MUSEO TOOLBOX

ATELIER PROGRAMMATION & RÉSEAUX DE NEURONES

Encadrants : Yousra Hamrouni, Nicolas Karasiak et Claude Monteil
Préparé par : Anne-Sophie Tronc Laures, Sam Antonetti,
Arthur Duflos, Hélène Ternisien de Boiville
21 février 2020
Université Toulouse Jean Jaures & ENSAT

Résumé

Créé il y a plus de 20 ans, le Master 2 SIGMA a pour objectif de former des spécialistes de géomatique appliquée aux problématiques de l'aménagement des espaces. Il comprend également une forte composante de programmation et d'algorithmie. Chaque année, les étudiants, répartis en groupes d'atelier, doivent répondre à des commandes provenants d'organismes publiques, d'entreprises privées ou de laboratoires de recherches. En Février 2020, quatre étudiants du master ont travaillé avec les doctorants du laboratoire DYNAFOR en s'appuyant des travaux déjà réalisés sur l'utilisation du langage python en télédétection.

Cet atelier avait deux objectifs. Le premier était de proposer d'implémenter les filtres spatiaux et les traitements avec plus processeurs à la bibliothèque python Museo ToolBox développée par Nicolas Karasiak. Le second objectif était de mettre en place une méthode de Deep Learning (réseaux de neurones) pour la prédiction à partir d'images satellites des peupliers du sud-ouest de la France. L'ambition était de contribuer aux travaux de thèse de Yousra Hamrouni. A l'issue de cet atelier, les étudiants ont pu proposer des améliorations pour Museo ToolBox qui ont été revues et intégrées par Nicolas Karasiak dans les branches develop et spatial_filter de son répertoire GitHub (https://github.com/nkarasiak/MuseoToolBox/tree/develop)
, (https://github.com/nkarasiak/MuseoToolBox/tree/spatial_filter).

Les étudiants ont également pu développer une chaine de traitement de Deep Learning qui est disponible sur le répertoire Github de l'atelier (https://github.com/HTDBD/SIGMA_neural_network). Le code permet à partir d'une image et de polygones d'entrainer, de valider et de prédire un modèle issu de TensorFlow.

Au cours des 6 semaines d'ateliers, les étudiants ont pu entrainer des réseaux de neurones sur différents jeux de données (Cartes historiques de la Finlande et orthophotos de l'IGN). Cependant, selon le modèle et le jeu de données utilisés, les résultats obtenus sont hétérogènes.

Table des matières

TABLE DES FIGURES

INTRODUCTION

A chaque promotion, les étudiants de SIGMA participent à des ateliers professionnels au cours desquels ils doivent répondre à une commande. Cette année l'un de ces ateliers avait pour objectif d'explorer les possibilités qu'offre le langage python dans le traitement d'images satellites.

Le laboratoire Dynafor accueille deux doctorants, Yousra Hamrouni et Nicolas Karasiak, dont les travaux de recherches se concentrent sur l'utilisation des images satellitaires pour l'étude des espaces forestiers. La thèse de Nicolas Karasiak porte sur la télédétection hypertemporelle des essences forestières tandis que la thèse de Yousra Hamrouni porte plus particulièrement sur la télédétection hypertemporelle des peupliers.
Cet atelier s'intègre à leurs recherches et s'appuient sur les travaux déjà réalisés par ces deux doctorants.

Au cours de sa thèse Nicolas Karasiak a développé une bibliothèque python spécialisée dans le traitement d'images, Museo Toolbox. Plus compréhensible et moins opaque que GDAL, elle a permis aux étudiants de SIGMA d'appréhender rapidement les bases de la télédétection. La première partie de cet atelier lui est consacrée. Publiée sur la plateforme GitHub, le code source de cette bibliothèque est en open source et il est libre à chacun de proposer des améliorations, des évolutions ainsi que des critiques. Pour cet atelier Nicolas Karasiak avait identifié deux pistes d'améliorations à la portée des étudiants. La première consistait à créer une fonction spatiale (partie II) qui est essentielle pour les traitements d'images. La seconde était en relation avec la première. Les fonctions spatiales étant généralement très chronophages, il fallait trouver un moyen d'améliorer les temps de calculs de la bibliothèque. De ce fait, la seconde piste se concentrait sur le multiprocessing ou l'utilisation des plusieurs cœurs pour un traitement.

En France depuis 2009, la présence de peupliers a drastiquement baissé suite à l'évolution de différents facteurs économiques. L'objectif de la thèse de Yousra Hamrouni est d'étudier à l'aide d'images satellites l'évolution de la population des peupliers dans le sud-ouest de la France. Sur ces images, les peupliers sont reconnaissables par la forme de leur implantation et de leur canopée. Une telle singularité amène à penser que de nouvelles méthodes d'intelligence artificielle seraient capables d'apprendre et de reconnaitre les peupliers sur des images satellites. La seconde partie de l'atelier a donc eu pour objectif d'étudier l'utilisation de méthodes de Deep Learning tels que les réseaux de neurones pour prédire la présence de peupliers dans le sud-ouest de la France.

L'ensemble des commandes et attentes de cet atelier nécessitait d'établir des canaux de communications et des outils de partage dédiés à l'écriture de scripts python. Tout au long de cet atelier, les encadrants et les étudiants ont pu dialoguer par le biais de l'application Discord. Les étudiants ont également appris à utiliser le protocole git. Git Hub est un plateforme web qui permet à chacun de créer son propre répertoire et d'y ajouter des codes ou des logiciels, de le versionner, le partager et l'améliorer.

Motivés et curieux d'en apprendre plus sur la création de bibliothèque sous python et de réseaux de neurones, nous, quatre étudiants de la promotions 2019-2020 de SIGMA, vous proposons sous la forme de ce rapport les résultats et les échecs rencontrés lors de cet atelier.

Partie I : Museo Toolbox

A) MULTIPROCESSING, PLUSIEURS COEURS POUR MTB

1) Cahier des charges

Le traitement d'images requiert des ressources et des temps qui peuvent être conséquents, en effet, de lourdes images ne peuvent être totalement chargées en mémoire et parcourir le raster pixel par pixel est extrêmement chronophage. La bibliothèque MuseoToolbox permet de diminuer le temps de calcul en utilisant des outils optimisés tels que la bibliothèque Numpy et le fonctionnement par block permet de diminuer la taille de la mémoire nécessaire.

Par défaut, python et le logiciel par lequel il est utilisé n'utilisent qu'un processeur. Cependant, chaque ordinateur dispose d'au moins 4 processeurs. Avoir plusieurs processeurs permet à un ordinateur de lancer plusieurs tâches simultanément. Dans cette partie de l'atelier, le but était d'ajouter une méthode de parallélisme (ou multiprocessing) à MuseoToolbox afin d'accélérer encore le temps d'exécution. Il s'agit pour un calcul d'utiliser plusieurs processeurs. Il nous a été conseillé de nous tourner vers la bibliothèque Joblib qui est utilisée par de grandes bibliothèques comme Scikit-learn.

2) Répartition des tâches et avancement

Nous avons défini des tâches et points clefs qui nous ont permis d'avancer rapidement dans cette première partie d'atelier.

Tout comme l'autre groupe dédié à la fonction spatiale, notre parcours s'est divisé en 5 tâches principales.

1 Phase de recherches

2 Phase d'écriture du code

3 Phase d'écriture des commentaires

4 Phase de présentation

5 Phase d'intégration dans GitHub

Figure 1 : Diagramme de Gantt de la partie MuseoToolbox

Une fois la phase de recherche terminée, nous nous sommes concentrés sur la construction de MuseoToolbox et plus particulièrement de la classe RasterMath. Comprendre ce code nous a demandé un certain temps bien que nous ayons déjà eu l'occasion de travailler avec cette bibliothèque, nous n'étions pas entrés dans le code source. Nous avons accompli cette étape individuellement, en nous regroupant en cas d'incompréhension, et si nous n'arrivions toujours pas à saisir une particularité, nous pouvions demander à Nicolas Karasiak.

Sur la majeure partie de la programmation, nous avons travaillé sur un même ordinateur, le deuxième ordinateur étant utilisé pour des recherches annexes, avec parfois des échanges de fragments de code via message privé sur Discord. Nous avons également travaillé de notre côté sur certaines pistes comme ça a été le cas pour la gestion des masques des blocks, ou l'optimisation de la liste intermédiaire de blocks.

3) Nouvelle fonctions et mise à jour

- ✓ **Get_block_coords**

```
def get_block_coords(self, block_number=0):
    """
    Get position of a block :

        order as [x,y,width,height]

    Parameters
    ----------
    block_number, int, optional (default=0).
        Position of the desired block.

    Returns
    --------
    List of positions of the block [x,y,width,height]

    """
    if block_number > self.n_blocks:
        raise ValueError(
            'There are only {} blocks in your image.'.format(
                self.n_blocks))
    else:

        row = [l for l in range(0, self.n_lines, self.y_block_size)]
        col = [c for c in range(0, self.n_columns, self.x_block_size)]
```

```
        row_number = int(block_number / self.n_x_blocks)
        col_number = int(block_number % self.n_x_blocks)

        width = min(self.n_columns - col[col_number], self.x_block_size)
        height = min(self.n_lines - row[row_number], self.y_block_size)

        tmp = self._generate_block_array(
            col[col_number], row[row_number], width, height, self.mask)

        if self.return_3d is False:
            tmp = self._manage_2d_mask(tmp)
            tmp = np.ma.copy(tmp)

        return [col[col_number], row[row_number], width, height]
```

Cette fonction est une adaptation de la fonction *get_block* qui était déjà présente dans MuseoToolbox. Afin de réécrire les blocs à la bonne position dans le raster, nous avions besoin de la position (i, j) du premier pixel du bloc, ainsi que du nombre de lignes et de colonnes qui le composent. La fonction renvoie donc une liste contenant ces informations que nous pourrons récupérer lors de l'écriture.

✓ write_block & write_few_blocks

```
def _write_few_blocks (self, idx_blocks, tab_blocks, idx_func):
    """
    Uses _write_block to write some blocks from a list to a raster

    Parameters
    ----------
    idx_blocks : list.
        List of indexes of all blocks
    tab_blocks : list.
        List of values or tab that will be written in the output image
    idx_func : int
        function's index

    Returns
    -------
    None.

    """
    for i in range  (len(idx_blocks)):
        self._write_block( idx_blocks[i] ,tab_blocks[i] , idx_func)


def _write_block(self, idx_block, tab_block, idx_func):
    """
    Write a block at a position on a output image

    Parameters
    ----------
    idx_block : int.
        List of indexes of all blocks
    tab_blocks : numpy tab.
        List of values or tab that will be written in the output image
    idx_func : int
        function's index

    Returns
    -------
    None.
```

```
"""
maxBands = self.outputs[idx_func].RasterCount
for ind in range(maxBands):
    # write result band per band
    indGdal = ind + 1
    curBand = self.outputs[idx_func].GetRasterBand(indGdal)

    resToWrite = tab_block[..., ind]
    coords = self.get_block_coords(idx_block)
    if self.return_3d is False:
        # need to reshape as block
        tmparr = resToWrite.reshape(coords[3], coords[2])
        resToWrite = self.reshape_ndim(tmparr)

    curBand.WriteArray(resToWrite,coords[0],coords[1])
    curBand.FlushCache()
```

Dans nos premières versions de la méthode Run (procédure principale d'écriture d'un raster de RasterMath), nous n'avions pas prévu de fonction séparée pour écrire les blocks une fois calculés dans le raster de sortie. Afin de clarifier le code, nous avons créé la procédure *_write_block* qui prend en entrée un block (résultat de la fonction définie par l'utilisateur) et l'écrit au bon endroit dans un fichier Tif par exemple. La sortie est définie en même temps que la fonction par l'utilisateur. La bonne position du block est déterminée directement à partir de son indice grâce à la fonction *get_block_coords*.
La procédure *_write_few_block* est pensée pour écrire une liste de block car cela correspond au fonctionnement du run, elle réitère la méthode *_write_block* sur chaque élément de la liste.

✓ run_Parallel & run_Parallel_short

Afin de pouvoir gérer le multiprocessing nous avons recréé une procédure similaire au run, en commençant par n'ajouter que les fonctionnalités de base, c'est-à-dire lire un raster, utiliser des fonctions dessus et l'écrire dans un raster de sortie. Nous n'avons pas traité les masques immédiatement. Joblib propose une fonction *delayed* qui s'applique à un objet de classe *Parallel*, cette fonction permet de diviser une liste entre les différents processeurs afin que chacun d'entre eux exécute une fonction sur les éléments de cette liste. Les éléments se répartissent entre chaque processeur et sont ensuite disponibles dans une liste avec le même index qu'en entrée. Le principal problème rencontré a été de nous passer de la programmation objet, Joblib ne pouvant prendre de fonction utilisant un self en entrée. Nous avons donc contourné le problème en créant une liste de blocs intermédiaires qui pourrait être parcourue par la fonction de Joblib, la longueur de la liste étant déterminé en fonction du nombre de processeurs à utiliser.

Nous avons testé différentes fonctions pour vérifier quelle partie du code était la plus chronophage. Nous avons pu observer que la création de toutes ces listes nous faisait perdre un certain temps. La méthode *run_Parallel_short* a permis de mettre en évidence ce temps perdu, elle chargeait tous les blocks en mémoire et les écrivait tous en une fois. Mais avec de lourds rasters ce fonctionnement pouvait poser des problèmes.

