



Faculté
des Sciences

Département des Sciences Informatiques

UMONS
Université de Mons

Apprentissage automatique du jeu du démineur

Projet réalisé par Nico SALAMONE
pour l'obtention du diplôme de Master en sciences informatiques

Année académique 2017–2018

Directeur: Dr. Alexandre DECAN

Service: Service de Génie Logiciel

Rapporteurs: Dr. Hadrien MÉLOT

Dr. Tom MENS

Remerciements

Je tiens à remercier mon directeur de projet Dr. Alexandre DECAN pour son aide et ses conseils, notamment en ce qui concerne les réseaux de neurones artificiels et le langage de programmation Python.

Table des matières

1	Introduction	3
2	Mise en contexte	4
2.1	Jeu du démineur	4
2.2	Apprentissage automatique et notions connexes	5
2.3	Réseaux de neurones artificiels	6
3	Réalisation d'une intelligence artificielle pour le jeu du démineur	10
3.1	Contexte technique	10
3.2	Réseau de neurones	11
3.2.1	Hypothèse et objectif	11
3.2.2	Fonctionnement du réseau	13
3.2.3	Génération d'un jeu de données et un d'un jeu d'entraînement	15
3.2.4	Évaluation des résultats du réseau	20
3.3	Intelligence artificielle	21
3.3.1	Hypothèses et objectif	22
3.3.2	Fonctionnement de l'intelligence artificielle	22
3.3.3	Implémentation de l'intelligence artificielle	23
3.3.4	Évaluation des résultats de l'intelligence artificielle	24
3.4	Résultats obtenus	24
4	Amélioration de l'intelligence artificielle	28
4.1	Ratio de sous-grilles où la case centrale est une case bombe	28
4.2	Ordre des sous-grilles dans le jeu d'entraînement	29
4.3	Nombre de sous-grilles contenues dans le jeu d'entraînement	30
4.4	Paramètres de l'entraînement du réseau de neurones	33
4.5	Nouvelle structure du réseau de neurones	37
4.6	Utilisation de jeux de données sans doublons	39
4.7	Intelligence artificielle utilisant des drapeaux	39
4.8	Modification de l'intelligence artificielle avec drapeaux	42
5	Performances de l'intelligence artificielle après suppression de certaines hypothèses	46
6	Conclusion	48
	Références	51

1 Introduction

Nous entendons de plus en plus parler de l'apprentissage automatique. Pourquoi un tel engouement pour cette discipline ? L'apprentissage automatique permet d'aborder certains problèmes, difficilement réalisables par des algorithmes classiques, plus facilement. Nous pouvons prendre l'exemple des voitures autonomes. Imaginez un instant que vous devez réaliser une intelligence artificielle sans apprentissage automatique pour ce problème. Comment vous y prendriez-vous ? En envisageant tous les paramètres et toutes les situations possibles et imaginables ? Cela deviendrait très vite difficile et fastidieux. C'est à ce moment précis que l'apprentissage automatique tire toute son utilité : elle permet à une machine d'apprendre par elle-même tous les paramètres, ou du moins les plus « influençants ». De ce fait, il n'est donc plus nécessaire de devoir les coder « en dur » [3]. Une implémentation des voitures autonomes est le projet Waymo de Google¹.

La méthode d'apprentissage automatique qui sera explorée dans ce rapport est les réseaux de neurones artificiels. L'objectif est de réaliser une intelligence artificielle, basée sur cette méthode, pour le jeu du démineur et de comparer comment la structure et les paramètres d'un réseau de neurones peuvent influencer sur la qualité des prédictions faites par celui-ci. Dans notre cas, l'apprentissage utilisé sera de type supervisé, c'est-à-dire que le réseau devra apprendre sur base d'un ensemble d'exemples (ou encore jeu d'entraînement) constitué de paires entrée-sortie [12]. Une approche non supervisée, où l'apprentissage du réseau se fait à l'aide d'entrées uniquement [5], n'est pas adaptée à notre problème ; celle-ci est en effet utilisée pour extraire des informations utiles à partir de données.

Pour résoudre ce problème, nous construirons tout d'abord une première intelligence artificielle utilisant un réseau de neurones artificiels. Pour ce faire, nous :

- décrirons le fonctionnement du réseau et la façon dont il a été construit et entraîné grâce à un jeu d'entraînement ;
- détaillerons l'objectif de l'intelligence artificielle et la manière dont elle opère ;
- évaluerons le réseau et l'intelligence artificielle avec des mesures (ou indicateurs) dont certaines seront conçues par nos soins ; cela nous permettra de mieux connaître leur performance.

Nous améliorons ensuite l'intelligence artificielle en faisant varier les différents paramètres qui entrent en jeu : la structure du réseau, le jeu d'entraînement, *etc*. Et enfin, nous supprimerons certaines hypothèses faites et regardons comment l'intelligence artificielle s'en sort malgré tout.

1. <https://www.google.com/selfdrivingcar/>.

2 Mise en contexte

Cette section présente les notions nécessaires à la compréhension de la suite de ce rapport. Nous parlerons tout d'abord du jeu du démineur. Plus précisément, nous verrons l'objectif de ce jeu, les règles et le déroulement d'une partie. Ensuite, nous définirons l'apprentissage automatique ainsi que quelques notions liées. Nous aborderons enfin les réseaux de neurones artificiels. Nous expliquerons leur fonctionnement ainsi que les différents éléments à connaître au préalable pour leur utilisation.

2.1 Jeu du démineur

Le jeu du démineur est un jeu de réflexion qui se joue à un seul joueur. Il se déroule sur une grille de taille $n \times m$ cases, avec n le nombre de lignes et m le nombre de colonnes. Cette grille, nommée la *grille du jeu*, contient b bombes placées aléatoirement, avec $1 \leq b < nm$. La *densité de bombes* d est définie comme étant le rapport entre le nombre de bombes et le nombre de cases. En d'autres mots, celle-ci se calcule comme suit :

$$d = \frac{b}{nm}.$$

Concernant les cases, il y en a de deux types : les *cases vides*, ne contenant rien, et les *cases bombes*, contenant une bombe. Les cases ont également deux états : soit elles sont *démasquées*, soit elles sont *masquées*. Le joueur peut voir et connaître le type des cases démasquées, mais pas le type des cases masquées. La seule action possible pour l'utilisateur est de *démasquer* les cases masquées qui deviendront alors des cases démasquées.

Le but du démineur est d'identifier toutes les cases bombes et de démasquer toutes les cases vides sans démasquer une seule case bombe. Pour aider le joueur, toutes les cases vides ont un *marquage*. Un marquage mk sur une case vide t est un nombre naturel indiquant le nombre de cases bombes adjacentes à t (mk vaut donc 8 au maximum). L'utilisateur dispose également de *drapeaux* pouvant être uniquement placés sur les cases masquées. Le but de ceux-ci est de repérer ces cases comme étant des cases bombes. Cela permet au joueur de se rappeler où se trouvent toutes les cases bombes qu'il a déjà déduites.

Pour illustrer ce qui vient d'être dit, prenons l'exemple de la [Figure 1](#). Celle-ci présente une grille de 6×5 contenant exactement quatre bombes (représentés par le symbole « **B** »). Nous pouvons y voir les cases vides avec leur marquage ainsi que les cases bombes (les marquages nuls ne sont pas représentés dans la figure).

Initialement, lorsque le joueur s'apprête à commencer la partie, toutes les cases de la grille du jeu sont masquées. Le joueur peut donc démasquer n'importe laquelle.

À partir de la grille de la figure précédente, supposons que le joueur ait joué quelques coups et obtient la grille de la [Figure 2](#). Notons une coordonnée comme suit: (x, y) , avec x la coordonnée en abscisse et y la coordonnée en ordonnée. Également, disons que l'origine $(1, 1)$ est située en bas à gauche. Nous pouvons constater que le joueur a placé un drapeau (symbolisé par « **F** ») sur la case $(4, 1)$. Ce qui suggère que cette case est une case bombe. Comment le joueur a-t-il trouvé cela? Par la case $(5, 2)$, nous savons qu'exactement une de ses cases voisines est une case bombe. Comme toutes celles-ci sont des cases démasquées et sont vides mise à part la case $(4, 1)$, nous pouvons en conclure

1	2	2	1	
1	B	B	1	
1	3	3	2	
	1	B	1	
	1	2	2	1
		1	B	1

 FIGURE 1 – Exemple d'une grille de 6×5 avec quatre bombes

que cette dernière est une case bombe (le même raisonnement peut être fait à partir de la case $(5, 1)$).

La Figure 3 montre la fin de la partie précédente. Nous pouvons y voir que c'est une partie gagnante: le joueur a démasqué toutes les cases vides sans démasquer une case bombe. Nous constatons également qu'il a placé trois drapeaux sur des cases masquées. Cela signifie que le joueur pense que ces cases sont des cases bombes. En effet, c'est bien le cas (voir la Figure 1).

			1	
			1	
1	3		2	
	1		1	
	1	2	2	1
		1	F	1

FIGURE 2 – Exemple d'une partie en cours

1	2	2	1	
1	F		1	
1	3	3	2	
	1	F	1	
	1	2	2	1
		1	F	1

FIGURE 3 – Exemple d'une fin de partie

2.2 Apprentissage automatique et notions connexes

L'*apprentissage automatique* (*machine learning* en anglais) [3, 8] est un domaine très vaste de l'intelligence artificielle. Cela permet à un programme d'apprendre à réaliser une tâche sans que celle-ci soit explicitement programmée. C'est très utile notamment quand nous connaissons les entrées et les sorties du problème, mais pas l'algorithme sous-jacent (en général, celui-ci est très complexe et dépend de beaucoup de paramètres). Il peut aussi arriver que ce problème contient des informations ou des relations « cachées », c'est-à-dire qui ne sont pas directement visibles et qui nécessite de la réflexion. Nous pouvons citer par exemple la reconnaissance vocale ou encore la détection de courriels indésirables. Ce sont tous deux des problèmes où il est très difficile de décrire formellement un algorithme.

Il y a principalement deux types de problèmes dans l'apprentissage automatique: les *problèmes de classification* et les *problèmes de régression* [5]. Dans le premier cas, nous devons assigner une classe c parmi n classes à une certaine entrée. L'idée est donc de « coller une étiquette » à cette dernière. Dans le cas des problèmes de régression, à la place d'attribuer une classe à une entrée, nous attribuons une valeur (continue) à celle-ci. Plus généralement, résoudre un de ces deux types de problèmes revient à approximer une fonction, appelée la *fonction cible*.

Essentiellement, deux apprentissages différents existent: l'*apprentissage supervisé* (*supervised learning* en anglais) et l'*apprentissage par renforcement* (*reinforcement learning* en anglais) [4,5]. Ceux-ci se basent sur un *jeu d'entraînement* T (*training set* en anglais). Celui-ci est un ensemble de données consistant à entraîner le programme à réaliser une tâche. Dans l'apprentissage supervisé, ce jeu d'entraînement est un ensemble de paires entrée-sortie. Chaque paire, parfois appelée *exemple*, représente un échantillon de la fonction cible. Le jeu d'entraînement dans l'apprentissage par renforcement contient cette fois-ci uniquement des entrées. Le programme s'entraîne sur chacune, prédit un résultat et reçoit une indication si ce dernier est « bon » ou « mauvais »¹. Le programme peut donc s'ajuster en conséquence.

2.3 Réseaux de neurones artificiels

Cette section présente les éléments importants des réseaux de neurones artificiels. Cela a pour but d'améliorer la compréhension des sections suivantes. Tout ce qui suit a pour référence les documents [4,6,7,12].

Étant une méthode de l'apprentissage automatique, les *réseaux de neurones artificiels* sont des modèles mathématiques s'inspirant des réseaux de neurones biologiques. Ils sont constitués de simples unités de traitement appelées des *neurones*. Un neurone n_i peut être relié à un neurone n_j , avec $i \neq j$, par une *connexion*. Celle-ci permet de transporter une certaine information a_i du neurone n_i vers le neurone n_j (les connexions sont donc dirigées). Les connexions ont également un *poids* $w_{i,j}$ qui détermine la force du signal: plus un poids $w_{i,j}$ est élevé, plus l'information a_i sera considérée comme importante.

Chaque neurone n_j reçoit en entrée une suite de *valeurs d'activation* a_1, a_2, \dots, a_k ² correspondant à la sortie des neurones n_1, n_2, \dots, n_k connectés à n_j . À partir de ces valeurs, le neurone n_j réalise une somme pondérée et stocke le résultat dans une variable in_j . Ce dernier vaut donc:

$$in_j = \sum_{i=0}^k w_{i,j} a_i.$$

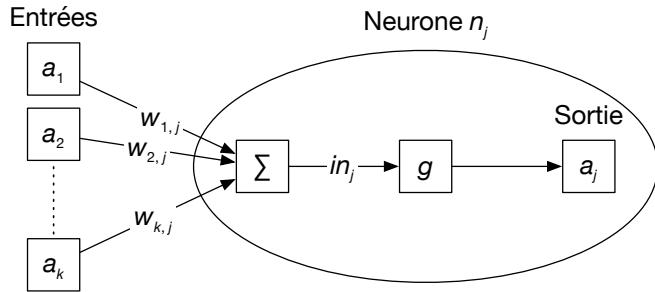
Le résultat in_j est ensuite envoyé à une *fonction d'activation* g . La valeur renournée par celle-ci est stockée dans une variable a_j et correspond à la sortie (valeur d'activation) du neurone n_j . Plus précisément, nous avons:

$$a_j = g(in_j) = g\left(\sum_{i=0}^k w_{i,j} a_i\right).$$

La [Figure 4](#) illustre ce processus.

1. Cette indication n'est pas nécessairement binaire, cela peut être une valeur continue.

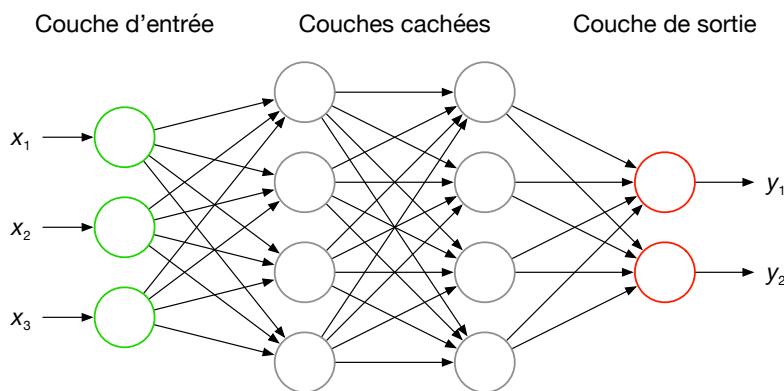
2. Nous pouvons voir les valeurs d'activation comme des informations qui passent et qui sont modifiées de neurone en neurone.


 FIGURE 4 – Calcul de la sortie a_j d'un neurone n_j

Les fonctions d'activation possèdent un *seuil d'activation* qui permet de déterminer l'état *d'activation* d'un neurone n_j . Si sa valeur d'activation a_j est supérieure à ce seuil, alors n_j est dit *actif*. Inversement, si a_j est inférieure au seuil, alors n_j est dit *inactif*. Un neurone n_j actif enverra de l'information à ces voisins, c'est-à-dire tous les neurones n_k tels qu'il existe une connexion entre n_j et n_k , contrairement à un neurone n_j inactif. Si la fonction d'activation utilisée n'a pas de seuil d'activation, alors les neurones sont actifs.

