

Projet Informatique 1ère année  
PRO 3600

Détection des styles architecturaux par une Intelligence  
Artificielle

Membres du projet : Elliot Cole, Brun Baptiste, Naama Mohamed, Abri  
Saad, Mateo Zoughebi

Mohamed Sellami PRO 3600- Département INF

Version 18/02/2022



# Table des matières

<b>1 Introduction</b>	<b>2</b>
<b>2 Cahier des charges</b>	<b>3</b>
2.1 Partie application . . . . .	3
2.2 Partie IA. . . . .	4
<b>3 Développement</b>	
3.1 Analyse du problème et spécification fonctionnelle . . . . .	
3.2 Conception préliminaire . . . . .	
3.3 Conception détaillée . . . . .	
3.4 Codage . . . . .	
3.5 Tests unitaires . . . . .	
3.6 Tests d'intégration . . . . .	
3.7 Tests de validation . . . . .	
<b>4 Manuel utilisateur</b>	
4.1 Production de l'exécutable . . . . .	
4.2 Réalisation des tests . . . . .	
4.3 Utilisation. . . . .	
<b>5 Conclusion</b>	
<b>6 Annexe</b>	
Annexe 1 . . . . .	<b>7</b>

# Introduction

Ce document décrit le développement de l'application *Scan'INT* permettant l'identification de styles architecturaux sur une photo, dans le cadre du module PRO 3600. Le développement de l'application implique deux domaines d'étude: L'Intelligence Artificielle ( IA ) et la création d'une application web. L'intelligence artificielle développée permet d'ajouter une nouvelle dimension aux applications d'identification des monuments par géolocalisation. Dans le cadre de ce module nous nous restreindrons à l'identification de 25 styles architecturaux pour la création de l'IA.

## 2 Cahier des charges

### 2.1 Partie application web (front-end):

- Fonctionnalités :

L'application aura pour but principal de scanner un bâtiment de sorte à obtenir son style architectural.

Cette application web aura:

- Une interface dynamique qui propose à l'utilisateur d'ajouter une photo d'un bâtiment depuis son explorateur de fichiers
- A partir de cette photo, l'analyser et sortir en réponse le style architectural du bâtiment que prédit le réseaux de neurones en 'back-end'

- Contrainte:

- Apprentissage du langage de programmation : HTML/CSS/JavaScript
- Gestion de l'Intelligence artificielle depuis l'application web via un serveur python par la bibliothèque Flask

- Livrable :

- Application web développée en JavaScript en local(annexe 1).

### 2.2 Partie IA

- Contraintes :

- Apprendre de façon complète un langage de programmation: Python avec les bibliothèques suivantes : fastai (torch, torchvision), numpy, json,time, os, random,matplotlib. Ainsi que le Notebook collaboratif : GoogleColab
- Constitution d'une base de donnée assez large pour pouvoir entraîner notre IA (database trouvé sur Kaggle)

- Fonctionnalités :

- Pouvoir reconnaître le style architectural d'un bâtiment (dans un premier temps un édifice religieux) à partir d'une photo prise dans l'application mobile.

- Livrable :
  - Réseau de neurones convolutifs fonctionnel sous Python à l'aide de Fastai.

### 3) Développement

#### Conception préliminaire

A haut niveau, le logiciel se décompose en deux parties, une partie “front-end” : c’est la partie IHM de l’application, elle sera à terme une interface logiciel sur web écrite en JavaScript/HTML..

La partie “back-end” est notre réseau de neurones convolutifs qui permettra de prédire l’architecture d’une photo que l’on enverra au programme.

Enfin une partie “intégration” sera mise en place pour faire le raccordement : ce sera un serveur python écrit en Flask

Les structures de données utilisées seront ici des images d’architectures que l’on trouvera sur <https://www.kaggle.com/brsdincer/architecture-art-cycleGAN-process/data> . Elles seront ensuite adaptées au format 224x224 pixels par souci d’homogénéité.

Ici un descriptif des modules de notre logiciel :

#### ***Partie Back-end :***

**Fichier Entraînement\_IA\_Bâtiment.ipynb**

*Importation des modules:*

Tout d'abord on commence par l'importation des différents modules que l'on va utiliser, notamment la bibliothèque Fast AI.

Fast AI va nous permettre d'avoir des résultats plus rapidement en termes de DeepLearning. C'est ce qui est utilisé lorsqu'on traite d'intelligence artificielle standard

### Etude des données :

La deuxième partie nous permet de télécharger notre dataset via le site Kaggle. Cela passe donc par le téléchargement d'un token API qui permet à notre IDE de se connecter sur Kaggle. Ensuite il suffit d'extraire notre fichier, et spécifier l'emplacement son emplacement pour que notre environnement de travail puisse y avoir accès.

### Augmentation de la base de données :

Une fois le dataset téléchargé et importé, on peut augmenter notre base de données. En effet, afin d'optimiser nos chances de réussite, il est nécessaire d'avoir une base de donnée conséquentes.

Pour cela il suffit d'appliquer des transformations "primaire" à l'image (inclinaison, rotation, symétrie...)

### Constitution de notre réseau de neurone convolutifs :

On va utiliser un réseau de neurones largement utilisé et déjà pré-entraînés sur notre modèle : resnet34. Ce réseau de neurones convolutifs ayant 34 couches de neurones. Celui-ci s'entraîne sur notre dataset d'image augmentée.

## **Partie Front-end :**

### **Fichier app.py**

Le serveur python écrit en Flask sera composé des modules suivants :

Une fonction `validate_image(stream)` qui permettra de signaler un message d'erreur si l'utilisateur envoie une image du mauvais format (autre format que jpg etc.)