Afin de diminuer le temps d'exécution sans pour autant surcharger la mémoire de l'ordinateur, l'utilisateur pourra choisir une taille maximale qu'il autorise, le code calcule le nombre de blocs que l'on peut charger avant de dépasser

cette limite et cela détermine la longueur de la liste intermédiaire. De plus, l'ensemble des blocs n'est plus chargé mais seuls les indices de ces blocs sont générés dans une liste.

```
def run_parallel (self, n_jobs = 1, size = '1G'):
        """
        Function under construction and used for tests
        Apply a function of idx [0] to an image with parallel mode using joblib.
        Warning : this function does not write the results in an output.

        Parameters
        ----------
        n_jobs : int, optional, ( default value : 1)
            Numbers of workers or process that will work in parallel.
        size : str, optional ( default value : '1G')
            maximun size of ram the program can use to store temporary the results

        Returns
        -------
        None
        """
        #create a list of all the blocks
        size_value=size[:-1]
        if size[-1]=='M':
            size_value=1048576*int(size_value)
        elif size[-1]=='G':
            size_value=1073741824*int(size_value)
        elif size[-1]=='T':
            size_value=1099511627776*int(size_value)
        else :
            raise ValueError(' {} is not a valid value. use for example 100G, 10M, 1T '.format(size))
        tmp=self.get_random_block()
        length=int(size_value/tmp.size*tmp.itemsize)

        if length>self.n_blocks:
            length=self.n_blocks


        if n_jobs < 0 :
            n_jobs = cpu_count()
        elif n_jobs == 0 :
            raise ValueError(' {} is not a valid value.'.ormat(n_jobs))

        self.pb = ProgressBar(self.n_blocks, message=self.message)

        for i in range(0,self.n_blocks,length):
            idx_masked=[]
            idx_blocks=[]
            if i <=self.n_blocks-length :
                for j in range(i,i+length):
                    if not self.get_block(j).size ==0:
                        idx_blocks+=[j]
                    else:
                        idx_masked+=[j]
            else :
                for j in range(i,self.n_blocks):
                    if not self.get_block(j).size ==0:
                        idx_blocks+=[j]
                    else:
                        idx_masked+=[j]

            for idx,fun in enumerate(self.functions):
                self.pb.add_position(self._position)

                #fun get the first function of the rastermath object
```

```
                fun = self.functions[idx]
                if self.functionsKwargs[idx] is not False:
                    res=Parallel(n_jobs, verbose = self.verbose)(delayed(fun)(self.get_block(i).data,**self.functionsKwargs[idx])for i in
idx_blocks)
                else :
                    res=Parallel(n_jobs,verbose = self.verbose)(delayed(fun)(self.get_block(i).data)for i in  idx_blocks)

                self._write_few_blocks(idx_blocks , res ,idx)
                if len(idx_masked)!=0:
                    self._write_few_blocks(idx_masked,[self.outputNoData[idx] for i in idx_masked] ,idx)
                self._position += 1


        for idx in range(len(self.outputs)):
            self.outputs[idx] = None
        self.pb.add_position(self.n_blocks)
```

4) Points bloquants et difficultés

- ✓ **Joblib, une bibliothèque peu transparente et Rapidité de run par MTB**

Joblib est une bibliothèque de multiprocessing, utilisée par des librairies telles que Scikit-learn. La documentation de Joblib est peu fournie si l'on compare à la documentation de Multiprocessing, et le fait de n'utiliser qu'une seule fonction de la classe *Parallel*, ne permet pas forcément d'avoir un code très souple. Lors de nos premiers essais avec la méthode *Run_Parallel*, nos temps d'exécution étaient bien supérieurs à ceux du run de MTB, environ 20 à 50 fois plus longs. Ci-dessous, quelques exemples de temps d'exécutions pour 3 fonctions chronophages :

```
def fun(X):
    for l in range (X.shape[0]):
        X[l,...]
    return X


def fun2(X):
    return X/2

def fun3 (X):
    for j  in range (10):
        for l in range (X.shape[0]):
            X[l,...]
        for l in range (X.shape[0]):
            X[l,...]
    return X
```

TEST AVEC 1 processeur

temps run parallel sans boucle blocks : **102.07369089126587**

temps run parallel avec boucle blocks : **100.61493611335754**

temps run MTB : **2.7261879444122314**

TEST 4 processeurs

T1

temps run parallel sans boucle blocks : **29.215853929519653**

temps run parallel avec boucle blocks : **41.26543593406677**

temps run MTB : **2.645871877670288**

T2

temps run parallel hélène sans boucle blocks : **30.587169647216797**

temps run parallel arthur avec boucle blocks : **39.223149061203**

temps run MTB : **2.60693097114563**

Nous avons perdu un certain temps à essayer de comprendre d'où cela provenait. Nous nous sommes demandés pourquoi cette bibliothèque était autant utilisée si nous n'arrivions pas à avoir des résultats satisfaisants et justifiant l'utilisation de plusieurs cœurs. Nous avons découvert par hasard en utilisant uniquement les données sans le masque d'un tableau Numpy que Joblib n'était pas optimisé pour lire la dimension mask des tableaux Numpy. En ajoutant le .data dans le process, nous avons obtenu une vitesse satisfaisante (similaire au run originel avec un seul processeur).

- ✓ **Multiprocessing**

Lors de notre phase de recherche, de nombreux utilisateurs avaient recours à la bibliothèque Multiprocessing, une bibliothèque un peu plus transparente que Joblib. Le choix dans les méthodes de multiprocessing sont plus nombreux, et semblent également plus complexes, avec entre autres la possibilité de passer par des Pool. Les différents petits tests effectués et les résultats de certains utilisateurs montrent un réel intérêt pour le multiprocessing et laissaient supposer de tels résultats pour son application des MuseoToolbox. Cependant sous Windows l'utilisation de multiprocessing nécessite que la condition suivante soit vérifiée : `if __name__ == '__main__'`

Or dans le cas d'une implémentation de multiprocessing dans une bibliothèque, cette condition n'est pas vérifiée et le multiprocessing n'est pas lancé. Nous n'avons pas trouvé d'exemple d'utilisation de cette bibliothèque dans une autre librairie. Afin de ne pas perdre de temps, nous avons confirmé notre choix de la bibliothèque Joblib.

- ✓ **La magie de GDAL**

Une fois que notre code s'exécutait sans erreur, nous avons vérifié la qualité des résultats. Lorsque nous demandions plusieurs fonctions à appliquer au raster, seulement la dernière s'écrivait, en visualisant les images sur QGIS les premières avait toutes une unique valeur, 0, mais avaient les bonnes dimensions, et la bonne position. Parfois, aucune ne s'écrivait et lorsque nous redémarrions le noyau la dernière s'écrivait. Après différentes recherches et l'aide de Nicolas Karasiak, il est apparu que GDAL n'écrivait que le dernier raster. Il s'agissait donc de trouver la bonne place pour le *FlushCache*.

- ✓ **La gestion des masques**

Au terme de la partie consacrée à MuseoToolbox et au multiprocessing, nous n'avons pas réussi à intégrer la gestion des masques. Comme nous avons procédé par étape, nous nous sommes d'abord concentrés sur l'ajout des traitements en parallèle sur des blocks entiers. Une fois cette fonctionnalité implémentée dans le code, l'ajout du traitement des

masques demandait de modifier le fonctionnement de notre méthode principale. La gestion des masques représentait également une difficulté pour le second groupe en charge de la fonction spatiale. De ce fait et afin de ne pas déborder sur la partie Deep Learning de l'atelier, Nicolas Karasiak a terminé l'optimisation des codes et l'intégration de notre *run Parallel* sur le Github de MuseoToolbox.

5) Habitudes de codes et astuces

 # le débogage et les variables courtes

Afin de faciliter le débogage nous avons privilégié des variables courtes, l'auto-complétion n'étant pas disponible dans la console de débogage, des variables de 1 ou 2 caractères permettent de réduire le risque d'erreur de frappe.

 # décortiquer son code

La création de fonctions ou procédures indépendantes permet d'alléger le code principal et facilite sa compréhension à l'instar de la procédure *write_block*.

 # bien connaitre ses boucles

Nous avons dû créer une boucle qui parcourait les blocs avec un certain pas (pas correspondant au début au nombre de processeurs souhaité, puis dépendant de la taille de la mémoire allouée), et il a fallu faire plusieurs tests simples et certains à la main pour comprendre qu'il y avait une erreur dans la boucle. Ci-dessous, la boucle sans erreur avec en plus la création d'une liste des indices de blocks totalement masqués :

```
if i <=self.n_blocks-length :
    for j in range(i,i+length):
        if not self.get_block(j).size ==0:
            idx_blocks+=[j]
        else:
            idx_masked+=[j]
else :
    for j in range(i,self.n_blocks):
        if not self.get_block(j).size ==0:
            idx_blocks+=[j]
        else:
            idx_masked+=[j]
```

6) Conclusion

Cette partie de l'atelier nous a permis de faire un premier pas vers les traitements réalisés avec le multiprocessing. Avec l'ajout du multiprocessing, le temps de traitement de fonctions lourdes sur 4 cœurs est divisé par 2 par rapport à un traitement sur un seul cœur. Nous avons pu aborder des notions tels que les Pool et la mémoire partagée qui n'ont pas été utilisés mais pourront être approfondi si nous devons à nouveau travailler sur le multiprocessing à l'avenir. Travailler

sur une bibliothèque déjà bien développée, telle que MuseoToolbox, a représenté un défi intéressant puisqu'il a fallu se l'approprier et comprendre son fonctionnement. Certaines caractéristiques des bibliothèques utilisées dans le traitement ont pu nous laisser perplexes et nous faire perdre une ou plusieurs journées.

B) FILTRES SPATIAUX

1) Cahier des charges

L'objectif est d'incorporer des fonctionnalités à la classe RasterMath de Museo ToolBox pour permettre à l'utilisateur d'appliquer des fonctions spatiales issues de son propre code sur des rasters, par exemple des filtres spatiaux.
Le principe d'une fonction spatiale est de calculer la valeur d'un pixel selon la valeur des pixels voisins.
RasterMath est une classe qui permet de faciliter et améliorer les traitements d'images raster pour l'utilisateur. Comme vu précédemment, les méthodes de RasterMath permettent de créer un ou plusieurs blocs de pixels de l'image et d'appliquer un ensemble de fonctions sur ces blocs.
Par défaut, dans RasterMath, les blocs seront en deux dimensions, c'est-à-dire que les pixels des différentes bandes sont alignés dans un tableau à une dimension, mais il existe la possibilité d'appliquer une fonction et de retourner des résultats en trois dimensions, soit garder les bandes comme troisième dimension. Ce point est très important car pour pouvoir appliquer une fonction « filtre spatial », il faudra « forcer » le mode en trois dimensions.

2) Répartition des tâches et avancement

Les étapes suivantes ont été suivies :

- Recherche sur le filtre spatial : Ces recherches nous ont permis de nous rendre compte qu'il existait une très grande diversité de filtres spatiaux et qu'une difficulté était la gestion des bords (de l'image ou du bloc dans notre cas).
- Ecriture / Adaptation du code à partir du fichier __init__.py de RasterMath.
- Intégration dans Github via GitKraken.
- Vérification / modification du code suite à l'intégration des fonctions de multi-processing.
- Rédaction du rapport sur la partie filtre spatial.

3) Nouvelles fonctions et mise à jour

Pour que les méthodes de RasterMath puissent fonctionner sur des fonctions spatiales, des fonctions et/ou procédures existantes ont dues être modifiées et des nouvelles ont été créées.
Auparavant, pour appliquer une fonction à une image, RasterMath extrayait les blocs avec la dimension par défaut (256x256) ou selon ce que l'utilisateur avait rentré avec la procédure *custom_block_size*. Le traitement de fonctions

spatiales, prenant en compte les pixels et leurs voisins, a nécessité d'agrandir les blocs par rapport à ce qui était entré pour pouvoir appliquer les fonctions sur tous les pixels d'un bloc. Les blocs agrandis comprennent ainsi la totalité du bloc originel plus une partie des blocs adjacents pour permettre aux calculs sur les pixels en bordure de bloc de s'effectuer correctement. L'utilisateur choisi un décalage (offset) qui permet de fixer la taille de l'agrandissement de ces blocs.

Les blocs en bordure d'image sont traités différemment. Quel que soit l'offset défini par l'utilisateur, le bloc ne s'agrandit pas en dehors de l'image puisqu'il ne peut y avoir de prise en compte de pixels inexistants. Il a donc fallu prendre en compte les 9 positions possibles des blocs dans l'image.

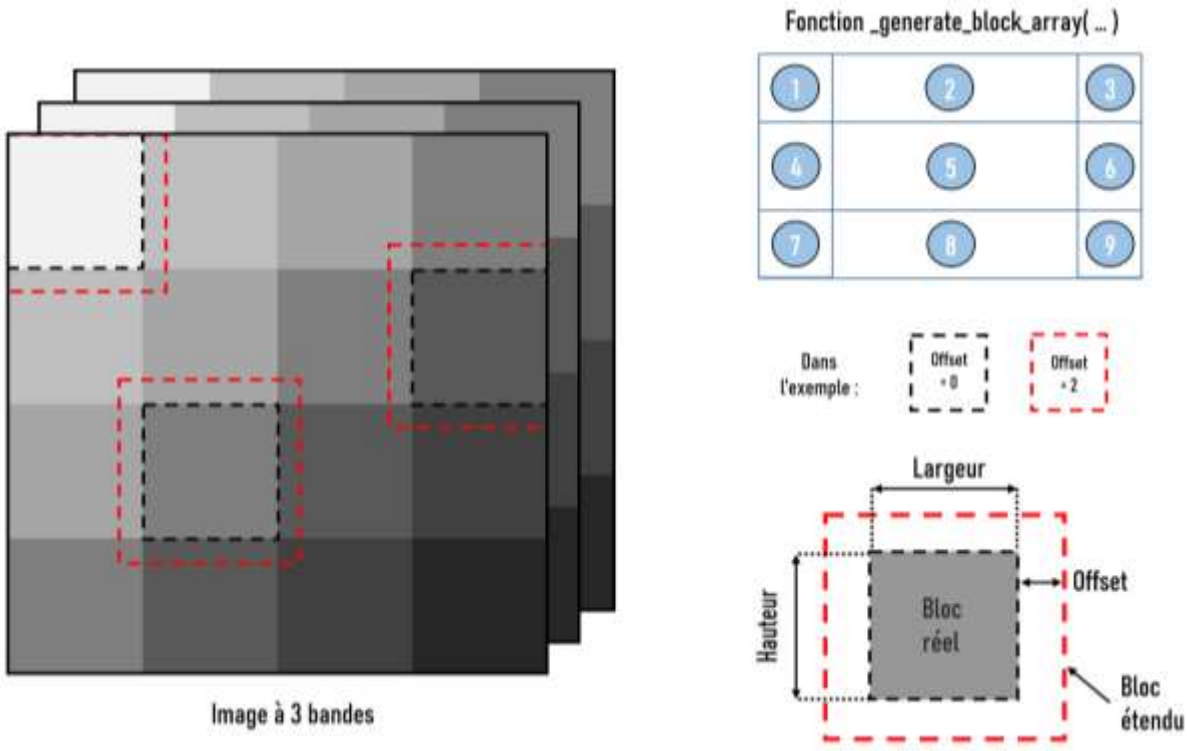

Figure 2 - Fonctionnement de la fonction _generate_block_array dans la fonction get_block, c'est-à-dire appliqué à la génération des blocs étendus d'une image pour l'application d'une fonction spatiale

La fonction modifiée pour appliquer ce principe est la fonction privée *_generate_block_array* qui permet de générer la matrice d'un bloc d'une image à partir de sa position et de ses dimensions. Quelques autres fonctions comme *get_block* et *get_random_block* ont été adaptées en modifiant l'appel et les arguments d'entrée en conséquence pour pouvoir fonctionner avec ces modifications.

