Institut de Mathématiques de Toulouse, INSA Toulouse

Neural networks, Introduction to deep learning

ML Training- CERFACS
October, 2021

Béatrice Laurent

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

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

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

Several activation functions can be considered.

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

The following Figure represents the activation function described above.

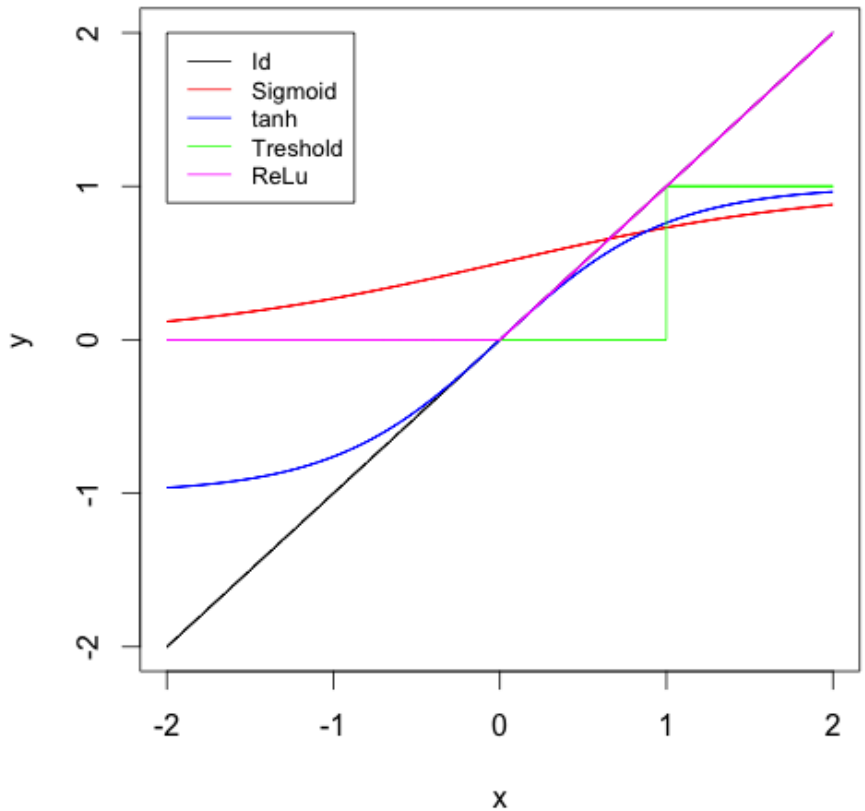
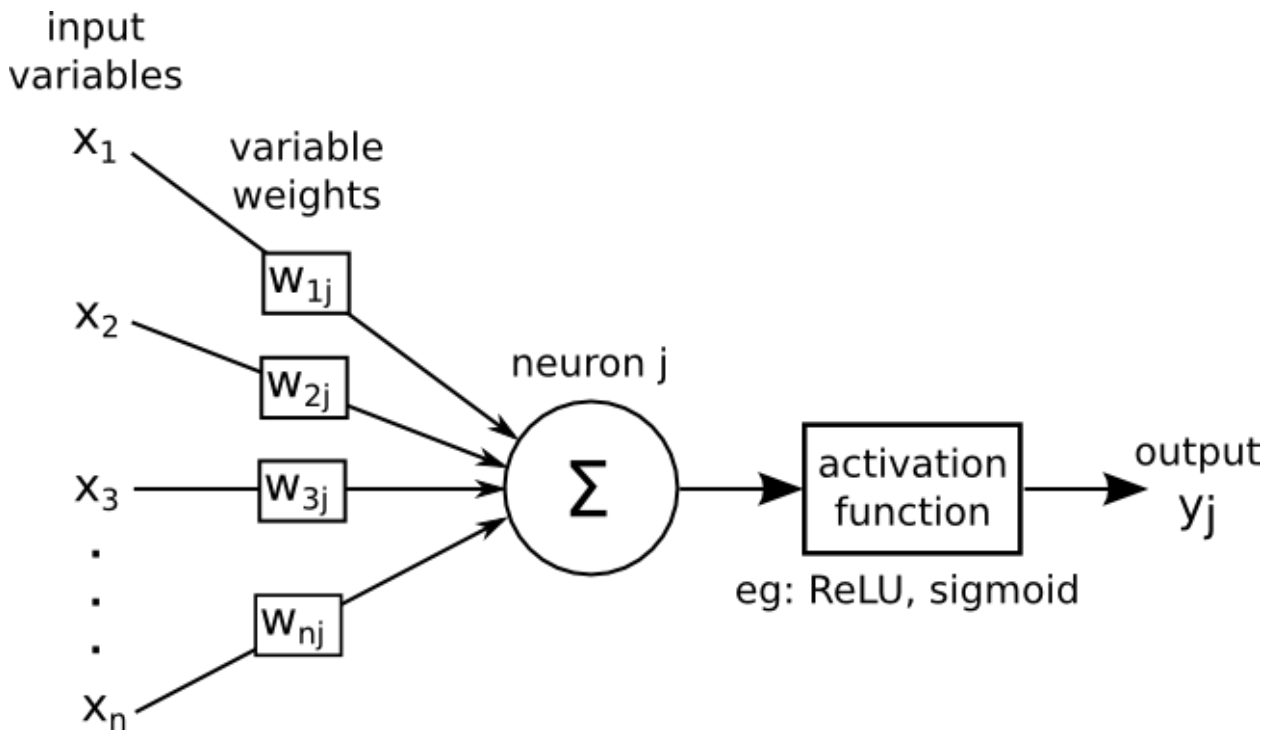


FIGURE – Activation functions

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.

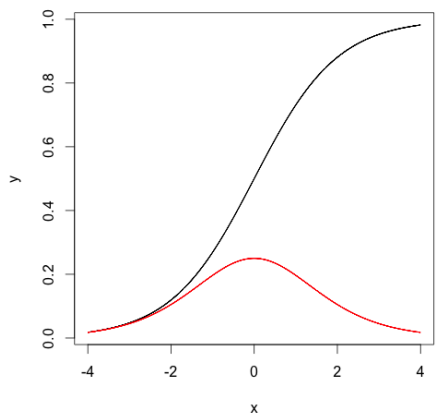


FIGURE – Sigmoid function (in black) and its derivatives (in red)

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 piecewise linear and has many of the properties that make linear model easy to optimize with gradient-based methods.

Outline

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

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.

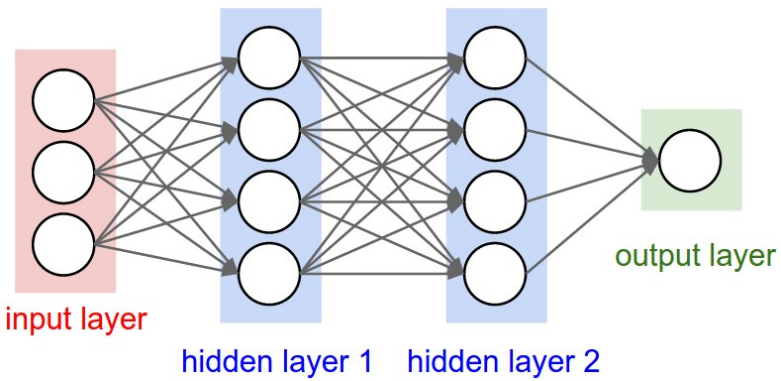


FIGURE – A basic neural network.

Multilayer perceptron

- The output of a neuron can also be the input of a neuron of the same layer or of neuron of previous layers : this is the case for recurrent neural networks.

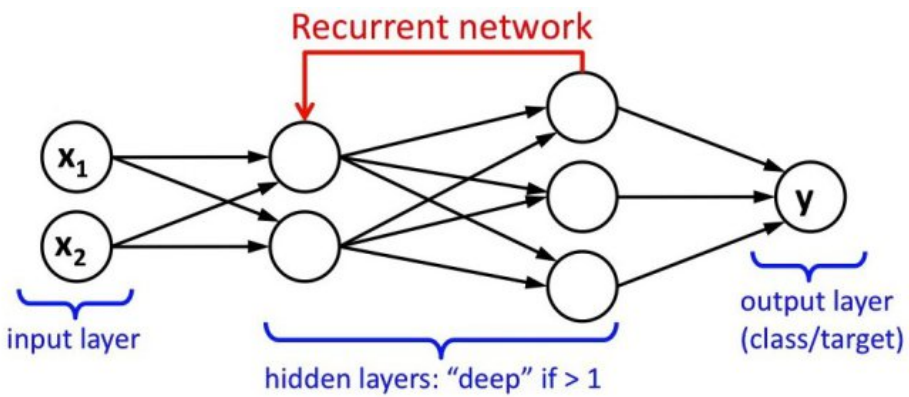


FIGURE – A recurrent neural network

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 (the identity 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 $k - 1$.

Outline

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

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 $(X_i, Y_i)_{1 \leq i \leq n}$.
- As usual, the estimation is obtained by minimizing a loss function, generally with a gradient descent algorithm.
- We first have to choose the loss function between the output of the model $f(x, \theta)$ for an entry x and the target y.

Loss functions

- If the regression model, we generally consider the loss function associated to the $\mathbb{L}_2$ norm :

$$\ell(f(x,\theta),y) = \|y - f(x,\theta)\|^2.$$

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

$$\begin{aligned} \ell(f(x,\theta),y) &= -[y\log(f(x,\theta)) + (1-y)\log(1-f(x,\theta))] \\ &= -\log(P_\theta(Y = y/X = x)) \end{aligned}$$

Loss functions

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

$$\ell(f(x,\theta),y) = -\left[\sum_{j=1}^{k} \mathbb{1}_{y=j} \log P_\theta(Y = j/X = x)\right].$$

- Ideally we would like to minimize the classification error, but it is not smooth, this is why we consider the cross-entropy.

Penalized empirical risk

- Denoting θ the vector of parameters to estimate, we consider the expected loss function

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

  associated to the 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 $\Omega(\theta)$:

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

- One can also consider $\mathbb{L}^1$ regularization (LASSO penalty), 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 often used.
- In order to compute the gradient, a clever method, called **Backpropagation algorithm** is considered.

Outline

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

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

$$\sum_{i=1}^{n} R_i(\theta) = \sum_{i=1}^{n} \|Y_i - f(X_i, \theta)\|^2$$

with

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

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 $k - 1$.

Backpropagation algorithm for regression

- Let us introduce the notations

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

- 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

- 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 1 and 2.
- 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.

Outline

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

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 :

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

where B is a subset of $\{1, \ldots, n\}$ with cardinality m.

The stochastic gradient descent algorithm

- In the previous algorithm, we do not compute the gradient for the empirical 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).

To apply the SGD algorithm, we need to compute the gradients $\bigtriangledown_\theta \ell(f(X_i, \theta), Y_i)$. For this, we use the Backpropagation algorithms.

Stochastic Gradient Descent algorithm

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

$$\tilde{\nabla}_\theta = \frac{1}{m} \sum_{i \in B} \nabla_\theta \ell(f(X_i, \theta), Y_i) + \lambda \nabla_\theta \Omega(\theta),$$

    where B is a batch with cardinality m.
  - Update the parameters

$$\theta^{(r+1)} = \theta^{(r)} - \varepsilon_r \tilde{\nabla}_\theta,$$

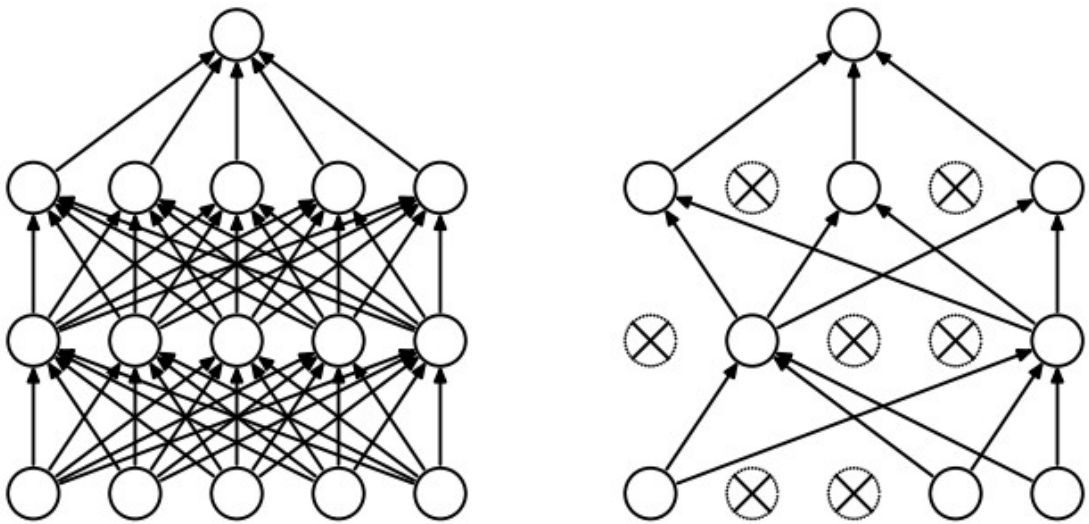
    where $\varepsilon_r > 0$ is the learning rate. that satisfies $\varepsilon_r \to 0$, $\sum_r \varepsilon_r = \infty$, $\sum_r \varepsilon_r^2 < \infty$, for example $\varepsilon_r = 1/r$.

Regularization

- We have already mentioned $\mathbb{L}^2$ or $\mathbb{L}^1$ penalization.
- 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

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

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 ..

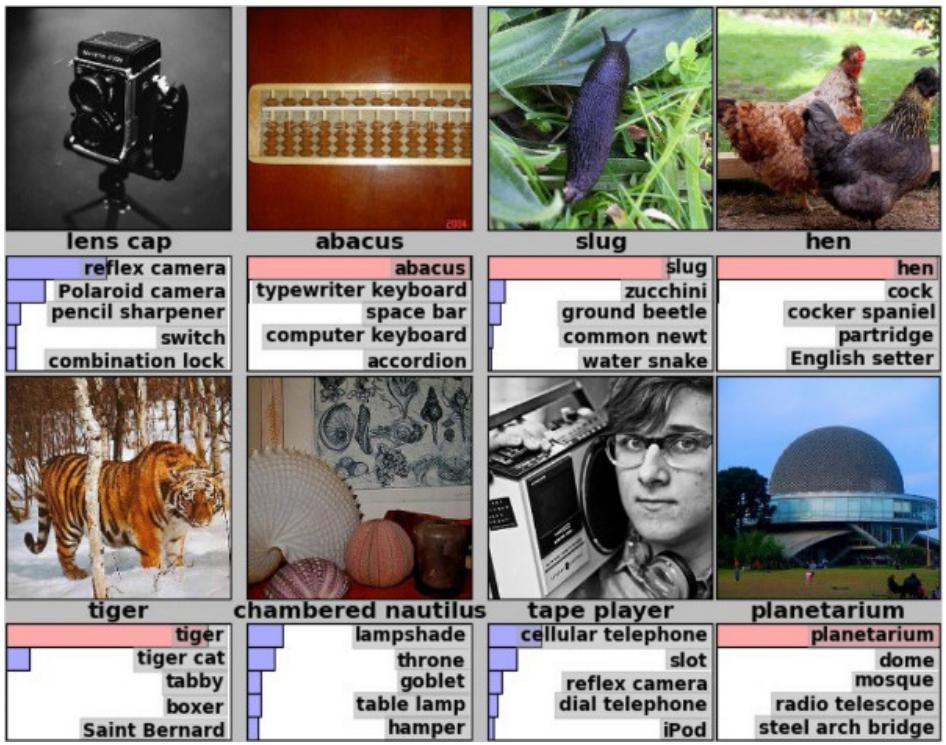


FIGURE – Image annotation

Convolutional neural networks

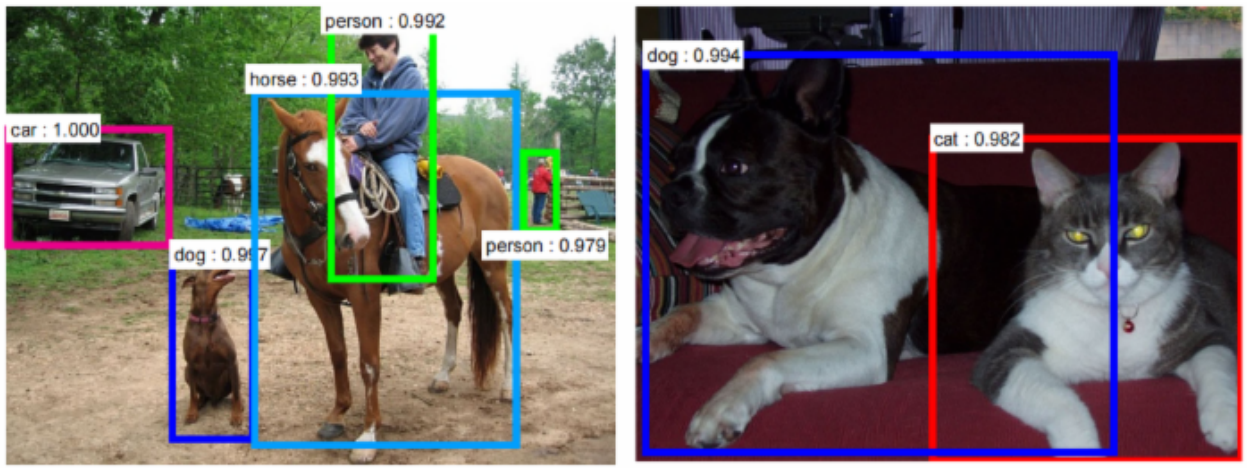
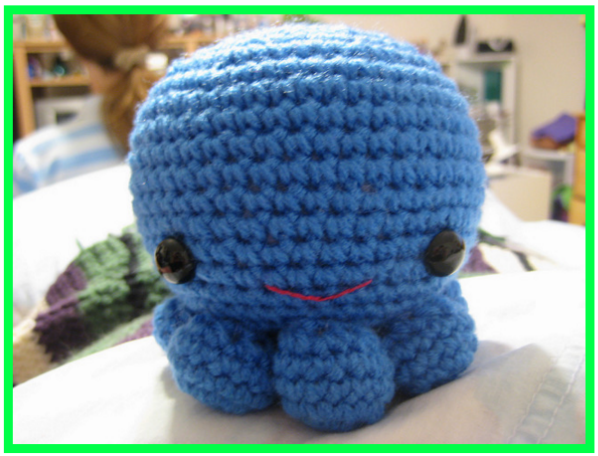


FIGURE – Image Segmentation.

