
Rapport de Mini Projet

# Outils Mathématiques pour l'ingénieur

**Réalisateurs :**

ACHKHITY YASSINE
KHAIDER BRAHIM

**Sous la Supervision de :**

Mr. KAMOUSS
Mr. LAAZAIZ

June 1, 2024

# Contents


**1 Introduction** **2**

**2 Théorie des Automates Finis** **2**
2.1 Définition . . . 2
2.2 Fonction de Transition . . . 2
2.3 Automates Déterministes et Non-Déterministes . . . 3
2.3.1 Procédure d'acceptation . . . 4
2.3.2 Équivalence entre AFI et AFD . . . 4
2.4 la minimisation d'un automate . . . 6
2.4.1 Construction du tableau de distinction . . . 6
2.5 Application à la Reconnaissance d'Images . . . 7

**3 Phase initiale** **7**
3.1 Introduction . . . 7
3.2 Objectif . . . 7
3.3 Méthodologie . . . 7
3.4 Conception Orientée Objet . . . 8
3.5 Conclusion . . . 11

**4 Phase Appication** **12**
4.1 Introduction . . . 12
4.1.1 Présentation de la Classification d'image . . . 12
4.1.2 Aperçu des Réseaux de Neurones Artificiels . . . 12
4.2 Concepts de Base de Réseaux de Neurones . . . 12
4.2.1 Neurone Artificiel . . . 12
4.2.2 Fonction d'Activation . . . 13
4.3 Préparation des Données . . . 13
4.3.1 Collecte des Données: Introduction aux Datasets MNIST et EMNIST 13
4.3.2 Prétraitement des Données . . . 13
4.3.3 Normalisation et Redimensionnement des Images . . . 14
4.3.4 Division des Données en Ensembles d'Entrainement, de Validation et de Test . . . 14
4.4 Conception et Entraînement d'un Réseau de Neurones . . . 14
4.4.1 Configuration des Hyperparamètres . . . 14
4.4.2 Utilisation de Libraries (TensorFlow, Keras, PyTorch) . . . 14

**5 Implémentation du Code Python** **14**
5.1 Digit recognition: . . . 14
5.2 Character recognition: . . . 16

**6 Automate et la reconnaissance des manuscrits: Réalisation Pratiques** **19**
6.1 Paramètres de l'automate de reconnaissance des chiffres . . . 20
6.2 Exemple d'utilisation . . . 22
6.3 Reconnaissance des Chiffres Manuscrits . . . 23
6.4 Exemple d'Automate Cellulaire pour la Reconnaissance . . . 23

**7 Futures améliorations** **23**

**8 Conclusion** **24**

# 1 Introduction

Les automates, également connus sous le nom de machines à états finis, sont des modèles mathématiques utilisés pour représenter et analyser des systèmes à états discrets. Ils jouent un rôle crucial dans divers domaines, notamment l'informatique théorique, l'ingénierie, la linguistique et même la biologie. Ces systèmes sont caractérisés par un ensemble d'états, une fonction de transition qui décrit comment passer d'un état à un autre en réponse à des entrées spécifiques, un état initial à partir duquel le système commence son fonctionnement, et un ou plusieurs états finaux ou acceptants qui déterminent si une entrée donnée est acceptée par l'automate.

**Applications des Automates**
Les automates ont une multitude d'applications pratiques :

- **Analyse Lexicale :** Dans la compilation des langages de programmation, les DFA sont utilisés pour scanner le code source et identifier les tokens.
- **Reconnaissance de Modèles :** Les automates sont utilisés dans les logiciels de reconnaissance optique de caractères (OCR), dans les filtres anti-spam, et dans les systèmes de reconnaissance vocale.
- **Vérification de Modèles :** Les automates servent à vérifier les propriétés de systèmes complexes, comme les protocoles de communication ou les circuits électroniques, en s'assurant qu'ils respectent certaines spécifications.

Les automates, en tant que modèles mathématiques de systèmes à états finis, sont des outils puissants pour la modélisation, l'analyse et la vérification de systèmes discrets. Leur capacité à représenter des processus de décision complexes avec des états et des transitions les rend indispensables dans de nombreux domaines de l'informatique et de l'ingénierie. Comprendre les différents types d'automates et leurs applications permet de mieux appréhender les enjeux de la conception et de l'analyse des systèmes informatiques modernes.

# 2 Théorie des Automates Finis

## 2.1 Définition

Un automate fini est un quintuplet $(A, Q, I, T, E)$ où :

- $A$ : un alphabet utilisé pour la construction de mots.
- $Q$ : un ensemble fini d'états.
- $I$ : un sous-ensemble de $Q$ formant les états initiaux.
- $T$ : un sous-ensemble de $Q$ formant les états terminaux.
- $E$ : un ensemble de triplets appelés transitions, $E \subseteq Q \times A \times Q$.

## 2.2 Fonction de Transition

La fonction de transition $\delta$ est définie comme $\delta : Q \times A \to Q$, où $\delta(q, a)$ donne l'état suivant à partir de l'état $q$ avec l'entrée $a$.

## 2.3 Automates Déterministes et Non-Déterministes

L'exécution d'un automate fini indéterministe peut s'avérer très inefficace s'il comporte beaucoup de points de choix : si à chaque état l'automate a le choix entre deux transitions, alors pour analyser un mot de longueur $n$ il faudra envisager, dans le pire des cas, de l'ordre de $2^n$ transitions (si on a de la chance, et que l'on choisit toujours la "bonne" dérivation en premier, on pourra cependant trouver une dérivation en $n$ transitions). Pour éliminer ces points de choix, et rendre l'exécution efficace, il faut que l'automate soit déterministe, c'est-à-dire qu'il ait un seul état initial et que, étant donnés un état $S_i \in K$ et un symbole $a \in T$, il existe une seule transition possible.

**Définition (Automate Fini Déterministe = AFD)** : Un automate fini déterministe est défini par un quintuplet $(K, T, M, S_0, F)$ tel que

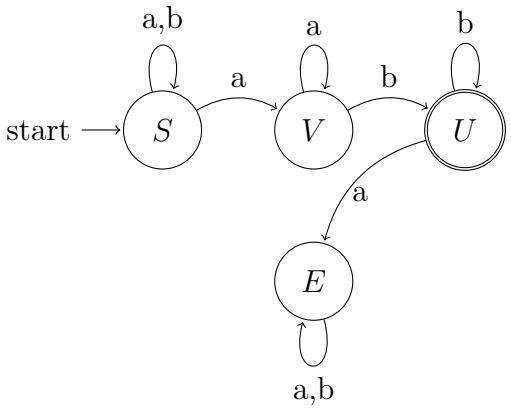
- $K$ est un ensemble fini d'états.
- $T$ est le vocabulaire terminal (correspondant à l'alphabet sur lequel est défini le langage).
- $M$ est une fonction de $K \times T$ dans $K$, appelée fonction de transition (M($S_i$, $a$) donne l'état unique dans lequel l'automate doit aller quand il se trouve dans l'état $S_i$ et que le mot à analyser commence par le symbole $a$).
- $S_0 \in K$ est l'état initial.
- $F \subseteq K$ est l'ensemble des états finaux.