Après avoir permis la création d'un bloc agrandi, il a fallu étudier la possibilité d'ajouter une fonction spatiale dans RasterMath. La fonction *add_function* permettait à la fois de faire les vérifications sur la fonction entrée par l'utilisateur (image en sortie, nombre de bandes, type de données, caractère spatial …) et de l'ajouter à un objet RasterMath. Une fonction *_check_add_fonction* a été créée spécifiquement pour faire les vérifications. Un nouvel attribut *_function_is_3d* de type booléen qui indique si la fonction à traiter est une fonction spatiale ou non a été ajouté pour faire les vérifications nécessaires selon qu'elles s'appliquent à une fonction spatiale ou non.

Pour intégrer la possibilité d'utiliser des fonctions spatiales, il a fallu créer une nouvelle fonction nommée *_add_spatial_function.* La différence avec *add_function* est l'utilisation d'un nouvel attribut de RasterMath, *_offsets*, qui correspond à la liste des offsets (décalages), et la vérification de « spatial=True » dans *_check_add_function*.

La fonction qui permet de lancer les calculs, la fonction run, a également été modifiée pour s'adapter aux changements. La fonction *_iter_for_spatial_fonction* renvoie un objet de type générateur qui génère donc les blocs spatiaux des pixels des blocs de l'image. C'est-à-dire il parcourt les pixels des blocs de l'image un par un et renvoie les blocs composés du pixel et de ses voisins dont le nombre varie en fonction de sa position dans l'image et de l'offset.

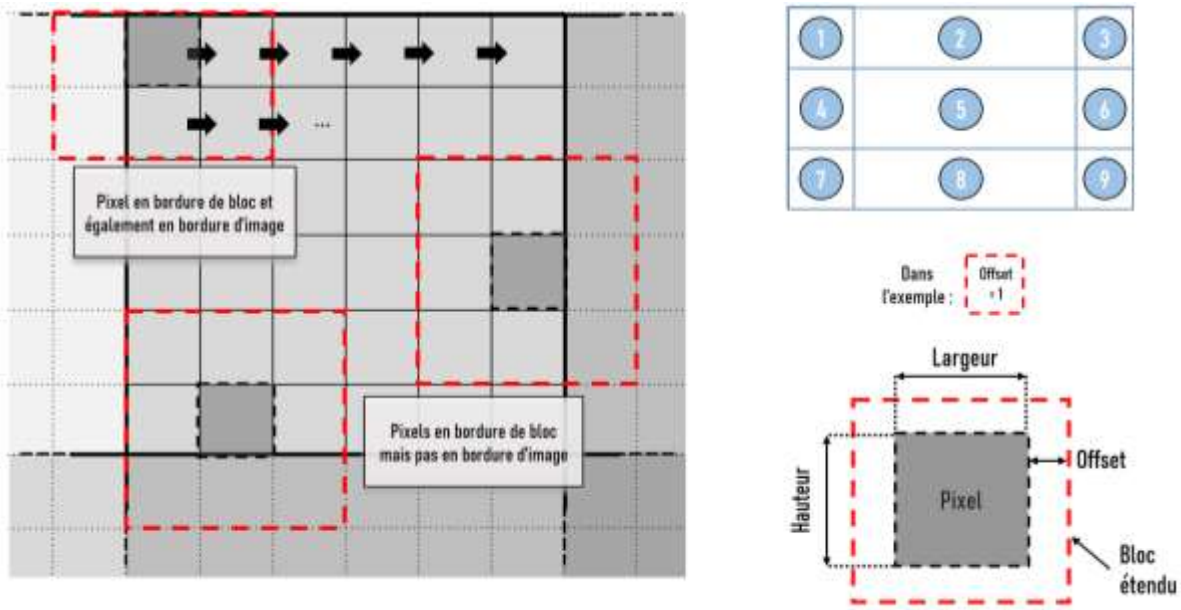

Figure 3 - Fonctionnement de la fonction _generate_block_array dans la fonction _iter_for_spatial_function, c'est-à-dire appliqué à la génération des blocs spatiaux (pixel et ses voisins) dans un bloc de l'image.

4) Points bloquants et difficultés

Les points recensés dans le tableau suivant sont les difficultés principales rencontrées lors de la manipulation de RasterMath :

POINTS GENERAUX	
Comprendre la structure des méthodes de RasterMath.	
SAM	ANNE-SOPHIE
Comprendre le format des données en entrée et en sortie en fonction du nombre de dimension des matrices Comprendre les imbrications et les utilisations des fonctions et procédures au sein de la classe. Faire en sorte de s'imprégner du code préexistant et savoir l'utiliser pour créer les nouvelles fonctionnalités Optimiser le code pour faire en sorte de traiter un maximum de cas possibles sans répéter les lignes de code.	Savoir réutiliser les méthodes de RasterMath, notamment comprendre le caractère généraliste de la classe : application à n'importe quelle fonction choisie par l'utilisateur. Visualiser la différence entre fonction spatiale et non spatiale : Facilité par la création de schémas. Dans le code, repérer tout ce qui doit être modifier pour pouvoir rentrer une fonction spatiale. Savoir retranscrire pour chaque fonction le passage de la 2D à la 3D.

5) Habitudes de codes et astuces

La conduite de cette partie du projet a permis de prendre des habitudes de codage, à apprendre à utiliser des fonctionnalités de python et à manipuler de nouveaux outils :

- ✓ **Programmation Python :**
  - Découverte et apprentissage du mode « Débogage » permettant de faciliter la recherche de l'emplacement des bugs et leur résolution.
  - Approfondissement de la connaissance et de l'usage des bibliothèques numpy et gdal.
  - Découverte de fonctionnalités de mise en page et de raccourcis utiles pour faciliter le codage en python.

- ✓ **Commentaires / Doc string :**
  Définition des paramètres selon la convention Numpy docstyle.

✓ **GitHub :**

- Découverte et manipulation de l'intégration continue à l'aide de Travis.
- Apprentissage de l'utilisation du logiciel GitKraken pour Github.

6) Conclusion

Le code intègre la gestion des filtres spatiaux et est fonctionnel. Les masques sont gérés et n'empêche pas RasterMath de faire les calculs sur les images avec les fonctions spatiales définies par l'utilisateur, mais il semblerait après quelques tests qu'il faille retravailler un peu dessus pour ne prendre en compte que les pixels non masqués dans les calculs de la fonction.
Notre groupe n'a également pas eu le temps de mixer la partie concernant la gestion des filtres spatiaux avec la partie sur le multi-processing dans RasterMath.
Différentes pistes d'améliorations du script sont envisageables :

- Affichage d'un message d'erreur quand l'utilisateur choisit add_spatial_function avec un offset=0 car cela ne représente aucun intérêt et cela ralentit le traitement.
- Plus généralement, affichage de messages d'information ou d'erreurs personnalisés lors du traitement spatial d'images.
- Optimisation et amélioration de l'organisation du code pour faciliter sa compréhension

Suite à ce travail, Nicolas Karasiak a modifié le code pour l'optimiser, le rendre compatible avec le travail en parallèle de RasterMath et l'intégrer dans la branche Spatial du Github de RasterMath.

C) CONCLUSION & CRITIQUE - PARTIE I

Dans la partie des filtres spatiaux comme la partie multiprocessing, cet atelier nous a permis d'approfondir des notions vues au cours de ce master. Au terme de ces semaines de travail, les deux branches de MuseoToolbox sont fonctionnelles. Cependant la commande n'a pas totalement été remplie, en effet nous n'avons pas parallélisé le traitement des filtres spatiaux. Le fait de nous être divisé en deux groupes avec des méthodes de programmation différentes a produit deux procédures run non homogènes. Les deux travaux ne sont pas compatibles. Un travail était nécessaire pour optimiser le code avant fusion. Nicolas Karasiak a donc fusionné les deux branches pendant que nous passions à la partie sur le Deep Learning.

Partie II : Réseau de neurones

B) QU'EST-CE QU'UN RÉSEAU DE NEURONES ?

1) Processus global

Un réseau de neurones artificiels est un système informatique dont la conception est inspirée du fonctionnement des neurones biologiques. Le Deep Learning fait partie intégrante des méthodes du Machine Learning (Figure 1). Ces réseaux, inclus dans les champs de recherche et d'action du Deep Learning, sont devenus prépondérants en raison de leurs hautes performances et de la possibilité de les appliquer à un grand nombre de domaines. Notamment, ces systèmes apprennent à effectuer des tâches telles que la classification ou la prédiction grâce à un processus itératif.

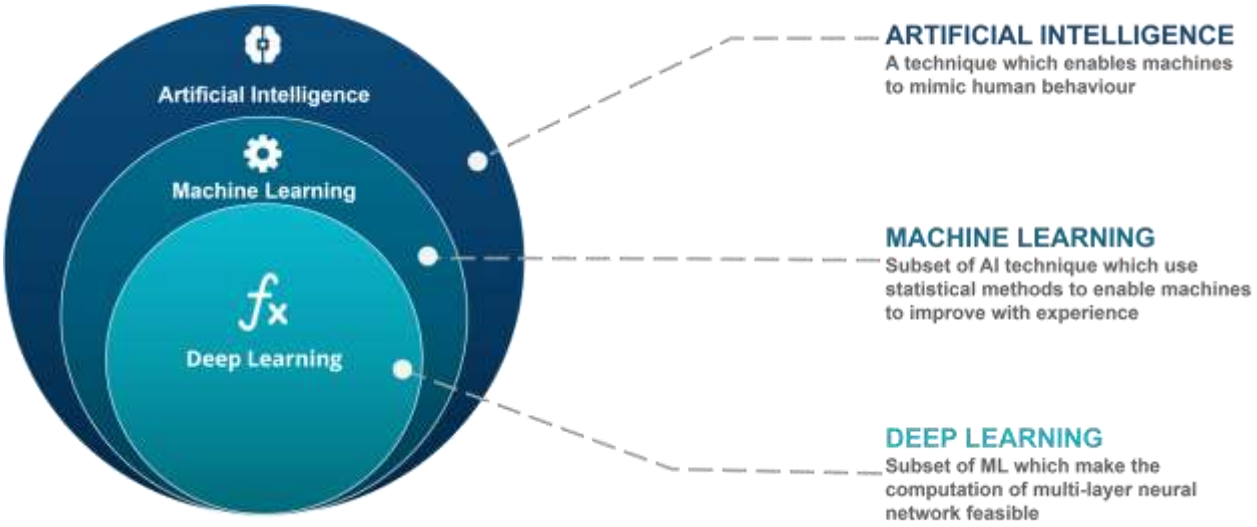

Figure 4 : Organisation des branches de l'intelligence artificielle

a) *Le neurone artificiel*

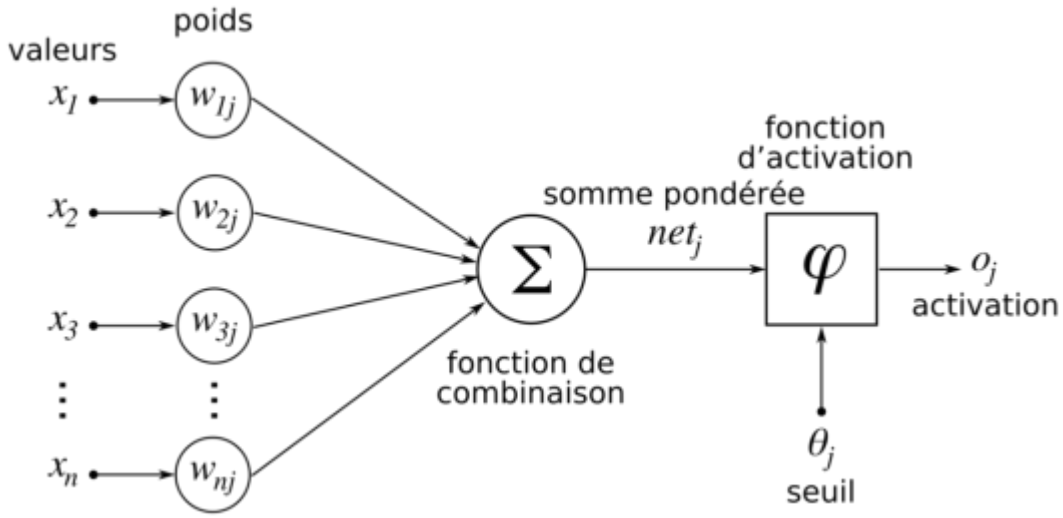

Figure 5 : Voix schématique d'un neurone artificiel avec un index j.[1]

En entrée, le neurone reçoit *n* variables venant de neurones en amont. Le neurone fait le calcul de la somme pondérée de ces variables en fonction de la force des connections entre neurones, appelées poids (w1, w2, w3…). Une fonction d'activation (ReLU, Sigmoid …) est appliquée à cette somme pondérée, donnant la valeur de l'état du neurone qui va être comparée à une valeur de seuil, pour donner la réponse en sortie, appelée niveau d'activation du neurone.
Les fonctions d'activation sont utilisées pour introduire une certaine non-linéarité dans le modèle ; elles déterminent la réaction du neurone en fonction de l'information reçue.

