INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE
Département de Génie Electrique & Informatique

Université de Toulouse

LAAS CNRS

Projet intégrateur - 5 SDBD

Janvier 25, 2023

" AI-based Hashiwokakero puzzle solver | SOA "

Étudiants :

Salma	Aboumzrag	aboumzrag@insa-toulouse.fr
Jose	Daniel Calderon	calderon@insa-toulouse.fr
Andréa	Chenel	chenel@insa-toulouse.fr
Théo	Fontana	fontana@insa-toulouse.fr
Pierre	Laur	plaur@insa-toulouse.fr
Ludovic	Mocquais	mocquais@insa-toulouse.fr
Walid	Ouahi	ouahi@insa-toulouse.fr
Coumba	Soumaré	soumare@insa-toulouse.fr

Tuteurs :

Marie-José	Huguet	huguet@laas.fr
Mohamed	Siala	msiala@insa-toulouse.fr
Sami	Yangui	Yangui@insa-toulouse.fr
Nawal	Guermouche	Nawal.Guermouche@laas.fr

Mots clés : Machine Learning, Hashiwokakero, SOA

Table des matières

Remerciements

En préambule à ce rapport, nous tenons à remercier tous ceux qui ont contribué de près ou de loin à la réalisation de ce projet.

Nous souhaitons adresser nos remerciements les plus sincères aux personnes qui nous ont apporté leur aide et qui ont contribué à l'élaboration de ce projet ainsi qu'à la réussite de cette formidable année universitaire.

Ces remerciements vont tout d'abord à nos professeurs pour leur soutien et leur encadrement ainsi que pour leur disponibilité tout en long de la réalisation de ce projet, et pour leur inspiration, encouragements, aide et leur suivi.

Nos remerciements iront également vers tous ceux qui ont accepté avec bienveillance de participer au jury de ce mémoire.

Merci à tous et à toutes.

Introduction

Hashiwokakero, ou *Hashi* est un casse-tête qui se joue sur une grille rectangulaire. Le but premier de ce jeu est de connecter les cercles entre eux, aussi appelés îles, à l'aide de traits, aussi appelés ponts.

À l'intérieur de chaque cercle, se trouve un chiffre allant de 1 à 8 qui indique le nombre de ponts qui devront être reliés à l'île.

A la fin de la résolution, chaque île devra être reliée à toutes les autres îles en utilisant les différents ponts créés.

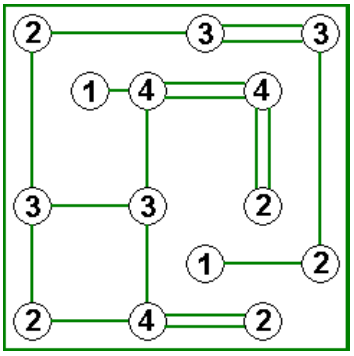

Figure 1 : Exemple d'une grille de Hashiwokakero

Le joueur se doit de respecter quelques règles assez simples. Les principales règles de ce jeu sont les suivantes :

- Les ponts doivent être représentés par une ligne droite ;
- Les ponts sont uniquement verticaux ou horizontaux ;
- Deux ponts ne peuvent pas se croiser ;
- Les ponts doivent commencer et finir sur deux îles différentes ;
- Les îles doivent être reliées par un pont simple ou double ;
- Le nombre de ponts connectés à une île doit être égal au chiffre indiqué sur l'île ;
- Toutes les îles doivent être reliées entre elles au sein d'un même groupe.

Nous avons créé un solveur pour ces grilles. L'utilisateur renseigne une image d'une grille de Hashi faite à la main et le solveur retourne la grille complétée. La Figure 1 est un exemple d'une grille de Hashi correctement complétée.

Overview

Pour identifier les différents éléments de la grille et résoudre le problème, notre application sépare le travail en plusieurs tâches. La Figure 2 détaille l'architecture de notre système.

En entrée, notre application prend une image d'une grille de Hashi faite à la main. Dans un premier temps, il nous a fallu développer la reconnaissance des îles afin d'obtenir leurs positions sur la grille. Une fois les îles extraites de la grille, il nous a fallu identifier les chiffres présents à l'intérieur des cercles afin de connaître le nombre de ponts à générer. Enfin, une fois ces informations extraites de la grille, il nous faut résoudre le problème à l'aide d'un solveur avant de retourner la grille résolue.

Nous ne manquerons pas de présenter l'architecture orientée service que nous avons choisie pour orchestrer notre application.

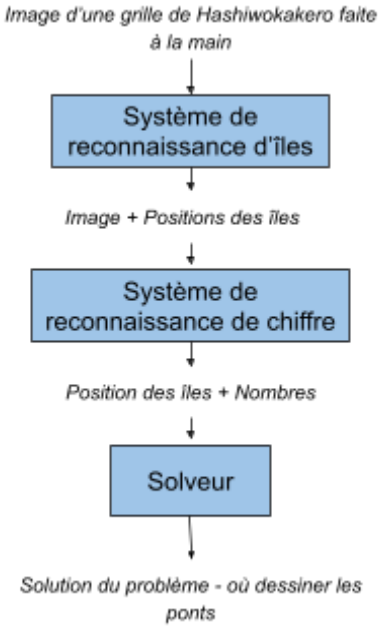

Figure 2 : Architecture du solveur

I - Génération du dataset

Nous commençons par générer un jeu de données utilisable d'énigmes de Hashi dessinées à la main qui nous permettra de tester les multiples composants de notre application, et d'avoir une source d'apprentissage pour notre système de reconnaissance des chiffres, cf. partie Reconnaissance des chiffres.

La génération de l'ensemble de données nécessite la création de plusieurs parties :

- Base de données d'énigmes de Hashi basées sur du texte (**Figure 3**), sous forme de matrices avec des chiffres pour les îles et des zéros pour les espaces vides.
- Ensemble de données de cercles dessinés à la main (**Figure 4**), réalisés à partir de cercles dessinés sur papier que nous transformons ensuite en images séparées.
- Une base de données de chiffres dessinés à la main (**Figure 5**) à utiliser à l'intérieur de nos cercles, ici nous utilisons la base de données MIST qui est une base de données publique.
- Ensemble de données de quelques fonds qui sont simplement des images de feuilles de papier(**Figure 6**).

