TENSORFLOW

**Table des matières**

[I. Introduction 2](#_gjdgxs)

[II. Gradient / Optimisation 10](#_ihv636)

[III. Ecrire la régression linéaire 16](#_3im3ia3)

[IV. Modèle à couche : multilayer perceptron 25](#_1ade6im)

[V. Modèle à couche : Convolution Neural Network 31](#_3v3yxl4)

[VI. Couches et modèles personnalisés 42](#_2mgun19)

[VII. Callbacks 49](#_4cg47ux)

[VIII. Transfer Learning 56](#_3qqcayc)

# I. Introduction

Bienvenue sur le premier exercice du cours **introduction à TensorFlow** de DataScientest.com. Au fil de ce cours, vous aurez l’occasion de comprendre les aspects fondamentaux de l’apprentissage automatique avec **TensorFlow** : descente de gradient, réseau de neurones, perceptron, etc.

### Compétences requises

* Keras
* Programmation orientée objet

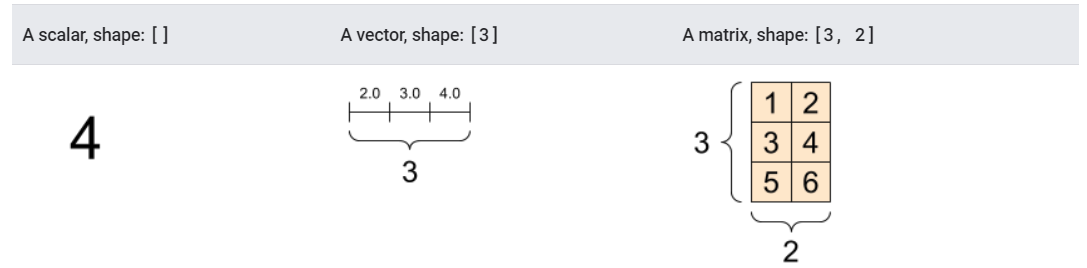
### Contexte

Avant de commencer, quelques mots sur **TensorFlow**. **TensorFlow** est le deuxième framework d’apprentissage automatique créé et utilisé par *Google* pour concevoir, construire et former des modèles d’apprentissage automatique. Par ailleurs, vous pouvez également utiliser la bibliothèque **TensorFlow** pour effectuer des calculs numériques.

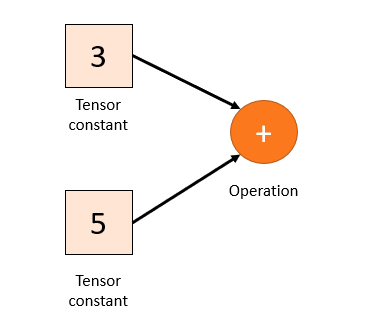
Cela n'a l'air de rien de particulier mais les calculs sont effectués avec des graphiques de flux de données. Dans ces graphiques, les nœuds représentent des opérations mathématiques (+, -, \* ...), tandis que les arêtes représentent des tenseurs. Les tenseurs sont généralement des tableaux de données multidimensionnels. Le nom «TensorFlow» dérive d'ailleurs des opérations que les réseaux de neurones effectuent sur des tableaux de données multidimensionnels/tenseurs! Il s'agit littéralement de flux de tenseurs.

### Comment fonctionne TensorFlow?

Le **tensor** est l'unité centrale de données dans tensorflow. C'est comme un tableau numpy, nous pouvons le concevoir comme l'abstraction à **n dimensions des matrices**. Un tenseur de dimension zéro est un scalaire (float) ou une constante. Un tenseur à 1 dimension est une liste ou un vecteur.



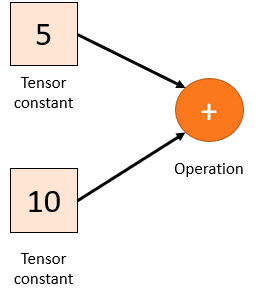
Ces tenseurs sont transmis aux opérations qui effectuent des calculs sur eux. Une **opération** est communément appelée **op**. Les opérations peuvent prendre plusieurs tenseurs en entrée, effectuent des opérations (addition, ...) et peuvent retourner plusieurs tenseurs. Par exemple, une opération comme tf.add peut prendre deux entrées 3 et 5 et produire leur somme 8.



Les **tensors** et les **opérations** sont reliés entre eux dans un graphe de calcul. Un graphe de calcul est défini en considérant les opérations comme des nœuds et les tenseurs comme des arêtes.

### Première mise en pratique

Pour introduire les concepts importants de **tensorflow**, nous allons dans le premier exemple calculer le résultat de 5+10 à l'aide de ce package en établissant le graph suivant :



### Tensor

Nous allons avoir besoin de définir les **constantes 5 et 10** comme des tenseurs, il existe pour cela la fonction constant du package **tensorflow**. La fonction a comme argument :

tf.constant(value, dtype=None, shape=None, name='Const')

* **value :** La valeur de la constante. La valeur de la constante peut être une valeur float, un string, une matrice array, ...
* **shape** (optionnel) : La forme de **value**. Si présent, il spécifie les dimensions du tenseur **value**. S'il n'est pas présent, la forme de value est utilisée.
* name (optionnel): Nom du tensor (très utile pour le retrouver dans le graphe des opérations).
* Importer le module **tensorflow** sous le nom **tf**.
* Créer une constante **x1** qui vaut 5.
* Créer une constante **x2** qui vaut 10.
* Afficher les deux constantes précédemment créées à l'aide de print.

import tensorflow as tf

x1 = tf.constant(5)

x2 = tf.constant(10)

print(x1)

print(x2)

tf.Tensor(5, shape=(), dtype=int32)

tf.Tensor(10, shape=(), dtype=int32)

### Opérations

**TensorFlow** dipose de nombreux outils afin d'effectuer des **opérations**. Une **opération**, aussi appelée *op*, est une fonction de **TensorFlow** qui :

* Prend en argument un nombre entier de Tensor.
* Effectue des calculs sur ceux-ci.
* Renvoie un nombre entier de Tensor.

Quelques fonctions faisant le parallèle entre les opérateurs numpy et tensorflow :

|  |
| --- |

| **Fonction Numpy** | **Fonction Tensorflow** |
| --- | --- |
| np.add or + | tf.add or + |
| np.subtract or - | tf.subtract or - |
| np.matmul | tf.matmul |
| np.transpose | tf.transpose |
| np.linalg.inv | tf.linalg.inv |
| np.cos | tf.cos |
| np.sin | tf.sin |
| np.tan | tf.tan |
| np.log | tf.log |
| np.exp | tf.exp |
| np.reshape | tf.reshape |
| np.concatenate | tf.concat |
| np.expand\_dims | tf.expand\_dims |
| np.arange | tf.range |

* A l'aide de la fonction add du module **TensorFlow** qui prend deux arguments de type Tensor, effectuer l'addition de x1 et x2. Stocker le résultat dans une variable resultat.
* Afficher le résultat de la variable **resultat** à l'aide de la fonction print.

resultat = tf.add(x1, x2)

print(resultat)

tf.Tensor(15, shape=(), dtype=int32)

Nous pouvons remarquer ici que **resultat** est un tensor de valeur **15**. Une des grandes particularités de la version 2.0+ de **tensorflow** est la mise en place du "eager execution". Le "Eager execution" ou l'exécution rapide consiste à évaluer les opérations immédiatement. Les opérations renvoient des valeurs concrètes au lieu de créer un graphique de calcul à exécuter ultérieurement.

La méthode numpy appliquée à un tensor retourne sa valeur sous forme de tableau numpy ou de scalaire

* Afficher la valeur du tensor **resultat**.

resultat.numpy()

15

### Manipulation des tensors : Approximé le nombre pi avec Monte Carlo

Pour rappel, les tensors peuvent être représentés comme des objets array. Comme dans les objets arrays, il est possible de **créer** et **manipuler** de la même façon les tensors :

# Import du module numpy sous l'alias 'np'

import tensorflow as tf

# Création d'une matrice de dimensions 5x10 remplie de zéros

X = tf.zeros(shape = (5, 10))

# Création d'une matrice à 3 dimensions 3x10x10 remplie de uns

X = np.ones(shape = (3, 10, 10))

Il est aussi possible de créer un tensor à partir d'une liste ou d'un array à l'aide du constructeur tf.constant:

# Création d'un tensor à partir d'une liste définie en compréhension

X = tf.constant([2\*i for i in range(10)]) # 0, 2, 4, 6, ..., 18

# Création d'un tensor à partir d'un array.

X = np.array([[1, 3, 3],

[3, 3, 1],

[1, 2, 0]])

X = tf.constant(X)

Voici un tableau des concordance entre numpy et tensorflow:

| **Fonction Numpy** | **Fonction Tensorflow** |
| --- | --- |
| np.zeros | tf.zeros |
| np.ones | tf.ones |
| np.eye | tf.eye |
| np.transpose | tf.transpose |
| np.random.normal | tf.random.normal |
| np.random.uniform | tf.random.uniform |

* Créer un tensor aléatoire X de taille [10000,2] suivant une loi uniforme de paramètre 0, 1.
* Afficher le min et le max du tensor à l'aide de la fonction reduce\_min et reduce\_max de tensorflow. Est-ce que le résultat est cohérent ?

X = tf.random.uniform([10000, 2])

print('Min of X :', tf.reduce\_min(X).numpy())

print('Max of X :', tf.reduce\_max(X).numpy())

Min of X : 8.010864e-05

Max of X : 0.99996424

## Indexation d'un tensor

L'indéxation sur les tensors est très similaire à celle des arrays. En effet, l'indéxation doit se faire en indiquant l'index que nous voulons accéder **sur chaque dimension**:

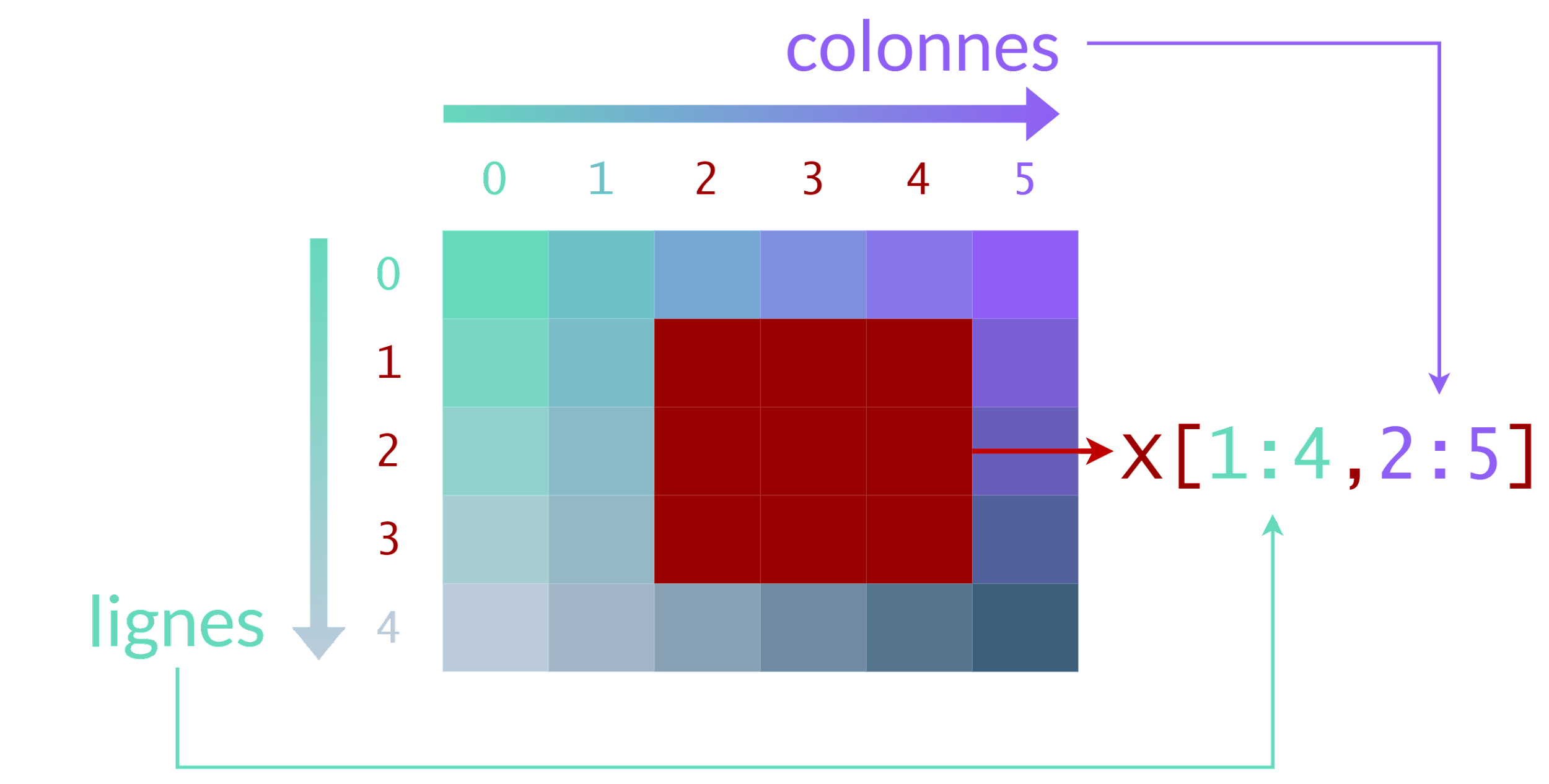
import tensorflow as tf

X = tf.zeros(shape = [10,10])

## affichage de l'élément à l'index (4, 3)

print(X[4,3])

Il est également possible de la même façon de récupérer une partie du tensor :



X = tf.zeros(shape = [4,5])

## affichage une sous matrice de X

print(X[1:4,2:5]

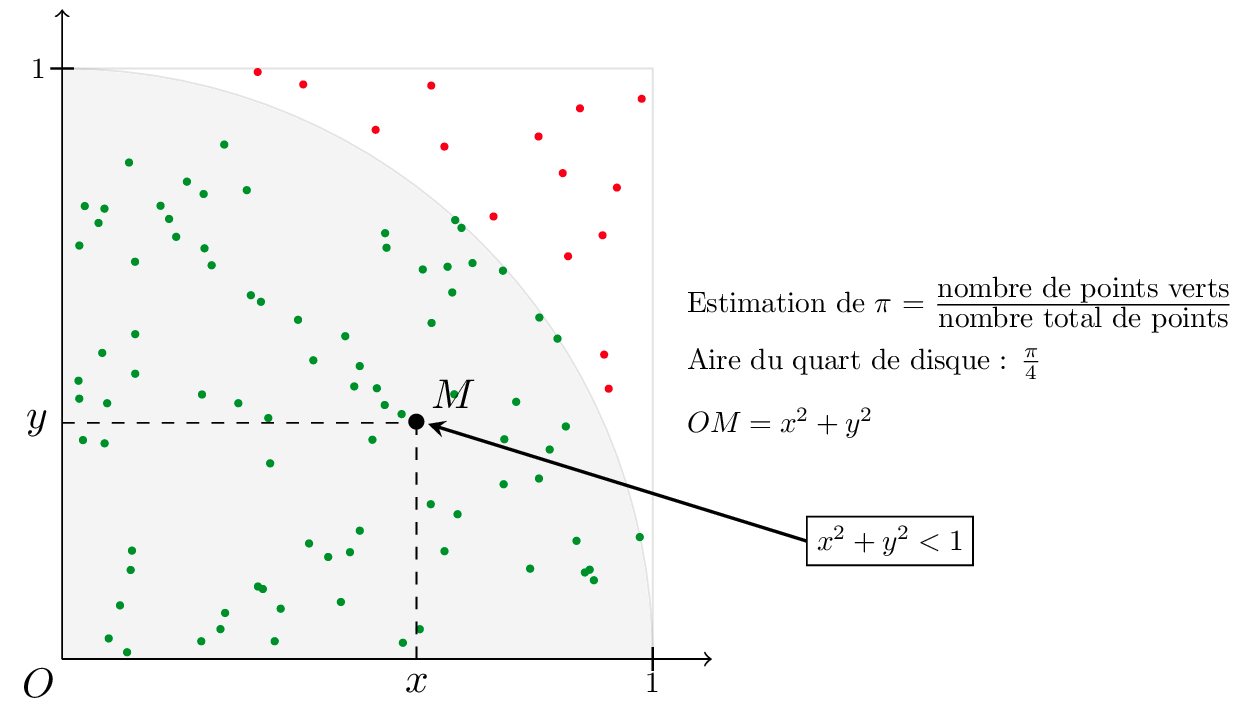
* Que retourne X[:,0]\*\*2+X[:, 1]\*\*2 < 1 ?

