Rapport de Soutenance 1

Mattéo Baussart
Lucas Duport
Julie Blassiau
Matthieu Correia

EPITA

2022 Projet OCR

Table des matières

1 Plan de soutenance

Introduction

Démonstration

- Réseau de neurones : Entraînement & Prédiction
- Prétraitement de l'Image
- Traitement de l'Image
- Solver

Conclusion

- Éléments terminés
- Éléments commencés
- Éléments à commencer

2 Présentation

2.1 Mattéo

Depuis que j'ai pour la première fois utilisé un ordinateur, je m'intéresse au fonctionnement des logiciels. D'un autre côté, les maths m'ont toujours paru fascinantes. Donc lorsque j'ai découvert le langage Python, la cryptographie est vite devenue une passion. C'est de cette manière que je me suis plongé dans la programmation. Puis, en commençant à plus m'intéresser aux jeux vidéo, je me suis lancé dans la création d'un petit jeu vidéo avec un ami. Ce jeu n'a jamais été fini, mais l'expérience a été très enrichissante. A côté, j'ai eu plusieurs occasions de créer des petits jeux vidéo simples, pour le lycée ou en tant que projets personnels. Ces expériences m'ont beaucoup plu et c'est pourquoi, par ce projet, je souhaite renouveler le plaisir et la satisfaction que m'ont apportés ces expériences de programmation, mais aussi de travail de groupe.

2.2 Lucas

J'ai toujours eu cette attirance vers les nouvelles technologies en général, cela a commencé avec la célèbre Nintendo DS et depuis cela n'a cessé. À peine entré au collège, mon passe-temps était de me former et de créer un petit blog quand je rentrais de classe. C'est cette passion qui m'a tirée vers un autre lycée, qui lui proposait la spécialité Numérique et Sciences de l'Informatique, un choix décisif pour mon orientation. Enfin dans la continuité logique de mon parcours, me voici dans le supérieur dans un endroit qui m'ouvre les portes de la science et de la technologie. Un projet de groupe de cette ampleur correspond parfaitement à mes attentes de l'enseignement supérieur : tant il est complet et élaboré, tant il requiert de la discipline et de l'autonomie. Après le projet mené au semestre précédent, il me semble intéressant d'avoir un sujet plus cadré car cela correspond certainement plus aux types de projets qui seront menés en entreprise, car il présente plus de contraintes et d'étapes précises. De plus, le sujet est intéressant surtout que l'OCR possède un nombre applications infini.

2.3 Julie

Contrairement à la majorité des élèves de l'EPITA, je n'ai commencé à m'intéresser à l'informatique que tardivement. Les jeux vidéos ne me passionnaient en rien et j'avais des a priori sur ce domaine. Mais en terminale, j'ai commencé à coder en rentrant chez moi et cela s'est révélé amusant. J'ai d'autant plus apprécié quand je me suis rendue compte que c'était étroitement lié aux mathématiques. Depuis ma première année à Epita, je prends peu à peu confiance. Le projet S2 ne s'était malheureusement pas passé comme je l'aurais souhaité, je suis donc ravie d'évoluer avec ce nouveau groupe sur ce projet très intéressant. En effet, il sort de ce qu'on a pu faire jusqu'ici, mélange un grand nombre de connaissances et nous en fait, pour ma part, apprendre un paquet ! C'est le premier projet concret que je réalise et plus il avance plus on se prend dedans...

2.4 Matthieu

Dès mon plus jeune âge, je me suis intéressé à la science et aux découvertes. Lorsque je rentrais de l'école je regardais des documentaires sur la chaîne *RMC Découverte* où passaient de nombreux thèmes allant de la faune et la flore aux grandes guerres mondiales mais également des sujets plus poussés comme le nucléaire, la physique quantique et ses condensats de Bose-Einstein par exemple ainsi que les voyages spatiaux supraluminiques grâce au théorique vaisseau à distorsion d'Alcubierre. La science me fascine car elle rend possible l'impossible avec assez d'efforts et de patience. Une vidéo que j'ai vu il y a quelques années nommée *Timelapse of the futur* m'a profondément impactée car je me suis rendu compte de l'infinité dans laquelle nous nous trouvons, mais que même elle possède une fin où même l'inexistence cessera d'exister. Cette considération est principalement ce qui m'a poussé dans mes études d'ingénieur car je veux participer aux avancées dans l'espoir de permettre à l'humanité d'échapper à ce destin funeste qui équivaudrait à n'avoir jamais existé. Pour nuancer cette pensée, une série nommée *The Expanse* qui relate de ce que la vie humaine pourrait devenir d'ici quelques centenaires après la démocratisation du voyage spatial me redonne de l'espoir et me rapporte à un idéal accessible.

3 Avancement & Répartition des tâches

<table>
<tr><th colspan="2"></th><th>Matthieu</th><th>Mattéo</th><th>Lucas</th><th>Julie</th></tr>
<tr><td rowspan="4">Réseau</td><td>Paramètres du Réseau de neurones</td><td rowspan="4">100%</td><td colspan="3" rowspan="4"></td></tr>
<tr><td>XOR Proof</td></tr>
<tr><td>Sauvegarde et chargement des données</td></tr>
<tr><td>Fonctionnement du Réseau de neurones</td></tr>
<tr><td rowspan="5">Prétraitement</td><td>Chargement de l'image</td><td rowspan="5"></td><td colspan="2" rowspan="4">100%</td><td rowspan="5"></td></tr>
<tr><td>Redimensionnement automatique</td></tr>
<tr><td>Suppression des couleurs</td></tr>
<tr><td>Rotation manuelle</td></tr>
<tr><td>Application des filtres</td><td colspan="2">80%</td></tr>
<tr><td rowspan="5">Traitement</td><td>Récupération des lignes</td><td rowspan="5"></td><td colspan="3" rowspan="3">95%</td></tr>
<tr><td>Sélection des segments</td></tr>
<tr><td>Construction du carré</td></tr>
<tr><td>Extraction de la grille</td><td colspan="3" rowspan="2">100%</td></tr>
<tr><td>Extraction des cellules</td></tr>
<tr><td rowspan="2">Solver</td><td>Solver</td><td colspan="3" rowspan="2"></td><td rowspan="2">100%</td></tr>
<tr><td>Traitement des fichiers</td></tr>
</table>

4 Réseau de Neurones - Matthieu

Qu'il soit biologique ou artificiel, l'intérêt d'un réseau de neurones provient de la façon dont des neurones sont reliés pour former un système complexe. Chaque neurone peut prendre des décisions simples basées sur des calculs mathématiques, et ensemble, de nombreux neurones peuvent ainsi analyser des problèmes complexes et fournir des réponses précises. Concrètement, un réseau de neurones apprend à accomplir une tâche en examinant des exemples étiquetés et en induisant leur sémantique. Un réseau de neurones est donc une boucle de rétroaction corrective, qui donne plus d'importance aux données qui permettent de faire des suppositions correctes et moins à celles qui mènent à des erreurs, conduisant à terme à une compréhension globale du problème.

