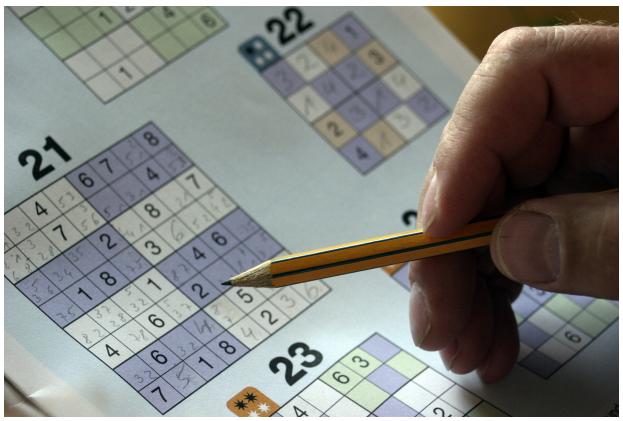
Projet en lien avec le cours SYS819 à l'École de Technologie Supérieure de Montréal

Résolution de sudokus avec un ResNet11-SPD : Modélisation des grilles comme des images à faible résolution



Vermorel Guilhem

<vermorelguilhem@gmail.com>

<https://github.com/guilhemvermorel>

réalisé le 20/01/2023

Table des matières

Extrait. Le sudoku est un sujet passionnant. Derrière ce jeu solitaire dont les règles sont plutôt simples, se cache une logique mathématique complexe qui a plus d'une fois essayé d'être solutionnée de manière automatique. Néanmoins, entre les résolutions bruts qui sont très longues et les méta-heuristiques de type réseaux de neurones qui ne donnent pas toujours des résultats concluant, il est difficile de trouver une solution viable à ce problème. L'idée de ce rapport est de tester une nouvelle approche du sudoku, en le modélisant comme une image 9x9 à faible résolution et d'utiliser une architecture de réseau de neurones à convolution adéquate : un *ResNet* [[10]] en remplaçant les blocs de *pooling* par des couches *SPD-Conv* [[1]]. Une couche *SPD-Conv* est composée d'un bloc *space-to-death* suivi d'une convolution *non-strided*. En comparant les résultats obtenus avec ceux présents dans les papiers, il s'avère que les accuracys obtenus sont meilleurs pour une complexité plus faible. Vous pouvez retrouver le code utilisé en open-source sur mon GitHub : `https://github.com/guilhemvermorel/Sudoku-solver-CNN`

I Introduction

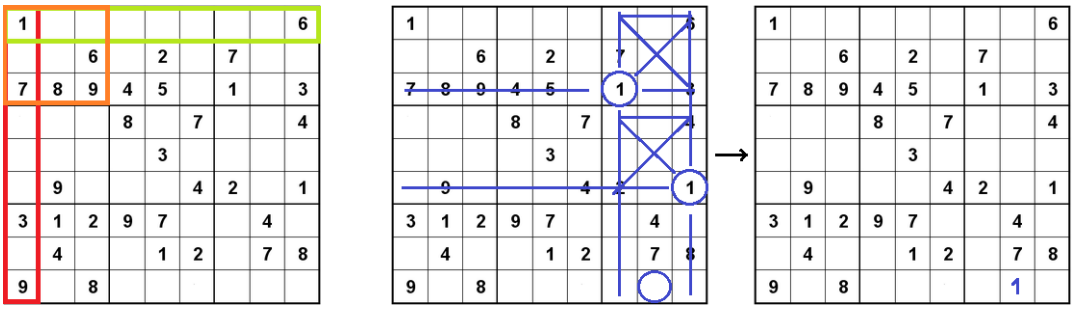
I.1 Contexte

La problématique d'apprendre à une intelligence artificielle à jouer à un jeu existe depuis ses débuts. En effet, en 1952 déjà, Arthur Samuel sortait un programme capable de comprendre le jeu de dame avec lequel il battra le champion du monde de l'époque. Au fil des années, et avec l'amélioration des technologies, les IAs ont gagné en puissance de calcul et se sont adaptés à des jeux de plus en plus complexe, des échecs, en passant par des jeux vidéo en tout genre, en allant jusqu'à battre le champion du monde du jeu de go, un des jeux avec le plus de combinaisons possibles de coup par tour.
Le sudoku est un jeu de grille où le but est de remplir l'ensemble des cases avec une série de chiffres, tous différents, qui ne doivent jamais se trouver plus d'une fois sur la même ligne, la même colonne ou la même zone (voir Figure 1 (a)). La résolution du sudoku est un problème mathématique complexe qui devient de plus en plus dur à résoudre à mesure que l'on enlève le nombre de cases remplies à l'état initial (voir Figure 1 (b)). Par la suite, on nommera les indices, ces cases remplies initialement.

Le but de ce projet est donc de **résoudre n'importe quel sudoku 9x9 ayant une unique solution, quelque soit son nombre d'indice, et de façon automatique à l'aide d'un réseau de neurones à convolution**.

Pour un sudoku classique de 9x9, en comptant toutes les rotations et permutations possibles, il n'existe pas moins de 10^{21} grilles remplies différentes. Des

solutions existent déjà pour résoudre le sudoku de manière brut, mais les temps de calculs sont néanmoins trop longs. L'idée serait ainsi de ne pas tester toutes les combinaisons possibles du jeu, mais plutôt d'utiliser un réseau de neurones à convolution comme méta-heuristique pour le résoudre plus rapidement.



