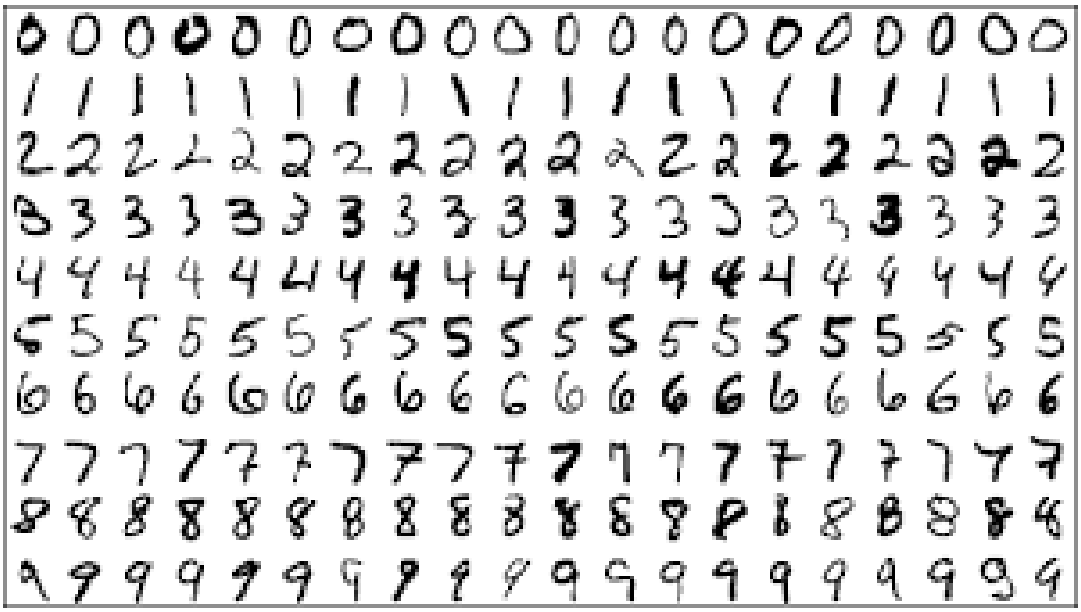
SIMPLON.CO

Projet N°13
Mars 2022



RAPPORT

Classification des chiffres manuscrits

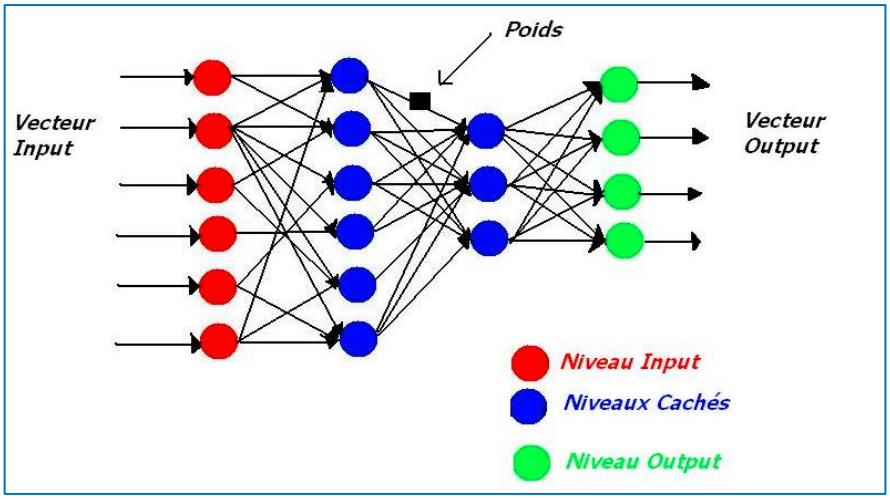
Classifier des chiffres manuscrits

Auteurs : **Maïna LE DEM**

Indice: A
Date d'émission: 28/03/2022

SOMMAIRE

1 : Réseau de neurone



Les réseaux de neurones, communément appelés des réseaux de neurones artificiels sont des **imitations simples des fonctions d'un neurone dans le cerveau humain** pour résoudre des problématiques d'apprentissage de la machine

Le neurone est une unité qui est exprimée généralement par une fonction sigmoïde.

