Convolutional Neural Networks and Generative Networks

Alasdair Newson

Equipe IMAGES - Télécom Paris
alasdair.newson@telecom-paris.fr

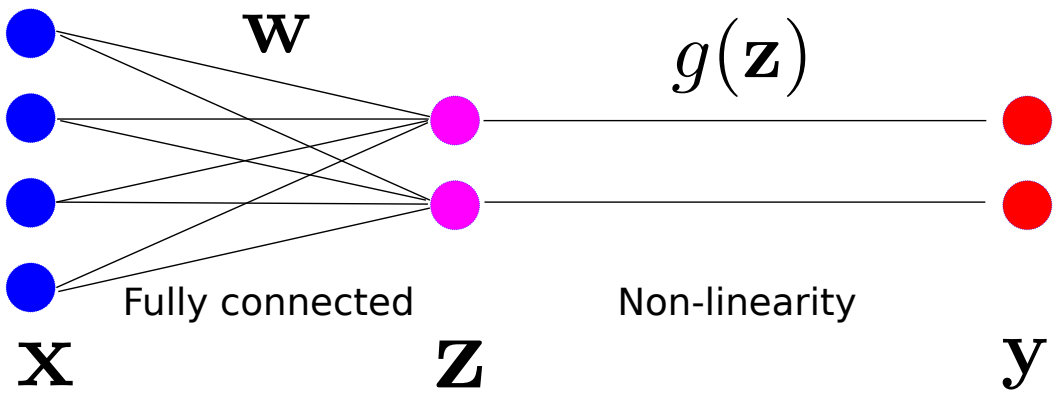
17 April, 2023

Summary

1. Convolutional Neural Networks
   - Introduction, notation
     - Convolutional Layers
   - Down-sampling and the receptive field
     - Image classification datasets and well-known CNNs
   - Applications of CNNs
     - Adversarial Examples
     - Visualising CNNs
2. Generative Models
   - Autoencoders
     - Vanilla Autoencoder
     - Autoencoder variants
   - Generative models
     - Variational autoencoders
     - Generative Adversarial Networks
     - Texture and style models
3. Summary

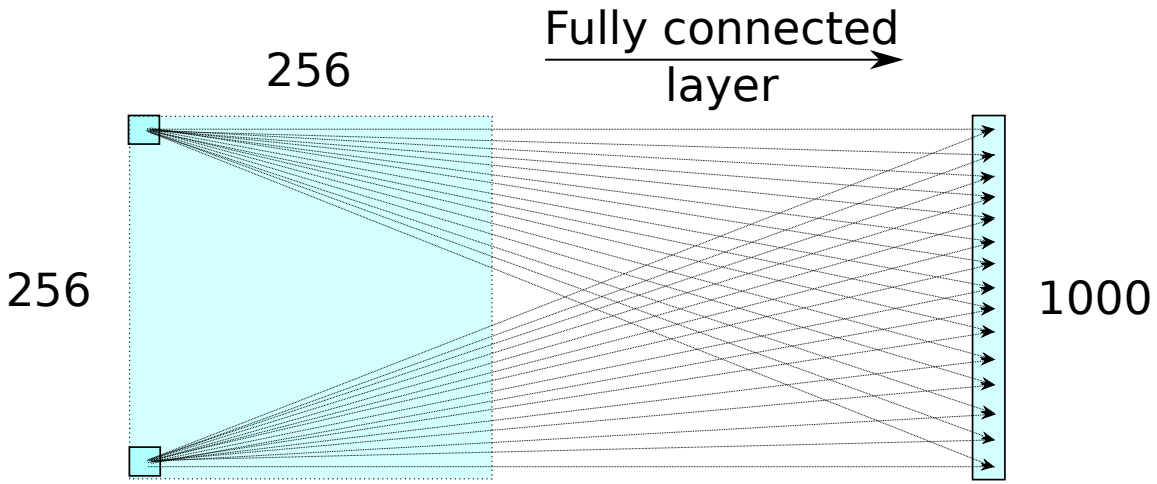
Introduction

- Neural networks provide a highly flexible way to model complex dependencies and patterns in data
- Last lesson, we saw the following elements :
  - MLPs : fully connected layers, biases
  - Activation functions : sigmoid, soft max, ReLU
  - Optimisation : gradient descent, stochastic gradient descent, momentum
  - Regularisation : weight decay, dropout, batch normalisation



Introduction

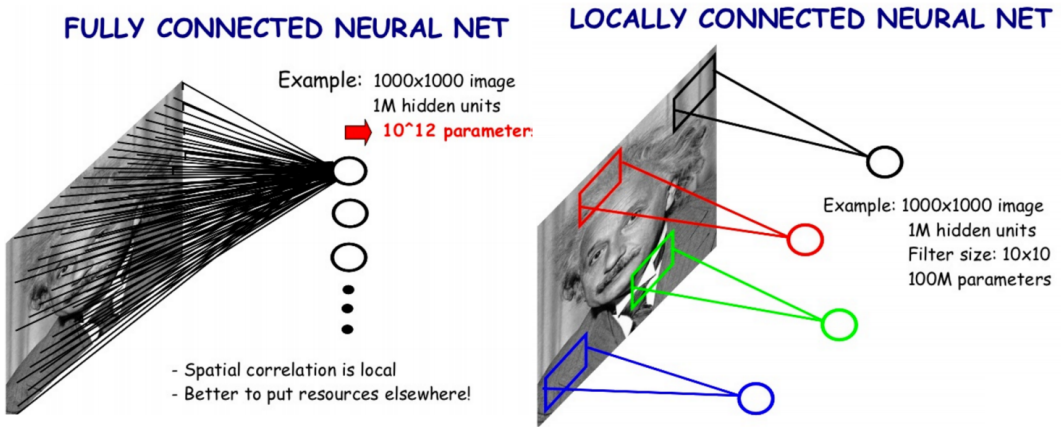
- Until now, each layer of the network contained at fully connected layers
- Unfortunately, there are great drawbacks with such an approach



- Each hidden unit is connected to each input unit;
- There is high redundancy in these weights :
  - In the above example, 65 million weights are required;

Introduction

- For many types of data with **grid-like topological structures** (eg. images), it is not necessary to have so many weights
- For these data, the **convolution** operation is often extremely useful
- Reduces the number of parameters to train
  - Training is faster
  - Convergence is easier : smaller parameter space



- A neural network with convolution operations is known as a **Convolutional Neural Network** (CNN)

Introduction - some history

- "Neocognitron" of Fukushima* : first to incorporate notion of receptive field into a neural network, based on work on animal perception of Hubert and Weisel†
- Yann LeCun first to propose **back-propagation** for training convolutional neural networks‡
  - Automatic learning of parameters instead of hand-crafted weights
  - However, training was very long : required 3 days (in 1990)

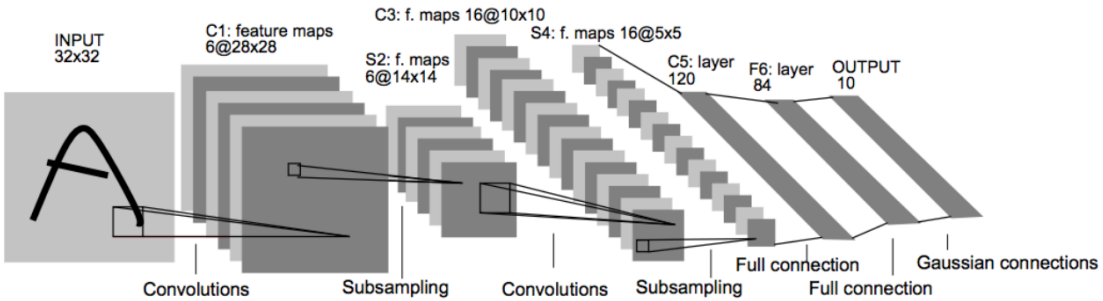


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

* ***Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position***, *Fukushima, K., Biological Cybernetics, 1980*

† ***Receptive fields and functional architecture of monkey striate cortex***, *Hubel, D. H. and Wiesel, T. N, 1968*

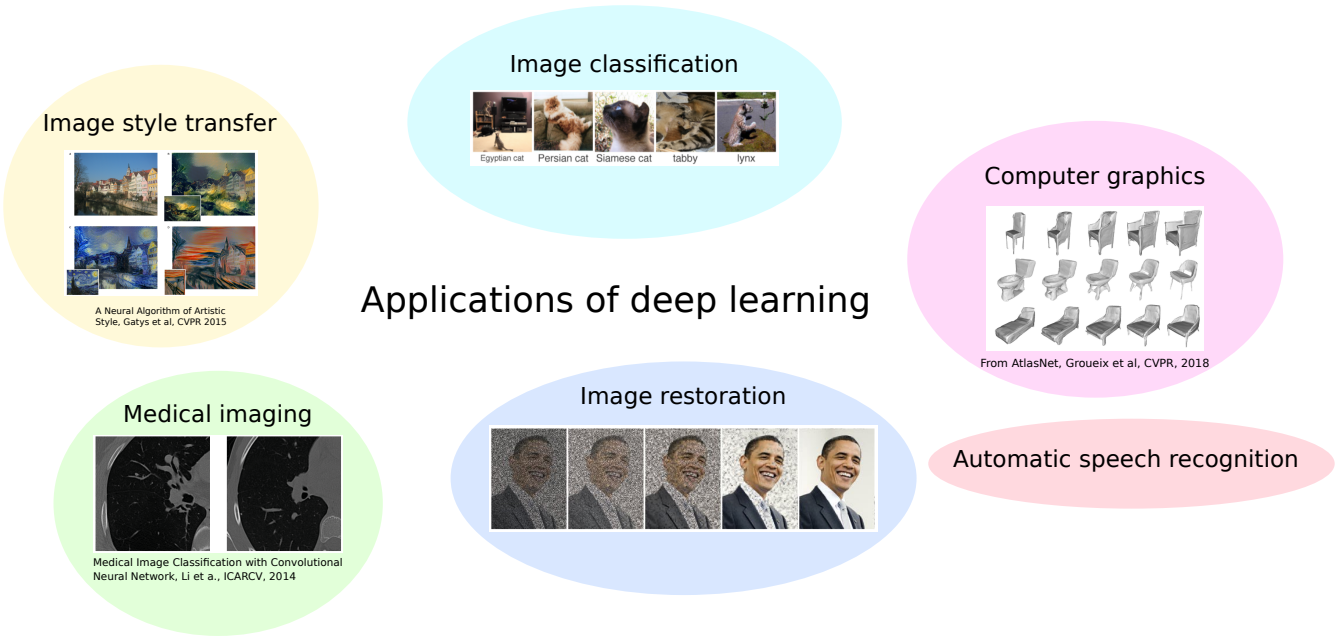
‡ ***Backpropagation Applied to Handwritten Zip Code Recognition***, *LeCun, Y. et al., AT&T Bell Laboratories*

Introduction - some history

- In the years 1998-2012, research continued on shallow and deep neural networks, but other machine learning approaches were preferred (GMMs, SVMs etc.)
- In 2012, Alex Krizhevsky et al. used **Graphics Processing Units** (GPUs) to carry out backpropagation on a very deep convolutional neural network
  - Greatly outperformed classic approaches in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)
- GPUs turned out to be very efficient for training neural nets (lots of parallel computations)
- Signalled the beginning of deep learning revolution

Introduction - some history

- In the past years, CNNs have completely revolutionised many domains
- CNNs produce competetive/best results for most problems in image processing and computer vision

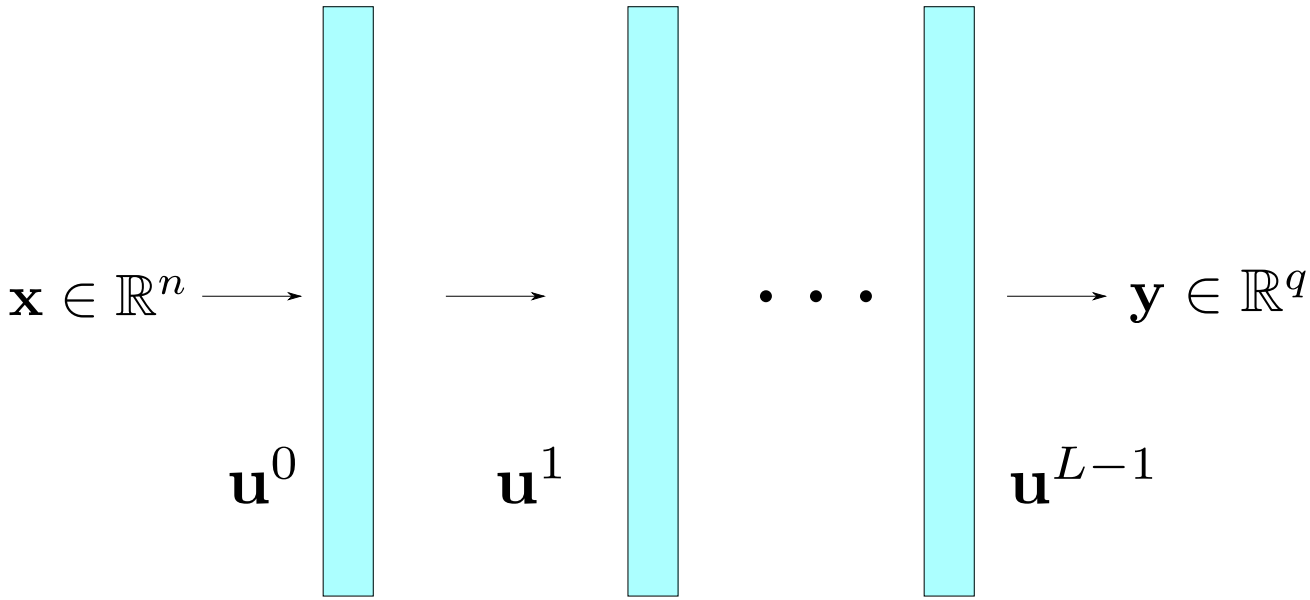


- Being applied to an ever-increasing number of problems

Introduction - some notation

Notations

- $\mathbf{x} \in \mathbb{R}^n$: input vector
- $\mathbf{y} \in \mathbb{R}^q$: output vector
- $\mathbf{u}_\ell$: feature vector at layer ℓ
- θ_ℓ : network parameters at layer ℓ



Neural network with L layers

Introduction

- A "**Convolutional Neural Network**" (CNN) is simply a concatenation of :
  1. Convolutions (filters)
  2. Additive biases
  3. Down-sampling ("Max-Pooling" etc.)
  4. Non-linearities

- In this lesson, we will be mainly concentrating on **convolutional** and **down-sampling** layers

Convolutional Layers

Convolution operator

Let f and g be two integrable functions. The **convolution operator** $*$ takes as its input two such functions, and outputs another function $h = f * g$, which is defined at any point $t \in \mathbb{R}$ as :

$$h(t) = (f * g)(t) = \int_{-\infty}^{+\infty} f(\tau) g(t - \tau) d\tau.$$

- Intuitively, the function h is defined as the inner product between f and a *shifted* version of g;

Convolutional Layers

- In many practical applications, in particular for CNNs, we use the **discrete convolution** operator, which acts on discretised functions;

Discrete convolution operator

Let f_n and g_n be two summable series, with $n \in \mathbb{Z}$. The discrete convolution operator is defined as :

$$(f * g)(n) = \sum_{i=-\infty}^{+\infty} f(i)g(n-i)$$

- Intuitively, the function h is defined as the inner product between f and a *shifted* version of g
- In practice, the filter is of small spatial support, around 3×3, or 5×5
- Therefore, only a **small number** of parameters need to be trained (9 or 25 for these filters)

Properties of convolution

1. Associativity : $(f * g) * h = f * (g * h)$
2. Commutativity : $f * g = g * f$
3. Bilinearity : $(\alpha f) * (\beta g) = \alpha\beta(f * g)$, for $(\alpha, \beta) \in \mathbb{R} \times \mathbb{R}$
4. Equivariance to translation : $(f * (g + \tau))(t) = (f * g)(t + \tau)$

Convolutional Layers

Associativity, commutativity

- Associativity+commutativity implies that we can carry out convolution in any order
- There is no point in having two or more consecutive convolutions
  - This is true in fact for any linear map

Equivariance to translation

- Equivariance implies that the convolution of any shifted input $(f + \tau) * g$ contains the **same information** as $f * g$ [†]
- This is useful, since we want to detect objects **anywhere in the image**

[†] if we forget about border conditions for a moment

Convolutional Layers - 2D Convolution

- Most often, we are going to be working with **images**
- Therefore, we require a 2D convolution operator : this is defined in a very similar manner to 1D convolution :

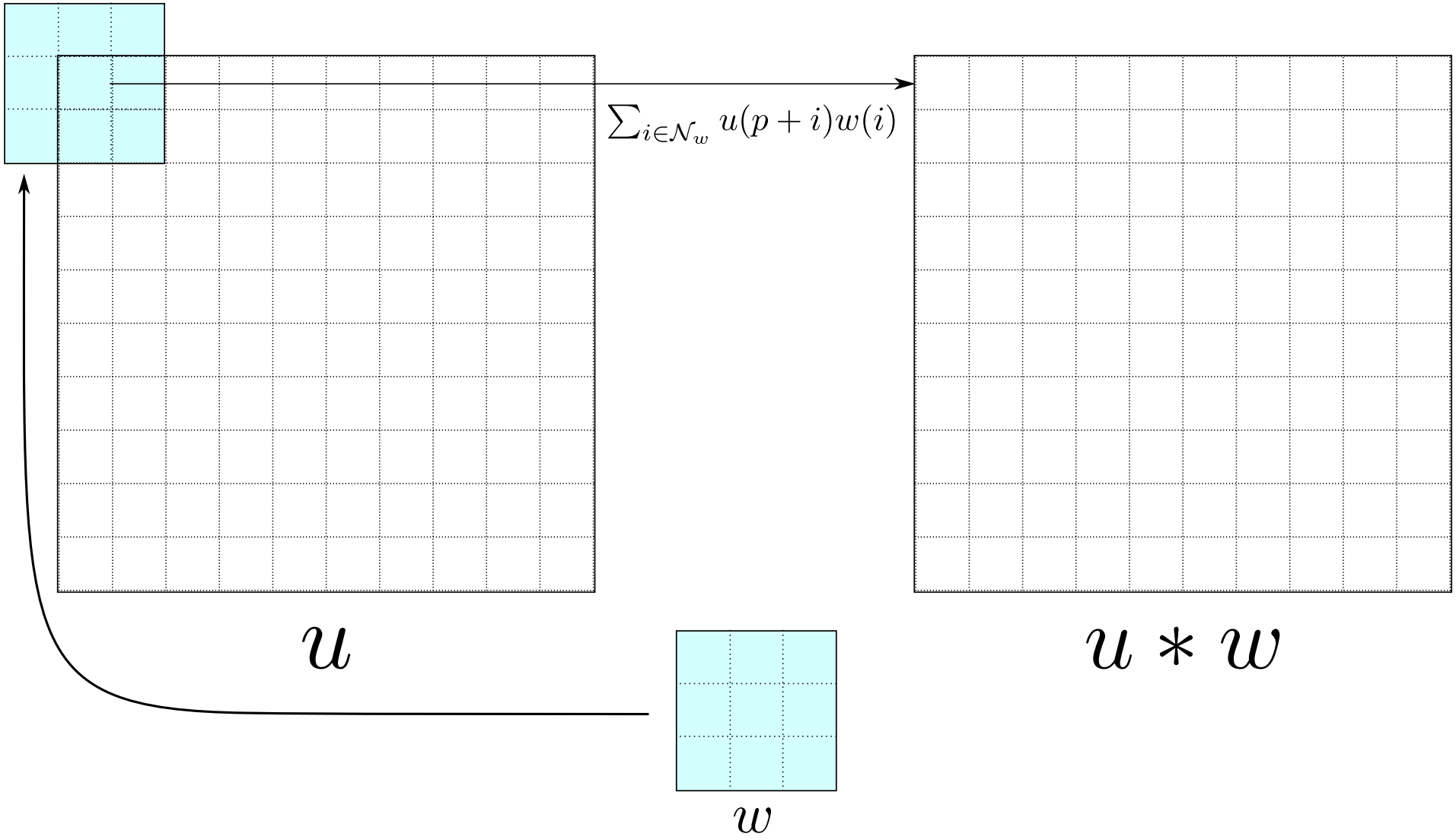
2D convolution operator

$$(f * g)(s,t) = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} f(i,j)g(s-i,t-j)$$

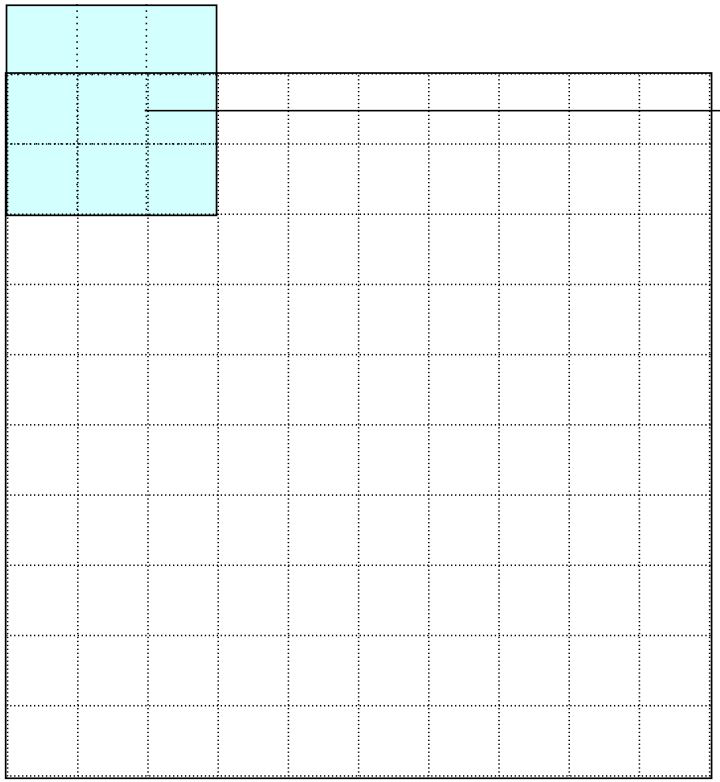
Important remarks for the rest of the lesson!

- We are going to denote the **filters with $\mathbf{w}$**
- For lighter notation, we write $\mathbf{w}(i) =: w_i$ (and the same for x_i etc.)

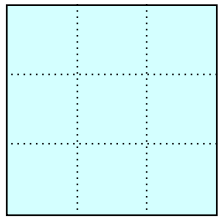
Convolutional Layers : Visual Illustration



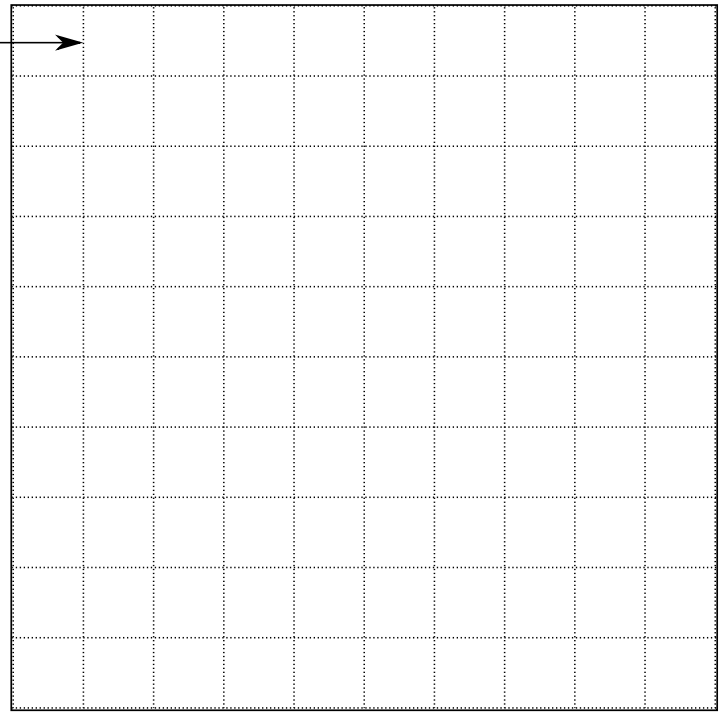
Convolutional Layers : Visual Illustration



