RAPPORT DE PROJET IA

# Reconnaissance optique d'écriture manucrite (Handwriting OCR) Architecture CRNN sur le dataset RIMES

**Réalisé par :**
Joachim DRUHET
Quentin DAVIN

*Polytech Dijon*
*Ingénierie Logicielle et Intelligence Artificielle*
*Cycle Ingénieur - 5ème Année*
*Année 2025-2026*

# Table des matières

# 1 Introduction

Si la reconnaissance optique de caractères imprimés est aujourd'hui une technologie mature, le passage à l'écriture manuscrite (*Handwriting OCR*) soulève des défis d'une toute autre nature. Là où une machine lit aisément des lettres standardisées et séparées, l'écriture humaine introduit une variabilité extrême et, surtout, une continuité dans le tracé. En effet, dans l'écriture cursive, les ligatures entre les lettres empêchent toute segmentation simple : il est impossible de découper une image mot par mot ou lettre par lettre sans perdre le contexte. Le problème ne consiste donc plus simplement à classifier une forme isolée, mais à transcrire une séquence temporelle complète à partir d'une image, sans disposer des coordonnées précises de chaque caractère.

Ce projet, réalisé dans le cadre du module d'Intelligence Artificielle et Optimisation encadré par M. Ambard, vise à répondre à cette problématique en construisant une chaîne de traitement complète basée sur le Deep Learning. Notre travail s'appuie sur le dataset RIMES, qui regroupe des courriers administratifs fictifs mais réalistes, pour entraîner un réseau de neurones hybride de type CRNN (*Convolutional Recurrent Neural Network*).

Au-delà de la simple implémentation du modèle avec PyTorch, notre démarche a consisté à maîtriser l'ensemble du pipeline : du nettoyage rigoureux des données brutes jusqu'à l'intégration du modèle final dans une interface utilisateur, en passant par la résolution des problèmes de convergence typiques de ce genre d'architecture.

# 2 Architecture du modèle

## 2.1 Choix de l'Architecture : CRNN

L'architecture retenue est un **CRNN (Convolutional Recurrent Neural Network)**. Ce modèle hybride, devenu la référence pour la reconnaissance de texte, combine la capacité d'extraction visuelle des réseaux convolutifs (CNN) avec la capacité de modélisation séquentielle des réseaux récurrents (RNN).

Le pipeline se décompose en trois blocs distincts :

#### 1. L'extracteur de caractéristiques (CNN)

La première partie du réseau analyse l'image brute. Nous utilisons une série de couches de convolution standard (`Conv2d`) suivies de fonctions d'activation `ReLU` et de normalisation `BatchNorm`. L'objectif est d'extraire des motifs visuels (traits, courbes, boucles) indépendamment de leur position.

*Détail d'implémentation :* Une spécificité importante de notre architecture réside dans les couches de *pooling*. Alors que les premières couches réduisent l'image classiquement en $2 \times 2$, les dernières couches utilisent un *max pooling* rectangulaire de type $(2, 1)$. Cette asymétrie est cruciale : elle permet de réduire la hauteur de l'image (inutile pour le texte qui se lit horizontalement) tout en conservant une résolution maximale sur l'axe de la largeur (l'axe temporel), préservant ainsi la distinction entre des lettres étroites voisines (comme 'i' et 'l').

#### 2. La transformation Map-to-Sequence

En sortie du CNN, nous obtenons un tenseur de caractéristiques de dimension ($batch, channels, heigh$ Pour alimenter le RNN, nous devons transformer ce volume 3D en une séquence temporelle. Nous fusionnons les dimensions *channels* et *height* pour ne garder que l'axe *width* comme axe temporel. Chaque colonne de pixels de l'image traitée devient ainsi un vecteur d'entrée pour le réseau récurrent.

#### 3. La modélisation séquentielle (Bi-LSTM)

La séquence de vecteurs est injectée dans un RNN bidirectionnel (**Bi-LSTM**).

- **LSTM (Long Short-Term Memory) :** Contrairement à un RNN simple, le LSTM possède une mémoire interne qui lui permet de gérer les dépendances à long terme. Cela est indispensable pour l'écriture manuscrite où le début d'un mot influence la lecture de la fin (gestion du contexte).
- **Bidirectionnel :** Le réseau lit la séquence de gauche à droite et de droite à gauche. Cela permet de lever des ambiguïtés graphiques en utilisant le contexte passé et futur d'un caractère.