Image Classification

Your specialized huggable recommendation

Go for it! Hug it out. With a score of **0.695779**, we're somewhat sure.



Your specialized huggable recommendation

Don't hug that. Please. With a score of **0.999875**, we're pretty sure.

Your specialized huggable recommendation

Don't hug that. Please. With a score of **0.778686**, we're pretty sure.

Your specialized huggable recommendation

Go for it! Hug it out. With a score of **0.958104**, we're pretty sure.

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.

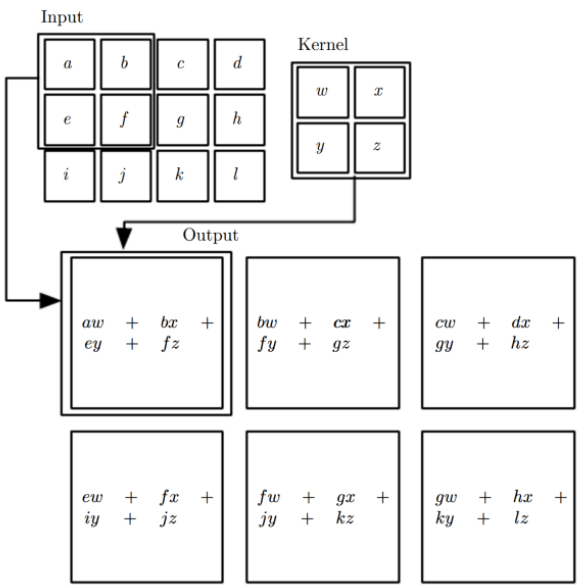


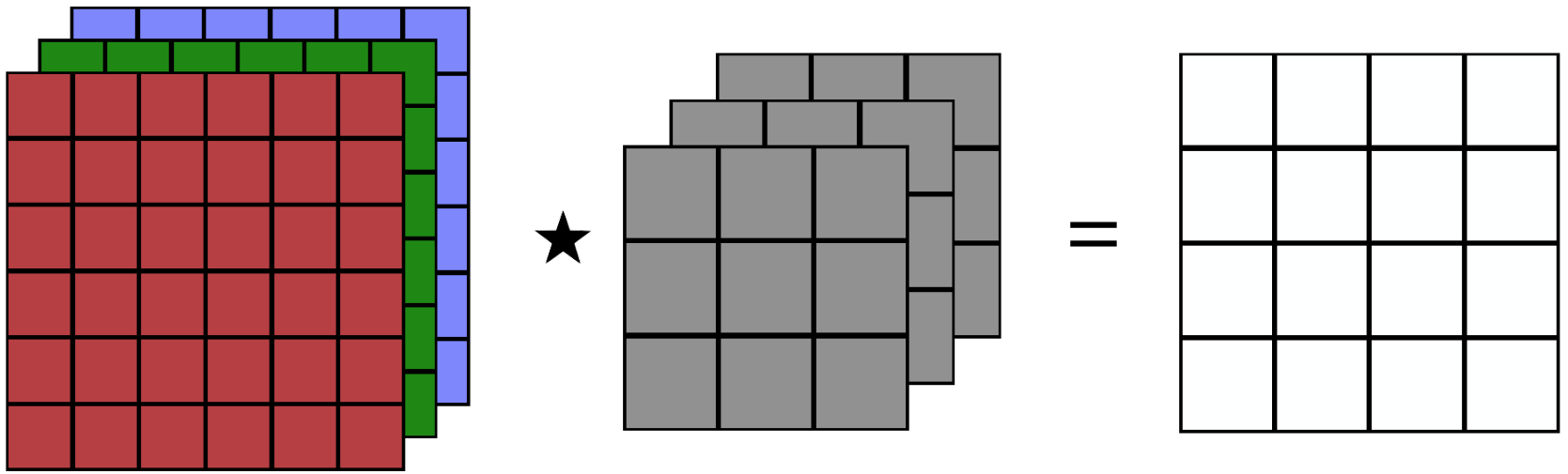
FIGURE – 2D convolution

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. For example, to keep the same size between the input and the output, we use the option padding = SAME in Keras.

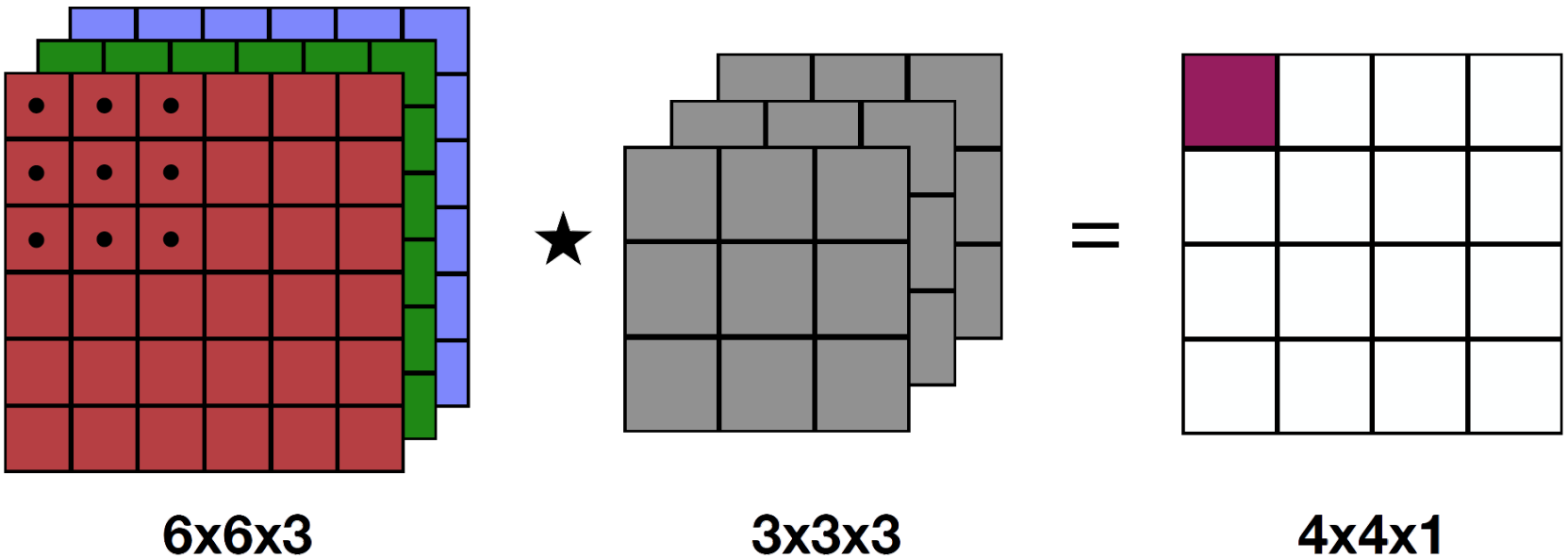
Convolution - channels

Colored image in 3 dimensions : (height, width, channels (RGB))



Convolution - channels

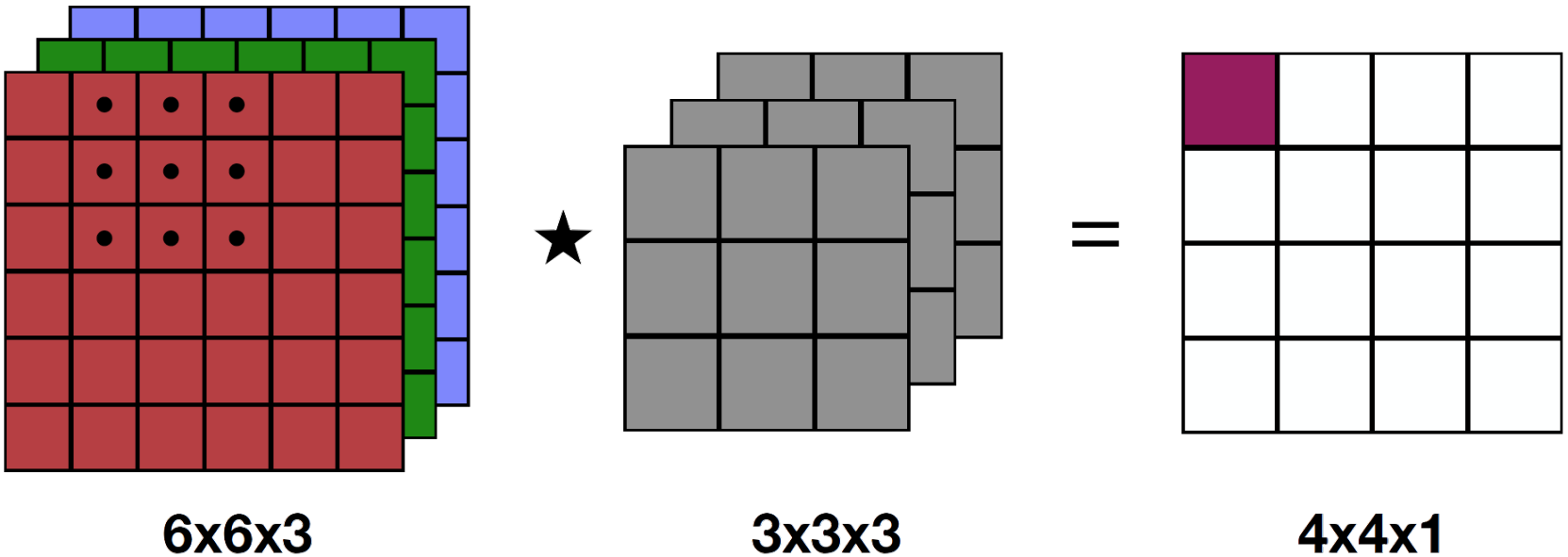
Colored image in 3 dimensions : (height, width, channels (RGB))



$$(F \star img)(x, y) = \sum_c \sum_m \sum_n F^c(n, m) \cdot Img^c(x + m, y + n)$$

Convolution - channels

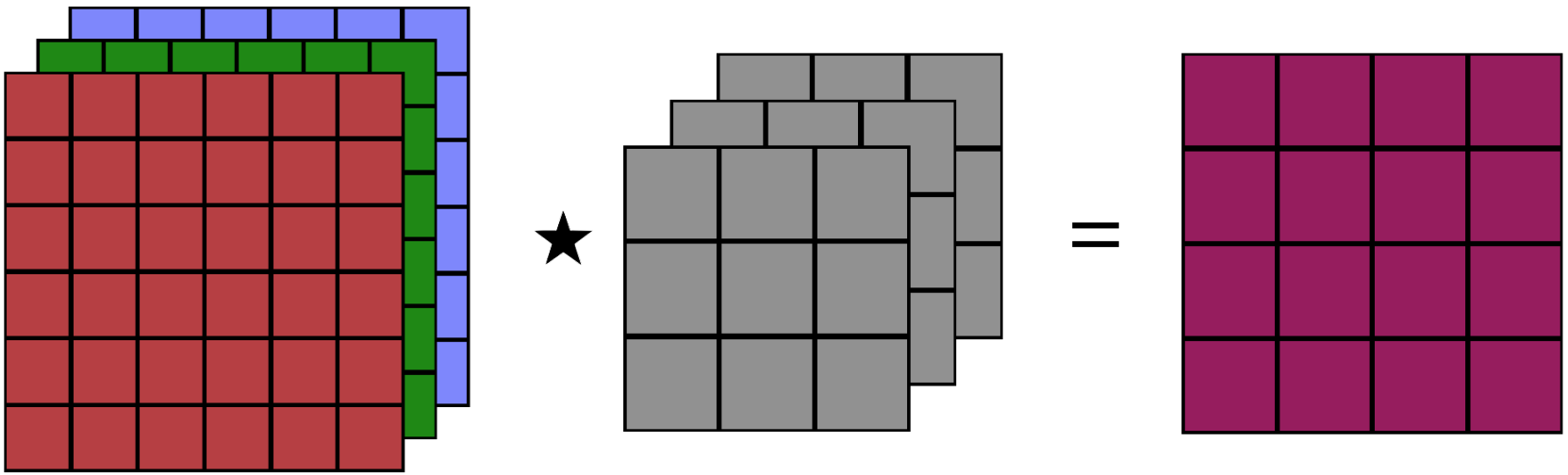
Colored image in 3 dimensions : (height, width, channels (RGB))



$$(F \star img)(x, y) = \sum_c \sum_m \sum_n F^c(n, m) \cdot Img^c(x + m, y + n)$$

