TIPE

Réseau de neurones et application à la reconnaissance de panneaux routiers

Dupuis Benjamin

Numéro candidat: 10367

Introduction (1/6)

Sommaire

Problématique

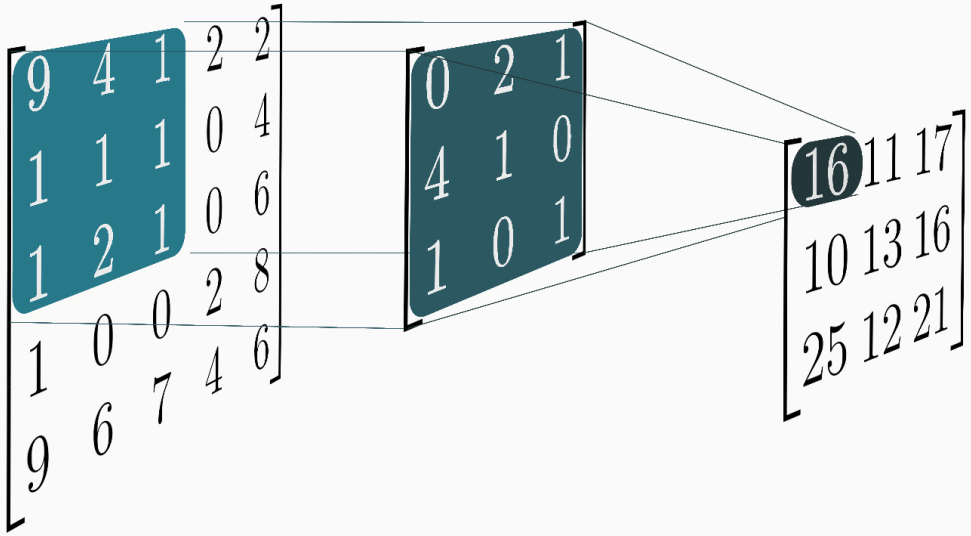
Les réseaux de neurones peuvent-ils permettre de reconnaitre des panneaux routiers ?

Source : Faming Shao, https://doi.org/10.3390/s18103192 [9]

Objectifs:

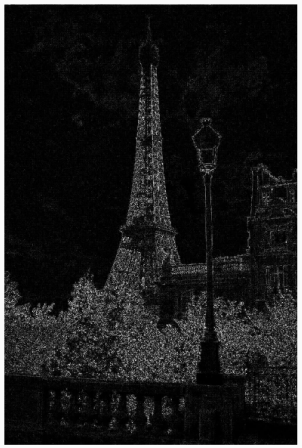
- .Compréhension
- .Implémentation
- .Optimisation et application

Filtres: présentation



$$(A * B)[1,1] = \underline{9}\times\underline{0} + \underline{4}\times\underline{2} + \underline{1}\times\underline{1} + \underline{1}\times\underline{4} + \underline{1}\times\underline{1} + \underline{1}\times\underline{0} + \underline{1}\times\underline{1} + \underline{2}\times\underline{0} + \underline{1}\times\underline{1} = \underline{16}$$

$$* \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} =$$



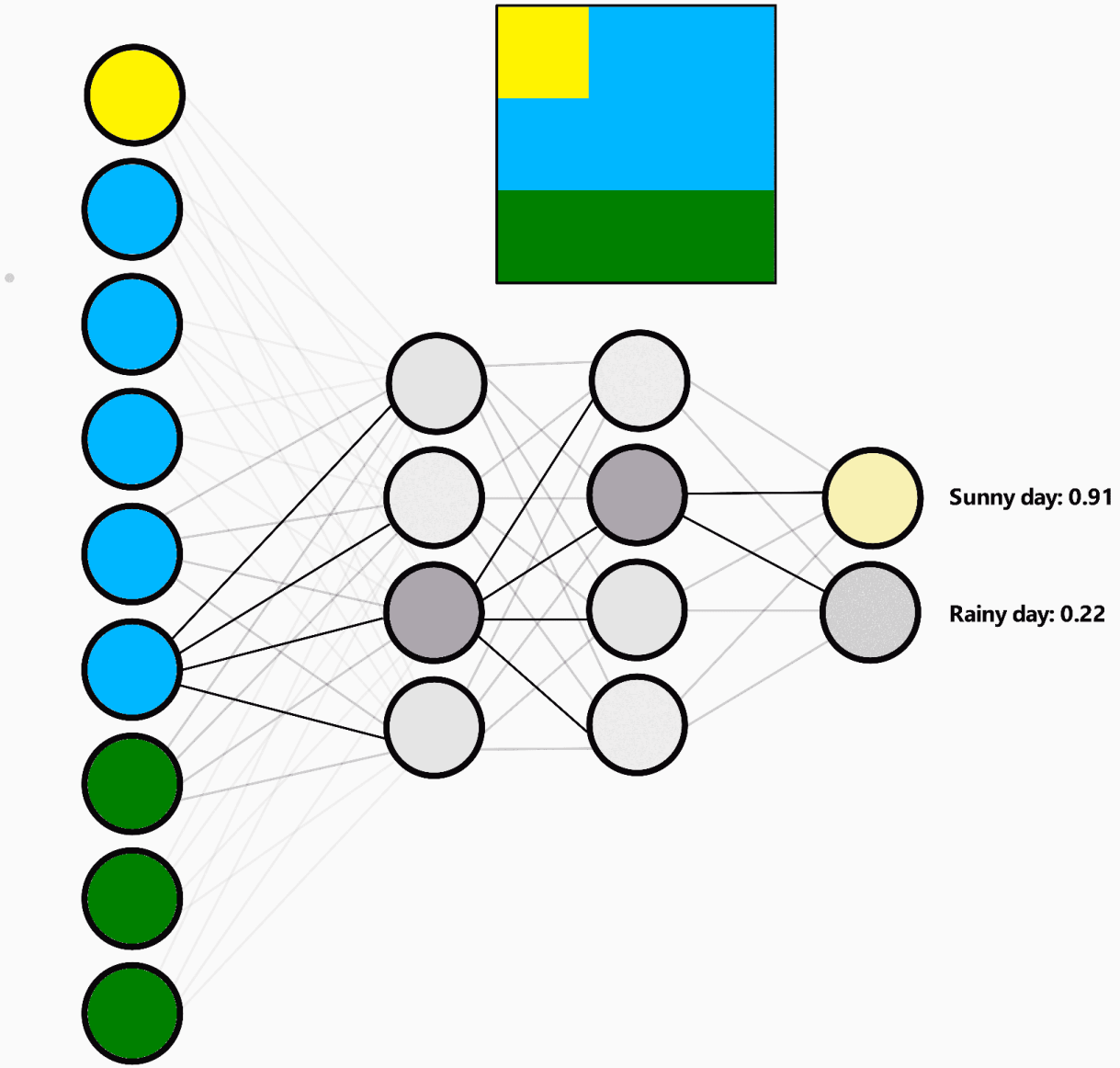
Reconnaissance de bords

$$* \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} =$$

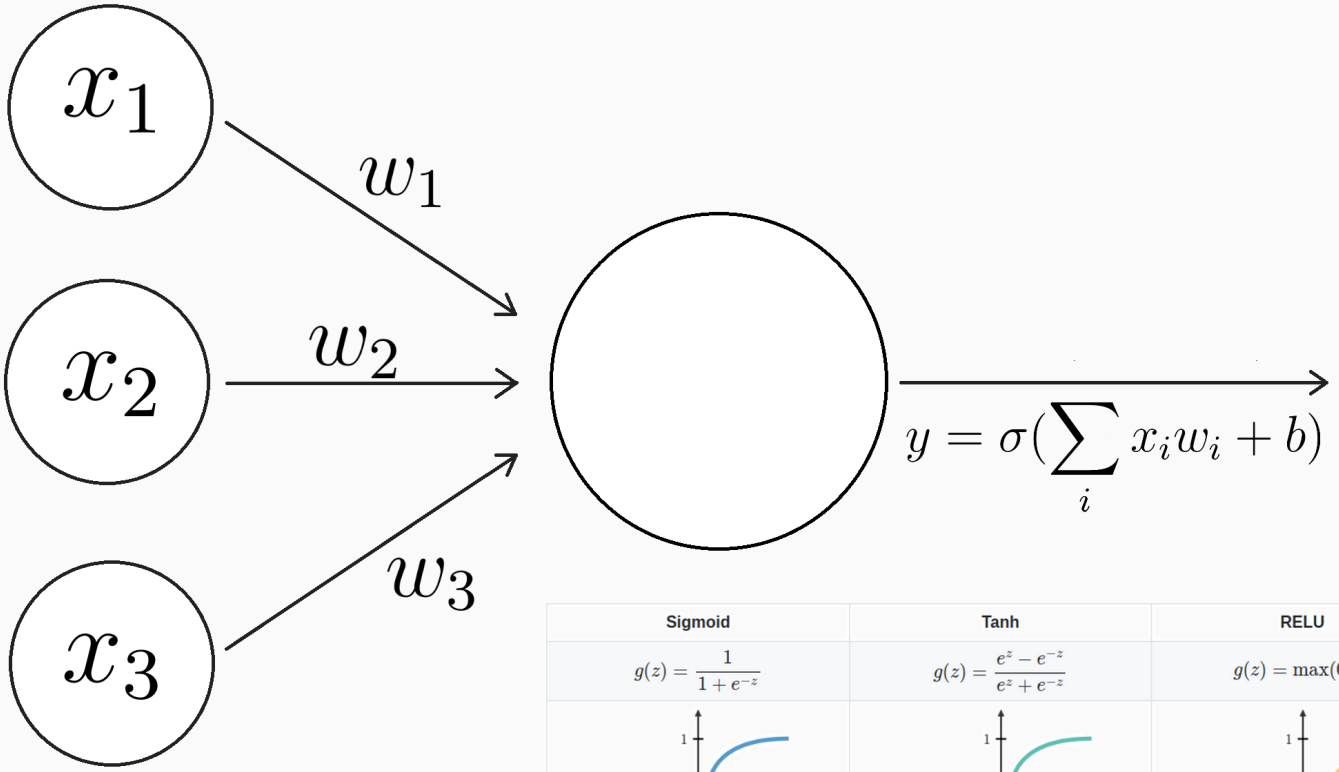
Floutage léger

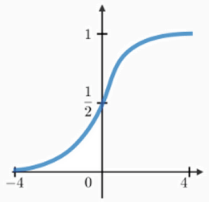
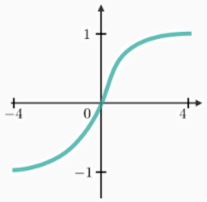
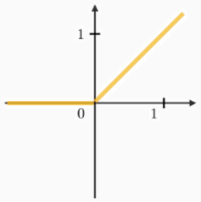
Présentation (2/6)