### 2.2 La CTC Loss (Connectionist Temporal Classification)

L'apprentissage supervisé classique nécessite que chaque entrée soit alignée avec sa sortie. Or, dans notre cas, l'image en entrée est large (ex : 1024px) et produit une séquence de caractéristiques longue (ex : 256 pas de temps), tandis que le label cible est court (ex : le mot "chat"). Nous ne savons pas à quel "pas de temps" correspond le caractère 'c'.

Pour résoudre ce problème sans segmentation manuelle, nous utilisons la **CTC Loss**. Son fonctionnement repose sur deux principes :

1. **Le token "Blank" :** Le réseau prédit une probabilité pour chaque caractère de l'alphabet, plus un caractère spécial *silence* (ou *blank*). Cela permet au modèle de dire "ici, je suis entre deux lettres" ou "je ne suis pas sûr".
2. **Le décodage :** La CTC considère toutes les voies possibles qui mènent au texte cible. Par exemple, pour écrire "bar", le réseau peut prédire `-b-aa-r-` ou `b-a-rr--`. L'erreur calcule la somme des probabilités de tous ces alignements valides et cherche à maximiser cette somme pour la bonne transcription.

Cela permet un entraînement "bout-en-bout" (*end-to-end*) : on donne l'image et le texte au modèle, et il apprend seul à aligner les lettres sur les motifs visuels.

## 3 Pipeline de données et prétraitements

### 3.1 Normalisation et gestion des dimensions

Pour garantir un apprentissage stable, les données brutes doivent subir un traitement rigoureux avant d'entrer dans le réseau. Nous avons mis en place une chaîne de transformations standardisée :

#### 1. Homogénéisation des canaux

Le dataset RIMES contient un mélange d'images en couleurs (RGB) et en niveaux de gris. Pour éviter les erreurs de dimensions lors de la création des batchs (empilement des

tenseurs), nous forçons la conversion de toutes les images en différents niveaux de gris (1 canal). Cela réduit également la charge mémoire sur le GPU.

**2. Redimensionnement intelligent**

Les architectures CRNN imposent généralement une hauteur d'entrée fixe. Nous redimensionnons toutes les images à une hauteur de 32 pixels tout en conservant leur ratio d'aspect original pour ne pas déformer l'écriture. La largeur étant variable, nous appliquons un *padding* (remplissage) avec des pixels blancs à droite de l'image, jusqu'à atteindre une largeur maximale de sécurité (1024 pixels).

**3. Normalisation statistique**

Les valeurs des pixels (de 0 à 255) sont transformées en tenseurs flottants et normalisées sur l'intervalle $[-1, 1]$ (Moyenne 0.5, Écart-type 0.5). Cette étape centre les données autour de zéro, ce qui aide l'algorithme d'optimisation à converger plus vite.

## 3.2 Gestion dynamique des batchs

L'entraînement par batchs présente une difficulté technique : comment gérer des images de largeurs différentes dans un même tenseur ? Nous avons développé une fonction de collation personnalisée (`collate_fn`) qui résout ce problème pour la CTC Loss.

Si une image ne fait que 200px de large sur les 1024px du tenseur global (le reste étant du blanc), la fonction de coût ne doit pas essayer de trouver du texte dans la zone vide. Notre fonction calcule donc un vecteur `input_lengths` contenant la largeur "utile" de chaque image divisée par 4 (car le CNN réduit la largeur spatiale par un facteur 4 via le pooling). Cela permet au modèle de focaliser son attention uniquement sur la zone contenant de l'information manuscrite, ignorant le bruit potentiel du padding.

# 4 Difficultés rencontrées et solutions techniques

Durant l'apprentissage, nous avons fait face à plusieurs obstacles majeurs qui ont nécessité des ajustements architecturaux spécifiques.

## 4.1 Stagnation de l'apprentissage (CTC Collapse)

Lors des premières phases d'entraînement, nous avons constaté un blocage systématique : la fonction de perte (`Loss`) stagnait autour de 3.5 sans jamais descendre. Le modèle prédisait uniquement des chaînes vides ou des caractères répétés incohérents.

**Diagnostic :** Ce phénomène est appelé CTC Collapse. Le modèle tombe dans un minimum local où il prédit le caractère "Blank" (silence) en permanence, car c'est statistiquement l'option la moins risquée au début. Cela est souvent dû à des gradients instables dans le RNN qui empêchent le réseau de sortir de cet état.

**Solution :** Nous avons appliqué une technique de Gradient Clipping. Le principe est de fixer un seuil maximal (ici, une norme de 5.0). Si le vecteur gradient dépasse cette norme, il est redimensionné pour respecter le seuil. Cette régularisation a permis de stabiliser la rétro propagation et a débloqué la descente de gradient dès les premières époques.

