UQAC

Université du Québec
à Chicoutimi

Projet : 8INF804 : Traitement numérique des images - Hiver 2022

Réalisé par :

Nihal OUHAOUADA | *OUHN11629909*

Pierre LACLAVERIE | *LACP03119904*

Thibaud SIMON | *SIMT15039901*

Yann REYNAUD | *REYY07110005*

Sommaire

Introduction

L’objectif de ce travail est de présenter la recherche et l'implémentation de l’utilisation de réseaux antagonistes (GAN) pour la génération d’images. En premier lieu, une analyse du dataset sera faite, ensuite nous présenterons la méthode utilisée pour générer nos images, suivie d’une explication de la démarche et l’analyse du code produit. Enfin les résultats seront analysés et une mise en perspective du résultat.

Quand la quantité de données que l’on récupère n’est pas suffisante et qu’il n’est pas possible d’en avoir de nouvelles dans les mêmes conditions, il est possible d’en générer… à partir des mêmes données qui ont été collectées.

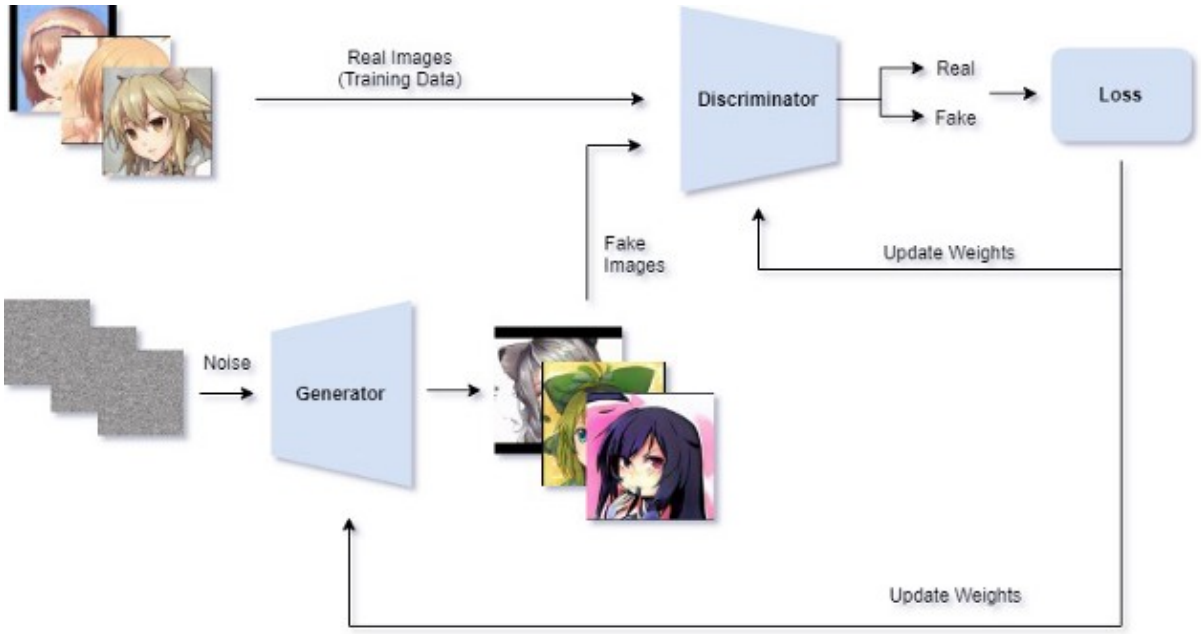
Intention et motivation

La fabrication d’une série animée nécessite un travail d’écriture et de dessin. L’idée de notre application vient dans ce même contexte, et sert à créer de nouveaux personnages animés qui feront l’objet d’animation par la suite. Notre application va également offrir la possibilité de créer de nouveaux avatars pour, par exemple, personnaliser l’identité sur un forum, un site web, … Nous allons pour cela exploiter la technologie récente, plus particulièrement les techniques d’apprentissage automatique, pour produire un maximum d’images possibles.

Au cours de ce travail pratique, nous nous sommes concentrés sur des images de face d’animés. Nous avons pour objectif de générer de nouvelles images de faces d’animés à l’aide d’un GAN en ayant au départ un dataset de faces de personnages d’animés et cela en utilisant l’ensemble de données “animes faces” et l’outil open source Tensorflow.