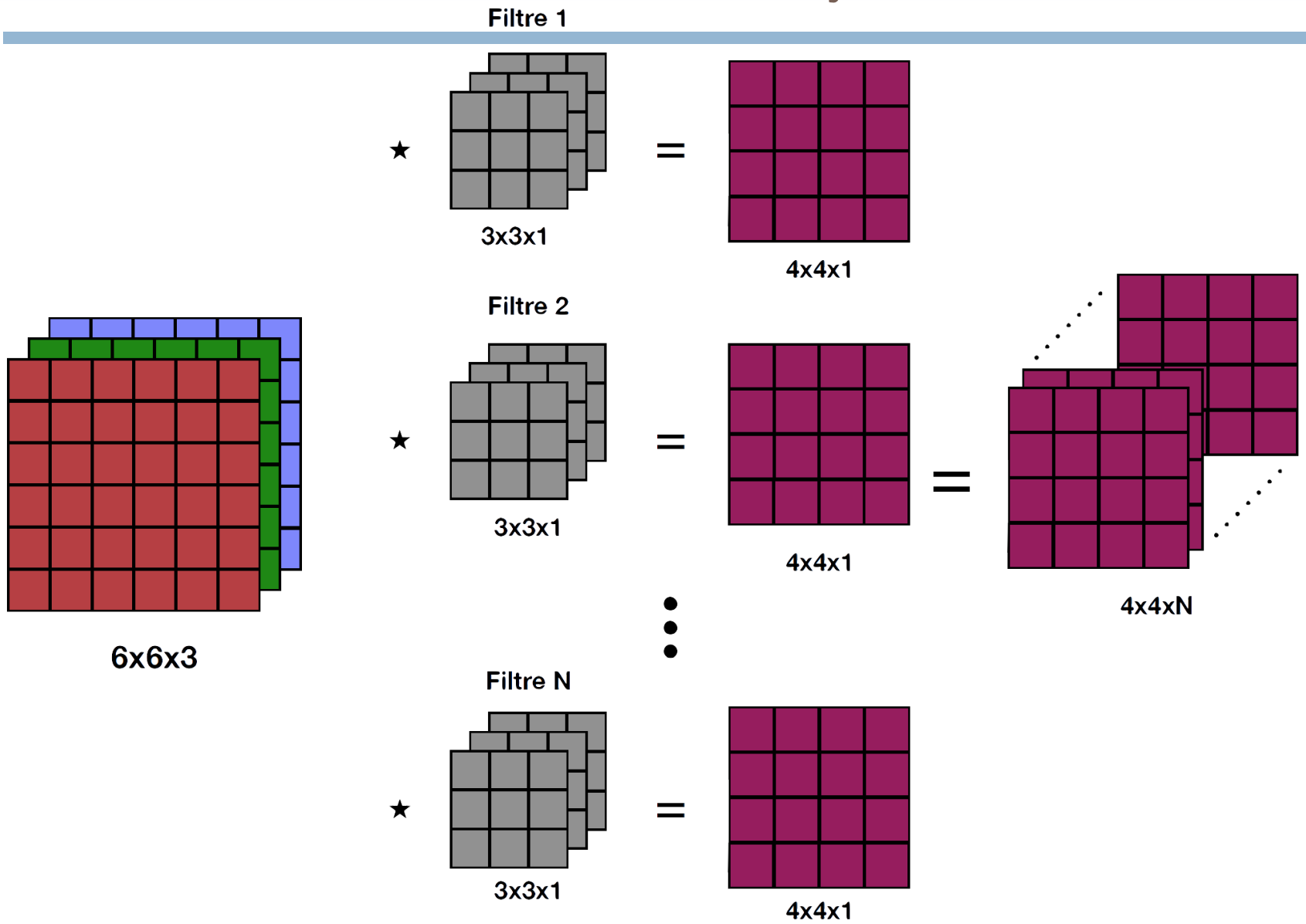
Convolution - channels

Colored image in 3 dimensions : (height, width, channels (RGB))

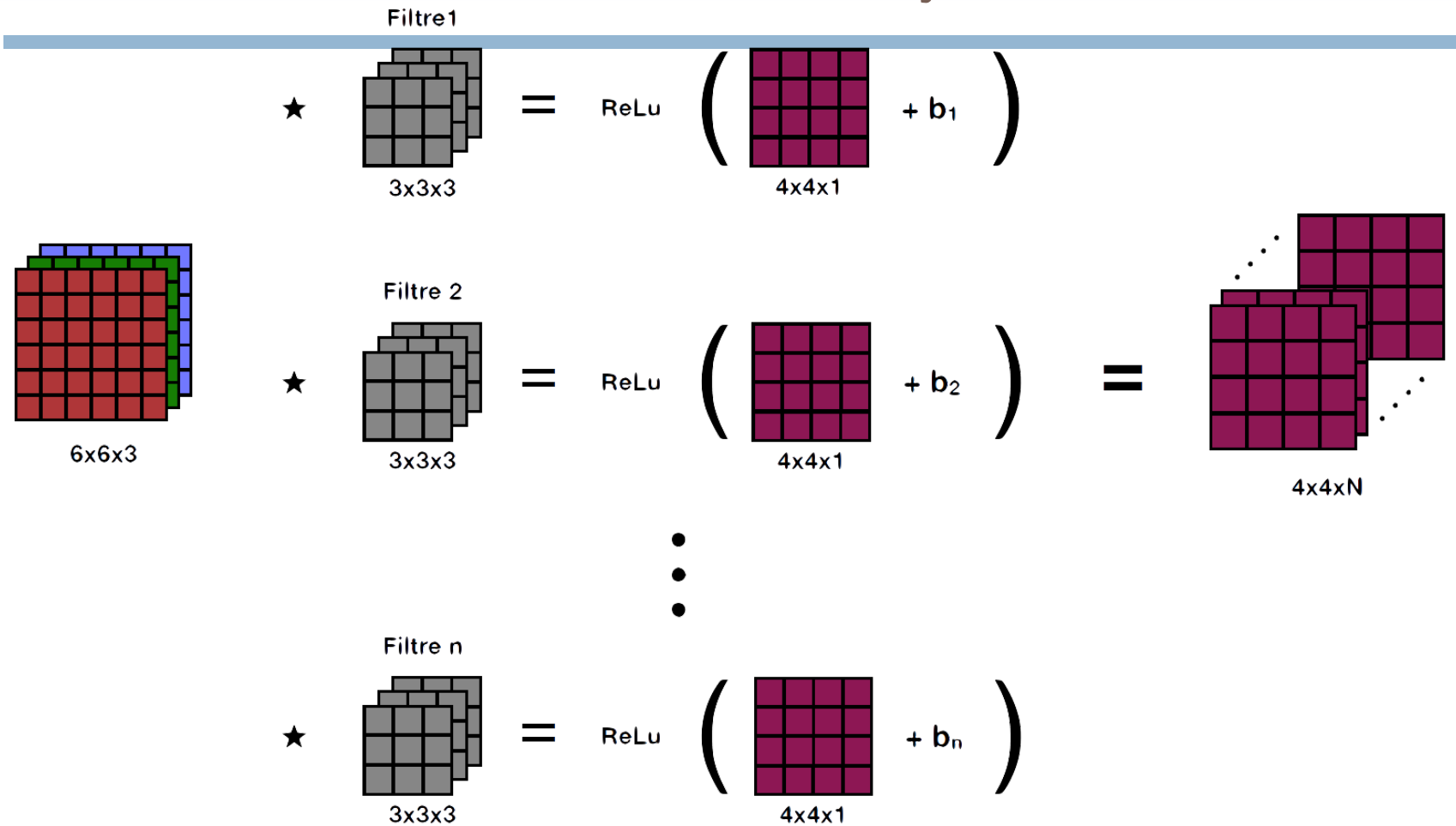


$$(F \star img)(x, y) = \sum_{c} \sum_{m} \sum_{n} F^{c}(n, m) \cdot Img^{c}(x + m, y + n)$$

Convolution - Layer



Convolution - Layer



- Input Dimension : (h, w, c_i)
- Filter Dimension : (f, f), Nb filter : N
- Output Dimension : $(h - f + 1, w - f + 1, N)$

Dimensions formula - Convolutional Layer

Input Dimensions : $(h \times w \times c)$

- h : height of the image,
- w : width of the image,
- c : channel of the image.

Convolution parameter : (f, p, s)

- f : filter size
- p : padding size
- s : stride size
- n : number of output channels

Output Dimensions :

$$\left(\left[\frac{h + 2p - f}{s} + 1 \right] \quad \times \quad \left[\frac{w + 2p - f}{s} + 1 \right] \quad \times \quad n \right)$$

Number of parameters : $N_p = \big((h \cdot w \cdot c) + 1\big) \cdot n$

Convolution layer

- 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 refered 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.

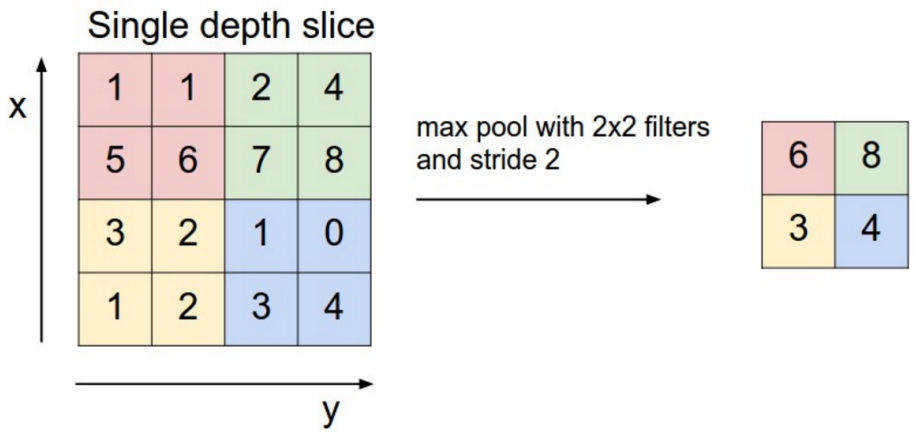


FIGURE – Maxpooling and effect on the dimension

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 experimental than 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.

LeNet-5 and Mnist Classification

Objectif :

- Manuscrit number from 0 to 9 hand-written
- Dimension (32,32,1)
- Image for training : $N_{train} = 60.000$
- Image for testing : $N_{test} = 10.000$

LeNet-5 and Mnist Classification

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

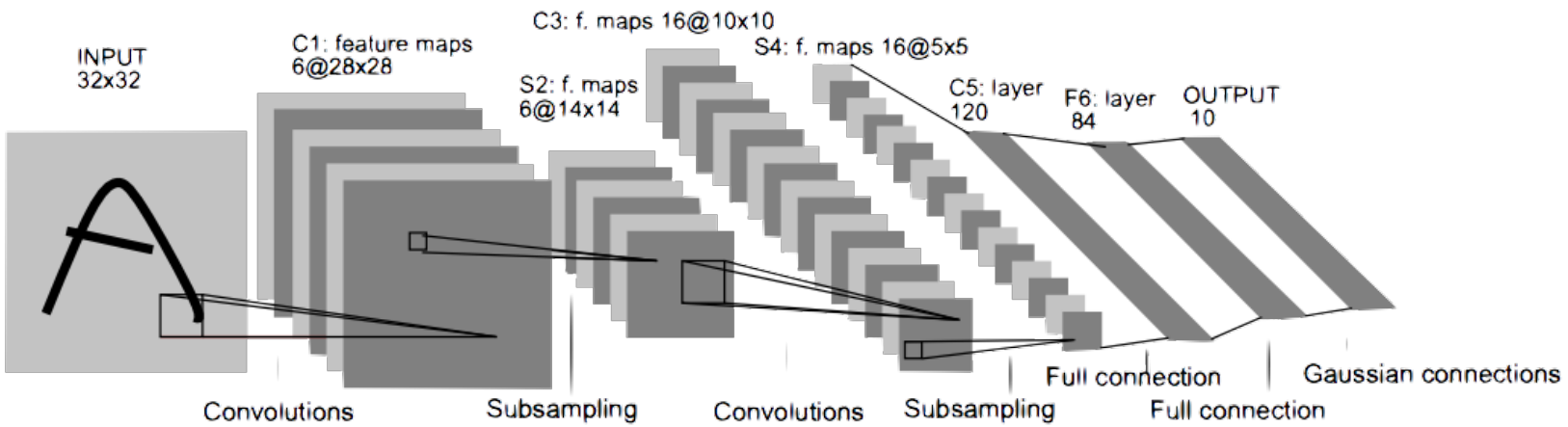


FIGURE – Architecture of the network Le Net. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998)

Size and height decrease while channel increase.

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 (227×227) onto 1000 classes.
- AlexNet is composed of 5 convolution layers, 3 max-pooling layers and fully connected layers.

Alex Net (Krizhevsky, A. et al (2012))

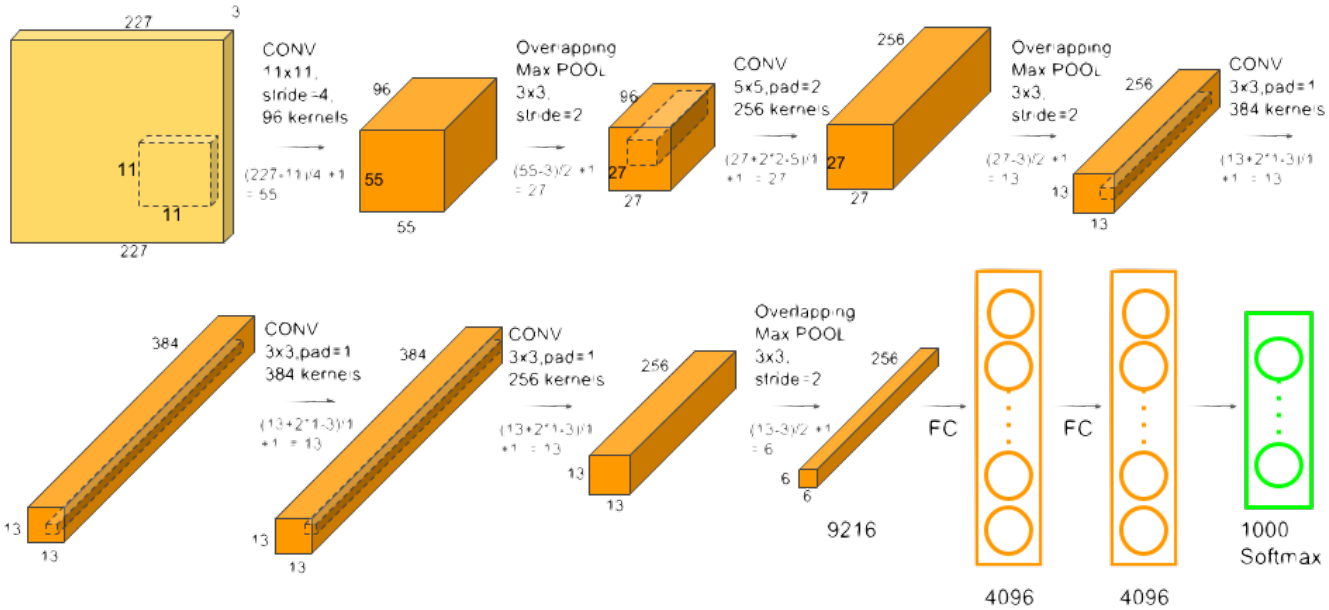
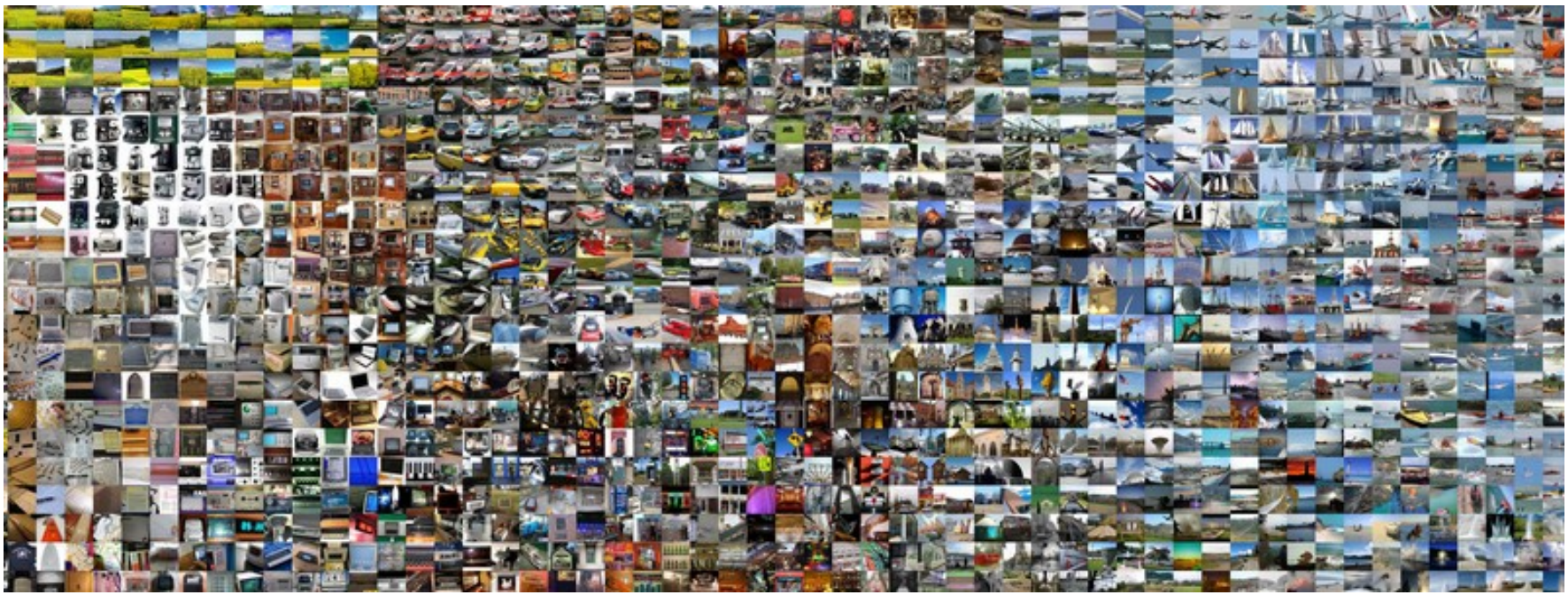


Image source : https://www.learnopencv.com/understanding-alexnet/

- $\simeq 60M$ parameters
- Similar to LeNet-5 but Much bigger
- Only ReLu is different.
- Multiple GPUs
- Learned on huge amount of data : **ImageNet** .

