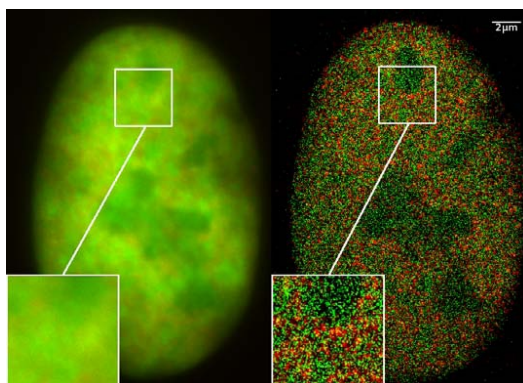


ECOLE POLYTECHNIQUE DE L'UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS  
Département Informatique  
64 avenue Jean Portalis  
37200 Tours, France  
Tél. +33 (0)2 47 36 14 14  
[polytech.univ-tours.fr](http://polytech.univ-tours.fr)

## Projet Recherche & Développement 2018-2019

# Problème de super-résolution par application de CNN



**Tuteurs académiques**  
Donatello CONTÉ  
Maxime MARTINEAU

**Étudiant**  
Vincent LIUZZI (DI5)



# Liste des intervenants

Nom	Email	Qualité
Vincent LIUZZI	<a href="mailto:vincent.liuzzi@etu.univ-tours.fr">vincent.liuzzi@etu.univ-tours.fr</a>	Étudiant DI5
Donatello CONTÉ	<a href="mailto:donatello.conté@univ-tours.fr">donatello.conté@univ-tours.fr</a>	Tuteur académique, Département Informatique
Maxime MARTINEAU	<a href="mailto:maxime.martineau@univ-tours.fr">maxime.martineau@univ-tours.fr</a>	Tuteur académique, Département Informatique



# Avertissement

Ce document a été rédigé par Vincent Liuzzi susnommé l'auteur.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Donatello Conté et Maxime Martineau susnommés les tuteurs académiques.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable des tuteurs académiques et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



## Pour citer ce document

Vincent Liuzzi, *Problème de super-résolution par application de CNN*, Projet Recherche & Développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

```
@mastersthesis{
  author={Liuzzi, Vincent},
  title={Problème de super-résolution par application de CNN},
  type={Projet Recherche \& Développement},
  school={Ecole Polytechnique de l'Université François Rabelais de Tours},
  address={Tours, France},
  year={2018-2019}
}
```

# Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	vi
<b>1 Introduction</b>	<b>1</b>
1 Contexte la réalisation.....	1
1.1 La super résolution .....	1
1.2 Contexte du projet .....	1
1.3 Objectifs .....	2
2 Description générale.....	3
2.1 Environnement du projet .....	3
2.2 Caractéristiques des utilisateurs .....	3
2.3 Fonctionnalités et du système .....	3
2.4 Structure générale du système .....	3
<b>2 État de l'art</b>	<b>6</b>
1 Le Deep Learning .....	6
1.1 Réseau de neurone .....	7
1.1.1 Principe.....	7
1.1.2 Entraînement .....	8
1.1.3 Problèmes liés à l'entraînement .....	8

1.2	Réseau de neurone à convolution (CNN) .....	8
2	Protocole expérimental .....	9
3	Super résolution en temps réel en utilisant un CNN à "sous-pixel" (ESPCN) [2].....	10
3.1	Principe général .....	10
3.2	Jeu de données .....	11
3.3	Conclusion .....	11
4	Super résolution avec réseau spatio-temporels et compensation de mouvement (VESPCN) [3] .....	11
4.1	Principe général .....	11
4.2	Jeu de données .....	13
4.3	Conclusion .....	13
5	Révélateur de détail (DETAIL) [1].....	14
5.1	Principe général .....	14
5.2	Jeu de données .....	16
5.3	Conclusion .....	16
6	Comparaison des méthodes .....	16
6.1	Tableau comparatif .....	16
6.2	Choix final .....	18
<b>3</b>	<b>Analyse du système</b> .....	<b>19</b>
1	Modèle ESPCN .....	19
1.1	Jeu de données .....	19
1.2	Premier module : Prétraitement des données .....	19
1.3	Deuxième module : Construction du modèle .....	19
1.4	Troisième module : Post-traitement .....	20
2	Modèle VESPCN.....	20
2.1	Jeu de données .....	20
2.2	Construction du modèle .....	21
<b>4</b>	<b>Mise en oeuvre</b> .....	<b>22</b>
1	Installation de l'environnement .....	22
1.1	Anaconda .....	22
1.2	TensorFlow et Keras.....	22
1.3	Rodeo .....	22
1.4	PyCharm .....	23
2	Pré processing.....	23
2.1	Base de données test .....	23
2.2	Séquençage des vidéos.....	23
2.3	Extraction des patches.....	24
2.4	Sauvegarde en physique ou dans la mémoire vive? .....	24

3	Définition d'une architecture de fichiers.....	24
4	Fonctionnement de l'entraînement .....	24
5	Séparation du dataset .....	25
6	Générateur .....	25
7	Évaluation du modèle.....	27
8	Entraînement, validation, évaluation.....	27
8.1	Entraînement et validation.....	27
8.2	Évaluation .....	28
9	Organisation du code .....	28
9.1	Modèle général et modèles dérivés .....	28
9.2	Gestion des paramètres .....	28
10	Utilisation du réseau et résultats obtenus.....	28
10.1	Utilisation du réseau.....	28
10.2	Résultats obtenus .....	29
<b>5</b>	<b>Bilan et conclusion S9</b>	<b>31</b>
1	Tâches faites.....	31
2	Retards .....	31
3	Tâches à faire .....	32
3.1	Fin de l'analyse .....	32
3.2	Mise en place de l'environnement.....	32
3.3	Module ESPCN .....	32
3.4	Module VESPCN.....	32
<b>6</b>	<b>Bilan et conclusion du S10</b>	<b>33</b>
1	Retards et tâches non faites.....	33
2	Raisons du retard et difficultés .....	33
3	Bilan sur ce qui fonctionne.....	34
4	Conclusion générale .....	34
	<b>Annexes</b>	<b>35</b>
<b>A</b>	<b>Gestion de projet</b>	<b>36</b>
1	Tâches à faire au semestre 9 .....	36
1.1	Tâche 1 : Recherches générales.....	36
1.2	Tâche 2 : Tutoriels sur la classification d'image .....	37
1.3	Tâche 3 : Préparation de la présentation & présentation.....	37
1.4	Tâche 4 : Lecture et étude des articles.....	37
1.5	Tâche 5 : Rédaction du rapport .....	37
1.6	Tâche 6 : Préparation de la soutenance et soutenance.....	37

2	Tâches à faire au semestre 10 .....	38
2.1	Tâche 7 : Fin de l'analyse .....	38
2.2	Tâche 9 : Mise en place de l'environnement de développement .....	38
2.3	Tâche 10 : Construction du modèle ESPCN.....	38
2.4	Tâche 11 : Entraînement du modèle ESPCN.....	38
2.5	Tâche 12 : Validation et corrections du modèle ESPCN .....	38
2.6	Tâche 13 : Documentation d'utilisation ESPCN .....	38
2.7	Tâche 14 : Cahier de test ESPCN.....	39
2.8	Tâche 15 : Construction du modèle VESPCN .....	39
2.9	Tâche 16 : Entraînement modèle VESPCN .....	39
2.10	Tâche 17 : Validation et correction du modèle VESPCN .....	39
2.11	Tâche 18 : Complétion de la documentation d'utilisation .....	39
2.12	Tâche 19 : Complétion du cahier de test.....	39
2.13	Tâche 20 : Rédaction du rapport .....	40
2.14	Tâche 21 : Préparation de la soutenance et passage. ....	40
3	Décalage du planning.....	40
<b>B</b>	<b>Description des interfaces</b> .....	<b>42</b>
1	Interfaces matériel/logiciel .....	42
2	Interfaces homme/machine .....	42
3	Interfaces logiciel/logiciel.....	42
<b>C</b>	<b>Spécifications fonctionnelles</b> .....	<b>43</b>
1	Test & validation.....	43
2	Transformation des vidéos en basse résolution .....	43
3	Base de données et jeu de données.....	43
4	Entrées et sorties du système .....	43
<b>D</b>	<b>Spécifications non fonctionnelles</b> .....	<b>44</b>
1	Contraintes de développement et conception .....	44
2	Contraintes de fonctionnement et d'exploitation .....	44
3	Controlabilité.....	44
<b>E</b>	<b>Guide d'installation et d'utilisation</b> .....	<b>45</b>
1	Installation.....	45
1.1	Python .....	45
1.2	Anaconda .....	45
1.2.1	Téléchargement.....	45
1.2.2	Variables d'environnement.....	45
1.3	Tensorflow et Keras .....	46



1.3.1	Version CPU.....	46
1.3.2	Version GPU.....	46
1.4	Environnement de développement .....	47
2	Utilisation .....	47
2.1	Arborescence de fichiers.....	47
2.2	Paramètres par défaut.....	48
2.3	Séquençage des vidéos.....	48
2.4	Extraction des patches.....	48
2.5	Entraînement du réseau.....	49
<b>F</b>	<b>Guide développeur</b>	<b>50</b>
	<b>Bibliographie</b>	<b>56</b>

# Table des figures

## 1 Introduction

1	Illustration de la détection de fréquence cardiaque par vidéo.....	2
2	Diagramme de cas d'utilisation pour la partie test et validation.....	4
3	Diagramme de cas d'utilisation pour la partie entraînement .....	4

## 2 État de l'art

1	Schéma du contexte dans lequel est inscrit le deep learning .....	6
2	Schéma d'un réseau de neurone simple.....	7
3	Schéma expliquant la convolution .....	9
4	Schéma de l'architecture du modèle ESPCN. ....	10
5	Schéma de l'architecture du modèle VESPCN.....	12
6	Schéma illustrant la fusion précoce .....	12
7	Schéma illustrant la compensation de mouvement.....	13
8	Résultat de la compensation de mouvement : carte d'erreur .....	14
9	Fonctionnement du zero-upsampling.....	15
10	Schéma de l'architecture du modèle DETAIL.....	15
11	Tableau comparatif de 4 méthodes de super résolution selon 7 critères. ....	16

## 3 Analyse du système

1	Tableau de conversion des pixels du canal r dans l'image HR finale.....	20
---	---	----

## 4 Mise en oeuvre

1	Interface de l'IDE RODEO .....	23
2	Architecture des fichiers sur le disque.....	25

3	Code python du générateur.....	26
4	Code python de la fonction s'occupant du training.....	27
5	Code python de la classe Parameters. ....	29
6	Graphes des métriques en fonction des époques (pour 10 époques) .....	30
 <b>A Gestion de projet</b>		
1	Diagramme de Gantt des tâches réalisées au S9.....	36
2	Diagramme de Gantt complet. Tâches réalisées au S9 et prévisionnelles au S10.....	41

# 1

## Introduction

Ce document contient les spécifications système concernant le Projet de Recherche et de Développement n°10 : "Problème de super résolution par application de CNN" ainsi que les recherches concernant l'état de l'art de la super-résolution.

Il contiendra dans une première partie les différentes caractéristiques et spécifications du projet : contexte, environnement, fonctionnalités et spécifications du système. Dans une seconde partie, nous retrouverons l'état de l'art et enfin l'analyse du système.

Ce projet étant plutôt exploratoire, la partie recherche sera donc plus importante que la partie développement.

Proposé par l'équipe Reconnaissance de Forme et Analyse d'Images (RFAI), le projet est encadré par Donatelo Conté et Maxime Martineau. Le projet sera réalisé par Vincent Liuzzi.

## 1 Contexte la réalisation

### 1.1 La super résolution

Obtenir une vidéo en haute résolution est parfois compliqué car le matériel nécessaire n'est pas toujours disponible. En effet, les objectifs et capteurs haute résolution ont souvent un coût très élevé. Pour contourner ce problème, on peut utiliser des algorithmes qui vont augmenter la taille et la qualité de l'image : **La super résolution**.

On trouve de nombreux domaines d'application comme la surveillance par exemple. Les caméras de sécurité ne filment généralement pas en très haute résolution. Cela reste utile pour observer la globalité d'une scène mais il peut être très difficile de reconnaître des détails comme un visage ou une plaque d'immatriculation.

La super résolution trouve aussi son utilité dans l'astronomie où l'acquisition des corps célestes n'est pas toujours aisée compte tenu de la distance nous séparant de ceux-ci, ainsi que de la qualité des appareils n'étant pas suffisante pour observer clairement les astres sans traitement.

### 1.2 Contexte du projet

Cela aura pour but, à terme, de détecter les émotions d'une personne via la webcam d'un ordinateur. On pourra faire cela grâce à la détection de la dilatation de la pupille.

On trouve déjà des programmes qui font cela grâce aux expressions du visage, mais ce n'est pas toujours précis et peut-être faussé. Un sourire peut évoquer de la joie mais certaines personnes peuvent sourire nerveusement lorsqu'elles ont peur par exemple. De plus, si des gens participent à une expérience de détection d'émotion, tous ne seront pas naturels et vont exagérer ou contrôler leurs expressions faciales. Il est donc plus précis de se baser sur des caractéristiques physiologiques que l'on ne peut contrôler comme la température du corps, la transpiration ou le rythme cardiaque.

Il existe des techniques pour capter des émotions via ces caractéristiques comme le grossissement de certaines zones de la vidéo pour capter le rythme cardiaque. Par exemple, on peut grossir la vidéo au niveau des veines ou du ventre pour capter la fréquence cardiaque ou encore exagérer les couleurs du visage car, quand on respire, les couleurs du visage changent de manière minime dû à l'afflux sanguin qui remonte dans le cerveau.

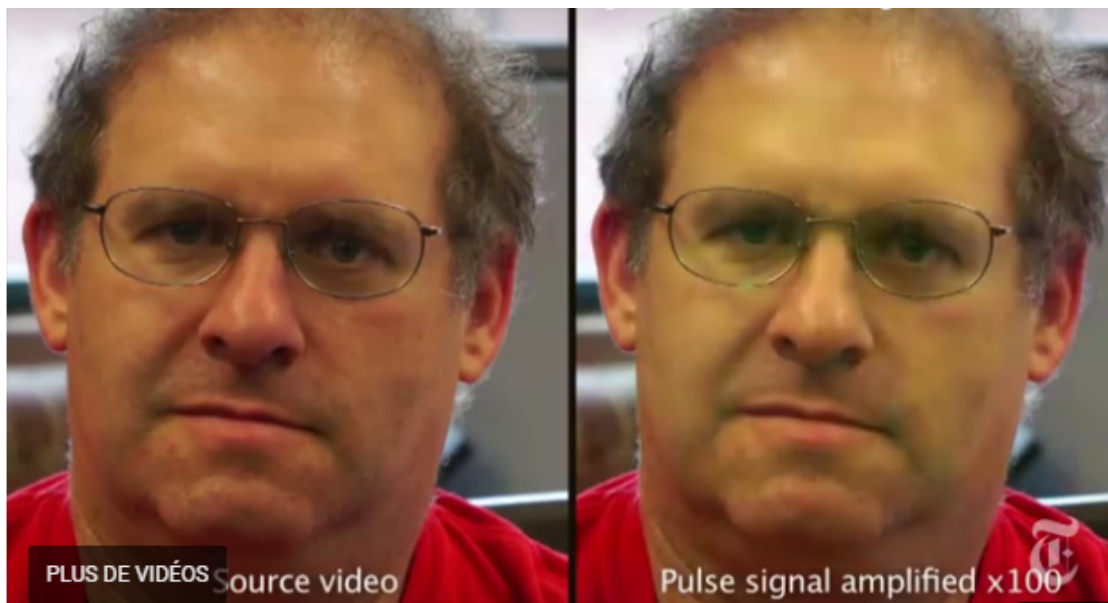


Figure 1 – Illustration de la détection de fréquence cardiaque par vidéo.

Une webcam classique étant souvent de faible qualité et les dilatations de la pupille étant minimales, il nous faudra donc augmenter la qualité de la vidéo pour pouvoir zoomer sur les yeux de la personne.

### 1.3 Objectifs

Malgré le contexte entourant ce projet, son objectif final n'est pas de pouvoir détecter telle ou telle émotion. L'objectif est de convertir une vidéo d'entrée basse résolution en vidéo haute résolution (1920x1080 environ).

La grosse partie de ce projet est donc de trouver et implémenter un algorithme permettant d'effectuer ceci. Cet algorithme correspond en fait à un modèle du réseau de neurones que l'on va utiliser. Cette partie sera développée dans la partie 2. **Etat de l'art.**

## 2 Description générale

### 2.1 Environnement du projet

Pour ce projet, il n'y a pas d'existant à proprement parler. Il faudra créer et implémenter un nouveau modèle de réseau de neurones, tester et évaluer les résultats soi-même.

Cependant, le modèle va être repris de méthodes déjà existantes. De même, il sera plus simple de prendre les mêmes jeux de données déjà utilisés pour comparer les résultats obtenus.

### 2.2 Caractéristiques des utilisateurs

Il n'y a pas d'utilisateur cible visé. On peut considérer que les utilisateurs seront des scientifiques. Ils connaîtront donc le contexte d'utilisation et seront familiers avec l'environnement, la technologie utilisée ainsi que les protocoles nécessaires à l'exécution de l'algorithme.

### 2.3 Fonctionnalités et du système

Pour notre système, nous pouvons séparer les fonctionnalités principales en deux parties : La partie de test et la partie d'entraînement du réseau de neurones. Bien évidemment, il est difficile d'évoquer cette dernière n'ayant pas encore développé ce qu'était un réseau de neurones et en quoi il consiste.

La partie la plus simple est la validation. Voici le diagramme de cas d'utilisation associé :

Les tâches sont assez simplistes car l'utilisateur n'a pas une place importante dans ce projet. Tout ce que l'utilisateur va faire une fois que l'algorithme est construit et le réseau de neurones en place, c'est lancer le programme, ce qui correspond à charger l'algorithme et récupérer les résultats. Après cela, il va devoir évaluer les résultats puis le valider en fonction du protocole de validation mis en place. Ces deux derniers points seront développés dans la partie **Spécifications fonctionnelles**.