4.1 Paramètres du réseau : Acquisition et initialisation

```
hiddenN = 2
toLoopTrain = 4
toLoopValidate = 4
epoch = 250
epochInterval = 500
l_rate = 0.1
l1Norm = 0.0
l2Norm = 0.0
cost_func = MSE
optimizer = false
track = true
StatsFile = stats.txt
NNName = XOR
toExceed = 99.0
```

Fichier de configuration NN.cfg

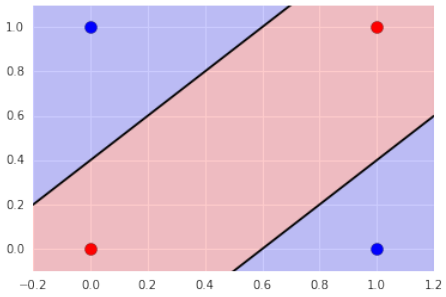
Afin de pouvoir entrainer notre réseau, il est nécessaire de renseigner les paramètres et hyper-paramètres sur lesquels se baser. Généralement, ces paramètres sont inscrits de façon définitive dans le code ou insérés en variables depuis la commande d'exécution du programme principal. Cependant, entrer ces paramètres manuellement lors de l'exécution peut être fastidieux et mener à des erreurs, et d'un autre côté inscrire ces paramètres statiquement manquerai de flexibilité et impliquerait une recompilation à chaque changement de paramètre, or cette manœuvre récurrente impacterai le temps de recherche des hyper-paramètres. L'option alternative a donc été de réunir ces paramètres dans un fichier de configuration (*.cfg) dont on renseigne le chemin lors de l'invocation du programme principal. Ce fichier contient la quasi-totalité des paramètres nécessaires pour la structure 'NNParam' qui permet d'acheminer les hyper-paramètres à travers le code, à savoir :

— **hiddenN** : Nombre de neurones dans la couche cachée
— **toLoopTrain** : Nombre d'exemples parmi lesquels entrainer le réseau
— **toLoopValidate** : Nombre d'exemples parmi lesquels tester le réseau
— **epoch** : Nombre d'époques par tentative d'entraînement
— **epochInterval** : Intervalle d'époques entre deux validation du réseau
— **l_rate** : Valeur de la vitesse d'apprentissage
— **l1Norm & l2Norm** : Valeurs pour les régularisations L1 et L2 (à éviter pour XOR)
— **cost_func** : Fonction d'erreur à utiliser lors de la rétropropagation du gradient
— **optimizer** : Utilisation ou non de l'optimisateur 'Adam' (à éviter pour XOR)
— **track & StatsFile** : Si 'track', enregistrement des valeurs d'erreur par époque dans le fichier 'StatsFile'
— **NNName** : Nom du réseau de neurones, utilisé lors de la sauvegarde
— **toExceed** : Précision du réseau à dépasser avant de sauvegarder

Ces paramètres sont parsés dans la fonctions 'main' et instanciés dans la structure 'NNParam' avant d'être passés au réseau.

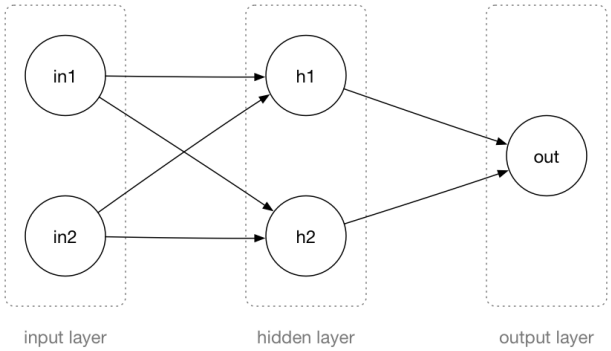
4.2 Proof Of Concept : XOR

L'opérateur OU EXCLUSIF, également connu comme XOR, est une combinaison des opérateurs AND, OR et NAND. Contrairement à ces deux opérateurs qui peuvent caractériser leur séparabilité linéaire en usant d'un hyperplan unique, l'opérateur XOR nécessite lui une combinaison de deux hyperplans obliques.



Hyperplans de séparation XOR

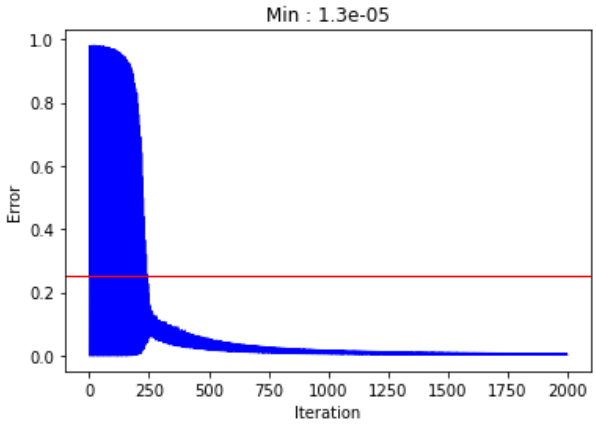
Afin de réaliser un réseau de neurones capable d'apprendre le comportement de l'opérateur XOR, il nous faut donc un minimum de deux neurones dans la couche cachée. Cet opérateur n'admet que deux sorties possibles (0 ou 1), il peut donc être considéré comme un problème de classification, cependant nous le considérons dans notre cas comme un problème de régression où la sortie est considérée comme positive si elle est supérieure ou égale à 0.5, ou nulle dans le cas inverse.