X[:,0]\*\*2+X[:, 1]\*\*2 < 1

<tf.Tensor: shape=(10000,), dtype=bool, numpy=array([ True, True, True, ..., True, True, True])>

Maintenant que les bases sont posées, essayons d'estimer la valeur pi à l'aide de la méthode de Monte Carlo. La méthode consiste à :

* Tirer aléatoirement des points de coordonnées x, y suivant une loi uniforme de paramètre 0 et 1.
* Compter l'ensemble des points dans le cercle de centre (0,0) et de rayon 1.
* Diviser la somme précédente par le nombre de point, le résultat sera une approximation de *π*4.



* Dans une boucle for avec len(X) itérations, vérifier si la condition X[i,0]\*\*2+X[i, 1]\*\*2 < 1 est vrai. Si oui, incrémenter une variable **a** (initialisé à 0).
* Afficher la valeur a.
* À l'aide de a, trouver une approximation de pi.

a = 0

for i in range(len(X)):

if X[i,0]\*\*2+X[i, 1]\*\*2 < 1:

a += 1

print('Point in the cercle :', a)

print('PI Approximation :', a/len(X)\*4)

Point in the cercle : 7890

PI Approximation : 3.156

Pour l'instant, nous ne remarquons pas de grande différence entre **tensorflow** et **numpy**. Si vous rechercher à faire des calcules matricielles ou des opérations, il est préférable pour une question d'optimisation d'utiliser **numpy**. Nous verrons dans prochain exercice que le graphe de calcul de **tensorflow** rend possible le calcul du gradient ou des primitives d'une fonction.

## Ce qu'il faut retenir

### Tensor

Le **tensor** est l'unité centrale de données dans tensorflow. C'est comme un tableau numpy, nous pouvons le concevoir comme l'abstraction à n dimensions des matrices.

### Opération

Les tenseurs sont transmis aux opérations qui effectuent des calculs sur eux. Les **opérations** peuvent prendre plusieurs tenseurs en entrée, effectuent des opérations (addition, ...) et peuvent retourner plusieurs tenseurs. Quelques fonctions faisant le parallèle entre les opérateurs numpy et tensorflow :

| **Fonction Numpy** | **Fonction Tensorflow** |
| --- | --- |
| np.matmul | tf.matmul |
| np.transpose | tf.transpose |
| np.linalg.inv | tf.linalg.inv |
| np.cos | tf.cos |
| np.sin | tf.sin |
| np.tan | tf.tan |
| np.log | tf.log |
| np.exp | tf.exp |
| np.reshape | tf.reshape |
| np.concatenate | tf.concat |
| np.expand\_dims | tf.expand\_dims |
| np.arange | tf.range |

### Graphe

Les **tensors** et les **opérations** sont reliés entre eux dans un graphe de calcul. Un graphe de calcul est défini en considérant les opérations comme des nœuds et les tenseurs comme des arêtes.

### Eager Execution :

Une des grandes particularités de la version 2.0+ de **tensorflow** est la mise en place du "eager execution". Le "Eager execution" ou l'exécution rapide consiste à évaluer les opérations immédiatement. Les opérations renvoient des valeurs concrètes au lieu de créer un graphique de calcul à exécuter ultérieurement.

# II. Gradient / Optimisation

L'exercice précédent a permis de présenter quelques notions importantes sur tensorflow, comme les opérations, les constantes... Mais, pour l'instant, nous ne remarquons pas de grande différence entre **tensorflow** et **numpy**. Dans cet exercice, nous allons voir une des grande force de tensorflow, le calcul du gradient ou des primitives d'une fonction à l'aide du graphe de calcul.

Pour introduire ces notions, nous allons calculer le gradient à un point donnée de la fonction f et trouver son minimum :

*f*(*x*)=*x*²−2*x*+3

Pour effectuer ces tâches, il va être nécessaire d'introduire la notion de **variable** sur tensorflow.

## Les variables sur Tensorflow

Nous n'avons que pour l'instant travailler sur des constantes sur tensorflow. Elles peuvent être très rapidement limitées puisqu'une fois définie, il n'est plus possible de modifier leur valeur. C'est la qu'intervient les **variables**. Les opérations effectuées sur les constantes peuvent aussi l'être sur les variables, mais, elles ont pour particularité de pouvoir être modifier après définition.

Sur **tensorflow**, le constructeur Variable permet de définir une variable. Il est alors possible avec cet objet de pouvoir modifier sa valeur au courant de l'exécution. Pour instancier cet objet, il est important de spécifier en argument la valeur initiale de la variable **initial\_value**.

La méthode assign appliquée à la **variable** permet d'assigner une nouvelle valeur à la variable :

## Definition of the variable X.

X = tf.Variable(3.0)

## Assign a new value at X.

X.assign(8.0)

* Importer tensorflow sous le nom tf.
* Créer une variable **x** à l'aide de fonction Variable en l'initialisant à 3.0.
* Définir une fonction f avec comme argument **x** retournant la valeur de f(x)

*f*(*x*)=*x*²−2*x*+3

* Evaluer *f*(3).

Les opérations classiques (+, -, \*, /) fonctionnent avec les tenseurs flottants. De plus, il est possible d'ajouter des flottants, entiers, etc. à des tenseurs.

import tensorflow as tf

x = tf.Variable(3.0)

def f(x):

return x\*\*2 - 2\*x + 3

f(x).numpy()

6.0

* À l'aide de la méthode assign, changer la valeur de x dans le but d'évaluer f(4).
* Afficher la valeur f(4).

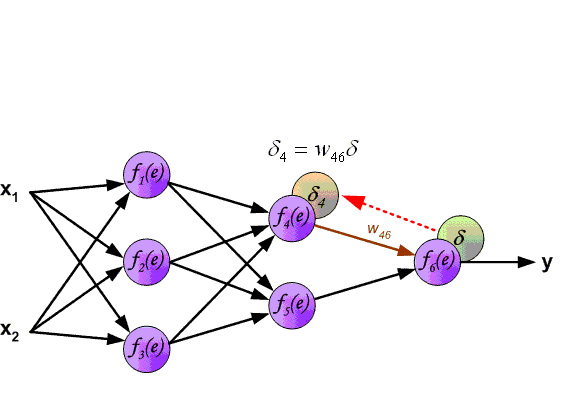
x.assign(4.0)

f(x).numpy()

11.0

Pour l'instant, nous n'avons pas vu de grande différence entre **Tensorflow** et **Numpy**. Nous allons maintenant calculer le gradient en x de la fonction *f* pour x=4.

En effet, **Tensorflow** permet de calculer les propriétés d'une fonction comme sa primitive ou sa dérivée. Il rend également possible la propagation d'une erreur de la sortie du graphique vers l'entrée (backpropagation). Cette propriété est la base des méthodes d'optimisations en Deep Learning.



TensorFlow fournit la fonction GradientTape pour calculer le gradient en fonction de ses variables d'entrées. Tensorflow "enregistre" toutes les opérations exécutées dans la bande de GradientTape. À l'aide des opérations exécutées dans la bande, tensorflow déduit le gradient de la fonction.

Prenons par exemple la fonction *y*=*x*∗*x*. Le gradient pour *x*=3 peut être calculer par:

x = tf.Variable(3.0)

with tf.GradientTape() as tape:

y = x \* x

grad = tape.gradient(y, x)

* Calculer le gradient de la fonction f pour x = 4.0.

with tf.GradientTape() as tape:

function = f(x)

grad = tape.gradient(function, x)

print(grad.numpy())

6.0

* Calculer analytiquement ∂*f*(*x*)/∂*x* pour x = 4.

# gradient\_f = 2\*x-2

print('Gradient of f for x=4 :', 2\*4-2)

Gradient of f for x=4 : 6

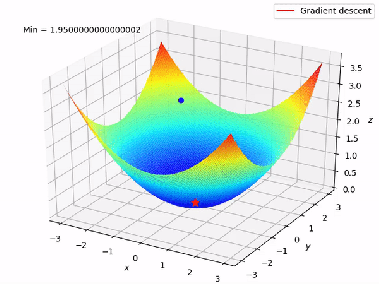
Une fois le gradient calculé, il est très facile de trouver un minimum local de la fonction à l'aide de la méthode de gradient.

Voici une courte explication de la méthode :

from IPython.display import YouTubeVideo

YouTubeVideo('Ka4AnSXxF2g', width=800, height=500)

Par conséquent, la descendre de gradient consiste initialement à choisir une position aléatoire, et de suivre l’opposé de la pente (gradient) à chaque itération :



*X**t*+1=*X**t*−*η*⋅∂*J**/*∂*X*

Avec :

* *X**t* : position

* *η* : pas du gradient, autrement dit, la vitesse d’apprentissage

* *J* : fonction à minimiser
* En utilisant l'algorithme de descente de gradient, trouver le minimum de la fonction.
* Quelle est la valeur du gradient en ce point ?

learning\_rate = 0.1

nb\_iteration = 100

x.assign(4.0)

for i in range(nb\_iteration):

# Compute f(4)

with tf.GradientTape() as tape:

function = f(x)

# Compute gradient of f(4)

grad = tape.gradient(function, x)

x.assign(x - learning\_rate\*grad)

print('Gradient :', grad.numpy())

print('Solution :', x.numpy())

Gradient : 4.7683716e-07

Solution : 1.0000002

## Optimizer sur Tensorflow

Dans la partie précédente, nous avions défini l'algorithme de descente de gradient manuellement. Tensorflow est également doté d'un ensemble de méthodes d'optimisation prédéfini, les méthodes se trouvent dans le sous-package tensorflow.keras.optimizers.

Il existe notamment l'optimizer SGD correspondant à la descente de gradient :

optimizer = tf.keras.optimizers.SGD(

learning\_rate=0.01,

momentum=0.0,

nesterov=False)

* Définir un optimizer SGD sous le nom **optimizer** avec un learnin\_rate=0.1.

optimizer = tf.keras.optimizers.SGD(learning\_rate = 0.1)

Une fois la méthode d'optimizer définie, la méthode apply\_gradients permet d'appliquer une itération de la méthode d'optimisation (ex :descente de gradient). La méthode nécessite en argument le rapport entre les valeurs du gradient et les variables:

# Update parameters with an iteration of gradient descent

optimizer.apply\_gradients(zip(grads, variables))

**Pour chaque itération**, il est alors nécessaire de suivre les étapes suivantes pour trouver le minimum de la fonction *y*=*x*2 :

* Calculer le gradient de la fonction à minimiser
* Appliquer une descente de gradient à l'aide de la méthode apply\_gradients

# Gradient compute.

with tf.GradientTape() as tape:

y = x \* x

grad = tape.gradient(y, x)

# Compute the optimizer method, refresh variables.

optimizer.apply\_gradients(zip([grad], [x]))

* Trouver le minimum de la fonction f avec la méthode d'optimisation SGD.

  Nous pouvons remarquer que le package **keras** est disponible sur **tensorflow**. Les prochains exercices permettront de mieux comprendre ce lien.

nb\_iteration = 1000

x = tf.Variable(initial\_value=3.0)

optimizer = tf.keras.optimizers.SGD(learning\_rate=0.1)

for i in range(nb\_iteration):

# Compute

with tf.GradientTape() as tape:

function = f(x)

# Compute gradient

grad = tape.gradient(function, x)

optimizer.apply\_gradients(zip([grad], [x]))

x.numpy()

1.0000002

## Ce qu'il faut retenir

### Constante sur Tensorflow

Les constantes sont des tensors qui ne peuvent pas changer de valeur. Généralement, les fonctions ci-dessous génèrent des constantes sur **tensorflow**.

| **Fonction Numpy** | **Fonction Tensorflow** |
| --- | --- |
| np.zeros | tf.zeros |
| np.ones | tf.ones |
| np.eye | tf.eye |
| np.transpose | tf.transpose |
| np.random.normal | tf.random.normal |
| np.random.uniform | tf.random.uniform |

La fonction constant permet également de convenir un objet en constante.

### Variable sur Tensorflow

Les variables quant à elle sont des tensors modifiables après définition. Les opérations effectuées sur les constantes peuvent aussi l'être sur les variables. Elles deviennent très intéressantes quand nous sommes face à un problème d'optimisation où il va être nécessaire d'ajuster les paramètres (variables) au cours des itérations.

### Gradient / Optimisation

**Tensorflow** permet de calculer les propriétés d'une fonction comme sa primitive ou sa dérivée. Il rend également possible la propagation d'une erreur de la sortie du graphique vers l'entrée (backpropagation). Cette propriété est la base des méthodes d'optimisations en Deep Learning.

# III. Ecrire la régression linéaire

Les deux exercices précédents ont permis d'introduire les notions suivantes : concept de graphe, d'opération, tensors, gradients et optimisation en **TensorFlow**.

Nous allons maintenant nous confronter à un vrai problème de machine learning, résoudre un problème de régression à l'aide d'une régression linéaire.

Pour simplifier l'exercice, le jeu de données sera généré artificiellement. Nous allons utiliser la fonction make\_regression de **sklearn.datasets.samples\_generator** dans le but de traiter un **problème simple de régression**.

* Exécuter la cellule suivante pour changer et afficher le jeu de données.

%matplotlib inline

from sklearn.datasets import make\_regression

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

import numpy as np

X, y = make\_regression(n\_samples=100,

n\_features=1,

n\_informative=1,

noise=10,

random\_state=0)

X = np.squeeze(X)

# plot data

plt.scatter(X, y, color='blue', alpha=0.5)

plt.xlabel('x')

plt.ylabel('y')

plt.title('Our dataset')

# display plot

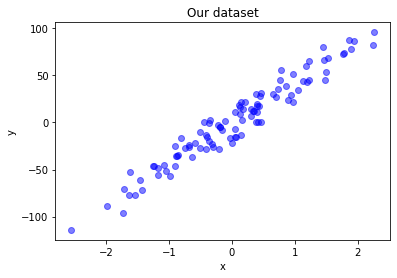
print('Shape of X :', X.shape)

print('Shape of y :', y.shape)

plt.show()

Shape of X : (100,)

Shape of y : (100,)



* Combien y'a-t-il d'observation ?
* Combien y'a-t-il de colonnes qui décrivent l'observation

print(X.shape)

print('nb\_observation :', len(X))

print('nb\_columns :', 1)

(100,)

nb\_observation : 100

nb\_columns : 1

Nous disposons d’un ensemble de couples {(*x**i*,*y**i*)}1≤*i*≤*N* de nombres réels. La régression linéaire consiste à trouver une relation linéaire entre x et y:

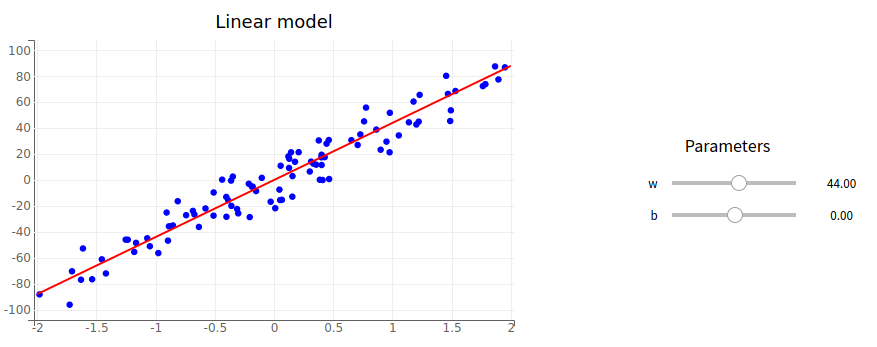
*y*̂ *i*=*b*+*w*⋅*x**i*

Les inconnues du problème sont **w** et **b** (géométriquement, ce sont la pente de la droite et son ordonnée à l’origine).

* Exécuter la cellule suivante pour afficher la droite de régression.

from interaction\_intro import show\_linear

show\_linear(X, y)



Résoudre ce problème revient à trouver w et b qui minimisent l’erreur quadratique moyenne (MSE). Notre problème s’écrira alors :



* Exécuter la cellule suivante pour afficher la fonction de perte de notre modèle en fonction de w et b.
* Graphiquement, pour quelle couple (b,w) la fonction de perte est minimum ? Est-ce que le résultat est cohérent ?

  Dans le graphique 3D suivant : l'axe x correspond à **b**, l'axe y correspond à **w**, et l'axe z correspond à la valeur de la MSE

from interaction\_intro import show\_MSE

show\_MSE(X,y)

Nous recherchons le couple (w, b) qui minimise la fonction de coût.

Il est possible de donner une formule explicite de **w** et **b**. Mais, en dehors de ce cas particulier, il n’y a guère d’espoir de pouvoir résoudre aussi facilement un tel problème de minimisation.

En Deep Learning, nous sommes confrontés à des problèmes de minimisation complexe et sans solution explicite. Pour introduire les concepts fondamentaux du Deep Learning, nous allons utiliser une descente de gradient pour résoudre notre problème d'optimisation.