Concernant la partie *entraînement*, il faut savoir que pour qu'un réseau neuronal soit fonctionnel, il faut l'entraîner. Voici comment cela se présente sous forme de diagramme :

L'utilisateur va choisir les poids des noeuds du réseau de neurones (Voir la partie 2. **État de l'art : Introduction sur les CNN**), choisir le jeu de données qui va être utilisé (les nombreuses vidéos basses résolutions à transformer en vidéos hautes résolutions). Une fois la "préparation" terminée, il ne reste plus qu'à lancer l'algorithme d'entraînement.

### 2.4 Structure générale du système

Le système va être séparé en plusieurs modules différents qui vont correspondre à des étapes. Tout d'abord, il va falloir préparer les données. En effet, comme cela va être développé plus tard, nous travaillons avec des vidéos, mais ces vidéos vont devoir être transformées en séquences d'images.

Le module central va être l'algorithme en lui-même qui peut être séparé en plusieurs sous modules qui seront détaillés dans la partie **Analyse et conception** :

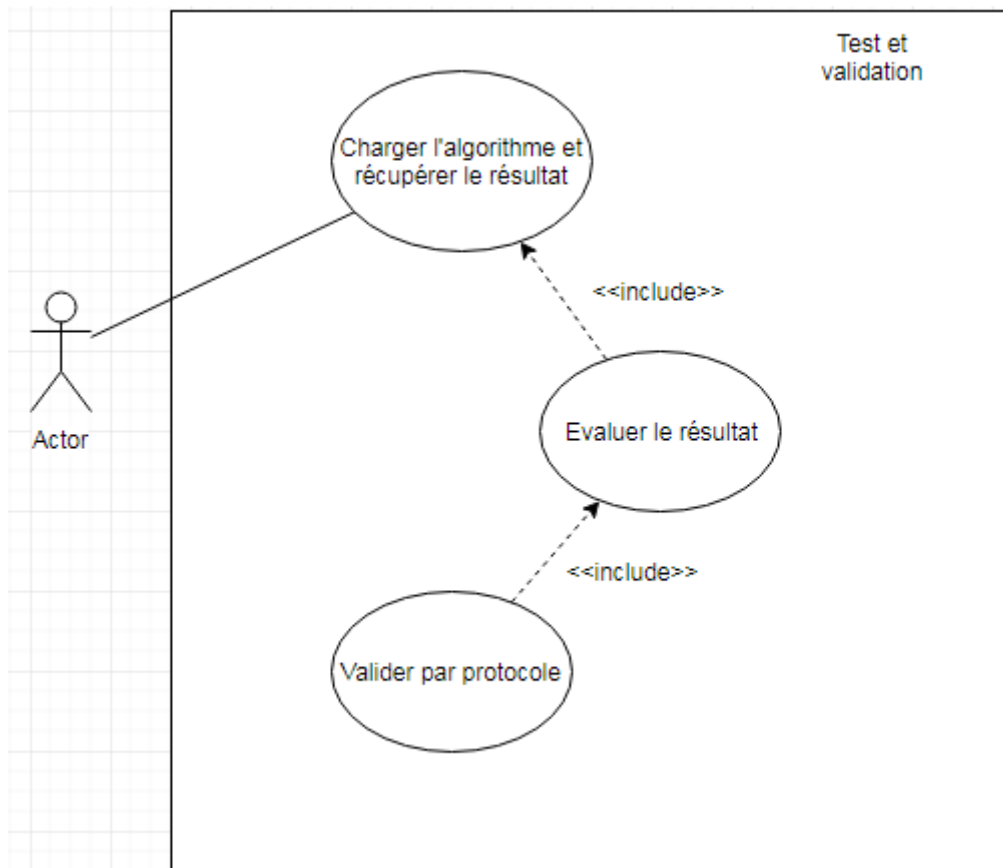


Figure 2 – Diagramme de cas d'utilisation pour la partie test et validation

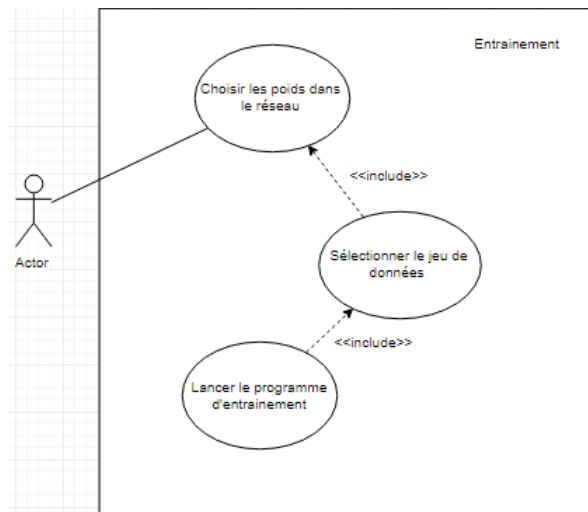


Figure 3 – Diagramme de cas d'utilisation pour la partie entraînement

- estimation de mouvement
- compensation de mouvement
- module de convolution sous-pixel adapté à la temporalité

Enfin, le dernier module sera du post-processing, qui servira à reconstruire la vidéo générée à partir de séquences d'images.

Le système va se découper en deux parties majeures. Comme la partie **Analyse** va l'expliquer, nous allons implémenter deux méthodes qui sont étudiées dans la partie **Etat de l'art**. D'abord,

nous allons élaborer le module de convolution sous pixel, appelé **ESPCN** par la suite. Il sera entraîné et testé. Une fois qu'il sera fonctionnel, nous allons les modules concernant le mouvement et la temporalité pour créer le système final, la méthode **VESPCN** qui utilise par conséquent le module **ESPCN**.

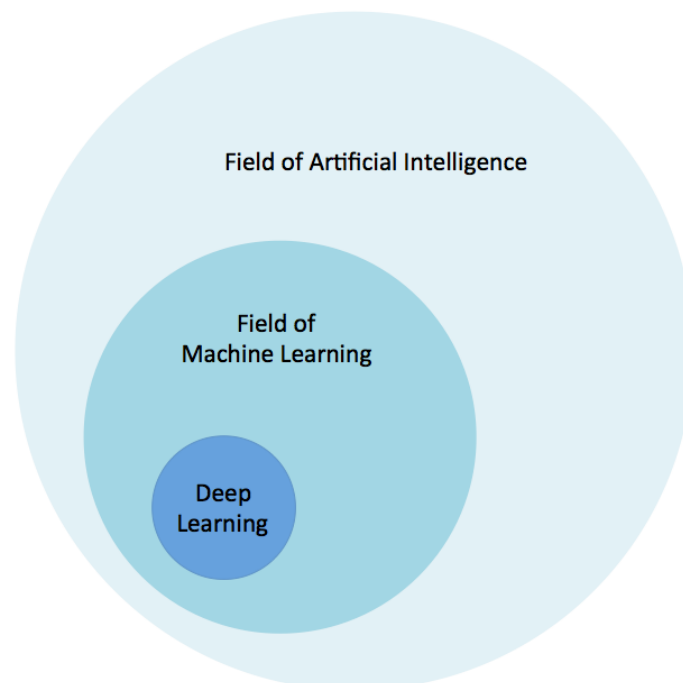


# 2

## État de l'art

### 1 Le Deep Learning

Le Deep Learning, ou "apprentissage profond" en français, est l'une des principales techniques utilisées par l'intelligence artificielle. Il est dérivé du Machine Learning (apprentissage automatique) qui permet à un ordinateur d'apprendre sans avoir été programmé explicitement à cet effet.



**Figure 1** – Schéma du contexte dans lequel est inscrit le deep learning

Il y a de très nombreux domaines où le Deep Learning est utilisé. On le retrouve par exemple dans les différents systèmes de reconnaissance faciale ou de traduction en temps réel, dans Google Photos qui est capable de reconnaître des objets sur vos photos, dans les voitures auto-

nomes ou la robotique en général.

C'est un concept qui émerge depuis les années 2000 mais le principe sur lequel se base le Deep Learning (les neurones artificiels) ne date pas d'aujourd'hui.

En 1943, deux chercheurs publient leur modèle du **neurone formel**, directement inspiré du neurone biologique. C'est une entité qui prend une ou plusieurs entrées, effectue un traitement et produit une sortie.

En 1957 est inventé le **perceptron**, le début des réseaux de neurones que nous allons détailler dans la partie suivante.

On voit en 1986 l'apparition des **perceptrons multicouches**, qui est une évolution des principes précédents et qui se rapproche de ce que nous connaissons du Deep Learning.

## 1.1 Réseau de neurone

### 1.1.1 Principe

Un réseau de neurones est un système qui se base sur le fonctionnement des neurones du cerveau humain. Un neurone isolé n'est pas capable d'effectuer un grand calcul ou effectuer une tâche. C'est lorsqu'on regroupe les neurones et qu'on les connecte entre eux d'une certaine façon que les résultats sont convaincants. Voici comment se présente un réseau de neurones simple (ou un perceptron multicouches)

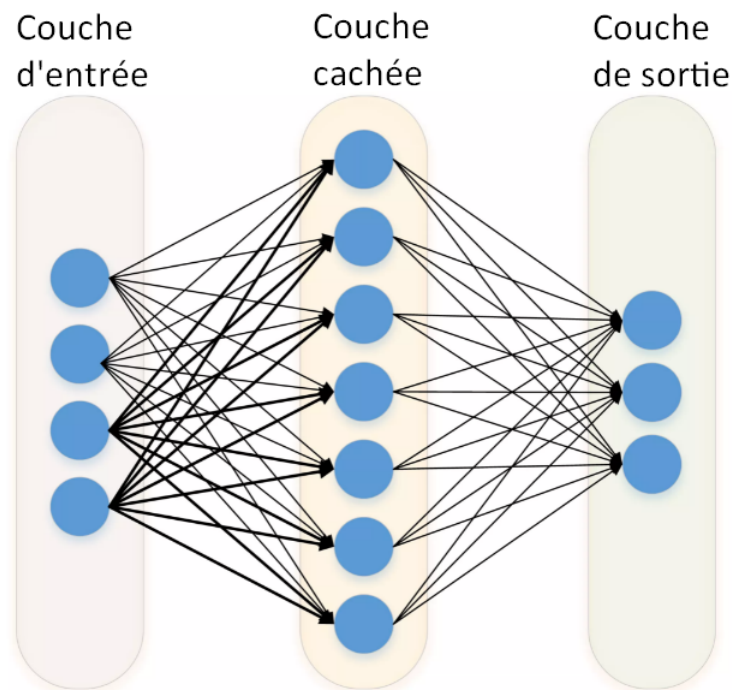


Figure 2 – Schéma d'un réseau de neurone simple

Tout d'abord, il faut préciser que tous les neurones sont organisés en "couches" et chaque couche a son utilité. Précédemment, j'ai utilisé le terme "**modèle**". Un modèle est en fait la façon dont est construit le réseau.

On retrouve la **couche d'entrée** qui est la première couche de notre réseau. C'est d'ici que partent les données de départ que l'on veut analyser (du texte ou une image par exemple).

La deuxième couche est la **couche cachée**. Elle n'a une utilité que pour le réseau et n'a pas

de contact direct avec l'extérieur. Elle reçoit des données de la couche d'entrée, effectue un traitement mathématique et renvoie des données à la couche de sortie.

Cette dernière donne le résultat obtenu par le réseau en fonction de ses manipulations effectuées.

Pour considérer un réseau d'apprentissage **profond**, il faut que ce réseau possède au moins deux couches cachées.

### 1.1.2 Entraînement

Un concept important du Deep Learning est l'entraînement ou l'apprentissage. En effet, pour que notre réseau soit fonctionnel et produise de bons résultats, il faut lui apprendre comment faire. Pour cela, on va lui donner un très grand nombre de données d'entrée et lui dire le résultat attendu. À l'instar du principe général du Deep Learning, l'entraînement fonctionne de la même manière que l'humain lorsqu'il apprend. Si on veut apprendre à un très jeune enfant ce qu'est un chat, on ne va pas lui décrire toutes les caractéristiques qui font que c'est un chat. Lorsqu'il va croiser des chats, ses parents lui diront "ceci est un chat". Lorsque l'enfant va avoir croisé un grand nombre de chat en sachant d'avance que c'en est un, il sera capable d'en reconnaître tout seul. C'est ainsi que fonctionne aussi l'entraînement d'un réseau neuronal artificiel.

### 1.1.3 Problèmes liés à l'entraînement

Il faut cependant faire attention au jeu de données qu'on lui fournit. En effet si le jeu de données est trop petit, le réseau commettra beaucoup d'erreur aussi bien en phase d'apprentissage qu'en phase d'utilisation. On appelle ça l'**underfitting**. Il ne faut pas non plus un trop grand jeu de données sinon le modèle sera effectivement très performant en phase d'apprentissage, mais lors de l'utilisation réelle, il aura du mal à se généraliser et à fournir des prédictions correctes face à des données qu'il n'a jamais rencontrées. C'est l'**overfitting**. Un des enjeux de l'apprentissage est donc de trouver un juste milieu dans le jeu de données que l'on fournit au modèle.

## 1.2 Réseau de neurone à convolution (CNN)

Nous avons parlé dans la partie précédente d'image, mais pour en traiter, il est largement préférable d'utiliser un réseau de neurones à convolution. Ceci fonctionne exactement comme un réseau de neurones classique sauf qu'on trouvera à l'intérieur une ou plusieurs couches de convolution.

Pour définir ce qu'est la convolution, il faut savoir qu'une image est représentée en entrée par une matrice en 2 dimensions pour une image en niveau de gris. La couleur est représentée par une troisième dimension de profondeur 3 pour représenter les 3 couleurs de base : Rouge, Vert et Bleu. Ces trois dimensions sont aussi appelés **canaux**.

La convolution est un outil mathématique qui agit un peu comme un filtrage. On définit une taille **n** au début, et cela va correspondre à une fenêtre qui va se déplacer à travers l'image et effectuer des opérations sur les différentes zones de l'image.

Dans chaque zone, le filtre va appliquer une opération avec les  $n \times n$  pixels que la fenêtre a sélectionnés, comme par exemple choisir le pixel avec la plus grande valeur ou une moyenne de ces pixels. La dimension de l'image résultante de cette couche aura donc une taille plus

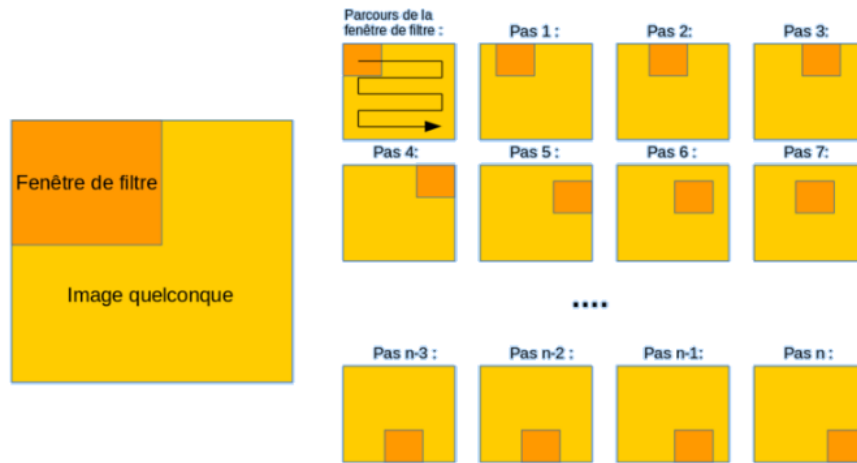


Figure 3 – Schéma expliquant la convolution

petite que l'originale. Cette sortie est aussi appelée **carte des caractéristiques** (ou feature maps).

Nous avons vu le principe global du deep learning. Mais pour aborder un problème de super résolution, le modèle est bien plus compliqué. Nous allons étudier cela à travers 3 articles scientifiques.

## 2 Protocole expérimental

Nous retrouvons un point commun avec la majorité des méthodes de super-résolution : Le protocole expérimental de test et d'évaluation.

### Evaluation

Le principe pour évaluer un algorithme de résolution est simple. Nous cherchons à savoir si l'augmentation en résolution est cohérente et efficace. Nous allons donc prendre des vidéos ou images HR et en créer des **copies BR** en baissant donc la résolution. Ce sont ces copies BR que nous allons passer dans l'algorithme. Le résultat de l'algorithme va être comparé avec l'image ou la vidéo **originale** pour étudier la différence.

La différence est souvent exprimée en **PSNR**. C'est une mesure qui permet d'évaluer la différence pixel par pixel de deux images, elle est exprimée en décibels. Plus la valeur du PSNR est élevée, plus l'image / vidéo créée est proche de l'original. On trouve aussi une différence exprimée en **SSIM**, qui elle évalue la similarité de structure entre deux images. Cette valeur est comprise entre 0 et 1, plus la valeur est proche de 1, plus les images sont similaires.

Cependant ces mesures **objectives** ne reflètent pas forcément la réalité ni la "beauté" d'une image et peuvent ne pas correspondre à la perception humaine. En effet, il est possible de retrouver deux PSNR très éloignées mais des différences presque indiscernables à l'oeil nu. De même, il est aussi probable d'avoir deux images avec le même PSNR et où il y a une différence de qualité très évidente pour l'oeil humain.

### Préparer le jeu de données

Pour entraîner le réseau, il faut bien construire et choisir son jeu de données. Pour de la super-résolution vidéo, le plus courant est de prendre un certain nombre de vidéos et d'en extraire des **séquences** plus ou moins longues.

Ces séquences sont en fait une suite de 10,20,30 ou plus images consécutives qui sont extraites

de la vidéo. Le plus intéressant est de choisir une séquence où il y a un mouvement. Comme pour un réseau de neurones en général, il faut bien choisir le nombre de séquences à extraire ainsi que le nombre d'images de chaque séquence pour éviter l'underfitting ou l'overfitting.

### 3 Super résolution en temps réel en utilisant un CNN à "sous-pixel" (ESPCN) [2]

#### 3.1 Principe général

Pour résoudre un problème de super résolution avec un réseau de neurones, il faut déjà savoir ce qu'on trouvera en entrée et en sortie du réseau. Nous retrouverons en entrée une vidéo **basse résolution BR** que nous voulons transformer en vidéo **haute résolution HR**.