**Exemple**

Par exemple, l'AFD $(K, T, M, S_0, F)$ tel que

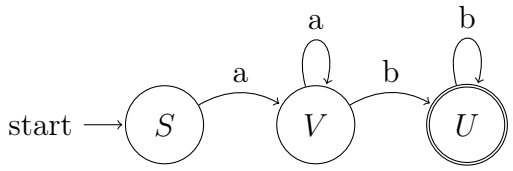
$$
\begin{aligned}
K &= \{S, V, U, E\} \\
T &= \{a, b\} \\
M &= \{(S, a) \to V, (S, b) \to E, (V, a) \to V, (V, b) \to U, \\
&\quad (U, a) \to E, (U, b) \to U, (E, a) \to E, (E, b) \to E\} \\
S_0 &= S \\
F &= \{U\}
\end{aligned}
$$

est représenté graphiquement par le graphe :



et accepte le langage $L = \{a^n b^p \mid n > 0, p > 0\}$.

L'état $E$ de cet automate correspond à un état d'erreur : l'automate va dans cet état dès lors qu'il reconnait que le mot ne fait pas partie du langage, et y reste jusque la fin de l'analyse. Dans un souci de simplification, on ne représente généralement pas cet état, et on représente l'automate par le graphe :




### 2.3.1 Procédure d'acceptation

Un AFD fonctionne comme un AFI, et on définit de la même manière les notions de configuration, dérivation entre configurations et acceptation d'un mot, la seule différence étant que la fonction de transition $M$ détermine de façon unique le nouvel état dans lequel l'automate doit se placer au moment de faire une dérivation. L'exécution d'un automate fini déterministe est résumée dans la procédure "accepte" suivante :

```
procédure accepte
entrée : un AFD (K, T, M, S0, F)
         un tableau de caractères u indicé de 1 à n
sortie : retourne vrai si u[1..n] appartient au langage, faux sinon
debut
    etatCrt ← S0
    i ← 1
    tant que i  n faire
        etatCrt ← M(etatCrt, u[i])
        i ← i + 1
    fin tant que
    si etatCrt  F alors retourne vrai sinon retourne faux
fin
```

### 2.3.2 Équivalence entre AFI et AFD

Pour déterminer si un mot $u$ de longueur $n$ est accepté, un AFD effectue exactement $n$ transitions, tandis qu'un AFI en effectue de l'ordre de $2^n$. L'exécution d'un AFD est donc nettement plus efficace que celle d'un AFI. En contrepartie, on peut se demander si les AFI sont plus généraux, c'est-à-dire s'ils acceptent plus de langages que les AFD. La réponse, négative, est donnée par le théorème suivant.

**Théorème (équivalence entre AFD et AFI)** : La famille des langages acceptés par un AFD est identique à la famille des langages acceptés par un AFI (autrement dit, s'il existe un AFI reconnaissant un langage donné, alors il existe un AFD reconnaissant le même langage).

La démonstration de ce théorème est réalisée en donnant un algorithme permettant de construire à partir d'un AFI un AFD reconnaissant le même langage. Chaque état de l'AFD correspond à un ensemble d'états de l'AFI.

**Procédure rendDéterministe**

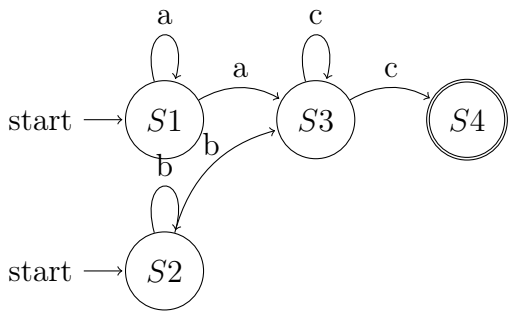
```
procédure rendDéterministe
entrée : un AFI A = (K, T, M, I, F)
```

```
sortie : un AFD A0 = (K0, T, M0, S00, F0) tel que L(A) = L(A0)
début
    S00 ← I
    K0 ← {I}
    vus ←
    tant que K0  vus faire
        soit U un état de K0 tel que U  vus
        pour tout l  T faire
            V ← {Sj / Si  U, (Si, l, Sj)  M}
            M0(U, l) ← V
            K0 ← K0  {V}
        fin pour
        vus ← vus  {U}
    fin tant que
    F0 ← {U  K0 / U  F  }
fin
```

Considérons par exemple l'AFI $(K, T, M, I, F)$ suivant :

$$
\begin{aligned}
K &= \{S1, S2, S3, S4\} \\
T &= \{a, b, c\} \\
M &= \{(S1, a, S1), (S1, a, S3), (S2, b, S2), (S2, b, S3), (S3, c, S3), (S3, c, S4)\} \\
I &= \{S1, S2\} \\
F &= \{S4\}
\end{aligned}
$$

correspondant au graphe suivant :




Pour plus de commodité, on représente la relation de transition $M$ par la table suivante :

| | $a$ | $b$ | $c$ |
|---|---|---|---|
| $S1$ | $\{S1, S3\}$ | $\emptyset$ | $\emptyset$ |
| $S2$ | $\emptyset$ | $\{S2, S3\}$ | $\emptyset$ |
| $S3$ | $\emptyset$ | $\emptyset$ | $\{S3, S4\}$ |
| $S4$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

On construit ensuite les états de l'AFD et leur fonction de transition. Au départ, l'AFD a un seul état qui est composé de l'ensemble des états initiaux de l'AFI : sur notre exemple, l'état initial de l'AFD est $\{S1, S2\}$. À chaque fois qu'on ajoute un nouvel état dans l'AFD, on détermine sa fonction de transition en faisant l'union des lignes correspondantes dans la table de transition de l'AFI : sur notre exemple, pour l'état

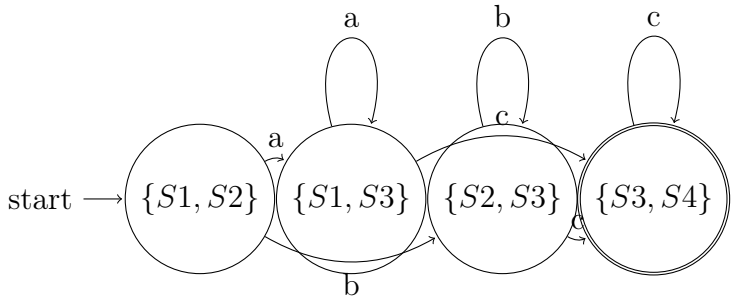
$\{S1, S2\}$, on fait l'union des lignes correspondant à $S1$ et $S2$, et on détermine la fonction de transition :

| $M$ | $a$ | $b$ | $c$ |
|---|---|---|---|
| $\{S1, S2\}$ | $\{S1, S3\}$ | $\{S2, S3\}$ | $\emptyset$ |