Architecture réseau XOR

L'architecture de ce réseau comporte une couche d'entrée à deux neurones (en accord avec l'arité de l'opérateur XOR), puis une couche cachée contenant deux neurones avec une fonction d'activation LeakyRelu et enfin une couche de sortie comportant un neurone unique avec une fonction d'activation Sigmoïde émettant une valeur entre 0 et 1. Il apparaît que les deux neurones de la couches cachée simulent respectivement les opérateur OR et NAND, tandis que celui de la couche de sortie équivaut à l'opérateur AND.

On remarquera que l'initialisation des poids du réseau joue un rôle primordial dans la convergence, c'est pourquoi nous utilisons la méthode d'initialisation de He afin de tenter de prévenir l'explosion ou la disparition de gradient ainsi que d'obtenir une convergence plus rapide. La ligne rouge sur le graphe indique la limite où le réseau a complètement appris l'opérateur et ne commet plus d'erreur selon la fonction d'erreur de la différence au carré (MSE).



Erreur par itération

Ce graphe est donc le résultat de l'apprentissage du réseau sur 500 époques (ce qui correspond à 2000 itérations) avec pour paramètres 0.1 pour le taux d'apprentissage ainsi que la fonction MSE pour le calcul de l'erreur et la rétropropagation du gradient. On peut donc observer sur ce graphe que notre réseau a pu converger dès la 250ème itération environ, soit près de 62 époques nécessaires. Cependant, il peut arriver que l'initialisation des poids prédispose le réseau à stagner sur un optimum local, auquel cas il n'y a plus d'espoir pour cette tentative et il est nécessaire de recommencer l'apprentissage.

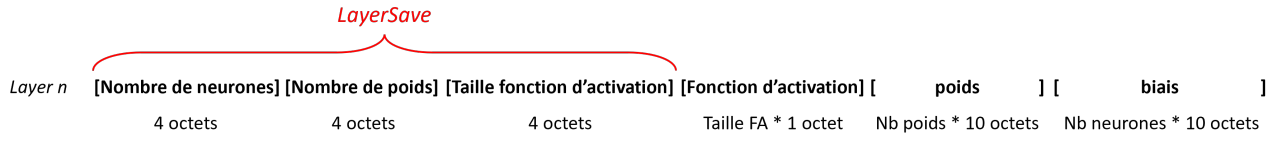
4.3 Données du réseau : Sauvegarde et chargement

Notre réseau de neurones contient de nombreux paramètres nécessaires à son bon fonctionnement. Actuellement, l'architecture choisie produit 6 poids, 3 biais et 2 fonctions d'activation, soit une dizaine de paramètres environ. Cependant, lorsque nous devrons adapter notre architecture pour la reconnaissance de chiffres il est prévu que ce nombre atteigne la vingtaine de milliers. Au vu de l'ampleur que peut prendre ce nombre, il est inconcevable de stocker ces données sous format textuel, c'est pourquoi nous choisirons plutôt le format binaire. La sauvegarde des paramètres se compose de plusieurs étapes dont la première consiste à recueillir le nom du réseau que l'on vient d'entraîner, soit en l'entrant manuellement lorsque demandé soit en le renseignant dans le fichier de configuration du réseau. Ce nom servira lors de la création du fichier de sauvegarde qui suit le format suivant :

NeuralNetData_**[1]**layers_**[2]**_**[3]**.dnn

Où **[1]** représente le nombre de couches du réseau (ex : 3), **[2]** le nom du réseau (ex : OCR), et **[3]** la performance du réseau (ex : 88.82 pour 88.82% de précision). Cette notation nous permet une meilleure lisibilité mais a également une utilité lors du chargement des données.
La prochaine étape consiste à faire migrer les paramètres importants d'une couche vers une structure 'LayerSave' plus épurée où se concentrent les informations caractérisant le réseau comme le nombre de neurones, le nombre de connexions avec la couche précédente ainsi que la taille du nom de la fonction d'activation. Ces informations sont ensuite passées au fichier binaire à l'aide de la fonction 'fwrite' qui nous permet de sérialiser la structure 'LayerSave' mentionnée plus tôt. La dernière étape consiste à écrire le nom de la fonction d'activation puis tous les poids et biais de la couche. Ce processus s'opère sur chaque couche pour permettre de sauvegarder un réseau de neurones.

Afin de charger le réseau de neurones en mémoire, il est nécessaire d'effectuer le processus inverse, à savoir commencer par déterminer le nombre de couches sauvegardées en scannant le nom du fichier avec la fonction 'sscanf' pour récupérer le paramètre **[1]** mentionné précédemment. Il faut ensuite désérialiser la structure 'LayerSave' qui va nous indiquer la taille du nom de la fonction d'activation ainsi que le nombre de poids et de biais, que l'on récupèrera à l'aide de la fonction 'fread', et réitérer ce procédé pour le nombre de couches attendu. Nous nommerons ce format *'dnn'* que nous pouvons représenter de manière suivante :



4.4 Fonctionnement général

Comme nous l'avons vu précédemment, le réseau de neurones se subdivise en deux parties majeures, l'une étant la prédiction tandis que l'autre se concentre sur l'entraînement. Afin de créer un réseau de neurones, il est nécessaire de déclarer une structure 'NNParam' qui contiendra la majorité des paramètres et hyper-paramètres, ainsi qu'une structure 'Network' qui supportera l'ossature du réseau. Cette structure est ainsi composée de plusieurs structures 'Layer' symbolisant les différentes couches (entrée, sortie et cachées), que l'on incorpore au réseau lors de l'initialisation. Actuellement, la conception de l'architecture du réseau est statique et ne contient qu'une couche cachée (bien que l'on puisse changer dynamiquement le nombre de neurones) mais il est prévu de développer ce système pour que l'on puisse renseigner dans le fichier de configuration (*.cfg) le nombre de couches cachées ainsi que leurs spécificités (nombre de neurones et fonctions d'activation). Lors de l'ajout des couches cachées et de sortie, il est nécessaire de choisir une fonction d'activation parmi celles implémentées, à savoir :