Maintenant que nous avons décrit le fonctionnement d'un neurone, nous allons nous intéresser à la *topologie* des réseaux de neurones. Celle-ci définit l'agencement des neurones et des connexions. Il en existe plusieurs types, mais nous n'allons en présenter qu'un seul: les réseaux *feedforward*. Un réseau de neurones reçoit en entrée un *vecteur d'entrée* $x = (x_1, x_2, \dots, x_k)$ et fournit en sortie un *vecteur de sortie* $y = (y_1, y_2, \dots, y_l)$.

Les réseaux *feedforward* sont des réseaux où les neurones sont organisés en couches. Nous retrouvons dans l'ordre: une *couche d'entrée* (contenant exactement k neurones), aucune ou plusieurs *couches cachées* et une *couche de sortie* (contenant exactement l neurones). Les réseaux ne possédant pas de couches cachées sont appelés des *perceptrons* et les réseaux en possédant au minimum deux sont appelés des *réseaux profonds*. Dans les réseaux *feedforward*, les neurones de chaque couche reçoivent en entrée les sorties des neurones de la couche précédente. Ils envoient ensuite leur sortie aux neurones de la couche suivante. En ce qui concerne la première couche, la couche d'entrée, les entrées des neurones sont les valeurs du vecteur d'entrée x . La sortie des neurones de la couche de sortie correspond au vecteur de sortie y . La Figure 5 montre un exemple d'un réseau *feedforward* avec deux couches cachées.


 FIGURE 5 – Exemple d'un réseau *feedforward*

Nous allons désormais voir l'*apprentissage* d'un réseau de neurones. Cela consiste à apprendre à celui-ci à réaliser une tâche (plus généralement il approxime une fonction). Nous avons besoin d'un jeu d'entraînement T . En particulier, pour l'apprentissage supervisé, celui-ci est un ensemble de paires entrée-sortie, c'est-à-dire un ensemble d'exemples où pour chacun d'entre eux nous connaissons le *vrai vecteur de sortie* yt pour chaque vecteur d'entrée x . Un jeu d'entraînement T de taille k est donc de la forme:

$$T = \{(x_1, yt_1), (x_2, yt_2), \dots, (x_k, yt_k)\},$$

où x_i est le vecteur d'entrée du réseau et yt_i est le vrai vecteur de sortie, $\forall i \in \{0, 1, \dots, k\}$.

Nous devons encore parler de deux notions importantes et nécessaires à l'apprentissage: la rétropropagation et l'optimizer. Pour ce faire, nous avons besoin de calculer une erreur e_i pour chaque exemple (x_i, yt_i) du jeu d'entraînement. Les étapes de ce calcul sont décrites ci-après.

1. Envoyer l'entrée x_i au réseau.
2. Récupérer la sortie yp_i , appelée le *vecteur de sortie prédit*.
3. Calculer l'erreur e_i à partir d'une *fonction de perte* (*loss function* en anglais) prenant en entrée yt_i et yp_i . Nous avons donc $e_i = loss(yt_i, yp_i)$, où $loss$ est la fonction de perte.

La *rétropropagation* (*backpropagation* en anglais) est le concept permettant de propager une certaine erreur e , calculée à partir de l'ensemble des erreurs e_i , de neurone en neurone et de couche en couche. (en général des neurones de la couche de sortie vers les neurones de la couche d'entrée). Cela permet de connaître la *contribution* c_i de chaque neurone n_i à cette erreur (« erreur locale »).

L'*optimizer* est un algorithme permettant d'ajuster les poids des connexions du réseau dans le but de diminuer l'erreur e . Cela peut être fait de différentes manières. Certains *optimizers* peuvent ajuster chaque poids $w_{i,j}$ à partir de $w_{i,j}$ et de l'erreur e , d'autres à partir de $w_{i,j}$ et de la contribution des neurones à l'erreur e . Les *optimizers* qui utilisent cette deuxième façon de faire sont ceux qui utilisent la rétropropagation.

L'apprentissage d'un réseau de neurones est fait par un *algorithme d'apprentissage* (*fit algorithm* en anglais). Celui-ci est réalisé en *ep itérations* (*epochs* en anglais). Lors d'une itération, le réseau explore et s'entraîne sur tout le jeu d'entraînement (celui-ci est donc visité au total ep fois). La fréquence de mise à jour des poids des connexions du réseau par l'*optimizer* est déterminée par la *taille du batch* b ; cela est réalisé tous les b exemples. En d'autres mots, lors d'un *batch*, l'erreur e est calculée à partir des erreurs e_i de ces b exemples et l'*optimizer* se sert de celle-ci pour corriger les poids du réseau. Quand la taille du *batch* vaut 1, alors nous disons que l'apprentissage est *online* et quand celle-ci est supérieure à 1, l'apprentissage est dit *offline*.

Ces étapes sont décrites de façon très haut niveau dans l'[Algorithme 1](#).

Une fois que le réseau a fini son apprentissage, nous pouvons évaluer sa performance. Nous utilisons pour cela un *jeu de test* (*test set* en anglais). Celui-ci est exactement de la même forme que le jeu d'entraînement. L'idée est d'envoyer les vecteurs d'entrée du jeu de test au réseau, de récupérer les vecteurs de sortie prédis et de comparer ces derniers avec les vrais vecteurs de sortie. Pour avoir une idée correcte de sa performance, il est important que le jeu de test soit différent du jeu d'entraînement. En effet, cela permet d'évaluer les performances du réseau pour de nouvelles situations qu'il n'a jamais rencontrées lors de son entraînement.

Algorithme 1 FITNEURALNETWORK($network, T, ep, loss, opt, b$)

Entrée:

Un réseau de neurones $network$, un jeu d'entraînement T , un nombre d'itérations ep , une fonction de perte $loss$, un optimizer opt et une taille du batch b .

Sortie:

Le réseau de neurones $network$ entraîné.

```

1: pour  $i$  allant de 1 à  $ep$  faire
2:    $j \leftarrow 1$ 
3:    $e \leftarrow$  Créer un tableau vide de taille  $b$ 
4:   pour chaque  $example$  dans  $T$  faire
5:      $(x, yt) \leftarrow example$ 
6:     Envoyer  $x$  à  $network$  et récupérer le vecteur de sortie prédit  $yp$ 
7:      $e[j] \leftarrow loss(yt, yp)$ 
8:     si  $j = b$  alors
9:       Mettre à jour les poids des connexions de  $network$  à partir de  $opt$  et de  $e$ 
10:       $j \leftarrow 1$ 
11:    sinon
12:       $j \leftarrow j + 1$ 
13: retourner  $network$ 

```

3 Réalisation d'une intelligence artificielle pour le jeu du démineur

Dans cette section, nous présentons notre démarche complète pour la réalisation d'une première intelligence artificielle pour le jeu du démineur. Celle-ci sera basée sur un réseau de neurones artificiels construit grâce à un apprentissage supervisé *via* un jeu d'entraînement. Plus précisément, nous verrons:

- le contexte technique contenant les outils utilisés;
- le réseau de neurones avec sa structure, ses paramètres, le jeu de données et le jeu d'entraînement;
- l'intelligence artificielle utilisant ce réseau et son fonctionnement;
- les résultats obtenus pour cette première intelligence artificielle.

3.1 Contexte technique

Pour réaliser notre implémentation, nous utiliserons le langage de programmation **Python** et la librairie **Keras** [1] pour manier les réseaux de neurones.

Le langage de programmation **Python** dispose d'une syntaxe très légère et très conviviale. Nous ne devons donc pas nous ennuyer avec une syntaxe « inutilement » lourde. Cela est par exemple très intéressant pour concevoir rapidement des prototypes ainsi qu'explorer de nouvelles approches. Un autre gros point fort de **Python** est l'immense catalogue d'outils et de librairies disponibles (voir quelques exemples ci-après). Toutefois, ce langage est moins rapide que beaucoup d'autres; cela est majoritairement dû au fait que **Python** est un langage interprété (aucune compilation n'est présente).

Outre la simplicité de **Python**, la librairie **Keras**, disponible dans ce langage, est une interface de programmation applicative (API) de très haut niveau reposant sur des librairies très populaires dans le domaine des réseaux de neurones comme **Theano**¹ et **Tensorflow**². Nous appelons ces librairies des *backends*. Il existe beaucoup d'autres librairies permettant de réaliser le même travail. Cependant, ces dernières sont généralement plus bas niveau et plus difficiles à utiliser. Pour illustrer la simplicité d'utilisation de **Keras**, le [Code source 1](#) montre la création et l'entraînement d'un réseau de neurones pour un problème de classification en seulement quelques lignes de code.

Nous utiliserons également d'autres outils et librairies de **Python**. Ils sont cités et décrits brièvement ci-après.

- **Jupyter**⁴: utilitaire permettant de réaliser des *notebooks*.
- **Matplotlib**⁵: librairie permettant de réaliser très simplement des graphiques.
- **Numpy**⁶: librairie offrant la possibilité de manipuler très efficacement et rapidement des données. Cela nous sera notamment utile pour manier le jeu de données.

1. <http://deeplearning.net/software/theano/>.

2. <https://www.tensorflow.org>.

3. Exemple repris de l'adresse suivante: <https://keras.io/getting-started/sequential-model-guide/>.

4. <http://jupyter.org>.

5. <https://matplotlib.org>.

6. <http://www.numpy.org>.

3.2 Réseau de neurones

```
1 # For a single-input model with 2 classes (binary classification):
2 model = Sequential()
3 model.add(Dense(32, activation='relu', input_dim=100))
4 model.add(Dense(1, activation='sigmoid'))
5 model.compile(optimizer='rmsprop',
6                 loss='binary_crossentropy',
7                 metrics=['accuracy'])
8
9 # Generate dummy data
10 import numpy as np
11 data = np.random.random((1000, 100))
12 labels = np.random.randint(2, size=(1000, 1))
13
14 # Train the model, iterating on the data in batches of 32 samples
15 model.fit(data, labels, epochs=10, batch_size=32)
```

CODE SOURCE 1 – Exemple simple d'utilisation de la librairie Keras³

3.2 Réseau de neurones

Dans cette section, nous décrirons toutes les informations relatives au réseau de neurones artificiels comme sa conception et son entraînement. Premièrement, nous émettrons une certaine hypothèse et décrirons l'objectif et la fonction cible. Deuxièmement, nous détaillerons le fonctionnement du réseau de neurones avec sa structure et ses différents paramètres. Troisièmement, nous expliquerons la création du jeu de données et du jeu d'entraînement. Et quatrièmement, nous présenterons quelques méthodes pour mesurer les performances du réseau de neurones.

3.2.1 Hypothèse et objectif

Nous faisons l'hypothèse que toutes les grilles que l'intelligence artificielle rencontrera seront des grilles de taille 10×10 contenant 10 bombes (c'est le cas le plus classique et le plus courant dans le jeu du démineur). Le réseau de neurones sera donc conçu pour ce type de grilles. La raison de ce choix est de se concentrer uniquement sur un seul type de grilles. En effet, une intelligence artificielle « universelle », conçue pour tous les types, est plus difficile à réaliser et potentiellement moins performante.

L'objectif du réseau de neurones est de fournir une information utile à l'intelligence artificielle. Dans notre cas, il devra être capable de prédire si une case est une case bombe ou non. Pour répondre à cette question, il n'est pas nécessaire de montrer toute la grille au réseau; effectivement, seules les informations locales suffisent. Dès lors, nous travaillons avec un système de *sous-grilles* permettant d'extraire le « voisinage » de chaque case. Cela a pour effet de limiter l'entrée du réseau et donc de réduire, par exemple, le temps de calcul de l'entraînement de celui-ci.

De manière plus formelle, nous définissons la sous-grille sg d'une case t comme étant une grille, extraite de la grille du jeu, où t est le centre de sg . Les sous-grilles sont donc *carrées*, c'est-à-dire que le nombre de lignes nr et le nombre de colonnes nc de chacune sont égaux. Aussi, nous avons que nr et nc sont impairs. Nous définissons le *diamètre* d d'une sous-grille comme étant le nombre de lignes (ou de colonnes) de celle-ci. Nous obtenons

donc la relation suivante:

$$d = nr = nc.$$

Également, le *rayon* r d'une sous-grille peut être défini à partir du diamètre de celle-ci comme ceci:

$$r = \frac{d - 1}{2}.$$

La [Figure 6](#) montre un exemple d'une sous-grille de taille 5×5 de rayon 2 pour une certaine case (coloriée en rouge).

Une taille de 5×5 pour les sous-grilles nous permet d'avoir suffisamment d'informations pour évaluer la probabilité que la case centrale t_c soit une case bombe. En effet, ce sont les cases adjacentes à t_c qui déterminent si t_c est une case bombe (dépendant de leur marquage). Toutefois, ces cases permettent également de déterminer si leurs cases adjacentes sont également des cases bombes ou vides. Prenons l'exemple de la [Figure 7](#) montrant une sous-grille de taille 3×3 (de rayon 1). Nous pouvons voir qu'une case bombe a été repérée et qu'un drapeau a été déposé sur celle-ci (représenté par « F »). Avec une sous-grille de taille 3×3 , nous pouvons croire que la case t_1 en haut à gauche (avec « 1 » comme marquage) de la case centrale t_c signifie que t_c est peut-être une case bombe. Or, ce n'est pas le cas. Effectivement, la case au-dessus de t_1 , disons t_2 , est une case bombe (un drapeau est posé dessus). Cela n'est pas possible avec une grille de taille 5×5 , car nous voyons le drapeau et pensons donc que t_2 est une case bombe.

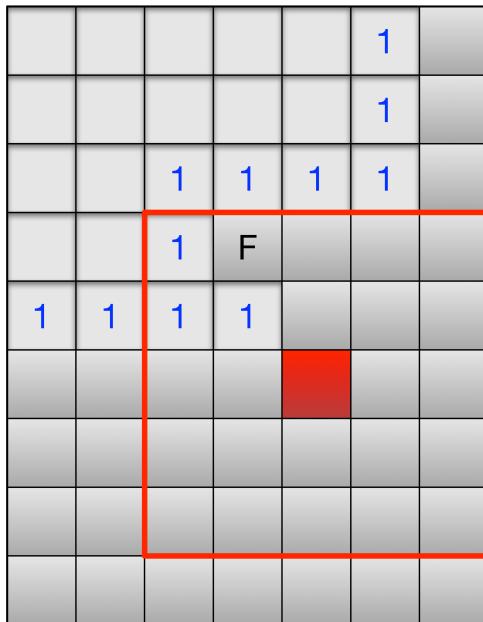


FIGURE 6 – Exemple d'une sous-grille 5×5

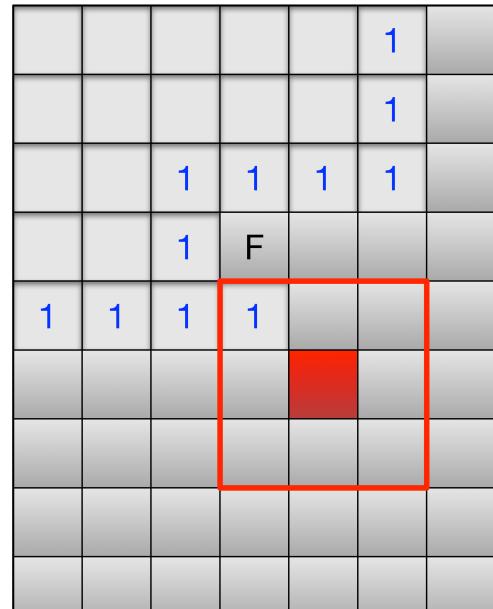


FIGURE 7 – Exemple d'une sous-grille 3×3

Avec ce système de sous-grilles, nous devons faire attention aux sous-grilles qui *débordent* de la grille du jeu g . Une sous-grille sg déborde si une de ses cases *déborde*, c'est-à-dire une case t se trouvant dans sg et ne se trouvant pas dans g . La [Figure 8](#) montre un exemple d'une sous-grille qui déborde. Nous voyons qu'exactement cinq de ses cases débordent; ce sont celles tout à gauche dans la sous-grille. Comme l'entrée d'un réseau de neurones est toujours de taille fixe, nous ne pouvons pas réduire la taille des

sous-grilles. Au lieu de cela, nous remplaçons chaque case t qui déborde par un nouvel objet nommé *case mur*. Nous avons donc désormais deux nouveaux types de cases: les cases murs, contenant un mur, et les *cases libres*, ne contenant pas de mur.