Autrement dit, quand on est dans l'état "$S1$ ou $S2$" et qu'on lit un $a$, on va dans l'état "$S1$ ou $S3$" ($M(\{S1, S2\}, a) = \{S1, S3\}$), quand on est dans l'état "$S1$ ou $S2$" et qu'on lit un $b$, on va dans l'état "$S2$ ou $S3$" ($M(\{S1, S2\}, b) = \{S2, S3\}$) et quand on est dans l'état "$S1$ ou $S2$" et qu'on lit un $c$, on va dans l'état "vide", correspondant à l'état d'erreur ($M(\{S1, S2\}, c) = \emptyset$). On rajoute ensuite les états $\{S1, S3\}$ et $\{S2, S3\}$ à l'AFD et on détermine leur fonction de transition selon le même principe. De proche en proche, on construit la table de transition suivante pour l'AFD :

| $M$ | $a$ | $b$ | $c$ |
|---|---|---|---|
| $\{S1, S2\}$ | $\{S1, S3\}$ | $\{S2, S3\}$ | $\emptyset$ |
| $\{S1, S3\}$ | $\{S1, S3\}$ | $\emptyset$ | $\{S3, S4\}$ |
| $\{S2, S3\}$ | $\emptyset$ | $\{S2, S3\}$ | $\{S3, S4\}$ |
| $\{S3, S4\}$ | $\emptyset$ | $\emptyset$ | $\{S3, S4\}$ |

L'ensemble des états de l'AFD est $K_0 = \{\{S1, S2\}, \{S1, S3\}, \{S2, S3\}, \{S3, S4\}\}$. Les états de l'AFD contenant un état final de l'AFI sont des états finaux. Ici, l'AFI a un seul état final $S4$ et l'ensemble des états finaux de l'AFD est $F_0 = \{\{S3, S4\}\}$. Cet AFD correspond au graphe suivant :




## 2.4 la minimisation d'un automate

La minimisation d'un automate déterministe fini (DFA) est le processus consistant à réduire le nombre d'états de l'automate tout en conservant le même langage reconnu. Cette section décrit le processus de minimisation à l'aide de la construction d'un tableau de distinction.

### 2.4.1 Construction du tableau de distinction

1. **Initialisation :** Créer une table où chaque paire d'états $(p, q)$ est marquée comme non distinguée.

2. **Marquage des paires distinguables :** Marquer les paires $(p, q)$ où $p$ est un état final et $q$ ne l'est pas (ou vice versa).

3. **Propagation des distinctions :** Pour chaque paire $(p, q)$ non marquée, si il existe un symbole de l'alphabet tel que la paire des états de destination $(p', q')$ soit déjà marquée, alors marquer $(p, q)$.

4. **Fusion :** Fusionner toutes les paires d'états non marquées pour former le DFA minimal.

## 2.5 Application à la Reconnaissance d'Images

Les automates déterministes sont utilisés dans divers domaines, y compris la reconnaissance d'images. Par exemple, dans la reconnaissance de formes, un DFA peut analyser des séquences de pixels pour identifier des motifs spécifiques dans une image. Chaque pixel ou groupe de pixels peut être considéré comme une entrée pour l'automate, permettant de détecter des formes ou des objets prédéfinis en suivant les transitions définies par l'automate.

# 3 Phase initiale

## 3.1 Introduction

Dans le cadre de notre formation en Intelligence Artificielle et Génie Informatique à l'ENSAM Casablanca, nous avons entrepris un mini-projet sur les automates. Ce projet vise à appliquer les concepts théoriques des automates à des problèmes pratiques, nous permettant ainsi de développer nos compétences en modélisation et en programmation.

## 3.2 Objectif

L'objectif de cette phase initiale est de créer une modélisation orientée objet d'un automate fini et d'implémenter les fonctionnalités de base nécessaires à sa manipulation. Cette base nous permettra de développer des applications plus complexes dans la phase suivante du projet.

## 3.3 Méthodologie

Pour réaliser cette phase, nous avons suivi les étapes suivantes :

1. **Définition des classes :**
   - **Etat** : Représente les états de l'automate.
   - **Alphabet** : Représente l'ensemble des symboles utilisés.
   - **Transition** : Représente les transitions entre les états.
   - **Automate** : Regroupe les états, l'alphabet et les transitions.
2. **Implémentation des classes :** Nous avons défini les attributs et méthodes de chaque classe en utilisant le langage Python. Les classes comprennent des méthodes pour ajouter, supprimer et modifier les états, l'alphabet et les transitions.
3. **Lecture de l'automate :** Nous avons développé une fonction capable de lire un automate à partir d'une liste structurée, instanciant ainsi un objet de la classe Automate.
4. **Fonctionnalités principales :**
   - Conversion d'un automate non déterministe en automate déterministe.
   - Complétion d'un automate.
   - Minimisation d'un automate.
   - Affichage graphique du graphe de transitions.

## 3.4 Conception Orientée Objet

### Classe Alphabet

#### Description

La classe `Alphabet` représente un ensemble de symboles utilisés dans un automate. Les symboles sont stockés dans un ensemble (`set`) pour assurer l'unicité et permettre des opérations efficaces d'ajout et de suppression.

#### Attributs

- `_symboles` : un ensemble (`set`) contenant les symboles de l'alphabet.

#### Méthodes

- `__init__(self, symboles)` : Constructeur qui initialise l'alphabet avec un ensemble de symboles donnés.
- `__repr__(self)` : Retourne une représentation en chaîne de caractères de l'alphabet.
- `ajouter_symbole(self, symbole)` : Ajoute un symbole à l'alphabet.
- `supprimer_symbole(self, symbole)` : Supprime un symbole de l'alphabet.
- `get_symboles(self)` : Retourne l'ensemble des symboles de l'alphabet.
- `set_symboles(self, symboles)` : Définit l'ensemble des symboles de l'alphabet.

### Classe Transition

#### Description

La classe `Transition` modélise une transition entre deux états d'un automate, activée par un symbole de l'alphabet.

#### Attributs

- `_etat_source` : L'état source de la transition.
- `_symbole` : Le symbole déclenchant la transition.
- `_etat_destination` : L'état destination de la transition.

#### Méthodes

- `__init__(self, etat_source, symbole, etat_destination)` : Constructeur qui initialise la transition avec un état source, un symbole, et un état destination.
- `__repr__(self)` : Retourne une représentation en chaîne de caractères de la transition.
- `get_etat_source(self)` : Retourne l'état source de la transition.
- `set_etat_source(self, etat_source)` : Définit l'état source de la transition.
- `get_symbole(self)` : Retourne le symbole de la transition.