Avant de passer à la mise en œuvre de notre architecture , voyons un peu ce que c’est un GAN : il est constitué de 2 réseaux de neurones : le générateur et le discriminateur qui fonctionnent de manière antagoniste. Le générateur a pour but de créer de fausses images, le discriminateur quant à lui va devoir distinguer les vraies images de fausses. Le processus devrait s'arrêter lorsque le discriminateur devient incapable de discriminer (comme son nom l’indique) les fausses images des vraies. Et c’est à partir de ce moment que nous allons pouvoir utiliser notre générateur pour remplir sa fonction.

L’architecture générale de notre GAN ressemblera à ce qui est présenté dans l’image suivante :



Architecture GAN (source)

Analyse du Dataset

Nous avons choisi de prendre le dataset anime faces pour notre projet. Celui-ci comporte plus de 21 000 images tirées du site getchu.com. L’algorithme utilisé est tiré de ce github, il permet de localiser toutes les faces (têtes) de personnages tirés d’animé.
(auteur: Soumik Rakshit, publication kaggle il y a 3 ans)

Pourquoi avons-nous choisi ce dataset?

- Les animés sont connus du grand public donc beaucoup de ressources disponibles
- La quantité d’images est importante : apprentissage avec des données variées
- La taille du dataset est petite : environ 500 Mo, taille raisonnable
- La résolution des images est petite (64px) donc le traitement est plus rapide.

Post-traitement: Une inspection des images à l'œil a montré que quelques (250/21000) images n’étaient pas pertinentes, celles-ci ont été supprimées.

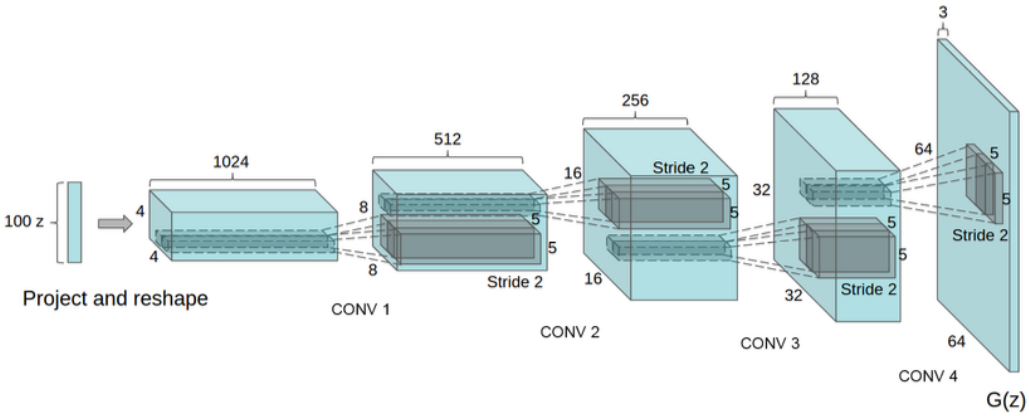
Architecture de réseaux de neurones

- Création des modèles :

  Générateur :

  Le générateur prend un vecteur de bruit en entrée, d’une dimension définie (**latent_dim**), et génère une image à partir de celui-ci. L’image est générée aléatoirement en suivant la loi ‘standard_normal’ de

numpy. Plus d'informations à ce sujet ici.
Le générateur subira (détails dans l'implémentation) plusieurs opérations de reshaping pour obtenir, à partir du vecteur de bruit en 3 dimensions, notre sortie à taille et canaux souhaités (64x64, rgb)



Représentation visuelle du générateur et son reshaping, issue de la publication sur le DCGAN. Source web: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Discriminateur :

Concernant le discriminateur, il s'agit d'un classificateur binaire régulier. Nous utiliserons la fonction d'activation **LeakyReLU** (variante de ReLU) ainsi que l'optimiseur **Adam** pour son efficacité. Pour les pertes, on utilise la cross-entropy binaire (**binary_crossentropy**).

