Institut de Mathématiques de Toulouse, INSA Toulouse

Neural networks, Introduction to deep learning

ML Training- CERFACS
October, 2022

Béatrice Laurent-Brendan Guillouet

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
- 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.

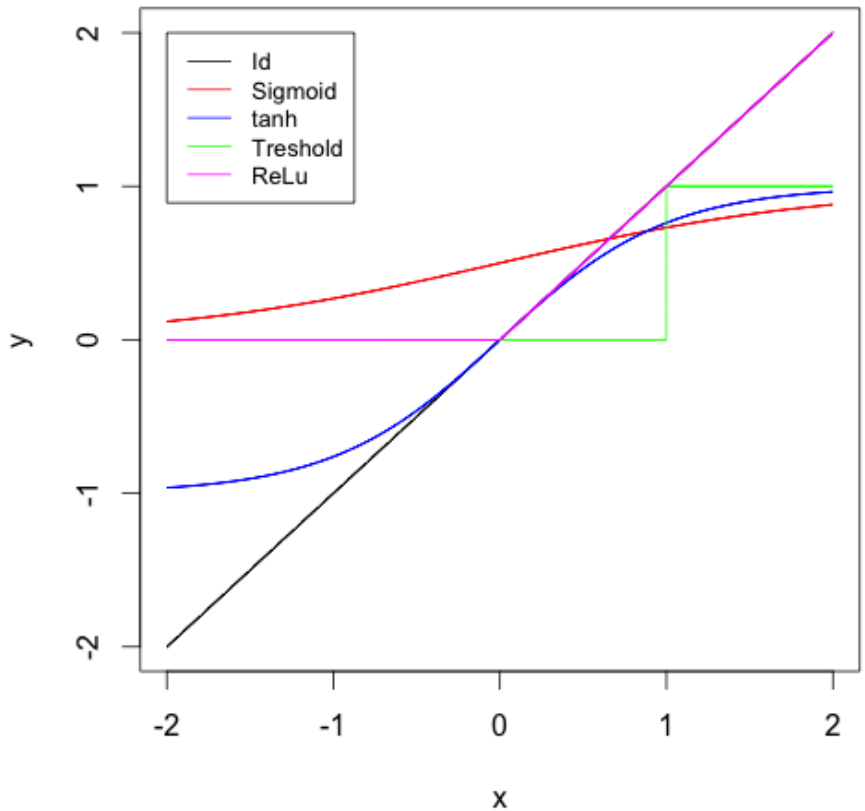
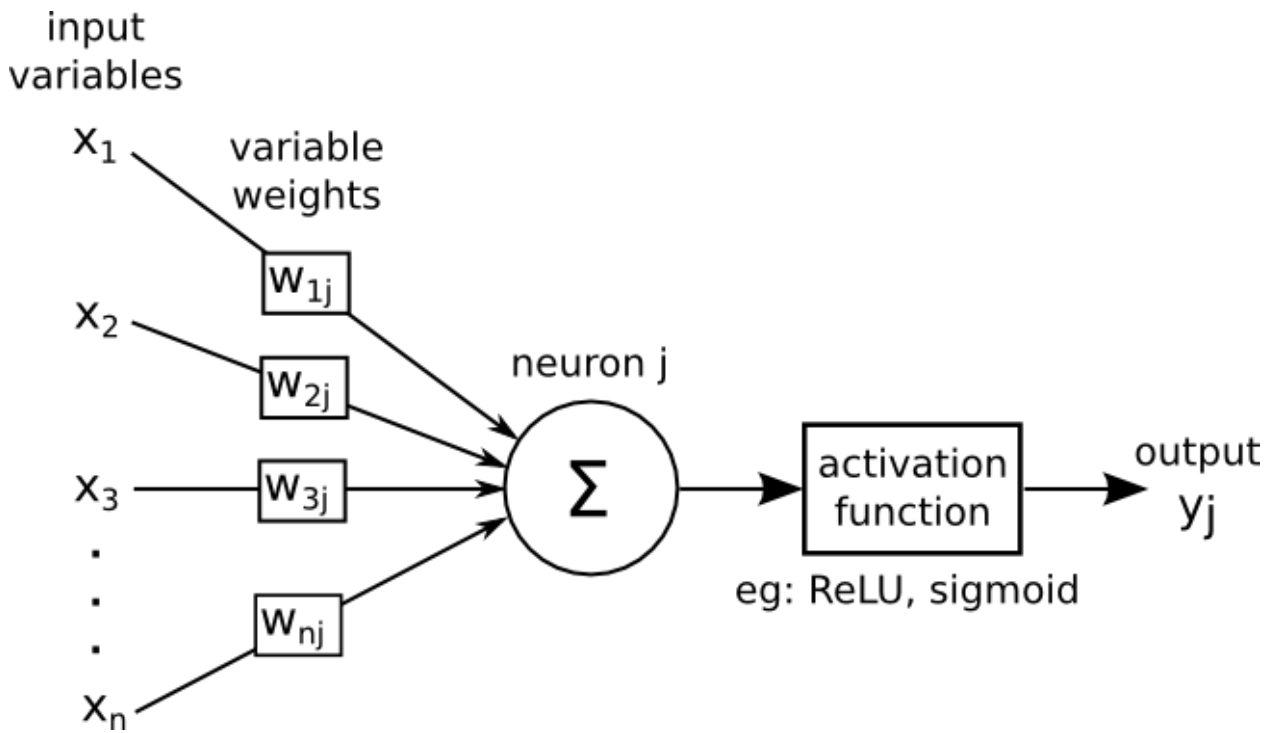


Figure – Activation functions

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.

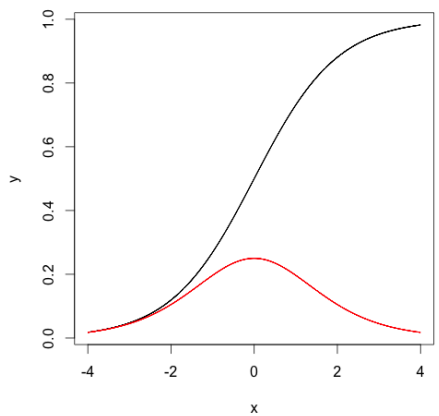


Figure – Sigmoid function (in black) and its derivatives (in red)

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
- 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.

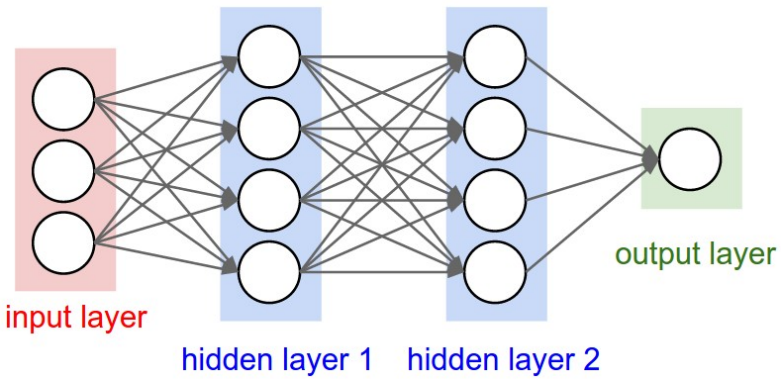


Figure – A basic neural network.

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.

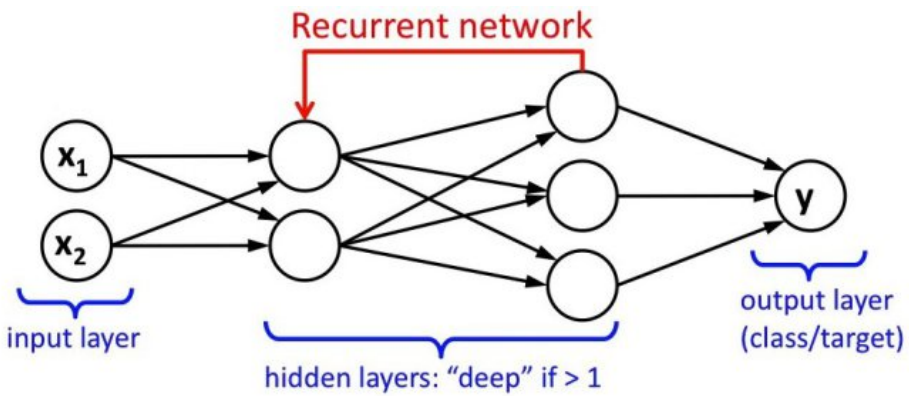


Figure – A recurrent neural network

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
- 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 $f(x,\theta) = P_\theta(Y = 1/X = x)$,

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

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
- 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 equal to

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

with

$$\ell(f(x,\theta),y)=\|y-f(x,\theta)\|^2=\sum_{k=1}^{K}(y_k-f_k(x,\theta))^2.$$

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 $k - 1$.

Using the previous notations, we want to compute the gradients

$$\text{Output weights}\frac{\partial \ell(f(x,\theta),y)}{\partial W_{i,j}^{(L+1)}} \quad \text{Output biases}\frac{\partial \ell(f(x,\theta),y)}{\partial b_i^{(L+1)}}$$

$$\text{Hidden weights}\frac{\partial \ell(f(x,\theta),y)}{\partial W_{i,j}^{(k)}} \quad \text{Hidden biases}\frac{\partial \ell(f(x,\theta),y)}{\partial b_i^{(k)}}$$

for $1 \leq k \leq L$.

We use the chain-rule : if $z(x) = \phi(a_1(x), \ldots, a_J(x))$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial a_j}\frac{\partial a_j}{\partial x_i} = \langle \nabla \phi, \frac{\partial \boldsymbol{a}}{\partial x_i} \rangle.$$

Hence, using $f(x,\theta) = \psi(a^{(L+1)}(x))$, and recalling

$$\ell(f(x,\theta),y) = \sum_{k=1}^{K}(y_k - \psi(a_k^{(L+1)}(x)))^2.$$

we have

Backpropagation algorithm for regression

$$\frac{\partial \ell(f(x,\theta),y)}{\partial a_k^{(L+1)}(x)} = -2(y_k - \psi(a_k^{(L+1)}(x)))\psi'(a_k^{(L+1)}(x))$$

$$\frac{\partial \ell(f(x,\theta),y)}{\partial b_i^{(L+1)}} = \sum_{k=1}^{K} \frac{\partial \ell(f(x,\theta),y)}{\partial a_k^{(L+1)}(x)} \frac{\partial a_k^{(L+1)}(x)}{\partial b_i^{(L+1)}} = \frac{\partial \ell(f(x,\theta),y)}{\partial a_i^{(L+1)}(x)}$$

Hence,

$$\begin{aligned} \nabla_{b^{(L+1)}} \ell(f(x,\theta),y) &= \nabla_{a^{(L+1)}(x)} \ell(f(x,\theta),y) \\ \nabla_{a^{(L+1)}(x)} \ell(f(x,\theta),y) &= -2(y - \psi(a^{(L+1)}(x)) \odot \psi'(a^{(L+1)}(x)), \end{aligned}$$

where $\odot$ denotes the element-wise product.

Backpropagation algorithm for regression

Let us now compute the partial derivative of the loss function with respect to the output weights.

$$\frac{\partial \ell(f(x), y)}{\partial W_{i,j}^{(L+1)}} = \sum_k \frac{\partial \ell(f(x), y)}{\partial a_k^{(L+1)}(x)} \frac{\partial a_k^{(L+1)}(x)}{\partial W_{i,j}^{(L+1)}}$$

and

$$\frac{\partial a_k^{(L+1)}(x)}{\partial W_{i,j}^{(L+1)}} = (h^{(L)}(x))_j \mathbb{1}_{i=k}.$$

Hence

$$\nabla_{W^{(L+1)}} \ell(f(x), y) = \nabla_{a^{(L+1)}(x)} \ell(f(x, \theta), y)(h^{(L)}(x))^T.$$

Backpropagation equations

- By similar computations, we prove that for all hidden layer $k \in \{1, \ldots, L\}$,

$$\begin{aligned} \nabla_{h^{(k)}(x)} \ell(f(x), y) &= (W^{(k+1)})^T \nabla_{a^{(k+1)}(x)} \ell(f(x), y) \\ \nabla_{a^{(k)}(x)} \ell(f(x), y) &= \nabla_{h^{(k)}(x)} \ell(f(x), y) \odot \phi'(a^{(k)}(x)) \end{aligned}$$

and

$$\begin{aligned} \nabla_{W^{(k)}} \ell(f(x), y) &= \nabla_{a^{(k)}(x)} \ell(f(x), y)(h^{(k-1)}(x))^T \\ \nabla_{b^{(k)}} \ell(f(x), y) &= \nabla_{a^{(k)}(x)} \ell(f(x), y) \end{aligned}$$

Backpropagation algorithm

We can now summarize the backpropagation algorithm.
We use the Backpropagation equations to compute the gradient by a two pass algorithm.

- **Forward pass :**
  - We fix the value of the current weights $\theta^{(r)} = (W^{(1,r)}, b^{(1,r)}, \ldots, W^{(L+1,r)}, b^{(L+1,r)})$,
  - 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.