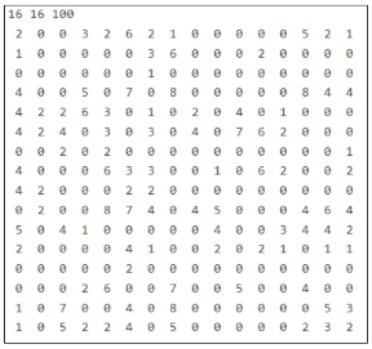

Figure 3 : Exemple de données textuelles utilisées pour la génération du Dataset

Figure 4 : Exemple de cercles utilisés pour la génération du Dataset

Figure 5 : Exemple de données issues du Dataset MINIST

Figure 6 : Exemple de Background utilisés pour la génération du Dataset

Avec ces différentes pièces, nous pouvons assembler l'image d'un puzzle de Hashi en quelques étapes :

1. Nous prenons un texte aléatoire basé sur le puzzle Hashiwokakero, et le parcourons ligne par ligne.
2. Pour chaque numéro de chaque ligne, nous choisissons un numéro aléatoire dessiné à la main dans notre base de données.
3. Pour chacun des chiffres choisis, nous choisissons au hasard un cercle de notre ensemble de données.
4. Nous fusionnons les nombres et les cercles ensemble et créons une image nombre-cercle de 80x80.
5. Nous concaténons les images de nombres et de cercles de la ligne, tout en ajoutant du remplissage entre eux pour créer les espaces vides. Cela forme une image de ligne.
6. Après avoir répété les étapes précédentes pour toutes les lignes, nous les concaténons, et ajoutons du remplissage, pour former l'image finale.

Le résultat de ces étapes est un puzzle Hashi dessiné à la main, utilisable par les différentes parties de notre application. Nous pouvons ensuite répéter ce processus pour toutes les énigmes Hashi basées sur du texte, ce qui crée notre ensemble de données d'énigmes Hashi "dessinées à la main" utilisables.

II - Reconnaissance d’îles

La première étape de la résolution d’un puzzle par un ordinateur est la reconnaissance du contexte de ce dernier. En effet, ici, notre puzzle ne peut être résolu automatiquement sans connaître certains détails clés comme la position des espaces vides, celle des îles ainsi que les chiffres que les îles contiennent. Ici, nous allons nous intéresser aux positions des espaces vides et des îles. Pour ce faire, notre programme doit être capable de reconnaître sur une image l'existence de cercles et par la même occasion, il doit pouvoir définir leur position en accordance avec leur position dans le puzzle original. C’est le rôle du service Island que de répondre à ce besoin.

a) Format des entrées et sorties :

Le format d’entrée de notre service est une image dessinée à la main d’un puzzle Hashiwokakero que nous recevons depuis le service Orchestrateur. Cette image est ensuite traitée par notre service comme nous le verrons ultérieurement.

Il renvoie une image du puzzle sous forme de matrice contenant les coordonnées des cercles à leur position respective, ainsi que la position des espaces vides.

b) Nettoyage :

Pour pouvoir analyser l’image du puzzle, nous devons tout d’abord effacer toutes les imperfections de cette dernière. Pour ce faire nous procéderons par étapes.

Premièrement, nous transformons l'image afin qu’elle contienne uniquement des nuances de gris. Dans un second temps nous faisons une copie de l’image, que nous dilatons avant de la flouter légèrement. Celà permet de créer une image ou les lignes disparaissent, ne gardant ainsi que l'arrière-plan.

Nous soustrayons ensuite l'image originale en niveaux de gris à la nouvelle image pour supprimer l'arrière-plan. Sur l’image résultante, nous appliquons une limite qui dit qu'au dessus d’une certaine nuance de gris, les couleurs deviennent noir absolu, sinon, elles deviennent blanche absolu. Nous obtenons ainsi une image binaire.

c) Création de la grille :

Pour pouvoir créer la grille représentant notre puzzle, nous allons d’abord devoir trouver les cercles présents sur notre image binaire. Pour celà nous allons utiliser la fonction Hough Circle de la librairie OpenCV de python.

La transformée de Hough est une méthode mathématique permettant de détecter des formes diverses dans une image. La méthode de détection de cercle provenant de cette dernière est également baptisée HCT (Hough circle transform). C’est cette fonction qui correspond à celle que nous allons utiliser.

La fonction Hough Circle requiert plusieurs paramètres. Notamment, les rayons minimum et maximum des cercles que nous essayons de détecter, ainsi que la distance minimale entre deux de ces cercles. Un autre paramètre important est le seuil d’accumulation qui peut être défini comme la fiabilité des cercles détectés.

Dans notre cas nous avons défini les paramètres tels que :

- Rayon minimum = 8 px
- Rayon maximum = 19 px
- Distance minimum = 55 px
- Seuil d’accumulation = 18

Cette fonction renvoie sous forme de liste les couples contenant les coordonnées de centre de cercles et leur rayon qu'elle a trouvés sur l'image binaire. Ces cercles correspondent très strictement aux paramètres précédemment désignés.

Notre service utilise ensuite cette liste pour calculer la matrice finale. Pour ce faire, il récupère d'abord dans la liste le rayon le plus grand. Il range alors la liste de la plus petite à la grande coordonnée y. Il crée ensuite la première itération de la délimitation entre deux lignes de cercles représentée par une coordonnée nommée *yBound*. Cette coordonnée est calculée par l'équation suivante :

$$yBound = y + 2 * maxRadius$$

Ici, y représente la coordonnée du cercle examiné, et *maxRadius* représente la constante récupérée plus tôt du rayon maximum de notre liste. On entre ensuite dans une boucle qui examine chaque cercle de notre liste de cercle, en vérifiant pour chacun si la coordonnée y de leur centre est au-dessus ou en dessous de cette limite. Si elle est au dessus, on note que le cercle est sur la même ligne que le précédent; sinon, on crée une nouvelle ligne. La délimitation *yBound* est ensuite recalculée pour chaque cercle, en utilisant la même formule.