Objectif d'un réseau



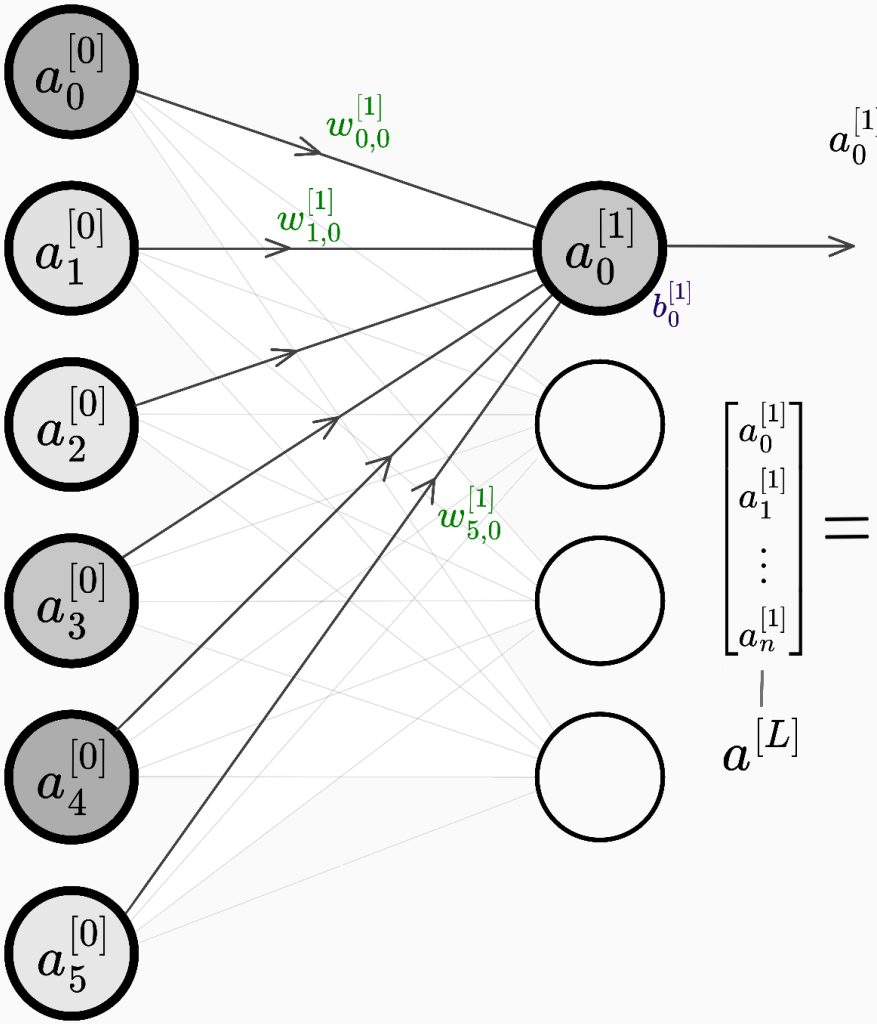
Le neurone artificiel (perceptron)



Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$
		

Caractéristiques: .Non-linéarité .Différentiabilité .Rapidité de calcul

Propagation avant (FeedForward)



$$a_0^{[1]} = \phi(a_0^{[0]} w_{0,0}^{[1]} + \ldots + a_5^{[0]} w_{5,0}^{[1]} + b_0^{[1]})$$

$$\underbrace{\begin{bmatrix} a_0^{[1]} \\ a_1^{[1]} \\ \vdots \\ a_n^{[1]} \end{bmatrix}}_{a^{[L]}} = \phi\left(\underbrace{\begin{bmatrix} w_{0,0} & w_{0,1} & \ldots & w_{0,n} \\ w_{1,0} & w_{1,1} & \ldots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \ldots & w_{k,n} \end{bmatrix}}_{w^{[L]}} \underbrace{\begin{bmatrix} a_0^{[0]} \\ a_1^{[0]} \\ \vdots \\ a_n^{[0]} \end{bmatrix}}_{a^{[L-1]}} + \underbrace{\begin{bmatrix} b_0^{[1]} \\ b_1^{[1]} \\ \vdots \\ b_n^{[1]} \end{bmatrix}}_{b^{[L]}} \right)$$

Fonction coût

.S'applique a une image pré-labellisée

.Mesure l'erreur sur une prédiction

.Différentiable

.Fonctions coût classiques:

.Distance euclidienne: $C_{W,B}(y) = ||y - \hat{y}||_2^2$

.Entropie-croisée:

$C_{W,B}(y) == -\sum_i \hat{y}_i * ln(y_i) + (1 - \hat{y}_i) * ln(1 - y_i)$

Descente du gradient

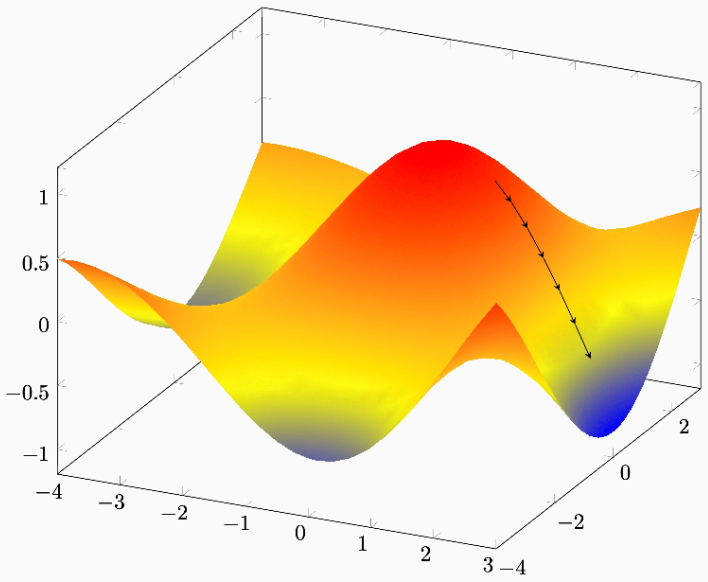


Illustration : Descente du gradient pour une fonction de deux variables.

Calcul des dérivées partielles:

.

Par rapport à $a_j^{[k]}$:

$$\frac{\partial C}{\partial a_j^{[k]}} = \sum_i \frac{\partial C}{\partial a_i^{[k+1]}} \frac{\partial a_i^{[k+1]}}{\partial a_j^{[k]}} = \sum_i \frac{\partial C}{\partial a_i^{[k+1]}} . \sigma'(a_i^{[k+1]}) . w_{j,i}^{[k]}$$

.

Par rapport à $w_{i,j}^{[k]}$:

$$\frac{\partial C}{\partial w_{i,j}^{[k]}} = \frac{\partial C}{\partial a_j^{[k]}} \frac{\partial a_j^{[k]}}{\partial w_{i,j}^{[k]}} = \frac{\partial C}{\partial a_j^{[k]}} . \sigma'(a_j^{[k]})$$

.

Par rapport à $b_j^{[k]}$:

$$\frac{\partial C}{\partial b_j^{[k]}} = \frac{\partial C}{\partial a_j^{[k]}} \frac{\partial a_j^{[k]}}{\partial b_j^{[k]}} = \frac{\partial C}{\partial a_j^{[k]}} . \sigma'(a_j^{[k]})$$

Données et premières tentatives (3/6)

Premier essai

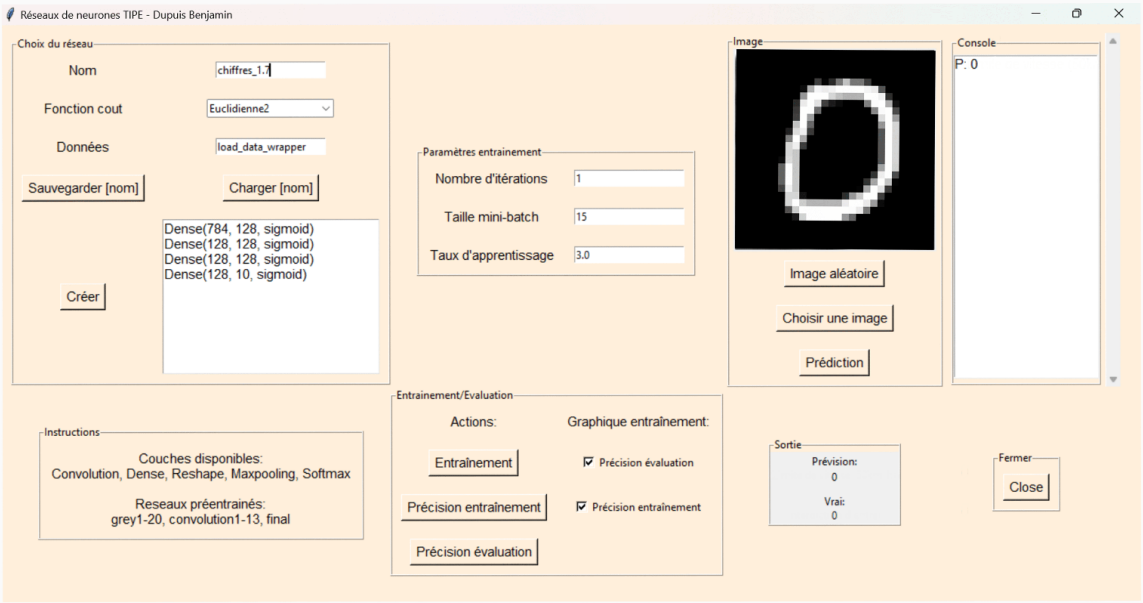
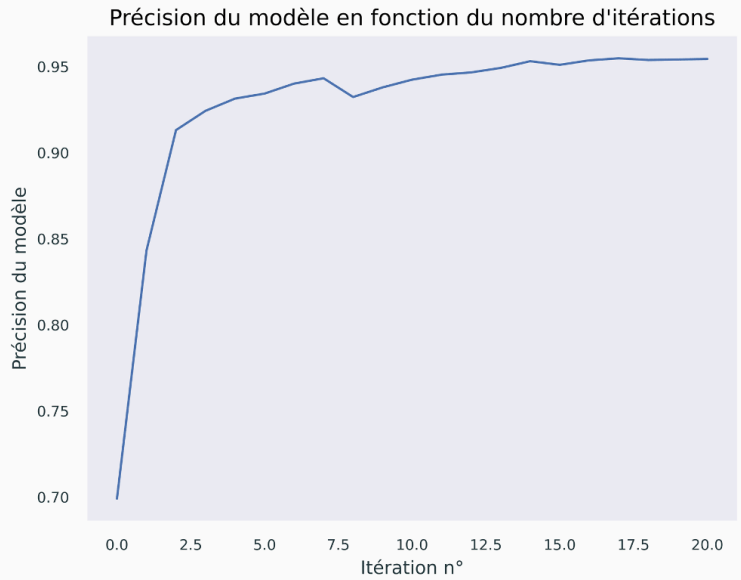
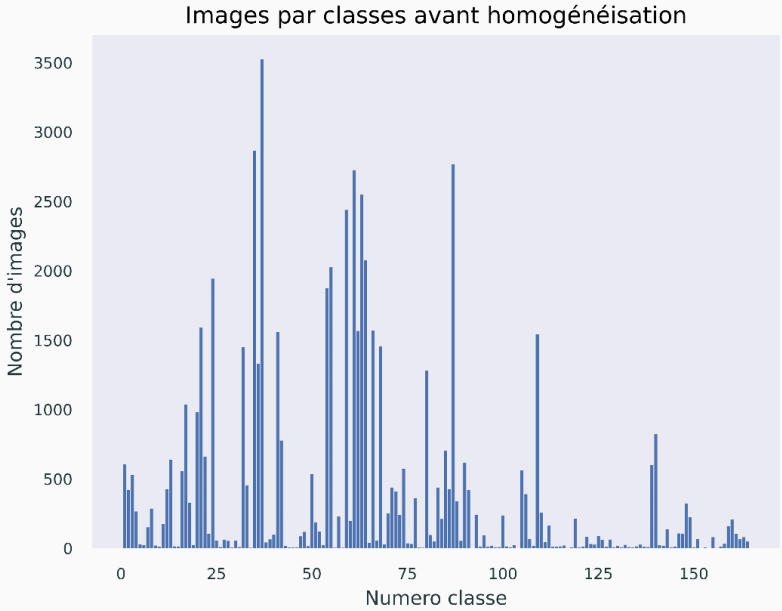