(a) Le jeu du sudoku (b) Résolution

FIGURE 1 – Comment jouer au sudoku

I.2 Motivation

Ma motivation première est mon amour pour ce jeu qui m'a poussé à vouloir en apprendre plus. De par son côté mathématique et logique qui sont fascinantes et très prenantes, le sudoku rassemble des joueurs de toutes générations, que l'on soit petit ou grand, et j'en fais partie.
Dans un second temps, résoudre un sudoku participe à trouver le moyen de répondre à des problèmes mathématiques complexes [[2]]. Ici, plus particulièrement de type NPcomplet, qui font partis des problèmes les plus difficile à résoudre informatiquement. C'est à dire que le temps de résolution augmente de façon exponentielle à mesure qu'on augmente la taille du problème. C'est un des enjeux de l'informatique aujourd'hui.
Enfin, ce projet met en scène un aspect du Deep Learning qui est de plus en plus utilisé aujourd'hui dans les entreprises : l'apprentissage de réseaux de neurones à convolution. Il m'a permis de rentrer au coeur de cette notion complexe, de comprendre l'architecture type d'un réseau, de tester différentes approches, différentes modélisations dans le but d'obtenir les meilleurs résultats possible.

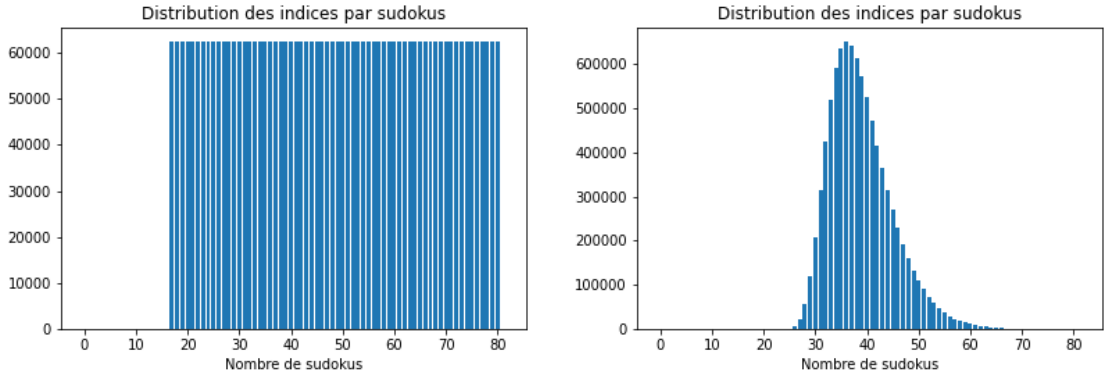
II Travaux connexes

II.1 Datasets utilisés

Le premier dataset est le principal dataset que l'on utilisera par la suite. Il vient de kaggle et il contient 4 millions de sudokus avec ses solutions, ainsi que le nombre

d'indices pour chacun [[3]]. Pour chaque sudoku, les lignes de chiffres de la grille sont juxtaposées les unes aux autres et les chiffres manquants dans les puzzles non-résolus sont remplacés par des 0. De plus, le nombre d'indices varie ici de 17 à 81 et est distribué suivant une loi uniforme dans le dataset, pour chaque nombre d'indices se trouve 62500 données (Voir Figure 2 (a)).
L'avantage de ce dataset est sa plus grande plage d'indices, qui permettra au modèle d'être capable de résoudre un plus grand nombre de sudoku, ce que l'on recherche initialement. De plus, le fait que celui-ci soit distribué de façon égale donnera au modèle une meilleure généralisation des données et donc de meilleurs résultats.

Le second dataset intervient en second plan, au cas où le premier ne satisfait pas. Il vient également de kaggle, mais contient cette fois-ci 9 millions de puzzles avec ses solutions [[4]]. Les données sont semblable aux données du premier à l'exception cette fois-ci que le nombre d'indices dans chaque sudoku suit une loi normal centrée autour de 36 indices (voir Figure 2 (b)).
Son grand nombre de données est son avantage principal ainsi que ses puzzles qui sont garantis comme n'ayant qu'une seule solution.



(a) Dataset 1 (b) Dataset 2

FIGURE 2 – Distributions des indices selon chaque dataset

.

II.2 Choix des métriques

Par la suite, nous choisirons 3 métriques différentes avec 3 difficultés d'obtention différentes :

- La première métrique que l'on peut présenter est l'accuracy par cases, elle compare le nombre total de cases qui ont été correctement trouvé par le

modèle sur l'ensemble des cases du testset [[8]]. En effet, cette métrique va rapidement être grande et illisible à mesure que l'algorithme comprend qu'il ne doit pas toucher aux cases initialement remplies. Néanmoins, il va nous permettre de tester l'efficacité de notre modèle sur ce point, mais surtout il va nous permettre de voir si celui-ci sort quelque chose, et quelque chose de cohérent qui plus est. On pourrait la définir de la façon suivante, M représentant l'ensemble des éléments du testset et k un puzzle aléatoire parmi cet ensemble :

$$accuracy1 = \frac{1}{M} \sum_{k=1}^{M} \frac{Nombre\_de\_cases\_prédites_k}{Nombre\_de\_cases_k}$$

- La seconde métrique, elle, est la plus largement représentée dans les papiers, il s'agit de l'accuracy par cases vides, elle compare le nombre totale de cases vides qui ont été correctement trouvées par le modèle sur l'ensemble des cases vides du testset [[5],[6],[7]]. Celle-ci va nous permettre de véritablement tester les performances du modèles pour ce qui est de son efficacité à remplir le puzzle. Elle se définit comme ce qui suit :

$$accuracy2 = \frac{1}{M} \sum_{k=1}^{M} \frac{Nombre\_de\_cases\_vides\_prédites_k}{Nombre\_de\_cases\_vides_k}$$