Image net

http://image-net.org/



- 1000 classes
  - 1,2 M training images,
  - 100k test images.

VGG16

Use always same layer :

- Convolution : 3×3 filter, stride $= 1$, padding$=$ SAME
- Max Pooling : 2×2 filter, stride $= 2$

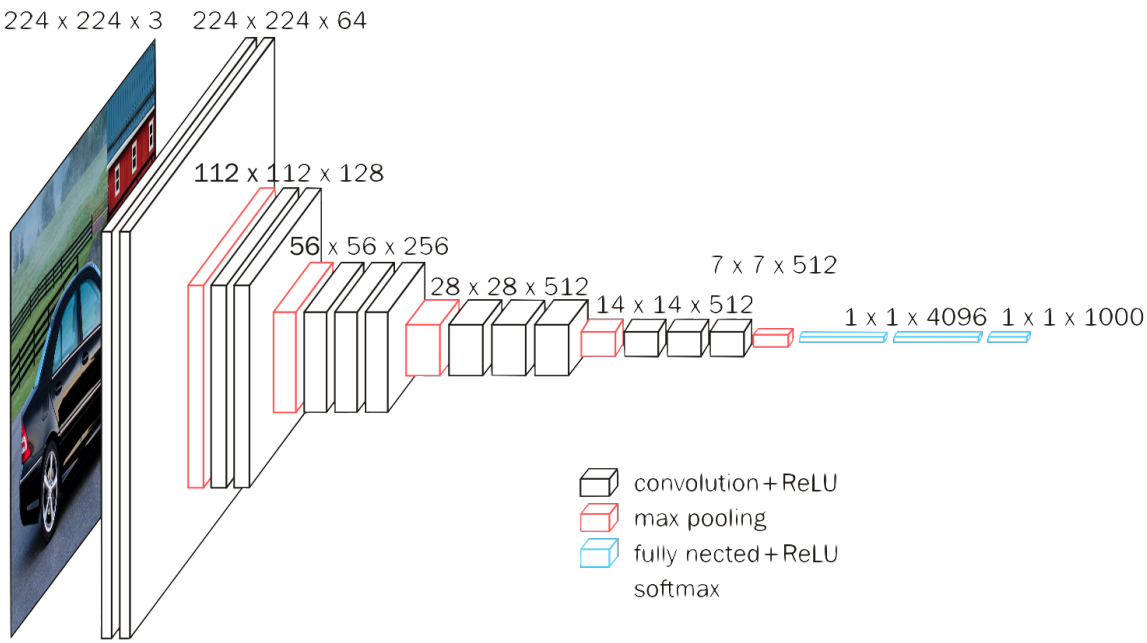
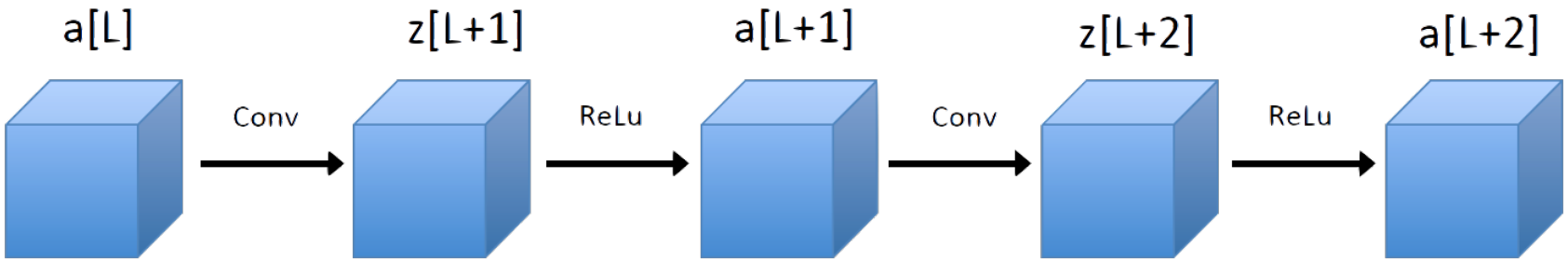


FIGURE – Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition (2014). 138M parameters.

Image source : https://blog.datawow.io/cnn-models-ef356bc11032, realized with Tensorflow.

ResNet Block - He et al. (2016)

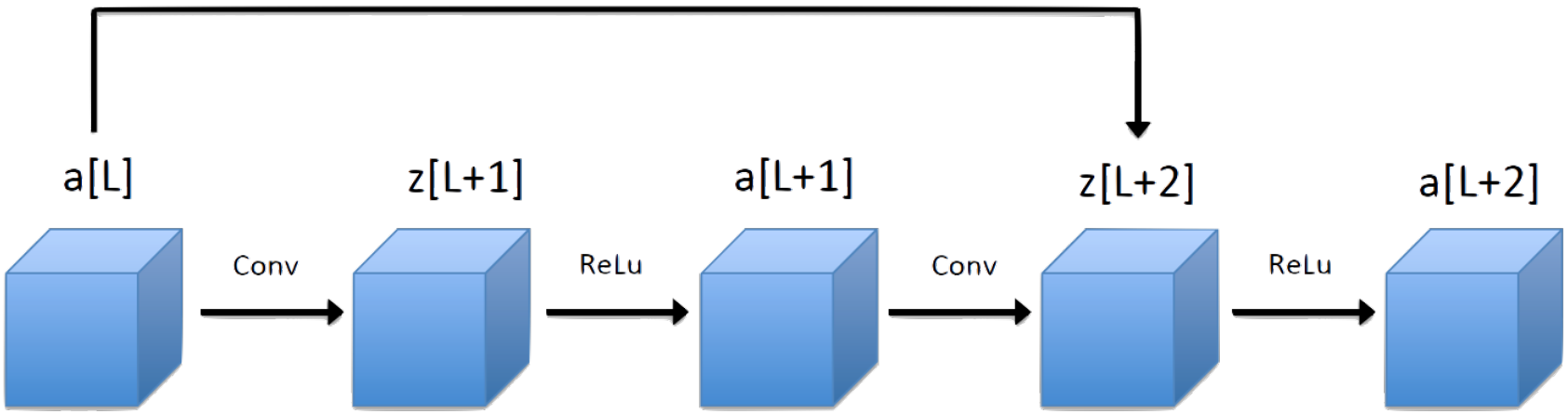
Convolution with padding = SAME



$$
\begin{aligned}
z[l+1] &= Conv(a[l]) \\
a[l+1] &= ReLu(z[l+1]) \\
z[l+2] &= Conv(a[l+1]) \\
a[l+2] &= ReLu(z[l+2] \qquad\qquad)
\end{aligned}
$$

ResNet Block - He et al. (2016)

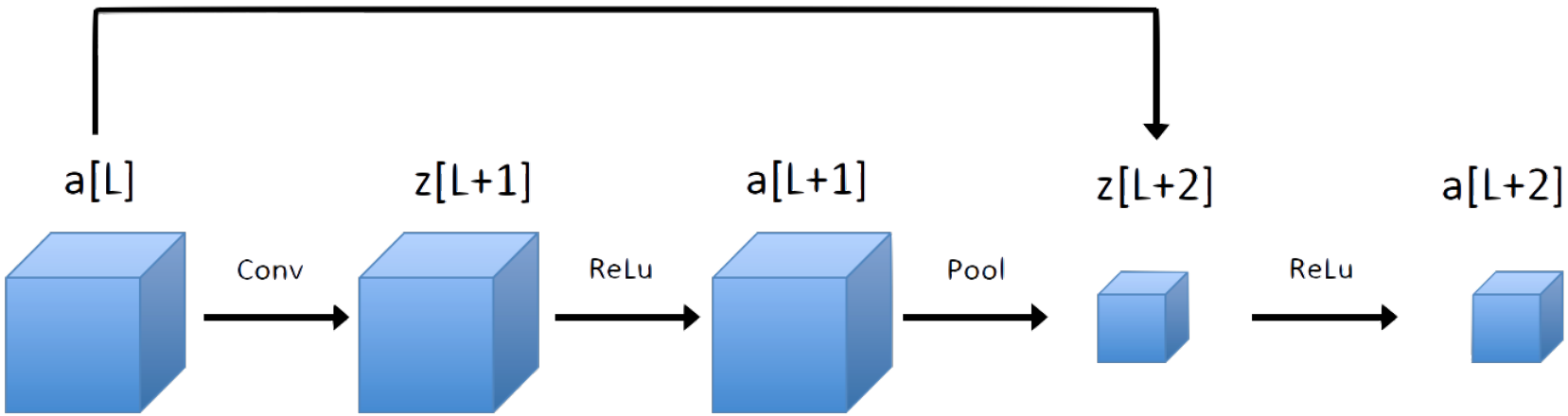
Convolution with padding = SAME



$$z[l+1] = Conv(a[l])$$
$$a[l+1] = ReLu(z[l+1])$$
$$z[l+2] = Conv(a[l+1])$$
$$a[l+2] = ReLu(z[l+2] + \quad a[l])$$

ResNet Block - He et al. (2016)

MaxPooling



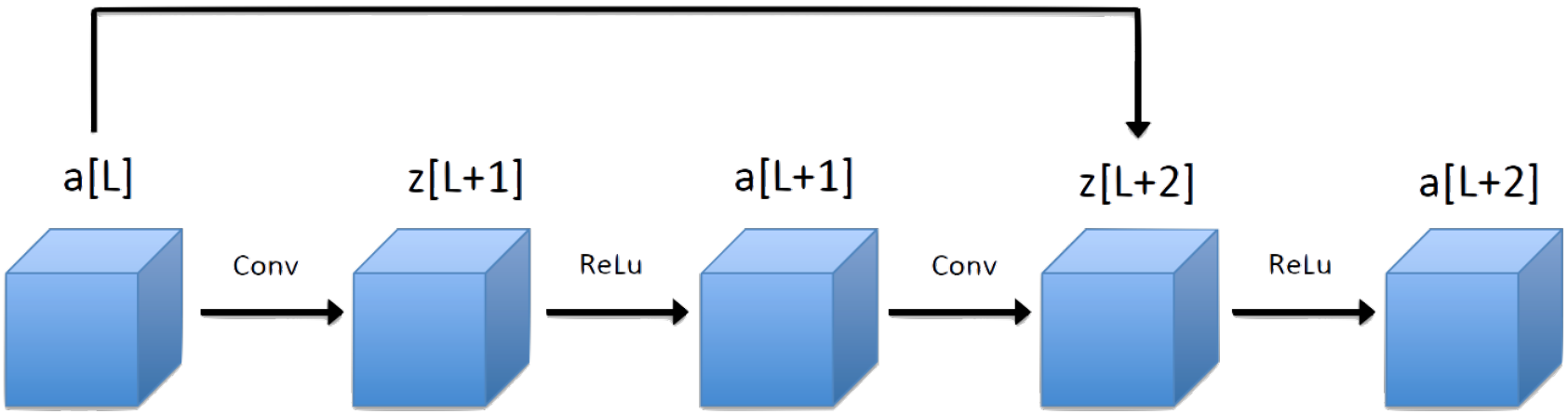
$$z[l+1] = Conv(a[l])$$
$$a[l+1] = ReLu(z[l+1])$$
$$z[l+2] = MaxPool(a[l+1])$$
$$a[l+2] = ReLu(z[l+2] + W \cdot a[l])$$

ResNet Block - Properties

Making a convolutional network too deep can hurt its performance.

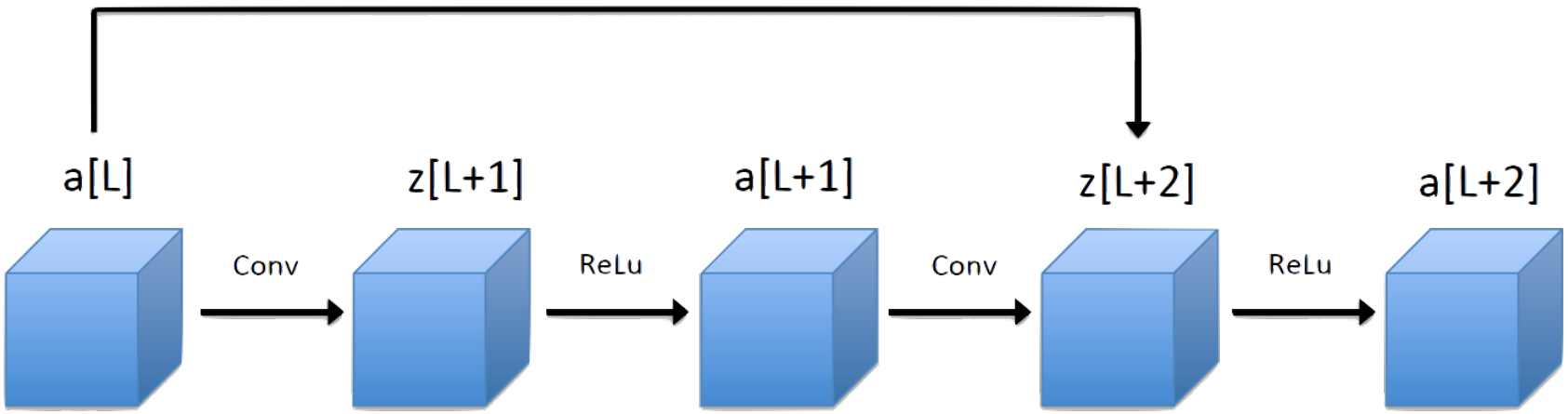
ResNet Block - Properties

Making a convolutional network too deep can hurt its performance.
With Resnet, you can make your network deeper...
Convolution with padding = SAME



ResNet Block - Properties

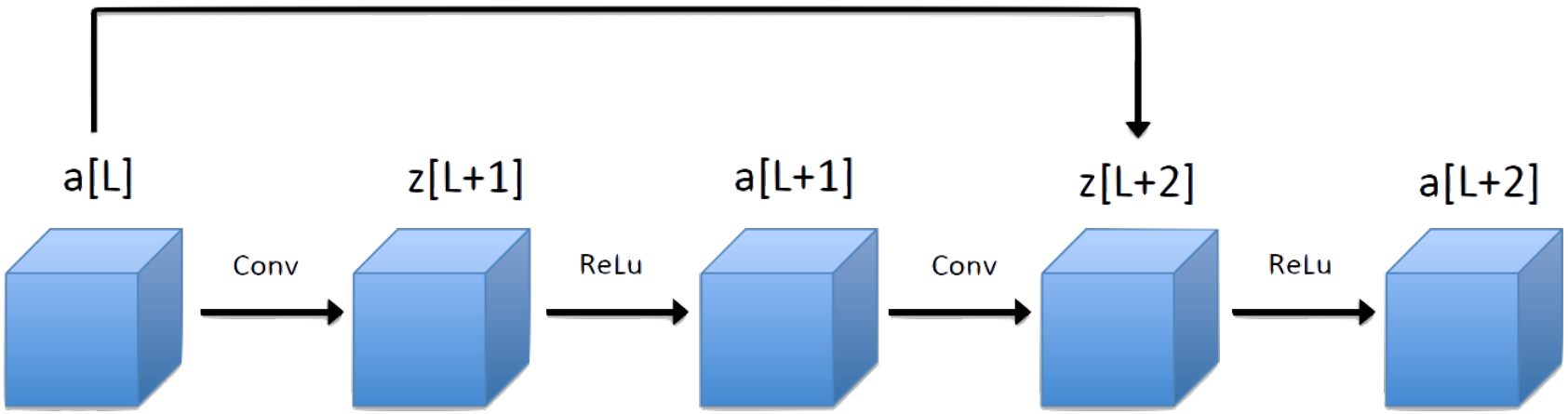
Making a convolutional network too deep can hurt its performance.
With Resnet, you can make your network deeper...
Convolution with padding = SAME



$$z[l+1] = Conv(a[l])$$
$$a[l+1] = ReLu(z[l+1])$$
$$z[l+2] = Conv(a[l+1])$$
$$a[l+2] = ReLu(z[l+2] + a[l])$$
$$= ReLu\Big(w[l+1] \cdot a[l+1] + b[l+1]\Big) + a[l]$$

ResNet Block - Properties

Making a convolutional network too deep can hurt its performance.
With Resnet, you can make your network deeper...
Convolution with padding = SAME



$$z[l+1] = Conv(a[l])$$
$$a[l+1] = ReLu(z[l+1])$$
$$z[l+2] = Conv(a[l+1])$$
$$a[l+2] = ReLu(z[l+2] + a[l])$$
$$= ReLu\Big(w[l+1] \cdot a[l+1] + b[l+1]\Big) + a[l]$$

- ..without hurting its performance..
- ... and with a bit of luck, improving it !