- None
- Sigmoïd
- Softmax
- Relu
- Selu
- Step
- Argmax
- LeakyRelu

Une fois l'architecture du réseau instanciée, il est nécessaire de charger les exemples pour l'apprentissage dans les tableaux à doubles dimensions 'inputTrain' et 'outputTrain' de la structure 'NNParam' afin de pouvoir plus tard utiliser ces données pour l'apprentissage, et similairement dans les tableaux 'inputTest' et 'outputTest' qui contiennent les même types de données mais qui eux serviront lors de la validation du modèle sur des données étrangères. Lors de leur initialisation, toutes les couches autre que celle d'entrée se voient attribuées des biais initialisés à 0 ainsi que des poids aléatoires selon une distribution gaussienne de moyenne 0 et d'écart-type 1, cependant pour les couches possédant les fonctions d'activation 'Relu' ou 'LeakyRelu' on préférera un écart-type de $\sqrt{\frac{2}{in}}$ (où 'in' représente le nombre de neurones de la couche précédente) en accord avec la méthode d'initialisation de He. Une fois le réseau initialisé, il est désormais possible de fournir un vecteur à la couche d'entrée qui va déclencher la propagation à travers les couches du réseau pour nous permettre de récupérer un vecteur en sortie, qui lors de la phase d'apprentissage va être comparée au vecteur témoin qui contient les valeurs que l'on attendait pour cette passe. En usant d'une des fonctions d'erreur implémentées (MAE | MSE | RMSE | CrossEntropy), nous pouvons déterminer la valeur globale de l'erreur (qu'il est possible d'enregistrer pour chaque époque dans un fichier 'StatsFile'), puis lors de la rétropropagation du gradient nous déterminons l'importance de chaque biais et poids qui ont mené à cette valeur et effectuons les changements nécessaires afin de faire diminuer l'erreur de chacun de ces composants pour les prochaines prédictions. Nous utilisons donc la méthode de descente de gradient en ligne, qui à l'inverse des versions en 'Batch' ne traite qu'un exemple d'apprentissage à la fois. Cette correction de l'erreur en fonction de l'importance dans le résultat final est le méchanisme qui est au cœur du phénomène d'apprentissage et donc du fonctionnement du réseau de neurones. Pour effectuer une prédiction, le comportement est le même jusqu'à l'obtention du vecteur de sortie, mais au lieu d'effectuer une rétropropagation de l'erreur, nous appliquons soit une fonction 'Step' si la valeur est unique, soit une fonction 'Argmax' afin de normaliser la réponse avec uniquement des 0 et un 1.

5 Prétraitement de l'Image - Mattéo | Lucas

5.1 Chargement de l'image

Pour ce faire, nous chargeons une image à l'aide de la librairie SDL2 image. C'est à l'aide de cette librairie que nous manipulons les images puisqu'elle possède déjà un certain nombre d'outils pour nous aider (ex : récupérer les pixels d'une image). Tout d'abord, à l'aide du nom du fichier de l'image, nous ouvrons l'image avec la SDL pour récupérer une surface. Une surface est une structure qui va grandement nous aider à manipuler les images puisqu'elle contient tout ce dont nous avons besoin : un format, une longueur, une largeur, un pitch et surtout un pointeur sur la liste de pixels que contient l'image. Pour créer cette surface à partir d'une image, nous pouvons spécifier un format. Dans notre cas, nous avons choisi le format RGB888 (comme utilisé dans le TP n°6), ce format est utilisé car un octet réservé à la valeur alpha ne nous intéresse pas. Pour s'épargner les structures SDL à chaque ligne, nous avons créé notre propre structure "Image" qui possède seulement les attributs qui nous intéressent. Nous y trouvons les dimensions de l'image (largeur, hauteur), ainsi qu'un tableau de taille hauteur × largeur contenant la valeur des pixels. Ces valeurs étant comprises entre 0 et 255, le type utilisé pour les stocker est unsigned char. Cette structure nous permet de parcourir les pixels plus librement dans le suite du projet car cela nous évite d'utiliser la SDL pour chaque traitement, comme la suppression des couleurs.

5.2 Redimensionnement automatique de l'image

Nous calculons un ratio entre la taille réelle de l'image et la taille maximum que nous souhaitons gérer. Ensuite nous ajustons soit la hauteur, soit la largeur de l'image en fonction du ratio le plus important. Si la largeur dépasse le maximum autorisé, elle est ramenée à ce maximum, et la hauteur est adaptée en conséquence pour conserver les proportions. Cette fonctionnalité a pour but de toujours traiter des images de taille correcte sans pour autant mettre en péril l'échelle de l'image car cela peut avoir un effet néfaste sur son traitement.

$$x = \frac{new\_x}{ratio\_x} \qquad\qquad y = \frac{new\_y}{ratio\_x}$$

5.3 Suppression des couleurs

Pour chaque pixel que contient l'image, nous appliquons un traitement qui consiste à faire une moyenne sur le rouge, le vert et le bleu du pixel, car les nuances de gris s'obtiennent lorsque la valeur rouge du pixel est la même que la bleue et la verte. Le « niveau de gris » dépend directement de la valeur de cette moyenne. Par exemple nous aurons un gris foncé avec rouge = vert = bleu = 50 ou un gris clair avec rouge = vert = bleu = 220. Nous calculons la moyenne entre les valeurs r v b du pixel, et nous mettons toutes ses valeurs à cette moyenne.

$$gris = \frac{rouge + vert + bleu}{3}$$

Nous avons finalement une Image (en nuance de gris), prête à l'emploi.

5.4 Rotation manuelle

La rotation manuelle est essentielle pour plusieurs raisons. D'abord, si l'utilisateur a la possibilité de redresser son image à la main, alors notre algorithme fonctionnera plus rapidement (surtout si nous faisons confiance à l'utilisateur et que donc nous n'appliquons pas de rotation automatique par dessus). De plus, une fois la rotation manuelle implémentée, dès que la grille est détectée, nous pouvons tourner l'image automatiquement grâce à l'angle de la grille sur l'image. Les bords de la grille s'alignent alors automatiquement avec les bords de l'image.

Au début, nous avons cherché plusieurs façons de tourner l'image avec la SDL. Nous avions deux principales possibilités, la première : utiliser des outils de la librairie SDL, la seconde appliquer la rotation directement sur la liste des pixels de l'image, en passant par les différentes structures de la librairie pour pouvoir utiliser les fonctions qui nous intéressent. Dans un premier temps, nous avons essayé en passant par la SDL. Nous sommes arrivés à nos fins en utilisant des surfaces pour manipuler les images, or il a fallu s'arranger avec les différentes structures SDL pour pouvoir exécuter les fonctions qui nous intéressent car celles-ci imposent certains types. En suivant ces étapes fastidieuses, nous avons finalement une surface contenant notre image d'origine tournée.

