Mise en place d'un réseau de neurones pour la détection d'objets sur une image

Introduction et problématique

Apprentissage automatique

Reconnaissance d'images

Réseaux de neurones artificiels

Réseaux à convolution

Transport de données de l'entrée à la sortie

Essentiel dans les voitures autonomes

Comment implémenter un réseau de neurones à convolution dans un langage classique de type Python ?

Plan - Déroulement

I. Théorie

1. Réseaux de neurones classiques
2. Rétropropagation du gradient
3. Réseaux de neurones à convolution

II. Implémentation d’un tel réseau

1. Architecture du réseau
2. Explication couche par couche
3. Le dataset

III. Optimisation et fonctions secondaires

1. Optimisation avec numpy
2. Fonctions de sauvegarde
3. Visualisation de l’action du réseau

IV. Résultats

1. Visualisation
2. Apprentissage

I.
Théorie

Bibliographie

[1] David Kriesel : A Brief Introduction to Neural Networks :
http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf

[2] J.G. Makin : Backpropagation :
https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf

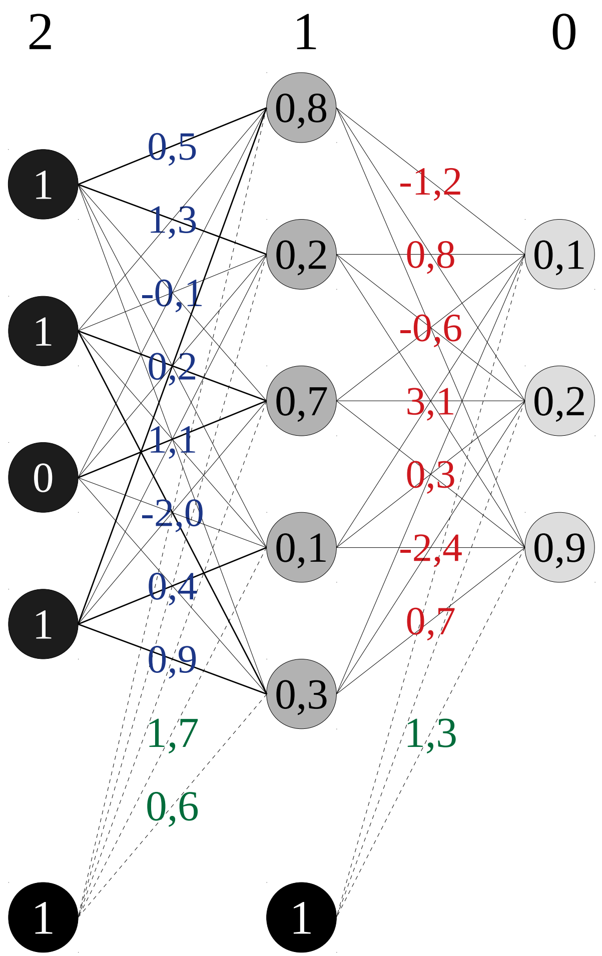
[3] David Stutz : Understanding Convolutional Neural Networks :
https://davidstutz.de/wordpress/wp-content/uploads/2014/07/seminar.pdf

[4] Matthew D. Zeiler- Rob Fergus : Visualizing and Understanding Convolutional Networks :
https://cs.nyu.edu/%7Efergus/papers/zeilerECCV2014.pdf

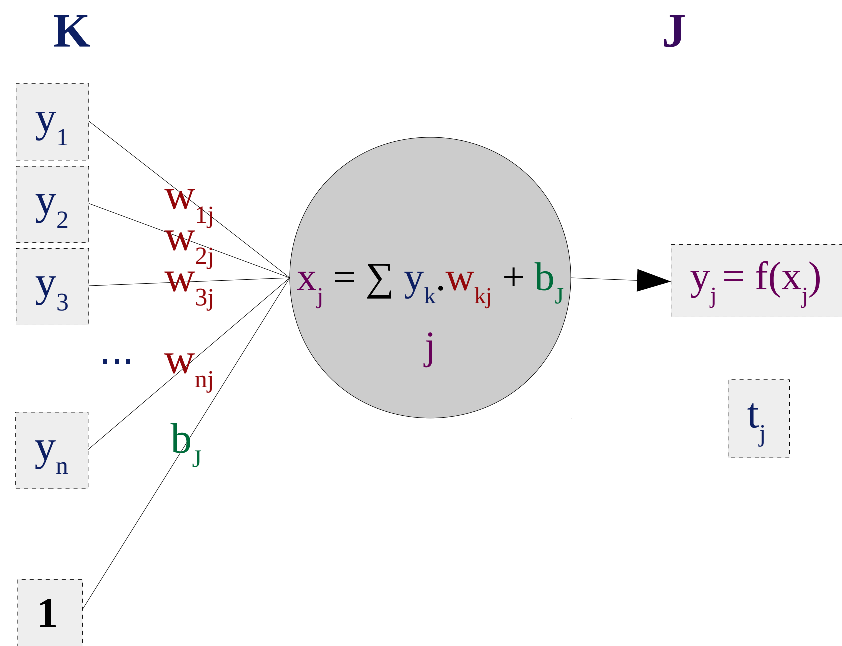
[5] Alex Krizhevsky- Ilya Sutskever- Geoffrey E. Hinton : ImageNet Classification with Deep Convolutional Neural Networks :
https://papers.nips.cc/paper/4824-imagenet-classification-withdeep-convolutional-neural-networks.pdf

I. 1. Réseaux de neurones classiques

[1] David Kriesel : A Brief Introduction to Neural Networks :
http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf



Réseau de neurones



Neurone de McCulloch-Pitts

$$x_j = \sum y_k.w_{kj} + b_J$$

$$y_j = f(x_j)$$

Calcul d'une sortie

$$E = g(y_j, t_j)$$

Fonction d'erreur

I. 2. Rétropropagation du gradient

[2] J.G. Makin : Backpropagation :
https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf

$$\Delta \omega_{kj} = -\alpha \frac{\partial E}{\partial \omega_{kj}}$$

$$\frac{\partial E}{\partial \omega_{kj}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_j}\frac{\partial x_j}{\partial \omega_{kj}}$$

$$\delta_j = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_j}$$

$$\frac{\partial x_j}{\partial \omega_{kj}} = y_k$$

$$\frac{\partial y_j}{\partial x_j} = \frac{df}{dx}(x_j)$$

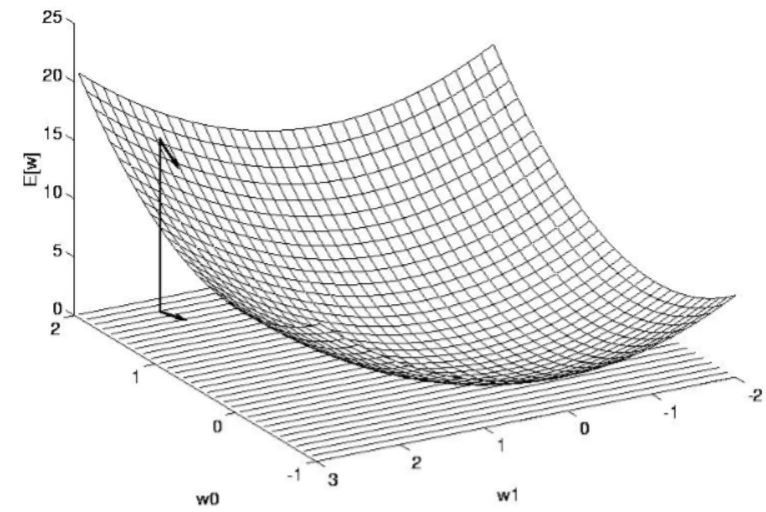
$$\frac{\partial E}{\partial y_j} = \sum_{i\in I}\frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_i}\frac{\partial x_i}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_i} = -\delta_i$$

$$\frac{\partial x_i}{\partial y_j} = w_{ji}$$

$$\frac{\partial E}{\partial y_j} = \frac{\partial g}{\partial y}(y_j, t_j)$$

Calcul du gradient



J.G. Makin : Backpropagation

$$\Delta \omega_{kj}(n) = \alpha\,\delta_j\,y_k + \eta\,\Delta\omega_{kj}(n-1) - \alpha\,\varepsilon\,\omega_{kj}$$

$$\delta_j = -\frac{\partial g}{\partial y}(y_j, t_j)\frac{df}{dx}(x_j)$$

$$\delta_j = \Big(\sum_{i\in I}\delta_i\,\omega_{ji}\Big)\frac{df}{dx}(x_j)$$

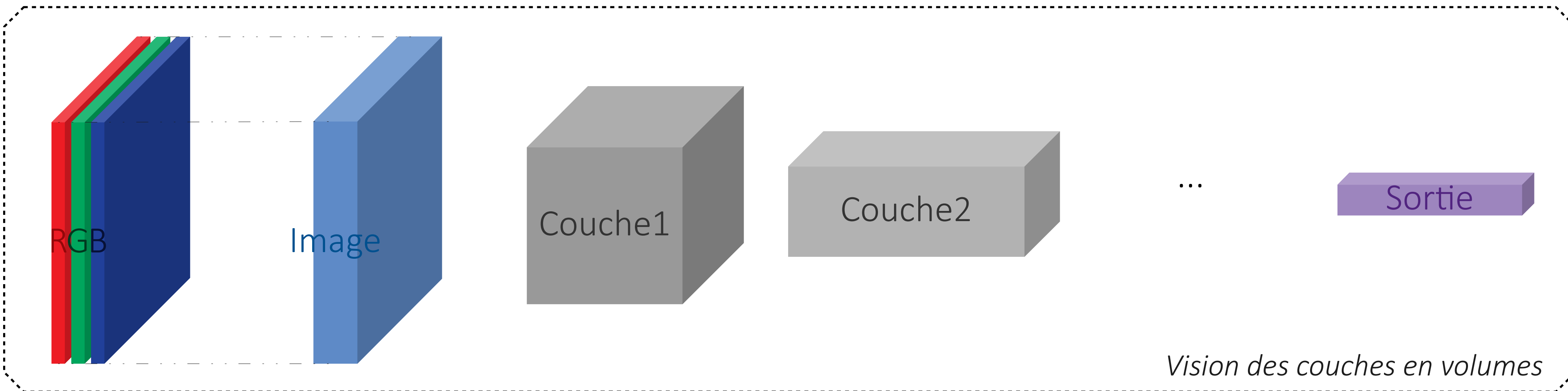
Modification d'un poids

[5] Alex Krizhevsky- Ilya Sutskever- Geoffrey E. Hinton : ImageNet Classification with Deep Convolutional Neural Networks :
https://papers.nips.cc/paper/4824-imagenet-classification-withdeep-convolutional-neural-networks.pdf 0

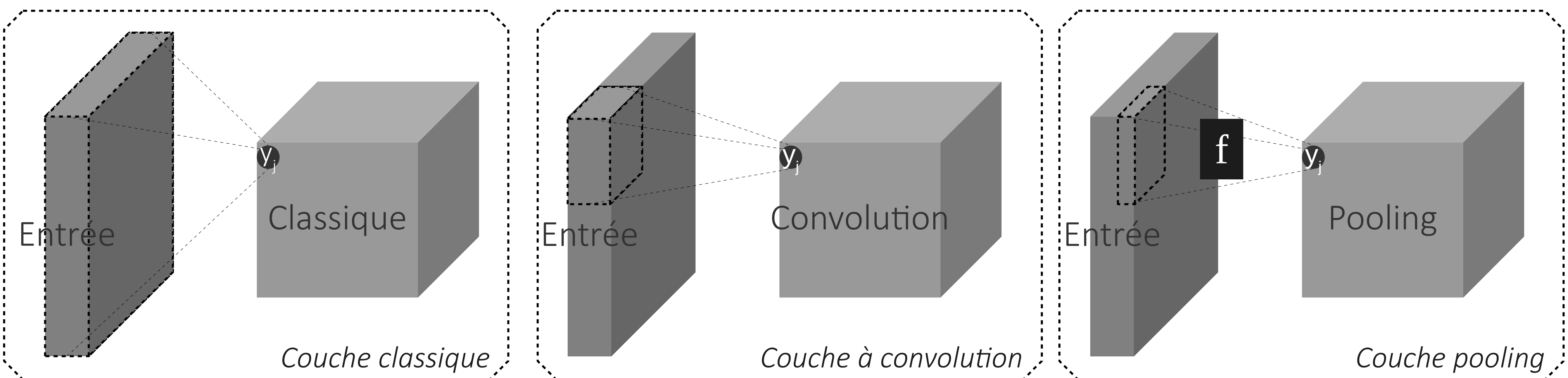
I. 3. Réseaux de neurones à convolution

[3] David Stutz : Understanding Convolutional Neural Networks :
https://davidstutz.de/wordpress/wp-content/uploads/2014/07/seminar.pdf

[4] Matthew D. Zeiler- Rob Fergus : Visualizing and Understanding Convolutional Networks :
https://cs.nyu.edu/%7Efergus/papers/zeilerECCV2014.pdf

