

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique
Université des Sciences et Technologies
Houari Boumediene



M2 : HPC

Rapport DL :
Font Style Transfer
Génération de mélodies

KHENE Soraya
MEKHAZNI Ryham
KEFSI Nourhane

25 janvier 2025

Table des matières

1	Création de Dataset	1
1.1	Bibliothèques Utilisées	2
1.2	Création du Dataset de Textes	2
1.2.1	Génération de Textes	2
1.2.2	Génération des Images de Texte	2
1.3	Création du Dataset de Caractères	3
1.3.1	Génération des Images de Caractères	3
1.3.2	Rotation des Images	4
1.3.3	Enregistrement des Images	4
2	Modèles de transfer font style	5
2.1	Modèle CNN	5
2.1.1	Architecture du Modèle	5
2.1.2	Prétraitement des Données	6
2.1.3	Entraînement du Modèle	6
2.1.4	Fonctionnement du Modèle	6
2.1.5	Résultats	7
2.2	Modèle duo cGAN classificateur	9
2.2.1	Étapes d'Entraînement de cGAN	10
2.2.2	Pipeline de Génération	12
2.2.3	Résultats	12
2.2.4	Problèmes	13
2.3	Modèle Pix2Pix	14
2.3.1	Préparation des données	14
2.3.2	Augmentation des données	14
2.3.3	Architecture du modèle	14
2.3.4	Entraînement	17
2.3.5	Résultats	17
2.3.6	Exemple de génération	18
2.4	Modèle cycleGAN	19
2.4.1	Chargement et Prétraitement des Données	19
2.4.2	Architecture des Modèles	20
2.4.3	Fonctions de Perte	21
2.4.4	Optimisation	21
2.4.5	Étapes d'Entraînement	21
2.4.6	Résultats	22
3	L'interface utilisateur pour le text font transfer :	23
3.1	Simulation de transfert de style (Mock Test) :	23

4	Modèle de génération de mélodies	26
4.1	Bibliothèques utilisées	26
4.2	Les étapes principales du model	26
5	Difficultés rencontrées	31

Table des figures

1	Text Arial -> Calibri	3
2	Le caractère "f" en Arial transformé en Times	4
3	cnn model.	5
4	Prétraitement des Donnée	6
5	Epoch 1	8
6	Epoch 2	8
7	epoch 6	8
8	Epoch 8	8
9	Epoch 26	8
10	Epoch 27	8
11	Conditional GAN architecture	9
12	Classifier architecture	10
13	Architecture cGAN	10
14	Generateur (model_7)	11
15	sequential_7	11
16	Discriminateur (model_8)	11
17	sequential_8	11
18	Résultat cGAN Algerian font	13
19	l'architecture Pix2Pix	15
20	Un exemple de generation	18
21	Cycle GAN	19
22	Architecture du model generateur	20
23	Architecture du model discriminateur	21
24	Résultats après 25 epochs	22
25	Résultats apres 300 epochs	22
26	simulation	23
27	interaction avec le model	24
28	Interface	24
29	Chargement d'une image	25
30	interaction avec le model	25
31	Architecture LSTM	28
32	Model Training	29
33	Melodie generation	30
34	Problèmes avec Kaggle	31

Liste des tableaux

1	Partie descendante de l'architecture du générateur (U-Net) pour des images en niveaux de gris.	15
2	Partie montante de l'architecture du générateur (U-Net) pour des images en niveaux de gris.	15
3	Architecture du Discriminateur (PatchGAN) pour des images en niveaux de gris.	17

Introduction

Dans le contexte de ce projet, nous avons étudié deux champs captivants de l'apprentissage profond : le transfert de style typographique (font transfer) et la création musicale (music generation). Ces deux thèmes s'intègrent dans le cadre plus vaste de la créativité produite par des modèles d'apprentissage profond, visant à reproduire ou à concevoir de nouvelles œuvres artistiques à partir d'exemples préexistants.

Le transfert de style typographique est basé sur le concept de changer l'apparence du texte tout en conservant son sens. Ce projet a nécessité l'emploi d'un modèle génératif (telle qu'une GAN ou un réseau neuronal) pour conférer un style particulier à un texte déterminé, en s'appuyant sur des modèles de polices existants. Par ailleurs, la génération musicale implique l'entraînement d'un modèle qui peut produire de nouvelles séquences musicales à partir d'une collection d'exemples musicaux. Ce travail a fait appel à des réseaux de neurones récurrents, plus précisément des LSTM, afin d'apprendre à créer de nouvelles mélodies sur la base de motifs assimilés à partir de compositions musicales de genres particuliers.

Ce rapport vise à exposer les phases de conception, d'entraînement et de validation de ces deux modèles, en soulignant les méthodes d'apprentissage employées, les obstacles surmontés et les performances réalisées. Nous examinerons les facettes théoriques et concrètes de ces deux initiatives dans le but de démontrer l'application potentielle des technologies d'intelligence artificielle dans des activités créatives.

1 Création de Dataset

Nous avons créé deux types de datasets pour l'entraînement et le test de nos modèles utilisés dans le transfert de style de police.

- Le premier dataset contient des paires d'images (**input**, **target**) où l'image "input" contient un texte en anglais dans le style *Arial*, et l'image "target" contient le même texte dans le style *Calibri*.
- Le deuxième dataset contient des paires d'images représentant tous les caractères existants dans la langue anglaise. Chaque image "input" contient un caractère dans un style de police, tandis que l'image "target" contient le même caractère dans un autre style de police.

Dans cette section, nous présentons les bibliothèques utilisées et les étapes détaillées pour la création de chaque dataset.

1.1 Bibliothèques Utilisées

Nous avons utilisé plusieurs bibliothèques pour générer nos datasets :

- **nltk** : La bibliothèque Natural Language Toolkit (NLTK) est utilisée pour générer des textes aléatoires à partir du corpus "brown" intégré, permettant de créer des échantillons de texte réalistes pour le dataset.
- **PIL (Pillow)** : Utilisée pour la manipulation des images. Elle permet de créer des images avec du texte rendu dans des polices spécifiques et d'ajuster la taille du texte pour qu'il tienne dans l'image.
- **random** : Utilisée pour la génération aléatoire de caractères et d'autres variations dans les données.

1.2 Création du Dataset de Textes

Pour créer ce dataset, nous avons suivi les étapes suivantes :

1.2.1 Génération de Textes

Nous avons utilisé le corpus "brown" de NLTK pour générer des textes aléatoires. Ces textes sont constitués de mots choisis au hasard, avec des longueurs variables allant de 5 à 25 mots. Cette diversité garantit une couverture des différents types de mots et de structures de phrases, essentielle pour la robustesse du modèle.

1.2.2 Génération des Images de Texte

Pour chaque texte généré, une image est créée en utilisant la police **Arial**. Voici les étapes principales :

- Le texte est dessiné dans l'image à l'aide de la bibliothèque PIL.
- La taille de la police est ajustée pour s'assurer que le texte tienne dans l'image sans être coupé.
- Les paires d'images (input, target) sont générées, où l'image "input" contient le texte en **Arial** et l'image "target" contient le texte en **Calibri**.

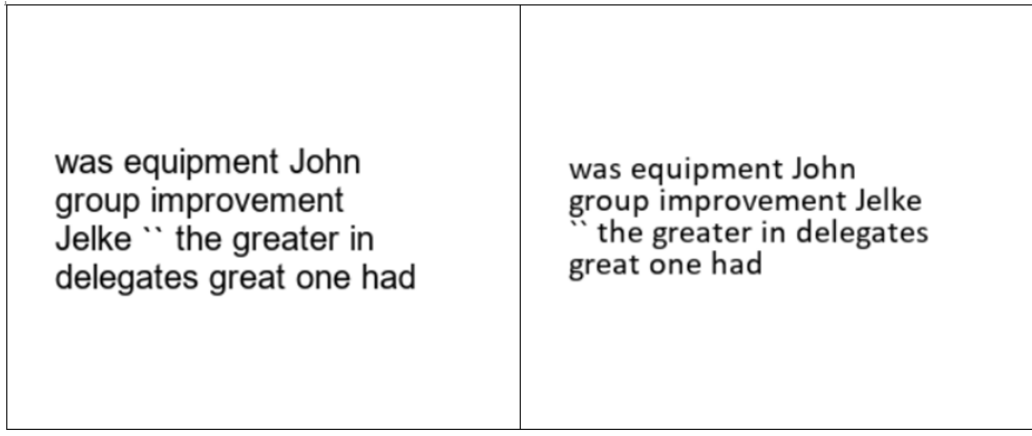


FIGURE 1 – Text Arial -> Calibri

Nous avons choisi uniquement deux styles de polices pour générer un grand nombre d’images et capturer efficacement les caractéristiques spécifiques des styles (en raison des limitations des ressources).

1.3 Création du Dataset de Caractères

Le deuxième dataset est basé sur des caractères individuels plutôt que sur des textes. Contrairement au premier dataset, celui-ci inclut plusieurs styles de polices et sert à entraîner des modèles capables de transférer les styles d’un caractère à un autre.

1.3.1 Génération des Images de Caractères

Pour chaque combinaison de styles de police, plusieurs paires d’images (input, target) sont générées :



FIGURE 2 – Le caractère "f" en Arial transformé en Times

- L'image "input" contient un caractère dans un style de police donné.
- L'image "target" contient le même caractère dans un autre style de police.
- La bibliothèque PIL est utilisée pour créer les images.

1.3.2 Rotation des Images

Chaque image est ensuite tournée dans trois orientations possibles (0° , 180° , 270°) pour augmenter la diversité du dataset. Après chaque rotation, l'image est redimensionnée pour s'assurer qu'elle conserve une taille constante de 128x128 pixels.

1.3.3 Enregistrement des Images

Les images générées sont enregistrées dans un répertoire spécifique à chaque police, avec un nom de fichier qui reflète :

- Le caractère représenté.
- La couleur du texte.
- La couleur de fond.
- L'angle de rotation (par exemple, `A_black_white_0.png` pour un caractère "A" noir sur fond blanc, avec une rotation de 0°).

