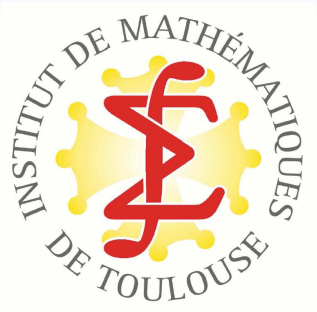


Neural Networks and Introduction to Deep Learning

Formation continue
SII- Nov 2018

Béatrice Laurent-Philippe Besse

INSA Toulouse, Institut de Mathématiques de Toulouse

Outline

- Introduction
- Neural Networks
- Convolutional Neural Networks
- Autoencoders and GANs
- Recurrent Neural Networks

Introduction

- Deep learning is a set of learning methods attempting to model data with complex architectures combining different non-linear transformations.
- The elementary bricks of deep learning are the neural networks, that are combined to form the deep neural networks.

There exist several types of architectures for neural networks :

- The multilayer perceptrons, that are the oldest and simplest ones
- The Convolutional Neural Networks (CNN), particularly adapted for image processing
- The recurrent neural networks, used for sequential data such as text or times series.

Introduction

- Deep learning architectures are based on deep cascade of layers.
- They need clever stochastic optimization algorithms, and initialization, and also a clever choice of the structure.
- They lead to very impressive results, although very few theoretical fondations are available till now.
- These techniques have enabled significant progress in the fields of sound and image processing, including facial recognition, speech recognition, computer vision, automated language processing, text classification (for example spam recognition).

Outline

- Introduction
- Neural Networks
- Convolutional Neural Networks
- Autoencoders and GANs
- Recurrent Neural Networks

Neural networks

- An artificial neural network is non linear with respect to its parameters θ that associates to an entry x an output $y = f(x, \theta)$.
- The neural networks can be use for regression or classification.
- The parameters θ are estimated from a learning sample.
- The function to minimize is not convex, leading to local minimizers.
- The success of the method came from a universal approximation theorem due to Cybenko (1989) and Hornik (1991).
- Le Cun (1986) proposed an efficient way to compute the gradient of a neural network, called backpropagation of the gradient, that allows to obtain a local minimizer of the quadratic criterion easily.

Artificial Neuron

Artificial neuron

- a function f_j of the input $x = (x_1, \ldots, x_d)$
- weighted by a vector of connection weights $w_j = (w_{j,1}, \ldots, w_{j,d})$,
- completed by a neuron bias b_j,
- and associated to an activation function ϕ :

$$y_j = f_j(x) = \phi(\langle w_j, x \rangle + b_j).$$

Activation functions

Activation functions

- The identity function $\phi(x) = x$
- The sigmoid function (or logistic) $\phi(x) = \frac{1}{1+e^{-x}}$
- The hyperbolic tangent function ("tanh")

$$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- The hard threshold function $\phi_\beta(x) = \mathbb{1}_{x \geq \beta}$
- The Rectified Linear Unit (ReLU) activation function $\phi(x) = \max(0, x)$

Activation functions

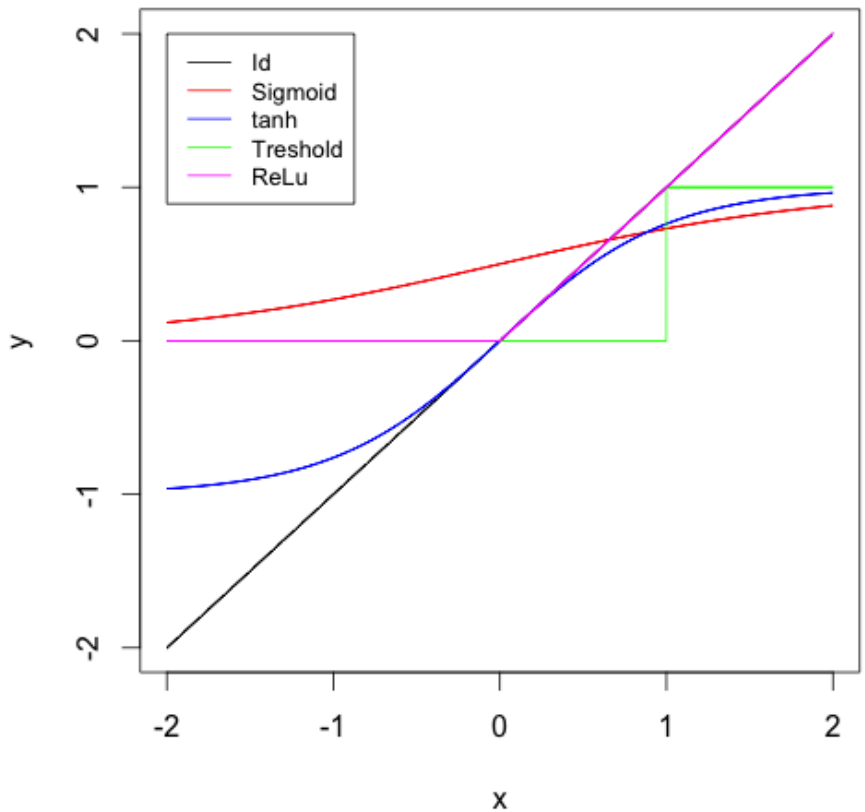
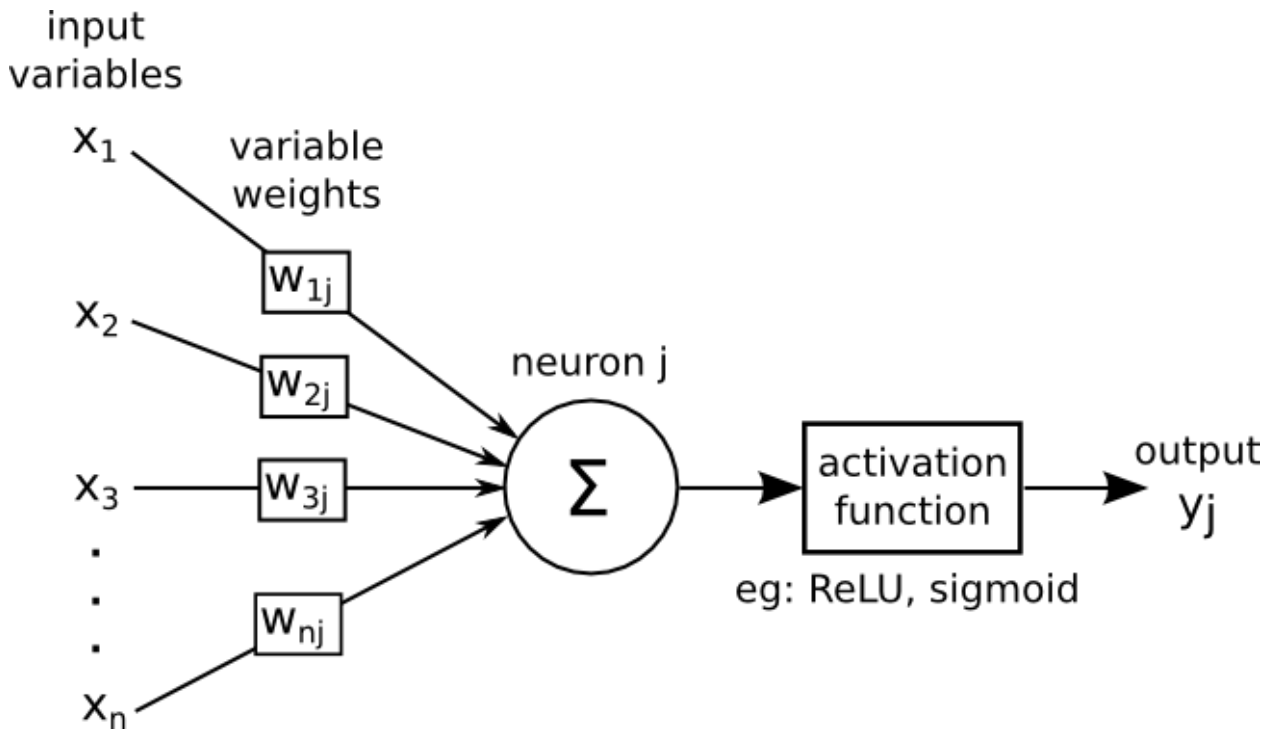


FIGURE: Activation functions

Artificial Neuron

Schematic representation of an artificial neuron where $\Sigma = \langle w_j, x \rangle + b_j$.



- Historically, the sigmoid was the mostly used activation function since it is differentiable and allows to keep values in the interval $[0, 1]$.
- Nevertheless, it is problematic since its gradient is very close to 0 when $|x|$ is not close to 0.

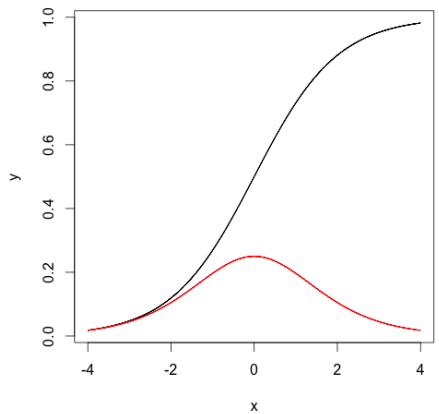


FIGURE: Sigmoid function (in black) and its derivatives (in red)

Activation functions

- With neural networks with a high number of layers, this causes troubles for the backpropagation algorithm to estimate the parameters.
- This is why the sigmoid function was supplanted by the rectified linear function (ReLu).
- This function is not differentiable in 0 but the probability to have an entry equal to 0 is generally null.
- The ReLU function and its derivative are equal to 0 for negative values, hence it is advised
  - to add a small positive bias to ensure that each unit is active.
  - to consider a variations of the ReLU function such as

$$\phi(x) = \max(x, 0) + \alpha \min(x, 0)$$

where α is either a fixed parameter set to a small positive value, or a parameter to estimate.

Multilayer perceptron

- A multilayer perceptron (or neural network) is a structure composed by several hidden layers of neurons where the output of a neuron of a layer becomes the input of a neuron of the next layer.
- On last layer, called output layer, we may apply a different activation function as for the hidden layers depending on the type of problems we have at hand : regression or classification.