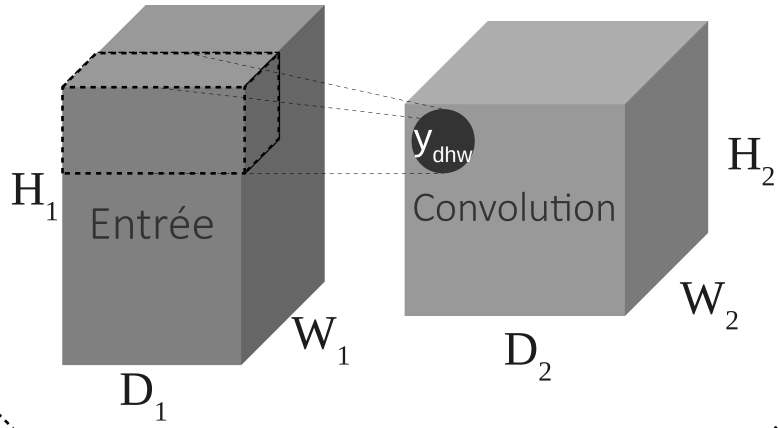


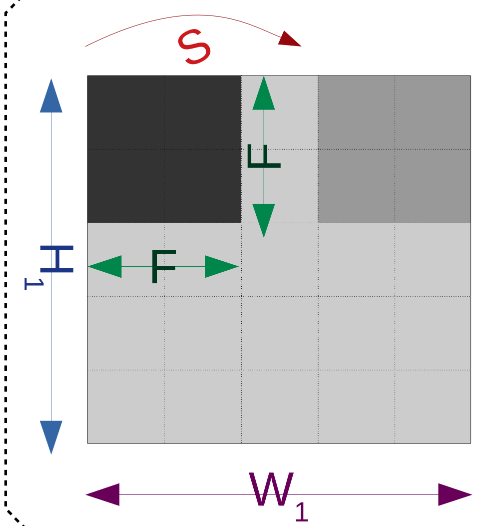
Vision des couches en volumes

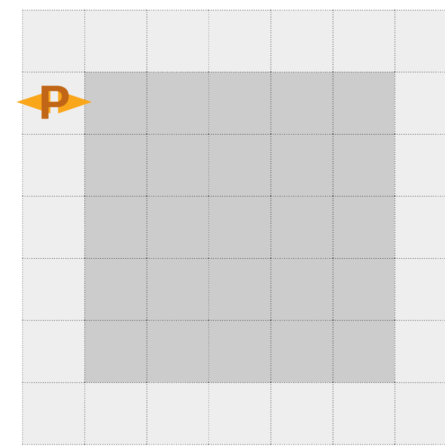


Couche classique   *Couche à convolution*   *Couche pooling*

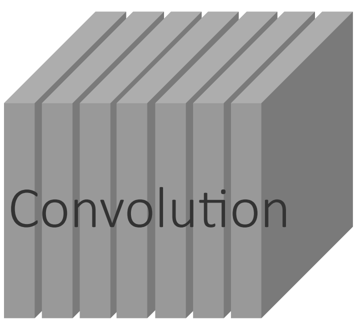
I. 3. a. Convolution







Pour y_{dhw} :
$F^2.D_1$ poids

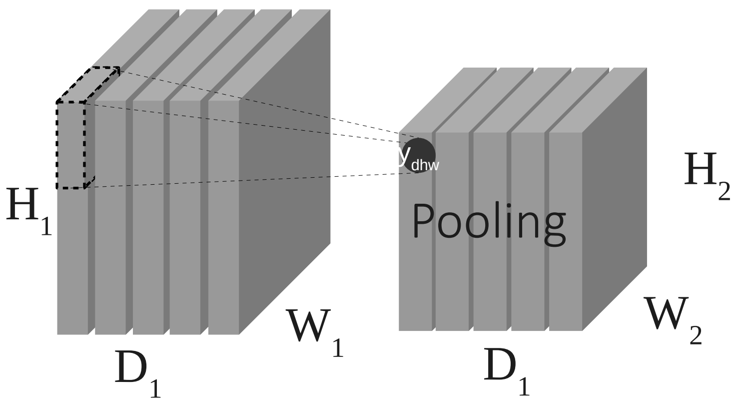


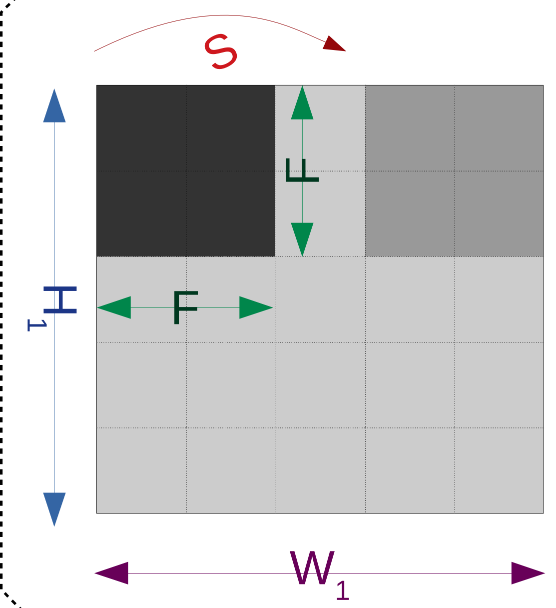
→ Ensemble de $(F^2.D_1.D_2)$ poids

$$H_2 = \frac{H_1 - F + 2 \times P}{S} + 1$$

$$W_2 = \frac{W_1 - F + 2 \times P}{S} + 1$$

I. 3. b. Pooling





y_{dhw} = f(restriction)

ex :

f : {entrées} → max({entrées})

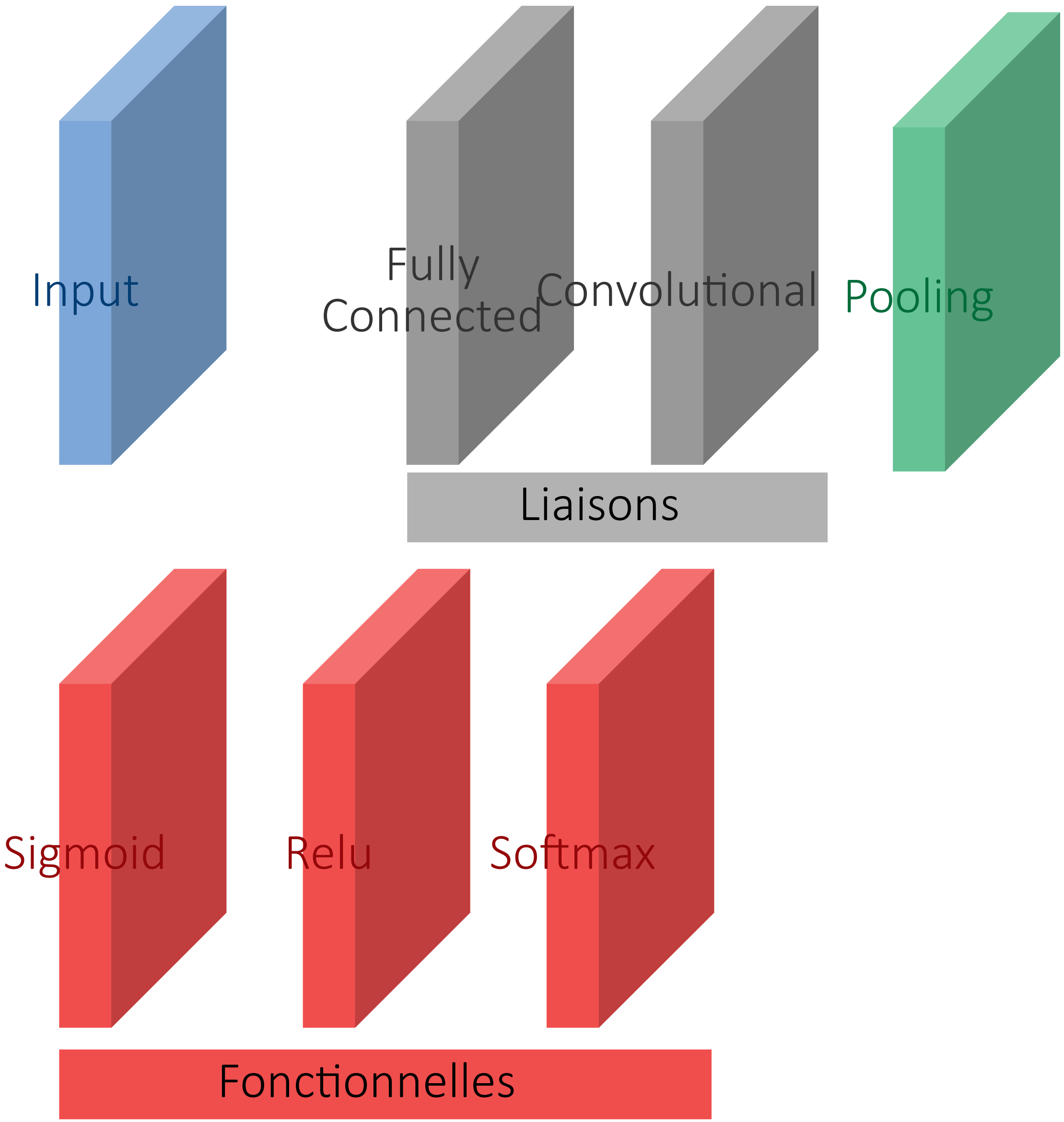
→ Pooling :
réduction des données

$$H_2 = \frac{H_1 - F}{S} + 1$$

$$W_2 = \frac{W_1 - F}{S} + 1$$

II. Implémentation du réseau

II. 1. Architecture du réseau

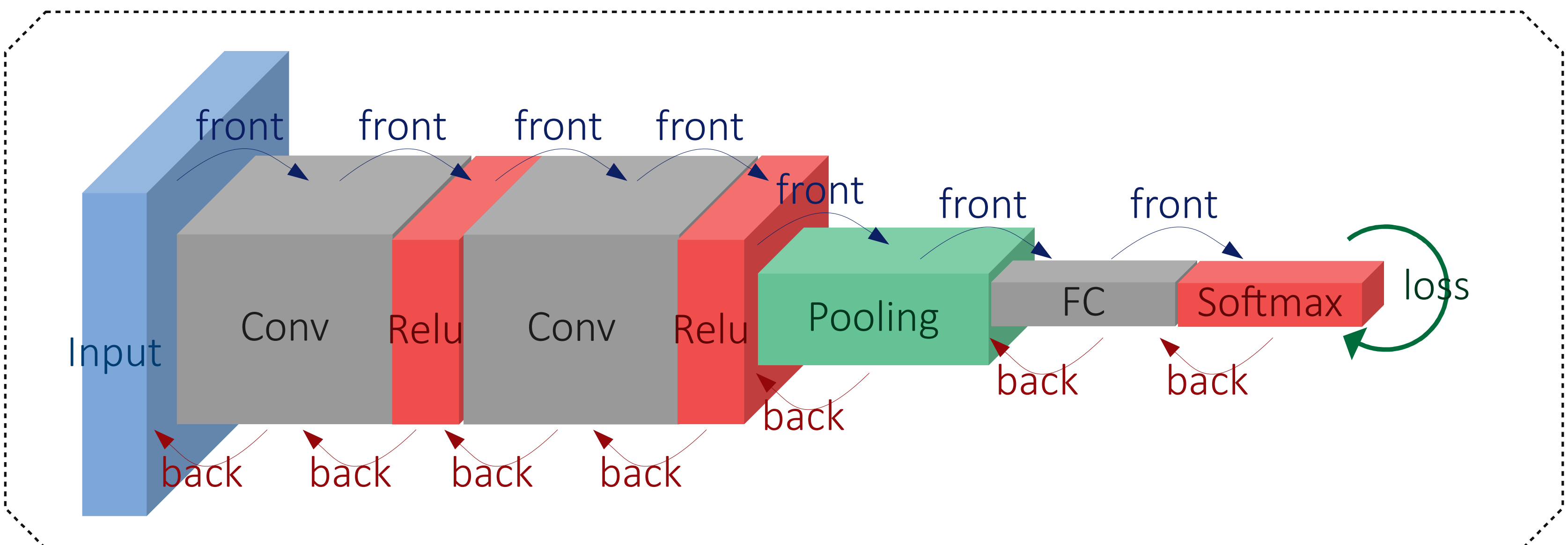



```
class Network: ...

class Input: ...

class FullyConnected: ...

class Convolutional: ...

class Pooling: ...

class Sigmoid: ...

class Relu: ...

class Softmax: ...
```

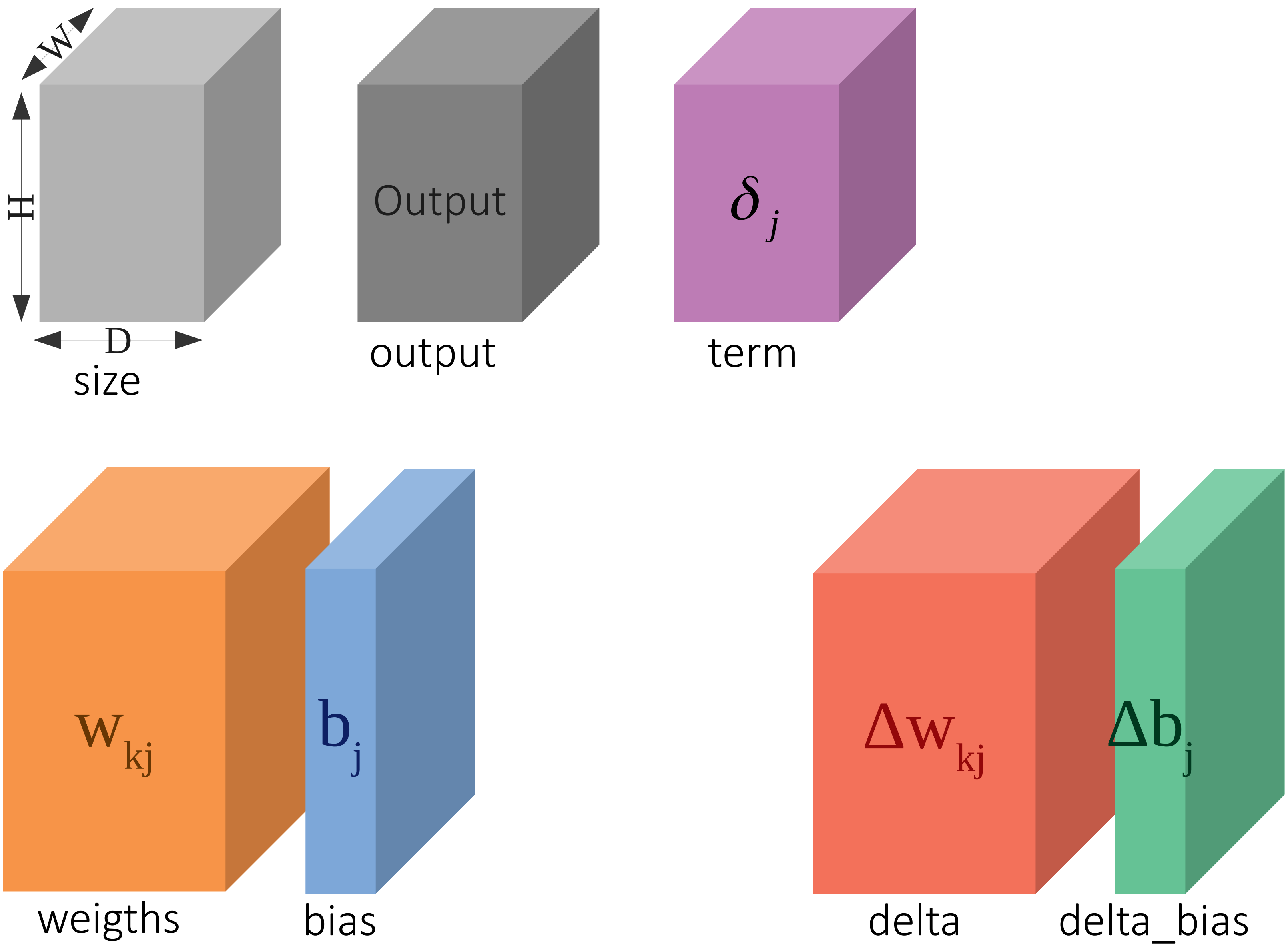
II. 1. b. Network

```python
def __init__(self, name, B, layers_list, cat):
    self.id = name
    n = len(layers_list)
    self.depth = n
    self.batch_size = B
    self.layers = []
    self.loss = np.array([])
    self.accuracy = np.array([])

    # LAYERS
    Type, param = layers_list[0]
    self.layers.append(Input(B, param))
    for i in range(1, n):
        Type, param = layers_list[i]
        self.layers.append(Type(param, self.layers[i-1].size))
```