2 Modèles de transfer font style

2.1 Modèle CNN

Dans cette section, nous présentons le modèle de réseau de neurones convolutionnel (**CNN**) utilisé pour le transfert de style de police. Ce modèle repose sur une architecture encodeur-décodeur qui transforme les images d'une police donnée vers un autre style de police.

2.1.1 Architecture du Modèle

L'architecture du modèle suit un schéma encodeur-décodeur classique, où l'encodeur extrait les caractéristiques pertinentes de l'image d'entrée, tandis que le décodeur génère l'image de sortie en rétablissant les dimensions spatiales originales. Voici les détails de l'architecture :

- **Entrée** : Le modèle prend en entrée des images en niveaux de gris de taille 256x256 pixels.
- **Encodeur** : L'encodeur comprend trois couches convolutionnelles qui réduisent progressivement la taille de l'image tout en extrayant des caractéristiques à différents niveaux. Les filtres utilisés sont respectivement de tailles 64, 128 et **256**.

```
# Build the CNN model (Encoder-Decoder)
def build_cnn_model():
    inputs = layers.Input(shape=(256, 256, 1))
    x1 = layers.Conv2D(64, (3, 3), padding="same", activation="relu")(inputs)
    x2 = layers.Conv2D(128, (3, 3), strides=2, padding="same", activation="relu")(x1)
    x3 = layers.Conv2D(256, (3, 3), strides=2, padding="same", activation="relu")(x2)

    # Upsample and concatenate with skip connections
    x4 = layers.Conv2DTranspose(128, (3, 3), strides=2, padding="same", activation="relu")(x3)
    x4 = layers.Concatenate()([x4, x2]) # Skip connection
    x5 = layers.Conv2DTranspose(64, (3, 3), strides=2, padding="same", activation="relu")(x4)
    x5 = layers.Concatenate()([x5, x1]) # Skip connection

    # Output layer
    outputs = layers.Conv2D(1, (3, 3), padding="same", activation="tanh")(x5) # Tanh for output
    return tf.keras.Model(inputs, outputs)
```

FIGURE 3 – cnn model.

- **Décodeur** : Le décodeur utilise des couches convolutionnelles transposées (ou de sur-échantillonnage) pour augmenter les dimensions spatiales des caractéristiques extraites. Des **connexions de saut (skip connections)** sont utilisées pour permettre au modèle de récupérer des informations importantes de l'encodeur.

```

def load_image(image_path):
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=1) # Grayscale
    image = tf.image.resize(image, [IMG_HEIGHT, IMG_WIDTH])
    image = image / 255.0 # Normalize to [0, 1]
    return image

def load_data(image_dir):
    input_paths = sorted([os.path.join(image_dir, f) for f in os.listdir(image_dir) if "input" in f])
    target_paths = sorted([os.path.join(image_dir, f.replace("input", "target")) for f in os.listdir(image_dir) if "target" in f])

    inputs = [load_image(p) for p in input_paths]
    targets = [load_image(p) for p in target_paths]
    dataset = tf.data.Dataset.from_tensor_slices((inputs, targets))
    return dataset

# Data augmentation
def data_augmentation(input_image, target_image):
    if tf.random.uniform(()) > 0.5:
        input_image = tf.image.flip_left_right(input_image)
        target_image = tf.image.flip_left_right(target_image)
    return input_image, target_image

```

FIGURE 4 – Prétraitement des Données

- **Sortie** : La couche de sortie génère l'image résultante avec une activation **tanh**, ce qui permet de s'assurer que les valeurs des pixels sont normalisées dans la plage $[-1, 1]$.

2.1.2 Prétraitement des Données

Les données d'entrée se composent de paires d'images, où chaque paire contient une image source (dans une police de caractères donnée) et une image cible (dans la police souhaitée). Ces images sont prétraitées en les redimensionnant à 256x256 pixels et en les normalisant, de sorte que les valeurs des pixels soient comprises entre 0 et 1.

2.1.3 Entraînement du Modèle

Le modèle est entraîné pendant 50 époques avec une taille de lot de 32. Il utilise l'optimiseur Adam et la fonction de perte d'erreur quadratique moyenne (MSE). Pour améliorer la généralisation, une augmentation des données est effectuée en appliquant des transformations aléatoires telles que le retournement horizontal des images. Un callback personnalisé permet d'afficher les images générées à la fin de chaque époque.

2.1.4 Fonctionnement du Modèle

Le modèle reçoit une image d'entrée dans une police de caractère et génère une image correspondante dans une autre police. Pendant l'entraînement, les

images de sortie sont comparées aux images cibles à l'aide de la fonction de perte, et le modèle apprend à minimiser cette erreur pour améliorer la qualité des images générées.

2.1.5 Résultats

Le modèle CNN a été entraîné sur 50 époques avec un lot de 32, en utilisant la fonction de perte MSE et l'optimiseur Adam¹ (1×10^{-4}). Bien qu'il génère des images de texte avec un style similaire à la cible, certaines zones apparaissent trop sombres, rendant le texte peu lisible.

Ces problèmes sont liés à la complexité du transfert de style, qui nécessite de préserver le contenu tout en adaptant les caractéristiques visuelles. Malgré l'utilisation de couches profondes et de connexions *skip*, des artefacts comme des zones sombres et des lettres floues persistent, ce qui indique la nécessité d'ajustements dans l'architecture ou le prétraitement des données.

Enfin, la qualité est limitée par la résolution des images et la complexité des styles. Des améliorations supplémentaires sont indispensables pour garantir une meilleure lisibilité du texte généré.

1. L'optimiseur Adam (Adaptive Moment Estimation) est un algorithme de descente de gradient robuste et largement utilisé en apprentissage profond. Il est particulièrement efficace pour entraîner des réseaux de neurones complexes, car il ajuste dynamiquement les taux d'apprentissage (learning rate) en fonction des gradients.

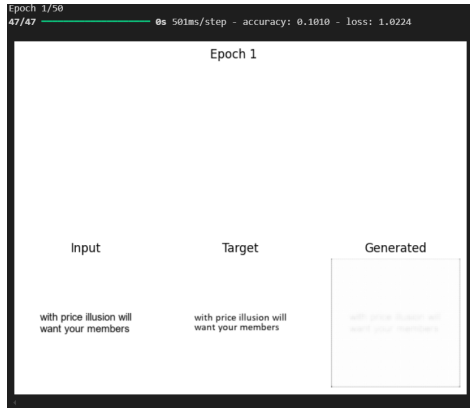


FIGURE 5 – Epoch 1

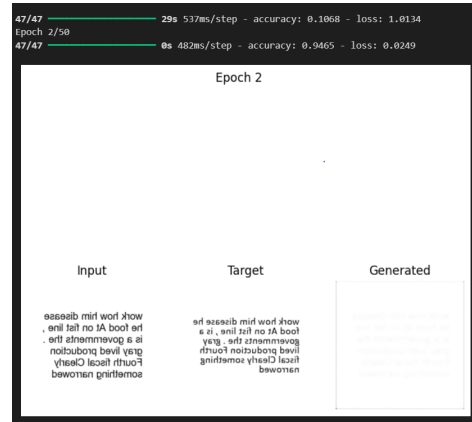


FIGURE 6 – Epoch 2

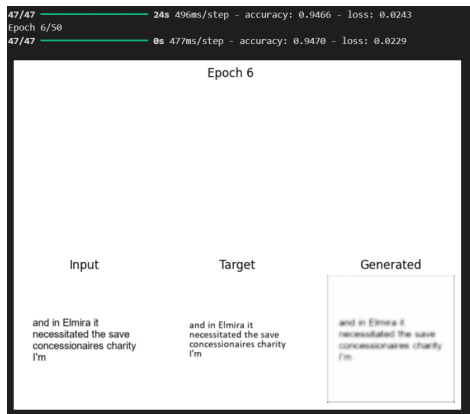


FIGURE 7 – epoch 6

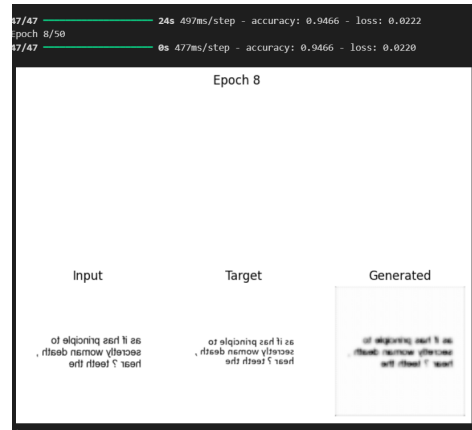


FIGURE 8 – Epoch 8

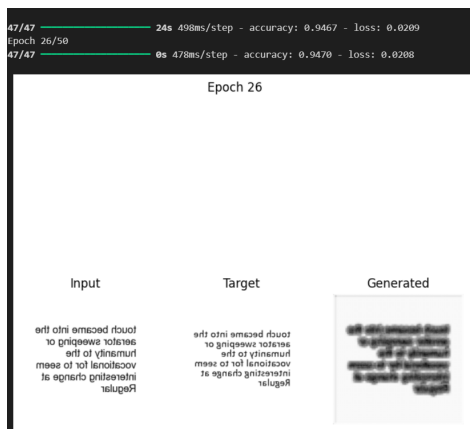


FIGURE 9 – Epoch 26

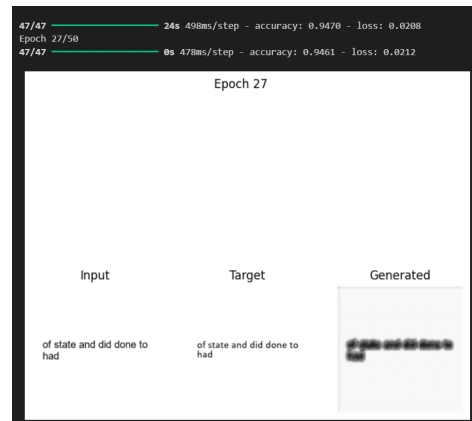


FIGURE 10 – Epoch 27

2.2 Modèle duo cGAN classificateur

Composée de deux sous models : Classificateur et Generateur GAN conditionnel

- **Le modèle cGAN (Conditional Generative Adversarial Network)** : représente une avancée majeure dans la génération et la discrimination d'images, permettant aux machines d'apprendre par elles-mêmes de manière plus précise.