Une fonction `home()` lié à `@app.route('/')` : cette fonction retourne une template en sortie celle de `home.html` : chaque fois que l'utilisateur va sur le site il va se retrouver sur cette page

Une fonction `upload_files()` lié à `@app.route('/app.html, methods=[POST])` : cette fonction s'exécute lorsque l'utilisateur envoie une requête POST, quand il clique sur upload une image, celle-ci s'enregistre dans le répertoire courant

Une fonction `def solutions()` lié à `@app.route("/analyze", methods=[POST])` qui permettra d'analyser la photo et de sortir la prédiction du bâtiment en sortie

### Conception détaillée :

Pour la partie "**back-end**" (fichier `Entrainement_IA_Bâtiment.ipynb`), voici un descriptif détaillé des différents modules :

Outre l'importation des différents modules (`fastai`, `matplotlib` etc.), il faut se rendre sur : <https://www.kaggle.com/nomutilisateur/account> puis sur "Create New API Token".

La méthode de `os` `upload` permet d'importer le token `kaggle.json` :

```
uploaded = files.upload()
```

On crée le répertoire où l'on va mettre notre dataset

```
! mkdir -p ~/.kaggle/  
! mv kaggle.json ~/.kaggle/
```

C'est le chemin, d'accès de notre dataset

```
chemin = Config.data_path()/'architecture'  
chemin.mkdir(parents=True, exist_ok=True)
```

On télécharge le dataset

```
! kaggle datasets download -d wwymak/architecture-dataset
```

On s'assure que le dataset est bien téléchargé, en le voyant apparaître dans le répertoire de travail

```
os.listdir()
```

Une fois bien téléchargé, on le met dans le répertoire, en dezipant

```
! unzip -q -n architecture-dataset.zip -d {chemin}
```

### Etude des données :

Nous allons utiliser le jeu de données Architecture de Zhe Xu, qui présente 25 styles d'architecture. Notre modèle devra apprendre à faire la différence entre ces 25 catégories distinctes.

```
os.listdir('/root/.fastai/data/architecture/arcDataset')
```

Le chemin de notre data set est celui ci-dessous :

```
chemin = '/root/.fastai/data/architecture/architectural-styles-dataset'
```

La première chose que nous faisons lorsque nous abordons un problème est d'examiner les données. Nous devons toujours bien comprendre la nature du problème et l'aspect des données avant de pouvoir déterminer comment le résoudre. Examiner les données signifie comprendre comment les répertoires de données sont structurés, quelles sont les étiquettes et à quoi ressemblent quelques exemples d'images. Cela va permettre de sélectionner des images différentes, afin de s'assurer que nos résultats ne sont pas le fruit du hasard :

```
np.random.seed(42)
```

On va transformer nos images afin d'augmenter notre dataset, on en parlera plus tard :

```
tfms = get_transforms(do_flip=True, flip_vert=False, max_rotate=10,  
max_zoom=1.1, max_lighting=0.2, max_warp=0.2, p_affine=0.75, p_lighting=0.75)
```



### Affichage des images, et études brèves du dataset

Dans cet ensemble de données particulier, les étiquettes sont stockées dans le nom du dossier qui contient les images de chaque classe. Nous devons les extraire pour pouvoir classer les images dans les bonnes catégories. La bibliothèque fastai à une fonction faite pour cela, "ImageDataBunch.from\_folder".

```
data = ImageDataBunch.from_folder(chemin, train=".", valid_pct=0.2,  
    ds_tfms=tfms, size=224, num_workers=4, padding_mode='reflection',  
    bs=64).normalize(imagenet_stats)
```

On affiche 9 images de notre set aléatoirement :

```
data.show_batch(rows=3, figsize=(9, 9))
```

On regroupe les différentes catégories ensemble :

```
data.classes, data.c, len(data.train_ds), len(data.valid_ds)
```

### Augmentation de notre base de donnée

La fonction 'get transforms' nous a permis d'obtenir plus d'images pour l'entraînement, dans ce cas nous avons 9588 images au lieu des 4979 de l'ensemble de données original.

Ceci est réalisé en faisant quelques changements aux images et en les traitant comme des images nouvelles.

Retournement (juste horizontal), zoom, lumière, rotation, etc...

```
def _plot(i,j,ax):  
    x,y = data.train_ds[3]  
    x.show(ax, y=y)
```

```
plot_multi(_plot, 3, 3, figsize=(8,8))
```

### Apprentissage : Constitution de notre réseau de neurones convolutifs

Maintenant que notre dataset d'images à été traité, nous pouvons dès à présent l'utiliser pour entraîner notre première couche de réseaux de neurones convolutifs. Pour cela, on va utiliser un réseau de neurones largement utilisé et déjà pré-entraînés sur notre modèle : resnet34. Ce réseau de neurones convolutifs ayant 34 couches de neurones

Ici, on va utiliser la fonction "cnn\_learner" de la bibliothèque pytorch : Elle prend en entrée l'objet "data" (qui est notre dataset traité et normalisé précédemment), le nom du modèle (resnet34) et les paramètres que l'on cherche à observer (ici "l'accuracy" (précision) et le taux d'erreur)

```
learn = cnn_learner(data, models.resnet34, metrics=[accuracy])
```

`learn.fit_one_cycle(5)` C'est cette fonction qui va appliquer à notre variable learn, un entraînement de 5 epochs (5 tours)

`learn.save('stage-1-resnet34')` Par la fonction `.save()` de f permet d'enregistrer le réseaux "learn" dans le répertoire courant

`learn.export()` Maintenant on exporte notre modèle dans le dossier root, il est entraîné et doit être utilisé sur des images tests

`!cp /root/.fastai/data/architecture/export.pkl .` #On le copie dans le répertoire courant

`image = open_image(imagetest.jpg)` On ajoute manuellement, dans le répertoire courant, l'image que l'on veut prédire (à renommer imagetest impérativement)

`learn = load_learner(chemin)` La méthode `load_learner` de fastai prend en paramètre un chemin et renvoie le modèle dans une variable learn

`prediction, pred_idx, outputs = learn.predict(image)` #la modèle d'instance `predict(image)` renvoie notre prédiction sous la forme d'un objet de la classe `Prediction` (bibliothèque fastai)  
`prediction.obj` La méthode `obj` de `prediction` renvoie sous forme de string, la prédiction