Cependant cette méthode présente plusieurs désavantages :

— Nous utilisons un maximum d'outils prédéfinis et, même si la documentation SDL est bien comprise, il se peut que la manipulation des différentes structures provoque finalement une erreur pour diverses raisons.

— Ce n'est pas du tout optimal, car nous devons créer une fenêtre qui sera masquée pour l'utilisateur pour pouvoir procéder au rendu de notre image, mais ce rendu n'est même pas visible pour l'utilisateur...

Même si l'autre méthode fonctionnait, nous avons fini par implémenter la rotation de l'image directement à partir des pixels pour avoir une rotation « propre ».
Il nous faut d'abord calculer la taille de la nouvelle image après rotation, qui dépend de la taille de l'image de l'origine ainsi que de l'angle de rotation. Nous utilisons les relations trigonométriques pour calculer la taille de la nouvelle image (new_w, new_h) à partir de la taille de l'ancienne image (w, h) et l'angle de rotation alpha.

$$new\_w = |h * sin(\alpha)| + |w * cos(\alpha)|$$

$$new\_h = |w * sin(\alpha)| + |h * cos(\alpha)|$$

Ensuite, par des relations trigonométriques similaires, il faut retrouver l'emplacement du pixel source à partir de la destination. On a la relation :

$$x = \left(new\_x - \frac{new\_w}{2}\right) * cos(\theta) - \left(new\_y - \frac{new\_h}{2}\right) * sin(\theta) + \frac{w}{2}$$

$$y = \left(new\_x - \frac{new\_w}{2}\right) * sin(\theta) + \left(new\_y - \frac{new\_h}{2}\right) * cos(\theta) + \frac{h}{2}$$

new_x et new_y étant les coordonnées sur la nouvelle image, on calcule leur position initiale et on leur ôte le centre de rotation. X et y peuvent être décimaux, leur partie décimale sera utilisée comme un pourcentage d'appartenance aux pixels voisins et cela aura une incidence sur la valeur du pixel en (x,y).

Par exemple, si x = 7.2 et y = 1.6, on considérera qu'horizontalement, le pixel est constitué à 80% du x = 7 et 20% du x = 8, et que verticalement il est constitué à 40% du y = 1 et 60% du y = 2. On utilisera ces pourcentages pour calculer les poids des 4 pixels avoisinants. Dans notre exemple :

$$weight(7,1) = 0.8 * 0.4 = 0.32$$

$$weight(8,1) = 0.2 * 0.4 = 0.08$$

$$weight(7,2) = 0.8 * 0.6 = 0.48$$

$$weight(8,2) = 0.2 * 0.6 = 0.12$$

En multipliant ces poids par les valeurs de leur pixel respectif, et en sommant ces produits, on obtient donc la valeur de new_pixels(new_x, new_y). Ce calcul permet de réduire la perte d'information lors des rotations et d'avoir un rendu plus naturel. Cette pondération est utilisée pour les autres transformations qui donnent des coordonnées non-entières, comme le
redimensionnement et l'extraction de grille, ce qui nous permet d'obtenir une image tournée à l'angle souhaité par l'utilisateur.

Nous avons ajouté la possibilité de faire la rotation manuelle avec une interface. Lorsque le programme se lance avec une image voulue, une fenêtre s'ouvre affichant l'image. Ensuite, en utilisant les flèches (gauche/droites) ou les lettres (q/d), il est possible de pré-visualiser la rotation de l'image pour un certain angle. En appuyant sur la touche "s", une image nommée "rotated.png" est sauvegardée dans le dossier actuel. En quittant la fenêtre, la rotation finale de l'image est conservée pour les prochaines étapes.

5.5 Application des filtres

5.5.1 Filtre gaussien

Nous appliquons le filtre gaussien sur un pixel en prenant en considération une zone de 5 pixels voisins. Nous appliquons donc la matrice gaussienne en 5x5 ci-contre sur les pixels de l'image :

$$\begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix}$$

Ainsi nous obtenons une nouvelle image avec du bruit en moins.

5.5.2 Saturation

La saturation consiste à choisir un seuil. Ensuite, tous les pixels dont la valeur est inférieure à ce seuil sont mis à 0, les autres à 255. Le seuil est choisi de telle sorte que 12.5% des pixels soient plus sombres que ce seuil. Cette valeur à été trouvée après plusieurs essais. Après cette saturation binaire, le résultat attendu est un fond blanc avec des lignes noires représentant les différents éléments présents sur cette image. Après ceci, il suffit d'inverser les couleurs pour avoir des valeurs élevées correspondant aux lignes que l'on souhaite détecter. Néanmoins, son point faible est son manque de flexibilité. Pour une image donnée, un seuil de 12.5% pourrait être trop bas ou trop élevé. Aussi, ce procédé est totalement inefficace dans le cas où la grille est plus claire que le fond.

5.5.3 Edge detection

Pour ces raisons, nous prévoyons par la suite de remplacer cette transformation par une autre conçue spécialement pour cela. Cette transformation est la edge detection. Elle s'effectue en appliquant une matrice dite "canny" à l'image. Après son utilisation, on retrouve des valeurs élevées lorsqu'il y a des changements brusques d'intensité. Autrement dit, on retrouve des valeurs élevées aux bordures des lignes, ce qui ce que nous voulons.