Basé sur le concept des GANs (Generative Adversarial Networks), ce modèle repose sur une interaction continue entre deux réseaux neuronaux : le **générateur** et le **discriminateur**. Le générateur a pour mission de produire des images artificielles les plus réalistes possible afin de tromper le discriminateur, tandis que ce dernier doit différencier les images issues de la banque de données (qu'il identifie comme vraies) de celles générées par le générateur (considérées comme fausses). Grâce à un mécanisme de rétroaction basé sur des récompenses positives et négatives, le discriminateur apprend progressivement à affiner sa capacité de classification.

Simultanément, le générateur s'améliore également pour produire des images toujours plus convaincantes, tout cela orchestré par l'algorithme de Descente de Gradient. Ce processus collaboratif et compétitif permet au modèle de créer des objets de plus en plus réalistes et pertinents, illustrant la puissance de l'apprentissage par confrontation dans l'intelligence artificielle.

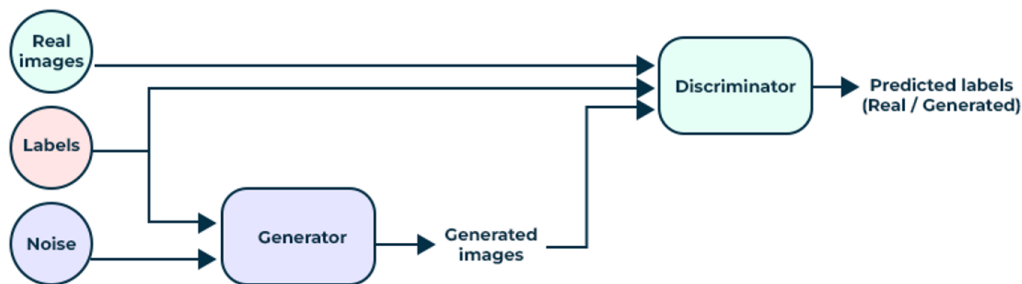


FIGURE 11 – Conditional GAN architecture

- **Classificateur** : il s'agit d'un modèle simple basée sur les CNNs. Configurée avec binary crossentropy comme loss function, accuracy comme metrique d'évaluation, et Adam comme optimizer. Nous avons obtenu accuracy de 98%.

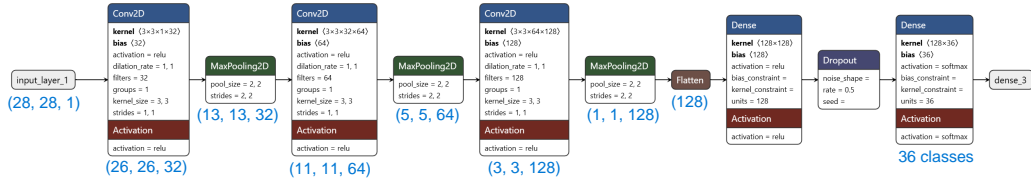


FIGURE 12 – Classifier architecture

2.2.1 Étapes d'Entraînement de cGAN

1. **Préparation des Données** : Cela inclut le chargement des images réelles (x_{train}) et de leurs labels correspondants (y_{train}). Les images normalisée entre -1 et 1, car la fonction d'activation **tanh** dans le générateur redimensionne la sortie dans cette plage.
2. **Initialisation des Modèles** : Définition des modèles **Générateur** (g_{model}) et **Discriminateur** (d_{model}).

Le **Générateur** prend deux entrées : un vecteur latent/noise (z) et un label ($label$). Il produit une image conditionnée par le label. L'idée est de prendre un ce vecteur de faible dimension (par exemple, un vecteur de bruit aléatoire de taille $latent_dim = 100$) et de le transformer en une sortie plus grande et plus structurée (image).

Le **Discriminateur** prend une image et un label en entrée et sort une probabilité indiquant si l'image est réelle ou générée (fausse).

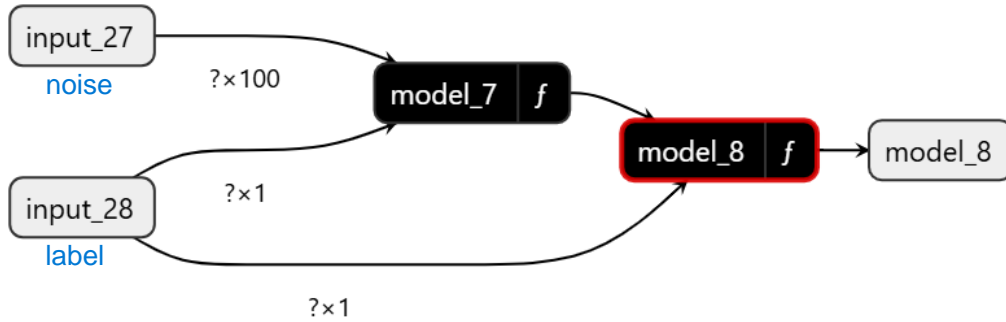


FIGURE 13 – Architecture cGAN

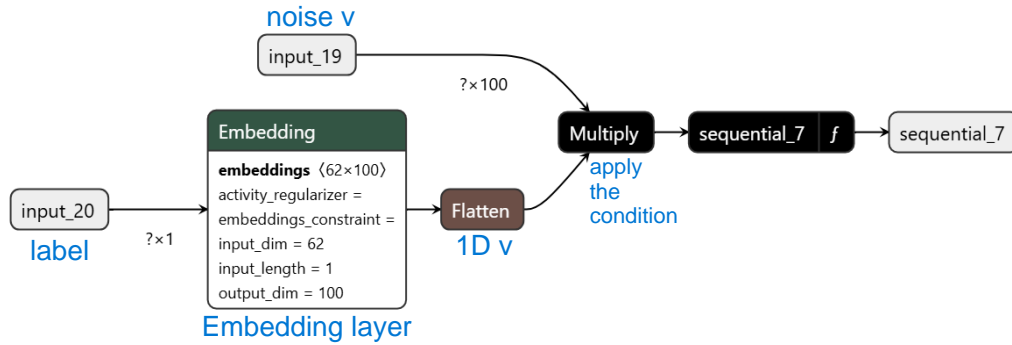


FIGURE 14 – Générateur (model_7)

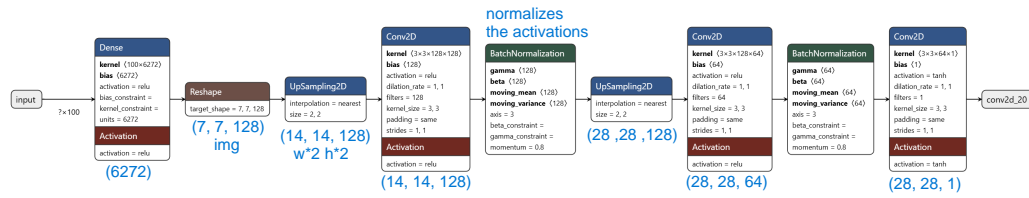


FIGURE 15 – sequential_7

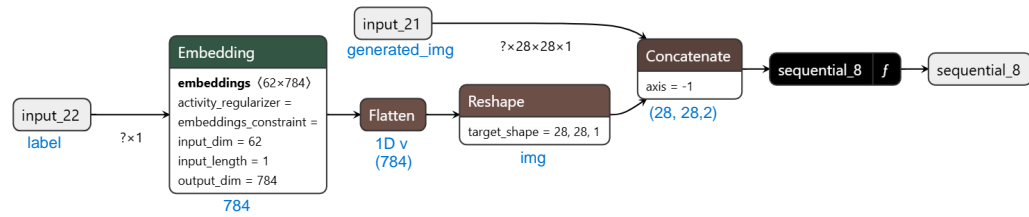


FIGURE 16 – Discriminateur (model_8)

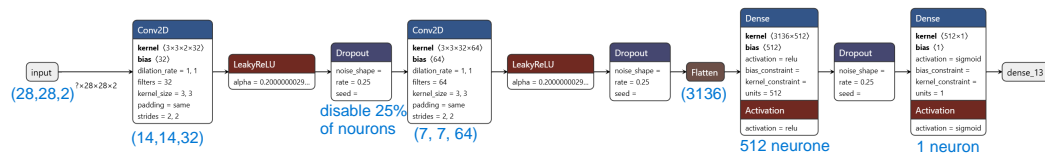


FIGURE 17 – sequential_8

3. **Entraînement du Discriminateur** : Le discriminateur est entraîné à différencier les images réelles des images générées (fausses), en ajustant ses poids pour améliorer l'exactitude de la classification.

— **Images réelles** : Les images réelles provenant de l'ensemble d'entraînement sont envoyées dans le discriminateur avec leurs labels, et la perte est calculée en utilisant les labels réels (`valid`).

- **Images générées** : Les images générées par le générateur sont envoyées dans le discriminateur avec leurs labels, et la perte est calculée en utilisant les labels fausse (**fake**).

Mis à jour les poids du discriminateur en fonction de ces pertes.

4. **Entraînement du Générateur** : Le générateur est entraîné indirectement via le cGAN combiné. Ce dernier intègre à la fois le générateur et le discriminant (fixé comme non entraînable). L'objectif est que le générateur produise des images réalistes conditionnées sur les labels, capables de tromper le discriminant.
5. **Itérations sur les Époques** : Pendant chaque époque, l'entraînement du discriminateur et du générateur sont alternée. À chaque itération :
 - Le **discriminateur** est mis à jour avec les pertes des images réelles et fausses.
 - Le **générateur** est mis à jour pour améliorer la qualité des images générées et tromper le discriminateur.