Dans les techniques "classiques" de super résolution, l'augmentation de résolution se faisait plus ou moins progressivement. Cela a pour incidence qu'une grande partie des opérations et des calculs se faisait dans un espace HR ce qui est contre performant. En effet, il est plus simple et moins coûteux de travailler sur une matrice  $256 \times 256$  que  $1000 \times 1000$  par exemple. Le principe de base ici est donc de passer en HR **uniquement à la toute fin du réseau** afin d'effectuer les opérations dans un espace en BR et améliorer grandement les performances.

Ainsi, l'extraction des caractéristiques (abordé dans la partie précédente) se produit dans un espace basse résolution. Nous nous retrouvons donc avec des filtres de plus petite taille pour regrouper les mêmes informations qu'avec une image et un filtre de taille supérieure. Cela signifie donc que les temps de calcul et la complexité sont grandement réduits.

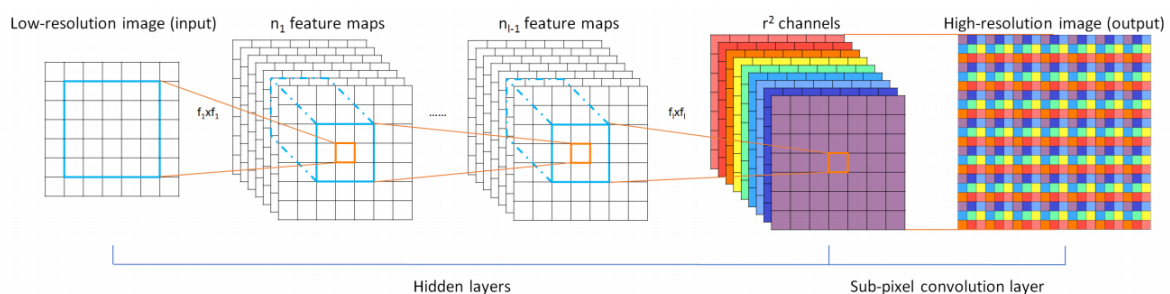


Figure 4 – Schéma de l'architecture du modèle ESPCN.

Il nous reste donc la dernière partie et la plus importante à aborder : la couche de convolution de sous-pixel. En effet, c'est cette dernière couche du réseau qui va se charger de la transformation en haute résolution. Elle produit une image en haute résolution à partir de plusieurs cartes de caractéristiques BR avec un seul filtre de mise à l'échelle par carte. Ces cartes de caractéristiques sont dans plusieurs canaux et la dernière étape va consister en l'agrégation de tous ces canaux en une image de sortie HR. Les canaux sont en fait considérés comme des **sous pixels** qui vont venir s'assembler et former l'image finale.

On remarque sur le schéma qu'il y a  $r^2$  canaux.  $r$  est le facteur par lequel on veut augmenter la résolution. Si l'on veut multiplier la résolution de notre vidéo par 3, alors nous aurons 9 canaux en parallèle (comme sur le schéma). Les canaux sont donc en BR et chaque pixel de ces canaux va venir former un sous pixel de l'image HR.

L'entraînement, lui, va consister en la génération d'une image LR à partir d'une image HR connue, puis nous allons calculer la marge d'erreur entre l'image HR de départ et l'image générée par le modèle. Ou plus précisément, l'erreur quadratique moyenne pixel par pixel va

être calculée en tant que fonction objective, c'est-à-dire une fonction qui va servir de critère pour déterminer la meilleure optimisation.

### 3.2 Jeu de données

Le jeu de données utilisé pour la partie image provient de bases de données publiques avec en tout plus de 50 000 images d'entraînement et environ 300 pour le test. Concernant la vidéo, il est utilisé 15 vidéos en 1920x1080 de 5 et 10 secondes.

### 3.3 Conclusion

Cette méthode présente beaucoup d'avantages. Tout d'abord, il est proposé une toute nouvelle couche changeant de ce qu'il se faisait précédemment en super-résolution. En effet, la première approche à laquelle on pourrait penser pour la super résolution aurait été une couche de **déconvolution** (ou convolution inverse) qui impliquerait donc d'avoir une sortie de plus grande taille qu'à l'entrée. Cependant, les coups de calculs sont assez importants.

Cette nouvelle couche permet donc d'accéder à un résultat qualitatif pour des coûts de calcul minimes par rapport à ce qui se faisait habituellement. La vitesse d'exécution est aussi 10 fois plus importante que les méthodes précédentes.

Bien que cette méthode semble convenable, nous allons voir par la suite qu'il existe d'autres moyens intéressants d'exploiter la vidéo lors d'un problème de super-résolution.

## 4 Super résolution avec réseau spatio-temporels et compensation de mouvement (VESPCN) [3]

### 4.1 Principe général

L'aspect que l'on peut reprocher à la méthode précédente et qu'elle traite la vidéo image par image. On appelle ça de la *"single image Super Resolution" (SISR)*. La vidéo est considérée comme une suite d'images indépendantes mises bout à bout. La méthode ici s'oppose à ce principe en se basant sur du *"multi-image Super Resolution"* qui considère qu'il y a plusieurs observations possibles d'une même scène. Passer de la super résolution image à la vidéo implique d'introduire une nouvelle dimension : **le temps**. Ceci permettrait d'améliorer grandement les performances.

Cependant, il est important de préciser que la méthode ESPCN traite de manière efficace les images. C'est seulement son extension aux vidéos qui pose problème. L'idée ici est donc de **combiner** l'efficacité des **convolutions sous pixel** avec la performance des réseaux spatio-temporels et de la **compensation de mouvement**.

Voici le schéma du modèle. Il est composé en deux parties : la partie estimation et compensation de mouvement et une partie de convolution sous pixel (vu dans l'analyse de la méthode précédente).

Dans les réseaux spatio-temporels, on suppose que les données d'entrée sont des blocs d'information. Ainsi, au lieu d'avoir une simple image BR, nous aurons une séquence d'images consécutives BR. Le but ici va être de traiter une image, avec son image suivante et son image précédente **compensée**. C'est ici que le module de compensation de mouvement entre en jeu.

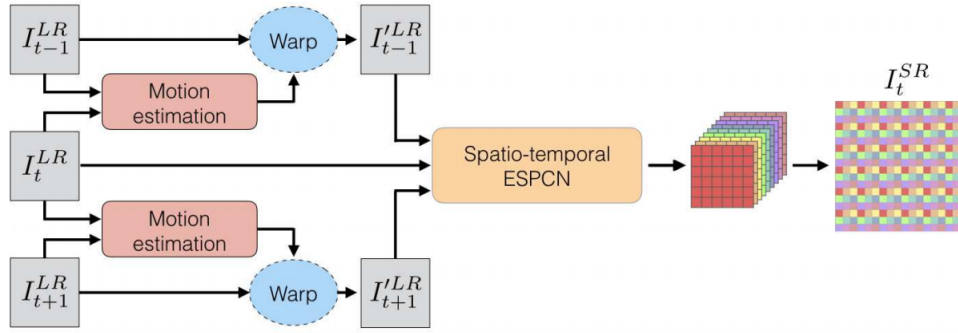


Figure 5 – Schéma de l'architecture du modèle VESPCN.

Avant cela, il se doit d'être introduit brièvement la notion de fusion précoce car elle sera abordée plus tard.

### Fusion précoce

Une fusion précoce consiste à rassembler toutes les images d'entrées (consécutives donc) en un même niveau de caractéristiques. On va donc avoir autant de canaux qu'il y a d'images d'entrée, comme dans le schéma suivant :

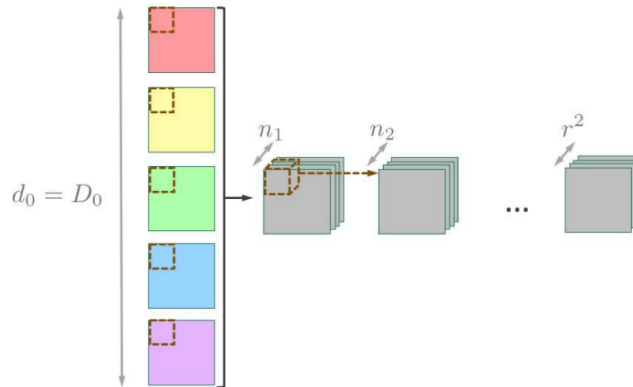


Figure 6 – Schéma illustrant la fusion précoce

$d_0$  correspond au nombre d'images d'entrée, et  $D_0$  à la profondeur temporelle du réseau. Cela signifie donc que dans le schéma, il y a 5 images consécutives qui vont être rassemblées en 5 canaux qui vont être traités par la suite comme si c'était une seule image avec plusieurs cartes des caractéristiques, comme on pourrait trouver dans un réseau de neurones de super résolution à image simple.

### Compensation de mouvement

Nous pouvons voir dans le schéma précédent que dans le module *compensation de mouvement*, nous avons deux mêmes processus qui travaillent avec seulement des données de départs différentes. Nous allons décortiquer ce processus à l'aide du schéma de la figure 7.

Dans la réalité, la compensation d'image va se faire par blocs de 3 images consécutives. Mais pour expliquer le principe plus simplement, nous allons considérer une image et sa suivante. On peut séparer cela en 4 parties ou étapes :



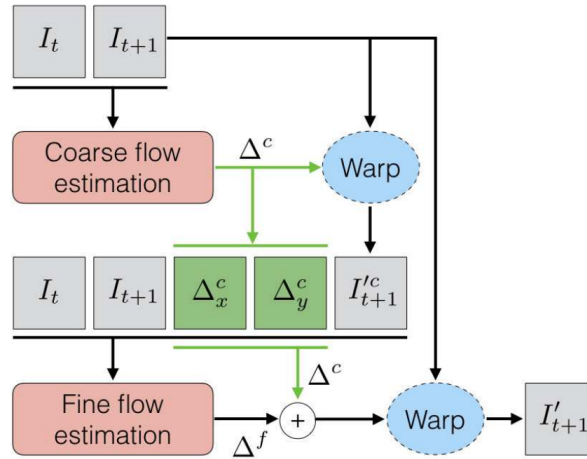


Figure 7 – Schéma illustrant la compensation de mouvement

1. Tout d'abord, nous allons estimer grossièrement le flux de données (bloc "Coarse flow estimation"). Pour cela, une **fusion précoce** entre l'image de référence  $I_t$  et l'image suivante  $I_{t+1}$  va être réalisée. Par la suite, l'image va subir 2 convolutions pour les réduire.
2. De cela ressort un flux estimé ( $\Delta^c$ ) qui va être agrandi par une convolution sous-pixel et le résultat de tout cela va être appliqué pour déformer l'image cible ( $I_{t+1}^c$ ).
3. Cette image déformée, l'estimation grossière et les deux images de départ vont passer à travers une estimation de flux plus fine. Cela se fait grâce une convolution par pas de 2 et une phase finale d'amélioration x2 de la résolution de l'image. Nous obtenons le flux  $\Delta^f$ .
4. L'image finale (donc l'image suivante compensée est obtenue en déformant l'image cible avec les deux flux provenant des deux estimations

## 4.2 Jeu de données

La base de données utilisée est la Consumer Digital Video Library (CDVL) qui contient 115 vidéos non compressées full HD. 100 de ces vidéos sont pour l'entraînement, et les 15 autres pour le test.

La résolution des vidéos est réduite et 30 échantillons aléatoires sont extraits de chaque paire de vidéos HR-BR pour obtenir 3000 échantillons d'entraînement, dont 5% sont utilisés pour la validation.

## 4.3 Conclusion

La compensation de mouvement est très efficace car elle permet d'obtenir de meilleurs résultats.

Dans l'encadré orangé, nous avons l'image précédente et suivante **sans** compensation. Les deux images à droite dans cet encadré représentent une carte d'erreur, qui est obtenue en soustrayant l'image suivante ou précédente avec celle de référence. On obtient donc l'écart entre ces images. Dans l'encadré bleu, nous avons la même chose mais **avec** compensation. On remarque facilement que les écarts sont bien moindres. C'est en fait comme si nous avions obtenu l'image  $I_{t-1/2}$  et  $I_{t+1/2}$  car nous avons une image intermédiaire qui permet de fluidifier le mouvement et augmenter la qualité.

Il est aussi démontré que pour des performances similaires, cette méthode utilise 20% d'opérations en moins que la méthode *ESPCN* précédente, et cela est dû au réseau spatio-temporel.

La combinaison de deux modules produit un résultat qui surclasse toutes les méthodes de super résolution précédentes car les vidéos sont de meilleure qualité et surtout plus cohérentes.



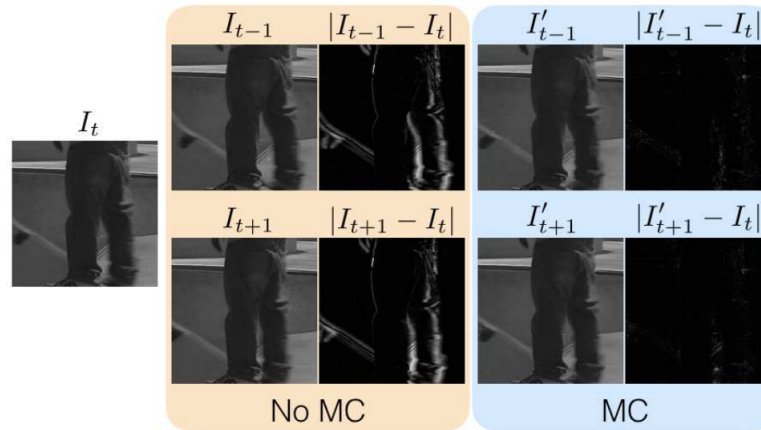


Figure 8 – Résultat de la compensation de mouvement : carte d'erreur

## 5 Révélateur de détail (DETAIL) [1]

Cette méthode a été imaginée après la précédente. Il y a d'ores et déjà des éléments de comparaison sur lesquels s'appuyer. Tout d'abord, ce qui n'a pas été dit c'est que les techniques de compensation telles que celle du VESPCN nécessitent un réglage au cas par cas des paramètres et un calcul lourd. En effet, la technique pour compenser les inter-images était d'aligner toutes les autres images à celle de référence, en utilisant la déformation inverse. Cependant, ce n'est pas optimal pour de la super résolution.

### 5.1 Principe général

Nous retrouverons 3 modules dans l'architecture. De l'**estimation de mouvement**, de la **compensation de mouvement** et enfin de la **fusion de détails**.

#### Estimation de mouvement

Estimer un mouvement avec un réseau de neurones n'est pas nouveau comme nous l'avons vu dans la méthode précédente. Le module que l'on va retrouver ici prendra deux images BR et créera un champ de mouvement. Ce module a déjà étudié car c'est le module d'estimation de mouvement de la méthode VESPCN qui est choisi pour construire ce modèle.

#### Couche SPMC (Sub-Pixel Motion Compensation)

Cette couche sert à construire des images HR à partir d'images BR, du flow obtenu par l'estimation de mouvement et d'un facteur d'augmentation de résolution. Elle contient deux sous modules. Le premier est un générateur de grille d'échantillonnage.

Pour rappel, le principe de la compensation de mouvement consiste à déplacer des pixels pour créer des images intermédiaires. Ce sous module va s'occuper de calculer ces nouvelles coordonnées en fonction du flux obtenu précédemment. Ces coordonnées transformées sont transposées dans une image agrandie. Ce processus est fait grâce au "zero upsampling". Il s'agit d'agrandir la matrice de l'image comme dans la figure 9.

Dans ce schéma, chaque pixel blanc est initialisé à 0 (d'où le **zero-upsampling**). Ici l'augmentation de résolution est de 2x. Nous avons donc 3/4 des pixels ayant 0 comme valeur.

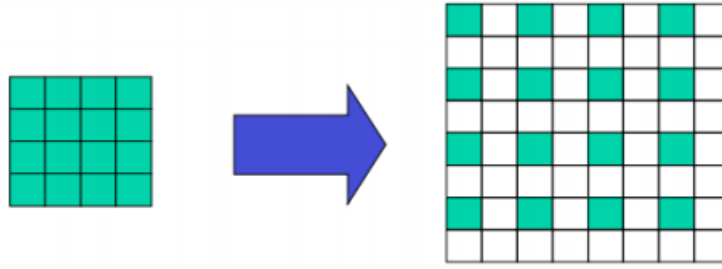


Figure 9 – Fonctionnement du zero-upsampling.

Nous avons ensuite le sous module d'échantillonneur d'image différentiable. La différentiabilité est un outil mathématique qui concerne les approximations linéaires et la dérivation. L'image de sortie est construite dans un espace plus grand et ce module va calculer, pour chaque pixel de sortie, la dérivée partielle du pixel d'entrée correspondante.

Dans ce module, la compensation de mouvement est faite en même temps que l'augmentation de résolution contrairement à la majorité des travaux précédents qui étaient faits en deux parties.

### Fusion de détails

La couche précédente produit donc une série d'images compensées. Cependant, ces images sont de grande taille, elles sont déjà en HR. En gardant la même logique que le principe de la méthode ESPCN, travailler sur de grands éléments n'est pas optimal. De plus, suite au zero-upsampling, nous nous retrouvons avec 15/16 des pixels à la valeur 0 (à cause de la multiplication de la résolution par 4). Voici le schéma du réseau pour clarifier les explications :

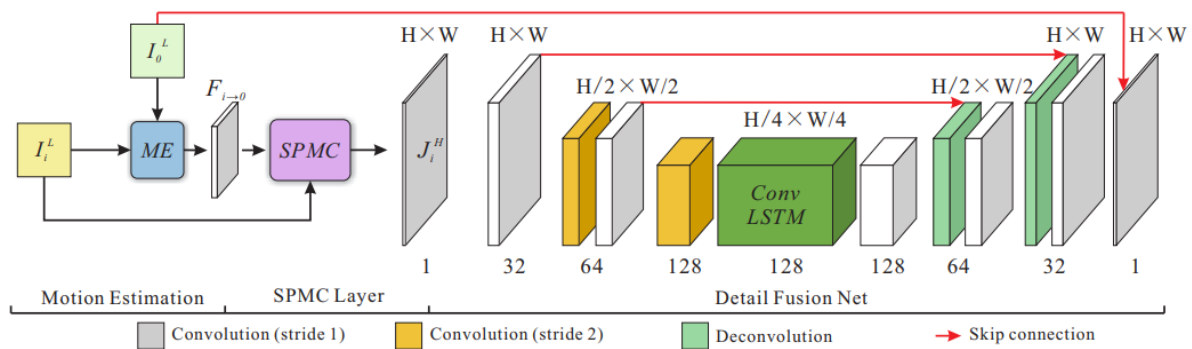


Figure 10 – Schéma de l'architecture du modèle DETAIL.