ResNet

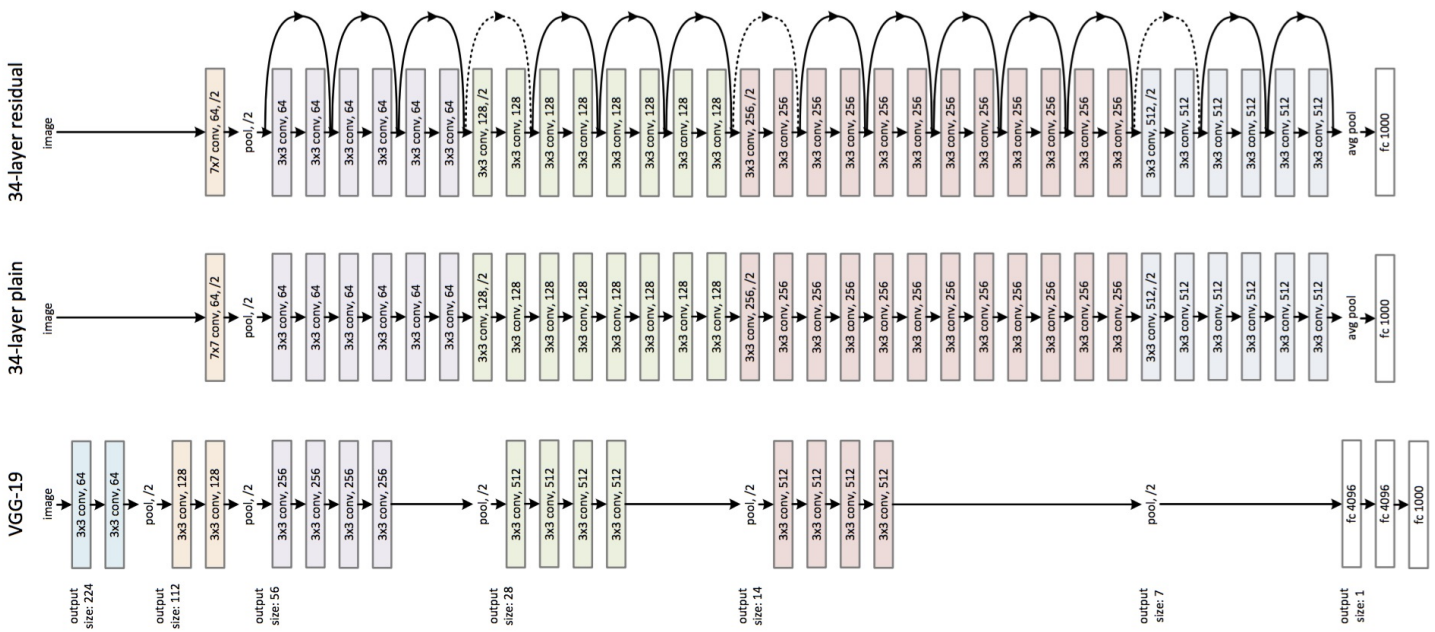
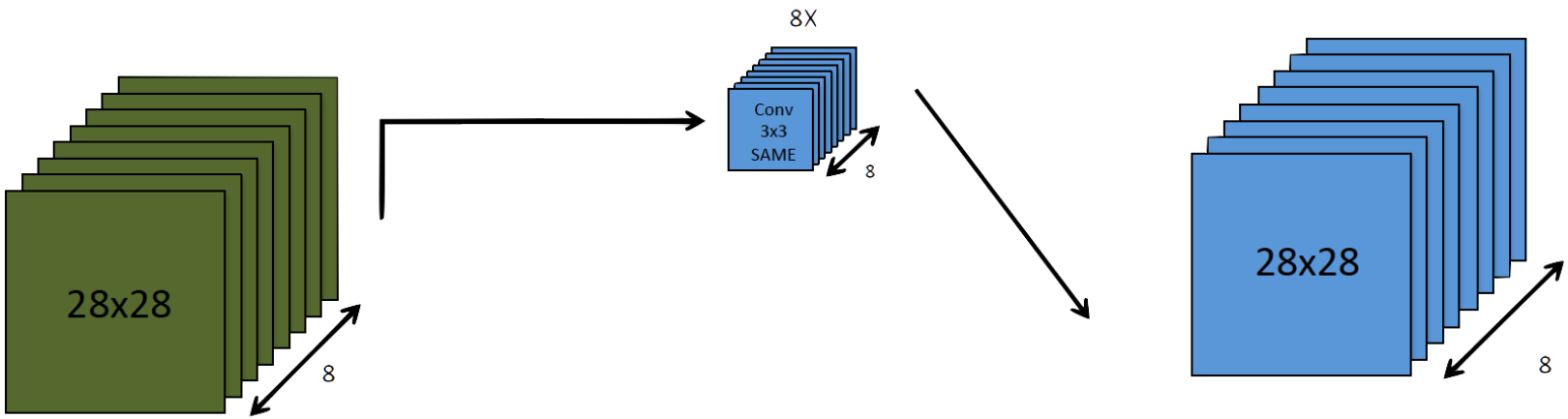


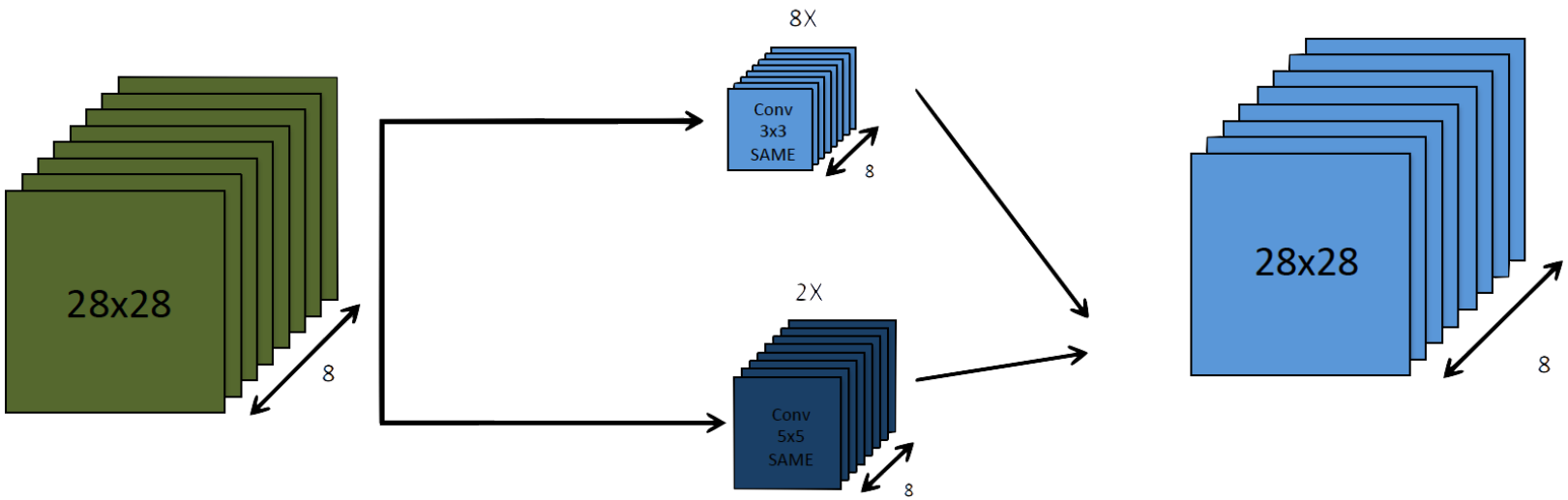
Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

Inception Block - Szegedy et al. (2015)



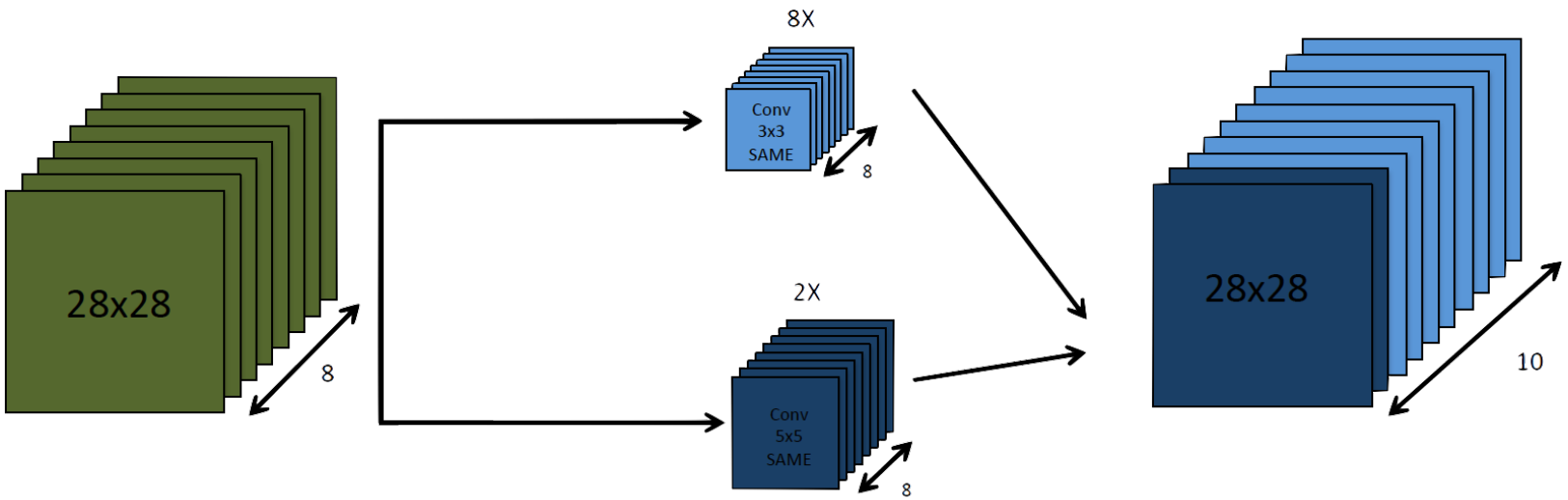
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



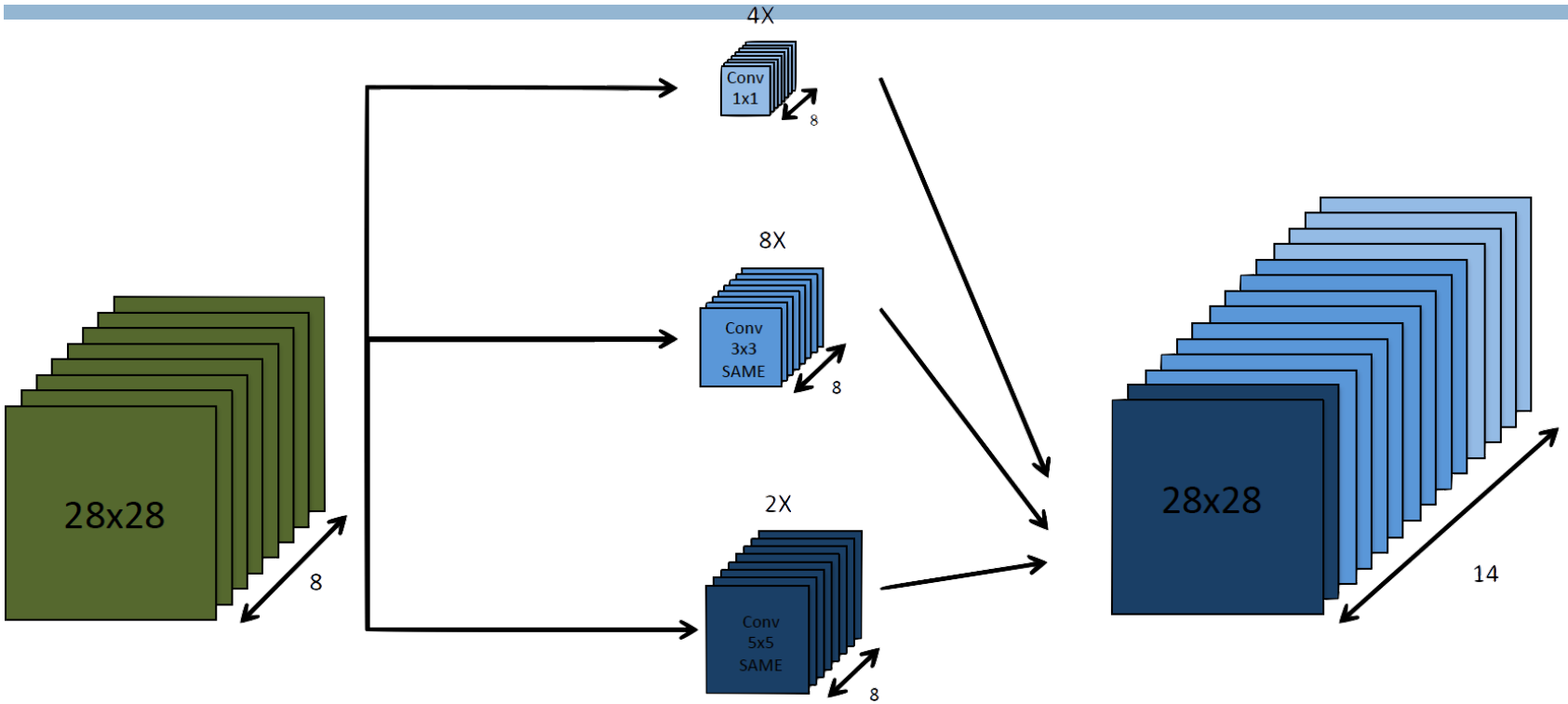
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