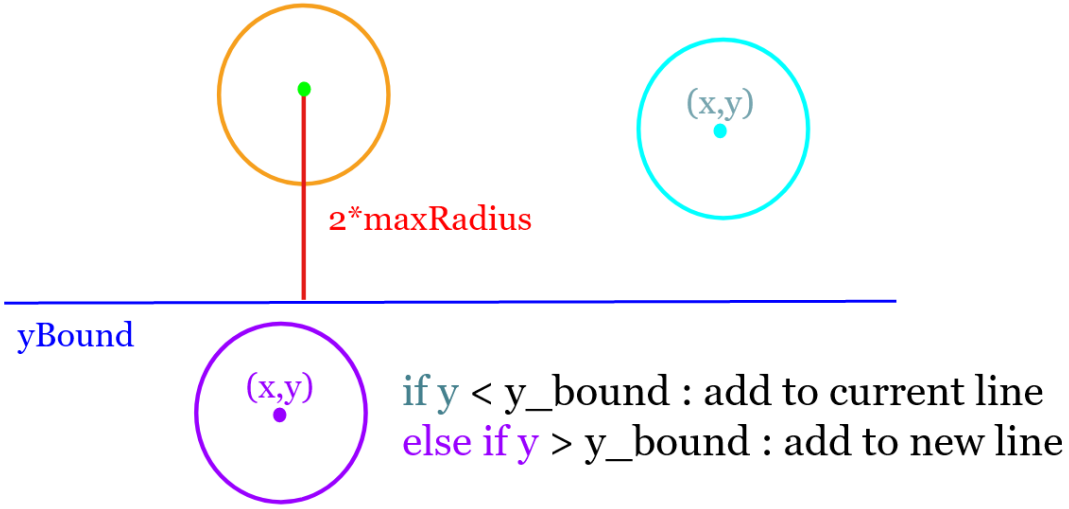

Figure 7 : Délimitation

Notre service reproduit ensuite ces mêmes étapes en utilisant la coordonnée x pour noter les colonnes des cercles.

Le service ayant maintenant pour chaque cercle le couple (ligne, colonne) donnant sa position dans le puzzle. Il crée une matrice vide de dimension le nombre de lignes fois le nombre de colonnes et il y place les couples coordonnées-rayon des cercles en fonctions de positionnement ligne-colonne. Il remplit ensuite les espaces vides par des couples coordonnées-rayon nuls.

La matrice résultante de ces opérations est la matrice finale attendue. Elle est transformée un string JSON et renvoyée au service Orchestrateur.

III - Reconnaissance des chiffres

Une fois les îles détectées, il faut procéder à l'identification des chiffres à l'intérieur des cercles. En récupérant en entrée l'image ainsi que la matrice de coordonnées identifiant les positions des chiffres, l'objectif est de générer une matrice associant les chiffres à leur valeur. Le Machine learning et le deep learning jouent alors un rôle important puisqu'il présente différents modèles d'identification et de reconnaissance de chiffres.

a) Objectifs principaux et méthodes utilisées

Le but de l'algorithme de reconnaissance de chiffres est d'obtenir une matrice comparative avec l'ensemble des positions et la probabilité de détection de chaque chiffre (de 1 à 8) sur ces positions. La première étape fut d'identifier le modèle dont les paramètres de performance, de précision et de temps d'exécution sont les plus adaptés à la résolution de notre problème.

Ainsi, afin d'obtenir la précision la plus haute possible, nous avons décidé d'appliquer un CNN (Réseau neuronal convolutif). Cette méthode, entraînée par le dataset MNIST dans un temps, s'est révélée comme la méthode la plus efficiente comparée à des méthodes comme la méthode MLP (Multilayer Perceptron) seule.

En effet, avec la méthode CNN choisie, l'application d'un certain nombres de couches extrayant des caractéristiques de l'image précède l'application du perceptron multicouche (MLP) .

La structure du Réseau Neuronal Convolutif est la suivante :

- Couche convolutive :

La première couche permet l'extraction des caractéristiques principales de l'image par l'application de filtres spécifiques. Chaque filtre appliqué dans une sous-couche vise une caractéristique spécifique de l'image (par exemple les contours de l'image ou outline filter, un filtre sur les lignes horizontales ou verticales de l'image, etc). L'entraînement du réseau avec l'application d'un nombre suffisant de filtres sur les images du dataset permet d'affiner les caractéristiques associées à chaque classe (chiffres de 0 à 9).

- Couche de pooling :

Cette couche induit la réduction de la taille de l'image avec l'extraction des poids maximaux (max pooling) par région de l'image.
L'application de plusieurs couches convolutives et de pooling permettent de spécialiser nos caractéristiques extraites en sortie.

- Perceptron multicouche (MLP):

Le perceptron multicouche permet de combiner les caractéristiques extraites par les couches précédentes et de les associer aux classes (correspondant aux neurones de la couche de sortie du MLP). Le perceptron multicouche calcule l'erreur entre la sortie attendue et la sortie générée en propageant la donnée d'entrée. Cette erreur est ensuite utilisée par rétropropagation pour alimenter le modèle MLP.