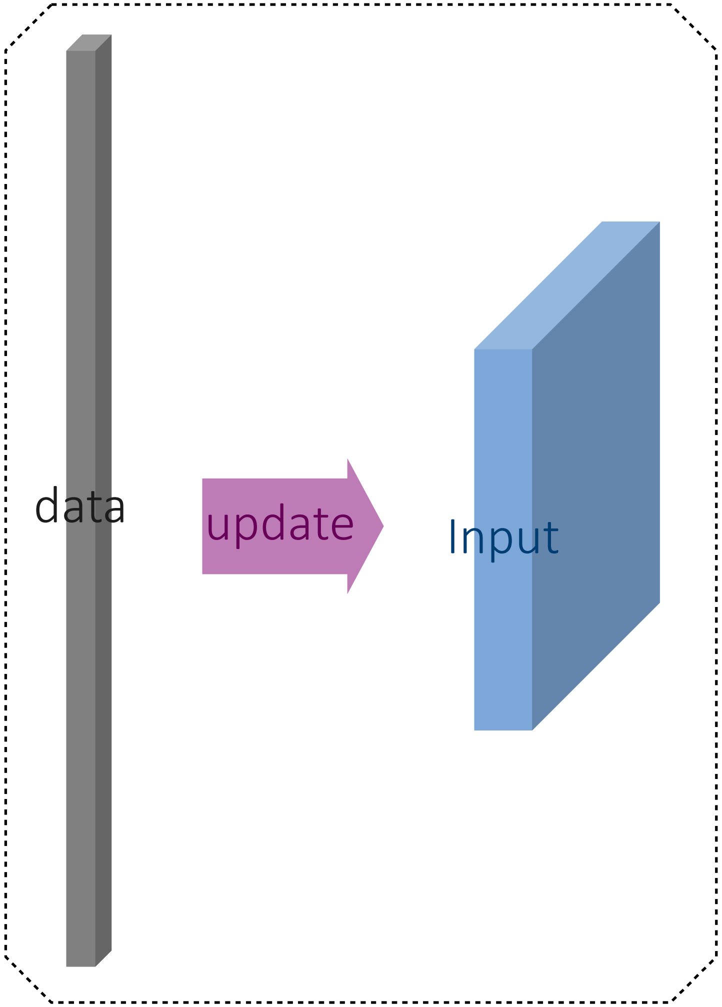
```python
def train(self, example):
    n = self.depth
    img, label = example
    self.layers[0].update(img)

    # propagation avant
    for i in range(1, n):
        self.layers[i].front(self.layers[i-1].output)

    # calcul de l'erreur
    self.layers[n-1].loss(label)

    # rétropropagation
    for i in range(1, n):
        self.layers[n-i].back(self.layers[n-i-1])
```



II. 1. c. Structure d'une couche

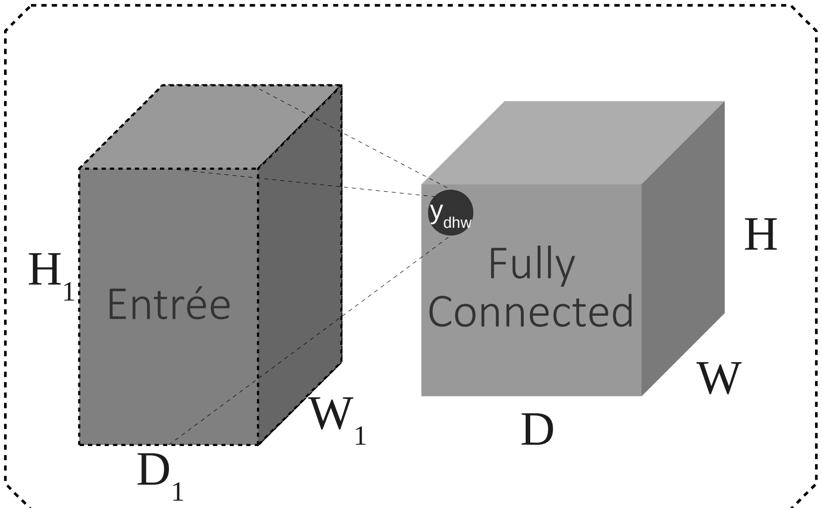


II. 2. a. Input



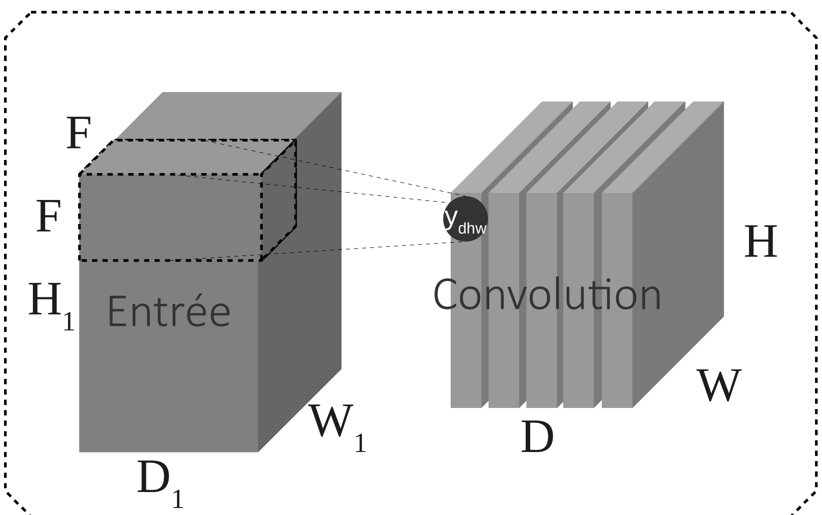

```python
class Input:
    """
            INPUT
    + s'occupe de la transition entre le réseau e
    - size : tuple des dimensions (D, H, W)
    - output : tableau des valeurs d'entrée
    > update : fait prendre à output une nouvelle
    > response : donne la carte des responsabilit
    """
    def __init__(self, B, parameters):
        self.id = "Input"
        D, H, W = parameters
        self.size = B, D, H, W
        self.output = np.zeros(self.size)
        self.term = np.zeros(self.size)
        self.tempo = [0]

    def define(self): ...

    def update(self, out):
        start = time.clock()
        self.output = np.reshape(out, self.size)
        self.tempo[0] += time.clock() - start
```

II. 2. b. Couches de Liaison




```python
def __init__(self, parameters, prev_size):
    FullyConnected.ident += 1
    D, H, W, rate = parameters
    B, D1, H1, W1 = prev_size
    self.id = "FullyConnected_n°" + str(FullyConnected.ident)
    self.size = (B, D, H, W)
    self.weigths = np.random.randn(D, H, W, D1, H1, W1) * 0.01 / sqrt(D1*H1*W1)
    self.delta = np.zeros((D, H, W, D1, H1, W1))
    self.bias = np.random.randn(D, H, W) * 0.01 / sqrt(D1*H1*W1)
    self.delta_bias = np.zeros((D, H, W))
    self.distrib = np.ones((B))
```

Ensemble de D*H*W sets de $D_1*H_1*W_1$ poids
+ 1 biais pour chaque set : D*H*W biais

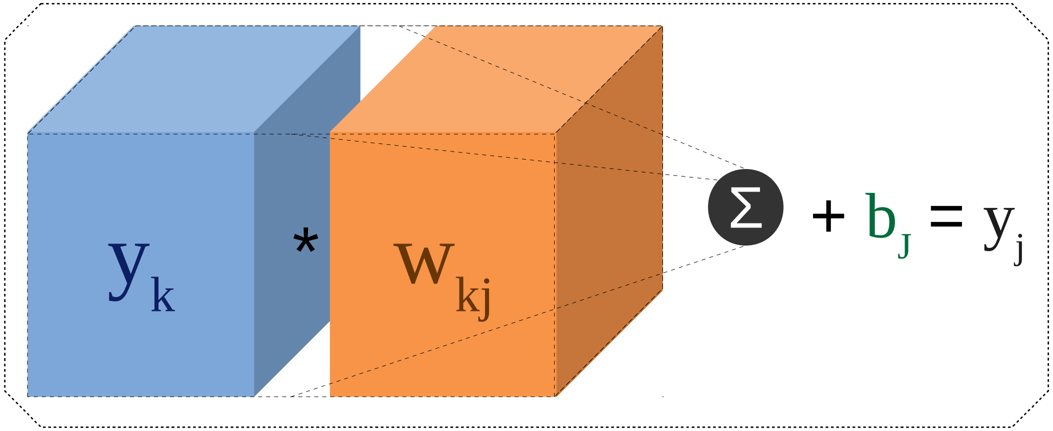



```python
def __init__(self, parameters, prev_size):
    Convolutional.ident += 1
    self.id = "Convolutional_n°" + str(Convolutional.ident)
    B, D1, H1, W1 = prev_size
    D2, F, S, P, rate = parameters
    if P == -1 and S == 1 :
        P = int((F - 1)/2)
    self.hyper = F, S, P
    H2 = int(((H1 - F + 2*P) // S) + 1)
    W2 = int(((W1 - F + 2*P) // S) + 1)
    self.size = B, D2, H2, W2
    self.weigths = np.random.randn(D2, D1, F, F) / sqrt(D1*H1*W1)
    self.delta = np.zeros((D2, D1, F, F))
    self.bias = np.ones((D2)) / sqrt(D1*H1*W1)
    self.delta_bias = np.zeros((D2))
    self.distrib = np.ones((B))
```

Ensemble de D sets de D_1*F*F poids
+ 1 biais pour chaque set : D biais

II. 2. b. Couches de Liaison

Front :



FullyConnected :

```
def front(self, inp):
    B, D, H, W = self.size
    self.output = np.transpose(np.einsum('bijk,...ijk', inp, self.weigths), axes = [3, 0, 1, 2])
    + np.einsum('dhw,...', self.bias, self.distrib)
```

Convolutional :

```
def front(self, inp):
    B, D1, H1, W1 = inp.shape
    B, D, H, W = self.size
    F, S, P = self.hyper
    padded_inp = np.zeros((B, D1, H1 + 2*P, W1 + 2*P))
    padded_inp[:, :, P:H1+P, P:W1+P] = inp
    for b in range(B):
        for h in range(H):
            for w in range(W):
                self.output[b, :, h, w] = np.transpose(np.einsum('...ijk,ijk', self.weigths, padded_inp[b, :, h*S:h*S+F, w*S:w*S+F]))
                + self.bias
```

II. 2. b. Couches de Liaison

Back :

FullyConnected :

```
def back(self, prev_layer):
    B, D, H, W = self.size

    # terme d'erreur :
    prev_layer.term = np.transpose(np.einsum('bdhw,dhw...', self.term, self.weigths), axes = [3, 0, 1, 2])

    # biais :
    self.delta_bias = (self.speed / B * np.sum(self.term, axis = 0)) + (self.moment*self.delta_bias)
    self.bias += self.delta_bias

    # poids :
    self.delta = self.speed / B * (np.einsum('bijk,b...', prev_layer.output, self.term) - self.white*self.weigths) + (self.moment*self.delta)
    self.weigths += self.delta
```

Convolutional :

```
def back(self, prev_layer):
    B, D, H, W = self.size
    B, D1, H1, W1 = prev_layer.size
    F, S, P = self.hyper
    delta = np.zeros(np.shape(self.delta))
    padded_prev_out = np.zeros((B, D1, H1 + 2*P, W1 + 2*P))
    padded_prev_out[:, :, P:H1+P, P:W1+P] = prev_layer.output
    padded_prev_term = np.zeros((B, D1, H1 + 2*P, W1 + 2*P))

    # biais :
    self.delta_bias = (self.speed / (H*W*B) * np.sum(self.term, axis = (3, 2, 0))) + (self.moment*self.delta_bias)

    for b in range(B):
        for h in range(H):
            for w in range(W):
                # on envoie le terme d'erreur dans les neurones précedents
                padded_prev_term[b, :, h*S:h*S+F, w*S:w*S+F] += np.einsum('d..., d...', self.term[b, :, h, w], self.weigths)
                delta += np.einsum('ijk,...', padded_prev_out[b, :, h*S:h*S+F, w*S:w*S+F], self.term[b, :, h, w])

    # poids :
    self.delta = self.speed * ((delta / (H*W*B)) + (self.moment*self.delta) - (self.white*self.weigths))

    self.weigths += self.delta
    self.bias += self.delta_bias
    prev_layer.term = padded_prev_term[:, :, P:H1+P, P:W1+P]
```