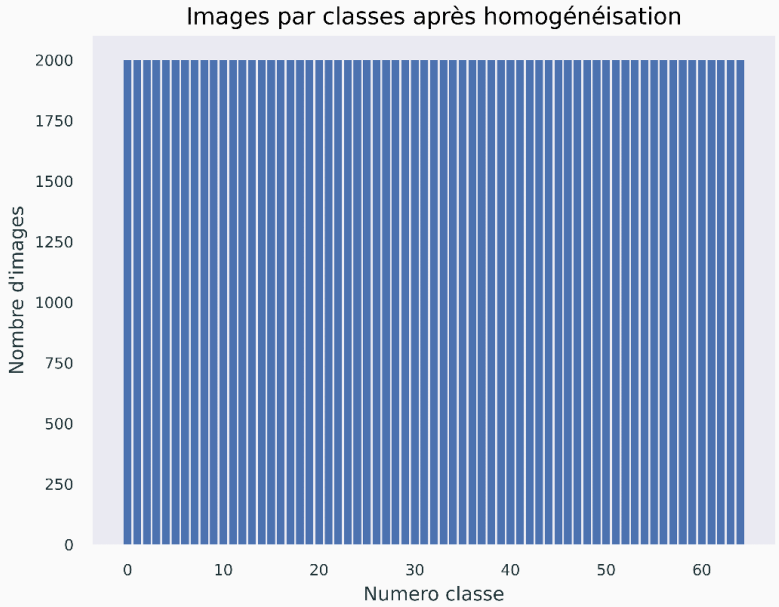
Illustration : Interface permettant de superviser les paramètres d'apprentissage



Graphique : Performances d'un modèle (2 couches de 128 neurones)

Création d'un jeu de données

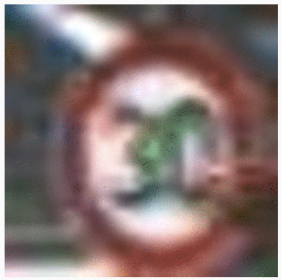




.164 Classes puis 65 Classes

.Entraînement: 130.000 images
.Evaluation: 19.684 images

.Allemagne, Belgique, Croatie
.Espagne, France, Pays-Bas



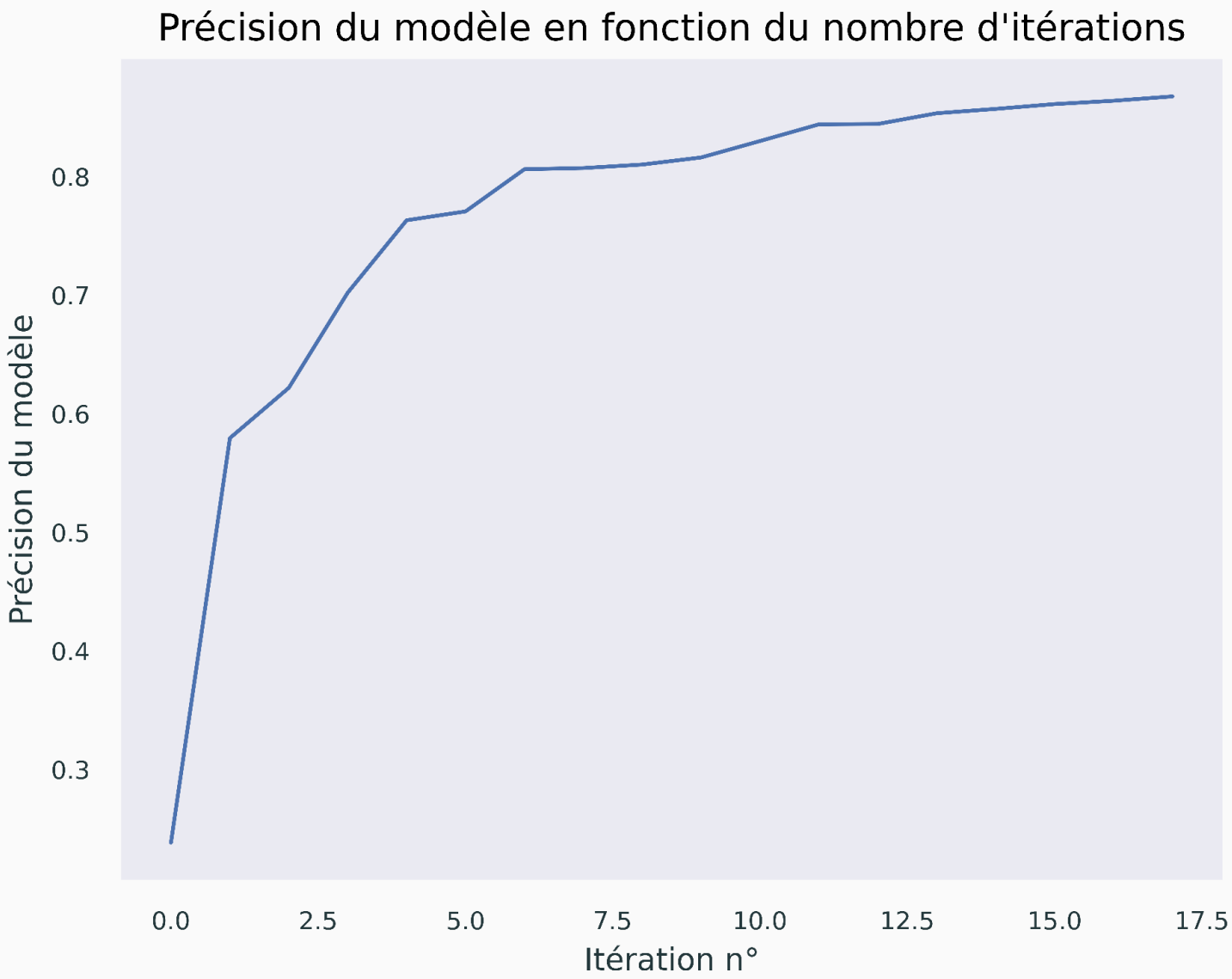
Limitation de vitesse (30 km/h)

Contournement par la gauche

Cédez-le-passage

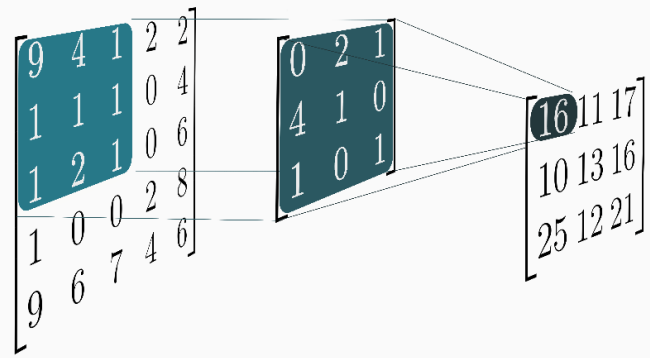
Virage à gauche

Statistiques



Améliorations et modèle expérimental (4/6)

Couche de convolution



$$(A * B)[1,1] = \underline{9}\times\underline{0} + \underline{4}\times\underline{2} + \underline{1}\times\underline{1} + \underline{1}\times\underline{4} + \underline{1}\times\underline{1} + \underline{1}\times\underline{0} + \underline{1}\times\underline{1} + 2\times\underline{0} + \underline{1}\times\underline{1} = \underline{16}$$

Illustration : Opération de Convolution

Soient $m, n \in \mathbb{N}^*$ tel que:

$$.A \in \mathcal{M}_n(\mathbb{R})$$

$$.B \in \mathcal{M}_m(\mathbb{R})$$

$$.A * B = M \in \mathcal{M}_{n-m+1}(\mathbb{R})$$

.

Propagation avant:

$$\forall (i,j) \in [1,m]^2,$$

$$M[i,j] = \sum_{l=1}^{m} \sum_{c=1}^{m} A[l+i-1, c+j-1]B[l,c]$$

.

On peut alors calculer la dérivée partielle de la fonction coût par rapport à B[l,c]:

$$\frac{\partial C}{\partial B[l,c]} = \frac{\partial C}{\partial M[i,j]} \frac{\partial M[i,j]}{\partial B[l,c]} = \frac{\partial C}{\partial M[i,j]} A[l+i-1, c+j-1]$$

.

Résumé

Création d'un réseau

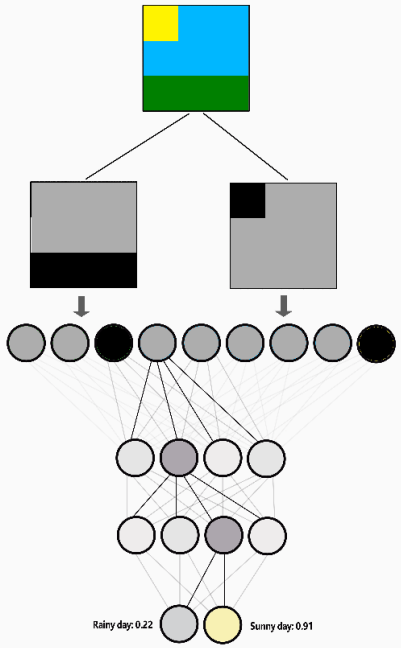
Choix d'une structure de réseau, de ses caractéristiques générales.

Entraînement sur une image

Pour une image labellisée, propagation de celle-ci et ajustement des poids en conséquence.

Entraînement du réseau

Itération sur toutes les images d'entraînement un certain nombre de fois.