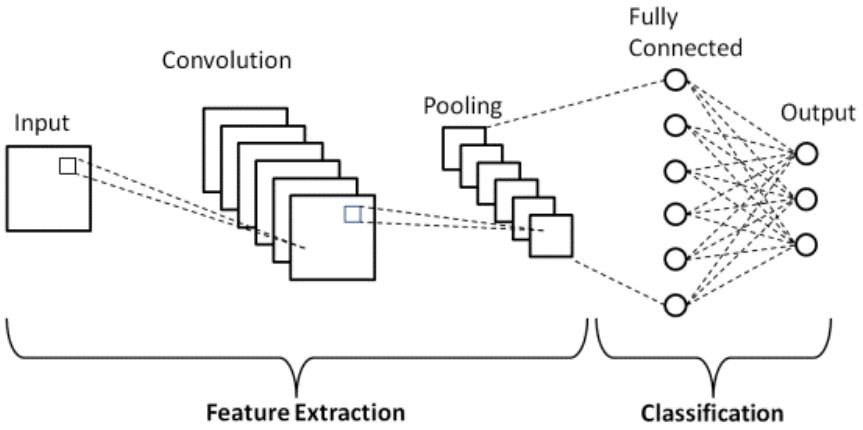

Figure 8 : Structure du modèle CNN

Dans notre cas, nous avons sélectionné le modèle *Alexnet* après comparaison avec les différentes méthodes et entraînements effectués. Les premiers entraînements effectués sur un modèle à trois couches donnaient déjà de très bons résultats, emmenant jusqu'à 99% de précision. Cependant, ces tests concernaient un entraînement sur le dataset MNIST.
La variable la plus sensible de la partie reconnaissance des chiffres étant que les chiffres analysés et extraits de la grille Hashiwokakero sont des chiffres cerclés. Cette variable entraîne une baisse considérable de la précision d'analyse allant jusqu'à 45% de précision pour les chiffres cerclés. L'image ci-dessous illustre un de nos résultats avec cette méthode, comprenant dans la première liste les chiffres que notre modèle prédit et en deuxième les chiffres corrects.

```
 Images with circles
98/98 [==============================] - 0s 4ms/step
[6 3 3 ... 3 4 2]
[6 5 3 ... 5 4 2]
0.44694533762057875
```

Figure 9 : Probabilité de reconnaissance des chiffres cerclés

Le modèle *Alexnet* choisi est lui constitué de 5 couches convolutives comprenant des filtres (Gaussian filter, Sobel filter, Edge detection filter notamment). Celui-ci est bien plus lourd `Total params: 21,598,922` que le modèle utilisé précédemment comme le montre le nombre de paramètres du modèle : .

Les couches de convolution appliquées dans le modèle s'appuient sur l'API Keras et sur le paramètre Conv2D. Cette méthode prend en compte le **nombre de filtres appliqués** (dans la première couche par exemple il y a 96 filtres), la **taille du kernel** étant les dimensions de la zone sur laquelle les filtres sont appliqués (couche 1 : en 11x11), les "**strides**" signifiant la taille du décalage lors de l'application du filtre sur l'image (4 pixels dans la première couche) et le **padding** ("same" dans notre cas) signifiant que l'on souhait conserver la taille du volume d'entrée en sortie.

```python
def define_alexnet_model():
    model = models.Sequential()

    model.add(layers.experimental.preprocessing.Resizing(224, 224, interpolation="bilinear", input_shape=(28,28,1)))

    model.add(layers.Conv2D(96, 11, strides=4, padding='same'))
    model.add(layers.Lambda(tf.nn.local_response_normalization))
    model.add(layers.Activation('relu'))
    model.add(layers.MaxPooling2D(3, strides=2))

    model.add(layers.Conv2D(256, 5, strides=4, padding='same'))
    model.add(layers.Lambda(tf.nn.local_response_normalization))
    model.add(layers.Activation('relu'))
    model.add(layers.MaxPooling2D(3, strides=2))

    model.add(layers.Conv2D(384, 3, strides=4, padding='same'))
    model.add(layers.Activation('relu'))

    model.add(layers.Conv2D(384, 3, strides=4, padding='same'))
    model.add(layers.Activation('relu'))

    model.add(layers.Conv2D(256, 3, strides=4, padding='same'))
    model.add(layers.Activation('relu'))

    model.add(layers.Flatten())
    model.add(layers.Dense(4096, activation='relu'))
    model.add(layers.Dropout(0.5))

    model.add(layers.Dense(4096, activation='relu'))
    model.add(layers.Dropout(0.5))

    model.add(layers.Dense(10, activation='softmax'))
    model.summary()
    model.compile(optimizer='adam', loss=losses.sparse_categorical_crossentropy, metrics=['accuracy'])
    return model
```

Figure 10 : Structure de modèle Alexnet

b) Apprentissage du modèle de convolution

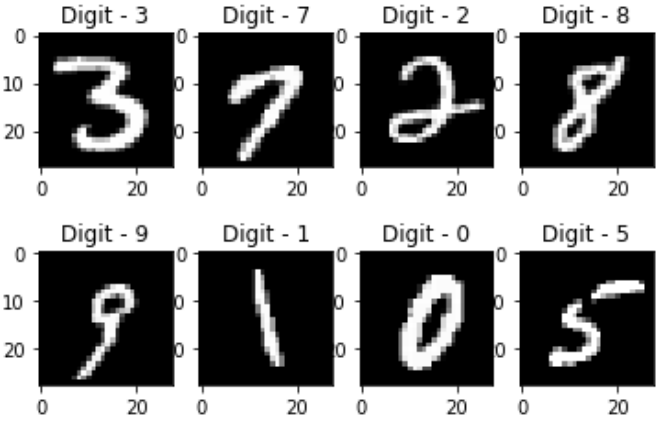

Figure 11 : Training dataset (MNIST) for the CNN Model

Afin d'effectuer l'entraînement de notre modèle CNN nous nous sommes ainsi basés sur le dataset MNIST pour commencer. Celui-ci, constitué d'une large banque de données (70 000 chiffres écrits à la main), nous permettait d'avoir un modèle performant. Cependant, comme mentionné précédemment, l'entraînement du modèle met en lumière une faiblesse liée au format d'entrée des chiffres que l'on souhaite reconnaître. Contrairement au MNIST dataset, les chiffres reçus en entrée suite à la reconnaissance des îles sont des chiffres cerclés.

Deux solutions étaient possibles:

- Conserver un entraînement du modèle avec le dataset MNIST et effectuer un redécoupage des images à l'entrée en supprimant les cercles entourant les chiffres.
- Modifier notre entraînement du modèle en l'alimentant avec un dataset évolué et prenant en compte la possibilité de cercle autour des chiffre

Nous avons ainsi opté pour la deuxième solution. Pour cela nous générons un dataset à partir de notre dataset de grilles de Hashiwokakero. L'objectif de cette solution est de spécialiser notre modèle sans perdre la performance apportée par le MNIST dataset.