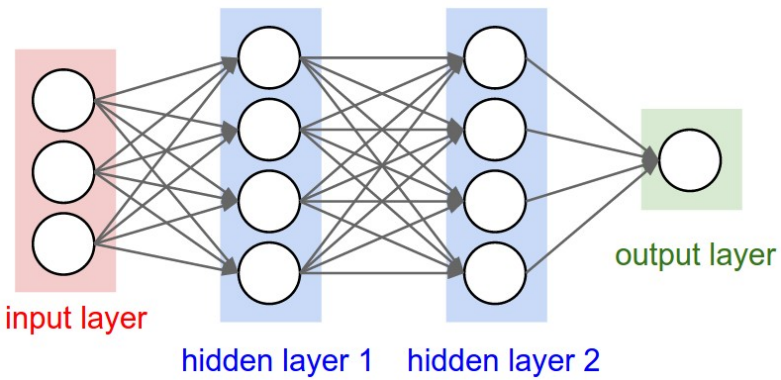


FIGURE: A basic neural network.

Multilayers perceptrons

- The parameters of the architecture are the number of hidden layers and of neurons in each layer.
- The activation functions are also to choose by the user. On the output layer, we apply no activation function in the case of regression.
- For binary classification, the output gives a prediction of $\mathbb{P}(Y = 1/X)$ since this value is in $[0, 1]$, the sigmoid activation function is generally considered.
- For multi-class classification, the output layer contains one neuron per class i, giving a prediction of $\mathbb{P}(Y = i/X)$. The sum of all these values has to be equal to 1.
- The multidimensional function **softmax** is generally used

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

Mathematical formulation

Multilayer perceptron with L hidden layers :

- We set $h^{(0)}(x) = x$.
  For $k = 1, \ldots, L$ (hidden layers),

$$\begin{aligned} a^{(k)}(x) &= b^{(k)} + W^{(k)} h^{(k-1)}(x) \\ h^{(k)}(x) &= \phi(a^{(k)}(x)) \end{aligned}$$

  For $k = L + 1$ (output layer),

$$\begin{aligned} a^{(L+1)}(x) &= b^{(L+1)} + W^{(L+1)} h^{(L)}(x) \\ h^{(L+1)}(x) &= \psi(a^{(L+1)}(x)) := f(x, \theta). \end{aligned}$$

  where ϕ is the activation function and ψ is the output layer activation function

- At each step, $W^{(k)}$ is a matrix with number of rows the number of neurons in the layer k and number of columns the number of neurons in the layer $k - 1$.

Estimation of the parameters

- Once the architecture of the network has been chosen, the parameters (the weights w_j and biases b_j) have to be estimated from a learning sample.
- As usual, the estimation is obtained by minimizing a loss function with a gradient descent algorithm.
- We first have to choose the loss function.
- It is classical to estimate the parameters by minimizing the loss function which is the opposite of the log likelihood.
- Denoting θ the vector of parameters to estimate, we consider the expected loss function

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}(\log(p_\theta(Y/X)).$$

Loss functions

- If the model is Gaussian, namely if $P_\theta(Y/X = x) \sim \mathcal{N}(f(x,\theta), I)$, maximizing the likelihood is equivalent to minimize the quadratic loss

$$L(\theta) = \mathbb{E}_{(X,Y)\sim P}(\|Y - f(X,\theta)\|^2).$$

- For binary classification, with $Y \in \{0,1\}$, maximizing the log likelihood corresponds to the minimization of the cross-entropy. Setting $f(X,\theta)) = P_\theta(Y = 1/X)$,

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}\left[Y \log(f(X,\theta)) + (1-Y)\log(1 - f(X,\theta))\right].$$

- This loss function is well adapted with the sigmoid activation function since the use of the logarithm avoids to have too small values for the gradient.

Loss functions

- For a multi-class classification problem, we consider a generalization of the previous loss function to k classes

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}\left[\sum_{j=1}^{k} \mathbb{1}_{Y=j} \log p_\theta(Y = j/X)\right].$$

- Ideally we would like to minimize the classification error, but it is not smooth, this is why we consider the cross-entropy (or eventually a convex surrogate).

Penalized empirical risk

- The expected loss can be written as

$$L(\theta) = \mathbb{E}_{(X,Y)\sim P}\left[\ell(f(X,\theta), Y)\right]$$

  and it is associated to a loss function ℓ.

- In order to estimate the parameters θ, we use a training sample $(X_i, Y_i)_{1\leq i\leq n}$ and we minimize the empirical loss

$$\tilde{L}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(X_i,\theta), Y_i)$$

  eventually we add a regularization term.

- This leads to minimize the penalized empirical risk

$$L_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(X_i,\theta), Y_i) + \lambda\Omega(\theta).$$

Penalized empirical risk

- We can consider $\mathbb{L}^2$ regularization.

$$\begin{aligned}\Omega(\theta) &= \sum_k \sum_i \sum_j (W_{i,j}^{(k)})^2 \\ &= \sum_k \|W^{(k)}\|_F^2\end{aligned}$$

  where $\|W\|_F$ denotes the Frobenius norm of the matrix W.
- Note that only the weights are penalized, the biases are not penalized.
- It is easy to compute the gradient of $\Omega(\theta)$:

$$\bigtriangledown_{W^{(k)}} \Omega(\theta) = 2W^{(k)}.$$

- One can also consider $\mathbb{L}^1$ regularization, leading to parcimonious solutions :

$$\Omega(\theta) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|.$$

Penalized empirical risk

- In order to minimize the criterion $L_n(\theta)$, a **stochastic gradient descent algorithm** is used.
- In order to compute the gradient, a clever method, called **Backpropagation algorithm** is considered.
- It has been introduced by Rumelhart et al. (1988), it is still crucial for deep learning.

The stochastic gradient descent algorithm

- Initialization of $\theta = (W^{(1)}, b^{(1)}, \ldots, W^{(L+1)}, b^{(L+1)})$.
- **For** $j = 1, \ldots, N$ iterations :
  - At step j :

$$\theta = \theta - \varepsilon \frac{1}{m} \sum_{i \in B} \left[\bigtriangledown_\theta \ell(f(X_i, \theta), Y_i) + \lambda \bigtriangledown_\theta \Omega(\theta) \right].$$

The stochastic gradient descent algorithm

- In the previous algorithm, we do not compute the gradient for the loss function at each step of the algorithm but only on a subset B of cardinality m (called a **batch**).
- This is what is classically done for big data sets (and for deep learning) or for sequential data.
- B is taken at random without replacement.
- An iteration over all the training examples is called an **epoch**.
- The numbers of epochs to consider is a parameter of the deep learning algorithms.
- The total number of iterations equals the number of epochs times the sample size n divided by m, the size of a batch.
- This procedure is called **batch learning**, sometimes, one also takes batches of size 1, reduced to a single training example (X_i, Y_i).

Backpropagation algorithm for regression

- We consider the regression case and explain how to compute the gradient of the empirical quadratic loss by the Backpropagation algorithm.
- To simplify, we do not consider here the penalization term, that can easily be added.
- Assuming that the output of the multilayer perceptron is of size K, and using the previous notations, the empirical quadratic loss is proportional to

$$\sum_{i=1}^{n} R_i(\theta)$$

with

$$R_i(\theta) = \sum_{k=1}^{K} (Y_{i,k} - f_k(X_i, \theta))^2.$$

Backpropagation algorithm for regression

- In a regression model, the output activation function ψ is generally the identity function, to be more general, we assume that

$$\psi(a_1, \ldots, a_K) = (g_1(a_1), \ldots, g_K(a_K))$$

  where $g_1, \ldots, g_K$ are functions from $\mathbb{R}$ to $\mathbb{R}$.
- Computation of the partial derivatives of R_i with respect to the weights of the output layer.
  - Recalling that

$$a^{(L+1)}(x) = b^{(L+1)} + W^{(L+1)} h^{(L)}(x),$$

    we get

$$\frac{\partial R_i}{\partial W_{k,m}^{(L+1)}} = -2(Y_{i,k} - f_k(X_i, \theta)) g_k'(a_k^{(L+1)}(X_i)) h_m^{(L)}(X_i).$$

Backpropagation algorithm for regression

- Computation of the partial derivatives of R_i with respect to the weights of the previous layer.
  - Differentiating now with respect to the weights of the previous layer

$$\frac{\partial R_i}{\partial W_{m,l}^{(L)}} = -2\sum_{k=1}^{K}(Y_{i,k} - f_k(X_i,\theta))g_k'(a_k^{(L+1)}(X_i))\frac{\partial a_k^{(L+1)}(X_i)}{\partial W_{m,l}^{(L)}}.$$

with

$$\begin{aligned} a_k^{(L+1)}(x) &= \sum_j W_{k,j}^{(L+1)} h_j^{(L)}(x), \\ h_j^{(L)}(x) &= \phi\left(b_j^{(L)} + \langle W_j^{(L)}, h^{(L-1)}(x)\rangle\right). \end{aligned}$$

  - This leads to

$$\frac{\partial a_k^{(L+1)}(x)}{\partial W_{m,l}^{(L)}} = W_{k,m}^{(L+1)}\phi'\left(b_m^{(L)} + \langle W_m^{(L)}, h^{(L-1)}(x)\rangle\right) h_l^{(L-1)}(x).$$

Backpropagation algorithm for regression

- Let us introduce the notations

$$\begin{aligned}\delta_{k,i} &= -2(Y_{i,k} - f_k(X_i,\theta))g_k'(a_k^{(L+1)}(X_i)) \\ s_{m,i} &= \phi'\left(a_m^{(L)}(X_i)\right)\sum_{k=1}^{K} W_{k,m}^{(L+1)}\delta_{k,i}.\end{aligned}$$

- Then we have

$$\frac{\partial R_i}{\partial W_{k,m}^{(L+1)}} = \delta_{k,i}h_m^{(L)}(X_i) \quad (1)$$

$$\frac{\partial R_i}{\partial W_{m,l}^{(L)}} = s_{m,i}h_l^{(L-1)}(X_i), \quad (2)$$

known as the **backpropagation equations**.