Par conséquent, le réseau de neurones recevra des sous-grilles de taille 5×5 en entrée et devra prédire si oui ou non la case centrale t de celles-ci est une case bombe.

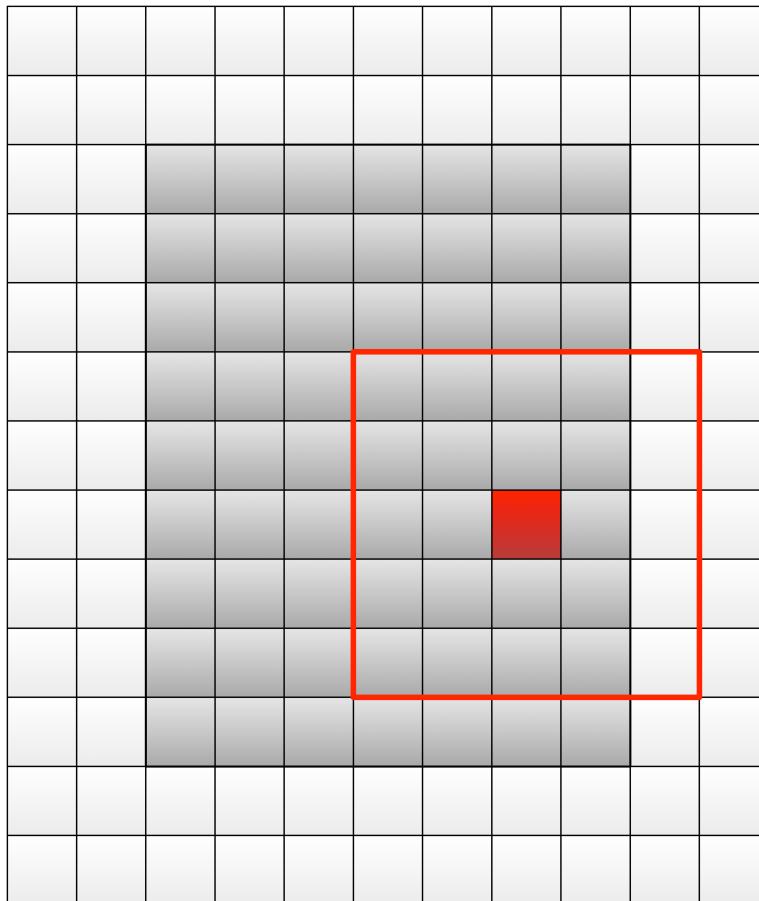


FIGURE 8 – Exemple d'une sous-grille de 5×5 qui déborde

3.2.2 Fonctionnement du réseau

Comme le réseau de neurones doit être capable de prédire si oui ou non la case centrale d'une sous-grille est une case bombe (voir la section précédente), celui-ci sera donc entraîné avec un jeu d'entraînement de la forme suivante:

$$T = \{(sg_1, yt_1), (sg_2, yt_2), \dots, (sg_k, yt_k)\},$$

où sg_i est une sous-grille et yt_i est égal à 1 si la case centrale de sg_i est une case bombe, 0 sinon, $\forall i \in \{0, 1, \dots, k\}$.

Dès lors, pour une sous-grille sg donnée, le réseau prédira 1 si la case centrale de sg est une case bombe et 0 dans le cas contraire. Il est possible que celui-ci prédise une valeur comprise entre 0 et 1; cette valeur sera interprétée comme un « degré de fiabilité » permettant de connaître le niveau de certitude du réseau. Effectivement, si par exemple le

réseau nous prédit une valeur de 0,98, alors nous considérerons qu'il est sûr et certain que la case centrale de sg est une case bombe, tandis que s'il nous prédit une valeur de 0,5, alors nous interpréterons cela comme une incertitude de la part du réseau (il éprouve des difficultés à dire si la case centrale de sg est une case bombe ou vide). Vu que les valeurs prédites par le réseau sont continues (nombre compris entre 0 et 1), nous sommes donc en face à un problème de régression.

Pour réaliser l'entraînement du réseau, nous devons utiliser un encodage pour chaque objet du jeu (type de cases). Un *encodage* consiste juste à attribuer une valeur numérique à l'objet. Pour chaque case t , nous définissons arbitrairement l'encodage suivant⁷:

- si t est démasquée:
 - si t est une case vide et a un marquage i : i ;
 - sinon si t est une bombe: -1 ;
 - sinon (t est une case mur): -2 ;
- sinon (t est masquée):
 - si t contient un drapeau: -4 ;
 - sinon (t ne contient pas de drapeau): -3 .

Nous devons également définir la structure du réseau de neurones ainsi que certains paramètres. Comme les auteurs de la référence [9] ont travaillé sur le même problème que le nôtre avec une procédure similaire et qu'ils ont obtenu de bons résultats, nous utilisons l'une de leurs structures et leurs paramètres.

Concernant la structure, il s'agit d'un réseau *feedforward* composé de cinq couches contenant 25 (nombres de cases d'une sous-grille), 300, 256, 128 et 1 (nombre de valeurs prédites par le réseau) neurones respectivement. Chaque couche l contient un certain nombre de neurones et utilise une fonction d'activation. Cette dernière est utilisée par tous les neurones de la couche l . Parmi les fonctions d'activation utilisées, nous trouvons la fonction d'activation unité de rectification linéaire (*rectified linear unit* en anglais, souvent abrégée par RELU) et la fonction sigmoïde. La première est définie comme suit [11]:

$$relu(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 1 \end{cases},$$

et la seconde comme [11]:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}.$$

Le Tableau 1 montre la structure utilisée.

Les autres paramètres concernent l'entraînement. Le nombre d'itérations est fixé à 1 et la taille du *batch* à 10. Cela signifie que les sous-grilles sont parcourues au total qu'une seule fois et que les poids du réseau sont mis à jour tous les 10 sous-grilles (voir la Section 2.3). Également, nous utilisons RMSPROP comme *optimizer* et l'erreur quadratique moyenne *mse* comme fonction de perte et qui est définie comme suit:

$$mse = \frac{1}{n} \sum_{i=1}^n (t_i - p_i)^2,$$

7. Rappel: seules les cases vides ont un marquage et seules les cases masquées peuvent contenir un drapeau.

Paramètre Couche	Nombre de neurones	Fonction d'activation
Couche d'entrée	25	—
1 ^{re} couche cachée	300	Unité de rectification linéaire
2 ^e couche cachée	256	Unité de rectification linéaire
3 ^e couche cachée	128	Unité de rectification linéaire
Couche de sortie	1	Sigmoïde

Tableau 1 – Structure du réseau de neurones

où t_i est une vraie valeur, p_i est une valeur prédite et n correspond au nombre de vraies valeurs (et également au nombre de valeurs prédites). De manière plus précise, comme chaque vrai vecteur de sortie yt et chaque vecteur de sortie prédit yp ont une taille de 1, t_i correspond à la première composante de yt et p_i à la première composante de yp .

3.2.3 Génération d'un jeu de données et un d'un jeu d'entraînement

Dans cette section nous parlons d'une étape très importante pour l'apprentissage supervisé du réseau de neurones: le jeu de données et le jeu d'entraînement. Ce dernier est utilisé pour l'apprentissage du réseau où celui-ci tente d'approximer la fonction cible.

Le *jeu de données* (*data set* en anglais) est un ensemble de données (souvent « brutes ») liées au problème en question. Cet ensemble est un tableau à deux dimensions où chaque ligne représente un *individu* et chaque colonne une *variable* (ou encore *feature* en anglais) qui est en réalité un *attribut* de celui-ci [11]. Les données du jeu de données sont sélectionnées, éventuellement formatées ou transformées, pour créer un jeu d'entraînement. Un jeu d'entraînement, déjà abordé dans les Sections 2.2 et 2.3, est un ensemble de paires contenant chacune un vecteur d'entrée et un vrai vecteur de sortie.

Nous pouvons obtenir un jeu de données de différentes manières. Nous pouvons par exemple récupérer des échantillons (exemples) dans la « nature » ou encore générer celui-ci. Dans le premier cas, il s'agit tout simplement de récolter des échantillons, souvent lors d'une expérience. Dans le second, c'est un algorithme qui a la possibilité de générer tous les individus ainsi que tous leurs attributs. Nous pouvons nous demander quel est l'intérêt d'utiliser une technique d'intelligence artificielle comme les réseaux de neurones si nous possédons un tel algorithme. En fait, il y a beaucoup de problèmes où il est impossible de générer le jeu de données au complet en cause de la taille très importante (souvent exponentielle) de celui-ci. Dans une telle situation, nous générerons qu'une (petite) partie du jeu de données complet (un sous-ensemble).

Dans notre cas, pour le jeu du démineur, nous utilisons cette deuxième méthode; c'est-à-dire que nous générerons des données relatives au jeu en question par un algorithme. En effet, comme les règles du jeu sont « simples » et déterministes, nous pouvons par conséquent générer des sous-grilles et calculer ensuite si oui ou non la case centrale de celles-ci est une case bombe. Nous ne pouvons pas nous permettre de générer toutes les sous-grilles existantes, car celles-ci sont beaucoup trop nombreuses.

Avant de présenter la génération du jeu de données et du jeu d'entraînement, nous

devons introduire deux nouvelles notions. Effectivement, nous faisons désormais (et dans la suite du rapport) la distinction entre deux types de sous-grilles: d'un côté les *sous-grilles sans masque*, où aucune case de celles-ci n'est masquée, et de l'autre les *sous-grilles avec masque*, où il existe au moins une case de celles-ci qui est masquée. Le jeu de données contiendra uniquement des sous-grilles sans masque tandis que le jeu d'entraînement sera composé uniquement de sous-grilles avec masque.

Commençons par la génération du jeu de données. Celui-ci contient des sous-grilles de taille 5×5 sans masque. Il est nécessaire que celles-ci soient *vraisemblables*. En d'autres mots, elles doivent correspondre à la réalité, c'est-à-dire aux sous-grilles extraites à partir des cases de la grille du jeu (de taille 10×10 et contenant 10 bombes). Nous ne voulons pas par exemple avoir une sous-grille (de 5×5) avec 20 bombes alors que la grille du jeu n'en possède que 10. De manière assez similaire, une sous-grille possédant les 10 bombes de la grille du jeu devrait être très rare.

Pour ce faire, nous avons besoin de construire des sous-grilles de 7×7 (de rayon 3) et d'extraire pour chacune la sous-grille de 5×5 « située au centre » de celle-ci. Plus précisément, pour chaque sous-grille sg_1 de 7×7 , la sous-grille sg_2 de 5×5 est formée en supprimant la première ligne, la dernière ligne, la première colonne et la dernière colonne de sg_1 . En réalité, nous avons besoin de considérer des sous-grilles de 7×7 pour calculer le marquage des cases situées aux extrémités de la sous-grille de 5×5 vu que le marquage d'une case dépend de ses cases voisines directes. Prenons un exemple pour illustrer cela. La [Figure 9](#), où les marquages nuls ne sont pas représentés, montre une sous-grille de 7×7 et la [Figure 10](#) représente la sous-grille de 5×5 extraite à partir de celle-ci. Nous voyons que générer une sous-grille de 7×7 et d'ensuite extraire la sous-grille de 5×5 donne une sous-grille vraisemblable et « correcte »; celle-ci peut en effet provenir d'une extraction d'une sous-grille de 5×5 à partir d'une case de la grille du jeu. Si nous n'utilisons pas une sous-grille de 7×7 , générer la sous-grille de 5×5 de la [Figure 10](#) serait plus difficile en cause des marquages situés aux extrémités de cette dernière. Nous pouvons par exemple citer les deux marquages (valant « 1 ») des deux cases situées en haut à gauche de la sous-grille de 5×5 puisque, dans celle-ci, aucune bombe n'est adjacente à ces deux cases.

Nous allons maintenant parler de la génération d'une sous-grille de 7×7 sans masque, en sachant que toutes les sous-grilles de cette taille sont construites de la même façon.

Pour avoir une sous-grille vraisemblable, nous avons besoin d'ajouter des cases murs dans celle-ci (discuté dans la [Section 3.2.1](#)). Le *mur de gauche*, le *mur de droite*, le *mur du haut* et le *mur du bas* contiennent l'ensemble des cases murs situées à gauche, à droite, en haut et en bas respectivement. L'*épaisseur* pour le mur de gauche et le mur de droite (respectivement pour le mur du haut et le mur du bas) représente le nombre de colonnes (respectivement le nombre de lignes) où chaque case est une case mur. La [Figure 11](#) montre un exemple d'une sous-grille de 7×7 avec une épaisseur de 1 pour le mur de droite, une épaisseur de 3 pour le mur du bas et une épaisseur de 0 pour les deux autres murs (les murs sont symbolisés par « W »).

L'[Algorithme 2](#) montre le calcul de l'épaisseur de chaque mur de la sous-grille sg de 7×7 en cours de construction. Ce calcul se base sur la taille de la grille du jeu et la taille de sg . L'épaisseur est aléatoire et peut valoir au maximum r , où r est le rayon de sg . Si l'épaisseur du mur de gauche (respectivement du mur du haut) est supérieure à 0, alors l'épaisseur du mur de droite (respectivement du mur du bas) est égale à 0, et réciproquement. En effet, si nous n'imposons pas ces conditions, nous pourrions par

1	1				1	1
B	1		1	1	2	B
1	1		1	B	2	1
			1	1	1	
			1	1	1	
1	1	1	1	B	1	
1	B	1	1	1	1	

FIGURE 9 – Exemple d'une sous-grille de 7×7

1		1	1	2
1		1	B	2
		1	1	1
		1	1	1
1	1	1	B	1

FIGURE 10 – Sous-grille de 5×5 extraite de la sous-grille de 7×7

1	1					W
B	1		1	1	1	W
1	1		1	B	1	W
			1	1	1	W
W	W	W	W	W	W	W
W	W	W	W	W	W	W
W	W	W	W	W	W	W

FIGURE 11 – Exemple d'une sous-grille de 7×7 avec des murs

exemple avoir une épaisseur supérieure à 0 pour le mur de gauche et le mur de droite de sg ; cela est impossible vu la taille de la grille du jeu et la taille de sg .

Algorithme 2 COMPUTEWALLTHICKNESS($r, nro, ncol$)

Entrée:

Le rayon r de la sous-grille en construction, le nombre de lignes nro de la grille du jeu et le nombre de colonnes $ncol$ de la grille du jeu.