Maintenant que les bases sur les modèles de régression sont établies, implémentons la régression linéaire :

* Créer une classe sous le nom Linear:
  + Dans la fonction \_\_init\_\_:
    - Définir l'attribut **w** comme une variable initialisée par une loi normale de dimension **[1]** à l'aide de la fonction normal de **tf.random**.
    - Définir l'attribut **b** comme une variable initialisée par une loi normale de dimension **[1]** à l'aide de la fonction normal de **tf.random**.
  + Définir un callable en créant une fonction \_\_call\_\_ avec comme argument **self** et **inputs** retournant *y*̂ *i*=*b*+*w*⋅*i**n**p**u**t**s*

class Linear():

def \_\_init\_\_(self):

self.w = tf.Variable(tf.random.normal([1]), name='weight')

self.b = tf.Variable(tf.random.normal([1]), name='bias')

def \_\_call\_\_(self, inputs):

return inputs \* self.w + self.b

* Définir un modèle linéaire sous le nom model.
* Quelle est la prédiction du modèle pour x=3 ?

import tensorflow as tf

model = Linear()

model(3).numpy()

array([-1.4667366], dtype=float32)

## Entraînement

Dans la cellule précédente, nous avons défini notre modèle. Nous allons maintenant définir notre fonction de perte à minimiser ainsi qu'une méthode d'optimisation.

**Tensorflow** est doté d'un ensemble de fonction de perte dans le sous-module tensorflow.keras.losses. Ici, nous allons utiliser la fonction de perte mean\_squared\_error prenant comme arguments le tensor **y\_true** et **y\_pred**.

L'optimisateur va permettre de trouver les paramètres **b** et **w** qui minimise notre fonction de perte.

**Tensorflow** est également doté d'un ensemble de méthode d'optimisation se trouvant dans le sous-module tensorflow.keras.optimizers. En argument de ses fonctions, il est possible de modifier les hyperparamètres de la méthode d'optimisation.

Pour rappel, le pipeline d'une itération de la méthode d'optimisation consiste à :

* Calculer le gradient de la fonction de perte par :

with tf.GradientTape() as tape:

# Model prediction.

y\_pred = model(X)

# Compute the loss function.

loss\_value = loss(y\_true, y\_pred)

# Compute the gradient function.

grads = tape.gradient(loss\_value, model.parameters)

* Définir un optimisateur et appliquer une descente de gradient à l'aide de la méthode apply\_gradients de l'optimisateur en précisant en argument le rapport entre les valeurs du gradient et les variables :

# Update the weights of the model.

optimizer.apply\_gradients(zip(grads, model.paramaters))

* Créer un objet **model** de la classe Linear.
* Définir un optimizer sous le nom **optimizer** à l'aide de la fonction Adam de **tf.keras.optimizers** avec un **learning\_rate** de 0.1.
* Dans une fonction train\_op avec comme argument **model**, **inputs** et **targets** :
  + Calculer le gradient de la fonction de coût mean\_squared\_error de **tf.keras.losses**.
  + Entraîner le modèle à l'aide de la méthode apply\_gradients de **optimizer** en précisant comme argument la relation entre le gradient et la liste des poids du modèle.
  + Retourner la valeur réelle de la fonction de perte.
* Entraîner le modèle sur 1000 itérations en utilisant le jeu de données **X** et **y**. Stocker le résultat de la fonction de coût pour chaque itération dans la liste **grads**.

# Définition du modèle

model = Linear()

# Définition d'un optimisateur Adam

optimizer = tf.keras.optimizers.Adam(learning\_rate=0.1)

def train\_op(model, inputs, targets):

with tf.GradientTape() as tape:

# Prédiction de notre modèle

y\_pred = model(inputs)

# Calcule de l'erreur de notre modèle

loss\_value = tf.keras.losses.mean\_squared\_error(targets, y\_pred)

# Calculer le gradient de la fonction de perte

grads = tape.gradient(loss\_value, [model.w, model.b])

# Descente de gradient

optimizer.apply\_gradients(zip(grads, [model.w, model.b]))

# Retourner la valeur de la fonction de perte

return loss\_value.numpy()

steps = 1000

# Entraînement du modèle

grads = [train\_op(model, X, y) for i in range(steps)]

* Quel est le couple (w, b) pour lequel notre modèle à convergé ? Est-ce que la valeur est cohérente ?

print('b :', model.b.numpy())

print('w :', model.w.numpy())

b : [-0.7918286]

w : [42.248653]

## Évaluation du modèle

* Afficher graphiquement l'évolution de la fonction de perte en fonction de l'itération.
* Calculer la prédiction **y\_pred** de notre modèle pour tout le jeu de données.
* Afficher graphiquement les points (*x**i*,*y**i*) et la prédiction de notre modèle (*x**i*,*y**p**r**e**d**i*).

# Prédiction de notre modèle

y\_pred = model(X)

# Afficher l'évolution de la fonction de perte

plt.figure(figsize=(12,4))

plt.subplot(121)

plt.plot(grads)

plt.xlabel('Iteration')

plt.title('Valeur de la fonction de perte')

plt.subplot(122)

# Afficher les points (x, y).

plt.scatter(X, y, alpha=0.5, label = 'True value')

# Afficher la prédiction de X.

plt.plot(X, y\_pred, 'r', label = 'Prediction')

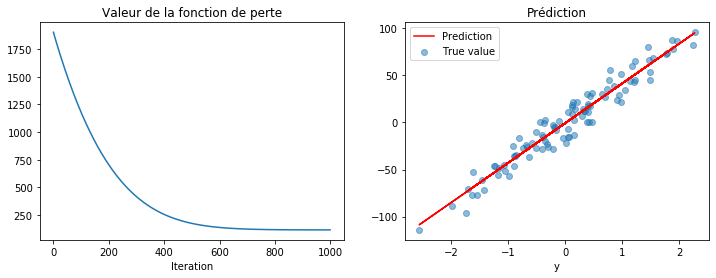
plt.xlabel('x')

plt.xlabel('y')

plt.title('Prédiction')

plt.legend()

plt.show()



## Modèle Polynomial : Lien entre Tensorflow et Keras

Nous avons précédemment utilisé un modèle linéaire pour résoudre notre problème de régression. Pour vous entraîner à manipuler des graphes ainsi qu'introduire une relation entre Keras et Tensorflow, nous allons maintenant utiliser un modèle polynomial d'ordre 3 sur le même problème :

*y*̂ *i*=*b*+*w*0*x**i*+*w*1*x*2*i*+*w*2*x*3*i*

* Exécuter la cellule suivante pour afficher la courbe de régression d'un modèle polynomial.

from interaction\_intro import show\_polynomial

show\_polynomial(X, y)

Dans la partie précédente, nous avons résolu le problème de régression en créant une classe Linear, puis pour chaque époque, nous avions calculé le gradient de la fonction de perte et enfin nous avions rétropopagé l'erreur à l'aide d'un optimisateur Adam.

Si, la classe Linear hérite de **tensorflow.keras.Model**, elle peut être considérée comme un modèle Keras. En conséquence, la classe héritera notamment de la méthode compile pour préciser la méthode d'optimisation et la fonction de coût, et la méthode fit pour entraîner le modèle.