2.2.2 Pipeline de Génération

Nous avons testé ce modèle duo avec des images du dataset contenant des caractères alphanumériques. Pipeline pour générer :

1. L'utilisateur fournit une image contenant un caractère écrit dans le font A.
2. Classifier l'image et obtenir une classe (label) en sortie (e.g., des entiers représentant des classes).
3. Générer une image correspondante à cette classe dans le font B avec le générateur cGAN.

2.2.3 Résultats

Voici le résultat généré par cGAN pour plusieurs classes donnée, après entraînement de 30000 epochs, batch de taille 128

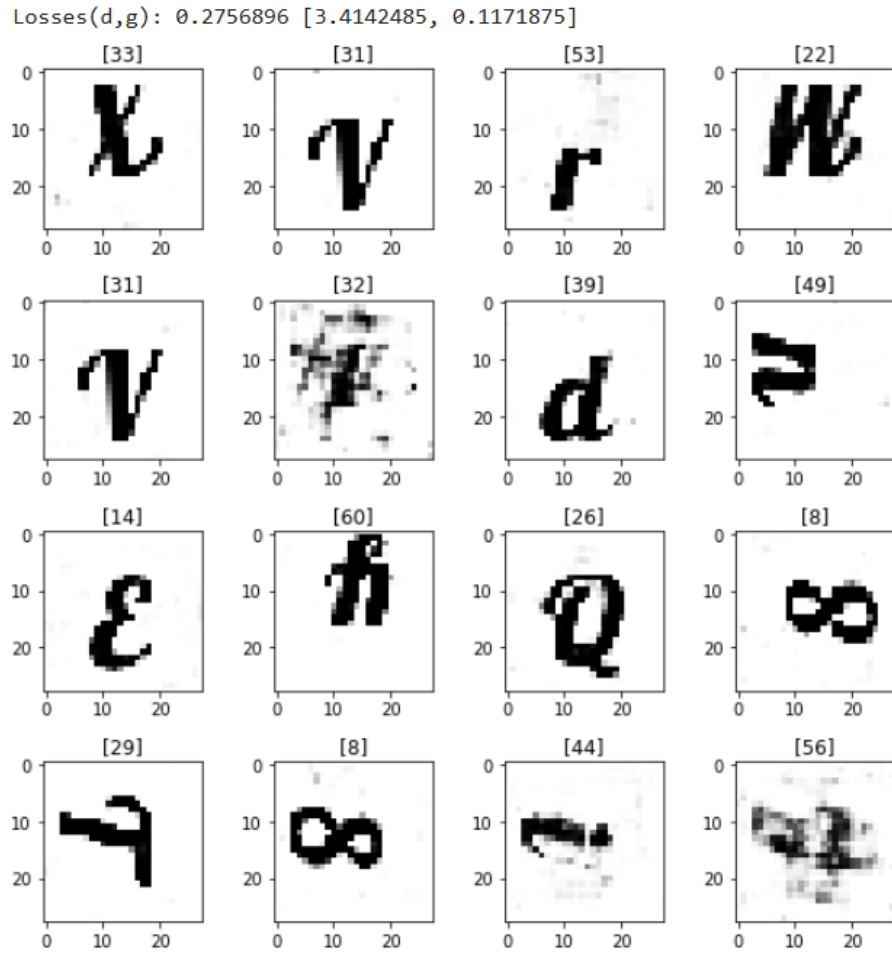


FIGURE 18 – Résultat cGAN Algerian font

2.2.4 Problèmes

1. Comme vous l'avez peut-être remarqué dans l'image précédente, avec le modèle actuel, il n'y a pas de contrôle concernant la taille et l'orientation des caractères générés.
2. Il n'y a pas de moyen de le généraliser à du texte.

2.3 Modèle Pix2Pix

Nous avons exploité plusieurs types de réseaux de neurones pour trouver le meilleur qui s'adapte à notre problème (transfert de style de police), l'un de ces modèles est **Pix2Pix** que nous allons explorer dans cette sous-section. Ce modèle est une variante de **CGAN**, spécialisé dans le transfert image-image. Nous allons explorer les différentes étapes de création de ce modèle et discuter de ses résultats.

2.3.1 Préparation des données

Les données utilisées dans ce projet proviennent du premier dataset contenant des paires d'images de texte, avec une image d'entrée (texte original) et une image cible (texte stylisé). Le dataset est organisé dans un répertoire contenant des fichiers au format **PNG**, où chaque image d'entrée est accompagnée de son image cible correspondante. Les images sont redimensionnées à une taille de **256x256** pixels pour être utilisées par le modèle. De plus, une normalisation est effectuée pour amener les valeurs des pixels dans la plage de $[-1, 1]$, ce qui est essentiel pour l'entraînement des modèles **GAN**.

2.3.2 Augmentation des données

Une augmentation des données a été appliquée pour améliorer la généralisation du modèle et éviter le surapprentissage. Cette augmentation inclut un retournement horizontal aléatoire des images d'entrée et des images cibles, ce qui permet d'augmenter la diversité des données d'entraînement.

2.3.3 Architecture du modèle

1. Générateur (U-Net) Entrée et Sortie

- Entrée : Une image de taille $256 \times 256 \times 3$ (niveau de gris, 1 canal).
- Sortie : Une image de taille $256 \times 256 \times 3$ (niveau de gris, 1 canal).

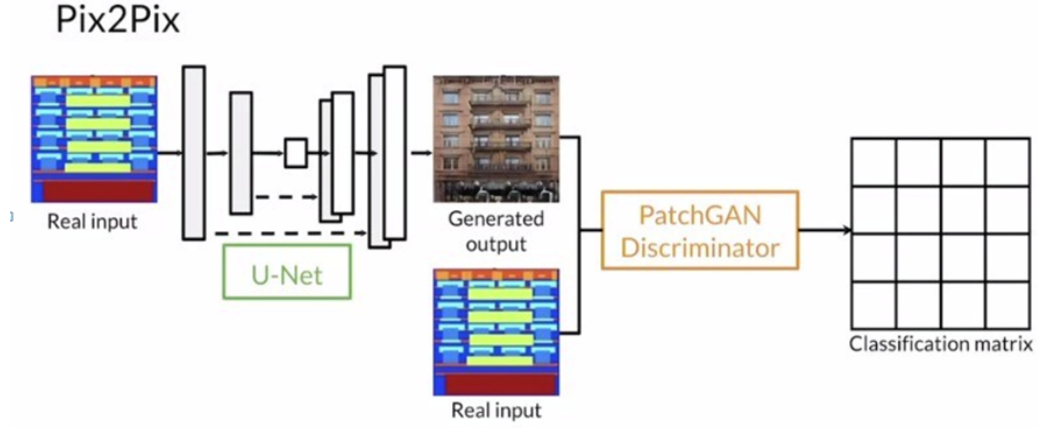


FIGURE 19 – l'architecture Pix2Pix

- (a) **Partie Descendante (Encodage)** La partie descendante du U-Net utilise des convolutions pour extraire des caractéristiques de bas niveau à partir de l'image d'entrée.

Couche	Dimensions Entrée	Opération	Dimensions Sortie	Taille du Kernel	Nombre de Kernels	Stride
1	256×256×1	Conv2D+Relu	128×128×64	4×4	64	2
2	128×128×64	Conv2D+BatchNorm+Relu	64×64×128	4×4	128	2
3	64×64×128	Conv2D+BatchNorm+Relu	32×32×256	4×4	256	2
4	32×32×256	Conv2D+BatchNorm+Relu	16×16×512	4×4	512	2
5	16×16×512	Conv2D+BatchNorm+Relu	8×8×512	4×4	512	2
6	8×8×512	Conv2D+BatchNorm+Relu	4×4×512	4×4	512	2
7	4×4×512	Conv2D+BatchNorm+Relu	2×2×512	4×4	512	2
8	2×2×512	Conv2D	1×1×512	4×4	512	2

TABLE 1 – Partie descendante de l'architecture du générateur (U-Net) pour des images en niveaux de gris.

- (b) **Partie Montante (Décodage)** La partie montante utilise des transpositions de convolution (ou convolutions fractionnelles) pour restaurer les dimensions originales de l'image.

Couche	Dimensions Entrée	Opération	Dimensions Sortie	Taille du Kernel	Nombre de Kernels	Stride
1	1×1×512	Transposed Conv+Relu	2×2×512	4×4	512	2
2	2×2×1024	Transposed Conv+BatchNorm+Relu	4×4×512	4×4	512	2
3	4×4×1024	Transposed Conv+BatchNorm+Relu	8×8×512	4×4	512	2
4	8×8×1024	Transposed Conv+BatchNorm+Relu	16×16×512	4×4	512	2
5	16×16×1024	Transposed Conv+BatchNorm+Relu	32×32×256	4×4	256	2
6	32×32×512	Transposed Conv+BatchNorm+Relu	64×64×128	4×4	128	2
7	64×64×256	Transposed Conv+BatchNorm+Relu	128×128×64	4×4	64	2
8	128×128×128	Transposed Conv+ReLU	256×256×1	4×4	1	2

TABLE 2 – Partie montante de l'architecture du générateur (U-Net) pour des images en niveaux de gris.

- (c) **Caractéristiques Importantes**

- **Skip Connections** : Chaque couche de l’encodage est connectée à la couche correspondante du décodage, permettant de transférer des détails de bas niveau directement vers les couches de sortie.
- **Activation** : ReLU est utilisé pour toutes les couches sauf la dernière, où une activation tanh est appliquée.
- **Taille des Batches** : Le modèle est entraîné avec des lots de 32 images à la fois.
- **Buffer Size** : Un buffer de 10 000 images est utilisé pour améliorer les performances de chargement des données.