Sortie:

L'épaisseur lw, rw, tw et bw du mur de gauche, de droite, du haut et du bas respectivement.

```

1:  $lwp \leftarrow r/ncol$  # Probabilité que le mur de gauche ait une épaisseur supérieure à 0.
2:  $rwp \leftarrow r/ncol$  # Probabilité que le mur de droite ait une épaisseur supérieure à 0.
3:  $twp \leftarrow r/nro$  # Probabilité que le mur du haut ait une épaisseur supérieure à 0.
4:  $bwp \leftarrow r/nro$  # Probabilité que le mur du bas ait une épaisseur supérieure à 0.
5:  $lw, rw, tw, bw \leftarrow 0, 0, 0, 0$ 
6:  $rand \leftarrow$  Tirer un nombre réel aléatoire compris entre 0 et 1
7:  $thick \leftarrow$  Tirer un nombre entier aléatoire compris entre 1 et  $r$ 
8: si  $rand < lwp$  alors
9:    $lw \leftarrow thick$ 
10: sinon si  $rand < (lwp + rwp)$  alors
11:    $rw \leftarrow thick$ 
12:  $rand \leftarrow$  Tirer un nombre réel aléatoire compris entre 0 et 1
13:  $thick \leftarrow$  Tirer un nombre entier aléatoire compris entre 1 et  $r$ 
14: si  $rand < twp$  alors
15:    $tw \leftarrow thick$ 
16: sinon si  $rand < (twp + bwp)$  alors
17:    $bw \leftarrow thick$ 
18: retourner  $lw, rw, tw, bw$ 

```

Maintenant que nous avons calculé l'épaisseur des murs, il nous suffit d'ajouter les cases murs. Par exemple, supposons que notre sous-grille de 7×7 en construction ait une épaisseur de 2 pour le mur de gauche. Alors, les cases des deux premières colonnes de celle-ci seront des cases murs.

Il ne nous reste plus qu'à ajouter les bombes et les marquages pour achever la construction de notre sous-grille sg de 7×7 . L'[Algorithme 3](#) présente le calcul du nombre de bombes pour cette dernière. Pour ce faire, cet algorithme utilise le rapport entre le nombre de cases libres⁸ de sg et le nombre de cases de la grille du jeu. Le nombre de bombes est aléatoire et peut être de b au maximum, où b est le nombre de bombes de la grille du jeu.

Nous connaissons maintenant le nombre de bombes b que la sous-grille en construction doit contenir. Dès lors, il nous suffit de placer b bombes sur b cases libres différentes aléatoirement. Pour chaque case libre t étant également une case bombe et pour chaque case vide at adjacente à t , nous incrémentons le marquage de at de 1.

La construction de notre sous-grille de 7×7 est à présent terminée. Nous pouvons donc en extraire la sous-grille de 5×5 située au centre comme expliqué précédemment.

8. Rappel: une case libre est une case ne contenant pas de mur.

Algorithme 3 COMPUTENUMBEROFBOMBS(*freetilsg*, *nro*, *ncol*, *bomg*)

Entrée:

Le nombre de cases libres *freetilsg* de la sous-grille en cours de construction, le nombre de lignes *nro*, le nombre de colonnes *ncol* et le nombre de bombes *bomg* de la grille du jeu.

Sortie:

Le nombre de bombes *bomsg* de la sous-grille.

```

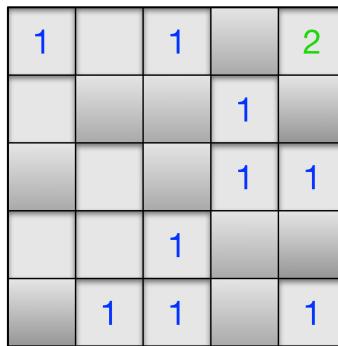
1: tilg  $\leftarrow$  nro.ncol # Nombre de cases de la grille du jeu.
2: ratio  $\leftarrow$  freetilsg/tilg
3: bomsg  $\leftarrow$  0
4: pour i allant de 1 à bomg faire
5:   rand  $\leftarrow$  Tirer un nombre réel aléatoire compris entre 0 et 1
6:   si rand  $<$  ratio alors
7:     bomsg  $\leftarrow$  bomsg + 1
8: retourner bomsg

```

Nous pouvons désormais créer le jeu de données avec des sous-grilles de 5×5 construites avec la méthode décrite ci-avant. Dans notre cas, nous générerons un jeu de données où la case centrale de chaque sous-grille est une case bombe et un autre où la case centrale de chaque sous-grille est une case vide (nous ajustons pour cela le placement des bombes). Appelons-le premier *jeu de données avec bombe* et le second *jeu de données sans bombe*.

Nous pouvons maintenant passer à la génération du jeu d'entraînement. Pour réaliser cela, nous allons générer *n* sous-grilles avec masque pour chaque sous-grille *sg* sans masque contenue dans les deux jeux de données. Une sous-grille *msg* avec masque est tout simplement créée en dupliquant *sg* et en *masquant* quelques cases libres. Masquer une case *t* revient à transformer *t* en une case masquée. Pour être plus précis, nous masquons toujours la case centrale de *sg*, car lorsque nous extrayons une sous-grille à partir d'une case *t*, nous voulons savoir si *t* est une case bombe ou non (il est donc sous-entendu que *t* est une case masquée). Également, toutes les cases bombes sont masquées, car le réseau ne rencontrera jamais une case bombe démasquée en pratique. En effet, si une case bombe est démasquée, la partie est perdue et donc terminée. De plus, nous masquons un nombre *m* aléatoire de cases libres pas encore masquées. La [Figure 12](#) montre un exemple d'une sous-grille de 5×5 avec masque construite à partir de la sous-grille sans masque de la [Figure 10](#). Ces sous-grilles avec masque ne sont pas vraisemblables. En réalité, nous voulons que le réseau de neurones « réfléchisse » avec les marquages et les murs uniquement et non pas avec les cases masquées qui correspondent simplement à un manque d'information.

Nos deux jeux de données contiennent chacun 1.000.000 de sous-grilles sans masque. Ceux-ci sont utilisés pour former le jeu d'entraînement et le jeu de test. Pour créer le jeu d'entraînement, nous prenons d'abord dans l'ordre 50.000 sous-grilles sans masque dans le jeu de données sans bombe et 50.000 autres sous-grilles dans le jeu de données avec bombe. Ensuite, à partir de chaque sous-grille parmi ces 100.000, nous générerons 10 sous-grilles avec masque. Le jeu d'entraînement contient par conséquent un total de 1.000.000 de sous-grilles avec masque. Cela semble être une quantité acceptable dans un premier


 FIGURE 12 – Exemple d'une sous-grille de 5×5 avec masque

temps. Nous ajusterons cela dans la [Section 4](#); par exemple en prenant 100.000 sous-grilles sans masque dans les deux jeux de données au lieu de 50.000.

3.2.4 Évaluation des résultats du réseau

Nous présentons dans cette section différentes *méthodes d'évaluation* qui permettent de connaître la qualité des prédictions du réseau de neurones. Cela nous permet d'avoir une estimation sur la performance réelle de notre réseau et de savoir s'il a atteint l'objectif que nous avons fixé dans la [Section 3.2.1](#). Également, cela peut être utile pour comparer différents réseaux de neurones entre eux et déterminer lequel répond le mieux à l'objectif. Toutes ces méthodes se basent sur un jeu de test qui est généré de la même manière que le jeu d'entraînement⁹ et qui est différent du jeu d'entraînement (voir la [Section 2.3](#)).

Dans un premier temps, nous pouvons utiliser les méthodes d'évaluation directement implémentées dans la librairie Keras [2]. Ces méthodes s'appuient sur deux choses: les vrais vecteurs de sortie du jeu de test et les vecteurs de sortie prédits par le réseau de neurones. Parmi les méthodes présentes dans la librairie, nous utiliserons l'erreur absolue moyenne *mae* et l'erreur quadratique moyenne *mse*. La deuxième a déjà été définie dans la [Section 3.2.2](#) et la première est définie comme suit:

$$mae = \frac{1}{n} \sum_{i=1}^n |t_i - p_i|,$$

où t_i est une vraie valeur, p_i est une valeur prédite et n correspond au nombre de vraies valeurs (et également au nombre de valeurs prédites).

Nous avons également développé nos propres méthodes d'évaluation, non présentes dans librairie Keras. Toutes celles-ci se basent sur une liste d'erreurs $el = [e_1, e_2, \dots, e_n]$, où n est le nombre de sous-grilles avec masque dans le jeu de test. Chaque erreur e_i est calculée à partir du i^{e} vrai vecteur de sortie et du i^{e} vecteur de sortie prédit. Dans notre cas, cette erreur est tout simplement l'erreur absolue *ae*, calculée comme ceci:

$$ae = |t - p|,$$

9. Le jeu de test est créé dans notre cas en prenant 5.000 sous-grilles sans masque dans les deux jeux de données et en générant 10 sous-grilles avec masque pour chacune de celles-ci (le jeu de test contient donc au total 100.000 sous-grilles avec masque).

où t est une vraie valeur et p est une valeur prédictée.

Notre première méthode nous permet d'obtenir la répartition des erreurs de la liste d'erreurs décrite ci-dessus. Cela est très utile pour par exemple répondre à la question « combien d'erreurs sont inférieures à 0,1? » ou encore « quelle quantité d'erreurs sont comprises entre 0,6 et 0,9? ». De manière plus détaillée, nous découpons la plage des erreurs, c'est-à-dire l'intervalle où toutes les erreurs sont comprises, en n intervalles. Pour chaque intervalle, nous calculons le pourcentage du nombre d'erreurs qui se trouvent dans celui-ci. La Figure 13 montre un exemple, sous forme graphique, de résultats obtenus par cette méthode avec 10 intervalles. Nous voyons par exemple qu'un peu plus de 40% des erreurs sont inférieures à 0,1 et qu'environ 30% sont situées entre 0,3 et 0,7.

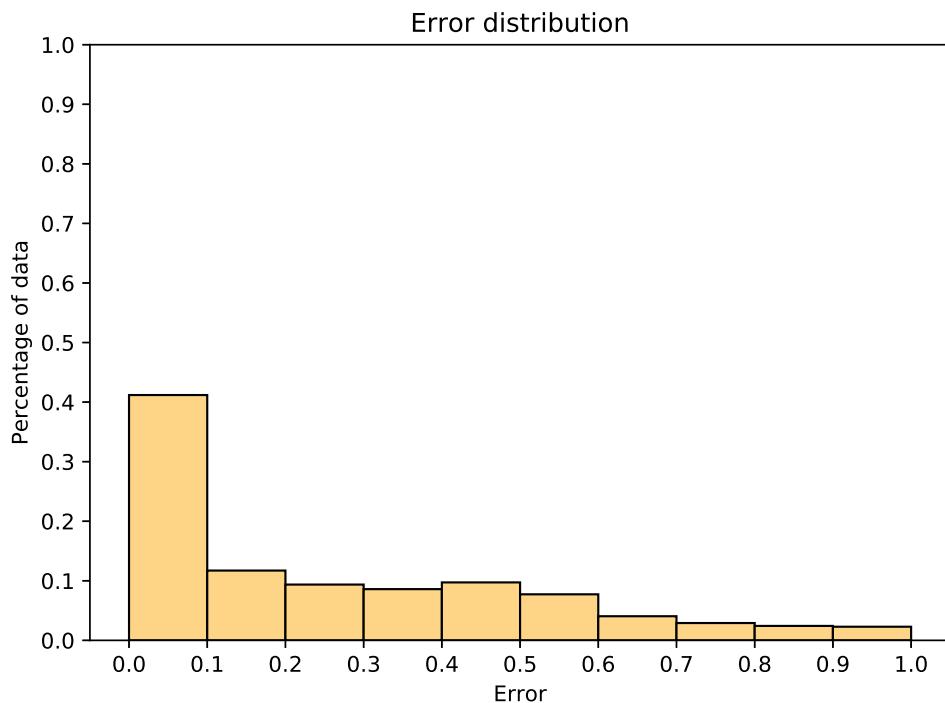


FIGURE 13 – Exemple d'une répartition d'erreurs obtenue par notre méthode

Notre seconde méthode consiste à réaliser un diagramme de dispersion (nuage de points) des erreurs en fonction du nombre de cases vides démasquées. Cela permet d'avoir une idée des erreurs commises par le réseau pour les sous-grilles où la plupart des cases sont masquées et pour les sous-grilles où la plupart sont démasquées. En toute logique, nous devrions avoir des erreurs plus élevées en moyenne dans ce premier cas que dans le second (car moins d'informations sont disponibles). La Figure 14 contient un exemple d'un diagramme de dispersion de ce type. Nous constatons, comme dit précédemment, que l'erreur commise pour une sous-grille contenant 24 cases démasquées est beaucoup plus faible en moyenne que pour une sous-grille ne contenant qu'une seule case démasquée.

3.3 Intelligence artificielle

Dans cette section, nous présentons notre intelligence artificielle basée sur le réseau de neurones décrit dans la section précédente. Nous verrons l'objectif de celle-ci, son

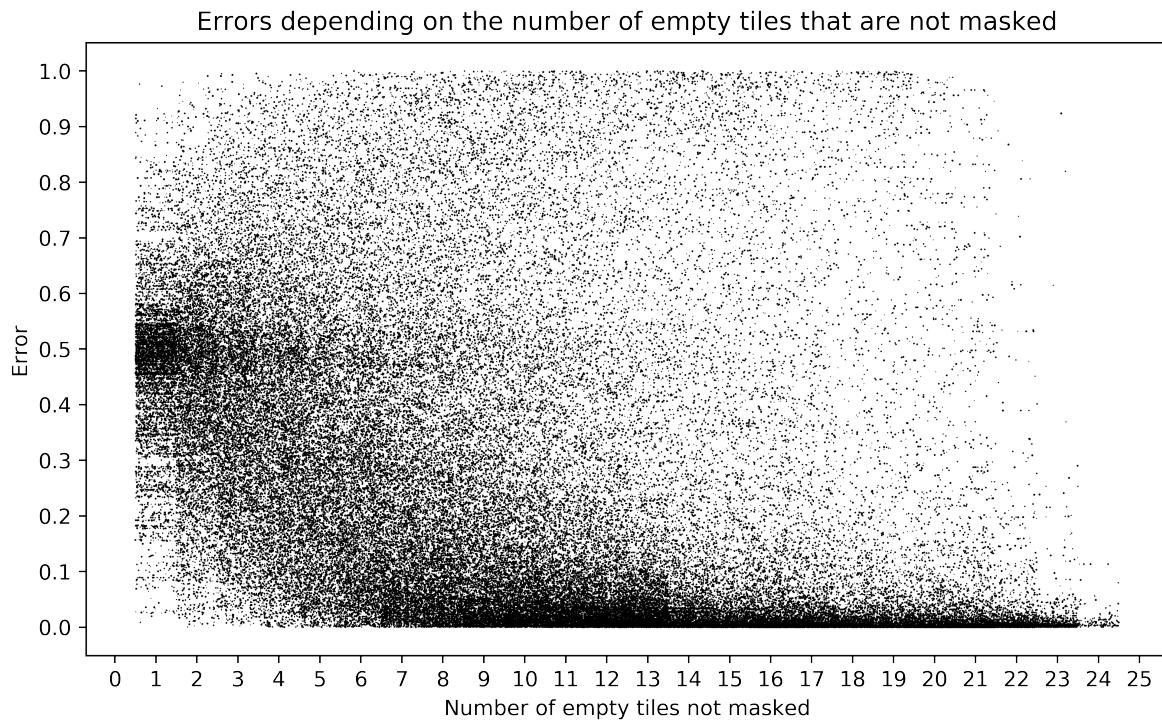


FIGURE 14 – Exemple d'un diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées

fonctionnement et les mesures pouvant servir à estimer sa performance. Également, nous parlerons de notre implémentation, sans entrer dans les détails.

3.3.1 Hypothèses et objectif

Notre première hypothèse est identique à celle faite pour le réseau de neurones, c'est-à-dire que nous supposons que l'intelligence artificielle sera utilisée uniquement sur des grilles de taille 10×10 contenant exactement 10 bombes. La justification est également identique (voir la [Section 3.2.1](#)). Comme seconde hypothèse, nous présumons que notre réseau de neurones a été conçu et entraîné comme précisé dans la [Section 3.2](#) et qu'il réalise de très bonnes prédictions.