Exemple sur les modifications à faire sur la classe Linear :

```python class Linear(tf.keras.Model): def **init**(self):

# Initialisation of tensorflow.keras.Model class

super(Linear, self).\_\_init\_\_()

# Variables definitions

self.w = tf.Variable(tf.random.normal([1]), name='weight')

self.b = tf.Variable(tf.random.normal([1]), name='bias')

def \_\_call\_\_(self, inputs):

return inputs \* self.w + self.b

* Créer une classe sous le nom Polynomial:
  + Dans la fonction \_\_init\_\_:
    - Initialiser les éléments hérités de notre classe.
    - Définir les attributs **w0, w1, w2, b**
  + Dans la fonction \_\_call\_\_ avec comme argument **self**, **inputs** et **training=True** retournant *y*̂ *i*=*b*+*w*0⋅*x**i*+*w*1⋅*x*2*i*+*w*2⋅*x*3*i*

  Il est nécessaire d'initialiser la classe **tensorflow.keras.Model**, pour pouvoir utiliser ses méthodes. Pour le faire, il est nécessaire de préciser dans l'initialisation de la classe super(Linear, self).\_\_init\_\_().

class Polynomial(tf.keras.Model):

def \_\_init\_\_(self):

super(Polynomial, self).\_\_init\_\_()

self.w0 = tf.Variable(tf.random.normal([1]), name='w0')

self.w1 = tf.Variable(tf.random.normal([1]), name='w1')

self.w2 = tf.Variable(tf.random.normal([1]), name='w2')

self.b = tf.Variable(tf.random.normal([1]), name='bias')

def \_\_call\_\_(self, inputs, training = True):

return self.w0 \* inputs + self.w1 \* inputs \*\* 2 + self.w2 \* inputs \*\* 3 + self.b

* Créer un objet **model** de la classe Polynomial.
* Définir un optimizer sous le nom **optimizer** à l'aide de la fonction Adam de **tf.keras.optimizers** avec un **learning\_rate** de 0.1.
* Compiler le modèle à l'aide de sa méthode compile en précisant notre **optimizer** et une fonction de perte **'mean\_squared\_error'**.
* Entraîner le modèle sur les données **X** et **y** à l'aide de la méthode fit:
  + L'entraînement devra se faire sur 100 epochs (paramètre **epochs**)
  + Les batchs devront avoir une taille de 16 (paramètre **batch\_size**)
  + La sortie de l'entraînement devra être stockée dans une variable nommée **training\_history**.

model = Polynomial()

optimizer = tf.keras.optimizers.Adam(learning\_rate = 0.1)

model.compile(optimizer = optimizer, loss = 'mean\_squared\_error')

training\_history = model.fit(X, y, epochs = 100, batch\_size= 16)

* Afficher graphiquement l'évolution de **loss** en fonction des itérations.
* Afficher graphiquement les points (*x**i*,*y**i*) et la prédiction de notre modèle (*x**i*,*y**p**r**e**d**i*).

# Prédiction de notre modèle

y\_pred = model(np.linspace(-2.5, 2.5, 100))

# Afficher l'évolution de la fonction de perte

plt.figure(figsize=(12,4))

plt.subplot(121)

plt.plot(training\_history.history['loss'])

plt.xlabel('Iteration')

plt.title('Valeur de la fonction de perte')

plt.subplot(122)

# Afficher les points (x, y).

plt.scatter(X, y, alpha=0.5, label = 'True value')

# Afficher la prédiction de X.

plt.plot(np.linspace(-2.5, 2.5, 100), y\_pred, 'r', label = 'Prediction')

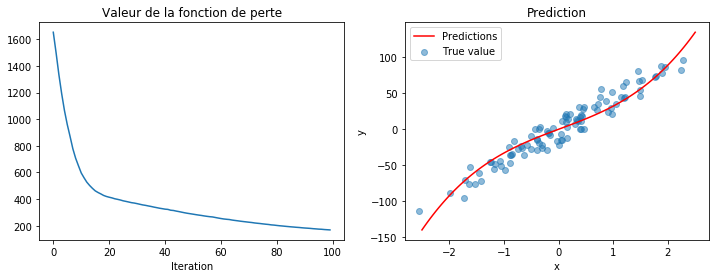
plt.xlabel('x')

plt.ylabel('y')

plt.title('Prediction')

plt.legend()

plt.show()



## Ce qu'il faut retenir

### Gradient / Optimisation

**Tensorflow** permet de calculer les propriétés d'une fonction comme sa primitive ou sa dérivée. Il rend également possible la propagation d'une erreur de la sortie du graphique vers l'entrée (backpropagation). Cette propriété est la base des méthodes d'optimisations en Deep Learning.

### Fonction de perte

**Tensorflow** est doté d'un ensemble de fonction de perte dans le sous-module tensorflow.keras.losses. Voici un tableau de correspondance :

| **Fonction de perte** | **Fonction Tensorflow** |
| --- | --- |
| Entropie croisée: problème de classification binaire | binary\_crossentropy |
| Entropie croisée: problème multi classification | categorical\_crossentropy |
| Erreur absolue moyenne: problème de régression | mean\_absolute\_error ou mae |
| Erreur quadratique moyenne: problème de régression | mean\_squared\_error ou mse |

### Optimizer

**Tensorflow** est également doté d'un ensemble de méthode d'optimisation se trouvant dans le sous-module tensorflow.keras.optimizers. En argument de ses fonctions, il est possible de modifier les hyperparamètres de la méthode d'optimisation.

### Hérédité des propriétés des modèles de keras.

Il est possible de faire hériter à notre modèle la classe **tensorflow.keras.Model** dans le but de la considérer comme un modèle Keras. En conséquence, la classe héritera notamment de la méthode compile pour préciser la méthode d'optimisation et la fonction de coût, et la méthode fit pour entraîner le modèle.

# IV. Modèle à couche : multilayer perceptron

Dans les exercices précédents, nous avons introduit le concept de graphe, d'optimisation et quelques relations entre keras et tensorflow. L'objectif de cet exercice est de pouvoir réintroduire quelques notions vues dans le module **Introduction au Deep Learning avec Keras**.

Pour ce faire, nous allons traiter un problème simple de classification sur la base **MNIST**.

* Exécuter la cellule suivante pour charger et visualiser le jeu de données **MNIST**.

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

from tensorflow.keras.datasets.mnist import load\_data

# Load MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = load\_data()

# Show 10 randoms numbers.

j = 1

plt.figure(figsize=(14, 5))

for i in np.random.randint(low=0, high=len(X\_train), size=[10]):

plt.subplot(2, 5, j)

plt.axis('off')

plt.imshow(X\_train[i], cmap='gray')

plt.title('Number ' + str(y\_train[i]))

j+=1

# Reshape

X\_train = X\_train.reshape([-1, 28\*28])

X\_test = X\_test.reshape([-1, 28\*28])

# Shape of X\_train and y\_train

print('Shape of X:', X\_train.shape)

print('Shape of y:',y\_train.shape)

Shape of X: (60000, 784)

Shape of y: (60000,)



## Tensorflow et Keras

La version de **tensorflow** 2.0+ a été construite autour du framework **keras**. La structure de **keras** se trouve dans le sous-package **tensorflow.keras**.

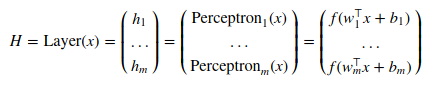
Vous pouvez retrouver toutes les "couches de neurones" de **Keras** dans **tensorflow.keras.layers**.

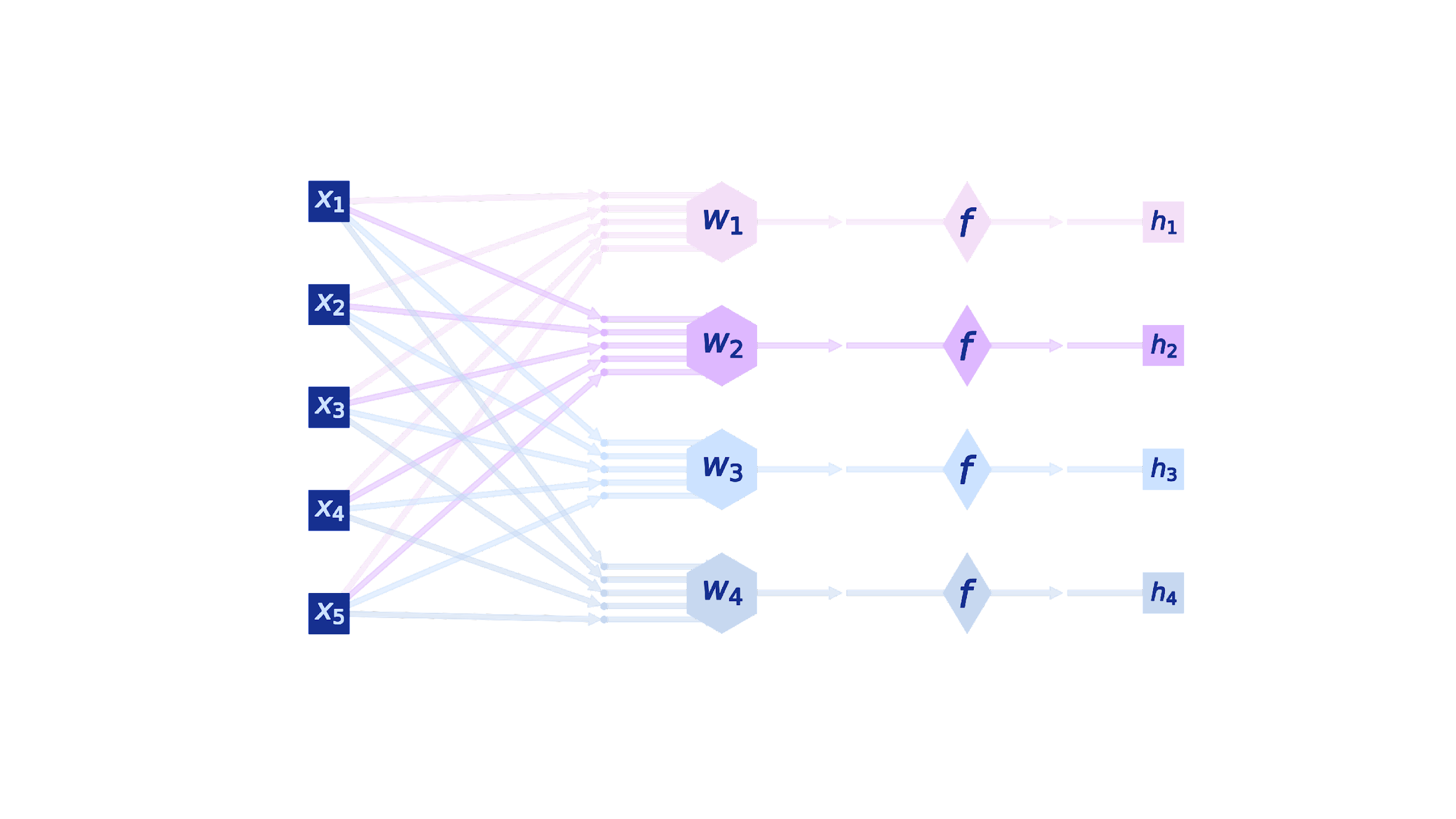
| **Définition** | **Fonction** |
| --- | --- |
| Dense layer | Dense |
| Convolution 2D | Conv2D |
| Dropout | Dropout |
| Batch Normalization | BatchNormalization |
| Average Pooling 2D | AveragePooling2D |
| Flatten | Flatten |
| RNN | RNN |
| LSTM | LSTM |
| GRU | GRU |

## Rappel sur les couches Denses

Une couche *Dense* ou *Fully-connected* correspond à une **pile** de perceptrons dont la sortie est **un vecteur des sorties de chaque Perceptron**. Un vecteur de poids est associé à chaque Perceptron de la couche.

Pour un vecteur d'entrée donné *x*, la sortie *H* d'une couche Perceptron est :





Dans l'animation interactive suivante, nous avons illustré pour vous chaque étape de ces opérations:

* La couche d'entrée est de dimension 5, donc la taille du vecteur d'entrée et du vecteur de poids de chaque neurone de la couche dense suivante sera de 5.
* La première couche dense contient 3 neurones, donc la taille de son vecteur de sortie sera de 3 et la taille du vecteur de poids de chaque neurone de la couche dense suivante sera de 3.
* La deuxième couche dense contient 4 neurones, donc la taille de son vecteur de sortie sera de 4.
* Pour interagir avec la figure, cliquez sur le bouton ***Next*** pour aller à l'étape suivante et ***Previous*** pour revenir à l'étape précédente.

from interaction\_layers import show\_dense

show\_dense()

## Multi layers perceptrons

Le modèle Perceptron multicouche est une **séquence de couches Perceptron** dont l'entrée est la sortie de la **couche précédente**.

Considérons un modèle Perceptron à 3 couches. Pour un vecteur d'entrée donné *x*, la sortie de la première couche est :

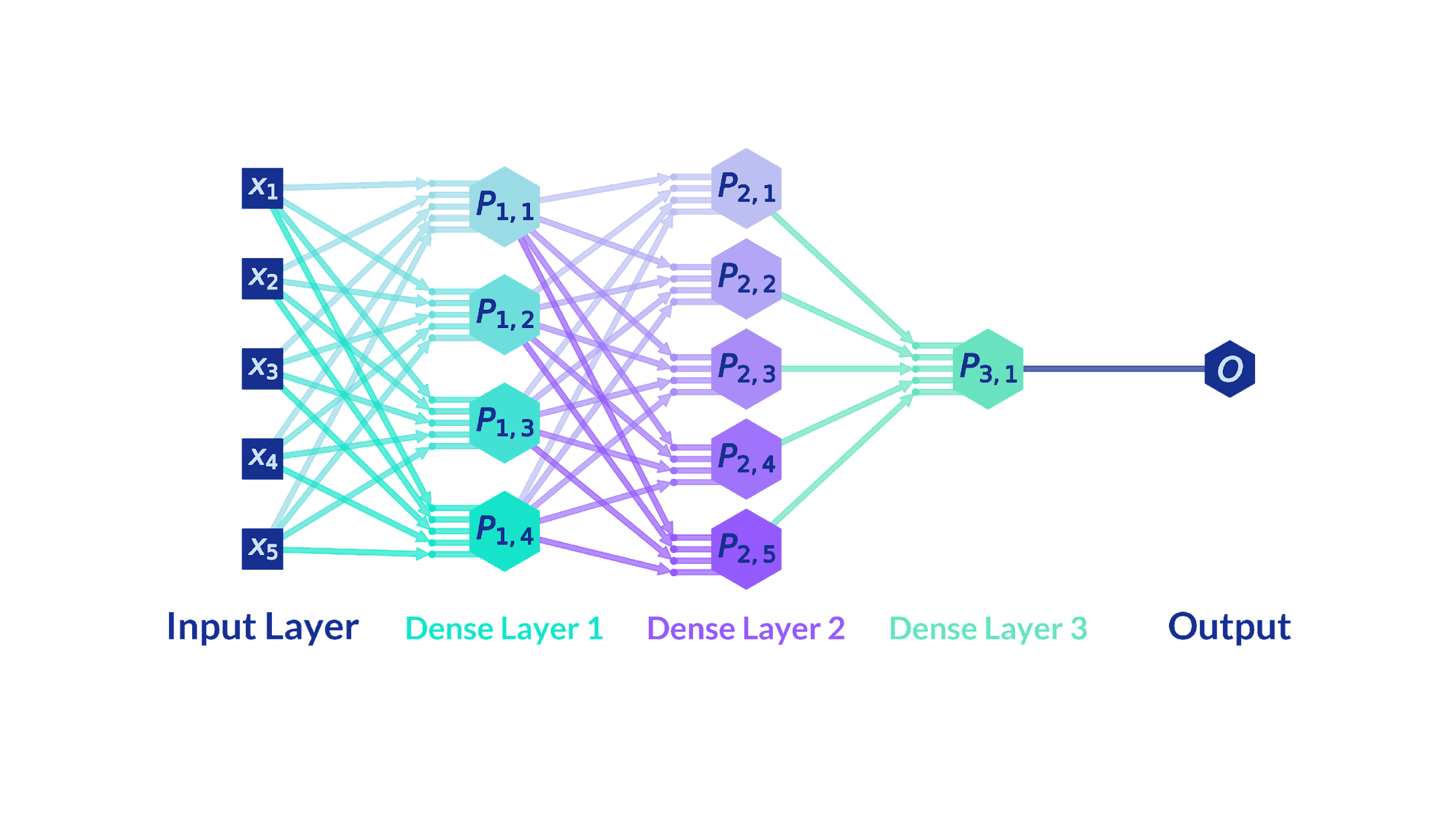
*H*1=Layer1(*x*)

Ensuite, le vecteur *H*1 est donné en entrée à la deuxième couche :

*H*2=Layer2(*H*1)

Enfin, le vecteur *H*2 est donné en entrée à la troisième couche pour obtenir la sortie finale du modèle :

*O*=Layer3(*H*2)



Les perceptrons multicouches peuvent être construits **sequentiellement** en empilant les couches denses les unes après les autres.

## Implémentation sur Keras

Keras permet d'implémenter très facilement des réseaux de neurones multi-couche :

from tensorflow.keras import Sequential

from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(4, activation='relu'))

model.add(Dense(5, activation='relu'))

model.add(Dense(1, activation=None))

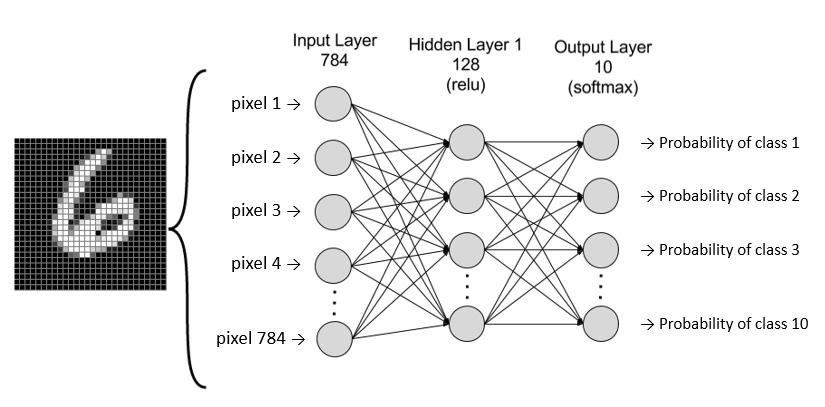
model.compile(loss)

model.fit(X\_train, y\_train, batch\_size=32, epochs=10)

* Le constructeur Sequential permet de mettre en place une architecture de neurone par couche. Ensuite, la méthode add permet d'ajouter les couches de neurones une après l'autre.
* La couche Dense correspond un ensemble de perceptron connecté à la couche précédente.
* La méthode compile permet de spécifier une fonction de perte et une méthode d'optimisation.
* La méthode fit permet d'entraîner le modèle sur l'ensemble d'entraînement.

## Modèle

Nous allons implémenter le modèle illustrée dans l'image suivante:



* Instancier un modèle séquentiel sous le nom **model** à l'aide du constructeur Sequential de **tensorflow.keras**.
* Ajouter au **model** une couche Dense avec 128 neurones, une fonction d'activation **'relu'** et une input\_shape **[28\*28]**.
* Ajouter au **model** une couche Dense avec 10 neurones et une fonction d'activation **'softmax'**.

import tensorflow as tf

from tensorflow.keras.layers import Dense

# Sequential

model = tf.keras.Sequential()

# Hidden layer 1

model.add(Dense(128, activation='relu', input\_shape=[28\*28]))

# Output Layer

model.add(Dense(10, activation='softmax'))

* Compiler le modèle à l'aide de sa méthode compile. Comme nous avons un problème de classification, nous allons utiliser la fonction de perte **'sparse\_categorical\_crossentropy'**. L'optimiseur sera **'adam'** et la métrique sera **['accuracy']**.

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

Entraîner le modèle sur les données **X\_train** et **y\_train** à l'aide de la méthode fit:

* L'entraînement devra se faire sur 20 epochs (paramètre **epochs**)
* Les batchs devront avoir une taille de 64 (paramètre **batch\_size**)
* La performance du modèle devra être évaluée sur un échantillon de validation (X\_test, y\_test) (paramètre **validation\_data**)
* La sortie de l'entraînement devra être stockée dans une variable nommée **history**.

history = model.fit(X\_train, y\_train, epochs = 20, batch\_size = 64, validation\_data = (X\_test, y\_test))

* Afficher la courbe de la fonction de coût et de précision en fonction de l'epoch en exécutant la cellule suivante.

import matplotlib.pyplot as plt

plt.figure(figsize=(12,4))

plt.subplot(121)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model loss by epoch')

plt.ylabel('loss')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='right')

plt.subplot(122)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

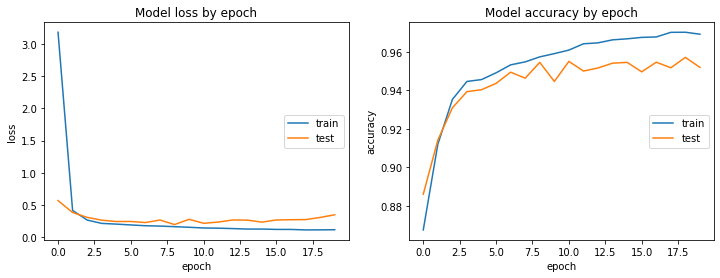
plt.title('Model accuracy by epoch')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='right')

plt.show()



Dans cet exercice, nous avons vu que nous pouvons définir des modèles par couche de la même manière que sur keras. Dans le prochain exercice, nous allons présenter les **datasets** sur **tensorflow** et entraîner un modèle plus adapté à de l'image : Réseau de neurones convolutifs (CNN)

## Ce qu'il faut retenir

La version de tensorflow 2.0+ a été construite autour du framework [**keras**](https://www.tensorflow.org/guide/keras).

Vous pouvez retrouver toutes les fonctionnalités de **keras** dans **tensorflow.keras**. Il y'a notamment les "couches de neurones" dans **tensorflow.keras.layers** :

| **Définition** | **Fonction** |
| --- | --- |
| Dense layer | Dense |
| Convolution 2D | Conv2D |
| Dropout | Dropout |
| Batch Normalization | BatchNormalization |
| Average Pooling 2D | AveragePooling2D |
| Flatten | Flatten |
| RNN | RNN |
| LSTM | LSTM |
| GRU | GRU |

# V. Modèle à couche : Convolution Neural Network

Dans l'exercice précédent, nous avons vu instancié un modèle perceptron à multi couche à l'aide du sous-module keras. Ce type de couche n'est pas toujours très adapté pour extraire de l'information sur des données non-structurée (texte, audio, image ...). Ici, nous allons utiliser des couches plus adaptées à de l'image : Réseau de neurones convolutifs (CNN).

Nous allons également présenter comment automatiser les étapes de pré-processing avec les objets dataset sur **tensorflow** .

* Exécuter la cellule suivante pour charger et visualiser le jeu de données **MNIST**.

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

from tensorflow.keras.datasets.mnist import load\_data

# Load MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = load\_data()

# Show 10 randoms numbers.

j = 1

plt.figure(figsize=(14, 5))

for i in np.random.randint(low=0, high=len(X\_train), size=[10]):

plt.subplot(2, 5, j)

plt.axis('off')

plt.imshow(X\_train[i], cmap='gray')

plt.title('Number ' + str(y\_train[i]))

j+=1

# Reshape

X\_train = X\_train.reshape([-1,28,28,1])

X\_test = X\_test.reshape([-1,28,28,1])

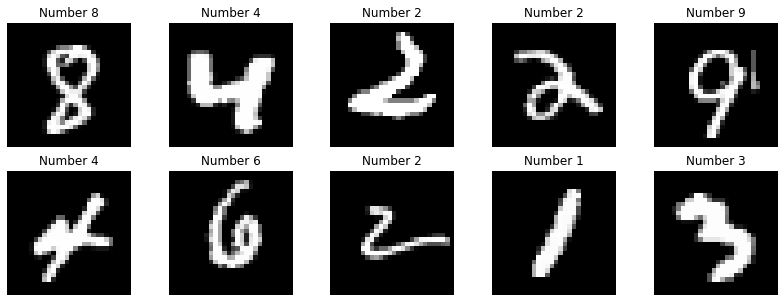
# Shape of X\_train and y\_train

print('Shape of X:', X\_train.shape)

print('Shape of y:',y\_train.shape)

Shape of X: (60000, 28, 28, 1)

Shape of y: (60000,)



Il arrive très régulièrement qu'il soit nécessaire d'ajouter un pré-processing sur nos données. Les datasets sur tensorflow sont conçus pour automatiser sous forme d'un pipeline l'ensemble des opérations.

## Dataset avec Tensorflow

Tensorflow possède un sous ensemble **tensorflow.data.dataset** ([plus d'informations](https://www.tensorflow.org/api_docs/python/tf/data/Dataset)) permettant d'appliquer un ensemble de transformation sur le jeu de données ainsi que de créer des itérateurs ou des générateurs. La fonction from\_tensor\_slices permet de transformer un **tuple** de tensor (ou array) en un objet **Dataset**.