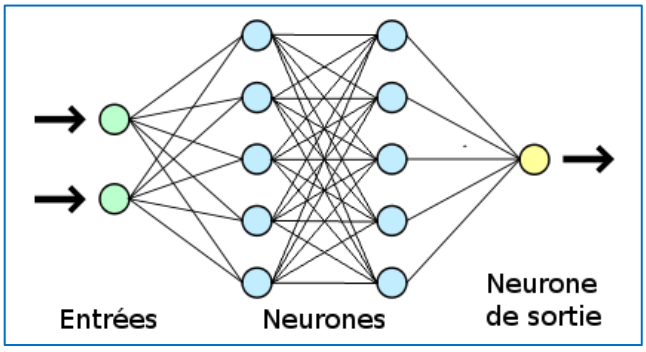
$$f(x) = \frac{1}{1 + e^{-x}}$$

Les architectures de réseaux neuronaux peuvent être divisées en 4 grandes familles :

- Réseaux de neurones Feed fowarded
- Réseaux de neurones récurrent (RNN)
- Réseaux de neurones à résonance
- Réseaux de neurones auto-organisés

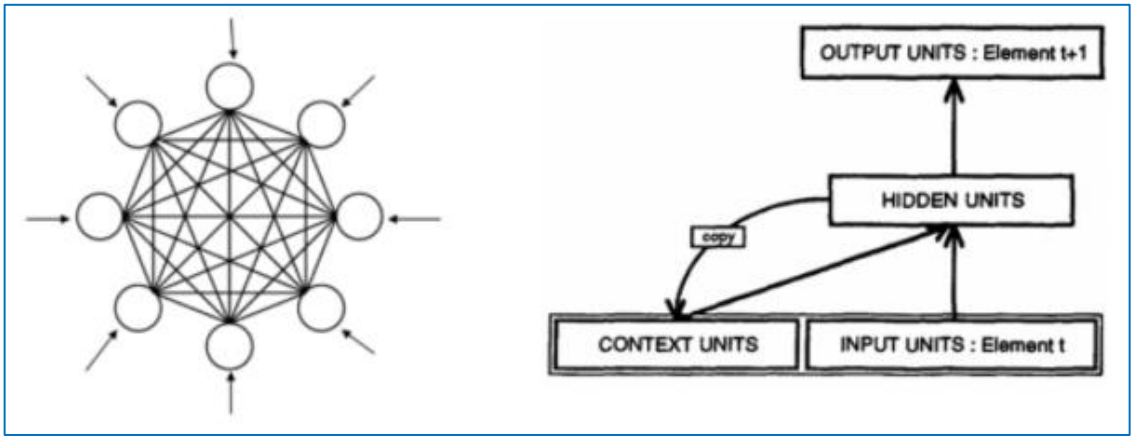
1.1 - *Réseau de neurones Feed Forward:*

- Feed-fowarded (propagation avant) signifie tout simplement que la donnée traverse le réseau d'entrée à la sortie sans retour en arrière de l'information.
- Typiquement, dans la famille des réseaux à propagation avant, on distingue les réseaux monocouches (perceptron simple) et les réseaux multicouches (perceptron multicouche)
- Le nombre de couches correspond aux nombres de matrices de poids dont disposent le réseau. Un perceptron multicouche est donc mieux adapté pour traiter les types de fonctions non-linéaires.
- Pour le traitement d'informations complexes et très variées, il est envisageable de créer plusieurs réseaux de neurones distincts dédiés à traiter chacun une partie de l'information. Ces réseaux de neurones sont appelés des réseaux neuronaux convolutifs (Convolutional Neural Networks).