Notre objectif est d'avoir une intelligence artificielle ayant une performance décente pour le jeu du démineur. Pour rappel, le but de ce jeu est de détecter l'emplacement de toutes les cases bombes et de démasquer toutes les cases vides sans démasquer une seule case bombe. Dès lors, nous désirons une intelligence artificielle capable de gagner un nombre raisonnable de parties.

3.3.2 Fonctionnement de l'intelligence artificielle

Pour atteindre notre objectif, nous utilisons notre réseau de neurones et le considérons comme notre *oracle*.

Mettons-nous au cours d'une partie. Au premier tour, nous nous retrouvons avec une grille où toutes les cases sont masquées. Il n'y a donc aucun moyen de connaître celle

qui a la probabilité la plus faible d'être une case bombe. Dès lors, pour ce premier tour, nous choisissons aléatoirement une case et la démasquons. Pour chacun des prochains tours, nous récupérons toutes les cases masquées de la grille du jeu courante. Pour chaque case masquée t parmi celles-ci, nous extrayons la sous-grille sg de 5×5 où t est la case centrale de sg , envoyons sg au réseau de neurones et récupérons le vecteur de sortie prédict. Parmi tous les vecteurs de sortie prédicts obtenus, nous prenons celui dont la première composante est la plus petite. Disons qu'il s'agit du i^e vecteur. Alors, nous démasquons la i^e case masquée. Ce choix est cohérent avec le fait que le réseau prédit 0 quand la case centrale d'une sous-grille est une case vide et 1 quand il s'agit d'une case bombe (chaque valeur intermédiaire est interprétée comme un « degré de fiabilité »). Bien évidemment, pour que cela fonctionne correctement, il est nécessaire que le réseau de neurones fasse de bonnes prédictions (ce que nous avons supposé dans cette section).

3.3.3 Implémentation de l'intelligence artificielle

Cette section décrit brièvement l'implémentation de l'intelligence artificielle. Pour réaliser celle-ci, nous devons d'abord concevoir une interface de programmation applicative (API) pour le jeu du démineur. Celle-ci se décompose en trois couches décrites ci-après.

1. La première couche contient une *grille sans masque*, c'est-à-dire une grille où toutes les cases sont démasquées (similaire à une sous-grille sans masque). Cette couche est représentée par la classe `Grid`. Cette dernière s'occupe de construire la grille et de placer les murs¹⁰ et les bombes (l'épaisseur des quatre murs ainsi que les positions des bombes sont reçues en paramètre). Elle génère également les marquages à partir des cases bombes. Nous trouvons dans cette classe des méthodes utilitaires permettant de récupérer des informations sur la grille comme la position des bombes ou encore le type d'une case à partir de sa position (case vide ou case bombe).
2. La deuxième couche contient une *grille avec masque* où au moins une case est une case masquée (comparable à une sous-grille avec masque). En réalité, il s'agit d'une grille de la première couche dont certaines cases sont masquées. La classe correspondante est `MaskedGrid` et étend la classe `Grid`. Elle regroupe, en plus des méthodes de la classe `Grid`, des méthodes relatives aux cases masquées. Nous retrouvons par exemple une méthode pour démasquer une case.
3. La troisième et dernière couche constitue le jeu du démineur. La classe `Minesweeper` représente celle-ci et contient un objet de type `MaskedGrid` (agrégation entre les deux classes). Cette classe comprend donc une grille avec masque et implémente les règles du jeu. Nous y trouvons par exemple le calcul du *score* du joueur que nous définissons simplement comme étant le nombre de cases vides démasquées dans la grille.

L'implémentation de l'intelligence artificielle découle directement de ce qui a été dit dans la [Section 3.3.2](#). Lors du premier tour, nous sélectionnons une case au hasard et la démasquons. Pour chaque tour qui suit, nous parcourons tout d'abord chaque case t de la grille du jeu et si t est une case masquée, nous extrayons la sous-grille de 5×5 où t est la case centrale. Nous obtenons donc une liste de sous-grilles. Ensuite, nous envoyons

¹⁰. Les murs ne font pas partie du jeu de base. Cependant, ils facilitent l'implémentation des sous-grilles vu que celles-ci les utilisent.

3.4 Résultats obtenus

celles-ci au réseau de neurones et récupérons les vecteurs de sortie prédicts. Et enfin, nous prenons celui où sa première composante est la plus petite et démasquons la case centrale de la sous-grille correspondant à ce vecteur.

3.3.4 Évaluation des résultats de l'intelligence artificielle

Dans cette section, nous présentons quelques méthodes d'évaluation dont le but est de connaître les performances de notre intelligence artificielle en pratique sur de vraies parties du jeu du démineur. Cela peut être également utile pour comparer deux intelligences artificielles entre elles.

Nous utilisons pour cela deux méthodes basées sur une liste de n scores. Cette dernière est simplement créée en faisant jouer l'intelligence artificielle sur n parties et en récupérant les n scores correspondants.

Notre première méthode consiste à calculer la répartition des scores à l'aide du minimum, du maximum, de la moyenne, du premier quartile, du deuxième quartile (c'est-à-dire la médiane) et du troisième quartile. Ces trois quartiles sont équivalents aux 25^e, 50^e et 75^e percentiles respectivement¹¹.

Notre seconde méthode d'évaluation repose simplement sur le calcul du taux de victoires correspondant au pourcentage de parties gagnées. Il nous suffit pour cela de calculer le nombre de victoires, ce qui est revient à compter le nombre de scores maximums atteignables parmi les n scores de la liste des scores. Dans notre cas, comme la grille du jeu est de taille 10×10 et contient 10 bombes et comme le score est égal au nombre de cases vides démasquées, le score maximum atteignable est de 90. Le taux de victoires wr est donc calculé comme suit:

$$wr = \frac{m}{n},$$

où m est le nombre de victoires et n est le nombre de parties jouées par l'intelligence artificielle (équivaut au nombre de scores compris dans la liste des scores).

3.4 Résultats obtenus

Dans cette section, nous exposons les résultats que nous avons obtenus pour l'évaluation du réseau de neurones et de l'intelligence artificielle.

Commençons par l'évaluation du réseau de neurones. Nous obtenons une erreur absolue moyenne de 0,182 et une erreur quadratique moyenne de 0,115. Également, la répartition des erreurs est montrée dans la Figure 15 (10 intervalles sont utilisés). Nous constatons qu'environ 62% des erreurs sont inférieures à 0,1, que 10% sont comprises entre 0,1 et 0,2 et que 28% sont supérieures à 0,2. Ce sont des résultats corrects, mais peut-être insuffisants. En effet, il y a environ 12% des erreurs qui sont supérieures à 0,6 et 5% supérieures à 0,9. Cela pourrait provoquer une baisse de performances de l'intelligence artificielle.

Regardons maintenant le diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées. Cela est montré dans la Figure 16. Les points présents dans celle-ci forment une sorte de « C ». Cela signifie que plus le nombre de cases vides démasquées est élevé, c'est-à-dire que plus d'informations sont disponibles pour le réseau, et plus

11. Le i^{e} percentile représente la valeur v de la donnée qui sépare les données en deux groupes: le premier contient les $i\%$ des données dont la valeur est inférieure à v et le second les $(100 - i)\%$ des données dont la valeur est supérieure à v .

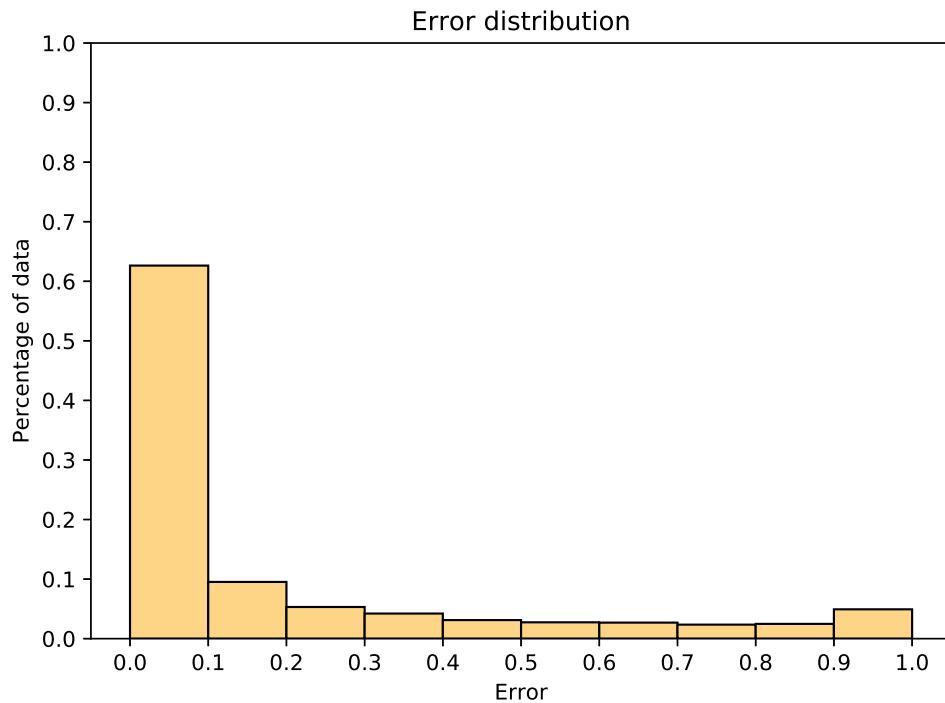


FIGURE 15 – Répartition des erreurs

les erreurs sont faibles en moyenne. Également, plus ce nombre diminue et plus les erreurs se dispersent dans l'intervalle $[0, 1]$. En d'autres mots, plus la quantité d'informations est faible, plus le réseau doute et plus les erreurs (et la première composante des vecteurs de sortie prédits) sont dispersées et « aléatoires ». Nous pouvons en conclure que la quantité d'informations disponibles pour le réseau influence bel et bien sa performance, ce qui est un comportement rationnel. Cependant, lorsque cette quantité est très faible, les erreurs devraient se situer idéalement vers 0.5. Dans notre cas, elles sont fortement dispersées; cela pourrait diminuer considérablement les performances de l'intelligence artificielle. Effectivement, prenons une sous-grille avec masque où la case centrale t est une case bombe et où une seule case vide est démasquée. Le réseau de neurones pourrait prédire une valeur extrêmement petite (proche de 0). Dans ce cas, la probabilité que t soit démasquée par l'intelligence artificielle est assez élevée, beaucoup plus que si la valeur prédictive aurait été de 0.5. Si cette case t est démasquée, alors la partie est perdue et donc terminée.

Passons désormais à l'évaluation de l'intelligence artificielle. Pour ce faire, nous allons utiliser nos méthodes d'évaluation et donc calculer le score pour n parties, où n est fixé à 1.000 dans notre cas. En parallèle de cela, nous comparerons notre intelligence artificielle à une *intelligence artificielle aléatoire* fonctionnant comme suit: pour un tour donné, parmi toutes les cases masquées, une case est choisie aléatoirement et est démasquée.

Considérons tout d'abord la répartition des scores montrée dans la Figure 17 sous forme d'un diagramme en boîte (*box plot* en anglais). Ce genre de graphique est composé comme suit:

- la boîte représente l'intervalle allant du premier quartile au troisième quartile;
- la ligne rouge horizontale correspond au deuxième quartile (c'est-à-dire la médiane);

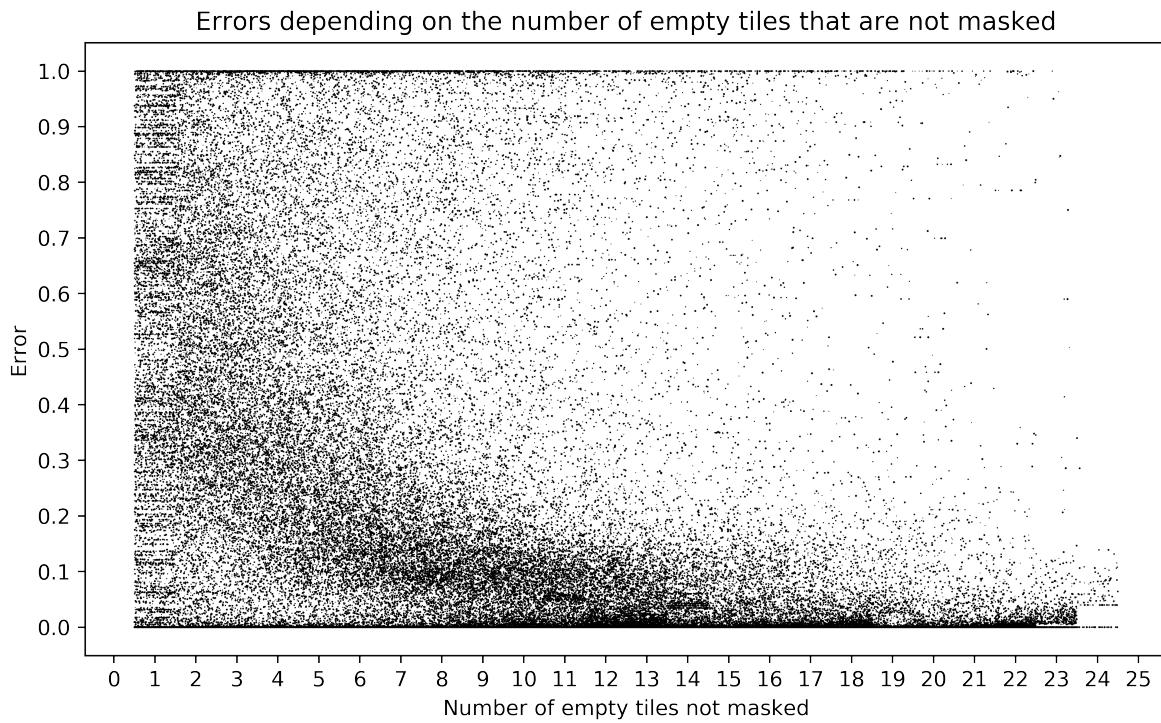


FIGURE 16 – Diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées

- la ligne orange jaune en pointillé représente la moyenne;
- les traits aux extrémités correspondent au minimum et au maximum après l'exclusion des *outliers* (petits cercles dans la figure) qui sont des « exceptions » ou encore des « cas particuliers » (données rares, peu représentatives).

Dans cette figure, nous voyons que, pour notre intelligence artificielle, le score moyen est égal à 68, la médiane à 82 et le troisième quartile à 88. Cela signifie que pour 50% des parties (respectivement 25%) le score est supérieur à 82 (respectivement 88). Aussi, comme le premier quartile vaut 61, le score est compris entre 61 et 82 pour 25% des parties et est inférieur à 61 pour également 25% des parties. Notre intelligence artificielle obtient donc de bons scores. Concernant l'intelligence artificielle aléatoire, la moyenne des scores est égale à 61, le premier quartile à 55, la médiane à 71 et le troisième quartile à 78. Ces mesures sont bien inférieures à celles de notre intelligence artificielle. Nous pouvons donc en conclure que les scores obtenus par cette dernière sont nettement supérieurs.

Terminons l'évaluation des deux intelligences artificielles par le taux de victoires. La notre a un taux de victoires de 18,2% tandis que l'intelligence artificielle aléatoire de 0,1%. Nous constatons que notre intelligence artificielle est bien plus efficace et que l'usage du réseau de neurones contribue à l'augmentation de performances. Toutefois, un taux de victoires de 18% n'est pas exceptionnel; environ une partie sur cinq uniquement est gagnée. Dans la prochaine section, nous tenterons d'améliorer notre intelligence artificielle.