- `set_symbole(self, symbole)` : Définit le symbole de la transition.
- `get_etat_destination(self)` : Retourne l'état destination de la transition.
- `set_etat_destination(self, etat_destination)` : Définit l'état destination de la transition.

## Classe Etat

### Description

La classe `Etat` représente un état dans un automate. Chaque état possède un nom, une liste de transitions, et des indicateurs pour déterminer s'il est initial ou final.

### Attributs

- `_nom` : Le nom de l'état.
- `_transitions` : Une liste des transitions associées à l'état.
- `_initial` : Un booléen indiquant si l'état est initial.
- `_final` : Un booléen indiquant si l'état est final.

### Méthodes

- `__init__(self, nom)` : Constructeur qui initialise l'état avec un nom.
- `__repr__(self)` : Retourne une représentation en chaîne de caractères de l'état.
- `ajouter_transition(self, transition)` : Ajoute une transition à l'état.
- `supprimer_transition(self, transition)` : Supprime une transition de l'état.
- `get_nom(self)` : Retourne le nom de l'état.
- `set_nom(self, nom)` : Définit le nom de l'état.
- `get_transitions(self)` : Retourne la liste des transitions de l'état.
- `set_transitions(self, transitions)` : Définit la liste des transitions de l'état.
- `is_initial(self)` : Retourne vrai si l'état est initial.
- `set_initial(self, initial)` : Définit si l'état est initial.
- `is_final(self)` : Retourne vrai si l'état est final.
- `set_final(self, final)` : Définit si l'état est final.

## Classe Automate

### Description

La classe `Automate` représente un automate fini. Elle contient les définitions et méthodes nécessaires pour manipuler les états, les transitions et l'alphabet de l'automate.

**Attributs**

- `_alphabet` : L'alphabet de l'automate.
- `_etats` : Une liste des états de l'automate.
- `_etats_initiaux` : Une liste des états initiaux de l'automate.
- `_etats_finaux` : Une liste des états finaux de l'automate.
- `_transitions` : Une liste des transitions de l'automate.

**Méthodes**

- `__init__(self, alphabet, etats, etats_initiaux, etats_finaux)` : Constructeur de la classe Automate.
- `ajouter_transition(self, transition)` : Ajoute une transition à l'automate.
- `supprimer_transition(self, transition)` : Supprime une transition de l'automate.
- `ajouter_etat(self, etat)` : Ajoute un état à l'automate.
- `supprimer_etat(self, etat)` : Supprime un état de l'automate.
- `ajouter_etat_initial(self, etat)` : Ajoute un état initial à l'automate.
- `supprimer_etat_initial(self, etat)` : Supprime un état initial de l'automate.
- `ajouter_etat_final(self, etat)` : Ajoute un état final à l'automate.
- `supprimer_etat_final(self, etat)` : Supprime un état final de l'automate.
- `lire_automate(self, entree)` : Initialise l'automate à partir d'une entrée donnée.
- `est_complet(self)` : Vérifie si l'automate est complet.
- `est_deterministe(self)` : Vérifie si l'automate est déterministe.
- `rendre_deterministe(self)` : Transforme l'automate en un automate déterministe.
- `rendre_complet(self)` : Transforme l'automate en un automate complet.
- `afficher_graphe(self)` : Affiche le graphe de l'automate.
- `get_alphabet(self)` : Retourne l'alphabet de l'automate.
- `set_alphabet(self, alphabet)` : Définit l'alphabet de l'automate.
- `get_etats(self)` : Retourne la liste des états de l'automate.
- `set_etats(self, etats)` : Définit la liste des états de l'automate.
- `get_etats_initiaux(self)` : Retourne la liste des états initiaux de l'automate.
- `set_etats_initiaux(self, etats_initiaux)` : Définit la liste des états initiaux de l'automate.

- `get_etats_finaux(self)` : Retourne la liste des états finaux de l'automate.
- `set_etats_finaux(self, etats_finaux)` : Définit la liste des états finaux de l'automate.
- `get_transitions(self)` : Retourne la liste des transitions de l'automate.
- `set_transitions(self, transitions)` : Définit la liste des transitions de l'automate.
- `automate_to_list(automate)` : Cette fonction convertit l'automate en listes exploitables contenant les informations sur l'alphabet, les états, les états initiaux, les états finaux et les transitions.
- `mot(etat, alphabet, transitions)` : Retourne l'état cible pour une transition donnée dans l'automate.
- `chercher_cle(dictionnaire, element_recherche)` : Recherche une clé dans un dictionnaire à partir d'une valeur spécifiée.
- `equivalent(etat1, etat2, classes, alphabets, transitions)` : Vérifie si deux états sont équivalents dans le contexte de l'algorithme de minimisation.
- `minimiser(automate)` : Applique l'algorithme de minimisation à l'automate donné, en réduisant le nombre d'états tout en préservant le comportement.

**Algorithme de Minimisation**

L'algorithme de minimisation fonctionne en plusieurs étapes :

1. Initialisation des classes d'équivalence en fonction des états finaux et non finaux de l'automate.
2. Itération sur les états pour les regrouper en classes d'équivalence en fonction de leur comportement similaire.
3. Création d'un nouvel automate à partir des classes d'équivalence identifiées.

## 3.5 Conclusion

Cette phase initiale nous a permis de poser les bases de notre automate. Nous avons défini les classes nécessaires et implémenté les principales fonctionnalités pour manipuler un automate fini. Ces fondations sont cruciales pour la phase suivante du projet, où nous développerons des applications spécifiques basées sur cette modélisation.

# 4 Phase Appication

## 4.1 Introduction

### 4.1.1 Présentation de la Classification d'image

**Définition de la classification d'images** La classification d'images est un domaine de l'intelligence artificielle et de la vision par ordinateur qui consiste à attribuer des étiquettes ou des catégories à des images en fonction de leur contenu visuel. Cette tâche implique l'analyse des caractéristiques visuelles des images pour les classer dans des catégories prédéfinies. Les algorithmes de classification d'images utilisent généralement des techniques d'apprentissage automatique, notamment l'apprentissage profond, pour identifier des motifs et des caractéristiques distinctives dans les données visuelles.

**Applications de la classification d'images dans divers domaines** La classification d'images trouve des applications dans de nombreux domaines. En médecine, elle est utilisée pour l'analyse des images médicales, comme la détection de tumeurs dans les radiographies ou les IRM. Dans l'industrie automobile, elle est cruciale pour le développement des véhicules autonomes, permettant la reconnaissance des panneaux de signalisation et des obstacles sur la route. En sécurité, elle aide à la reconnaissance faciale et à la surveillance. D'autres applications incluent la recherche d'images sur Internet, l'archivage et la gestion de collections d'images, ainsi que le commerce électronique pour la reconnaissance de produits.