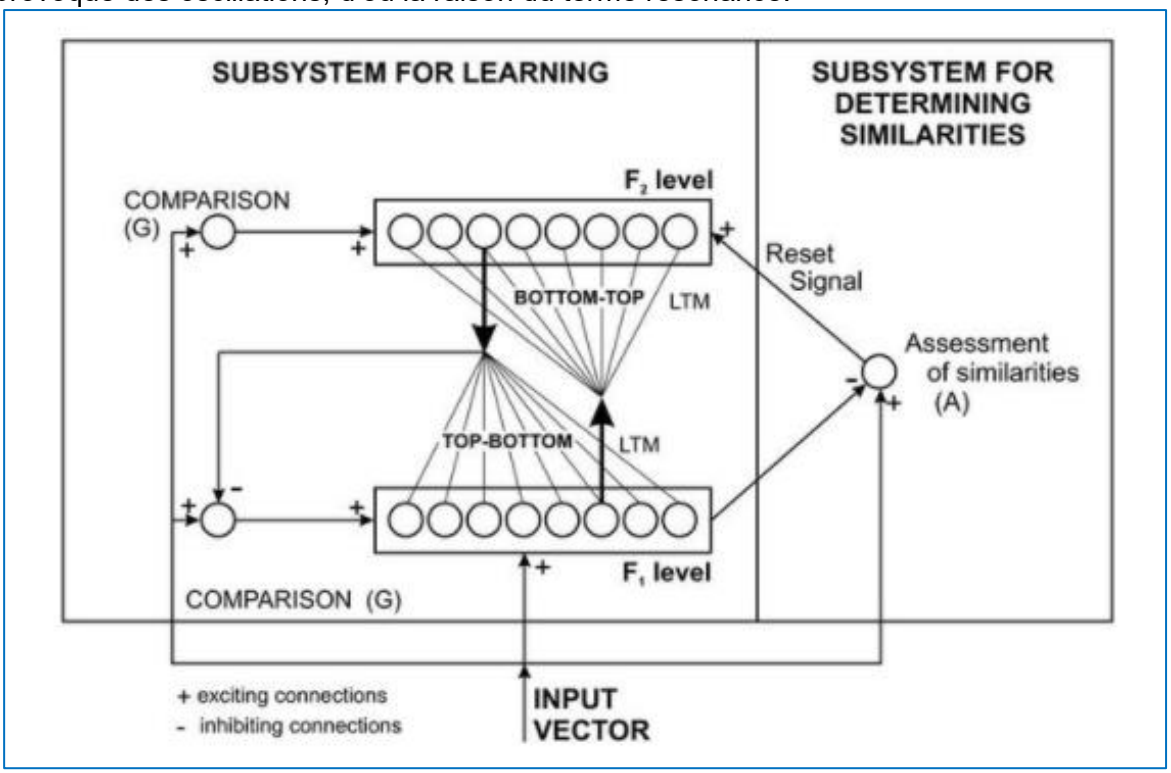
1.2 - *Réseau de neurones récurrents RNN:*

- Traitent de l'information en cycle: Ces cycles permettent au réseau de traiter l'information plusieurs fois en la renvoyant à chaque fois au sein du réseau.
- A la fois en propagation en avant et en rétro propagation.
- Capacité de prendre en compte des informations contextuelles suite à la récurrence du traitement de la même information. Cette dynamique auto-entretient le réseau.



1.3 - *Réseau de neurones à résonnance :*

L'activation de tous les neurones est renvoyée à tous les autres neurones au sein du système. Ce renvoi provoque des oscillations, d'où la raison du terme résonance.



1.4 - *Les Réseaux de neurones auto-organisés:*

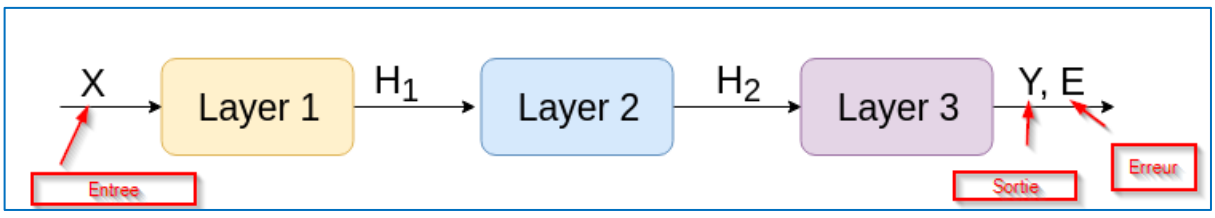
- méthodes d'apprentissage non-supervisée, les réseaux neuronaux auto-organisés sont capables d'étudier la répartition de données dans des grands espaces comme par exemple pour des problématiques de clustérisassions ou de classifications.