- La dernière métrique enfin, est sûrement la plus simple et la plus connu de toutes, mais également la plus difficile à obtenir, puisqu'elle représente l'accuracy standard calculant le nombre de sudoku entièrement rempli sur l'ensemble des sudokus. Elle correspond totalement à la problématique que l'on s'est fixé mais cela risque d'être compliqué d'obtenir de bon résultats avec. En effet, on comprend bien qu'une erreur pour résoudre une seule case du sudoku assignerait la valeur fausse au sudoku entier. On voit sa définition ici :

$$accuracy3 = \frac{Nombre\_de\_sudokus\_résolus}{Nombre\_de\_sudokus}$$

II.3 Méthodes adoptées

Dans les papiers, on trouve beaucoup de réseaux de neurones à convolution très simple dans l'utilisation, qui empilent juste des couches de convolutions à la suite :

- Un réseau constitué de 16 couches, dont 15 couches de convolutions successives composées de 512 filtres 3x3. [[5],[6]]

- Un réseau constitué de 11 couches, dont 10 couches de convolutions successives composées de 512 filtres 3x3. [[7]]

- Un réseau constitué de 6 couches : 3 couches de convolutions composées de 64 filtres 3x3 sur les deux première et de 128 filtres 1x1 sur la dernière. [[8]]

Dans ce projet, l'architecture que l'on exploitera est un réseau de neurones à convolution inspiré du *ResNet50* [[10]], le *ResNet50-SPD* [[1]], il s'agit d'une suite de 4 couches *SPD-conv* entre lesquels on place une suite de plusieurs couches de convolution classiques sans utiliser de couche de *pooling*, pour finir sur une dernière couche *fully connected*.
Cette architecture, inventée par Raja Sunkara and Tie Luo a pour but de s'intéresser au image de faible dimension pour de la classification, que l'on pourrait apparenter à la résolution de nos sudokus. En effet, les algorithmes classiques de machine learning sont utilisés sur des images avec une très bonne résolution et marche moins bien sur des images avec un plus petit format. Dans les grandes images on a beaucoup d'informations redondantes et le but des couches de *pooling* est de ressortir l'essentiel de cette information en supprimant celle qui est inutile et également de réorganiser les données dans un nouvel espace qui favorise la disparité. Or dans le cas des petites images, chaque information compte et il serait peu efficace d'en réduire la dimension encore. Ainsi, pour résoudre ce problème, nous pouvons les remplacer par des couches de *SPD-conv*. Elles consistent à enchaîner des couches *SPD* (*space-to-depth*) suivis de couches de convolution *non-strided*, c'est à dire avec une valeur de *stride* égale à 1 (voir Figure 3).
Les couches *SPD* empilent 4 blocs les uns à la suite des autres dans l'axe de la profondeur. Ces blocs correspondent à l'image d'entrée dont l'on récupère une donnée tous les deux pixels sur les 2 axes spatiaux, chacun décalé d'un pixel pour récupérer l'ensemble des données une seul fois (voir Figure 3 (c)). Cette action va permettre de combiner les *features maps* entre elles sans perdre d'informations des données d'entrée.

Pour ce qui est des hyperparamètres choisis, nous commencerons avec ceux présents dans la Table 1, qui correspondent à ceux qui sont les plus représentés dans les papiers.

Activ. Func.	Optim.	Loss Func.	lr	batch s.	weight decay	n. epochs
ReLu	Adam	Cross entropy	0.0001	32	0.00001	50

Table 1 – Choix des hyperparamètres

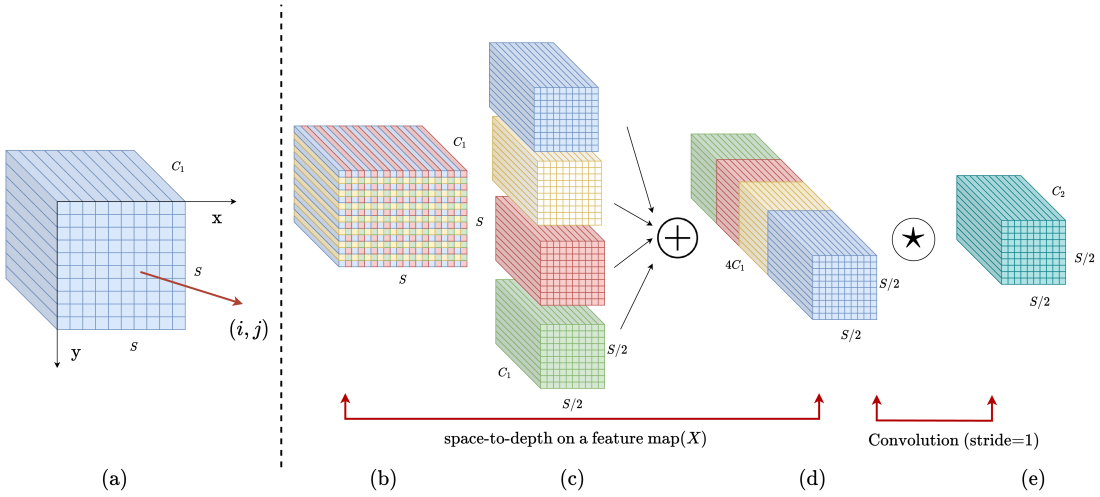


FIGURE 3 – Configuration des couches *SPD-conv* [[1]]

Nous comparerons les performances obtenus et l'emprunte mémoire de notre modèle avec les hyperparamètres sélectionnés en fonction des autres modèles déjà présent dans les papiers.