Expérimentations et interprétations (5/6)

Liste paramètres

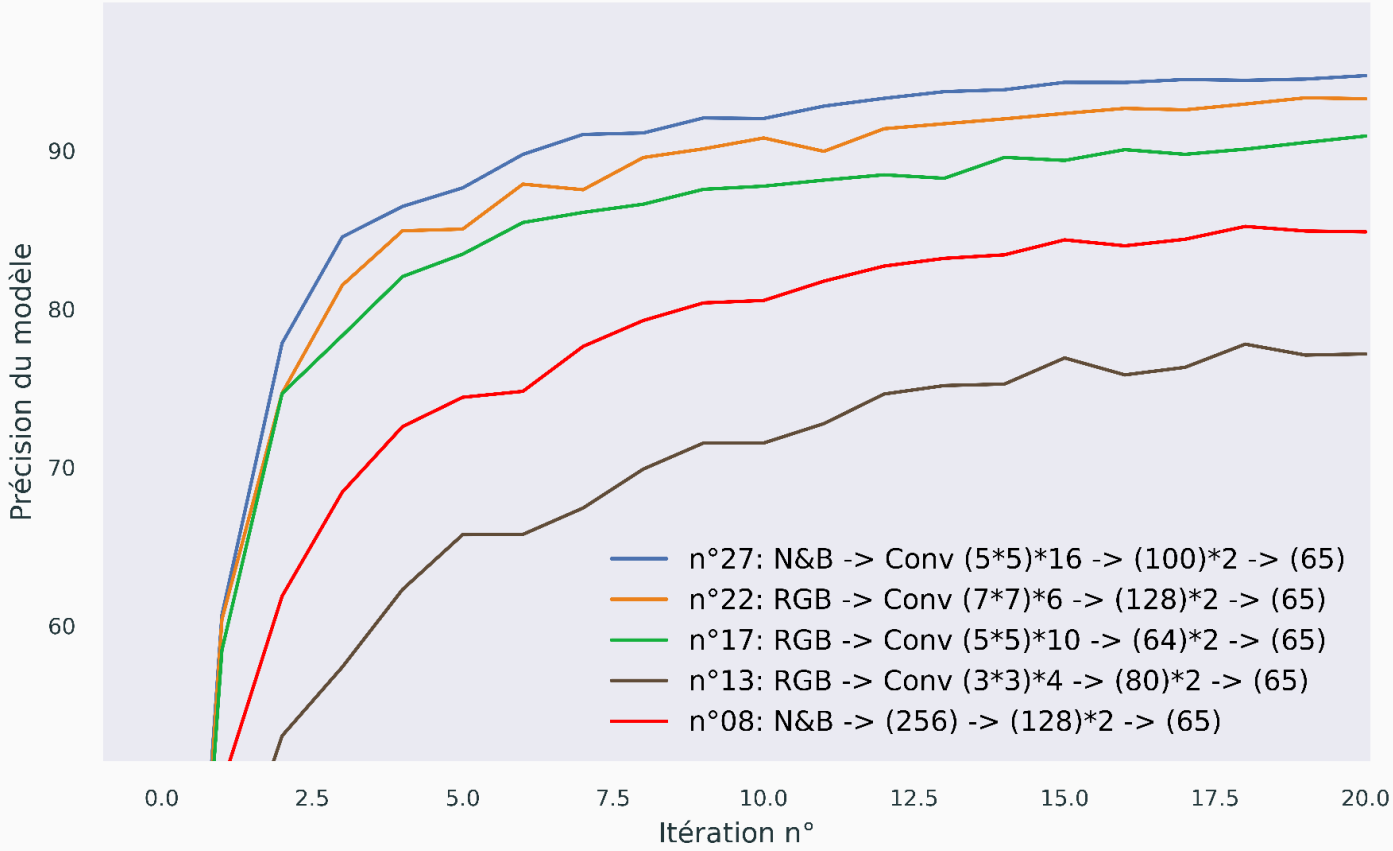
Paramètres du réseau

- .Choix des couches
- .Fonctions d'activation

Paramètres d'entraînement

- .Fonction coût
- .Choix du pas
- .Nombre d'itérations

Choix structure réseau



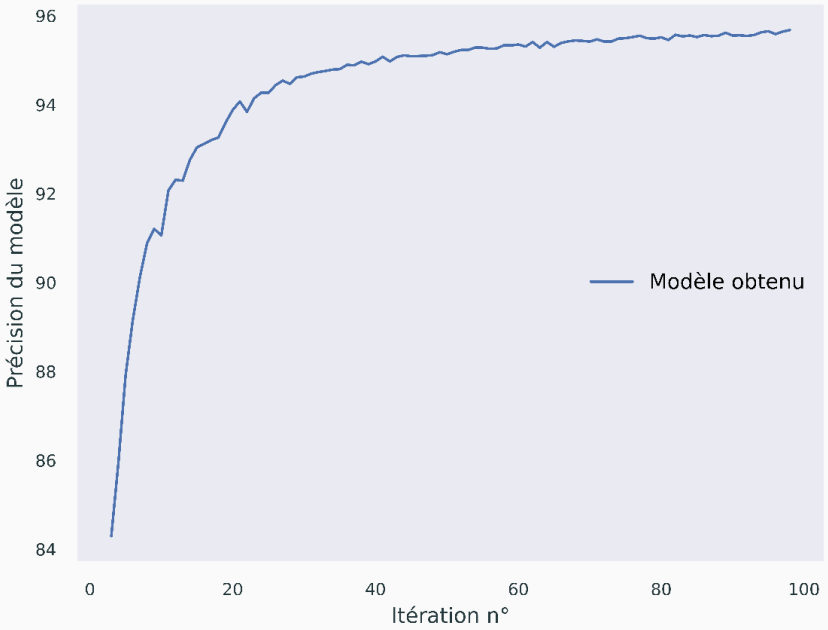
Graphique : Comparaison des performances de différentes structures de réseau

Choix fonction coût, fonction d'activation

	Fonctions d'activation			
	Sigmoid $= \frac{1}{1+e^x}$	TanH. $= \frac{e^x - e^{-x}}{e^x + e^{-x}}$	ReLU $= max(0, x)$	Identité x
Distance euclidienne $= \|\|y - \hat{y}\|\|_2^2$	0.9431	0.9243	0.9395	0.0267
Entropie croisée $= -\sum_i \hat{y}_i * ln(y_i)$ $-\sum_i (1 - \hat{y}_i) * ln(1 - y_i)$	0.9502	0.9183	0.0076	0.0104

Tableau : Évaluation après 20 itérations pour différentes fonctions d'activation et coût

Performances finales



Graphique : Résultat final (95.7%)

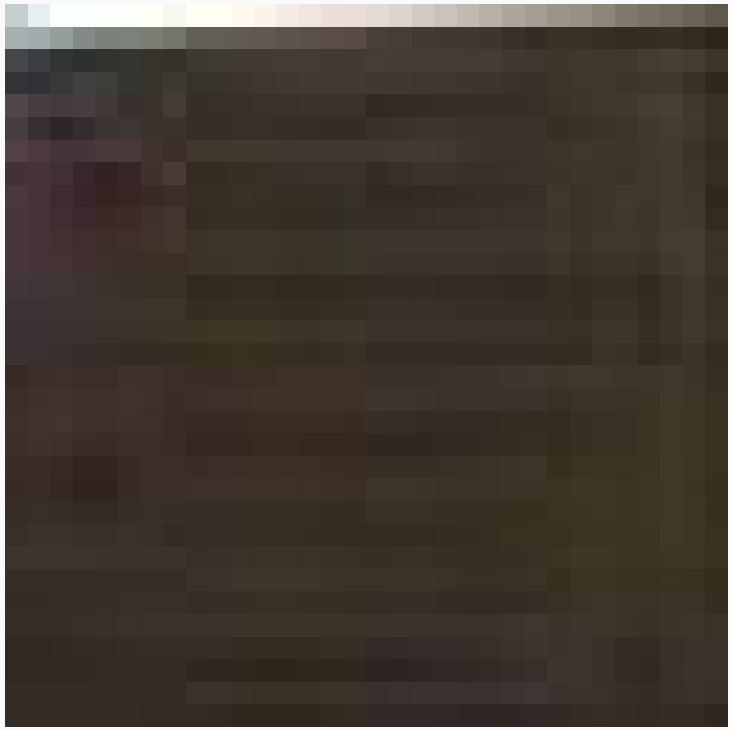


Image : Exemple de panneau mal identifié

Application à la reconnaissance de panneaux (1)

Image : Image par un jour nuageux

Image : Image de nuit

Application à la reconnaissance de panneaux (2)

Image : Image prise en voiture

Image : Panneau extrait, correctement identifié comme "Sens interdit".