Backpropagation algorithm for regression

- The values of the gradient are used to update the parameters in the gradient descent algorithm.
- At step $r+1$, we have :

$$W_{k,m}^{(L+1,r+1)} = W_{k,m}^{(L+1,r)} - \varepsilon_r \sum_{i \in B} \frac{\partial R_i}{\partial W_{k,m}^{(L+1,r)}}$$

$$W_{m,l}^{(L,r+1)} = W_{m,l}^{(L,r)} - \varepsilon_r \sum_{i \in B} \frac{\partial R_i}{\partial W_{m,l}^{(L,r)}}$$

where B is a batch and $\varepsilon_r > 0$ is the learning rate that satisfies $\varepsilon_r \to 0$, $\sum_r \varepsilon_r = \infty$, $\sum_r \varepsilon_r^2 < \infty$, for example $\varepsilon_r = 1/r$.

Backpropagation algorithm for regression

- We use the Backpropagation equations to compute the gradient by a two pass algorithm.
- In the *forward pass*, we fix the value of the current weights $\theta^{(r)} = (W^{(1,r)}, b^{(1,r)}, \ldots, W^{(L+1,r)}, b^{(L+1,r)})$, and we compute the predicted values $f(X_i, \theta^{(r)})$ and all the intermediate values $(a^{(k)}(X_i), h^{(k)}(X_i) = \phi(a^{(k)}(X_i)))_{1 \leq k \leq L+1}$ that are stored.
- Using these values, we compute during the *backward pass* the quantities $\delta_{k,i}$ and $s_{m,i}$ and the partial derivatives given in Equations **??** and **??**.
- We have computed the partial derivatives of R_i only with respect to the weights of the output layer and the previous ones, but we can go on to compute the partial derivatives of R_i with respect to the weights of the previous hidden layers.
- In the back propagation algorithm, each hidden layer gives and receives informations from the neurons it is connected with.
- Hence, the algorithm is adapted for parallel computations.

Initialization

- The input data have to be normalized to have approximately the same range.
- The biases can be initialized to 0. The weights cannot be initialized to 0 since for the tanh activation function, the derivative at 0 is 0, this is a saddle point.
- They also cannot be initialized with the same values, otherwise, all the neurons of a hidden layer would have the same behaviour.
- We generally initialize the weights at random : the values $W_{i,j}^{(k)}$ are i.i.d. Uniform on $[-c, c]$ with possibly $c = \frac{\sqrt{6}}{N_k + N_{k-1}}$ where N_k is the size of the hidden layer k. We also sometimes initialize the weights with a normal distribution $\mathcal{N}(0, 0.01)$ (see Gloriot and Bengio, 2010).

Optimization algorithms

- Many algorithms can be used to minimize the loss function with hyperparameters to calibrate.
- The Stochastic Gradient Descent (SGD) algorithm :

$$\theta_i^{new} = \theta_i^{old} - \varepsilon \frac{\partial L}{\partial \theta_i}(\theta_i^{old}),$$

where ε is the *learning rate* , and its calibration is very important for the convergence of the algorithm.

  - If it is too small, the convergence is very slow and the optimization can be blocked on a local minimum.
  - If the learning rate is too large, the network will oscillate around an optimum without stabilizing and converging.
  - A classical way to proceed is to adapt the learning rate during the training : recommended to begin with a "large " value of ϵ, and reduce this value during the successive iterations : The observation of the evolution of the loss function can give indications on the way to proceed.

Optimization algorithms

- Another stopping rule, called *early stopping* is also used : we stop learning when the loss function for a validation sample stops to decrease.
- *Batch learning* is used for computational reasons, backpropagation algorithms need to store intermediate values : for big data sets, such as millions of images, this is not feasible, all the more that the deep networks have millions of parameters to calibrate.
- The batch size m is also a parameter to calibrate. Small batches generally lead to better generalization properties.
- The particular case of batches of size 1 is called *On-line Gradient Descent*. The disadvantage of this procedure is the very long computation time.

SGD algorithm

Summary of Stochastic Gradient Descent algorithm

- Fix the parameters ε : learning rate, m : batch size, nb : number of epochs.
- For $l = 1$ to nb epochs
- For $l = 1$ to n/m,
  - Take a random batch of size m without replacement in the learning sample : $(X_i, Y_i)_{i \in B_l}$
  - Compute the gradients with the backpropagation algorithm

$$\tilde{\nabla}_\theta = \frac{1}{m} \sum_{i \in B_l} \nabla_\theta \ell(f(X_i, \theta), Y_i).$$

  - Update the parameters

$$\theta^{new} = \theta^{old} - \varepsilon \tilde{\nabla}_\theta.$$

SGD algorithm

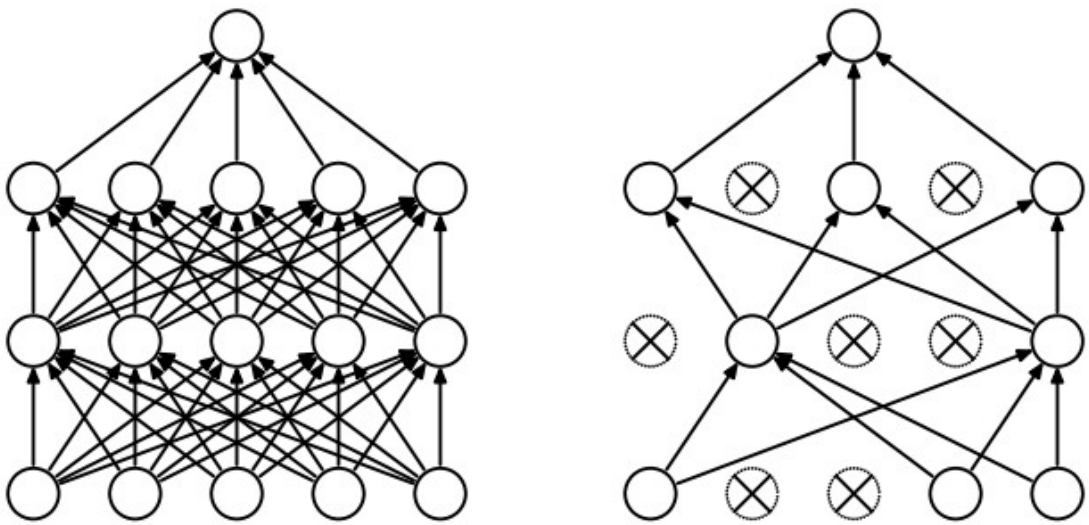
- Since the choice of the learning rate is delicate, variations of the algorithm that are less sensitive to this parameter have been proposed.
- **Nesterov accelerated gradient** : Nesterov (1983) and Sutskever et al. (2013)
- **RMSProp** algorithm, due to Hinton (2012) .

Regularization

- We have already mentioned $\mathbb{L}^2$ or $\mathbb{L}^1$ penalization ; we have also mentioned early stopping.
- For deep learning, the mostly used method is the **dropout** introduced by Hinton et al. (2012).
- With a certain probability p, and independently of the others, each unit of the network is set to 0.
- It is classical to set p to 0.5 for units in the hidden layers, and to 0.2 for the entry layer.
- The computational cost is weak since we just have to set to 0 some weights with probability p.

Dropout

- This method improves significantly the generalization properties of deep neural networks and is now the most popular regularization method in this context.
- The disadvantage is that training is much slower (it needs to increase the number of epochs).



(a) Standard Neural Net (b) After applying dropout.

FIGURE: Dropout

Outline

- Introduction
- Neural Networks
- Convolutional Neural Networks
- Autoencoders and GANs
- Recurrent Neural Networks

Convolutional neural networks

- For some types of data, especially for images, multilayer perceptrons are not well adapted.
- They are defined for vectors. By transforming the images into vectors, we loose spatial informations, such as forms.
- The **convolutional neural networks (CNN)** introduced by LeCun (1998) have revolutionized image processing.
- CNN act directly on matrices, or even on tensors for images with three RGB color chanels.

Convolutional neural networks

- CNN are now widely used for image classification, image segmentation, object recognition, face recognition ..