- Fonction de perte et optimiseur

GAN :

Nous avons utilisé l'optimiseur Adam pour son efficacité. Celui-ci est une méthode de descente de gradient stochastique permettant une estimation grâce aux moments de 1er et 2nd ordre. Cet optimiseur a aussi pour avantages de ne pas demander trop de mémoire, et n'a pas besoin de beaucoup de réglages sur les hyperparamètres pour un bon fonctionnement. C'est aussi tout simplement le plus utilisé sur les exemples de DCGAN.

Pour les pertes, nous avons utilisé la cross entropie binaire (**binary_crossentropy**). En effet, l'entraînement du GAN repose sur le fait d'entraîner en rivalité deux réseaux de neurones. D'une part un générateur qui va produire des images et d'autre part un discriminateur qui va chercher à vérifier si les images créées par le générateur sont de fausses images ou non. L'objectif du générateur est de tromper le discriminateur. Inversement, l'objectif du discriminateur est d'être assez robuste pour percer à jour le générateur. Dans le meilleur des cas, le discriminateur doit se tromper en moyenne la moitié du temps. La réponse du discriminateur sera Vrai ou Faux pour qualifier les images produites par le générateur. Deux valeurs d'où le

binary. La cross-entropy, ou entropie croisée, augmente si la valeur prédite par l'algorithme a tendance à ne pas être le bon label, la "vérité terrain". Plus d'informations sur la documentation ou ici.

Discriminateur :

Nous avons également utilisé Adam comme optimiseur au vu de son efficacité générale. Comme énoncé précédemment, Adam est plutôt simple d'utilisation au niveau des hyperparamètres. Tout comme pour le générateur, on a opté pour des pertes utilisant la méthode **binary_crossentropy**.

- Exécution de l'entraînement

Dans la fonction **train_GAN** nous réalisons l'entraînement de nos modèles. Nous commençons par diviser le nombre d'images du dataset tronqué par le nombre de batchs afin d'obtenir le nombre de batchs par époques.

Discriminateur : On introduit la notion de **half-batch** pour travailler sur autant d'échantillons réels que de faux. On utilise ainsi 50 % de vraies images (**get_real_images)** et on génère 50% de fausses images (**generate_fake_images)** qu'on concatène dans un seul ensemble.

GAN : après avoir entraîné le discriminateur nous entraînons alors le GAN (fonction **train_on_batch**), récupérant ainsi les pertes. A noter que le GAN s'entraîne via du bruit (**generate_latent_points**) qui est labellisé par défaut à 1.

Méthodologie

Nous commençons par réaliser les imports nécessaires, puis connecter notre drive au google colab, et décompresser le fichier ZIP du dataset, situé sur le drive, dans l'environnement de travail local.

Nous pouvons maintenant charger les images décompressées dans notre programme, en utilisant la fonction intégrée **flow_from_directory**:

```
def load_real_images():
 datagen = ImageDataGenerator(rescale=1./255)
 X = datagen.flow_from_directory('/content/faces/data',
                                 target_size= (64,64),
                                 batch_size=8192,
                                 class_mode='binary')
 #Charger dans une liste et renvoyer comme array numpy
 ...
```

Le générateur prend en entrée un vecteur de bruit. Nous avons donc créé une fonction pour le générer, en utilisant une distribution normale:

```
def generate_latent_points(latent_dim, nb_of_samples):
 #Generate points in the latent space
 X = np.random.randn(latent_dim * nb_of_samples)
 #Reshape into a batch of inputs for the network
 X = X.reshape(nb_of_samples, latent_dim)
 return X
```

Voici les définitions de discriminateurs et générateurs, correspondants à l'architecture présentée par DCGAN et expliquée au-dessus.

```
def define_Discriminator(in_shape=(64,64,3)):
 model = Sequential()
 model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
 model.add(LeakyReLU(alpha=0.2))
 model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
 model.add(LeakyReLU(alpha=0.2))
 model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
 model.add(LeakyReLU(alpha=0.2))
 model.add(Flatten())
 model.add(Dropout(0.4))
 model.add(Dense(1, activation='sigmoid'))
 optimizer = Adam(lr=OPT_LR, beta_1=0.5)
 model.compile(loss='binary_crossentropy', optimizer=optimizer,
metrics=['accuracy'])
 return model
```