Pour la partie "**intégration**" (fichier app.py), voici un descriptif détaillé des différents modules :

La fonction `validate_image(stream)` permet de retourner une erreur si le format de l'image est différent de jpeg ou jpg :

```
def validate_image(stream):
    header = stream.read(512)
    stream.seek(0)
    format = imghdr.what(None, header)
    if not format:
        return None
```

```
return '.' + (format if format != 'jpeg' else 'jpg')
```

La fonction `too_large` permet de rediriger l'erreur HTML "ErreroHandler 413", on écrit à la place "File is too large" avec l'id de l'erreur

```
@app.errorhandler(413)
def too_large(e):
    return "File is too large", 413
```

```
#=====mise de l'image dans le dossier=====
```

La fonction `home` retourne une template en sortie celle de `home.html` : chaque fois que l'utilisateur va sur le site il va se retrouver sur la page `home.html`:

```
@app.route('/')
def home():
    return render_template('/home.html')
```

La fonction `index()` est similaire à la fonction précédente, elle s'exécute lors d'une requête GET (c'est à dire lorsque l'utilisateur va rejoindre la page par URL) et renvoie vers le template de `app.html`

```
@app.route('/app.html')
def index():
    files = os.listdir(app.config['UPLOAD_PATH'])
    return render_template('app.html', files=files)
```

La fonction `upload_files()` est exécuté lorsqu'il y a une requête POST c'est à dire lorsque le user upload une image, celle ci se retrouve dans le répertoire courant au niveau local.

```
@app.route('/app.html', methods=['POST'])
def upload_files():
    uploaded_file = request.files['file']
    filename = secure_filename(uploaded_file.filename)
    if filename != "":
        file_ext = os.path.splitext(filename)[1]
        if file_ext not in app.config['UPLOAD_EXTENSIONS'] or \
            file_ext != validate_image(uploaded_file.stream):
            return "Invalid image", 400
        uploaded_file.save(os.path.join(app.config['UPLOAD_PATH'], filename))
    return "", 204
```

La fonction `analyze` est exécutée lorsqu'il y a une requête POST c'est à dire lorsque le user clique sur Analyser, c'est la que la méthode prédiction est envoyé, par la méthode `model.prediction` de `fastai`. On retourne cette réponse sur JavaScript

```
@app.route('/upload', methods=["POST"])
def analyze():
```

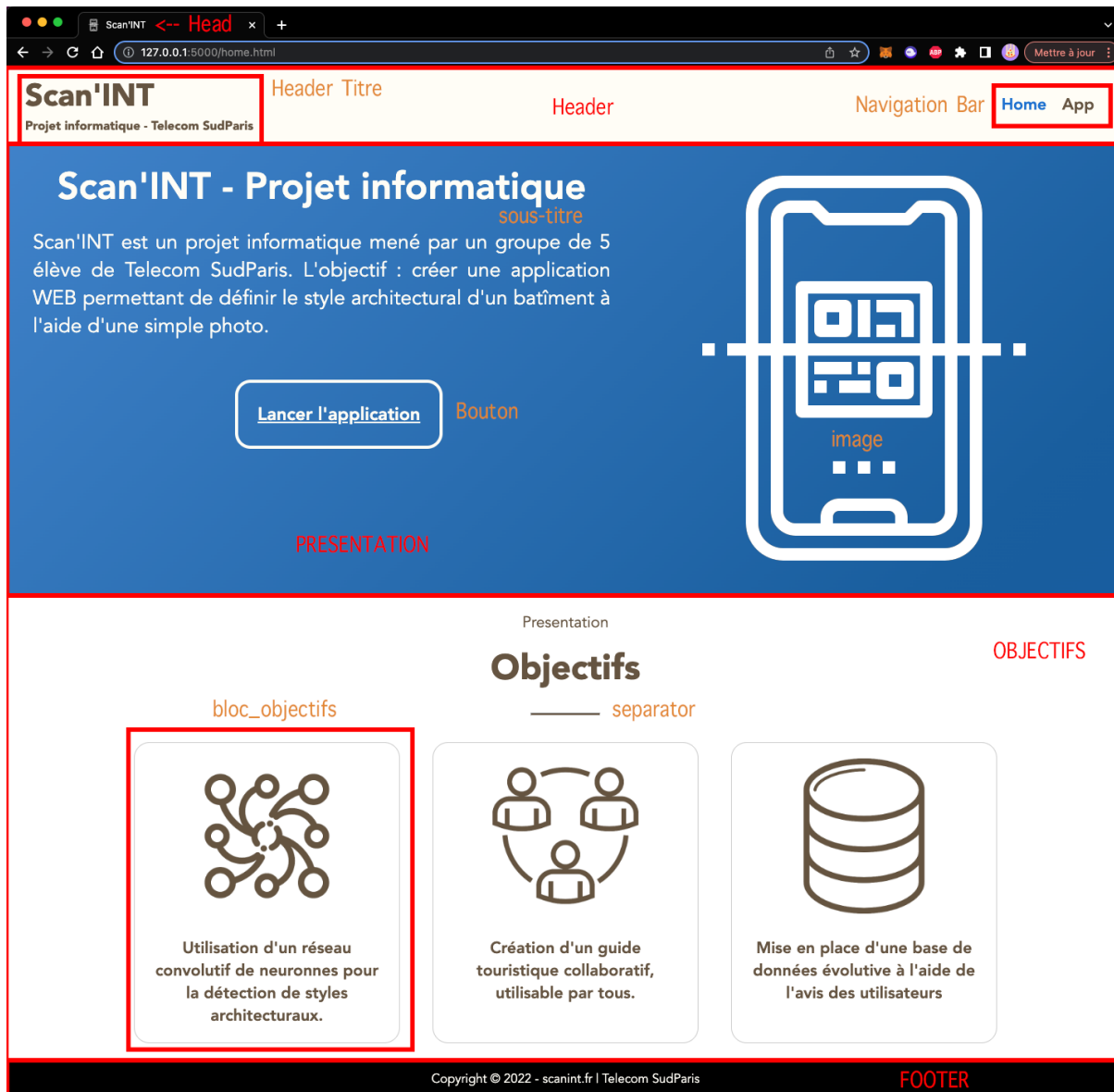
```
    img = open_image(io.BytesIO(file)) #On met l'image dans le bon format pour que le model puisse l'utiliser
    prediction = model.predict(img)[0]

    response = {"result": str(prediction)}
    return jsonify(response)
```