- Pour une classification binaire, la fonction sigmoïde permet de prédire la probabilité d'appartenir à la classe positive.
- Pour une classification multi-classes, il y aura autant d'unités de sortie que de classes à prédire. Il est alors possible d'utiliser la fonction *softmax*, qui est une généralisation de la fonction sigmoïde. Les valeurs de sortie sont ramenées entre 0 et 1 et leur somme est égale à 1.

b) *Le réseau de neurones multi-couches*

Les neurones sont organisés en couches. Les connexions ne se font qu'entre les neurones de deux couches différentes. En règle générale, pour chaque neurone de la couche, il existe un nombre de connexions égal au nombre de neurones de la couche suivante.
Il existe trois types de couches :

- La couche d'entrée reçoit toutes les informations nécessaires pour expliquer le phénomène ou le problème posé.

[1] Source : Par Chrislb — Made by Chrislb, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=674150

- Les couches intermédiaires ou couches cachées n'ont pas de contact avec l'extérieur. On parle de deep-learning à partir de deux couches « cachées ».
- La couche de sortie prédit une valeur qui est comparée à une valeur connue et une perte est calculée.

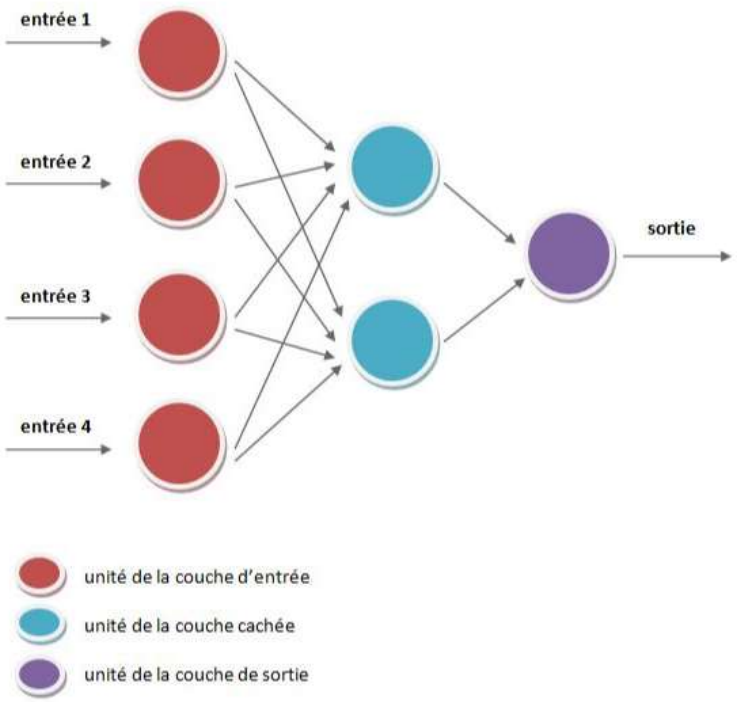

Figure 6 : Réseau de neurones en couches[2]

Pour mettre en place un réseau de neurones multi-couches, il faut au moins deux jeux de données :

- Des données pour l'apprentissage du modèle. Plus il y a de paramètres dans le modèle, plus il faudra de données pour apprendre les valeurs de ces paramètres, sans risquer le sur-apprentissage.
- Des données pour la validation.

En dernier lieu, un jeu de données test permet d'évaluer la capacité du modèle à généraliser sa prédiction.

[2] Source : Par Jimmy-neutron — Travail personnel, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=23224844

c) *Apprentissage d'un réseau de neurons*

L'apprentissage d'un réseau de neurones multi-couches consiste à trouver les bons poids des neurones afin d'avoir une prédiction satisfaisante en sortie.

Les réseaux à apprentissage supervisé[3] les plus utilisés sont les réseaux avec rétro-propagation du gradient.

Il s'agit de décomposer les erreurs à chaque couche selon le théorème de dérivation des fonctions composées. La mise à jour des poids se fait en alternant une phase montante (forward) et une phase descendante (backward). La mise à jour des poids se fait au moment de la phase descendante.

Le nombre de cycles d'apprentissage, appelés « epoch », fait partie du nombre de paramètres du modèle. Cela correspond au nombre de fois où l'ensemble des données d'apprentissage passent dans le modèle.

2) Réseaux de neurones convolutifs

Les réseaux de neurones convolutifs[4] sont les plus performants pour classer des images. En entrée, une image est fournie sous la forme d'une matrice de pixels.

Le réseau de neurone convolutif reçoit l'image et va appliquer deux types d'opérations :

- Des filtres ou noyaux de convolutions qui vont permettre d'extraire les caractéristiques des images. Une image est passée par cette succession de filtres pour créer une nouvelle image, appelée carte de convolution. Au fur et à mesure que l'on progresse dans le réseau, les couches de convolution vont détecter des entités de plus en plus complexes. Le Convolutional Neural Network accumule les détails de l'image pour obtenir une représentation globale.
- Des opérations de simplification (pooling) qui permettront d'alléger les calculs et de ressortir les principales opérations.

[3] Le réseau de neurones est entraîné sur des images avec les classes qui sont spécifiées au modèle.

[4] Convolutionnal Neural Network (CNN)

a) *La convolution*

Dans le machine learning, une convolution mélange le filtre convolutif et la matrice d'entrée pour entraîner les pondérations des valeurs des pixes. Le calcul de la convolution dépend de la taille du noyau (kernel) choisi. Un exemple de calcul est donné par la figure 3 :

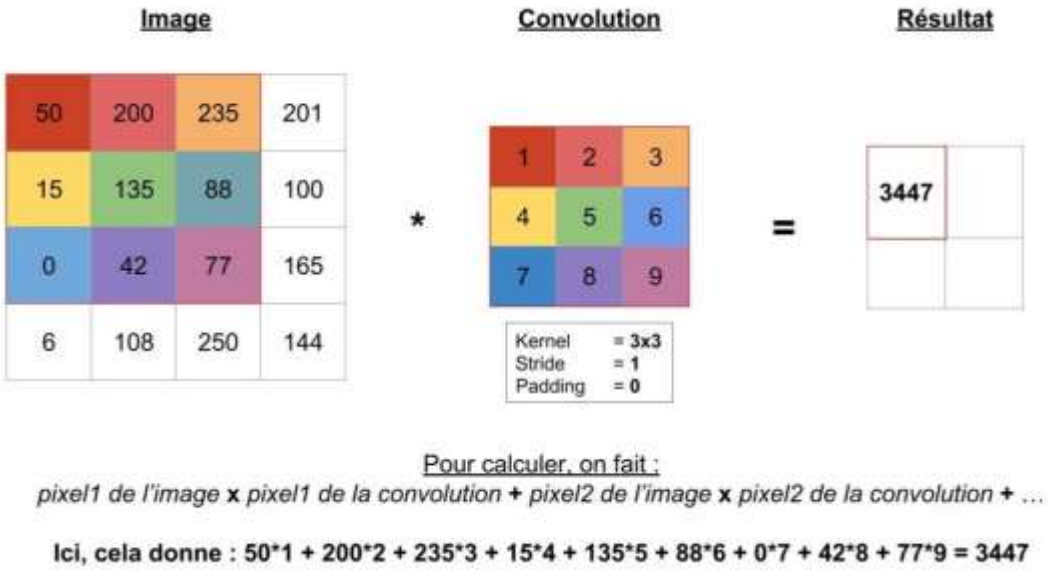

Figure 7 : Calcul d'une étape de la convolution (crédit : Lambert Rosique)

Un autre paramètre, le « stride », définit le nombre de pixels de décalage pour appliquer de nouveau la convolution. L'image, après application de la convolution, est plus petite que l'image d'origine. Le « padding » permet d'ajuster la taille de l'image de sortie, en paramétrant la manière dont on peut dépasser de l'image d'entrée pour appliquer la convolution.

b) *Le "pooling"*

Le pooling consiste à remplacer un carré de pixels par une valeur unique. Le plus utilisé est le « max-pooling » qui calcul la valeur maximale du carré : il permet de simplifier efficacement l'image. Ensuite, le carré suivant est défini selon le « stride » (décalage) choisi, généralement 1 ou 2.

c) *Le " flattening"*

Cela consiste à récupérer toutes les valeurs de toutes les images pour en faire un seul vecteur :

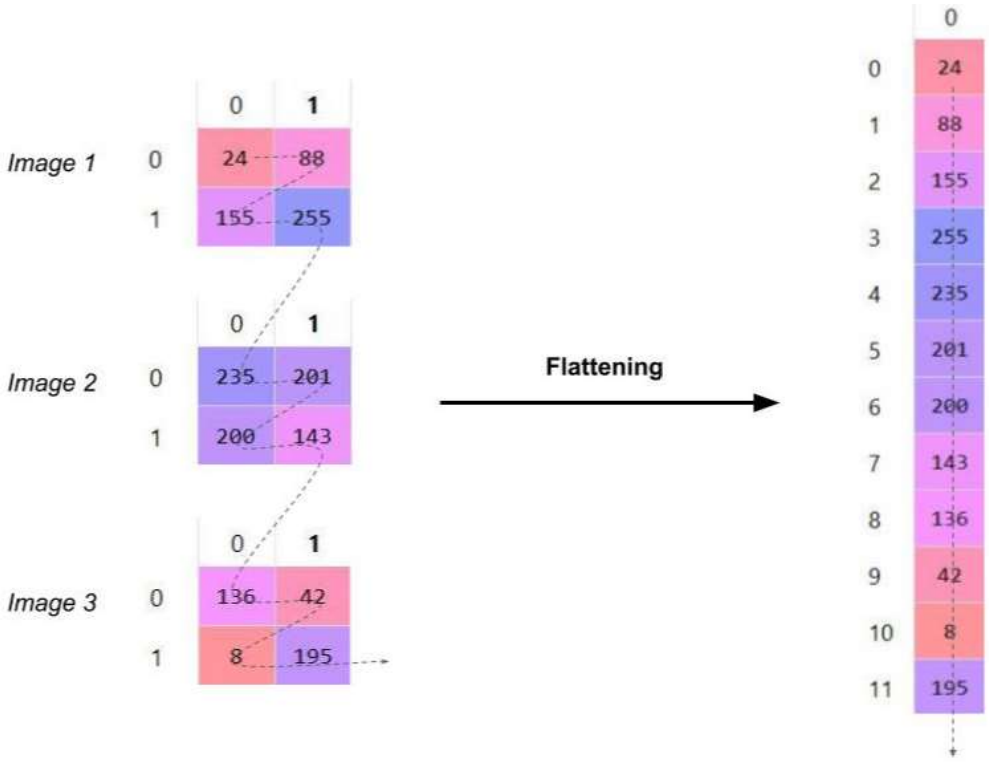

Figure 8 : Mise à plat des images finales en sortie des filtres + simplifications (crédit : Lambert Rosique)

d) *Le "Dense" ou "Fully connected"*

Elle reçoit en entrée le vecteur de nombres, sortie de « Flatten », et renvoie des probabilités pour chaque classe de prédiction en passant par le réseau de neurones artificiel.

e) *Autres couches*

La couche « Drop-out » est fréquemment utilisée dans un réseau CNN. Elle réduit les chances de sur-apprentissage en abandonnant des neurones de manière aléatoire à chaque « epoch ».

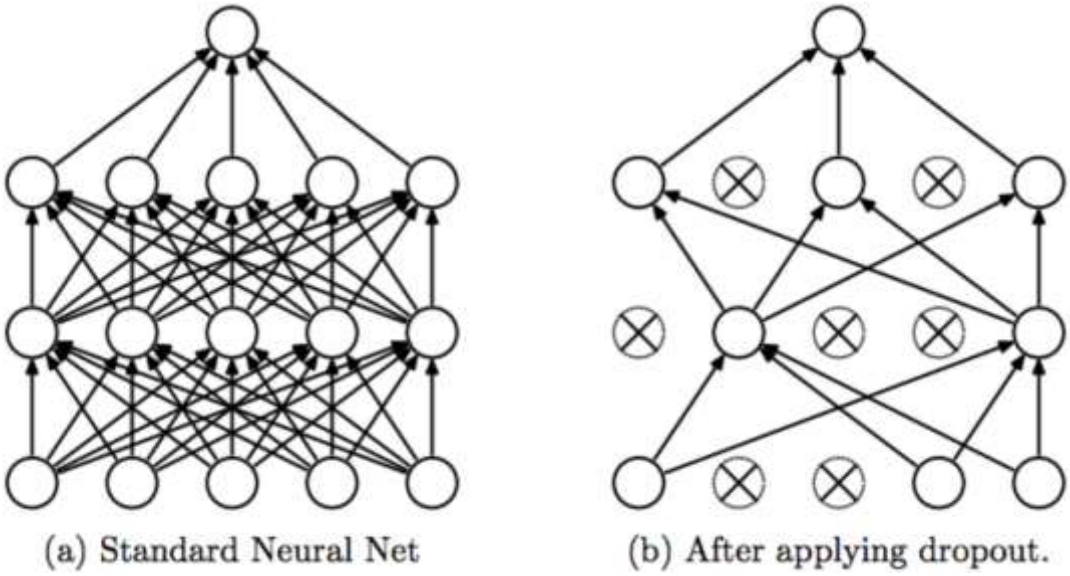

Figure 9 : Schéma de l'effet de l'application de la fonction Drop-Out[5]

f) *Schéma global d'un réseau de convolution*

Pour finir sur les CNN, voici un schéma récapitulatif d'un réseau :

Figure 10 : Un CNN est organisé en deux parties :

l'extraction de l'information et l'analyse de cette information (crédit : Lambert Rosique)

[5] Source : http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

3) Prototyper un réseau avec Keras

Différentes librairies peuvent être utilisées pour faire du Deep Learning :

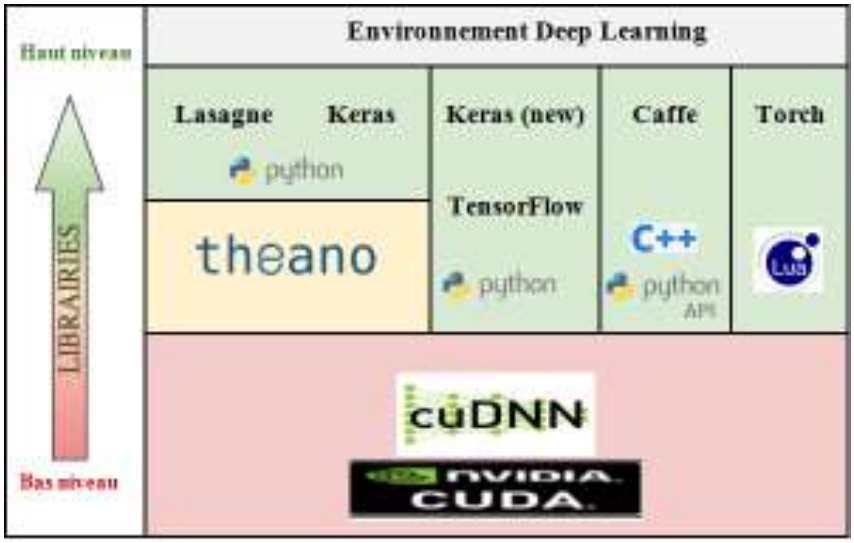

Figure 11 : Librairies utilisées pour le Deep Learning dans différents langages[6].