De même, nous créons notre modèle de générateur. Celui-ci requiert de connaître la dimension du vecteur de points latents d'entrée, afin de définir les dimensions de sa couche d'entrée; nous l'avons donc placé en argument de la fonction.

```
def define_Generator(latent_dim):
 model_G = Sequential()
 #16x16 image
 nb_nodes =  16 * 16 * 256
 model_G.add(Dense(nb_nodes, input_dim=latent_dim))
 model_G.add(LeakyReLU(alpha=0.2))
 model_G.add(Reshape((16, 16, 256)))
 #Upsample to 32x32
 model_G.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
 model_G.add(LeakyReLU(alpha=0.2))
 #Upsample to 64x64
 model_G.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
 model_G.add(LeakyReLU(alpha=0.2))
```

```
 model_G.add(Conv2D(3, (7,7), activation='tanh', padding='same'))
 return model_G
```

La première phase correspondant à l'entraînement du discriminateur, nous avons besoin d'extraire des images de la dataset de départ et de générer des fausses images pour que le discriminateur puisse s'entraîner sur une dataset contenant des fausses et des vraies images labellisées.
Ainsi nous avons 2 fonctions.
La première permet d'extraire des vraies images du dataset et de les labelliser "1" car ce sont des vraies images.

```
def get_real_images(dataset, nb_of_samples):
 i = np.random.randint(0, dataset.shape[0], nb_of_samples)
 X = dataset[i]
 y = np.ones((nb_of_samples, 1))
 return X, y
```

La deuxième génère des fausses images grâce au modèle du générateur à partir du bruit d'entrée, puis les leur donne le label "0" (fausses images).

Enfin, nous pouvons créer notre modèle du GAN en réunissant ces 2 modèles en un. Celui-ci nous servira pour l'entraînement du générateur à chaque époque, c'est pourquoi on désactive l'entraînement des poids du discriminateur, ainsi nos résultats ne sont pas détériorés, ils s'améliorent tour-à-tour et non pas en même temps.

```
def define_GAN(model_G, model_D):
 #Discriminator's weights not trainable
 model_D.trainable = False
 model = Sequential()
#Adding both models in an only one
 model.add(model_G)
 model.add(model_D)
 optimizer = Adam(lr=OPT_LR, beta_1=0.5)
 model.compile(loss='binary_crossentropy', optimizer=optimizer)
 return model
```

Une fois le modèle du GAN définis, nous pouvons implémenter une fonction d'entraînement du gan (deuxième phase de l'entraînement).

Pour ce faire nous créons une boucle itérative pour chaque époque à la main.

```
def train_GAN(model_G, model_D, model_GAN, dataset, latent_dim, nb_epochs =
50, nb_batch = 128):
 batch_per_epoch = int(dataset.shape[0] / nb_batch)
 half_batch = int(nb_batch / 2)
 #Range for epochs enumeration
```

```
 for i in range(nb_epochs):
 #Enumerate batches over the training set
   for j in range(batch_per_epoch):
     X_real, y_real = get_real_images(dataset, half_batch)
     X_fake, y_fake = generate_fake_images(model_G, latent_dim, half_batch)
     #Concatenate fake and real images / labels after generating them
     X, y = np.vstack((X_real, X_fake)), np.vstack((y_real, y_fake))
     #Then training discriminator and GAN
     d_loss, _ = model_D.train_on_batch(X, y)
     X_gan = generate_latent_points(latent_dim, nb_batch)
     y_gan = np.ones((nb_batch, 1))
     g_loss = model_GAN.train_on_batch(X_gan, y_gan)
     print('%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, batch_per_epoch, d_loss,
g_loss))
   summarize_performance(i, model_G, model_D, dataset, latent_dim)
```

Pour chaque époque nous voulons sauvegarder le modèle discriminateur et afficher l'accuracy du discriminateur sur les vraies et fausses images, c'est pourquoi nous utilisons la fonction **summarize_performance** implémentée ci-dessous :

```
def summarize_performance(epoch, model_G, model_D, dataset, latent_dim,
nb_of_samples=100):
 #Saving the generator's model for a given epoch
 model_G.save('/content/drive/My Drive/Colab Notebooks/model_anime_'
+str(epoch)+ '.h5')
 #Evaluating Discriminator's model on real and fake images
 X_real, y_real = get_real_images(dataset, nb_of_samples)
 _, acc_real = model_D.evaluate(X_real, y_real, verbose=0)
 x_fake, y_fake = generate_fake_images(model_G, latent_dim, nb_of_samples)
 _, acc_fake = model_D.evaluate(x_fake, y_fake, verbose=0)
 print('Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
```