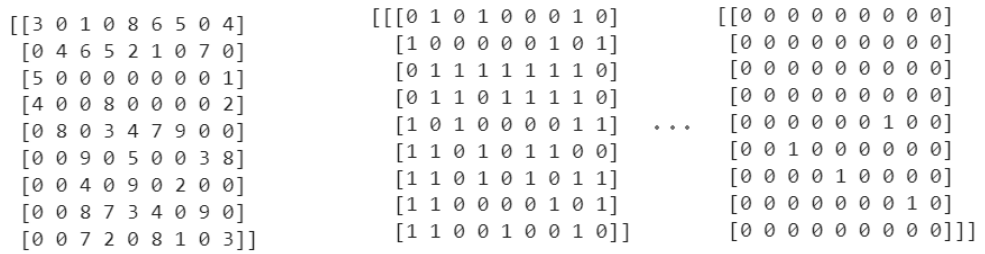
III Implémentation

III.1 Mise en place du CNN

Après la revu de littérature, notre projet c'est plus focalisé sur une approche qui consiste à modéliser les sudokus comme de petites images 9x9 pour utiliser l'architecture du *ResNet50-SPD* afin de classifier chaque case de sortie entre les 9 valeurs possibles (de 1 à 9). Ainsi, on obtient un algorithme qui classifie les 81 cases du sudoku sur ces 9 valeurs, et qui le résout par la même occasion.

Pour ce faire, on a d'abord choisit d'encoder les valeurs afin d'enlever la relation d'ordre qu'il existe dans le sudoku à cause des valeurs des cases allant de 0 à 9 (rappel : 0 signifiant que la case est vide). Pour ce faire on a d'abord transformé les sudokus du dataset (voir organisation du dataset1/2) en un tableau 9x9 (voir Figure 4 (a)), puis en un tableau binaire 10x9x9 avec l'axe 0 correspondant aux valeurs possible des cases et les axes 1 et 2 correspondant à la localisation spatiale dans la grille (voir Figure 4 (b)). Par exemple, la valeur de 1 du sudoku selon les dimensions (9,5,2), veut dire qu'il y a un 9 sur le sudoku à l'intersection entre la $6^{ème}$ ligne et la $3^{ème}$ colonne.

Pour ce qui est des valeurs de sorties, le sudoku est cette fois-ci modélisé comme

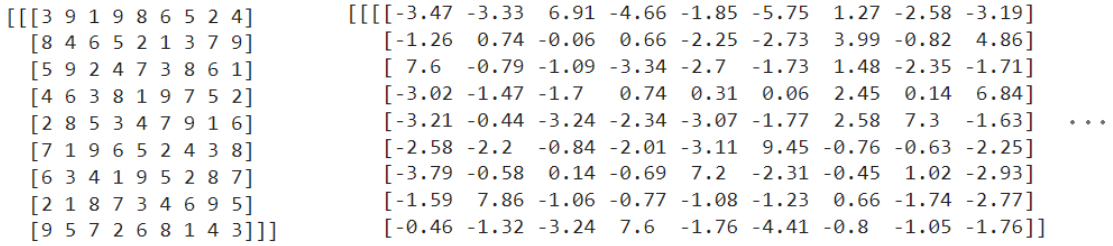


(a) Sudoku en array (b) Sudoku encodé

FIGURE 4 – Représentation des sudokus en entrée du modèle

.

un tableau 9x9x9 avec les axes 1 et 2 représentant les dimensions spatiales et l'axe 3 les valeurs possible des cases (Voir Figure 5 (b)), de même que les valeurs des *target* pour qu'elles soient comparé à l'*output* dans la *loss function* (Voir Figure 6 (b)). Par exemple, si la valeur du sudoku selon les dimensions (0,1,8) est maximale pour toute la ligne du tableau, alors la case du sudoku à l'intersection entre la $1^{ère}$ ligne et la $2^{ème}$ colonne vaut 9 (On peut notamment vérifier que cette valeur est une mauvaise sortie sur la *target*, mais que l'algorithme hésitait avec 7 au vu de la valeur selon les dimensions (0,1,6)).



(a) Sudoku en array (b) Sudoku encodé

FIGURE 5 – Représentation des sudokus en sortie du modèle

.

Enfin, les données d'entraînement, de validation et de test sont divisés de la façon suivante : 0.6375/0.1125/0.25 et chaque accuracy évoqué dans la partie II.2 a été codé à la main. L'apprentissage étant effectué sur la plateforme Google Colab [[9]] avec une capacité de RAM et de temps de RAM limité, il est nécessaire d'enregistrer le modèle à plusieurs reprises (toutes les 5 *epochs*) pour pouvoir effectuer l'entraînement en plusieurs fois.

(a) Sudoku en array (b) Sudoku encodé

FIGURE 6 – Représentation des sudokus cibles du modèle

.

III.2 Changement du modèle initiale

Après une bonne compréhension du réseau de neurones *ResNet50* et des couches *SPD* , il a été possible de faire des réarrangement dans la taille des canaux pour qu'il puisse prendre en entrée au format d'*array* 10x9x9 et en sortie 9x9x9.
Il fallait également changer la fonction *SPD* pour qu'elle puisse être utilisée sur des images de taille impair. En effet, si l'on revient à la Figure 3, on peut remarquer que celle-ci ne fonctionne que pour des images pairs et même des images avec des dimensions en 2^n si l'on veut les enchaîner par la suite (on divise par 2 les dimensions selon x et y à chaque *SPD*). Pour cela, on ne constitue plus 4 blocs récupérés tous les 2 pixels selon les 2 axes spatiaux, mais 9 blocs récupérés tous les 3 pixels selon ces 2 axes spatiaux de la façon suivante :

$$f_{0,0} = X[0:9:3, 0:9:3], f_{0,1} = X[0:9:3, 1:9:3], ...$$

$$..., f_{1,2} = X[1:9:3, 2:9:3], f_{2,2} = X[2:9:3, 2:9:3]$$

Néanmoins, avec cette nouvelle couche, la dimension de l'*input* réduit drastiquement, elle est divisée par 3 à chaque opération. Ainsi, après 2 opérations, l'*array* passe de 10x9x9 à un *array* 10x1x1. Or, le modèle initial du *ResNet50-SPD*, stipule d'en faire 4, mais en enlevant la première *SPD-conv* et en réduisant le modèle des dernières couches de convolution et de la dernière couche *SPD-conv*, on obtient le modèle de la Figure 7. Afin de baisser le nombre de paramètres et d'avoir un temps de calcul plus faible (très utile quand on fait l'entraînement sur Colab), on ne mettra qu'une couche de convolution entre les réseaux. On obtient un *ResNet11-SPD*, avec 10 couches de convolution et une couche *fully connected*.