Backpropagation algorithm

- **Backward pass :** For $(x, y) = (X_i, Y_i)$, i in the batch B,
  - Compute the output gradient
    $\nabla_{a^{(L+1)}(x)} \ell(f(x), y) = -2(y - \psi(a^{(L+1)}(x)) \odot \psi'(a^{(L+1)}(x))$.
  - For $k = L + 1$ to 1
    - Compute the gradient at the hidden layer k

$$\begin{aligned} \nabla_{W^{(k)}} \ell(f(x), y) &= \nabla_{a^{(k)}(x)} \ell(f(x), y)(h^{(k-1)}(x))^T \\ \nabla_{b^{(k)}} \ell(f(x), y) &= \nabla_{a^{(k)}(x)} \ell(f(x), y) \end{aligned}$$

    - Compute the gradient at the previous layer

$$\nabla_{h^{(k-1)}(x)} \ell(f(x), y) = (W^{(k)})^T \nabla_{a^{(k)}(x)} \ell(f(x), y)$$

      and

$$\nabla_{a^{(k-1)}(x)} \ell(f(x), y) = \nabla_{h^{(k-1)}(x)} \ell(f(x), y) \odot \phi'(a^{(k-1)}(x))$$

Outline

- Neural Networks
- Multilayer perceptrons
- Estimation of the parameters
- Backpropagation algorithms
- Optimization algorithms
- Convolutional Neural Networks
- 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

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

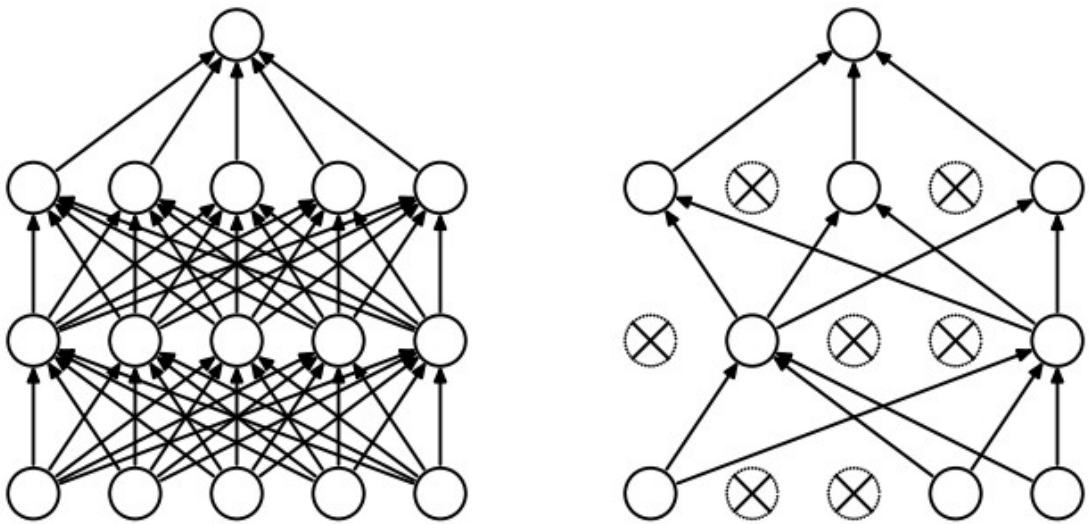
    where $\varepsilon_r > 0$ is the learning rate. that satisfies $\varepsilon_r \to 0$, $\sum_r \varepsilon_r = \infty$, $\sum_r \varepsilon_r^2 < \infty$, for example $\varepsilon_r = 1/r$.

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
- 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 ..

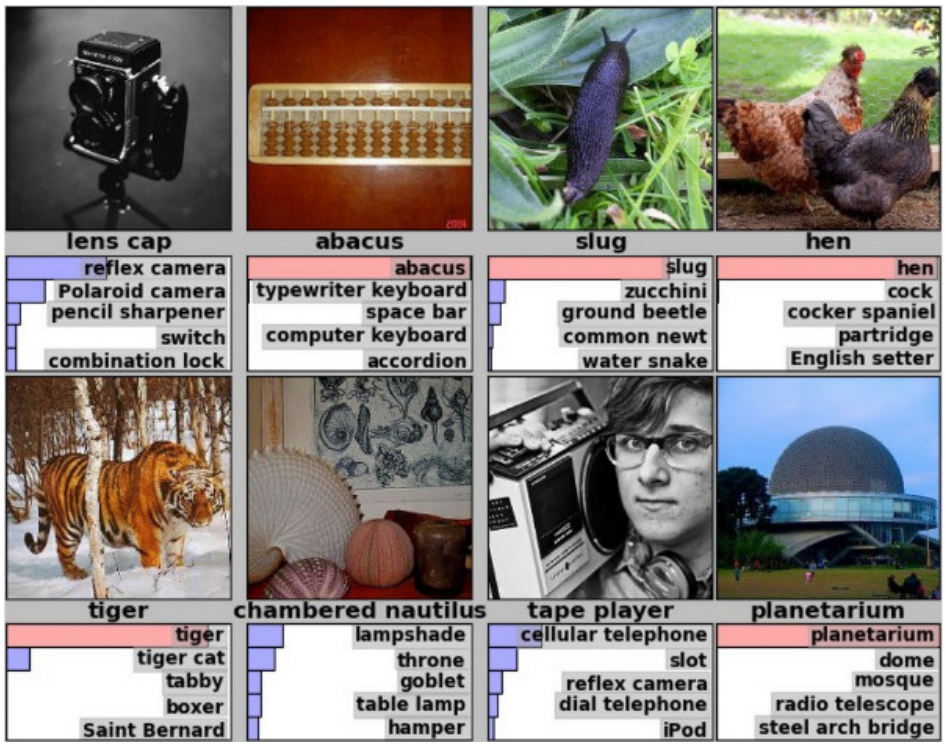


Figure – Image annotation

Convolutional neural networks

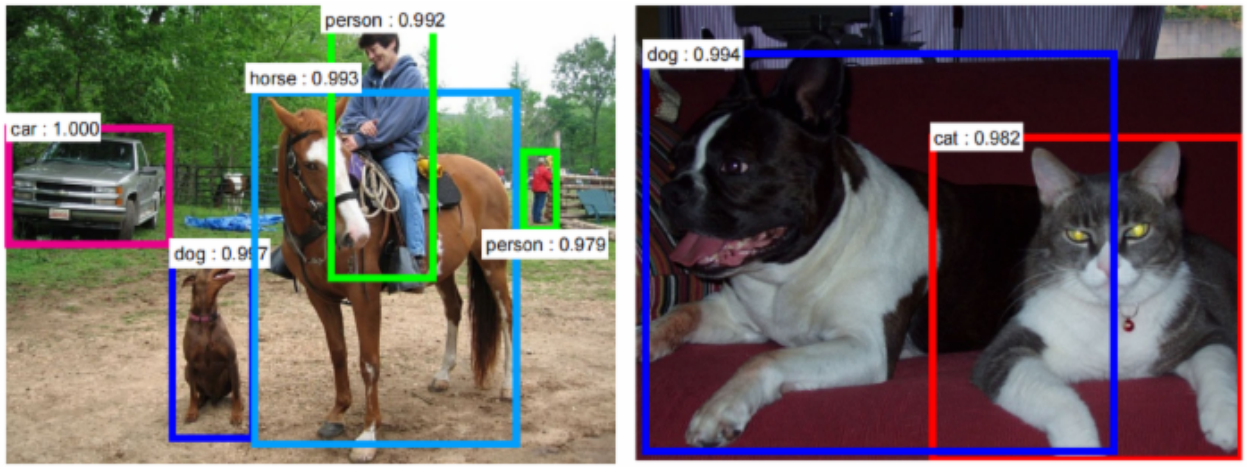
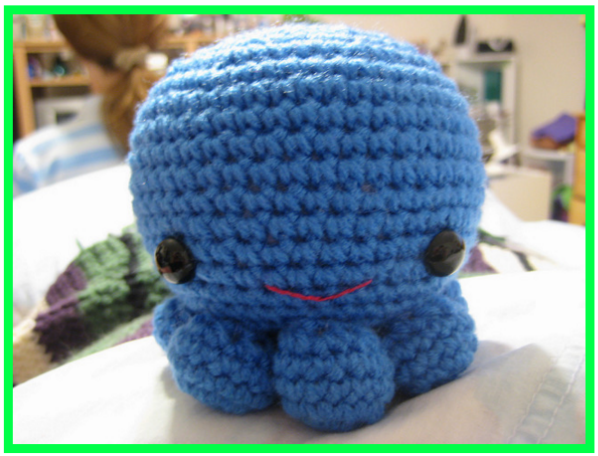


Figure – Image Segmentation.

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.

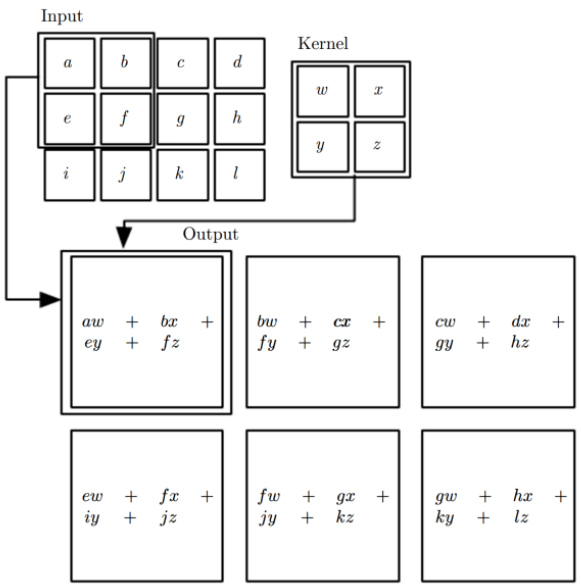


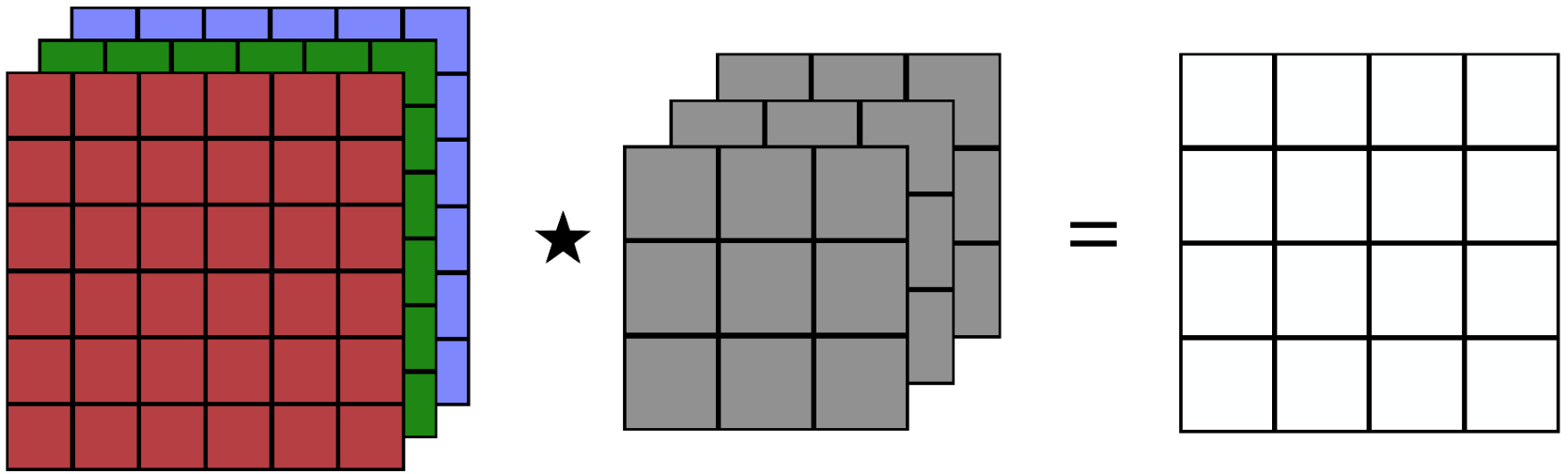
Figure – 2D convolution

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.