Les parties "Motion Estimation" et "SPMC Layer" ont été abordées dans les parties précédentes. Nous nous retrouvons dans la partie **Detail Fusion Net**. Tout d'abord, la taille de l'image est réduite par 4 grâce à deux convolutions. Les coûts de calcul sont ainsi réduits. Nous pouvons voir des flèches rouge **skip connection**. Elles servent à accélérer l'entraînement.

On retrouve au milieu un module ConvLSTM (résultat de la recherche suivante <https://bit.ly/2zD0AfZ>) qui, sans rentrer dans les détails, capture les corrélations spatio-temporelles de manière très efficace. Suite à cela nous aurons deux **déconvolutions** qui vont ramener les images BR bien traitées à la taille souhaitée (le quadruple ici). Nous remarquons sur le schéma des rectangles blancs après chaque convolution et déconvolution. Ceci correspond à un filtre d'activation **ReLU** (pour Rectified Linear Units).

Un **filtre d'activation** est quelque chose que l'on peut ajouter à la sortie d'une couche. Ce filtre en particulier applique une fonction  $\max(x, 0)$  sur chaque pixel d'input. Ce qui signifie

que chaque valeur négative vaudra maintenant 0 et chaque valeur positive ou nulle restera inchangée.

## 5.2 Jeu de données

Pour un entraînement propre, il faut que les données n'aient aucun bruit et soient de bonne qualité mais tout en restant riche en détail. Aucun jeu de données vidéo public n'étant disponible à ce jour, il a été collecté ici 975 séquences de vidéos **1080p HD**.

Les données originales sont réduites en **270x480, 180x320** et **135x240**. Enfin, parmi ces 975 séquences, 945 sont gardées pour l'entraînement et 30 pour le test et la validation.

## 5.3 Conclusion

Cette méthode exploite donc la temporalité couplée à une couche de convolution sous pixel et ajoute aussi un module intéressant de fusion de détail. Cette fusion de détail à partir de détails internes est plus efficace et plus sûre que la synthétisation et création de détails extérieurs car avec cette dernière, il est possible à la longue que des détails supplémentaires apparaissent même lorsque l'on utilise une seule image. Les temps de calcul restent corrects mais offrent une qualité de rendu supérieur à l'état de l'art actuel.

# 6 Comparaison des méthodes

## 6.1 Tableau comparatif

Afin de bien remarquer les différences entre les méthodes et ce qui peut les départager, j'ai décidé de réaliser un tableau comparatif. Nous comparerons donc les méthodes ESPCN, VESPCN et DETAIL. Nous incluons aussi une autre méthode : DRAFT. Elle n'a pas été analysée en détail car elle n'a pas été jugée la plus pertinente pour notre cas, mais elle apparaît dans ce tableau car elle comporte tout de même des avantages.

Critères \ Techniques	DRAFT	ESPCN	VESPCN	DETAIL
Temporalité	Estimation seulement	Non	Oui	Oui
Imbrication de méthode	Non	Non	Oui, module ESPCN	Oui, module EM VESPCN
Efficacité	++	+	+++	+++
Qualité ou vitesse ?	?	Vitesse	Qualité	Qualité
Disponibilité du code	Oui	Non	Non	Non
Jeu de données	Disponible avec le code	Publique	Publique	Jeu de données personnel
Technologie	Caffe	Inconnue	Inconnue	Tensorflow

Figure 11 – Tableau comparatif de 4 méthodes de super résolution selon 7 critères.

Pour commencer, les couleurs dans le tableau ont toutes un sens. Elles définissent l'intérêt des critères pour notre méthode et le poids qu'elles auront dans le choix final. L'échelle se constitue ainsi, en allant du moins au plus pertinent : **rouge**, **orange**, **jaune** et **vert**.

De plus, les méthodes sont rangées par ordre chronologique de sortie de l'article scientifique les concernant. La méthode *DRAFT* est parue en 2015, les méthodes *ESPCN* et *VESPCN* sont parues en 2016 alors que la méthode *DETAIL* a vu le jour en 2017.

Un aspect important de la super résolution vidéo qui a été vu dans la partie précédente est la **temporalité**. Nous remarquons que la méthode *ESPCN* n'aborde pas du tout la temporalité, elle pratique de la super résolution en **single image**, contrairement aux méthodes *VESPCN* et *DETAIL* qui elles, pratiquent du **multi-image** en ne traitant donc pas les images une à une indépendamment les unes des autres. La méthode *DRAFT* quant à elle mentionne seulement l'estimation de mouvement sans creuser la temporalité.

La méthode *ESPCN* reste tout de même attractive de par son efficacité pour traiter les images. C'est pour cela que la méthode *VESPCN* utilise ce système pour le coupler à la temporalité. Cette dernière méthode comporte une manière de traiter l'estimation de mouvement efficace qui sera reprise par la méthode *DETAIL*.

Concernant l'**efficacité**, elle est évaluée selon nos besoins : la vidéo. Elle concerne surtout la **qualité de rendu finale**. Voilà pourquoi la méthode *ESPCN* n'est pas très bien notée bien qu'elle reste attractive pour traiter les images. De plus, dans son article, elle met surtout en avant sa **vitesse** d'exécution avant sa qualité de résultat. Pour le reste, la comparaison reste compliquée car il est difficile de vraiment évaluer objectivement à l'oeil nu les résultats, et ces méthodes ne se comparent pas forcément entre elles avec le PSNR. Cependant, il est possible de déduire leur efficacité. Par exemple, les méthodes *DRAFT* et *DETAIL* se comparent objectivement toutes deux à une même méthode : *BayesSR*.

Concernant la méthode *DRAFT*, on remarque qu'en moyenne (sur 3 tests) son PSNR est plus élevé que celui de la méthode *BayesSR* de 3,3%. Cependant, son SSIM est moins élevé de 1,1%. Pour la méthode *DETAIL* en revanche, on voit qu'en moyenne, (sur 8 tests) son PSNR est plus haut que celui de la méthode *BayesSR* de 9% et son SSIM plus élevé de 4,3%.

Comparé à une même méthode, on peut conclure que **la méthode *DETAIL* est plus performante que la méthode *DRAFT*** au niveau de la qualité de rendu final.

De plus, la méthode *VESPCN* est comme une version évoluée d'*ESPCN* puisqu'elle en reprend le principe en l'améliorant. On peut donc en déduire qu'**elle est plus efficace**. Enfin, concernant la différence entre *DETAIL* et *VESPCN*, il est moins simple de vraiment les départager. *DETAIL* étant sortie plus tard, elle se compare à *VESPCN* en terme de PSNR et se montre seulement 0,78% plus efficace sur un même jeu de vidéos. Ces éléments ne nous **permettent donc pas** forcément **de les départager** au niveau de qualité de rendu, bien que ce soit ce dernier élément que ces deux méthodes mettent en avant.

La méthode *DRAFT* est la seule méthode qui **met à disposition son code**, c'est pour cela que cette méthode apparaît dans le tableau comparatif car cela peut peser dans la balance. Les autres méthodes proposent leur algorithme, les paramètres utilisés etc mais pas le code source.

L'avantage de ces techniques est qu'elles utilisent quasiment toute un jeu de données **disponible à tous**. La méthode *DRAFT* utilise un jeu de données qu'ils ont confectionné mais qui est mis à disposition avec le code source, contrairement à *DETAIL* qui utilisent aussi un jeu de données personnel mais qui n'est pas mis à disposition. *ESPCN* et *VESPCN* utilisent une base de données **publique**, à laquelle nous pourrions donc accéder plus tard.

Enfin, le dernier critère de sélection est la technologie utilisée. Il est dit que les modèles des méthodes *DETAIL* et *DRAFT* ont été développées en **Python** et respectivement grâce à **Tensorflow** et **Caffe** (surcouche de Tensorflow). Les méthodes *ESPCN* et *VESPCN* ne précisent pas la technologie ni le langage utilisé.

## 6.2 Choix final

Le choix final s'est effectué par élimination. Tout d'abord, le seul réel avantage de la méthode *DRAFT* est qu'elle mettait à disposition tout son code et ses jeux de données. Cela était donc attractif car même si le principe reste moins intéressant que les autres (pas de temporalité), cela aurait permis de rendre un projet fonctionnel et d'avoir un logiciel de manière sûre à la fin, quitte à évoluer vers d'autres techniques par la suite. Cependant, le code fourni est en **MatLab**. Or, nous trouvions plus simple d'implémenter notre modèle en Python. Traduire tout le code MatLab en Python perd tout l'avantage qu'avait cette technique.

*ESPCN* traitant le problème en **single image** et faisant partie entièrement d'un sous module de *VESPCN*, cela nous a permis de l'éliminer et de réfléchir aux deux dernières.

*VESPCN* et *DETAIL* n'ont pas énormément de différences, autant au niveau du principe général que de ce qu'elles peuvent mettre à disposition. La méthode *DETAIL* proposait cependant un schéma un petit peu plus simple mais nous avons préféré nous tourner vers les méthodes *ESPCN ET VESPCN*. En effet, cela permettrait de découper le projet en deux parties majeures et d'implémenter 2 méthodes. La première partie serait donc **l'implémentation du module ESPCN** afin de le rendre fonctionnel. Ensuite, en rajoutant les modules nécessaires de compensation de mouvement et d'estimation de mouvement, il est possible d'**améliorer** la première partie pour créer le modèle de la méthode *VESPCN*.

# 3

## Analyse du système

Pour des raisons expliquées en conclusion, cette analyse n'est pas complète. En effet, l'analyse VESPCN notamment n'a pas pu être faite profondément. Comme dit précédemment, le système va être séparé en deux parties : ESPCN et VESPCN. Nous allons étudier l'un après l'autre, car le second module est une évolution du premier.

### 1 Modèle ESPCN

#### 1.1 Jeu de données

Le jeu de données choisi va être conforme à ce qui a été utilisé pour l'expérimentation de cette méthode indiquée dans l'article. Nous allons nous concentrer uniquement sur la partie vidéo, et non la partie image.

On trouvera d'abord un premier set de 8 vidéos HD en 1920 x 1080 d'approximativement 10 secondes disponible ici : <https://media.xiph.org/video/derf/> puis un set de 7 vidéos HD aussi de 5 secondes de long, publiquement accessibles ici : <http://ultravideo.cs.tut.fi/#testsequences>.

#### 1.2 Premier module : Prétraitement des données

Comme expliqué dans la partie expérimentale, les vidéos doivent être traitées pour être passées dans l'algorithme.

Nous considérons que les vidéos sont déjà en basse résolution. La résolution aura auparavant été divisée par  $r$ , facteur par lequel nous voudrions par la suite augmenter la résolution.

Une vidéo n'étant rien d'autre qu'une suite d'images assemblées, nous allons extraire les images et les enregistrer indépendamment. **Keras** sait très bien s'occuper de cela.

#### 1.3 Deuxième module : Construction du modèle

Notre modèle comportera 3 couches de convolution à laquelle s'ajoutera la couche de convolution sous pixel.

Les trois couches sont une suite de **convolution 2D** qui font s'effectuer sur les images avec des filtres respectivement de taille 5, 3 et 3. Nous avons vu ce qu'était une convolution dans la partie 2.1.2. La convolution 1D est appliquée à un vecteur et une convolution 2D est appliquée à une matrice, ce qui est notre cas pour traiter une image. Avec un pas de 1 et une telle taille de filtre, la taille de l'image reste la même après la convolution qu'avant. Le but ici est de ne surtout pas perdre d'information ni de baisser en résolution. Dans la première couche, il y aura 64 filtres de taille 5x5, dans la deuxième il y aura 32 filtres de taille 3x3 et dans la dernière on aura un seul filtre de taille 3x3.

La dernière partie du modèle comporte la couche de **convolution sous pixel**. Comme expliqué précédemment, c'est le coeur du système et c'est cela qui va se charger de l'augmentation de résolution. À la fin des trois convolutions précédentes, nous nous retrouvons avec  $r$  canaux,  $r$  étant pour rappel le facteur d'augmentation de résolution. Ces  $r$  canaux, étant des sous-parties de l'image finale vont s'assembler de la façon suivante. Si nous considérons que dans l'image HR, un pixel correspond en fait à une matrice de pixel  $r \times r$ , alors un pixel de coordonnées  $(x,y)$  de chaque canal viendra former matrice  $r \times r$  se situant en  $(x,y)$  dans l'image HR.

Prenons un exemple. Avec un facteur d'augmentation de résolution de 3, nous auront 9 canaux en entrée de cette dernière couche. Si nous prenons le pixel  $(0,0)$  de chaque canal, alors le pixel  $(0,0)$  du canal 1 constituera le pixel  $(0,0)$  de l'image HR, ce même pixel du canal 2 constituera le pixel  $(1,0)$  de l'image HR alors que le pixel  $(0,0)$  du canal 9 constituera le pixel  $(2,2)$ . Voici un tableau généralisant ceci :

Coordonnées du pixel dans le canal $r$	(0,0)	(1,1)	(5,5)	( $r, r$ )
Canal 1 dans l'image HR	(0,0)	(3,3)	(15,15)	( $3 * r, 3 * r$ )
Canal 2 dans l'image HR	(1,0)	(4,3)	(16,15)	( $3 * r + 1, 3 * r$ )
Canal 3 dans l'image HR	(2,0)	(5,3)	(17,15)	( $3 * r + 2, 3 * r$ )
Canal 5 dans l'image HR	(1,1)	(4,4)	(16,16)	( $3 * r + 1, 3 * r + 1$ )
Canal 9 dans l'image HR	(2,2)	(5,5)	(17,17)	( $3 * r + 2, 3 * r + 2$ )

**Figure 1** – Tableau de conversion des pixels du canal  $r$  dans l'image HR finale.

## 1.4 Troisième module : Post-traitement

Une fois la séquence d'images avec une résolution augmentée, il faut retransformer la séquence d'images en vidéo. Encore une fois, **Python** est capable de réaliser cela facilement.

## 2 Modèle VESPCN

Pour ce modèle, le pré-traitement et le post-traitement des données sont le même que pour le modèle précédent. Nous n'allons donc pas le redétailler.

### 2.1 Jeu de données

Le jeu de données est disponible ici : <http://www.cdv1.org>. 115 vidéos sont tirées de cette base de données publique. Nous en utiliserons 100 pour l'entraînement.

De ces 100 vidéos réduites en résolution **en 3 ou 4 fois** 30 échantillons sont extraits aléatoirement de chaque paire HR-BR (pour la comparaison future) pour obtenir **3000** échantillons. Le système étant séparé en deux, nous utiliserons des images simples pour le module ESPCN et des blocs d'images consécutives déjà compensées pour le réseau spatio-temporel.

## 2.2 Construction du modèle

*C'est malheureusement cette partie qui demande le plus de temps et d'étude et c'est celle qui sera la moins complète car cette méthode à tout de même des subtilités complexes.*

Tout d'abord, pour estimer le mouvement par rapport à une image, nous allons l'estimer à partir de son image précédente ET suivante. Nous travaillerons donc avec des **blocs de 3 images** consécutives. Ce nombre est aussi la profondeur de la dimension temporelle.

### Fusion précoce

Nous retrouverons plusieurs sous modules qui sont conformes à ceux étudiées dans l'état de l'art. Le rassemblement des images en canaux par fusion précoce est possible avec le module **PIL** de Python par exemple. Toutes les convolutions vont être appliquées avec un filtre en **3x3**. Elles peuvent être réalisées grâce à une fonction **conv2D** proposée par Keras par exemple.

L'aspect important à noter est que le système est une **jointure** entre deux sous-systèmes. La partie temporelle (compensation et estimation) et la partie de convolution sous pixel. Cela signifie donc que les deux systèmes sont **entraînaibles séparément** (d'où les données d'entrée différentes).



# 4

## Mise en oeuvre

### 1 Installation de l'environnement

#### 1.1 Anaconda

En premier lieu, j'ai choisi **Anaconda** comme distribution Python. Elle permet de faciliter la gestion des paquets. Je l'ai choisi car c'est une distribution **open-source**, avec de nombreux "**packages**" et est très **populaire** ce qui signifie qu'il sera facile de trouver des exemples ou des tutoriels avec Anaconda.

Anaconda utilise le système *conda* pour gérer les paquets. Ainsi, pour en installer un, il suffira de taper en ligne de commande : `conda install nom-du-paquet`.

La version de Python utilisée ici sera la version 3.6 et non 3.7 car pour l'instant, Keras et Tensorflow ne fonctionnent qu'avec Python 3.6.

#### 1.2 TensorFlow et Keras

Comme expliqué dans la partie des spécifications, nous allons utiliser la bibliothèque Python **TensorFlow** et sa surcouche **Keras**. Nous allons les installer simplement avec :

```
conda install tensorflow et conda install keras.
```

Si nous vérifions les versions de ces derniers avec un petit code Python :

```
import keras;
print(keras.__version__)
```

Nous voyons que nous utilisons la dernière version de Keras, la version 2.2.4. Pour TensorFlow, nous utilisons la version 1.5.1.

#### 1.3 Rodeo

Comme environnement de développement, j'ai choisi RODEO. C'est un IDE open-source qui permet entre autres d'exécuter du code ligne par ligne. Cela me permet aussi de découvrir un nouvel outil.

Le code présent dans l'éditeur dans l'image suivante permet de vérifier la bonne installation de Keras et de TensorFlow.