**Importance de la reconnaissance de chiffres et de caractères** La reconnaissance de chiffres et de caractères est une application spécifique mais essentielle de la classification d'images. Cette technologie est utilisée dans la reconnaissance optique de caractères (OCR), qui permet la conversion d'images de texte manuscrit ou imprimé en texte numérique. Elle est fondamentale pour l'automatisation des processus administratifs, la numérisation de documents, la lecture automatique des plaques d'immatriculation et la gestion des chèques bancaires. La reconnaissance précise des chiffres et des caractères facilite également l'accessibilité, permettant aux personnes malvoyantes d'accéder à des contenus écrits via des dispositifs de lecture numérique.

### 4.1.2 Aperçu des Réseaux de Neurones Artificiels

**Pourquoi utiliser des réseaux de neurones pour la classification** Les réseaux de neurones sont largement utilisés pour la classification en raison de leur capacité à apprendre des modèles complexes à partir de données. Contrairement aux méthodes traditionnelles de classification qui reposent sur des règles prédéfinies, les réseaux de neurones peuvent découvrir des motifs et des relations non linéaires dans les données, ce qui les rend très efficaces pour la classification d'images, de texte, de son et d'autres types de données complexes. Mathématiquement, un réseau de neurones est une composition de fonctions non linéaires qui permet d'approximer des fonctions arbitraires, ce qui lui confère une grande capacité de modélisation.

## 4.2 Concepts de Base de Réseaux de Neurones

### 4.2.1 Neurone Artificiel

**Structure de fonctionnement** Un neurone artificiel est l'unité de base d'un réseau de neurones artificiels. Mathématiquement, un neurone artificiel effectue une opération

de somme pondérée suivie d'une fonction d'activation. Soit $x_1, x_2, ..., x_n$ les entrées du neurone, $w_1, w_2, ..., w_n$ les poids associés à chaque entrée, et $b$ le biais. La somme pondérée est calculée comme suit :

$$z = \sum_{i=1}^{n} (x_i \cdot w_i) + b$$

La sortie du neurone est alors obtenue en appliquant une fonction d'activation $\sigma(z)$ à cette somme pondérée.

### 4.2.2 Fonction d'Activation

**Différents types de fonctions d'activation** Les fonctions d'activation introduisent des non-linéarités dans les réseaux de neurones. Parmi les fonctions d'activation couramment utilisées, on trouve la fonction sigmoïde, définie comme $\sigma(z) = \frac{1}{1+e^{-z}}$, la fonction ReLU (Rectified Linear Unit) définie comme $\sigma(z) = \max(0, z)$, la fonction tangente hyperbolique (tanh) définie comme $\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, etc.

**Choix de la fonction d'activation en fonction de cas d'utilisation** Le choix de la fonction d'activation dépend souvent de la nature du problème de classification et des caractéristiques des données. Par exemple, la fonction sigmoïde est couramment utilisée pour la classification binaire car elle produit des sorties dans l'intervalle [0, 1], tandis que la fonction ReLU est populaire dans les réseaux de neurones profonds en raison de sa convergence rapide et de sa facilité de calcul.

## 4.3 Préparation des Données

### 4.3.1 Collecte des Données: Introduction aux Datasets MNIST et EMNIST

Les ensembles de données MNIST et EMNIST sont largement utilisés dans le domaine de la vision par ordinateur pour la classification de chiffres manuscrits. MNIST contient des images de chiffres écrits à la main de 0 à 9, tandis que EMNIST étend MNIST en incluant des lettres de l'alphabet anglais en plus des chiffres. Ces ensembles de données sont précieux pour l'entraînement et l'évaluation des modèles de classification d'images.

### 4.3.2 Prétraitement des Données

**Techniques de nettoyage des données** Le prétraitement des données est une étape cruciale dans le processus de préparation des données. Pour les ensembles de données d'images comme MNIST et EMNIST, les techniques de nettoyage des données peuvent inclure la suppression du bruit, la correction des distorsions, et la normalisation des intensités de pixel.

**Prétraitement spécifique aux images** Le prétraitement spécifique aux images comprend des opérations telles que la mise à l'échelle des intensités de pixel, la conversion en niveaux de gris, et l'augmentation des données par des opérations de rotation, de retournement, ou de zoom. Ces techniques permettent d'améliorer la robustesse du modèle et d'augmenter la diversité des données d'entraînement.

### 4.3.3 Normalisation et Redimensionnement des Images

La normalisation et le redimensionnement des images sont des étapes importantes pour assurer la cohérence des données avant l'entraînement du modèle. La normalisation consiste à mettre à l'échelle les valeurs des pixels pour qu'elles se situent dans un intervalle spécifique, généralement [0, 1] ou [-1, 1]. Le redimensionnement vise à uniformiser la taille des images, ce qui facilite le traitement par le modèle.

### 4.3.4 Division des Données en Ensembles d'Entrainement, de Validation et de Test

Pour évaluer la performance d'un modèle de classification, les données sont généralement divisées en trois ensembles : un ensemble d'entraînement utilisé pour ajuster les paramètres du modèle, un ensemble de validation utilisé pour sélectionner les hyperparamètres et éviter le surapprentissage, et un ensemble de test utilisé pour évaluer la performance finale du modèle sur des données non vues auparavant.

## 4.4 Conception et Entraînement d'un Réseau de Neurones

### 4.4.1 Configuration des Hyperparamètres

Les hyperparamètres d'un réseau de neurones sont des paramètres qui ne sont pas appris directement par le modèle lors de l'entraînement, mais qui affectent le processus d'apprentissage. Parmi les hyperparamètres les plus importants, on trouve le taux d'apprentissage ($\alpha$), qui contrôle la taille des pas effectués lors de la mise à jour des poids du réseau, le nombre d'epochs ($N$), qui détermine le nombre de fois que l'ensemble de données est parcouru lors de l'entraînement, et la taille du lot ($batch$), qui spécifie le nombre d'échantillons utilisés pour estimer le gradient à chaque étape de l'entraînement. Le choix des hyperparamètres peut avoir un impact significatif sur la performance du modèle et nécessite souvent une recherche approfondie pour être optimisé.

### 4.4.2 Utilisation de Libraries (TensorFlow, Keras, PyTorch)

Les bibliothèques telles que TensorFlow, Keras et PyTorch fournissent des outils puissants pour la conception et l'entraînement de réseaux de neurones. Elles facilitent la mise en œuvre des architectures de réseaux complexes, la gestion des données d'entraînement et de validation, ainsi que l'optimisation des hyperparamètres. Mathématiquement, ces bibliothèques utilisent des techniques d'optimisation telles que la descente de gradient stochastique (SGD), l'optimisation par momentum, ou l'optimisation adaptative comme Adam pour ajuster les poids du réseau afin de minimiser une fonction de perte définie, comme l'erreur quadratique moyenne (MSE) pour la régression ou l'entropie croisée pour la classification. Ces bibliothèques offrent également des fonctionnalités de régularisation, telles que l'abandon (dropout) ou la régularisation L1/L2, pour prévenir le surapprentissage et améliorer la généralisation du modèle.