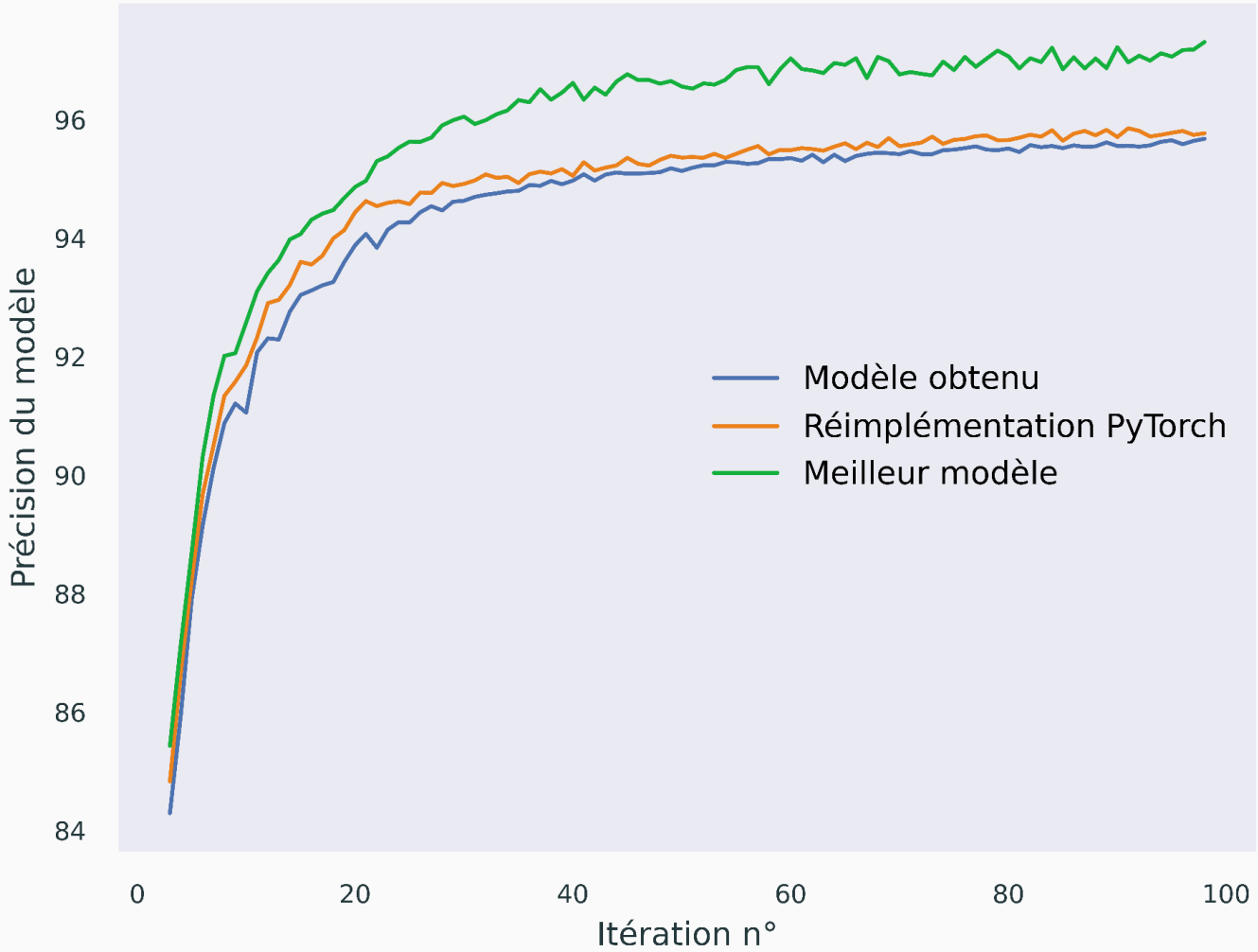
.20 images par seconde

.981/985 identifications

.Fonctionne sur la nuit

Conclusion et mise en perspective (6/6)

Comparaison avec PyTorch



Conclusion

Possibilités:

.Expérimenter avec d'autres couches

.Construire un fondement théorique solide

.Appliquer à d'autres problèmes

Documents annexes.

Humains et reconnaissance de formes (supprimé)

Source : Emmanuel Gill

Source : La Voix du Nord

Choix pas d'entraînement

	Pas (Fonctions d'activation)			
	1	0.01	0.001	0.0001
Réseau 27 Conv + FF	0.8994	0.9223	0.9395	0.8813
Réseau 22 Conv + FF	0.9075	0.9111	0.9350	0.8913
Réseau 08 FF seulement	0.8451	0.8436	0.8412	0.8385

Tableau : Évaluation après 100 itérations pour différents pas et réseaux (Sigmoid, ReLU, distance Euclidienne)

Neurone humain

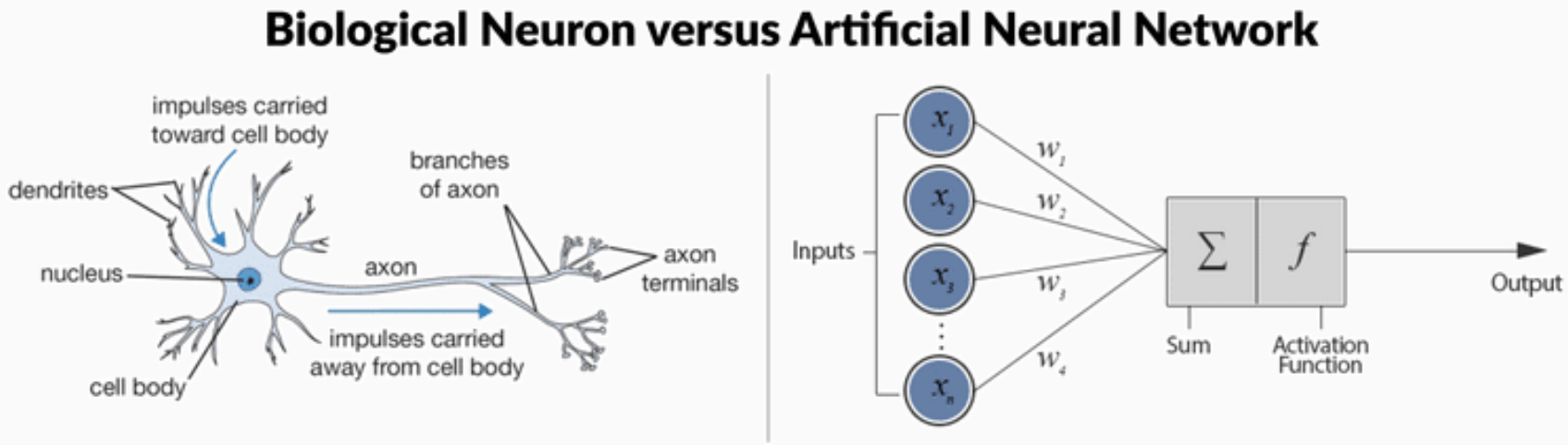


Figure 13: Source: https://datascientest.com/

Algorithme descente du gradient

Algorithm 1 Descente du gradient

Entrée: $C(x)$, *eta*, x_0, N

Initialiser $k \leftarrow 0$

tant que $k < N$ **faire**

Calculer le gradient $\nabla f(x_k)$

Mettre à jour le point : $x_{k+1} \leftarrow x_k - \eta \nabla f(x_k)$

$k \leftarrow k + 1$

fin tant que

retourne x_k

Preuve descente du gradient (cas fortement convexe)

$C : \mathbb{R}^n \longrightarrow \mathbb{R}^p$, fonction coût à minimiser.

Hypothèses:

.C est différentiable

.C est c-convexe ($\exists c > 0, \forall x, y, \forall t \in [0,1], C(tx + (1-t)y \leq tC(x) + (1-t)C(y) - \frac{ct(1-t)}{2}||x-y||^2$)

.∇C est k-lipschitzienne

.η "petit" et strictement positif (détail plus bas)

Preuve :

Notons y l'unique point réalisant le minimum de C (unique car convexe)

Comme C est différentiable, $\nabla C(y) = 0$

$\forall k \in \mathbb{N}, ||x_{k+1} - y||_2^2 = ||x_k - \eta \nabla C(x_k) - y||_2^2$

$= ||x_k - y||_2^2 - 2\eta\langle \nabla C(x_k) - \nabla C(y), x_k - y\rangle + \eta^2 ||\nabla C(x_k) - \nabla C(y)||_2^2$

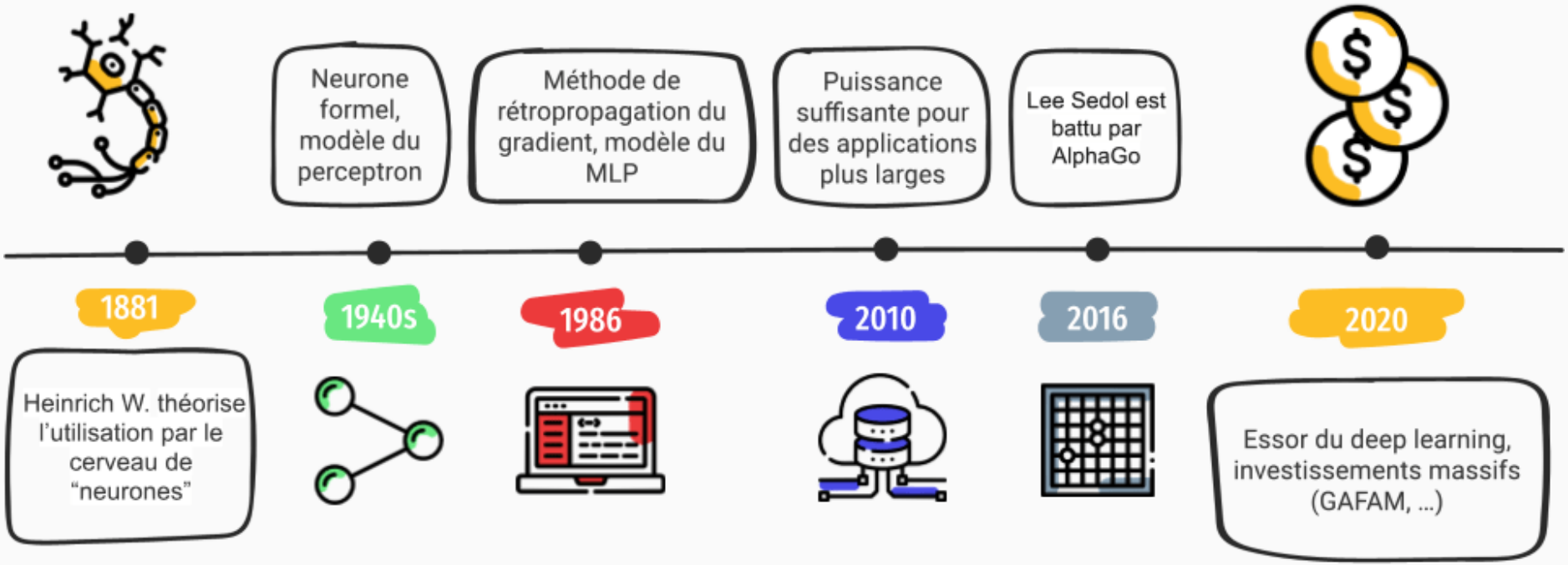
Or, $2||\nabla C(x_k) - \nabla C(y)||_2^2 \leq C^2 ||x_k - y||_2^2$ et

$\langle \nabla C(x_k) - \nabla C(y), x_k - y\rangle \geq c||x_k - y||_2^2$

donc $||x_{k+1} - y||_2 \leq (1 - 2c\eta + \eta^2 C^2)^{\frac{1}{2}} ||x_k - y||_2$

donc pour $0 < \eta < \frac{2c}{C}, 0 < 1 - 2c\eta + \eta^2 C^2 < 1$, on a bien $x_k \underset{k \to +\infty}{\to} y$!

Un peu d'histoire



References

[1] Álvaro Arcos-García, Juan A. Álvarez-García, and Luis M. Soria-Morillo. "Evaluation of deep neural networks for traffic sign detection systems". In: *Neurocomputing* 316 (2018), pp. 332–344. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2018.08.009. URL: https://www.sciencedirect.com/science/article/pii/S092523121830924X.

[2] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.1980 (), pp. 193–202.

[3] Jiuxiang Gu et al. "Recent Advances in Convolutional Neural Networks". In: (2015). arXiv:1512.07108.

Bibliographie ii

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.1989 (), pp. 359–366.

[5] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.1998 (), pp. 2278–2324. DOI: 10.1109/5.726791.

[6] Yann LeCun et al. "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems 2* (1989).

[7] David E. Rumelhart and James L. McClelland. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations* (1986).

[8] Citlalli Gómez Serna and Yassine Ruichek. "Classification of Traffic Signs: The European Dataset". In: *IEEE Access* (2018).

Bibliographie iii

[9] F. Shao et al. “Real-Time Traffic Sign Detection and Recognition Method Based on Simplified Gabor Wavelets and CNNs”. In: *Sensors* 18.10 (2018), p. 3192. DOI: 10.3390/s18103192. URL: https://doi.org/10.3390/s18103192.