Démarche:

- Donner une entrée au modèle.
- Propager cette entrée à travers le réseau de neurones jusqu'à récupérer la sortie.
- Une fois la sortie obtenue, nous pouvons la comparer à la sortie voulue et donc calculer une erreur.
- **On ajuste les paramètres du modèle pour diminuer l'erreur précédemment** calculée. Pour cela on soustrait à chaque paramètre la dérivée de l'erreur par rapport à lui-même (gradient descendant).

```python
# Base class
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    # computes the output Y of a layer for a given input X
    def forward_propagation(self, input):
        raise NotImplementedError

    # computes dE/dX for a given dE/dY (and update parameters if any)
    def backward_propagation(self, output_error, learning_rate):
        raise NotImplementedError
```

```python
from layer import Layer

# inherit from base class Layer
class ActivationLayer(Layer):
    def __init__(self, activation, activation_prime):
        self.activation = activation
        self.activation_prime = activation_prime

    # returns the activated input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    # Returns input_error=dE/dX for a given output_error=dE/dY.
    # learning_rate is not used because there is no "learnable" parameters.
    def backward_propagation(self, output_error, learning_rate):
        return self.activation_prime(self.input) * output_error
```



$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$

Où **α** est un paramètre dans l'intervalle [0,1] que nous fixons et qui est appelé le **taux d'apprentissage.** Quoi qu'il en soit, l'important ici est **∂E/∂w** (la dérivée de E par rapport à **w**). Nous devons être en mesure de trouver la valeur de cette expression pour n'importe quel paramètre du réseau, **quelle que soit son architecture.**

```python
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)

    # set loss to use
    def use(self, loss, loss_prime):
        self.loss = loss
        self.loss_prime = loss_prime

    # predict output for given input
    def predict(self, input_data):
        # sample dimension first
        samples = len(input_data)
        result = []

        # run network over all samples
        for i in range(samples):
            # forward propagation
            output = input_data[i]
            for layer in self.layers:
                output = layer.forward_propagation(output)
            result.append(output)

        return result

```

```
32      # train the network
33      def fit(self, x_train, y_train, epochs, learning_rate):
34          # sample dimension first
35          samples = len(x_train)
36
37          # training loop
38          for i in range(epochs):
39              err = 0
40              for j in range(samples):
41                  # forward propagation
42                  output = x_train[j]
43                  for layer in self.layers:
44                      output = layer.forward_propagation(output)
45
46                  # compute loss (for display purpose only)
47                  err += self.loss(y_train[j], output)
48
49                  # backward propagation
50                  error = self.loss_prime(y_train[j], output)
51                  for layer in reversed(self.layers):
52                      error = layer.backward_propagation(error, learning_rate)
53
54              # calculate average error on all samples
55              err /= samples
56              print('epoch %d/%d   error=%f' % (i+1, epochs, err))
```

2 : Classification des images par CNN

CNN (ou ***ConvNet***) = **Convolutional Neural Network** *(Réseau neuronal convolutif)*

Un avantage majeur des réseaux convolutifs est l'utilisation d'un poids unique associé aux signaux entrant dans tous les neurones d'un même noyau de convolution. Cette méthode réduit l'empreinte mémoire, améliore les performances[3] et permet une invariance du traitement par translation. C'est le principal avantage du réseau de neurones convolutifs par rapport au perceptron multicouche, qui, lui, considère chaque neurone indépendant et affecte donc un poids différent à chaque signal entrant.

2.1 - *Avantage :*

Contrairement à un modèle MLP (Multi Layers Perceptron) classique qui ne contient qu'une partie classification, l'architecture du Convolutional Neural Network dispose en amont d'une partie convolutive et comporte par conséquent deux parties bien distinctes :

- Une partie convolutive : Son objectif final est d'extraire des caractéristiques propres à chaque image en les compressant de façon à réduire leur taille initiale. En résumé, l'image fournie en entrée passe à travers une succession de filtres, créant par la même occasion de nouvelles images appelées cartes de convolutions. Enfin, les cartes de convolutions obtenues sont concaténées dans un vecteur de caractéristiques appelé code CNN.
- Une partie classification : Le code CNN obtenu en sortie de la partie convolutive est fourni en entrée dans une deuxième partie, constituée de couches entièrement connectées

appelées perceptron multicouche (MLP pour Multi Layers Perceptron). Le rôle de cette partie est de combiner les caractéristiques du code CNN afin de classer l'image. Pour revenir sur cette partie, n'hésitez pas à consulter [l'article sur le sujet] .