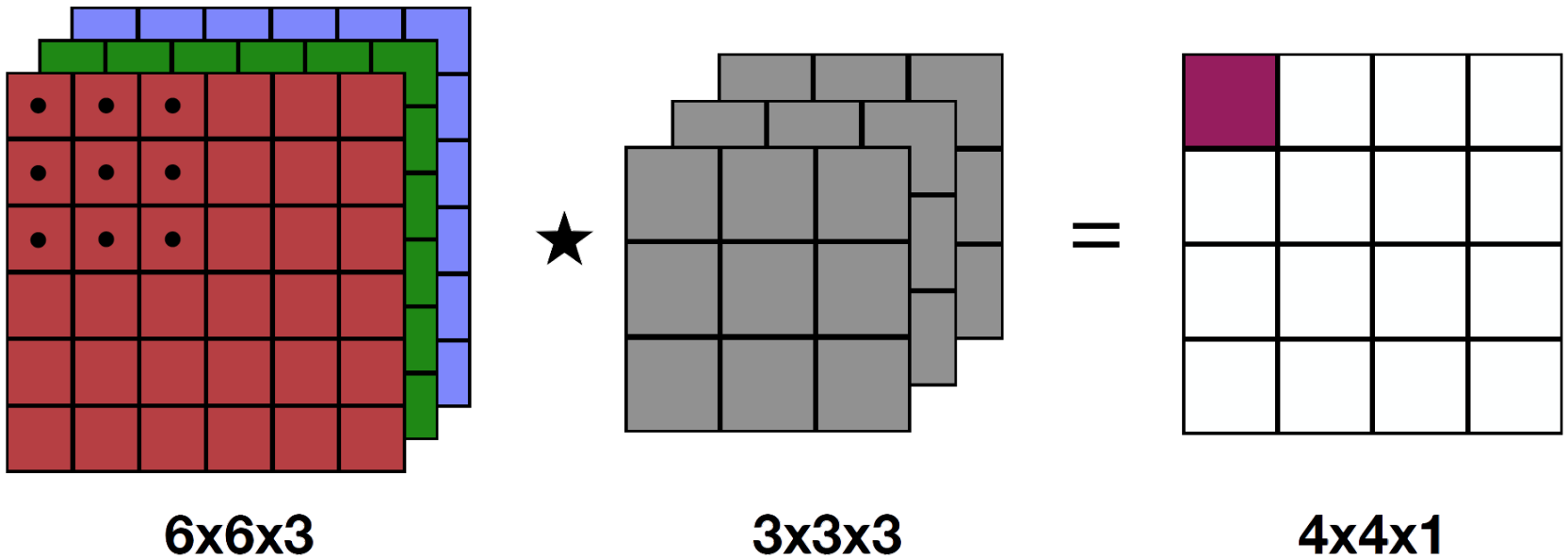
Convolution - channels

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



Convolution - channels

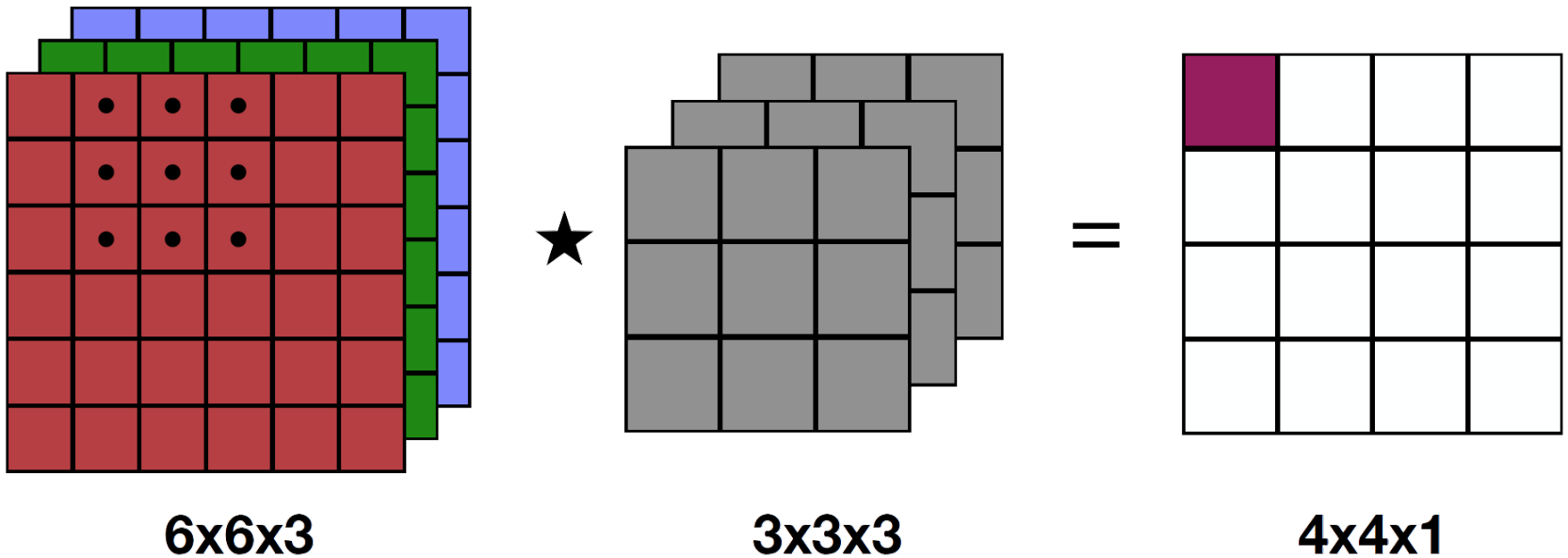
Colored image in 3 dimensions : (height, width, channels (RGB))



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

Convolution - channels

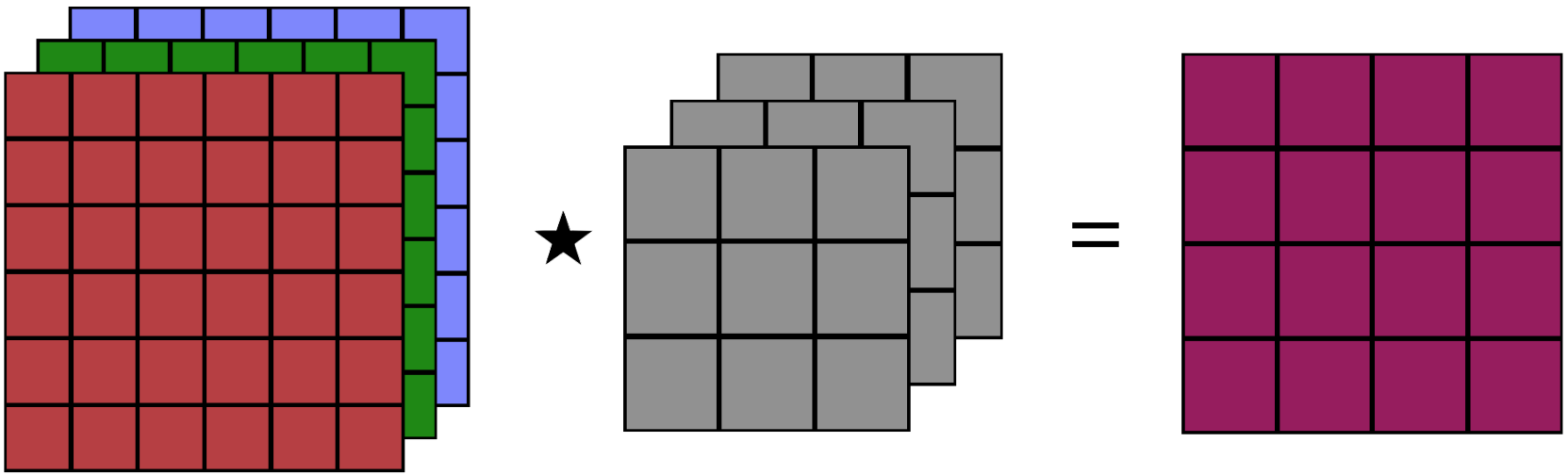
Colored image in 3 dimensions : (height, width, channels (RGB))



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

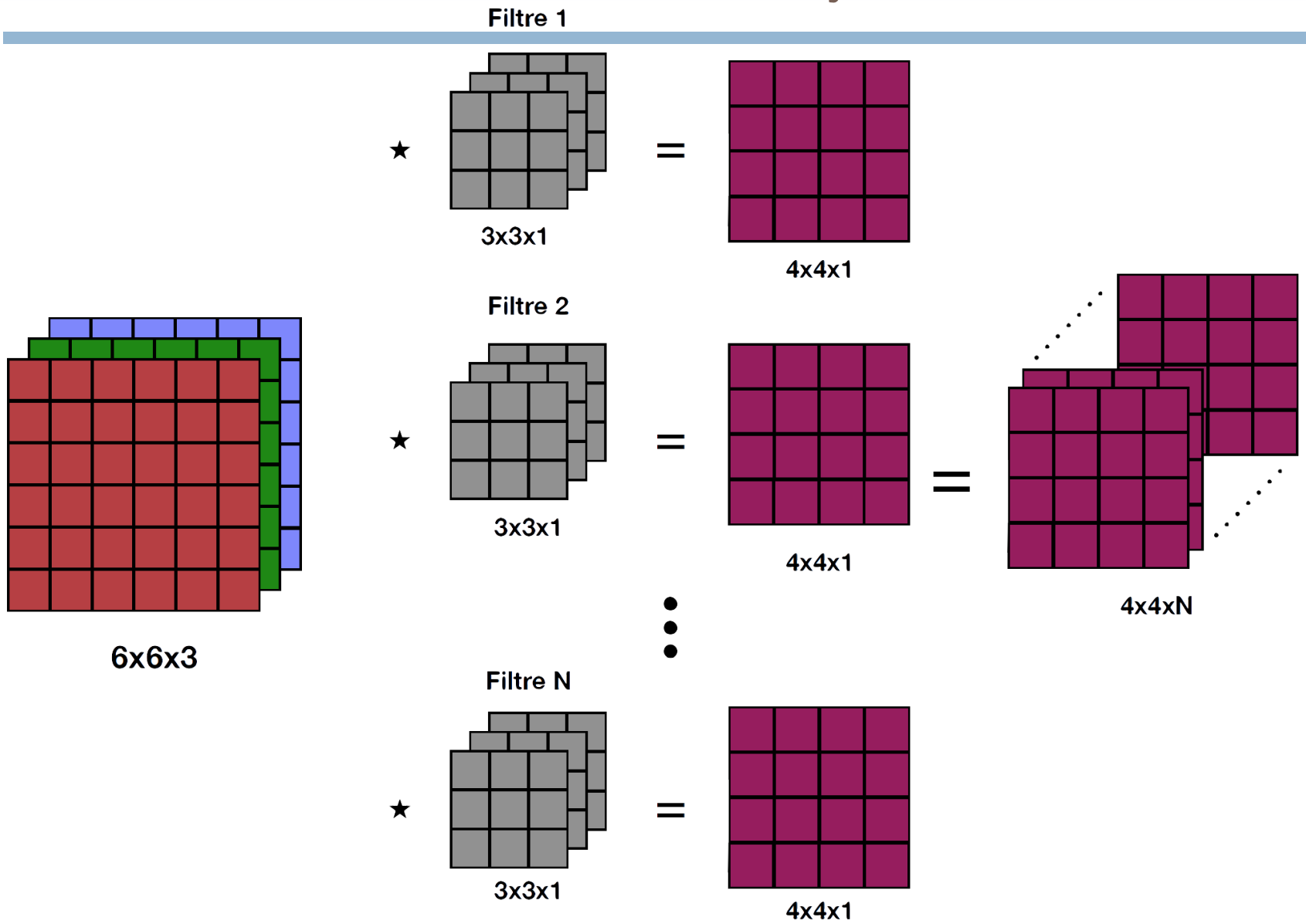
Convolution - channels

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

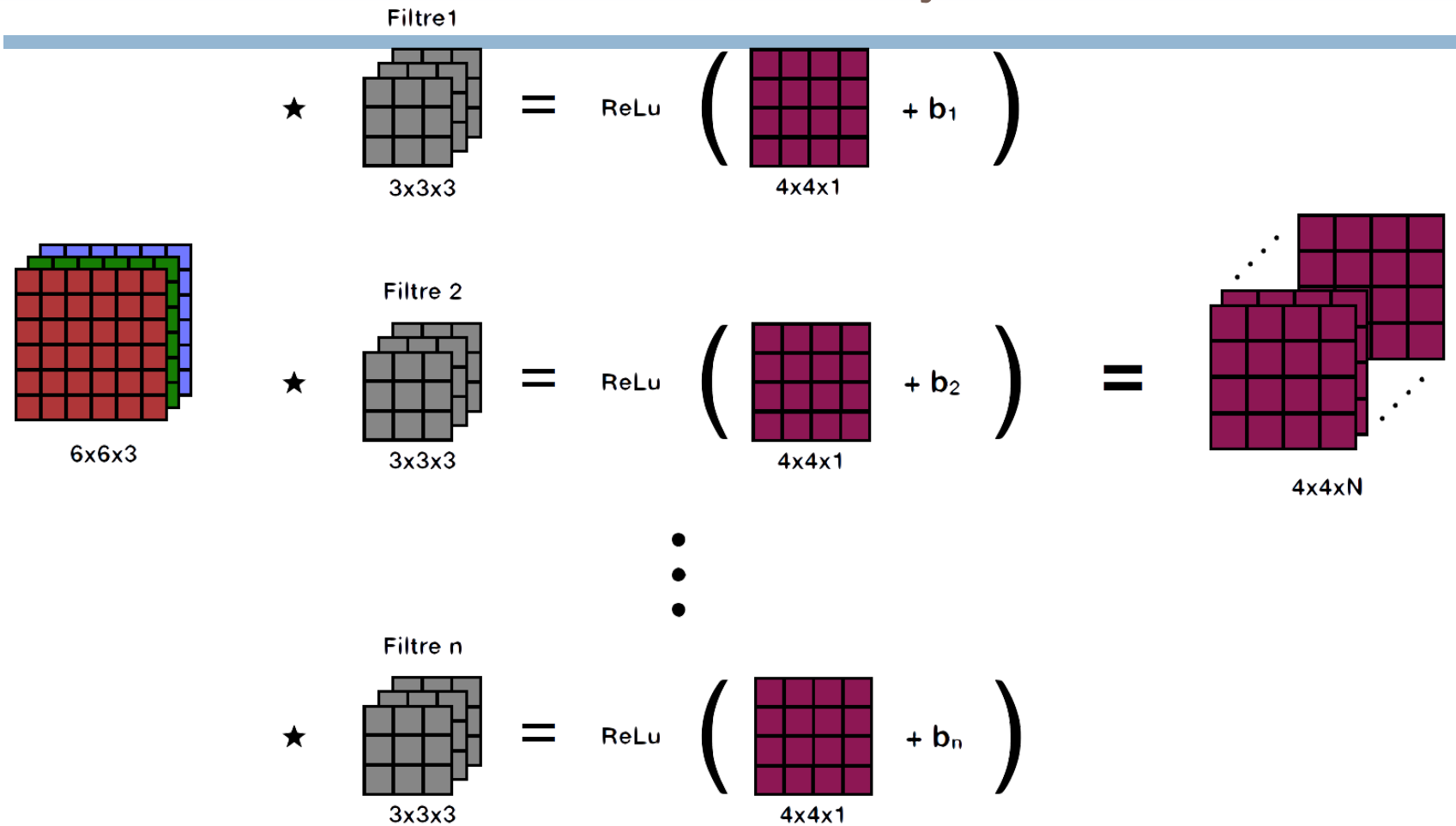


$$(F \star img)(x, y) = \sum_{c} \sum_{m} \sum_{n} F^{c}(n, m) \cdot Img^{c}(x + m, y + n)$$

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.

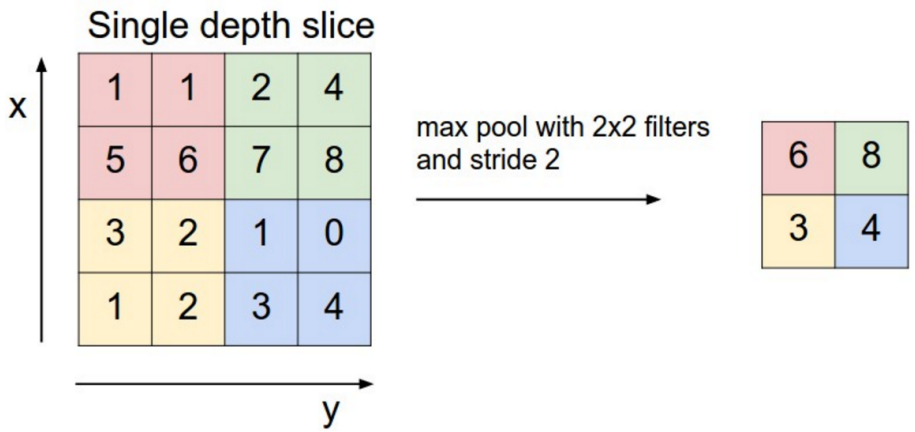


Figure – Maxpooling and effect on the dimension

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.