[10] Valentyn N. Sichkar and Sergey A. Kolyubin. “Effect of various dimension convolutional layer filters on traffic sign classification accuracy”. In: *Scientific and Technical Journal of Information Technologies, Mechanics and Optics* (2019).

[11] J. Stallkamp et al. “Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition”. In: *Neural Networks* (2012). DOI: 10.1016/j.neunet.2012.02.016. URL: http://www.sciencedirect.com/science/article/pii/S0893608012000457.

Liste des codes sources

Listing 1: network.py

```
import numpy as np
from layers import *
from utils.classes import *
from utils.activation_functions import *
from utils.cost_functions import *
from loaders.pano_loader import *

class Network(object):

    def __init__(self, loader, layers: list, nb_classes=43,
    ↪  cost_function=DistanceEuclidienne2,
    ↪  classes=classes_fr()):
        """
        :param layers: [Layer1, Layer2, ...]
        """
        self.classes = classes
        self.layers = layers
        self.loader = loader
        self.nb_classes = nb_classes
        self.num_layers = len(layers)
        self.cost = cost_function
        self.monitor = ([],[])

    def feedforward(self, entree):
        """
        Renvoie le vecteur sortie pour un vecteur d'entree entree
        :return: np.array a 1 dimension
        """
        sortie = entree
        for layer in self.layers:
            sortie = layer.feedforward(sortie)
        return sortie
```

```
    def feedforwardClasse(self, entree):
        """
        Renvoie la classe pour un vecteur d'entree a
        :return: string
        """
        sortie = self.feedforward(entree)
        return self.classes[np.argmax(sortie)]

    def train(self, epochs=1, eta=2,
              test_accuracy=False, train_accuracy=False,
              mini_batch_size=20):
        """
        Entraine et retourne les evaluations si demandées
        :return: (train_accuracy, test_accuracy) (listes
↪   d'entiers a une dimension)
        """

        train, test = self.loader()
        start = time.time()

        if train_accuracy:
        ↪  self.monitor[0].append(self.accuracy(train,
        ↪  '(train)')[0])
        if test_accuracy:
        ↪  self.monitor[1].append(self.accuracy(test,
        ↪  '(test)')[0])

        for epoch in range(epochs):

            n_train = len(train)
            print(f'Epoch {epoch + 1} started...')

            np.random.shuffle(train)

            batches = [
                train[k:k + mini_batch_size]
                for k in range(0, n_train, mini_batch_size)]

            for batch in batches:
                self.train_batch(batch, eta)

            print(f"\nEntraînement: itération n°{epoch + 1}
            ↪  finie.")
            print(f"Temps: {time.time() - start:.2f}s")
```

```
            if train_accuracy:
                self.monitor[0].append(self.accuracy(train,
                '(train)')[0])
            if test_accuracy:
                self.monitor[1].append(self.accuracy(test,
                '(test)')[0])

        return train_accuracy, test_accuracy

    def train_batch(self, batch, eta):
        batch_size = len(batch[0])
        for entree in batch:
            image, label = entree[:, :, :, 0],
            delta_vecteur(int(entree[0, 0, 0, 0]),
            self.nb_classes)
            output = self.feedforward(image)
            nabla = self.cost.prime(output, label)
            for layer in reversed(self.layers):
                nabla = layer.backpropagation(eta, nabla)

        for layer in reversed(self.layers):
            layer.backpropagation_update(eta, batch_size)
            layer.reset_backpropagation()

    def accuracy(self, data, add_str=''):
        """
        :param data: np.array de taille (nb_images, image,
etiquette)
        :return: (nb_juste:int, nb_total:int)
        """
        print('Evaluation précision...')
        total = len(data)
        juste = 0
        for couple in data:
            juste += int(np.argmax(self.feedforward(couple[:, :,
            :, 0])) == int(couple[0, 0, 0, 0]))
        print(f"Précision: {juste}/{total} ou {juste / total:.4%}
        {add_str}")
        return juste, total

    def save(self, nom):
        """"
        Sauvegarde le reseaux sous ../network/{nom}
        """
        with open('./networks/' + nom, "wb") as f:
            pickle.dump(self, f)
```

Listing 2: layers.py

```
            f.close()
        print(f'Network {nom} saved')


def load(nom):
    """
    Charge le reseau {nom}
    """
    with open("./networks/" + nom, "rb") as f:
        net = pickle.load(f)
        f.close()
        print(f'Network {nom} loaded!')
        return net


#Fonction Auxiliaire
def delta_vecteur(j, size):
    """
    :return: np.array de taille (size, 1) avec un 1 a la position
↪   j, 0 partout ailleurs
    """
    e = np.zeros((size, 1))
    e[j] = 1.0
    return e

if __name__ == '__main__':
    print('Ceci est un module, pas un script.')


import numpy as np
from scipy import signal
from utils.activation_functions import *
import time

# @ = np.dot, * = np.multiply (hadamart)

class Layer():
    def backpropagation_update(self, eta, batch_size):
        pass

    def reset_backpropagation(self):
        pass
```

```python
class Dense(Layer):
    def __init__(self, entree_dim, sortie_dim, func):
        self.entree_dim, self.dim_sortie = entree_dim, sortie_dim
        self.func = func
        self.poids = np.random.randn(sortie_dim, entree_dim)
        self.biais = np.random.randn(sortie_dim, 1)
        self.poids_batch = np.zeros(self.poids.shape)
        self.biais_batch = np.zeros(self.biais.shape)

    def feedforward(self, entree):
        self.entree = entree
        self.sortie = self.func.fn(self.poids @ self.entree +
        ↪  self.biais)
        return self.sortie

    def backpropagation(self, eta, nabla_sortie):  # Une image,
    ↪  reset met a jour les poids.
        derivee_activation = self.func.prime(self.sortie)
        nabla_sortie = nabla_sortie * derivee_activation  #
        ↪  Derive la fonction d'activation
        nabla_poids = (nabla_sortie @ self.entree.T)
        nabla_entree = (self.poids.T @ nabla_sortie)  #
        ↪  Jacobienne
        self.poids_batch += eta * nabla_poids
        self.biais_batch += eta * nabla_sortie
        return nabla_entree

    def backpropagation_update(self, eta, batch_size):
        self.poids -= eta * self.poids_batch / batch_size
        self.biais -= eta * self.biais_batch / batch_size

    def reset_backpropagation(self):
        self.poids_batch = np.zeros(self.poids.shape)
        self.biais_batch = np.zeros(self.biais.shape)

    def name(self):
        return f"Dense({self.entree_dim}, {self.dim_sortie},
        ↪  {self.func.__name__})"


class Convolution(Layer):
    """stride (déplacement) = 1, padding (ajout de zeros) = 0"""
    def __init__(self, img_dim, kernel_cote, sortie_prof, func):
        self.func = func
        self.entree_dims = img_dim
```

```
        self.entree_haut, self.entree_larg, self.entree_prof =
        ↪  self.entree_dims
        self.sortie_dims = (self.entree_haut - kernel_cote + 1,
        ↪  self.entree_larg - kernel_cote + 1, sortie_prof)
        self.sortie_prof = sortie_prof
        self.filtre_dim = (kernel_cote, kernel_cote,
        ↪  self.entree_prof, sortie_prof)
        self.filtre = np.random.rand(*self.filtre_dim)
        self.biais = np.random.rand(*self.sortie_dims)
        self.filtre_batch = np.zeros(self.filtre.shape)
        self.biais_batch = np.zeros(self.biais.shape)

    def feedforward(self, entree):
        self.entree = entree
        self.sortie = np.copy(self.biais)
        for i in range(self.sortie_prof):
            for j in range(self.entree_prof):
                #On utilise scipy car c'est plus rapide, mais on
                ↪  pourrait l'implementer facilement avec des
                ↪  boucles
                #La fonction de scipy est plus rapide car elle
                ↪  est ecrite en C, et utilise la FFT
                self.sortie[:, :, i] +=
                ↪  signal.correlate2d(self.entree[:, :, j],
                ↪  self.filtre[:, :, j, i], "valid")
        self.sortie = self.func.fn(self.sortie)
        return self.sortie

    def backpropagation(self, eta, nabla_sortie):
        derivee_activation = self.func.prime(self.sortie)
        nabla_sortie = nabla_sortie * derivee_activation
        nabla_filtre = np.zeros(self.filtre_dim)
        nabla_entree = np.zeros(self.entree_dims)

        for i in range(self.sortie_prof):
            for j in range(self.entree_prof):
                nabla_filtre[:, :, j, i] =
                ↪  signal.correlate2d(self.entree[:, :, j],
                ↪  nabla_sortie[:, :, i], "valid")
                nabla_entree[:, :, j] +=
                ↪  signal.convolve2d(nabla_sortie[:, :, i],
                ↪  self.filtre[:, :, j, i], "full")
        self.filtre_batch += nabla_filtre
        self.biais_batch += nabla_sortie
        #print(time.time() - st)
        #print('conv\n')
```

```
        return nabla_entree

    def name(self):
        return f"Convolution({self.entree_dims},
        ↪  {self.filtre_dim[0]}, {self.sortie_prof},
        ↪  {self.func.__name__})"

    def backpropagation_update(self, eta, batch_size):
        self.filtre -= eta * self.filtre_batch / batch_size
        self.biais -= eta * self.biais_batch / batch_size

    def reset_backpropagation(self):
        self.filtre_batch = np.zeros(self.filtre.shape)
        self.biais_batch = np.zeros(self.biais.shape)


class Reshape(Layer):
    def __init__(self, dim_entree, dim_sortie):
        self.dim_entree = dim_entree
        self.dim_sortie = dim_sortie

    def feedforward(self, entree):
        self.entree = entree
        temp = np.reshape(entree, self.dim_sortie)
        return temp

    def backpropagation(self, eta, nabla_sortie):
        return np.reshape(nabla_sortie, self.dim_entree)

    def name(self):
        return f"Reshape({self.dim_entree}, {self.dim_sortie})"


class Softmax(Layer):
    """Normalise pour avoir une distribution de probabilite"""
    def __init__(self):
        pass

    def feedforward(self, entree):
        expo = np.exp(entree - np.max(entree))
        self.sortie = expo / np.sum(expo)
        return self.sortie

    def backpropagation(self, eta, nabla_sortie):
        jacobian = np.diag(self.sortie.flatten()) -
        ↪  np.outer(self.sortie, self.sortie)
```

```
        nabla_entree = jacobian @ nabla_sortie
        return nabla_entree

    def name(self):
        return "Softmax"



class Maxpooling(Layer):
    """
    filtre carre, cote*cote, aucun parametre (donc rien a update
↪   en backprop), filtre_cote divise la hauteur ET la largeur
    """

    def __init__(self, entree_dim, filtre_cote):
        self.entree_haut, self.entree_larg, self.entree_prof =
        ↪  entree_dim
        self.entree_dim = entree_dim
        self.filtre_cote = filtre_cote
        self.sortie_haut = self.entree_haut // self.filtre_cote
        self.sortie_larg = self.entree_larg // self.filtre_cote
        self.sortie_prof = self.entree_prof

    def feedforward(self, entree):
        self.entree = entree
        cote = self.filtre_cote
        sortie = np.zeros((self.sortie_haut, self.sortie_larg,
        ↪  self.sortie_prof))
        for couche in range(self.sortie_prof):
            for y in range(self.sortie_haut):
                for x in range(self.sortie_larg):
                    sortie[x, y, couche] = np.max(entree[y *
                    ↪  cote: (y + 1) * cote, x * cote: (x + 1) *
                    ↪  cote, couche])
        return sortie

    def backpropagation(self, grad_sortie, eta):
        grad_entree = np.zeros((self.entree_haut,
        ↪  self.entree_larg, self.entree_prof))
        entree = self.entree
        cote = self.filtre_cote
        for couche in range(self.sortie_prof):
            for x in range(self.sortie_larg):  # j
                for y in range(self.sortie_haut):
                    origine = entree[x * cote: (x + 1) * cote, y
                    ↪  * cote: (y + 1) * cote, couche]
```

```
                    x_max, y_max = np.where(np.max(origine) ==
                    ↪  origine)
                    x_max, y_max = x_max[0], y_max[0]
                    grad_entree[x * cote: (x + 1) * cote, y *
                    ↪  cote: (y + 1) * cote, couche][x_max,
                    ↪  y_max] = grad_sortie[
                        x, y, couche]
        return grad_entree

    def name(self):
        return f"Maxpooling({self.entree_dim},
        ↪  {self.filtre_cote})"


#Jamais servie
class Greyed(Layer):
    """Convertit l'image en niveaux de gris"""
    def __init__(self, first = True):
        self.rgb_weights = np.array([0.2989, 0.5870, 0.1140])
        self.first = first

    def feedforward(self, entree):
        self.entree = entree
        return (entree[..., :3] @ self.rgb_weights)[..., None]  #
        ↪  [..., None] pour ajouter la dimension manquante

    def backpropagation(self, nabla_sortie, eta):
        if self.first:
            return nabla_sortie
        nabla_entree = np.zeros_like(self.entree)
        nabla_entree[..., :3] = (nabla_sortie[..., 0, None] @
        ↪  self.rgb_weights[None, :])
        return nabla_entree

    def name(self):
        return "Greyed"
```