En python, Keras est une librairie de haut niveau qui permet de manipuler les réseaux de neurones en quelques lignes de code et fournit des blocs de construction élevés pour développer des réseaux de neurones profonds. Pour les opérations de bas niveau, Keras s'appuie sur d'autres bibliothèques spécialisées. Keras est soutenue par une large communauté et utilisée par des grandes entreprises telles que Netflix, Uber, Google ou la NASA.

Dans le code Python, un modèle est compilé grâce à la fonction « compile » de Keras dont les paramètres sont :

- ✓ « Le loss » : Il s'agit de l'écart entre la prédiction des labels et les classes « réelles ». Pour calculer cette valeur, le modèle doit définir une fonction de perte. Pour nos modèles, nous avons choisi la fonction « sparse_categorical_crossentropy ». L'entropie croisée consiste à calculer la différence moyenne entre les distributions de probabilité réelle et prévue pour toutes les classes du problème. Une valeur d'entropie croisée parfaite est 0.

- ✓ « L'optimizer » correspond à une implémentation particulière de l'algorithme de descente de gradient, c'est-à-dire qu'il va définir comment évolue le modèle pour s'adapter aux données d'entraînement.

[6] Source : https://blog.octo.com/classification-dimages-les-reseaux-de-neurones-convolutifs-en-toute-simplicite/

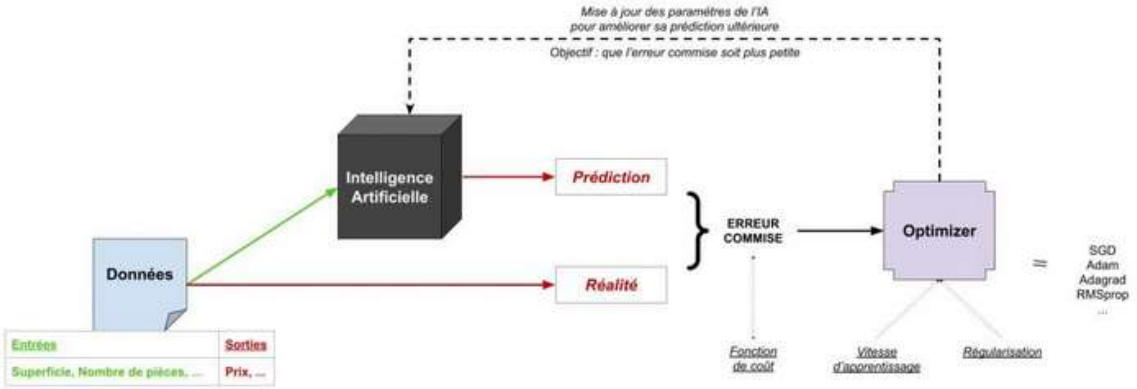

Figure 12 : Schéma explicatif du rôle de l'optimizer[7]

Nous avons utilisé pour nos modèles l'optimizer Adam, qui est le plus utilisé, du fait de son efficacité et de sa stabilité.

- ✓ « Metrics » : Ce sont des fonctions utilisées pour évaluer la performance du modèle. Nous avons choisi la fonction « accord globale », mais comme nous allons le voir dans les résultats, ce choix n'est pas forcément optimal.

4) Le modèle Inception V3

Le modèle Inception est un modèle de reconnaissances d'images couramment utilisé. Avant sa création, les réseaux CNN empilaient des couches de convolution de plus en plus profondément pour obtenir de meilleures performances.
Le principe d'Inception est de faire fonctionner des filtres de plusieurs tailles au même niveau, constituant ainsi des modules. Le réseau devient alors plus large, plutôt que plus profond. Les sorties des filtres de même niveau sont concaténées et envoyées au module de démarrage suivant.

[7] http://penseeartificielle.fr/meilleur-optimizer-ranger-radam-lookahead-fastai/

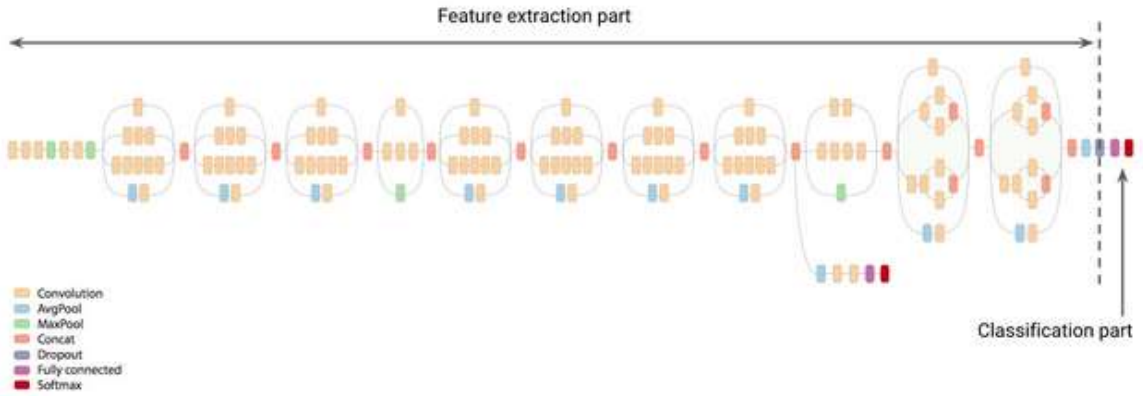

Figure 13 : Diagramme général du modèle Inception V3[8]

Améliorant les modèles Inception V1 et V2, le modèle Inception V3 empile 11 modules de démarrage qui regroupent des couches et des filtres convolutionnels avec des unités linéaires rectifiées comme fonction d'activation.

Dans le cadre de notre atelier, nos tests ont porté sur un modèle CNN que nous avons adapté[9] avant de tester le modèle Inception V3. En effet, ce modèle est à priori très performant sur la prédiction d'images et une de nos encadrante, Yousra Hamrouni a fait une formation sur les réseaux de neurones au cours de laquelle elle a effectué plusieurs tests sur ce modèle.

Dans la suite du rapport, nous détaillons les données utilisées, les modèles et leurs paramétrages avant de donner les résultats. L'objectif est de pouvoir tester la prédiction d'images de peupliers géoréférencées.

[8] Source : https://www.researchgate.net/figure/Inception-v3-architecture_fig1_328900168

[9] Cf. Partie du rapport Modèles et Résultats

B) DONNÉES UTILISÉES

Au cours de cette seconde partie de l'atelier, l'une des questions primordiales était le choix des données qui seraient utilisées pour entrainer et valider nos réseaux de neurones. La commande initiale étant de construire un réseau capable de prédire la présence de peupliers au sein d'un espace défini au préalable comme forestier, il nous était indispensable de travailler avec des images satellites. En effet, sur une image satellite les peupliers sont reconnaissables par la régularité de leurs plantations et la forme de leur cime.
Les choix de provenance et de résolution des images satellites a été fait suivant les conseils de Yousra Hamrouni ainsi que des autres encadrants.

En premier lieu, nous avons tenté de télécharger via des flux WMS et des API des données Sentinel et IGN. Le but était de pouvoir rendre le réseau de neurones autonome en lui fournissant un flux de données potentiellement constant et complet. Malheureusement les images Sentinel ne proposent qu'une résolution à 20m et celle-ci n'est pas suffisante pour l'étude des peupliers.

Nous avons alors reproduit le même schéma sur le flux proposer par l'IGN. Un script a été établi grâce à la clef d'activation du laboratoire qui permet d'obtenir l'ensemble des données proposées par l'institut. Cependant pour des questions de rapidité et de stockage mémoire, ce script n'a pas été utilisé et nous avons commencé par travailler sur les orthophotos à trois bandes de la région d'étude enregistrées sur leurs ordinateurs localement.
Sur les images à 5m, les caractéristiques des peupliers ne se dénotent pas clairement. A une résolution d'1m certains résultats peuvent être espérés mais le groupe, appuyé par les encadrants, a commencé ses recherches sur des images ayant une résolution de 50cm.

Afin d'entrainer et de valider les modèles, il était également nécessaire d'obtenir des données vectorielles qui comprenaient les labels que le modèle devait apprendre. Pour les peupliers, il nous fallait donc des polygones d'espace de foret qui représentaient à la fois des essences de peupliers et d'autres essences. Au cours de ses travaux de recherches, Yousra Hamrouni a construit un tel jeu de données. Elle a donc pu nous fournir, au format shapefile, des polygones avec 6 classes pour les 6 essences que ces polygones répertoriaient. Nous avons simplifié les polygones en les classant en 2 classes (peupliers et non peupliers).

Un premier code a alors été mis en place pour créer des sets de petites images d'entrainement et de validation des modèles à partir des rasters et de polygones de labels. Ce script est présent sur le GitHub de l'atelier et se nomme « extract_label_nn.py ». Ce dernier permet de créer des images de la taille de pixel souhaités et de les enregistrer sous un nom qui comprend le label.

Peupliers du Sud-Ouest de la France

Dans un premier temps, nous avons cherché à exploiter les données rasters de l'IGN pour prédire la présence de peupliers, cependant au vue de la résolution et la quantité des données, chaque traitement était long et nécessitait beaucoup de mémoire. Les premières extractions d'images, à partir de notre script, ainsi que les apprentissages étaient trop lourds pour certains de nos ordinateurs et il a été nécessaire d'utiliser l'application Google Colaboratory pour faire tourner les scripts.

Google Colaboratory est un outil de collaboration développé par Google. Il permet de travailler à plusieurs sur un script en langage python uniquement et de lancer en bénéficiant des capacités offertes par l'outil. C'est à dire une architecture CPU et 12 G de mémoire vive.

Malgré l'utilisation de ces outils, nous étions ralentis par la lourdeur des calculs. Une partie de l'équipe était alors dédié à l'étude des modèles et de leurs couches. Sans données, il leur était impossible d'avancer.

Difficultés et contraintes

La quantité de données s'est donc avérée être une difficulté plutôt qu'un atout pour notre atelier. De plus celui-ci ne durant qu'un mois et demi, nous étions contraints par le temps. Sans jeu de données viable, il nous était impossible d'approfondir nos connaissances sur les modèles de réseau de neurones à tester et utiliser. Afin de pouvoir continuer nos recherches et tester nos avancées, il a été décidé de s'appuyer sur un autre jeu de données qui nous a permis de comprendre la construction d'un réseau de neurones.

Carte Historique de la Finlande

Figure 14 : Extrait de la carte de Finlande de Jussi Lampinen

Au cours de la 4ème semaine, Nicolas Karasiak a été contacté par un chercheur finlandais, Jussi Lampinen, qui travaille pour l'université de Turku. Ses recherches consistent à numériser l'occupation du sol des cartes historiques.

Il nous a alors fourni des données vectorielles et rasters sur lesquelles il travaille afin d'étudier la possibilité d'effectuer ce travail par le biais du Deep Learning.

Ces données, nous ont permis de momentanément mettre les données de peupliers en pause et travailler sur des données issues de cartes historiques de la Finlande. Ces dernières étaient beaucoup plus légères à traiter et offraient plus de polyvalence à notre groupe pour entrainer nos modèles.

Ces cartes historiques comprennent plusieurs classes mais ici seules 6 classes sont étudiées : *Forest, Mesic grassland, Moist grassland, Arable, Pasture, Peatland.* Les données vectorielles proposées par Jussi Lampinen n'étant pas assez denses, nous avons donc créer de nombreux polygones de chaque classe nous permettant de créer les données d'entraînement et de validation de nos modèles avec le code mentionné précédemment.

L'échantillon mis à notre disposition était assez léger et permettait à chacun d'effectuer de nombreux tests pour comprendre l'architecture d'un réseau de neurones. Pour ces cartes, nous avons créé des images de 30 pixels par 30 pixels et avons obtenu 1202 éléments en entrainement et 1494 en validation. Ces jeux de données ont permis d'entrainer certains des modèles présentés dans la partie suivante (Modèles et résultats).

Retour sur les peupliers occitans

Une fois que l'ensemble de l'équipe eut pris connaissance des fonctionnements d'un réseau et ait pu faire des tests sur ses machines ou sur Google Colaboratory, il nous a été possible de réutiliser le modèle issu de l'analyse des cartes de Finlande sur les données de peupliers.

Dans un premier temps, suite à une erreur, le code ne sélectionnait que deux essences d'arbres : le peuplier et le robinier. Un paramètre du code empêchait de prendre la bonne colonne dans la table attributaire du fichier gpkg. Cependant cette erreur a tout de même permis de produire environs 1900 images en entrainement et 1700 en validation (jeu de données 1). En corrigeant cette erreur et en prenant l'ensemble des autres essences d'arbres le script produisaient plus de 55 000 images de 30x30 en validation et 54 000 en entrainement (jeu de données 2).