6 Traitement de l'Image - Lucas | Julie | Mattéo

6.1 Récupération des lignes

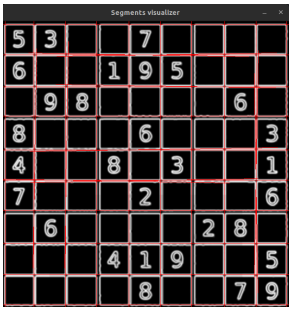
Pour la récupération des lignes, nous avons utilisé l'algorithme de Hough. En effet, Hough caractérise une droite par sa pente et son ordonnée à l'origine. Mais dans notre cas, il est plus avantageux de la caractériser par des coordonnées polaires (ρ et θ). ρ étant la distance entre la droite et l'origine du repère et θ est l'angle que fait la perpendiculaire de la droite avec l'axe x. D'après ces paramètres, on peut en déduire l'équation cartésienne de la droite :

$$\rho = x * cos(\theta) + y * sin(\theta)$$

Pour synthétiser, on initialise un tableau à deux dimensions avec en entrée ρ et θ. ρ varie entre 0 et la longueur de la diagonale (soit ρ_max) et θ entre 0 et 359. La manière de remplir le tableau est la suivante : pour chaque valeur de ρ et de θ, on calcule la valeur moyenne des pixels se trouvant sur la ligne correspondante. Pour autant, on ne parcourt pas les pixels de n'importe quelle manière, on parcourt les y et on récupère les x quand les droites sont verticales, et inversement lorsque les droites sont horizontales :

$$x = \frac{\rho - y * sin(\theta)}{cos(\theta)}$$

$$y = \frac{\rho - x * cos(\theta)}{sin(\theta)}$$



Visualisation des lignes

Cela permet d'éviter de parcourir l'entièreté des pixels de l'image pour la droite. Cette disjonction de cas permet également d'éviter les divisions par 0 et les grands sauts de pixels. On stocke donc la valeur moyenne trouvée dans notre tableau.

6.2 Sélection des segments

Nous prenons la valeur moyenne des pixels de la droite comme seuil. Et ensuite, on comble les trous qui sont en dessous d'une certaine taille (arbitraire), et nous conservons seulement le plus grand segment sur cette droite-ci. Cela sert à prendre en compte même des segments qui ne seraient pas entièrement continus en termes de pixels mais qui paraîtraient continus à l'œil nu.

6.3 Construction du carré

6.3.1 Le mode opératoire

Nous avons pensé à construire le carré petit à petit, en mettant au fur et à mesure des segments bout-à-bout si nous leur trouvions des cohérences pour construire un carré. Au cours de cette recherche, différents filtres sont appliqués pour juger de la cohérence des potentiels autres côtés du carré.

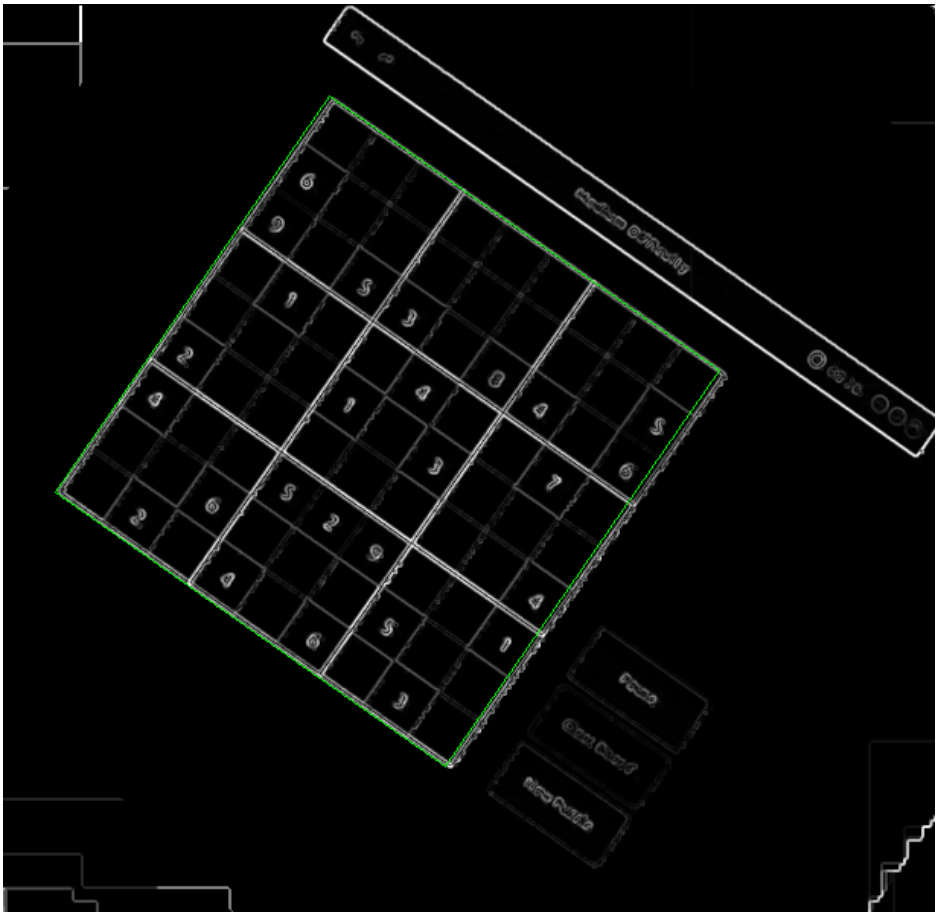
6.3.2 Nos filtres

Nous filtrons par rapport :

- Aux angles : nous construisons un carré, cela implique que le côté suivant du carré doit avoir un angle d'environ 90° par rapport au segment. (Une marge d'erreur est accordée.)
- À la longueur : les propriétés géométriques d'un carré impliquent que ses côtes doivent être de même longueur. Ainsi les potentiels autres côtés qui ne font pas la même taille que le premier segment sont éliminés. (Une marge d'erreur est accordée.)
- À la position : l'extrémité du côté actuel doit se trouver à proximité de l'extrémité du potentiel prochain côté. Nous éliminons donc les segments qui ont leur extrémités trop éloignés du côté actuel. Mais pour ne pas éliminer un segment voisin dont nous regarderions la mauvaise extrémité, il nous est possible de changer de sens un segment. (Une marge d'erreur est accordée.)

6.3.3 La finition

C'est ainsi que nous trouvons les quatre segments qui sont les différents côtés du carré. Ensuite, nous extrapolons ces segments pour être certain de récupérer précisément les coins du carré et non des segments dont les extrémités ne seraient pas exactement les coins du carré. (Cette différence serait dûe à la marge d'erreur accordée sur la position). Les marges d'erreur ont été choisies arbitrairement de manière à ce que les résultats soient les plus précis possibles en fonction des différents cas testés.