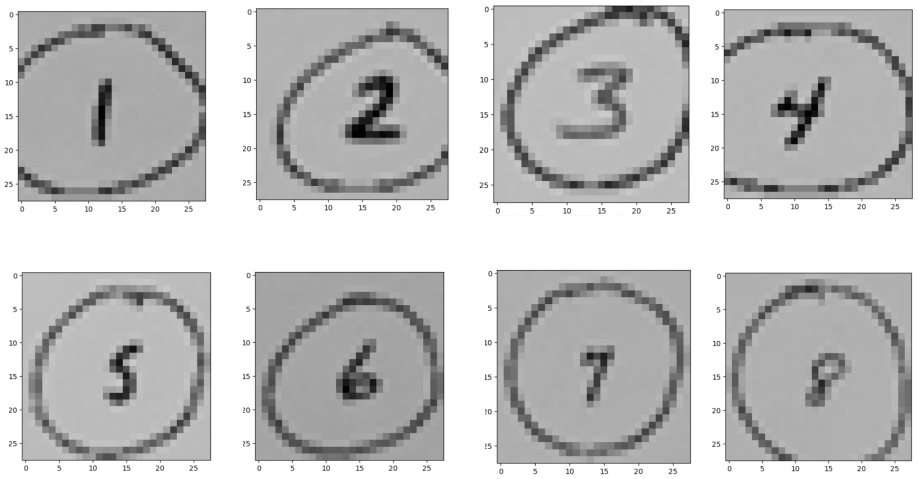

Figure 12 : Dataset généré des grilles d'Hashiwokakero

Notre dataset généré selon le format spécifié ci-dessus comprend ainsi 30900 images de chiffres allant de 1 à 8 contrairement au dataset MNIST contenant des 0 et 9. Celui-ci s'obtient par la découpe de grilles d'Hashiwokakero chiffre par chiffre avec en entrée l'image de la grille, la matrice des coordonnées des positions des différents chiffres sur l'image ([0,0,0] étant l'absence de chiffre dans la position de la matrice concernée) ainsi qu'une matrice des chiffres réels (0 étant l'absence de chiffre).

Cette spécialisation du modèle nous permet d'augmenter considérablement la précision de notre modèle qui atteint une précision de 95% pour les chiffres cerclés et 98% pour des chiffres non cerclés.

Concernant le format de sortie, on renvoie pour chaque chiffre testé une liste représentant la classe et la probabilité d'appartenance du chiffre à la classe. Les classes étant les chiffres de 1 à 8. Cela nous permet d'enlever les probabilités d'obtention d'un 0 ou d'un 9.

c) Limites et pistes d'améliorations

L'utilisation du modèle Alexnet ainsi que l'alimentation et spécialisation du dataset avec des données de chiffres cerclés améliorent le modèle mais celui-ci présente tout de même des faiblesses.
Un des avantages du format est que bien qu'une probabilité supérieure puisse être trouvée pour un chiffre incorrect, le modèle du solver possédera tout de même les probabilités des autres chiffres reconnus. Ainsi, s'il arrive qu'un 1 soit reconnu comme un 7 en premier, si la seconde probabilité est celle que ce chiffre soit un 1, c'est le 1 qui sera testé en deuxième choix.
Une des limites du modèle pourrait également être le cas où un espace de bruit du hashiwokakero est reconnu comme un cercle par la reconnaissance d'îles. Dans ce cas, le système de reconnaissance de chiffres pourrait reconnaître un 0 tout en attribuant des probabilités aux autres chiffres et donc introduire une variable pour le solver. La solution serait d'éliminer la liste de la matrice dans les cas où les probabilités des chiffres de 1 à 8 sont trop faibles en sortie, signifiant que le chiffre est soit un 0, soit un 9, soit du vide.

IV - Résolution de la grille

Une fois que les positions des îles ont été identifiées ainsi que les chiffres inscrits à l'intérieur de ces îles, il faut réaliser un solveur afin de résoudre la grille. Pour ce faire, nous avons utilisé la Programmation par Contraintes. Nous savons que cette méthode est simple et efficace : elle a déjà été utilisée auparavant pour résoudre des grilles de Hashiwokakero dans le papier de *Coelho et al.*, publié en 2019 : [1905.00973] Benchmark Instances and Branch-and-Cut Algorithm for the Hashiwokakero Puzzle. Ce papier nous donne une base de travail et nous garantit que notre méthode fonctionne pour la résolution de nos grilles de Hashi.

a) Format des entrées et sorties

Figure 13 : Comment la grille est-elle résolue ?

Les étapes précédentes permettent de donner au solveur une bonne représentation de la grille.

Entrées : Cette partie de l'application prend un fichier JSON en entrée, comme indiqué sur la Figure 6. Ce fichier est composé des dimensions de la grille, et d'un tableau indiquant pour chaque île :

- sa position sur la grille
- la probabilité pour chaque chiffre entre 1 et 8 qu'il soit présent sur cette île.

```
{
    "width": 5,
    "height": 4,
    "islands": [
        {
            "row": 0,
            "col": 0,
            "digits_probabilities": [
                0.5,
                0.0,
                0.1,
                0.4,
                0.4,
                0.4,
                0.1,
                0.0
            ]
        },
        {
```

Figure 14 : Exemple de fichier d'entrée

Sorties : Le solveur retourne, au format texte, une image de la grille résolue, le taux de confiance (produit des probabilités des chiffres) et le temps nécessaire pour résoudre la grille. La grille résolue indique les ponts placés entre les bonnes îles. La Figure 7 présente un exemple de sortie du solveur.

```
Solution :
 2=====4=====4
             ‖
 2--------3  ‖
 |        ‖  ‖
 |  2=====4  ‖
 |           ‖
 3=====4=====4

Confidence : 81.0%
Successfully solved the grid in 0.014 seconds
```

Figure 15 : Exemple de fichier de sortie

b) Modèle de base

Nous nous inspirons du papier cité précédemment pour créer un modèle de Programmation par Contraintes et le résoudre à l'aide d'un solveur. On choisit d'utiliser *OR-Tools*, un solveur open-source qui obtient de très bons résultats dans les compétitions de solveurs.

On définit des variables pour tout couple d'îles (i,j) :

- La variable binaire y_{ij} vaut 1 si et seulement si les îles *i* et *j* doivent être connectées par un pont
- La variable entière x_{ij} indique le nombre de ponts que l'on doit construire entre *i* et *j*. Les valeurs que peuvent prendre les x_{ij} sont 0, 1 ou 2.