dataset = tf.data.from\_tensor\_slices(tf.range(10))

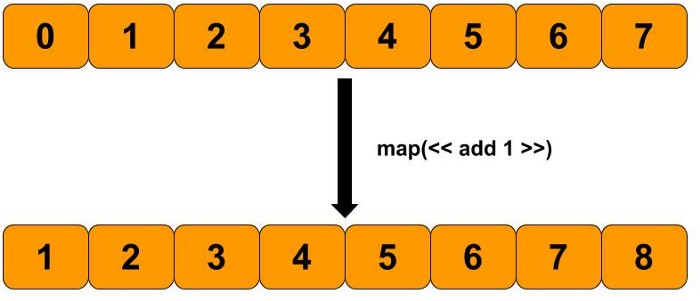
# Create a dataset with [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Une fois l'objet **Dataset** instancié, Tensorflow permet d'appliquer un pipeline de transformation en chaînant les appels de méthode sur l'objet tensorflow.data.Dataset. Les transformations ne seront faites uniquement quand l'algorithme en aura besoin. Les transformations les plus utilisées :

* map: Ajoute dans le pipeline une opération map permettant d'appliquer une opération à chaque élément du dataset.

# Add 1 at each element

dataset = dataset.map(lambda x : x+1)

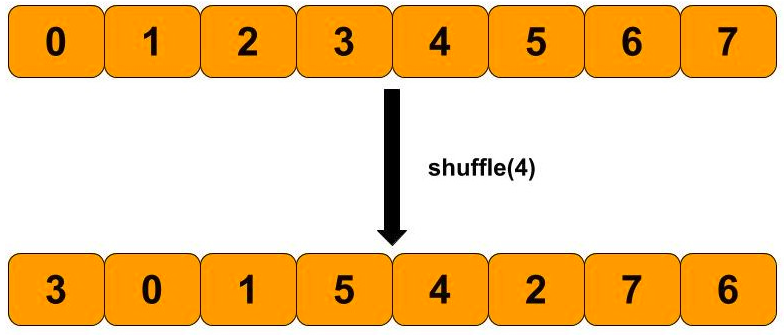


L'argument num\_parallel\_calls permet d'effectuer l'opération en multi-tâches.

* shuffle: Mélange de façon aléatoire le jeu de données. Le paramètre de cette méthode permet de déterminer combien d'éléments vont être mélangés au fur et à mesure. C'est à dire que la base va être séparer en plusieurs sous-bases de cette taille dans laquelle les éléments seront mélangés aléatoirement.

# Shuffle the dataset

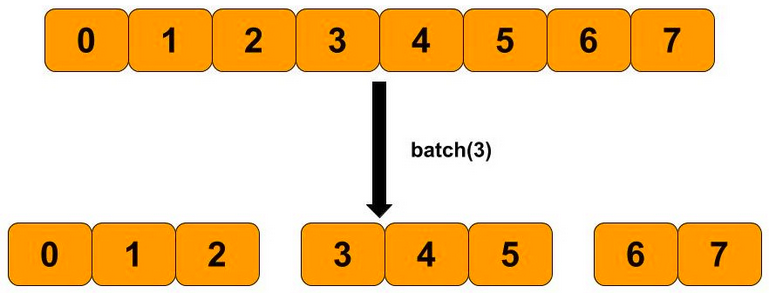
dataset = dataset.shuffle(buffer\_size=4)



* batch: Sépare, dans l'ordre, **le dataset** en lot de données. Il y aura autant d'élément dans chaque lot de données que le batch\_size fournit en paramètre de cette méthode. Sauf dans le dernière où il y en aura au plus le **batch\_size** :

# Batch at every 4 elements

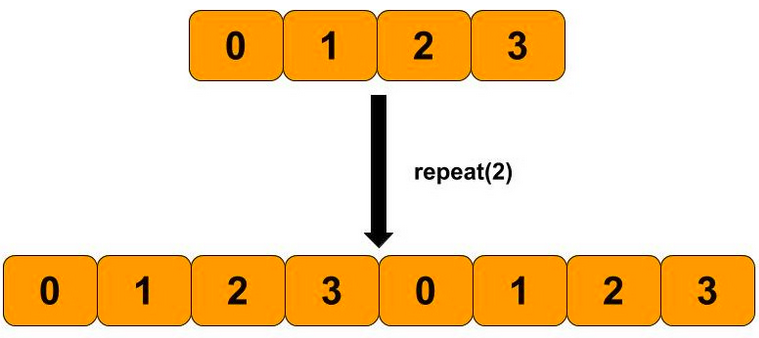
dataset = dataset.batch(batch\_size=3)



* repeat: Recopie autant de fois que voulu la base de données. Si le paramètre de l'argument **count** est None, alors le dataset sera répété indéfiniment.

# Duplicate the dataset

dataset = dataset.repeat(count=2)



Exemple d'un pipeline permettant de redimensionner les images et de les regrouper en lot :

# Create a dataset

dataset = tf.data.Dataset.from\_tensor\_slices((X, y))

# Resize each element

dataset = dataset.map(lambda x, y : [tf.image.resize(x, (128,128)), y])

# Shuffle and batch

dataset = dataset.shuffle(20000).batch(64)

* Importer le package tensorflow sous le nom **tf**.
* Définir un objet **dataset** à l'aide de la fonction from\_tensor\_slices du tuple **(X\_train, y\_train)**.
* Appliquer la transformation transform à **dataset** à l'aide de la méthode map.
* Mélanger le jeu de données à l'aide de la méthode shuffle en précisant un **buffer\_size** de 20000.
* Séparer le jeu de données en batch à l'aide de la méthode batch en précisant un **batch\_size** de 64.

def transform(x, y):

# Reshape data

img = tf.reshape(x, (28,28,1))

# Augmentation data

img = tf.image.random\_crop(img, [25,25,1])

# Return resize image

return tf.image.resize(img, (28,28)), y

import tensorflow as tf

# Create a dataset

dataset = tf.data.Dataset.from\_tensor\_slices((X\_train, y\_train))

# Reshape each element

dataset = dataset.map(transform)

# Shuffle and batch

dataset = dataset.shuffle(20000).batch(64)

Notre objet **Dataset** est une séquence de batchs. Il se comporte comme la séquence range(N) de 0 à N. La méthode take appliquée au **Dataset** permet d'extraire un nombre d'éléments de la séquence. Ce nombre d'élément doit être précisé dans l'argument **count** de la méthode.

* Exécuter la cellule suivante pour afficher les 2 premiers éléments de la séquence de **dataset**

for (X\_b, y\_b) in dataset.take(count=2):

print('Shape of the batch X :', X\_b.shape)

print('Target elements of the batch :', y\_b.numpy(), '\n')

Shape of the batch X : (64, 28, 28, 1)

Target elements of the batch : [1 1 0 4 5 7 7 1 2 9 7 7 8 5 0 6 4 2 9 8 1 9 0 5 3 8 3 4 3 5 3 1 0 7 7 8 3

1 2 1 2 3 8 2 1 4 7 0 3 5 1 8 9 6 4 9 0 1 1 7 7 5 5 8]

Shape of the batch X : (64, 28, 28, 1)

Target elements of the batch : [4 2 7 2 4 2 6 9 8 2 6 4 3 9 9 8 2 0 8 6 4 4 3 3 9 7 2 9 3 8 7 2 4 6 6 9 0

5 8 9 5 2 0 3 3 2 4 2 8 1 3 2 9 2 4 4 3 8 8 4 2 6 0 2]

Maintenant que le dataset est mis en forme, il est temps de s'intéresser au modèle. Ici, nous allons utiliser un réseau de neurones convolutifs (en anglais CNN ou ConvNet pour Convolutional Neural Networks).

## Rappel sur les convolutions

Les couches de convolution sont très utilisées pour la *classification d'images*. Avant d'aborder directement les opérations effectuées par une couche de convolution, il est important de comprendre l'opération à la base du neurone de cette couche, **le produit de convolution**.

D'un côté, nous avons une matrice d'entrée, souvent appelée **tuile** ou *convolution patch* en anglais, et d'un autre nous avons une matrice de convolution, souvent appelée **noyau de convolution**, **filtre** ou *convolution kernel* en anglais.

Les étapes du produit de convolution sont les mêmes que celles du produit scalaire:

* Produit terme à terme des deux matrices.
* Somme des produits.

Dans la figure interactive suivante, nous illustrons la convolution d'une image de dimensions 4x4 par un noyau de convolution de dimensions 3x3:

* Etape 1: Découpage de l'image en plusieurs tuiles de même taille que le kernel. Chaque tuile a une couleur différente.
* Etape 2: Produits de convolutions entre chaque tuile et le noyau.
* Etape 3: Activation et concaténation des produits en une nouvelle matrice.
* Pour interagir avec la figure, cliquez sur le bouton *Next* pour aller à l'étape suivante et *Previous* pour revenir à l'étape précédente.

from interaction\_layers import show\_conv

show\_conv()

Pour quelle raison faisons-nous cette opération ? Quels sont les avantages à utiliser cette technique ?

Comme vous allez le voir, la convolution d'une image est très facilement interprétable.

Dans la figure interactive suivante, nous illustrons la convolution d'une photo du Taj-Mahal par un noyau de convolution de dimensions 3x3.

La photo est en noir et blanc pour qu'elle corresponde à une matrice de pixels simple, mais l'opération peut être réalisée sur les matrices des composantes rouges, vertes et bleues d'une photo en couleur.

* Pour interagir avec cette figure, vous pouvez soit sélectionner un des noyaux de convolutions de la liste, soit modifier manuellement les coefficients du noyau.

from interaction\_layers import show\_filter

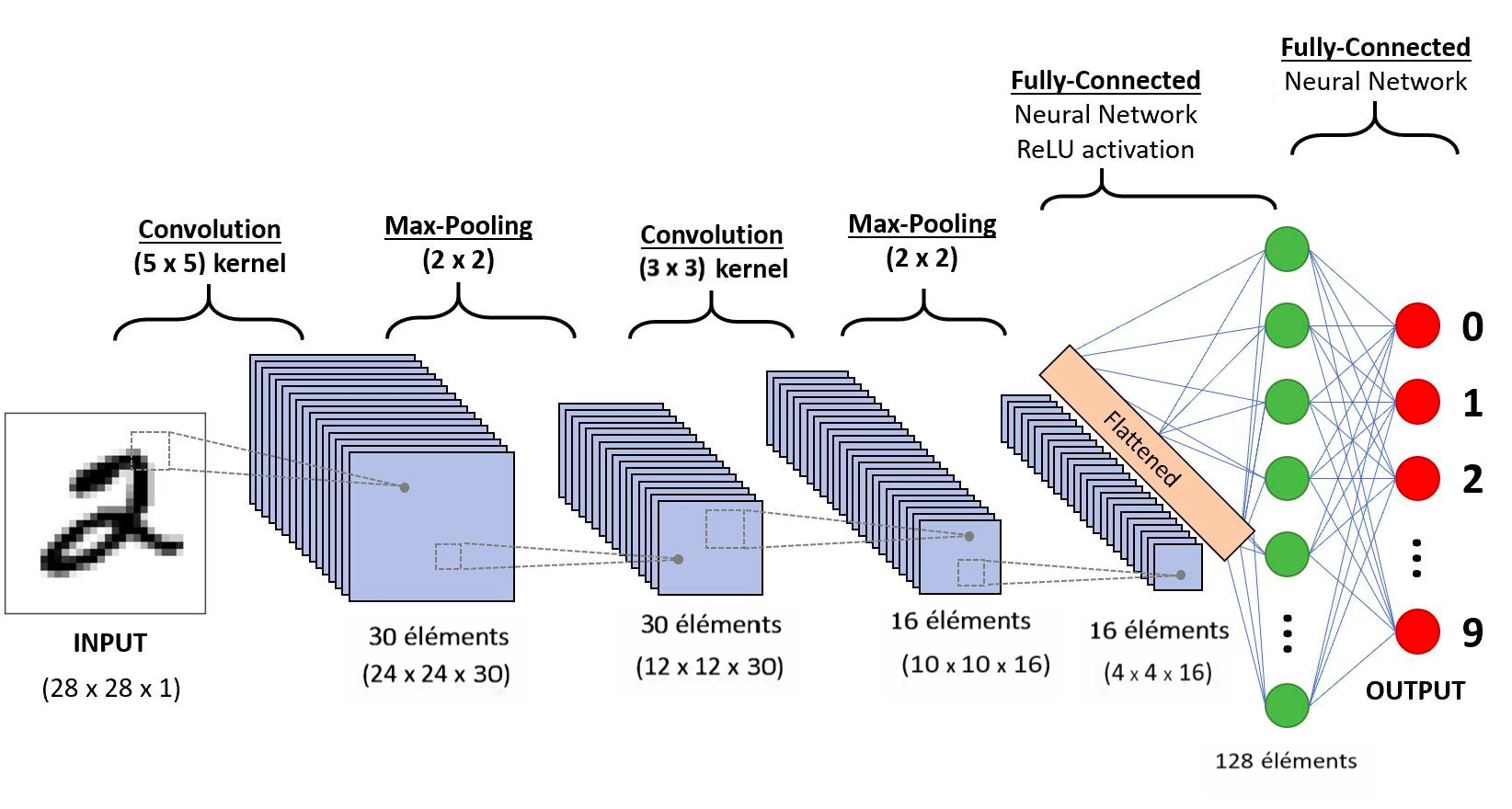
show\_filter()

La convolution d'une image peut être vue comme une extraction de features. Par exemple, les noyaux servant à détecter les bords pourraient être utiles pour classifier certaines formes géométriques.

L'intérêt des neurones de convolution est que par l'entraînement il pourront trouver eux-mêmes les **meilleurs noyaux de convolution à utiliser pour extraire des features à partir d'une image**.

## Architecture de LeNet

Nous allons maintenant instancier le modèle LeNet pour résoudre la problématique. L'architecture est constituée des couches suivantes:

* **Convolution 1** : 30 filtres, dim input (28, 28, 1), dim noyau (5, 5), Activation ***ReLU***.
* **Max-Pooling 1** : Dim pool (2, 2).
* **Convolution 2** : 16 filtres, dim noyau (3, 3), activation ***ReLU***.
* **Max-Pooling 2** : Dim pool (2, 2).
* **Aplatissement** : Flatten.
* **Dropout** : Connexions coupées à 20%.
* **Dense 1** : 128 neurones, activation ***ReLU***.
* **Dense 2** : 10 neurones, activation ***softmax***.
* Instancier un modèle séquentiel lenet et ajouter l'ensemble des couches.

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dropout, Dense

lenet = tf.keras.Sequential()

lenet.add(Conv2D(filters = 30, # Number of output filters

kernel\_size = (5, 5), # Kernel shape

input\_shape = (28, 28, 1), # Input shapeD

activation = 'relu')) # Activation function

lenet.add(MaxPooling2D(pool\_size = (2, 2)))

lenet.add(Conv2D(filters = 16,

kernel\_size = (3, 3),

activation = 'relu'))

lenet.add(MaxPooling2D(pool\_size = (2, 2)))

lenet.add(Flatten())

lenet.add(Dropout(rate = 0.2))

lenet.add(Dense(units = 128, activation = 'relu'))

lenet.add(Dense(units = 10, activation = 'softmax'))

* Compiler le modèle à l'aide de sa méthode compile. Comme nous avons un problème de classification, nous allons utiliser la fonction de perte **'sparse\_categorical\_crossentropy'**. L'optimiseur sera **'adam'** et la métrique sera **['accuracy']**.

lenet.compile(loss = 'sparse\_categorical\_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

* Entraîner le modèle sur le **dataset** à l'aide de la méthode fit:
* L'entraînement devra se faire sur 20 epochs (paramètre **epochs**)
* La perfomance du modèle devra être évaluée sur un échantillon de validation (X\_test, y\_test) (paramètre **validation\_data**).
* La sortie de l'entraînement devra être stockée dans une variable nommée **history**.

history = lenet.fit(dataset, # Training dataset

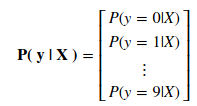
epochs = 20, # Number of epochs

validation\_data=(X\_test, y\_test) # Validation dataset

)

## Évaluation

La fonction d'activation **softmax** utilisée à la fin de notre modèle, fait que celui-ci retourne pour chaque image une probabilité de prédiction de chaque classe.



Pour prédire la classe de l'image, il suffit alors de trouver pour quelle classe la probabilité est maximale :

*y*̂ =argmax*y*(*P*( *y* | *X* )

* Prédire la probabilité des classes du jeu de données **X\_test**.
* Prédire dans **y\_pred** la classe la plus probable à l'aide de la fonction argmax de **tensorflow** en précisant **axis=1**.

# Probability prediction

y\_prob = lenet(X\_test)

# Label prediction

y\_pred = tf.argmax(y\_prob, axis=-1).numpy()

* Afficher le score de précision à l'aide de la fonction accuracy\_score de **sklearn.metrics**.
* Afficher la matrice de confusion à l'aide de la fonction confusion\_matrix de **sklearn.metrics**.

from sklearn.metrics import accuracy\_score

from sklearn.metrics import confusion\_matrix

print('Accuracy :', accuracy\_score(y\_test, y\_pred))

confusion\_matrix(y\_test, y\_pred)

Accuracy : 0.9857

array([[ 963, 0, 0, 0, 0, 1, 14, 2, 0, 0],

[ 1, 1128, 1, 0, 0, 0, 0, 5, 0, 0],

[ 0, 1, 1020, 7, 0, 0, 0, 4, 0, 0],

[ 0, 0, 0, 1006, 0, 2, 0, 2, 0, 0],

[ 0, 0, 1, 0, 972, 0, 1, 3, 0, 5],

[ 0, 0, 0, 11, 0, 879, 1, 0, 0, 1],

[ 0, 5, 6, 0, 4, 12, 928, 0, 3, 0],

[ 0, 0, 4, 0, 0, 0, 0, 1023, 1, 0],

[ 0, 0, 7, 4, 1, 7, 0, 1, 953, 1],

[ 0, 0, 2, 2, 10, 1, 0, 8, 1, 985]])

## Boucle d'entraînement personnalisée

Nous avons utilisé la méthode fit pour entraîner notre modèle. La méthode peut parfaitement convenir dans de nombreux modèles de deep learning. Toutefois, il arrive parfois que le format des données ne permette pas de l'utiliser.

Il est alors nécessaire de définir une méthode fit personnalisée pour entraîner notre modèle. L'approche est très similaire à l'exercice sur la régression linéaire.

## Fonction d'entraînement

Comme dans les exercices d'introduction, nous allons maintenant définir notre fonction de perte à minimiser ainsi qu'une méthode d'optimisation.

Ici, nous avons un problème de classification, nous allons alors utiliser la fonction de perte sparse\_categorical\_crossentropy de tensorflow.keras.losses. Etant donné qu'il y a plusieurs prédictions, il faudra prendre la moyenne de ce que retourne la fonction précédente (pour avoir la perte moyenne) que l'on peut faire grâce à la fonction reduce\_mean de **tensorflow**.

* Définir un optimizer sous le nom **optimizer** à l'aide de la fonction Adam de tensorflow.keras.optimizers avec un learning\_rate de 0.001.
* Dans une fonction train\_op avec comme argument **model**, **inputs** et **targets** :
  + Calculer le gradient de la moyenne reduce\_mean de la fonction de coût sparse\_categorical\_crossentropy. La liste des paramètres des objets Sequential sont contenues dans l'attribut **trainable\_variables** de l'objet.

with tf.GradientTape() as tape:

# model prediction

y\_pred = model(inputs, training=True)

# compute the loss function

loss\_value = tf.reduce\_mean(loss(targets, y\_pred))

# Compute the gradient of loss function

grads = tape.gradient(loss\_value, model.trainable\_variables)

* + Entraîner le modèle à l'aide de la méthode apply\_gradients de **optimizer** en précisant comme argument la relation entre le gradient et les poids du modèle.

optimizer.apply\_gradients(zip(grads, model.trainable\_variables))

* + Retourner la valeur réelle de la fonction de coût.

# Define Adam optimizer

optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001)

# Training function

def train\_op(model, inputs, targets):

with tf.GradientTape() as tape:

# model prediction

y\_pred = model(inputs, training=True)

# compute the loss function

loss\_value = tf.reduce\_mean(tf.keras.losses.sparse\_categorical\_crossentropy(targets, y\_pred))

# Compute the gradient of loss function

grads = tape.gradient(loss\_value, model.trainable\_variables)

# Gradient descent

optimizer.apply\_gradients(zip(grads, model.trainable\_variables))

# Return the loss function value

return loss\_value.numpy()

## Entraînement

Maintenant que le dataset et la fonction d'entraînement sont définis, il est temps d'entraîner le modèle. Pour chaque époque les opérations suivantes vont être nécessaires :

# Entraîner le modèle sur une époque

for X\_t, y\_t in dataset:

# Entraîner le modèle sur un batch

train\_op(model, X\_t, y\_t)

# Fonction de coût pour l'ensemble de validation

loss = tf.reduce\_mean(loss(y\_test, lenet(X\_test))).numpy()

* Entraîner le modèle sur 5 epochs. Afficher pour chaque epoch la valeur de la fonction de perte sur le jeu de données de test.

grads = []

epochs = 5

# Entraînenement du modèle

for i in range(epochs):

# Pour chaque epoch

for X\_t, y\_t in dataset:

#Entraîner le modèle pour chaque batch

train\_op(lenet, X\_t, y\_t)

# Fonction de coût pour l'ensemble de validation

loss = tf.reduce\_mean(tf.keras.losses.sparse\_categorical\_crossentropy(y\_test, lenet(X\_test.reshape([-1,28,28,1])))).numpy()