Une fois l'entraînement terminé et les modèles du discriminateur pour différentes époques enregistrés, nous pouvons utiliser ces modèles pour générer des "fausses images", afin de comparer concrètement les résultats pour diverses époques nous avons implémenté la fonction suivante :

```
def plot_images(model, nb_imgs=6):
   X = generate_fake_images(model, LATENT_DIM, nb_imgs)
   # plot the result
   plt.figure(figsize=(19, 19))
   i=1
   plt.axis('off')
   for img in X[0]:
     plt.subplot((nb_imgs)+1, nb_imgs, i)
     plt.imshow(np.clip((img*255), 0, 255).astype('uint8'))
```

```
      i=i+1
      plt.axis('off')
    plt.show()
      plt.title(f"époque {i+1}")
```

Il nous suffit alors de l'utiliser avec les modèles de différentes époques pour comparer les résultats.

Définissons quelques constantes :

```
NUM_EPOCHS = 100
BATCH_SIZE = 64
OPT_LR = 2e-4
LATENT_DIM = 100
```

Maintenant que nous avons toutes les fonctions nécessaires au bon fonctionnement de notre GAN, nous pouvons appeler ces fonctions dans notre **programme principal** :

```
# Discriminateur
discriminateur = define_Discriminator()

# Générateur
generateur = define_Generator(LATENT_DIM)

# GAN
model_GAN = define_GAN(generateur, discriminateur)

# Chargeons nos images
dataset = load_real_images()

#entraînons notre modèle
train_GAN(generateur, discriminateur, model_GAN, dataset[0], LATENT_DIM,
nb_epochs = NUM_EPOCHS, nb_batch = BATCH_SIZE)
```

Ici, nous définissons les dimensions de notre vecteur de points latents, puis nous définissons nos 2 modèles (Générateur et discriminateur) pour ensuite faire le modèle du GAN. On définit alors notre dataset en appelant la fonction **load_real_images**, puis nous entraînons le modèle du GAN et affichons les images résultant de la prédiction du modèle du générateur pour diverses époques.

Nous pouvons dorénavant passer à la partie test du modèle (À exécuter ici) :

Commençons par importer le modèle sauvegardé sur notre drive :

```
generateur = load_model('/content/model_anime_150.h5')
```

(ici il s'agit du modèle de l'époque 150)

Puis nous appliquons notre fonction **plot_images** via ce modèle

```
plot_images(generateur)
```

Choix d'implémentation

Dans la partie précédente, nous avons donné les grandes lignes de notre méthodologie. Maintenant nous allons détailler chacun des choix faits précédemment.

- Chargement des données :

Au cours de cette partie, nous allons expliquer le choix de :

- Datagenerator
- Paramètres de flow_from_directory

Nous avons utilisé Datagenerator afin de pouvoir charger efficacement le dataset. En effet, quand ce dernier commence à être important, cette méthode permet de ne pas surcharger la mémoire vive de l'appareil utilisé.

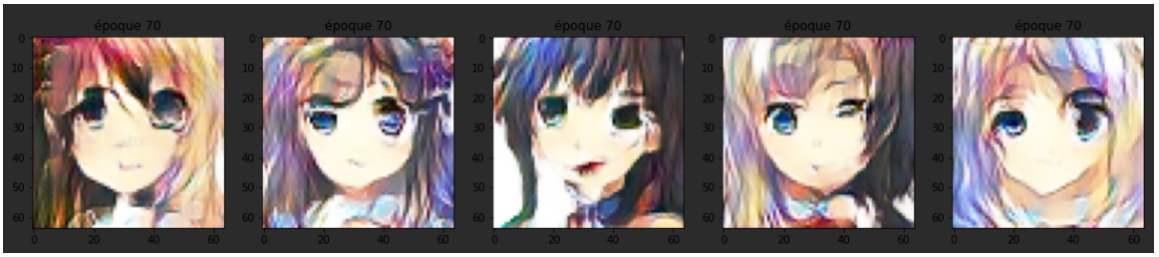
Pour flow_from_directory:

Le choix d'un class_mode à binary est nécessaire car nous avons comme sortie uniquement la réponse vraie ou fausse.