Listing 3: cost_functions.py

```
import numpy as np


class Cost():
```

```
    pass


class DistanceEuclidienne2(Cost):
    """
    Renvoie le carré de la distance euclidienne entre a et y
    """

    @staticmethod
    def fn(a, y):
        return np.mean(np.power(y - a, 2))

    @staticmethod
    def prime(a, y):
        return 2 * (a - y) / np.size(y)


class DistanceEuclidienne(Cost):
    """
    Renvoie la distance euclidienne entre a et y
    """

    @staticmethod
    def fn(a, y):
        return np.linalg.norm(a - y)

    @staticmethod
    def prime(a, y):
        return (a - y) / DistanceEuclidienne2.fn(a, y) *
        ↪  np.size(y)


class EntropieCroisee(Cost):
    """
    Mettre softmax en dernier (Prend une distribution de propa en
↪   entrée))
    """

    @staticmethod
    def fn(a, y):
        return np.sum(np.nan_to_num(-y * np.log(a) - (1 - y) *
        ↪  np.log(1 - a)))

    @staticmethod
    def prime(a, y):
        return (a - y) / (a * (1 - a))
```

Listing 4: activation_functions.py

```
import numpy as np
class activations:
    """Fonction d'activation, (fn, prime) possible"""
    pass

class sigmoid(activations):
    @staticmethod
    def fn(z):
        return 1.0/(1.0+np.exp(-z))
    @staticmethod
    def prime(z):
        return sigmoid.fn(z)*(1-sigmoid.fn(z))

class tanh(activations):
    @staticmethod
    def fn(z):
        return (np.tanh(z) + 1)/2
    @staticmethod
    def prime(z):
        return (1-np.tanh(z)**2)/2

class ReLU(activations):
    @staticmethod
    def fn(z):
        return np.maximum(0, z)
    @staticmethod
    def prime(z):
        return np.where(z > 0, 1, 0)

class identity(activations):
    @staticmethod
    def fn(z):
        return z
    @staticmethod
    def prime(z):
        return 1
```

Listing 5: pytorch.py

```
import torch
import torch.nn as nn
```

```
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import torch.nn.functional as F
import numpy as np
from loaders.pano_loader import *
import pickle

def delta_vecteur(i, n):
    vecteur = np.zeros(n)
    vecteur[i] = 1
    return vecteur

train_data, test_data = EUD_loader_grey()

# Convertit les numpy arrays en torch tensors
train_data = torch.tensor(train_data, dtype=torch.float32)
test_data = torch.tensor(test_data, dtype=torch.float32)

# Créer un jeu de données custom
class CustomDataset(Dataset):
    def __init__(self, data):
        self.labels = data[:, 0, 0, 0, 0].clone().detach().long()
        self.images = data[:, :, :, 0,
        ↪  0].clone().detach().float()

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, index):
        return self.images[index], self.labels[index]


train_dataset = CustomDataset(train_data)
test_dataset = CustomDataset(test_data)

train_loader = DataLoader(train_dataset, batch_size=15,
↪  shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=15,
↪  shuffle=False)

# Examples de CNNs pour le rendu final
class CNN1(nn.Module):
    """
    (32*32) C-> (16, 28*28) MP-> (16, 14*14) -> (512) -> (65)
    """
    def __init__(self):
```

```
        super(CNN1, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 5, padding=0, stride=1)
        self.MP1 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(14 * 14 * 16, 512)
        self.fc2 = nn.Linear(512, 65)

    def forward(self, x):
        x = F.sigmoid(self.conv1(x))
        x = self.MP1(x)
        x = x.view(-1, 14*14*16)
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        return x

class CNN2(nn.Module):
    """
    (32*32) -> (256) -> (256) -> (65)
    """
    def __init__(self):
        super(CNN2, self).__init__()
        self.fc1 = nn.Linear(32*32, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 65)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = x.view(-1, 32*32)
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = F.sigmoid(self.fc3(x))
        x = self.softmax(x)
        return x

class CNN3(nn.Module):
    """
    (32*32) -> (128) -> (128) -> (65)
    """
    def __init__(self):
        super(CNN3, self).__init__()
        self.fc1 = nn.Linear(32*32, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 65)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = x.view(-1, 32*32)
```

```
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = F.sigmoid(self.fc3(x))
        x = self.softmax(x)
        return x

class CNN4(nn.Module):
    """
    (32*32) -> (256) -> (65)
    """
    def __init__(self):
        super(CNN4, self).__init__()
        self.fc1 = nn.Linear(32*32, 256)
        self.fc2 = nn.Linear(256, 65)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = x.view(-1, 32*32)
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = self.softmax(x)
        return x

class CNN5(nn.Module):
    """
    (32*32) MP-> (16*16) C-> (16, 12*12) -> (128) -> (128) ->
↪   (65)
    """
    def __init__(self):
        super(CNN5, self).__init__()
        self.MP1 = nn.MaxPool2d(2, 2)
        self.conv1 = nn.Conv2d(1, 16, 5, padding=0, stride=1)
        self.fc1 = nn.Linear(16 * 12 * 12, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 65)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.MP1(x)
        x = F.sigmoid(self.conv1(x))
        x = x.view(-1, 16*12*12)
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = F.sigmoid(self.fc3(x))
        x = self.softmax(x)
        return x
```

```
# Défini la fonction coût
criterion = nn.CrossEntropyLoss()
resultats = np.empty((5, 20)) # 5 modèles, 20 epoches, enregistre
↪  les résultats

def eval(i, t, loader, model, numeroReseau):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for images, labels in loader:
            images = images.unsqueeze(1)  # Add a channel
            ↪  dimension
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    resultats[numeroReseau][i] = accuracy
    print(f"Epoch {i}/{t}, Accuracy: {accuracy:.3f}%
    ↪  ({correct}/{total})")

def train(model, numeroReseau, optimizer):
    # Create the training loop
    num_epochs = 20
    eval(0, num_epochs, test_loader, model, numeroReseau)
    for epoch in range(num_epochs):
        model.train()
        i = 0
        for images, labels in train_loader:
            i += 1
            if i % 3000 == 0:
                print(i)
            images = images.unsqueeze(1)  # Add a channel
            ↪  dimension
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
        eval(epoch, num_epochs, test_loader, model, numeroReseau)

if __name__ == "__main__":
    for i in range(1,6):
        print(f"Début évaluation modèle {i}:")
```

```
        exec(f'model = CNN{i}()')
        optimizer = optim.SGD(model.parameters(), lr=0.5)
        train(model, i, optimizer)
    print("Fin évaluation modèles.")
    print(resultats)
    with
    ↪  open('C:\\Users\\benyo\\PycharmProjects\\PanneauxCNN\\resultats.pickle',
    ↪  'wb') as f:
        pickle.dump(resultats, f)
    print('Done !')
```

Listing 6: panneaux_loaders.py

```
import pickle
import os

data_dir =
↪  os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪  '..', 'data')

def pano_loader_hidden(x):
    print('loading data')
    with open(os.path.join(data_dir, str(x)), 'rb') as f:
        data = pickle.load(f)
    print('data loaded')
    return data['train'], data['test']

def pano_loader_grey():
    """
    :return: (train: [num, haut, larg, prof, y], test: [num,
↪  haut, larg, prof, y])
    train et test sont des numpy array de shape (nb_images, 32,
↪  32, 1, 1)
    """
    return pano_loader_hidden('grey_255_LHE.pickle')

def pano_loader_RGB():
    """
    :return: (train: [num, haut, larg, prof, y], test: [num,
↪  haut, larg, prof, y])
    train et test sont des numpy array de shape (nb_images, 32,
↪  32, 3, 1)
    """
    return pano_loader_hidden('255_mean.pickle')
```

```
def EUD_loader_grey():
    """
    :return: (train: [num, haut, larg, prof, y], test: [num,
↪   haut, larg, prof, y])
    test et train sont des numpy array de shape (nb_images, 32,
↪   32, 1, 1)
    """
    return pano_loader_hidden('EUD_GREY_255_LHE.pickle')

def EUD_loader_RGB():
    """
    :return: (train: [num, haut, larg, prof, y], test: [num,
↪   haut, larg, prof, y])
    test et train sont des numpy array de shape (nb_images, 32,
↪   32, 3, 1)
    """
    return pano_loader_hidden('EUD_RGB_AUGMENTE.pickle')

def EUD_loader_ORIGINAL_REDUIT():
    """
    :return: (train: [num, haut, larg, prof, y], test: [num,
↪   haut, larg, prof, y])
    test et train sont des numpy array de shape (nb_images, 32,
↪   32, 3, 1)
    """
    return pano_loader_hidden('EUD_ORIGINAL_REDUIT.pickle')

if __name__ == '__main__':
    with open('../data/255_mean.pickle', 'rb') as f:
        data = pickle.load(f)
    print(type(data))
    print(data.keys())
```