u

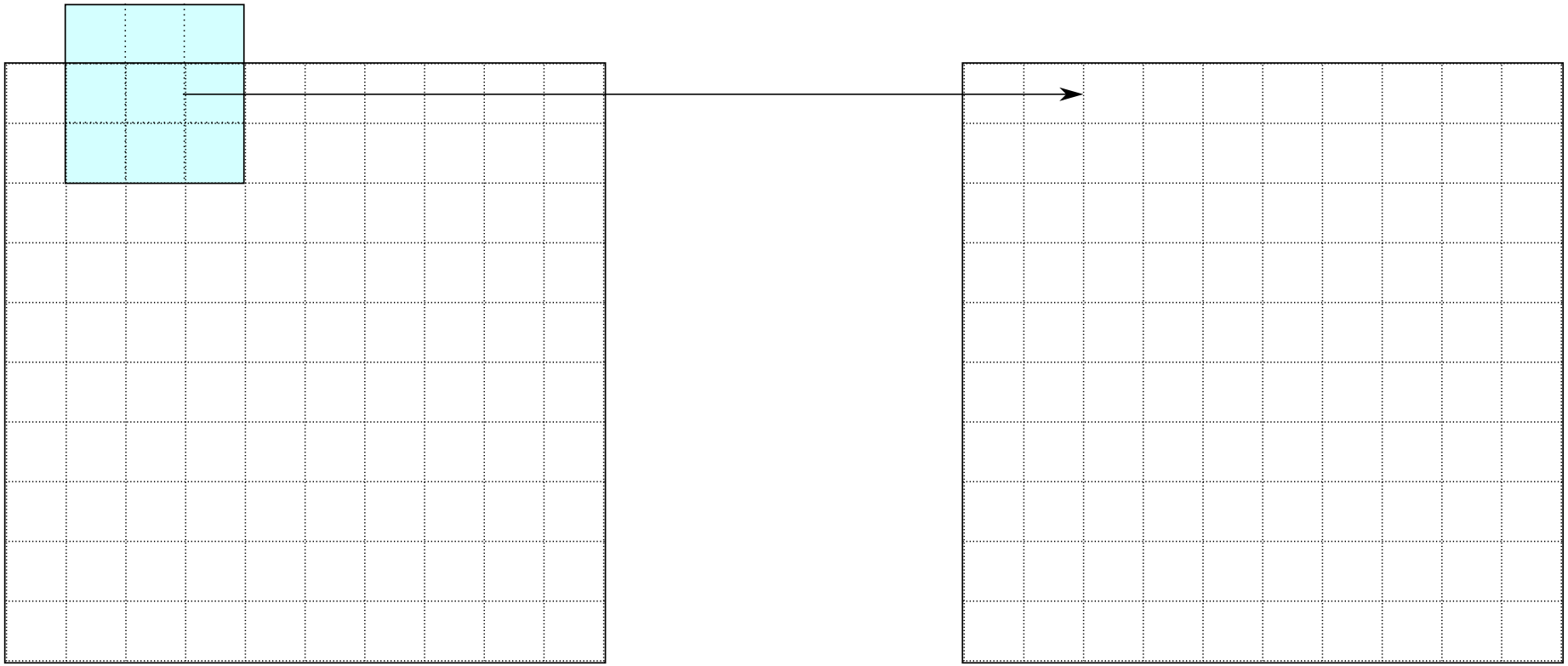


w



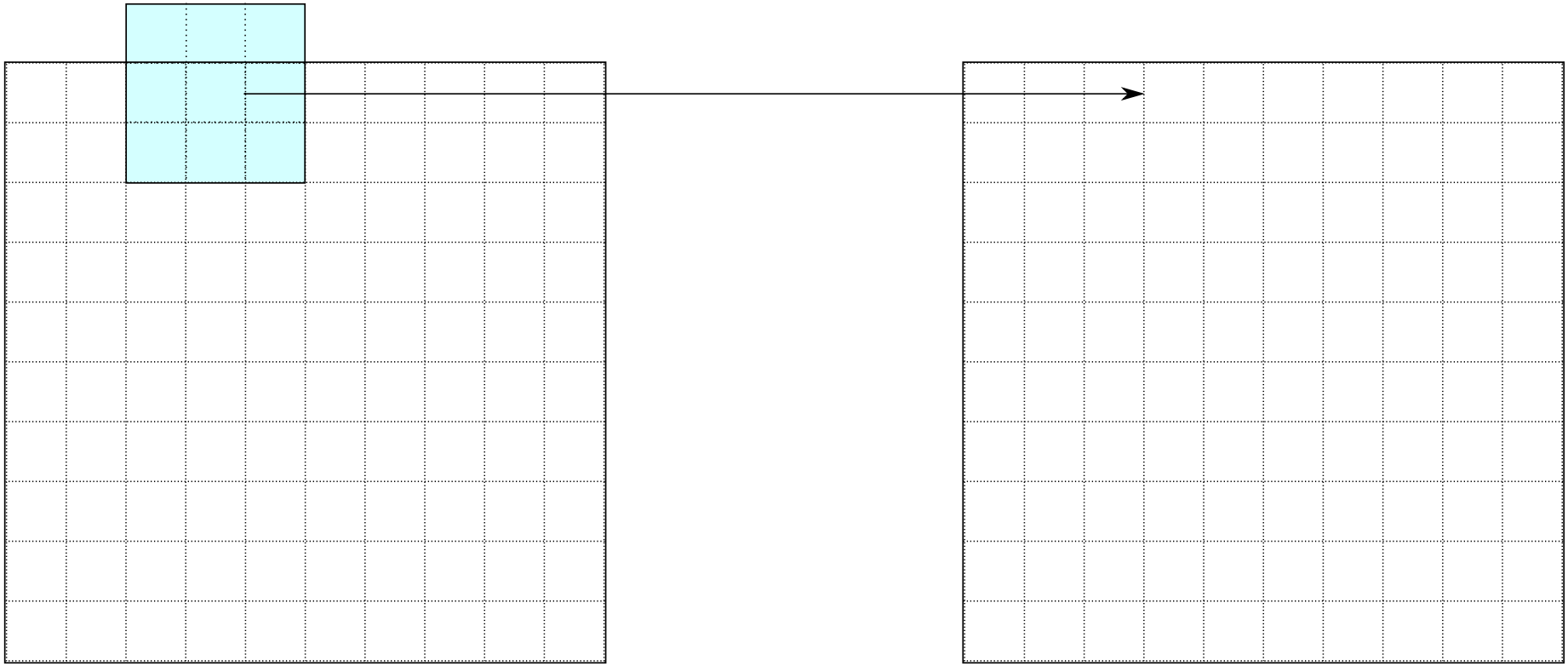
$u * w$

Convolutional Layers : Visual Illustration

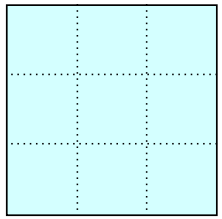


u $u * \varphi$

Convolutional Layers : Visual Illustration



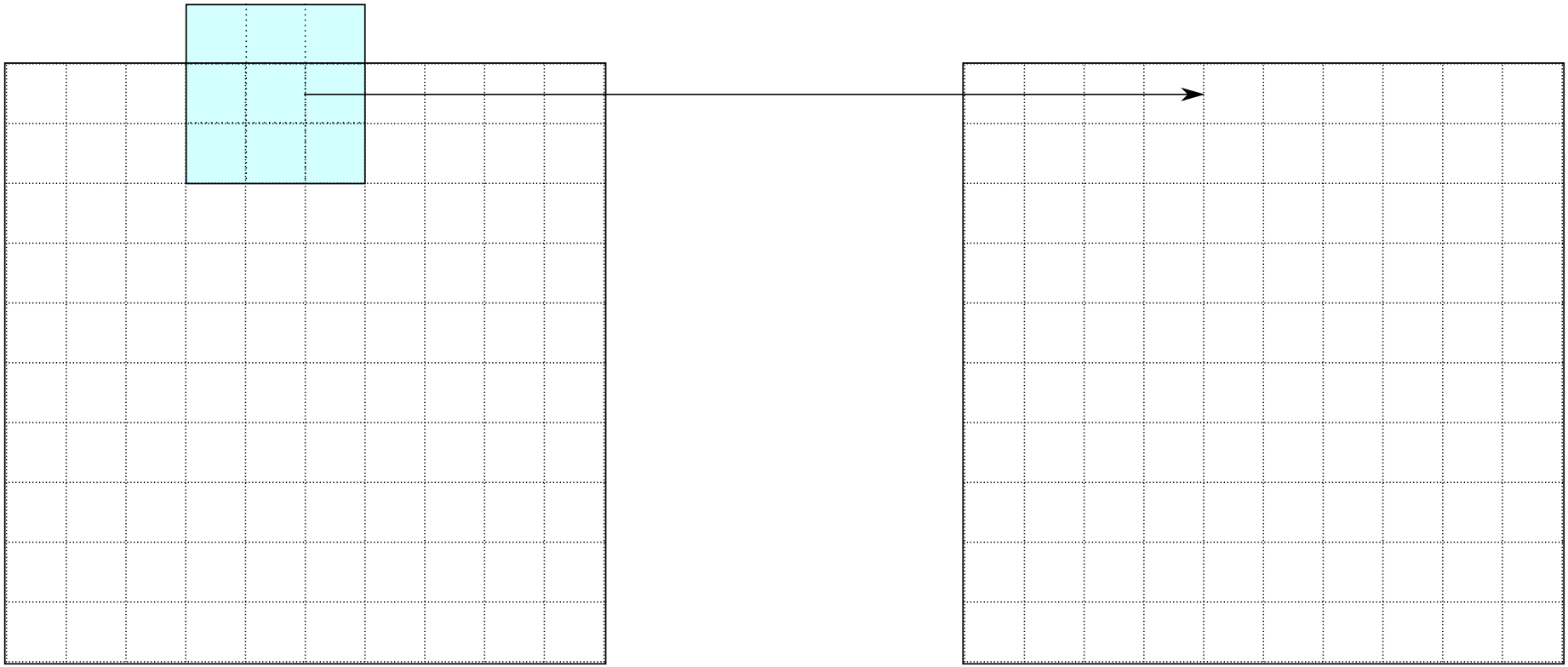
u



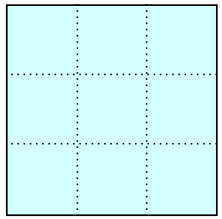
w

$u * w$

Convolutional Layers : Visual Illustration



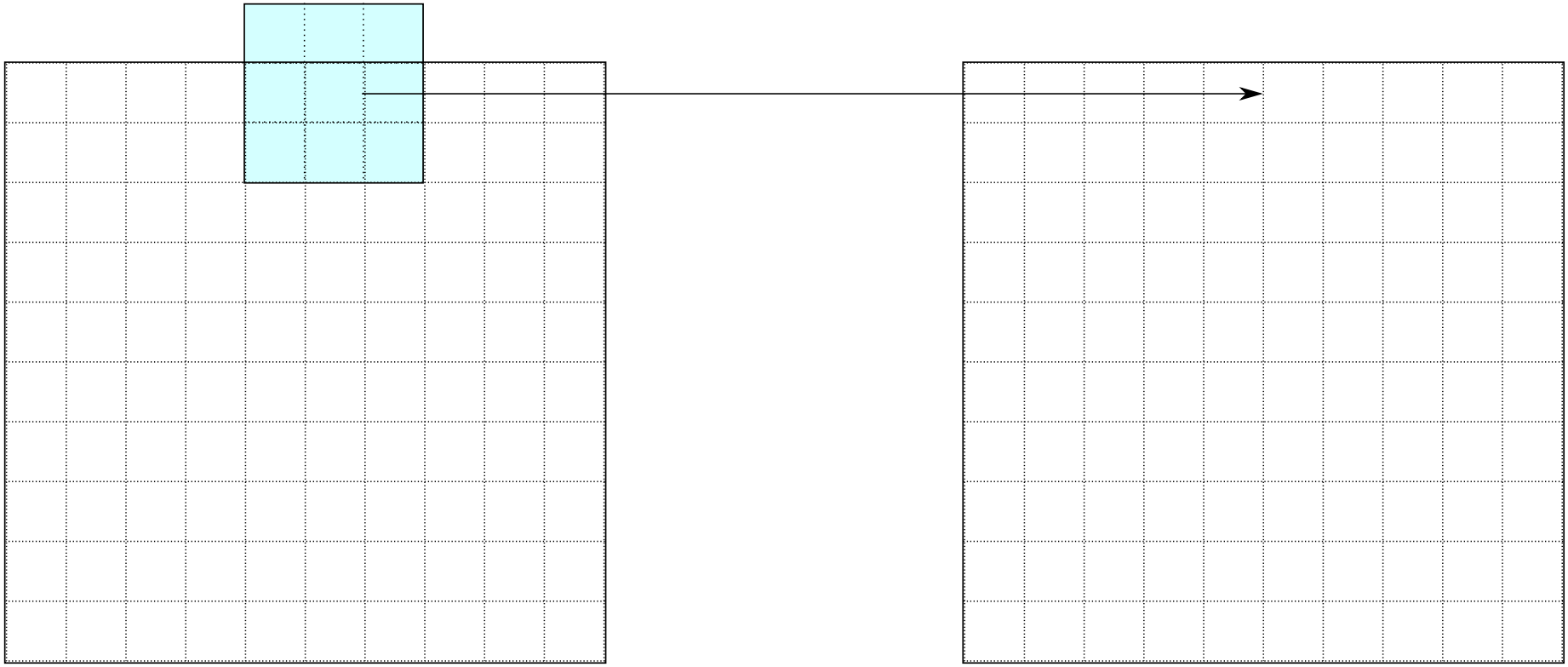
u



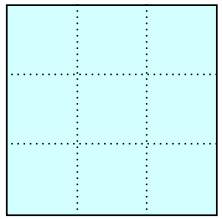
w

$u * w$

Convolutional Layers : Visual Illustration



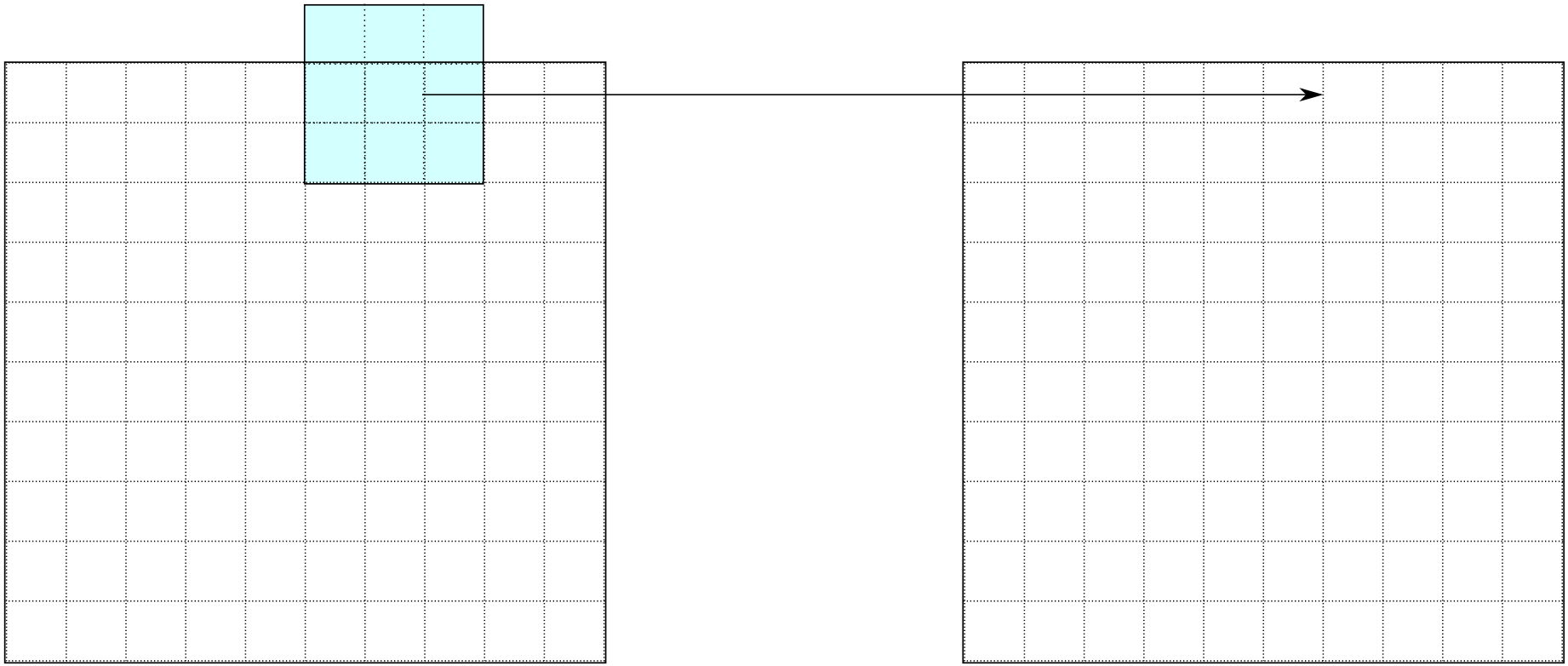
u



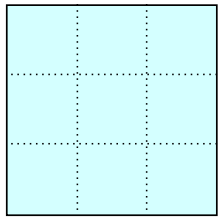
w

$u * w$

Convolutional Layers : Visual Illustration



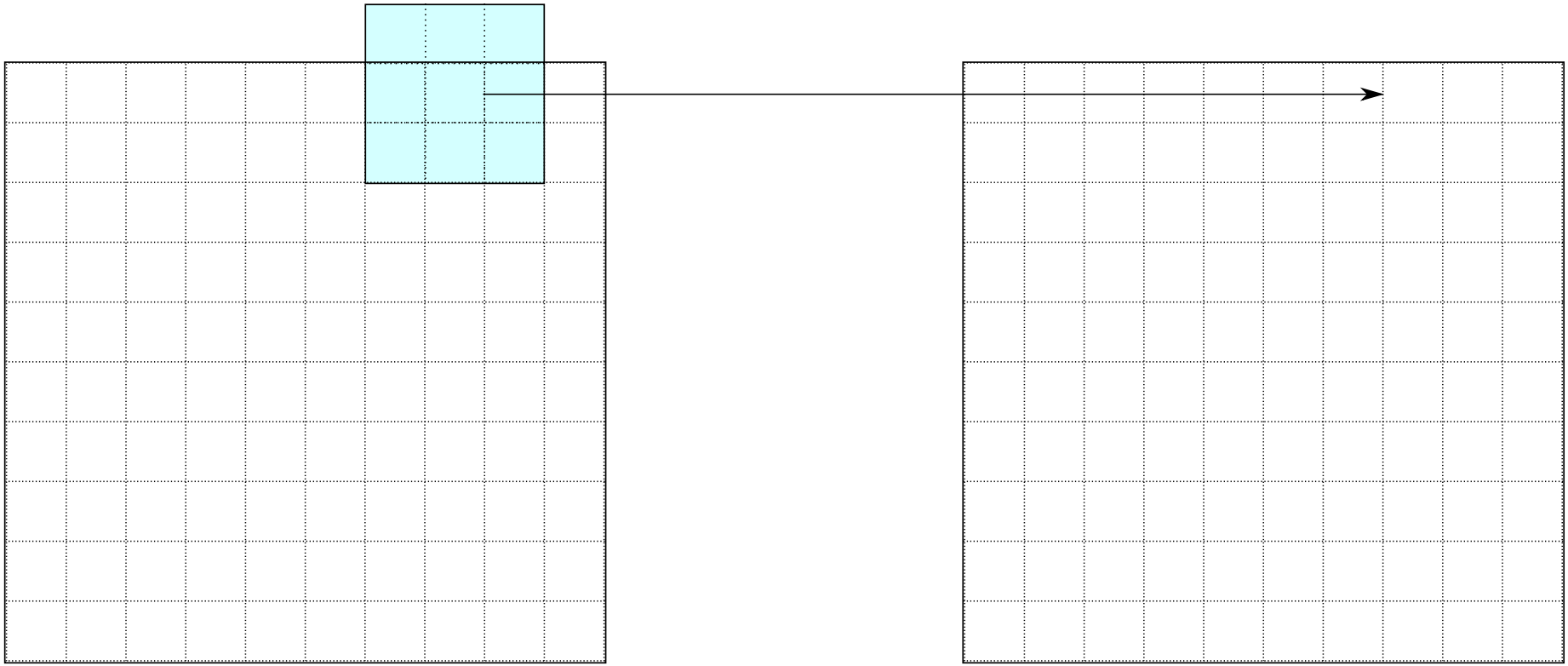
u



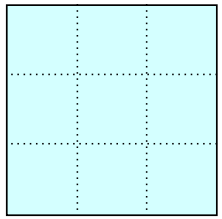
w

$u * w$

Convolutional Layers : Visual Illustration



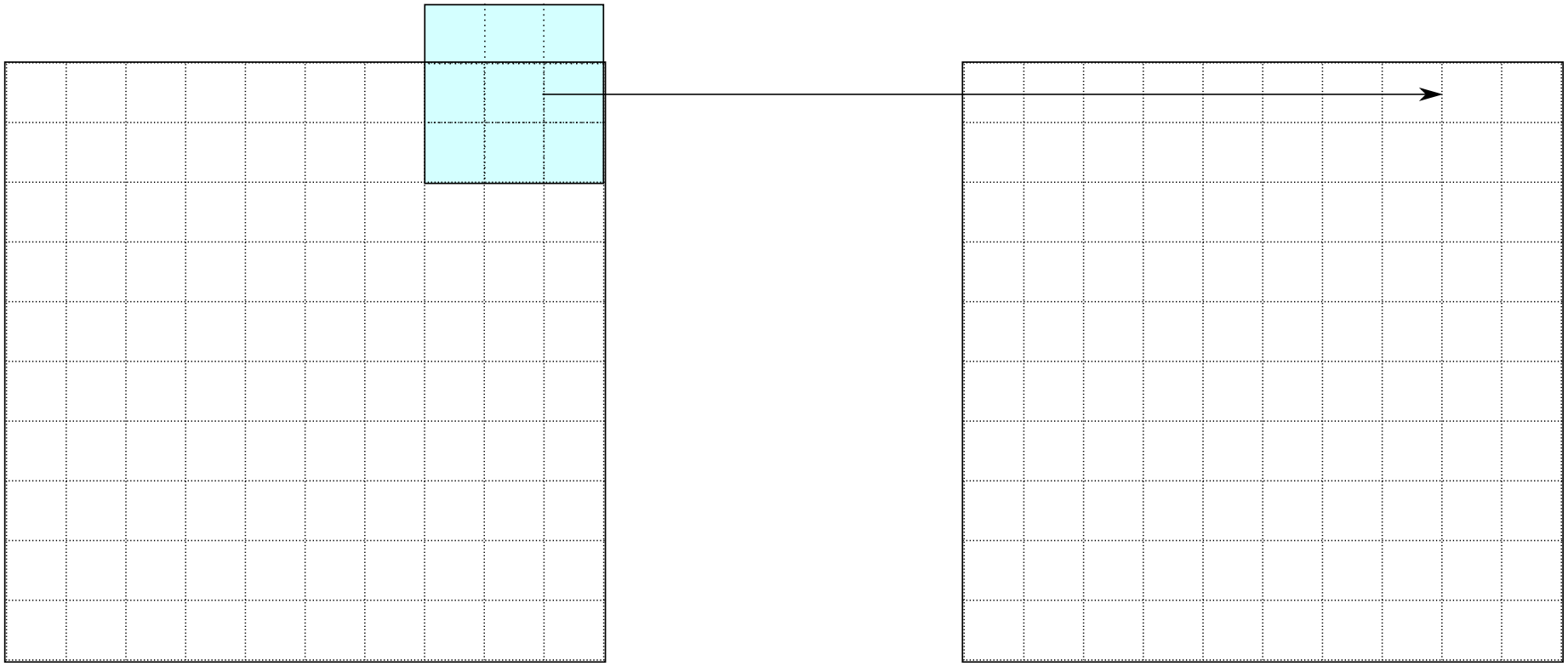
u



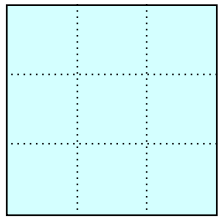
w

$u * w$

Convolutional Layers : Visual Illustration



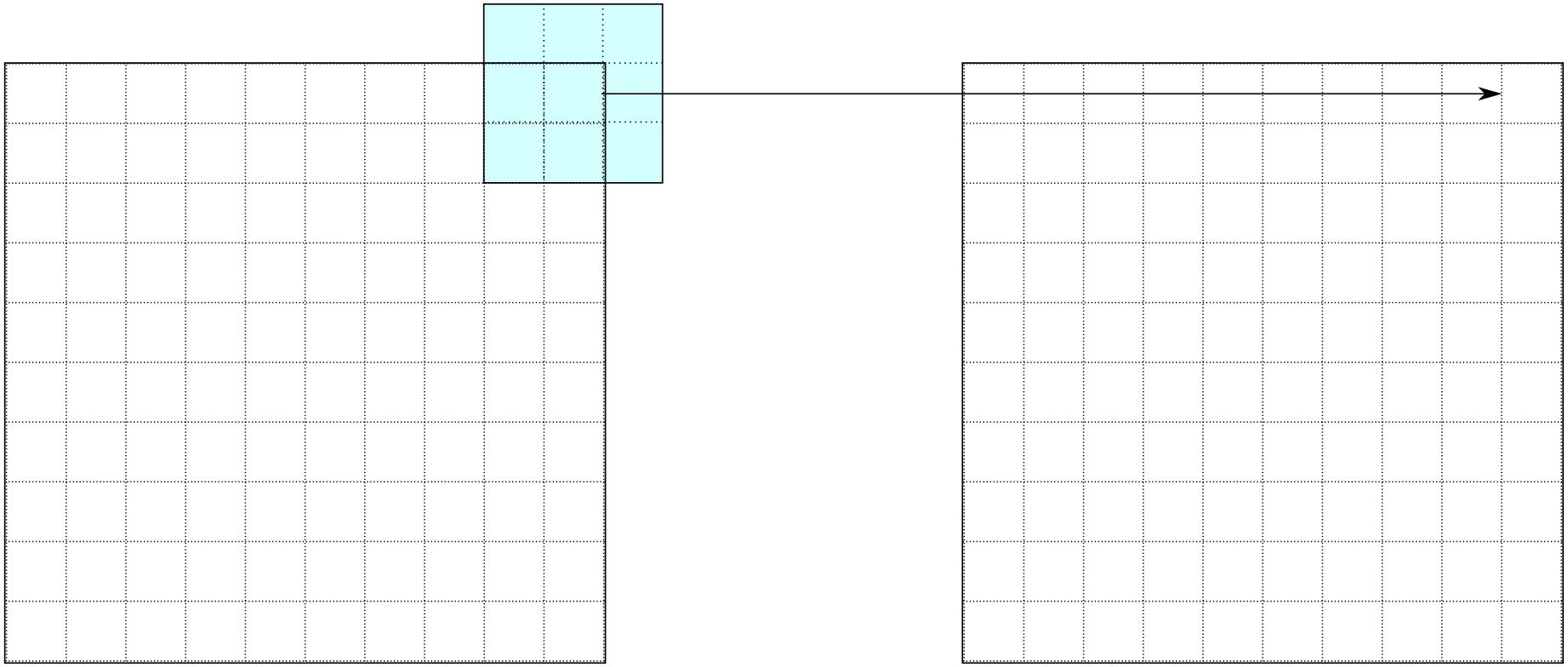
u



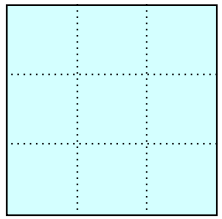
w

$u * w$

Convolutional Layers : Visual Illustration



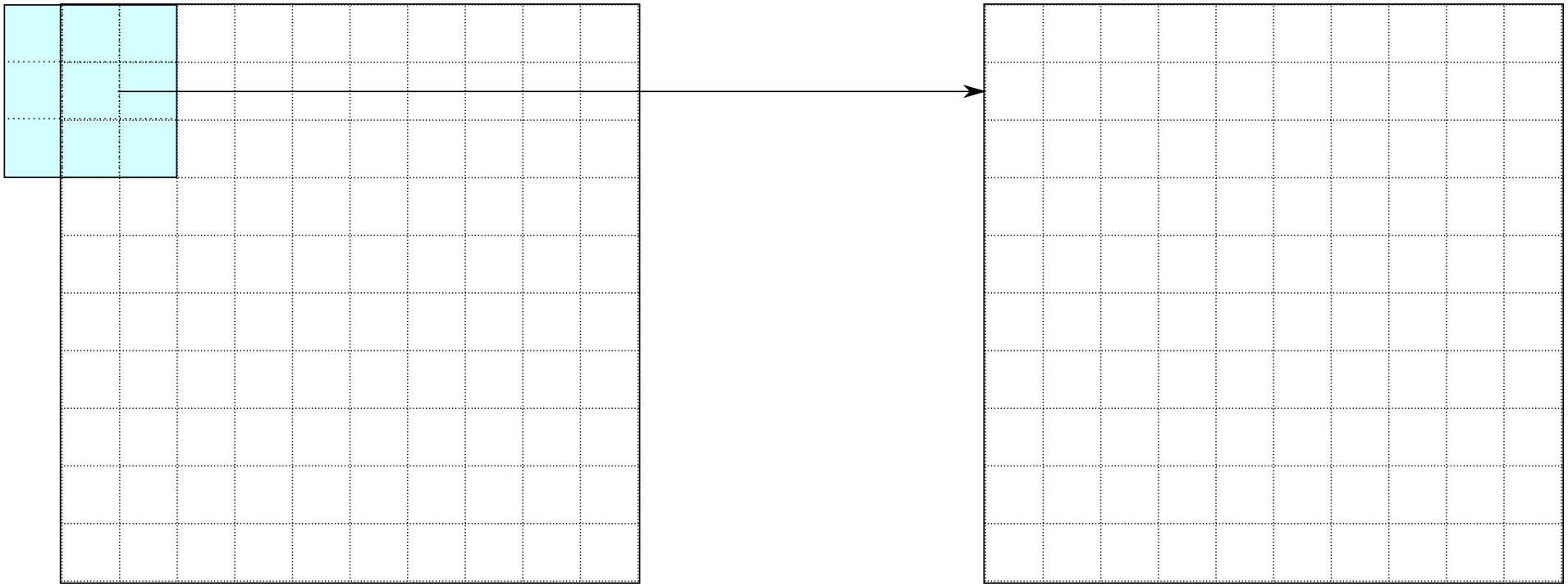
u



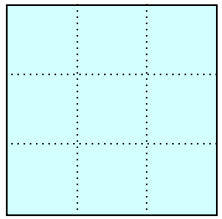
w

$u * w$

Convolutional Layers : Visual Illustration



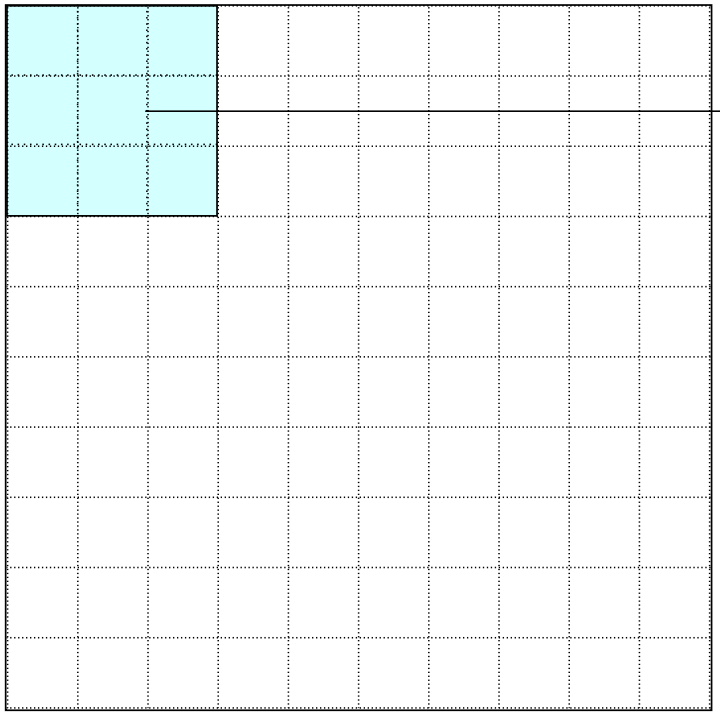
u



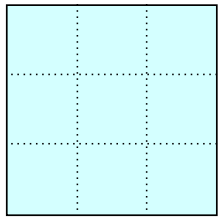
w

$u * w$

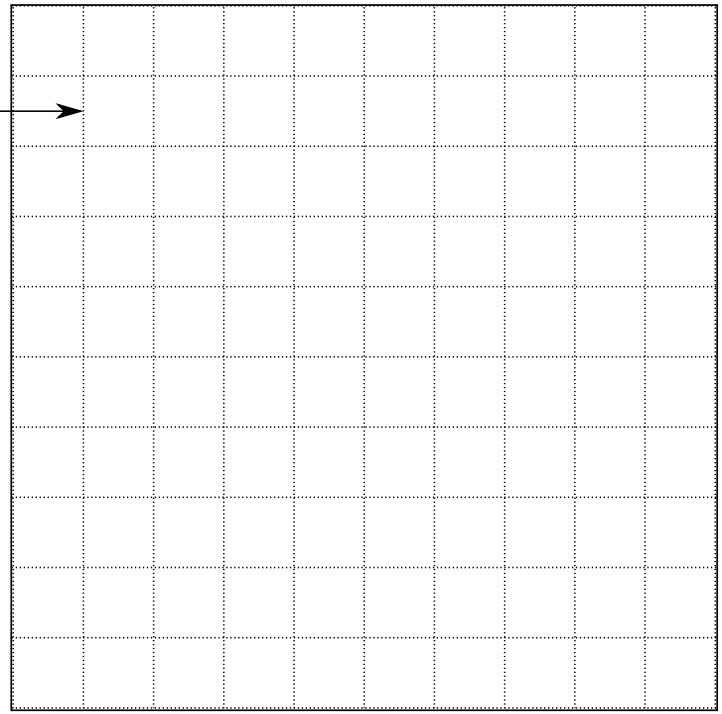
Convolutional Layers : Visual Illustration



u

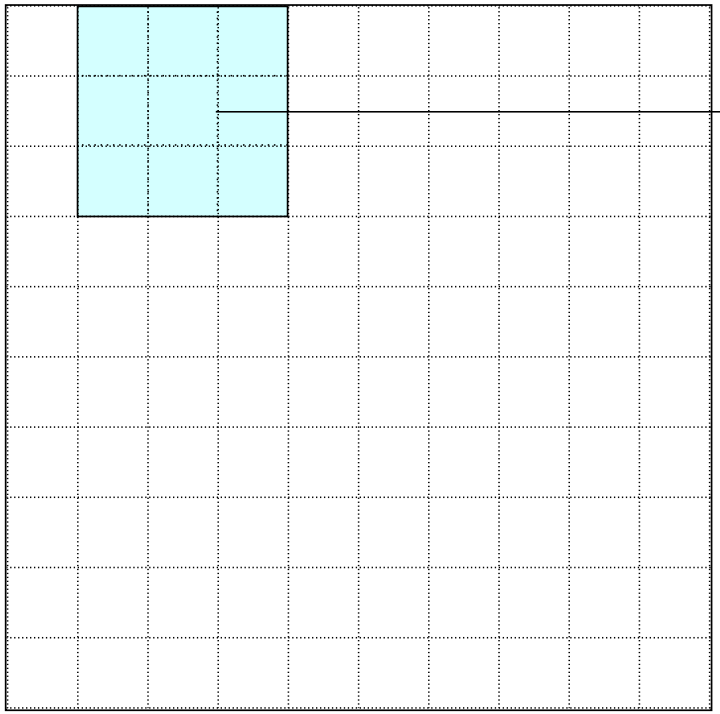


w

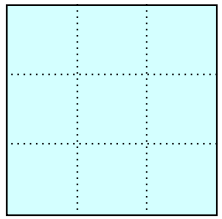


$u * w$

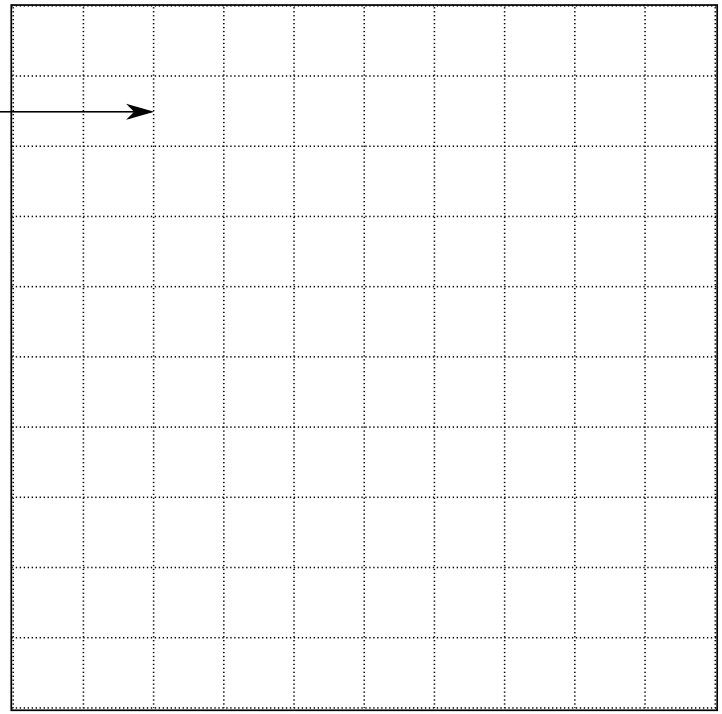
Convolutional Layers : Visual Illustration



u



w



$u * w$

Convolutional Layers

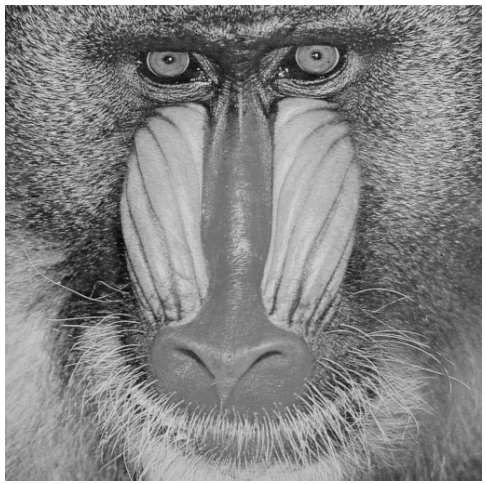
- The filter weights w_i determine what type of "feature" can be detected by convolutional layers;
- Example, sobel filters :