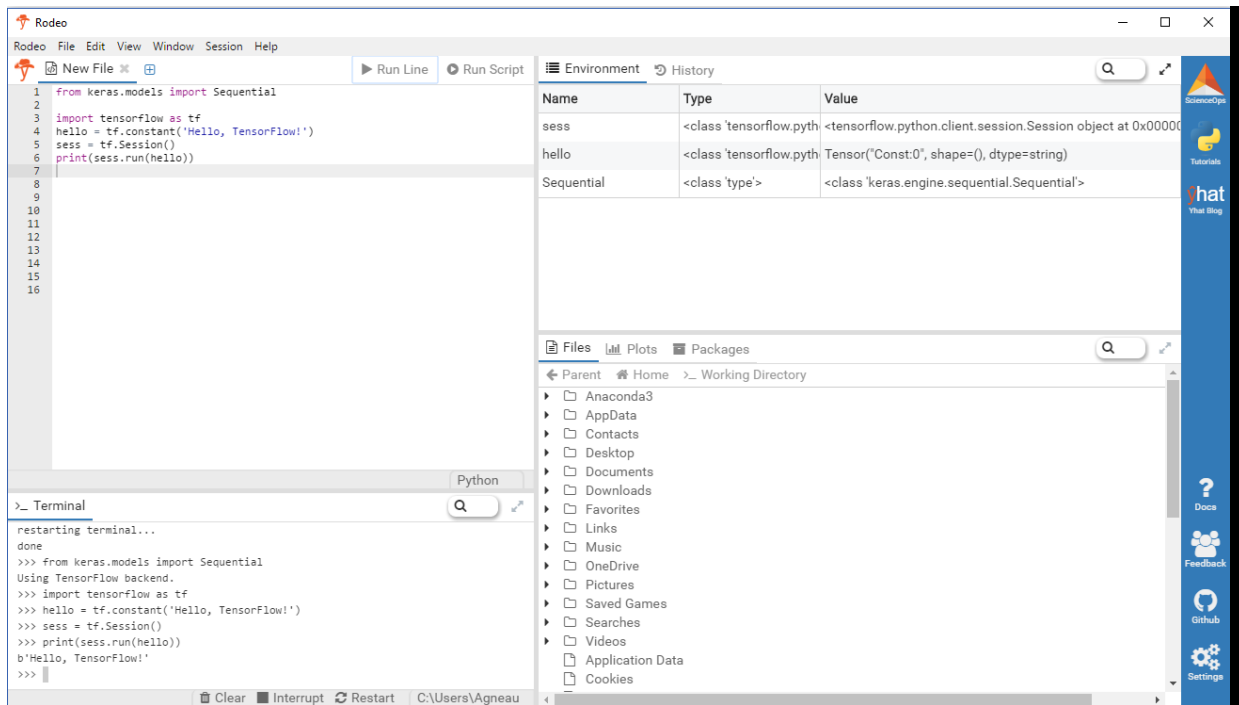


Figure 1 – Interface de l'IDE RODEO

## 1.4 PyCharm

Après quelques utilisations de Rodeo et mon changement de machine, j'ai préféré opter pour PyCharm avec lequel j'étais un peu plus familier et que je trouve plus fonctionnel et "user friendly".

## 2 Pré processing

La première partie, avant de s'attaquer au modèle était de gérer le pré traitement des données.

### 2.1 Base de données test

En premier lieu, j'ai téléchargé 23 vidéos d'une dizaine de secondes illustrant plusieurs scènes. Celles-ci ont pour but uniquement d'avoir une base pour commencer le traitement. Les vidéos sont en qualité 1920x1080, soit le standard HD.

### 2.2 Séquençage des vidéos

Comme expliqué précédemment, on ne peut travailler avec une vidéo telle quelle. Il faut la séquencer ou autrement dit, extraire des images de ces vidéos. Pour cela, un script Python à part s'en occupera. Il sera exécuté en dehors du script principal d'entraînement et générera des images dans les dossiers appropriés. Pour être plus précis, il générera 30 images (ou "frame") consécutives par vidéo. Ces images seront enregistrées en physique sur le disque de la machine.

### 2.3 Extraction des patches

Malheureusement, le réseau choisi ne pourra pas traiter des images de cette taille pour l'entraînement. En effet, pour s'entraîner le réseau va avoir besoin d'une image LR, mais aussi de l'image HR correspondante pour qu'il puisse avoir une sorte de "modèle". Comme expliqué précédemment, l'entraînement du modèle va consister en le **calcul de poids** pour faire en sorte que l'image HR transformée à partir de l'image LR corresponde le plus à l'image HR originale. Les patches extraits des images HR seront de taille  $17r * 17r$ .  $r$  étant le facteur d'augmentation de résolution des vidéos.

### 2.4 Sauvegarde en physique ou dans la mémoire vive?

Tous ces patches doivent être stockés quelque part lors du lancement du réseau. Il a donc fallu y réfléchir car les deux traitements sont différents. Stocker dans la mémoire vive permettrait d'y accéder plus facilement mais ferait perdre énormément de capacité de calcul à la machine. En effet :

Si l'on prend une image  $1920*1080$  et que l'on extrait des patches avec  $r = 3$ , cela nous fait donc environ **800** patches en  $51*51$ . Sachant que nous avons pour l'instant 23 vidéos, 30 frames par vidéo et 800 patches par frame, cela nous donne un total d'un peu plus de **550 000** patches. Cela fait donc une très grande quantité de donnée à stocker dans la mémoire vive qui sert normalement pour effectuer des calculs, entraîner le réseau etc...

Même si cela prendra beaucoup de place physique sur le disque, on considérera que l'espace de la machine est suffisant et que c'est la meilleure solution.

Pour stocker ces fichiers, il faut définir une architecture de fichiers.

## 3 Définition d'une architecture de fichiers

Pour bien gérer le système, il faut avoir une architecture bien précise. Les images, vidéos et scripts python doivent avoir un emplacement bien défini. Ceci a bien évidemment été faite au tout début mais elle est expliquée ici car les notions nécessaires ont été introduites dans la partie précédente.

Les classes dans un dossier *classes*, mais les scripts dans le dossier parent. En effet, les scripts ont besoin des classes donc c'est pour éviter de devoir remonter dans l'arborescence pour l'import de ces scripts. Alors que les classes n'ont pas besoin des scripts.

Les vidéos du dataset sont contenues dans un dossier *train* avec deux sous dossiers *videos* et *images*. On retrouvera donc dans le dossier *images* les frames extraites des vidéos avec des sous-dossiers ayant le nom des vidéos. Dans chaque dossier contenant les images, nous retrouverons toutes les frames et des sous dossiers correspondant à chaque frame et contenant les patches. Voici un exemple pour que ce soit plus clair :

## 4 Fonctionnement de l'entraînement

Lors de cette partie développement, j'ai passé une grande partie du temps à me renseigner sur des concepts, les comprendre et les appliquer. En effet, le deep learning est un domaine très nouveau pour moi et qui met du temps à assimiler. Le fonctionnement de l'entraînement en fait parti.

Un cycle complet d'entraînement est composé de plusieurs **époques** (ou epochs en anglais).

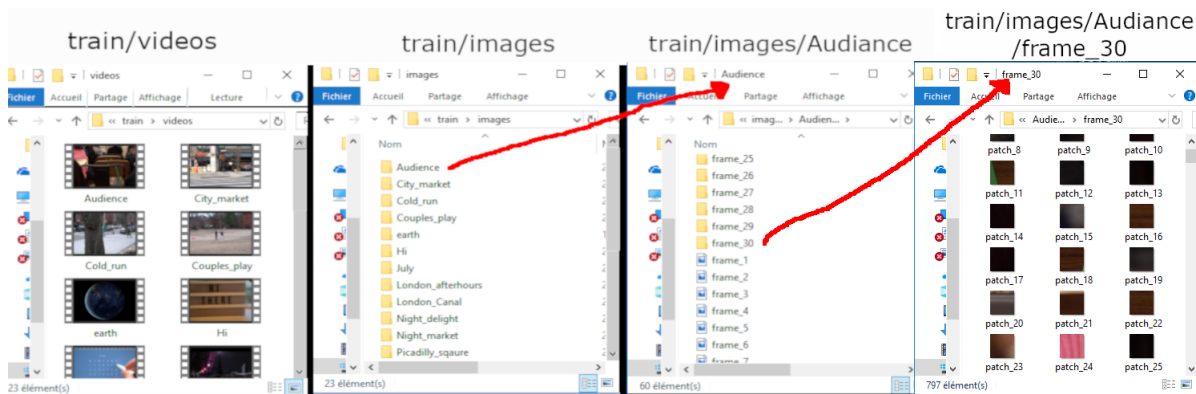


Figure 2 – Architecture des fichiers sur le disque

Durant une époque, tout le dataset est traité. Le réseau ajuste donc ses poids avec ces époques. Plus il y en a, plus le réseau va pouvoir ajuster ses poids et être efficace. Cependant, à partir d'un certain nombre dépendant de la complexité du réseau et du dataset, le réseau va rentrer dans ce que l'on a défini dans la partie **Etat de l'art** : l'overfitting. Le réseau se calquera trop sur les données d'entraînement et aura du mal à généraliser.

Ensuite, dans une époque, il y a plusieurs étapes, que l'on peut schématiser simplement comme suit :

1	2	3	4	5	...	n-1	n
---	---	---	---	---	-----	-----	---

Chaque case peut-être assimilée à **une étape de l'époque**. Dans chaque étape va être traitée un **lot** de donnée (ou batch). Imaginons que notre dataset est composé de 500 données et que nous avons 10 étapes. Par époque, chaque étape va donc traiter  $500/10 = 50$  données.

Ainsi, au lieu de traiter tout le dataset d'un coup, nous allons les traiter par **lot**. Et pour ceci, nous avons besoin d'un **générateur**.

## 5 Séparation du dataset

Avant de parler du générateur, nous allons d'abord voir comment est géré le dataset lors de l'entraînement. Nous avons déjà deux parties : La partie **training** (disons 80% du dataset complet) et la partie **test** (20% donc du dataset).

La partie training sont les données qui servent au réseau pour s'entraîner, ajuster ses poids etc... La partie test, elle, sert à utiliser le réseau entraîné sur de nouvelles données. Le réseau sera alors évalué d'un manière décrite plus tard.

Cependant, pour avoir un réseau plus efficace, nous ajoutons 10% de données de **validation** à la place de données de training. A chaque fin d'époques, le réseau va se confronter à des données de validation et il va ainsi pouvoir se tester et voir où il en est. Ces 10% de données de validation sont toujours les mêmes mais ce n'est pas un problème car le réseau n'apprendra pas sur ces données et les "oubliera".

A la création du réseau, il y a donc une fonction qui sépare tout le dataset en 3 parties avec les pourcentages cités plus hauts, qui peuvent être changés.

## 6 Générateur

Le générateur est un concept de python inconnu pour moi avant, mais très bien adaptable au deep learning.

Un générateur est une fonction qui, à chaque fois qu'elle va être appelée, va renvoyer une valeur différente. C'est donc de ceci que nous allons nous servir pour **alimenter les batches** à chaque étape de l'époque.

D'abord, il nous définir une taille de nos lots, une *batch\_size*. Cette taille de batch est de l'ordre en général de 32, 64 ou 128 par exemple, mais nous allons partir sur 128. J'ai donc instancié un générateur général pour les différentes parties du dataset.

Là où est l'intérêt d'avoir stocké les patches en physique, est que lors du lancement du programme, il faut bien avoir toutes les données. Au lieu d'avoir tous les patches, nous allons stocker tous les **chemins** vers les patches. Nous aurons ainsi à garder des **chaînes de caractère plutôt que des images**. Voici le code du générateur que nous allons analyser pour comprendre le fonctionnement : (voir **figure 3 - Code python du générateur**)

A chaque fois que la fonction *generator* va être appelée, tout ce qui se trouve dans la boucle

```
def generator(self, patches_HR_paths, batch_size, r, patches_paths = None):
    """
    Pour ne pas stocker une immense liste d'images, on va donner à s'entraîner des batches, ou des lots.
    Ces lots seront générés par ce générateur.

    Il prend simplement aléatoirement batch_size chemins dans patches_HR_paths_train, lit les images,
    les convertit et les renvoie grâce au mot clé yield.

    A chaque fois que cette fonction sera appelée, elle reprendra son exécution au while True et renverra des
    nouvelles valeurs avec le yield

    :param trainOrValidate: Determine si il faut piocher dans la liste de training ou de validation
    :return: Une liste d'images en ndarray
    """
    while True:

        batches_paths = np.random.choice(a=patches_HR_paths, size=batch_size)

        x_train = [] # patches LR
        y_train = [] # patches HR

        for path in batches_paths:
            patch_HR = io.imread(path) # On lit l'image à partir du chemin

            patch_LR = rescale(patch_HR, 1.0 / r, anti_aliasing=False) # Downscale le patch HR

            x_train.append(patch_LR) # On l'ajoute au x_train (data)
            y_train.append(patch_HR) # On l'ajoute au y_train (labels)

        x_train = np.array(x_train) # Convertit les list en np array
        y_train = np.array(y_train)

        yield x_train, y_train
```

Figure 3 – Code python du générateur

While true va être exécuté et la fonction renverra ce qu'il y a après le mot clé **yield**, qui fait donc office de *return*.

Dans un premier temps, nous allons choisir un lot (128 données) au hasard parmi le dataset de training (70% du dataset complet).

Ensuite, pour chaque donnée de ce lot, nous récupérons l'image à partir du chemin. A partir de cette image, nous créons le patch LR correspondant au patch HR.

Nous mettons ces deux images dans deux listes que nous formatons en *numpy array* pour que ce soit compréhensible par le réseau et que nous renvoyons grâce au mot clé *yield*.

Nous voyons ainsi que seulement 128 \* 2 images sont traitées à la fois, au lieu de 550000 \* 2 images. La mise en place du générateur n'était pas chose aisée car il a fallu bien comprendre le

principe des lots, de l'entraînement et de la séparation du dataset et ce générateur est une part très importante du projet car c'est par là que tout passera.

## 7 Évaluation du modèle

Comme dit précédemment, le modèle, après entraînement va être évalué avec la partie **test** du dataset. Il va être évalué selon deux "métriques", deux critères.

La première, c'est le **loss**. C'est l'interprétation de l'évolution du modèle. C'est une addition des erreurs faites lors du test ou de l'entraînement/validation. Plusieurs calculs sont possibles, mais nous allons utiliser **MSE**, pour **Mean Squared Error** qui revient à faire la moyenne des erreurs au carré, chose assez courant dans les CNN. Plus la loss est petite, mieux le réseau se comporte. Nous avons ensuite le **PSNR** (ou Peak Signal to Noise Ratio) qui traduit la différence entre 2 images sous forme de bruit, donc exprimé en décibel. Le PSNR va donc être calculé entre le patch HR original, et le patch HR obtenu grâce au réseau à partir du patch LR. Plus le PSNR est grand, plus les images sont conformes.

## 8 Entraînement, validation, évaluation

Pour toutes ces tâches, j'ai utilisé des fonctions de Keras toutes dédiées.

### 8.1 Entraînement et validation

Au lieu d'utiliser la fonction *fit* classique de Keras, nous allons donc utiliser *fit\_generator* qui prendra en paramètre les générateurs dont nous avons parlé plus tôt. Voici comment le training est implémenté : En premier paramètre nous retrouvons le générateur qui va être appelé de

```
def trainWithGenerator(self):
    """
    Entraîne le réseau avec la fonction fit_generator qui prend en paramètre un générateur (donc la fonction précédente).
    Il sauvegarde le modèle si il a besoin d'être entraîné et affiche le résumé du modèle avec ses couches
    :return: History objet
    """
    print(self.model.summary())
    # plot_model(self.model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
    nb_samples = len(self.patches_HR_paths_train)
    history = self.model.fit_generator(self.generator(self.patches_HR_paths_train, self.batch_size, self.r),
                                     steps_per_epoch=nb_samples//self.batch_size,
                                     epochs=self.epochs,
                                     verbose=1,
                                     validation_data=self.generator(self.patches_HR_paths_validate, self.batch_size, self.r),
                                     validation_steps=int((len(self.patches_HR_paths_validate)/self.batch_size)))

    if self.is_training:
        self.save('espcn')

    #plot_model(history) # Affiche la courbe du model
    return history
```

Figure 4 – Code python de la fonction s'occupant du training.

sorte à ce qu'il traite la partie "training" du dataset. Pour calculer le nombre d'étapes par époque, c'est très simple. Si nous voulons traiter TOUT le dataset en une époque et qu'à chaque état, un lot d'une taille définie est traité, alors il nous suffit de diviser le nombre total de données par la taille du lot pour obtenir le nombre d'étapes.

On donne en tant que **validation** le générateur adapté pour les données de validation. De la même manière, le nombre d'étape par validation est calculé en divisant le nombre de données du dataset de validation par la taille des lots.

Le modèle est ensuite sauvegardé dans un fichier JSON ainsi que ses poids.

## 8.2 Évaluation

Concernant l'évaluation du modèle, on va procéder de la même manière que précédemment en utilisant *evaluate\_generator* au lieu de *evaluate* classique, en utilisant le générateur adapté pour la partie **test** du dataset.

## 9 Organisation du code

### 9.1 Modèle général et modèles dérivés

Afin de créer un code évolutif, j'ai créé une classe de modèle générique de laquelle vont hériter les modèles créés par la suite. Dans ce modèle générique nous retrouvons déjà tous les attributs, à savoir : la taille d'input des patchs (17 dans nos exemples), la dimension des couleurs (3 pour des images colorées, 1 pour du noir et blanc), un booléen qui indique si le réseau s'entraîne ou pas, le facteur d'augmentation de résolution, le learning rate, la taille des lots et le nombre d'époques.

Nous retrouvons aussi 4 fonctions en plus du constructeur, dont trois implémentées. Tout d'abord, une fonction vide **build\_model** qui servira à construire le modèle. Elle n'est pas généralisable car elle dépend vraiment du modèle en question mais est indispensable. Nous avons deux fonctions **load** et **save** qui vont charger et sauvegarder le modèle sous format JSON et mh5 pour les poids. Enfin, la fonction **train** qui est de base générique à chaque modèle qui n'utilise pas de générateur. Elle pourra cependant être modifiée si besoin.