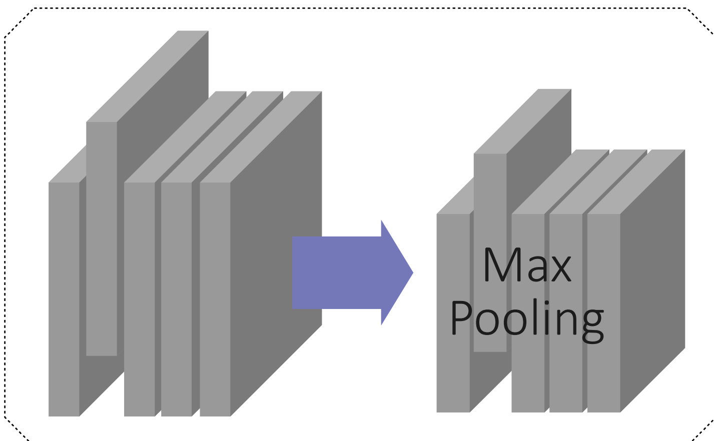
$$\Delta\omega_{kj}(n)=\alpha\,\delta_j\,y_k+\eta\,\Delta\omega_{kj}(n-1)-\alpha\,\varepsilon\,\omega_{kj}$$

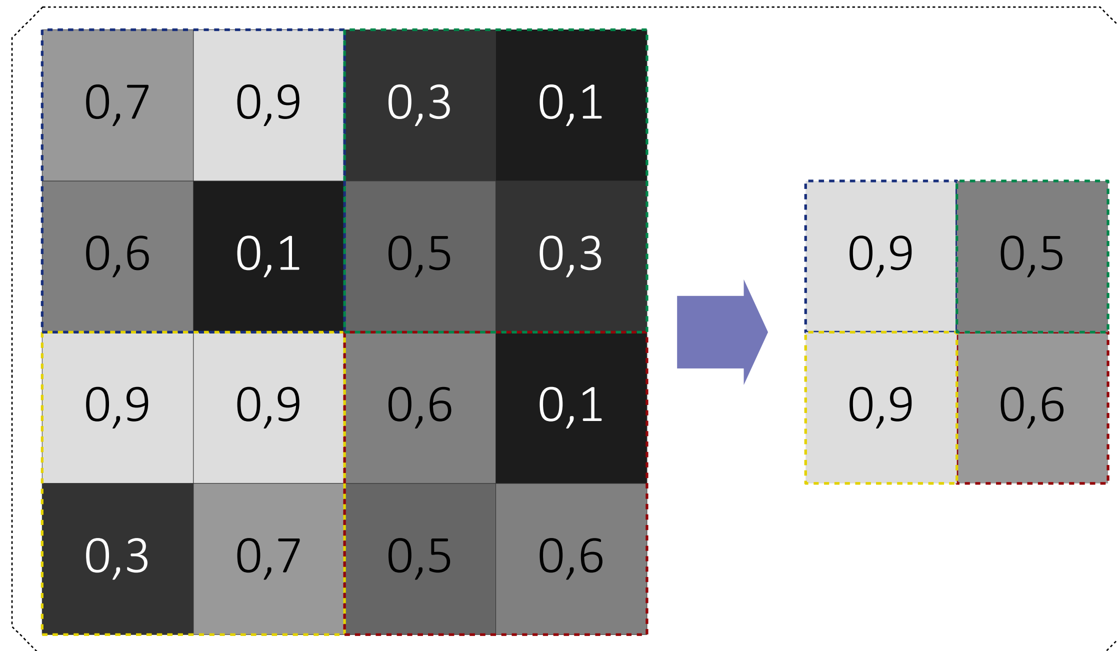
$$\delta_j=-\frac{\partial g}{\partial y}(y_j,t_j)\frac{df}{dx}(x_j)$$

$$\delta_j=\Big(\sum_{i\in I}\delta_i\,\omega_{ji}\Big)\frac{df}{dx}(x_j)$$

II. 2. c. Pooling

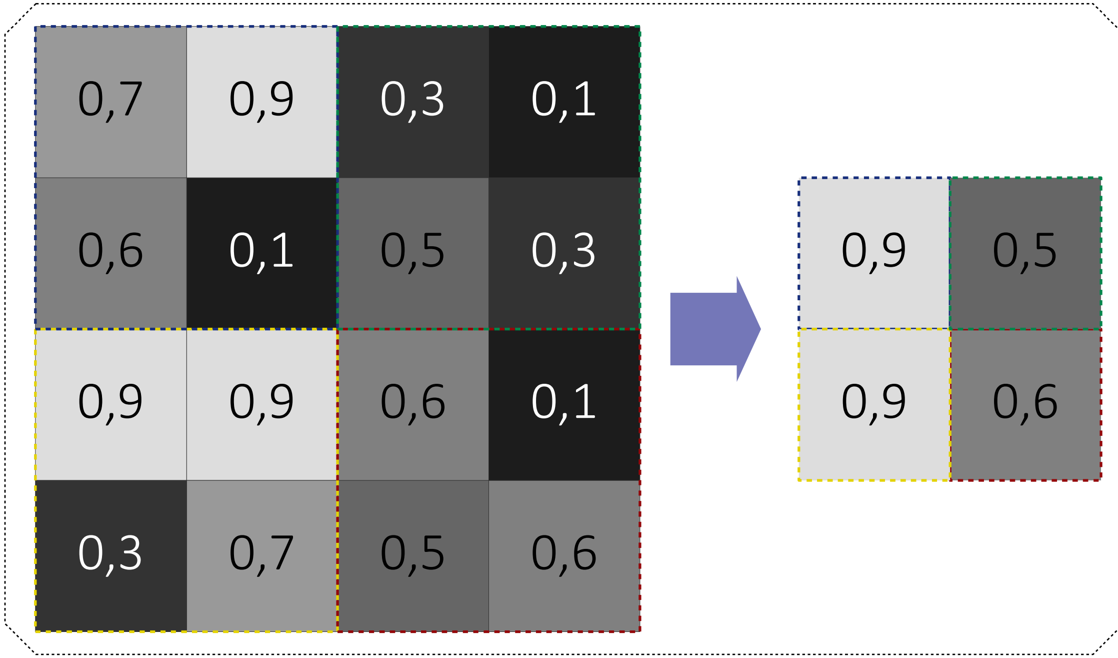
Front :

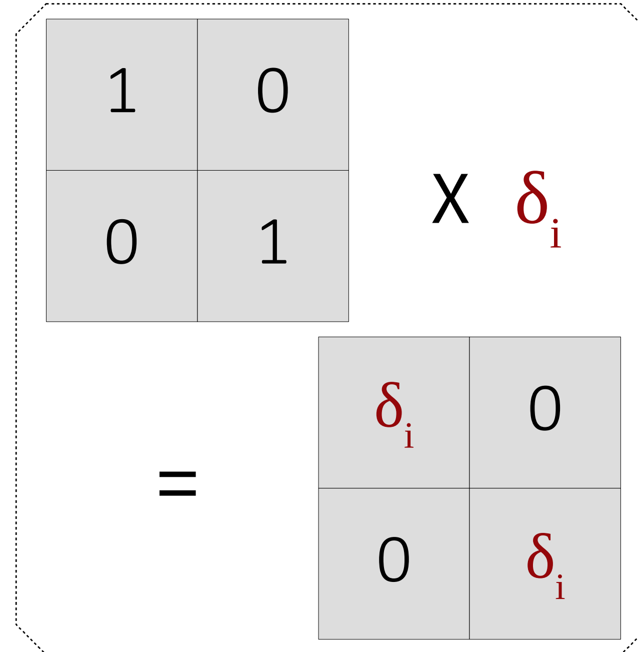





```python
def front(self, inp):
    B, D, H, W = self.size
    F, S = self.hyper
    for h in range(H):
        for w in range(W):
            self.output[:, :, h, w] = np.max(np.max(inp[:, :, h*S:h*S+F, w*S:w*S+F], axis = 2), axis = 2)
```

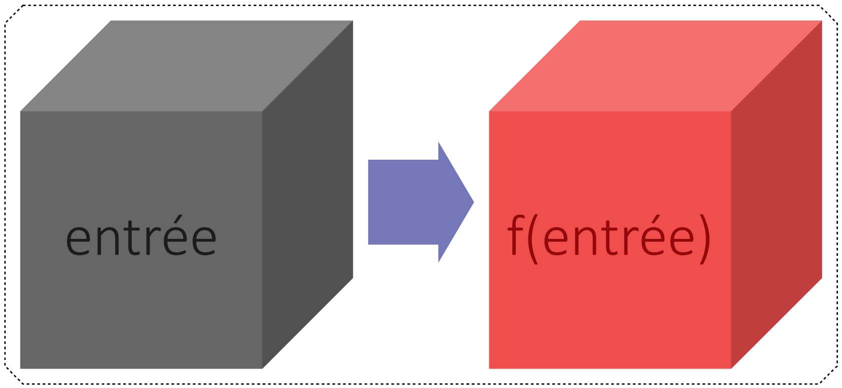
II. 2. c. Pooling






```python
def back(self, prev_layer):
    B, D, H, W = self.size
    F, S = self.hyper
    prev_layer.term = np.zeros(np.shape(prev_layer.term))
    ones = np.ones((F, F))

    for h in range(H):
        for w in range(W):
            M = prev_layer.output[:, :, h*S:h*S+F, w*S:w*S+F]
            - np.transpose(np.einsum('b...,ij' , self.output[:, :, h, w], self.distrib), axes = [1, 0, 3, 2])
            M[M == 0] = 1
            M[M != 1] = 0
            prev_layer.term[:, :, h*S:h*S+F, w*S:w*S+F] += np.einsum('...,...ij', self.term[:, :, h, w], M)
```

II. 2. d. Couches Fonctionnelles

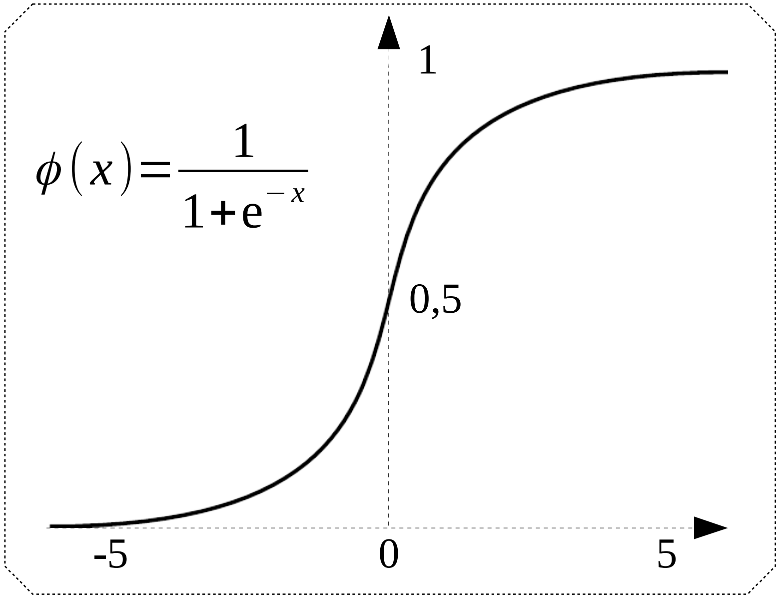


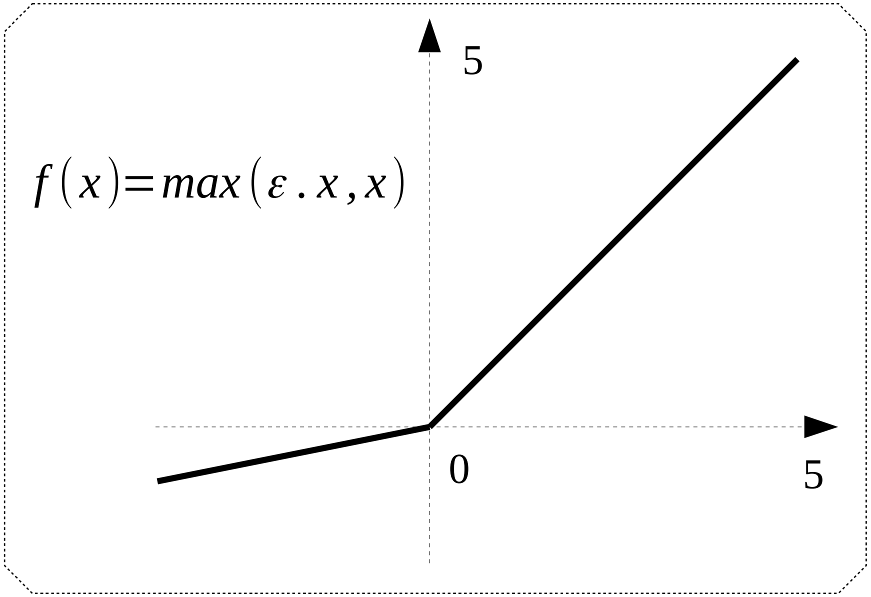
Sigmoid :

```python
def front(self, inp):
    def sigmoid(array):
        return 1 / (1 + np.exp(-array))
    self.output = sigmoid(inp)
```

ReLU :

```python
def front(self, inp):
    def ReLU(array):
        return np.maximum(array, 0.01*array)
    self.output = ReLU(inp)
```





II. 2. d. Couches Fonctionnelles

$$\Delta\omega_{kj}(n)=\alpha\,\delta_j\,y_k+\eta\,\Delta\omega_{kj}(n-1)-\alpha\,\varepsilon\,\omega_{kj}$$

$$\delta_j=-\frac{\partial g}{\partial y}(y_j,t_j)\frac{df}{dx}(x_j)$$

$$\delta_j=\left(\sum_{i\in I}\delta_i\omega_{ji}\right)\frac{df}{dx}(x_j)$$

Sigmoid :

```python
def back(self, prev_layer):
    # on adapte le terme d'erreur
    prev_layer.term = self.term * self.output * (1 - self.output)
```