(d) **Calcul des pertes** Trois types de pertes sont utilisés pour entraîner le générateur :

- **GAN Loss** :² Cette perte permet de guider le générateur pour qu’il produise des images réalistes en trompant le discriminateur.
- **L1 Loss** : Cette perte permet de préserver la structure du texte en encourageant une reconstruction proche de l’image cible.
- **Style Loss** : La perte de style est calculée en utilisant la matrice de Gram,³ ce qui permet de capturer les caractéristiques stylistiques de l’image cible et de les appliquer au texte généré.

2. **Discriminateur** Le discriminateur du modèle Pix2Pix est basé sur une architecture appelée PatchGAN, conçue pour discriminer les patches locaux d’une image plutôt que l’image entière. Cela aide à capturer les détails locaux, ce qui est essentiel pour des tâches comme la génération d’images.

2.

i. **gan loss of generator** :

$$\mathcal{L}_{cGAN}(G, D) = E_{x,y} [\log D(x, y)] + E_{x,z} [\log (1 - D(x, G(x, z)))]$$

ii. **Gan loss for discriminator**

$$\mathcal{L}_D = -E_{x \sim p_{data}(x)} [\log D(x)] - E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

3. La matrice de Gram est utilisée pour capturer les corrélations entre les différentes activations dans un réseau de neurones convolutifs, représentant ainsi la texture ou le style d’une image. Elle est calculée en multipliant la matrice des activations de chaque couche par sa transposée. Cela permet d’obtenir une représentation des interactions de style entre les différentes parties de l’image.

(a) **Architecture du Discriminateur dans Pix2Pix (PatchGAN)**

- **Entrée :** Une image concaténée (l'image d'entrée $256 \times 256 \times 3$ et l'image générée $256 \times 256 \times 3$), de dimensions $256 \times 256 \times 6$.
- **Sortie :** Une matrice de probabilité de taille $30 \times 30 \times 1$, chaque valeur indiquant si le patch local correspondant est réel ou généré.

Couche	Dimensions Entrée	Opération	Dimensions Sortie	Taille du Kernel	Nombre de Kernels	Stride
1	$256 \times 256 \times 1$	Conv2D+LeakyReLU	$128 \times 128 \times 64$	4×4	64	2
2	$128 \times 128 \times 64$	Conv2D+BatchNorm+LeakyReLU	$64 \times 64 \times 128$	4×4	128	2
3	$64 \times 64 \times 128$	Conv2D+BatchNorm+LeakyReLU	$32 \times 32 \times 256$	4×4	256	2
4	$32 \times 32 \times 256$	Conv2D+BatchNorm+LeakyReLU	$16 \times 16 \times 512$	4×4	512	2
5	$16 \times 16 \times 512$	Conv2D	$15 \times 15 \times 1$	4×4	1	2

TABLE 3 – Architecture du Discriminateur (PatchGAN) pour des images en niveaux de gris.

2.3.4 Entraînement

L'entraînement a été effectué avec 400 époques. Les images ont été chargées et prétraitées avant d'être envoyées dans le modèle pour l'entraînement. L'optimisation a été réalisée en utilisant l'algorithme Adam, avec des taux d'apprentissage ajustés spécifiquement pour le générateur et le discriminateur. Le modèle a été formé en utilisant la stratégie MirroredStrategy pour entraîner le modèle sur plusieurs GPU, ce qui permet d'optimiser l'utilisation des ressources matérielles disponibles.

2.3.5 Résultats

Au fur et à mesure des époques d'entraînement, le modèle a montré une diminution progressive des pertes du générateur (Gen Loss) et du discriminateur (Disc Loss), ce qui indique une amélioration des performances du modèle.

Les résultats des premières époques montrent les pertes suivantes :

- Epoch 1 : Gen Loss = 8.16, Disc Loss = 1.39
- Epoch 2 : Gen Loss = 6.32, Disc Loss = 1.38
- Epoch 3 : Gen Loss = 5.47, Disc Loss = 1.36
- ...
- Epoch 15 : Gen Loss = 5.16, Disc Loss = 1.19

Cela montre que, bien que les images générées deviennent plus réalistes, le modèle nécessite davantage d'époques et un ajustement des hyperparamètres

pour atteindre une qualité visuelle satisfaisante. Des améliorations sont possibles par l’augmentation de la taille du dataset et l’ajustement des poids de la fonction de perte.

2.3.6 Exemple de génération

Voici un exemple des images générées après 15 époques :

Epoch 12/400: Gen Loss: 5.052728652954102, Disc Loss: 1.2402054071426392



Epoch 13/400: Gen Loss: 5.078715801239014, Disc Loss: 1.2256262302398682



FIGURE 20 – Un exemple de generation

- Image d’entrée : Un texte écrit dans la police Arial.
- Image cible : Un texte dans la police Calibri.
- Image générée : Le texte de l’entrée converti en style Calibri par le modèle Pix2Pix.

2.4 Modèle cycleGAN

Cette partie implémente un modèle de transfert de style basé sur un CycleGAN. L'objectif principal est de former deux générateurs (Gen_G et Gen_F) et deux discriminateurs (Disc_Font1 et Disc_Font2) pour effectuer une conversion bidirectionnelle entre deux ensembles d'images (polices d'écriture) tout en préservant la cohérence du texte.

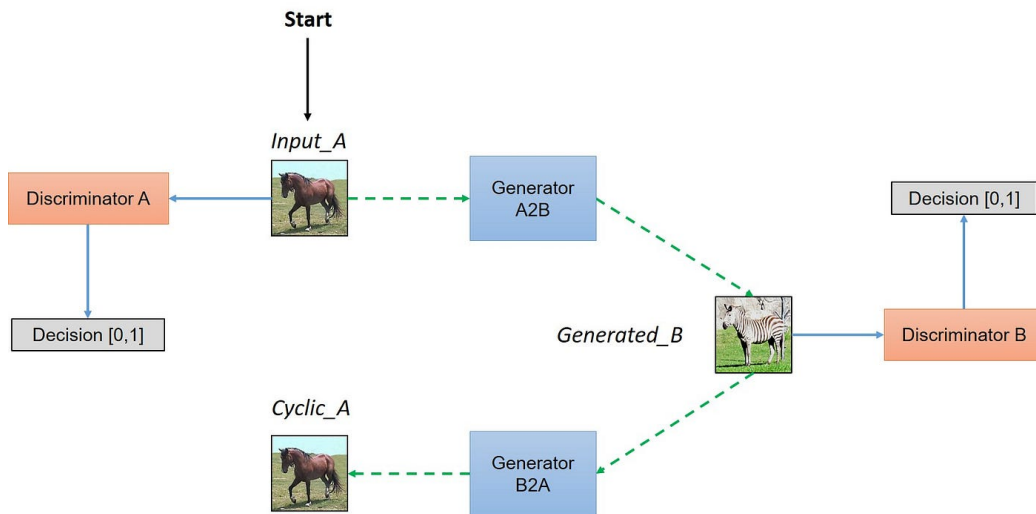


FIGURE 21 – Cycle GAN

2.4.1 Chargement et Prétraitement des Données

- **Dossier des données :** Les images sont chargées depuis un répertoire contenant des paires d'images d'entrée ("_input_") et de cibles ("_target_").
- **Prétraitement des images :**
 - Les images sont converties en niveaux de gris, redimensionnées à une taille fixe (256x256), et normalisées dans l'intervalle $[-1, 1]$.⁴

4. Normaliser dans l'intervalle $[-1, 1]$ permet :

1. **D'accélérer l'entraînement des réseaux neuronaux** en s'assurant que les entrées aient une échelle cohérente et centrée autour de 0, ce qui facilite la convergence des algorithmes d'optimisation comme l'Adam.
2. **De maintenir la cohérence avec l'activation des fonctions des modèles génératifs**, notamment les tanh, qui produisent des sorties dans cet intervalle.
3. **De réduire le risque d'explosions ou de saturations dans les gradients**, rendant l'entraînement plus stable.

- Chaque image est ajoutée comme tenseur avec une seule dimension de canal.

2.4.2 Architecture des Modèles

Générateur (Gen_G et Gen_F) :

- **Entrée** : Une image en niveaux de gris de dimension $256 \times 256 \times 1$.
- **Structure** :
 - **Couche 1** : Convolution avec 64 filtres (3×3), activation **ReLU**, pas de 1.
 - **Couche 2** : Convolution avec 128 filtres (3×3), activation **ReLU**, réduction de taille par un pas de 2.
 - **Couche 3** : Convolution transposée avec 64 filtres (3×3), activation **ReLU**, augmentation de taille par un pas de 2.
 - **Couche 4 (sortie)** : Convolution avec 1 filtre (3×3), activation **tanh**.
- **Sortie** : Une image transformée de taille $256 \times 256 \times 1$.

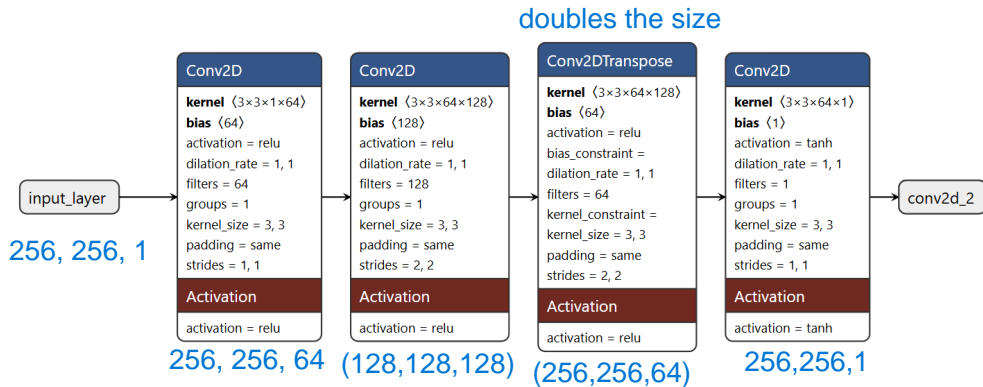


FIGURE 22 – Architecture du model generateur

Discriminateur (Disc_Font1 et Disc_Font2) :

- **Entrée** : Une image en niveaux de gris de dimension $256 \times 256 \times 1$.
- **Structure** :
 - **Couche 1** : Convolution avec 64 filtres (4×4), activation **ReLU**, réduction de taille par un pas de 2.
 - **Couche 2** : Convolution avec 128 filtres (4×4), activation **ReLU**, réduction de taille par un pas de 2.
 - **Couche 3** : Aplatissement des caractéristiques et passage par une couche dense avec 1 neurone, activation 'sigmoid'.