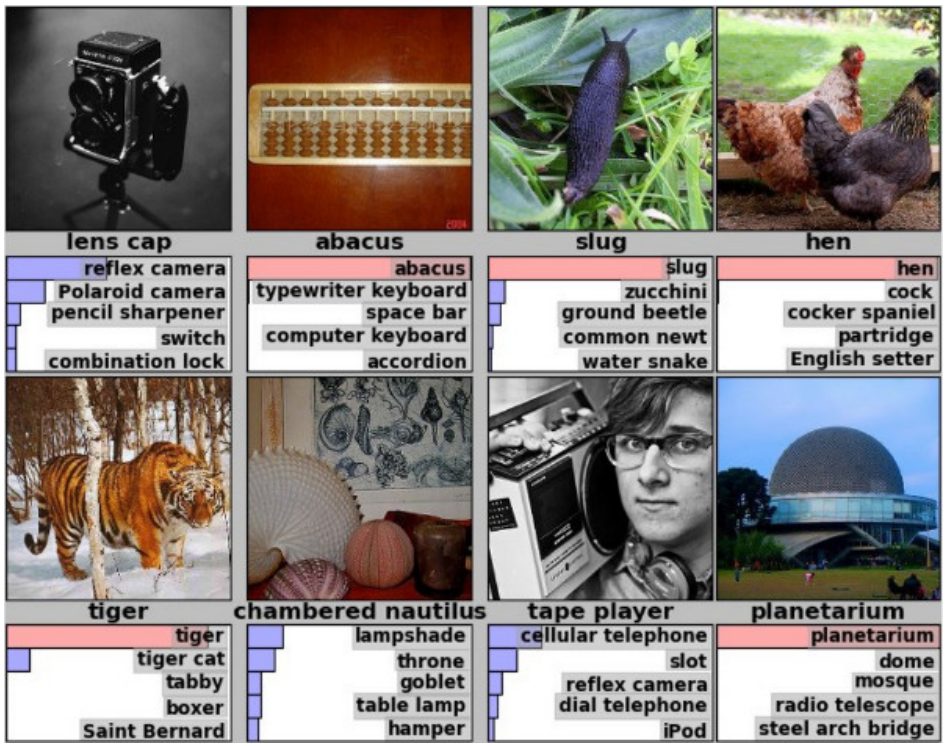


FIGURE: Image annotation

Convolutional neural networks

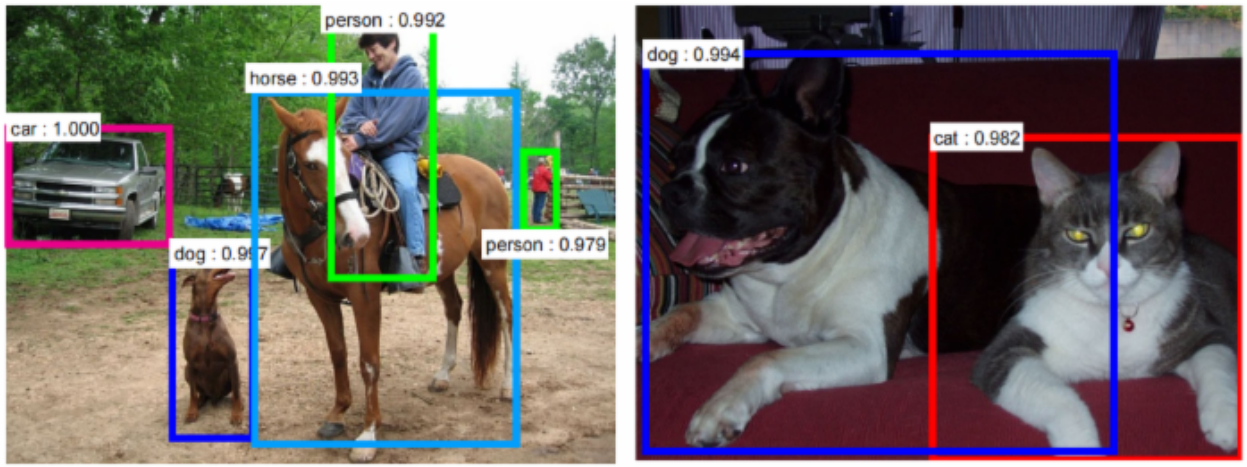


FIGURE: Image Segmentation.

Layers in a CNN

- A Convolutional Neural Network is composed by several kinds of layers, that are described in this section :
  - convolutional layers
  - pooling layers
  - fully connected layers

Convolution layer

- For 2-dimensional signals such as images, we consider the 2D-convolutions : $(K * I)(i,j) = \sum_{m,n} K(m,n) I(i+n, j+m)$.
- K is a convolution kernel applied to a 2D signal (or image) I.

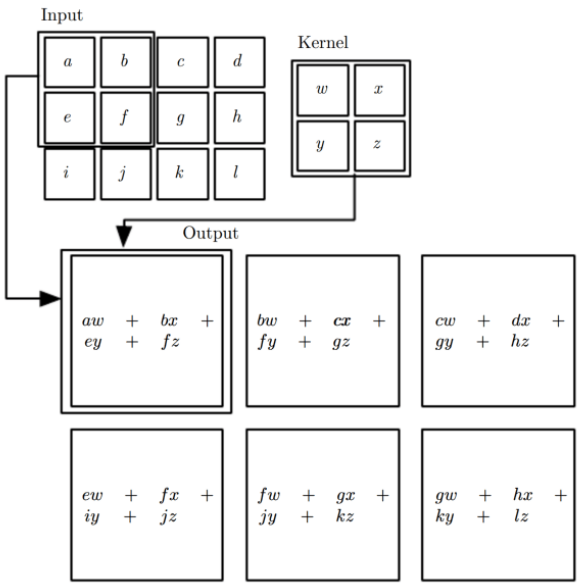


FIGURE: 2D convolution

Convolution layer

- The principle of 2D convolution is to drag a convolution kernel on the image.
- At each position, we get the convolution between the kernel and the part of the image that is currently treated.
- Then, the kernel moves by a number s of pixels, s is called the *stride*.
- When the stride is small, we get redondant information.
- Sometimes, we also add a *zero padding*, which is a margin of size p containing zero values around the image in order to control the size of the output.

Convolution layer

- Assume that we apply C_0 kernels, each of size $k \times k$ on an image.
- If the size of the input image is $W_i \times H_i \times C_i$ (W_i denotes the width, H_i the height, and C_i the number of channels, typically $C_i = 3$), the volume of the output is $W_0 \times H_0 \times C_0$, where C_0 corresponds to the number of kernels that we consider, and

$$W_0 = \frac{W_i - k + 2p}{s} + 1, \quad H_0 = \frac{H_i - k + 2p}{s} + 1.$$

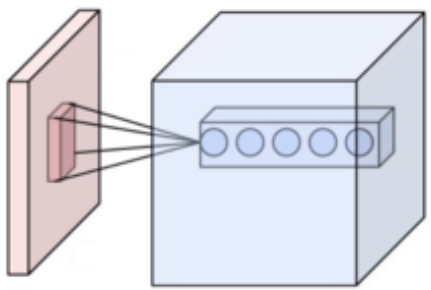
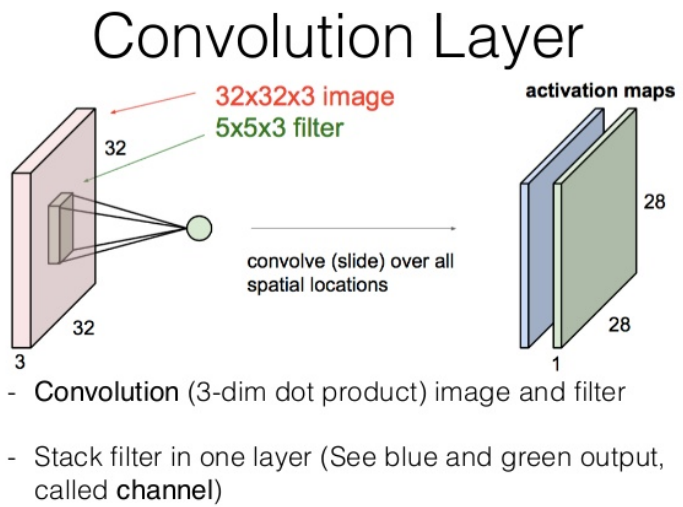


FIGURE: 2D convolution - Units corresponding to the same position but at various depths : each unit applies a different kernel on the same patch of the image.

Convolution layer



- If the image has 3 channels and if K_l $(l = 1, \ldots, C_0)$ denote $5 \times 5 \times 3$ kernels, the convolution with the image I with the kernel K_l corresponds to the formula :

$$K_l * I(i,j) = \sum_{c=0}^{2} \sum_{n=0}^{4} \sum_{m=0}^{4} K_l(n,m,c) I(i+n-2, i+m-2, c).$$

Convolution layer

- For images with C^i channels, the shape of the kernel is (k, k, C^i, C^0) where C^0 is the number of output channels.
- The convolution operations are combined with an activation function ϕ (ReLu in general) : if we consider a kernel K of size $k \times k$, if $\boldsymbol{x}$ is a $k \times k$ patch of the image, the activation is obtained by sliding the $k \times k$ window and computing $z(\boldsymbol{x}) = \phi(K * x + b)$, where b is a bias.
- CNN learn the filters (or kernels) that are the most useful for the task that we have to do (such as classification). Several convolution layers are considered : the output of a convolution becomes the input of the next one.

Pooling layer

- CNN also have *pooling* layers, which allow to reduce the dimension, also referred as *subsampling*, by taking the mean or the maximum on patches of the image (mean-pooling or max-pooling).
- Like the convolutional layers, pooling layers acts on small patches of the image.
- If we consider 2×2 patches, over which we take the maximum value to define the output layer, with a stride 2, we divide by 4 the size of the image.

Pooling layer

- Another advantage of the pooling is that it makes the network less sensitive to small translations of the input images.

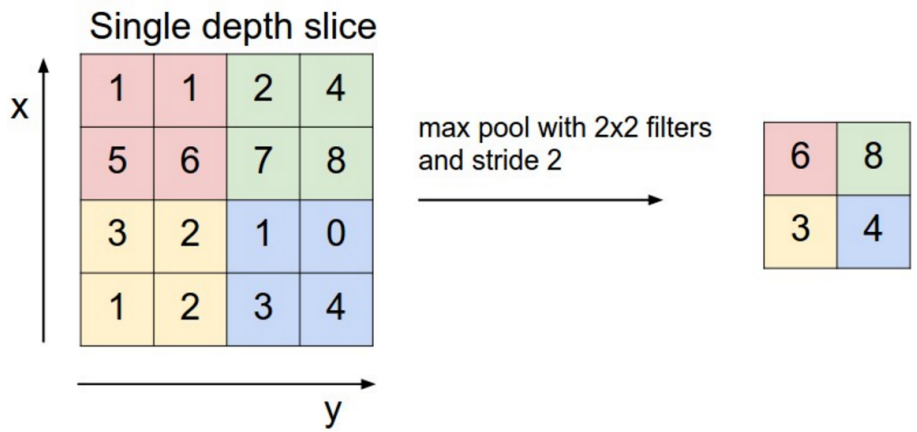


FIGURE: Maxpooling and effect on the dimension

Fully connected layers

- After several convolution and pooling layers, the CNN generally ends with several *fully connected* layers.
- The tensor that we have at the output of these layers is transformed into a vector and then we add several perceptron layers.

Architectures

- We have described the different types of layers composing a CNN.
- We now present how this layers are combined to form the architecture of the network.
- Choosing an architecture is very complex and this is more engineering that an exact science.
- It is therefore important to study the architectures that have proved to be effective and to draw inspiration from these famous examples.
- In the most classical CNN, we chain several times a convolution layer followed by a pooling layer and we add at the end fully connected layers.

Architectures

The **LeNet** network, proposed by the inventor of the CNN, Yann LeCun (1998) is of this type. This network was devoted to digit recognition. It is composed only on few layers and few filters, due to the computer limitations at that time.