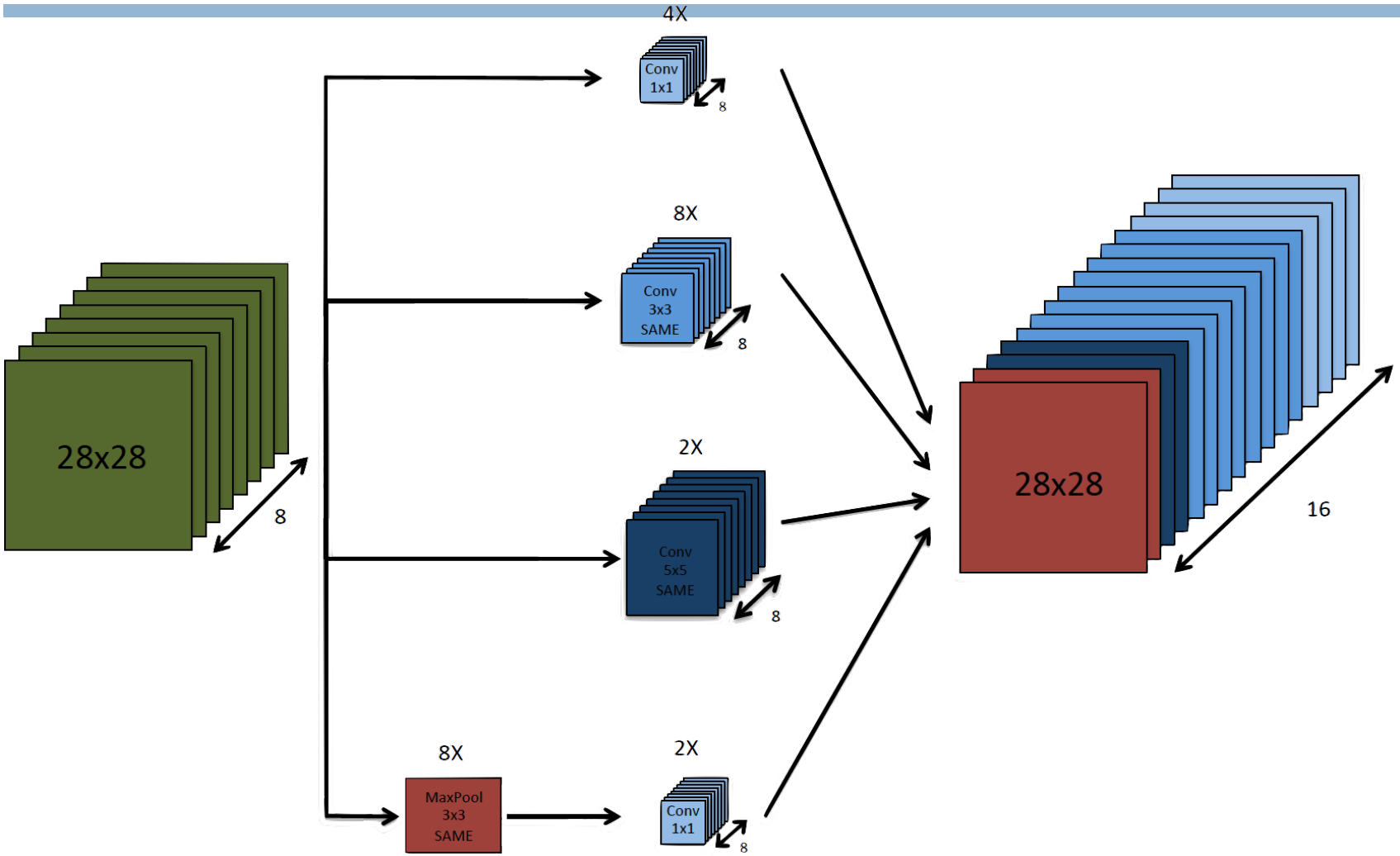
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



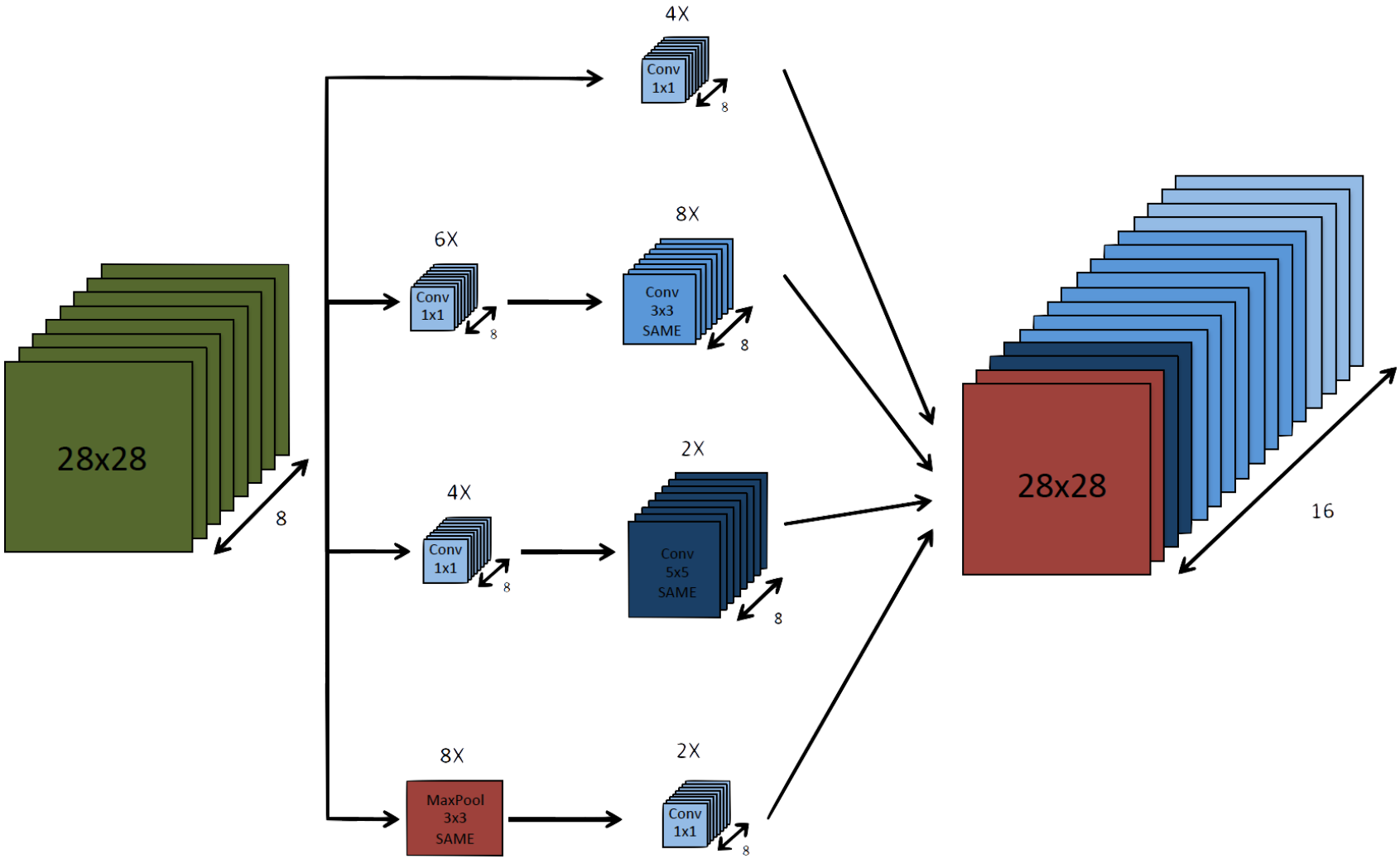
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



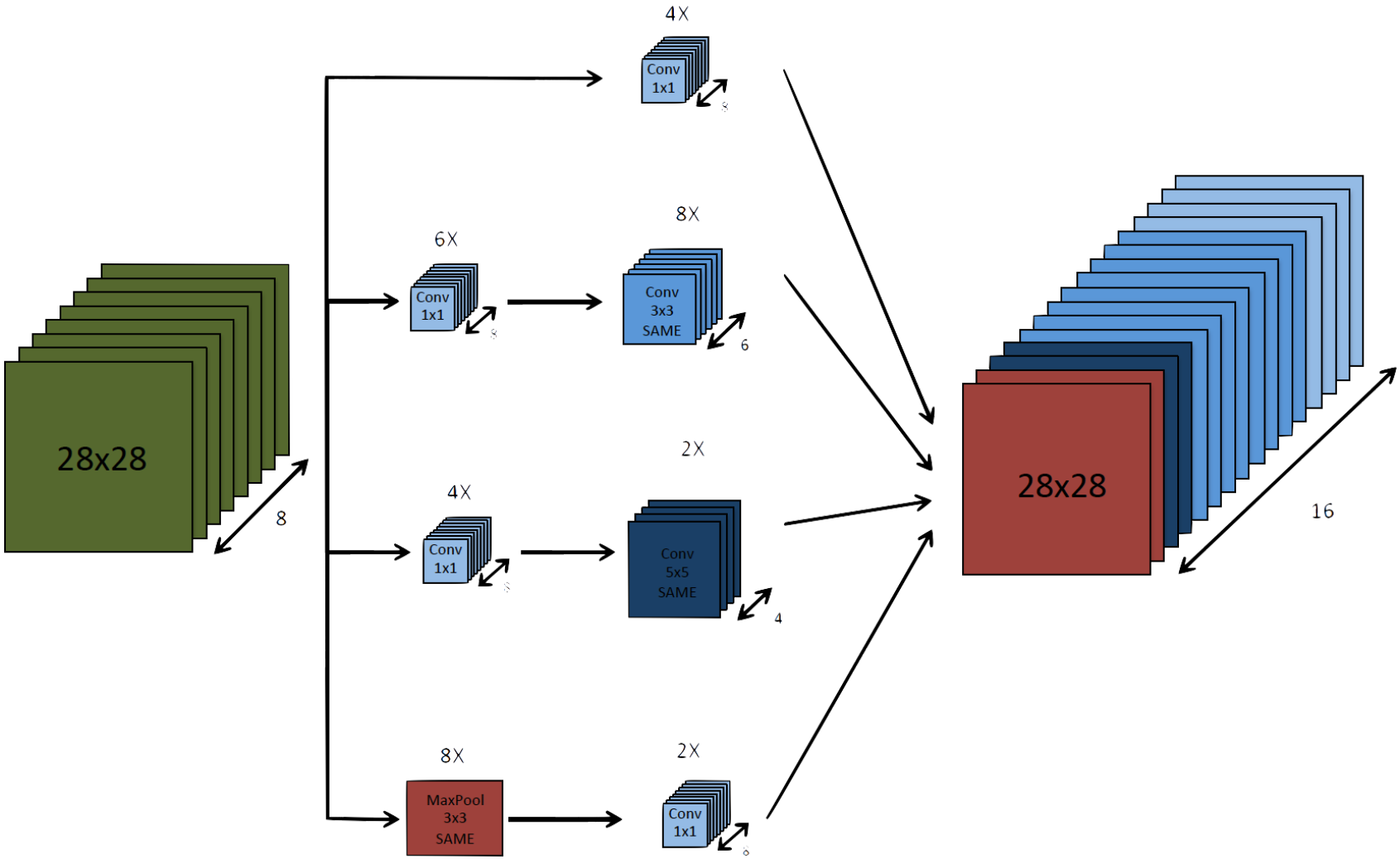
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



Convolution with 3x3 filters and padding = SAME

State of the art

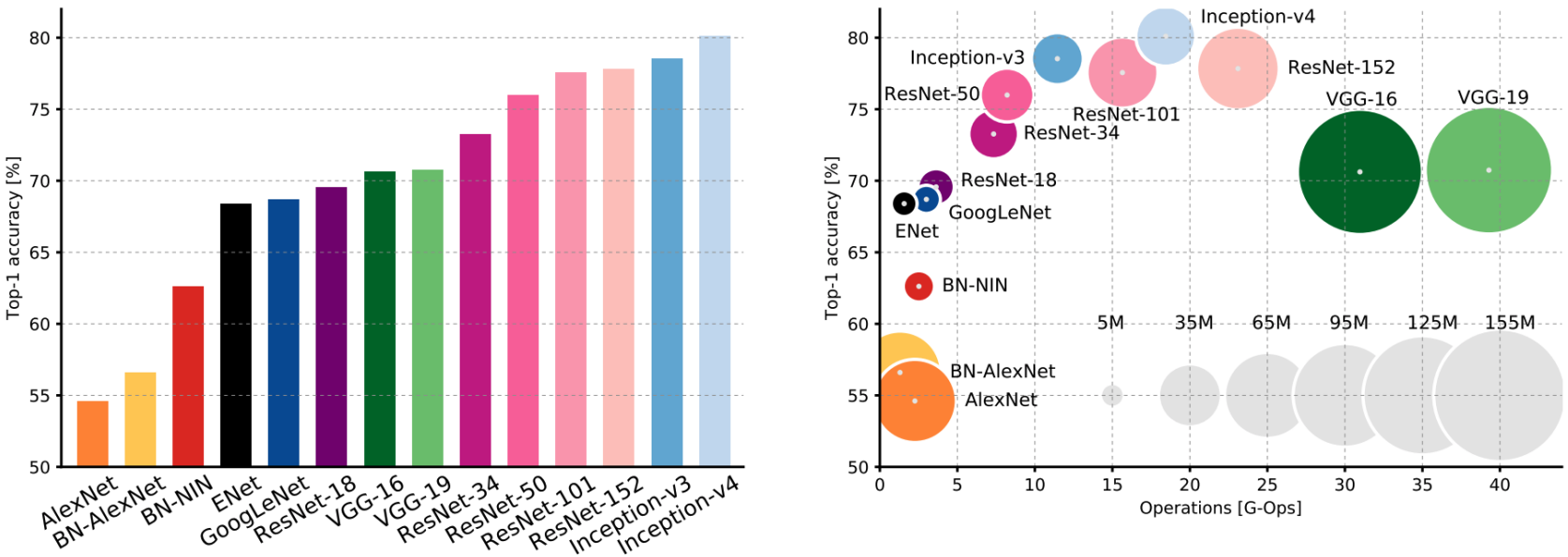


Image source : **canziani2016analysis**

- In constant evolution,
- New architectures regularly proposed.

Librairies and frameworks for DeepLearning

In practice

Pre-trained model

- Some models using previous architectures have already be trained on massive amount of data (ImageNet).
- Theses models are available on easily usable on deep learning Framework such as Keras (Python library).
- Two ways to used them :
  - **Transfer Learning**
  - **Fine Tuning**

Data Augmentation

Localisation and Segmentation

Localisation : YOLO, SSD, Fast-RCNN
Segmentation.

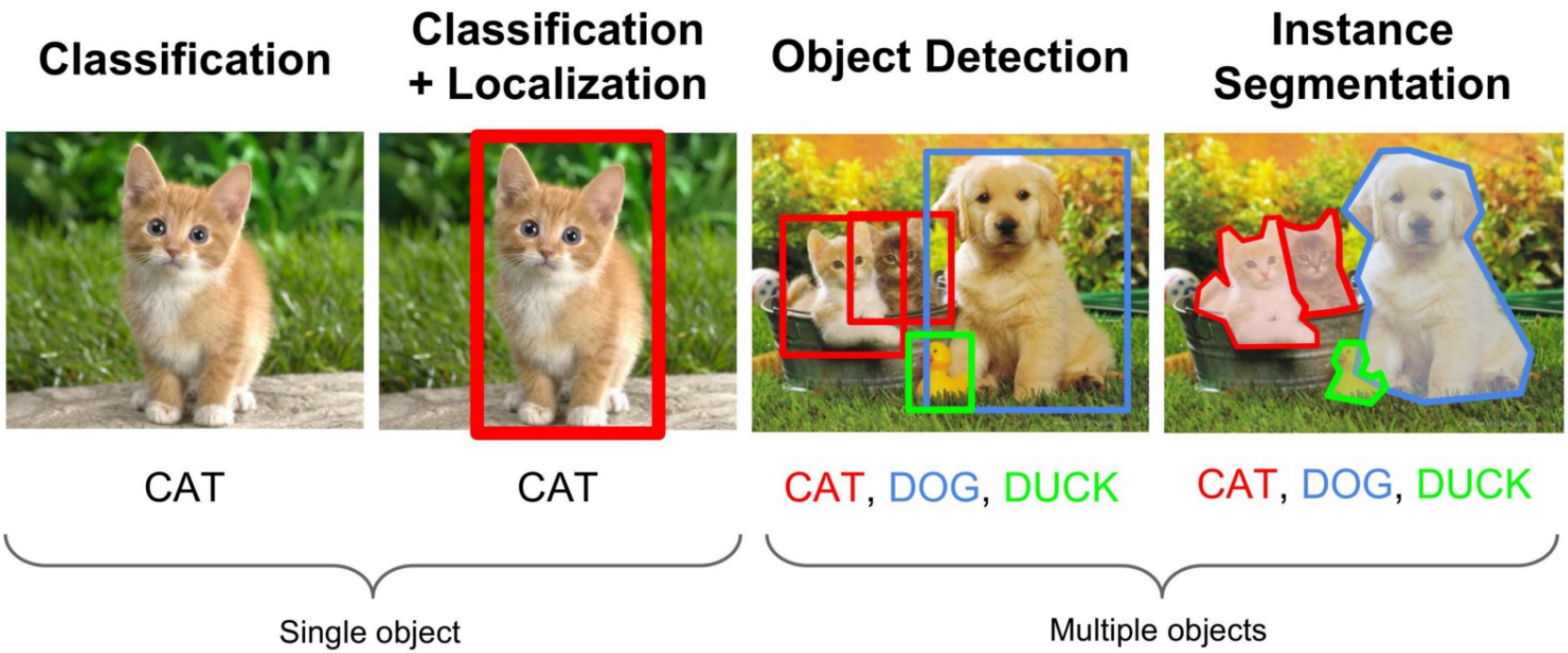
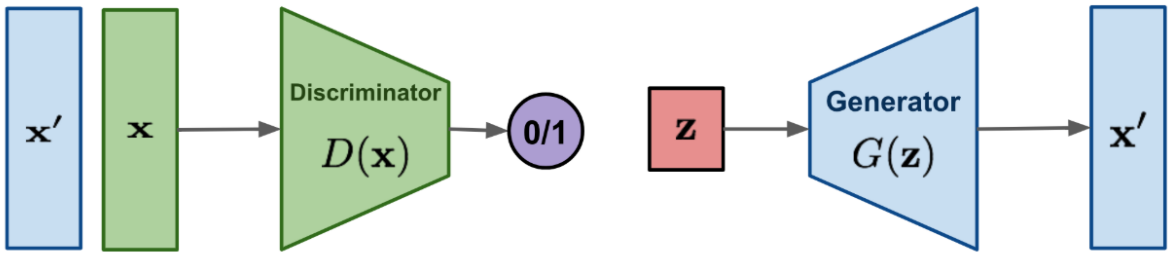
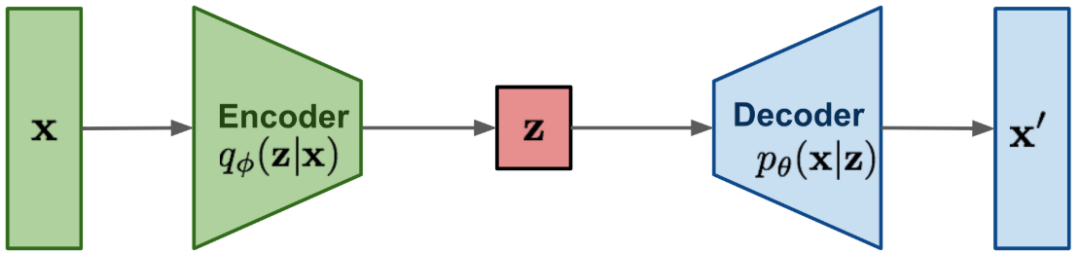