La classe **ESPCN** hérite donc de la classe générique **BaseModel** qui implémente, en plus des fonctions mères, toutes celles servant à ce dont j'ai évoqué plus haut.

### 9.2 Gestion des paramètres

Les différents modules du programme se lancent actuellement par ligne de commande. Tous les attributs de la classe de modèle font parti des paramètres à lancer dans la ligne de commande. En Python, ceci s'exécute ainsi :

`python nom_du_script.py --param1 valDuParam1 --param2 valDuParam2` et ceci pour un dizaine de paramètre. Si aucun des paramètres n'est spécifié, alors une valeur par défaut lui est assigné. C'est avec ceci que nous allons faire en sorte de faciliter la gestion des paramètres.

En effet, j'ai créé une classe **parametersESPCN.py** qui centralise **tous** les paramètres donc le réseau aura besoin comme par exemple les chemins vers les images, le taux de données d'entraînement que l'on choisi ou encore tous les autres attributs cités précédemment. Donc si on ne précise aucun argument lors du lancement des scripts, ce sont ces valeurs qui seront prises en compte.

## 10 Utilisation du réseau et résultats obtenus

### 10.1 Utilisation du réseau

Pour utiliser le réseau, il faut effectuer ce que l'on appelle une **prédiction** qui va être réalisée avec la fonction **predict\_generator**. Pour la prédiction, nous allons utiliser une vidéo LR, dont



```

class Parameters:
    def __init__(self):
        self.videos_path = "train/videos"
        self.HR_path = "train/images"
        self.LR_path = "train/images/LR"
        self.r = 3
        self.nb_frames_per_video = 30
        self.input_size = 17
        self.colors_dimension = 3
        self.learning_rate = 0.01
        self.batch_size = 128
        self.epochs = 30
        self.stride = 14
        self.percentage_train = 0.7
        self.percentage_validation = 0.1
        self.n1 = 64
        self.n2 = 32
        self.f1 = 5
        self.f2 = 3
        self.f3 = 3

    def parse_fichier(self):
        pass

```

Figure 5 – Code python de la classe Parameters.

nous avons pas la version HR. C'est donc un nouveau module qui sera dans un nouveau fichier **predict\_video.py**.

Nous allons cette fois-ci extraire **toutes** les frames de la vidéo, et non plus 30. Nous allons toujours extraire les patches de ces frames et les enregistrer en physique sur le disque.

Il faut ensuite développer et utiliser un autre générateur pour que Keras l'utilise.

Une fois la prédiction réalisée, cela nous ressort des images en format *numpy array*. Il nous faut reconstituer les images à partir des patches puis la vidéo à partir des images. Pour l'instant, la reconstitution des images à partir des patches ne fonctionne pas ce qui nous empêche d'avoir un résultat visuel ici.

## 10.2 Résultats obtenus

Le réseau n'a jamais pu être entraîné correctement (environ 100 epochs) par manque de machine plus puissante que la mienne (GTX 1060). En effet il aurait fallu entre 36 et 48h pour autant d'époques. Voici donc le graphique obtenu pour seulement 10 époques :

On voit que les valeurs ne sont pas exceptionnelles mais suivent cependant une bonne progression avec le loss qui diminue et le PSNRLoss (calcul associé au PSNR) qui augmente.



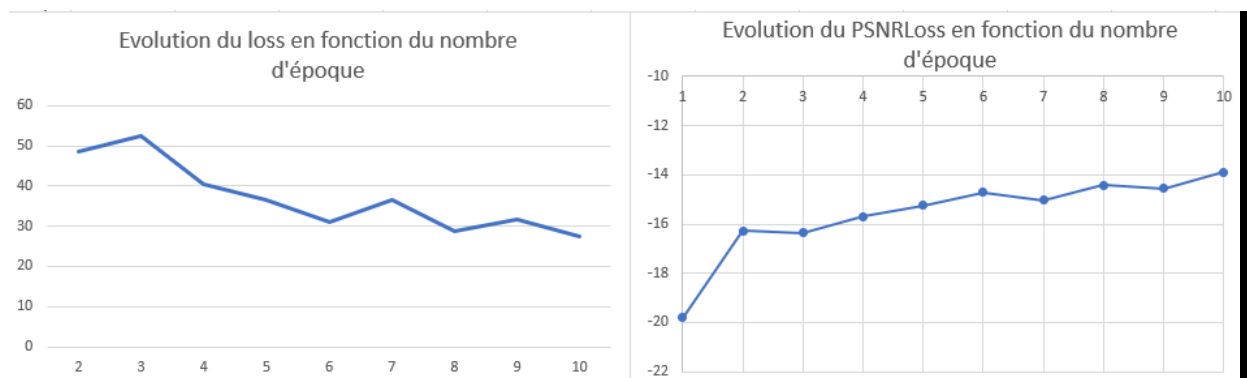


Figure 6 – Graphes des métriques en fonction des époques (pour 10 époques)

# 5

## Bilan et conclusion S9

### 1 Tâches faites

La grosse partie de ce premier semestre était consacrée à la **recherche**. En effet, le Deep Learning était un concept tout nouveau pour moi. Il a donc fallu que je m'informe sur tout le principe avant de pouvoir étudier vraiment le sujet. Pour me familiariser avec ce concept, j'ai aussi suivi des tutoriels sur la **classification d'image** (utilisation commune du Deep Learning) afin de me familiariser avec l'environnement Python et Keras.

Mi-Octobre, j'ai réalisé une présentation à mes tuteurs pour montrer ce que j'ai compris des réseaux de neurones et de la super-résolution.

À partir de là, j'ai pu attaquer l'état de l'art en étudiant 4 articles scientifiques proposant des méthodes de super-résolution. Ces articles sont assez pointus et en anglais. Comme c'est la première fois que j'ai affaire à des documents de ce genre, l'analyse de ces documents a pris une très grande partie de mon temps.

Une fois les méthodes étudiées, il a fallu en choisir une que je vais implémenter au prochain semestre. Une fois la méthode choisie, l'analyse du système peut être réalisée. C'est ici que le retard a été pris.

### 2 Retards

Un seul retard a été pris et il concerne l'analyse du système. En effet, l'état de l'art ayant duré longtemps, en ajoutant d'autres difficultés d'indisponibilité, le choix de la méthode finale n'a pu être faite que le **03/12/18**, soit une semaine avant le rendu de ce rapport. En effet, le choix d'une méthode était important et il me fallait du temps pour bien comprendre les enjeux et fonctionnements des méthodes détaillées dans les articles. L'analyse n'ayant pu être faite avant le choix, il a été difficile de réaliser l'analyse poussée et détailler de deux systèmes (**ESPCN** et **VESPCN**) en si peu de temps, en plus de la rédaction du rapport et de la préparation de la soutenance. La grosse partie manquante étant l'analyse de la deuxième partie, ceci n'est pas très pénalisant car c'est la première partie (le modèle *ESPCN*) qui va être implémenté en premier.

### 3 Tâches à faire

Cette partie abordera les tâches restantes à faire de manière globale. Cela sera détaillé dans l'annexe A avec un diagramme de Gantt et une précision de chaque tâche.

#### 3.1 Fin de l'analyse

La première tâche de ce second semestre est de rattraper le petit retard pris et de **finir l'analyse** et d'aller un peu plus en profondeur afin de mieux aborder la suite du projet.

#### 3.2 Mise en place de l'environnement

Afin d'attaquer la partie développement, il faudra **mettre en place** l'environnement (IDE, installation de Python, Keras). Une fois cela fait, l'implémentation du premier modèle peut démarrer

#### 3.3 Module ESPCN

La première tâche à **implémenter** est le module ESPCN. C'est la base du second module. Une fois ce modèle construit, entraîné, fonctionnel et validé, la suite du projet peut être réalisée. Cette tâche n'est cependant pas aisée car le code n'est pas rendu disponible et l'algorithme n'est pas clairement écrit, il faudra le déduire.

#### 3.4 Module VESPCN

Une fois le module précédent terminé, on y ajoutera le système d'**estimation et de compensation mouvement** afin de créer le modèle VESPCN. Ce modèle sera aussi à construire, entraîner et valider indépendamment du précédent. C'est de cette architecture que nous attendons une sortie de vidéo HR à partir d'une vidéo LR.

# 6

## Bilan et conclusion du S10

### 1 Retards et tâches non faites

Plusieurs choses n'ont pas pu être terminées d'être implémentées. Tout d'abord, l'entraînement complet du réseau n'a pas pu être effectué comme expliqué dans la partie **résultats**.

Nous avons ensuite la reconstitution en patch en image qui n'a pas été implémentée. Cela nous empêche d'avoir un résultat visuel du résultat du réseau.

Enfin, il pourrait être possible d'améliorer les résultats en effectuant des ajustements comme les paramètres du réseau ainsi que l'implémentation du modèle.

### 2 Raisons du retard et difficultés

La première raison du retard qui a été accumulé est due à une panne de ma machine personnelle qui a commencé en Novembre. Jusqu'à fin Janvier, j'ai renvoyé 4 fois ma machine en réparation pour finalement devoir en acheter une autre. C'est à cause de cela que je n'ai pas cherché de solution dès le début pour trouver une machine de remplacement car j'étais censé récupérer la mienne "la semaine prochaine". J'aurais cependant dû chercher une solution de dépannage. Je n'ai donc pas pu m'avancer sur le développement pendant les vacances de Noël et travailler sur les machines virtuelles de l'école n'était pas chose aisée.

De plus, il n'y a **aucun existant** sur l'article scientifique que j'ai analysé. Il y a quelques fois des dépôts git à disposition pour voir comment a été implémenté le modèle en question mais ici il n'y avait **ni dépôt, ni détails d'implémentation**. Sachant que c'est un tout nouveau domaine que j'ai dû découvrir, j'ai dû me concentrer à voir comment marchaient les concepts et trouver le moyen d'implémenter le modèle en question.

Enfin, on trouve beaucoup d'explications et d'exemples de CNN mais beaucoup sur la **classification** et très peu sur la super résolution qui est un nouveau domaine. J'avais donc moins de bases pour mes recherches.

En autocritique, je pense que j'aurais dû aller voir mes tuteurs plus souvent sur la partie technique lorsque je bloquais mais j'essayais de faire le maximum tout seul pour résoudre les problèmes car j'ai considéré que cela faisait parti de ma démarche de PR&D.

### 3 Bilan sur ce qui fonctionne

Préprocessing	Training	Prédictions
Séquançage de vidéos	Entrainement du réseau	Préprocessing
Extraction des patches	Training complet	Transformation des patches en image height

### 4 Conclusion générale

Grâce à ce projet, j'ai pu découvrir un nouveau domaine : le deep learning. J'ai aussi pu être confronté à de grandes phases de recherches car même si la charge de développement n'est pas excessive, il y a eu beaucoup de recherches dans ce second semestre.

J'ai aussi pu être confronté à des problèmes de gestion de projet. Même si le planning n'est pas respecté, il y a quand même une cohérence (voir partie **Décalage du planning** en annexe).

## Annexes

# A

## Gestion de projet

### 1 Tâches à faire au semestre 9

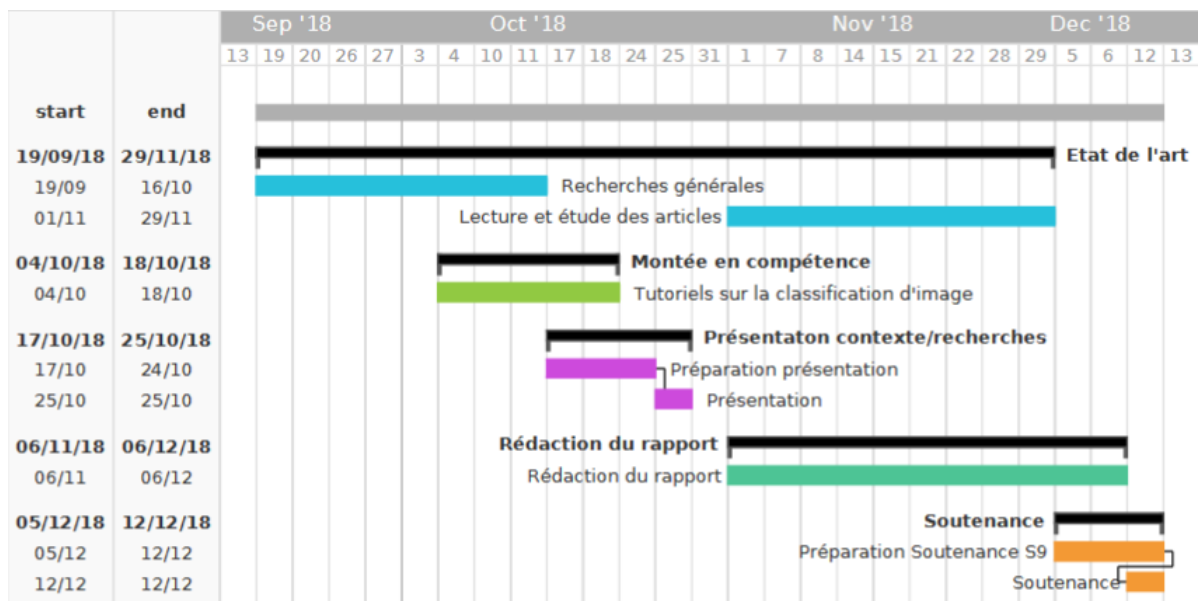


Figure 1 – Diagramme de Gantt des tâches réalisées au S9.

#### 1.1 Tâche 1 : Recherches générales

**Date de début :** 19 Septembre 2018

**Date de fin :** 16 Octobre 2018

**Description de la tâche :** Cette tâche comporte les recherches générales faites sur le Deep Learning en général et la super résolution.

## 1.2 Tâche 2 : Tutoriels sur la classification d'image

**Date de début :** 04 Octobre 2018

**Date de fin :** 18 Octobre 2018

**Description de la tâche :** Une fois les premières recherches faites, ces tutoriels ont servis à prendre en main l'environnement de développement et de manipuler une première forme du Deep Learning. Cette tâche rassemble donc la mise en place de l'environnement, le tutoriel sur de la classification d'image et un autre tutoriel visant à visualiser la sortie de chaque couche. Ce dernier n'a pas été totalement concluant mais n'étant pas le sujet principal de ce projet, j'ai préféré de pas y accorder trop de temps.

## 1.3 Tâche 3 : Préparation de la présentation & présentation..

**Date de début :** 17 Octobre 2018

**Date de fin :** 24 Octobre 2018

**Description de la tâche :** Cette tâche contient la finalisation des recherches générale, le regroupement des résultats des tutoriels et la création d'une diapositive pour présenter le travail actuel aux tuteurs de ce projet. Cette tâche est importante car elle marque le début de la seconde partie de l'Etat de l'art. C'est à partir de là que j'ai eu connaissance des articles scientifiques.

## 1.4 Tâche 4 : Lecture et étude des articles

**Date de début :** 01 Novembre 2018

**Date de fin :** 29 Novembre 2018

**Description de la tâche :** C'est la partie centrale de l'Etat de l'art. Les articles ont été lus dans un premier temps de manière globale afin de comprendre le principe, puis plus profondément afin de pouvoir comprendre le système et choisir une méthode.

## 1.5 Tâche 5 : Rédaction du rapport

**Date de début :** 06 Novembre 2018

**Date de fin :** 06 Décembre 2018

**Description de la tâche :** La rédaction du rapport comporte la mise en place LaTeX de l'environnement, ainsi que les étapes petit à petit afin d'être au plus possible à jour. Les spécifications ainsi que l'analyse n'ont plus être faites qu'après avoir choisi la méthode (le 03/12/18). La partie 'Etat de l'art' s'est écrite en parallèle que la fin de la tâche 'Lecture et étude des articles'.

## 1.6 Tâche 6 : Préparation de la soutenance et soutenance

**Date de début :** 05 Décembre 2018

**Date de fin :** 12 Décembre 2018

**Description de la tâche :** La soutenance a été préparée la semaine précédente le jour où elle se déroule. Cela comprend le choix du thème, la construction du plan et du diapo ainsi que l'entraînement à l'oral.



## 2 Tâches à faire au semestre 10

### 2.1 Tâche 7 : Fin de l'analyse

**Date de début :** 19 Décembre 2018

**Date de fin :** 20 Décembre 2018

**Description de la tâche :** La première à chose à faire dans cette deuxième est de rattraper le retard en finissant la première partie. Une fois l'analyse approfondie et terminée, le développement peu démarrer.

### 2.2 Tâche 9 : Mise en place de l'environnement de développement

**Date de début :** 09 Janvier 2019

**Date de fin :** 09 Janvier 2019

**Description de la tâche :** Ayant eu ma machine personnelle en réparation pendant 1 mois et demi en milieu d'année, il me faudra réinstaller tout l'environnement de développement.

### 2.3 Tâche 10 : Construction du modèle ESPCN

**Date de début :** 09 Janvier 2019

**Date de fin :** 30 Janvier 2019

**Description de la tâche :** Implémentation du modèle ESPCN sous Keras en accord avec l'article et donc l'analyse

### 2.4 Tâche 11 : Entraînement du modèle ESPCN

**Date de début :** 30 Janvier 2019

**Date de fin :** 06 Janvier 2019

**Description de la tâche :** Ceci concerne l'entraînement du réseau de neurones précédemment construit. Cette tâche est compliquée à quantifier car cela dépend énormément de la machine utilisée, et les configurations de celle-ci ne sont pas connues à ce jour. Pour raccourcir le temps d'entraînement, il sera possible de réduire le jeu de données.

### 2.5 Tâche 12 : Validation et corrections du modèle ESPCN

**Date de début :** 07 Février 2019

**Date de fin :** 20 Février 2019

**Description de la tâche :** Cette tâche comprend la validation du modèle et des résultats obtenus, mais aussi les correctifs à apporter ainsi qu'un ré-entraînement possible.

### 2.6 Tâche 13 : Documentation d'utilisation ESPCN

**Date de début :** 30 Janvier 2019

**Date de fin :** 07 Février 2019

**Description de la tâche :** Parallèlement à l'entraînement (qui peut se faire en tâche de fond), la documentation d'utilisation de ce module sera écrite. Ainsi, en cas de retard sur le deuxième modèle, la documentation d'un modèle **fonctionnel** sera disponible.