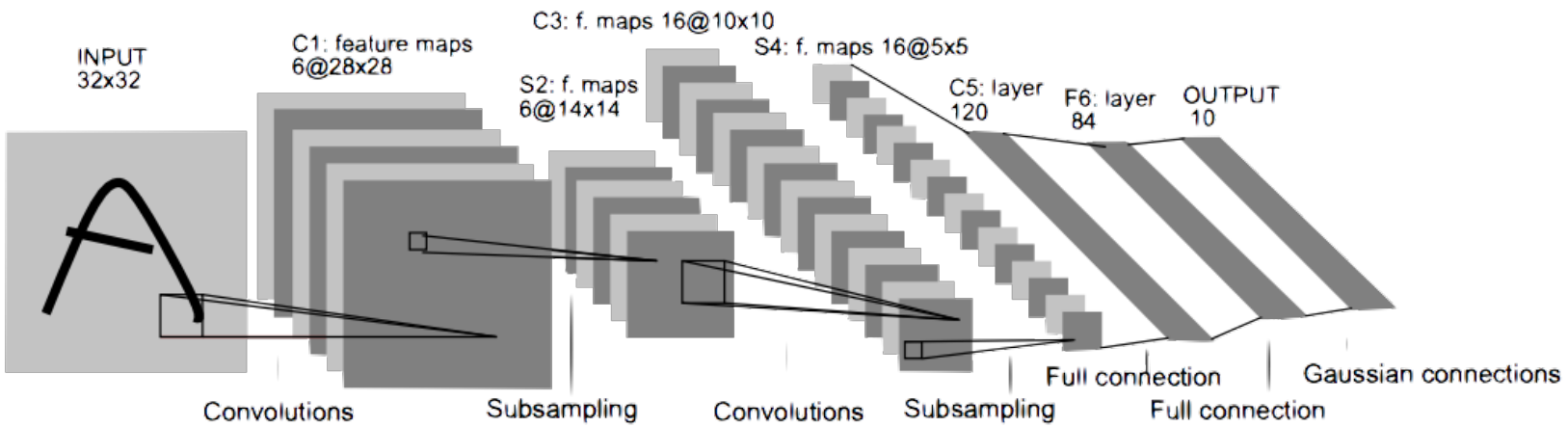


Figure – Architecture of the network Le Net. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998)

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))

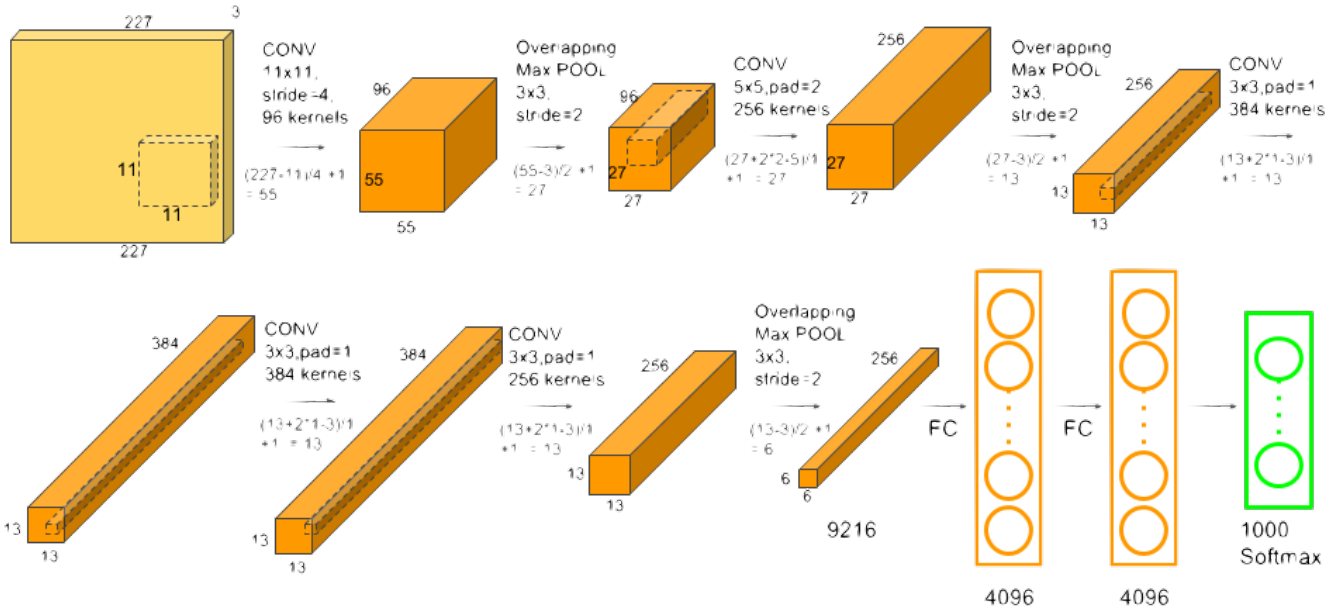
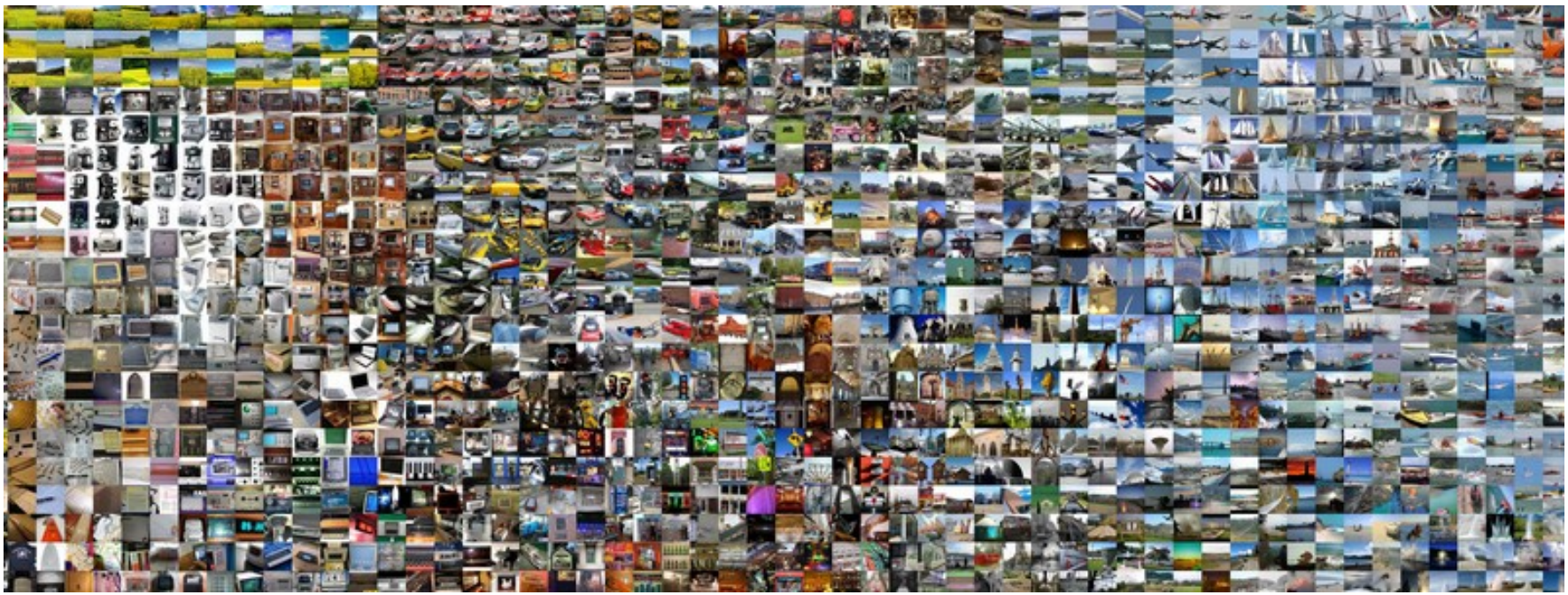


Image source : https://www.learnopencv.com/understanding-alexnet/

- $\simeq 60M$ 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 $= 2$

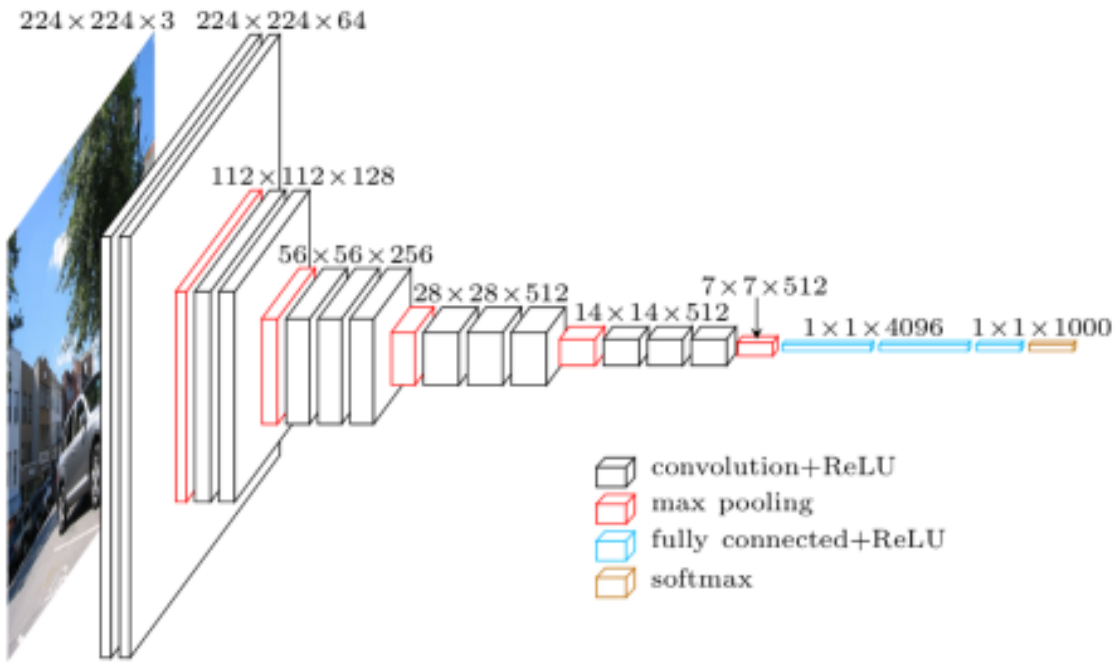
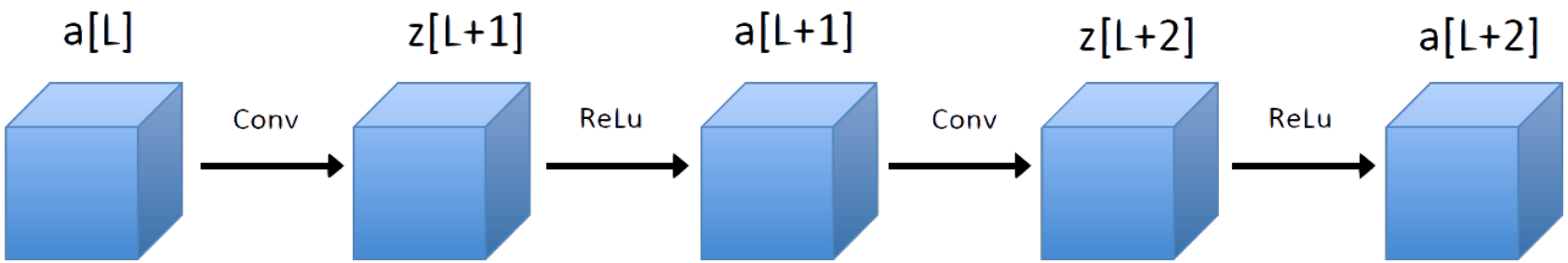


Figure – Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition (2014). 138M parameters.

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

ResNet Block - He et al. (2016)

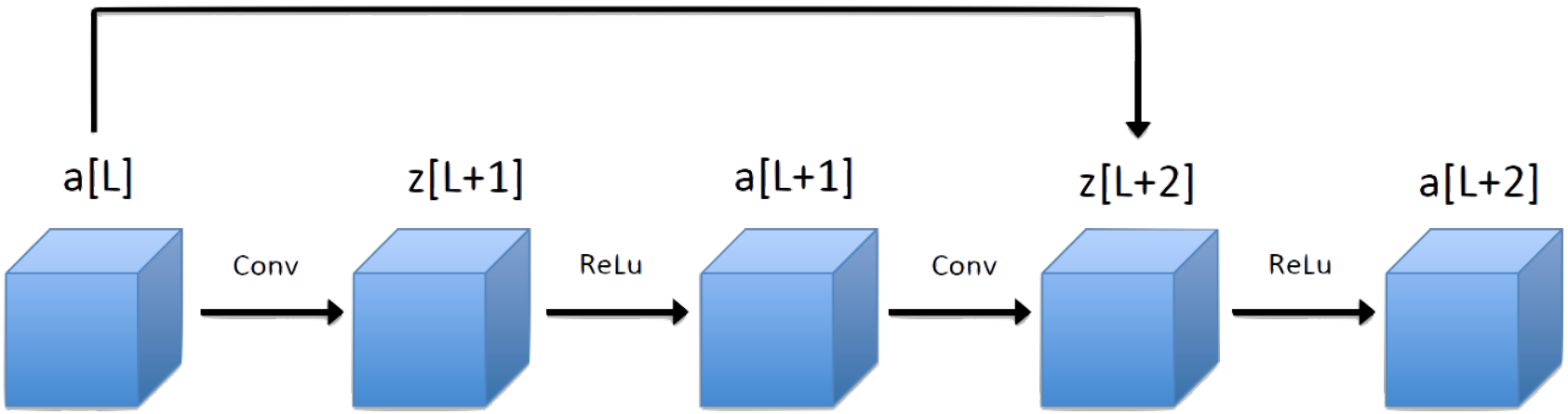
Convolution with padding = SAME



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

ResNet Block - He et al. (2016)