Pour la partie “**front-end**” (fichier home.html, home.css, app.html, app.css, dropzone.js), voici une présentation du site :

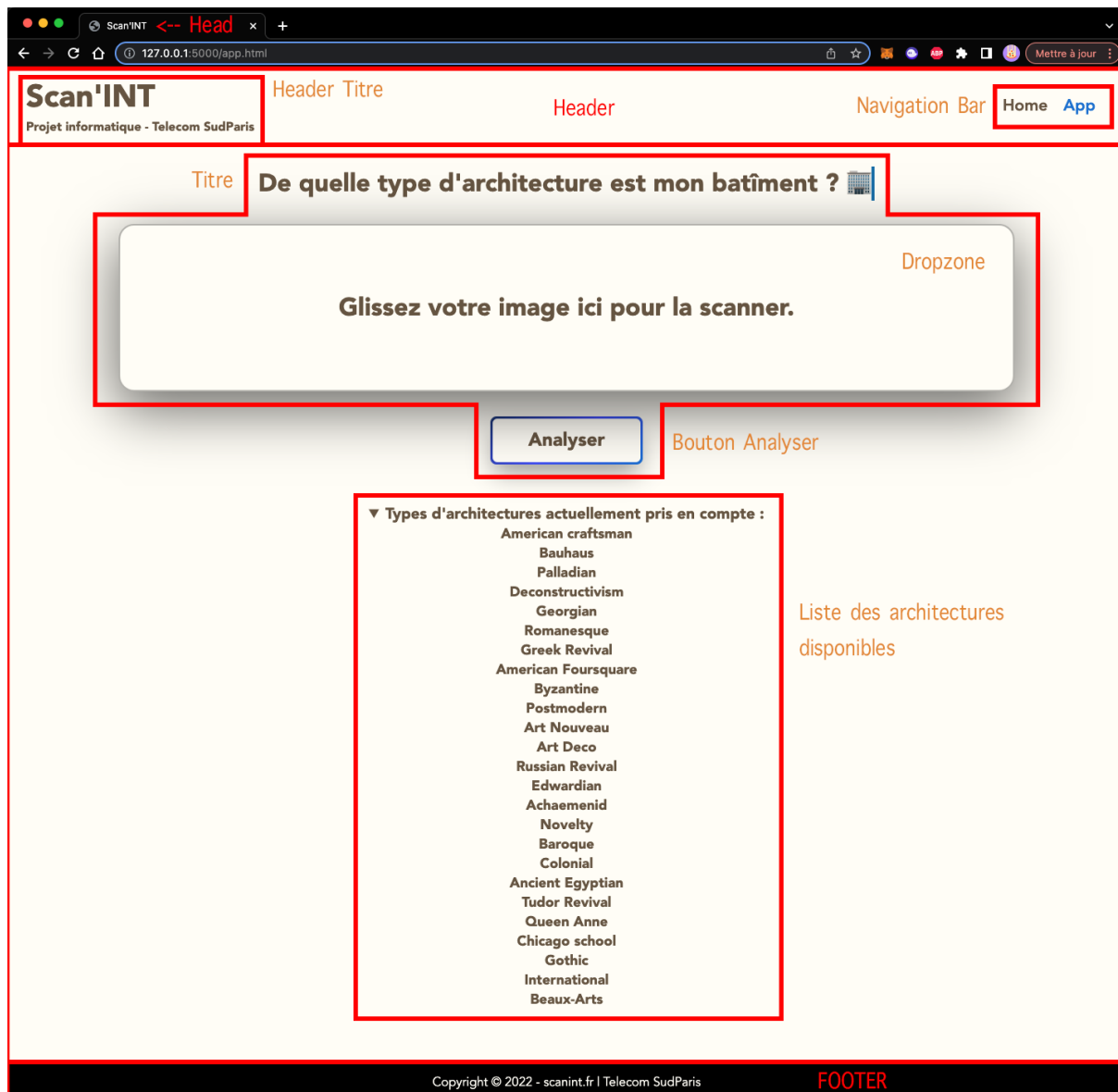
*La légende des photos ci-dessous fait référence aux nom de class/balises utilisées dans le code HTML/CSS.*

- **Page d'accueil : (home.html + home.css)**



Sur cette page, on peut trouver une brève présentation de Scan'INT ainsi que les 3 principaux objectifs de l'application. On y trouve aussi une “navigation bar” qui permet de naviguer entre les deux pages du site : la page d'accueil ‘home’ et la page application ‘App’. On peut aussi rejoindre cette seconde page à l'aide du bouton “lancer l'application” qui renvoie lui aussi sur la page application.