— **Sortie** : Une valeur indiquant si l'image est réelle ou générée.

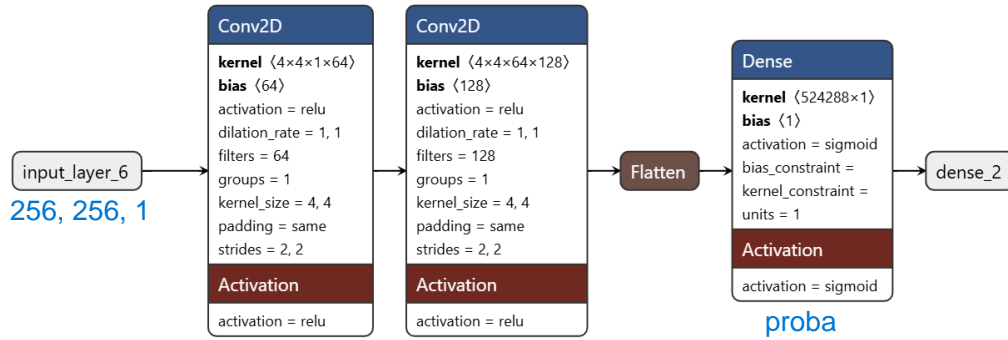


FIGURE 23 – Architecture du model discriminateur

2.4.3 Fonctions de Perte

- **Perte Adversariale** : Elle mesure à quel point le discriminateur accomplit bien sa tâche de classification, ce qui permet de guider l'entraînement à la fois du discriminateur et du générateur. Elle se calcule en minimisant la distance entre les sorties des discriminateurs pour les images générées et les vraies.
- **Perte de Cohérence Cyclique** : Imposée pour garantir que l'image transformée revient à son état d'origine après un cycle complet de transformation (Font1 -> Font2 -> Font1). Elle se calcule en moyennant la somme des différences absolues entre les pixels de l'image réelle et de l'image cyclée.

2.4.4 Optimisation

- Optimiseurs Adam utilisés pour les générateurs et les discriminateurs.
- Paramètres : Taux d'apprentissage initial 2×10^{-5} et $\beta_1 = 0.5$.

2.4.5 Étapes d'Entraînement

1. Passage avant :

- Les générateurs transforment les images d'entrée et produisent des images générées.
- Les discriminateurs évaluent les vraies et fausses images.
- Calcule des pertes.

2. Calcul des gradients :

- Les gradients des générateurs et des discriminateurs sont calculés pour leurs pertes respectives.

3. Mise à jour des poids :

- Les poids des générateurs et des discriminateurs sont mis à jour en utilisant leurs gradients.

2.4.6 Résultats

Le modèle a commencé de générer des images très similaires à celle de l'entrée, mais on ne voit aucune trace du fond cible.

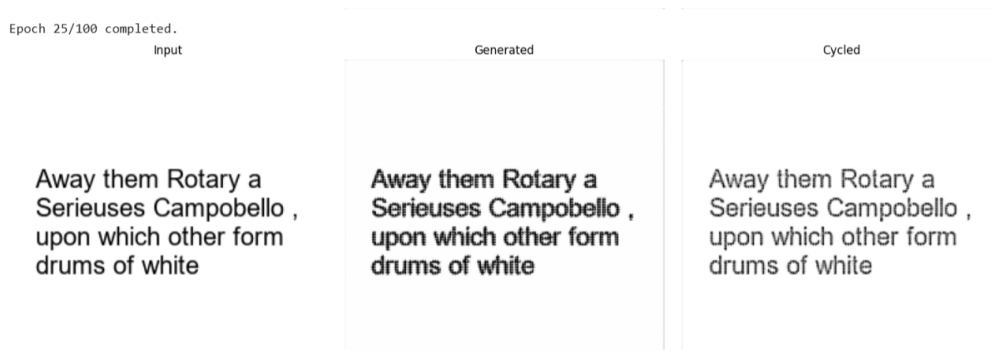


FIGURE 24 – Résultats après 25 epochs

Au fil des epochs, les images ont commencé à devenir bruyantes.

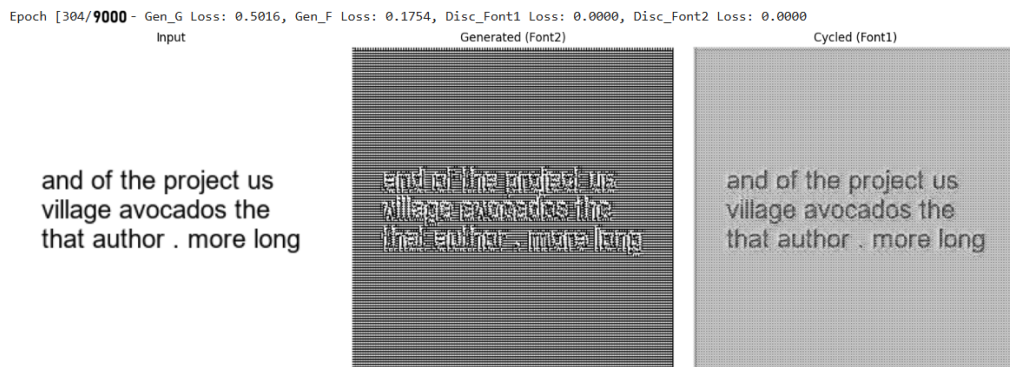


FIGURE 25 – Résultats apres 300 epochs

Malheureusement, nous n'avons pas pu l'entraîner plus, car pour atteindre 304 epochs, il a pris plus de 2 heures et ensuite Kaggle a simplement planté.

3 L'interface utilisateur pour le text font transfer :

L'interface utilisateur a été développée avec Gradio, une bibliothèque Python permettant de créer des interfaces web interactives pour les modèles de machine learning. L'interface offre les fonctionnalités suivantes :

1. Téléchargement d'une image contenant du texte.
2. Sélection de la police d'entrée et de sortie parmi les options disponibles.
3. Affichage de l'image transformée avec possibilité de téléchargement.

L'organisation de l'interface est simple et intuitive, divisée en deux colonnes :

1. Téléchargement de l'image et sélection des polices elle offre aussi une possibilité de capturer une image directement en utilisant une caméra.
2. Affichage du résultat transformé avec une icône de téléchargement.

3.1 Simulation de transfert de style (Mock Test) :

Étant donné les limitations actuelles du modèle, une version simulée a été implémentée pour tester et visualiser le flux de l'application. Cette approche applique des modifications simples aux images d'entrée, comme l'ajout de motifs et de textes simulant la transformation vers la police sélectionnée. Cette méthode permet de présenter l'interface utilisateur et de vérifier le fonctionnement des différents éléments sans dépendre d'un modèle d'apprentissage complexe.

```
# Fonction simulée pour le transfert de style
def mock_font_style_transfer(image, input_font, output_font):
    # Redimensionner l'image au format attendu
    image = image.resize((256, 256)).convert("L") # Conversion en niveaux de gris
    image_array = np.array(image) # Convertir en tableau NumPy

    # Simuler une transformation en dessinant des lignes ou des motifs sur l'image
    simulated_image = Image.fromarray(image_array).convert("RGB")
    draw = ImageDraw.Draw(simulated_image)
    draw.rectangle([10, 10, 118, 118], outline="red", width=3)
    draw.text((20, 20), f"{output_font}", fill="blue")

    return simulated_image
```

FIGURE 26 – simulation

Mais sinon si on voulait utiliser un modèle pour générer des images le code aurait été de cette façon : Cette méthode utilise un modèle d'apprentissage

profond préentraîné (TensorFlow). L'image d'entrée est redimensionnée et normalisée pour être traitée par le modèle. Celui-ci génère une nouvelle image où le texte est transformé dans le style de la police souhaitée. Le modèle accepte des images en niveaux de gris, les redimensionne à une taille standard (256x256) et génère une version stylisée. Deux polices de base, "Times New Roman" et "Lobster", sont proposées pour le transfert de style

```

5
6 # Charger le modèle
7 model = tf.keras.models.load_model("model.h5")
8
9 # Liste des polices disponibles
10 input_fonts = ["Times New Roman", "Lobster"]
11 output_fonts = ["Times New Roman", "Lobster"]
12
13 # Fonction pour le traitement de l'image
14 def font_style_transfer(image, input_font, output_font):
15     # Redimensionner l'image au format attendu par le modèle
16     image = image.resize((256, 256)).convert("L") # Conversion en niveaux de gris
17     image_array = np.array(image) / 255.0 # Normalisation
18     image_array = np.expand_dims(image_array, axis=(0, -1)) # Préparer pour le modèle
19
20     # Utiliser le modèle pour générer une nouvelle image
21     generated_image = model.predict(image_array)[0]
22     generated_image = (generated_image * 255).astype(np.uint8) # Re-normaliser pour l'affichage
23
24     # Convertir en image PIL pour l'affichage
25     output_image = Image.fromarray(generated_image)
26     return output_image
27

```

FIGURE 27 – interaction avec le model

Voici un aperçu final de l'application :