Une fois les variables définies, il nous faut définir les contraintes que nous allons ajouter au modèle. Ces contraintes sont la traduction mathématique des règles du jeu, vues en introduction. Nous suivons toujours le travail de Coelho et al. en ajoutant les contraintes suivantes :

$$\sum_{i<k, i\in\delta(k)} x_{ik} + \sum_{j>k, j\in\delta(k)} x_{kj} = d_k \qquad k \in \mathcal{V} \qquad (1)$$

$$y_{ij} \leq x_{ij} \leq 2y_{ij} \qquad (i,j) \in \mathcal{E} \qquad (2)$$

$$y_{ij} + y_{kl} \leq 1 \qquad \{(i,j),(k,l)\} \in \Delta \qquad (3)$$

$$\sum_{\substack{i\in S, j\in\mathcal{V}\setminus S \\ \text{or } j\in S, i\in\mathcal{V}\setminus S}} y_{ij} \geq 1 \qquad S \subset \mathcal{V}, 1 \leq |S| \leq n-1 \qquad (4)$$

Figure 16 : Contraintes du problème de Coelho et al. 2019

Les contraintes (1) spécifient que le nombre de ponts connectés à une île doit être égal au chiffre inscrit sur l'île elle-même.

Les contraintes (2) spécifient que si deux îles sont connectées ($y_{ij} = 1$) alors le nombre de ponts entre elles doit être de 1 ou 2. Si elles ne sont pas connectées, le nombre de ponts entre elles doit être 0.

Les contraintes (3) spécifient que les ponts ne doivent pas se croiser.

Les contraintes (4) spécifient que toutes les îles doivent être connectées par des ponts. Pour tout sous-ensemble d'îles, il doit exister un pont qui sort du sous-ensemble. Le nombre de contraintes de ce type est beaucoup trop grand (2^n avec n le nombre d'îles), cette contrainte n'est donc pas implémentée dans le modèle.

A la place, on utilise le fait que si une solution est correcte, elle ne contient pas de sous-tours (sous-ensemble d'îles déconnectées du reste de la grille). On peut donc utiliser l'algorithme suivant :

Algorithm 1 Hashiwokakero Solver

```
Créer le modèle sans les contraintes (4)
sous_tour_existe ← True
while sous_tour_existe do
    solution ← resoudre_modele()
    sous_tour_existe ← detecter_sous_tours(solution)
    if sous_tour_existe then
        Ajouter des contraintes (4) au modèle pour éliminer les sous-tours
    end if
end while
```

Figure 17 : Algorithme d'élimination des sous-tours

c) Correction des erreurs

L'approche la plus simple consisterait à fixer chaque chiffre à sa valeur la plus probable. Avec cette méthode, si le système de reconnaissance d'images se trompe sur un seul chiffre, le solveur ne trouve pas de solution à la grille.

Pour corriger les erreurs, on définit pour chaque île *i* la variable entière d_i correspondant au chiffre identifié dans cette île. Elle prend donc ses valeurs entre 1 et 8.

On ajoute au modèle l'objectif de maximiser la probabilité que la grille soit correcte. En effet, cette partie du système prend en entrée la position des îles et la probabilité que chaque chiffre soit correct. La probabilité que la grille soit correcte se définit donc comme le produit de ces huit probabilités.

Le solveur cherche donc une solution à la grille qui maximise la probabilité que les chiffres soient corrects.

d) Vérification de l'unicité de la solution

Additionnellement, nous ajoutons la possibilité de réaliser des inférences supplémentaires si on sait que la grille donnée en entrée n'admet qu'une seule solution.

Si une seule solution est trouvée, la grille était correcte, nous avons donc trouvé la solution attendue. Si on trouve plusieurs solutions, nous pouvons affirmer que les chiffres utilisés ne sont pas corrects. Il faut donc que le solveur prenne en entrée la deuxième grille la plus probable au regard des probabilités données par le système de reconnaissance des chiffres. L'algorithme est le suivant :

Algorithm 2 Trouver la meilleure solution unique

```
solution_unique ← False
while not solution_unique do
  Modele1,Solution1 ← Maximiser les probabilités des chiffres avec Algorithm 1
  Modele2 ← Modele1
  Retirer l'objectif de maximisation de Modele2
  Fixer les chiffres de Modele2 à ceux trouvés dans Solution1
  Trouver toutes les solutions à Modele2
  if Plusieurs solutions then
    Interdire la combinaison de chiffres dans Modele1
  else
    solution_unique ← True
  end if
end while
```

Figure 18 : Algorithme de vérification de l'unicité de la solution

V - Architecture Orientée Service

1) Architecture et topologie

a) Déploiement sur cloud privée Openstack

Pour ce projet, notre application doit être déployée sur les serveurs OpenStack de l'INSA. OpenStack est une solution de cloud privé permettant de créer des machines et des réseaux virtuels. Pour notre projet nous avons utilisé un réseau privé et deux machines virtuelles. Une pour héberger l'application et l'autre pour héberger le code.

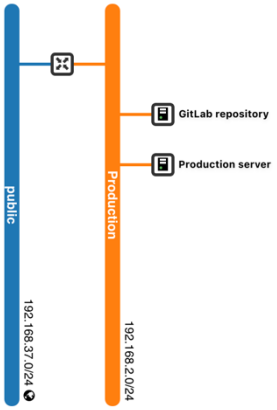

Figure 19 : Topologie des machines virtuelles OpenStack

b) Containerisation des services

Chacun des services décrit en partie 2 doit être hébergé dans une machine virtuelle sur les serveurs OpenStack de l'INSA. Nous avons décidé de packager chaque service dans un container Docker. Cela nous permet d'assurer une isolation entre nos services (chaque service ayant besoin de librairies différentes) et de nous assurer que ces services peuvent être rapidement packagés, démarrés, arrêtés, etc.

Nous créons donc un Dockerfile pour chacun des services. Chacun faisant usage d'une application Flask, les Dockerfile sont très similaires sinon identiques entre les services, seules les bibliothèques Python à installer sont différentes pour chaque service.

Dockerfile 195 bytes

```
1  # syntax=docker/dockerfile:1
2
3  FROM python:3.8-buster
4
5  WORKDIR /app
6  COPY . .
7
8  RUN pip3 install -r ./requirements.txt
9
10
11 EXPOSE 5000
12 ENTRYPOINT [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
13
```

Figure 20 : Exemple de Dockerfile

Nous avons également créé un Docker hub pour ce projet. Cela nous permet de garder une trace des images précédemment construites et donc de facilement retourner à ces versions si nécessaire pour construire les versions suivantes sur cette base.