Finalement, le réseau de neurones final contient 5 347 545 paramètres. Quand on compare ce chiffre avec ceux des autres modèles énoncés à la partie II.3, 33 millions [[5],[6]], 21 millions [[7]], 10 millions [[8]], il est 2 à 6.5 fois inférieur.

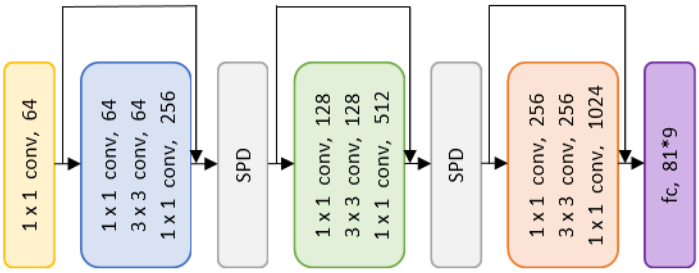


FIGURE 7 – Modèle *ResNet11-SPD*

IV Expériences

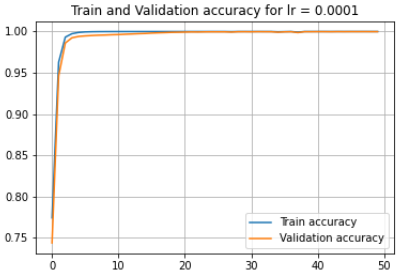
IV.1 Premiers résultats : sous-ensemble avec 1 seule case vide

En prenant le dataset 1, présenté dans la partie II.1, on décide dans un premier temps, de réaliser une étude préliminaire sur un sous-ensemble ne contenant que les sudokus avec un nombre d'indice égale à 80, autrement dis avec une unique case vide. Ce sous-ensemble comporte 62 500 valeurs, ce qui reste peu mais est très rapide à entraîner (moins d'une heure sur Colab), et après entraînement, on obtient les premiers résultats de la Figure 8.
L'algorithme converge rapidement (avant la $25^{ème}$ *epoch*), et les résultats sur le testset sont très bon :

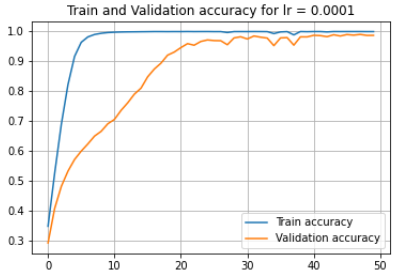
$Average\,loss$: 0.0001, $accuracy1$: 1.0000, $accuracy2$: 0.9991, $accuracy3$: 0.9991

Au vu des résultats, le modèle comprend très bien la logique du jeu : il ne doit pas changer les chiffres différents de 0 et doit remplacer le 0 en une valeur logique dans l'ordre des autres valeurs du tableau. Bien sûr, ce modèle ne comprend sûrement qu'un aspect très simple du sudoku, on regarde si c'est bien le cas en testant celui-ci sur plus de données (en gardant son entraînement sur le trainset à 80 indices). On obtient les résultats de la Table 2.

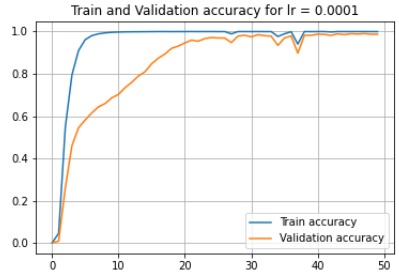
On voit bien que l'accuracy baisse drastiquement à mesure qu'on enlève le nombre de cases initiales du sudoku. Néanmoins, si l'on regarde l'accuracy2 du test effectué sur les sudokus avec 17 indices de départ, cette valeur est 1.25 fois plus grande qu'un algorithme qui choisirait les valeurs de façon aléatoire (qui vaudrait 0.1111, correspondant à 1 chance sur 9 pour chaque case vide). L'algorithme arrive donc à comprendre certains aspect du jeu avec des sudokus très complexe, alors qu'il n'est entraîné que sur une partie très simple de celui-ci.



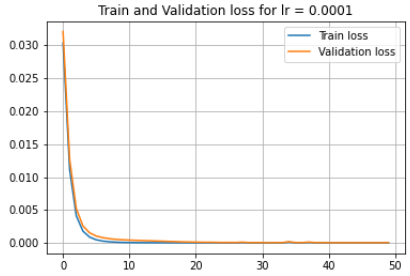
(a) Accuracy1



(b) Accuracy2



(c) Accuracy3



(d) Loss function

FIGURE 8 – Évolution des accuracys et de la fonction de perte pour le trainset et le validset pour le sous-ensemble avec 80 indices

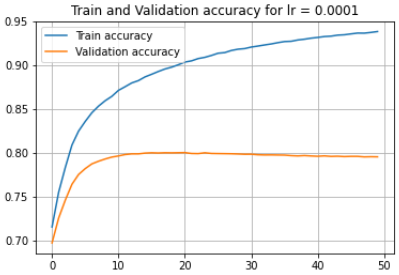
.