print('Epoch', i, ' Loss', loss)

grads.append(loss)

# Or

# lenet.compile('adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# lenet.fit(dataset, batch\_size=64, epochs=10)

import matplotlib.pyplot as plt

# Afficher l'évolution de la fonction de perte

plt.plot(grads)

plt.xlabel('Epoch')

plt.title('Valeur de la fonction de perte')

plt.show()

## Ce qu'il faut retenir

La version de tensorflow 2.0+ a été construite autour du framework [**keras**](https://www.tensorflow.org/guide/keras).

Vous pouvez retrouver toutes les fonctionnalités de **keras** dans **tensorflow.keras**. Il y'a notamment les "couches de neurones" dans **tensorflow.keras.layers** :

| **Définition** | **Fonction** |
| --- | --- |
| Dense layer | Dense |
| Convolution 2D | Conv2D |
| Dropout | Dropout |
| Batch Normalization | BatchNormalization |
| Average Pooling 2D | AveragePooling2D |
| Flatten | Flatten |
| RNN | RNN |
| LSTM | LSTM |
| GRU |  |

Globalement, la méthodologie pour résoudre un problème à l'aide d'outil de deep learning :

1. Définir un **dataset** permettant de mettre en forme les données et de partitionner en batchs.
2. Construire un modèle.
3. Compiler le modèle : définition d'une fonction de perte, métrique, optimizer.
4. Entraîner le modèle, il y'a deux manières équivalentes de le faire:
   * Méthode fit: problème simple.
   * Calculer le gradient de la fonction de coût puis rétropropager l'erreur: problème complexe.
5. Prédiction et évaluation du modèle.

# VI. Couches et modèles personnalisés

Les deux précédents exercices ont présenté quelques couches standards très utile en Deep learning (Dense, Conv2D...). Quelques fonctions de perte ou métriques prédéfinies ont été également utilisées.

Il s'agit maintenant de personnaliser le tout: de la création de couches et de modèles, la définition des loss et métriques, jusqu'à l'entrainement et la prédiction.

## Couche personnalisée : Classe Layer

La meilleure façon de créer ses propres couches personnalisées est de faire hériter la classe Layer. La classe devra également avoir au moins les 3 méthodes suivantes :

* \_\_init\_\_: Initialise la classe Layer et définit les hyperparamètres de la couche.
* build : Initialise les poids de la couche.
* call : Applique l'opération sur l'entrée.

Exemple pour la regression linéaire :

from tensorflow.keras.layers import Layer

class CustomLinear(Layer):

def \_\_init\_\_(self, units):

super(CustomLinear, self).\_\_init\_\_()

self.units = units

def build(self, input\_shape):

self.w = self.add\_weight(

shape = (input\_shape[-1], self.units),

initializer = "random\_normal",

trainable = True

)

self.b = self.add\_weight(

shape = (self.units,),

initializer = "random\_normal",

trainable = True

)

def call(self, inputs):

return tf.matmul(inputs, self.w) + self.b

Une fois la couche définie, l'appel de l'instance va automatiquement initialiser les poids de la couche :

# Définition d'une couche CustomLinear

linear = CustomLinear()

# Initialisation des poids ainsi que transforme l'entrée data

linear(data)

Nous allons maintenant créer une couche représentant une régression polynomiale d'ordre 2 contenant des poids w et un biais b, cette couche va transformer les entrées en leur appliquant une combinaison linéaire avec les poids.

* Importer le module tensorflow sous le nom tf
* Importer la classe Layer de tensorflow.keras.layers
* Créer une classe CustomPolynomial qui hérite de la classe Layer.
  + Dans le constructeur de cette classe \_\_init\_\_ avec comme argument self, units :
    - Initialiser les objets de la classe de base Layer.
    - Ajouter l'attribut units à la classe.
  + Dans la méthode build, initialiser deux attributs *w* et *b* à l'aide de la méthode add\_weight. Attention à bien multiplier la shape de w par deux (regression polynomiale d'ordre 2).
  + Dans l'appel call de la classe, concaténer l'entrée **inputs** avec **inputs^2**, puis appliquer une combinaison linéaire de ce résultat avec les poids w et qui rajoute un biais b.

def call(self, inputs):

# inputs\*\*2

inputs\_pow = tf.pow(inputs, 2)

# Concatenation inputs and inputs\*\*2

inputs\_process = tf.concat([inputs, inputs\_pow], axis=-1)

# Combinaison linéaire entre inputs\_process et w

return tf.matmul(inputs\_process, self.w) + self.b

import tensorflow as tf

from tensorflow.keras.layers import Layer

class CustomPolynomial(Layer):

def \_\_init\_\_(self, units):

super(CustomPolynomial, self).\_\_init\_\_()

self.units = units

def build(self, input\_shape):

self.w = self.add\_weight(

shape = (input\_shape[-1]\*2, self.units),

initializer = "random\_normal",

trainable = True

)

self.b = self.add\_weight(

shape = (self.units,),

initializer = "random\_normal",

trainable = True

)

def call(self, inputs):

inputs\_pow = tf.pow(inputs, 2)

inputs\_process = tf.concat([inputs, inputs\_pow], axis=-1)

return tf.matmul(inputs\_process, self.w) + self.b

* Instancier un objet customlayer de la classe CustomPolynomial à 8 unités
* Créer un tenseur inputs de dimensions (4,5) contenant que des 1.
* Appliquer la transformation de la couche customlayer à inputs.
* Afficher le résultat.

customlayer = CustomPolynomial(8)

inputs = tf.ones((4,5))

output = customlayer(inputs)

print(output)

tf.Tensor(

[[ 0.22367771 0.06185782 0.13460459 0.12191121 0.28009367 -0.00336447

0.14273143 0.20041849]

[ 0.22367771 0.06185782 0.13460459 0.12191121 0.28009367 -0.00336447

0.14273143 0.20041849]

[ 0.22367771 0.06185782 0.13460459 0.12191121 0.28009367 -0.00336447

0.14273143 0.20041849]

[ 0.22367771 0.06185782 0.13460459 0.12191121 0.28009367 -0.00336447

0.14273143 0.20041849]], shape=(4, 8), dtype=float32)

La classe Layer permet de stocker les poids de chaque couche dans les **attributs** weights, trainable\_weights et non\_trainable\_weights :

* weights : L'ensemble des poids de la couche.
* trainable\_weights : Seulement les poids entrainables
* non\_trainable\_weights : Seulement les poids non entrainables
* Afficher le nombre de poids :
  + de la couche customlayer.
  + entrainable de la couche customlayer.
  + non entrainable de la couche customlayer .

print("number of weights: ", len(customlayer.weights))

print("number of trainable weights: ", len(customlayer.trainable\_weights))

print("number of non trainable weights", len(customlayer.non\_trainable\_weights))

number of weights: 2

number of trainable weights: 2

number of non trainable weights 0

L'un des paramètres importants de la classe Layer est training, il permet de différencier le comportement du modèle en cas d'entraînement ou d'inférence (prédiction). Cette notion est notamment important dans les couches de Dropout et de BatchNormalization.

* Créer une couche personnalisée CustomDropout qui initialise un attribut rate.
* Dans l'appel call de la classe avec comme argument inputs et training = None :
  + S'il y a **entrainement**, retourne un dropout des entrées à l'aide de la fonction dropout de tf.nn (en paramètre les entrées et le taux rate).
  + S'il n'y a **pas d'entrainement**, retourne les entrées.

class CustomDropout(Layer):

def \_\_init\_\_(self, rate):

super(CustomDropout, self).\_\_init\_\_()

self.rate = rate

def call(self, inputs, training = None):

if training:

return tf.nn.dropout(inputs, rate = self.rate)

return inputs

## La classe Model

Grace à la classe Layer, il est très facile de créer des couches personnalisées compatibles avec le constructeur Sequential. Il est également possible de créer, de la même façon, un modèle personnalisé en faisant hériter cette fois la classe Model.

L'initialisation du modèle permet de définir toutes les couches nécessaires en attributs. L'appel du modèle, quant à lui, permet de passer nos entrées à travers les différentes couches du modèle :

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Softmax

class CustomModel(Model):

def \_\_init\_\_(self, units\_1, units\_2, n\_classes):

# Initialisation de la classe Model.

super(CustomModel, self).\_\_init\_\_()

# Définition de chaque couche de notre modèle.

self.linear\_1 = CustomLinear(units\_1)

self.dropout\_1 = CustomDropout(rate=0.2)

self.linear\_2 = CustomLinear(units\_2)

self.dropout\_2 = CustomDropout(rate=0.2)

self.linear\_3 = CustomLinear(n\_classes)

self.softmax = Softmax()

def call(self, inputs):

# Appliquer l'opération de chaque couche.

x = self.linear\_1(inputs)

x = tf.nn.relu(x)

x = self.dropout\_1(x)

x = self.linear\_2(x)

x = tf.nn.relu(x)

x = self.dropout\_2(x)

x = self.linear\_3(x)

return self.softmax(x)

L'avantage d'utiliser la classe Model par rapport à la classe Layer :

* Possède les méthodes suivantes : fit, evaluate, predict.
* Contient ses propres couches internes, accessibles à l'aide de model.layers.
* Permet de sauvegarder et sérialiser : save\_weights, load\_weights.

Ici, nous pouvons nous contenter du constructeur Sequential :

* Définir un modèle par couche vierge **model** à l'aide du constructeur Sequential.
* Ajouter une couche CustomPolynomial avec 32 units.
* Ajouter une couche CustomDropout avec 0.2 comme rate.
* Ajouter une couche d'activation ReLU de tensorflow.keras.layers.
* Ajouter une couche CustomPolynomial avec 64 units.
* Ajouter une couche CustomDropout avec 0.2 comme rate.
* Ajouter une couche d'activation ReLU de tensorflow.keras.layers.
* Ajouter une couche CustomPolynomial avec 1 unit.
* Ajouter une couche d'activation ReLU de tensorflow.keras.layers.

from tensorflow.keras.layers import ReLU

model = tf.keras.Sequential()

model.add(CustomPolynomial(32))

model.add(CustomDropout(0.2))

model.add(ReLU())

model.add(CustomPolynomial(64))

model.add(CustomDropout(0.2))

model.add(ReLU())

model.add(CustomPolynomial(1))

model.add(ReLU())

* Afficher le résultat que retourne le modèle pour un vecteur de 1 de taille [10,5].

model(tf.ones([10,5]))

<tf.Tensor: shape=(10, 1), dtype=float32, numpy=

array([[0.],

[0.],

[0.],

[0.],

[0.],

[0.],

[0.],

[0.],

[0.],

[0.]], dtype=float32)>

Nous allons maintenant traiter un problème de régresion avec notre modèle personnalisé fraichement créé.

* Charger le jeu de données *"airfoil\_data.csv"* dans le dataframe **df**.

import pandas as pd

df = pd.read\_csv('airfoil\_data.csv')

df.head()

|  | **Frequency** | **AngleOfAttack** | **ChordLength** | **FSVelocity** | **SuctionThickness** | **PressureLevel** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 800 | 0.0 | 0.3048 | 71.3 | 0.002663 | 126.201 |
| **1** | 1000 | 0.0 | 0.3048 | 71.3 | 0.002663 | 125.201 |
| **2** | 1250 | 0.0 | 0.3048 | 71.3 | 0.002663 | 125.951 |
| **3** | 1600 | 0.0 | 0.3048 | 71.3 | 0.002663 | 127.591 |
| **4** | 2000 | 0.0 | 0.3048 | 71.3 | 0.002663 | 127.461 |

* Séparer les données en features et target sachant que la variable cible est "PressureLevel".
* Séparer les données en un ensemble d'entrainement (**X\_train**, **y\_train**) et en un ensemble de test (**X\_test**, **y\_test**). On peut prendre une proportion de 20% pour l'ensemble de test.

from sklearn.model\_selection import train\_test\_split

features = df.drop(['PressureLevel'] , axis=1)

target = df.PressureLevel

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, target, test\_size = 0.2)

* Exécuter la cellule suivante pour normaliser nos données.

from sklearn.preprocessing import StandardScaler

X\_scaler = StandardScaler().fit(X\_train)

y\_train = y\_train.ravel().reshape(-1, 1)

y\_test = y\_test.ravel().reshape(-1, 1)

y\_scaler = StandardScaler().fit(y\_train)

# Features

X\_train = X\_scaler.transform(X\_train)

X\_test = X\_scaler.transform(X\_test)

# Target

y\_train = y\_scaler.transform(y\_train)

y\_test = y\_scaler.transform(y\_test)

* Compiler notre modèle **model** avec un optimizer Adam, une fonction de perte MeanSquaredError et une métrique MeanAbsoluteError.

model.compile(optimizer=tf.keras.optimizers.Adam(),

loss=tf.keras.losses.MeanSquaredError(),

metrics=[tf.keras.metrics.MeanAbsoluteError()])

* Entraîner le modèle sur X\_train et y\_train.

history = model.fit(X\_train, y\_train, batch\_size = 32 , epochs = 100 , validation\_data=(X\_test, y\_test))

## Ce qu'il faut retenir

Il est ainsi possible de créer des couches et des modèles personnalisés tout en profitant de l'API de keras qui fournit des méthodes utiles à la modélisation et l'entrainement.

Beaucoup de modèle nécessite d'utilise des couches personnalisées ou une structure bien particulière. C'est notamment le cas pour les modèles célèbres suivants:

* Seq2seq : modèle sequence-to-sequence
* Transformer : modèle de BERT, modèle CamemBERT, XLNet ...
* WaveNet : modèle basé sur des convolutions 1D.
* Spatial Transformer Networks : Remplacer la couche de pooling (<https://kevinzakka.github.io/2017/01/18/stn-part2/>)

Il est aussi possible d'accéder à d'avantages de couches/modèles prédéfinies sur [TensorFlow Hub](https://www.tensorflow.org/hub?hl=fr).

# VII. Callbacks

Dans l'exercice précédent, nous avons vu comment personnaliser une couche, un modèle pour la rendre compatible avec l'API de Keras. Nous allons maintenant nous intéresser à un outil bien pratique : les **callbacks**.

Les rappels (*callbacks*) sont des outils qui vous permettront de personnaliser vos entraînements et évaluations.

Il permet alors de connaître l'état interne d'un modèle, de le sauvegarder, d'afficher des statistiques intéressantes et même de changer des hyperparamètres pendant les étapes de l'entraînement.

Pour illustrer le fonctionnement des **callbacks**, nous allons utiliser le jeu de données : airfoil\_data.csv.

* Charger les données de airfoil\_data.csv dans df.

import pandas as pd

df = pd.read\_csv('airfoil\_data.csv')

df.head()

| **Frequency** | **AngleOfAttack** | **ChordLength** | **FSVelocity** | **SuctionThickness** | **PressureLevel** |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 800 | 0.0 | 0.3048 | 71.3 | 0.002663 | 126.201 |