Visualisation du carré final

6.4 Extraction de la grille

À partir des coordonnées des quatre coins de la grille, et des dimensions voulues, nous calculons les coefficients d'un matrice qui servira à la transformation. Cette matrice est faite de telle sorte que chaque coin de la grille devienne un coin du carré dans lequel nous souhaitons récupérer la grille. Les points à l'intérieur de la grille sont ainsi transformés pour conserver les proportions originales. L'intérêt de cette méthode est qu'elle permet de transformer n'importe quel quadrilatère convexe en un carré. De cette manière, la déformation éventuelle causée par la perspective de prise de photo est corrigée.

6.5 Extraction des cellules

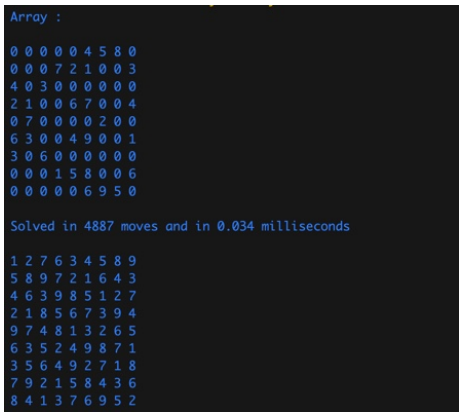
Le découpage de l'image est essentiel, notamment pour avoir chaque cellule sauvegardée dans un fichier différent, pour pouvoir les passer dans le réseau de neurones. Comme pour la rotation d'image, nous avons commencé par créer cette fonctionnalité en utilisant pleinement les possibilités de la librairie SDL. Cependant pour les mêmes raisons que la rotation, cela n'était ni intéressant à concevoir ni optimal. Pour découper l'image en sous-images, nous avons besoin de l'image à découper, d'un point (x,y), et de la longueur du côté de la sous-image que nous souhaitons. Nous partons du principe que les sous-images sont carrées. La sous-image est créée et ses pixels sont les pixels de l'image d'origine à partir du point (x,y) jusqu'au point (x+longueur, y+longueur). Bien sûr, à partir de notre structure Image, il nous est aisé de sauvegarder celle-ci dans un format PNG par exemple, ce qui nous sera utile pour alimenter le réseau de neurones. À présent, nous avons en notre possession un outil permettant de découper une image en sous-image simplement à l'aide des coordonnées et de la taille de celle-ci.

7 Solver - Julie

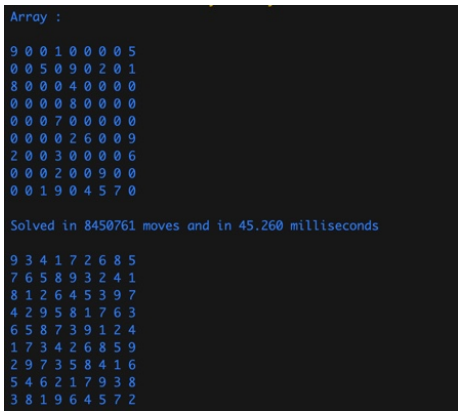
7.1 Solver

La résolution de sudoku peut paraître facile et en effet, quand elle est faite de manière brute elle s'implémente rapidement. Mais il existe des manières plus optimisées de la résoudre.
Pour la première version implémentée, nous avons simplement utilisé le back-tracking de manière classique. On émet des hypothèses sur notre liste de cellules vides. Tant que les hypothèses sont possibles, on avance sur les autres cases. Si l'hypothèse se révèle fausse, on passe à la valeur suivante. Si toutes les valeurs ont été testées mais qu'aucune n'est valable, alors on doit retourner sur l'ancienne hypothèse et passer à la valeur suivante sur cette case et ainsi de suite jusqu'à ce que toutes les hypothèses se révèlent correctes. Une hypothèse est considérée comme valable si sur toute la ligne, la colonne et le bloc de la case ne contienne pas la valeur testée. Pour ce faire, trois tableaux de booléens sont initialisés à False par défaut et lorsqu'il y a une valeur sur une case, à True. A la fin du programme, toutes les valeurs des tableaux sont à True, sauf si bien sûr la grille était initialement erronée. Dans le cas où la grille à plusieurs possibilités de solutions, le programme retournera la combinaison des premières hypothèses qu'il a trouvé comme valable. Nous avons testé ce programme sur deux grilles différentes.

Cette première grille est assez favorable, un grand nombre de cases sont déjà remplies, ce qui rend un nombre de coups (d'hypothèses émises) plus faibles que ce à quoi on peut s'attendre pour une méthode brute. Le temps d'exécution est quant à lui faible aussi.



V1 grille favorable



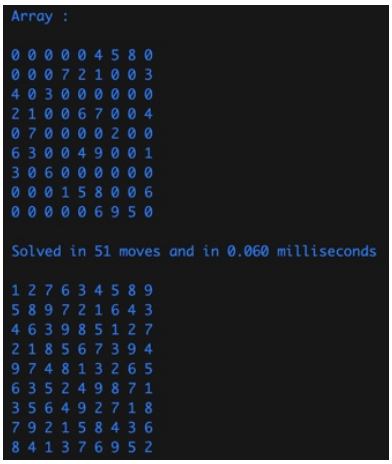
V1 grille défavorable

Cette deuxième grille est en revanche moins favorable. En effet, moins de cases sont remplies, cela laisse donc plus d'hypothèses à émettre et sur lesquelles se tromper. Le nombre de coups est dans ce cas-là bien plus élevé tout comme le temps d'exécution. Implémenter une version plus optimisée de la résolution de la grille est donc utile pour éviter un nombre trop grand de coups et rendre le temps d'exécution plus optimal.

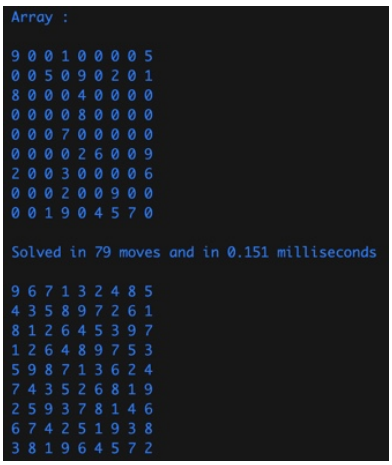
La complexité de cet algorithme est $O(9^n)$ où n est le nombre de cases vides - 9 solutions sont effectivement possibles à chaque case vide. Nous avons donc cherché à réduire n par des optimisations successives qui ressemblent à ce que chacun peut faire naturellement en résolvant une grille de sudoku. Réduire le nombre d'essais revient à trouver les cases dont on peut être sûr. À défaut, avec le moins de possibilités de valeurs possibles. C'est ce que l'on fait à chaque "coup" dans l'algorithme de résolution (back-tracking).