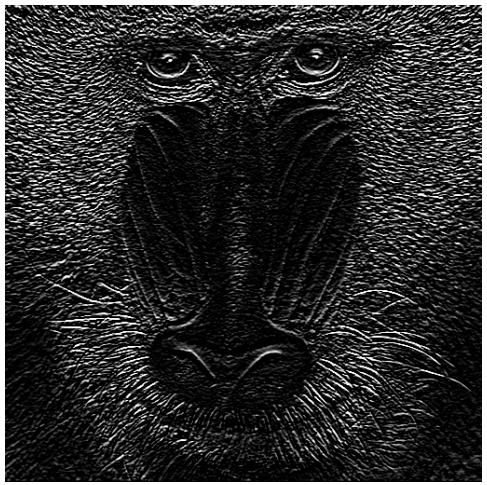
Horizontal edge

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Vertical edge

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$





Convolutional Layers

- We can also write convolution as a matrix/vector product, as in the case of fully connected layers

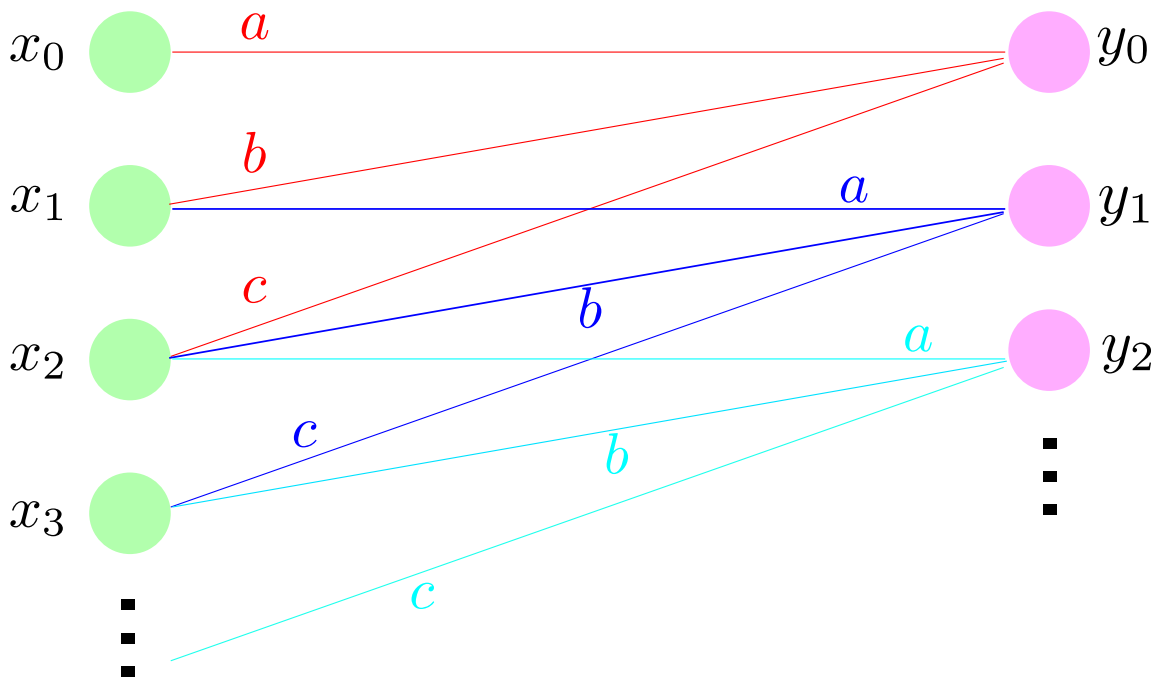
Example : discrete Laplacian operator

$$\mathbf{w} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \rightarrow A_{\mathbf{w}} = \begin{array}{c} \\ \\ \\ \\ \downarrow \\ \\ \\ \end{array} \overbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{n} \rightarrow \begin{pmatrix} 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & & \\ -1 & 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & \\ 0 & -1 & 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & -1 & 4 \end{pmatrix}$$

- This further illustrates the drastic reduction in weight parameters (9 instead of Kn)

Convolutional Layers

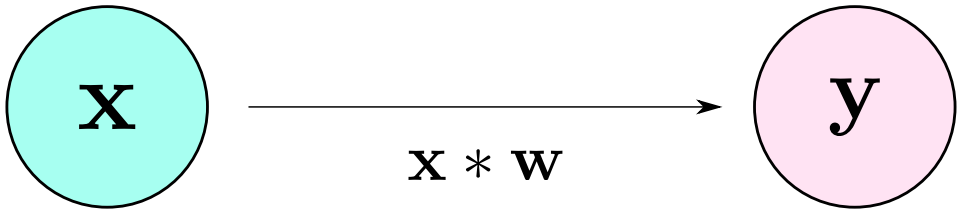
- At this point, it is good to have a more "neural network"-based illustration of CNNs



- We can see two of the main justifications for CNNs
  1. Sparse connectivity
  2. Weight sharing

Convolutional Layers

- Now that we understand convolution, how do we **optimize** a neural network with convolutional layers ? **Back-propagation**
- Consider a layer with just a convolution with $\mathbf{w}$



- We have the derivatives $\frac{\partial \mathcal{L}}{\partial y_i}$ available
- We want to calculate the following quantities :
  - $\frac{\partial \mathcal{L}}{\partial w_k}$ and
  - $\frac{\partial \mathcal{L}}{\partial x_i}$ (for further back-propagation)
- We shall use the abbreviation $\frac{\partial \mathcal{L}}{\partial y_i} =: dy_i$

Convolutional Layers

- First, $\frac{\partial \mathcal{L}}{\partial w_k}$. Each element y_i depends on the filter value w_k.
- Therefore, we can consider that the loss is a function of several variables :

$$\mathcal{L} = f\left(x_1, \ldots, x_n, w_1, \ldots, w_K, y_1(x_\cdot, w_\cdot), \ldots, y_m(x_\cdot, w_\cdot)\right)$$

- We use the multi-variate chain rule

Convolutional Layers

$$\frac{\partial \mathcal{L}}{\partial w_k} = \sum_i dy_i \frac{\partial y_i}{\partial w_k}$$ multi-variate chain rule

$$= \sum_i dy_i \frac{\partial (x * w)_i}{\partial w_k}$$

$$= \sum_i dy_i \frac{\partial \left(\sum_j x_j w_{i-j} \right)}{\partial w_k}$$

$$= \sum_i dy_i x_{i-k} = \sum_i dy_i x_{-(k-i)}$$ $k = i - j$

- More compactly : $dw_k = (dy * \text{flip}(x))_k$

Convolutional Layers

- Now, let us calculate $\frac{\partial \mathcal{L}}{\partial x_k}$

$$\frac{\partial \mathcal{L}}{\partial x_k} = \sum_i dy_i \frac{\partial y_i}{\partial x_k} \qquad \text{multi-variate chain rule}$$

$$= \sum_i dy_i \frac{\partial (x * w)_i}{\partial x_k}$$

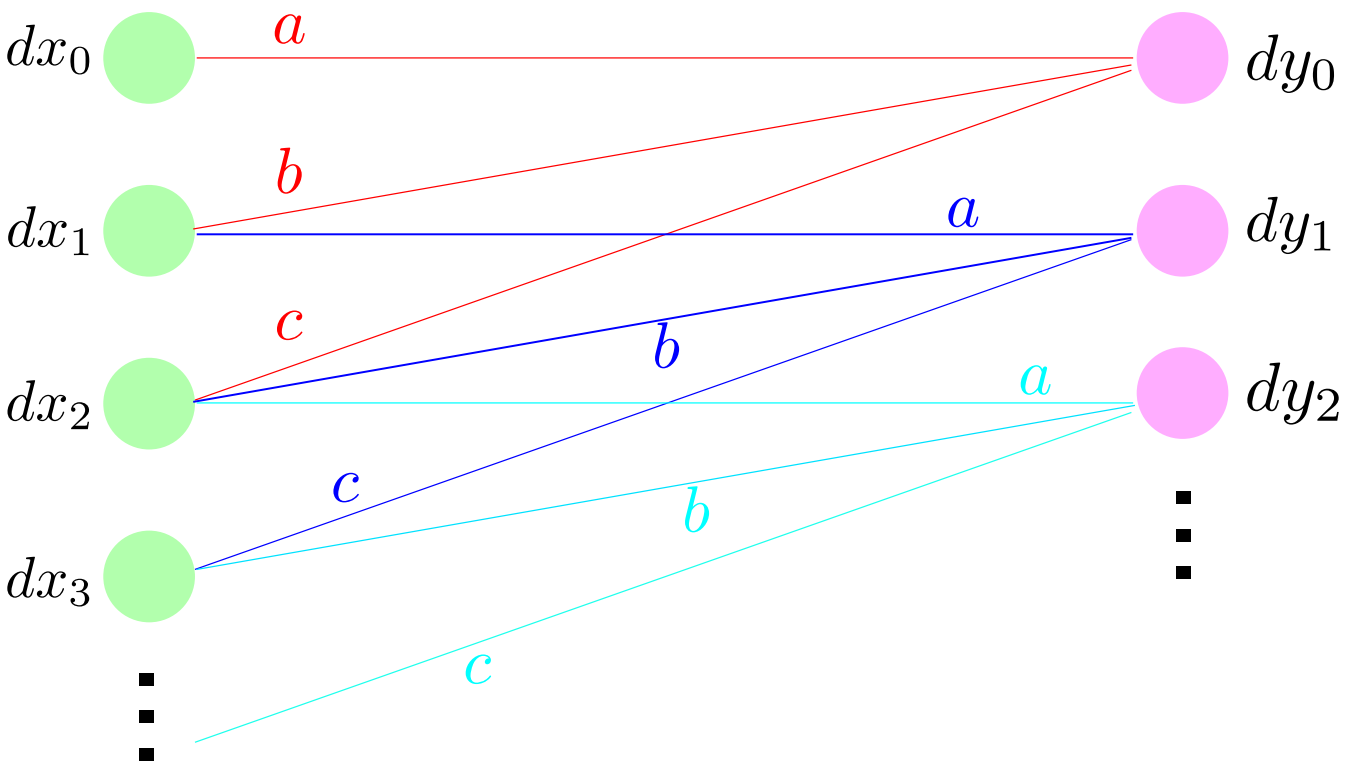
$$= \sum_i dy_i \frac{\partial \left(\sum_j x_j w_{i-j} \right)}{\partial x_k}$$

$$= \sum_i dy_i w_{i-k} = \sum_i dy_i w_{-(k-i)}$$

- More compactly : $dx_k = (dy * \text{flip}(w))_k$

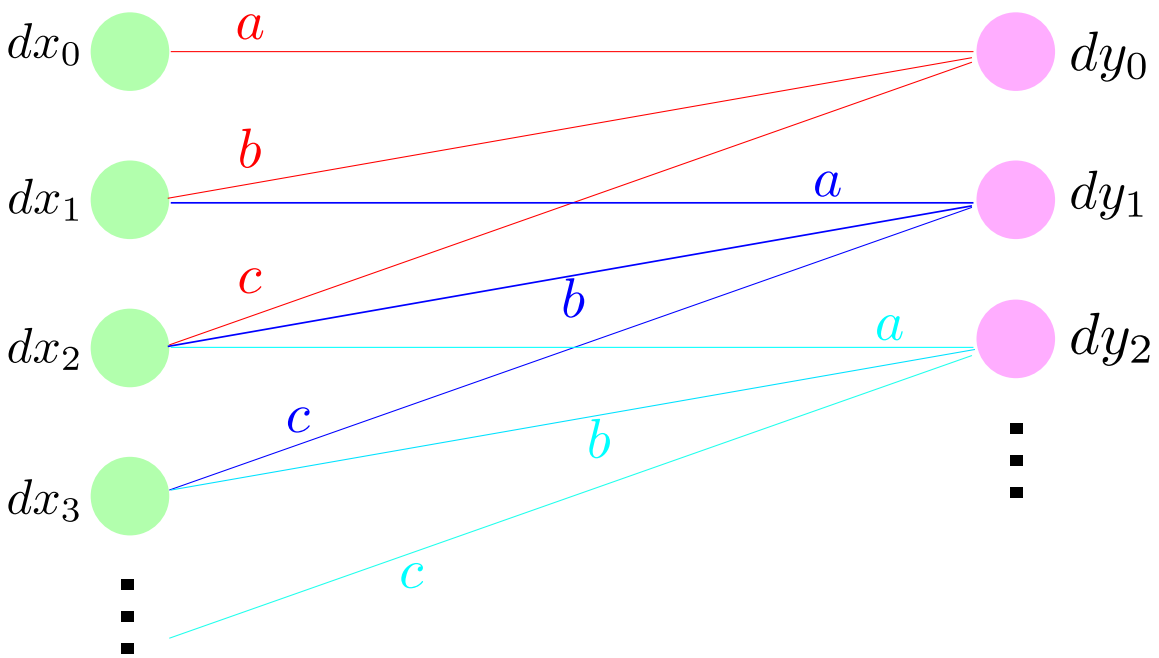
Convolutional Layers

- To make this clearer, take the illustration from above



- Say we want to calculate dx_2

Convolutional Layers



$$dx_2 = dy_0 \frac{\partial y_0}{\partial x_2} + dy_1 \frac{\partial y_1}{\partial x_2} + dy_1 \frac{\partial y_2}{\partial x_2} = dy_0 c + dy_1 b + dy_2 a$$

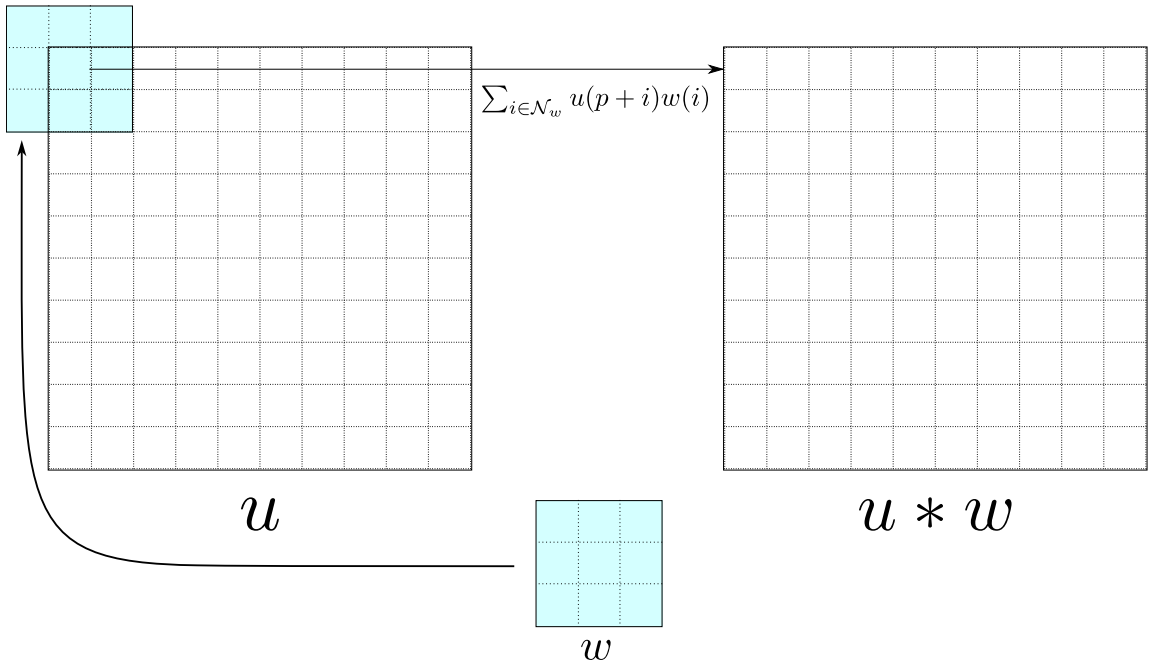
- As we can see, the order of the weights is flipped

Convolutional Layers

- Optimisation of loss w.r.t one parameter w_k involves the entire image
- **Weights are "shared" across the entire image**
- This notion of **weight sharing** is one of the main justifications of using CNNs
- In practice, we do not calculate dw_k and dx_k ourselves, we use the **automatic differentiation** tools of Tensorflow, Pytorch etc.

Convolutional Layers - border conditions

- The convolution operator poses a problem at the borders
- Theoretically, we consider functions defined over an infinite domain, but which have compact support
- In reality, we only have finite vectors/matrices to work on

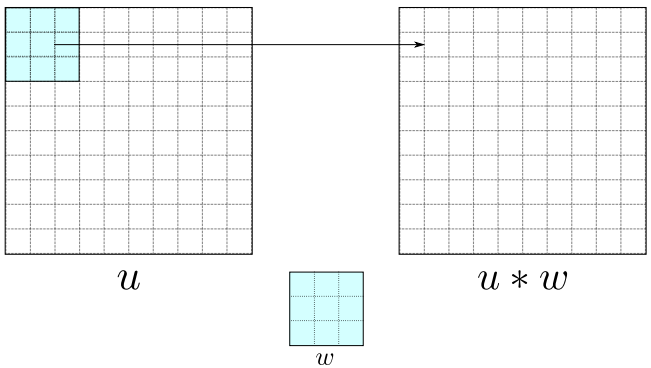


Convolutional Layers - border conditions

Two common approaches to border conditions

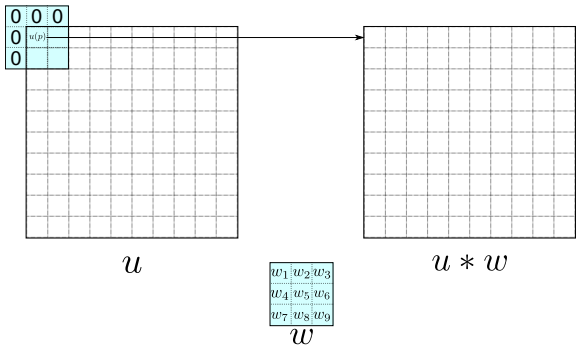
"VALID" approach

- Only take shift/dot products that do not extend beyond $\mathrm{Supp}(u)$
- Output size : $m - |w| + 1$



"SAME" approach

- Keep output size m
- Need to choose values outside of $\mathrm{Supp}(u)$: zero-padding



2D+feature convolution

- Several filters are used per layer, let us say K filters : $\{\mathbf{w}_1, \ldots, \mathbf{w}_K\}$
- The resulting vectors/images are then stacked together to produce the next layer's input $\mathbf{u}^{\ell+1} \in \mathbb{R}^{m \times n \times K}$

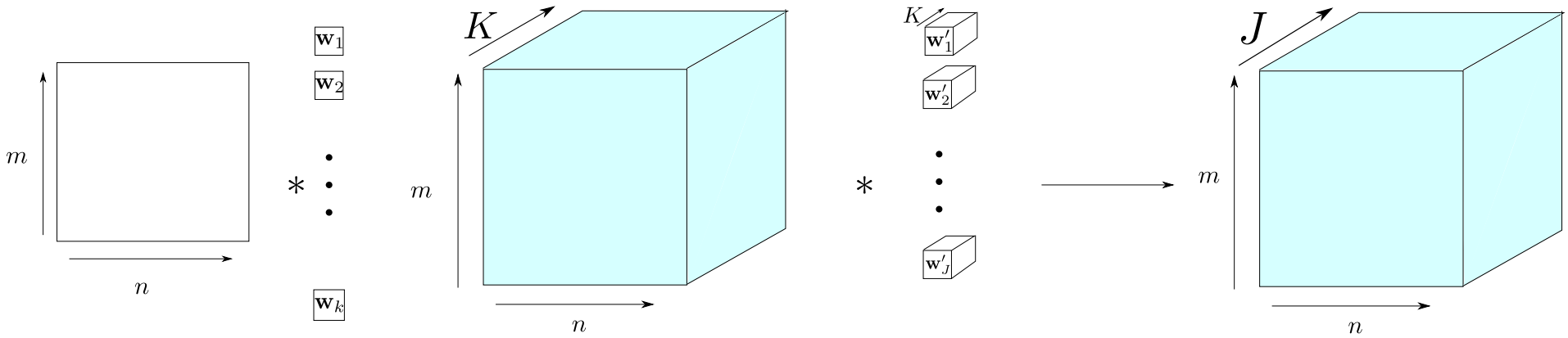
$$\mathbf{u}^{\ell+1} = [\mathbf{u} * \mathbf{w}_1, \ldots, \mathbf{u} * \mathbf{w}_K]$$

- Therefore, the next layer's weights must have a depth of K. The 2D convolution with an image of depth K is defined as

$$(\mathbf{u} * \mathbf{w})_{x,y} = \sum_{i,j,k} \mathbf{u}(i, j, \mathbf{k})\ \mathbf{w}(x - i, y - j, \mathbf{k})$$

Convolutional layers

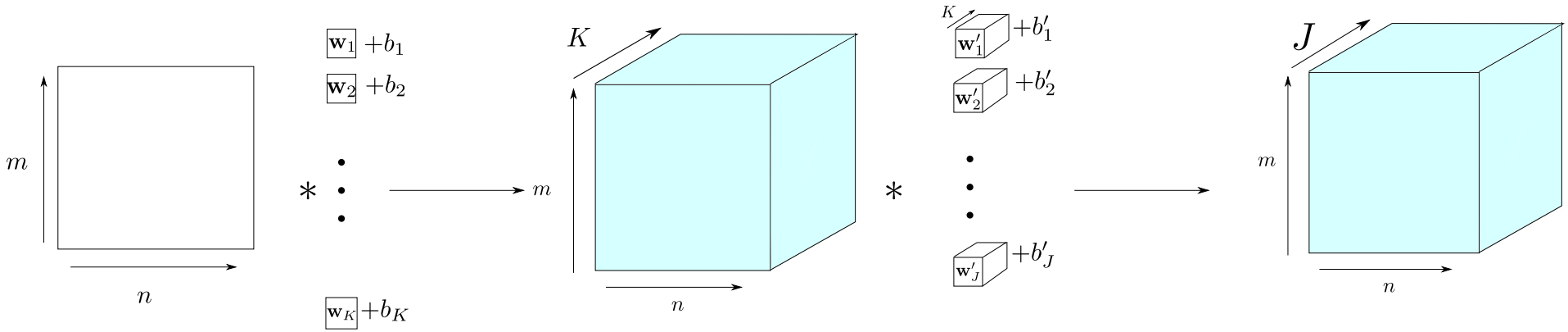
- Illustration of several consecutive convolutional layers with different numbers of filter



- Each layer contains "image" with a depth, where each channel corresponds to a different filter response
- Each layer is a concatenation of several features : rich information

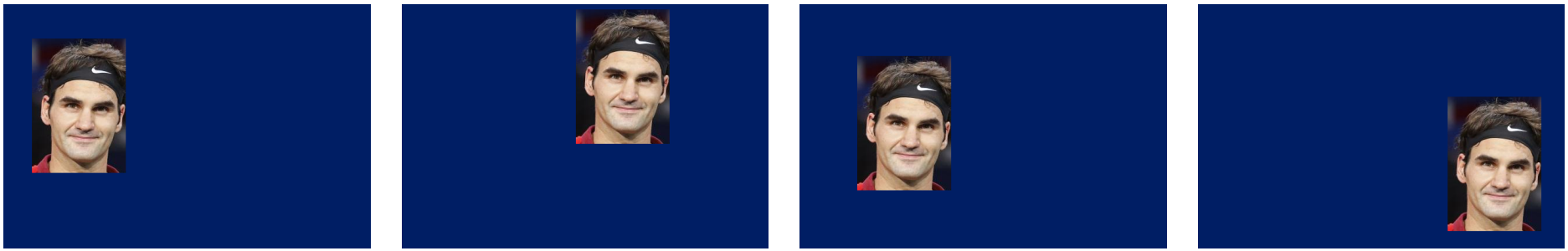
Convolutional layers - a note on Biases

- A note on biases in neural networks : **each output layer is associated with one bias**
- There is **not** one bias per pixel
- This is coherent with the idea of **weight sharing** (bias sharing)



Convolutional Layers

- In many cases, we are primarily interested in **detection**;
- We would like to detect objects **wherever they are in the image**

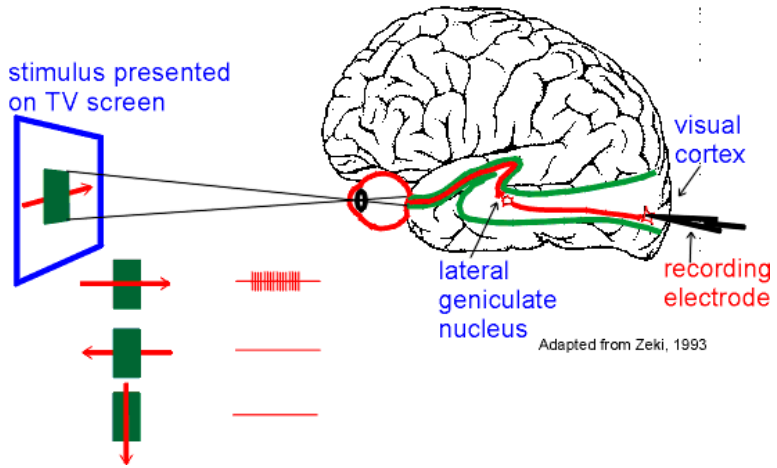


- Formally, we would like to have some **shift invariance property**;
- This is done in CNNs by using **subsampling**, or some variant :
  - Strided convolutions;
  - Max pooling;
- We explain these now;

Down-sampling and the receptive field

The Receptive Field

- Neural networks were initially inspired by the brain's functioning
- Hubel and Weisel[†] showed that the visual cortex of cats and monkeys contained cells which individually responded to different small regions of the visual field
- The region which an individual cell responds to is known as the "receptive field" of that cell



[†] ***Receptive fields and functional architecture of monkey striate cortex***, *Hubel, D. H.; Wiesel, T. N, 1968 Illustration from :* `http://www.yorku.ca/eye/cortfld.htm`

The Receptive Field

- This idea was imitated in convolutional neural networks by adding **down-sampling** operations

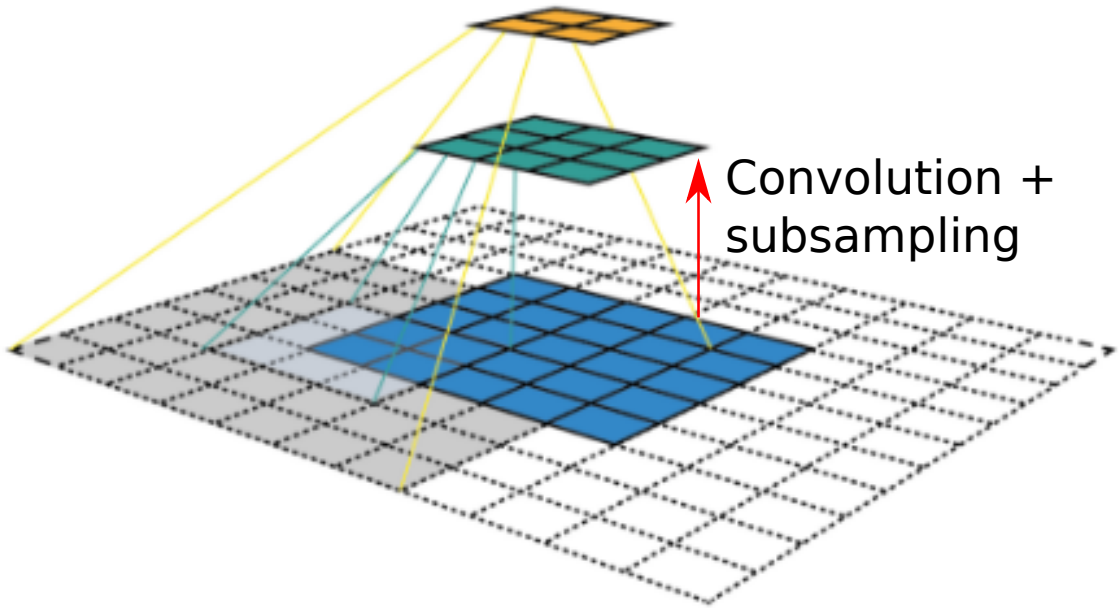


Illustration from : Applied Deep Learning, Andrei Bursuc, https://www.di.ens.fr/~lelarge/dldiy/slides/lecture_7/

