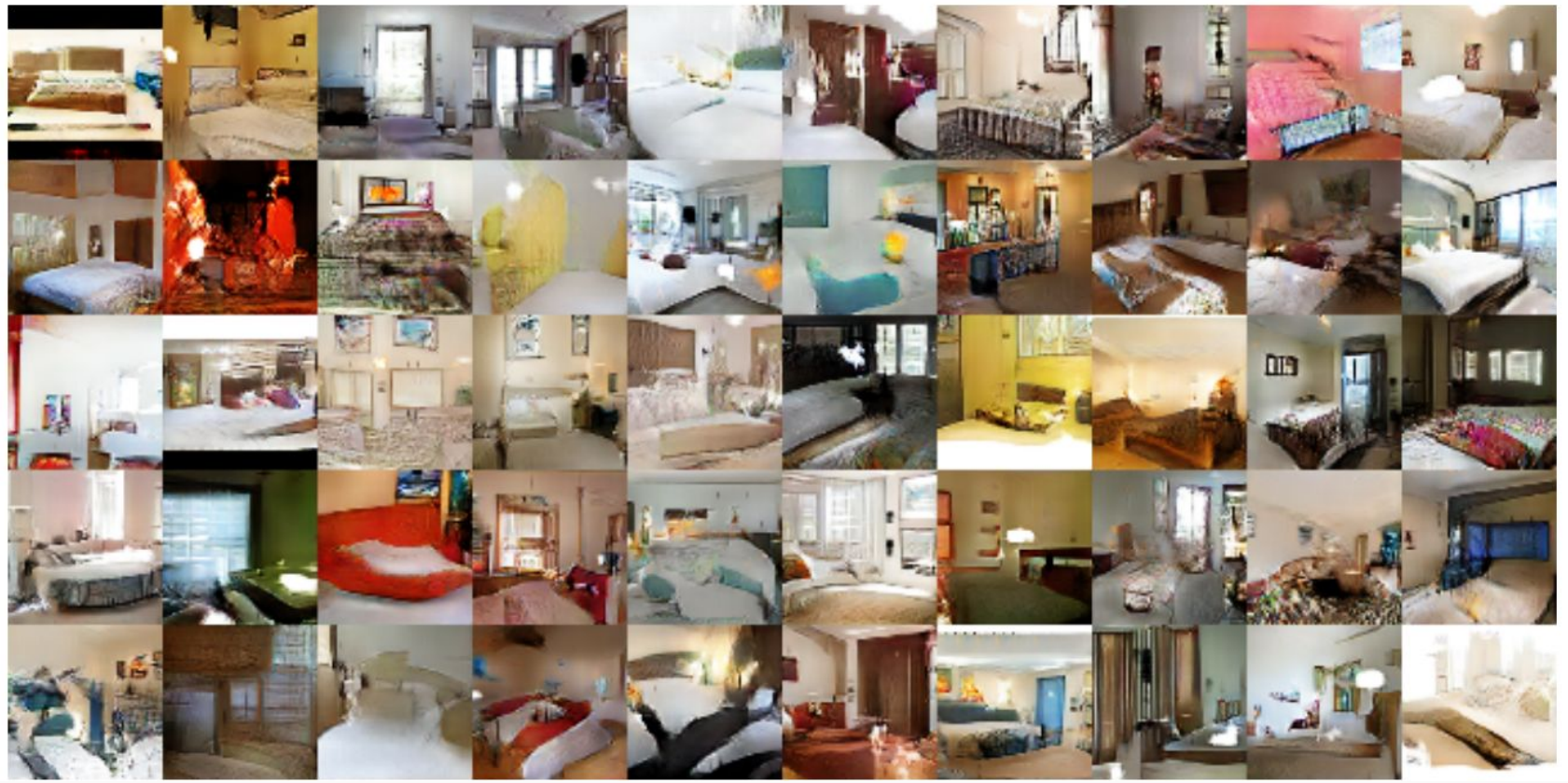
- **Generator:** upsampling network with fractionally strided convolutions,
- **Discriminator:** convolutional network with strided convolutions.



Generator

Convolutional GAN

(Radford et al., 2016)



Generations of realistic bedrooms pictures, from randomly generated latent variables.