Grâce à cette erreur, le groupe a d'abord pu entrainer ses modèles sur le jeu de données 1, moins conséquent et donc plus facile à traiter. A titre d'exemple, l'entraînement d'un modèle sur le jeu de données 1 avec 10 epochs prenait un peu plus d'une dizaine de minutes. En comparaison, avec le même nombre d'epochs mais le jeu de données 2, l'entrainement prenait plus de 20 heures et nécessitait une importante mémoire vive dont aucun ordinateur ne disposait. En traitant, en premier lieu, le jeu de données 1, nous avons pu obtenir de premiers résultats tangibles et valides qui démontraient que le réseau fonctionnait.

Le laboratoire Dynafor possède deux serveurs mis à disposition des chercheurs pour traiter des calculs qui nécessitent d'importants moyens. TOSCA, l'un des deux serveurs, possèdent plus de 48 coeurs et 400 G de mémoire vive (2299 Mhz par CPU : 48 au total, soit 110ghz). Nous avons utilisé la puissance de ce serveur afin de pouvoir entrainer, valider et prédire notre modèle à partir du jeu de données 2. Par ailleurs, la bibliothèque Tensorflow intègre au sein de son script un parallélisme de ses entrainements ce qui nous a permis d'utiliser l'ensemble des cœurs disponibles.

En conclusion, travailler à la fois sur la Finlande et les peupliers nous a permis d'allier efficacité et efficience. En travaillant uniquement sur les peupliers, nous n'aurions pas pu acquérir une telle connaissance des réseaux de neurones et aurions fini l'atelier avec beaucoup moins d'avancements. En travaillant uniquement sur les cartes de la Finlande, nous n'aurions pas appris à gérer au mieux une chaine de traitement pour de lourd jeu de données. Avancer rapidement avec un jeu de données léger permet d'être sûr des traitements lancés sur un lourd jeu de données qui devient opaque à chaque traitement

C) MODÈLES & RÉSULTATS

Pour commencer à appréhender les réseaux de neurones, plusieurs guides ont été suivis sur des modèles de données récupérés en ligne. Cela nous a permis de comprendre le fonctionnement des différentes couches et des différents paramètres des réseaux de neurones.

Pour effectuer des premiers tests, plus personnels, sur les modèles de réseaux de neurones avec les données qui nous intéressaient, nous avons commencé par créer des modèles inspirés de ceux déjà vus et adaptés à nos données en entrée.

a) MODELE 1 - Sans InceptionV3 – Prédictions de la carte d'occupation des sols

✓ Construction du modèle

Le premier modèle prend en entrée un jeu de données sur la carte d'occupation du sol. Il comporte une série de couches de convolution (Conv2D) mixés avec une couche d'activation Relu (Rectified Linear Unit - Unité linéaire rectifiée) et de MaxPooling pour effectuer un apprentissage à plusieurs niveaux. Des couches de Dropout sont intercalés dans cette série de couches afin de réduire le nombre d'interactions entre les neurones de sorte à réduire le surentrainement (ou overfitting). A la fin, une couche Flatten a été interposée pour aplatir le jeu de données afin de permettre aux couches Dense de connecter tous les neurones entre eux et finaliser l'entraînement avec une couche d'activation Softmax et 6 neurones de sortie pour prédire les 6 classes différentes.

Les données prises en entrée de ce modèle sont des imagettes de dimension (30,30,1). L'entraînement a été effectué avec 1494 imagettes et la validation avec 1202 imagettes. Nous avons décidé de conserver l'optimiseur « adam » puisqu'il paraît que c'est un des plus robustes et des plus utilisés. Après quelques tests avec d'autres optimiseurs (Adadelta, …) il s'est avéré que c'est celui qui donnait les meilleurs résultats pour notre modèle. Dans la portion de code ci-dessous, l'entraînement s'est fait avec 150 epochs mais nous avons fait tourner ce modèle avec plusieurs valeurs d'epochs allant de 10 à 150. Un schéma explicatif de chaque étape du modèle est affiché en figure 11

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape =(30,30,1), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(64, (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(128, (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(units = 64, activation = 'relu'))
model.add(Dense(units = 6, activation = 'softmax'))

model.compile (optimizer = 'adam', loss='sparse_categorical_crossentropy', metrics = ['accuracy'])

model.fit(train_features, train_classes, epochs=150, verbose=1)
```

Nous avons effectué l'entraînement du modèle plusieurs fois à partir de zéro pour voir ce que ça pouvait donner sur les variations lors de l'apprentissage. Les résultats présentés ci-dessous sont parmi les meilleurs ayant été obtenus par ce modèle sur ce jeu de données.

- ✓ **Résultats**

Val. Loss = 0.3616096977835611 – Val.acc = 0.88352746

La précision des prédictions de plus de 88% sur le jeu de validation est encourageante. Le modèle semble concluant pour des premiers essais. Cependant, la valeur du loss semble assez élevée ce qui indique une perte d'information non négligeable dans le modèle.

[Forest, Mesic grassland, Moist grassland, Arable, Pasture, Peatland] = [205, 93, 94, 216, 2, 592]

Les 6 classes étaient représentées sur cet essai (ce qui n'a pas été le cas à chaque fois), mais on peut voir qu'il y a des effectifs très variables. La répartition réelle des classes des imagettes de validation est :
[Forest, Mesic grassland, Moist grassland, Arable, Pasture, Peatland] = [211, 34, 143, 176, 52, 586].
Les classes 1 « Forest » et 6 « Peatland » semble également être plutôt bien identifiées par rapport aux autres. Les classes 3 « Moist grassland » et 5 « Pasture » sont vraiment sous-représentée. Et au contraire les classes 2 « Mesic grassland » et 4 « Arable » sont vraiment sous représentées. On peut faire les mêmes observations lors de la répétition du modèle sur le jeu de données.

- ✓ **Critiques**

Le modèle semble robuste mais il reste critiquable. En effet, la taille des imagettes en entrée (30x30) est petite ce qui limite grandement le nombre d'opérations de convolution qu'il est possible de faire dans le modèle. L'apprentissage du modèle par entraînement serait probablement plus robuste avec des imagettes de plus grande taille. Cependant, augmenter la taille des imagettes aurait eu pour effet de réduire considérablement le nombre de données d'entraînement, qui est déjà assez peu important. Usuellement, un réseau de neurones robustes se base sur plusieurs dizaines/centaines de milliers d'échantillons en entrée cependant notre modèle n'en prend qu'un peu moins de 1500. Il était donc inconcevable d'augmenter la taille des imagettes, pour rester cohérent avec l'apprentissage d'un réseau de neurones sur une quantité d'échantillons assez importante. D'un autre côté, le temps alloué à la création et à l'entraînement d'un réseau de neurones était assez court ce qui limitait les temps d'extraction disponibles d'extraction des jeux de données ainsi que les temps de calcul.

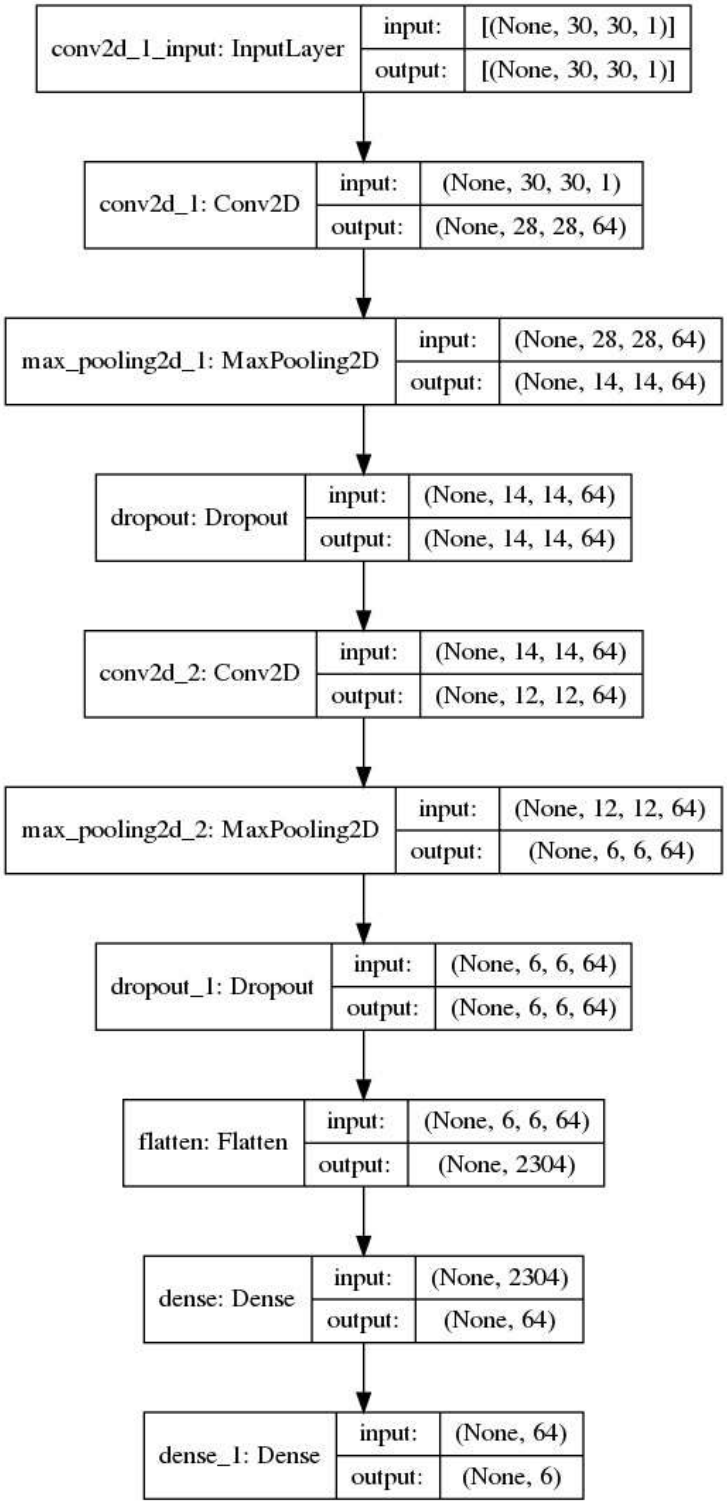

Figure 15 : Schéma de l'architecture du modèle 1

b) *MODELE 2 - Avec InceptionV3 et prédictions de la carte d'occupation des sols*

✓ **Construction du modèle**

Le second modèle prend en entrée le même jeu de données sur la carte d'occupation du sol que le précédent. La différence avec le premier est qu'on utilise le modèle InceptionV3 décrit précédemment pour entraîner le modèle. A la fin, une couche Flatten a été appliquée pour aplatir le jeu de données afin de permettre la finalisation de l'entraînement avec une couche d'activation Softmax et 6 neurones de sortie pour prédire les 6 classes différentes.
Les données prises en entrée de ce modèle sont des imagettes de dimension (90,90,3). En effet, le modèle inceptionV3 nécessite des imagettes en entrée avec une dimension minimale de (75,75,3). Pour éviter qu'il y ait une trop grosse transformation des données, les imagettes de dimension (30,30,3) ont subi un reshape avec un facteur 3 sur les deux premières dimensions pour multiplier par 9 le nombre de pixels.
Dans les portions de code ci-dessous, l'entraînement s'est fait avec 50 epochs mais nous avons fait tourner ce modèle avec plusieurs valeurs d'epochs allant de 10 jusqu'à 150 epochs.

```
Incep = InceptionV3(weights='imagenet', include_top=False, input_shape=(90, 90, 3))

model = Sequential()
model.add(Incep)
model.add(Flatten())
model.add(Dense(6, activation= 'softmax'))

model.compile (optimizer= "adam",
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