- **Page Application : (app.html + app.css + dropzone.js)**

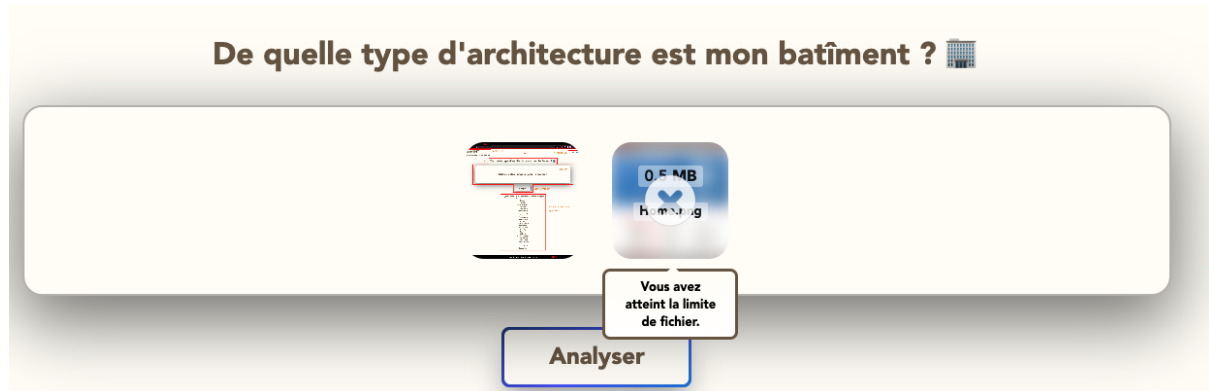


Sur cette page se trouve le coeur du projet, l'application web.

On y retrouve :

- le header avec la même navigation bar que sur la page d'accueil
- un titre animé qui rappelle la problématique à laquelle l'application tente de répondre
- une "dropzone", zone dans laquelle l'utilisateur pourra déposer l'image à étudier. Cette image est automatiquement transférée dans un dossier local de l'utilisateur nommé "upload" qui est ensuite utilisé pour la partie prédiction par l'IA.
- un bouton analyser, qui lance donc le scan de la photo qui a été préalablement téléchargée dans la dropzone par l'utilisateur
- une liste déroulante récapitulant tous les types d'architectures actuellement pris en charge par l'application

Le fichier “dropzone.js” permet notamment au site d’afficher quel fichier a déjà été téléchargé, quel type de fichier on peut utiliser, et bloque l’user lorsqu’il essaye d’en télécharger plus d’un :



Le code source du site se trouve sur le gitlab, dans les dossiers templates (.html) et static (.css / .js).

### Codage :

Les codes sources commentés des parties “back-end” et “front-end” ainsi que “integration” sont disponibles dans l’annexe ( cf annexe)

### Tests unitaires:

Pour le fichier Entrainement\_IA\_Bâtiment (2).ipynb :

Nous avons opéré plusieurs test unitaires afin de déterminer si le réseaux de neurones fonctionne correctement et est sans bug, à quel point il est efficace et combien de bonnes prédictions permet-il de faire.

La méthode *learn.fit\_one\_cycle* dans notre exemple a été entraînée sur 4 “epochs”, selon le CPU, le réseau apprend en 18 minutes et 43 secondes. De plus, on arrive a une précision allant jusqu’à 79%. On remarque aussi que la fonction de coût des jeux de données de validation et d’entraînement diminue de tours en tours, ce qui est bon signe (moins en moins de mauvaises prédictions).

Epochs = nombre de "tours" du réseaux de neurones que le modèle fait

Valid\_loss = résultat de la fonction de coût sur le jeu de données de validation

Train\_loss = résultat de la fonction de coût sur le jeu de données d'entraînement

Accuracy = précision du modèle : c'est la proportion de bonnes prédiction

Time = temps pour chaque epoch

epoch	train_loss	valid_loss	accuracy	error_rate	time
0	2.183656	1.135984	0.670840	0.329160	04:42
1	1.321748	0.896148	0.728743	0.271257	04:39
2	1.007899	0.704236	0.782994	0.217006	04:40
3	0.814357	0.672641	0.790297	0.209703	04:42

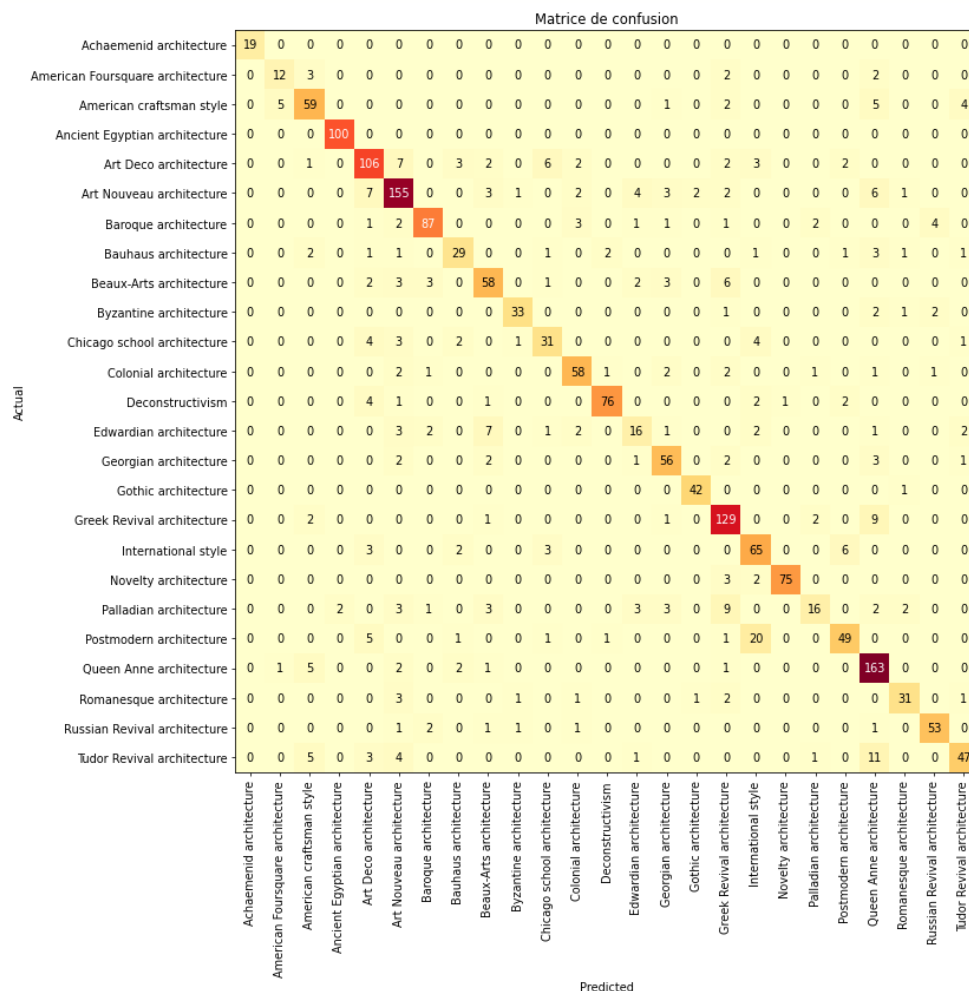
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader