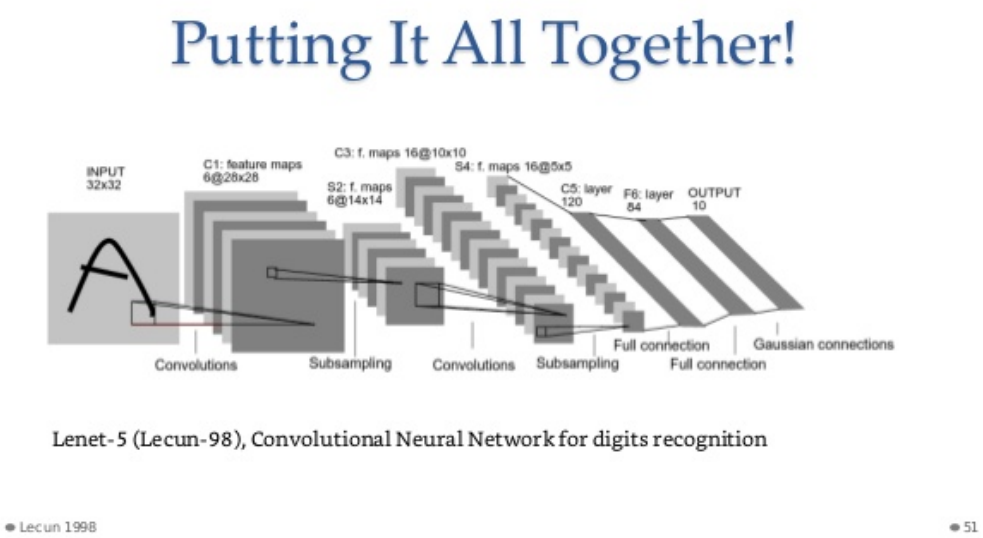


FIGURE: Architecture of the network Le Net. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998)

Architectures

- With the appearance of GPU (Graphical Processor Unit) cards, much more complex architectures for CNN have been proposed, like the network **AlexNet** (Krizhevsky (2012)).
- **AlexNet** won the ImageNet competition devoted to the classification of one million of color images (224×224) onto 1000 classes.
- AlexNet is composed of 5 convolution layers, 3 max-pooling 2×2 layers and fully connected layers.

Architectures

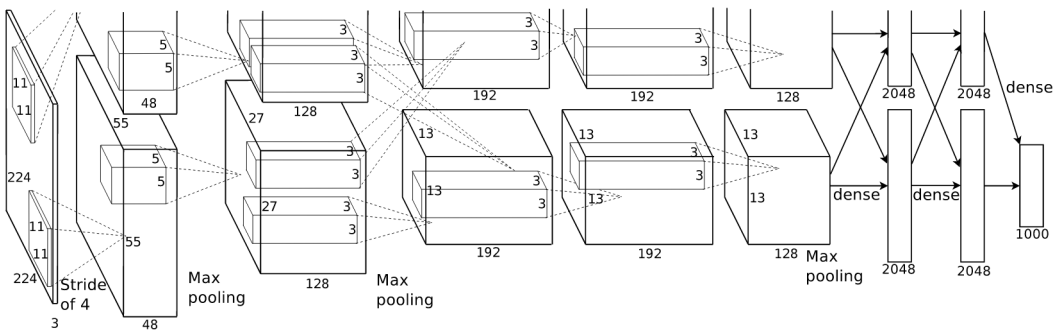


FIGURE: Architecture of the network AlexNet. Krizhevsky, A. et al (2012)

Architectures

Alexnet architecture

Input	227 * 227 * 3			
Conv 1	55*55*96	96	11 *11	filters at stride 4, pad 0
Max Pool 1	27*27*96		3 *3	filters at stride 2
Conv 2	27*27*256	256	5*5	filters at stride 1, pad 2
Max Pool 2	13*13*256		3 *3	filters at stride 2
Conv 3	13*13*384	384	3*3	filters at stride 1, pad 1
Conv 4	13*13*384	384	3*3	filters at stride 1, pad 1
Conv 5	13*13*256	256	3*3	filters at stride 1, pad 1
Max Pool 3	6*6*256		3 *3	filters at stride 2
FC1	4096	4096	neurons	
FC2	4096	4096	neurons	
FC3	1000	1000	neurons	(softmax logits)

Architectures

We present another example.

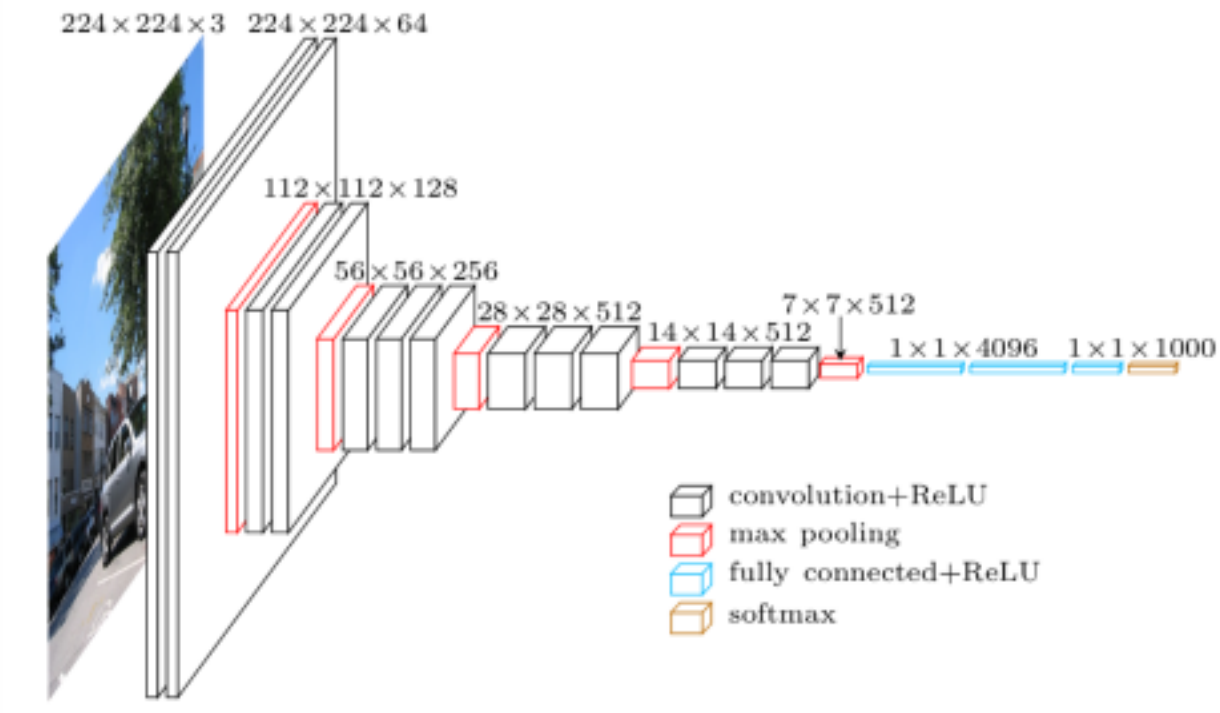


FIGURE: Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition (2014).

Architectures

The network that won the competition in 2014 is the network **GoogLeNet**(Szegedy et al. (2016)), which is a new kind of CNN, not only composed on successive convolution and pooling layers, but also on new modules called *Inception*, which are some kind of network in the network.

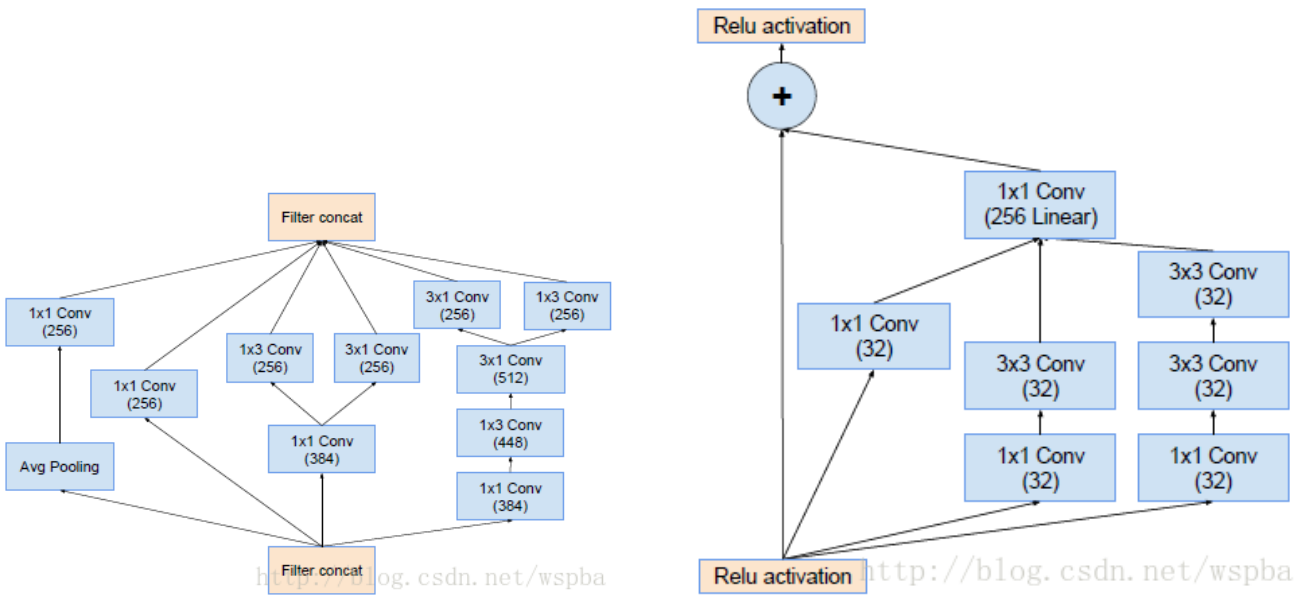


FIGURE: Inception modules, Szegedy et al. (2016)

Architectures

- The most recent innovations concern the **ResNet** networks (see Szegedy 2016).
- The originality of the ResNets is to add a connection linking the input of a layer (or a set of layers) with its output.
- In order to reduce the number of parameters, the ResNets do not have fully connected layers.
- GoogleNet and ResNet are much deeper than the previous CNN, but contain much less parameters.
- They are nevertheless much costly in memory than more classical CNN such as VGG or AlexNet.