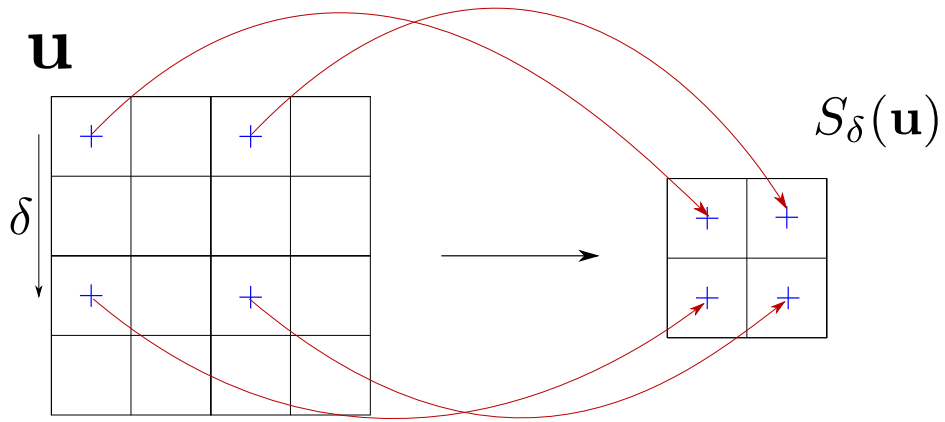
Strided convolution

- **Strided convolution** is simply convolution, followed by **subsampling**

Subsampling operator (for 1D case)

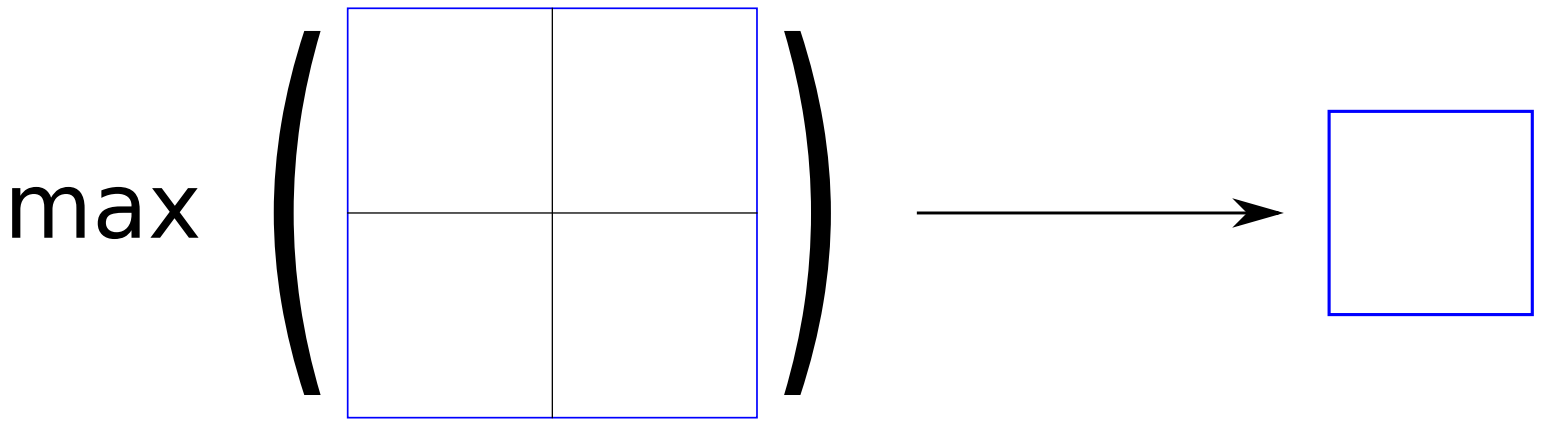
Let $\mathbf{x} \in \mathbb{R}^n$. We define the subsampling step as $\delta > 1$, and the subsampling operator $S_\delta : \mathbb{R}^n \to \mathbb{R}^{\frac{n}{\delta}}$, applied to $\mathbf{x}$, as

$$S_\delta(\mathbf{x})\ (t) = \mathbf{x}(\delta t), \ \text{ for } t = 0 \ldots \frac{n}{\delta}$$



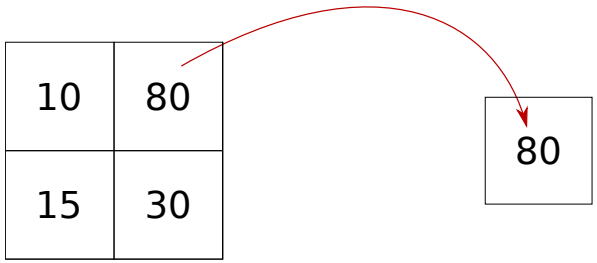
Max pooling

- **Max pooling** subsampling consists in taking the **maximum** value over a certain region
- This maximum value is the new subsampled value
- We will indicate the max pooling operator with S_m

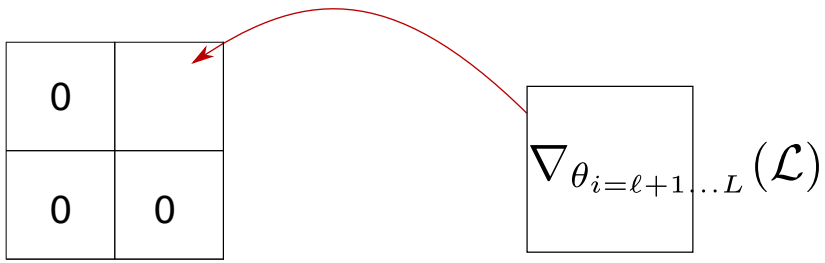


Max pooling

- Back propagation of max pooling only passes the gradient through the **maximum**



Max pooling



Back propagation

Down-sampling

- Conclusion : cascade of convolution, non-linearities and subsampling produces **shift-invariant** classification/detection
- We can detect roger wherever he is in the image !

Convolution + non-linearity +max pooling

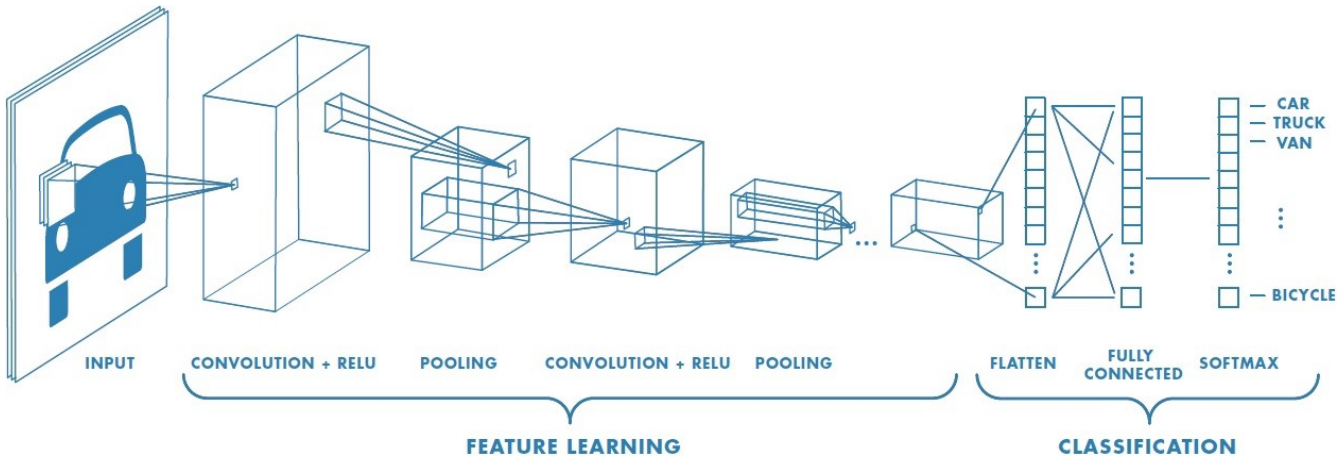
How to Build Your CNN

How to build your CNN ?

- We have looked at the following operations : convolutions, additive biases, non-linearities
- All of these elements make up convolutional neural networks
- However, how do we put these together tp create our own CNN ?
  - **Architecture ?**
  - Programming tools ?
  - Datasets ?

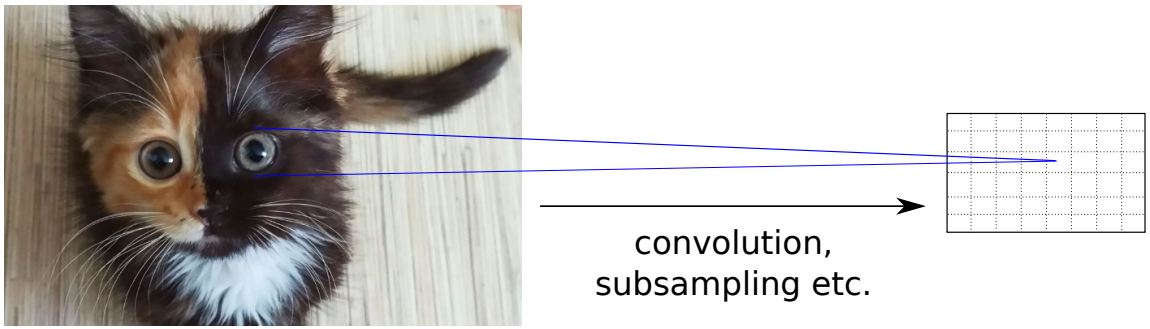
Architecture : vanilla CNN

- Simple classification CNN architecture often consists of a **feature learning section**
  - Convolution $\rightarrow$ biases $\rightarrow$ non-linearities $\rightarrow$ subsampling
  - This continues until a fixed subsampling is achieved
- After this, a **classification section** is used
  - Fully connected layer $\rightarrow$ non-linearity



Architecture

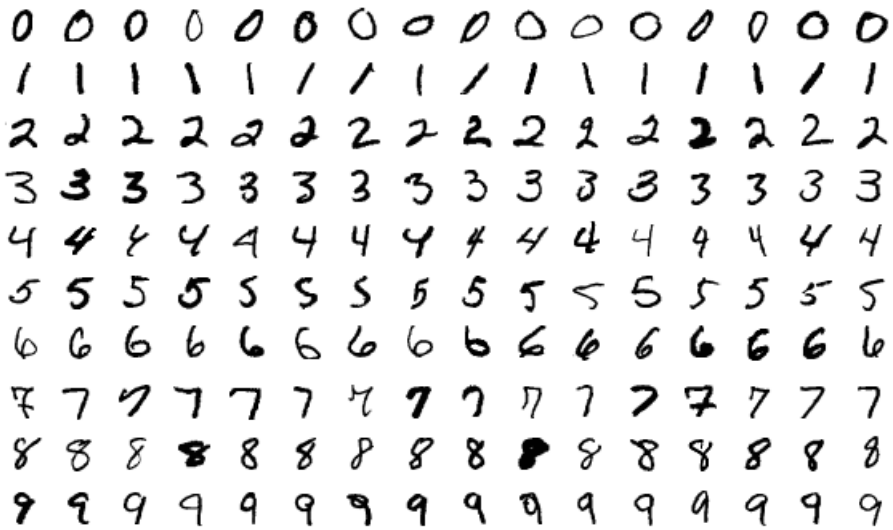
- Central question : how to choose number of layers ?
- Complicated, very little theoretical understanding, currently a hot topic of research
- However : there are a few rules of thumb to follow
  - Receptive field of the **deepest layer should encompass what we consider to be a fundamental brick** of the objects we are analysing



- Set number of layers and subsampling factors according to the problem

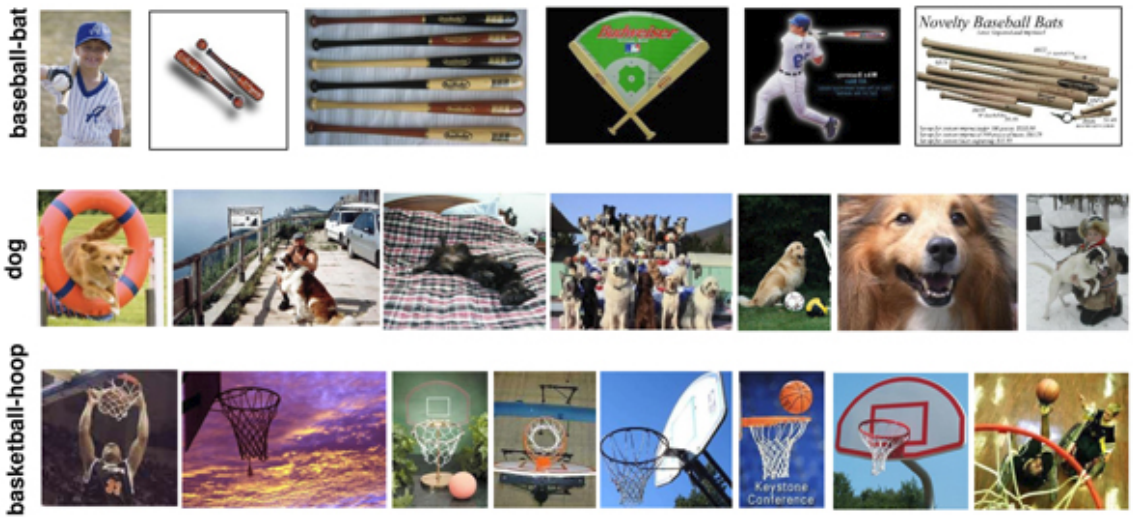
MNIST dataset

- MNIST is a dataset of 60,000 28×28 pixel grey-level images containing hand-written digits
- The digits are centred in the images and scaled to have roughly the same size
- Although quite a "simple" dataset, still used to display performance of modern CNNs



Caltech 101

- Produced in 2003, first major object recognition dataset
- 9,146 images, 101 object categories, each category contains between 40 and 800 images
- Annotations exist for each image : bounding box for the object and a human-drawn outline



ImageNet dataset

- Dataset created in 2009 by researchers from Princeton unverisity
- Very large dataset : 14,197,122 images, hand-annotated
- Used for the ImageNet Large Scale Visual Recognition Challenge, an annual benchmark competition for object recognition algorithms

LeNet (1989/1998)

- Created by Yann LeCun in 1989, goal : to recognise handwritten digits
- Able to classify digits with 98.9% accuracy, used by U.S. government to automatically read digits

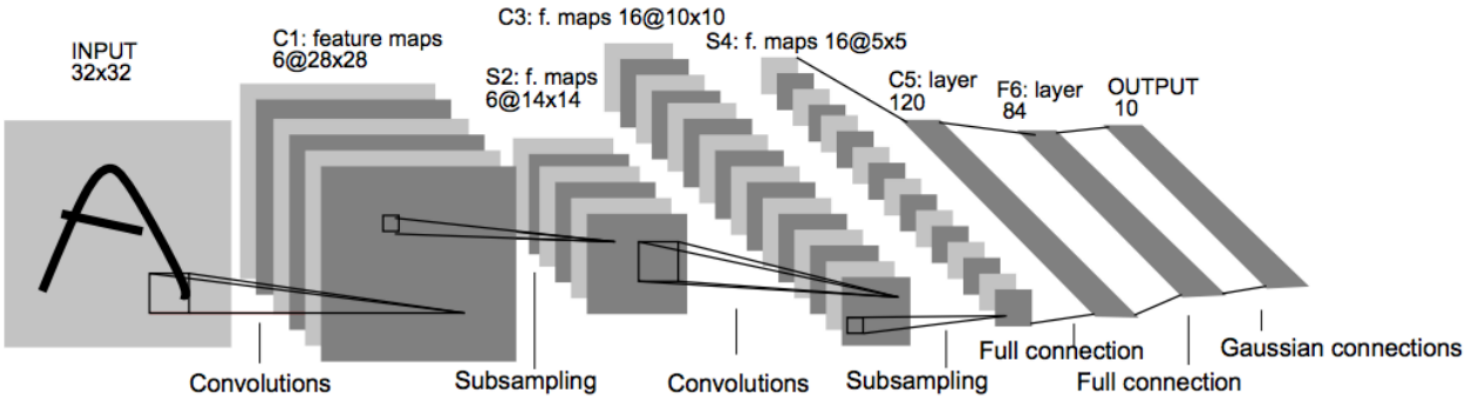
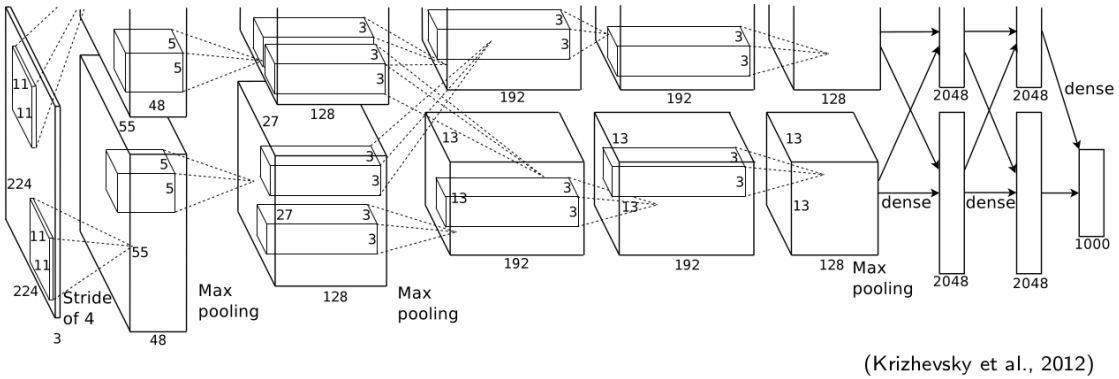


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Illustration from : ***Gradient-based Learning Applied to Document Recognition****, LeCun, Y. Bottou, L., Bengio, Y. and Haffner, Proceedings of the IEEE, 1989*

AlexNet (2012)

- AlexNet : created by Alex Krizhevsky in 2012
- Improved accuracy of ImageNet Large Scale Visual Recognition Challenge competition by **10 percentage points** (16.4%)
- First **truly deep neural network**
- Signaled beginning of dominance of deep learning in image processing and computer vision

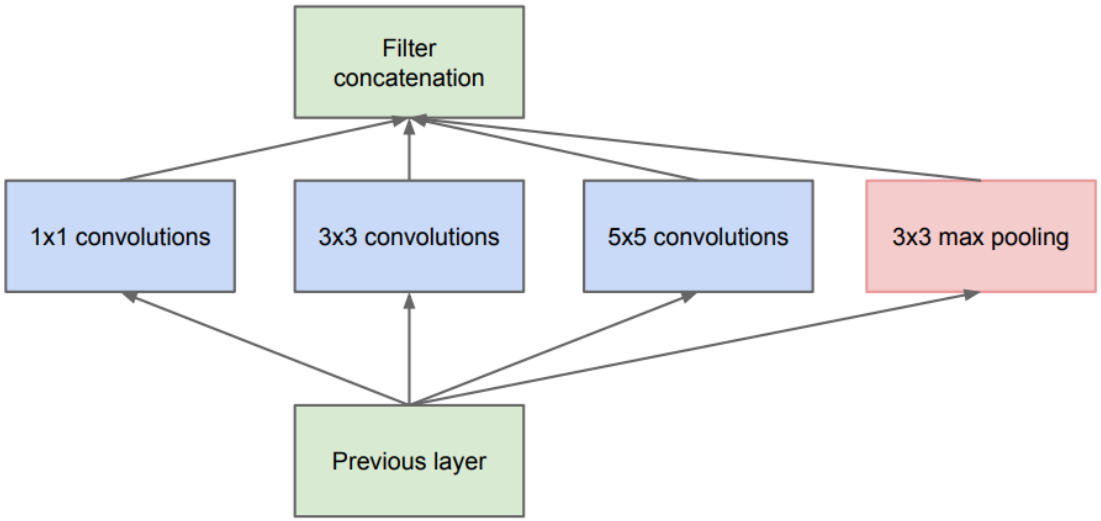


(Krizhevsky et al., 2012)

Illustration from : ***Imagenet classification with deep convolutional neural networks****, Krizhevsky, A., Sutskever, I. and Hinton, G. E, NIPS, 2012*

GoogLeNet (2015)

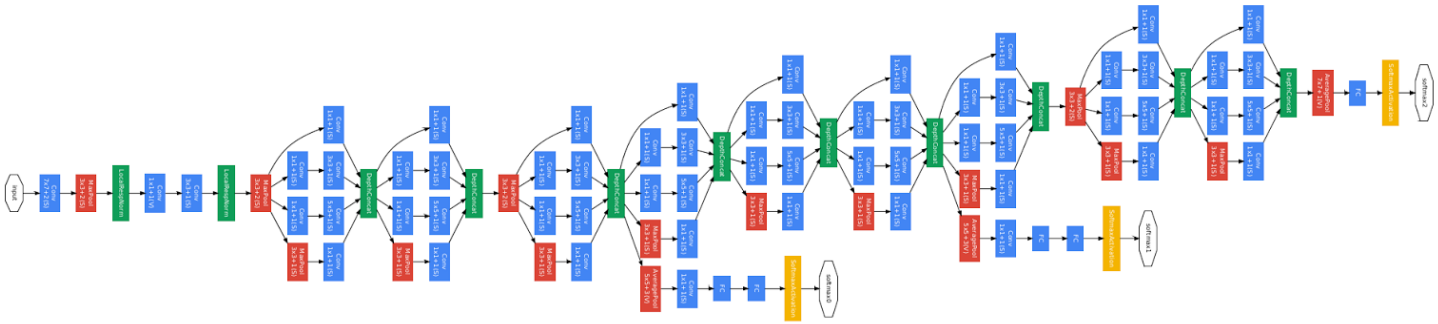
- In 2014/2015, Google introduced the "Inception" architecture/module
- Major attempt at reducing total number of parameters
- No fully connected layers, only convolutional
  - 2 million instead of 60 million for AlexNet
- Novel idea : have variable receptive field sizes in one layer



Going deeper with convolutions, *Szegedy et al, CVPR, 2015*

GoogLeNet (2015)

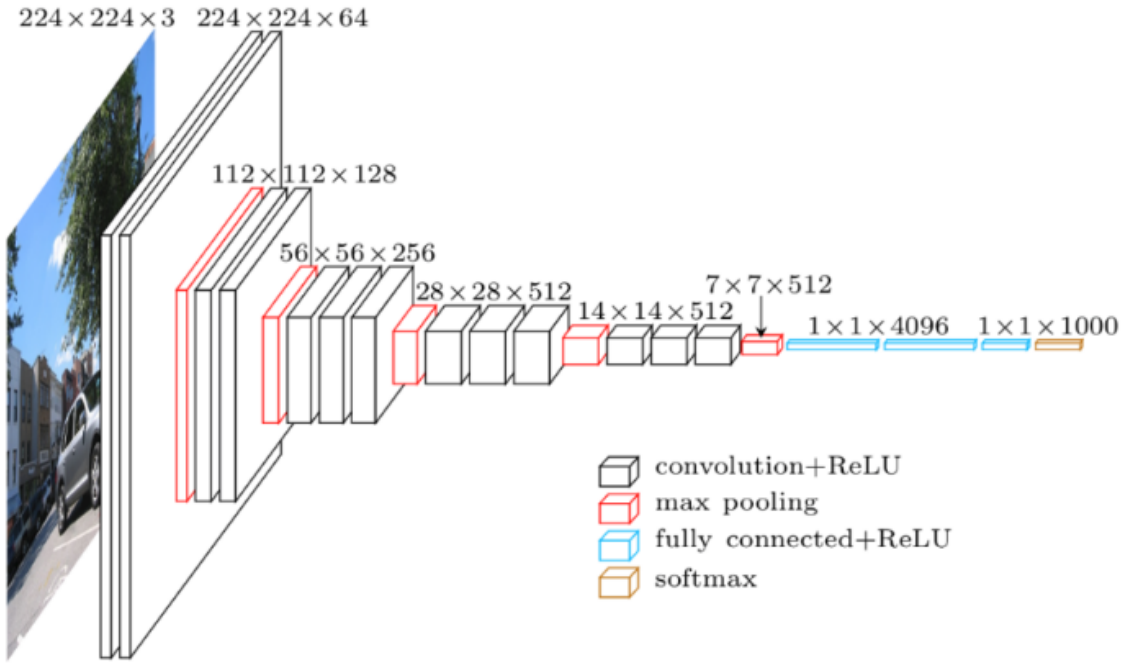
- Created by Google in 2014, GoogLeNet is a specific implementation of the "inception" architecture
- 6.6% test error rate on ImageNet (human error rate 5%)



Going deeper with convolutions, *Szegedy et al, CVPR, 2015*

VGG16 (2015)

- VGG16 is a 16-layer network, with small receptive fields (3×3 filters, with less subsampling)
- Around 7.5% test error on ILSVRC



Very Deep Convolutional Networks for Large-Scale Image Recognition, *Simonyan, K. and Zisserman, A., ICLR, 2015*
Illustration from Mathieu Cord,
https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/

Summary of advances in CNNs

Network	LeNet (1998)	AlexNet (2012)	GoogLeNet (2014)	VGG16 (2015)
Image size	28×28	$256 \times 256 \times 3$	$256 \times 256 \times 3$	$224 \times 224 \times 3$
Layers	3	8	22	16
Parameters	60,000	60 million	2 million	138 million

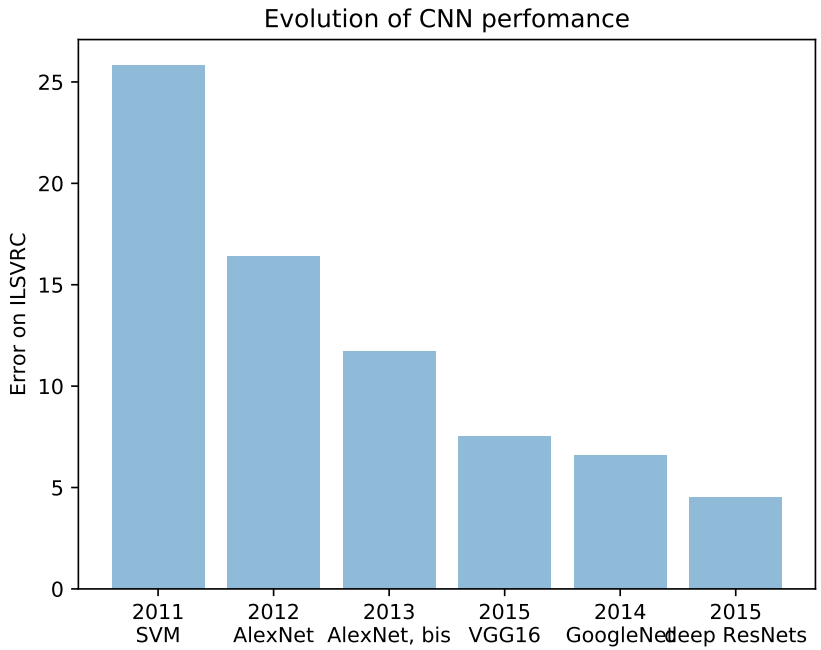


Image classification

- As we mentioned before, CNNs make sense for data with **grid-like** structures
- In particular, images are most often the target of CNNs
- Arguably the most common application of CNNs is to **image classification**
- Why is image classification important ? Closely linked to :
  - Object detection
  - Tracking
  - Image search (in large databases for example)
- In recent years, the best performing classification algorithms have been using neural networks

Image classification

- Why is image classification difficult ?
  - Images can vary in size, shape, position
  - We need to deal with variable lighting conditions, occlusions etc.