history = model.fit(train_features, train_classes, epochs=50, verbose=1)
```

✓ **Résultats**

Avec epochs = 50

Val. Loss = 0.05153570782446776 – Val.acc = 0.98668885

[Forest, Mesic grassland, Moist grassland, Arable, Pasture, Peatland] = [217, 33, 135, 176, 52, 589]

Avec epochs = 100

Val. Loss = 0.025807651022428605 – Val.acc = 0.9916805

[Forest, Mesic grassland, Moist grassland, Arable, Pasture, Peatland] = [219, 34, 139, 176, 52, 582]

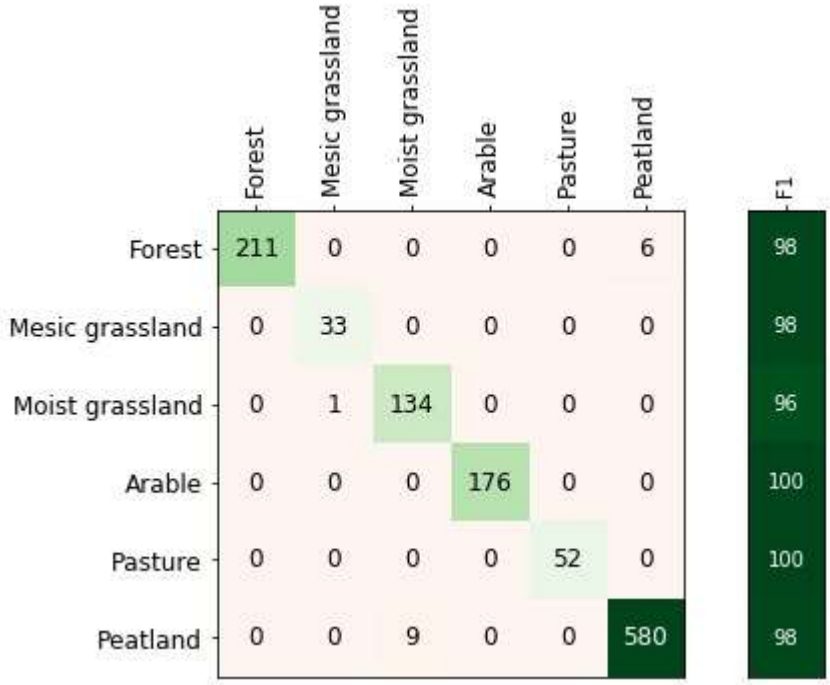

Figure 16 : Matrice de confusion sur la prédiction des données de validation avec 50 epochs

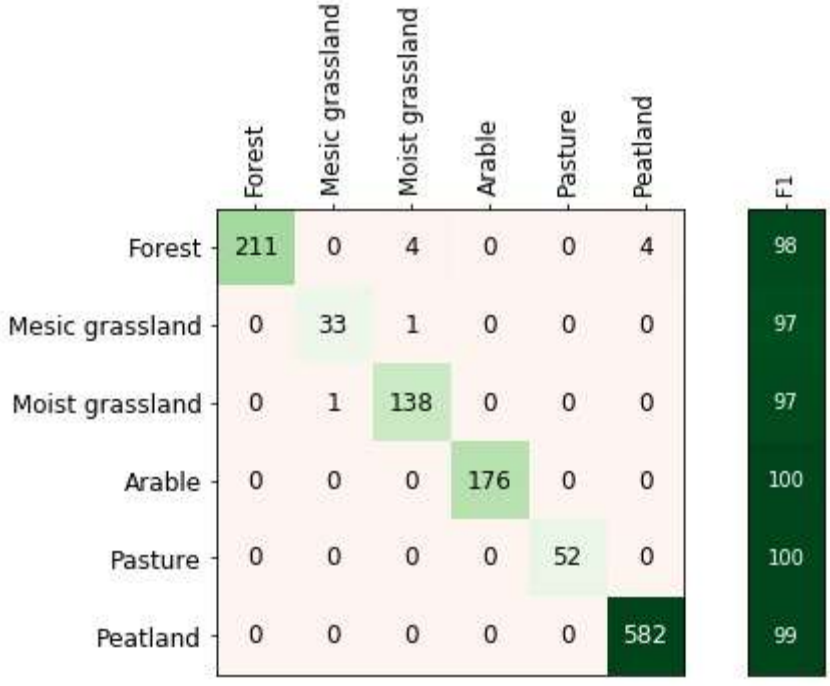

Figure 17 : Matrice de confusion sur la prédiction des données de validation avec 100 epochs

La précision des prédictions sur le jeu de validation est de plus de 98.5% pour des valeurs de 50 epochs ou supérieures ce qui en fait un modèle qui semble très robuste et efficace pour prédire un jeu de données qu'il n'a jamais vu. La valeur du loss est considérablement diminuée en comparaison avec le modèle 1 avec des valeurs de 5% au maximum. La perte d'informations est donc assez faible.
Les 6 classes sont représentées pour chacun des essais avec ce modèle. Les effectifs sont encore très variables mais la valeur de la précision nous indique que la répartition est proche de la réalité.
En effet, la répartition réelle des classes des imagettes de validation est la même que pour le modèle précédent.

[Forest, Mesic grassland, Moist grassland, Arable, Pasture, Peatland] = [211, 34, 143, 176, 52, 586].

Les données prédites par le modèle sont vraiment très proches de la réalité comme le montre la proximité des effectifs de chacune des classes prédites avec les classes réelles.

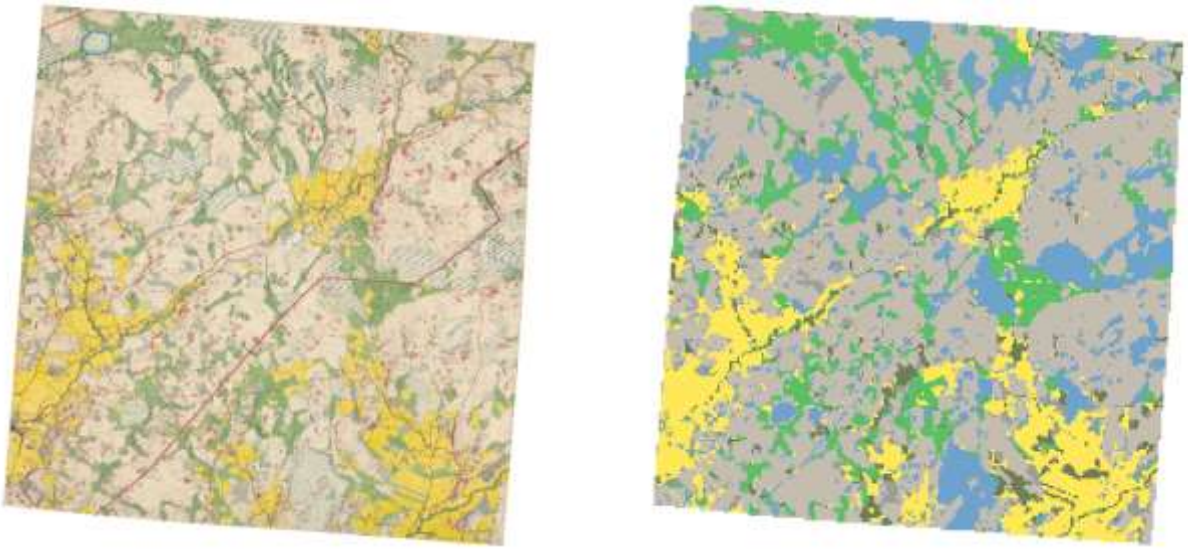

Figure 18 : Aperçu des résultats de la prédiction

A gauche, la carte d'occupation des sols finlandaises, A droite la prédiction du modèle

- ✓ **Critiques**

Le modèle semble très robuste, cependant quelques critiques peuvent être émises. En effet, comme pour le modèle précédent la taille des imagettes en entrée (90x90), issues d'un reshape à partir d'imagettes de dimension (30x30), est petite. La taille des images d'entraînement qu'a utilisé le modèle InceptionV3 pour s'entraîner est de 299x299, du coup des imagettes de cette taille-là serait préférable pour améliorer les prédictions. Cependant, de même que pour le modèle précédent, augmenter la taille des imagettes aurait eu pour effet de réduire considérablement le nombre de données d'entraînement, ce qui réduirait le nombre de données en entrée qui n'est déjà pas très élevé.

De plus, les données d'entraînement sont ne représentent qu'une seule et unique classe et ressemblent aux données de validation (d'autant plus que les imagettes sont petites). Si on applique ce modèle sur des bloc contenant plusieurs classes, le modèle a des chances d'être moins précis.

L'objectif final du modèle est de faire des prédictions sur une image entière subdivisée en blocs de 30x30. Si les blocs se retrouvent à cheval entre deux zones différentes, il risque d'y avoir une chance accrue d'erreur. En théorie, le bloc devrait prédire la classe la plus présente sur le bloc mais le manque de données d'entraînement sur des zones à cheval pourrait peut-être causer des lacunes de ce côté-là. Pour y remédier, il faudrait peut-être inclure davantage de données d'entrainement avec des chevauchements légers entre différentes zones pour rendre le modèle plus souple sur une prédiction sur une image entière et non sur un set de validation préconstruit.

L'opacité du modèle InceptionV3 ne permet également pas de connaître précisément les opérations faites à l'intérieur et il n'y a pas de possibilités de modifications ou d'adaptations à partir de celui-ci mais il faut reconnaître qu'il reste tout de même très efficace pour prédire des données.

c) *MODELE 3 - Avec InceptionV3 – Prédictions sur une ortho photo – Résolution de 50cm*

✓ *Construction du modèle*

Ce modèle prend en entrée un jeu de données sur l'ortho-photo avec une résolution de 50cm. Le modèle utilise encore le modèle InceptionV3 décrit précédemment. Une couche Flatten a été ajoutée à la fin pour aplatir le jeu de données afin de permettre la finalisation de l'entraînement avec une couche d'activation Sigmoïde pour obtenir 2 neurones de sortie afin de prédire la présence ou l'absence de peupliers dans l'imagette.

Les données prises en entrée de ce modèle sont des imagettes de dimension (90,90,3). De même que les modèles précédents, les imagettes de dimension (30,30,3) ont subi un reshape avec un facteur 3 sur les deux premières dimensions pour multiplier par 9 le nombre de pixels. Cette transformation a pour but d'éviter qu'il y ait une trop grosse transformation des données tout en remplissant les conditions nécessaires pour utiliser le modèle.

Dans les portions de code ci-dessous, l'entraînement s'est fait avec 50 epochs pour 1427 imagettes d'entraînement. La validation s'est faite sur 1906 imagettes.

```
Incep = InceptionV3(weights='imagenet', include_top=False, input_shape=(90, 90, 3))

model = Sequential()
model.add(Incep)
model.add(Flatten())
model.add(Dense(2, activation= 'sigmoid'))

model.compile (optimizer= "adam",
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

history = model.fit(train_features, train_classes, epochs=50, verbose=1)
```

✓ *Résultats*

Val. Loss = 0.68789 – Val.acc = 0.8326338

[Peuplier, Non peuplier] = [563, 1343]

La précision globale des prédictions de plus de 83% sur le jeu de validation semble correcte. Le modèle semble concluant pour des premiers essais sur les orhtos-photos. Toutefois, la valeur du loss semble assez élevée : la perte de précision entre les données d'entrainement et de validation est assez importante ce qui indique un potentiel surentrainement du modèle.

La répartition réelle des classes des imagettes de validation est : [Peuplier, non peuplier] = [840, 1066].

On observe que 563 imagettes du jeu de validation ont été prédites comme des imagettes de peuplier alors qu'il y en a 840 en réalité et 1343 ont été prédites comme du non peuplier alors qu'il y en a 1066 en réalité. Le modèle semble avoir été surentrainé et reconnaît un plus grand nombre d'imagettes de non peupliers que nécessaire.

✓ ***Critiques***

Le modèle est moins précis que le précédent, il comporte les mêmes couches mais il semble être moins adapté que le précédent aux jeux de validation et d'entraînement. Comme pour le modèle précédent, la taille des imagettes en entrée est de 90x90, issues d'un reshape à partir d'imagettes de dimension (30x30). Le problème concernant la taille des images reste le même.

Ce qui change avec le modèle précédent est que les résultats en sortie sont de nature binaire (Peuplier ou non peuplier). On constate que le nombre de peupliers est bien sous-évalué par rapport au nombre de peupliers réel sur le jeu de données de validation. Le nombre d'erreurs de classification d'imagettes de peupliers en non peuplier est assez important.

La structure de ce modèle semble toutefois assez correcte pour faire des prédictions à plus grande échelle. Le modèle suivant suivra la même structure mais sera entraîné sur un jeu de données d'entrainement beaucoup plus grand.

d) MODELE 4 – Sans InceptionV3 – Prédictions sur une ortho photo – Résolution de 50cm

✓ ***Construction du modèle***

Ce modèle prend en entrée un jeu de données sur l'ortho-photo avec une résolution de 50cm. Il est fait d'une structure similaire à celle décrite précédemment dans le modèle 1 sans InceptionV3. Une couche Flatten a été ajoutée à la fin pour aplatir le jeu de données afin de permettre la finalisation de l'entraînement avec une couche d'activation Sigmoïde pour obtenir 2 neurones de sortie afin de prédire la présence ou l'absence de peupliers dans les imagettes.
Les données prises en entrée de ce modèle sont 1678 imagettes de dimension (30,30,3). La moitié d'entre elles (839) sont des imagettes représentant des peupliers et les autres représentent des non peupliers.
Dans les portions de code ci-dessous, l'entraînement s'est fait avec 200 epochs. Nous avons rajouté dans la compilation du modèle, pour le paramètre « metrics » le paramètre *f1_m*, pour que soit pris en compte la valeur du F1 moyen dans les performances du modèle et pas seulement l'accord global, qui semble être une donnée pouvant nous induire en erreur.