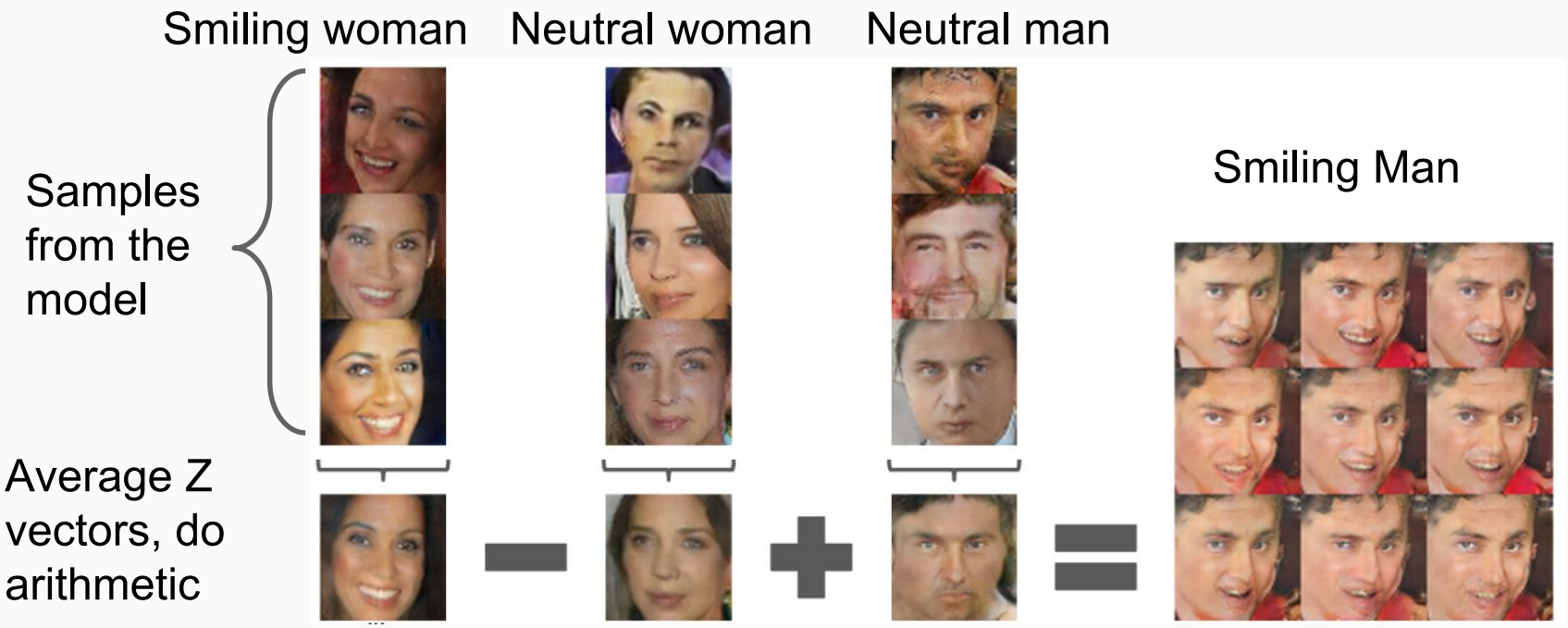
Convolutional GAN

(Radford et al., 2016)

Interpolation in between points in latent space.

Convolutional GAN – Arithmetic

(Radford et al., 2016)

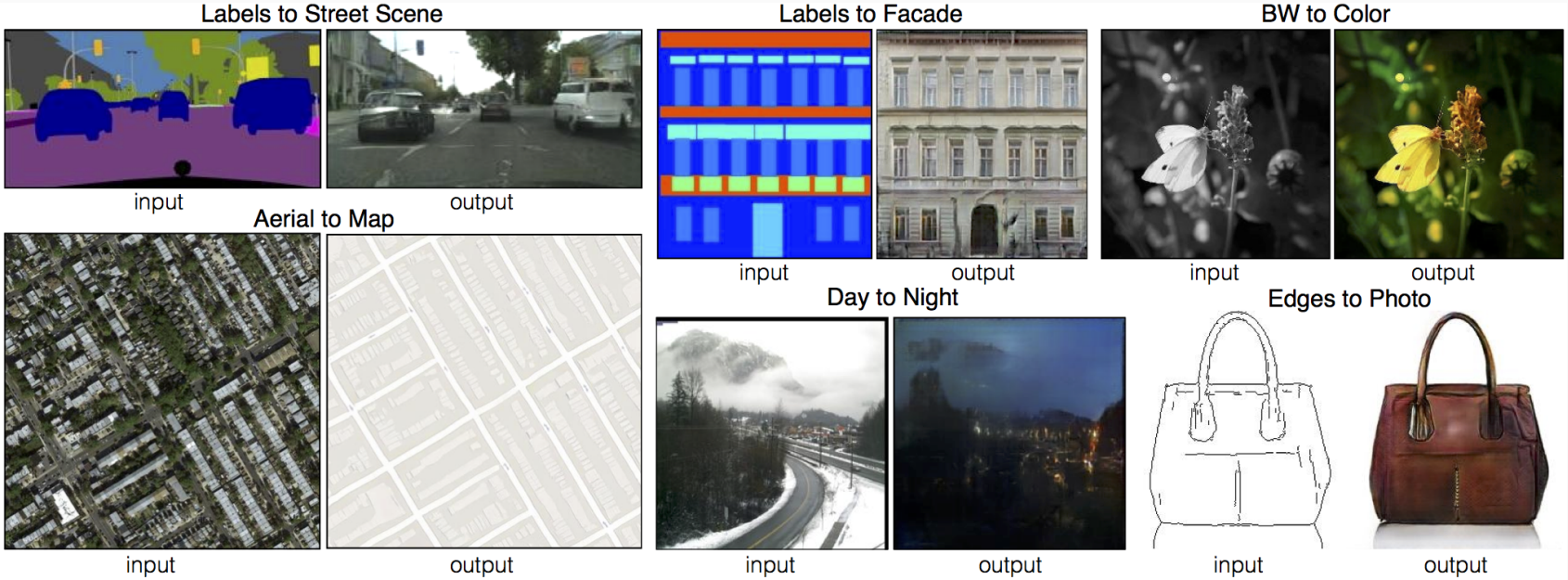


Convolutional GAN – Arithmetic

(Radford et al., 2016)

Style GAN (Karras et al., 2018)

Pix2pix: Image-to-Image Translation with Conditional Adversarial Nets (Isola et al., 2017)



- GAN conditioned on input image.
- Generator: U-net architecture
- Discriminator: Patch discriminator applied to each patch
- Opens the way for new creative tools

Questions?

Slides from Charles Deledalle

Sources, images courtesy and acknowledgment

K. Chatfield

P. Gallinari,

C. Hazırbaş

A. Horodniceanu

Y. LeCun

V. Lepetit

L. Masuch

A. Ng

M. Ranzato