Indices testés	Loss Func	Accuracy1	Accuracy2	Accuracy3
80	0.0001	1.0000	0.9991	0.9991
75	0.0005	0.9924	0.8969	0.5565
70	0.0018	0.9719	0.7939	0.1030
60	0.0083	0.8922	0.5963	0.0001
40	0.0440	0.5804	0.2757	0.0000
17	0.1097	0.2337	**0.1386**	0.0000

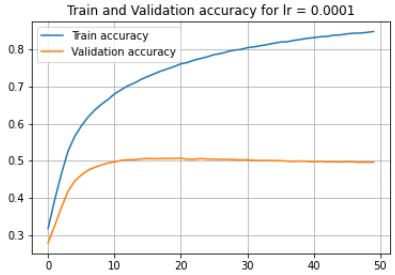
TABLE 2 – Résultats de tests obtenus en fonction du nombre d'indices du testset pour le modèle entraîner avec le sous-ensemble à 80 indices

IV.2 Preuve de sur-apprentissage : sous-ensemble avec tous les indices

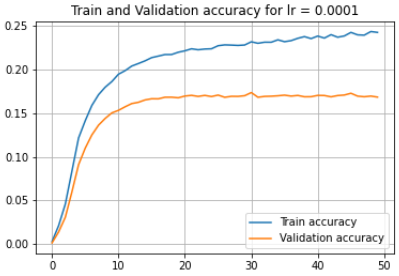
On décide ensuite de faire l'entraînement sur un sous-ensemble également, mais contenant une distribution uniforme d'indices entre 17 et 80 (200 000 données). Les résultats eux par contre sont très mauvais (Voir figure 9). En plus d'observer une forme de sur-apprentissage, les résultats sur l'ensemble de validation son très faibles, surtout pour l'accuracy2 qui ne dépassent pas 0.5.



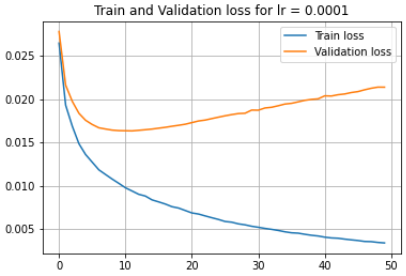
(a) Accuracy1



(b) Accuracy2



(c) Accuracy3



(d) Loss function

FIGURE 9 – Évolution des accuracys et de la fonction de perte pour le trainset et le validset pour le sous-ensemble avec tous les indices

.

Les résultats de l'évaluation du modèle sur le testset donnent :

$Average\,loss$: 0.0635, $accuracy1$: 0.5103, $accuracy2$: 0.2994, $accuracy3$: 0.0000

La Table 3, reprend les résultats d'évaluation selon le nombre d'indices des sudokus du testset. On voit que ceux-ci sont bien meilleurs que pour ceux de la Table 2 (presque 2 fois supérieur pour l'accuracy2 pour les sudokus les plus difficiles). Mais restent très faible pour les sudokus à difficulté moyenne et difficile.

Indices testés	Loss Func	Accuracy1	Accuracy2	Accuracy3
80	0.0001	0.9999	0.9974	0.9930
75	0.0001	0.9979	0.9730	0.8456
70	0.0005	0.9916	0.9398	0.5227
60	0.0032	0.9571	0.8378	0.0395
40	0.0273	0.7552	0.5211	0.0003
17	0.0763	0.4054	**0.2478**	0.0000

TABLE 3 – Résultats de tests obtenus en fonction du nombre d'indices du testset pour le modèle entraîner avec le sous-ensemble contenant tous les indices

Bien que le trop faible nombre de données du trainset doit jouer sur le sur-apprentissage du modèle, en exploitant le dataset 1, on se rend compte que les sudokus ne possèdent pas tous une unique solution. En effet, ils sont générés de façon aléatoire et les cases vides initiales sont elles aussi enlevées de façon aléatoire. Ainsi, à partir de 4 cases vides, il est possible d'avoir des solutions différentes (voir la Figure 10), et la probabilité d'en avoir augmente à mesure que le nombre d'indices baisse jusqu'au nombre d'indices égale à 17 dont la probabilité d'avoir une solution unique est quasi-nul. En effet, 17 est le nombre minimum d'indices avant de ne plus trouver aucune solution unique aux sudokus et on ne peut trouver que 50 000 sudokus avec 17 indices possiblement unique [[11]]. Or, il existe $6.67 * 10^{21}$ sudokus complets différents, donc une probabilité de $7.75 * 10^{-18}$ d'obtenir une solution unique lorsque l'on retire aléatoirement 64 cases d'un sudoku lui-même généré de façon aléatoire.
Cependant, avoir des sudokus avec une solution unique est indispensable pour notre modèle, dans le cas contraire on ne pourrait pas l'évaluer puisqu'il peut très bien avoir compris comment jouer au jeu sans pour autant avoir la même solution que la *target*, de même pour la *loss function* lors de l'entraînement.

1	4	**5**	**3**	2	7	6	9	8
8	3	9	6	5	4	1	**2**	7
6	**7**	2	9	**1**	8	**5**	4	3
4	9	6		8	**5**	**3**	7	
2	**1**	8	4	**7**	3	9	5	**6**
7	5	**3**		9	6	4	**8**	
3	**6**	7	**5**	4	2	8	1	**9**
9	8	**4**	7	6	1	2	**3**	5
5	2	1	8	3	**9**	**7**	6	4

(a) Sudoku initiale

1	4	**5**	**3**	2	7	6	9	8
8	3	9	6	5	4	1	**2**	7
6	**7**	2	9	**1**	8	**5**	4	3
4	9	6	1	8	**5**	**3**	7	2
2	**1**	8	4	**7**	3	9	5	**6**
7	5	**3**	2	9	6	4	**8**	**1**
3	**6**	7	**5**	4	2	8	1	**9**
9	8	**4**	7	6	1	2	**3**	5
5	2	1	8	3	**9**	**7**	6	4