| **1** | 1000 | 0.0 | 0.3048 | 71.3 | 0.002663 | 125.201 |
| **2** | 1250 | 0.0 | 0.3048 | 71.3 | 0.002663 | 125.951 |
| **3** | 1600 | 0.0 | 0.3048 | 71.3 | 0.002663 | 127.591 |
| **4** | 2000 | 0.0 | 0.3048 | 71.3 | 0.002663 | 127.461 |

* Exécuter la cellule suivante pour mettre en forme et normaliser les données.

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

features = df.drop(['PressureLevel'], axis = 1)

target = df.PressureLevel

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, target, test\_size = 0.2)

y\_train = y\_train.ravel().reshape(-1, 1)

y\_test = y\_test.ravel().reshape(-1, 1)

X\_scaler = StandardScaler().fit(X\_train)

y\_scaler = StandardScaler().fit(y\_train)

X\_train = X\_scaler.transform(X\_train)

X\_test = X\_scaler.transform(X\_test)

y\_train = y\_scaler.transform(y\_train)

y\_test = y\_scaler.transform(y\_test)

## Modèle de regression

* Créer model une instance de Sequential
* Ajouter au modèle une couche dense à 10 unités, dont la fonction d'activation est la tangente hyperbolique tanh en précisant l'input shape (X\_train.shape[1],)
* Ajouter au modèle une couche dense de sortie à 1 unité et dont la fonction d'activation est linéaire.
* Compiler le modèle avec l'optimiseur Adam, une MeanSquaredError comme perte et MeanAbsoluteError comme métrique.

import tensorflow as tf

from tensorflow.keras import Sequential

from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(10, activation = 'tanh', input\_shape = (X\_train.shape[1],)))

model.add(Dense(1, activation = 'linear'))

model.compile(optimizer=tf.keras.optimizers.Adam(),

loss=tf.keras.losses.MeanSquaredError(),

metrics=[tf.keras.metrics.MeanAbsoluteError()])

Les callbacks sont des objets de la classe callbacks.Callback de tensorflow.keras et vont agir comme *méthodes* à utiliser à certaines étapes de *l'entraînement ou évaluation*.

On peut les stocker dans une *liste* à passer comme *argument* à la méthode fit ou predict du modèle.

Nous verrons dans cet exercice plusieurs exemples de callbacks utiles, et à la fin nous créerons un callback personnalisé.

Le callback TerminateOnNan permet de *stopper* l'entraînement quand la fonction de perte a divergé (de valeur NaN). Ce callback peut être utile pour éviter de continuer à entraîner le modèle s'il a divergé ou d'arrêter le calcul en cas de données non conforme.

from tensorflow.keras import callbacks

TON = callbacks.TerminateOnNaN()

model.fit(X\_train, y\_train, batch\_size = 32, epochs = 10, callbacks = [TON])

* Créer TON une instance du callback TerminateOnNaN à l'aide de la commande callbacks.TerminateOnNaN().
* Entraîner le modèle sur 10 epochs, un batch\_size de 32, un validation\_split de 0.2 et en donnant à l'argument callbacks une liste contenant l'instance TON.

from tensorflow.keras import callbacks

TON = callbacks.TerminateOnNaN()

model.fit(X\_train, y\_train, batch\_size = 32, epochs = 10, validation\_split = 0.2, callbacks = [TON])

LearningRateScheduler est un callback qui permet de varier le taux d'apprentissage pendant l'entraînement.

Ce callback prend en argument une fonction qui met à jour le taux d'apprentissage selon l'époque.

* Définir une fonction decreasinglrUpdate qui prend en arguments l'époque et le taux d'apprentissage et qui divise le taux d'apprentissage chaque 3 époques.
* Créer lrScheduler une instance du callback LearningRateScheduler.
  + Préciser dans l'argument schedule la fonction qui met à jour le taux d'apprentissage.
  + Mettre 1 à l'argument verbose afin d'afficher le taux d'apprentissage durant l'entraînement.
* Entraîner le modèle sur 10 epochs, un batch\_size de 32, un validation\_split de 0.2 et en donnant à l'argument callbacks une liste contenant l'instance lrScheduler.

def decreasinglrUpdate(epoch,learning\_rate):

if epoch % 3 == 0:

return learning\_rate \* 0.1

else:

return learning\_rate

lrScheduler = callbacks.LearningRateScheduler(schedule = decreasinglrUpdate,

verbose = 1)

model.fit(X\_train, y\_train, batch\_size = 32, epochs = 10, validation\_split = 0.2, callbacks = [lrScheduler])

EarlyStopping est un callback très utilisé. Il permet de controler l'évolution des métriques en arrêtant l'entraînement quand ces dernières ne s'améliorent plus.

Ce callback est très utile pour réduire le surapprentissage d'un modèle.

* Créer early\_stopping une instance du callback EarlyStopping d'arguments:
  + monitor prend le nom de la métrique à controler, on prendra 'val\_loss' pour la perte de validation.
  + patience prend le nombre d'époques à attendre avant d'arreter l'entraînement, mettre ici 3.
  + mode permet de préciser si la métrique doit croître, décroitre. Mettre 'min' puisque la métrique est une perte à minimiser.
  + restore\_best\_weights est un booléen qui va prendre que les poids de la meilleure époque dans la métrique précisée. Mettre True.
* Entraîner le modèle sur 20 epochs, un batch\_size de 32, un validation\_split de 0.2 et en donnant à l'argument callbacks une liste contenant l'instance early\_stopping ainsi que lrScheduler.

early\_stopping = callbacks.EarlyStopping(monitor = 'val\_loss', patience = 3, mode = 'min', restore\_best\_weights = True)

model.fit(X\_train, y\_train, epochs = 20, batch\_size = 32, validation\_split = 0.2, callbacks = [early\_stopping, lrScheduler])

Un des callback importants à connaître est ModelCheckpoint, il permet de sauvegarder régulièrement un modèle pendant son entraînement. Ce qui peut être utile lors d'un long apprentissage.

* Stocker dans filepath le dossier de travail actuel qui fera office de lieu de sauvegarde du modèle.
* Créer checkpoint une instance du callback ModelCheckpoint d'arguments:
  + Mettre le chemin du dossier de sauvegarde dans l'argument filepath.
  + Mettre dans monitor la métrique à controler: ici la perte de validation val\_loss.
  + Assurez vous que le meilleur modèle ne soit pas écrasé et que le modèle sauvegarder le tout et non pas les poids seulement à l'aide des booléens save\_best\_only et save\_weights\_only.
  + Dans mode indiquer la décroissance de la métrique.
  + Dans save\_freq indiquer epoch pour que le modèle soit sauvegardé après chaque époque.
* Entraîner le modèle sur 5 epochs, un batch\_size de 32, un validation\_split de 0.2 et en donnant à l'argument callbacks une liste contenant l'instance early\_stopping ainsi que checkpoint.

import os

cwd = os.getcwd()

filepath = cwd

checkpoint = callbacks.ModelCheckpoint(filepath = filepath,

monitor = 'val\_loss',

save\_best\_only = True,

save\_weights\_only = False,

mode = 'min',

save\_freq = 'epoch')

model.fit(X\_train, y\_train, batch\_size = 32, epochs = 5, validation\_split = 0.2, callbacks = [early\_stopping, checkpoint])

Si on veut cette fois ci changer le taux d'apprentissage selon la métrique et non pas l'époque, on peut utiliser le callback ReduceLROnPlateau.

* Créer lr\_plateau une instance du callback ReduceLROnPlateau d'arguments:
  + monitor va controler la perte de validation.
  + patience le nombre d'époques à attendre avant d'arreter l'entraînement. Mettre 2.
  + verbose pour préciser combien d'informations seront affichées durant l'apprentissage, mettre 2.
  + Dans mode indiquer la décroissance de la métrique.
* Entraîner le modèle sur 50 epochs, un batch\_size de 32, un validation\_split de 0.2 et en donnant à l'argument callbacks une liste contenant l'instance early\_stopping ainsi que lr\_plateau.

lr\_plateau = callbacks.ReduceLROnPlateau(monitor = 'val\_loss',

patience=2,

verbose=2,

mode='min')

model.fit(X\_train, y\_train, batch\_size = 32, epochs = 50, validation\_split = 0.2, callbacks = [early\_stopping,lr\_plateau])

Il est possible d'utiliser TensorBoard qui est un meilleur outil pour visualiser le résumé de l'apprentissage de votre modèle.

Pour pouvoir l'utiliser dans vos projets, installez le dans votre environnement python. Puis la cellule suivante montre comment exécuter une instance de Tensorboard.

# %load\_ext tensorboard

# log\_dir = '/'

# tensorboard = callbacks.TensorBoard(log\_dir = log\_dir)

# model.fit(X\_train, y\_train, batch\_size = 32, epochs = 5, validation\_split = 0.2, callbacks = [tensorboard])

On remarque qu'il y a énormement de callback disponibles sous Keras.

Il est conseillé d'utiliser la *documentation* pour connaître les différentes possibilités qu'offre Keras.

Cependant, on peut vouloir créer des callbacks personnalisés. Dans ce cas on peut utiliser la classe Callback pour remplacer des *méthodes* qui sont appelées à differentes étapes de l'entraînement.

Ces méthodes sont nombreuses et prennent des arguments différents, par exemple la méthode on\_epoch\_end permet de réaliser une action spécifique à la fin d'une époque et prend en arguments l'époque epochet les résultats des métriques dans logs.

* A l'aide de la [documentation](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Callback): implémenter la classe CustomCallBack qui hérite de Callback:
  + Afficher le numéro du batch et la perte d'entraînement à la fin de l'entraînement d'un batch.
  + Afficher le numéro de l'époque, la perte d'évaluation et la métrique d'évalaution val\_mean\_absolute\_error à la fin d'une époque.
  + Afficher que la validation est finie à la fin de chaque test.
* Entraîner le modèle sur *10* epochs, un batch\_size de 64, un validation\_split de 0.2 et en donnant à l'argument callbacks une liste contenant uen instance de CustomCallBack. Préciser dans verbose la valeur 0 pour ne prendre en compte que les messages affichées par le callback (et non pas par la méthode fit).

class CustomCallBack(callbacks.Callback):

def on\_train\_batch\_end(self, batch, logs=None):

print('Pour le batch {}, erreur de train est {:7.2f}.'.format(batch, logs['loss']))

def on\_epoch\_end(self, epoch, logs=None):

print('La perte de validation pour époque {} est {:7.2f} and la métrique est {:7.2f}.'.format(epoch, logs['val\_loss'], logs['val\_mean\_absolute\_error']))

def on\_test\_end(self, logs=None):

print('Validation fini')

history = model.fit(X\_train, y\_train, batch\_size = 64, epochs = 10, validation\_split = 0.2, verbose = 0, callbacks = [CustomCallBack()])

## Ce qu'il faut retenir

Les rappels (*callbacks*) sont des outils qui permettent de contrôler l'entraînement et évaluation d'un modèle. Il est alors possible de connaître l'état interne d'un modèle, de le sauvegarder, d'afficher des statistiques intéressantes et même de changer des hyperparamètres pendant les étapes de l'entraînement.

Les callbacks suivants peuvent être très pratique en Deep Learning :

* Sauvegarder les meilleurs poids du modèle au cours de l'entraînement :

callbacks.ModelCheckpoint(filepath = filepath,

monitor = 'val\_loss',

save\_best\_only = True,

save\_weights\_only = False,

mode = 'min',

save\_freq = 'epoch')

* Réduire automatiquement le learning rate :

callbacks.ReduceLROnPlateau(monitor = 'val\_loss',

patience=5,

factor=0.5,

verbose=2,

mode='min')

* Arrêter l'entraînement si le modèle n'évolut plus. Très pratique pour ne pas gérer le nombre d'époque et laisser le modèle s'arrêter quand il n'évolut plus.

callbacks.EarlyStopping(monitor = 'val\_loss',

patience = 8,

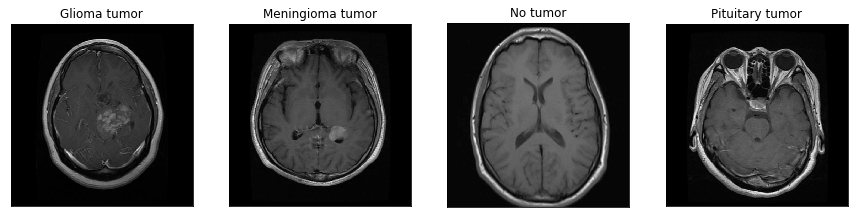
mode = 'min',

restore\_best\_weights = True)

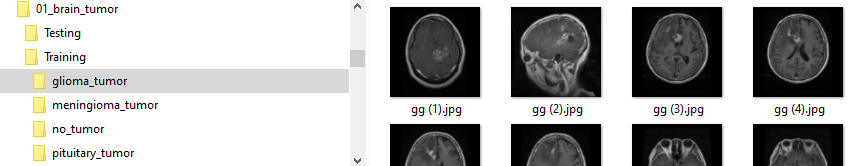
# VIII. Transfer Learning

Bienvenue dans le dernier exercice du module d'introduction à Tensorflow. Cet exercice a pour objectif de vous rappeler le fonctionnement du **transfer learning** et traiter toutes les étapes d'un problème de classification d'image.

Le jeu de données est constitué d'image de radiographie contenant quatre classes : **glioma tumor**, **meningioma tumor**, **no tumor**, **pituitary tumor**. Vous trouverez plus d'informations sur le jeu de données [ici](https://www.kaggle.com/sartajbhuvaji/brain-tumor-classification-mri).



Les images se trouvent dans le dossier Training et Testing. Et, une fois dans le dossier, les images de chaque classe se trouvent dans des dossiers séparés.



* Exécuter la cellule suivant permettant de créer un dataframe df contenant le chemin vers l'image et la classe correspondante.

import glob

import pandas as pd

# Trouver tous les chemins vers les fichiers qui finissent par .jpg

liste = glob.glob('./Training/\*/\*.jpg')

# Extraire le label de chaque image

liste = list(map(lambda x : [x, x.split('/')[2]], liste))

# Créer un dataframe pandas

df = pd.DataFrame(liste, columns=['filepath', 'nameLabel'])

df['label'] = df['nameLabel'].replace(df.nameLabel.unique(), [\*range(len(df.nameLabel.unique()))])

df.head()

| **filepath** | **nameLabel** | **label** |  |
| --- | --- | --- | --- |
| **0** | ./Training/meningioma\_tumor/m3 (67).jpg | meningioma\_tumor | 0 |
| **1** | ./Training/meningioma\_tumor/m1(122).jpg | meningioma\_tumor | 0 |
| **2** | ./Training/meningioma\_tumor/m2 (81).jpg | meningioma\_tumor | 0 |
| **3** | ./Training/meningioma\_tumor/m3 (77).jpg | meningioma\_tumor | 0 |
| **4** | ./Training/meningioma\_tumor/m (192).jpg | meningioma\_tumor | 0 |

## Exploration des données

Maintenant que la correspondance entre chemin et classe est faite, intéressons-nous maintenant aux images.

La fonction read\_file de tensorflow.io permet de charger l'information brute d'un fichier (image, fichier audio, dossier texte ...). Ensuite, la fonction decode\_jpeg de tensorflow.image permet de décoder l'information brute en un tensor de type uint8 ou uint16 :

# Charger l'information brute en mémoire

im = tf.io.read\_file(filepath)

# Decoder l'information en un tensorflow RGB (3 channels).

im = tf.image.decode\_jpeg(im, channels=3)

  On peut utiliser le package matplotlib ou OpenCV pour charger les images, mais, les fonctions de tensorflow sont plus performantes quand ils sont utilisés dans l'architecture du modèle.

* Charger une des images de notre dataframe.
* Afficher l'image à l'aide de la fonction imshow de matplotlib.pyplot.

import tensorflow as tf