2) Déploiement continue et Automatisation

a) répertoire Gitlab

Pour héberger le code de notre projet, nous avons déployé une instance GitLab dans une machine virtuelle sur les serveurs OpenStack. Ce choix a été fait pour faciliter la partie d'intégration et de déploiement continu (CI/CD) dans la suite du projet. GitLab propose en effet de nombreux outils directement intégrés à la plateforme pour faciliter ces étapes.

Nous avons créé un groupe pour l'ensemble du projet dans lequel nous avons ajouté un projet pour chacun des services que notre application utilise. Cela assure une bonne indépendance des services pour le développement. Cela facilite également le partage d'informations et de ressources entre les différents projets, notamment par le partage de variables d'environnement et de configuration par défaut des projets.

Différents utilisateurs ont également été ajoutés avec différents rôles :

- Des développeurs qui peuvent avoir accès au code de leurs projets, le modifier et lancer des pipelines CI/CD
- Des managers en charge des répertoires pouvant créer et supprimer des projets et gérer la configuration de l'instance GitLab ainsi des pipelines CI/CD.

Cela nous permet de mieux isoler le travail de chacun pour limiter les erreurs pouvant provoquer la perte de code.

b) GitLab runner

Pour effectuer les différentes opérations nécessaires à la réalisation des pipelines CI/CD GitLab a besoin d'instances de calcul appelées des GitLab runner. Il est possible d'accéder à des runner publics hébergés par GitLab via un système *pay as you go* avec un *free tier.* Nous avons cependant décidé d'installer notre propre instance de GitLab runner sur les serveurs OpenStack. Elle a été déployée sur le même serveur où seront déployés les services applicatifs. L'instance a été configurée pour pouvoir exécuter des pipelines en shell lorsqu'une modification est apportée au code source.

c) Stratégie CI/CD

Pour faciliter le déploiement nous utilisons un pipeline CI/CD. Les opérations à réaliser sont décrites dans un fichier nommé '.gitlab-ci.yml' situé à la racine de nos projets.

.gitlab-ci.yml 465 bytes Edit

```yaml
stages:
  - build
  - deploy

build:
  stage: build
  script:
    - echo $DOCKER_HUB_PWD | docker login --username hashiwokakero --password-stdin
    - docker build --network host -t hashiwokakero/solver:latest .
    - docker push hashiwokakero/solver:latest

deploy:
  stage : deploy
  script:
    - docker rm -f $(docker ps -aq --filter name=solver) || true
    - docker system prune -f
    - docker run -d --name solver -p 50004:5000 hashiwokakero/solver:latest
```

Figure 21 : Exemple de fichier .gitlab-ci.yml

Pour chacun de nos projets le pipeline est composé de deux étapes :

- Une étape *build* où une nouvelle image est construite avec la dernière version du code ajoutée. Lors de cette étape nous ajoutons également la nouvelle image dans notre docker hub.
- Une étape *deploy* où nous démarrons un container en utilisant l'image créée à l'étape précédente. Avant cela, une étape de suppression et de libération de l'espace alloué à la version précédente est effectuée pour éviter de stocker de nouvelles informations à chaque déploiement.

Ce Pipeline est déclenché à chaque fois qu'une modification est réalisée sur le répertoire github. Cela nous assure que les versions des containers déployés soient toujours les plus récentes.

Il est alors plus facile pour les développeurs de réaliser leurs tests, les étapes de construction d'images et de déploiement étant rendues automatiques.

Nous arrivons finalement à l'architecture décrite sur le diagramme en figure 22.

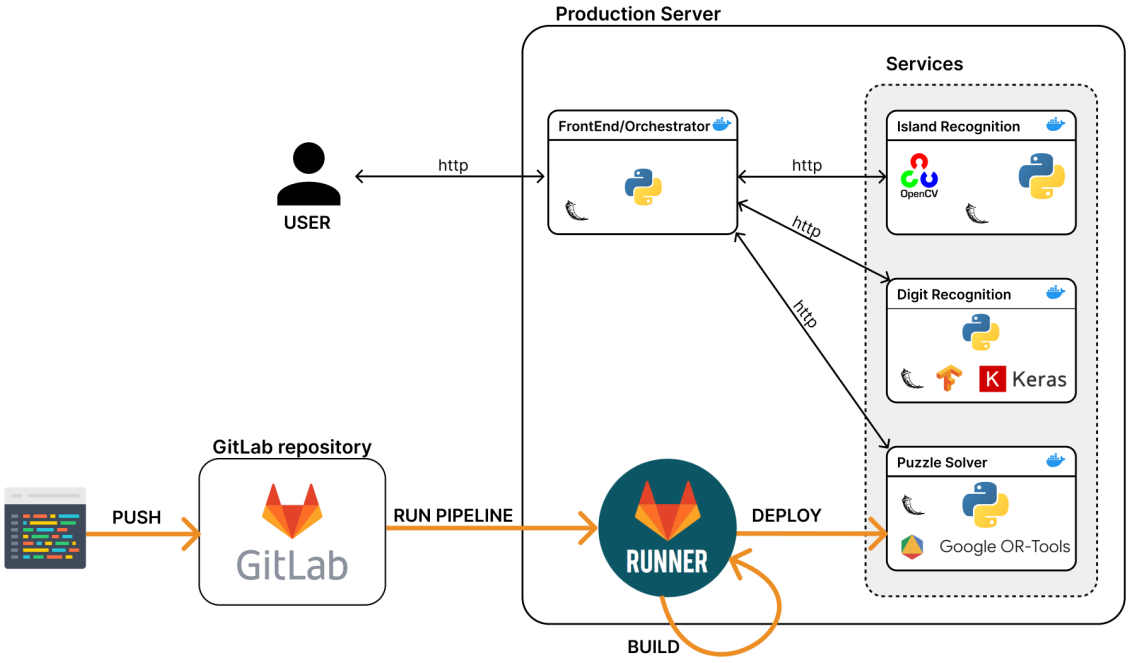

Figure 22 : Diagramme d'architecture générale du projet

3) Pistes d'améliorations

a) Amélioration des performances via des modifications d'architecture

Le fait que notre GitLab Runner soit situé sur la même machine virtuelle impose que lors de l'exécution d'un pipeline CI/CD les performances de l'application soient potentiellement impactées. En effet l'instance GitLab runner et les containers des différents services se partagent les ressources de la machine en mode best effort. Il serait intéressant de créer une nouvelle machine virtuelle uniquement dédiée au runner. La phase de build n'aurait alors pas d'impact sur l'application et le déploiement des nouvelles images sur le serveur de production pourrait être réalisé en utilisant ssh.