On va utiliser les méthodes de tests suivantes pour approfondir :

`interpretation = ClassificationInterpretation.from_learner(learn)` La bibliothèque fastai fournit la classe `ClassificationInterpretation`, on crée notre objet.

`interpretation.plot_confusion_matrix(figsize=(12,12), title = "Matrice de confusion", cmap = "YlOrRd", dpi=70)` Méthode d'instance permettant d'afficher la matrice de confusion, le coefficient (i,j) de cette matrice affiche combien de fois le j-ème élément a été prédit alors que la véritable valeur était le i-ème élément.

On obtient ainsi la matrice de confusion suivante :

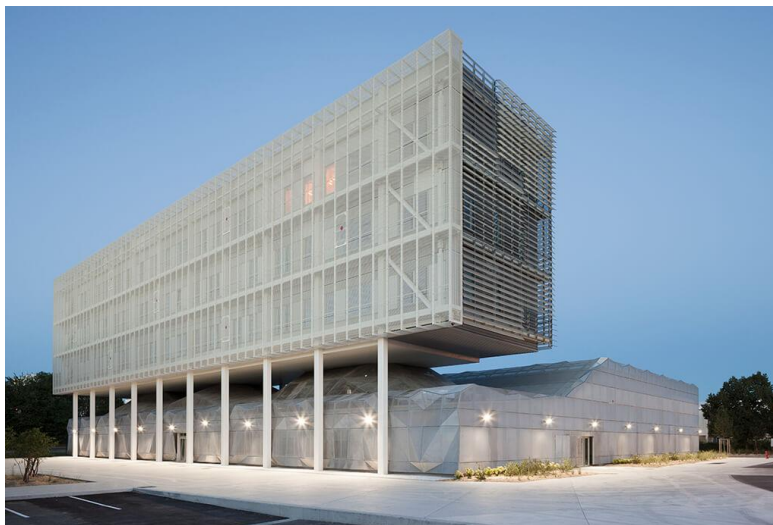


On remarque par exemple que l'architecture Queen Anne est très bien prédite : sur 175 image testées, on en a 163 de bien prédites soit 93% de réussite. Alors que pour l'architecture de l'école de Chicago, on a seulement 66% de réussite.

Afin de tester si l'IA fonctionne, on ajoute sur notre répertoire de travail une image que l'on souhaite prédire, qu'on renomme "imagetest.jpg" puis on exécute les cellules :

```
prediction,pred_idx,outputs = learn.predict(image) #la modèle d'instance predict(image) renvoie  
notre prédiction sous la forme d'un objet de la classe Prediction (bibliothèque fastai)  
prediction.obj #la méthode obj de prédiction renvoie sous forme de string, la prédiction
```

En ajoutant la photo du bâtiment Etoile de Télécom SudParis :



On obtient ceci :

```
img = open_image('etoile.jpg')
```

```
learn = load_learner(path)
```

```
prediction,pred_idx,outputs = learn.predict(img)  
prediction.obj
```

```
'Bauhaus architecture'
```

Ce qui semble être cohérent, ci dessus une image caractéristique de l'architecture Bauhaus :





On utilise ensuite le fichier test.py (code source en annexe), puis on spécifie le chemin vers l'image ainsi que le lien vers le fichier export.pkl comme suit :

### Test de validation :

On utilise le fichier test.py pour notre test de validation partie back (code source en annexe), puis on spécifie le chemin vers l'image ainsi que le lien vers le fichier export.pkl comme suit :

```

1 from fastai.vision import *
2
3
4
5 learner = load_learner("/home/anaana/téléchargements/")
6
7 img = open_image("/home/anaana/Bureau/art-nouveau.jpg")
8
9 prediction_pred_idx_outputs = learner.predict(img)
10 print(prediction_obj)
11
12
13

```

Run: test

/home/anaana/anaconda3/envs/pythonProject/bin/python /home/anaana/PycharmProjects/pythonProject/test.py

Art Nouveau architecture

Process finished with exit code 0

Le code compile bien ("Process finished with exit code 0") et on a bien le bon style architectural en sortie ici : Art-Nouveau Architecture.



#### **4) Manuel utilisateur :**

Production de l'exécutable :

Réalisation des tests :

Pour le test de la partie back-end : le fichier test.py. On télécharge le fichier export.pkl et on importe une image.

Utilisation :

#### **pour faire fonctionner le site**

- s'assurer d'avoir dans le même dossier les fichiers suivants :  
app.py  
basic.css  
projet.css  
export.pkl
- et les dossiers suivants :

static dans lequel se trouve :

app.css  
dropzone.css  
home.css  
client.js  
dropzone.js  
et un dossier image

- template dans lequel se trouve :

app.html  
home.html  
solution1.html

- \_\_pycache\_\_ dans lequel se trouve :  
app.python-38.pyc
- uploads ou les images vont finir

- installer les modules suivants avec pip install :

imgHDR  
os  
Flask

fastai version 1.0.61 avec la ligne de commande pip install --no-deps  
fastai==1.0.61  
json

- pour lancer le code, dans le terminale lancer la commande "flask run"  
puis controle clique sur l'ip fournie ou alors la copier coller dans  
la barre de recherche d'un navigateur
- dans le site web cliquer sur "lancer l'application" ou directement aller dans  
l'onglet app  
puis glisser l'image à prédire (jpg ou png seulement) puis cliquer sur analyser