# 5 Implémentation du Code Python

## 5.1 Digit recognition:

- Lors de notre recherche sur le projet on a implémenté un code Python qui permet la reconnaissance des chiffres de 0-9.

## Reconnaissance des chiffres manuscrits

Dans cette partie, nous examinerons un script Python qui utilise TensorFlow/Keras pour créer et entraîner un modèle de réseau neuronal pour la classification de chiffres manuscrits à partir de données MNIST. Le script comprend également la prédiction de chiffres manuscrits à partir d'images PNG.

## Code Python

```
import os
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

train_new_model = True

if train_new_model:
    # Chargement des donn es MNIST
    mnist = tf.keras.datasets.mnist
    (X_train, y_train), (X_test, y_test) = mnist.load_data()

    # Normalisation des donn es
    X_train = tf.keras.utils.normalize(X_train, axis=1)
    X_test = tf.keras.utils.normalize(X_test, axis=1)

    # D finition du mod le de r seau neuronal
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
    model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
    model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))

    # Compilation du mod le
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
     metrics=['accuracy'])

    # Entra nement du mod le
    model.fit(X_train, y_train, epochs=10)

    #   valuation   du mod le
    val_loss, val_acc = model.evaluate(X_test, y_test)
    print(val_loss)
    print(val_acc)

    # Sauvegarde du mod le
    model.save('handwritten_digits.model')
else:
    # Chargement d'un mod le pr -entra n
    model = tf.keras.models.load_model('handwritten_digits.model')

image_number = 1
while os.path.isfile('digits/digit{}.png'.format(image_number)):
    try:
        img = cv2.imread('digits/digit{}.png'.format(image_number))[:,:,0]
        img = np.invert(np.array([img]))
        prediction = model.predict(img)
        print("The number is probably a {}".format(np.argmax(prediction)))
        plt.imshow(img[0], cmap=plt.cm.binary)
```

```
            plt.show()
            image_number += 1
        except:
            print("Error reading image! Proceeding with next image...")
            image_number += 1
```

## Analyse

- Le script commence par l'importation des bibliothèques nécessaires : TensorFlow/Keras pour le modèle de réseau neuronal, NumPy pour la manipulation de tableaux, OpenCV pour la lecture d'images, et Matplotlib pour l'affichage.
- La variable `train_new_model` est utilisée pour déterminer si un nouveau modèle doit être entraîné ou si un modèle pré-entraîné doit être chargé.
- Si `train_new_model` est vrai, les données MNIST sont chargées et normalisées pour être prêtes à l'entraînement du modèle.
- Un modèle de réseau neuronal séquentiel est défini avec deux couches cachées de 128 unités chacune utilisant l'activation ReLU, et une couche de sortie de 10 unités utilisant l'activation softmax pour la classification des chiffres.
- Le modèle est compilé avec l'optimiseur Adam et la fonction de perte de catégorie croisée parcimonieuse, puis entraîné sur les données d'entraînement pendant 10 époques.
- La précision et la perte du modèle sont évaluées sur les données de test, et les résultats sont affichés.
- Le modèle entraîné est ensuite sauvegardé dans un fichier nommé `handwritten_digits.model`.
- Si `train_new_model` est faux, un modèle pré-entraîné est chargé à partir du fichier `handwritten_digits.model`.
- Le script boucle sur les fichiers d'images dans le dossier `digits` pour prédire les chiffres manuscrits. Chaque image est lue, inversée, et prédite par le modèle. Le résultat de la prédiction est affiché, et l'image est montrée avec Matplotlib.

## 5.2 Character recognition:

Cette partie permet la reconnaissance des lettres de l'alphabet de A-Z. Voici le code implémenté d'un modèle de classification des caractères manuscrits à l'aide de TensorFlow et Keras.

## Chargement des bibliothèques

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
    Dropout
from tensorflow.keras.utils import to_categorical
```

Chargement des bibliothèques nécessaires pour le traitement des données, la visualisation et la création du modèle de deep learning.
pandas est utilisé pour manipuler les données CSV.
tensorflow est la bibliothèque de deep learning utilisée pour construire et entraîner le modèle.
numpy est utilisé pour le calcul numérique.
matplotlib.pyplot est utilisé pour visualiser les images.
cv2 est une bibliothèque OpenCV utilisée pour le traitement d'images.
shuffle de sklearn.utils est utilisé pour mélanger les données.
Les différentes classes et fonctions de tensorflow.keras sont utilisées pour construire le modèle CNN.
train_test_split de sklearn.model_selection est utilisé pour diviser les données en ensembles d'entraînement et de test

## Chargement des données

```
data = pd.read_csv("A_Z Handwritten Data.csv")
data
```

Chargement des données de lettres manuscrites à partir d'un fichier CSV. (Kaggle Dataset)

## Préparation des données

```
X = data.drop('0', axis=1)
y = data['0']
```

Séparation des caractéristiques et des étiquettes.

## Visualisation de l'image et de son étiquette

```
img = X.T[0]
img = np.array(img)
img_reshaped = img.reshape((28, 28))
word_dict = {0:'A', 1:'B', 2:'C', 3:'D', 4:'E', 5:'F', 6:'G', 7:'H', 8:'I',
     9:'J', 10:'K', 11:'L', 12:'M', 13:'N', 14:'O', 15:'P', 16:'Q', 17:'R',
    18:'S', 19:'T', 20:'U', 21:'V', 22:'W', 23:'X', 24:'Y', 25:'Z'}
plt.imshow(img_reshaped)
plt.title(word_dict[y[0]])
```

Affichage d'une image manuscrite et de son étiquette correspondante.

## Séparation des données en ensembles d'entraînement et de test

```
train_size = int(len(X) * 0.7)
test_size = int(len(X) * 0.2)
validation_size = int(len(X) * 0.1)
train_x, test_x, train_y, test_y = train_test_split(X, y, test_size=0.2)
```

Division des données en ensembles d'entraînement et de test.

## Reshape des données

```
train_x = np.reshape(train_x.values, (train_x.shape[0], 28, 28))
test_x = np.reshape(test_x.values, (test_x.shape[0], 28, 28))

print("Train data shape: ", train_x.shape)
print("Test data shape: ", test_x.shape)
```

Reshape des données pour qu'elles puissent être utilisées par le modèle de deep learning.

## Encodage des étiquettes

```
train_yOHE = to_categorical(train_y, num_classes=26)
test_yOHE = to_categorical(test_y, num_classes=26)
print("New shape of train labels: ", train_yOHE.shape)
print("New shape of test labels: ", test_yOHE.shape)
```

Conversion des étiquettes en format one-hot encoding.

## Affichage des images après mélange