b) Ajout de tests dans les pipelines CI/CD

Actuellement, nos différents services ne possèdent pas de test. Il serait intéressant d'en ajouter pour s'assurer du bon fonctionnement de ces derniers. Nous pourrions alors ajouter ces phases de tests dans les pipelines CI/CD pour s'assurer que nous déployons des services passant les tests.

c) Séparation entre le développement et la production

Enfin, notre infrastructure ne dispose pas de séparation entre la partie développement/prototypage et la partie production accessible aux utilisateurs. Il nous faudrait ajouter des machines virtuelles pour que les équipes de développement puissent réaliser des tests sur des nouvelles versions de leurs services sans impacter les utilisateurs. Une fois la version prête à être déployée en production, elle pourrait être automatiquement passée en production en suivant un schéma de déploiement plus poussé pour assurer un service continu aux utilisateurs.

4) Gestion de projet et Organisation de l'équipe

a) Déroulement du projet

Nous présentons ici les différentes phases que nous avons suivies afin de réussir notre projet :

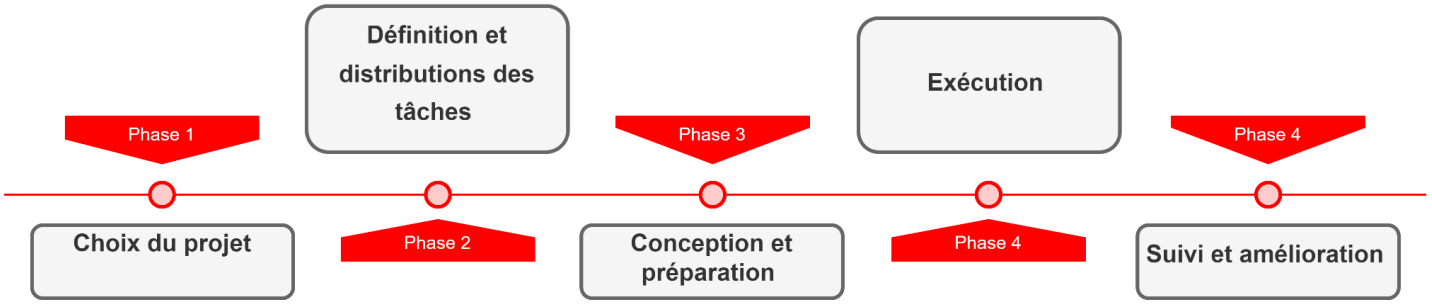

Figure 23 : Diagramme des différentes phases du projet

Après avoir choisi le sujet, nous nous sommes répartis les tâches en binôme afin d'être plus efficients. Nous avons mis au point des réunions hebdomadaires afin que chacun soit au courant de l'avancée des autres binômes.

b) Méthodologie de gestion

Pour réaliser ce travail, il était nécessaire d'utiliser une méthode agile. Donc on a mis en œuvre différentes pratiques agiles dans ce projet.

Nous avons utilisé ces pratiques agiles considérées parmis ceux les plus utilisées dans le développement de logiciels pour bien organiser notre travail et bénéficier de leurs avantages tels que les suivants :

- ➔ Minimisation des risques du projet
- ➔ Amélioration et garantie de la qualité
- ➔ Réduction du coût et du temps total sur l'ensemble du projet
- ➔ Amélioration de la communication entre toutes les parties prenantes

c) Outils utilisés

Pour permettre une bonne communication entre nous ainsi qu'avec les encadrants, et en plus des réunions hebdomadaires, nous avons utilisé Messenger et Google Drive comme moyens de communication interne et Discord comme moyen de communication externe.

Au sein des binômes, nous nous sommes aidés de Jira afin de suivre les tâches à faire, en cours et finies.

Pour pouvoir partager, organiser nos différents codes sources ainsi qu'assurer une intégration continue des différents microservices nous avons choisi GitLab comme plateforme de développement collaborative.

d) Progression globale du projet

Dans cette dernière section, nous allons discuter de notre progression sur le projet, au cours du dernier semestre.

Les premières semaines ont été consacrées à la découverte du projet et des outils liés à l'IA, ainsi qu'à la conception de l'architecture de l'application.

Nous avons ensuite procédé à des sprints d'une semaine chacun. Pour répartir le travail entre chacun d'entre nous, nous avons sélectionné des services indépendants et donné des points proportionnellement à leur difficulté afin que chaque membre ait la même charge de travail.

Conclusion

Nous avons développé une application qui résout une grille de Hashi faite à la main. Plusieurs services travaillent ensemble pour résoudre le problème. Premièrement, l'application identifie les îles et leurs positions sur la grille afin de les nettoyer. Ensuite, le chiffre à l'intérieur de l'île est extrait afin d'être communiqué au réseau de neurones et d'être identifié. Enfin, le problème est résolu et la solution est vérifiée.

Pour conclure, ce projet nous a beaucoup apporté. Il nous a permis à la fois de réinvestir des notions abordées tout au long de ce semestre et d'apprendre de nouvelles choses.

En effet, nous avons pu nous familiariser avec des technologies récentes et très sollicitées dans notre domaine. Nous avons également pu élargir nos connaissances conceptuelles en matière d'apprentissage automatique, d'IA, les microservices et l'intégration continue. Nous avons essayé de mettre en œuvre une application qui est entièrement microservices en utilisant tous les concepts de l'architecture SOA, ce qui rend l'application très flexible, modulaire et facile à mettre à jour.

Nous pouvons aussi souligner notre contribution concernant le travail d'équipe et la gestion d'équipe puisque nous avons mis en place un bon niveau de partage d'informations entre les membres du groupe ainsi qu'une répartition des tâches adaptée aux affinités de chacun selon différentes pratiques de développement logiciel agile.