## **pour utiliser le script de réseau de neurone**

créer un compte kaggle pour obtenir une API

et suivre les instructions données par les commentaires du code

## **5) Conclusion :**

L'application Scan'int est le fruit d'un travail en développement Web associé à l'exploitation d'un réseau convolutif de neurones. En effet, l'objectif de ce projet était avant tout de permettre à tout le monde d'utiliser un système de reconnaissance de bâtiments. Notre idée initiale fut la création d'une application mobile. Pour réaliser le prototype nous avons utilisé kivy un module python permettant de créer simplement une application tout en y associant notre réseau convolutif lui aussi fait en python. Cependant, nous souhaitons pour notre application un web-design plus évolué et nous nous sommes alors penché sur Swift. Cette possibilité de part des contraintes techniques ( outils optimisés pour ios et volumineux ) nous a alors poussé à nous tourner vers une application mobile faites en langage html/ CSS et java. La création du site nécessita donc l'apprentissage de nouveaux langages pour obtenir un design répondant à nos attentes. En parallèle de l'élaboration de la partie front nous avons travaillé sur l'exploitation du réseau de neurones convolutifs. En effet, nous avons eu à cœur de détailler chaque ligne de code nous permettant d'obtenir un fichier de poids correspondant à notre modèle entraîné. Une fois les parties Back et Front réalisées nous avons été confronté à un problème d'intégration de notre code. Effectivement,

nous avons utilisé le module flask de python afin d'héberger localement notre serveur et de lier le back au front. Malheureusement après un long travail d'apprentissage du module nous n'avons pas réussi à lier l'ensemble. Cependant cet échec nous a poussé à développer de nouvelles compétences notamment la compréhension du principe de routes de redirection utilisé dans la création de sites dynamiques. Le livrable de notre projet n'est donc pas intégralement satisfait, cependant ce dernier nous aura permis de toucher du doigt le métier de développeur, à la fois à travers l'aboutissement d'un livrable comme le front mais aussi aux échecs répétés que tout développeur connaît durant son parcours professionnel.

Pour terminer nous aimerions remercier notre encadrant mais aussi les professeurs de l'école et le club Minet pour leurs aides. De plus, nous souhaiterions dans le futur conclure notre projet et le publier en l'hébergeant dans un serveur externe.

## **Bibliographie :**

*Documentation sur Flask :* <https://flask.palletsprojects.com/en/2.1.x/>

Crée des dropzones sur JS : <https://cdnout.com/cdn/dropzone@5.7.1/>

Documentation sur Fastai : <https://docs.fast.ai/>

Classifier des images grâce à GoogleColab et Fastai :

<https://medium.com/unpackai/image-classification-model-using-fastai-and-google-colab-4d77fee8ea9d>

## **Annexe 1 :**

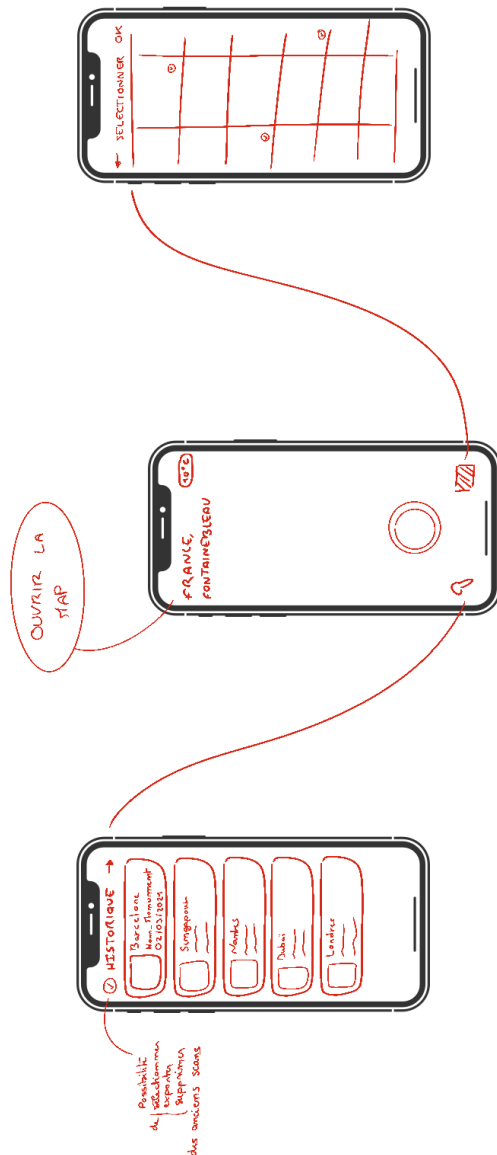


Figure 1 : Maquette graphique de l'application

## Annexe 2:



Gitlab : <https://gitlab.com/naamamohamedcpge/pro3600-22-sell-25>



Planning prévisionnelle (Trello) : <https://trello.com/b/tt4EygPj/pro3600-22-sell-25>