Image classification

- We have input datapoints $\mathbf{x}$, which we wish to classify into several, predefined classes $\{c_i, i = 1 \dots K\}$, where K is the number of classes
- As we have seen, convolution, non-linearities, subsampling allow for robust classification that is invariant to many perturbations

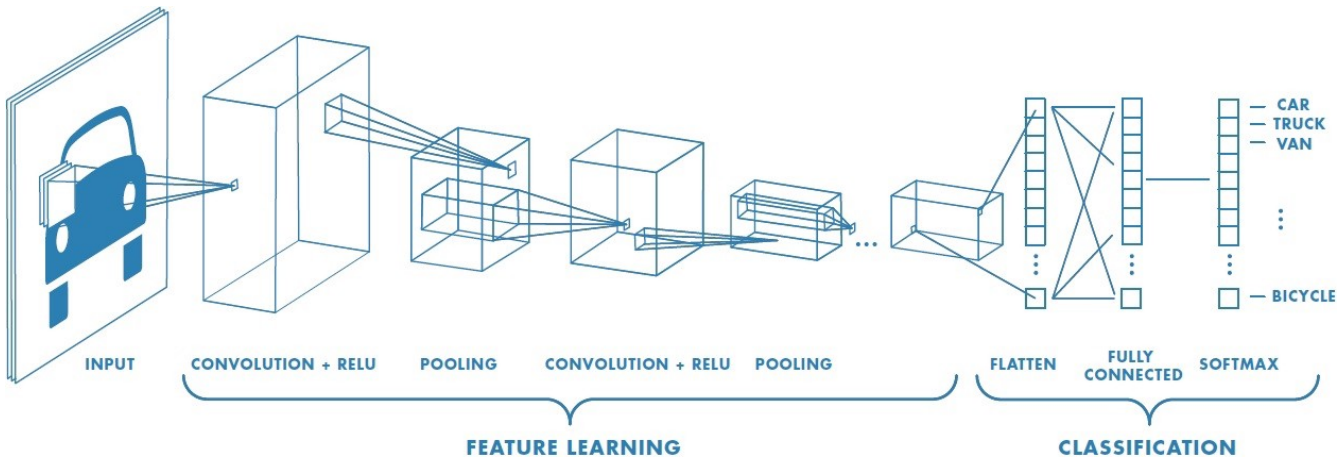
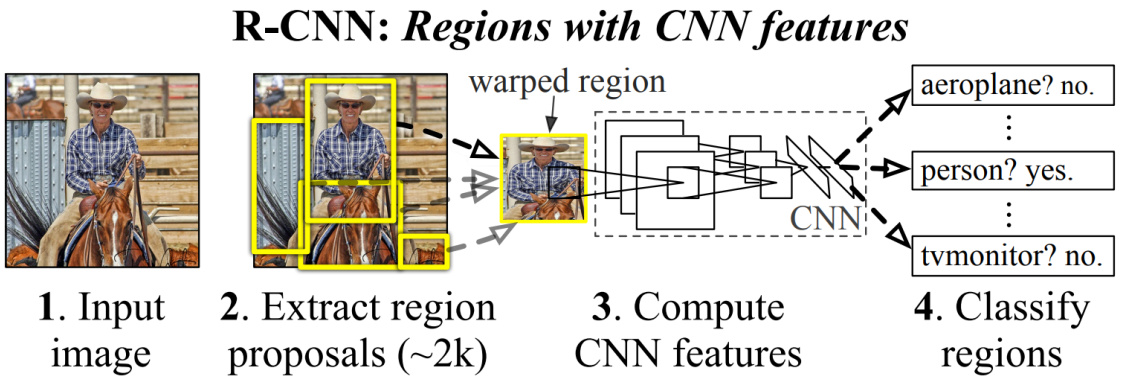


Image classification

- We can also **detect the position** of objects in images
- RNN† proposes a simple approach :
  1. Propose a list of bounding boxes in the image
  2. Pass the resized sub-images through a powerful classification network
  3. Classify each sub-image with your favourite classifier



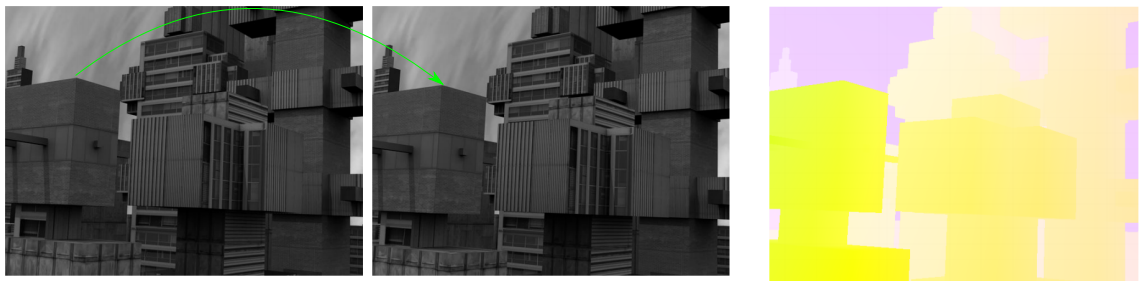
- Many variants on this work (Fast R-NN, Faster R-CNN) etc.

† ***Rich feature hierarchies for accurate object detection and semantic segmentation***, *Girschik, R. et al. CVPR 2014*

Motion estimation

- **Motion estimation** is a central task for many image processing and computer vision problems : tracking, video editing
- **Optical flow** involves estimating a vector field $(\mathbf{u}, \mathbf{v}) : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ where each vector points to the displacement of pixel (x, y) from an image I_1 to I_2

$$I_1(x, y) = I_2(x + \mathbf{u}(x, y), y + \mathbf{v}(x, y))$$

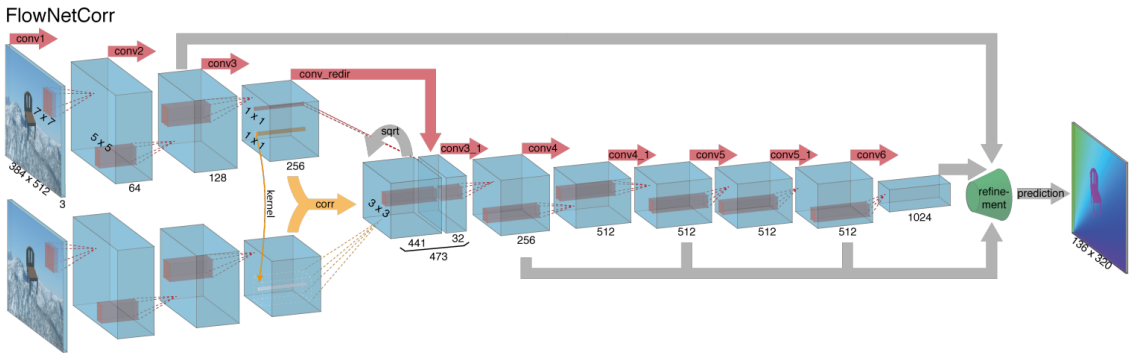


(a) Frame 1 (b) Frame 2 Optical flow

Illustration from : ***BriefMatch: Dense binary feature matching for real-time optical flow estimation****, Eilertsen, G, Forssén, P-E, Unger, J., Scandinavian Conference on Image Analysis, 2017*

Motion estimation with CNNs

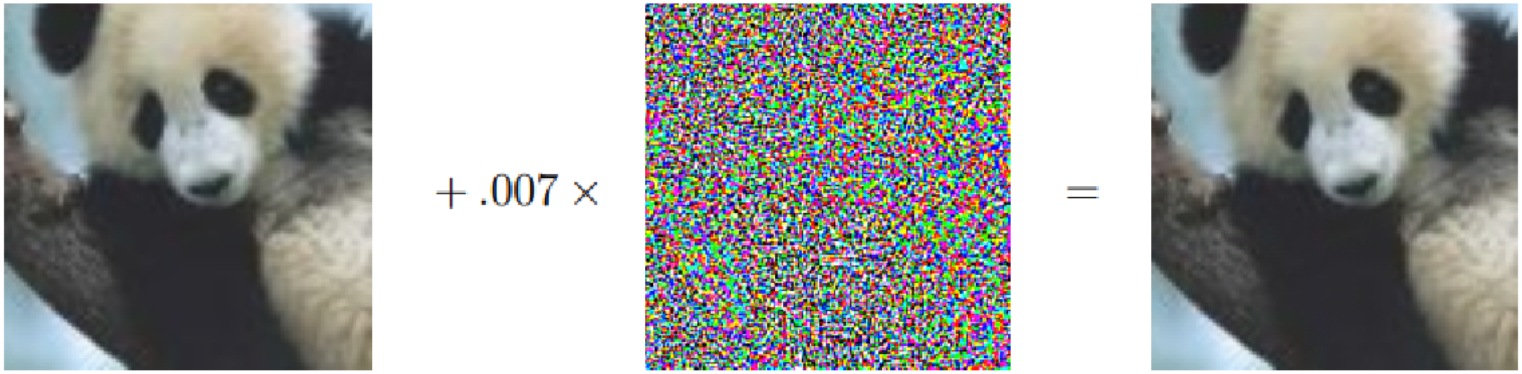
- A major challenge of optical flow estimation is to handle both fine and large-scale motions
  - This is difficult to do with classical, variational approaches
- CNNs have this multi-scale architecture already built in
- Example : FlowNet† uses this, first extracting meaningful features from the images (in parallel) and then combining them to create the optical flow



† ***FlowNet: Learning Optical Flow with Convolutional Networks***, *Fischer et al., ICCV 2015*

Adversarial examples

- We often get the impression that CNNs are the end all and be all of AI
- Consistently produce state-of-the-art results on images
- However, CNNs **are not infallible** : adversarial examples[†] !



- How was this image created ???

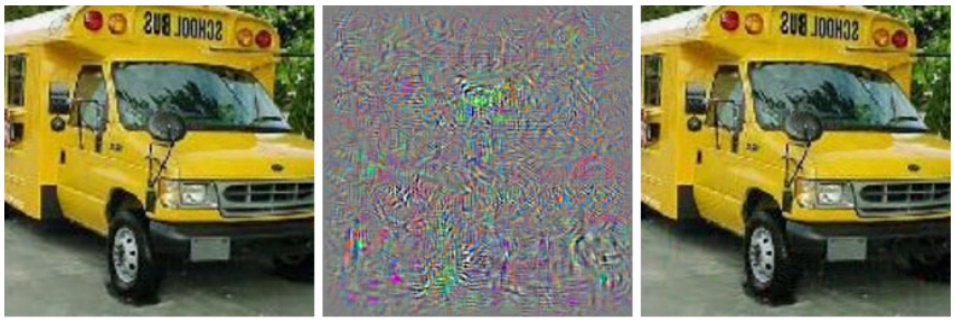
[†] ***Intriguing properties of neural networks***, *Szegedy, C. et al, arXiv preprint arXiv:1312.6199, 2013*

Adversarial examples

- Szegedy et al. propose[†] to add a small perturbation r that fools the classifier network f into choosing the wrong class c for $\hat{\mathbf{x}} = \mathbf{x} + \mathbf{r}$

$$\arg\min_{\mathbf{r}} |\mathbf{r}|_2^2, \ \text{s.t}\ f(\mathbf{x}+\mathbf{r}) = c,\ \mathbf{x}+\mathbf{r} \in [0,1]^n$$

- $\hat{x}$ is the closest example to x s.t $\hat{x}$ is classified as in class c
- Minimisation with box-constrained L-BFGS algorithm

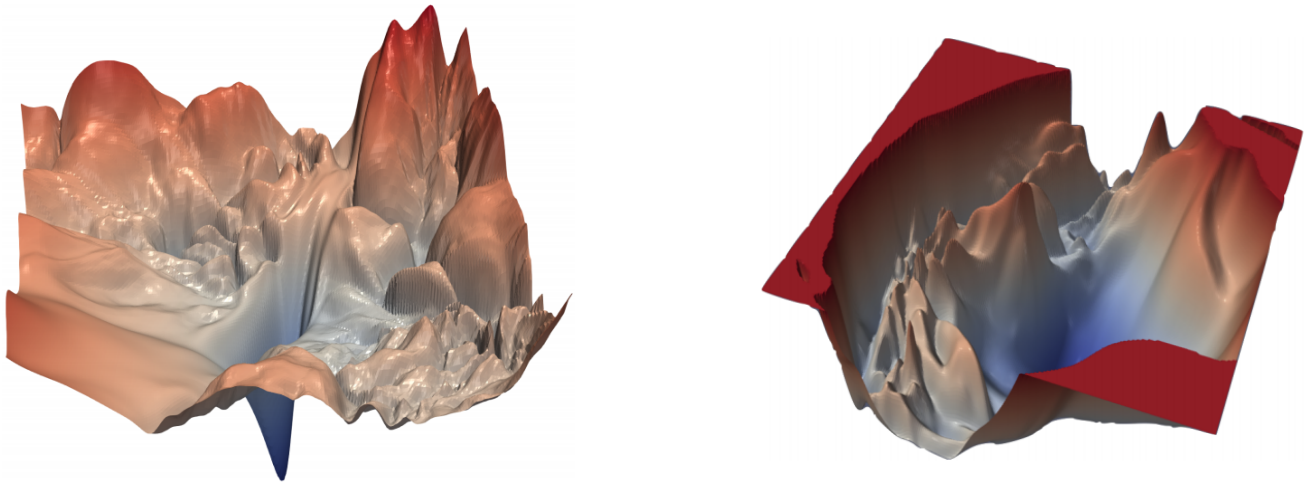


x r $\hat{x}$

[†] ***Intriguing properties of neural networks***, *Szegedy, C. et al, arXiv preprint arXiv:1312.6199, 2013*

Adversarial examples

- Common explanation : the space of images is very high-dimensional, and contains many areas that are unexplored during training time



Example of loss surfaces in commonly used networks (Res-Nets)

Illustration from ***Visualizing the Loss Landscape of Neural Nets****, Li, H et al, NIPS, 2018*

Adversarial examples

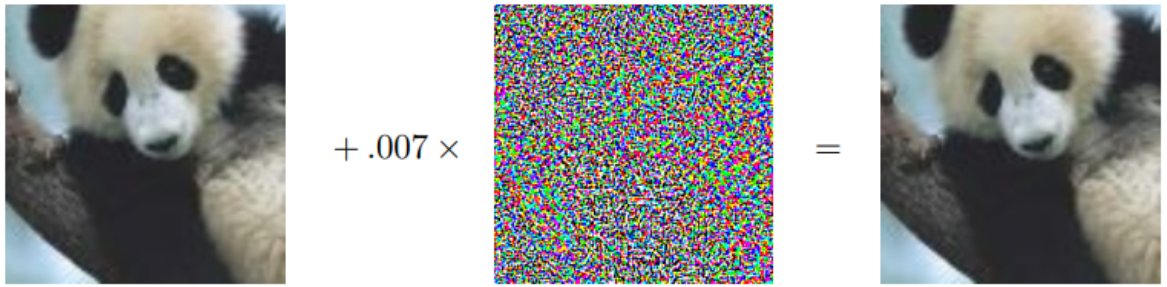
- Many approaches to adversarial examples exist. Goodfellow et al.[†]
- Consider the output of a fully connected layer $\langle \mathbf{w}, \hat{\mathbf{x}} \rangle = \langle \mathbf{w}, \mathbf{x} \rangle + \langle \mathbf{w}, \mathbf{r} \rangle$
- Let us set $\mathbf{r} = \mathrm{sign}(\mathbf{w})$. What happens to $\langle \mathbf{w}, \hat{\mathbf{x}} \rangle$?
  - Increase by nm as dimension n increases (m is average value of $\mathbf{w}$)
  - However, $|\mathbf{r}|_\infty$ does not increase with n
- Conclusion : we can add a small vector r that increases the output response $\langle \mathbf{w}, \hat{\mathbf{x}} \rangle$

[†] ***Explaining and Harnessing Adversarial Examples***, *Goodfellow, I.J, Shlens, J. and Szegedy, C., ICLR 2015*

Adversarial examples

- Goodfellow et al. consider a local linearisation of the network's loss around θ
  - $\mathcal{L}(\mathbf{x}_0) \approx f(\mathbf{x}_0) + \mathbf{w}\nabla_{\mathbf{x}}\mathcal{L}(\theta, \mathbf{x}_0)$
- Thus, the perturbation image $\hat{\mathbf{x}}$ is set to

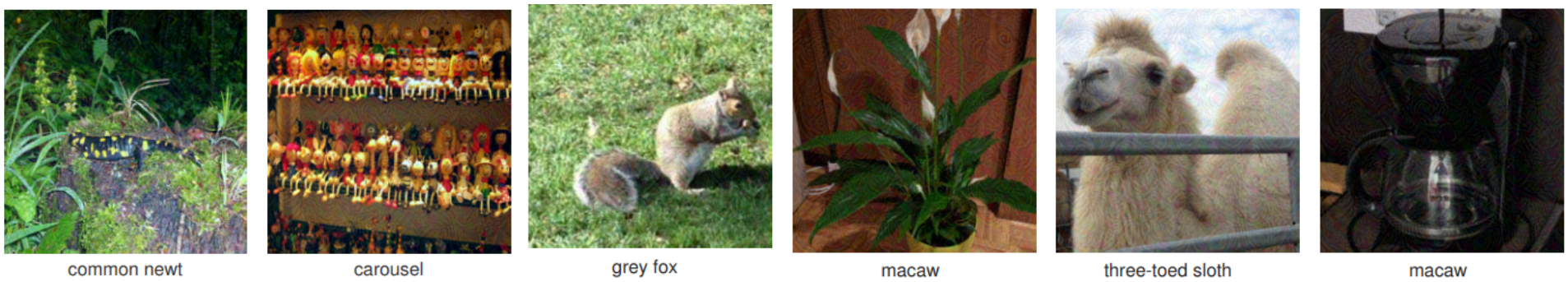
$$\hat{\mathbf{x}} = \mathbf{x} + \epsilon \text{sign}(\nabla_{\mathbf{x}}\mathcal{L}(\theta, \mathbf{x}))$$



† ***Explaining and Harnessing Adversarial Examples***, *Goodfellow, I.J, Shlens, J. and Szegedy, C., ICLR 2015*

Adversarial examples

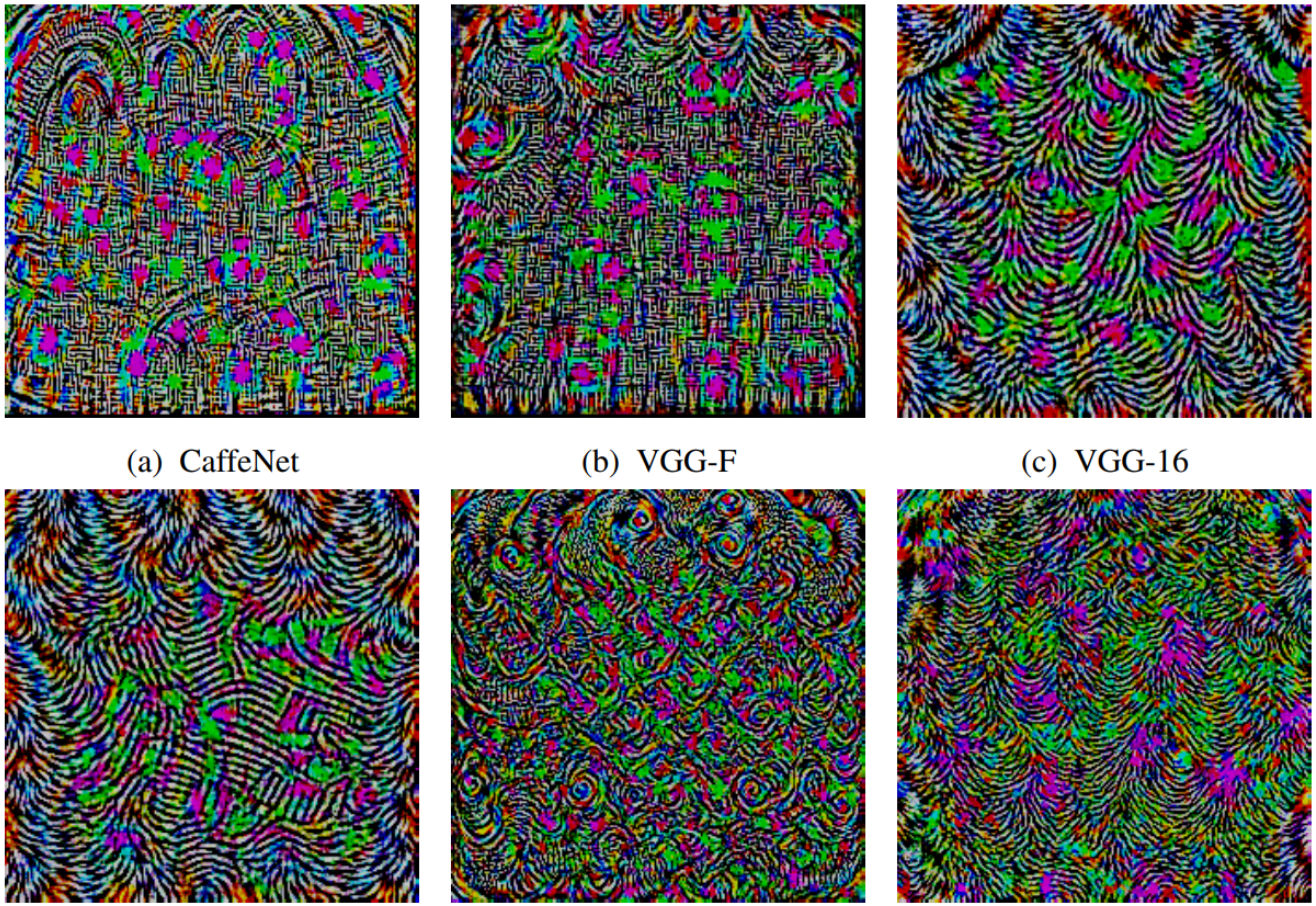
- Even worse, it is possible to create **universal adversarial** examples†
- Perturbations that fool a network for any image class



- Simple algorithm : initialise perturbation $\mathbf{r}$, go through database adding specific perturbations to $\mathbf{r}$, project onto set $\{\mathbf{r}, ||\mathbf{r}|| < \varepsilon\}$
- What do these perturbations look like ?

† ***Universal adversarial perturbations***, *Moosavi-Dezfooli, S-M, et al arXiv preprint (2017)*

Adversarial examples



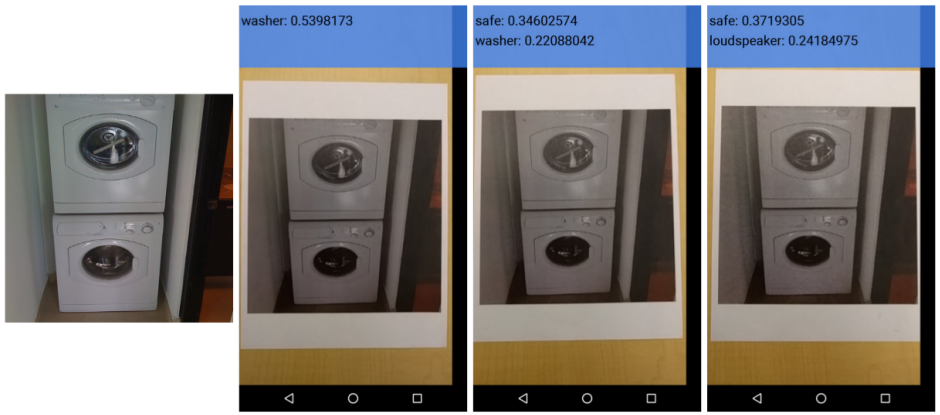
(a) CaffeNet (b) VGG-F (c) VGG-16

(d) VGG-19 (e) GoogLeNet (f) ResNet-152

† ***Universal adversarial perturbations***, *Moosavi-Dezfooli, S-M, et al arXiv preprint (2017)*

Adversarial examples

- Conclusion : CNNs are not necessarily robust
- Adversarial examples are a significant problem :
  - Even **printed photos** of adversarial examples work[†]



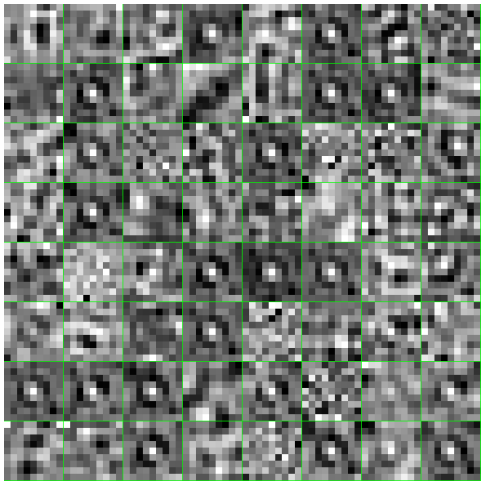
(a) Image from dataset (b) Clean image (c) Adv. image, $\epsilon = 4$ (d) Adv. image, $\epsilon = 8$

- Explaining and resisting adversarial examples is currently a hot research topic
  - DARPA explainable AI project

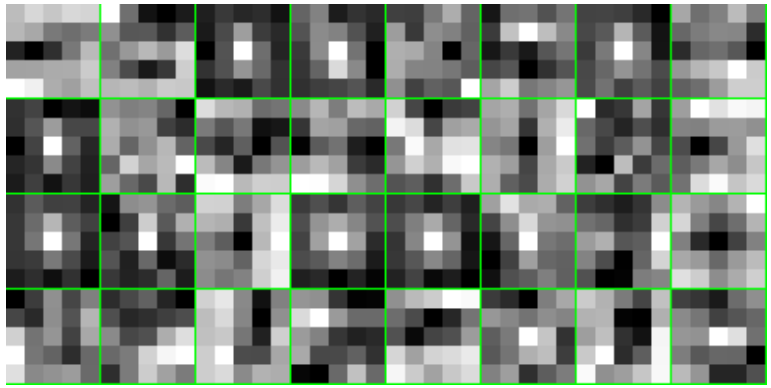
[†] ***Adversarial Examples in the Physical World***, *Kurakin, A., Goodfellow, I. J, Bengio, S. et al. ICLR workshop, 2017*

Visualising CNNs

- We would like to **understand what CNNs are learning**
  - Unfortunately filters are difficult to interpret (especially deeper layers)



Layer 1 filters



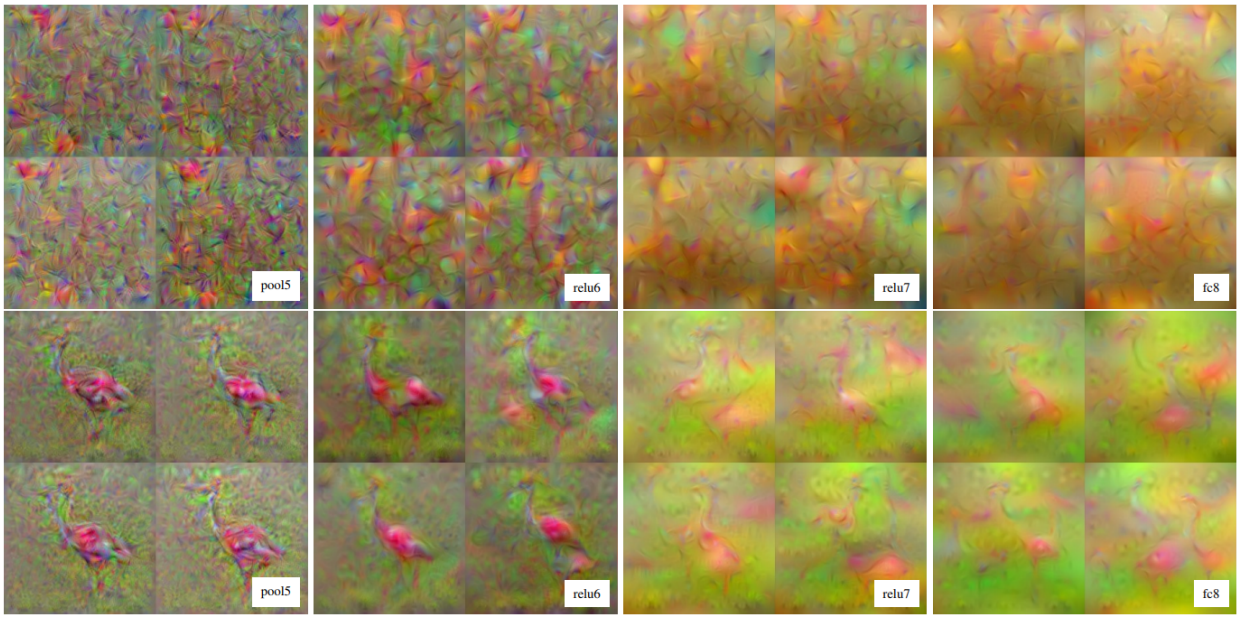
Layer 3 filters

- Therefore, much research has been dedicated to **visualising** CNNs

Visualising CNNs

- Idea : "invert" CNN : find $\mathbf{x}$ that gives a certain output
- Standard inverse problem with regularisation

$$\hat{x} = \arg\min_{\mathbf{x}} \|f(\mathbf{x}) - f_0\|_2^2 + \lambda\|\mathbf{x}\|_2^2 + \mu\|\nabla\mathbf{x}\|_2^2 \tag{1}$$



† *Mahendran and Vedaldi* ***Understanding Deep Image Representations by Inverting Them***, *Conference on Computer Vision and Pattern Recognition, 2014*

Visualising features

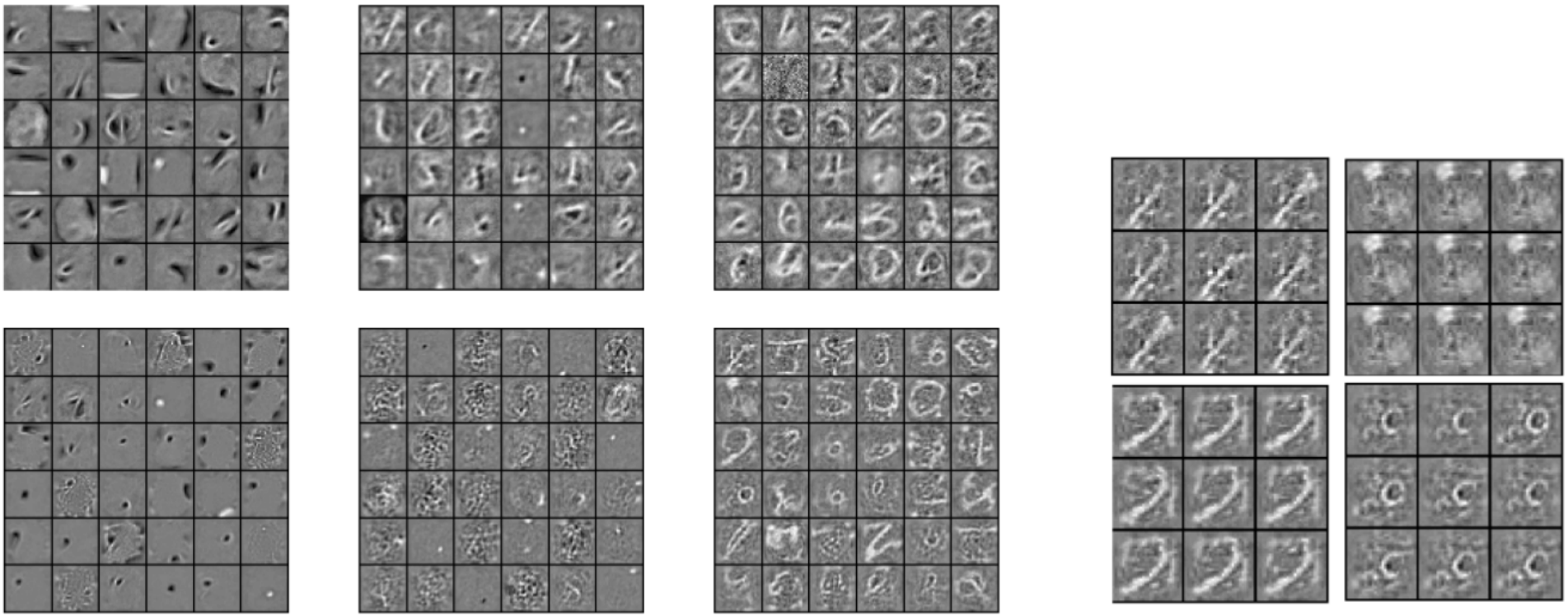
- Another approach to understanding CNNs : **maximise response** to a given filter
- Choose layer ℓ, filter number k and element ("pixel") (i, j)
- Random initialisation $\mathbf{x}_0$, constrain norm of solution $\mathbf{x}$