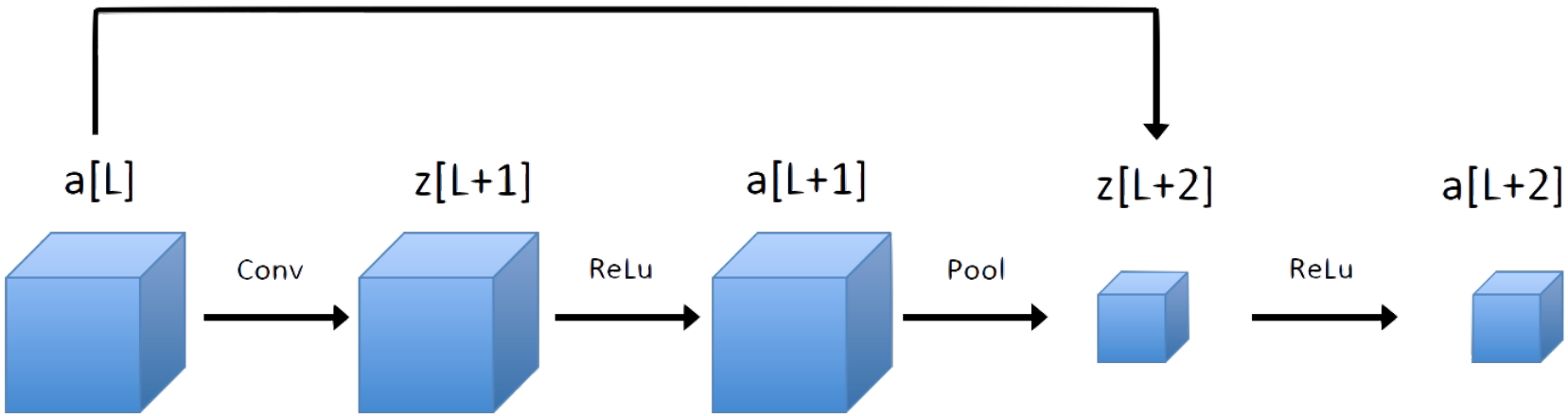
Convolution with padding = SAME



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

ResNet Block - He et al. (2016)

MaxPooling



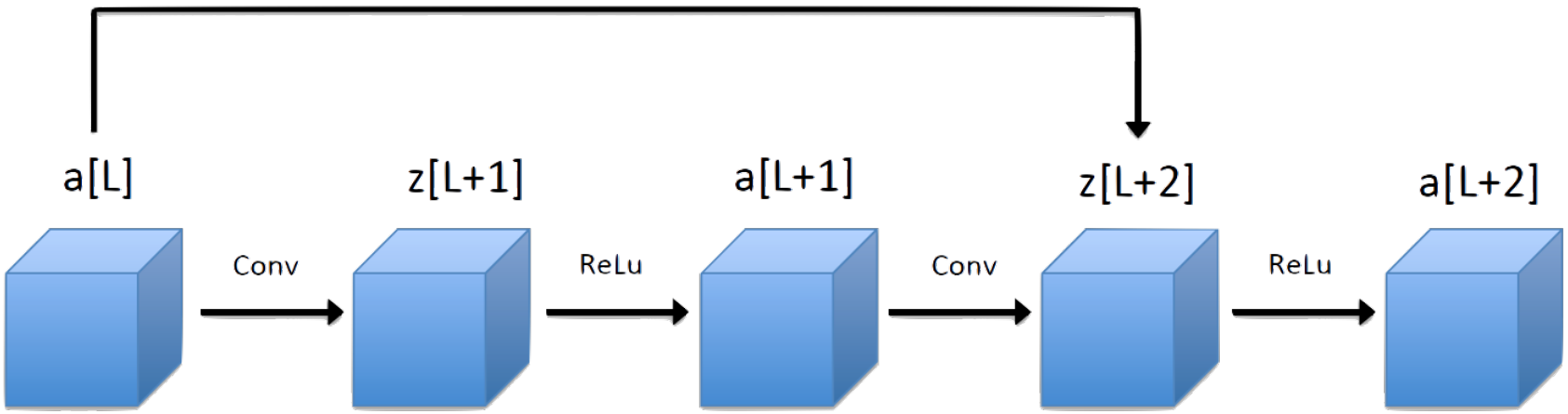
$$z[l+1] = Conv(a[l])$$
$$a[l+1] = ReLu(z[l+1])$$
$$z[l+2] = MaxPool(a[l+1])$$
$$a[l+2] = ReLu(z[l+2] + W \cdot a[l])$$

ResNet Block - Properties

Making a convolutional network too deep can hurt its performance.

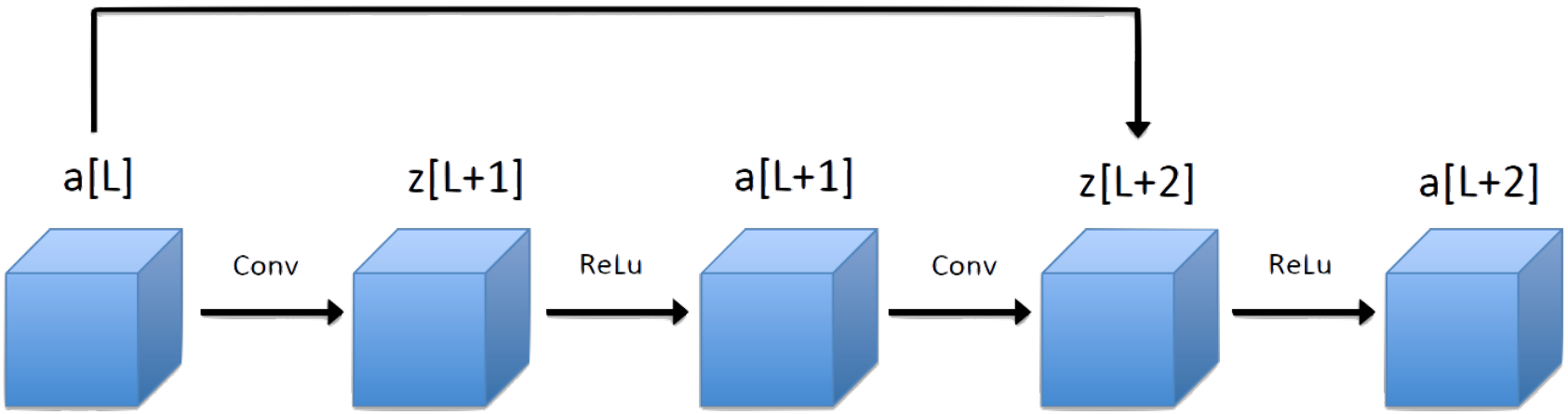
ResNet Block - Properties

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

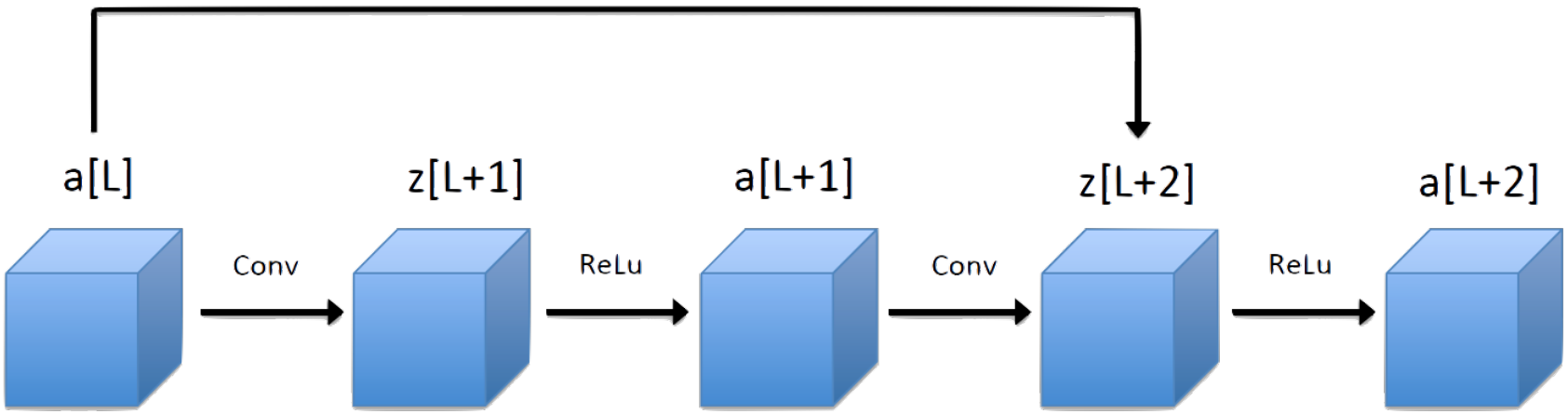
Making a convolutional network too deep can hurt its performance.
With Resnet, you can make your network deeper...
Convolution with padding = SAME



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

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

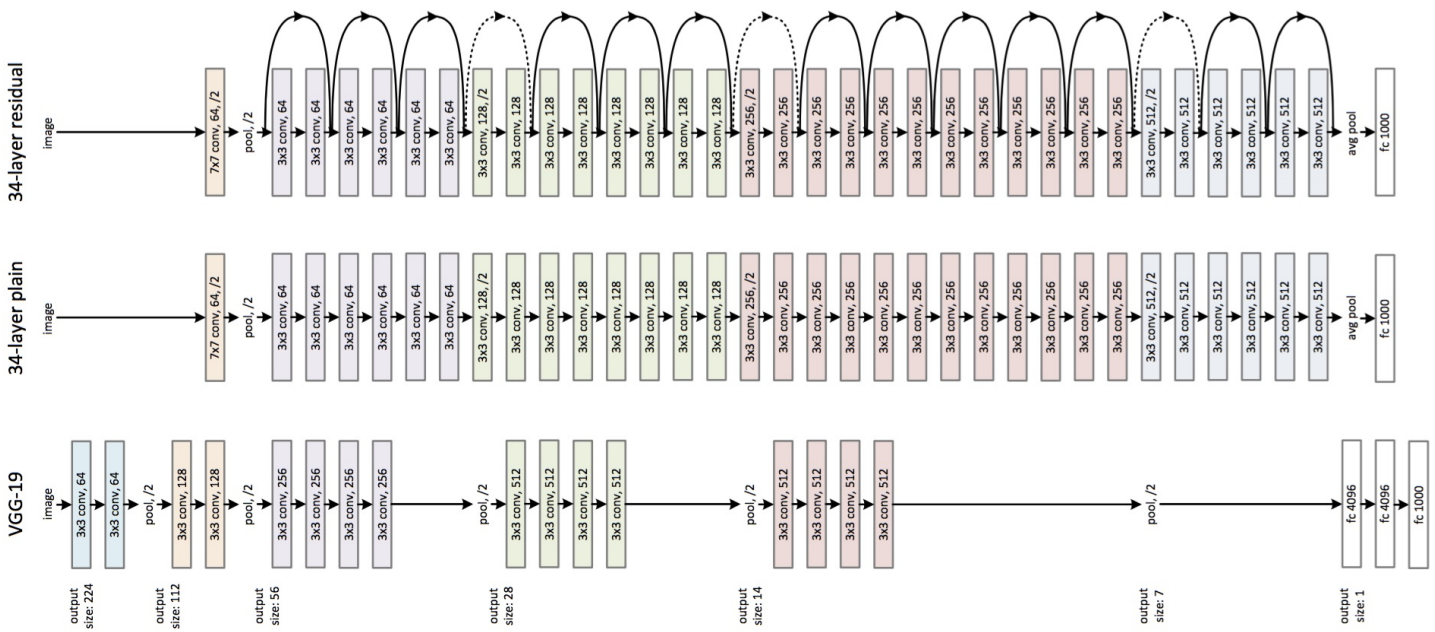
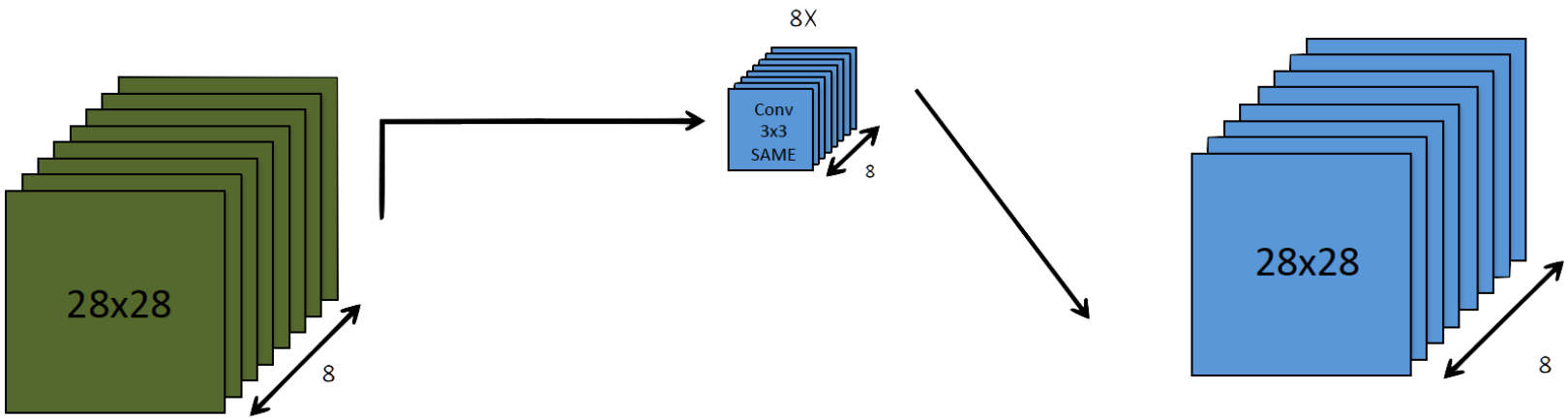