Listing 7: data_preprocessing.py

```
import os

import numpy as np
import pickle
from PIL import Image, ImageOps, ImageEnhance
import utils.classes as cl
import random
from loaders.pano_loader import *
```

```
data_dir =
↪  os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪  '..', 'data')
base_dir = os.path.join(data_dir, 'EuDataset')
train_dir = os.path.join(base_dir, 'Training')
test_dir = os.path.join(base_dir, 'Testing')


def create_data(dir):
    """Transforme le dossier original EuDataset
    :return: numpy array de dimensions (nb_images, 32, 32, 3, 1)
    """
    sub_dirs = os.listdir(dir)
    data = []
    for sub_dir in sub_dirs:
        print(int(sub_dir))
        sub_dir_path = os.path.join(dir, sub_dir)

        # For a file to be an image, we have to make sure it ends
        ↪  with .ppm
        images = [os.path.join(sub_dir_path, image) for image in
        ↪  os.listdir(sub_dir_path) if image.endswith('.ppm')]
        for image in images:
            img = Image.open(image)
            img = img.resize((32, 32))
            img = np.expand_dims(np.array(img), axis=-1)
            img[0, 0, 0, 0] = int(sub_dir)
            data.append(img)
    return np.array(data)


def update_noir_et_blanc(data: np.array):
    """Transforme un jeu de donnee
    Remarque: Transforme un jeu de donnee de la forme (nb_images,
↪   32, 32, 3, 1)
    :return: np.array transformee (noir et blanc, LHE)
    """

    # Creer un nouveau tableau de la forme adaptée
    new_data = np.empty((len(data), 32, 32, 1, 1))

    for i in range(len(data)):
        if i % 1000 == 0: print(i)
        img, label = data[i, :, :, :, 0] * 255, data[i, 0, 0, 0,
        ↪  0]
```

```
            img = Image.fromarray(img)

            # Noir et blanc
            img = img.convert('L')

            # LHE
            img = ImageOps.equalize(img)

            # Normalisation (Calculs plus rapide)
            img = np.asarray(img)
            img = img / 255
            img = np.expand_dims(img, axis=-1)
            img = np.expand_dims(img, axis=-1)

            img[0, 0, 0, 0] = label

            new_data[i] = img

    return new_data

def rdm_rotation(image: Image.Image):
    """
    :param image: PIL.Image
    :return: PIL.Image avec rotation aléatoire dans [-10, 10]
    """
    valeurs = [i for i in range(-4, 4)] # En degrés
    return image.rotate(np.random.choice(valeurs))

def rdm_brightness(image: Image.Image):
    """
    :param image: PIL.Image
    :return: PIL.Image avec luminosite augmentee (entre *0.5 et
↪   *1.5)
    """
    amount = np.random.uniform(0.8, 1.2)
    return ImageEnhance.Brightness(image).enhance(amount)

def rdm_flip(image: Image.Image):
    """
    :param image: PIL.Image
    :return: PIL.Image, symetrie verticale de l'entree
    """
    if random.random() < 0.5:  # Plus rapide que
    ↪   bool(random.getrandbits(1)) et que random.choice([True,
    ↪   False])
        return image
```

```
    return ImageOps.mirror(image)

def rdm_color(image: Image.Image):
    """
    :param image: PIL.Image
    :return: PIL.Image avec couleur augmentee (entre *0.8 et
↪   *1.2)
    """
    amount = np.random.uniform(0.8, 1.)
    return ImageEnhance.Color(image).enhance(amount)

def rdm_contrast(image: Image.Image):
    """
    :param image: PIL.Image
    :return: PIL.Image avec contraste augmentee (entre *0.8 et
↪   *1.2)
    """
    amount = np.random.uniform(0.8, 1.2)
    return ImageEnhance.Contrast(image).enhance(amount)

def rdm_sharpness(image: Image.Image):
    """
    :param image: PIL.Image
    :return: PIL.Image avec nettete augmentee (entre *0.8 et
↪   *1.2)
    """
    amount = np.random.uniform(0.7, 1.3)
    return ImageEnhance.Sharpness(image).enhance(amount)

def rdm_zoom(image: Image.Image):
    """
    :param image: PIL.Image
    :return: PIL.Image avec zoom aleatoire (entre 0.8 et 1.2)
    """
    f = np.random.uniform(1., 1.15)
    # Calculate new dimensions
    width, height = image.size
    new_width = int(width * f)
    new_height = int(height * f)

    # Resize the image
    resized_image = image.resize((new_width, new_height))

    # Calculate crop box dimensions
    left = (resized_image.width - width) // 2
    top = (resized_image.height - height) // 2
```

```
    right = (resized_image.width + width) // 2
    bottom = (resized_image.height + height) // 2

    # Crop the image
    return resized_image.crop((left, top, right, bottom))


def salt(image: Image.Image, amount=0.05):
    """Salt and Pepper noise (Chaque pixel a une proba d'etre
    ↪  detruit et est remplacer par du blanc ou du noir (1/2))
    :param image: PIL.Image
    :return: PIL.Image avec bruit
    """
    image = np.asarray(image)
    modified = np.copy(image)
    shape = image.shape
    nb_pixel = np.prod(shape)
    for i in range(round(amount * nb_pixel)):
        x = np.random.randint(0, shape[0])
        y = np.random.randint(0, shape[1])
        modified[x, y] = np.random.choice([0, 255])

    return Image.fromarray(modified)

def update_classes(data: np.array, dico):
    """Met a jour les labels avec un dico = {ancien_num,
    ↪  nouveau_num}
    :return: Nouvelles donnees)
    """
    a_supprimer = 0
    for i in range(len(data)):
        label = int(data[i,0,0,0,0])
        if label not in dico.keys():
            a_supprimer += 1
    new_data = np.empty((len(data) - a_supprimer, 32, 32, 3, 1))
    decalage = 0

    for i in range(len(data)):
        label = int(data[i,0,0,0,0])
        if label in dico.keys():
            new_data[i - decalage] = data[i]
            new_data[i - decalage, 0, 0, 0, 0] = dico[label]
        else:
            decalage += 1
    print(f'{a_supprimer} images supprimées.')
    return new_data
```

```
def save(train, test, name):
    """Save as usual data
    Compile les donnees dans un dictionnaire et l'enregistre
    :param train: np.array
    :param test: np.array
    """
    dico = {}
    classes = cl.classes_EUD_fr()
    dico['labels'] = list(classes.values())
    dico['train'] = train
    dico['test'] = test
    with open(os.path.join(data_dir, name + '.pickle'), 'wb') as
    ↪  f:
        pickle.dump(dico, f)
    print('Data saved !')

def augmente_classe(data: np.array, label: int, nb: int):
    """Augmente le nombre d'images d'une classe, normalise a la
    ↪  fin
    :param data: np.array Donnees, format classique
    :param label: int
    :param nb: int Nombre d'images voulues
    :return: np.array Donnees augmentees (Cette classe est
↪   augmentee)
    """
    images = [i for i in range(len(data)) if data[i, 0, 0, 0]
    ↪  == label]

    if len(images) > nb:
        #Retourne exactement nb images
        return np.array([data[i] for i in
        ↪  np.random.choice(images, nb)])

    new_data = np.zeros((nb, 32, 32, 3, 1))

    for i in range(nb):
        x = np.random.choice(images)
        img = Image.fromarray(data[x, :, :, :,
        ↪  0].astype(np.uint8))

        #Brightness
        img = rdm_brightness(img)
        #Color
        img = rdm_color(img)
        #Zoom
```

```
            img = rdm_zoom(img)
            #Contrast
            img = rdm_contrast(img)
            #Sharpness
            img = rdm_sharpness(img)

            #Remet le label
            img = np.array(img) / 255.
            img = np.expand_dims(img, axis=-1)
            img[0, 0, 0, 0] = data[x, 0, 0, 0, 0]

            new_data[i] = img
        return new_data

    def save_useless_now():
        with open("C:\\Users\\benyo\\Desktop\\train.pickle", 'rb') as
        ↪  f:
            train = pickle.load(f)
        with open("C:\\Users\\benyo\\Desktop\\test.pickle", 'rb') as
        ↪  f:
            test = pickle.load(f)
        save(train, test, 'EUD_RGB_ORIGINAL')

        ancien_to_nouveau = {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 7: 5, 8:
        ↪  6, 11: 7, 12: 8, 13: 9, 16: 10, 17: 11, 18: 12, 20: 13,
                             21: 14, 22: 15, 24: 16, 32: 17, 33: 18,
                             ↪  35: 19, 36: 20, 37: 21, 41: 22, 42:
                             ↪  23, 50: 24, 51: 25,
                             54: 26, 55: 27, 57: 28, 59: 29, 60: 30,
                             ↪  61: 31, 62: 32, 63: 33, 64: 34, 66:
                             ↪  35, 68: 36, 70: 37,
                             71: 38, 72: 39, 73: 40, 74: 41, 77: 42,
                             ↪  80: 43, 83: 44, 84: 45, 85: 46, 86:
                             ↪  47, 87: 48, 88: 49,
                             90: 50, 91: 51, 93: 52, 100: 53, 105:
                             ↪  54, 106: 55, 109: 56, 110: 57, 112:
                             ↪  58, 119: 59, 139: 60,
                             140: 61, 148: 62, 149: 63, 159: 64, 160:
                             ↪  65}

        data = EUD_loader_RGB()
        train = data[0]
        test = data[1]
        test = update_classes(test, ancien_to_nouveau)
        train = update_classes(train, ancien_to_nouveau)
        save(train, test, 'EUD_ORIGINAL_REDUIT')
```

```
    train, test = EUD_loader_ORIGINAL_REDUIT()
    new_train = []
    for i in range(65):
        print(i)
        increased = augmente_classe(train, i, 2000)
        new_train.append(increased)
    new_train = np.concatenate(new_train)
    save(new_train, test, 'EUD_RGB_AUGMENTE')

    train, test = EUD_loader_RGB()
    train = update_noir_et_blanc(train.astype(np.uint8))
    test = update_noir_et_blanc(test.astype(np.uint8))
    save(train, test, 'EUD_GREY_255_LHE')

if __name__ == '__main__':
    train, test = EUD_loader_ORIGINAL_REDUIT()
    new_train = []
    for i in range(65):
        print(i)
        increased = augmente_classe(train, i, 2000)
        new_train.append(increased)
    new_train = np.concatenate(new_train)
    save(new_train, test, 'EUD_RGB_AUGMENTE')
```