import matplotlib.pyplot as plt

# file path of the image

filepath = df.filepath[0]

# Read the file

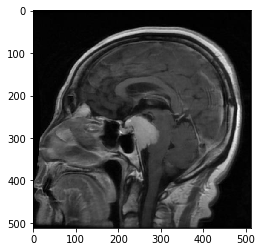
im = tf.io.read\_file(filepath)

# Decode the file

im = tf.image.decode\_jpeg(im, channels=3)

# Show the tensor

plt.imshow(im);



Les modèles en Deep learning n'accepte en générale que des entrées de même dimension, il est alors nécessaire de redimensionner toutes les images pour qu'elles aient la même taille. D'après la littérature, la dimension optimale entre perte d'information et complexité est aux alentours de (256,256).

La fonction resize de tensorfow permet de redimensionner une image en dimension size:

# Redimensionner l'image en (256,256)

im = tf.image.resize(im, size=(256,256))

* Redimensionner l'image chargée précédemment en (256,256).

# file path of the image

filepath = df.filepath[0]

# Read the file

im = tf.io.read\_file(filepath)

# Decode the file

im = tf.image.decode\_jpeg(im, channels=3)

# Resizing

tf.image.resize(im, size=(256,256))

## Charger le jeu de données

Le modèle a besoin pour s'entraîner d'images/labels. Ces images peuvent soit être tous chargées initialement ou soit être chargées quand le modèle en a besoin. Il arrive que le jeu de données soit constitué de centaines de millier d'images, dans ce cas, il n'est pas concevable de les charger au préalable en mémoire (nécessite de 40Go de RAM pour 150k images 256x256x3).

Dans ce cas, deux solutions sont disponibles :

* Comme le jeu de données est trop lourd, choisir un sous-échantillon (ex : 30 000 images) et le charger en mémoire.
* Charger les images pendant l'entraînement à l'aide d'un générateur. Solution à privilégier si le pré-processing n'est pas trop lourd.

Les deux solutions fonctionnent, dans cet exercice nous allons **choisir la deuxième**. Comme le jeu de test est généralement plus petit, il peut être chargé en mémoire.

  La structure des données nous permettrait d'utiliser les instances ImageDataGenerator de **tensorflow.keras.preprocessing.image** pour charger les données quand le modèle en a besoin. Mais, cette instance reste très limitée, convient généralement que pour des problèmes de classification.

* Séparer le jeu de données df.filepath et la variable cible df.label en un ensemble d'entraînement **X\_train\_path**, **y\_train**, et en un ensemble de test **X\_test\_path**, **y\_test**. Nous choisirons un rapport de 80% pour les données d'entraînements et une graine aléatoire 1234.
* Charger les images de **X\_test\_path** redimensionnées à [256,256,3] en mémoire dans la variable **X\_test**.

from sklearn.model\_selection import train\_test\_split

from tqdm import tqdm

import numpy as np

X\_train\_path, X\_test\_path, y\_train, y\_test = train\_test\_split(df.filepath, df.label, test\_size = 0.2, random\_state = 1234)

X\_test = []

for filepath in tqdm(X\_test\_path):

# Read the file

im = tf.io.read\_file(filepath)

# Decode the file

im = tf.image.decode\_jpeg(im, channels=3)

# Resizing

im = tf.image.resize(im, size=(256,256))

X\_test.append([im])

X\_test = tf.concat(X\_test, axis=0)

Maintenant les données de test chargées, il est nécessaire de définir un **dataset** permettant charger les images à chaque itération du modèle. Pour optimiser le temps de chargement/pré-traitement, il est possible de paralléliser chaque opération en utilisant une structure multi-tasking.

Les objet de type **dataset** sur tensorflow sont capables de le faire à l'aide de l'argument **num\_parallel\_calls** de la méthode map.

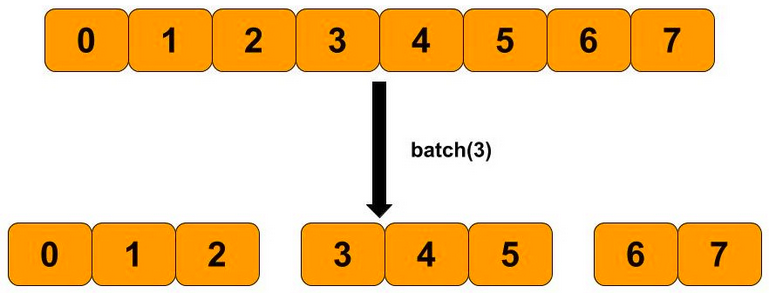
Pour rappel, le constructeur from\_tensor\_slices du package **tensorflow.data.Dataset** permet de convertir une liste d'array en un dataset.

dataset = tf.data.Dataset.from\_tensor\_slices((X\_path, y))

La méthode map du dataset permet d'appliquer une opération à chaque observation. Exemple :

dataset = dataset.map(lambda x, y : [load\_image(x), y])

La méthode batch du dataset permet de regrouper les observations sous forme de batch.



dataset = dataset.batch(batch\_size)

* Définir une fonction load\_image avec comme argument filepath retournant une image redimensionnée en (256,256).
* Définir un dataset dataset\_train de (X\_train\_path, y\_train) à l'aide de la fonction from\_tensor\_slices.
* À l'aide de la méthode map, appliquer la fonction load\_image à chaque valeur de **X\_train\_path**. Pour que le chargement s'effectue en multi-tasking, préciser l'argument num\_parallel\_calls égale à -1.
* Regrouper les observations sous forme de batch de taille 32.

@tf.function

def load\_image(filepath, resize=(256,256)):

im = tf.io.read\_file(filepath)

im = tf.image.decode\_png(im, channels=3)

return tf.image.resize(im, resize)

dataset\_train = tf.data.Dataset.from\_tensor\_slices((X\_train\_path, y\_train))

dataset\_train = dataset\_train.map(lambda x, y : [load\_image(x), y], num\_parallel\_calls=-1).batch(32)

Les modèles de classification d'image ou de détection d'objet utilise généralement une approche par transfer Learning.

### Quelques rappels sur le transfer leaning :

L'apprentissage par transfert est le phénomène par lequel un apprentissage nouveau est facilité grâce aux apprentissages antérieurs partageant des similitudes. Par exemple, les connaissances acquises lors de l’apprentissage de la reconnaissance des voitures peuvent s’appliquer lorsqu’on essaie de reconnaître des camions.

Les modèles existants (VGG, ResNet, ...) sont composés de deux grandes parties. La première appelée **backbone** est un ensemble de convolution permettant l'**extraction des features de l'image**. La seconde est une succession de dense layer qui a pour but de classifier.

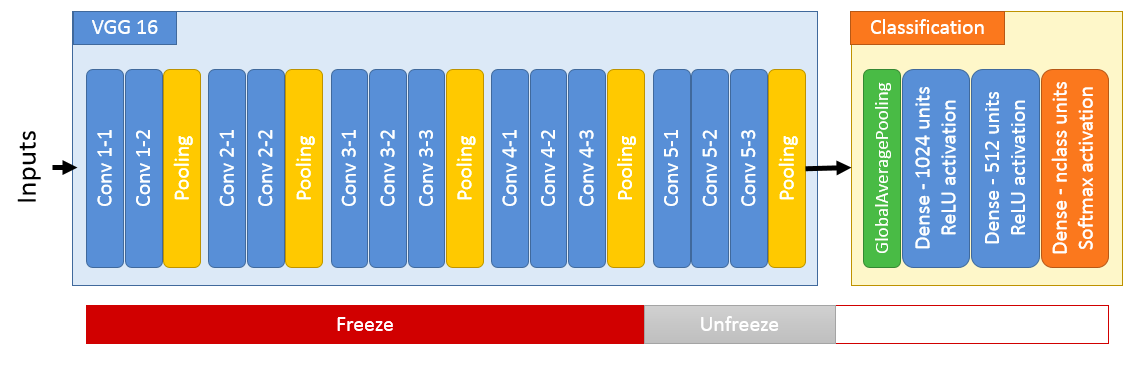
Les données du nouveau problème doit être assez semblable avec le jeu de données utilisé pour le pré-entrainement. Dans ce cas, la méthode de transfer learning consiste à utiliser le **backbone** d'un modèle pré-entraîné comme extraction de features. Ensuite des couches Dense sont ajouté pour traiter le problème de classification ou de regression.

%matplotlib inline

from interaction\_tl import show\_tl

show\_tl()

Lors du début de l'apprentissage, il est nécessaire de "freezer" (bloquer) les poids de la partie pré-entrainée (**backbone**) puisqu'ils sont proches des poids optimaux. Puis, au cours de l'entraînement, on peut "unfreeze" les couches pour affiner les poids du modèle : cet opération est appelée le **fine-tuning**.

Dans cet exercice, le modèle pré-entraîné sera le EfficientNet puisqu'il a montré de très bon résultat et de très bonne propriété pour le transfer learning.

Voici un exemple pour charger et freeze les poids d'une modèle pré-entraîné :

vgg16 = VGG16(include\_top=False, input\_shape=(256,256,3))

for layer in vgg16.layers:

layer.trainable = False

model = Sequential()

model.add(vgg16)

* Charger le modèle EfficientNetB1 de tensorflow.keras.applications sous le nom efficientNet. La partie classification ne sera pas prise et l'input\_shape sera (256,256,3).
* Freezer les poids du modèle.
* Afficher le résumé du modèle.

from tensorflow.keras.applications import EfficientNetB1

# Load the model efficientNet

efficientNet = EfficientNetB1(include\_top=False, input\_shape=(256,256,3))

# Freeze the blackbone

for layer in efficientNet.layers:

layer.trainable = False

efficientNet.summary()

### Partie Classification

* Ajouter le modèle pré-entraîné à un objet Sequential qui portera le nom de **model**.
* Ajouter à ce modèle une couche GlobalAveragePooling2D.
* Puis, ajouter quelques couches Dense et Dropout.
* Finir par une couche Dense avec 4 neurones et une fonction activation 'softmax'.
* Afficher le résumé du modèle.

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

efficientnetb1 (Functional) (None, 8, 8, 1280) 6575239

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

global\_average\_pooling2d (Gl (None, 1280) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 1024) 1311744

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 1024) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 512) 524800

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 512) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 4) 2052

=================================================================

Total params: 8,413,835

Trainable params: 1,838,596

Non-trainable params: 6,575,239

* Compiler le modèle avec votre fonction de perte sparse\_categorical\_crossentropy, un optimizer 'adam' et une métrique ['accuracy'].

model.compile('adam', 'sparse\_categorical\_crossentropy', metrics=['accuracy'])

## Entraînement du modèle[¶](https://train.datascientest.com/hub/laurentlabordef22@gmail.com/notebooks/223/674#Entra%C3%AEnement-du-mod%C3%A8le)

* Entraîner le modèle à l'aide la méthode fit sur 10 epochs en ajoutant les callbacks suivants :
  + Ajouter une sauvegarde des poids à chaque epoch à l'aide du callbacks ModelCheckpoint.
  + Diminuer le learning rate si la métrique de validation ne s'améliore pas dans les 5 dernières époques.

from tensorflow.keras import callbacks

# Save automatically the weights

checkpoint = callbacks.ModelCheckpoint(filepath = 'checkpoint',

monitor = 'val\_loss',

save\_best\_only = True,

save\_weights\_only = False,

mode = 'min',

save\_freq = 'epoch')

# Callback to reduce automatically the learning rate.

lr\_plateau = callbacks.ReduceLROnPlateau(monitor = 'val\_loss',

patience=5,

factor=0.1,

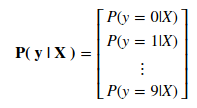
verbose=2,

mode='min')

model.fit(dataset\_train, epochs=10, validation\_data=(X\_test, y\_test), callbacks = [lr\_plateau, checkpoint])

## Évaluation

La fonction d'activation **softmax** utilisée à la fin de notre modèle, fait que celui-ci retourne pour chaque image une probabilité de prédiction de chaque classe.



Pour prédire la classe de l'image, il suffit alors de trouver pour quelle classe la probabilité est maximale :



* Prédire la probabilité des classes du jeu de données **X\_test**.
* Prédire dans **y\_pred** la classe la plus probable à l'aide de la fonction argmax de tensorflow en précisant axis=-1.

# Probability prediction

y\_prob = model.predict(X\_test, batch\_size=64)

# Label prediction

y\_pred = tf.argmax(y\_prob, axis=-1).numpy()

* Afficher le score de précision à l'aide de la fonction accuracy\_score de sklearn.metrics.
* Afficher la matrice de confusion à l'aide de la fonction confusion\_matrix de sklearn.metrics.

from sklearn.metrics import accuracy\_score

from sklearn.metrics import confusion\_matrix

print('Accuracy :', accuracy\_score(y\_test, y\_pred))

confusion\_matrix(y\_test, y\_pred)

Accuracy : 0.8885017421602788

array([[166, 3, 2, 6],

[ 8, 146, 0, 3],

[ 33, 4, 127, 2],

[ 3, 0, 0, 71]])

* Exécuter la cellule suivante pour afficher les prédictions de notre modèle sur 3 images.

indices\_random = tf.random.uniform([3], 0, len(X\_test), dtype=tf.int32)

plt.figure(figsize=(15,7))

for i, idx in enumerate(indices\_random):

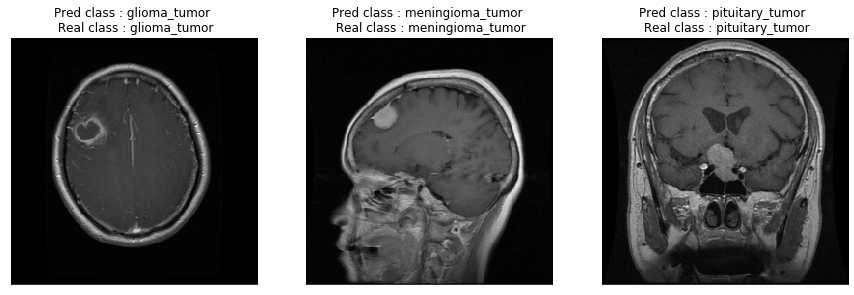
plt.subplot(1,3,i+1)

plt.imshow(tf.cast(X\_test[idx], tf.int32))

plt.xticks([])

plt.yticks([])

plt.title('Pred class : {} \n Real class : {}'.format(df.nameLabel.unique()[y\_pred[idx]], df.nameLabel.unique()[y\_test.values[idx]]))



## Ce qu'il faut retenir de ce module

## Méthodologie

Globalement, la méthodologie pour résoudre un problème à l'aide d'outil de deep learning :

1. Définir un **dataset** permettant de mettre en forme les données et de partitionner en batchs.
2. Construire un modèle : MLP, CNN, RNN, transfer learning ...
3. Compiler le modèle : définition d'une fonction de perte, métrique, optimizer.
4. Entraîner le modèle, il y'a deux manières équivalentes de le faire:
   * Méthode fit: problème simple.
   * Calculer le gradient de la fonction de coût puis rétropropager l'erreur: problème complexe.
5. Prédiction et évaluation du modèle.

## Dataset

Tensorflow possède un sous ensemble tensorflow.data.dataset qui permet d'appliquer toutes les étapes de pré-processing sur les données. C'est un pipeline d'opération appliqué sur les entrées. Un exemple :

* Charger les images + redimensionner (en paralélisant les calculs).
* Appliquer des méthodes d'augmentation de données.
* Normaliser nos données.
* Les regrouper en lot de données.

## Keras

La version de tensorflow 2.0+ a été construite autour du framework [**keras**](https://www.tensorflow.org/guide/keras).

Vous pouvez retrouver toutes les fonctionnalités de **keras** dans tensorflow.keras. Il y'a notamment :

* Les couches de neurones dans tensorflow.keras.layers. Il est possible de créer des couches personnalisées en faisant hériter la classe de la couche avec tensorflow.keras.layers.Layer.
* Les modèles pré-entrainés dans tensorflow.keras.applications.
* Les fonctions de pertes dans tensorflow.keras.losses
* Les métriques dans tensorflow.keras.metrics.
* Les optimizers dans tensorflow.keras.optimizers.

## Callbacks

Les rappels (*callbacks*) sont des outils qui permettent de contrôler l'entraînement et évaluation d'un modèle. Il est alors possible de connaître l'état interne d'un modèle, de le sauvegarder, d'afficher des statistiques intéressantes et même de changer des hyperparamètres pendant les étapes de l'entraînement.

Les callbacks suivants peuvent être très pratique en Deep Learning :

* Sauvegarder les meilleurs poids du modèle au cours de l'entraînement :

callbacks.ModelCheckpoint(filepath = filepath,

monitor = 'val\_loss',

save\_best\_only = True,

save\_weights\_only = False,

mode = 'min',

save\_freq = 'epoch')

* Réduire automatiquement le learning rate :

callbacks.ReduceLROnPlateau(monitor = 'val\_loss',

patience=5,

factor=0.5,

verbose=2,

mode='min')

* Arrêter l'entraînement si le modèle n'évolut plus. Très pratique pour ne pas gérer le nombre d'époque et laisser le modèle s'arrêter quand il n'évolut plus.

callbacks.EarlyStopping(monitor = 'val\_loss',

patience = 8,

mode = 'min',

restore\_best\_weights = True)