Image Generation

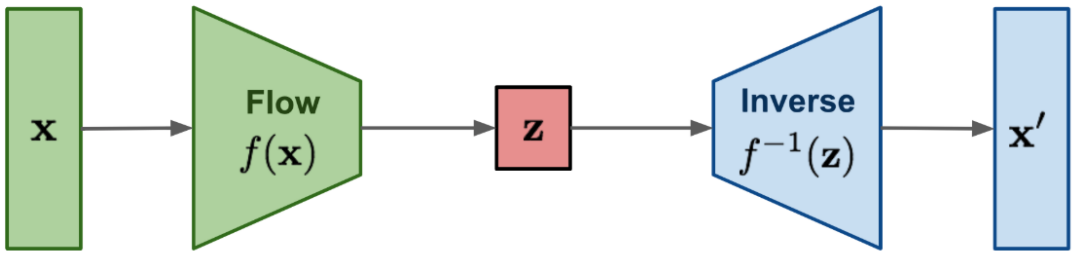
GAN: minimax the classification error loss.



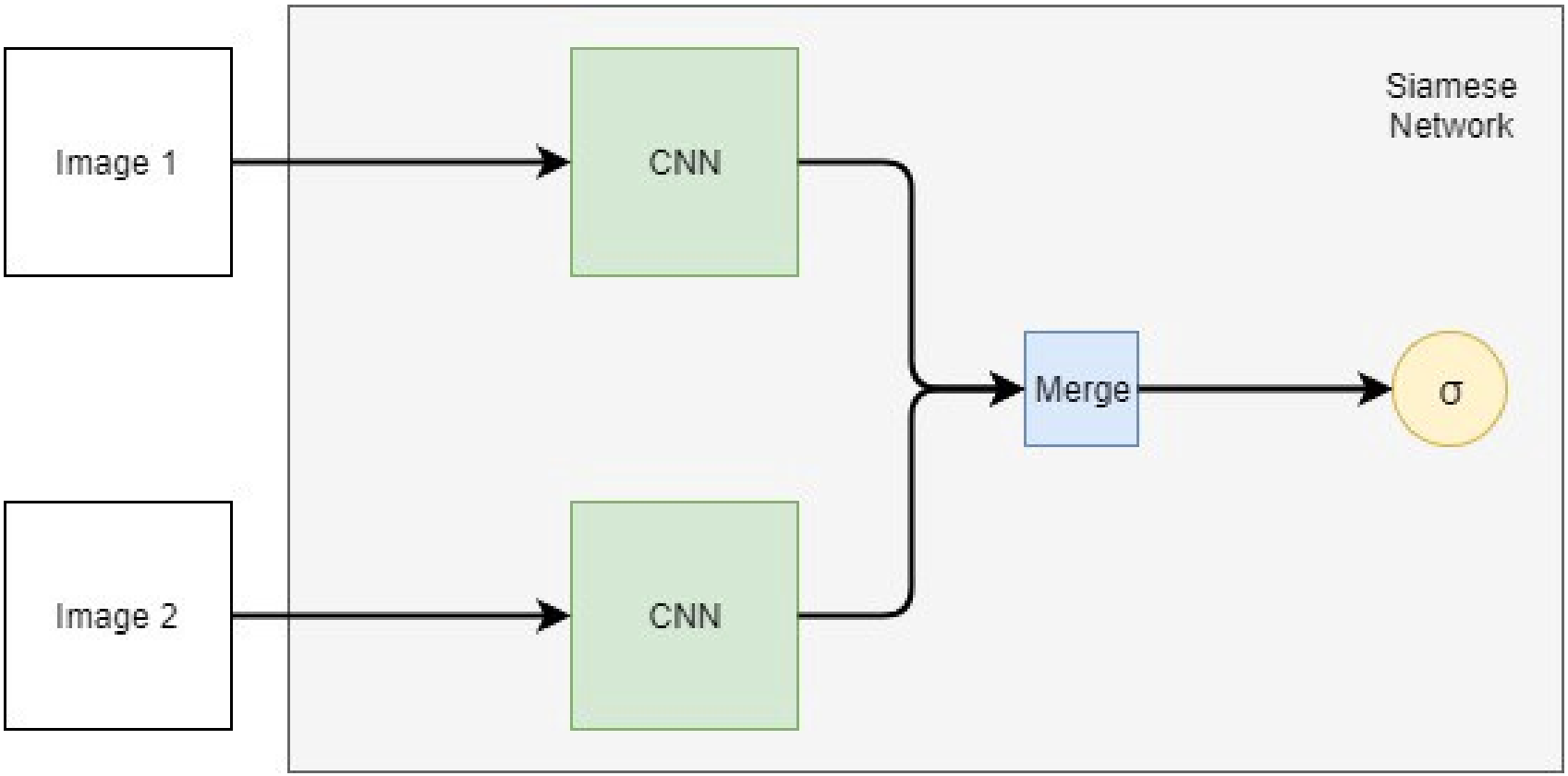
VAE: maximize ELBO.



Flow-based generative models: minimize the negative log-likelihood



Siamese Network - OneShotLearning



Outline

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

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, **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 neural 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

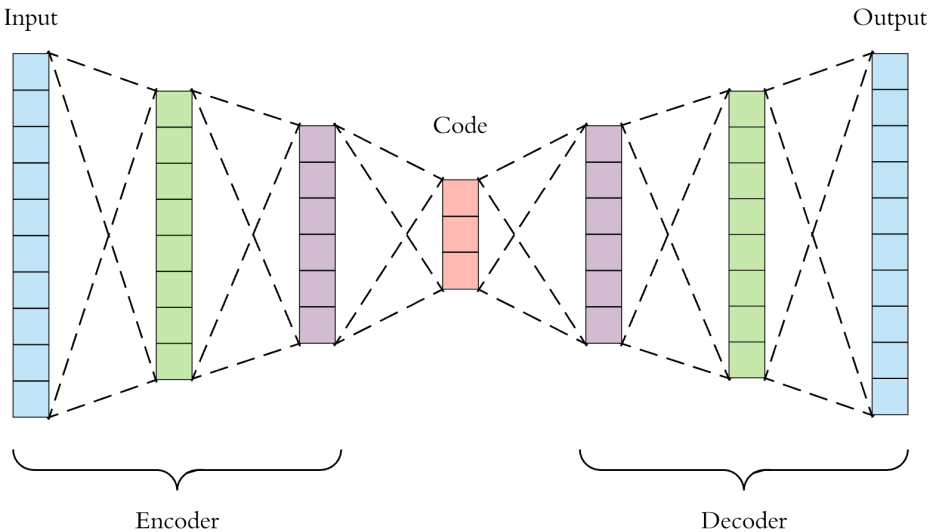


FIGURE – The structure of an autoencoder : mapping the input data to a reconstruction via a sparse internal representation code. Source : *towardsdatascience.com*

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 an $\mathbb{L}_1$ penalty $\Omega(\mathbf{h})$ on the code $\mathbf{h}$, this leads to minimize

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

with

$$\Omega(\mathbf{h}) = \lambda \sum_i |h_i|.$$

- This kind of autoencoders are typically used to learn features for classification purposes.

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 $\mathbf{x}$.
- Other kinds of autoencoders : Variational AutoEncoders (VAE) have been introduced by Kingma (2013), Rezende et al (2014).

Variational autoencoders

- We have seen that the encoders convert the input data into an encoding vector, corresponding to a sparse representation of the data.
- Each component of the encoding vector corresponds to some learned feature of the input data.
- An important fact is that, for each encoding dimension, a single value is given by the encoder.
- The decoder then takes these values as input to reconstruct the original data.
- **The variational autoencoder (VAE)** does not more return a single value for each component of the encoding vector, but a probability distribution.

Variational autoencoders

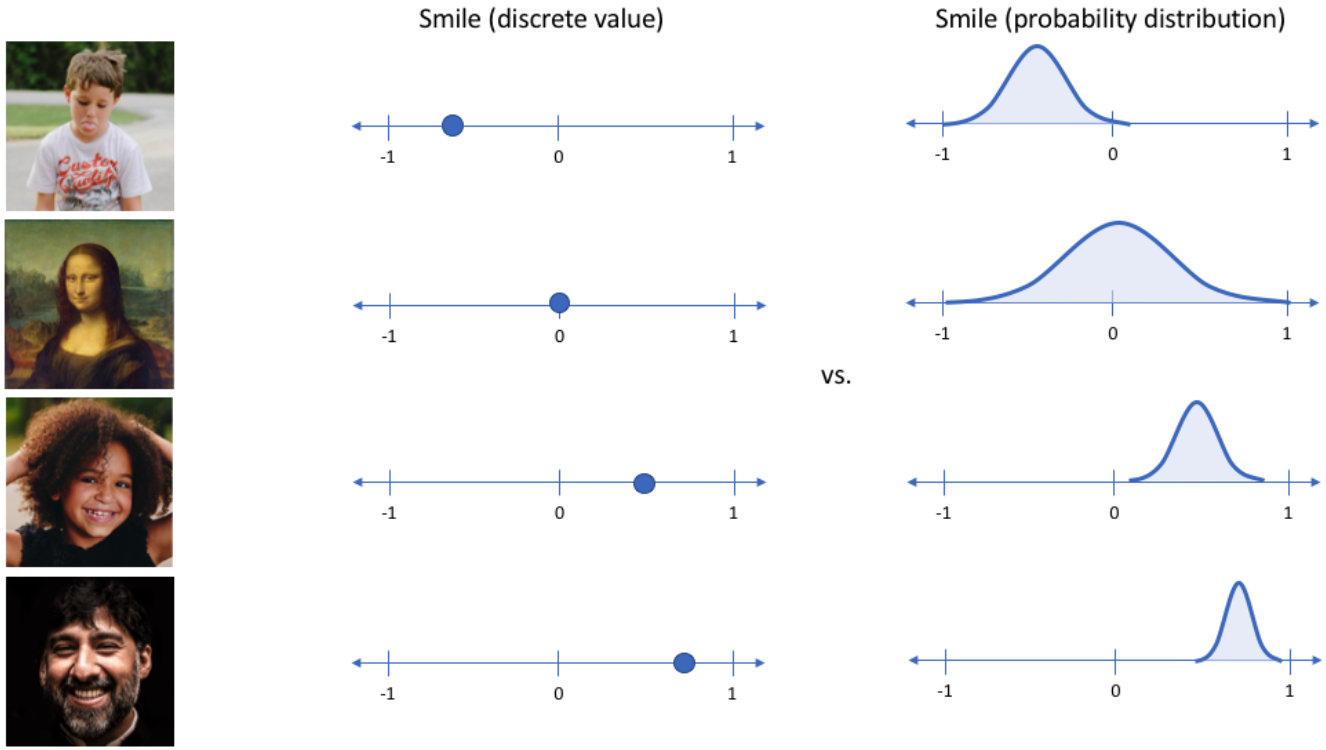


FIGURE – Source : https ://www.jeremyjordan.me

Variational autoencoders

- In VAEs, each latent variable is represented by a probability distribution, and to provide the inputs of the decoder, we sample from this probability distribution for each component of the encoding vector.
- The decoder should accurately reconstruct the input whatever the sampling for the encoding vector.
- By this way, we enforce close values in the latent space to provide quite similar reconstructions. Hence the reconstruction is smooth with respect to the latent space representation.

Variational autoencoders

- Let us assume that a hidden variable z with known prior distribution $p(z)$ (generally standard Gaussian variables) generates the observation x. We only observe x, we want to infer the distribution of z given x.
- We have $p(z/x) = \frac{p(x/z)p(z)}{p(x)}$, but $p(x)$ is intractable to compute in general.

$$p(x) = \int p(x/z)p(z)dz$$

  which is generally in integral in high dimension.
- The **variational approach** consists in approximating $p(z/x)$ by a simple distribution $q(z/x)$ (for example a normal distribution). The parameters of $q(z/x)$ (mean and standard deviation for a normal distribution) will be estimated in such a way that $q(z/x)$ is close to $p(z/x)$.
- We aim at minimizing $KL(q(z/x), p(z/x))$ (where KL denotes the Kullback-Leibler divergence).

Variational autoencoders

$$KL(q(z/x), p(z/x)) = -\int q(z/x) \log\left(\frac{p(z/x)}{q(z/x)}\right) dz$$

- $p(z/x) = p(x,z)/p(x)$ and $\int q(z/x)dz = 1$, hence we have

$$KL(q(z/x), p(z/x)) = -\int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) + \log(p(x)).$$

- Equivalently,

$$\log(p(x)) = KL(q(z/x), p(z/x)) + \int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) dz.$$

- $\log(p(x))$ is a fixed quantity (with respect to $q(z/x)$).
- Hence, minimizing $KL(q(z/x), p(z/x))$ is equivalent to maximize $\int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) dz$.

Variational autoencoders

- We want to maximize $\int q(z/x) \log\left(\frac{p(x,z)}{q(z/x)}\right) dz$. This is equal to

$$\begin{aligned} & \int q(z/x) \log\left(\frac{p(x/z)p(z)}{q(z/x)}\right) dz \\ = & \int q(z/x) \log\left(p(x/z)\right) dz + \int q(z/x) \log\left(\frac{p(z)}{q(z/x)}\right) dz \\ = & \mathbb{E}_{z\sim q(z/x)} \log(p(x/z)) - KL(q(z/x), p(z)). \end{aligned}$$

- We recall that we choose a very simple prior on z, $p(z)$: generally independent standard normal distributions, we also choose a simple distribution for $q(z/x)$: generally independent normal distributions.
- The means and standard deviations of the components of $q(z/x)$ are learned to maximize the above quantity, as well as the parameters of the neural networks corresponding to the encoder and decoder.

Variational autoencoders

We maximize $\mathbb{E}_{z\sim q(z/x)} \log(p(x/z)) - KL(q(z/x), p(z))$.