### 4.2 Convergence et taux d'apprentissage

Avec un taux d'apprentissage (*LearningRate*) fixe, le modèle apprenait rapidement au début mais peinait à affiner ses résultats en fin de parcours. Nous avons résolu ce problème en implémentant un `Scheduler` (`ReduceLROnPlateau`). Cet algorithme réduit dynamiquement le taux d'apprentissage (division par 2) lorsque la perte de validation cesse de diminuer pendant plusieurs époques, permettant d'atteindre un minimum global plus précis.

### 4.3 Incohérence des données (bug RGB/Grayscale)

Une erreur technique bloquante (`RuntimeError`) est apparue lors de la constitution des batchs. L'analyse a révélé que le jeu de données RIMES n'était pas homogène : certaines images étaient encodées en couleurs (3 canaux RGB) et d'autres en niveaux de gris (1 canal). Le CNN attendant une entrée fixe, nous avons modifié la classe `OCRDataset` pour forcer la conversion en niveaux de gris (`.convert("L")`) dès le chargement du fichier, garantissant des tenseurs uniformes de taille $[1, 32, W]$.

## 5 Validation et résultats

### 5.1 Validation préliminaire

Avant de lancer l'entraînement complet sur l'intégralité du jeu de données, nous avons validé l'intégrité de notre pipeline (chargement des données, propagation avant et arrière) par une méthode de "sur-apprentissage intentionnel". Nous avons restreint le jeu de données à un micro-échantillon (8 images). Le modèle ayant réussi à atteindre une perte de 0.0 sur cet échantillon en quelques époques, nous avons eu la confirmation que l'architecture était capable d'apprendre et que les dimensions des tenseurs étaient correctes.

### 5.2 Analyse de l'entraînement (dynamique de la perte)

L'entraînement final a été lancé sur le jeu de données complet avec un suivi rigoureux des métriques d'apprentissage (erreur d'entraînement, $TrainLoss$) et de généralisation (erreur de validation, $ValidationLoss$).

Nous observons deux phases distinctes (voir Figure 2) :

1. **Phase d'apprentissage (Itérations 1 à 8) :** La perte diminue rapidement sur les deux jeux de données. Le modèle apprend à reconnaître les structures des caractères. À l'époque 8, nous atteignons notre meilleur point de généralisation avec une erreur de validation minimale de 0.4944.
2. **Phase de sur-apprentissage (Itérations 9 à 30) :** Alors que la perte d'entraînement continue de chuter jusqu'à devenir quasi-nulle ($\approx 0.003$), la perte de validation commence à remonter progressivement jusqu'à 0.80.

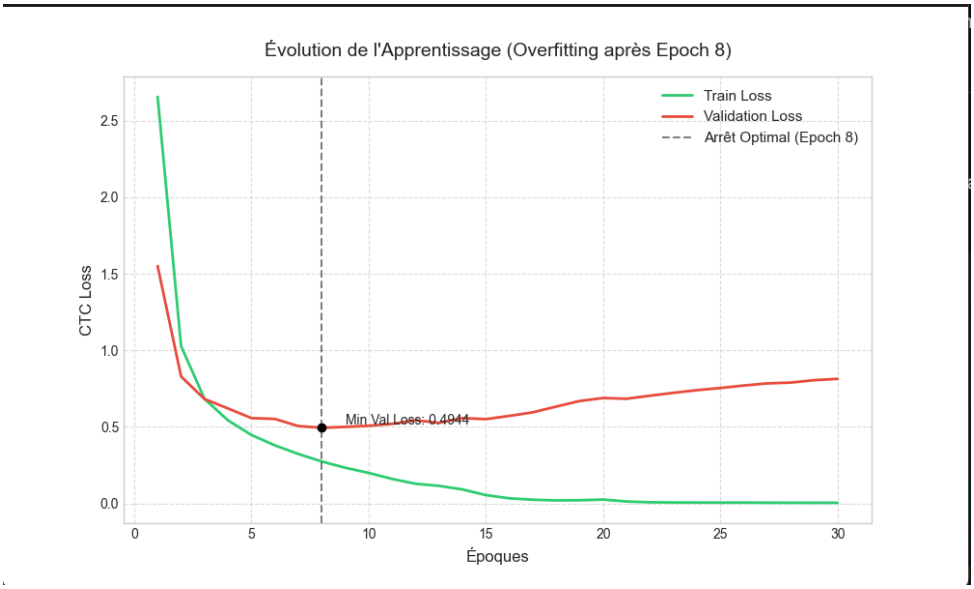


FIGURE 2 – Courbes d'apprentissage. On observe clairement le phénomène d'overfitting après la huitième itération : le modèle "apprend par cœur" les données d'entraînement mais perd sa capacité à généraliser.