**2.7 Tâche 14 : Cahier de test ESPCN**

**Date de début :** 07 Février 2019

**Date de fin :** 20 Février 2019

**Description de la tâche :** De même que pour la tâche précédente, le cahier de test sera rédigé en même temps que la phase de validation et de correction afin d'avoir un cahier de test d'un modèle fonctionnel disponible en cas de retard.

**2.8 Tâche 15 : Construction du modèle VESPCN**

**Date de début :** 21 Février 2019

**Date de fin :** 13 Mars 2019

**Description de la tâche :** Implémentation du modèle ESPCN sous Keras en accord avec l'article et donc l'analyse. Cette implémentation sera faite indépendamment de la première, puis viendra la compléter afin d'assembler ces deux parties.

**2.9 Tâche 16 : Entraînement modèle VESPCN**

**Date de début :** 13 Mars 2019

**Date de fin :** 20 Mars 2019

**Description de la tâche :** Ceci concerne l'entraînement du réseau de neurones VESPCN précédemment construit. Cette tâche est compliquée à quantifier car cela dépend énormément de la machine utilisée, et les configurations de celle-ci ne sont pas connues à ce jour. Pour raccourcir le temps d'entraînement, il sera possible de réduire le jeu de données.

**2.10 Tâche 17 : Validation et correction du modèle VESPCN**

**Date de début :** 21 Mars 2019

**Date de fin :** 27 Mars 2019

**Description de la tâche :** Cette tâche comprend la validation du modèle VESPCN et des résultats obtenus, mais aussi les correctifs à apporter ainsi qu'un ré-entraînement possible.

**2.11 Tâche 18 : Complétion de la documentation d'utilisation**

**Date de début :** 13 Mars 2019

**Date de fin :** 20 Mars 2019

**Description de la tâche :** Ceci aussi se fait en parallèle de l'entraînement. La documentation faite précédemment sera complétée conformément au modèle VESPCN.

**2.12 Tâche 19 : Complétion du cahier de test**

**Date de début :** 21 Mars 2019

**Date de fin :** 27 Mars 2019

**Description de la tâche :** Cette tâche se fait en parallèle de la validation. Le cahier de test fait précédemment sera complétée conformément au modèle VESPCN.

### 2.13 Tâche 20 : Rédaction du rapport

**Date de début :** 21 Février 2019

**Date de fin :** 27 Mars 2019

**Description de la tâche :** Le rapport sera rédigé au fur et à mesure du projet à partir du moment pour le module ESPCN est terminé. En effet, il est important de se concentrer sur cette tâche afin d'avoir quelque chose de fonctionnel à la fin.

### 2.14 Tâche 21 : Préparation de la soutenance et passage.

**Date de début :** 27 Mars 2019

**Date de fin :** 03 Avril 2019

**Description de la tâche :** La soutenance sera préparée la semaine précédente le jour où elle se déroulera. Cela comprend le choix du thème, la construction du plan et du diapo ainsi que l'entraînement à l'oral.

## 3 Décalage du planning

Finalement, le planning est resté à peu près le même sauf qu'il a été décalé de 3 semaines car j'ai réellement commencé mon développement fin Janvier. Si on regarde, la partie VESPCN aurait duré environ 3 semaines ce qui est cohérent. Dans l'élaboration de ce planning, j'avais fait en sorte que, si il y avait du retard, le module ESPCN allait être accompagné de ses documents et guides.

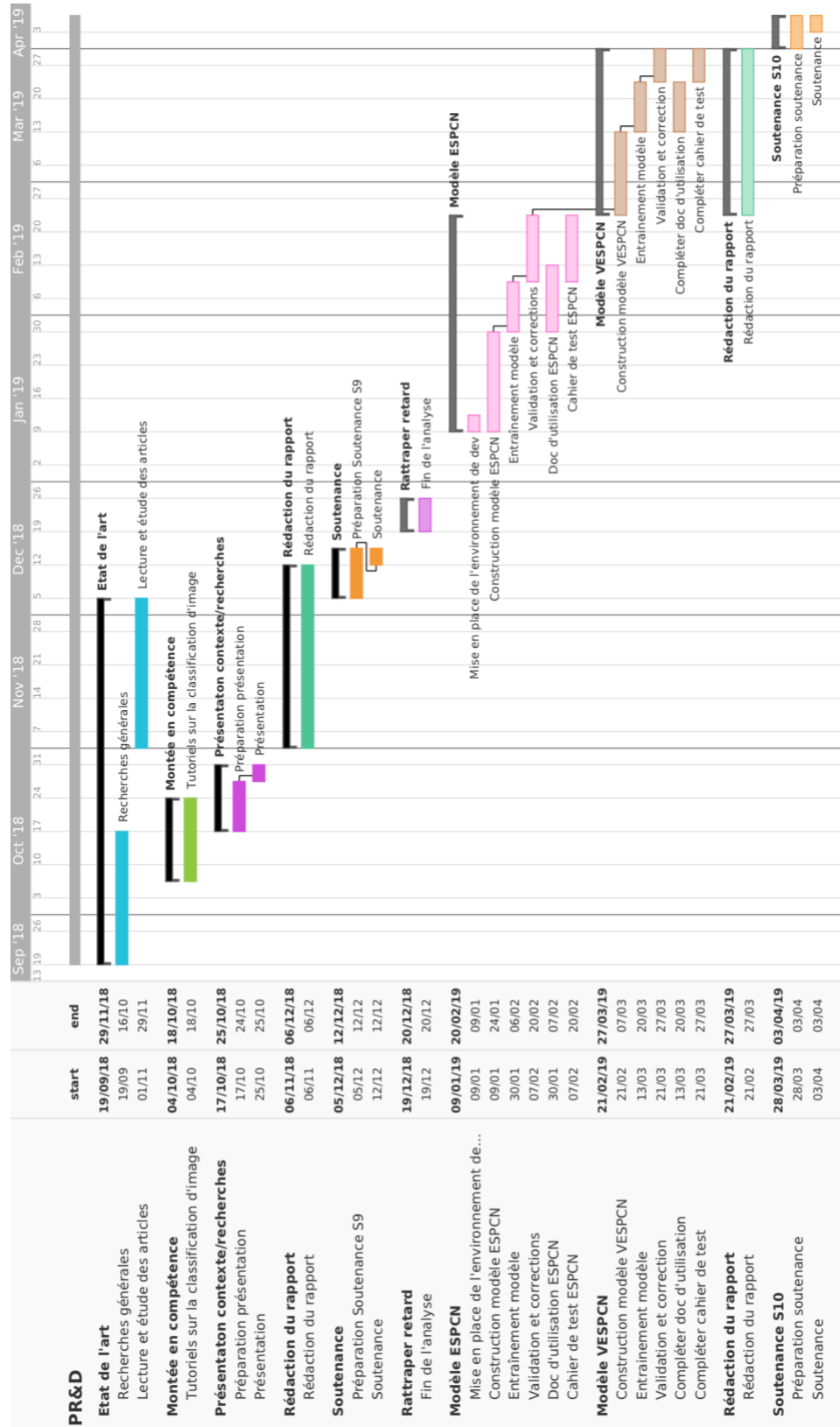


Figure 2 – Diagramme de Gantt complet. Tâches réalisées au S9 et prévisionnelles au S10.

# B

## Description des interfaces

### 1 Interfaces matériel/logiciel

Il n'y aura aucune interface matériel/logiciel car il n'y aura aucune communication avec quelconque matériel.

### 2 Interfaces homme/machine

On ne trouvera pas non plus d'interface homme/machine à proprement parler. Notre programme s'exécutera par ligne de commande, que ce soit pour l'entraînement, le test ou l'utilisation finale.

Il n'y aura **pas** de **visualisateur** de vidéo prévue. Le programme générera le fichier vidéo. L'utilisateur devra, si il le souhaite, la consulter de manière indépendante à notre système.

### 3 Interfaces logiciel/logiciel

Enfin, il n'y aura pas non plus d'interface logiciel/logiciel car tout sera géré par les bibliothèques de Python (Tensorflow et ses sur-couches).

# C

# Spécifications fonctionnelles

## 1 Test & validation

Pour tester et valider les résultats, nous allons prendre une vidéo **haute résolution** (HR) et la réduire en **basse résolution** (BR) puis passer cette dernière dans l'algorithme. Celui-ci va créer une vidéo **HR\*** qui va être comparée à la vidéo **HR** originale.

Concernant la comparaison, il y aura une première comparaison **visuelle** mais elle ne suffira pas. Il faudrait utiliser des mesures objectives comme par exemple avec **PSNR** ou **SSIM** ou autre, qui permettraient de comparer les vidéos image par image.

## 2 Transformation des vidéos en basse résolution

Pour passer les vidéos de **HR** en **BR**, on utilisera un logiciel **externe** qui ne fera pas parti de notre système. On peut penser à un logiciel comme VirtualDub par exemple qui est capable de ce genre de manipulation.

## 3 Base de données et jeu de données

Pour pouvoir entraîner le réseau de neurones, il faut utiliser un jeu de données adéquate. Pour cela, nous allons utiliser la même base de données que la méthode **ESPCN** et que la méthode **VESPCN**. Ces bases de données sont publics. Nous allons en plus choisir les mêmes jeux de données et en même quantité pour mieux pouvoir vérifier l'efficacité de notre système.

## 4 Entrées et sorties du système

En entrée nous allons avoir nos vidéos basses résolutions dont nous voulons que la résolution soit augmentée par l'algorithme. Pour cela, nous allons donner en entrée notre vidéo **traitée**, c'est à dire qu'elle sera sous la forme de séquences d'image successive.

Nous n'aurons cependant pas de sortie. La seule sortie sera un message de confirmation comme quoi tout s'est bien passé. Comme dit plus haut, il n'y aura **pas**, à la sortie de l'algorithme, la vidéo traitée qui se lance dans un visualisateur de média.

# D

## Spécifications non fonctionnelles

### 1 Contraintes de développement et conception

Concernant le langage de programmation, Python sera adopté ainsi que l'utilisation d'une bibliothèque adaptée comme Tensorflow. Keras, sur-couche de Tensorflow, sera utilisé si il contient tous les modules nécessaires à l'élaboration de l'algorithme.

### 2 Contraintes de fonctionnement et d'exploitation

Il n'y a pas de contrainte spécifique de développement liée au matériel. Il faut noter cependant qu'une machine effectuant plus de calculs à la seconde qu'une autre pourra effectuer les tâches d'entraînement et de test plus rapidement. Ainsi avec une machine plus "puissante" (avec un GPU plus performant) pourra prendre en compte un plus grand jeu de données pour un temps d'exécution plus court.

J'utiliserai donc ma machine personnelle pour tester le fonctionnement de mon algorithme pour des jeux de données relativement petits. Pour de vrais tests avec de jeux de données se rapprochant de la réalité, j'utiliserai les machines mises à disposition par l'école.

### 3 Controlabilité

Des messages seront affichés dans la console le long du programme afin d'en suivre la progression.

# E

# Guide d'installation et d'utilisation

Le projet évolue dans un environnement **Python** et la distribution **Anaconda**.

## 1 Installation

### 1.1 Python

Le programme évolue actuellement sur Python 3.6.8, téléchargeable ici : [Python 3.6.8](#), puis cliquez sur "*Windows x86 executable installer*" en bas de la page, puis exécutez le programme et suivez les instructions.

### 1.2 Anaconda

#### 1.2.1 Téléchargement

Anaconda étant une distribution Python, il faut télécharger la version adéquate à celle Python. Pour cela, allez à cette adresse : [Anaconda](#) et cliquez sur votre version de système d'exploitation qui convient, puis décompressez le tar.gz dans un dossier "Anaconda" à l'endroit où vous souhaitez l'installer.

#### 1.2.2 Variables d'environnement

Ensuite, si vous êtes sur Windows, alors il faudra ajouter Python en tant que variable d'environnement. Les instructions suivantes sont valables pour **Windows 10**.

Pour cela, cliquez sur l'icône Windows en bas à gauche et tapez "Modifier les variables d'environnement système". Dans l'onglet "Paramètres système avancés", cliquez sur *Variables d'environnement*.

Sélectionnez la variable **Path** et cliquez sur "Modifier", puis "Nouveau". Mettez ensuite le chemin complet vers le dossier *Anaconda* puis validez.



## 1.3 Tensorflow et Keras

Pour Tensorflow, choisissez entre la version CPU et GPU.

### 1.3.1 Version CPU

La version CPU est celle de base. Elle utilise les ressources de votre processeur et sera plus lente donc que la version GPU.

Le programme fonctionnent grâce à la bibliothèque **Tensorflow** et sa surcouche **Keras**. Pour cela, ouvrez "Anaconda Prompt". De là, tapez `conda install tensorflow`.

Pour Keras, effectuez là même commande en tapant `conda install keras` ou `conda install keras=2.2.4`.

### 1.3.2 Version GPU

La version GPU elle, fonctionnera avec votre processeur graphique / carte graphique. Elle sera bien plus rapide que la version précédente mais consommera plus de ressources.

#### Vérification GPU

Tout d'abord vérifiez bien que votre machine possède un GPU, de préférence NVidia. Les instructions suivantes fonctionnent pour les cartes graphiques NVidia.

#### Installation Tensorflow version GPU

Ensuite, installez la version GPU de Tensorflow. Pour cela, ouvrez "Anaconda Prompt".

De là, tapez `conda install tensorflow-gpu` ou `conda install tensorflow-gpu="1.13.1"` pour coller avec la version actuelle de ce projet.

Pour Keras, effectuez là même commande en tapant `conda install keras` ou `conda install keras=2.2.4`.

#### Installation Cuda

Enfin, il faut installer Cuda. Vous pouvez sélectionner votre configuration sur le lien suivant : [Télécharger CUDA](#). Le .exe pèse plus de 2Go. Exécutez l'installateur et suivez les instructions. Puis installez la librairie Cuda suivante :

<https://developer.nvidia.com/rdp/cudnn-archive#v7.5.2>

#### Vérification configuration

Une fois tout cela fait, vérifiez la bonne configuration. Voici le script Python pour tester :

```
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

Vous devriez obtenir quelque chose comme ceci :

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
```

```

incarnation: 8320990378049208634
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 4952267161
locality {
  bus_id: 1
  links {
  }
}
incarnation: 2490779436580148339
physical_device_desc: "device: 0, name: GeForce GTX 1060, pci bus id: 0000:01:00
]

```

La dernière étape est de vérifier si Keras fonctionne bien avec le GPU. Voici le script le vérifiant :

```

from keras import backend as K
K.tensorflow_backend._get_available_gpus()

```

## 1.4 Environnement de développement

Pour cela, le choix est libre. En effet, il est tout de même préférable d'utiliser la console pour exécuter le programme.

Mais vous pouvez installer PyCharm ([ici](#)) ou encore Rodeo ([ici](#)) par exemple.

## 2 Utilisation

Dans la partie suivante, des temps d'exécutions seront données. Ceux-ci ont été fait sur la configuration suivante qui servira de référence :

- PC Portable Omen by HP
  - Processeur : Inter Core i7-8750H 2.20 GHz
  - RAM : 12Go
  - Carte graphique : Nvidia GTX 1060, 4 Go dédiés
- 22 vidéos 1920x1080 d'une dizaine de secondes chacune.

### 2.1 Arborescence de fichiers

Pour que les scripts qui suivent puissent fonctionner tel quel, il faut que le dossier de travail respecte une certaine arborescence. Elle est ainsi :

```

script1.py
script2.py
scriptx.py
train
    videos
    images

```

Tout d'abord, il faudra un dossier *train* pour toutes les données d'entrée. Dans ce dossier *train* se trouvera deux sous-dossiers. Le dossier *videos* contient toutes les vidéos qui vont servir pour le training. Le dossier *images* aura aussi deux sous dossiers. Ces dossiers seront alimentés par la suite, mais doivent être présents pour être renseignés aux scripts suivants.  
Le dossier *test* n'est pas encore défini car cette partie n'est pas terminée.

## 2.2 Paramètres par défaut

Lorsque les prochains scripts vont être lancés, ils démarreront avec des paramètres. Ceux-ci peuvent-être définis dans la console, comme il sera décrit plus tard mais certains auront aussi des valeurs par défaut. Ces valeurs par défaut sont présentes dans une classe **Parameters**.

Dans le constructeur de la classe, il suffit de changer les paramètres de base du projet. Parmi ceux-ci nous retrouvons différents chemin d'accès, le coefficient multiplicateur de vidéo ou encore la taille des images d'entrée par exemple. Il suffit donc de changer une seule fois ces paramètres pour que tous les scripts les admettent comme valeur de paramètre par défaut.

## 2.3 Séquençage des vidéos

La première étape pour préparer les données est le séquençage de vidéos. Si toutes les vidéos sont bien dans **train/videos**, alors il suffit, dans le terminal, de lancer :

```
python sequence_videos.py -input_path path -output_path path -nb_frames number .
```

Les 3 paramètres trouvent leur valeurs par défaut dans la classe **Parameters**. Voici leur utilité :

- `input_path` : Chemin du dossier dans lequel se trouvent les vidéos
- `output_path` : Chemin du dossier de sortie dans lequel vont se retrouver les frames découpées.
- `nb_frames` : Nombre d'images consécutives à découper par vidéos.

Le script découpe donc les vidéos contenues dans **input\_path** et découpe chaque vidéos en **nb\_frames** images. Pour chaque vidéo, il y aura un dossier de créé dans **output\_path** du nom de la vidéo. Dans chacun de ses dossiers seront enregistrés toutes les frames correspondantes et seront nommées "frame\_x", x étant le numéro de la frame.

Cette opération prend, pour la configuration de référence, **30 secondes**.

## 2.4 Extraction des patches

Ce script aura pour but de découper des images en plusieurs images appelées **patches**. Le réseau de neurones traitera plusieurs les images par petits morceaux. Il est utilisable de la même manière que le précédent :

```
python extract_patches_HD.py -dir_path path -image_size size -r scale .
```

Les 3 paramètres trouvent leur valeurs par défaut dans la classe **Parameters**. Voici leur utilité :

- `dir_path` : Chemin du dossier dans lequel se trouvent les images en question et dans lequel seront les patches après opération.