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}} \mathbf{u}^{\ell}_{i,j,k}$$
$$\text{with } \|\mathbf{x}\| = \rho$$

- Optimisation : **gradient ascent**

† *Erhan, Bengio, Courville, Vincent,* ***Visualizing Higher-Layer Features of a Deep Network****, University of Montreal, 2009*

Visualising CNNs



Layer 1 Layer 2 Layer 3 Layer 4

Maximisation of different activations applied to MNIST dataset

† *Erhan, Bengio, Courville, Vincent,* ***Visualizing Higher-Layer Features of a Deep Network****, University of Montreal, 2009*

Visualising CNNs

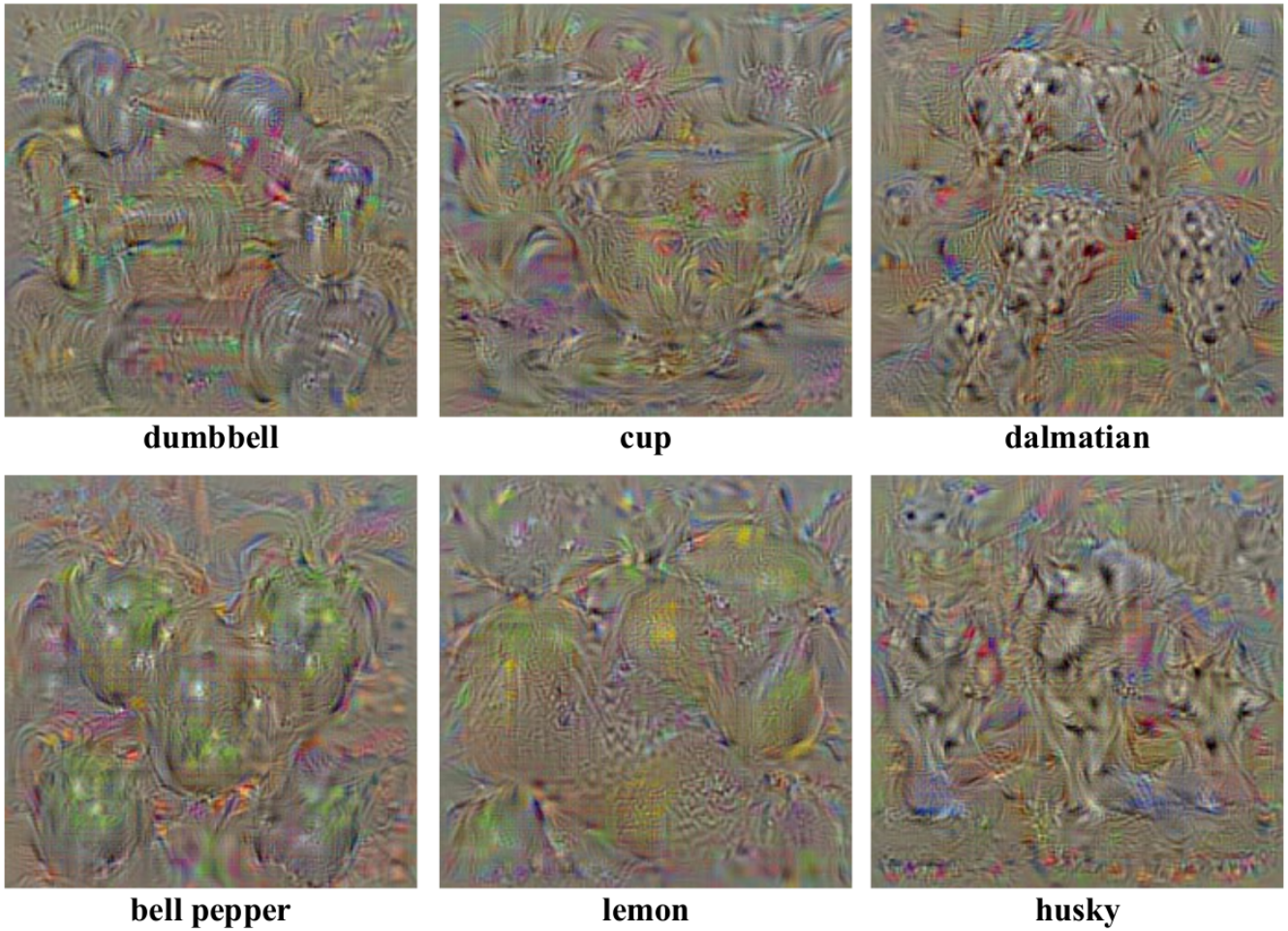
- Another approach of Simonyan et al.[†] proposes to see what images correspond to what classes
- Choose a class c, maximise the response of this class

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}} f(\mathbf{x})_c - \lambda \|\mathbf{x}\|_2^2$$

- Find an L_2-regularised image which maximises the score for a given class c
- Initialise with random input image $\mathbf{x}_0$

[†] *Simonyan, Vedaldi, Zisserman* ***Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps****, arXiv preprint arXiv:1312.6034, 2013*

Visualising CNNs

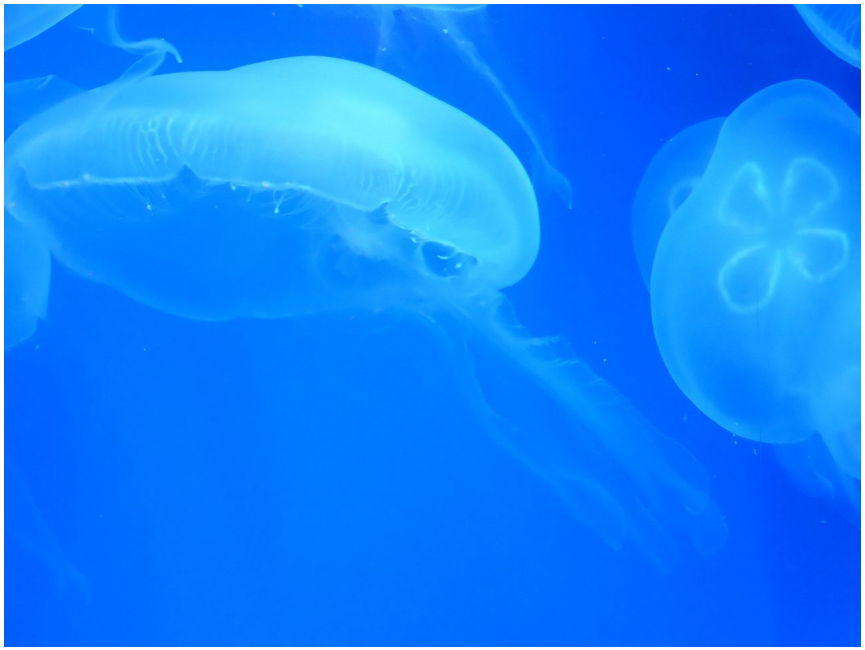


Class *model* visualisation

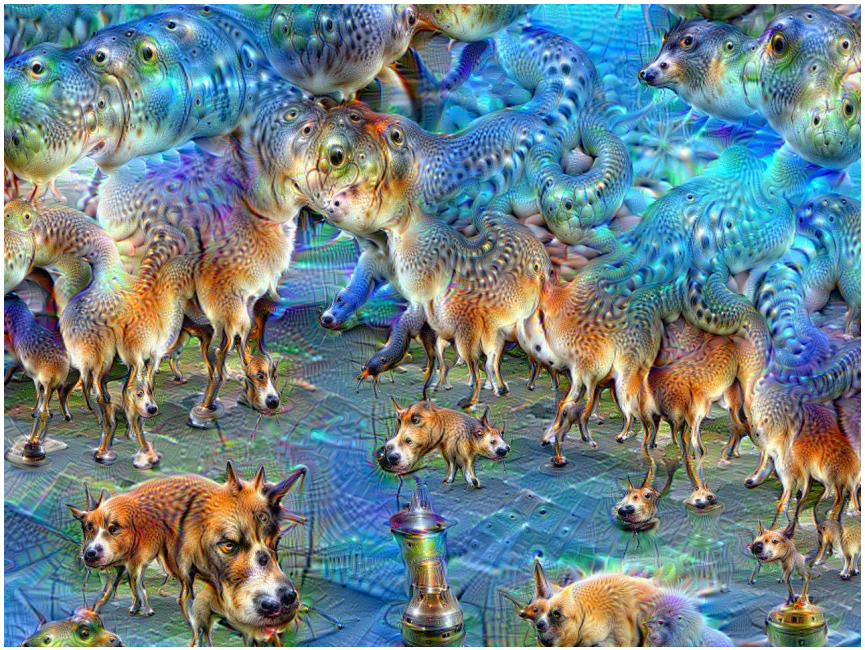
† *Simonyan, Vedaldi, Zisserman* ***Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps****, arXiv preprint arXiv:1312.6034, 2013*

Visualising CNNs

- Similar idea with Inception architecture of Google : "Deep Dream"
- Maximise a class from input image



Input image



Maximising "dogs" category

Deepdream - a code example for visualizing neural networks, *Mordvintsev, A., Olah, C. and Tyka, M., Google Research, 2015*

Conclusion

- CNNs represent the state-of-the art in many different domains/problems
- If you have an unsolved problem, there is a good chance CNNs will produce a good/excellent result
- **However : theoretical understanding is still relatively limited**
  - This leads to problems such as adversarial examples
  - It is not clear whether CNNs are truly robust/generalisable
  - This is a hot research topic, important if CNNs are to be used in industrial applications
- Next week : Recurrent Neural Networks (RNNs)
- Last lesson : Autoencoders, Generative Networks (GANs)

Summary

1. Convolutional Neural Networks
   - Introduction, notation
     - Convolutional Layers
   - Down-sampling and the receptive field
     - Image classification datasets and well-known CNNs
   - Applications of CNNs
     - Adversarial Examples
     - Visualising CNNs
2. Generative Models
   - Autoencoders
     - Vanilla Autoencoder
     - Autoencoder variants
   - Generative models
     - Variational autoencoders
     - Generative Adversarial Networks
     - Texture and style models
3. Summary

Introduction

- We have seen neural networks used for :
  - Classification/detection (MLPs, CNNs)
  - Modelling time-series, sequences (RNNs)
- All of these networks rely on the extraction of features to analyse data
- Idea : the network's internal representation of the data can be useful !
- **Autoencoders** and **generative models** use this idea

Introduction

- What do you think of these faces ?

Introduction

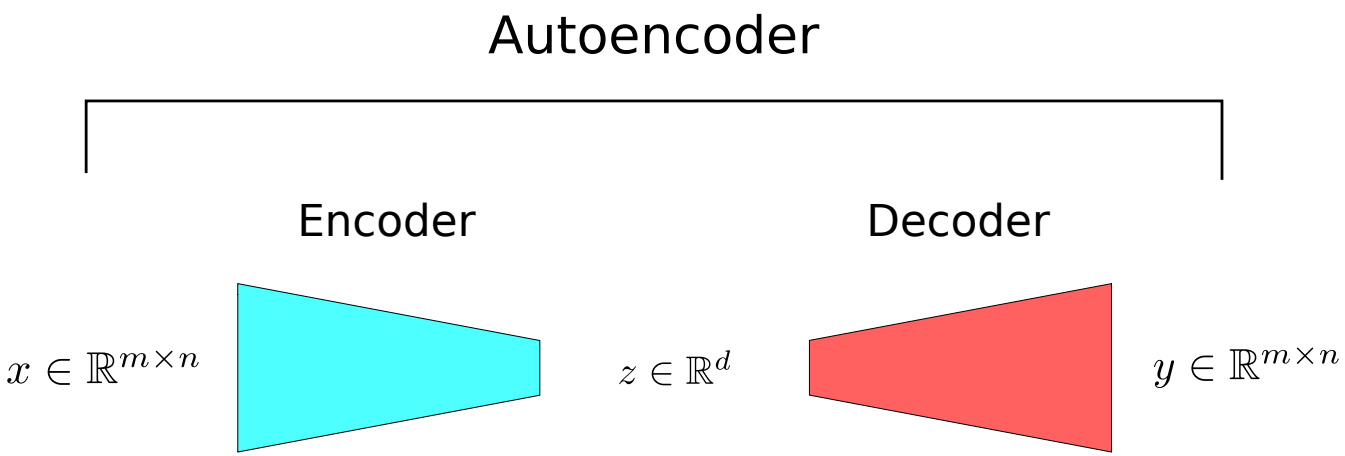
- What do you think of these faces ?

- These are all **generated by a generative model !!**
- Today, we are going to see how this is possible

Autoencoders

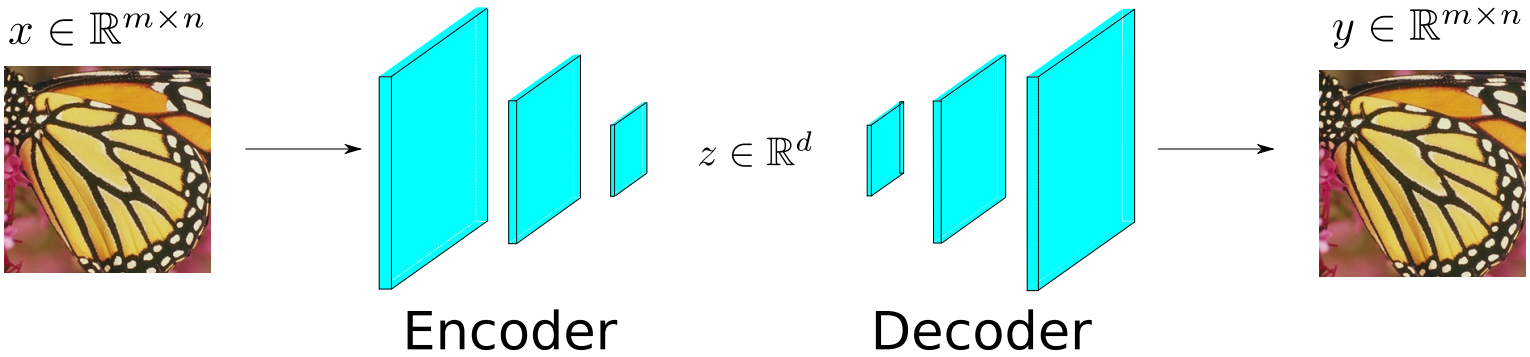
Autoencoders

- Autoencoders consist of two networks : an **encoder** and a **decoder**
  - Encoder : map data x to a smaller **latent space**
  - Decoder : map point z back from latent space to original data space
- Main idea : the latent space is a space where it is **easier to manipulate/understand data**
- More powerful and compact representation of data



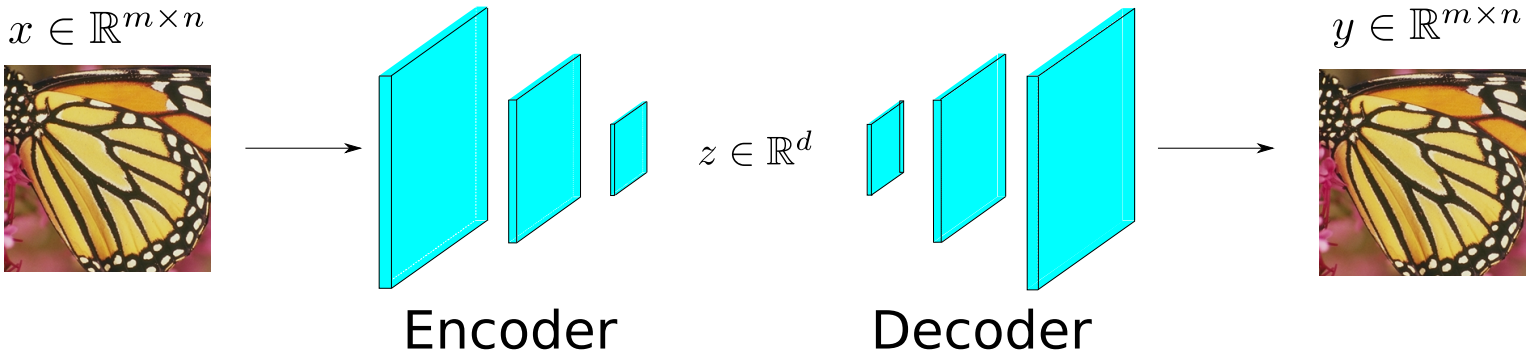
Autoencoders

- The autoencoder is trained to minimise some norm between the input x and the output y of the decoder
- In almost all cases, we have $d << mn$
- This forces the autoencoder to learn a compact and powerful latent space



Autoencoders

- Uses of autoencoders :
  - Data compression, dimensionality reduction
  - Classification (easier in latent space)
  - **Data generation/synthesis**



Autoencoders - some notation

- An AE is a neural network consisting of two sub-networks
  - The encoder Φ_e,

$$\Phi_e : \mathbb{R}^{mn} \to \mathbb{R}^d$$
$$x \mapsto \Phi_e(x) = z$$

  - The decoder Φ_d,

$$\Phi_d : \mathbb{R}^d \to \mathbb{R}^{mn}$$
$$z \mapsto \Phi_d(z) = y$$

- As in other neural networks, the main components of AEs are **mlp's/convolutions**, **biases** and **non-linearities**

Autoencoders

- The autoencoder is trained to reproduce the input x as an output y, in the sense of some norm, having gone through the bottleneck of the network
- The norm most often used is the sum of squared differences (ℓ_2-norm)

Autoencoding training minimisation problem

$$\begin{aligned}\mathcal{L}(x) &= \|y - x\|_2^2 \\ &= \sum_i^m \sum_j^n \left((\Phi_d \circ \Phi_e(x))_{i,j} - x_{i,j} \right)^2\end{aligned}$$

- Put simply : **output should look like input !**

Autoencoders - upsampling

- Most often, the autoencoder uses convolutions
- A key question is how to create the bottleneck : **downsampling**
- We know how to downsample :
  - Strided convolutions
  - Max pooling
- How about **upsampling** ?

Autoencoders - upsampling

- Upsampling can be carried out in several ways :
  - Simple interpolation (linear, bilinear, bicubic)
  - Transposed convolution
- **Transposed convolution** is probably the most common upsampling
  - Simultaneously upsamples and "convolves"
- Let's take a look at how this works

Autoencoders - transposed convolution

- Recall that we can write the convolution as a matrix/vector multiplication (see lecture on CNNs)
- For example, take the convolution with the Laplacian operator

$$w = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \rightarrow A_w = \begin{pmatrix} 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & & \\ -1 & 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & \\ 0 & -1 & 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & -1 & 4 \end{pmatrix}$$

- To carry out convolution $+$ stride with subsampling s, we just remove certain rows from the matrix A_w (the elements not retained during subsampling)

Autoencoders - transposed convolution

$$A_{w,s} = \begin{pmatrix} 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & & \\ -1 & 4 & -1 & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \overset{0}{\cdots} & -1 & \overset{0}{\cdots} & -1 & 4 \end{pmatrix}$$

- A_w now becomes a $\frac{mn}{s} \times mn$ matrix $A_{w,s}$
- Transposed convolution just consists of $A_{w,s}^T$

Autoencoders - transposed convolution

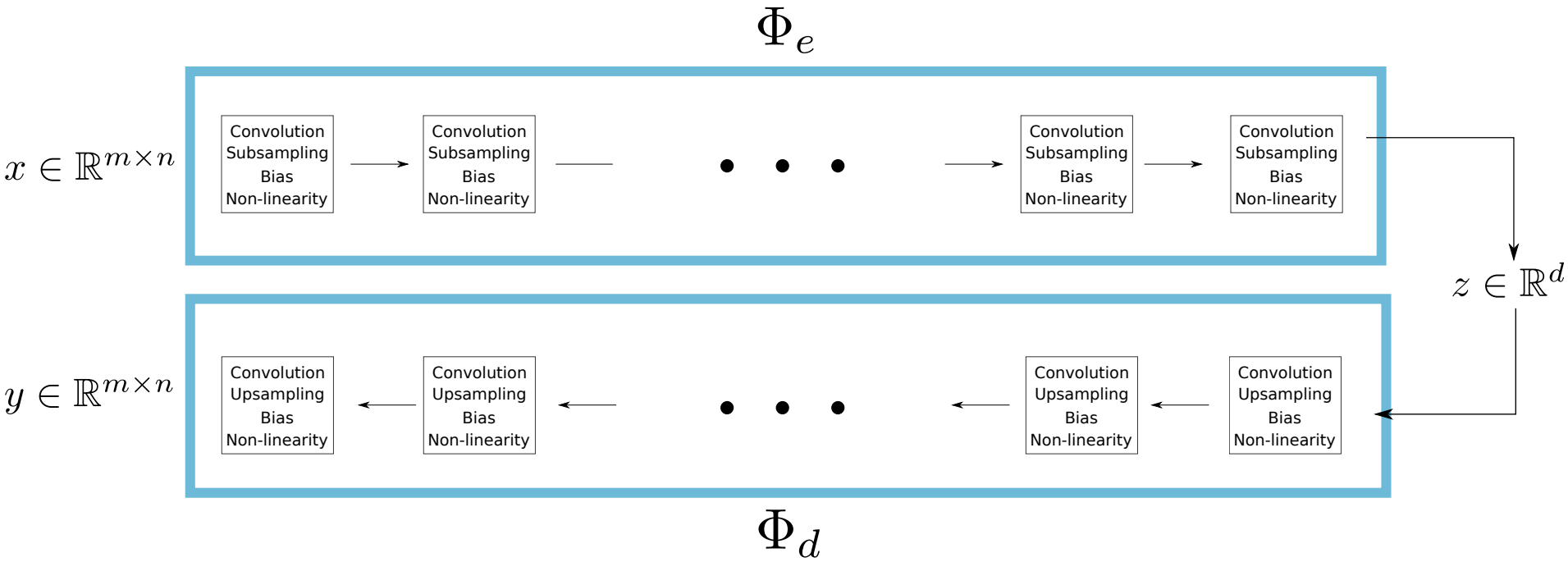
- Let's take a concrete example :
  - We wish to upsample a 1D signal x of size 2 to size 4
  - Convolutional filter $w = [1, 2, 1]^T$
- First, we look at the convolution matrix at the higher resolution signal size 4

$$A_w = \begin{pmatrix} \mathbf{2} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ 1 & 2 & 1 & 1 \\ \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{0} \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

- Therefore, we have :

$$A_{w,s} = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 0 \end{pmatrix} \quad A_{w,s}^T = \begin{pmatrix} 20 \\ 11 \\ 02 \\ 00 \end{pmatrix}$$

A generic autoencoder architecture



- $d << m * n$

Autoencoder variants

- Autoencoders come in many flavours, differ mainly by their loss functions
- Naive autoencoder loss $\mathcal{L}$ can lead to certain problems
  - Overfitting to data, poor robustness
  - Latent space difficult to interpret, not necessarily meaingful w.r.t data space
- As is often the case in deep learning, this can be addressed using **regularisation**
- In practice, this means adding extra terms to $\mathcal{L}$

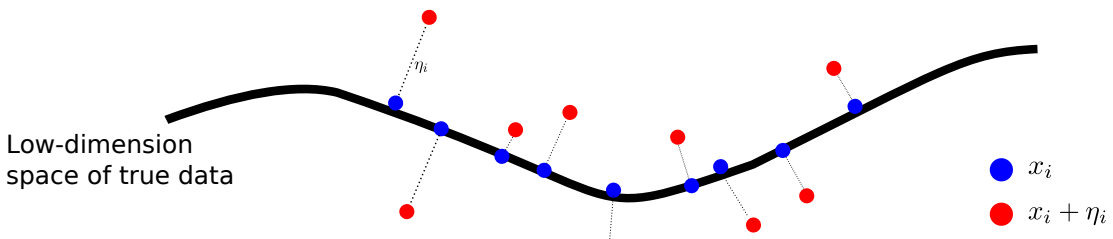
Denoising autoencoder

- First example : the denoising autoencoder
- We would like to make the encoder/decoder robust to **small perturbations** in the input data
- One solution : the **denoising autoencoder**

Denoising autoencoder

- Idea : add noise η to the input

$$\mathcal{L}(x) = \|\Phi_d \circ \Phi_e(x + \eta) - x\|_2^2$$



Sparse autoencoder

- Ideally, we want the code to be as **sparse** as possible
- Why ? We want the smallest vector which accurately describes the data
  - Combinations of elements more difficult to interpret
  - Useful for classification

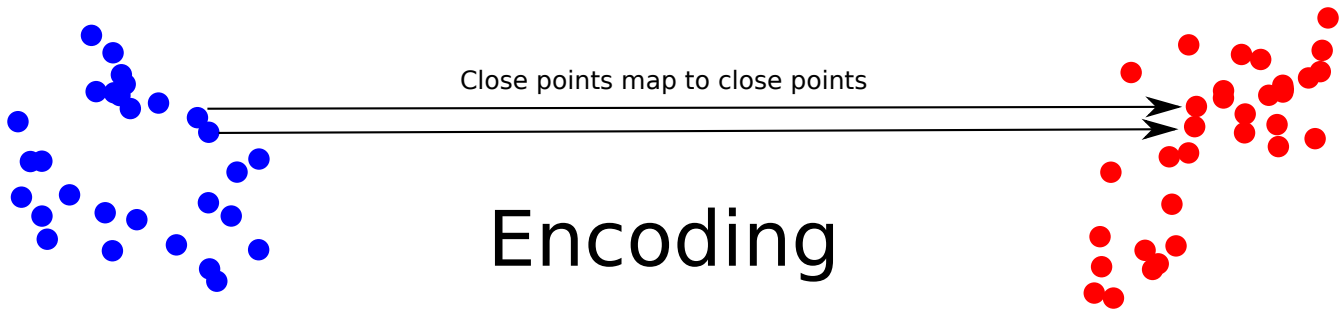
Sparse autoencoder

$$\mathcal{L}(x) = \|\Phi_d \circ \Phi_e(x) - x\|_2^2 + \lambda \|z\|_1$$

- The $\|z\|_1$ norm encourages sparsity in z

Contractive autoencoder

- Another approach to regularisation : contractive autoencoder
- Close points in data space map to close points in latent space (thus, contractive)



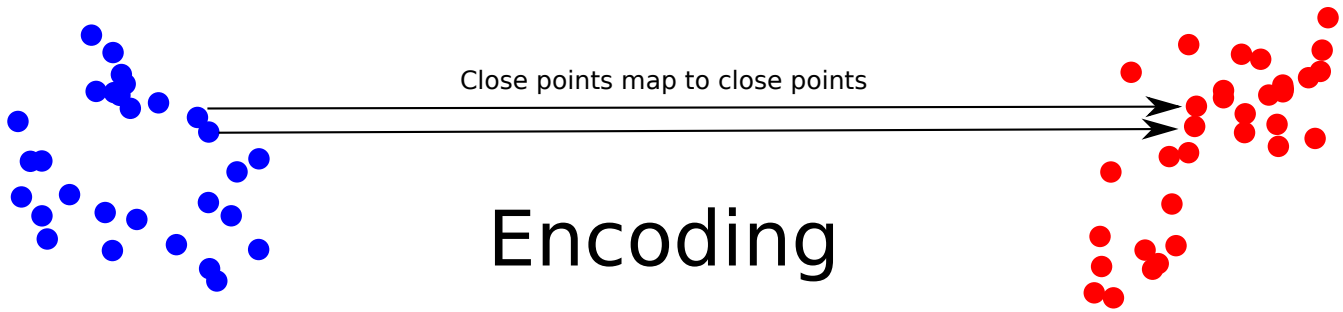
- How can we impose this ?
- $\frac{\partial z_j}{\partial x_i}$ should be small, for all couples (i, j)

Contractive autoencoder

- Add the Frobenius (ℓ_2) norm of the Jacobian of z w.r.t. x, $J_x z$, to the cost function

$$\mathcal{L}(x) = \|\Phi_d \circ \Phi_e(x) - x\|_2^2 + \lambda \|J_x E(x)\|_F^2$$
$$= \|\Phi_d \circ \Phi_e(x) - x\|_2^2 + \lambda \|J_x z\|_F^2$$

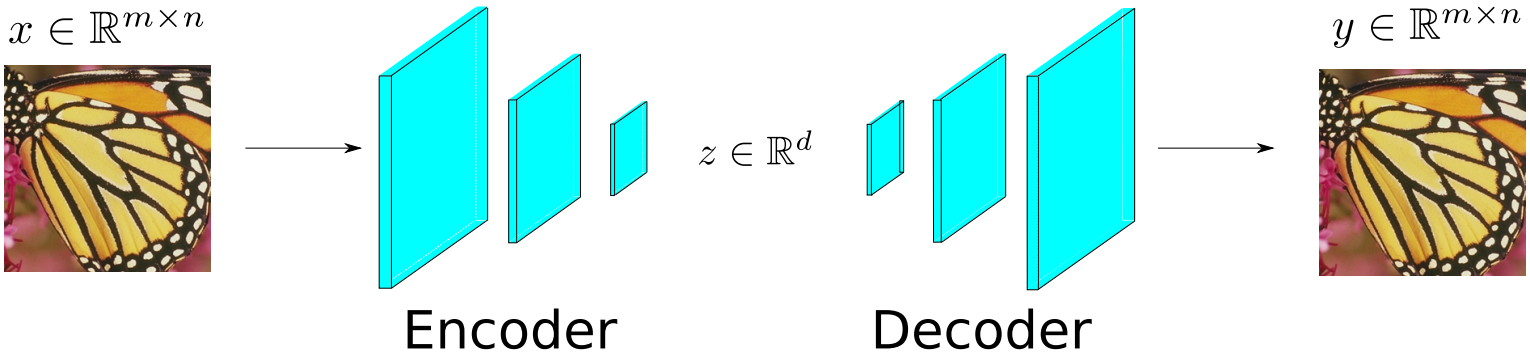
- Reminder, Jacobian : $J_x(z) = \begin{pmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_1}{\partial x_{mn}} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_d}{\partial x_1} & \cdots & \frac{\partial z_d}{\partial x_{mn}} \end{pmatrix}$



Autoencoders - summary

Important points to remember

- Autoencoder consists of two networks : encoder and decoder
- These compress to and from a **smaller dimension latent space**
- This latent space represents data more **powerfully and compactly**



Autoencoders

- Example of autoencoder use : interpolation of complex data

Interpolation of complex data[†]

[†] ***Generative Visual Manipulation on the Natural Image Manifold***, *J-Y. Zhu, P. Krähenbühl, E. Schechtman, A. Efros, CVPR 2016*

Generative models

Generative models

- In many applications, it is desirable to **synthesise** data
  - Video post-production
  - Data augmentation

- Several types of generative models exist :
  - Restricted Bolzmann machines, Deep Belief models
  - **Variational autoencoders**
  - **Generative Adversarial Networks**
  - **Texture synthesis and style transfer models**

- The common idea in these models is the internal representation/latent space of the network

Generative models

- Modern generative models produce highly realistic, (relatively) high-definition images

Synthesis examples from "Real NVP"[†]

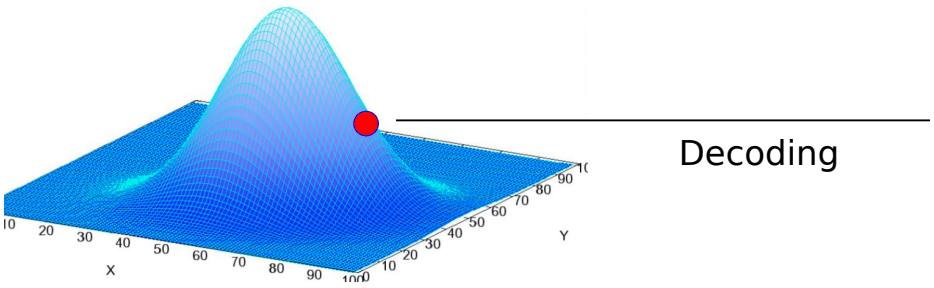
[†] ***Density estimation using Real NVP***, *L. Dinh, J. Sohl-Dickstein, S. Bengio, arXiv:1605.08803 2016*

Variational autoencoder

- Suppose we want to produce **random examples of data**, how would we go about this ?