Ce comportement est typique d'un modèle puissant qui finit par mémoriser le bruit du jeu d'entraînement. Grâce à notre mécanisme de sauvegarde conditionnelle (*ModelCheckpoint*), nous avons automatiquement conservé les poids du modèle à l'époque 8, garantissant les meilleures performances possibles.

## 5.3 Résultats qualitatifs (inférence)

Le modèle retenu a été intégré dans une interface graphique pour tester ses capacités en conditions réelles. Les résultats visuels confirment les métriques : le modèle parvient à transcrire avec succès des écritures cursives variées, gérant correctement les ligatures et l'espacement des caractères.

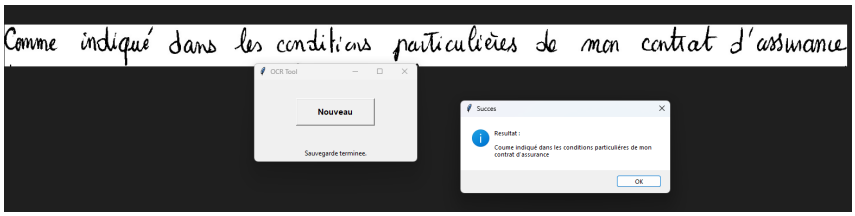


FIGURE 3 – Interface de démonstration. Le modèle CRNN prend l'image brute en entrée et restitue la séquence de caractères prédite.

Certaines erreurs persistent sur des graphies très ambiguës ou des ratures, mais la transcription globale reste lisible et cohérente.

## 5.4 Analyse des erreurs et cas limites

Bien que le modèle soit performant sur la majorité des données ($ValidationLoss \approx 0.49$), l'analyse qualitative des prédictions révèle des erreurs instructives sur la nature de l'architecture CRNN.

Prenons l'exemple d'une prédiction obtenue sur une lettre manuscrite :

> **Vérité :** *"habitation n° DPUET36, je souhaite vous faire part de mon récent déménagement"*
> **Prédiction :** *"habitation m° DFVET36, je souhaite vous faire pert de mon récent démchiagement"*

Cette erreur met en lumière deux types de limitations :

1. **Confusion topologique (m° / n°) :** Le modèle confond visuellement des lettres aux formes proches (le 'm' et le 'n', ou le 'a' et le 'e' dans "part/pert"). C'est une erreur purement visuelle : sans contexte, ces lettres sont très similaires dans l'écriture cursive.
2. **Absence de contexte sémantique (démchiagement) :** Le modèle a correctement détecté des boucles qui ressemblent graphiquement à "chi", mais il a produit un mot qui n'existe pas en français.

Cela démontre que notre modèle réalise une lecture purement "optique". Il transcrit ce qu'il voit, lettre par lettre, sans aucune compréhension linguistique ni vérification lexicale.

# 6 Conclusion et perspectives

Ce projet nous a permis d'implémenter avec succès une chaîne complète de reconnaissance d'écriture manuscrite. Nous avons surmonté les défis liés à l'hétérogénéité des données (normalisation, niveaux de gris) et à la stabilité de l'entraînement (Gradient Clipping). Le modèle final est fonctionnel et capable de généraliser correctement jusqu'à un certain point, avant d'entrer en sur-apprentissage (overfitting) dès la 9ème époque, preuve de la capacité du réseau à mémoriser les données.

Nous avons relevé plusieurs erreurs lors de la transcription, comme "démchiagement". Pour corriger ceux-ci, on peut améliorer notre modèle via deux axes :

— Le premier est d'enrichir notre jeu de données. Par exemple, pour réduire les confusions entre caractères proches (n/m), nous pourrions l'enrichir en appliquant des transformations aléatoires lors de l'entraînement : légères rotations, étirements élastiques ou ajout de bruit. Cela forcerait le CNN à devenir plus robuste aux variations graphiques subtiles.

— Changer l'algorithme de décodage. Actuellement, nous utilisons un décodage "Glouton" (Greedy Decoding) qui prend juste le caractère le plus probable à chaque instant. L'intégration d'un algorithme de Beam Search couplé à un dictionnaire français permettrait de pénaliser les séquences de caractères improbables. Ainsi, face à l'hésitation visuelle entre "démchiagement" et "déménagement", le système favoriserait "déménagement" car il existe dans le dictionnaire, corrigeant l'erreur optique par le sens.