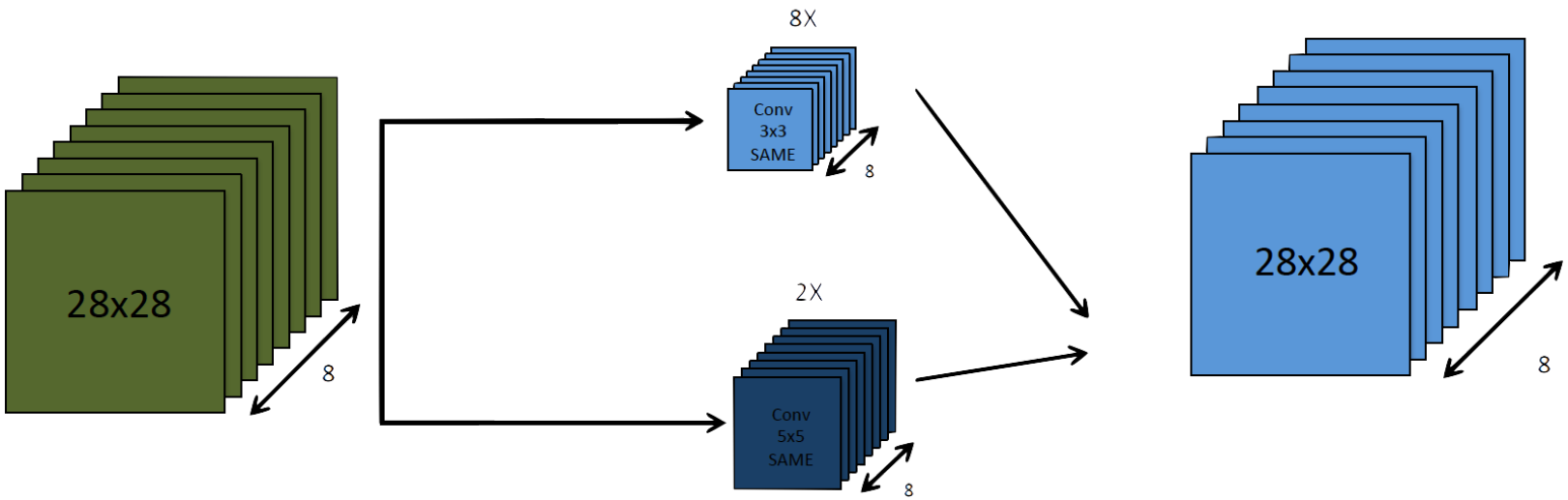
Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase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

Inception Block - Szegedy et al. (2015)



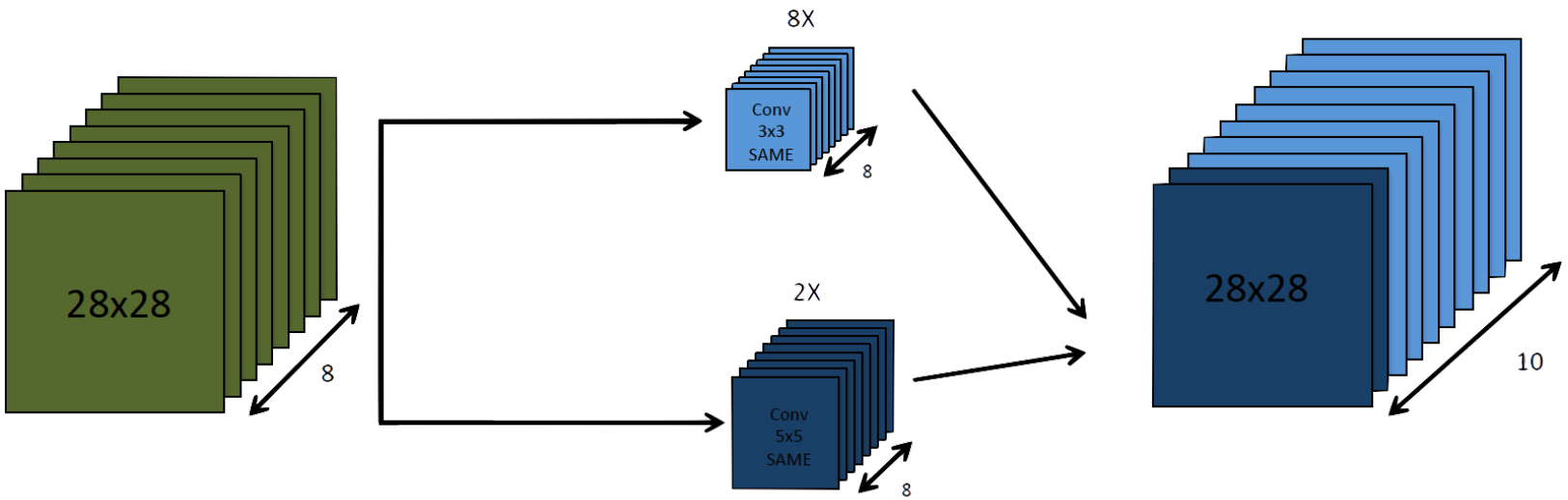
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



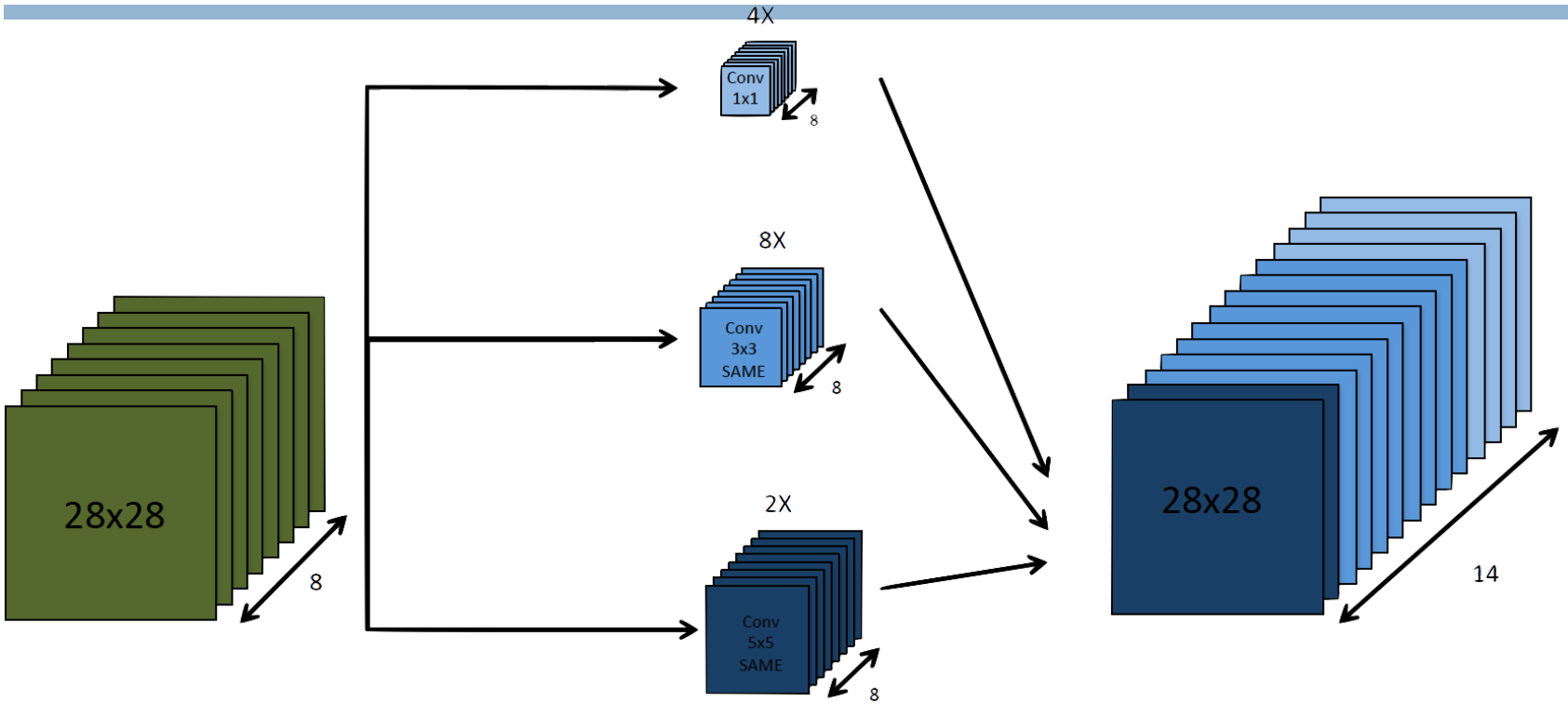
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



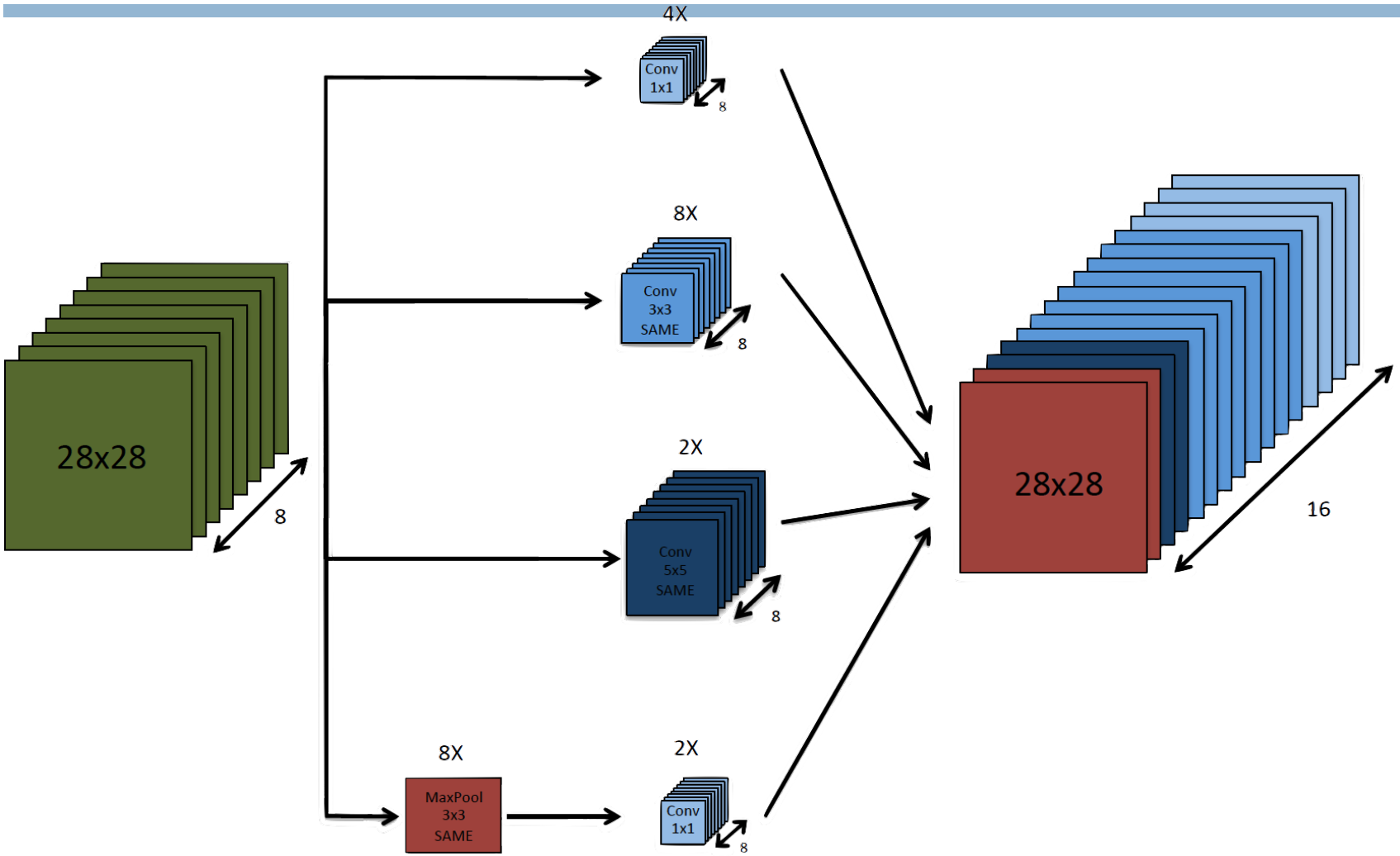
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



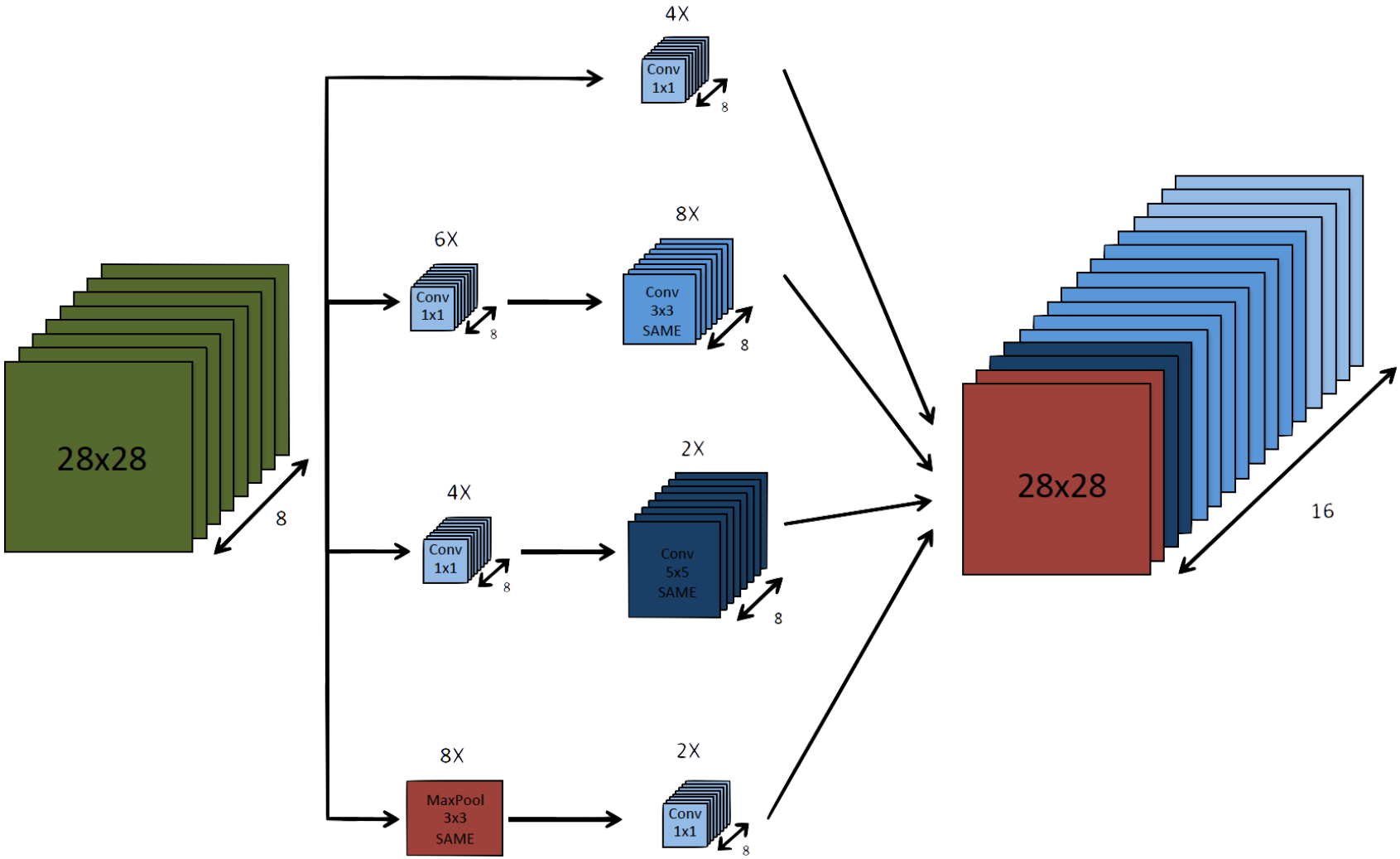
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