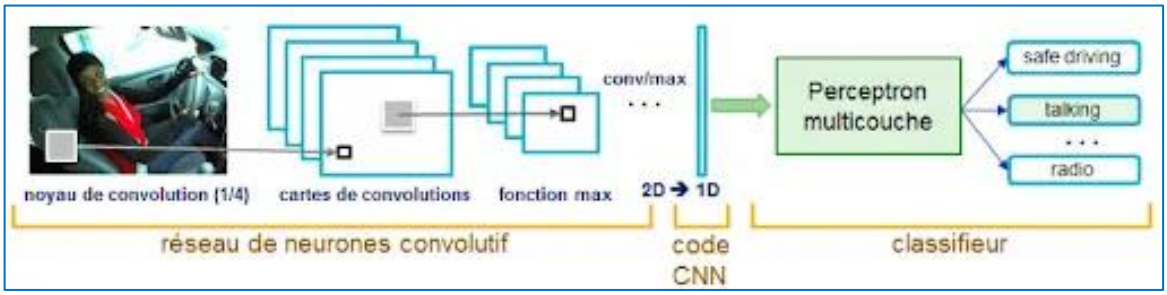


Schéma représentant l'architecture d'un CNN

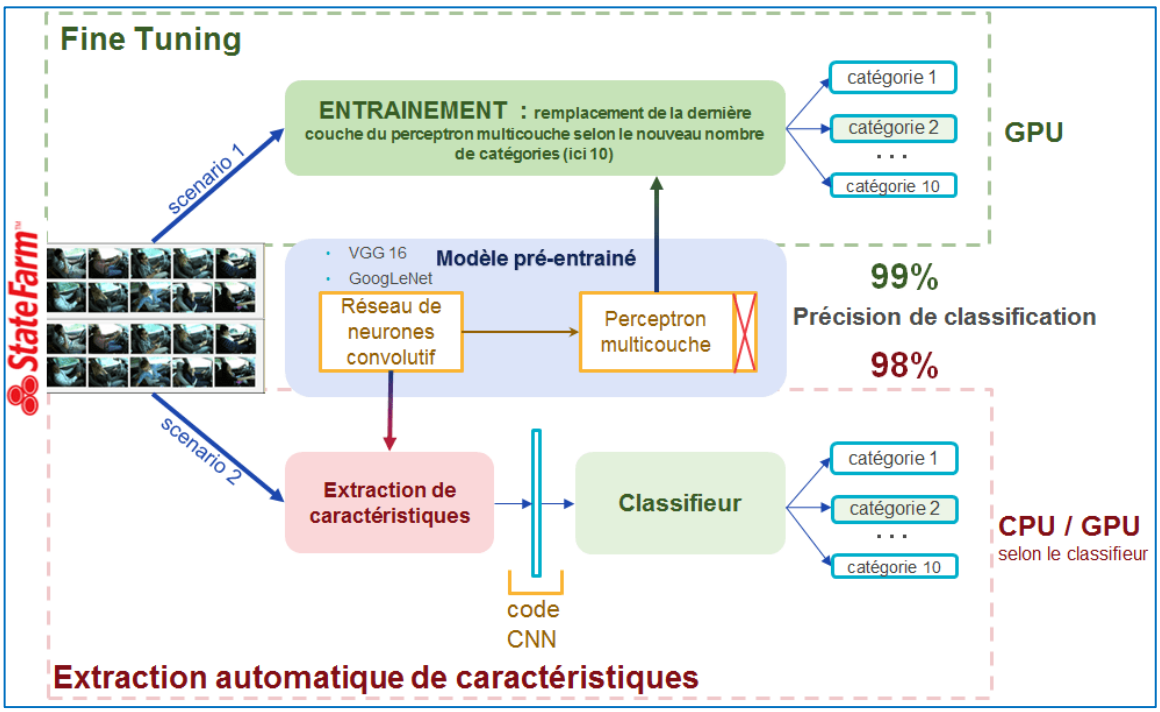
2.2 - *Transfert Learning :* *Adapter des CNN pré-entraînés*

Pour des usages pratiques, il est possible d'exploiter la puissance des CNN sans être un expert du domaine, avec du matériel accessible et une quantité raisonnable de données annotées. Toute la complexité de création de CNN peut être évitée en adaptant des réseaux pré-entraînés disponibles publiquement. Ces techniques sont appelées transfert learning, car on exploite la connaissance acquise sur un problème de classification général pour l'appliquer de nouveau à un problème particulier.