ReLU :

```python
def back(self, prev_layer):
    # on adapte le terme d'erreur
    M = np.copy(self.output)
    M[M >= 0] = 1
    M[M < 0] = 0.01
    prev_layer.term = self.term * M
```

$$\phi'(x)=\phi(x)(1-\phi(x))$$

$$f'(x)=\begin{cases}1\ si\ f(x)>0\\ \varepsilon\ si\ f(x)<0\end{cases}$$

loss :

```python
def loss(self, expect):
    out = np.copy(np.reshape(self.output, np.shape(expect)))
    err = expect - out
    self.term = np.reshape(err, np.shape(self.term))
    self.error = np.sum(err**2)/2
```

$$E=\frac{\sum_{j\in J}(y_j-t_j)^2}{2}$$

$$-\frac{\partial E}{\partial y_j}=t_j-y_j$$

II. 2. d. Couches Fonctionnelles

Softmax :

$$\sigma\left(\begin{bmatrix}2{,}3\\0{,}5\\1{,}2\\1{,}8\\2\end{bmatrix}\right)\approx\begin{bmatrix}0{,}35\\0{,}06\\0{,}12\\0{,}21\\0{,}26\end{bmatrix}$$

front :

```python
def front (self, inp):
    B, D, H, W = self.size
    def soft(array):
        array -= np.max(array)
        e = np.exp(array)
        return e / np.sum(e)
    for b in range(B):
        self.output[b] = soft(inp[b])
```

$$\sigma(X)_j=\frac{e^{x_j}}{\sum_{k=1}^{K}e^{x_k}}=\frac{e^{x_j-m}}{\sum_{k=1}^{N}e^{x_k-m}}\quad où\ X=\begin{bmatrix}x_1\\x_2\\\dots\\x_N\end{bmatrix}$$

loss / back :

```python
def back (self, prev_layer):
    prev_layer.term = self.term

def loss (self, expect):
    B, D, H, W = self.size
    out = np.copy(np.reshape(self.output, np.shape(expect)))
    err = expect - out
    self.term = np.reshape(err, np.shape(self.term))
    self.error = - np.stack([np.sum(expect[b]*np.log(out[b])) for b in range(B)])
```

$$L=-\sum_{j=1}^{N}t_j.\ln(y_j)$$

$$\delta_j=-\frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial x_j}=-\frac{\partial L}{\partial x_j}=t_j-y_j$$

II. 3. Le dataset

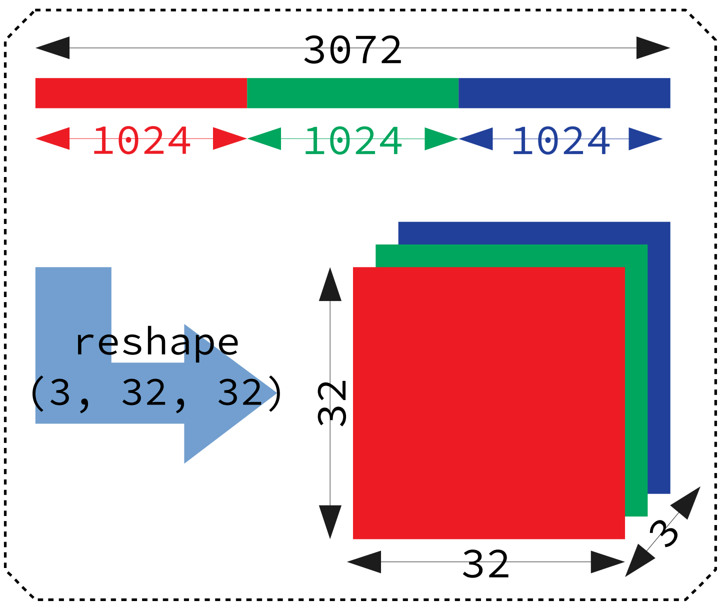
CIFAR-10

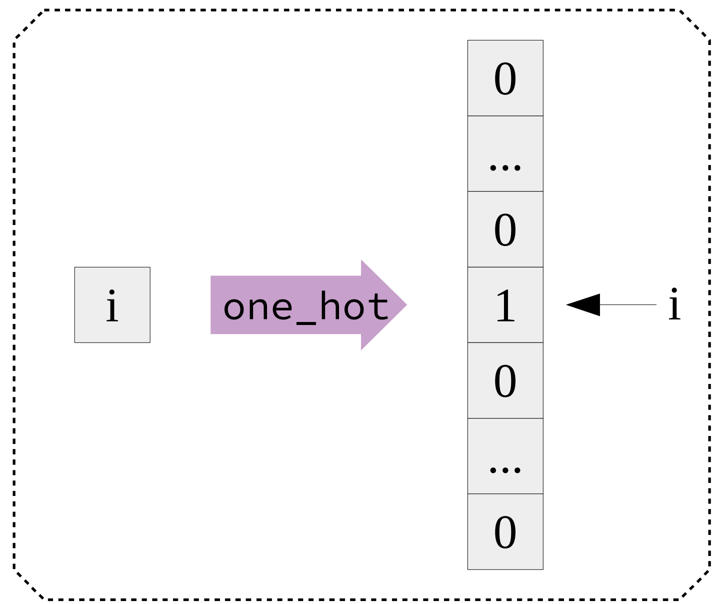
- batch_1.npz
- batch_2.npz
- batch_3.npz
- batch_4.npz
- batch_5.npz
- batch_test.npz

data : (10000x3072) array
labels : 10000 nb list

```python
def extract_batch(batch_path):
    with open(batch_path, 'rb') as fo:
        batch = pickle.load(fo, encoding='bytes')
    return batch[b'data'], batch[b'labels']

def reshape(features):
    return np.reshape(features, (len(features), 3, 32, 32))

def one_hot(labels):
    encoded = np.zeros((len(labels), 10))
    for idx, val in enumerate(labels):
        encoded[idx, val] = 1
    return encoded

def routine(batch_nb):
    (ftr, lbl) = extract_batch("CIFAR-10/data_batch_" + str(batch_nb))
    features = reshape(ftr)
    labels = one_hot(lbl)

    np.savez_compressed("Data/Data_Brut/batch_" + str(batch_nb), f = features, l = labels)
```

features :
(10000x3x32x32)





III. Optimisation et fonctions secondaires

III. 1. Optimisation

```
[(Input, (3, 32, 32)),
 (Convolutional, (16, 5, 1, 2, (0.01, 0.9, 0.0001))), (Relu, 0),
 (Pooling, (2, 2)),
 (Convolutional, (20, 5, 1, 2, (0.01, 0.9, 0.0001))), (Relu, 0),
 (Pooling, (2, 2)),
 (Convolutional, (20, 5, 1, 2, (0.01, 0.9, 0.0001))), (Relu, 0),
 (Pooling, (2, 2)),
 (FullyConnected, (1, 1, 10, (0.01, 0.9, 0.0001))), (Softmax, 0)]
```

```
IN (3x32x32)
CONV [F=5, S=1, K=16, P=2]
RELU (16x32x32)
POOL (16x16x16) [F=2, S=2]
CONV [F=5, S=1, K=20, P=2]
RELU (20x16x16)
POOL (20x8x8) [F=2, S=2]
CONV [F=5, S=1, K=20, P=2]
RELU (20x8x8)
POOL (20x4x4) [F=2, S=2]
FC (1x1x10)
SOFTMAX (1x1x10)

α=0,01 ; η=0,9 ; ε=0,005
```

→ Problème d'explosion / disparition du gradient
→ Normalisation des entrées
(+ tentative de créer une couche Norm)
→ Apprentissage impossible
(résultat invariant selon l 'entrée)
→ Remise en cause de la normalisation
(Retour au 1er problème)
→ Amélioration de l'initialisation des poids

Pour 1000 ex :

```
[Finished in 364.2s]
```

```
print("      term max : " + str(np.max(self.layers[n-i].term)))
print("           min : " + str(np.min(self.layers[n-i].term)))
print("     delta max : " + str(np.max(self.layers[n-i].delta)))
print("           min : " + str(np.min(self.layers[n-i].delta)))
print("delta_bias max : " + str(np.max(self.layers[n-i].delta_bias)))
print("           min : " + str(np.min(self.layers[n-i].delta_bias)))
```

III. 1. a. Optimisation avec Numpy

Mesure du temps :

```
def front(self, inp):
    start = time.clock()
    #[...]
    self.tempo[0] += time.clock() - start
```

Différentes procédures :

```
# technique 1
start1 = time.clock()
for n in range(N):
    out1 = out1 - out1
    for d in range(D):
        for h in range(H):
            for w in range(W):
                for i in range(I):
                    for j in range(J):
                        for k in range(K):
                            out1[d, h, w] += inp[i, j, k] * weigths[d, h, w, i, j, k]
end1 = time.clock()
```

```
# technique 2
start2 = time.clock()
for n in range(N):
    for d in range(D):
        for h in range(H):
            for w in range(W):
                out2[d, h, w] = np.sum(weigths[d, h, w] * inp)
end2 = time.clock()
```

```
# technique 3
start3 = time.clock()
for n in range(N):
    out3 = np.sum(np.sum(np.sum(weigths * inp, axis = 5), axis = 4), axis = 3)
end3 = time.clock()
```

```
# technique 4
start4 = time.clock()
for n in range(N):
    out4 = np.einsum('...ijk, ijk', weigths, inp)
end4 = time.clock()
```

Sortie :

```
# sortie console
print("\nOut : " + str(np.all(out1 == out2) and np
print("technique 1 : " + str(end1 - start1) + "\n
```

III. 1. a. Optimisation avec Numpy

FullyConnected :

```
FC_Front

Out : True

technique 1 : 0.06986800139100911
technique 2 : 0.032989646346127915
technique 3 : 0.0006796461460098907
technique 4 : 0.0001593392631201035


FC_Back

Delta : True
Prev_term : True

technique 1 : 0.004414679299526478
technique 2 : 0.04596371369226454
technique 3 : 0.0009296048952646407
technique 4 : 0.000297911560667663
```

Convolutional :

```
Conv_Front

Out : True

technique 1 : 25.805143939614197
technique 2 : 0.315979198296958
technique 3 : 0.2400763167105744
technique 4 : 0.05449553831184062


Conv_Back

Delta : True
Delta_Bias : True
Prev_term : True

technique 1 : 51.1044721849151
technique 2 : 0.7146845479050512
technique 3 : 0.2539486497130241
technique 4 : 0.21119815196661307
```

Pooling :

```
Pool_Front

Out : True

technique 1 : 0.17075882869782788
technique 2 : 0.016333029632193075


Pool_Back

Prev_term : True

technique 1 : 0.09175525036086185
technique 2 : 0.32677160151440887
technique 3 : 0.028837763556396112
```

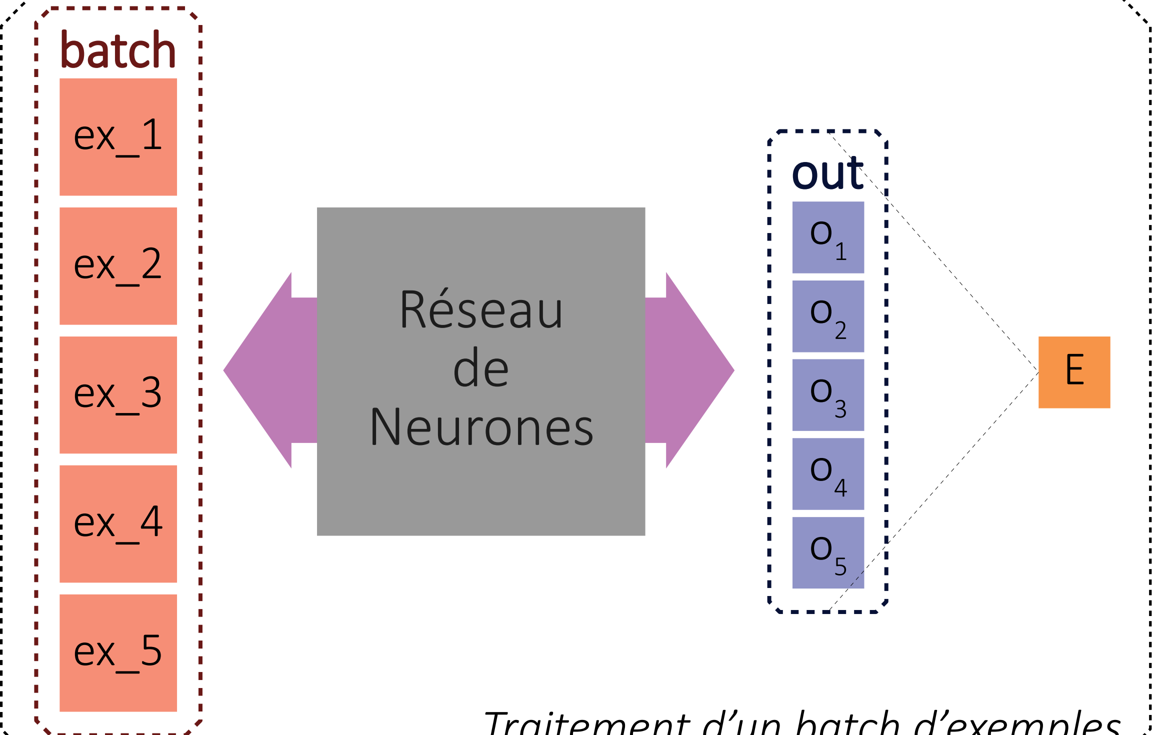
Pour 1000 ex :

```
[Finished in 78.3s]
```

III. 1. b. Batch

Batch de taille b :
ensemble de b exemples

1er : maximum de boucles
2e : méthode de base + boucle sur B
3e : einsum + boucle sur B
4e : einsum