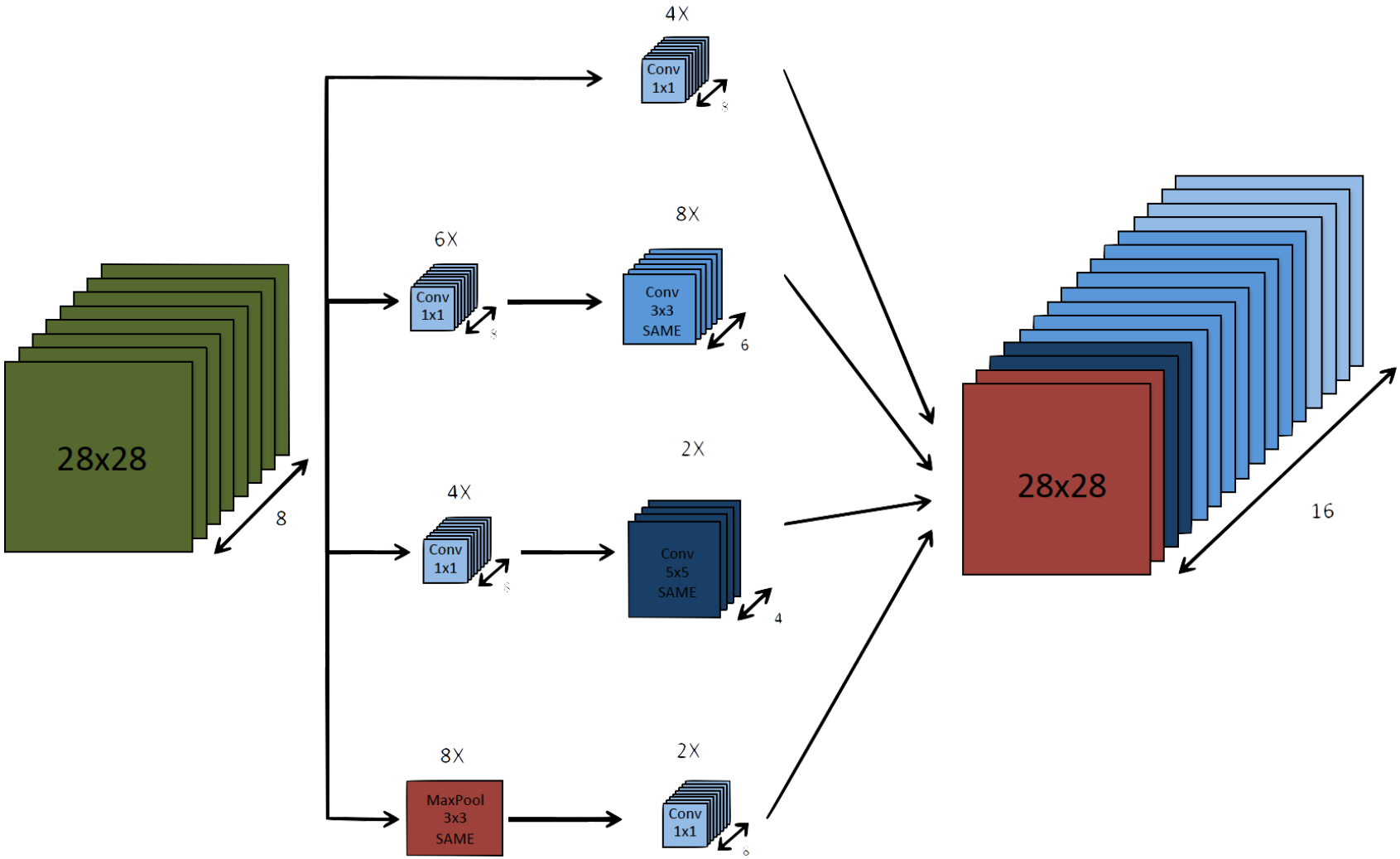
Convolution with 3x3 filters and padding = SAME

Inception Block - Szegedy et al. (2015)



Convolution with 3x3 filters and padding = SAME

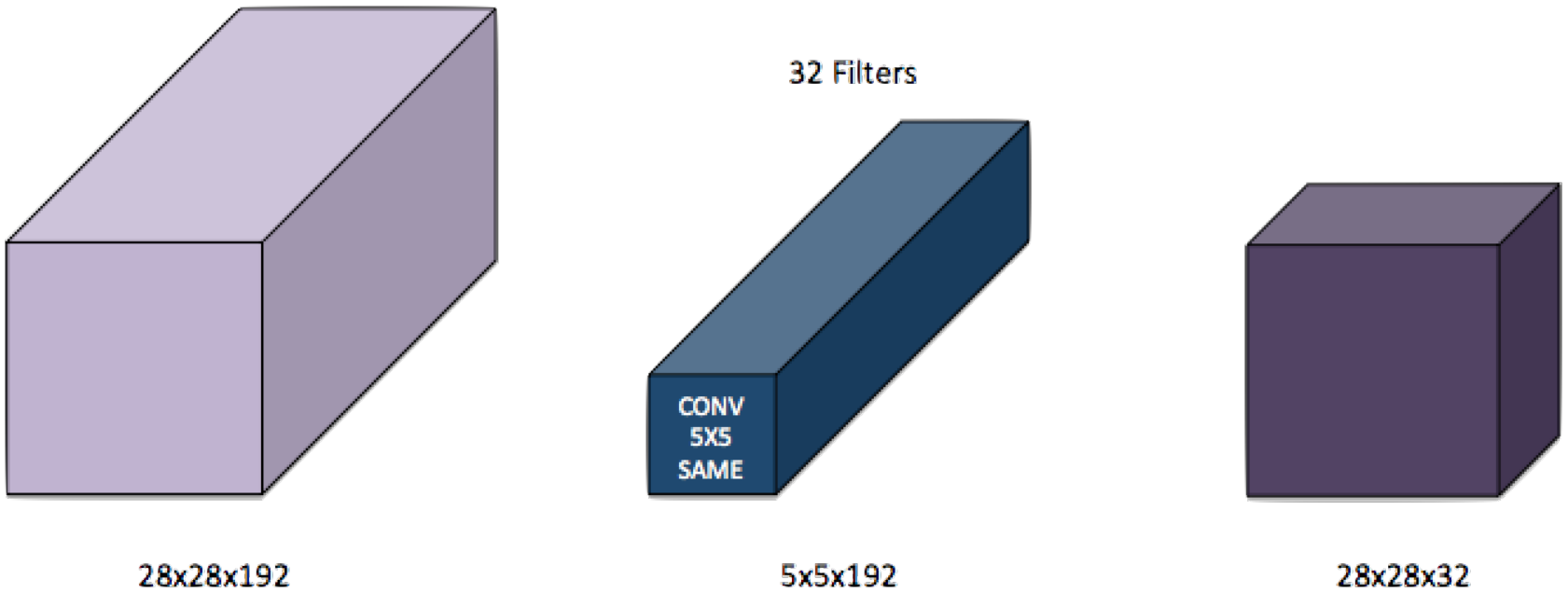
Inception Block - Szegedy et al. (2015)



Convolution with 3x3 filters and padding = SAME

1 x 1 convolution - Problem in computational cost

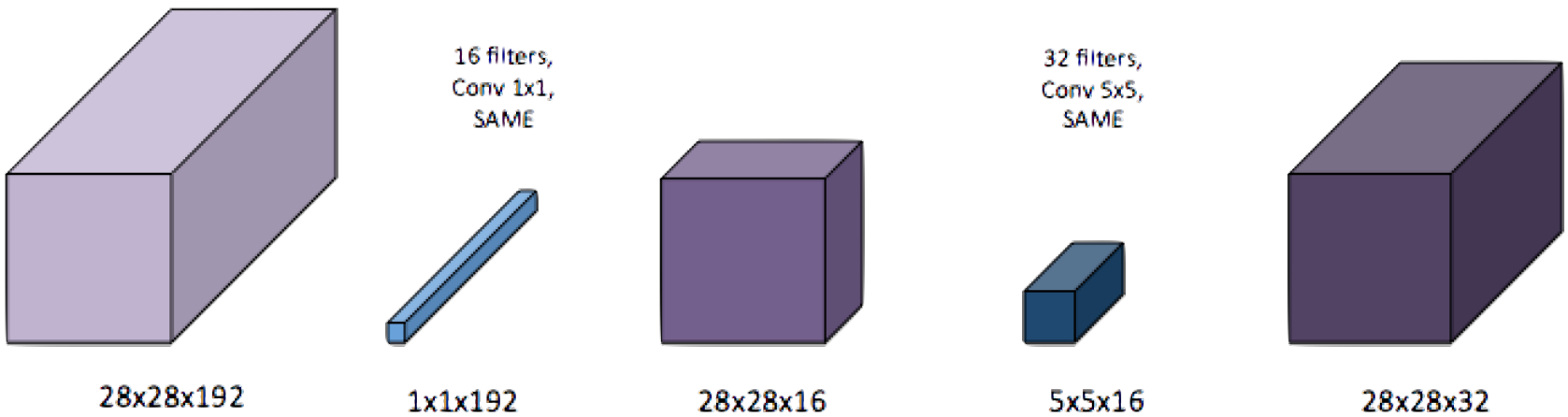
Network in Network (Lin et al. 2013)



- Parameters : 5x5x192x32 = 153.600
- Operations : 153.600 x 28x28 = 120.422.400

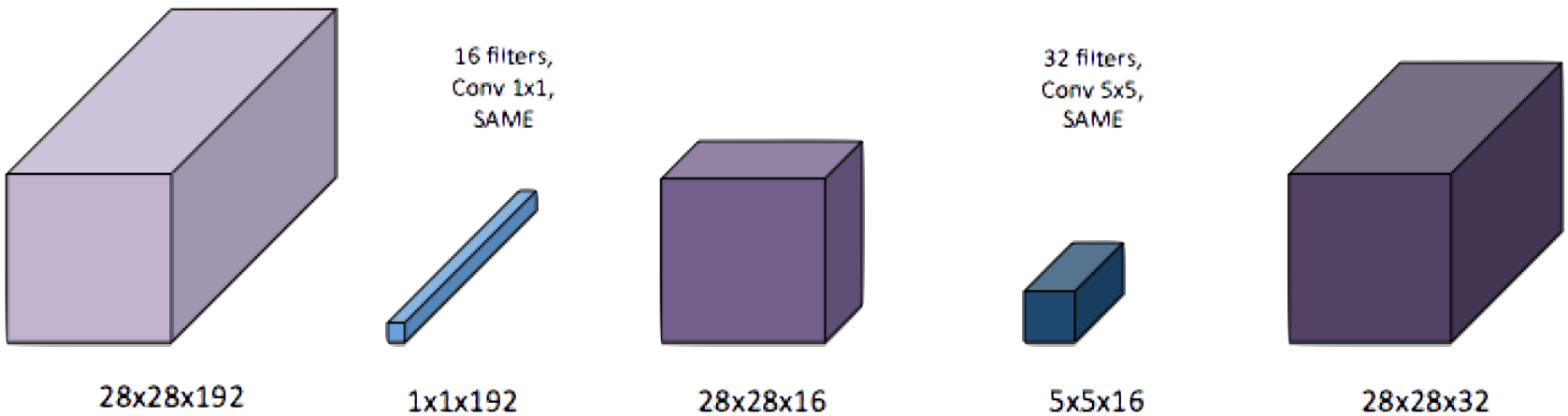
1 x 1 convolution - Problem in computational cost

Network in Network (Lin et al. 2013)



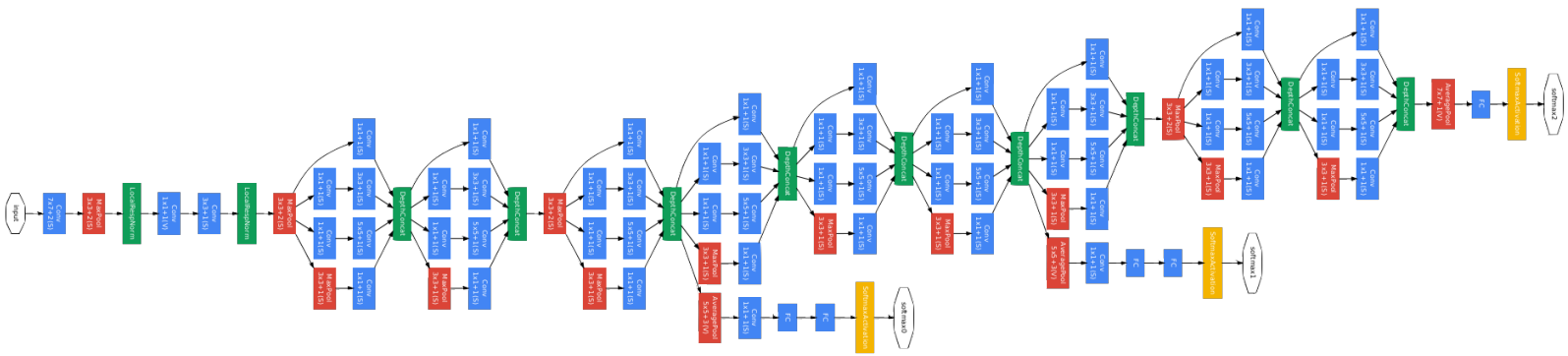
1 x 1 convolution - Problem in computational cost