Les expériences présentées dans cet article ont été réalisées en partant du classique VGG-16. La "connaissance" sur la classification d'images contenue dans un tel réseau peut-être exploitée de deux façons :

- comme un extracteur automatique de caractéristiques des images, matérialisé par le code CNN,
- comme une initialisation du modèle, qui est ensuite ré-entraîné plus finement (Fine Tuning) pour traiter le nouveau problème de classification.

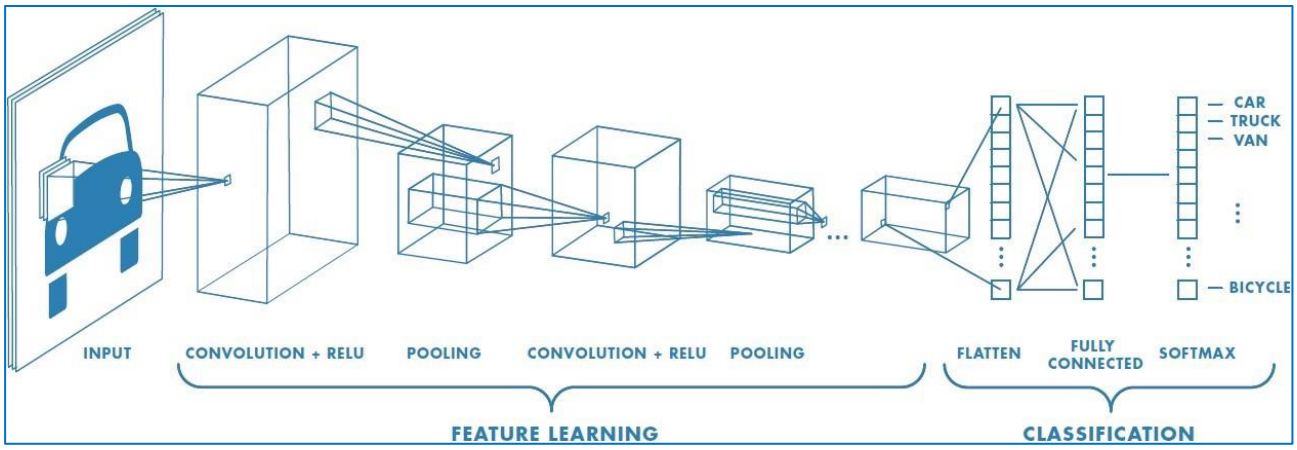
La première méthode fournit simplement un très bon modèle. Nous classifions correctement 98% des images de conducteurs. Le fine tuning demande un investissement un peu plus important, et permet d'améliorer la performance finale : 99% de précision sur notre problème.



3 : Réseau CNN

En apprentissage automatique, un réseau de neurones convolutifs ou réseau de neurones à convolution (en anglais CNN ou ConvNet pour Convolutional Neural Networks) est un type de réseau de neurones artificiels acycliques (feedforward), dans lequel le motif de connexion entre les neurones est inspiré par le cortex visuel des animaux. Les neurones de cette région du cerveau sont arrangés de sorte qu'ils correspondent à des régions qui se chevauchent lors du pavage du champ visuel. Leur fonctionnement est inspiré par les processus biologiques, ils consistent en un empilage multicouche de perceptrons, dont le but est de prétraiter de petites quantités d'informations. Les réseaux neuronaux convolutifs ont de larges applications dans la reconnaissance d'image et vidéo, les systèmes de recommandation et le traitement du langage naturel.

Perceptron: est un algorithme d'apprentissage supervisé de classifieurs binaires (c'est-à-dire séparant deux classes).



Exemple d'architecture d'un CNN

3.1 - *Couche convolution*

La composante clé des réseaux de neurones convolutifs, et constitue toujours au moins leur première couche.