Architectures

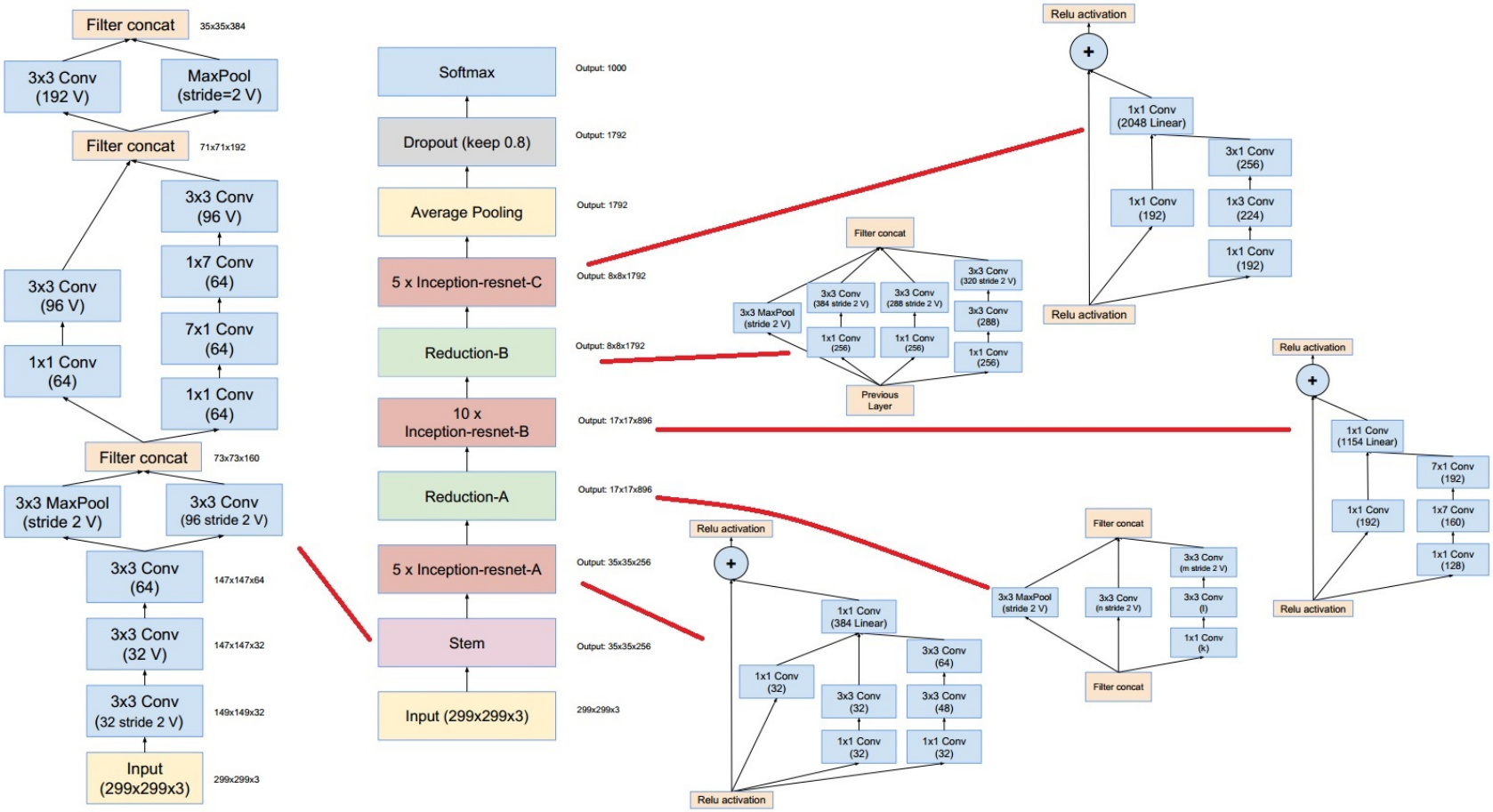


FIGURE: Inception-v4, Inception-resnet (Szegedy, C. et al. , 2016)

Architectures

The Figure shows a comparison of the depth and of the performances of the different networks, on the ImageNet challenge.

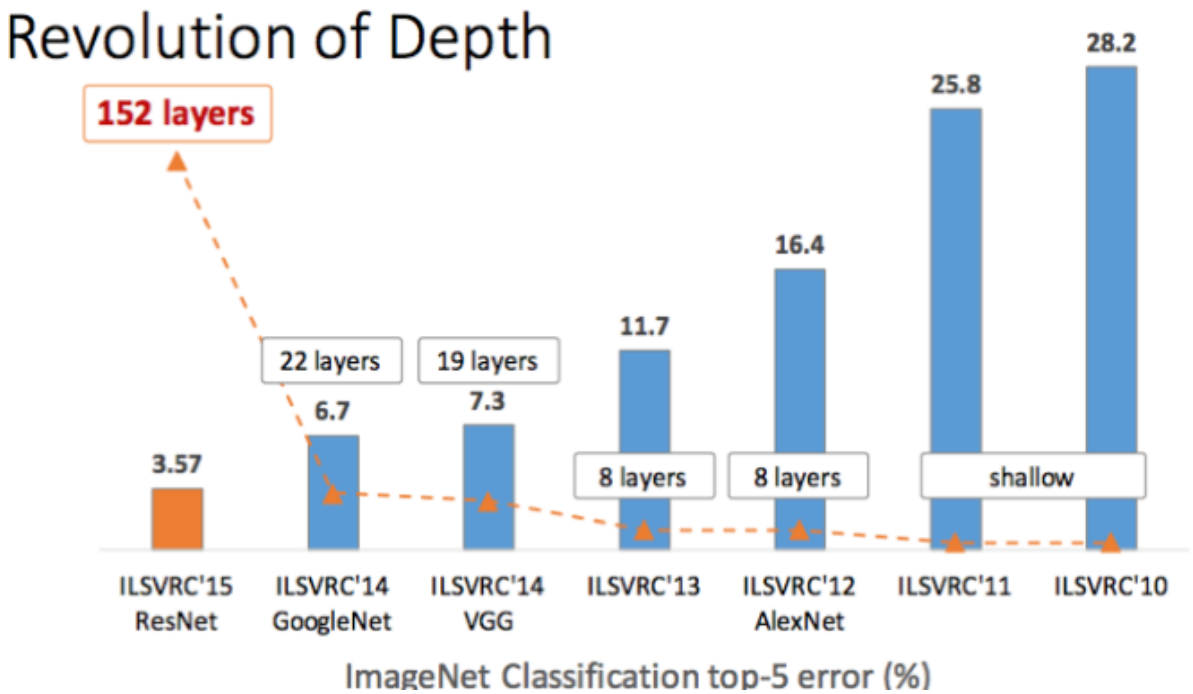


FIGURE: Evolution of the deepth of the CNN and the test error

Outline

- Introduction
- Neural Networks
- Convolutional Neural Networks
- Autoencoders and GANs
- Recurrent Neural Networks

Autoencoders

- An autoencoder is a neural network trained to learn an efficient data coding in an unsupervised way.
- The aim is to learn a sparse representation of the data, typically for the purpose of dimension reduction.
- Find underlying structures in the data, for clustering, visualization, more robust supervised algorithms.
- For complex data such as signals, images, text, there are various hidden latent structures in the data that we try to capture with the autoencoders.

Autoencoders

- An autoencoder learns to compress the data into a short code (**encoder**) and then uncompress that code to reconstruct something which is close to the original data (**decoder**).
- This forces the autoencoder to reduce the dimension, for example by learning how to ignore the noise, and by learning features in the original data.
- The autoencoder has a hidden layer, $\mathbf{h}$ that describes a **code** used to represent the input, and that is forced to provide a smart compression of the data. The autoencoder is hence a feedforward network consisting in two parts :
  - **An encoder function :** $\mathbf{h} = \mathbf{f}(\mathbf{x})$
  - **A decoder function** that provides a reconstruction : $\mathbf{r} = \mathbf{g}(\mathbf{h})$.

Autoencoders

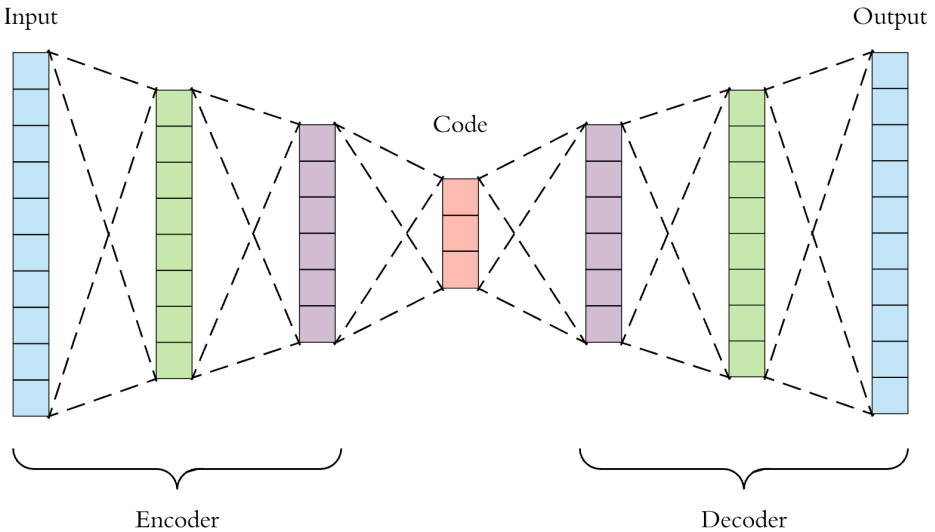


FIGURE: The structure of an autoencoder : mapping the input data to a reconstruction via a sparse internal representation code. Source : *towardsdatascience.com*

Hence, an autoencoder is a neural network trained to "copy" its input into its output, hence to learn the Identity function.
Of course, this would be useless without additional constraints, forcing the **code** to be sparse.

Autoencoders

- Encoders and decoders have arbitrary architectures such as Convolutional Neural Networs, Recurrent Neural Networks ..
- In order to obtain useful features from the autoencoder, the code $\mathbf{h}$ is forced to have a smaller dimension than the input data $\mathbf{x}$.
- As usual, the learning results from the minimization of a loss function $L(\mathbf{x}, g(f(\mathbf{x})))$ (for example the quadratic loss), with a regularization (or additional constraints) to enforce the sparsity of the code.
- When the encoder and decoder are linear and L is the quadratic loss, the autoencoder is equivalent to a PCA ; with nonlinear encoder and decoder functions, such as neural networks, we get nonlinear generalizations of the PCA, which might be more efficient.
- The regularization will lead the autoencoder to provide a sparse representation, with small derivatives, and to be robust to the noise and to missing data.