3.4 Résultats obtenus

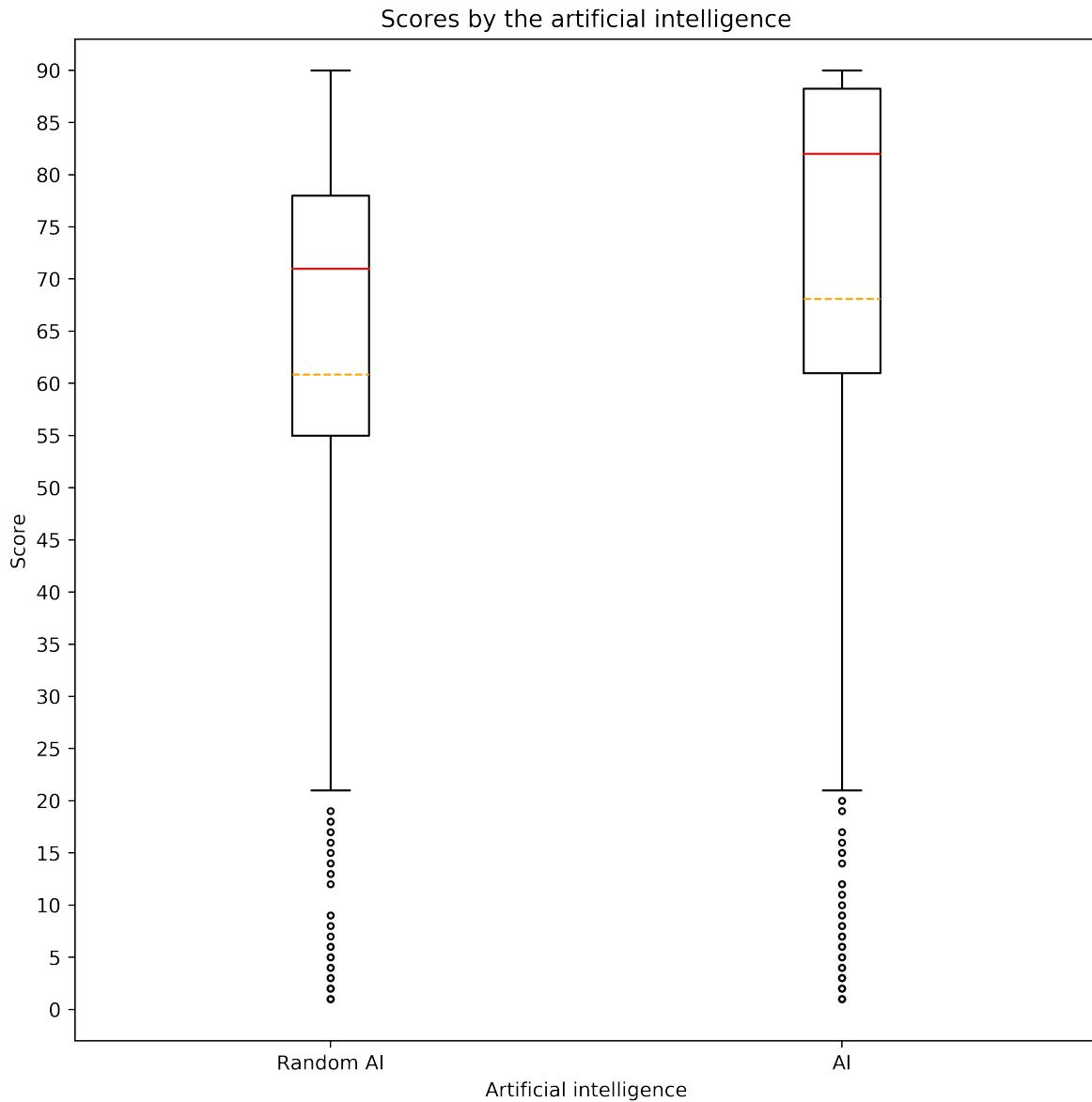


FIGURE 17 – Répartition des scores pour les deux intelligences artificielles

4 Amélioration de l'intelligence artificielle

Dans cette section, nous réaliserons différentes modifications et gardons celles qui améliorent les performances du réseau de neurones et de l'intelligence artificielle. Nous procéderons pas à pas: si une modification est conservée, alors elle sera maintenue tout au long du rapport pour toutes les prochaines modifications; il n'y a donc pas de retours en arrière. Concernant les modifications, nous retrouvons:

1. le ratio de sous-grilles où la case centrale est une case bombe;
2. l'ordre des sous-grilles du jeu d'entraînement;
3. le nombre de sous-grilles présentes dans le jeu d'entraînement;
4. les paramètres de l'entraînement du réseau de neurones;
5. la structure du réseau;
6. l'utilisation de jeux de données sans doublons;
7. la création d'une nouvelle intelligence artificielle utilisant des drapeaux;
8. l'ajout de deux paramètres dans l'intelligence artificielle du point précédent.

Dans cette section, nous ferons des analyses et des évaluations du réseau de neurones ainsi que de l'intelligence artificielle. Il est donc important de noter que toutes les erreurs commises par le réseau seront calculées en utilisant un jeu de test différent du jeu d'entraînement.

4.1 Ratio de sous-grilles où la case centrale est une case bombe

Notre première modification consiste à faire varier le ratio de sous-grilles où la case centrale est une case bombe. En effet, lorsque nous avons conçu le jeu d'entraînement, nous avions pris dans l'ordre 50.000 sous-grilles sans masque dans le jeu de données sans bombe et 50.000 autres dans le jeu de données avec bombe et avions généré 10 sous-grilles avec masque à partir de celles-ci. Cela correspond à un ratio de 0,5. Nous allons donc modifier ici ce ratio. Par exemple, nous pouvons prendre 70.000 sous-grilles sans masque dans le jeu de données sans bombe et 30.000 dans le jeu de données avec bombe, donnant donc un ratio de 0,7.

Le ratio de sous-grilles où la case centrale est une case bombe peut effectivement influencer les performances du réseau de neurones. Expliquons cela par un exemple en considérant un ratio r_1 de 0,5 et un ratio r_2 de 0,1. Avec r_1 , le réseau pourrait inférer que la probabilité qu'une case soit une case bombe est de 0,5, alors qu'avec r_2 , cette probabilité s'élèverait à 0,1. Dans notre cas, comme les grilles du jeu ont une taille de 10×10 et contiennent 10 bombes, cette probabilité est de 0,1 en réalité. Il est donc possible qu'un ratio inférieur à 0,5 et plus proche de 0,1 donne de meilleurs résultats.

Nous avons réalisé un diagramme de dispersion des erreurs en fonction du ratio de sous-grilles où la case centrale est une case bombe. Ce diagramme est présenté à la [Figure 18](#). Nous avons utilisé les ratios compris entre 0,25 et 0,75 avec un pas de 0,05. Malheureusement, contrairement à ce que nous avons imaginé dans le paragraphe précédent, les erreurs ne sont pas plus faibles lorsque le ratio diminue et se rapproche de 0,1. Également, nous constatons qu'aucun ratio ne semble fournir de meilleures performances pour le réseau de neurones et de ce fait pour l'intelligence artificielle aussi. Nous laissons donc ce ratio à 0,5 comme antérieurement.

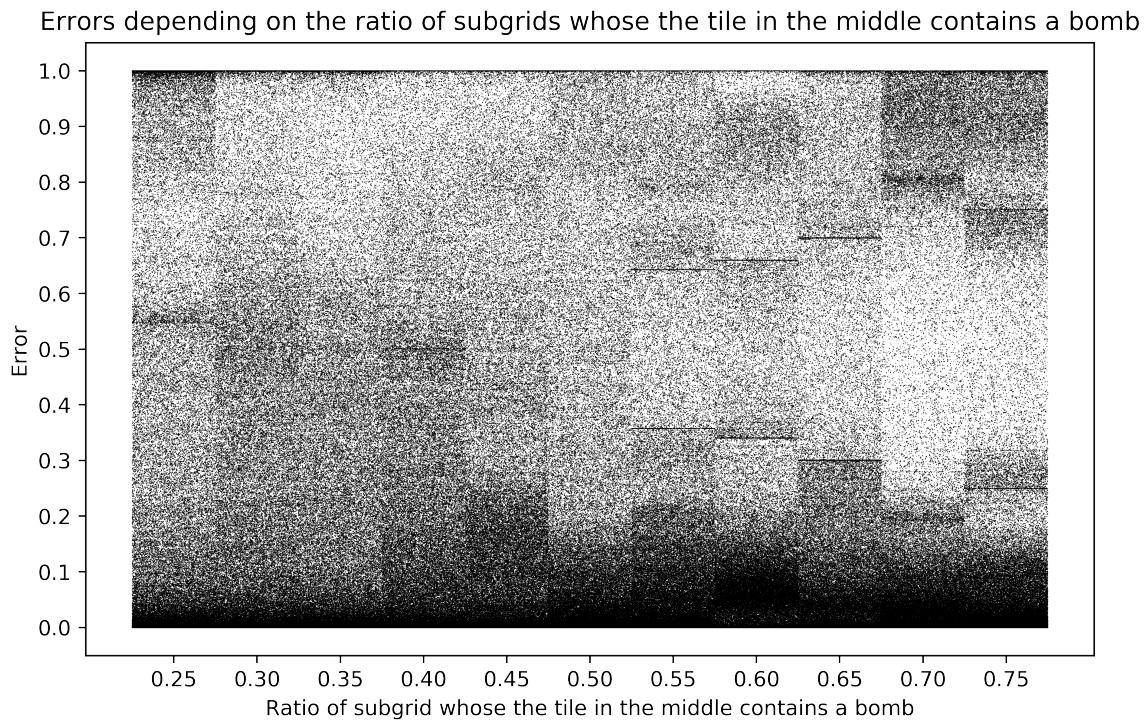


FIGURE 18 – Diagramme de dispersion des erreurs en fonction du ratio de sous-grilles où la case centrale est une case bombe

4.2 Ordre des sous-grilles dans le jeu d'entraînement

Nous allons désormais modifier l'ordre des sous-grilles avec masque dans le jeu d'entraînement. Actuellement, ce dernier est constitué de d'abord 500.000 sous-grilles avec masque où la case centrale est une case vide et d'ensuite 500.000 autres où la case centrale est une case bombe. Nous allons dans cette section modifier cet ordre et en essayer d'autres.

L'ordre des sous-grilles dans le jeu d'entraînement peut affecter les performances du réseau de neurones. En effet, dans le cas présent, la correction des poids du réseau se fait d'abord sur 500.000 sous-grilles où la case centrale est une case vide et puis sur 500.000 où la case centrale est une case bombe. Il serait peut-être plus intéressant d'alterner entre ces deux types de sous-grilles.

Dans cette analyse, nous avons utilisé sept ordres cités et décrits ci-après.

1. **NoBM BM**: contient d'abord 500.000 sous-grilles où la case centrale est une case vide et ensuite 500.000 où la case centrale est une case bombe. NoBM BM est l'ordre utilisé jusqu'ici.
2. **BM NoBM**: contient d'abord 500.000 sous-grilles où la case centrale est une case bombe et ensuite 500.000 où la case centrale est une case vide.
3. **Alternate n**: alterne entre n sous-grilles où la case centrale est une case vide et n sous-grilles où la case centrale est une case bombe. L'ordre est donc de la forme suivante:

$$\underbrace{\{e, e, \dots, e\}}_{n \text{ fois}}, \underbrace{\{b, b, \dots, b\}}_{n \text{ fois}}, \underbrace{\{e, e, \dots, e\}}_{n \text{ fois}}, \underbrace{\{b, b, \dots, b\}}_{n \text{ fois}}, \dots \}$$

où e est une sous-grille où la case centrale est une case vide et b est une sous-grille où la case centrale est une case bombe.

En particulier, nous avons utilisé les ordres **Alternate 1**, **Alternate 5**, **Alternate 10** et **Alternate 15**.

4. **Random**: les sous-grilles où la case centrale est une case vide et les sous-grilles où la case centrale est une case bombe sont mélangées aléatoirement; il s'agit d'un ordre aléatoire.

La [Figure 19](#) montre les erreurs en fonction de l'ordre des sous-grilles dans le jeu d'entraînement sous la forme d'un diagramme en boîte. Nous remarquons que la moyenne et les trois quartiles de l'ordre **NoBM BM** sont tous inférieurs à ceux des autres ordres. Cela signifie que les erreurs sont en moyenne plus faibles pour celui-ci. Comme cet ordre est celui que nous utilisons jusqu'à maintenant, nous n'effectuons pas de modification.

4.3 Nombre de sous-grilles contenues dans le jeu d'entraînement

Nous allons dans cette section modifier le nombre de sous-grilles présentes dans le jeu d'entraînement. Jusqu'à présent, ce dernier était construit en prenant 50.000 sous-grilles sans masque dans le jeu de données sans bombe et 50.000 sous-grilles dans le jeu de données avec bombe et en créant 10 sous-grilles avec masque pour chacune de ces 100.000 sous-grilles. Nous allons uniquement modifier le nombre de sous-grilles sans masque, c'est-à-dire que nous créons exactement le même nombre de sous-grilles avec masque par sous-grille sans masque et gardons la même proportion (ratio) de sous-grilles où la case centrale est une case bombe. Par exemple, nous calculerons les erreurs obtenues avec 500.000 sous-grilles sans masque. Dans ce cas, le jeu d'entraînement contiendra donc 2.500.000 sous-grilles avec masque où la case centrale est une case bombe et 2.500.000 autres où la case centrale est une case vide.

Modifier la taille du jeu d'entraînement peut impacter significativement les performances du réseau de neurones et de l'intelligence artificielle. En effet, augmenter cette taille signifie que le réseau de neurones rencontrera plus de situations (sous-grilles) lors de son entraînement. De ce fait, plus la taille est élevée, meilleures seront les performances. Néanmoins, il y a un effet de seuil. Si celui-ci est atteint, l'augmentation de la taille du jeu d'entraînement entraînera qu'une légère augmentation des performances.

La [Figure 20](#) montre l'évolution des erreurs en fonction du nombre de sous-grilles sans masque utilisées pour créer le jeu d'entraînement. Nous constatons qu'à partir de 100.000, l'augmentation du nombre de sous-grilles sans masque ne provoque qu'une faible diminution de l'erreur moyenne (effet de seuil). Par conséquent, nous pouvons fixer ce paramètre à 100.000, mais, comme le temps de calcul est encore raisonnable, nous le fixons à 1.000.000. Dès lors, à partir de maintenant, le jeu d'entraînement sera généré en prenant 500.000 sous-grilles sans masque dans le jeu de données sans bombe et 500.000 dans le jeu de données avec bombe et en créant 10 sous-grilles avec masque pour chacune.