Son but est de repérer la présence d'un ensemble de features dans les images reçues en entrée. Pour cela, on réalise un filtrage par convolution : le principe est de faire "glisser" une fenêtre représentant la feature sur l'image, et de calculer le produit de convolution entre la feature et chaque portion de l'image balayée. Une feature est alors vue comme un filtre : les deux termes sont équivalents dans ce contexte.

Par exemple, si la question est de distinguer les chats des chiens, les features automatiquement définies peuvent décrire la forme des oreilles ou des pattes.

3.2 - *Couche pooling*

Ce type de couche est souvent placé entre deux couches de convolution : elle reçoit en entrée plusieurs feature maps, et applique à chacune d'entre elles l'opération de pooling.

L'opération de pooling consiste à réduire la taille des images, tout en préservant leurs caractéristiques importantes.

Ainsi, la couche de pooling rend le réseau moins sensible à la position des features : le fait qu'une feature se situe un peu plus en haut ou en bas, ou même qu'elle ait une orientation légèrement différente ne devrait pas provoquer un changement radical dans la classification de l'image.

3.3 - *Activation*

La fonction d'activation sert avant tout à modifier de manière non linéaire les données. Cette non-linéarité permet de modifier spatialement leur représentation.

Dit simplement, la fonction d'activation permet de changer notre manière de voir une donnée.

Par exemple, si on a comme donnée : chaque semaine 50% des clients d'un magasin achètent des barres de chocolat; la fonction d'activation permettrait de changer la donnée en 50% des clients aiment le chocolat ou encore, 50% des clients prévoient d'acheter du chocolat chaque semaine.

3.4 - *Exemple avec tensor flow*

A. *Importer Tensor Flow*

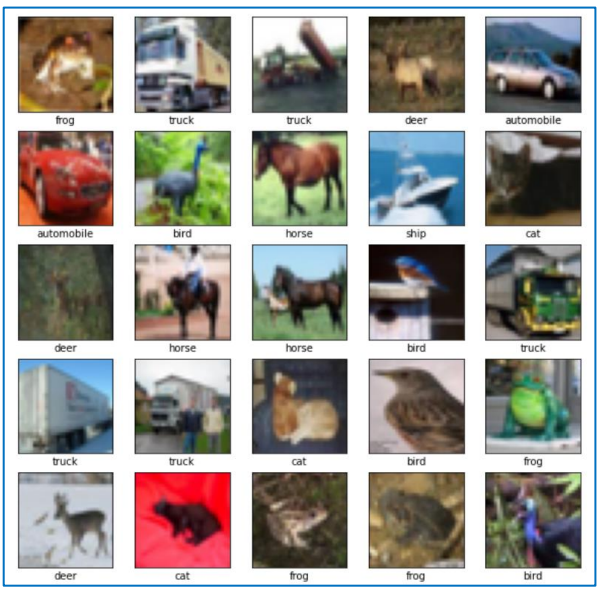
```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

B. *Télécharger et préparer le jeu de données CIFAR10*

```
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

C. *Vérifier les données*

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',  'dog',
'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



D. Créer la base convolutive

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32,
3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928

=================================================================
Total params: 56,320
Trainable params: 56,320
Non-trainable params: 0
_________________________________________________________________
```

E. Ajouter des couches denses sur le dessus

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 64)          36928

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

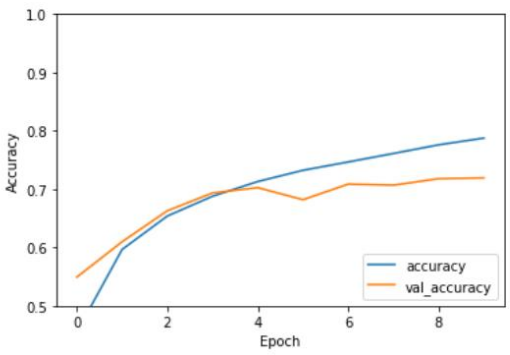
F. Compiler et entraîner le modèle

```
model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

G. Évaluer le modèle

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

```
313/313 - 1s - loss: 0.8475 - accuracy: 0.7192 - 634ms/epoch - 2ms/step
```




```
print(test_acc)
```

```
0.7192000150680542
```