Regularization for autoencoders

- The first kind of regularization is to induce sparsity.
- For sparse autoencoders, we add to the quadratic loss a sparsity penalty $\Omega(\mathbf{h})$ on the code $\mathbf{h}$, this leads to minimize

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}).$$

- This kind of autoencoders are typically used to learn features for classification purposes.
- This criterions can also be reinterpreted as the likelihood of a generative model with latent variables. Assume that the model has observable variables $\mathbf{x}$ and latent variables $\mathbf{h}$ with joint distribution $p(\mathbf{x}, \mathbf{h}) = p(\mathbf{h})p(\mathbf{x}/\mathbf{h})$. We can consider $p(\mathbf{h})$ as the model's distribution over the latent variables $\mathbf{h}$.

Regularization for autoencoders

- We have
$$\log p(\mathbf{x}, \mathbf{h}) = \log p(\mathbf{h}) + \log p(\mathbf{x}/\mathbf{h}).$$
- The term $\log p(\mathbf{h})$ can induce sparsity. This is the case for example for a Laplace prior :
$$p(\mathbf{h_i}) = \frac{\lambda}{2} e^{-\lambda |h_i|},$$
which leads to
$$-\log p(\mathbf{h}) = \sum_i \left(\lambda |h_i| - \log(\frac{\lambda}{2}) \right) = \Omega(\mathbf{h}) + C,$$
where C is constant (independent of $\mathbf{h}$). In this interpretation, the sparsity can be viewed as a consequence of the model's distribution on the latent variables.

Regularization for autoencoders

- A second kind of regularization is achieved by **denoising autoencoders**.
- Instead of minimizing the loss function $L(\mathbf{x}, g(f(\mathbf{x})))$, a denoising autoencoder minimizes $L(\mathbf{x}, g(f(\tilde{\mathbf{x}})))$, where $\tilde{\mathbf{x}}$ is a copy of $\mathbf{x}$ where some noise has been added.
- Hence the denoising autoencoder has to remove this noise to ressemble to the original object $\mathbf{x}$.
- The procedure can be summarized as follows :
  - Sample a training example x from the training data
  - Sample a noised version $\tilde{x}$ from a given corruption process $C(\tilde{\mathbf{x}}/\mathbf{x} = x)$.
  - Use $(x, \tilde{x})$ as a training example to estimate the encoder reconstruction distribution $p(\mathbf{x}/\tilde{\mathbf{x}})$, by minimizing the loss $L = -\log p(\mathbf{x}/\mathbf{h} = f(\tilde{\mathbf{x}}))$.

Regularization for autoencoders

- A third kind of regularization method consists in a penalization of the derivatives by minimizing the criterion

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}, \mathbf{x}),$$

where

$$\Omega(\mathbf{h}, \mathbf{x}) = \lambda \sum_i \|\nabla_x h_i\|^2.$$

- By this way, we force the encoder $\mathbf{h}$ to be robust to small variations in $\mathbf{x}$.

Regularization for autoencoders

- Autoencoders have many applications : they can be used for classification considering the latent code **h** as input to a classifier instead of **x**.
- The may also be used for semi-supervised learning for simultaneous learning of the latent code on a large unlabeled dataset and then the classifier on a smaller labeled dataset.
- They are also widely used for Generative models.
- They nevertheless have some limitations : autoencoders fail to capture a good representation of the data for complex objects such as images ; the generative models are sometimes of poor quality (blurry images for example).
- New kinds of autoencoders : VQ-VAE (Vector Quantised-Variational AutoEncoders) seem very promising to generate high quality images, videos, and speech (see *Neural Discrete Representation Learning*, A. van den Oord et al, Deep Mind).

Generative Adversarial Networks

- Generative Adversarial Networks (GANs) are generative algorithms that are known to produce state-of-the art samples, especially for image creation ; inpainting, speech and 3D modeling.
- They were introduced by Goodfellow et al (2014) The more recent advances are presented in Goodfellow (2016).
- The aim of GANS is to produce fake observations of a given distribution p^* from which we only have a true sample (for example faces images).
- This data are generally complex and it is impossible to consider a parametric model to capture the complexity of p^* nor to consider nonparametric estimators.
- In order to generate new data from the distribution p^*, GANs trains simultaneously two models :
  - a generative model G that captures the data distribution
  - a discriminative model D that estimates the probability that a sample comes from the training data rather than from the generator G.

Generative Adversarial Networks

- The generator admits as input a random noise (such as Gaussian or Uniform), and eventually latent variables with small dimension, and tries to transform them into fake data that match the distribution p^* of the real data.
- The generator and the discriminator have conflicting goals : simultaneously, the discriminator tries to distinguish the true observations from the fake ones produced by the generator and the generator tries to generated sample that are as undistinguishable as possible from the original data.
- In practice, G and D are defined by multilayers perceptrons learned from the original data via the optimization of a suitable objective function.
- Figure **??** gives a schematic representation of GANs.

Generative Adversarial Networks

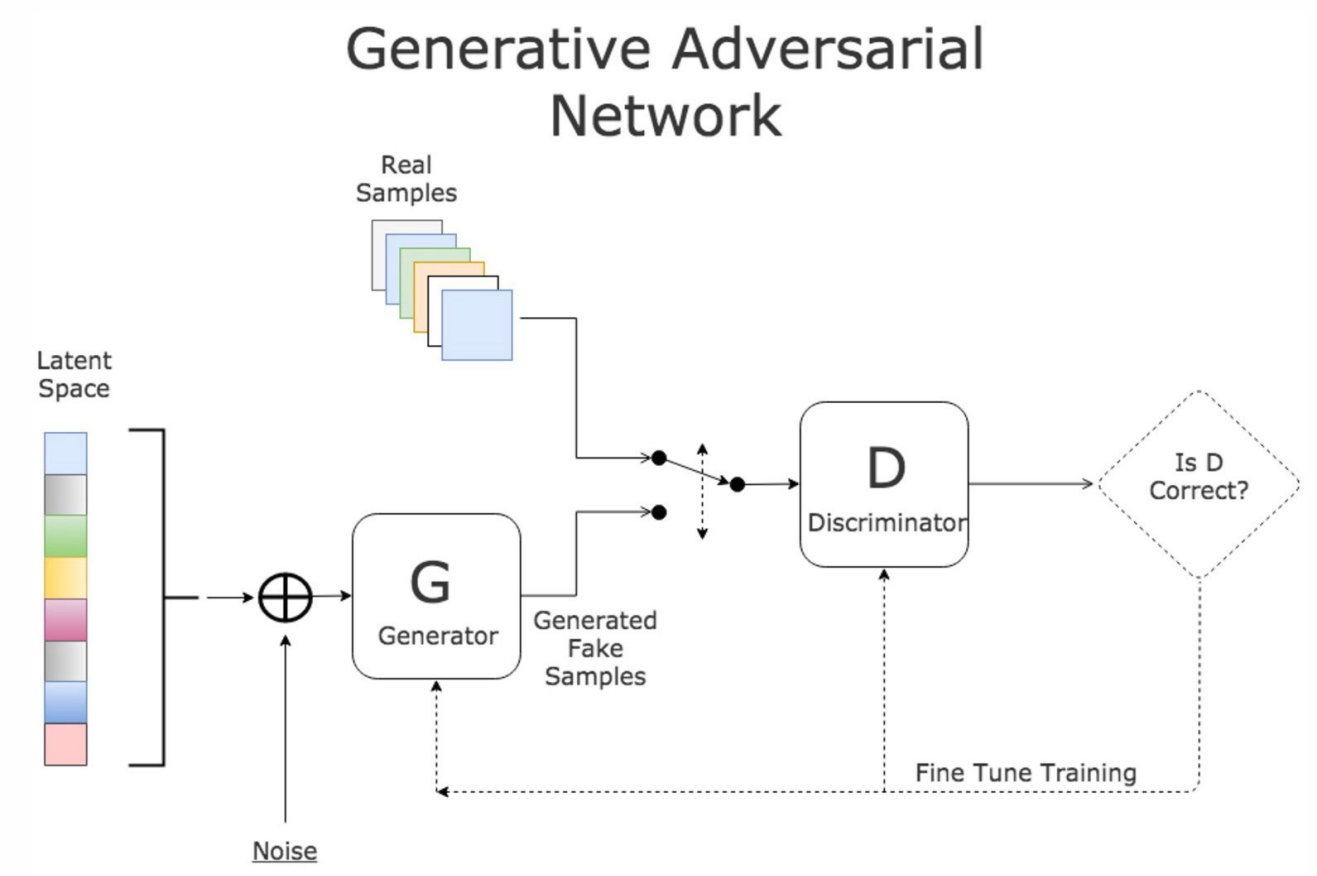


FIGURE: Source : blog.statsbot.co

Generative Adversarial Networks

- In the original paper (Goodfellow, 2014), the input of the generator is only a random noise z from distribution p_z.
- The generator is a multilayer perceptron with parameters θ_g and denoted $G(z, \theta_g)$.
- We also define another multilayer perceptron $D(x, \theta_d)$ with values in $[0, 1]$, that represents the probability that x comes from the real data rather than from the generator.
- D is trained to maximize the probability to assign the correct labels to the real (training) data as well as to the fake ones.
- Hence, $D(x)$ should be close to 1 when x is a real data, and close to 0 otherwise, namely $D(G(z))$ should be close to 0.

Generative Adversarial Networks

- At the same time, G is trained to maximize $D(G(z))$, or equivalently to minimize $\log(1 - D(G(z)))$; indeed $D(G(z))$ should be close to 1 so that $G(z)$ is undistinguishable from the real data.
- In other words, D and G play a two-players minimax game :

$$\min_G \max_D V(D, G)$$

where

$$V(D, G) = \mathbb{E}_{x \sim p^*}(\log(D(x))) + \mathbb{E}_{z \sim p_z}(\log(1 - D(G(z))).$$

- For a given generator, the optimal solution for the discriminator that realizes $\max_D V(D, G)$ is

$$D^*(x) = \frac{p^*(x)}{p^*(x) + p_g(x)},$$

where p_g is the distribution of $G(z)$.