Regardons maintenant l'effet de cette modification sur le réseau de neurones et sur l'intelligence artificielle. Commençons par le réseau de neurones. Nous obtenons à présent une erreur absolue moyenne de 0,179 au lieu de 0,182 et une erreur quadratique moyenne de 0,096 à la place de 0,115. La [Figure 21](#) montre la nouvelle répartition des erreurs. En comparaison avec celle obtenue avant cette modification ([Figure 15](#)), nous remarquons que la quantité d'erreurs situées dans les extrêmes (proches de 0 ou de 1) est moins importante

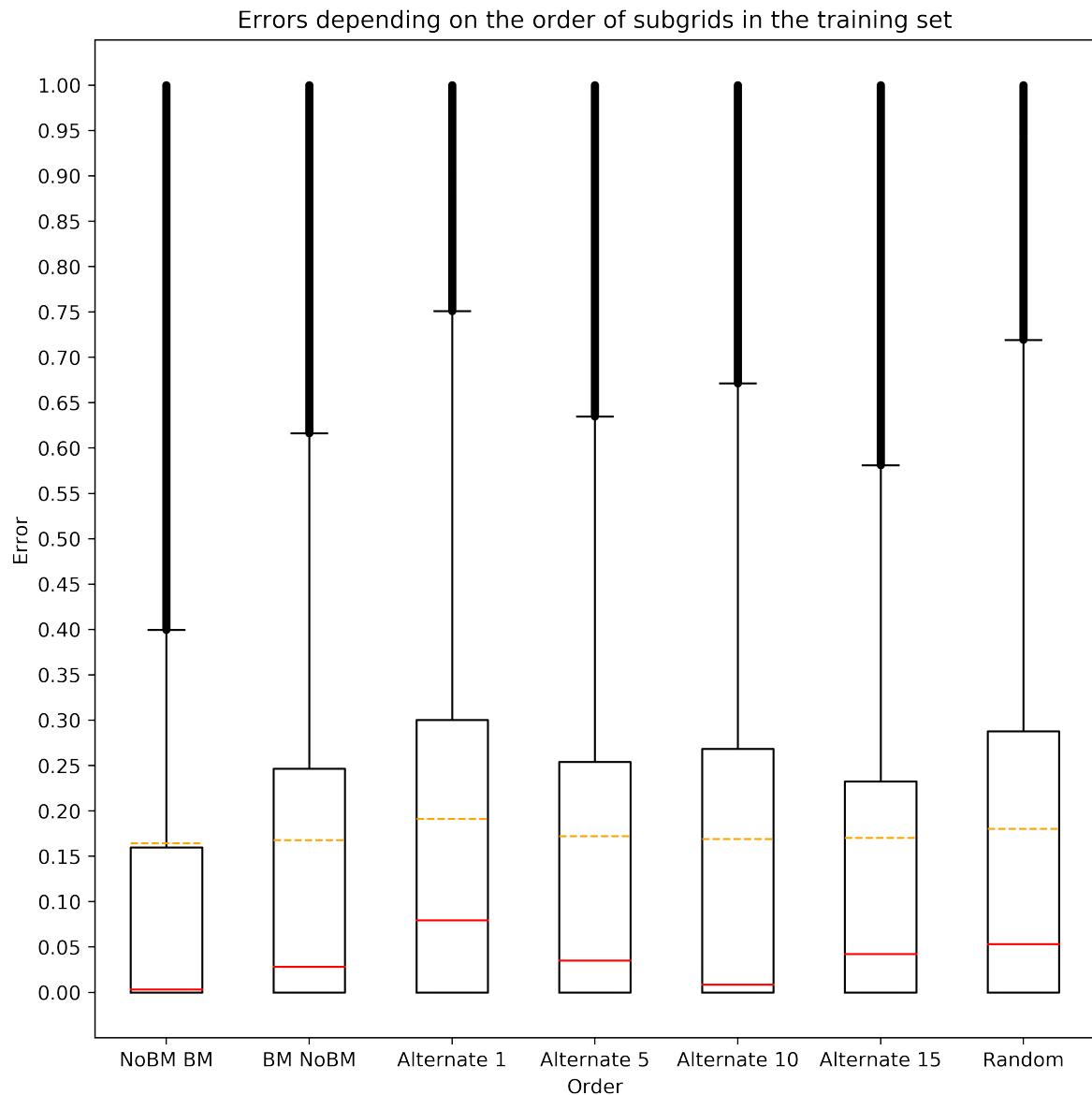


FIGURE 19 – Diagramme en boîte des erreurs en fonction de l'ordre des sous-grilles dans le jeu d'entraînement

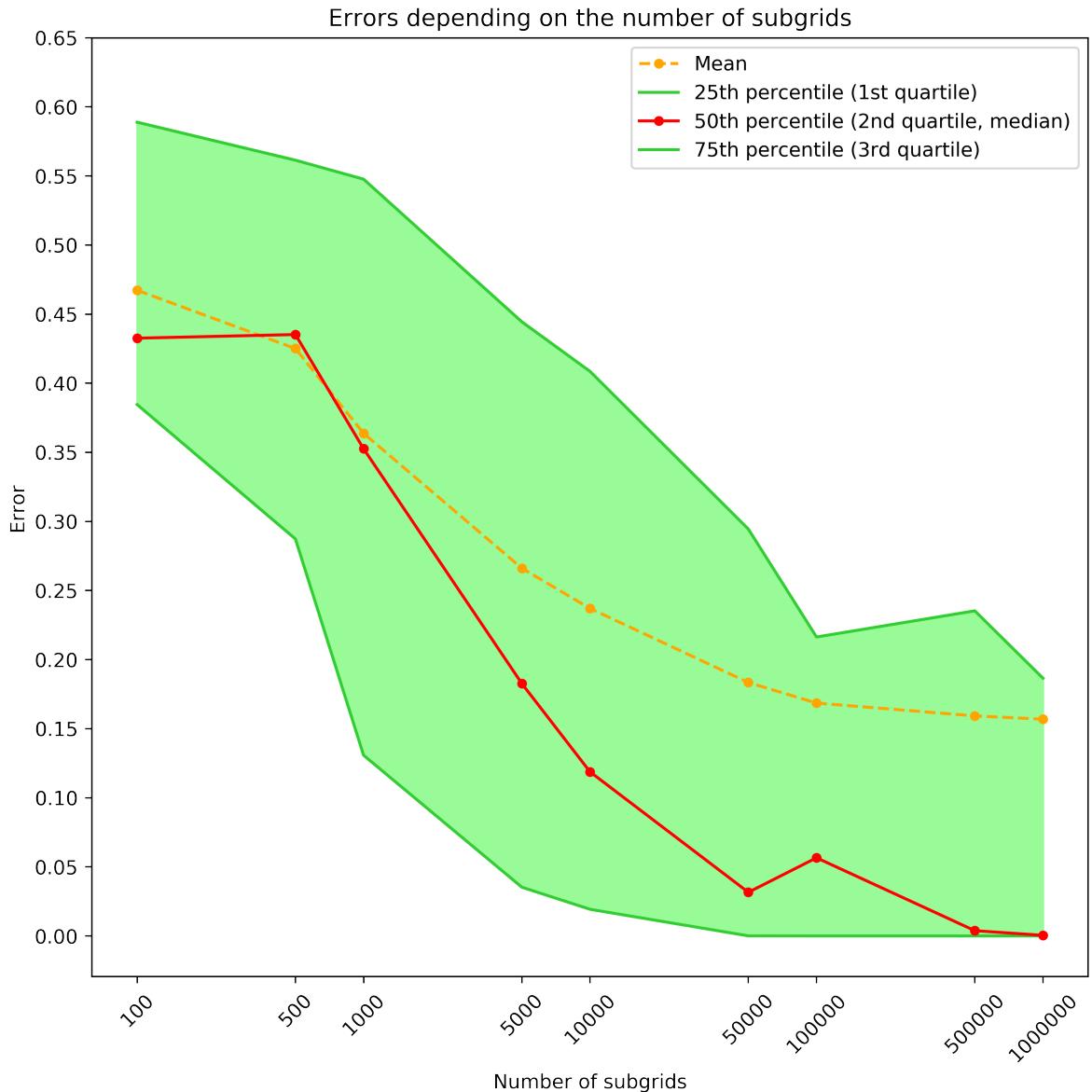


FIGURE 20 – Erreurs en fonction du nombre de sous-grilles sans masque permettant de générer le jeu d'entraînement

et qu’il y a davantage d’erreurs situées dans les alentours de 0,5. Également, les erreurs supérieures à 0,6 sont beaucoup moins abondantes: nous passons de 12% à 6% des erreurs. Concentrons-nous désormais sur le diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées montré à la Figure 22. Ce diagramme est assez similaire à celui que nous avions obtenu avant de réaliser la modification (Figure 16), excepté le fait que les erreurs ont plus tendance à être proches de 0,5 lorsque la quantité d’informations est faible. Comme dit dans la Section 3.4, cela est préférable, car les performances de l’intelligence artificielle pourraient diminuer de manière importante dans le cas contraire. En ce qui concerne celle-ci, la distribution des scores est très similaire par rapport à ce que nous avions obtenu antérieurement (voir la Figure 17). Nous constatons juste une légère baisse du premier quartile et une légère augmentation du troisième quartile. Également, le taux de victoires augmente faiblement; passant de 18,2% à 19,3%. Par conséquent, nous conservons cette modification.

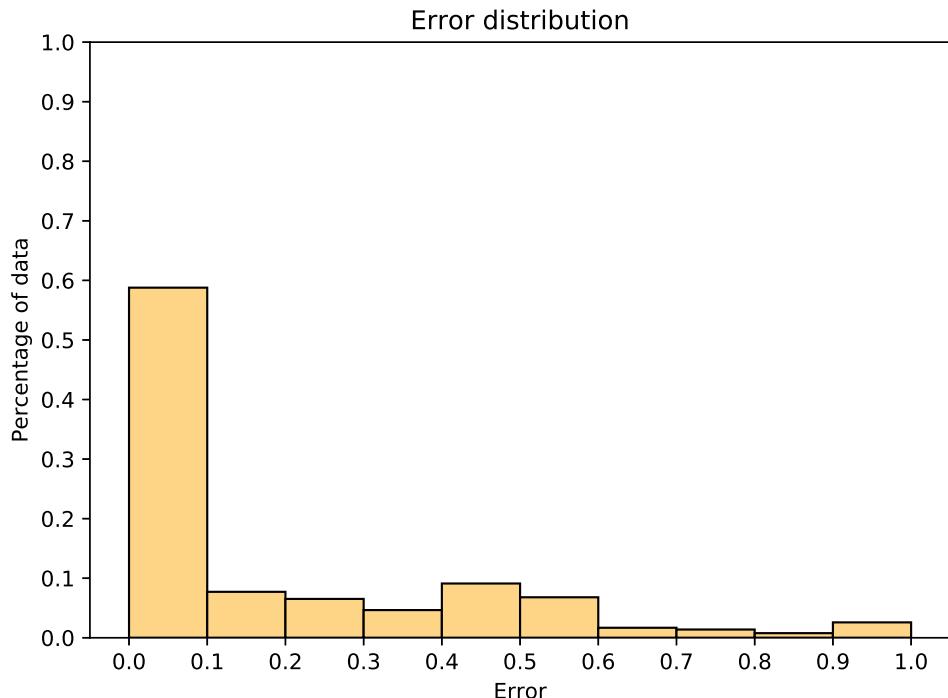


FIGURE 21 – Répartition des erreurs après avoir modifié la taille du jeu d’entraînement

4.4 Paramètres de l’entraînement du réseau de neurones

Nous allons maintenant modifier deux paramètres de l’entraînement du réseau de neurones: la taille du *batch* et le nombre d’itérations étant actuellement fixés à 10 et à 1 respectivement.

La taille du *batch* correspond à la fréquence de mise à jour des poids des connexions du réseau de neurones. Une valeur plus élevée pour ce paramètre permet d’éviter que le réseau ne se spécialise sur des cas particuliers. Toutefois, si cette valeur est trop élevée, le réseau pourrait ne pas remarquer certains détails ou certaines particularités importantes. Le nombre d’itérations *ep* permet de présenter *ep* fois tous les exemples du jeu d’entraînement

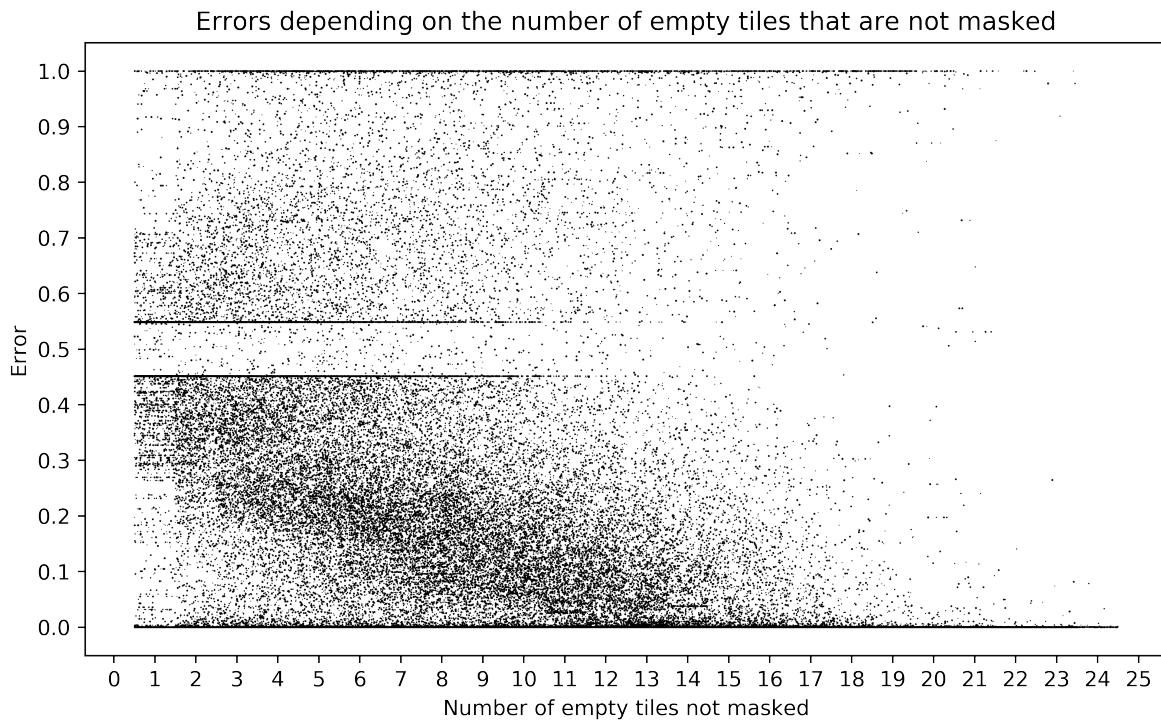


FIGURE 22 – Diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées après avoir augmenté la taille du jeu d'entraînement

au réseau de neurones. Si ce nombre est très élevé, le phénomène de *surapprentissage* (*overfitting* en anglais) peut se produire, signifiant que le réseau se spécialise beaucoup trop sur les exemples et ne les généralise pas suffisamment [11]. Ces deux paramètres sont en conséquence très importants et peuvent altérer les performances.

En ce qui concerne la taille du *batch*, nous en avons utilisé plusieurs allant de 10 à 10.000. Nous pouvons voir son effet sur les erreurs à la Figure 23. Nous voyons qu'une taille du *batch* fixée à 2.000 permet d'obtenir des erreurs plus faibles en moyenne. Nous fixons donc ce paramètre à cette valeur.

Regardons maintenant comment les erreurs évoluent en fonction du nombre d'itérations (avec 2.000 comme taille du *batch*). Cela est montré à la Figure 24. Nous constatons que la moyenne ainsi que les trois quartiles des erreurs sont les plus faibles lorsque le nombre d'itérations est égal à 6. Par conséquent, nous fixons ce paramètre à 6.