Une première manière d'optimiser est de trouver les cases évidentes. Pour une case donnée, on regarde si une et une seule valeur est possible en considérant la ligne, la colonne et le bloc de la case. Si on trouve une telle case, c'est celle par laquelle on va continuer le back-tracking. En effet, cette valeur considérée comme évidente permet d'émettre moins d'hypothèses incertaines.
La deuxième optimisation que nous avons rajouté a été de trouver les valeurs induites par les autres cases. Il s'agit, pour une ligne, une colonne, ou un bloc donné, de regarder si une valeur est nécessairement sur une case car absente des autres. Si on trouve une telle case, c'est celle par laquelle on va continuer l'algorithme de back-tracking. Comme pour la première optimisation, cela réduit le nombre d'hypothèses. En revanche, dans le back-tracking, on va prioriser une case évidente sur une case induite. S'il y a une case évidente, c'est avec elle qu'on continue le back-tracking.
La dernière optimisation a été de gérer les cases évidentes ou induites dans la pile des appels. On gère une pile des cases avec leur valeur hypothétique pour pouvoir remonter en cas d'erreur. Lorsque l'on émet une hypothèse sur une case ni évidente, ni induite, on l'inscrit dans la pile et on continue l'algorithme en cherchant des cases évidentes ou induites. Dans le cas où il n'y a plus d'hypothèse possible (on a essayé les valeurs de 1 à 9), on remonte dans la pile jusqu'à l'hypothèse d'avant, en sautant les cases évidentes ou induites pour en essayer une autre. En effet, comme les cases évidentes et induites ont des valeurs dont on est sûr, c'est forcément dans l'hypothèse retenue dans la pile qui est fausse.
Cette dernière optimisation est la meilleure que l'on ait trouvée et a un impact direct sur le nombre de coups et le temps mis pour résoudre une grille.

Pour cette première grille, on a une réduction de coups. Pour autant, toutes les optimisations ne sont pas nécessaires car elle se résout seulement avec des cases évidentes. En revanche le temps augmente (même s'il reste minime) car les calculs et les parcours de grille prennent du temps. Dans le cas d'une grille favorable, on aurait donc pu s'arrêter à la première optimisation.



V4 grille favorable



V4 grille défavorable

En revanche pour cette grille, la différence est flagrante. On passe de plus de 8 millions de coups à seulement 79. Idem pour le temps qui réduit beaucoup. Dans le cas d'une grille défavorable, la dernière optimisation est un réel gain de coups et de temps.

7.2 Traitement des fichiers

La grille que nous traitons dans l'algorithme du solver est sous forme de matrice. Or nous recevons la grille sous forme de fichier. Il a donc fallu être capable de lire un fichier (sans toute la partie traitement d'image) et le convertir sous forme de matrice. Puis, une fois résolue, repasser la matrice sous la forme du fichier souhaité. Pour ce faire, nous avons simplement utilisé de la lecture de fichier avec certaines conditions sur les retours à la ligne et les sauts de ligne. À chaque caractère lu, si c'est un chiffre (ou un point représentant le 0), il est ajouté à notre matrice. À la fin de la lecture du fichier, on le ferme et notre matrice est prête à être résolue ! De la même manière, à la fin de la résolution de la grille, chaque valeur est mise dans un nouveau fichier sous la même forme que le premier. Ce ne fut pas la partie la plus difficile du solver, seuls quelques petits bugs ont été résolus rapidement à deux surtout sur des problèmes de type et les conditions des retours à la ligne.

8 Hiérarchie du projet

```
OCR
├── .clang-format
├── rapport_1.pdf
├── AUTHORS
├── README
├── NeuralNetwork
│   ├── TrainedNetwork
│   │   ├── NN.cfg
│   │   ├── NeuralNetData_3layers_XOR_100.0.dnn
│   │   └── NeuralNetData_3layers_OCR_88.82.dnn
│   ├── Activations.c
│   ├── Activations.h
│   ├── Cost.c
│   ├── Cost.h
│   ├── IOHelper.c
│   ├── IOHelper.h
│   ├── Layer.c
│   ├── Layer.h
│   ├── main.c
│   ├── Makefile
│   ├── Network.c
│   ├── Network.h
│   ├── Rand.c
│   ├── Rand.h
│   ├── Tools.c
│   └── Tools.h
├── ImageProcessing
│   ├── Images
│   │   ├── image_01.jpeg
│   │   └── image_06.jpeg
│   ├── Makefile
│   ├── display.c
│   ├── display.h
│   ├── hough.c
│   ├── hough.h
│   ├── matrices.c
│   ├── matrices.h
│   ├── main.c
│   ├── openImage.c
│   ├── openImage.h
│   ├── tools.c
│   ├── tools.h
│   ├── transformImage.c
│   └── transformImage.h
└── Solver
    ├── Makefile
    ├── main.c
    ├── solver.c
    ├── solver.h
    └── grid.00
```

9 Bibliographie

9.1 Réseau de neurones

— [0,1] scaling dramatically increase training time

— Back-Propagation is very simple. Who made it Complicated ?

— Bias Update in Neural Network Backpropagation

— What is numerical stability ?

— Activation Functions in Neural Networks

— Weight Initialization in Neural Networks

— What should I do when my neural network doesn't learn?

— Debugging: Gradient Checking

— Implementing the XOR Gate using Backpropagation in Neural Networks

— How Neural Networks Solve the XOR Problem

— XOR problem with neural networks

9.2 Prétraitement de l'Image

— L'image considérée comme un signal

— How to rotate image by using pixel by pixel

— Basic image manipulation

— Documentation de la SDL

— Explication des formats SDL

9.3 Traitement de l'Image

— Computing a projective transformation

— Hough transform

9.4 Solver

— Lecture fichier en C

— Principe du back-tracking