Analyse des résultats

On peut voir qu'au bout de 150 epochs, le résultat n'est pas parfait mais il reste encore très satisfaisant. En effet, les visages présentent des défauts comme par exemple une malformation des yeux, des couleurs qui ne sont pas réalistes avec les styles connus ou encore pour certain le visage possède des irrégularités. Cependant, il est à noter que les images ainsi créées représentent bien "à l'œil nu" des personnages d'animés. Les principales caractéristiques du visage sont présentes : du front au menton en passant par les joues, la bouche, les yeux et le nez. Tous ces éléments sont présents dans chacune des images ainsi créées.

Il est important de constater que nous avons entraîné le modèle sur 150 époques afin d'être sûr de ne pas le sous-entraîner, mais notre système a atteint son équilibre (équilibre de Nash) bien avant celle-ci.



En effet nous remarquons qu'à partir de l'époque 70 nous avons des résultats satisfaisants et similaires à ceux de l'époque 150.

Perspectives de commercialisation

La solution proposée est fonctionnelle et propose des résultats satisfaisants. L'objectif ici est de créer des faces nouvelles à partir d'autres faces. Un acheteur potentiel serait une société spécialisée dans les manga qui souhaiterait par exemple créer de nouveaux personnages, de pouvoir générer un unique avatar pour un joueur ou un utilisateur par exemple.

Nos 2 concurrents se basent sur le modèle StyleGAN et BigGAN ce qui leur a permis d'avoir des performances supérieures aux nôtres. StyleGAN est introduit par des chercheurs de Nvidia, le point fort de cette architecture réside dans le fait qu'elle contrôle les détails dans leurs granularités tout en s'inspirant des GAN progressifs qui débutent leurs entraînement par une faible résolution et s'améliorent au fur-et-à-mesure que des couches sont ajoutées au réseau, ce qui diminue considérablement le temps d'entraînement. Des améliorations ont aussi été apportées à l'architecture progressive initiale comme par exemple le Mapping Network ou la troncature. Concernant l'élaboration d'un modèle BigGAN, de nombreuses méthodes ont été utilisées dans le but d'éviter le surajustement dans les données d'entraînement du discriminateur qui a pour effet de ne pas fournir des signaux significatifs au générateur. Parmis ces méthodes, nous citons : la troncature, le mécanisme d'auto-attention…

- This Waifu Does Not Exist.: Avec un simple clique sur le bouton "refresh", une image d'anime générée aléatoirement s'affiche sur l'écran et offre la possibilité d'écrire une histoire sur la même page mais en même temps l'application donne une mauvaise expérience utilisateur. Cette application se base sur StyleGAN,
- Artbreeder. anciennement appelé Gan Breeder, est un site web basé sur des techniques de machine learning, en particulier le StyleGAN ainsi que le BigGAN. Elle permet de générer de nouveaux personnages animés mais aussi les modifier.

Améliorations possibles

Pour ce qui est de l'équilibre de Nash, nous n'avons pas implémenté de solution permettant de le détecter. Nous avons dû le faire manuellement.

Concernant l'efficacité de notre solution, nous aurions pu avoir de bien meilleurs résultats en utilisant le StyleGAN (énoncé précédemment).

Afin d'avoir un dataset de départ plus homogène, nous aurions également pu pré-traiter les images grâce à des techniques d'augmentation.

On aurait pu ajouter une interface graphique pour notre application permettant de choisir le modèle de quelle époque on veut récupérer, puis l'utiliser afin de générer un nombre désiré (à entrer sur l'application) d'image d'animés.

Nous aurions également pu utiliser d'autres fonctions de perte que la binary crossentropy telles que Wasserstein.

De la même manière nous nous sommes arrêtés à l'optimiseur Adam, mais il aurait été intéressant de vérifier si d'autres modèles, moins performants de manière générale, seraient plus efficaces dans notre cas précis.