Network in Network (Lin et al. 2013)



- Parameters : 1x1x192x16, 5x5x16x32 = (3072, 12800)
- Operations : 3.072x28x28 + 12.800 = 2.408.448+ 10.035.200 = 12.443.648

Inception Model - Szegedy et al. (2015)



State of the art

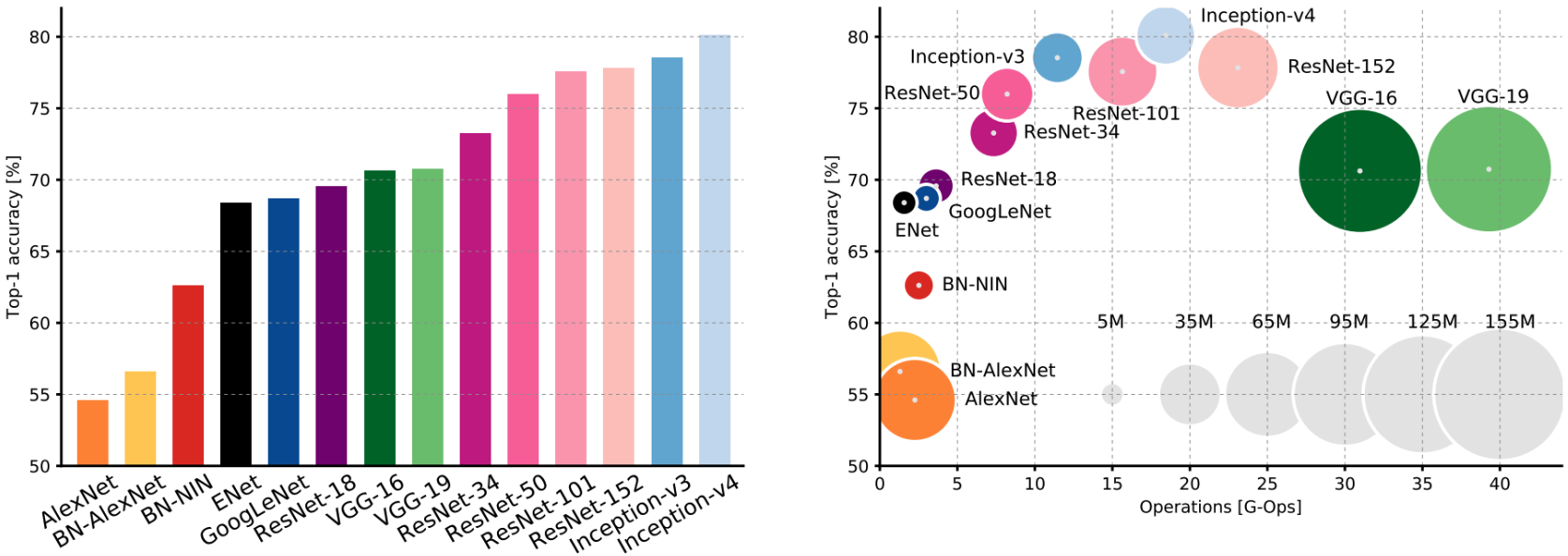


Image source : **canziani2016analysis**

- 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.

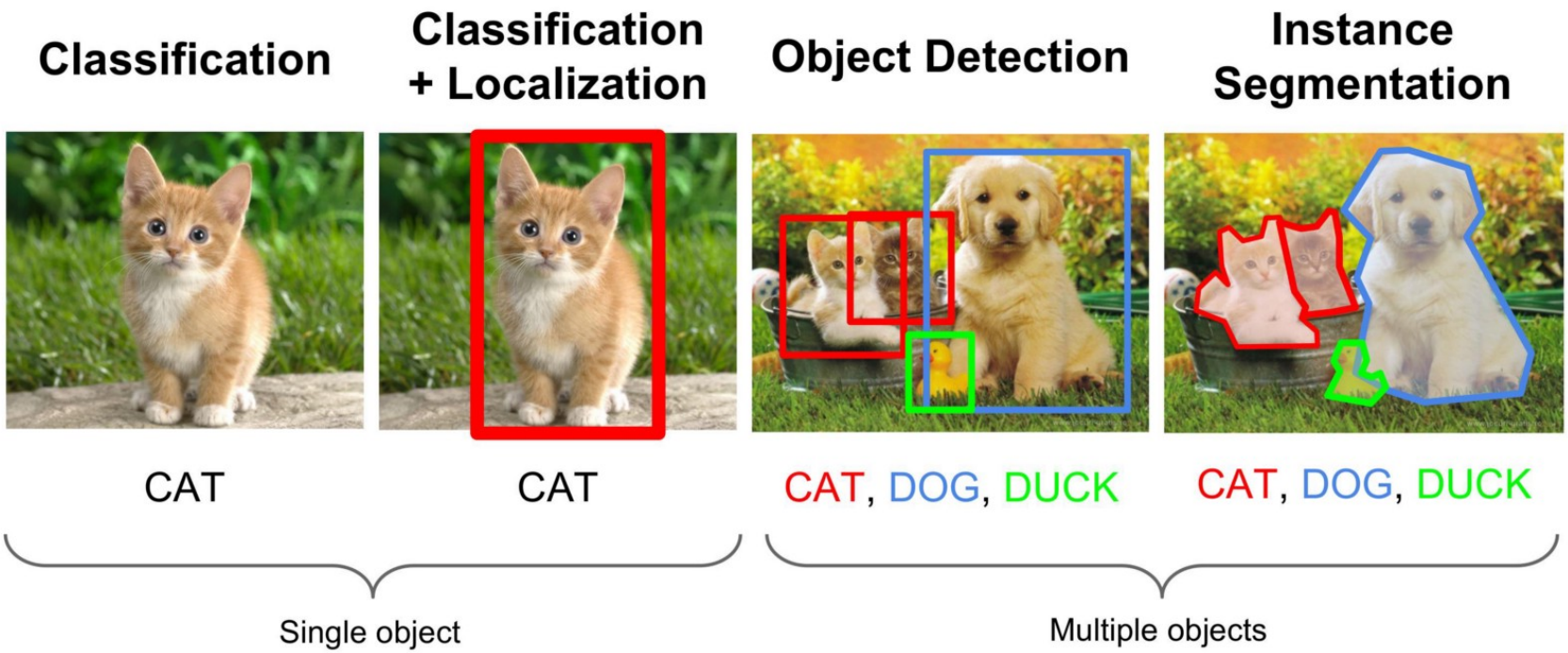
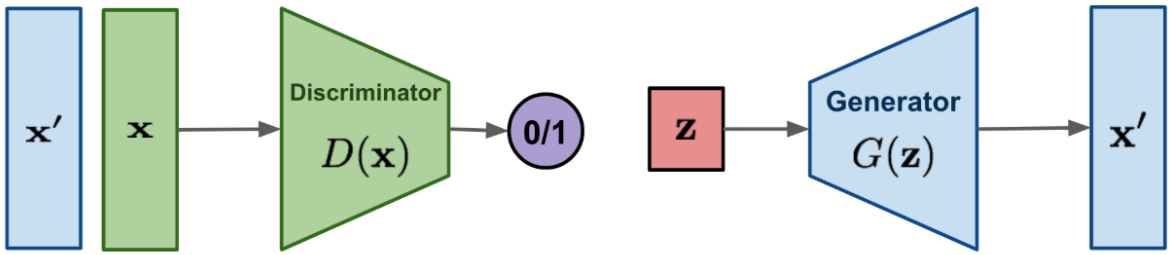
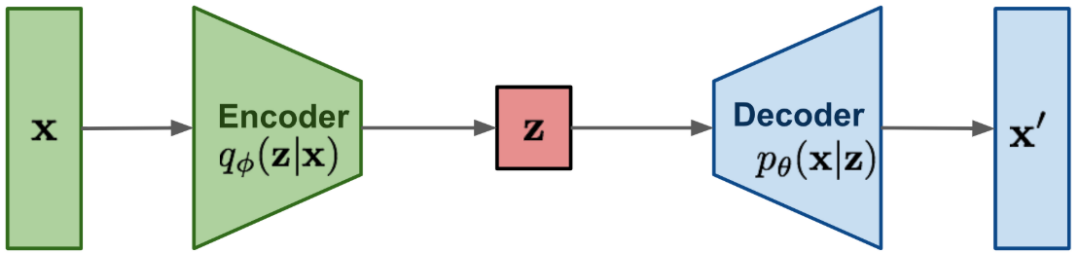


Image Generation

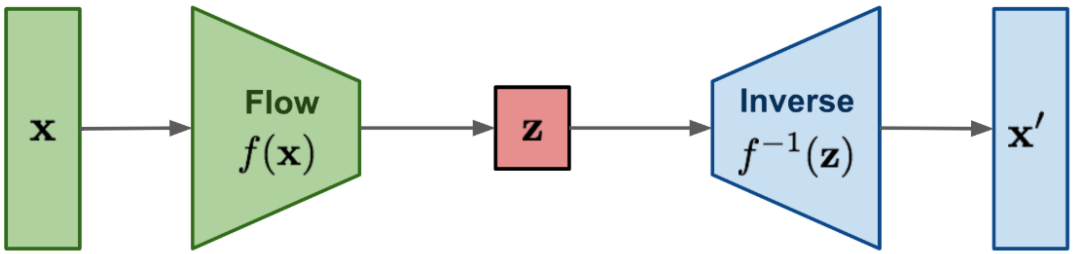
GAN: minimax the classification error loss.



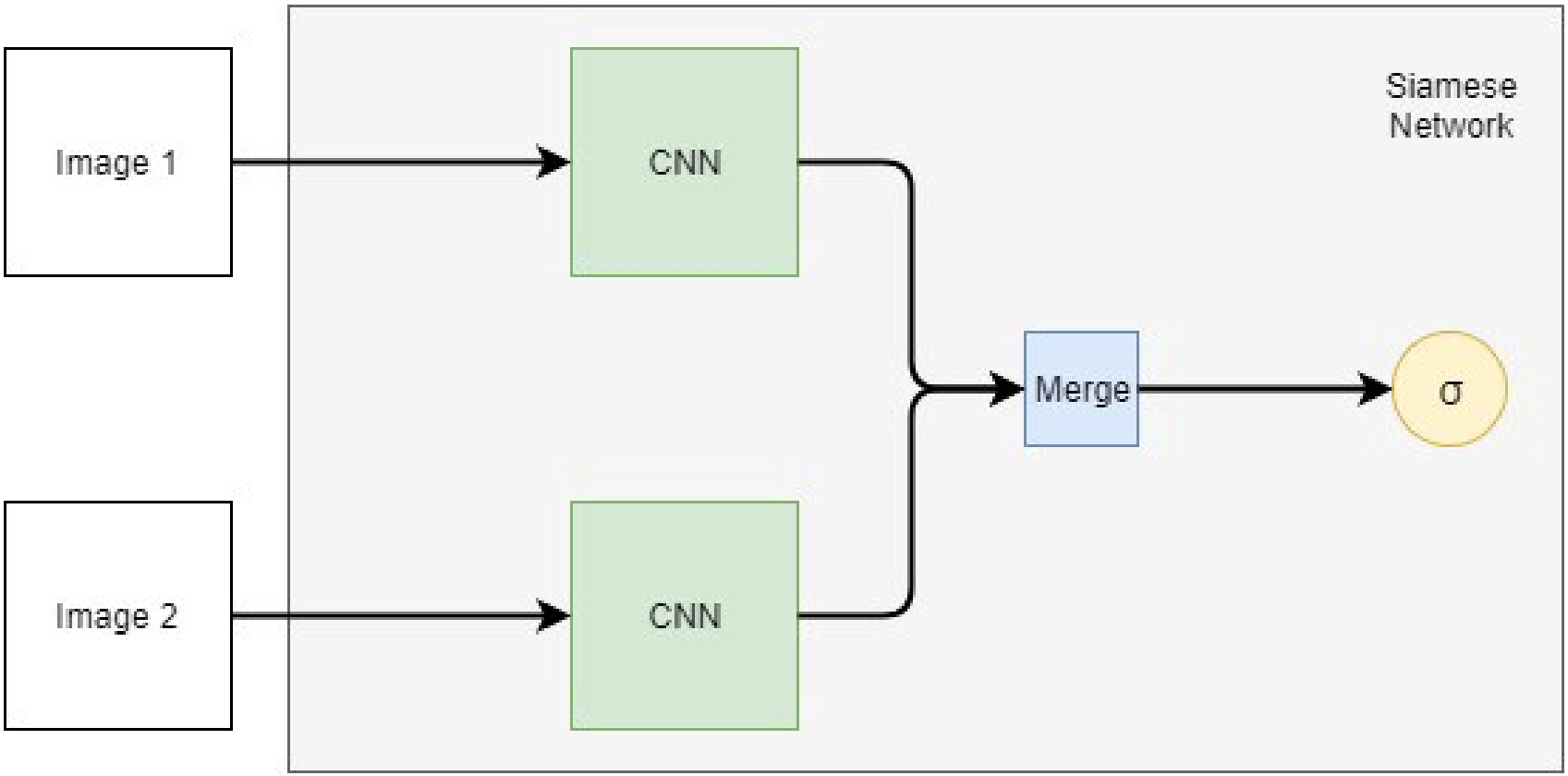
VAE: maximize ELBO.



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
- 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