Variational autoencoder

- Suppose we want to produce **random examples of data**, how would we go about this ?
- We can model the latent space in a **probabilistic manner**
- Synthesis will then consist of :
  1. Sampling in the latent space
  2. Decoding to produce the random image



Probabilistic model in latent space

Synthesis of random image

Variational autoencoder - variational Bayesian approach

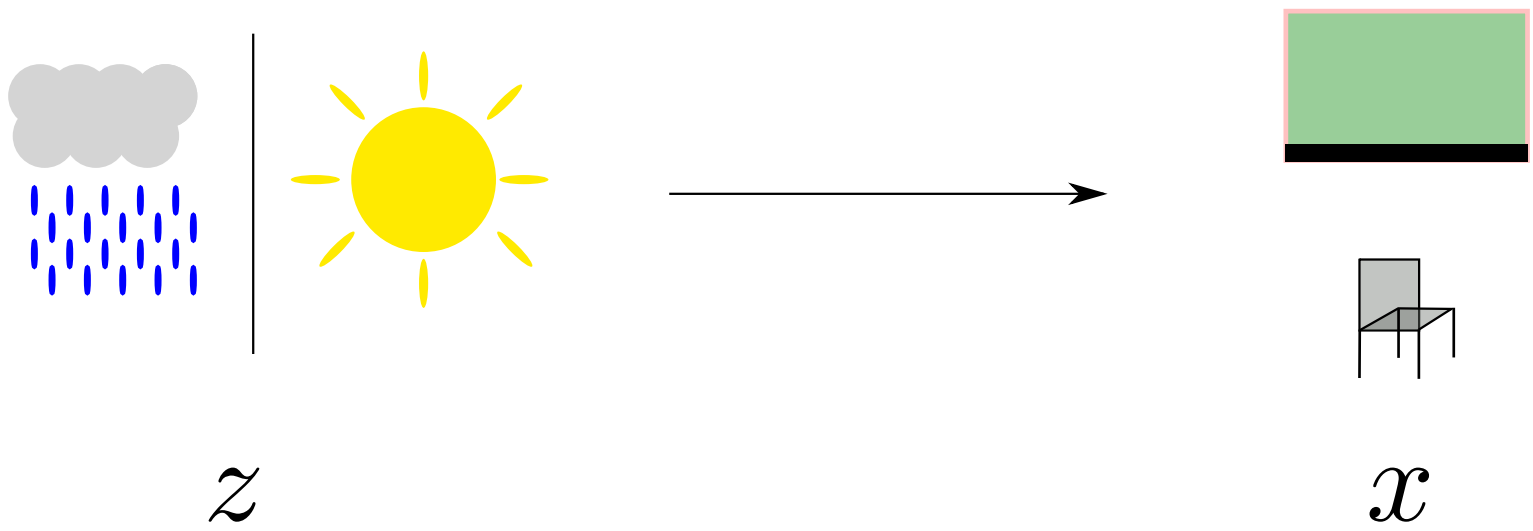
- The Variational Autoencoder (VAE) encourages the latent code z to follow a certain distribution, via the loss function
- This is in turn achieved by using a **Variational Bayesian** approach

Variational autoencoder - variational Bayesian approach

- The Variational Autoencoder (VAE) encourages the latent code z to follow a certain distribution, via the loss function
- This is in turn achieved by using a **Variational Bayesian** approach
- The Variational Bayesian approach is a methodology to approximate the **posterior distribution** of unobserved variables in graphical models
- We will need some tools from statistics : conditional probability, Bayes theorem, marginal distributions, distribution divergences ...
- Keep in mind that the main goal here is to **choose a loss function which will encourage the latent space to follow a probability distribution**

Variational autoencoder - variational Bayesian approach

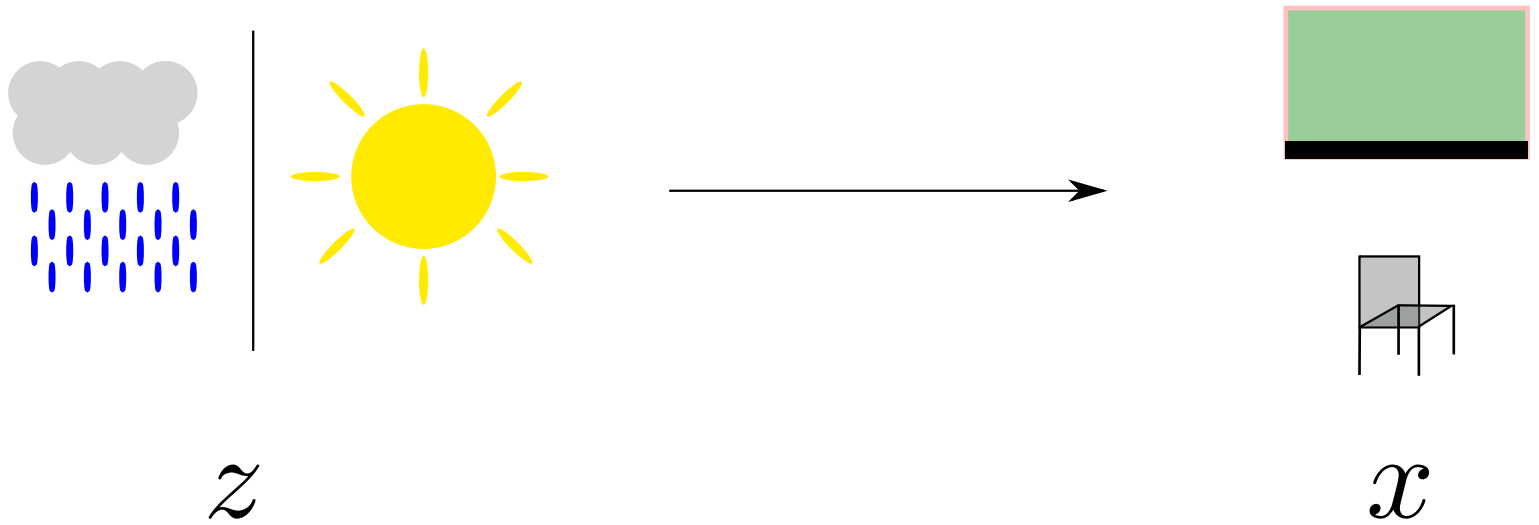
- Let's take an example. Suppose we have a student A, who comes to lessons or not. The event of A's presence in the lesson is x. This is the **observed variable**
- We also know that sometimes it rains, and that A's presence in the class depends on whether it rains or not. Let us denote the event of it raining with z. This is the **latent variable**
- Suppose that we do not directly know whether it is raining (we cannot look out of the window), but can only observe whether the student A is in the lesson



z x

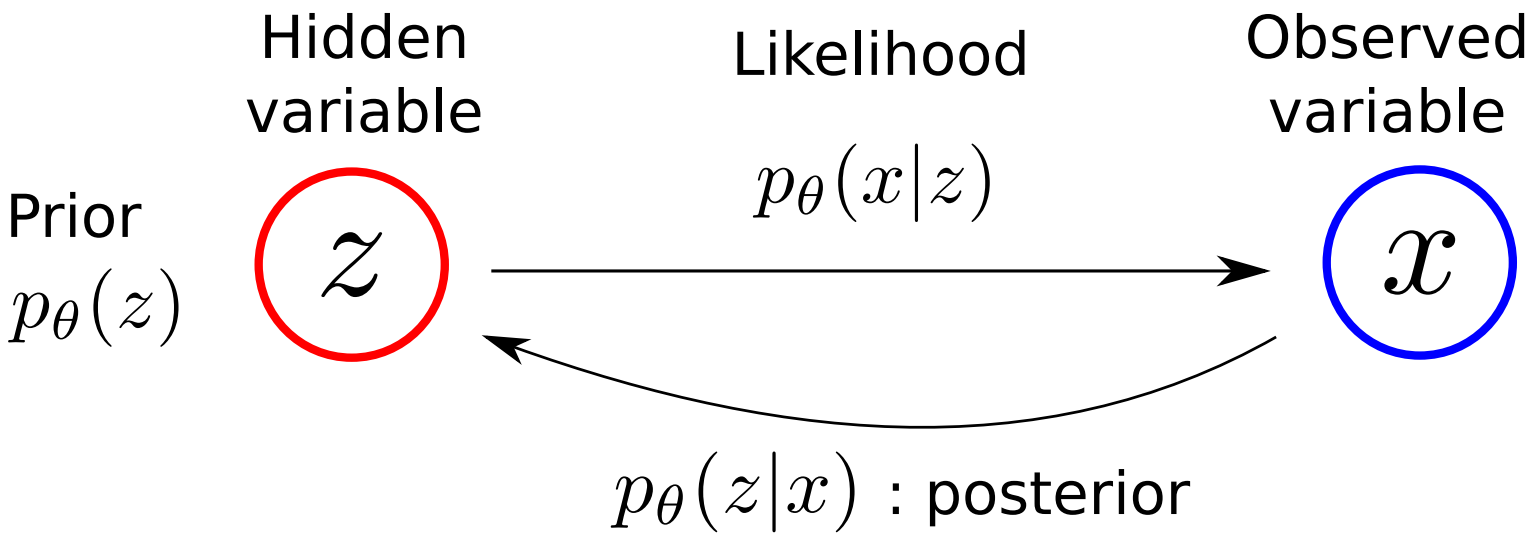
Variational autoencoder - variational Bayesian approach

- The probability of A being in the lesson, $\mathbb{P}(x)$, is the **marginal probability**
- The probability of it raining $\mathbb{P}(z)$ is the **prior probability**
- The probability of A being in the lesson given z, $\mathbb{P}(x|z)$ is the **likelihood**
- The probability of A it raining, given x, $\mathbb{P}(z|x)$ is the **posterior probability**



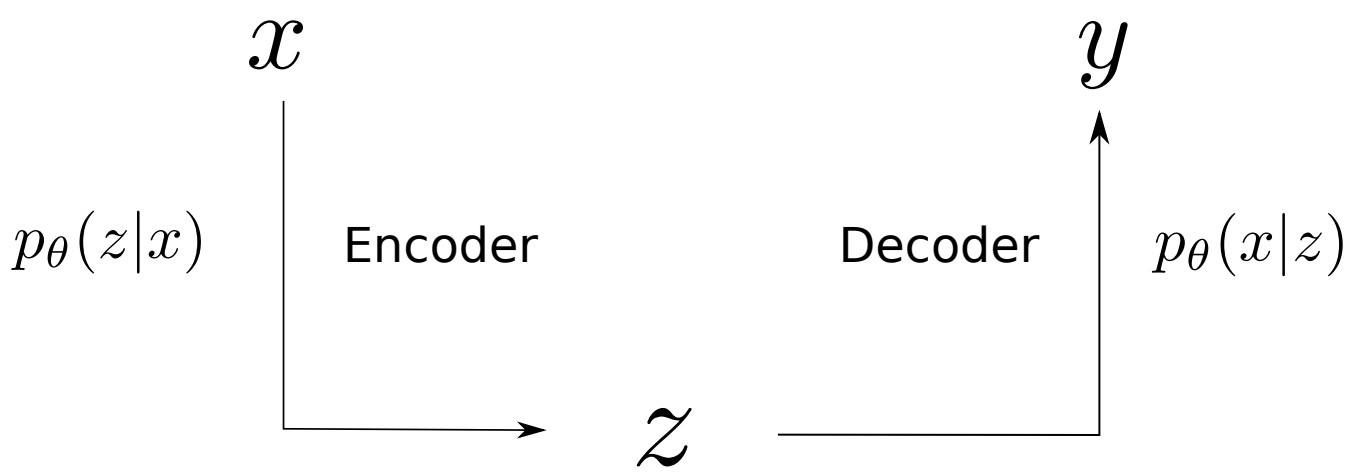
Variational autoencoder - variational Bayesian approach

- At this point, we assume that all the probabilities discussed have a **probability density function**
- We suppose that the distributions come from families of distrbutions parameterised by the parameters θ ($p_\theta(x)$, $p_\theta(x|z)$ etc)



Variational autoencoder - variational Bayesian approach

- There are some clear analogies with the autoencoder
- Encoder : posterior $p_\theta(z|x)$
- Decoder : likelihood $p_\theta(x|z)$



- We want to **impose the prior** $p_\theta(z)$!!

Variational autoencoder - variational Bayesian approach

- Often, we know (or we can at least estimate) the likelihood $p_\theta(x|z)$
- However, often the posterior distribution $p_\theta(z|x)$ is difficult to calculate, or intractable. Why is this the case ?

Variational autoencoder - variational Bayesian approach

- Often, we know (or we can at least estimate) the likelihood $p_\theta(x|z)$
- However, often the posterior distribution $p_\theta(z|x)$ is difficult to calculate, or intractable. Why is this the case ?

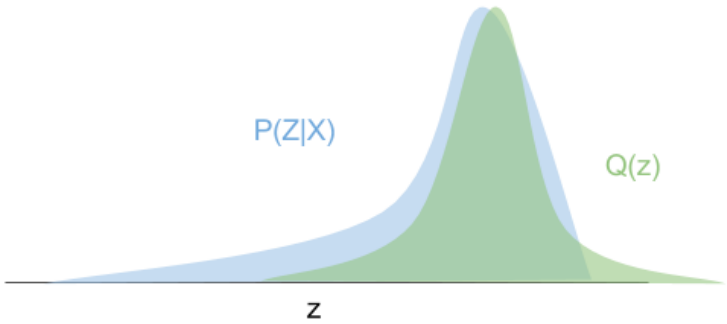
$$p_\theta(z|x) = \frac{p_\theta(x|z)\ \ p_\theta(z)}{p_\theta(x)} \qquad \text{Bayes's rule}$$

$$= \frac{p_\theta(x|z)\ \ p_\theta(z)}{\int p_\theta(x,z)dz} \qquad \text{Marginal distribution}$$

- $\int p_\theta(x,z)dz$ can be a very high-dimensional integral
- Calculating the posterior probability is known as the **inference problem**

Variational autoencoder - variational Bayesian approach

- So, how do we approach the problem of inference ?
- The variational Bayesian approach consists in using an **approximate distribution** $q_\phi(z|x) \approx p_\theta(z|x)$, which is easier to manipulate



- What does it mean for two probability distributions to be **similar** ?
  - Often, this is defined using the **Kullback-Leibler divergence**

Variational autoencoder

Kullback-Leibler divergence

Let p and q be two probability distributions defined over the same domain. The Kullback-Leibler divergence is defined as

$$KL(p \mid\mid q) = \int p(x) \log \frac{p(x)}{q(x)} dx \tag{2}$$

Variational autoencoder

Kullback-Leibler divergence

Let p and q be two probability distributions defined over the same domain. The Kullback-Leibler divergence is defined as

$$KL(p \mid\mid q) = \int p(x) \log \frac{p(x)}{q(x)} dx \tag{2}$$

The Kullback-Leibler divergence has some interesting properties :

- Non-negative : $KL(p \mid\mid q) \geq 0$
- $KL(p \mid\mid q) = 0 \iff p = q$ almost everywhere
- Non-symmetric : $KL(p \mid\mid q) \neq KL(q \mid\mid p)$

The second point is quite important. Why ? Because we know that **by minimising the KL divergence we are necessarily forcing p and q closer together**.

Variational autoencoder

- Let's come back to variational Bayesian methods now. We wish to approximate $p_\theta(z|x)$ with $q_\phi(z|x)$. To do this, we will find a $q_\phi*$:

$$q_\phi^* = \arg\min_{q_\phi} KL\left(q_\phi(z|x) \,||\, p_\theta(z|x)\right) \ ^\dagger \tag{3}$$

- Unfortunately, this does not help us much. Why ? **Because we don't know** $p_\theta(z|x)$!
- We will have to minimise $KL\left(q_\phi(z|x) \,||\, p_\theta(z|x)\right)$ some other way

† *You might notice that the q_ϕ before the p_θ. We do not go into the technical reasons here ...*

Variational autoencoders - The evidence lower bound

Evidence of Lower BOund (ELBO, lower bound of $\log p_\theta(x)$)

$$\text{ELBO}(q_\phi) = \mathbb{E}_{q_\phi}\left[\log(p_\theta(x,z))\right] - \mathbb{E}_{q_\phi}\left[\log q_\phi(z|x)\right]$$

- This is known as the "Evidence of Lower BOund" because :

$$\log p_\theta(x) = \text{ELBO}(q_\phi) + KL(q_\phi(z|x) \ \ || \ p_\theta(z|x)))$$

- KL divergence is positive : the ELBO is a lower bound for $\log p_\theta(x)$ (the "evidence")
- **By maximising the ELBO, we minimise the KL divergence between $q_\phi(z|x)$ and $p_\theta(z|x)$**

Variational autoencoders - Variational Autoencoder loss function

- We can rewrite the ELBO in the following manner

$$\text{ELBO}(q_\phi) = \mathbb{E}_{q_\phi}\left[\log(p_\theta(x|z))\right] - KL(q_\phi(z|x) \,||\, p_\theta(z))$$

- **We know all of these terms !** : we can use it as a VAE loss function !

Variational Autoencoder loss function

$$\mathcal{L}(x;\theta,\phi) = \overbrace{\mathbb{E}_{q_\phi}\left[\log(p_\theta(x|\ z))\right]}^{\text{Reconstruction error}} - \underbrace{KL(q_\phi(z|x) \,||\, p_\theta(z))}_{\text{Enforce the prior distribution}}$$

- **Important note !** : we want to **maximise** this loss function !

Variational Autoencoder summary

1. We modelled the autoencoding process using a probabilistic (Bayesian) framework, with an observed and a hidden variable
2. We wanted to calculate the posterior distribution $p_\theta(z|x)$, but this is complicated
3. We used an approximation $q_\phi(z|x)$ to $p_\theta(z|x)$
4. We used the ELBO as a loss function to minimise $KL(q_\phi(z|x)||p_\theta(z))$
   - Ensures a good reconstruction
   - Encourages the latent space to follow our chosen distribution (the prior $p_\theta(z)$)

Variational Autoencoder

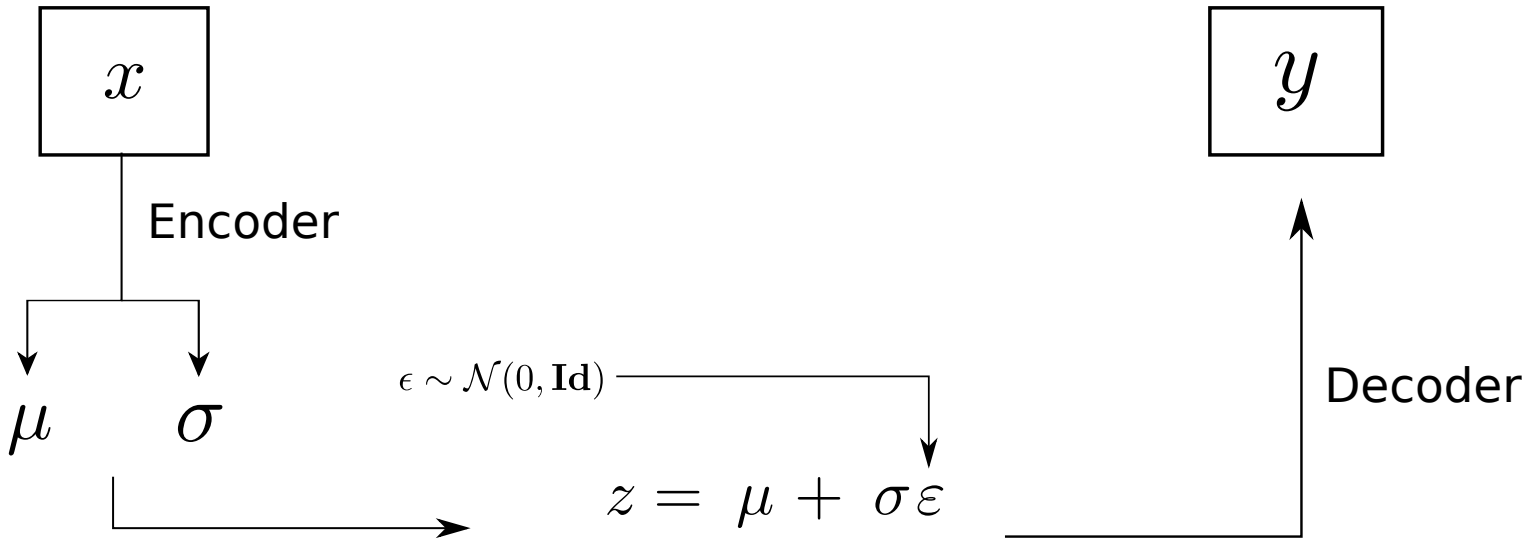
- There remains one more important detail : how to backpropagate through samples of z ? **Random variable, not differentiable**
- Solution : **"reparametrisation trick"**, make the random element an network input
- In the Gaussian case, where q_ϕ is a multivariate Gaussian vector, with mean μ and diagonal covariance matrix $\sigma \mathrm{Id}$, this gives

$$z = \mu + \sigma\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathrm{Id})$$

- μ and σ are produced by the encoder
- Thus, backpropagation can be carried out w.r.t to the parameters ϕ and θ

Variational autoencoder **in practice**

- The variational autoencoder is actually quite simple to implement
- Take the case of Gaussian $q_\phi(z|x)$

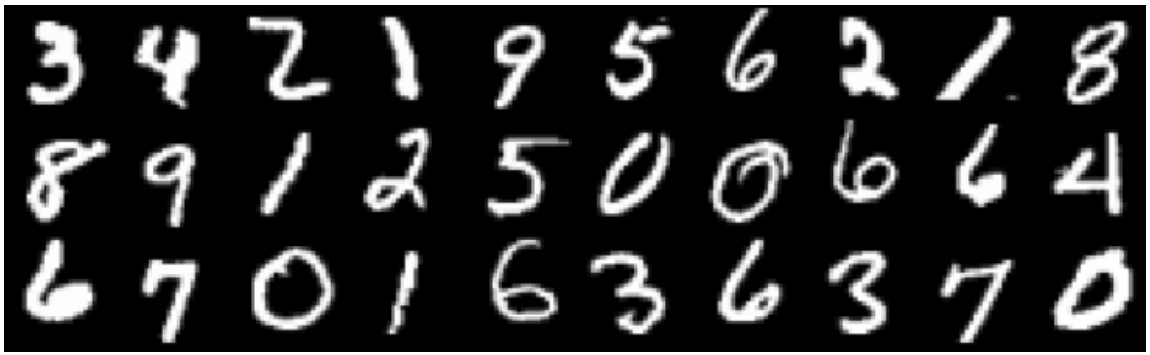


- Encoder and decoder can be MLPs, CNNs ...
- What is the loss in practice ?

Variational autoencoder **in practice**

- Let us take the following case, well-adapted to the **mnist dataset** :
  - Prior : $p_\theta(z) \sim \mathcal{N}(0, \mathrm{Id})$
  - Variational approximation : $q_\phi(z|x) \sim \mathcal{N}(\mu, \sigma \mathrm{Id})$, where $(\mu, \sigma) = E(x)$
  - Likelihood : $p_\theta(x|z) \sim Ber(y)$, where $y = D(z)$

$$\mathcal{L} = \overbrace{\sum_{i=1}^{mn} x_i \log y_i + (1 - x_i) \log(1 - y_i)}^{\text{Reconstruction error}} - \underbrace{\frac{1}{2} \sum_{j=1}^{d} \left(1 + \log\left(\sigma_j^2\right) - \mu_j^2 - \sigma_j^2\right)}_{\text{KL divergence}}$$



Variational autoencoder results

- Some results of variational autoencoders on mnist data : random samples

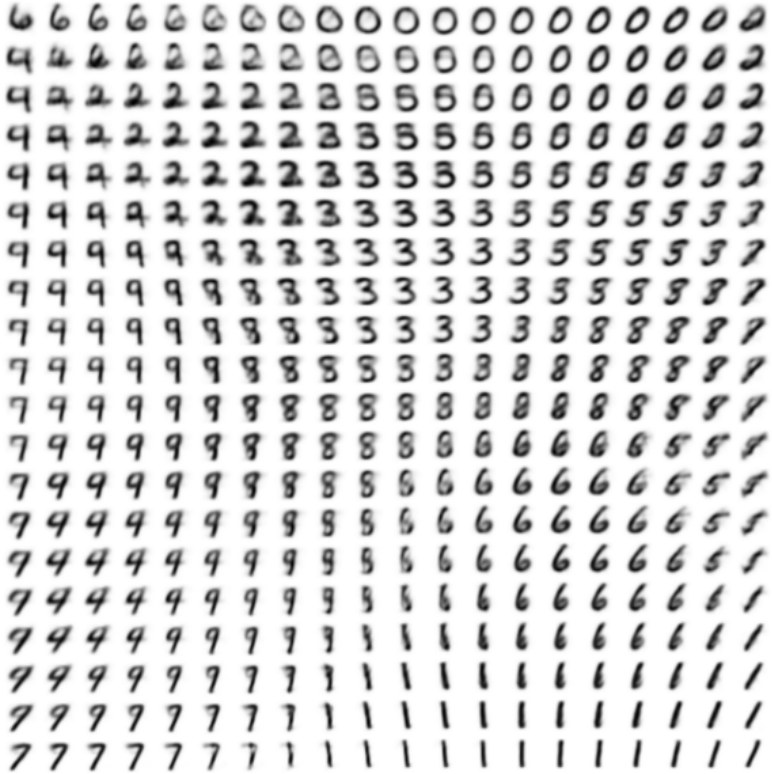
(a) 2-D latent space (b) 5-D latent space (c) 10-D latent space (d) 20-D latent space

Auto-Encoding Variational Bayes, *D. P. Kingma, M. Welling, arXiv preprint arXiv:1312.6114, 2013*

Variational autoencoder results

- Some results of VAEs on mnist, face data : uniform samples

(a) Learned Frey Face manifold



(b) Learned MNIST manifold

Auto-Encoding Variational Bayes, *D. P. Kingma, M. Welling, arXiv preprint arXiv:1312.6114, 2013*

Variational autoencoder results

- Some results of VAEs on more complex digits data

Variational Autoencoders : summary

- Rigourous framework to autoencode data onto a probabilisitcally modelled latent space
- Advantages
  - Theoretically-motivated, loss function meaningful
  - Learn to and from mapping (encoder and decoder)
- Drawbacks
  - Have to re-write loss function for each different model, not always easy
  - In practice, do not produce as complex examples as **Generative Adversarial Networks**, which we see now !