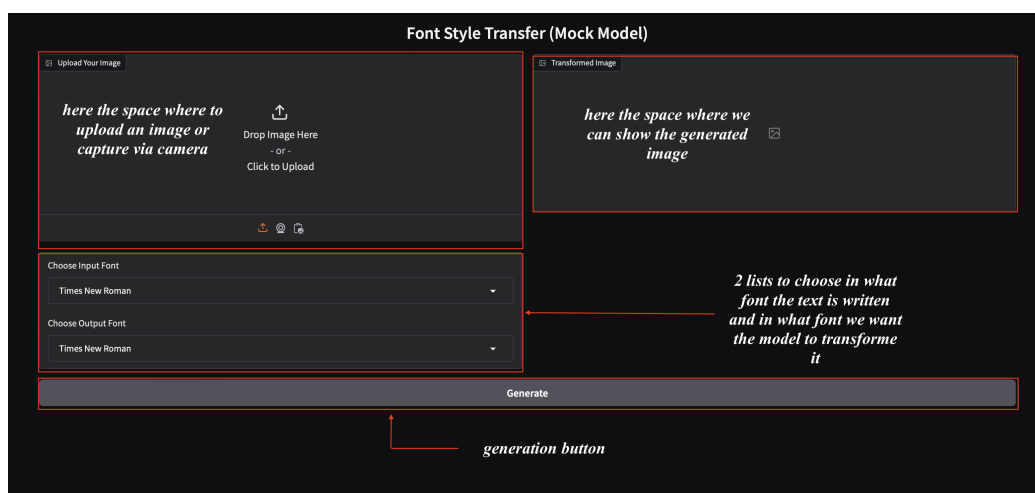


FIGURE 28 – Interface

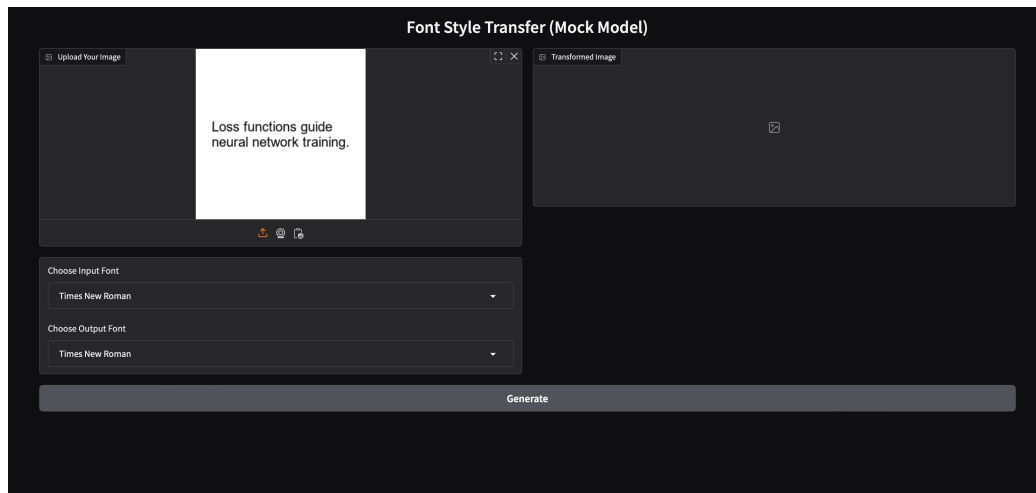


FIGURE 29 – Chargement d'une image

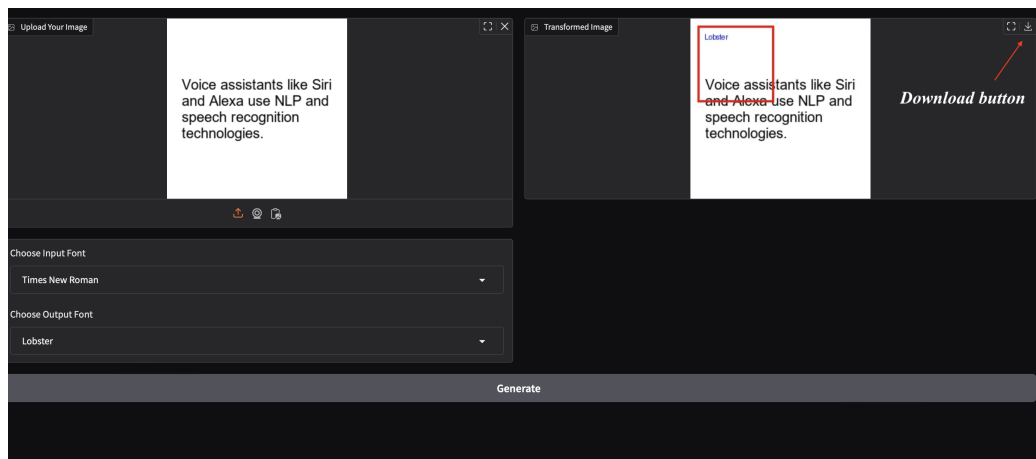


FIGURE 30 – interaction avec le model

4 Modèle de génération de mélodies

Le but de ce projet est de construire un modèle capable de générer des mélodies musicales à partir d'un dataset existant. Cela implique de travailler avec des données musicales (au format Kern) et de prétraiter ces données avant de les utiliser pour entraîner un modèle de deep learning en utilisant des techniques de traitement de données musicales et un réseau neuronal.

4.1 Bibliothèques utilisées

1. **music21** : Utilisée pour analyser et manipuler des données musicales. Elle facilite le chargement des fichiers musicaux, la vérification des durées, et les transpositions.
2. **TensorFlow/Keras** : Fournit les outils pour construire, entraîner, et évaluer le modèle de deep learning.
3. **NumPy** : Utilisé pour manipuler les données sous forme de tableaux et gérer les transformations.
4. **os et json** : Manipulent les fichiers et stockent les mappings (liens entre symboles musicaux et entiers).

Ces bibliothèques offrent un équilibre entre manipulation de données musicales (via music21) et les outils nécessaires pour l'entraînement et la gestion des réseaux de neurones (via TensorFlow/Keras).

4.2 Les étapes principales du model

Étape 1 : Chargement et prétraitement

1. Un dataset a été téléchargé via ce site <http://www.esac-data.org>. Les fichiers musicaux au format Kern (fichiers .krn) sont chargés à l'aide de music21. Chaque fichier est analysé pour s'assurer que les durées des notes respectent des valeurs acceptables (comme des noires, des blanches, etc.).
2. Transposition vers une tonalité commune, encodage et la sauvegarde : Les chansons sont transposées vers C majeur ou A mineur pour réduire la complexité et standardiser les données. Ensuite chaque note ou silence est transformé en une séquence symbolique. Par exemple : Les notes MIDI représentent les hauteurs. Les silences sont représentés par le symbole "r". Les durées sont gérées avec des répétitions du même symbole et enfin chaque chanson prétraitée est sauvegardée dans un fichier texte, ce qui facilite la création d'un fichier global pour l'entraînement.

3. Consolidation des données :Fusionner toutes les chansons encodées dans un fichier unique avec un délimite, ensuite créer un fichier JSON pour mapper chaque symbole (notes, silences) à un entier unique.

Étape 2 : Entraînement du modèle :

Architectur du modèle

L'architecture LSTM (Long Short-Term Memory) est une amélioration des réseaux de neurones récurrents (RNN) qui permet de mieux gérer les dépendances à long terme dans les séquences. Un bloc LSTM se compose généralement de plusieurs composants essentiels :

1. **Mémoire** : C'est le vecteur principal qui transporte l'information à travers le temps. Il peut être modifié par les portes, permettant au réseau de se souvenir ou d'oublier certaines informations.
2. **Porte d'entrée** : Cette porte décide quelles nouvelles informations doivent être ajoutées à la mémoire. Elle combine la sortie du réseau avec les nouvelles informations et détermine leur importance.
3. **Porte d'oubli** : Cette porte décide quelles informations de la mémoire doivent être oubliées. Elle génère un vecteur de valeurs entre 0 et 1 pour chaque valeur de la mémoire (1 signifie "tout garder", 0 signifie "tout oublier").
4. **Porte de sortie** : Elle détermine quelle information de la mémoire sera utilisée pour calculer la sortie cachée, qui est propagée à l'étape suivante du réseau.
5. **État caché** : C'est la sortie du bloc LSTM à un moment donné. Il combine l'information de la mémoire et l'activation de la porte de sortie pour donner une représentation à chaque instant.

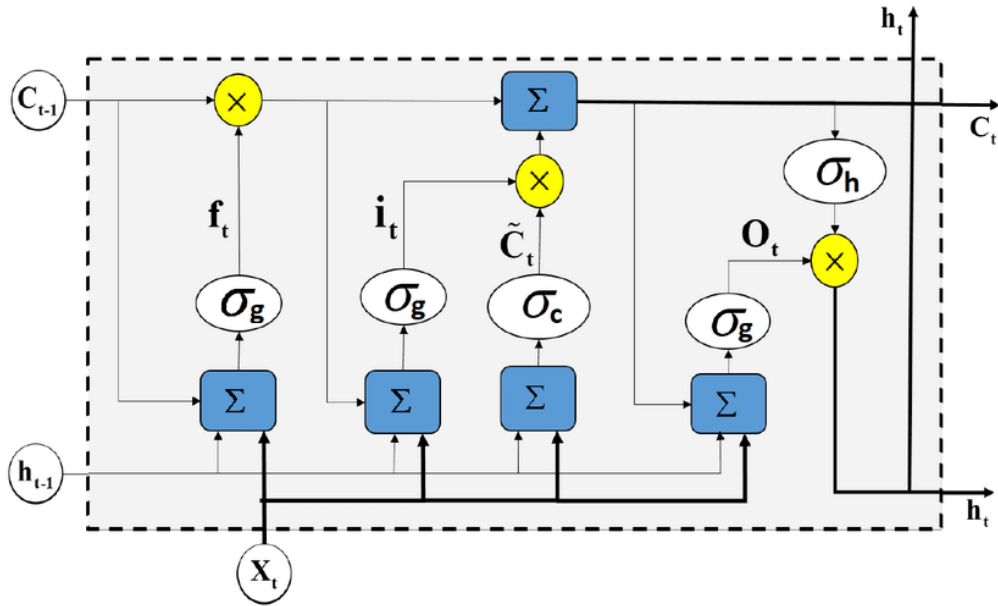


FIGURE 31 – Architecture LSTM

1. **Entrée** : Séquences encodées (représentées sous forme d'entrées One-Hot Encoded).
2. **Couche LSTM** : Capture des dépendances à long terme dans les séquences musicales. Une seule couche avec 256 unités.
3. **Couche de sortie Dense** : Prédit la prochaine note ou silence, activation Softmax pour produire une distribution de probabilités sur toutes les notes possibles.