```
from sklearn.utils import shuffle

shuff = shuffle(train_x[:100])
fig, ax = plt.subplots(3, 3, figsize=(10, 10))
axes = ax.flatten()

for i in range(9):
    shu = np.where(shuff[i] > 30, 200, 0)
    axes[i].imshow(shu, cmap="Greys")
    axes[i].axis('off')

plt.tight_layout()
plt.show()
```

Affichage des images après mélange pour vérifier la diversité des données.

## Affichage du nombre d'éléments par lettre

```
count = np.zeros(26, dtype='int')
for i in np.int0(y):
    count[i] += 1

alphabets = [word_dict[i] for i in range(26)]
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.barh(alphabets, count)
plt.xlabel("Number of elements")
plt.ylabel("Alphabets")
plt.grid()
plt.show()
```

Affichage du nombre d'exemples pour chaque lettre de l'alphabet.

## Prétraitement de l'image pour la prédiction

```
img_copy = cv2.GaussianBlur(img_copy, (7, 7), 0)
img_gray = cv2.cvtColor(img_copy, cv2.COLOR_BGR2GRAY)
_, img_thresh = cv2.threshold(img_gray, 100, 255, cv2.THRESH_BINARY_INV)
img_final = cv2.resize(img_thresh, (28, 28))
img_final = np.reshape(img_final, (1, 28, 28, 1))
```

Prétraitement de l'image pour qu'elle soit compatible avec le modèle de prédiction.

### Prédiction des étiquettes

```
predictions = model.predict(img_final)
top_indices = np.argsort(predictions[0])[-5:][::-1]
top_values = predictions[0][top_indices]

for i, index in enumerate(top_indices):
    print(f"Top {i+1} Prediction: Index = {index}, with probability value =
    {top_values[i]} of the alphabet '{word_dict[index]}'")
```

Prédiction des étiquettes et affichage des 5 meilleures prédictions avec leurs probabilités

### Conclusion

Le code implémente un modèle CNN pour la classification des caractères manuscrits de l'alphabet anglais, en utilisant TensorFlow et Keras. Chaque étape, de l'importation des bibliothèques au prétraitement des données, à la construction et à l'entraînement du modèle, est expliquée en détail.

## 6 Automate et la reconnaissance des manuscrits: Réalisation Pratiques

Dans cette partie, nous avons essayé d'implementer des fonctions essentielles pour le modèle de reconnaissance des chiffres manuscrits. Ces fonctions permettent de charger, traiter et prédire les chiffres à partir d'images.

### Fonctions ajoutées pour l'automate:

#### charger_image(self, filepath):

Cette fonction charge une image à partir du chemin spécifié. Utilise la bibliothèque OpenCV (cv2) pour lire l'image. Retourne l'image chargée.

```
def charger_image(self, filepath):
    img = cv.imread(filepath)
    return img
```

Listing 1: Code Python pour charger une image

#### conversion_grayscale(self, img):

Convertit une image en niveaux de gris. Utilise la fonction `cv.cvtColor` de OpenCV pour la conversion. Retourne l'image convertie.

```python
def conversion_grayscale(self, img):
    return cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

Listing 2: Code Python pour convertir une image en niveaux de gris

**inversion_couleurs(self, img):**

Inverse les couleurs de l'image. Utilise la fonction `cv.bitwise_not` de OpenCV. Retourne l'image avec les couleurs inversées.

```python
def inversion_couleurs(self, img):
    return cv.bitwise_not(img)
```

Listing 3: Code Python pour inverser les couleurs d'une image

**redimensionnement(self, img, size=(28, 28)):**

Redimensionne l'image à la taille spécifiée, par défaut (28, 28). Utilise la fonction `cv.resize` de OpenCV avec une interpolation. Retourne l'image redimensionnée.

```python
def redimensionnement(self, img, size=(28, 28)):
    return cv.resize(img, size, interpolation=cv.INTER_AREA)
```

Listing 4: Code Python pour redimensionner une image

**normalisation(self, img):**

Normalise l'image en divisant chaque pixel par 255.0. Prépare l'image pour la prédiction en la mettant dans la même échelle que celle utilisée pendant l'entraînement du modèle. Retourne l'image normalisée.

```python
def normalisation(self, img):
    return img / 255.0
```

Listing 5: Code Python pour normaliser une image

**prediction(self, img, model):**

Prépare l'image pour la prédiction en ajoutant les dimensions nécessaires. Utilise le modèle pré-entraîné pour prédire le chiffre manuscrit. Retourne l'indice de la classe prédite.

```python
def prediction(self, img, model):
    img = np.expand_dims(img, axis=0)
    img = np.expand_dims(img, axis=-1)
    prediction = model.predict(img)
    return np.argmax(prediction)
```

Listing 6: Code Python pour prédire le chiffre manuscrit

## 6.1 Paramètres de l'automate de reconnaissance des chiffres

L'alphabet et les états sont utilisés pour définir les transitions dans l'automate.

**Alphabet:**

L'alphabet est une liste d'actions que l'automate peut effectuer. Ici, il est défini comme une liste d'étapes du processus de traitement d'image:

- 'load'
- 'grayscale'
- 'invert'
- 'resize'
- 'normalize'
- 'predict'

```
alphabet = Alphabet(['load', 'grayscale', 'invert', 'resize', 'normalize',
    'predict'])
```

Listing 7: Définition de l'alphabet en Python

**États:**

Nous avons défini les états sous la forme des différentes étapes de traitement de l'image. Chaque état correspond à une étape dans le pipeline de traitement d'image.

```
etats = [
    Etat('initial'),      # tat initial avant toute action
    Etat('loaded'),       # Apr s le chargement de l'image
    Etat('grayscaled'),   # Apr s conversion en niveaux de gris
    Etat('inverted'),     # Apr s inversion des couleurs
    Etat('resized'),      # Apr s redimensionnement
    Etat('normalized'),   # Apr s normalisation
    Etat('predicted')     # Apr s pr diction
]
```

Listing 8: Définition des états en Python

**Transitions**

Les transitions définissent le passage d'un état à un autre en fonction de l'action effectuée. Définition des transitions: Chaque transition prend un état de départ, une action et un état d'arrivée. Ces transitions sont ajoutées à l'automate pour définir son comportement.

```
transitions = [
    Transition(etats[0], 'load', etats[1]),
    Transition(etats[1], 'grayscale', etats[2]),
    Transition(etats[2], 'invert', etats[3]),
    Transition(etats[3], 'resize', etats[4]),
    Transition(etats[4], 'normalize', etats[5]),
    Transition(etats[5], 'predict', etats[6]),
]

for transition in transitions:
    automate.ajouter_transition(transition)
```

Listing 9: Définition des transitions en Python

## 6.2 Exemple d'utilisation

L'exemple montre comment utiliser l'automate pour reconnaître des chiffres manuscrits.