```
model = Sequential()
model.add(Conv2D(32, (3,3), input_shape = (30,30,1), activation ='sigmoid'))
model.add(MaxPooling2D(pool_size(2,2)))
model.add(Dropout(0,5))
model.add(Conv2D(64, (3,3), activation ='sigmoid'))
model.add(MaxPooling2D(pool_size(2,2)))
model.add(Dropout(0,5))
model.add(Conv2D(8, (3,3), activation ='sigmoid'))
model.add(MaxPooling2D(pool_size(2,2)))
model.add(Dropout(0,5))
model.add(Flatten())
model.add(Dense(units = 64 , activation= 'sigmoid'))
model.add(Flatten())
model.add(Dense(units =2, activation= 'sigmoid'))
model.compile(optimizer = 'adam', loss='sparse_categorical_crossentropy', metrics=['accuracy', 'f1_m'])
model.fit (train_features, train_classes, epochs=200, verbose=1)
```

- ✓ ***Résultats***

Une prédiction a été faite sur les données de validation avec le modèle créé et les résultats suivants ont été obtenus :

Loss = 0.1898 – Accuracy = 0.9247 – f1_m = 0.51
[Peuplier, Non peuplier] = [3665, 50664]

La validation s'est faite sur 54 329 imagettes, dont seulement 733 images de peupliers.
D'après la valeur de l'accord global (92%), la précision des prédictions sur le jeu de validation semble très correcte et la valeur du F1 moyen indiquée est de 0.51. Ces valeurs indiquent que les prédictions semblent plutôt justes. Toutefois, malgré la valeur de précision importante indiquée par le modèle on constate un grand écart entre le nombre réel de peupliers et le nombre prédit !
Ce problème de détection des peupliers est confirmé par la matrice de confusion :

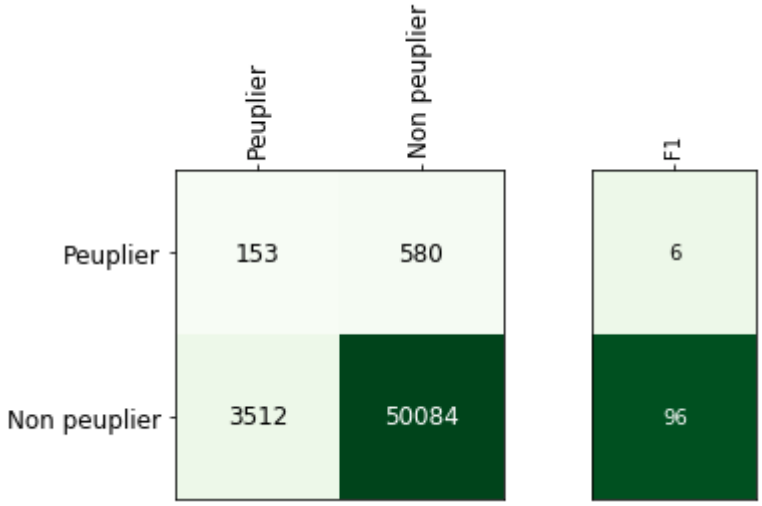

Figure 19 : Matrice de confusion des résultats du modèle 4 (jeu de validation).

Les données en ligne correspondent aux données de terrain, celles en colonne aux valeurs prédites par le modèle. On voit ici que les imagettes de peupliers ont été très mal prédites : seulement 6% des échantillons de peupliers ont été bien prédits. 580 imagettes de peupliers ont été prédites en tant que non peupliers et 3512 images de non peupliers parmi 53596 ont été prédites en tant que peupliers.

- ✓ ***Critiques***

Pour aller plus loin dans l'analyse, une prédiction a été faite sur le modèle d'entraînement. Les résultats et la matrice de confusion sur les données d'entraînements suivants ont été obtenus :

Loss = 0.5678 – Accuracy = 0.704 – f1_m = 0.76
[Peuplier, Non peuplier] = [465, 1213]

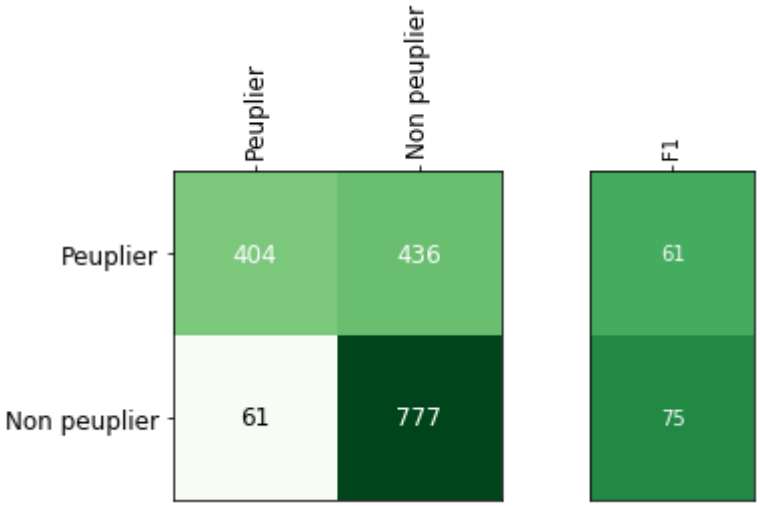

Figure 20 : Matrice de confusion des résultats du modèle 4 (jeu d'entraînement).

La figure 20 montre que même des prédictions effectuées sur les données avec lesquelles ont été entraîné le modèle sont mauvaises. Un modèle qui aurait bien appris avec les données d'entraînement aurait eu une précision très élevée et des scores de F1 plus importants.

Les résultats de ce modèle sont donc très décevants car les peupliers sont très mal classés. On s'aperçoit ici qu'on ne peut pas seulement se fier à l'accord global pour évaluer la performance du modèle. Ici, l'accord global de 0.86 nous induit en erreur car la matrice de confusion montre que les résultats du modèle sont très mauvais. Il aurait sans doute fallu prendre en compte la valeur du F1 dès les premiers tests du modèle pour s'assurer de la représentativité de l'accord global et pour voir si le modèle est bien entraîné ou pas.

Ces résultats ont été obtenus à la toute dernière minute du projet, mais ils conduisent à s'interroger sur l'ensemble des étapes du processus : choix du jeu de données, taille des images en entrée, paramètres du modèle…

Un modèle contenant InceptionV3 a été entraîné à l'aide de la totalité des imagettes extraites des orthophotos sur le serveur TOSCA mais le F1 des peupliers avoisinait 0 donc les résultats n'ont pas été présentés ici.

Parmi les 4 modèles expérimentés, on peut affirmer que seuls les modèles sur la carte de la Finlande sont plutôt bien entraînés. Ils fournissent des résultats plutôt bons et classent les images correctement. On peut également affirmer que les modèles que nous avons créés à partir de jeux de données tirés des ortho photos ont du mal à différencier les peupliers des non peupliers. On peut émettre l'hypothèse que les classes d'occupation du sol sur la carte de la Finlande sont plus facilement repérables grâce à leurs différences de couleurs et de textures facilement détectables à l'œil nu. En revanche, les modèles appliqués sur les ortho photos semblent ne pas distinguer les différentes espèces forestières, sans doute trop proches quant à leurs caractéristiques physiques. Un jeu de données plus uniforme et complet aurait peut-être pu apporter une amélioration aux modèles.

D) CONCLUSION & CRITIQUES PARTIE II

L'objectif initial de cette partie était d'entraîner des réseaux de neurones dans le but de détecter la présence de peupliers à partir d'orthophotos de l'IGN. Malgré le délai assez court pour la réalisation de cette étape, nous avons réussi à produire une carte de prédictions des peupliers à partir d'un modèle inspiré du modèle InceptionV3. Cependant, comme nous l'avons vu dans la partie précédente, les résultats obtenus sont médiocres. Après ces semaines de recherches et de résultats, parfois peu satisfaisants, il est possible d'identifier les difficultés majeures rencontrées et les pistes d'améliorations.

- ✓ **Compréhension et opacité Keras**

Il est important de souligner que la compréhension de modèles préexistants peut être compliqué. La bibliothèque Keras met à disposition de nombreux modèles pré-entrainés qui sont parfois difficiles à appréhender pour des néophytes des réseaux de neurones. Une étude plus approfondie des modèles utilisés ainsi que des nouveaux modèles sortis tels que le InceptionV4 pourrait améliorer ce point.

- ✓ **Résultats encourageants mais perfectibles**

Les précisions globales ou « Accuracy », des modèles que nous avons entrainés semblent de bonne qualité, cependant cette valeur n'est pas toujours fiable. Lors de la phase de tests, l'accuracy des modèles pouvait varier de plus ou moins 30 % selon l'entrainement sans raison apparente. La précision globale n'est donc probablement pas un bon indicateur pour décrire la précision du modèle de ce point de vue. Un autre indicateur comme le score F1 (moyenne harmonique de la précision et du rappel) ainsi qu'une matrice de confusion offre une meilleure certitude des résultats obtenus.

- ✓ **L'importance de la validation**

Lors de l'utilisation de réseaux de neurones pour la reconnaissance d'images sur une carte, il est important de créer des jeux de données de validation éloignées de ceux d'entrainement. Nous aurions pu obtenir des scores plus élevés mais faussés d'accuracy si les images d'entraînement et de validation étaient proches et donc similaires. Avec une méthode de validation pour créer nos échantillons nous nous assurons d'obtenir des résultats plus justes.

- ✓ **Données d'entrées, densité, taille ?**

Le modèle Inception V3 a été entraîné avec des images d'une taille de 299 par 299 pixels. A l'origine il a été conçu pour reconnaitre des races de chats et de chiens. Nous avons utilisé ce modèle avec des images d'une dimension

de 30 par 30 pixels en les redimensionnant. Nous n'offrons donc pas au modèle des données optimales pour son fonctionnement. Pour entraîner de façon plus robuste le modèle, il serait judicieux d'utiliser des images d'entrainement plus grandes. Cependant, la résolution de la prédiction sur l'ortho photo serait plus basse et donc les prédictions potentiellement moins précises.

Par ailleurs, pour améliorer les performances du modèle, il serait nécessaire d'augmenter le nombre d'échantillons de peupliers qui sont actuellement trop peu nombreux

- ✓ **Améliorer l'existant**

Dans l'ensemble, même si les résultats sur les peupliers laissent à désirer, les résultats sur les cartes de la Finlande sont encourageants. De plus, nous avons construit une chaine de traitement complète qui permet d'extraire un jeu de données, de le traiter et de prédire une carte. Aujourd'hui, peu de contributeurs GitHub et de spécialiste de réseaux de neurones partagent leurs méthodes pour créer les jeux de données labellisés à partir de raster. Cette chaine de traitement est disponible sur le GitHub de l'atelier et peut être facilement réutilisée et re-paramétrée à la guise de son futur utilisateur. En somme, il revient aux prochains utilisateurs d'améliorer et de partager leurs résultats en open source.

CONCLUSION DE L'ATELIER

COMPETENCES ACQUISES

En premier lieu, cet atelier a permis à l'ensemble du groupe d'approfondir les compétences en télédétection à partir de la programmation en Python.
Des aspects essentiels du traitement d'images ont étés abordés : les filtres spatiaux, l'amélioration de la vitesse de calcul de l'image, la gestion des masques. En travaillant à partir de la classe RasterMath de MuseoToolBox nous avons pu nous familiariser avec le traitement des images par bloc de pixels. Poursuivre dans la logique du code existant, nous a aidé à répondre aux problématiques posées : création d'une fonction spatiale et ajout de la possibilité de parallélisation à RasterMath. Nous avons approfondi la connaissance et l'utilisation de bibliothèques déjà connues (numpy, gdal) et abordés de nouvelles bibliothèques (joblib, multiprocessing).

Concernant les réseaux de neurones, nous avons découvert cette partie du MachineLearning pour le traitement d'images. Les recherches réalisées nous ont aidées à comprendre le fonctionnement et les paramètres d'un modèle de réseau de neurones. L'objectif était de pouvoir faire des essais sur la détection de peupliers à partir d'imagettes. Au vue de la lourdeur de ces données, nous avons commencé à tester les modèles de réseau sur une carte historique finlandaise d'occupation du sol avec six classes. Ainsi, nous avons pu obtenir des résultats sur un modèle que nous avons construit, puis sur le modèle très utilisé InceptionV3. Suite à cela, et ayant réussi à créer des imagettes de peupliers, ces modèles ont été testés sur celles-ci. Au final, nous avons pu nous rendre compte que dans ces manipulations, il était fondamental de réfléchir au choix du jeu de données et également de remettre en cause les paramètres choisis et les résultats obtenus. En effet, nous avons vu que même avec un accord global dépassant les 80%, on pouvait avoir un score de prédiction de la classe « peupliers » très éloigné de la réalité.

Enfin, l'atelier nous a permis de mieux connaître et d'utiliser Github, en y intégrant les codes réalisés pour la partie réseau de neurones.

PERSPECTIVES

Sur MuseoToolbox, l'intervention de Nicolas Karasiak a déjà permis d'optimiser le code pour rendre compatible à la fois l'ajout de fonctions spatiales et le traitement en parallèle.
Sur les réseaux de neurones, nous avons pu tester un premier processus allant de la création d'imagettes, la création d'un modèle et le test sur une image entière.

Ces étapes peuvent servir de base pour aller plus loin. En premier lieu, une réflexion peut être engagée sur les données en entrée du modèle, notamment sur la taille des images et le nombre d'échantillons. De même, il semble que des recherches et des essais peuvent être réalisés sur les paramètres du modèle : nombre d'epochs, remplacer l'accord global

par la valeur de prédiction de la classe (F1). Enfin, il faut sans doute aller plus loin dans la compréhension du modèle InceptionV3 ou se tourner vers d'autres modèles pour traiter les images.

Cet atelier nous a permis d'apprendre énormément de nouvelles notions en télédétection, tout en faisant le lien avec les aspects abordés en cours. Nous avons pu aborder le domaine des réseaux de neurones, domaine aux applications multiples et en évolution constante ces dernières années. L'atelier comprenait deux commandes étalées sur seulement 6 semaines, les deux premières ont été dédiées uniquement à l'amélioration de MuseoToolbox. Les 4 semaines restantes qui étaient réservées à l'étude de la création d'un réseau de neurones avec une telle densité de données, se sont avérées insuffisantes au vu de la tâche à accomplir. A l'avenir ce type d'étude pourrait constituer un atelier à part entière.

BIBLIOGRAPHIE

- ✓ **Filtres spatiaux :**

https://openclassrooms.com/fr/courses/5060661-initiez-vous-aux-traitements-de-base-des-images-numeriques/5217251-analysez-le-filtrage-spatial-et-la-convolution-par-masque
https://github.com/mfauvel/RS_Lab/blob/master/labworks_s8.org
https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.median_filter.html
https://towardsdatascience.com/image-filters-in-python-26ee938e57d2
https://github.com/shreyapamecha/User-Defined-function-for-Spatial-Filtering-/blob/master/Spatial_Filtering.py

- ✓ **Multi-processing :**

https://pypi.org/project/multicore/
http://www.celeryproject.org/
https://joblib.readthedocs.io/en/latest/
https://wltrimbl.github.io/2014-06-10-spelman/intermediate/python/04-multiprocessing.html
https://towardsdatascience.com/a-hands-on-guide-to-multiprocessing-in-python-48b59bfcc89e
https://joblib.readthedocs.io/en/latest/
https://docs.python.org/fr/3/library/multiprocessing.html
https://museotoolbox.readthedocs.io/en/latest/
https://stackoverflow.com/

- ✓ **Réseau de neurones :**

https://www.youtube.com/watch?v=R8IVMuZ_BEE
https://techwithtim.net/tutorials/python-neural-networks/text-classification-p1/
https://medium.com/france-school-of-ai/math%C3%A9matiques-des-r%C3%A9seaux-de-neurones-code-python-613d8e83541
https://lesdieuxducode.com/blog/2019/1/prototyper-un-reseau-de-neurones-avec-keras
https://keras.io/getting-started/functional-api-guide/
https://blog.octo.com/classification-dimages-les-reseaux-de-neurones-convolutifs-en-toute-simplicite/
http://penseeartificielle.fr/tp-reseau-de-neurones-convolutifs/
https://towardsdatascience.com/build-your-own-convolution-neural-network-in-5-mins-4217c2cf964f
https://www.learnopencv.com/neural-networks-a-30000-feet-view-for-beginners/
https://busy.org/@rerere/comment-fonctionne-un-reseau-de-neurones-inception-v3-4

https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

https://hackernoon.com/neural-networks-introduction-6048f69b68b0

https://meritis.fr/ia/deep-learning/
https://github.com/tensorflow/models/blob/master/research/inception/README.md

https://cloud.google.com/tpu/docs/inception-v3-advanced

The Hundred-Page Machine Learning Book en français, A. Burkov, 2019, A. Burkov