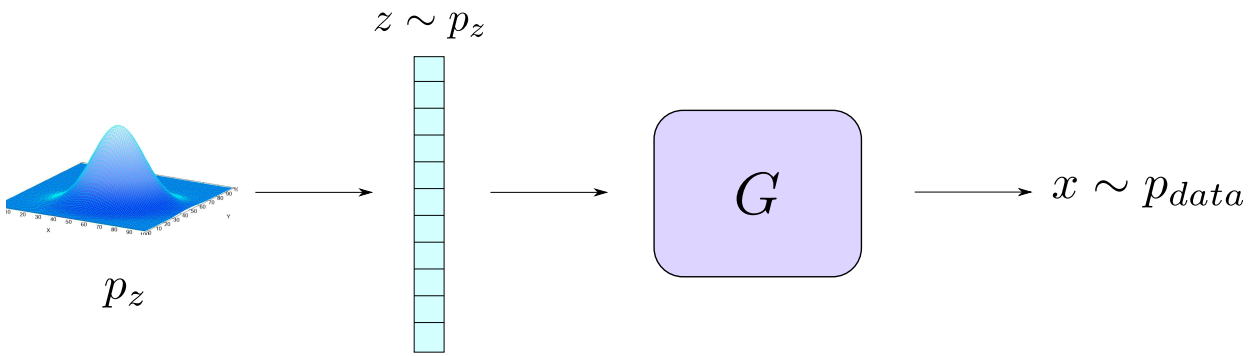
Generative Adversarial networks

- We saw that variational autoencoders were not straightforward to adapt to new situations
- **Generative adversarial networks** (GANs)* are another generative model that generate random examples of high-dimensional data

* ***Generative Adversarial Nets***, *Goodfellow et al, NIPS 2014*

Generative Adversarial networks

- The GAN contains only the **decoder** part of an autoencoder
  - The code z is **explicitly sampled** from a chosen distribution p_z (contrary to the VAE)
- The decoder is referred to here as the "Generator"



- We suppose that the data in the databse follows a distribution p_{data}
- We want to make the distribution of $x = G(z)$, p_G similar to p_{data}*

* *Why can we not do this via the KL divergence as before ? Too high dimensionality (previously, we worked in the latent space)*

Generative Adversarial networks

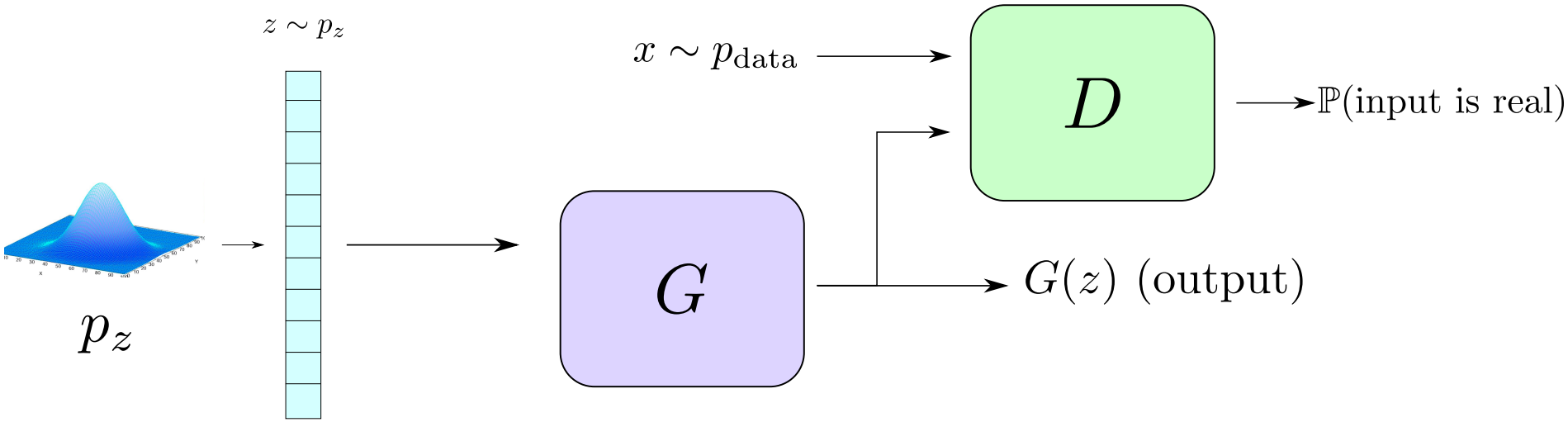
- However, with no reconstruction error, how do we make x look like the data ?
- Answer : Train another network : a **Discriminator** D (or "Adversarial Network")

Generative Adversarial networks

- However, with no reconstruction error, how do we make x look like the data ?
- Answer : Train another network : a **Discriminator** D (or "Adversarial Network")
- $D : \mathbb{R}^{mn} \to [0, 1]$ is trained to **identify "good" (or "true")** examples of the data
- $G : \mathbb{R}^{z} \to \mathbb{R}^{mn}$ is trained to **produce realistic data examples**
- The two networks are trained at the same time, and **each try to fool the other !**

Generative Adversarial networks

- The full GAN architecture looks like this



Generative Adversarial networks

- The discriminator is a really interesting idea, why ?
- Reliable and powerful image/data models are difficult to establish
- It is difficult to say whether an image is "good" or not
  - The discriminator acts as a **learned image norm** !
  - It can be used for other purposes also ...
- How is this is achieved ? Via a well-designed loss function

GAN loss

- Train generator G and the discriminator A in a minmax optimisation problem

$$\min_G \max_D \overbrace{\mathbb{E}_{x\sim p_{data}}[\log D(x)]}^{D \text{ is trying to recognize true data}} + \underbrace{\mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))]}_{\substack{G \text{ is trying to fool } D, \\ \text{but } D \text{ is trying not to be fooled}}}$$

GAN loss

- Train generator G and the discriminator A in a minmax optimisation problem

$$\min_G \max_D \overbrace{\mathbb{E}_{x\sim p_{data}}[\log D(x)]}^{D \text{ is trying to recognize true data}} + \underbrace{\mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))]}_{\substack{G \text{ is trying to fool } D, \\ \text{but } D \text{ is trying not to be fooled}}}$$

- Minimisation w.r.t G
  - Second term is low, $\implies$ $1-D(G(z))$ is close to 0 $\implies$ D is recognising $G(z)$ as a true data example : G **has fooled** D

GAN loss

- Train generator G and the discriminator A in a minmax optimisation problem

$$\min_G \max_D \overbrace{\mathbb{E}_{x\sim p_{data}}[\log D(x)]}^{D \text{ is trying to recognize true data}} + \underbrace{\mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))]}_{\substack{G \text{ is trying to fool } D, \\ \text{but } D \text{ is trying not to be fooled}}}$$

- Minimisation w.r.t G
  - Second term is low, $\implies$ $1-D(G(z))$ is close to 0 $\implies$ D is recognising $G(z)$ as a true data example : G **has fooled** D
- Maximisation w.r.t D
  - First term is high $\implies$ $D(x)$ is close to 1 : D **is learning to recognize true data**
  - Second term is high $\implies$ $1-D(G(z))$ is close to 1 : D **is not getting fooled by** G

Generative Adversarial networks

- At the beginning of the training, the examples from G are not very good : D can spot them easily
- At the end of training, the discriminator should not be able to tell the true data from the generated data : $p_G = p_{data}$
- Optimisation alternates between minimisation and maximisation steps

Generative Adversarial networks

- At the beginning of the training, the examples from G are not very good : D can spot them easily
- At the end of training, the discriminator should not be able to tell the true data from the generated data : $p_G = p_{data}$
- Optimisation alternates between minimisation and maximisation steps
- Are we sure that this loss is well-designed for this purpose ?
- In fact, we can prove that this is the case

Generative Adversarial networks

- First, we establish the following lemma :

Optimal GAN discriminator D^*

For a fixed G, the optimal D is $D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_G(x)}$

Generative Adversarial networks

- First, we establish the following lemma :

Optimal GAN discriminator D^*

For a fixed G, the optimal D is $D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_G(x)}$

$$\begin{aligned}\mathcal{L}(G, D) &= \mathbb{E}_{x\ \sim p_{data}}[\log D(x)] + \mathbb{E}_{z\ \sim p_z}[\log(1 - D(G(z)))] \\ &= \int_x p_{data}(x)\log(D(x))dx + \int_z p_z(z)\log(1 - D(G(z)))dz \\ &= \int_x p_{data}(x)\log(D(x)) + p_G(x)\log(1 - D(x))dx.\end{aligned}$$

Generative Adversarial networks

- First, we establish the following lemma :

Optimal GAN discriminator D^*

For a fixed G, the optimal D is $D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_G(x)}$

$$\begin{aligned}\mathcal{L}(G, D) &= \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \\ &= \int_x p_{data}(x)\log(D(x))dx + \int_z p_z(z)\log(1 - D(G(z)))dz \\ &= \int_x p_{data}(x)\log(D(x)) + p_G(x)\log(1 - D(x))dx.\end{aligned}$$

- For every x, the maximum of the previous equation w.r.t $D(x)$ is

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_G(x)}$$

Generative Adversarial networks

- Given this lemma, we can now state the main optimality theorem

Global optimum of the GAN loss function

The global optimum of the GAN loss function is achieved if and only if $p_G = p_{data}$. At this point $\mathcal{L}(G, D) = -\log 4$.

Generative Adversarial networks

- Given this lemma, we can now state the main optimality theorem

Global optimum of the GAN loss function

The global optimum of the GAN loss function is achieved if and only if $p_G = p_{data}$. At this point $\mathcal{L}(G, D) = -\log 4$.

- First, our previous lemma allows us to rewrite the loss function

$$\max_D \mathcal{L}(G, D) = \mathbb{E}_{x\ \sim p_{data}} [\log D^*(x)] + \mathbb{E}_{z\ \sim p_z} [\log(1 - D^*(G(z)))] \tag{4}$$

$$= \mathbb{E}_{x\ \sim p_{data}} \left[\log \frac{p_{data}(x)}{p_{data}(x) + p_G(x)}\right] + \mathbb{E}_{x\ \sim p_G} \left[\log \left(\frac{p_G(x)}{p_{data}(x) + p_G(x)}\right)\right]. \tag{5}$$

Generative Adversarial networks

- Given this lemma, we can now state the main optimality theorem

Global optimum of the GAN loss function

The global optimum of the GAN loss function is achieved if and only if $p_G = p_{data}$. At this point $\mathcal{L}(G, D) = -\log 4$.

- First, our previous lemma allows us to rewrite the loss function

$$\max_D \mathcal{L}(G, D) = \mathbb{E}_{x\ \sim p_{data}} \left[\log D^*(x)\right] + \mathbb{E}_{z\ \sim p_z} \left[\log(1 - D^*(G(z)))\right] \tag{4}$$

$$= \mathbb{E}_{x\ \sim p_{data}} \left[\log \frac{p_{data}(x)}{p_{data}(x) + p_G(x)}\right] + \mathbb{E}_{x\ \sim p_G} \left[\log \left(\frac{p_G(x)}{p_{data}(x) + p_G(x)}\right)\right]. \tag{5}$$

- Therefore, if $p_G = p_{data}$, then $\mathcal{L}(G, D) = \log \frac{1}{2} + \log \frac{1}{2} = -\log(4)$

Generative Adversarial networks

- Now, we are going to show that $-\log(4)$ is the optimal value of the loss function

Generative Adversarial networks

- Now, we are going to show that $-\log(4)$ is the optimal value of the loss function
- First, we remark that :

$$-\log(4) = \mathbb{E}_{x\ \sim p_{data}}\left[-\log(2)\right] + \mathbb{E}_{x\ \sim p_G}\left[-\log(2)\right]. \tag{6}$$

Generative Adversarial networks

- Now, we are going to show that $-\log(4)$ is the optimal value of the loss function
- First, we remark that :

$$-\log(4) = \mathbb{E}_{x\ \sim p_{data}}\left[-\log(2)\right] + \mathbb{E}_{x\ \sim p_G}\left[-\log(2)\right]. \tag{6}$$

- Therefore, by subtracting Equation 6 from Equation 5, we have

$$\begin{aligned}\mathcal{L}(G, D^*) &= -\log(4) + \int p_{data}(x) \log \frac{p_{data}(x)}{\frac{1}{2}\left(p_{data}(x) + p_G(x)\right)} dx \ + \\ &\qquad \int p_G(x) \log \frac{p_{data}(x)}{\frac{1}{2}\left(p_{data}(x) + p_G(x)\right)} dx \\ &= -\log(4) + KL\left(p_{data} \,||\, \frac{p_{data} + p_G}{2}\right) + KL\left(p_G \,||\, \frac{p_{data} + p_G}{2}\right).\end{aligned}$$

Generative Adversarial networks

- This can also be rewritten as

$$\mathcal{L}(G, D^*) = -\log(4) + 2\,\mathbf{JSD}\left(\mathbf{p_{data}} \,||\, \mathbf{p_G}\right).$$

- The **JSD** is the **Jensen-Shannon divergence**
- This is another distance between distributions. For p and q, we have :

$$JSD(p, q) = \frac{1}{2} KL\left(p \,||\, \frac{1}{2}(p+q)\right) + \frac{1}{2} KL\left(q \,||\, \frac{1}{2}(p+q)\right)$$

Generative Adversarial networks

- This can also be rewritten as

$$\mathcal{L}(G, D^*) = -\log(4) + 2\,\mathbf{JSD}\left(\mathbf{p_{data}} \,||\, \mathbf{p_G}\right).$$

- The **JSD** is the **Jensen-Shannon divergence**
- This is another distance between distributions. For p and q, we have :

$$JSD(p, q) = \frac{1}{2} KL\left(p \,||\, \frac{1}{2}(p+q)\right) + \frac{1}{2} KL\left(q \,||\, \frac{1}{2}(p+q)\right)$$

- The JSD is **non-negative and equal to zero if and only if** $p_{data} = p_G$
- Therefore $-\log(4)$ is the optimal value, and only reached when $p_{data} = p_G$

Generative Adversarial networks

- To summarise, we know that **by minimising the GAN loss, we are sure to encourage p_G to approximate p_{data}**
- In spite of this theoretical result (and others), training GANs is very difficult and is a current hot topic of research

Generative Adversarial networks

- Here are some results of the original GAN paper*

- In the space of four years, these results have been vastly improved on
- There are many, many GAN variants. We present a few now

* ***Generative Adversarial Nets***, *Goodfellow et al, NIPS 2014*

Conditional Generative Adversarial networks

- The Conditional GAN allows a label $\mathbf{c}$ to be added to the loss function
- It is then possible to generate examples of a given class

$$\min_G \max_D \left[\log D(x|\mathbf{c})\right] + \left[\log\left(1 - D(G(z|\mathbf{c}))\right)\right]$$

[*] ***Conditional Generative Adversarial Nets***, *Mirza, M. and Osindero, S., arXiv preprint arXiv:1411.1784, 2014*

Generative Adversarial networks

- Examples of results of Conditional GAN

* ***Conditional Generative Adversarial Nets***, *Mirza, M. and Osindero, S., arXiv preprint arXiv:1411.1784, 2014*

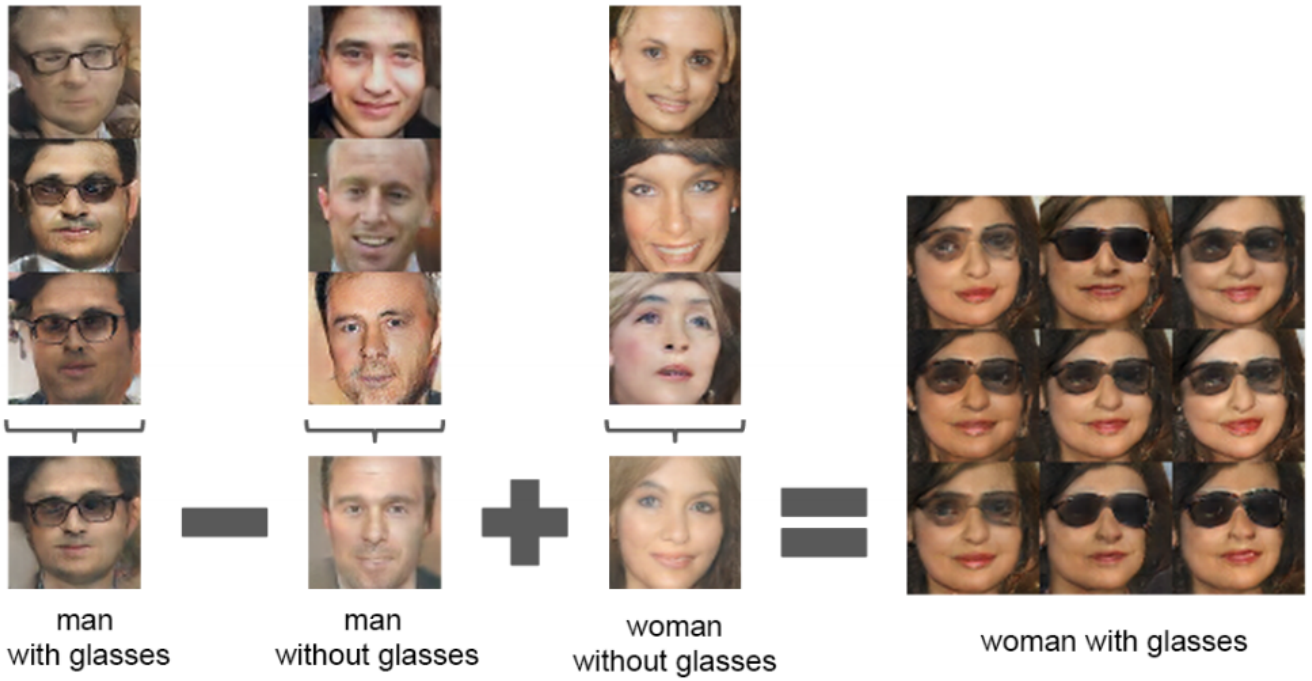
Generative Adversarial networks

- Deep Convolutional Generative Adversarial Networks (DCGAN)
- More complex data, sharper generation

* ***Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks***, *Radford, A. and Chintala, S., arXiv:1511.06434, 2015*

Generative Adversarial networks

- Deep Convolutional Generative Adversarial Networks (DCGAN)
- Linear algebra in the latent space



* ***Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks***, *Radford, A. and Chintala, S., arXiv:1511.06434, 2015*

Generative Adversarial networks

- A big problem of the GAN is known as **mode collapse**. This occurs when the training leads a GAN to produce the same result all the time

Generative Adversarial networks

- A big problem of the GAN is known as **mode collapse**. This occurs when the training leads a GAN to produce the same result all the time
- The "Wasserstein GAN"[†] modifies the loss function by using a different distance between probabilities : the Wasserstein distance

$$W(p,q) = \inf_{\gamma \in \prod(p,q)} \mathbb{E}_{(x,y)\sim\gamma} \left[\|x - y\| \right]$$

- $\prod(p,q)$ is the set of all the joint distributions of (x,y) whose marginals are p and q
- Some sequences of distributions **converge under the Wasserstein distance, but not other distances** (the KL divergence, for example)
- Using this formulation stabilises the GAN training

[†] ***Wasserstein GAN***, *Ajovsky, M., Chintala, S. and, Bottou, L. arXiv preprint arXiv:1701.07875, 2017*

Generative Adversarial networks

- Very recently, extremely realistic, high-resolution results have been achieved by different research teams[†]

[†] ***Progressive growing of gans for improved quality, stability, and variation***, *Karras, T., Aila, T., Laine, S., and Lehtine, J., arXiv preprint arXiv:1710.10196, 2017*

Generative Adversarial networks

- Very recently, extremely realistic, high-resolution results have been achieved by different research teams

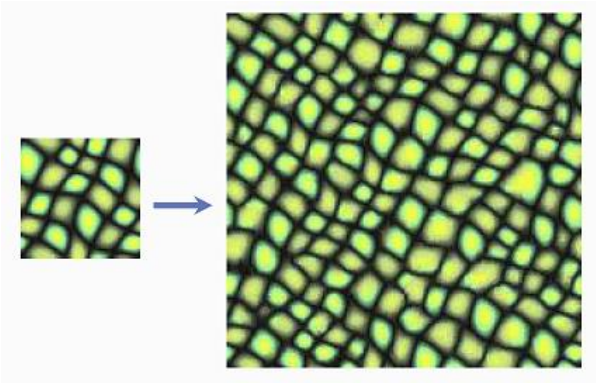
† ***Progressive growing of gans for improved quality, stability, and variation***, *Karras, T., Aila, T., Laine, S., and Lehtine, J., arXiv preprint arXiv:1710.10196, 2017*

Summary on GANs

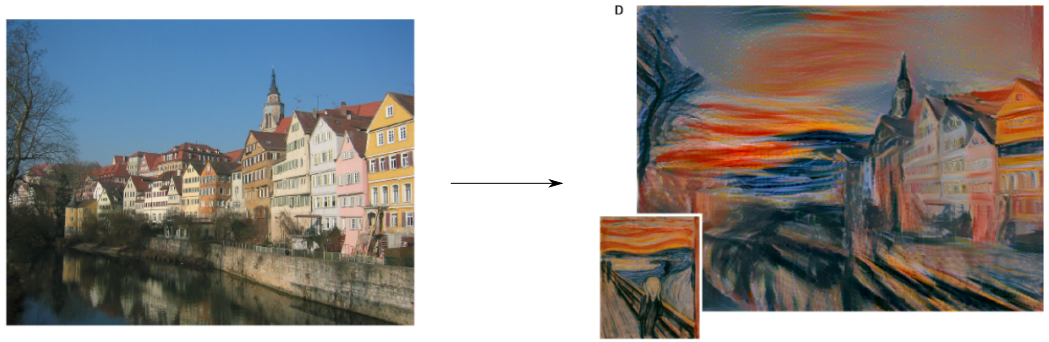
- GANs are a powerful and generic way of producing **random examples of complex images**
- However, they are **notoriously difficult to train**, this is a current area of research
- A significant disadvantage with respect to variational autoencoders is that **GANs do not train an encoder** : it is a one-way transformation
  - Often, for restoration or other inverse problems, it is useful to have an "inverse" transformation

Texture synthesis and style transfer

- We have seen two methods for creating random examples of complex images
- There is another type of generative model specifically designed for **texture synthesis and style transfer**
- These are two very common tasks in image and video editing



Texture synthesis



Style transfer

Texture synthesis and style transfer

- This set of methods is based on the work of Gatys et al.[†,‡], similar approach as **deep dream** and variants
- We suppose that texture and style are **coded somewhere in the layers of a neural network**
  - Small patterns, repetitve motifs
- We will iterate gradient descent on an image to encourage similar representations inside the network

$$x^{n+1} = x^n + \nabla_x \mathcal{L}$$

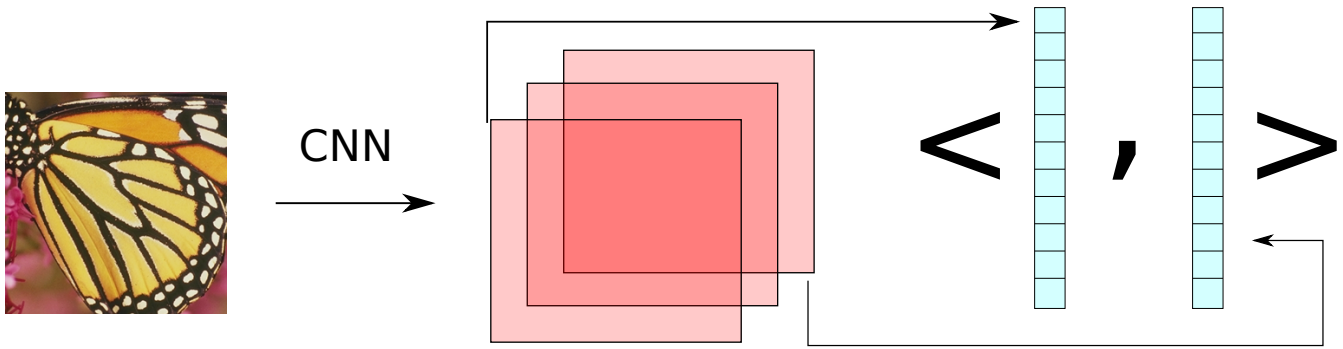
[†] ***Texture synthesis using convolutional neural networks***, *Gatys, L. A., Ecker, A. S, and Bethge, M., NIPS, 2015*
[‡] ***A Neural Algorithm of Artistic Style***, *Gatys, L. A., Ecker, A. S, and Bethge, M., arXiv:1508.06576, 2015*

Texture synthesis

- First, let's take the case of **texture synthesis**, suppose we have an image $\hat{x}$ whose texture we want to copy
- Let $u^{\ell}_{i,j}$ be the response of a neural network (often a classification CNN) to an image at layer ℓ for channel i at position j

Texture synthesis

- First, let's take the case of **texture synthesis**, suppose we have an image $\hat{x}$ whose texture we want to copy
- Let $u^{\ell}_{i,j}$ be the response of a neural network (often a classification CNN) to an image at layer ℓ for channel i at position j
- We define $C^{\ell}_{i,j} = \sum_k u^{\ell}_{i,k} u^{\ell}_{j,k}$, the dot product between two channels
- This represents the **correlations between different channels** in the CNN : a characteristic of the texture



Dot product between two feature layers

Texture synthesis

- Why is this quantity characteristic of texture ? Textures are **stationary random processes**

- This means that the statistical properties of textures **should not depend on spatial location**[†]
- Therefore, we carry out the dot product to remove spatial dependency

[†] ***A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients***, *Portilla, J and Simoncelli, E. P, International Journal of Computer Vision, 2000*

Texture synthesis and style transfer

- We try to find an output image y which minimises the following loss :

$$\mathcal{L}_{\text{Texture}}(y;\hat{x}) = \sum_{\ell} w_l \sum_{i,j} \frac{1}{2}\left(C^{\ell}_{i,j} - \hat{C}^{\ell}_{i,j}\right)^2$$

- w_ℓ is a weighting function for each layer

Texture synthesis algorithm

$y^0 \leftarrow$ random white noise
for $i = 1$ to N **do**
  $y^{n+1} = y^n - \varepsilon \nabla_x \mathcal{L}_{\text{Texture}}(y^n)$
end for

Texture synthesis

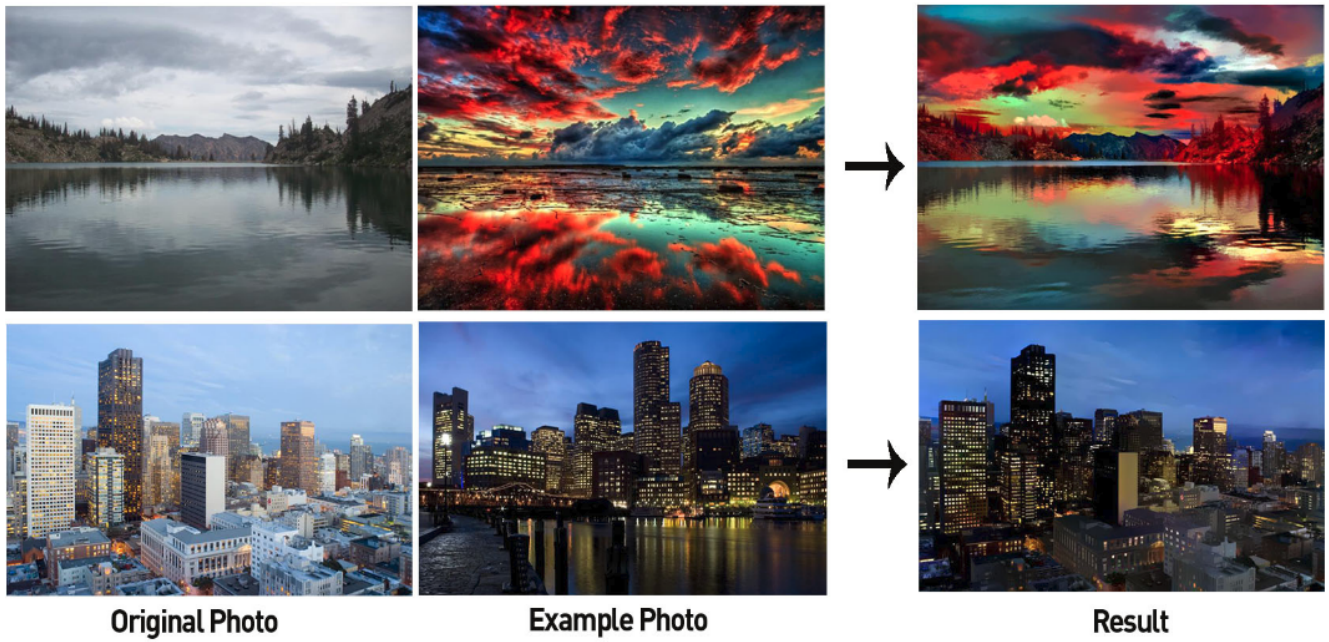
- Here are some image texture synthesis examples with this method, and variants

† ***Texture Networks: Feed-forward Synthesis of Textures and Stylized Image***, *Ulyanov et al., axrXiv, 2016*

Style transfer

- Now, we look at the case of **style transfer**. We have two images :
  - $\tilde{x}$ whose **content** we wish to copy
  - $\hat{x}$ whose **style** we wish to copy

- Contrary to texture, **content is spatially dependent**



† *Image from **Deep Photo Style Transfer**, Luan et al., arXiv:1703.07511v3, 2017*

Style transfer

- We define a content loss :

$$\mathcal{L}_{\text{Content}}(y; \tilde{x}, \ell) = \sum_{i,j} \left(u^{\ell}_{i,j} - \tilde{u}^{\ell}_{i,j} \right)^2$$

- The final loss is a mixture of content and style (texture) losses :

$$\mathcal{L}(y; \tilde{x}, \hat{x}) = \alpha \mathcal{L}_{content}(y; \tilde{x}) + \beta \mathcal{L}_{texture}(y; \hat{x})$$

- The algorithm to produce the end result is the same : random initialisation and iterate gradient descent

Style transfer - some results†

† *https://github.com/lengstrom/fast-style-transfer/*

Summary of texture synthesis and style transfer

- Inter-channel correlations of neural network layers provide a powerful model of texture/style
- This gives a **flexible, easy-to-implement** algorithm : random initialisation and iterate gradient descent
- Main drawback : only applicable to a restricted class of problems (texture, style)

Summary

1. Convolutional Neural Networks
   - Introduction, notation
     - Convolutional Layers
   - Down-sampling and the receptive field
     - Image classification datasets and well-known CNNs
   - Applications of CNNs
     - Adversarial Examples
     - Visualising CNNs
2. Generative Models
   - Autoencoders
     - Vanilla Autoencoder
     - Autoencoder variants
   - Generative models
     - Variational autoencoders
     - Generative Adversarial Networks
     - Texture and style models
3. Summary

Summary

Conclusion

We have seen three types of generative models

1. Variational autoencoders
   - An Autoencoder whose latent space is encouraged to follow a certain distribution
   - Variational Bayesian formulation
2. Generative Adversarial Networks
   - A generator trained to create realistic data
   - A discriminator trained to identify true and false data examples
3. Models for texture synthesis and style transfer
   - Iterative method to encourage similar intra-layer correlations of neural network
   - Start with an initial point, iterate gradient descent w.r.t image

Summary of advantages and weaknesses of Generative Models

Model/method	Advantages	Disadvantages
VAE	• Rigourous formulation • Encoder and decoder trained	• Loss must be rewritten for different probability distributions (not easy) • In practice, more limited applications than GANs
GAN	• Highly flexible • Applied to complex data, impressive results	• Difficult to train • No reverse transformation (encoder)
Texture/style model	• Versatile, simple algorithm • Produces best texture/style results	• Only applicable to limited cases