(b) Solution 1

1	4	**5**	**3**	2	7	6	9	8
8	3	9	6	5	4	1	**2**	7
6	**7**	2	9	**1**	8	**5**	4	3
4	9	6	2	8	**5**	**3**	7	**1**
2	**1**	8	4	**7**	3	9	5	**6**
7	5	**3**	1	9	6	4	**8**	2
3	**6**	7	**5**	4	2	8	1	**9**
9	8	**4**	7	6	1	2	**3**	5
5	2	1	8	3	**9**	**7**	6	4

(c) Solution 2

FIGURE 10 – 1 sudoku avec 77 indices et 2 solutions possibles

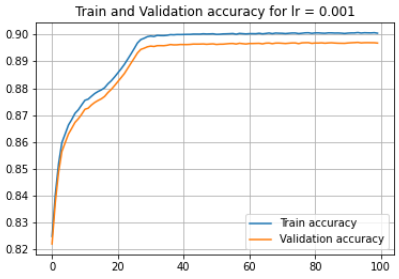
.

On doit donc changer de dataset.

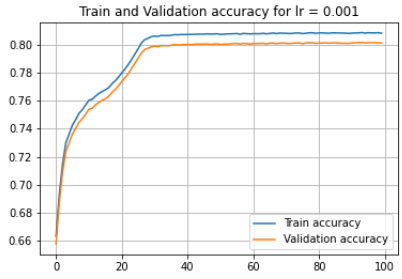
IV.3 Changement de dataset et augmentation de la taille de batch

On fera désormais l'entraînement du modèle sur le dataset 2 qui ne contient que des valeurs avec une unique solution, mais est normalement distribué et ne contient pas certains nombre d'indices de départ (voir la Figure 2 (b)). On ne pourra donc pas faire de tests en fonction du nombre d'indices de départ, mais l'algorithme devrait être en mesure de comprendre le jeu. On décide donc de prendre cette fois-ci l'ensemble du dataset pour avoir une meilleure généralisation. Néanmoins, celui-ci est trop conséquent pour pouvoir entraîner notre modèle sur Colab. On essaye de garder un maximum admissible, environ 2.6 millions de

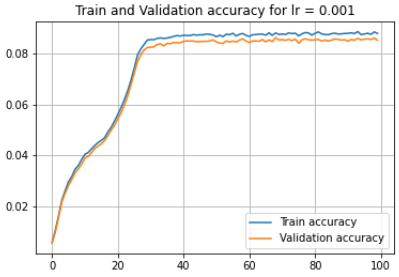
données dont 1.7 millions pour l'entraînement uniquement. Mais, cette fois-ci l'entraînement est trop long (20 minutes par *epoch*). Après quelques recherches, il est possible d'augmenter la *batch size* lors de l'entraînement et de continuer d'avoir de bons résultats à condition d'augmenter aussi le *learning rate* [[12]]. On passe donc la *batch size* à 512 et le *learning rate* à 0.001. On entraîne le modèle ainsi et on obtient les résultats de la Figure 11.



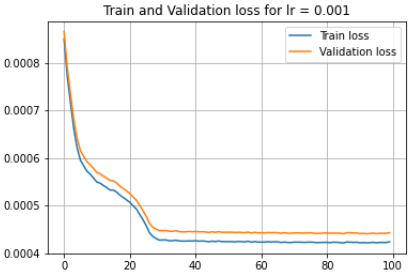
(a) Accuracy1



(b) Accuracy2



(c) Accuracy3



(d) Loss function

FIGURE 11 – Évolution des accuracys et de la fonction de perte pour le trainset et le validset pour le dataset 2 avec une taille de batch à 512

.

Et on obtient les résultats suivant sur le testset :

$Average\,loss$: 0.0004, $accuracy1$: 0.8968, $accuracy2$: **0.8015**, $accuracy3$: **0.0857**

Ceux-ci sont relativement bon pour l'accuracy2 mais restent encore très mauvais pour l'accuracy3. Cependant, ici l'algorithme essaye de prédire la valeur de l'ensemble des cases vides du sudoku en même temps, une approche qui semble inappropriée pour le résoudre.

IV.4 Résultats Finaux : une approche de test plus humaine

On essaye de mettre en place une approche plus humaine de la résolution du modèle. En effet, lorsque l'on résout un sudoku, on a tendance à prendre chaque

case l'une après l'autre, et on rempli en premier les cases dont l'on est sûr ou quasi-sûr d'avoir la bonne réponse (sinon on risque de devoir tout recommencer à un moment).
Ici, l'idée est donc la même, on évalue chaque sudoku l'un après l'autre (donc avec un *batch size* = 1), on regarde la sortie du modèle comme vu sur la Figure 5 (b), et on récupère les coordonnées de la valeur la plus élevé. Elles nous indiquent alors la case et le chiffre à assigner (entre 1 et 9) pour lesquelles l'algorithme est le plus sûr de lui. On remplace cette case-ci dans le sudoku initial avec la valeur prédite et on refait tourner le modèle. On recommence ainsi tant qu'il reste des cases vides dans le sudoku initial.
En faisant cette manipulation lors de l'évaluation sur le testset, on obtient les résultats finaux suivants :

$Average\,loss$: 0.0005, $accuracy1$: 0.9659, $accuracy2$: 0.9343, $accuracy3$: **0.8059**

Tous les résultats, et particulièrement l'accuracy3, augmentent drastiquement, l'approche initié est tout à fait justifiable et fonctionne. On forme 3 sous-ensembles du testset avec des nombres d'indices faibles, moyens et élevés en fonction de la distribution des données et on peut voir les résultats d'évaluation pour chacun d'eux dans la Table 4.