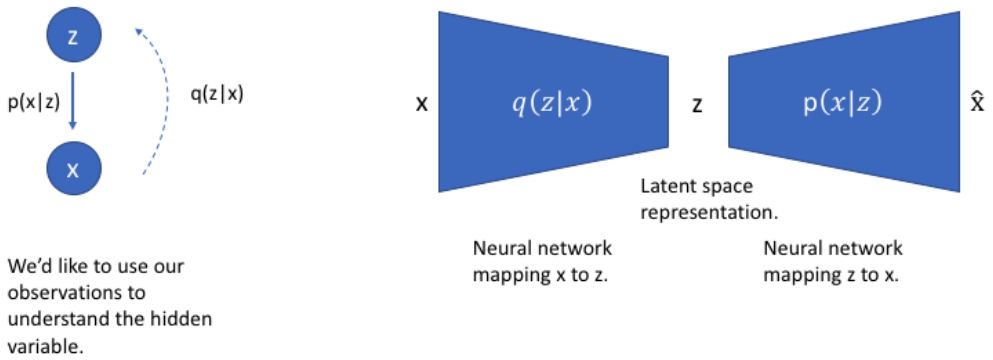


FIGURE – Source : https ://www.jeremyjordan.me

If we consider the loss function associated to the negative log-likelihood, we have

$$\ell(x, \hat{x}) = -\mathbb{E}_{z\sim q(z/x)} \log(p(x/z))$$

The VAE is trained by minimizing the penalized loss function :

$$\ell(x, \hat{x}) + KL(q(z/x), p(z)).$$

Variational autoencoders

- For the sake of simplification, we assume here that $q(z/x)$ is Gaussian with independent components. Instead of computing the latent variables as classical autoencoders, the VAE gives as output of the encoder the estimated mean and standard deviation for each latent variable z_j. The decoder will then sample from this distribution to compute $\hat{x}$.

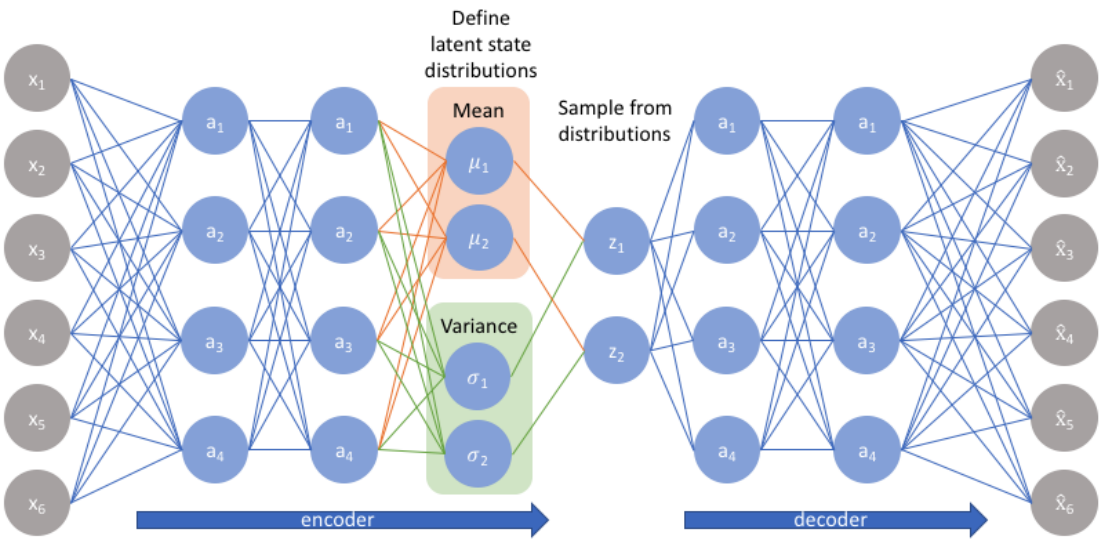


FIGURE – Source : https ://www.jeremyjordan.me

Variational autoencoders

- All the parameters are estimated by using the backpropagation equations.
- The problem is that here we have random variables (the variables z_j) in the network.
- In order to overcome this problem, we use a reparametrization trick, schematized below :

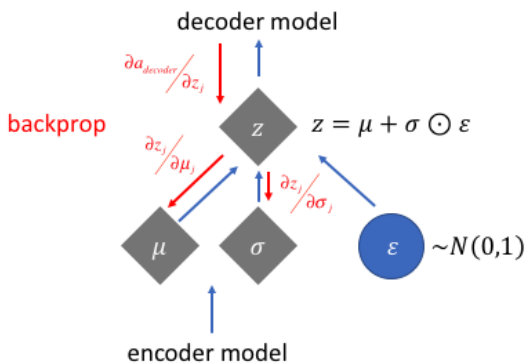


FIGURE – Source : https ://www.jeremyjordan.me

Variational autoencoders

- It is interesting to visualize the latent variables to interpret these features learned from the data.
- Variational autoencoder can be use to generate new data : we generate z from the learned distribution $q(z/x)$ and we then compute the corresponding $\hat{x}$ from the decoder.
- This provides smooth transformations of the original data, which can be interesting when we want to interpolate between observations (for example images or sound).

- Autoencoders have many applications : they can be used for classification considering the latent code as input to a classifier instead of $\mathbf{x}$.
- 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).
- Another way to generate new samples is to use Generative Adversarial Networks (GAN) presented in the next section.

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.
- The next figure gives a schematic representation of GANs.

Generative Adversarial Networks

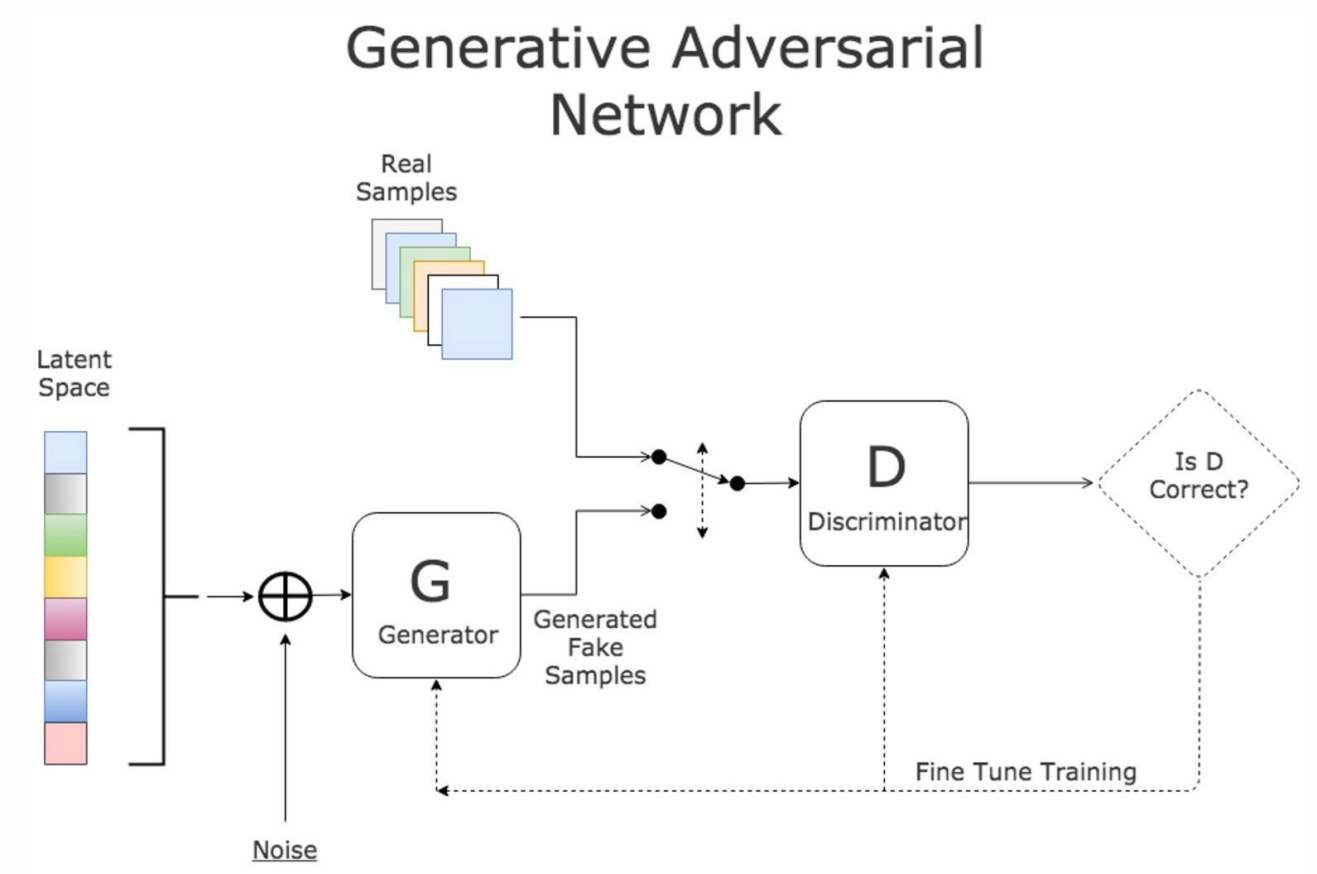


FIGURE – Source : blog.statsbot.co

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 : **the discriminator** $D(x, \theta_d)$ 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 :
  - $G(z)$ 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

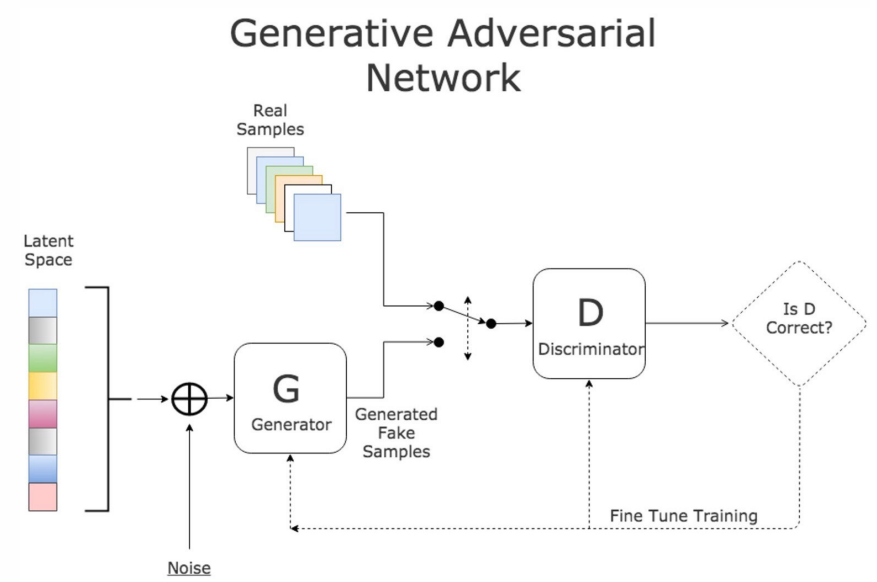


FIGURE – Source : blog.statsbot.co

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

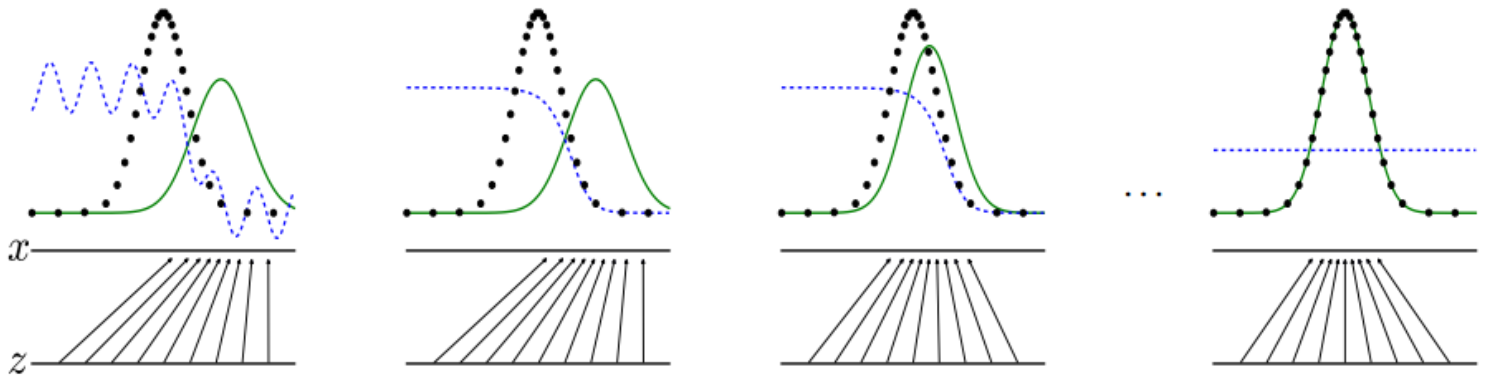


FIGURE – Successive steps in GANs training- In black, dotted line : the data generating distribution ; in green solid line : the generator distribution ; in blue, dashed line : the discriminator - z is sampled uniformly over the domain described by the lower lines- Source : cs.stanford.edu

- 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

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

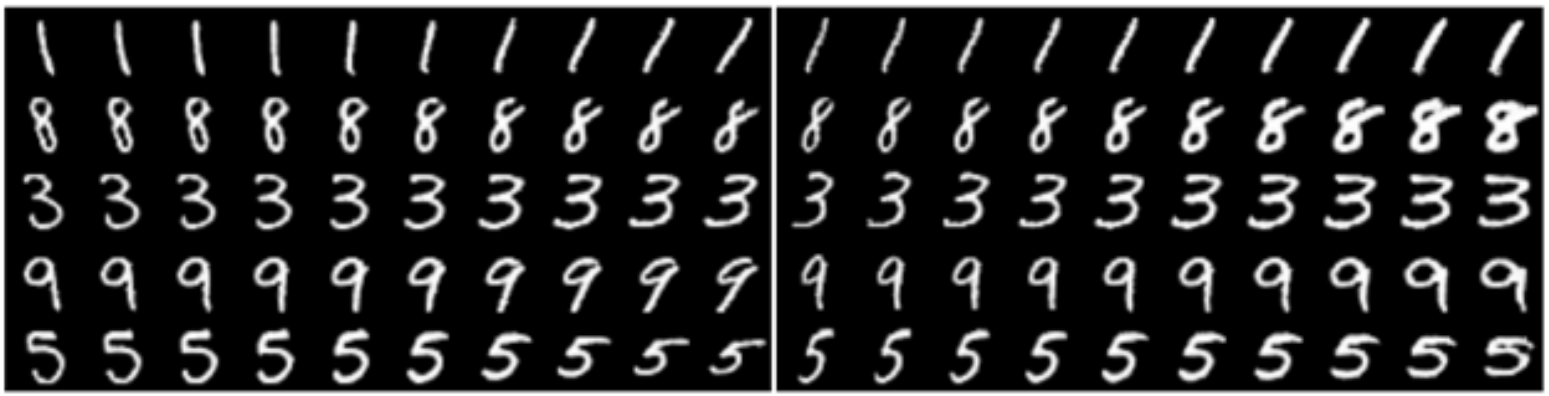
Generative Adversarial Networks

- 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 the next 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)

Generative Adversarial Networks

The latent variables obtained from an encoder can also be considered as latent input variables for GANs.

The literature on GANs, and more generally on deep learning is very abundant, the methods evolve very quickly and it crucial to train yourself permanently on these topics.

Beneficial uses of GANs are widespread, for example for pedagogical purposes, art, generate examples for images data sets, inpainting, 3D objects generation .. but also for malicious applications such as fake images, sound, videos or news.

Outline

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- Autoencoders, VAE and GAN
- Conclusion

Conclusion

- We have presented in this course the feedforward neural networks and explained how the parameters of these models can be estimated.
- The choice of the architecture of the network is also a crucial point. Several models can be compared by using a validation methods on a test sample to select the "best" model.
- The perceptrons are defined for vectors. They are not well adapted for some types of data such as images. By transforming an image into a vector, we loose spatial information, such as forms.

Conclusion

- 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 channels.
- They involve complex architectures, which are in constant evolution.
- Various libraries and frameworks have been developed : we will use Keras (Tensorflow) for simplicity reasons.
- GPU is required for training the deep networks.
- Few theoretical foundations available.

References

- François Cholet *Deep learning with Python*
- Ian Goodfellow, Yoshua Bengio and Aaron Courville *Deep learning*
- Bishop (1995) *Neural networks for pattern recognition*, Oxford University Press.
- Charles Ollion et Olivier Grisel *Deep learning course* https ://github.com/m2dsupsdlclass/lectures-labs
- Jeremy Jordan : Variational autoencoders, https ://www.jeremyjordan.me/
- Ali Ghodsi, Lec : Deep Learning, Variational Autoencoder, Oct 12 2017 [Lect 6.2]