Traitement d'un batch d'exemples

```
# technique 4
start4 = time.clock()
distrib = np.ones((B))
for n in range(N):
    out4 = np.transpose(np.einsum('bijk,...ijk', inp, weigths), axes = [3, 0, 1, 2])
    + np.einsum('dhw,...', bias, distrib)
end4 = time.clock()
```

III. 1. b. Batch

FullyConnected :

```
FC_Front

Out : True
technique 1 : 0.47478607193601363
technique 2 : 0.25221290897234855
technique 3 : 0.0009556579975283919
technique 4 : 0.0005958231213335326


FC_Back

Delta : True
Prev_term : True
technique 1 : 1.0797236256705371
technique 2 : 0.4147986151832195
technique 3 : 0.0036768856499134195
technique 4 : 0.0015284486661379937
```

Convolutional :

```
Conv_Front

Out : True
technique 1 : 1662.2624023150256
technique 2 : 25.99852667818793
technique 3 : 3.424109012221166
technique 4 : 3.7993167290745404


Conv_Back

Delta : True
Delta_Bias : True
Prev_term : True
technique 1 : 2393.45376159599
technique 2 : 42.1735197023754
technique 3 : 9.286126496495854
technique 4 : 10.880755328520166
```

Pooling :

```
Pool_Front

Out : True
technique 1 : 0.5224862815313145
technique 2 : 0.061666937895807905
technique 3 : 0.019400499237860913


Pool_Back

Prev_term : True
technique 1 : 0.4499948460167218
technique 2 : 0.13949586114375734
technique 3 : 0.04949674090792655
```

Pour 1000 ex :

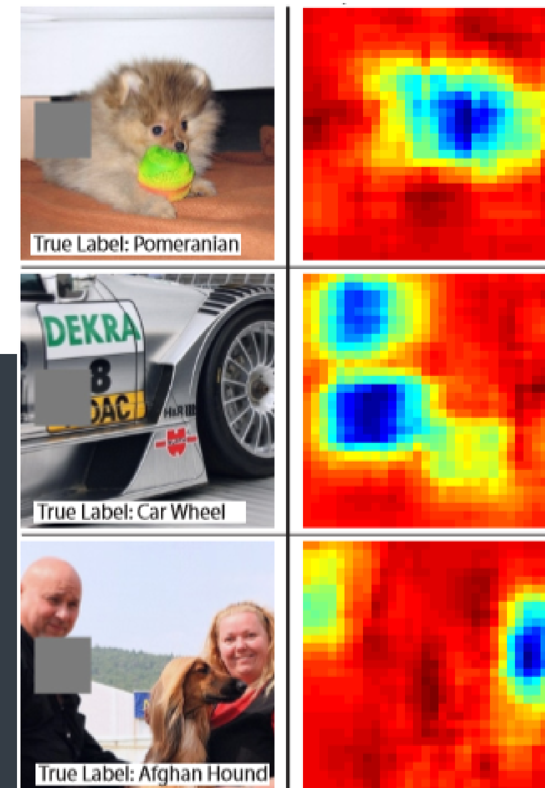
```
[Finished in 62.0s]
```

III. 2. Fonctions de sauvegarde

- Accuracy.npz
- Convolutional_n°1.npz
- Convolutional_n°2.npz
- Convolutional_n°3.npz
- FullyConnected_n°1.npz
- Loss.npz

```python
# ENREGISTREMENT / CHARGEMENT DU RÉSEAU

def load(self):
    loaded = np.load(self.id + "_save/Loss.npz")
    self.loss = loaded["l"]
    loaded = np.load(self.id + "_save/Accuracy.npz")
    self.accuracy = loaded["a"]
    for l in self.layers :
        if type(l) == FullyConnected or type(l) == Convolutional:
            loaded = np.load(self.id + "_save/" + l.id + ".npz")
            l.weigths = loaded["w"]
            l.delta = loaded["d"]
            l.bias = loaded["b"]
            l.delta_bias = loaded["db"]


def save(self):
    if not (self.id + "_save") in os.listdir("."):
        os.mkdir(self.id + "_save")
    np.savez_compressed(self.id + "_save/Loss", l = self.loss)
    np.savez_compressed(self.id + "_save/Accuracy", a = self.accuracy)
    for l in self.layers :
        if type(l) == FullyConnected or type(l) == Convolutional:
            np.savez_compressed(self.id + "_save/" + l.id, w = l.weigths, d = l.delta, b = l.bias, db = l.delta_bias)
```

III. 3. Visualisation de l’action du réseau



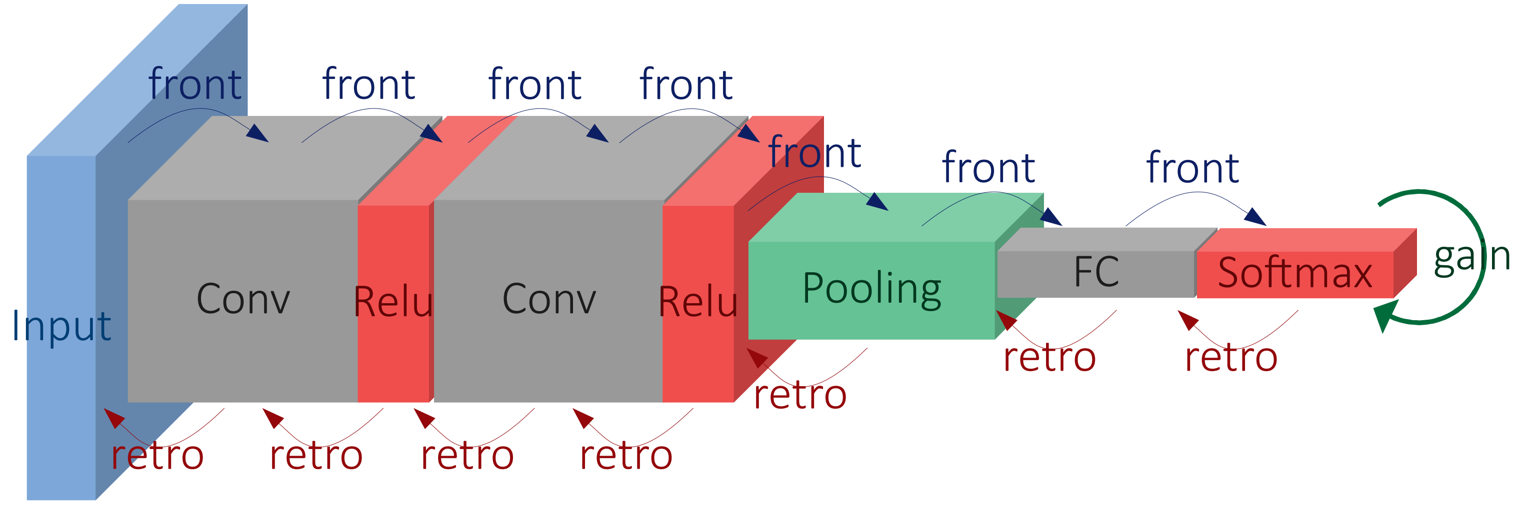
Matthew D. Zeiler- Rob Fergus :
Visualizing and Understanding
Convolutional Networks

```python
def visual(self, example, F, name):
    # fonction existante

    n = self.depth
    img, label = example
    self.layers[0].update(img)
    B, D, H, W = np.shape(img)
    H1 = H - F + 1
    W1 = W - F + 1
    heat_map = np.zeros((B, H1, W1))

    for h in range(H1):
        for w in range(W1):
            cache = np.copy(img)
            cache[:, :, h : h+F, w : w+F] = np.zeros((B, D, F, F))
            self.layers[0].update(cache)
            for i in range(1, n):
                self.layers[i].front(self.layers[i-1].output)
            heat_map[:, h, w] = self.layers[n-1].gain(label)
            print(heat_map[:, h, w])
    for b in range(B):
        imsave("Images/"+ name + "_" + str(b) + "_expect" + ".png", np.transpose(img[b], (1, 2, 0)))
        imsave("Images/"+ name + "_" + str(b) + "_visual" + ".png", -heat_map[b])
```

III. 3. a. Algorithme personnel

Adaptation de l'algorithme de rétropropagation :
→ Calcul de la justesse et non de l'erreur
→ Calcul récursif de la responsabilité de chaque neurone dans cette justesse
→ Final : responsabilité chaque pixel de l'image dans le résultat

```python
def responsability(self, example, name):
    # fonction développée personnellement

    n = self.depth
    img, label = example
    self.layers[0].update(img)

    # propagation avant
    for i in range(1, n):
        self.layers[i].front(self.layers[i-1].output)

    # calcul de la justesse
    self.layers[n-1].gain(label)

    # rétropropagation sans modification des poids
    for i in range(1, n):
        self.layers[n-i].retro(self.layers[n-i-1])
    self.layers[0].response(name)
```



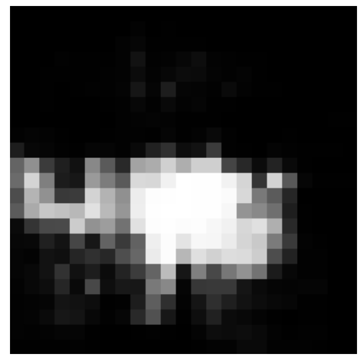
IV.
Résultats

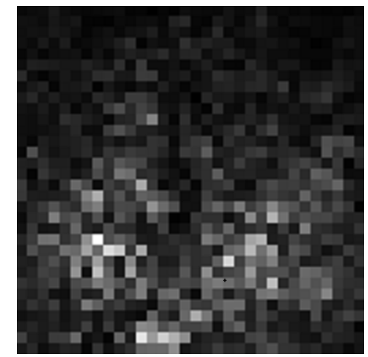
IV. 1. Visualisation

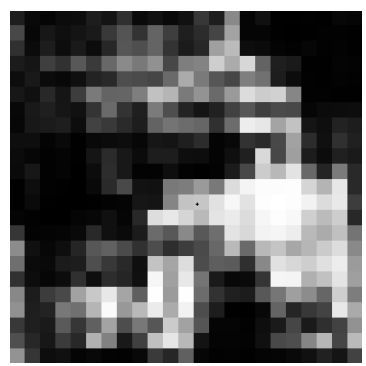
Image :

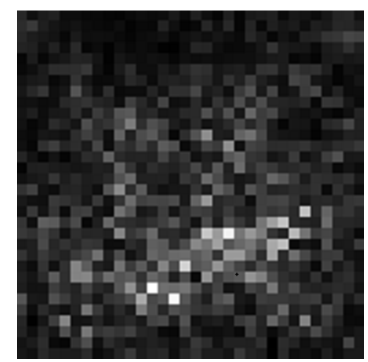
Algorithme de visualisation :

Fonction personelle :