Nombre d'indices	Loss Func	Accuracy1	Accuracy2	Accuracy3
23 à 30	0.0001	0.9999	0.9994	0.9971
36 à 38	0.0002	0.9968	0.9929	0.9672
49 à 77	0.0019	0.8861	0.8165	0.4429

TABLE 4 – Résultats de tests obtenus en fonction du nombre d'indices de départ pour le dataset 2

Un gros point problématique que l'on peut soulever est que le temps de calcul est très long avec cette méthode (2h30 pour 100 000 données de tests)

IV.5 Comparaisons avec les autres modèles

Pour comparer notre modèle avec ceux que l'on peut trouver dans les papiers, il est nécessaire d'effectuer l'entraînement sur le même dataset de départ. Étonnement, les trois autres modèles sont tous entraînés et testés sur le même dataset, possèdant 1 million de sudoku avec une unique solution chacun, disponible sur kaggle [[13]]. Finalement, après évaluation de notre modèle sur un sous-ensemble de 200 000 données, on obtient les résultats suivants :

$$Nb.de\,param. : 5millions,$$

$Average\,loss$: 0.0001, $accuracy1$: 0.9912, $accuracy2$: **0.9865**, $accuracy3$: **0.8557**

En comparaison, la Table 5 présente les résultats obtenus par les différents modèles trouvables dans les papiers.

Modèle utilisé	Nb. de param.	Accuracy1	Accuracy2	Accuracy3
Modèle de Akin-David et Mantey [[5]]	33 millions	?	0.86	$\simeq$0.10
Modèle de Ching (amélioration du précédent) [[6]]	33 millions	?	0.93	$\simeq$0.10
Modèle de Kyubyong [[7]]	21 millions	?	0.86	?
Modèle de Lyly123 [[8]]	10 millions	0.97	?	?

TABLE 5 – Résultats de tests obtenus par les modèles existants

Finalement, bien que l'on est pas toutes les accuracys pour les modèles existants, les résultats de notre modèle sont supérieur pour chacune d'elles, notamment pour l'accuracy3 avec une augmentation de 0.75. Il généralise, ainsi, bien mieux la résolution des sudokus pour moins de paramètres, 2 fois moins au minimum.

V Conclusion

En changeant le modèle du *ResNet50-SPD* [[1]], lui-même inspiré du *ResNet50* [[10]], nous avons pu l'adapter à nos entrée, les sudokus. En changeant, ensuite, l'approche de test pour résoudre chaque case vide l'une à la suite de l'autre, il nous a été possible d'obtenir des résultats excellent pour la résolution des cases vides (accuracy de 0.9865), mais également très bon pour résoudre le sudoku dans son intégralité (accuracy de 0.8557). Les scores sont beaucoup plus élevés que ceux que l'on peut trouver dans les papiers [[5],[6],[7],[8]] (augmentation de l'accuracy de 0.75 pour la deuxième évoquée plus haut), pour un nombre de paramètres bien inférieur (entre 2 et 6.5 fois inférieur). Néanmoins, le temps d'évaluation avec la dernière approche est trop longue pour qu'elle soit utiliser dans l'entraînement.
Enfin, pour réduire le temps de calcul lors de l'évaluation du modèle il pourrait être intéressant de tester des techniques d'optimisation de réseaux de neurones tels que la *quantization*, le *pruning*, la *factorization* ou encore la *distillation*.

Bibliographie

[1] Raja Sunkara et Tie Luo, *No More Strided Convolutions or Pooling : A New CNN Building Block for Low-Resolution Images and Small Objects*, https://arxiv.org/pdf/2208.03641v1.pdf, 2022.

[2] waytolearnx, *Différence entre un problème NP-Complet et NP-Difficile*, https://waytolearnx.com/2019/03/difference-entre-un-probleme-np-complet-et-np-difficile.html, 2019.

[3] ?? , dataset, *4 million of sudoku puzzles*, https://www.kaggle.com/datasets/informoney/4-million-sudoku-puzzleseasytohard, 2022.

[4] VAPONI , dataset, *9 Million Sudoku Puzzles and Solutions*, https://www.kaggle.com/datasets/rohanrao/sudoku, 2019.

[5] Charles Akin-David et Richard Mantey, *Solving Sudokus with Neural Network*, https://cs230.stanford.edu/files_winter_2018/projects/6939771.pdf, 2018.

[6] Chings, *Artificial Intelligence Solves Sudoku : Convolutional Neural Networks (CNN)*, https://medium.com/analytics-vidhya/how-to-solve-sudoku-with-convolutionalneural-networks-cnn-c92be8830c52, 2021.

[7] Kyubyong, *Can Convolutional Neural Networks Crack Sudoku Puzzles ?*, https://github.com/Kyubyong/sudoku, 2021.

[8] Lyly123, *Solve Sudoku with CNN acc 97%*, https://www.kaggle.com/code/lyly123/solve-sudoku-with-cnn-acc-97, 2022.

[9] Plateforme colaborative, *Google collab*, https://colab.research.google.com/.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren et Jian Sun, *Deep Residual Learning for Image Recognition*, https://arxiv.org/pdf/1512.03385.pdf, 2015.

[11] Pierre Barthélémy, *17 est le nombre de Dieu au sudoku*, https://www.lemonde.fr/passeurdesciences/article/2012/01/08/17-est-le-nombre-de-dieu-au-sudoku_5986165_5470970.html, 2012.

[12] Ibrahem Kandel, *The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset*, https://www.sciencedirect.com/science/article/pii/S2405959519303455, 2020.

[13] Kyubyong, dataset *1 million Sudoku games*, https://www.kaggle.com/datasets/bryanpark/sudoku, 2016.

[14] franzl34, image sur la page de garde : Pixabay, https://pixabay.com/fr.