**Étapes:**

1. Charger le modèle pré-entraîné de reconnaissance des chiffres manuscrits.
2. Créer l'automate avec les alphabets et états définis.
3. Définir les transitions pour chaque étape de traitement de l'image.
4. Simuler le processus de reconnaissance pour chaque image donnée:
   (a) Charger l'image.
   (b) Suivre les étapes de traitement en respectant les transitions.
   (c) Afficher l'image et la prédiction.

```
# Exemple d'utilisation pour la reconnaissance de chiffre manuscrit

# Charger le mod le de reconnaissance des chiffres manuscrits
model = tf.keras.models.load_model('handwritten_digits.model')

# Cr er l'automate avec les  tats  et transitions appropri s
alphabet = Alphabet(['load', 'grayscale', 'invert', 'resize', 'normalize',
    'predict'])
etats = [Etat('initial'), Etat('loaded'), Etat('grayscaled'), Etat('
    inverted'), Etat('resized'), Etat('normalized'), Etat('predicted')]
automate = Automate(alphabet, etats, [etats[0]], [etats[-1]])

# D finir les transitions pour le processus de reconnaissance de chiffre
transitions = [
    Transition(etats[0], 'load', etats[1]),
    Transition(etats[1], 'grayscale', etats[2]),
    Transition(etats[2], 'invert', etats[3]),
    Transition(etats[3], 'resize', etats[4]),
    Transition(etats[4], 'normalize', etats[5]),
    Transition(etats[5], 'predict', etats[6]),
]

for transition in transitions:
    automate.ajouter_transition(transition)

# Simuler le processus de reconnaissance pour une image donn e
for i in range(1, 12):
    image_path = f'C:/Users/yassine/OneDrive/Bureau/Automate Projet/phase
    application/digits/digit{i}.png'

    # Suivre les transitions de l'automate
    img = automate.charger_image(image_path)
    img = automate.conversion_grayscale(img)
    img = automate.inversion_couleurs(img)
    img = automate.redimensionnement(img)
    img = automate.normalisation(img)
    pred = automate.prediction(img, model)

    # Affichage de l'image et de la pr diction
    plt.imshow(img, cmap=plt.cm.binary)
```

```
38      plt.title(f"The number is probably a {pred}")
39      plt.show()
```

Listing 10: Exemple d'utilisation pour la reconnaissance de chiffre manuscrit

## 6.3 Reconnaissance des Chiffres Manuscrits

### Utilisation des Automates Cellulaires

Les automates cellulaires peuvent être utilisés pour traiter et analyser des images de chiffres manuscrits en utilisant les étapes suivantes :

### Étapes de la Modélisation

1. **Prétraitement :** Conversion des images en niveaux de gris et binarisation pour simplifier les données d'entrée.
2. **Segmentation :** Utilisation d'automates cellulaires pour identifier et isoler les chiffres dans l'image. Les cellules de l'automate représentent des pixels de l'image, et les règles de transition peuvent détecter les contours des chiffres.
3. **Extraction de Caractéristiques :** Application de règles d'automates cellulaires pour extraire des caractéristiques pertinentes telles que la forme et les intersections des traits.
4. **Classification :** Utilisation des états finaux de l'automate pour classifier les chiffres. Des réseaux de neurones peuvent être combinés avec les automates cellulaires pour améliorer la précision de la reconnaissance.

## 6.4 Exemple d'Automate Cellulaire pour la Reconnaissance

Considérons un exemple simple d'automate cellulaire à deux dimensions pour détecter les chiffres manuscrits. Soit une grille $G$ où chaque cellule peut être dans l'un des états suivants : 0 (blanc) ou 1 (noir). Le voisinage $N$ est constitué des huit cellules environnantes (voisinage de Moore).

La fonction de transition $\delta$ peut être définie de manière à détecter les contours des chiffres :

$$\delta(c_{i,j}) = \begin{cases} 1 & \text{si } \sum_{(k,l)\in N} c_{k,l} \geq 1 \text{ et } \sum_{(k,l)\in N} c_{k,l} < 4 \\ 0 & \text{sinon} \end{cases}$$

où $c_{i,j}$ représente l'état de la cellule $(i, j)$.

# 7 Futures améliorations

Notre passion pour l'informatique et les mathématiques nous a conduit à envisager l'intégration prochaine d'un code utilisant des automates cellulaires pour la reconnaissance des chiffres manuscrits. Par la suite, nous souhaitons étendre cette approche à l'identification des lettres en arabe ou dans d'autres langues. De plus, nous envisageons de développer une interface graphique interactive permettant de fournir directement les prédictions des caractères.

# 8 Conclusion

Ce projet a représenté une opportunité unique pour nous en tant qu'élèves ingénieurs, non seulement pour mettre en pratique nos connaissances en automates et en mathématiques, mais aussi pour explorer plusieurs bibliothèques en Python telles que OpenCV et TensorFlow, ainsi que des techniques de Machine Learning et de réseaux neuronaux. En guise de retour d'expérience, la mise en œuvre d'un automate de reconnaissance de caractères manuscrits s'est avérée être une tâche ardue, notamment au niveau de la conception. Nous avons réalisé que la seule solution était d'intégrer les étapes de prétraitement de l'image en tant qu'états de l'automate. Initialement, nous avions envisagé de concevoir un automate capable d'extraire les caractéristiques de l'image en entrée, telles que la détection de boucles ou de traits verticaux. Cependant, au cours de nos recherches, nous avons découvert les automates cellulaires, et nous prévoyons maintenant d'approfondir nos investigations afin d'intégrer les automates cellulaires à notre système de reconnaissance de manuscrits.

# References


[1]

[2] Stephen Wolfram, *A New Kind of Science*, Wolfram Media, 2002.

[3] R. Plamondon, S. N. Srihari, *Online and off-line handwriting recognition: a comprehensive survey*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2000.

[4] P. Zhang, T. D. Bui, C. Y. Suen, *A novel cascade ensemble classifier system with a high recognition performance on handwritten digits*, Pattern Recognition, 2007, Vol. 40, Issue 12, pp. 3415-3429. doi: 10.1016/j.patcog.2007.03.022.

[5] NeuralNine, *Handwritten digits recognition*, `https://github.com/NeuralNine/handwritten-digits-recognition/`.

[6] Kaggle website, `https://www.kaggle.com/`.

[7] Michael Sipser, *Introduction to the Theory of Computation*, Third Edition, Cengage Learning, 2012.

[8] Christine Solnon, *Théorie des langages.*

[9] Al Sweigart, *Automate the Boring Stuff with Python*, Second Edition, No Starch Press, 2019.

[10] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Second Edition, Addison-Wesley, 2001.

[11] GeeksforGeeks website, `https://www.geeksforgeeks.org/`.

[12] OpenAI, *ChatGPT*, `https://www.openai.com/chatgpt`.