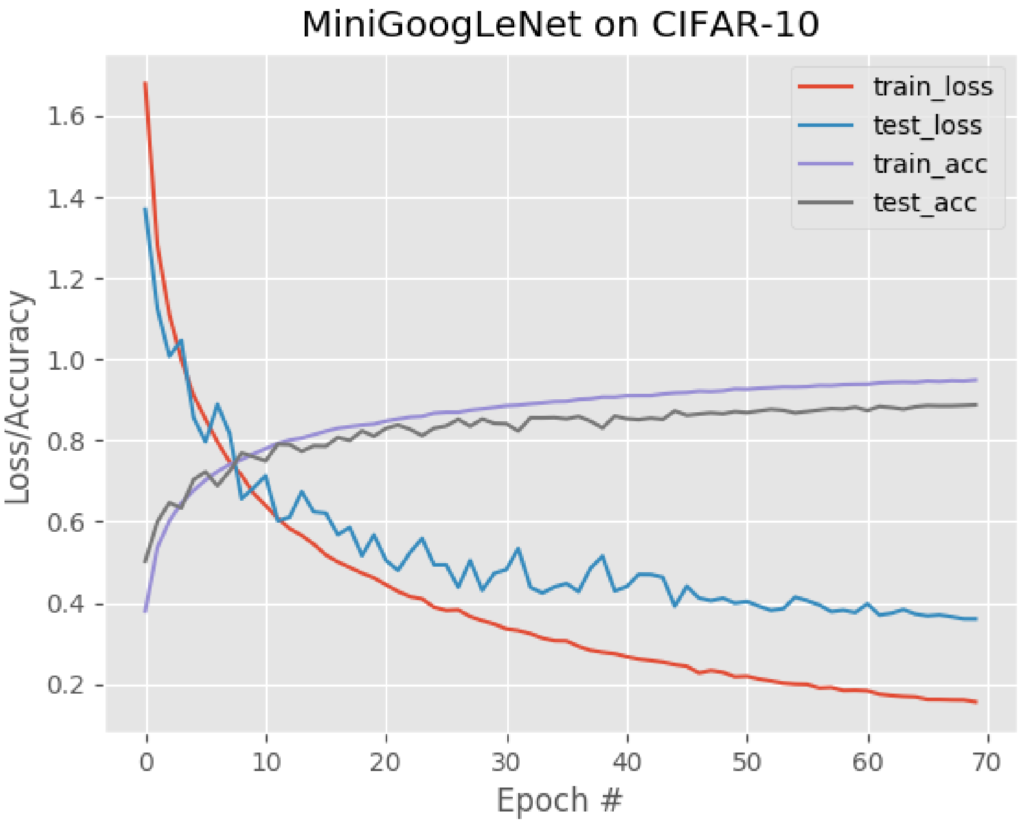


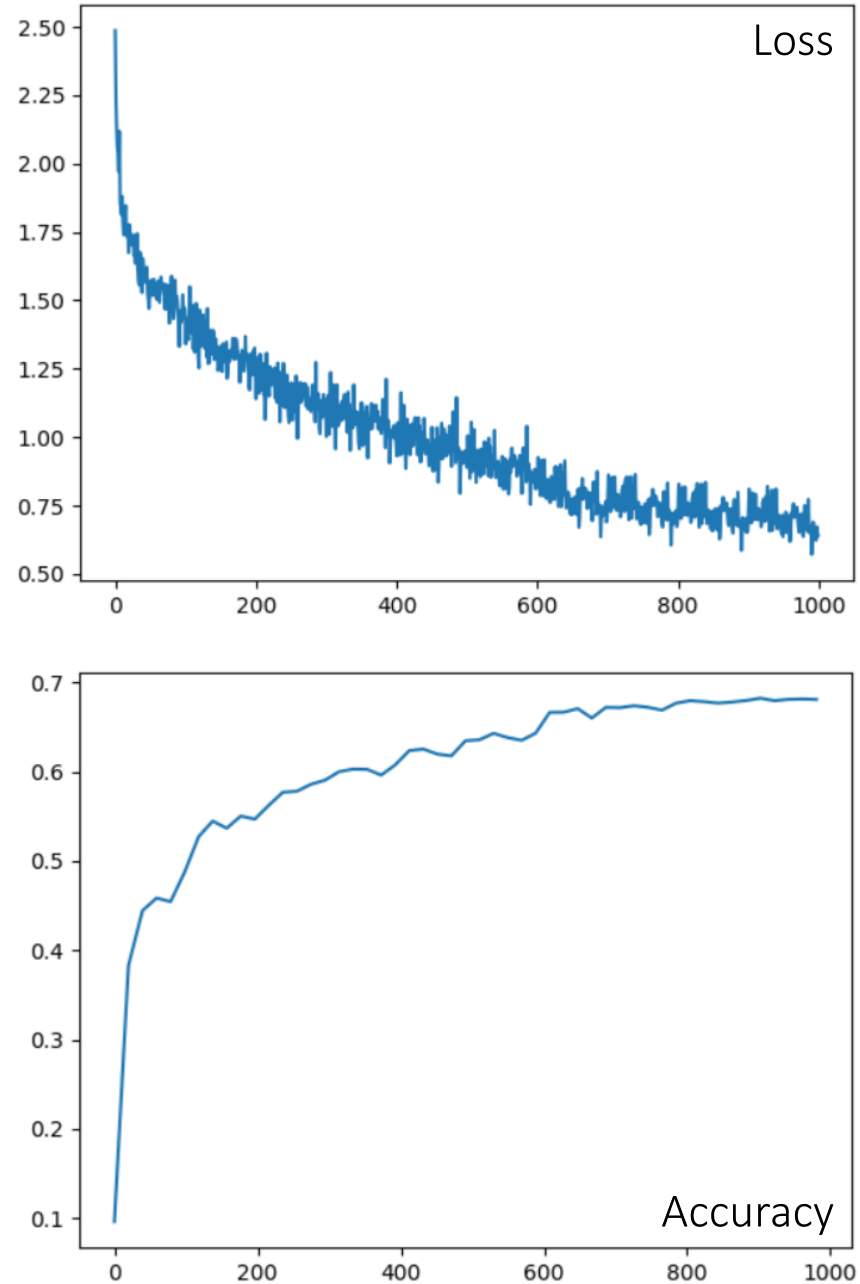






IV. 2. Apprentissage





Annexe

Network

```python
# ACCURACY / PRÉCISION DU RÉSEAU

def veri(self, example):
    n = self.depth
    img, label = example
    self.layers[0].update(img)

    # propagation avant
    for i in range(1, n):
        self.layers[i].front(self.layers[i-1].output)

    # calcul de la précision
    points = self.layers[n-1].score(label)
    print(points)
    return points


def verify(self, dataset):
    B = self.batch_size
    points = 0
    features, labels = dataset
    end = int(len(features)/B)
    for i in range(end):
        print("\n" + str(int(i/end*100)) + "%")
        ex = (features[i*B : (i+1)*B], labels[i*B : (i+1)*B])
        points += self.veri(ex)
    self.accuracy = np.append(self.accuracy, points/(end*B))
```

```python
# ENTRAINEMENT DU RÉSEAU

def train(self, example):
    n = self.depth
    img, label = example
    self.layers[0].update(img)

    # propagation avant
    for i in range(1, n):
        self.layers[i].front(self.layers[i-1].output)

    # calcul de l'erreur
    self.layers[n-1].loss(label)

    # rétropropagation
    for i in range(1, n):
        self.layers[n-i].back(self.layers[n-i-1])

    batch_loss = np.sum(self.layers[n-1].error)/self.batch_size
    print(batch_loss, end = "\n\n")
    self.loss = np.append(self.loss, batch_loss)


def training(self, dataset, dist):
    B = self.batch_size
    batch, (features, labels) = dataset
    start, end = dist
    end = int((end-start)/B + start)
    for i in range(start, end):
        print("\n BATCH : " + str(batch) + " - " + str(int(i/end*100)) + "%")
        ex = (features[i*B : (i+1)*B], labels[i*B : (i+1)*B])
        self.train(ex)
```

Network

```python
# TEST RAPIDE SANS MODIFICATION DES POIDS

def test(self, example):
    n = self.depth
    B = self.batch_size
    C = len(self.categories)
    img, label = example

    self.layers[0].update(img)
    for i in range(1, n):
        self.layers[i].front(self.layers[i-1].output)

    out = np.reshape(self.layers[n-1].output, newshape = (B, C))
    for b in range(B):
        for c in range(C):
            print("[" + str(label[b, c]) + "] [" + str(out[b, c]) + "] -> " + self.categories[c])
        print("")


def testing(self, dataset, dist):
    B = self.batch_size
    features, labels = dataset
    start, end = dist
    end = int((end-start)/B + start)
    for i in range(0, end-start):
        print("\nEX : " + str(start + i*B + 1))
        ex = (features[start + i*B : start + (i+1)*B], labels[start + i*B : start + (i+1)*B])
        self.test(ex)
```

Input

```python
def response(self, name):
    B, D, H, W = self.size
    self.term = np.absolute(self.term)
    mapp = np.sum(self.term, axis = 1)
    def normal(feat):
        min_val = np.min(feat)
        max_val = np.max(feat)
        return (feat - min_val) / (max_val - min_val)
    for b in range(B):
        imsave("Images/"+ name + "_" + str(b) + "_expect" + ".png", np.transpose(self.output[b], (1, 2, 0)))
        imsave("Images/"+ name + "_" + str(b) + "_response" + ".png", mapp[b])
```

Couches Fonctionnelles

```python
def gain (self, expect):
    B, D, H, W = self.size
    out = np.copy(np.reshape(self.output, np.shape(expect)))
    ga = expect
    self.term = np.reshape(ga, np.shape(self.term))
    return np.sum(out*ga, axis = 1)

def score(self, expect):
    B, D, H, W = self.size
    out = np.copy(np.reshape(self.output, np.shape(expect)))
    points = 0
    for b in range(B):
        if np.argmax(out[b]) == np.argmax(expect[b]):
            points += 1
    return points
```

Exemple d’apprentissage sur CIFAR-10

```python
def Train_CIFAR():
    lay_list = [(Input, (3, 32, 32)),
                (Convolutional, (16, 5, 1, 2, (0.01, 0.9, 0.0001))), (Relu, 0),
                (Pooling, (2, 2)),
                (Convolutional, (20, 5, 1, 2, (0.01, 0.9, 0.0001))), (Relu, 0),
                (Pooling, (2, 2)),
                (Convolutional, (20, 5, 1, 2, (0.01, 0.9, 0.0001))), (Relu, 0),
                (Pooling, (2, 2)),
                (FullyConnected, (1, 1, 10, (0.01, 0.9, 0.0001))), (Softmax, 0)]

    cat_list = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]
    net = Network("Test_CIFAR", 5, lay_list, cat_list)
    net.define()

    net.load()
    for b in range(1, 6):
        loaded = np.load("Data/Data_Brut/batch_" + str(b) + ".npz")
        dataset = loaded["f"], loaded["l"]
        print("################################ BATCH : " + str(b) + " ################################")
        net.training((b, dataset), (0, 10000))
        net.save()

        loaded = np.load("Data/Data_Brut/batch_test.npz")
        dataset = loaded["f"], loaded["l"]
        net.verify(dataset)
        net.save()
        print(net.accuracy)
```

Récupération des données du Loss et de l'Accuracy

```
def print_error():
    loaded = np.load("Test_CIFAR_save/Loss.npz")
    loss = loaded["l"]
    loaded = np.load("Test_CIFAR_save/Accuracy.npz")
    accuracy = loaded["a"]
    l = len(loss)
    p = len(accuracy)
    n = 1000

    err = np.array([])
    x = np.array([])
    for i in range(n):
        err = np.append(err, np.sum(loss[int(i*l/n) : int((i+1)*l/n)])/l*n)
        x = np.append(x, i)

    y = np.array([])
    for i in range(p):
        y = np.append(y, i*n/p)

    plt.plot(x, err)
    plt.plot(y, accuracy)
    plt.show()
```

Optimisation des fonctions avec Numpy

Création de tableaux aléatoires et analyse des résultats :

```
dim2 = (D, H, W)
dim1 = (I, J, K)
dim = (D, H, W, I, J, K)

inp = np.zeros(dim1)
weigths = np.zeros(dim)
out1 = np.zeros(dim2)
out2 = np.zeros(dim2)
out3 = np.zeros(dim2)

for i in range(I):
    for j in range(J):
        for k in range(K):
            inp[i, j, k] = random.randint(-4, 4)

for d in range(D):
    for h in range(H):
        for w in range(W):
            for i in range(I):
                for j in range(J):
                    for k in range(K):
                        weigths[d, h, w, i, j, k] = random.randint(-4, 4)
```

```
# sortie console
print("\nOut : " + str(np.all(out1 == out2) and np.all(out2 == out3) and np.all(out3 == out4)) + "\n")
print("technique 1 : " + str(end1 - start1) + "\ntechnique 2 : " + str(end2 - start2) + "\ntechnique 3 : "
```

FullyConnected : front()

```
# technique 1
start1 = time.clock()
for n in range(N):
    out1 = out1 - out1
    for d in range(D):
        for h in range(H):
            for w in range(W):
                for i in range(I):
                    for j in range(J):
                        for k in range(K):
                            out1[d, h, w] += inp[i, j, k] * weigths[d, h, w, i, j, k]
end1 = time.clock()
```

```
# technique 2
start2 = time.clock()
for n in range(N):
    for d in range(D):
        for h in range(H):
            for w in range(W):
                out2[d, h, w] = np.sum(weigths[d, h, w] * inp)
end2 = time.clock()
```

```
# technique 3
start3 = time.clock()
for n in range(N):
    out3 = np.sum(np.sum(np.sum(weigths * inp, axis = 5), axis = 4), axis = 3)
end3 = time.clock()
```

```
# technique 4
start4 = time.clock()
for n in range(N):
    out4 = np.einsum('...ijk, ijk', weigths, inp)
end4 = time.clock()
```

FullyConnected : back()

```
# technique 1
start1 = time.clock()
for n in range(N):
    prev_term1 = prev_term1 - prev_term1
    for d in range(D):
        for h in range(H):
            for w in range(W):
                for i in range(I):
                    for j in range(J):
                        for k in range(K):
                            delta1[d, h, w, i, j, k] = prev_out[i, j, k] * term[d, h, w]
                            prev_term1[i, j, k] += term[d, h, w] * weigths[d, h, w, i, j, k]
end1 = time.clock()
```

```
# technique 2
start2 = time.clock()
for n in range(N):
    prev_term2 = prev_term2 - prev_term2
    for d in range(D):
        for h in range(H):
            for w in range(W):
                delta2[d, h, w] = prev_out * term[d, h, w]
                prev_term2 += term[d, h, w] * weigths[d, h, w]
end2 = time.clock()
```

```
# technique 3
start3 = time.clock()
for n in range(N):
    delta3 = np.einsum('ijk,...', prev_out, term)
    prev_term3 = np.sum(np.sum(np.sum(np.transpose(np.transpose(term) * np.transpose(weigths)), axis = 0), axis = 0), axis = 0)
end3 = time.clock()
```

```
# technique 4
start4 = time.clock()
for n in range(N):
    delta4 = np.einsum('ijk,...', prev_out, term)
    prev_term4 = np.einsum('dhw,dhw...', term, weigths)
end4 = time.clock()
```

Convolutional : front()

```python
# technique 1 :
start1 = time.clock()
for n in range(N):
    out1 = out1 - out1
    for d in range(D):
        for h in range(H):
            for w in range(W):
                out1[d, h, w] += bias[d]
                for i in range(I):
                    for j in range(F):
                        for k in range(F):
                            out1[d, h, w] += pad_inp[i, h*S+j, w*S+k] * weigths[d, i, j, k]
end1 = time.clock()
```

```python
# technique 2 :
start2 = time.clock()
for n in range(N):
    for d in range(D):
        for h in range(H):
            for w in range(W):
                out2[d, h, w] = np.sum(weigths[d] * pad_inp[:, h*S:h*S+F, w*S:w*S+F]) + bias[d]
end2 = time.clock()
```

```python
# technique 3 :
start3 = time.clock()
for n in range(N):
    for h in range(H):
        for w in range(W):
            out3[:, h, w] = np.sum(np.sum(np.sum(weigths * pad_inp[:, h*S:h*S+F, w*S:w*S+F], axis = 3), axis = 2), axis = 1) + bias
end3 = time.clock()
```

```python
# technique 4 :
start4 = time.clock()
for n in range(N):
    for h in range(H):
        for w in range(W):
            out4[:, h, w] = np.einsum('...ijk,ijk', weigths, pad_inp[:, h*S:h*S+F, w*S:w*S+F]) + bias
end4 = time.clock()
```