### Annexe 3 :

Code source de EntraînementIABâtiment.ipynb :

```
"""Entraînement_IA_Bâtiment (2).ipynb

## Importation des modules
"""

# On s'assure que toutes les modifications apportées aux bibliothèques sont automatiquement
rechargées ici
# %reload_ext autoreload
# %autoreload 2

# On importe le module matplotlib afin de pouvoir afficher les graphiques pendant nos exécutions
# %matplotlib inline

# On importe le module files et os qui vont nous être utiles pour la suite du programme.
from google.colab import files
import os

# Le module fastai va nous permettre d'avoir des résultats plus rapidement en termes de
DeepLearning
from fastai.vision import *

# Évite les messages d'alertes dus à l'obsolescence du module
import warnings
warnings.filterwarnings("ignore", category=UserWarning, module="torch.nn.functional")

## Télécharger le dataset via Kaggle

uploaded = files.upload() #insérer le fichier kaggle.json

# On crée le répertoire où l'on va mettre notre dataset
! mkdir -p ~/.kaggle/
! mv kaggle.json ~/.kaggle/
```

```

# C'est le chemin, d'accès de notre dataset
chemin = Config.data_path()/'architecture'
chemin.mkdir(parents=True, exist_ok=True)

#On télécharge le dataset
! kaggle datasets download -d wwymak/architecture-dataset

# Une fois bien téléchargé, on le met dans le répertoire, en dezipant
! unzip -q -n architecture-dataset.zip -d {chemin}

# Le chemin de notre data set est celui ci-dessous :
chemin = '/root/.fastai/data/architecture/architectural-styles-dataset'

# On va transformer nos images afin d'augmenter notre dataset
tfms = get_transforms(do_flip=True, flip_vert=False, max_rotate=10, max_zoom=1.1,
                      max_lighting=0.2, max_warp=0.2, p_affine=0.75, p_lighting=0.75)

#Dans cet ensemble de données particulier, les étiquettes sont stockées dans le nom du dossier qui
contient les images de chaque classe. Nous devons les extraire pour pouvoir classer les images
dans les bonnes catégories. La bibliothèque fastai a une fonction faite pour cela,
`ImageDataBunch.from_folder`.#

data = ImageDataBunch.from_folder(chemin, train=".", valid_pct=0.2,
                                  ds_tfms=tfms, size=224, num_workers=4, padding_mode='reflection',
                                  bs=64).normalize(imagenet_stats)

#On affiche 9 images de notre set aléatoirement
data.show_batch(rows=3, figsize=(9, 9))

#On regroupe les différentes catégories ensemble.
data.classes, data.c, len(data.train_ds), len(data.valid_ds)

#Retournement (juste horizontal), zoom, lumière, rotation, etc...

def _plot(i,j,ax):
    x,y = data.train_ds[3]
    x.show(ax, y=y)

plot_multi(_plot, 3, 3, figsize=(8,8))

### Apprentissage

```



```
learn = cnn_learner(data, models.resnet34, metrics=[accuracy]) # on va utiliser un réseau de neurones largement utilisé et déjà pré-entraînés sur notre modèle : resnet34. Ce réseau de neurones convolutifs ayant 34 couches de neurones, par la fonction cnn_learner de fastai
```

```
#Time = temps pour chaque epoch, donc 4 ici  
learn.fit_one_cycle(4)
```

```
learn.save('stage-1-resnet34') #Par la fonction .save() de f permet d'enregistrer le réseaux "learn"  
dans le répertoire courant
```

```
interpretation = ClassificationInterpretation.from_learner(learn) #La bibliothèque fastai fournit la classe  
ClassificationInterpretation, on crée notre objet  
#interpretation  
interpretation.plot_confusion_matrix(figsize=(12,12), title = "Matrice de confusion", cmap = "YlOrRd",  
dpi=70) #Méthode d'instance permettant d'afficher  
# la matrice de confusion, le coefficient (i,j) de cette matrice affiche combien de fois le j-ème élément  
a été prédit alors que la véritable valeur était le i-ème élément
```

```
learn.export() #Maintenant on exporte notre modèle dans le dossier root, il est entraîné et doit être  
utilisé sur des images tests
```

```
!cp /root/.fastai/data/architecture/export.pkl . #On le copie dans le répertoire courant
```

```
image = open_image(imagetest.jpg) #On ajoute manuellement, dans le répertoire courant, l'image  
que l'on veut prédire (à renommer imagetest impérativement)
```

```
learn = load_learner(chemin) #La méthode load_learner de fastai prend en paramètre un chemin et  
renvoie le modèle dans une variable
```

```
prediction, pred_idx, outputs = learn.predict(image) #la méthode d'instance predict(image) renvoie notre  
prédiction sous la forme d'un objet  
#de la classe Prediction (bibliothèque fastai)  
prediction.obj #la méthode obj de prediction renvoie sous forme de string, la prédiction
```

## Annexe 4 : Code source de app.py

```
import imghdr  
from fastai.vision import *  
import os  
from flask import Flask, render_template, request, redirect, url_for, abort, send_from_directory  
from werkzeug.utils import secure_filename  
  
app = Flask(__name__)  
app.config['MAX_CONTENT_LENGTH'] = 2 * 1024 * 1024  
app.config['UPLOAD_EXTENSIONS'] = ['.jpg', '.png']
```

```

app.config['UPLOAD_PATH'] = 'uploads'

chemin="/" #Il faut mettre le export.pkl dans le repertoire courant
model = load_learner(chemin)

#=====validation
image=====

def validate_image(stream):
    header = stream.read(512)
    stream.seek(0)
    format = imghdr.what(None, header)
    if not format:
        return None
    return '.' + (format if format != 'jpeg' else 'jpg')

@app.errorhandler(413)
def too_large(e):
    return "File is too large", 413

#=====mise de l'image dans le dossier=====
@app.route('/')
def home():
    return render_template('/home.html')

@app.route('/app.html')
def index():
    files = os.listdir(app.config['UPLOAD_PATH'])
    return render_template('app.html', files=files)

@app.route('/app.html', methods=['POST'])
def upload_files():
    uploaded_file = request.files['file']
    filename = secure_filename(uploaded_file.filename)
    if filename != "":
        file_ext = os.path.splitext(filename)[1]
        if file_ext not in app.config['UPLOAD_EXTENSIONS'] or \
            file_ext != validate_image(uploaded_file.stream):
            return "Invalid image", 400
        uploaded_file.save(os.path.join(app.config['UPLOAD_PATH'], filename))
    return "", 204

@app.route('/upload', methods=["POST"])
def analyze():

    file = request.files['user-img'].read()
    img = open_image(io.BytesIO(file))
    prediction = model.predict(img)[0]

    response = {"result": str(prediction)}

```

```
return jsonify(response)
```