Generative Adversarial Networks

- Concerning the minimax game, in the space of arbitrary functions G and D, it can be shown that the optimal solution is that $G(z)$ recovers exactly the training data distribution ($p_g = p^*$) and D equals $1/2$ everywhere.
- In practice, D and G are trained from the original data X_i and the simulated random noise Z_i to realize

$$\min_G \max_D \left[\sum_{i=1}^{n} \log(D(X_i)) + \sum_{i=1}^{n} \log(1 - D(G(Z_i))) \right].$$

Goodfellow et al (2014) propose the following algorithm :

Algorithm : Minibatch SGD training of GANs.

for / training iterations **do**

for k steps **do**

- Sample a minibatch of m noise samples $z^{(1)}, \ldots, z^{(m)}$ from noise prior p_z
- Sample a minibatch of m examples $x^{(1)}, \ldots, x^{(m)}$ from the original data with distribution p^*
- Update the discriminator by ascending its stochastic gradient :

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log(D(x^{(i)})) + \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))) \right].$$

end for

- Sample a minibatch of m noise samples $z^{(1)}, \ldots, z^{(m)}$ from noise prior p_z
- Update the generator by descending its stochastic gradient :

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))).$$

end for

Generative Adversarial Networks

The generator never sees the training data, it is just updated via the gradients coming from the discriminator.
The next Figure presents a one-dimensional example to understand the successive steps in GANs training .

Generative Adversarial Networks

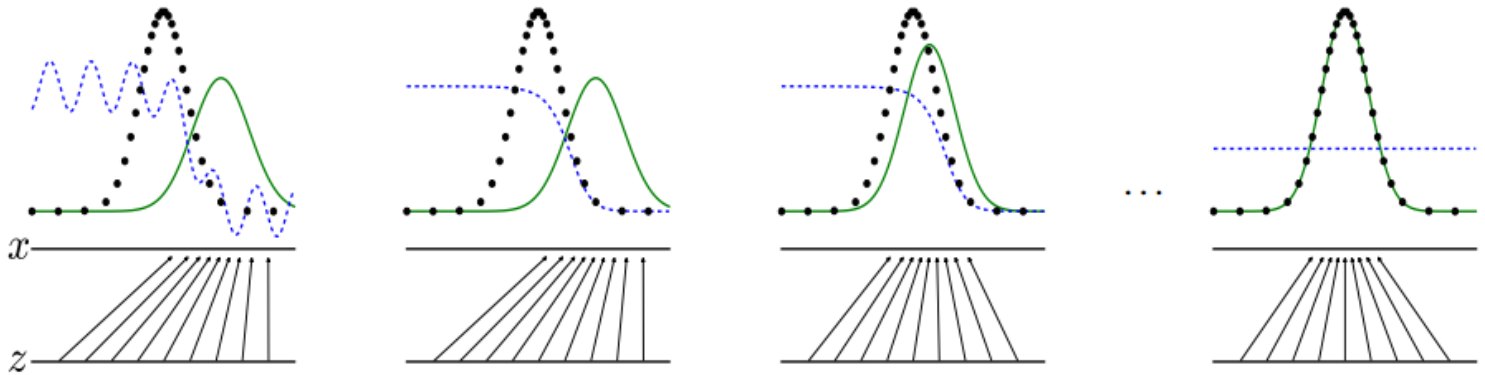


FIGURE: Successive steps in GANs training- In black, dotted line : the data generating distribution ; in green solid line : the generator distribution ; in blue, dashed line : the discriminator - z is sampled uniformly over the domain described by the lower lines- Source : cs.stanford.edu

- on the left, the initial values of the data distribution, the generator distribution and the discriminator.
- Then the discriminator has been trained , converging to $D^*(x)$.
- In the third picture, the generator has been updated, the gradient of D has guided $G(z)$ to high probability regions for the original data.
- On the right : after several training steps, p_g is very close to p^* and the discriminator does no more distinguish the true observations and those coming from the generator.

Generative Adversarial Networks

- Several extension of the original GANs have been proposed such as InfoGANs (Chen, 2016).
- The idea is to take as inputs of the generator not only a random noise but also interpretable latent variables learned from the data.
- This is done by maximizing the mutual information between a small subset of the GANs noise variables and the observations.
- It was shown that the procedure automatically discovers meaningful hidden representations in several image datasets.
- In order to discover the latent factors in an unsupervised way, the algorithm imposes a high mutual information between the generator $G(z, c)$ and the latent variables c.

Generative Adversarial Networks

- Let us recall that, in information theory, the mutual information between two random variables X and Y is a measure of the mutual dependence between the two variables.
- It represents the amount of information obtained about one variable through the other one. The mathematical formulation is

$$I(X,Y) = H(X) - H(X/Y) = H(Y) - H(Y/X)$$

where $H(X) = \int p_X \log(p_X)$ and $H(X/Y) = \int p_{X,Y} \log(p_{X,Y}/p_Y)$. If X and Y are independent, $I(X,Y) = 0$. In InfoGans, the original minimax game is replaced by

$$\min_G \max_D V_I(D,G)$$

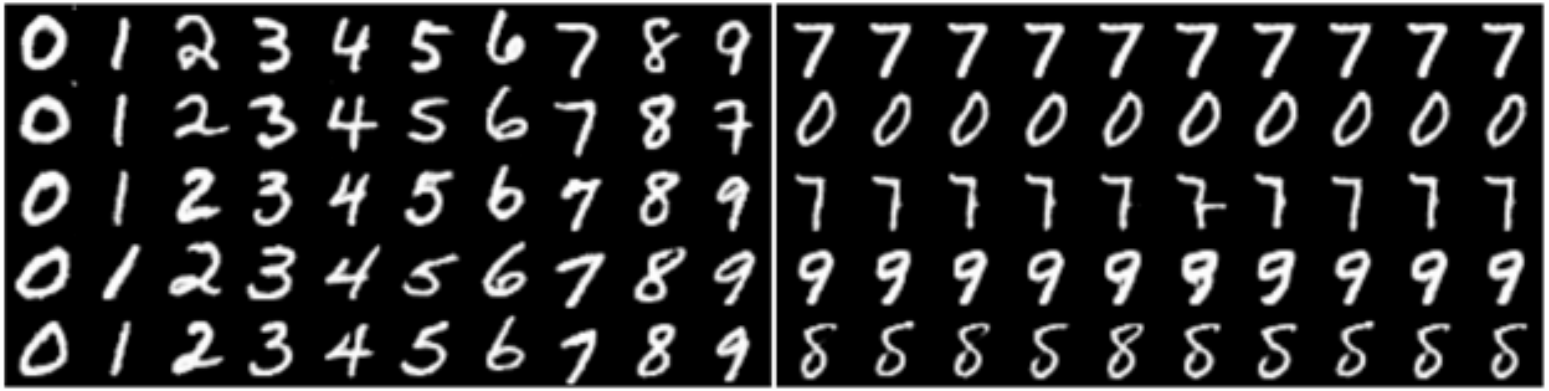
where

$$V_I(D,G) = V(D,G) - \lambda I(c, G(z,c)).$$

Generative Adversarial Networks

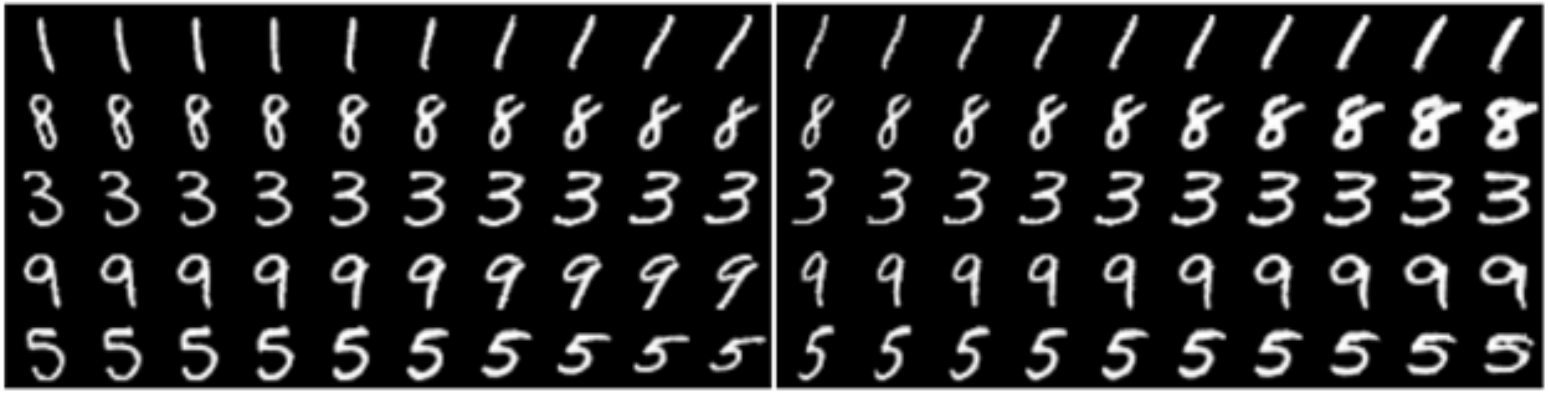
- Chen et al (2016) present an application to the MNIST data where they introduce a tree dimensional latent variable c :
  - the first component c_1 is a discrete random variable with 10 categories with equiprobability,
  - c_2, c_3 are two independent variables with uniform distribution on $[-1, 1]$.
- It appears that the latent variables are easily interpretable : the variable c_1 captures mostly the digit code, c_2 models rotation on digits and c_3 captures the width as shown in Figure **??**.

Generative Adversarial Networks



(a) Varying c_1 on InfoGAN (Digit type)

(b) Varying c_1 on regular GAN (No clear meaning)



(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation)

(d) Varying c_3 from -2 to 2 on InfoGAN (Width)

FIGURE: InfoGAN- The different rows correspond to different random generated samples $G(z, c)$ with fixed latent variables (except one) and fixed random variables z. Ref. Chen et al. (2016)

Outline

- Introduction
- Neural Networks
- Convolutional Neural Networks
- Autoencoders and GANs
- Recurrent Neural Networks

References

The main references for the course on Neural networks are :

- Ian Goodfellow, Yoshua Bengio and Aaron Courville *Deep learning*
- Bishop (1995) *Neural networks for pattern recognition*, Oxford University Press.
- Hugo Larochelle (Sherbrooke) :
  http ://www.dmi.usherb.ca/ larocheh/
- Charles Ollion et Olivier Grisel *Deep learning course*
  https ://github.com/m2dsupsdlclass/lectures-labs