Convolutional : back()

```python
# technique 1 :
start1 = time.clock()
for n in range(N):
    prev_term1 = prev_term1 - prev_term1
    delta1 = delta1 - delta1
    delta_bias1 = delta_bias1 - delta_bias1
    for d in range(D):
        for h in range(H):
            for w in range(W):
                delta_bias1[d] += term[d, h, w]
                for i in range(I):
                    for j in range(F):
                        for k in range(F):
                            delta1[d, i, j, k] += pad_prev_out[i, h*S+j, w*S+k] * term[d, h, w]
                            prev_term1[i, h*S+j, w*S+k] += term[d, h, w] * weigths[d, i, j, k]
end1 = time.clock()
```

```python
# technique 2 :
start2 = time.clock()
for n in range(N):
    prev_term2 = prev_term2 - prev_term2
    delta2 = delta2 - delta2
    delta_bias2 = delta_bias2 - delta_bias2
    for d in range(D):
        for h in range(H):
            for w in range(W):
                delta2[d] += pad_prev_out[:, h*S:h*S+F, w*S:w*S+F] * term[d, h, w]
                prev_term2[:, h*S:h*S+F, w*S:w*S+F] += term[d, h, w] * weigths[d]
                delta_bias2[d] += term[d, h, w]
end2 = time.clock()
```

```python
# technique 3 :
start3 = time.clock()
for n in range(N):
    prev_term3 = prev_term3 - prev_term3
    delta3 = delta3 - delta3
    delta_bias3 = np.sum(np.sum(term, axis = 2), axis = 1)
    for h in range(H):
        for w in range(W):
            prev_term3[:, h*S:h*S+F, w*S:w*S+F] += np.sum(np.transpose(np.transpose(term[:, h, w]) * np.transpose(weigths)), axis = 0)
            delta3 += np.einsum('ijk,...', pad_prev_out[:, h*S:h*S+F, w*S:w*S+F], term[:, h, w])
end3 = time.clock()
```

```
# technique 4 :
start4 = time.clock()
for n in range(N):
    prev_term4 = np.zeros(np.shape(prev_term4))
    delta4 = np.zeros(np.shape(delta4))
    delta_bias4 = np.sum(np.sum(term, axis = 2), axis = 1)
    for h in range(H):
        for w in range(W):
            prev_term4[:, h*S:h*S+F, w*S:w*S+F] += np.einsum('d..., d...', term[:, h, w], weigths)
            delta4 += np.einsum('ijk,...', pad_prev_out[:, h*S:h*S+F, w*S:w*S+F], term[:, h, w])
end4 = time.clock()
```

Pooling : front()

```
# technique 1 :
start1 = time.clock()
for n in range(N):
    for d in range(I):
        for h in range(H):
            for w in range(W):
                out1[d, h, w] = np.max(inp[d, h*S:h*S+F, w*S:w*S+F])
end1 = time.clock()
```

```
# technique 2 :
start2 = time.clock()
for n in range(N):
    for h in range(H):
        for w in range(W):
            out2[:, h, w] = np.max(np.max(inp[:, h*S:h*S+F, w*S:w*S+F], axis = 1), axis = 1)
end2 = time.clock()
```

Pooling : back()

```
# technique 1 :
start1 = time.clock()
for n in range(N):
    prev_term1 = np.zeros(np.shape(prev_term1))
    for d in range(I):
        for h in range(H):
            for w in range(W):
                for j in range(h*S, h*S + F):
                    for k in range(w*S, w*S + F):
                        if prev_out[d, j, k] == out[d, h, w] :
                            prev_term1[d, j, k] += term[d, h, w]

end1 = time.clock()
```

```
# technique 2 :
start2 = time.clock()
for n in range(N):
    prev_term2 = np.zeros(np.shape(prev_term2))
    for d in range(I):
        for h in range(H):
            for w in range(W):
                M = prev_out[d, h*S:h*S+F, w*S:w*S+F] - out[d, h, w]
                M[M == 0] = 1
                M[M != 1] = 0
                prev_term2[d, h*S:h*S+F, w*S:w*S+F] += term[d, h, w]*M
end2 = time.clock()
```

```
# technique 3 :
start3 = time.clock()
ones = np.ones((F, F))
for n in range(N):
    prev_term3 = np.zeros(np.shape(prev_term3))
    for h in range(H):
        for w in range(W):
            M = prev_out[:, h*S:h*S+F, w*S:w*S+F] - np.einsum('...,ij' ,out[:, h, w], ones)
            M[M == 0] = 1
            M[M != 1] = 0
            prev_term3[:, h*S:h*S+F, w*S:w*S+F] += np.einsum('...,...ij', term[:, h, w], M)
end3 = time.clock()
```

```python
# technique 1
start1 = time.clock()
for n in range(N):
    out1 = np.zeros(np.shape(out1))
    for b in range(B):
        for d in range(D):
            for h in range(H):
                for w in range(W):
                    out1[b, d, h, w] += bias[d, h, w]
                    for i in range(I):
                        for j in range(J):
                            for k in range(K):
                                out1[b, d, h, w] += inp[b, i, j, k] * weigths[d, h, w, i, j, k]
end1 = time.clock()
```

```python
# technique 2
start2 = time.clock()
for n in range(N):
    out2 = np.zeros(np.shape(out2))
    for b in range(B):
        for d in range(D):
            for h in range(H):
                for w in range(W):
                    out2[b, d, h, w] = np.sum(weigths[d, h, w] * inp[b]) + bias[d, h, w]
end2 = time.clock()
```

```python
# technique 3
start3 = time.clock()
for n in range(N):
    for b in range(B):
        out3[b] = np.einsum('ijk,...ijk', inp[b], weigths) + bias
end3 = time.clock()
```

FullyConnected : front()

```python
# technique 4
start4 = time.clock()
distrib = np.ones((B))
for n in range(N):
    out4 = np.transpose(np.einsum('bijk,...ijk', inp, weigths), axes = [3, 0, 1, 2])
    + np.einsum('dhw,...', bias, distrib)
end4 = time.clock()
```

FullyConnected : back()

```python
# technique 1
start1 = time.clock()
for n in range(N):
    prev_term1 = np.zeros(np.shape(prev_term1))
    delta1 = np.zeros(np.shape(delta1))
    for b in range(B):
        for d in range(D):
            for h in range(H):
                for w in range(W):
                    for i in range(I):
                        for j in range(J):
                            for k in range(K):
                                delta1[d, h, w, i, j, k] += prev_out[b, i, j, k] * term[b, d, h, w]
                                prev_term1[b, i, j, k] += term[b, d, h, w] * weigths[d, h, w, i, j, k]
end1 = time.clock()
```

```python
# technique 2
start2 = time.clock()
for n in range(N):
    prev_term2 = np.zeros(np.shape(prev_term2))
    delta2 = np.zeros(np.shape(delta2))
    for d in range(D):
        for h in range(H):
            for w in range(W):
                for b in range(B):
                    delta2[d, h, w] += prev_out[b] * term[b, d, h, w]
                    prev_term2[b] += term[b, d, h, w] * weigths[d, h, w]
end2 = time.clock()
```

```python
# technique 3
start3 = time.clock()
for n in range(N):
    delta3 = np.zeros(np.shape(delta3))
    for b in range(B):
        delta3 += np.einsum('ijk,...', prev_out[b], term[b])
        prev_term3[b] = np.einsum('dhw,dhw...', term[b], weigths)
end3 = time.clock()
```

```python
# technique 4
start4 = time.clock()
for n in range(N):
    delta4 = np.einsum('bijk,b...', prev_out, term)
    prev_term4 = np.transpose(np.einsum('bdhw,dhw...', term, weigths), axes = [3, 0, 1, 2])
end4 = time.clock()
```

Convolutional : front()

```
# technique 1 :
start1 = time.clock()
for n in range(N):
    out1 = np.zeros(np.shape(out1))
    for b in range(B):
        for d in range(D):
            for h in range(H):
                for w in range(W):
                    out1[b, d, h, w] += bias[d]
                    for i in range(I):
                        for j in range(F):
                            for k in range(F):
                                out1[b, d, h, w] += pad_inp[b, i, h*S+j, w*S+k] * weigths[d, i, j, k]
end1 = time.clock()
```

```
# technique 2 :
start2 = time.clock()
for n in range(N):
    for b in range(B):
        for d in range(D):
            for h in range(H):
                for w in range(W):
                    out2[b, d, h, w] = np.sum(weigths[d] * pad_inp[b, :, h*S:h*S+F, w*S:w*S+F]) + bias[d]
end2 = time.clock()
```

```
# technique 3 :
start3 = time.clock()
for n in range(N):
    for h in range(H):
        for w in range(W):
            for b in range(B):
                out3[b, :, h, w] = np.einsum('...ijk,ijk', weigths, pad_inp[b, :, h*S:h*S+F, w*S:w*S+F]) + bias
end3 = time.clock()
```

```
# technique 4 :
start4 = time.clock()
distrib = np.ones((B))
for n in range(N):
    for h in range(H):
        for w in range(W):
            out4[:, :, h, w] = np.transpose(np.einsum('...ijk,bijk', weigths, pad_inp[:, :, h*S:h*S+F, w*S:w*S+F]))
            + np.einsum('d,...', bias, distrib)
end4 = time.clock()
```

Convolutional : back()

```
# technique 1 :
start1 = time.clock()
for n in range(N):
    prev_term1 = np.zeros(np.shape(prev_term1))
    delta1 = np.zeros(np.shape(delta1))
    delta_bias1 = np.zeros(np.shape(delta_bias1))
    for b in range(B):
        for d in range(D):
            for h in range(H):
                for w in range(W):
                    delta_bias1[d] += term[b, d, h, w]
                    for i in range(I):
                        for j in range(F):
                            for k in range(F):
                                delta1[d, i, j, k] += pad_prev_out[b, i, h*S+j, w*S+k] * term[b, d, h, w]
                                prev_term1[b, i, h*S+j, w*S+k] += term[b, d, h, w] * weigths[d, i, j, k]
end1 = time.clock()
```

```
# technique 2 :
start2 = time.clock()
for n in range(N):
    prev_term2 = np.zeros(np.shape(prev_term2))
    delta2 = np.zeros(np.shape(delta2))
    delta_bias2 = np.zeros(np.shape(delta_bias2))
    for b in range(B):
        for d in range(D):
            for h in range(H):
                for w in range(W):
                    delta2[d] += pad_prev_out[b, :, h*S:h*S+F, w*S:w*S+F] * term[b, d, h, w]
                    prev_term2[b, :, h*S:h*S+F, w*S:w*S+F] += term[b, d, h, w] * weigths[d]
                    delta_bias2[d] += term[b, d, h, w]
end2 = time.clock()
```

```
# technique 3 :
start3 = time.clock()
for n in range(N):
    prev_term3 = np.zeros(np.shape(prev_term3))
    delta3 = np.zeros(np.shape(delta3))
    for b in range(B):
        for h in range(H):
            for w in range(W):
                prev_term3[b, :, h*S:h*S+F, w*S:w*S+F] += np.einsum('d..., d...', term[b, :, h, w], weigths)
                delta3 += np.einsum('ijk,...', pad_prev_out[b, :, h*S:h*S+F, w*S:w*S+F], term[b, :, h, w])
    delta_bias3 = np.sum(term, axis = (3, 2, 0))
end3 = time.clock()
```

```python
# technique 4 :
start4 = time.clock()
for n in range(N):
    prev_term4 = np.zeros(np.shape(prev_term4))
    delta4 = np.zeros(np.shape(delta4))
    for h in range(H):
        for w in range(W):
            prev_term4[:, :, h*S:h*S+F, w*S:w*S+F] += np.transpose(np.einsum('bd..., d...', term[:, :, h, w], weigths), axes = [3, 0, 1, 2])
            delta4 += np.einsum('bijk,b...', pad_prev_out[:, :, h*S:h*S+F, w*S:w*S+F], term[:, :, h, w])
    delta_bias4 = np.sum(term, axis = (3, 2, 0))
end4 = time.clock()
```

Pooling : front()

```python
# technique 1 :
start1 = time.clock()
for n in range(N):
    for b in range(B):
        for d in range(I):
            for h in range(H):
                for w in range(W):
                    out1[b, d, h, w] = np.max(inp[b, d, h*S:h*S+F, w*S:w*S+F])
end1 = time.clock()
```

```python
# technique 2 :
start2 = time.clock()
for n in range(N):
    for b in range(B):
        for h in range(H):
            for w in range(W):
                out2[b, :, h, w] = np.max(np.max(inp[b, :, h*S:h*S+F, w*S:w*S+F], axis = 1), axis = 1)
end2 = time.clock()
```

```python
# technique 3 :
start3 = time.clock()
for n in range(N):
    for h in range(H):
        for w in range(W):
            out3[:, :, h, w] = np.max(np.max(inp[:, :, h*S:h*S+F, w*S:w*S+F], axis = 2), axis = 2)
end3 = time.clock()
```

Pooling : back()

```python
# technique 1 :
start1 = time.clock()
for n in range(N):
    prev_term1 = np.zeros(np.shape(prev_term1))
    for b in range(B):
        for d in range(I):
            for h in range(H):
                for w in range(W):
                    for j in range(h*S, h*S + F):
                        for k in range(w*S, w*S + F):
                            if prev_out[b, d, j, k] == out[b, d, h, w] :
                                prev_term1[b, d, j, k] += term[b, d, h, w]

end1 = time.clock()
```

```python
# technique 2 :
start2 = time.clock()
ones = np.ones((F, F))
for n in range(N):
    prev_term2 = np.zeros(np.shape(prev_term2))
    for b in range(B):
        for h in range(H):
            for w in range(W):
                M = prev_out[b, :, h*S:h*S+F, w*S:w*S+F] - np.einsum('...,ij' ,out[b, :, h, w], ones)
                M[M == 0] = 1
                M[M != 1] = 0
                prev_term2[b, :, h*S:h*S+F, w*S:w*S+F] += np.einsum('...,...ij', term[b, :, h, w], M)
end2 = time.clock()
```

```python
# technique 3 :
start3 = time.clock()
ones = np.ones((F, F))
for n in range(N):
    prev_term3 = np.zeros(np.shape(prev_term3))
    for h in range(H):
        for w in range(W):
            M = prev_out[:, :, h*S:h*S+F, w*S:w*S+F]
            - np.transpose(np.einsum('b...,ij' ,out[:, :, h, w], ones), axes = [1, 0, 3, 2])
            M[M == 0] = 1
            M[M != 1] = 0
            prev_term3[:, :, h*S:h*S+F, w*S:w*S+F] += np.einsum('...,...ij', term[:, :, h, w], M)
end3 = time.clock()
```