- `image_size` : Taille des images d'entrée du réseau de neurones (donc la taille LR).
- `r` : Coefficient multiplicateur du réseau de neurones.

Le script découpe les images en patchs de taille `image_size * r` x `image_size * r`. Par exemple, si la taille d'entrée est de 17 \* 17 et que le coefficient `r` est de 3, les patchs seront de taille 51 \* 51. Les images découpées doivent se trouver dans le dossier indiqué par `dir_path`, donc logiquement le dossier de sortie du précédent script. Le script créera dans chaque dossier d'image un dossier par frame qui contiendra tous les patchs de ces images qui s'appelleront "patch\_x", de la même manière que pour les frames.

Cette opération prend, pour la configuration de référence, **environ 45 minutes**.

## 2.5 Entraînement du réseau

Le réseau est entraînable avec la commande dans le terminale suivante : `train.py`

Les arguments sont passables comme précédemment et en voici la description, sachant que toutes les valeurs par défaut sont dans `Parameters` :

- `r` : Coefficient multiplicateur du réseau de neurones.
- `HR_path` : Chemin du dossier dans lequel se trouvent les images HR. Typiquement : `train/images/HR`
- `LR_path` : Chemin du dossier dans lequel se trouvent les images LR. Typiquement : `train/images/LR`
- `colors_dimension` : Dimension de couleur. 3 si les vidéos sont en RGB, 1 si elles sont en niveau de gris.
- `learning_rate` : Taux d'apprentissage du réseau.
- `batch_size` : Taille des batchs, ou lots. Le réseau traitera les images lots par lots.
- `epochs` : Nombre d'époques de l'entraînement.
- `stride` : Valeur du pas de la convolution. Dans ce modèle, elle vaut 14 par défaut.

# F

## Guide développeur

### Versions

Keras :

### Parameters.py

La classe **Parameters** recense tous les paramètres propres à un réseau, à un projet.

**videos\_path** : Chemin absolu ou relatif du dossier où se trouvent les vidéos. Pour exemple, *train/videos*. (Chaîne de caractère)

**HR\_path** : Chemin absolu ou relatif du dossier où se trouvent les images haute résolution. Pour exemple, *train/images/HR*. (Chaîne de caractère)

**LR\_path** : Chemin absolu ou relatif du dossier où se trouvent les images basses résolutions. Pour exemple, *train/images/LR*. (Chaîne de caractère)

**r** : Facteur d'augmentation de résolution voulue. (Entier non signé)

**nb\_frames\_per\_videos** : Nombre de frames consécutives que l'on va extraire par vidéos (Entier)

**input\_size** : Taille des images d'entrées pour le réseau. (Entier non signé)

**colors\_dimension** : Nombre de canaux de couleur des vidéos. 1 pour noir et blanc, 3 pour RGB classique (Entier non signé)

**learning\_rate** : Taux d'apprentissage du réseau (float)

**batch\_size** : Taille des lots du réseau. (Entier non signé)

**epochs** : Nombre d'époques de l'entraînement du réseau. (Entier non signé)

Pour les utiliser, il suffit d'importer la classe puis de faire comme suit par exemple :

```
param = Parameters.Parameters()
```

Puis d'appeler les arguments comme tel :

```
upscaling_factor = param.r
```

## BaseModel.py

BaseModel peut-être assimilé à un squelette de modèle de réseau de neuronne. Il implante les fonctions les plus simples et communes à tous les modèles, de même pour les paramètres.

```
__init__(self,...)
```

Les paramètres suivants ont déjà été expliqués dans la partie précédente : **r**, **input\_size**, **colors\_dimension**, **learning\_rate**, **batch\_size** et **epochs**.

**is\_training** : Défini au lancement si le modèle a besoin d'être entraîné ou non. Si oui, alors on le sauvegarde, si non, alors on charge le modèle déjà entraîné pour gagner du temps.

```
build_model(self)
```

Prototype de fonction qui sera bien propre à chaque modèle.

**Retourne** : Rien

```
save(self, name)
```

Sauvegarde le modèle entraîné en format JSON dans le dossier **model**, sous le nom **name.json**. Les poids sont eux sauvegardés dans le dossier **model** sous le nom : **name\_weight.hdf5**.

**name** : Nom donné au réseau (Chaîne de caractère).

**Retourne** : Rien

```
load(self, name)
```

Charge le modèle en chargeant le fichier JSON et HDF5 vus précédemment.

**name** : Nom donné au réseau (Chaîne de caractère).

**Retourne** : Rien

```
train(self)
```

Entraîne le modèle avec la méthode **fit**. Affiche aussi le résumé du modèle et de ses couches. Si **if\_training** est à vrai, alors on save le model à la fin de l'entraînement.

**Retourne** : Objet *History*.

## ESPCN.py

Cette classe définit notre modèle, le modèle ESPCN. Elle hérite naturellement de la classe BaseModel.

On définit 5 paramètres (**n1**, **n2**, **f1**, **f2**, **f3**) qui correspondent aux paramètres des couches du réseau qui vont être vues plus tard.

`__init__(self,...)`

Le constructeur possède les mêmes paramètres que `BaseModel` et appelle donc le constructeur de cette classe parent. Il possède ensuite deux listes :

**patches\_HR\_pathes\_train** : C'est une list de string qui va contenir tous les chemins vers les fichiers images des patches qui vont être destinés à l'entraînement. En effet, il serait trop lourd de stocker directement les images en elles-mêmes. On stock les chemins (beaucoup moins volumineux) en chaîne de caractère pour ensuite les appeler à partir de là.

**patches\_HR\_pathes\_test** : C'est la même chose que l'attribut précédent sauf qu'il va concerner les patches pour la partie validation.

Ces deux listes sont remplies par la fonction **fillListPathPatches**.

`build_model(self)`

Cette fonction construit le modèle du réseau de neurone avec deux convolution et une convolution transposée qui remplace pour l'instant la convolution sous-pixel.

**Retourne** : L'objet Model correspondant au modèle construit.

`generateur(self)`

Pour ne pas stocker une immense liste d'images, on va donner à s'entraîner des batches, ou des lots. Ces lots seront générés par ce générateur. Il prend simplement aléatoirement *batch\_size* chemins dans *patches\_HR\_pathes\_train*, lit les images, les convertit et les renvoie grâce au mot clé **yield**. A chaque fois que cette fonction sera appelée, elle reprendra son exécution au **while True** et renverra des nouvelles valeurs avec le **yield**.

**trainOrValidate** : Determine si il faut piocher dans la liste de training ou de validation.

**Retourne** : Une liste d'images en nparray.

`trainWithGenerator(self)`

Entraîne le réseau avec la fonction `fit_generator` qui prend en paramètre un générateur (donc la fonction précédente). Il sauvegarde le modèle si il a besoin d'être entraîné et affiche le résumé du modèle avec ses couches.

**Retourne** : Un objet History.

`evaluateWithGenerator(self)`

Evalue le réseau avec la fonction `evaluate_generator` qui prend en paramètre un générateur (donc la fonction précédente). Il affiche la metric obtenue.

**Retourne** : Rien

`fillListPathPatches(self, path_patches_HR, percentage_train)`

Remplie deux listes de chemins d'images. La liste train sera composée de  $(percentage\_train * 100)\%$  de la liste *path\_patches\_HR*. Le reste ira dans la liste d'évaluation.

**path\_patches\_HR** : Chemin de toutes les images HR. (Récupérable dans les attributs de la

classe **Parameters**. (Chaîne de caractère)

**percentage\_train** : Pourcentage de patches qui iront en training. (Entier entre 0 et 1).

**Retourne** : Deux listes de chaînes de caractères correspondant aux différents chemins vers les fichiers images.

```
_upscale_block(self, init, r)
```

Effectue une convolution transposée.

**init** : Tenseur 4D correspondant à la couche du réseau

**r** : Facteur d'augmentation de résolution (Entier)

**Retourne** : Tenseur 4D correspondant à la couche sortante du réseau

```
load(self,name)
```

Appelle le constructeur de la classe mère et fonctionne de la même manière.

```
save(self,name)
```

Appelle le constructeur de la classe mère et fonctionne de la même manière.

```
sequence_videos.py
```

Les 3 paramètres du script trouvent leur valeurs par défaut dans la classe **Parameters**. Voici leur utilité :

- **input\_path** : Chemin du dossier dans lequel se trouvent les vidéos
- **output\_path** : Chemin du dossier de sortie dans lequel vont se retrouver les frames découpées.
- **nb\_frames** : Nombre d'images consécutives à découper par vidéos.

Le script découpe donc les vidéos contenues dans **input\_path** et découpe chaque vidéos en **nb\_frames** images. Pour chaque vidéo, il y aura un dossier de créé dans **output\_path** du nom de la vidéo. Dans chacun de ses dossiers seront enregistrés toutes les frames correspondantes et seront nommées "frame\_x", x étant le numéro de la frame.

```
videosToSequences(input_path, output_path)
```

Transforme une vidéo en une séquence aléatoire de 30 images consécutives

**input\_path** : Chemin du dossier où se trouvent toutes les vidéos - *Chaîne de caractère*

**output\_path** : Chemin du dossier où se trouveront les images - *Chaîne de caractère*

**Retourne** : Rien



```
downscale(input_path, output_path, scale)
```

Downscale les images et les enregistre sur le disque. **input\_path** Chemin du dossier où se trouvent toutes les images - *Chaine de caractère*

**output\_path** : Chemin du dossier où iront les séquences. Un dossier par vidéo sera créé à cet endroit. - *Chaine de caractère*

**scale** : Multiplicateur de réduction de résolution - *Entier non signé*

**Retourne** : Rien

## extract\_patches\_HD.py

C'est un script qui va, à partir du dossier *dir\_path*, extraire des patchs (morceaux d'image) de toutes les images contenues dans ce dossier.

La **taille des patchs** est gérée dans la fonction *image.\_extract\_patches\_2d* en deuxième argument. Le **nombre de patchs** à extraire est gérée dans la fonction *image.\_extract\_patches\_2d* en troisième argument. Ici, on prend la taille de l'image de base, qu'on divise par la taille du patch pour obtenir le nombre de patchs que peut contenir l'image.

**-dir\_path** : Chemin d'accès des images HR. Valeur par défaut définie à partir de la classe *Parameters*.

**-r** : Facteur d'augmentation de résolution Valeur par défaut définie à partir de la classe *Parameters*.

**-input\_size** : Taille des images d'entrée du réseau. Valeur par défaut définie à partir de la classe *Parameters*.

Le script créera dans chaque dossier d'image un dossier par frame qui contiendra tous les patchs de ces images qui s'appelleront "patch\_x", de la même manière que pour les frames.

## train.py

Le script crée une instance du modèle ESPCN, l'entraîne puis l'évalue. Voici la définition des paramètres du script :

- **r** : Coefficient multiplicateur du réseau de neurones. (Entier)
- **HR\_path** : Chemin du dossier dans lequel se trouvent les images HR. Typiquement : **train/images/HR** (Chaine de caractère)
- **LR\_path** : Chemin du dossier dans lequel se trouvent les images LR. Typiquement : **train/images/LR** (Chaine de caractère)
- **colors\_dimension** : Dimension de couleur. 3 si les vidéos sont en RGB, 1 si elles sont en niveau de gris. (Entier)
- **learning\_rate** : Taux d'apprentissage du réseau. (Entier)
- **batch\_size** : Taille des batchs, ou lots. Le réseau traitera les images lots par lots. (Entier)
- **epochs** : Nombre d'époques de l'entraînement. (Entier)
- **stride** : Valeur du pas de la convolution. Dans ce modèle, elle vaut 14 par défaut. (Entier)
- **is\_training** : Définit si le modèle doit s'entraîner ou pas.

**predict\_video.py**

Les paramètres du script sont exactement les mêmes que pour **train.py**

**sequencesToVideo(images\_paths, video\_path, size)**

Séquence les vidéos en plusieurs frames.

**images\_patches** : Chemin où les images iront

**video\_path** : Chemin vers la vidéo à extraire

**size** : Taille des patches à extraire

**sorted\_alphanumeric(data)**

Classe les données par ordre alphabétique en prenant en compte l'alphanumérique **data** :  
Données à trier

**generator(patches\_paths, batch\_size)**

Générateur qui va servir pour les prédictions.

**patches\_paths** : Chemin des patches

**batch\_size** : Taille du lot

**main**

Effectue le préprocessing et la prédiction. Le reste des opérations est mis en commentaire car c'est en cours.



## Bibliographie

- [1] *Detail-revealing Deep Video Super-resolution*. URL : [http://openaccess.thecvf.com/content\\_ICCV\\_2017/papers/Tao\\_Detail-Revealing\\_Deep\\_Video\\_ICCV\\_2017\\_paper.pdf](http://openaccess.thecvf.com/content_ICCV_2017/papers/Tao_Detail-Revealing_Deep_Video_ICCV_2017_paper.pdf).
- [2] *Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network*. URL : [https://www.cv-foundation.org/openaccess/content\\_cvpr\\_2016/papers/Shi\\_Real-Time\\_Single\\_Image\\_CVPR\\_2016\\_paper.pdf](https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Shi_Real-Time_Single_Image_CVPR_2016_paper.pdf).
- [3] *Real-Time Video Super-Resolution with Spatio-Temporal Networks and Motion Compensation*. URL : [http://openaccess.thecvf.com/content\\_cvpr\\_2017/papers/Caballero\\_Real-Time\\_Video\\_Super-Resolution\\_CVPR\\_2017\\_paper.pdf](http://openaccess.thecvf.com/content_cvpr_2017/papers/Caballero_Real-Time_Video_Super-Resolution_CVPR_2017_paper.pdf).

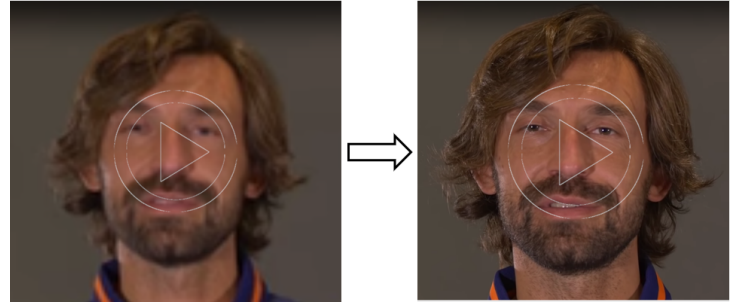
# Problème de super-résolution par application de CNN

Vincent Liuzzi

Encadrement : Donatello Conté et Maxime Martineau

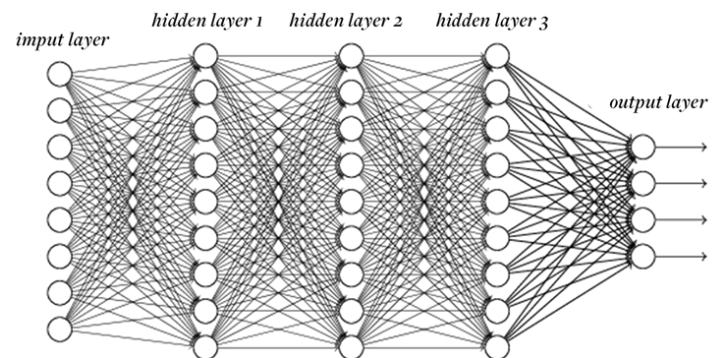
## Objectif

Transformer une vidéo basse résolution en vidéo haute résolution. Cet outil peut être utilisé sur des webcams (dont la qualité d'image est souvent médiocre) pour pouvoir analyser le dilatement de la pupille et détecter des émotions.



## Mise en oeuvre

Utilisation du Deep Learning grâce à l'application d'un réseau de neurones à convolution, ou CNN.



## Conclusion

Application prometteuse et améliorable avec quelques ajustements

# Problème de super-résolution par application de CNN

Vincent Liuzzi

Encadrement : Donatello Conté et Maxime Martineau

## Objectif

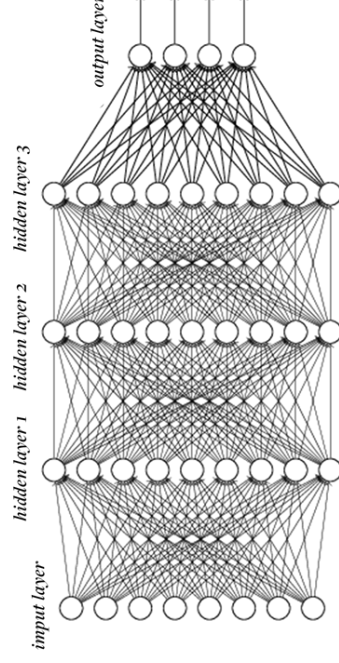
Transformer une vidéo basse résolution en vidéo haute résolution. Cet outil peut être utilisé sur des webcams (dont la qualité d'image est souvent médiocre) pour pouvoir analyser le dilatement de la pupille et détecter des émotions.

## Mise en oeuvre

Utilisation du Deep Learning grâce à l'application d'un réseau de neurones à la convolution, ou CNN.

## Conclusion

Application prometteuse et améliorable avec quelques ajustements



# Problème de super-résolution par application de CNN

## Résumé

Ce projet est un projet accès recherche. Il consiste à résoudre un problème de super-résolution appliqué à la vidéo. La super-résolution est le fait de transformer une image ou une vidéo basse résolution (avec peu de pixels) en un contenu de haute résolution. Cela se fait grâce à un réseau de neurone. C'est un système informatique inspiré du cerveau humain qui lui permet d'apprendre tout seul et d'effectuer des tâches complexes.

## Mots-clés

Recherche - Apprentissage profond - Réseau de neurone - Convolution - CNN - Super résolution - Vidéo

## Abstract

This project consists in solving a super-resolution problem applied to the video. Super-resolution is the process of transforming a low-resolution image or video (with few pixels) into high-resolution content. This is done through a neural network. It is a computer system inspired by the human brain that allows him to learn on his own and perform complex tasks.

## Keywords

Research – Deep Learning – Neural network – Convolution - CNN - Super resolution - Video

**Tuteurs académiques**

Donatello CONTÉ

Maxime MARTINEAU

**Étudiant**

Vincent LIUZZI (DI5)