Hyperparamètres :

- (a) Taille des séquences : 64.
- (b) Perte : **sparse_categorical_crossentropy** (adaptée pour des cibles catégoriques).
- (c) Optimiseur : Adam avec un taux d'apprentissage de 0.001.
- (d) Nombre d'épochs : 50.
- (e) Taille du batch : 64.

Entraînement :

- (a) Génération des données d'entraînement : transformation des chansons encodées en séquences d'entrée et cibles.

(b) Entraînement du modèle sur ces données.

Sauvegarde :

Sauvegarde du modèle entraîné au format HDF5 pour une réutilisation future.

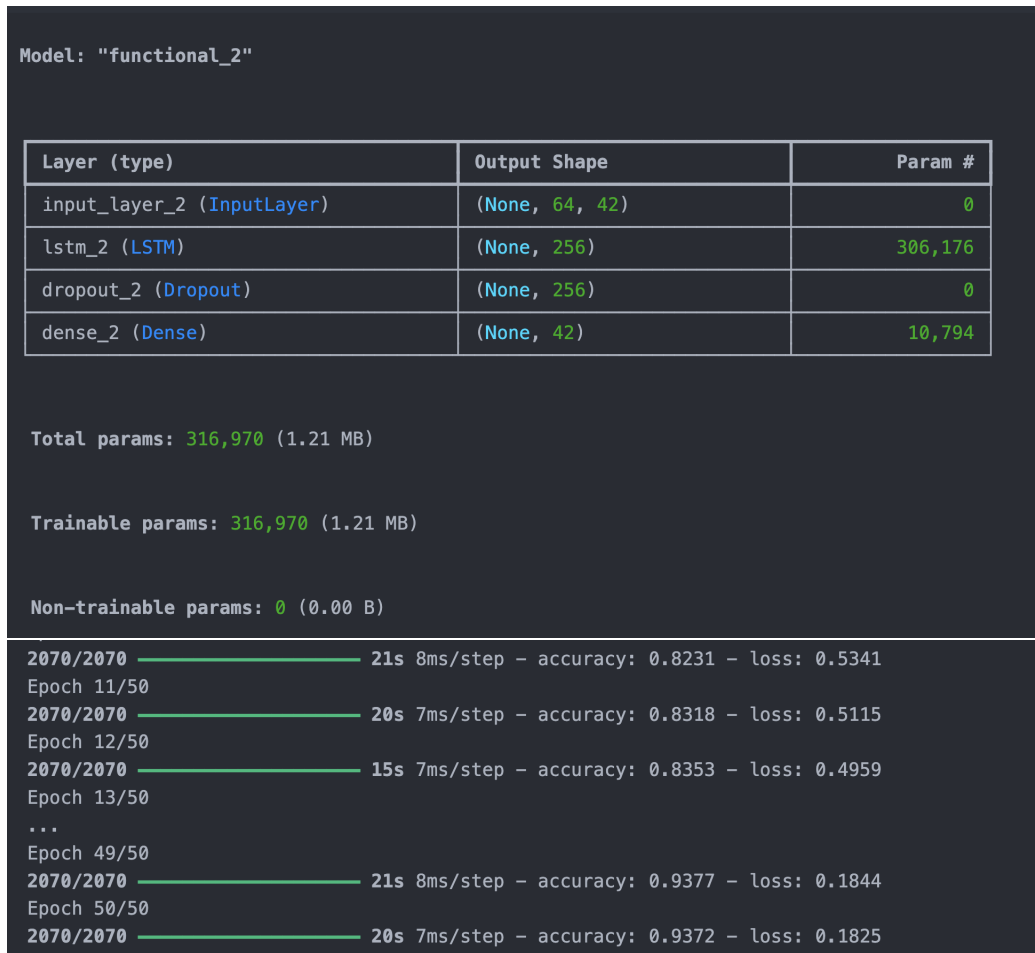


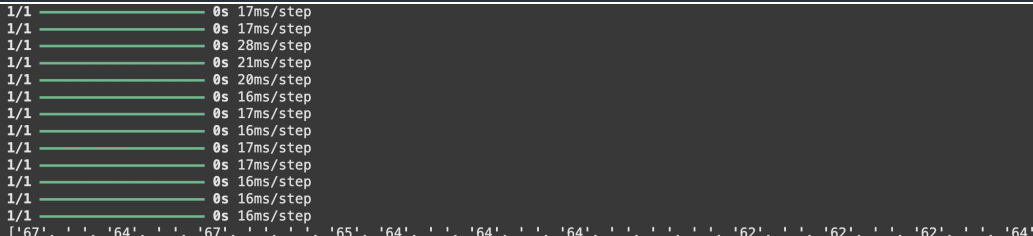
FIGURE 32 – Model Training

Étape 3 : Génération de mélodies :

1. Pour cela Nous Utilisoons le modèle entraîné pour prédire une nouvelle note à partir d'une séquence initiale.
2. Ajouter cette note à la séquence, puis prédire la suivante (boucle).

3. Création de nouvelles mélodies pour des compositions ou de l'accompagnement musical.

```
if __name__ == "__main__":
    mg = MelodyGenerator()
    seed = "67 _ 64 _ 67 _ _ 65 64 _ 64 _ 64 _ _"
    seed2 = "67 _ _ _ _ 65 _ 64 _ 62 _ 60 _ _ _"
    melody = mg.generate_melody(seed, 500, SEQUENCE_LENGTH, 0.3)
    print(melody)
    mg.save_melody(melody)
```



```
1/1 _____ 0s 17ms/step
1/1 _____ 0s 17ms/step
1/1 _____ 0s 28ms/step
1/1 _____ 0s 21ms/step
1/1 _____ 0s 20ms/step
1/1 _____ 0s 16ms/step
1/1 _____ 0s 17ms/step
1/1 _____ 0s 16ms/step
1/1 _____ 0s 17ms/step
1/1 _____ 0s 17ms/step
1/1 _____ 0s 16ms/step
1/1 _____ 0s 16ms/step
1/1 _____ 0s 16ms/step
1/1 _____ 0s 16ms/step
1/1 _____ 0s 16ms/step
['67', '_', '64', '_', '67', '_', '65', '64', '_', '64', '_', '64', '_', '62', '_', '62', '_', '62', '_', '64']
```

FIGURE 33 – Melodie generation

5 Difficultés rencontrées

Lors du projet, nous avons rencontré des obstacles liés aux plateformes en ligne, telles que les limitations de ressources sur Google Colab et Kaggle, qui ont retardé notre progression, notamment en raison de la fin de quota des GPU sur Kaggle et des problèmes de gestion des datasets sur Colab. De plus, l'absence de modèles préexistants de transfert de styles de police a compliqué la phase de conception, nous obligeant à expérimenter différentes approches, ce qui a ajouté une complexité imprévue au projet.

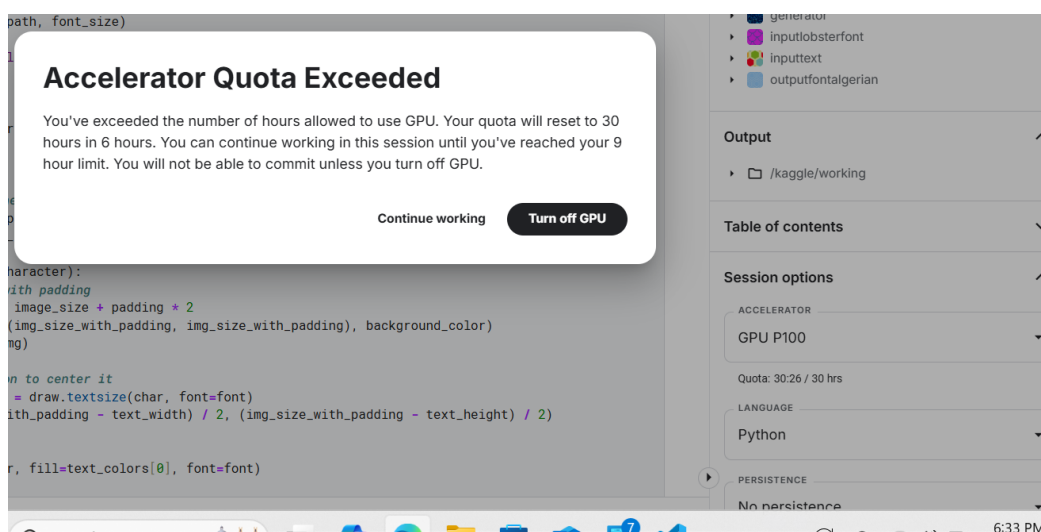


FIGURE 34 – Problèmes avec Kaggle

Conclusion

Ce projet nous a permis d'explorer deux domaines de l'apprentissage profond : le transfert de style de police et la génération musicale. Malgré nos efforts pour ajuster les hyperparamètres, le modèle de transfert de style n'a pas produit de résultats satisfaisants, ce qui montre la complexité de cette tâche et l'importance d'un réglage fin et d'un ensemble de données plus robuste. En revanche, le modèle de génération musicale a montré des résultats prometteurs, bien que son entraînement ait été limité à une petite quantité de données. Cette contrainte a affecté la qualité des résultats finaux. Ces expériences soulignent les défis pratiques liés à l'apprentissage profond et ouvrent la voie à des améliorations futures, notamment en optimisant les données d'entraînement, les architectures de modèles et les interfaces pour rendre ces projets plus complets et accessibles.