À présent que nous avons modifié la taille du *batch* et le nombre d'itérations, évaluons les performances du réseau de neurones et de l'intelligence artificielle. Considérons tout d'abord le réseau de neurones. L'erreur absolue moyenne passe de 0,179 à 0,119 et l'erreur quadratique moyenne de 0,096 à 0,062. Analysons la répartition des erreurs, présentée à la Figure 25. Environ 73% des erreurs sont inférieures à 0,1, ce qui correspond à une augmentation de 14% si nous comparons à la répartition obtenue avant cette modification (Figure 21). Également, environ 6% des erreurs sont supérieures à 0,6. Le diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées, montré à la Figure 26, ressemble assez bien au précédent (Figure 22). Cependant, nous remarquons que lorsque la quantité d'informations est très faible, les erreurs ont encore plus tendance

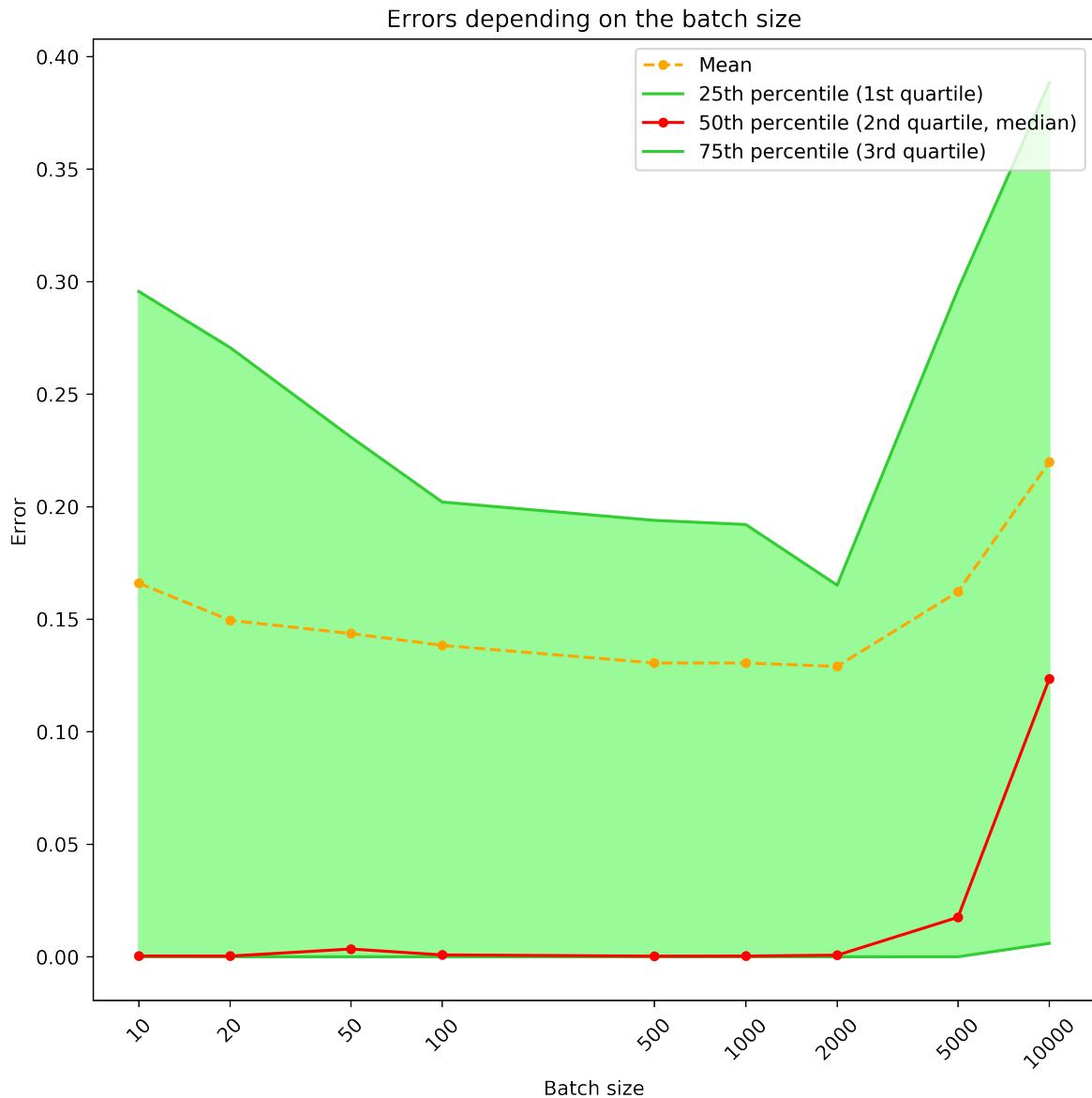


FIGURE 23 – Erreurs en fonction de la taille du *batch*

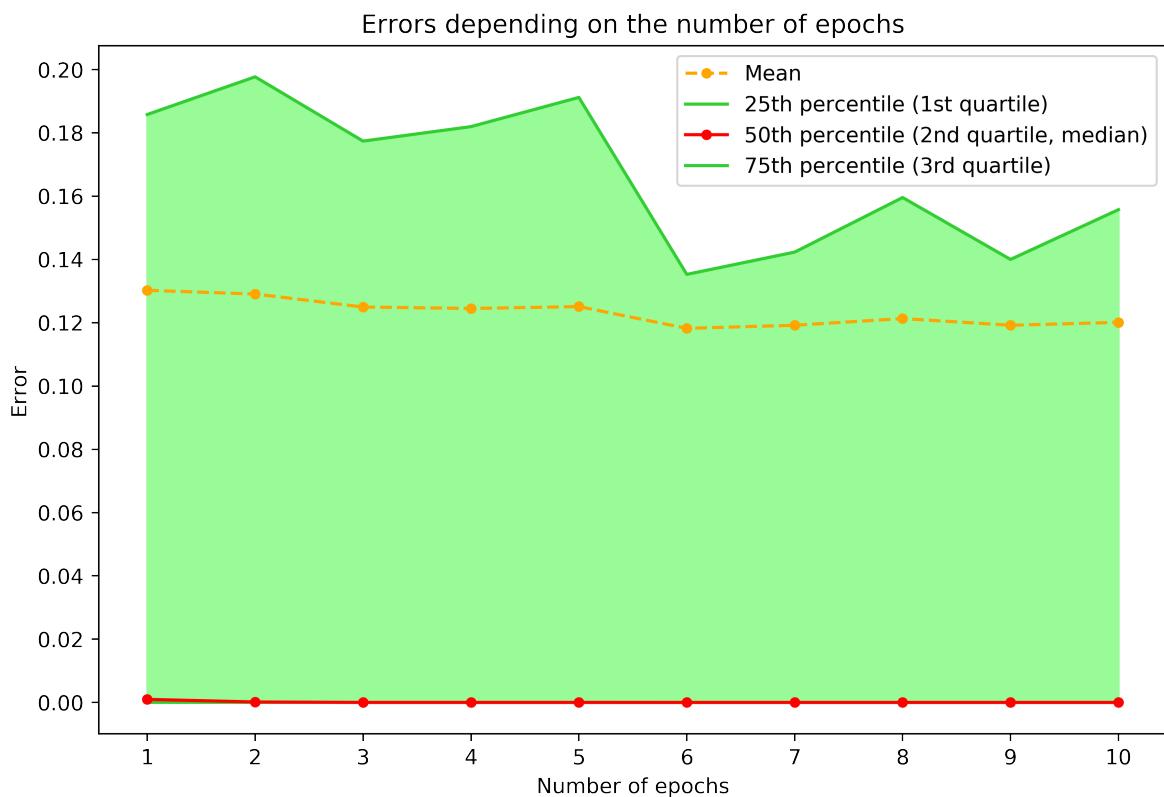


FIGURE 24 – Erreurs en fonction du nombre d’itérations

à se situer autour de 0,5. Quant à l'intelligence artificielle, la moyenne des scores obtenus est bien supérieure, passant de 67 à 76. Les trois quartiles augmentent également; le premier vaut désormais 79 et les deux autres 90. Comme 90 est le score maximum et correspond à une victoire, plus de 50% des parties jouées par l'intelligence artificielle sont gagnées. Confirmons cela en regardant le taux de victoires. Il est à présent de 58,6%, ce qui représente une différence importante de 39,3% comparée à précédemment.

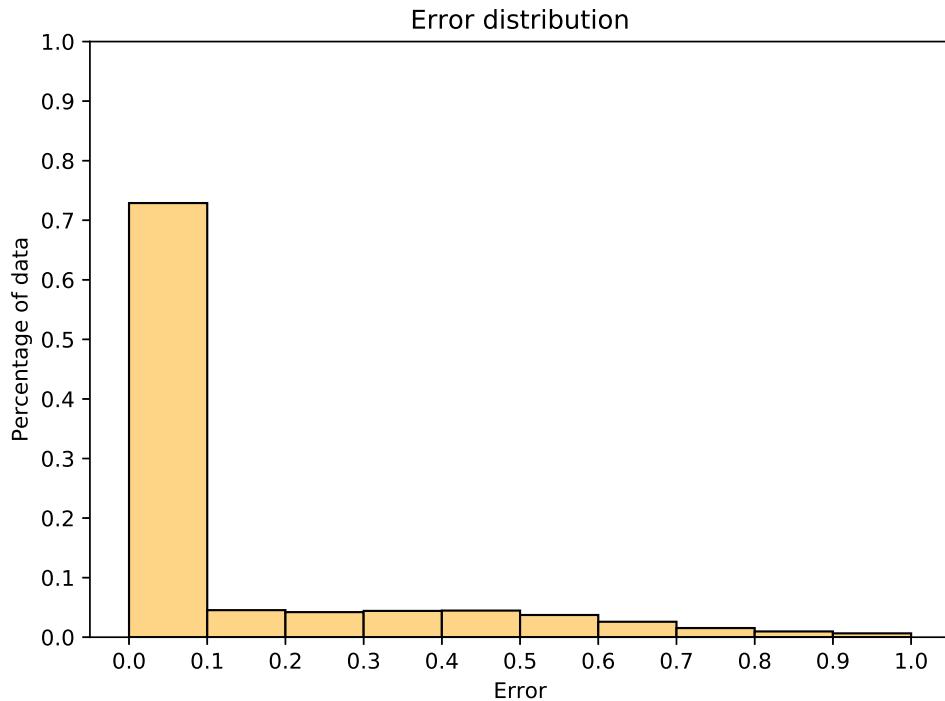


FIGURE 25 – Répartition des erreurs après avoir modifié les paramètres de l'entraînement du réseau de neurones

4.5 Nouvelle structure du réseau de neurones

Dans cette section, nous allons modifier la structure du réseau de neurones. Notre structure actuelle contient trois couches cachées contenant 300, 256 et 128 neurones respectivement.

Le nombre de neurones peut avoir une grande influence sur les performances. Effectivement, une quantité plus importante peut permettre au réseau d'intercepter plus d'informations et de repérer certaines subtilités d'un problème.

L'objectif ici est d'uniquement voir si l'augmentation du nombre de neurones total dans la structure améliore ou non les performances pour notre problème. Dès lors, nous décidons arbitrairement de doubler le nombre de neurones de chaque couche cachée et d'ajouter une nouvelle couche cachée contenant 1.024 neurones entre la première et la deuxième. Les fonctions d'activation restent inchangées. Le Tableau 2 résume la nouvelle structure du réseau de neurones ainsi obtenue.

En ce qui concerne l'évaluation du réseau de neurones, les constatations sont très similaires à précédemment. En effet, l'erreur absolue moyenne, l'erreur quadratique moyenne

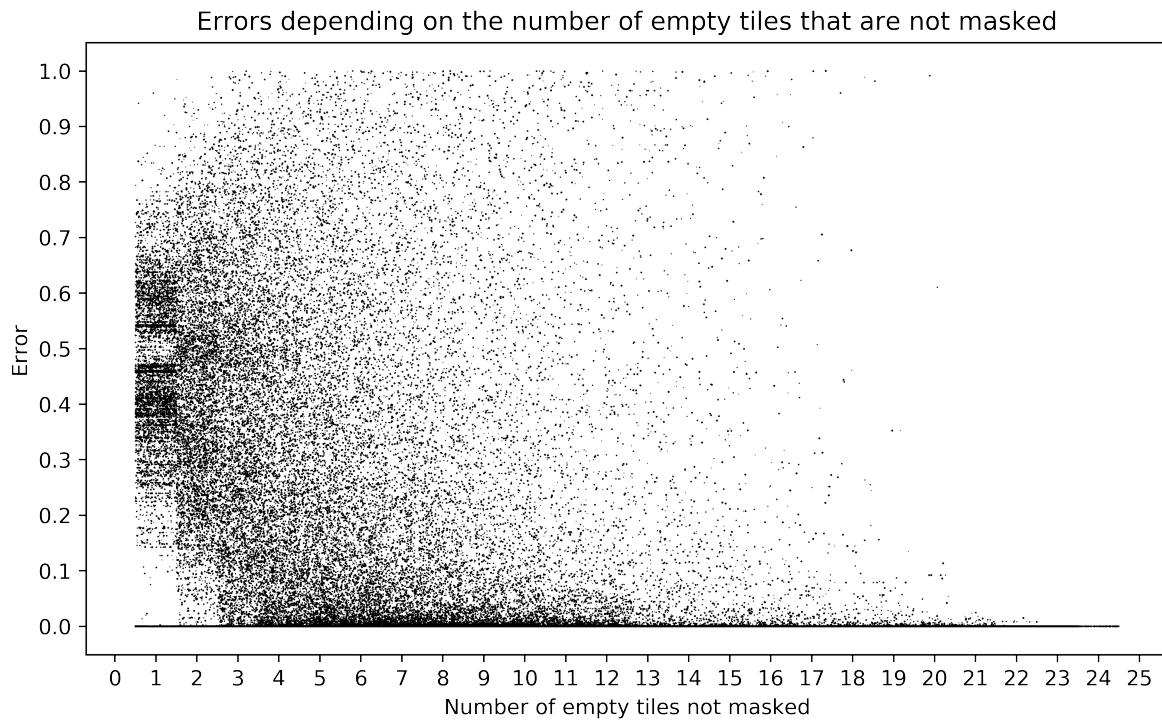


FIGURE 26 – Diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées après avoir modifié les paramètres de l’entraînement du réseau

Couche \ Paramètre	Nombre de neurones	Fonction d’activation
Couche d’entrée	25	—
1^{re} couche cachée	600	Unité de rectification linéaire
2^e couche cachée	1.024	Unité de rectification linéaire
3^e couche cachée	512	Unité de rectification linéaire
4^e couche cachée	256	Unité de rectification linéaire
Couche de sortie	1	Sigmoïde

Tableau 2 – Nouvelle structure du réseau de neurones

ainsi que la répartition des erreurs sont équivalentes à ce qui a été obtenu avant. Concernant l'intelligence artificielle, les mesures sont également très similaires. La moyenne des scores diminue très légèrement; elle vaut désormais 75 alors que, avant le changement de structure, elle était de 76. Le taux de victoires baisse lui aussi faiblement, passant de 58,6% à 58,4%. Nous pouvons en conclure que le nombre de neurones n'influe quasiment pas sur les performances dans notre cas. Par conséquent, nous ne conservons pas cette modification.

4.6 Utilisation de jeux de données sans doublons

Modifions maintenant le jeu de données sans bombe et le jeu de données avec bombe qui sont utilisés pour créer le jeu d'entraînement. Présentement, ceux-ci contiennent des doublons, c'est-à-dire que des mêmes sous-grilles sont présentes plusieurs fois. Parmi les 500.000 sous-grilles sans masque que nous prenons dans le jeu de données sans bombe (le jeu de données avec bombe respectivement), environ 47% (66% respectivement) sont des doublons. Cela est assez élevé. Dans cette section, nous allons donc créer deux nouveaux jeux de données sans doublons et observerons comment cela affecte les performances.

La quantité de doublons peut altérer les performances. Effectivement, utiliser des jeux de données sans doublons permet au réseau de neurones de rencontrer plus de situations et peut en conséquence le rendre plus efficace en pratique.

La création d'un jeu de données sans doublons contenant n sous-grilles est assez simple. Soit la i^{e} itération. Alors, le jeu de données ds actuellement en construction contient $i - 1$ sous-grilles. Il nous suffit de générer une sous-grille sg et de vérifier si elle est déjà présente dans ds . Si ce n'est pas cas, alors nous ajoutons sg dans ds et passons à l'itération suivante (si $i < n$). Dans le cas contraire, nous regénérerons une nouvelle sous-grille et vérifions de nouveau si elle est présente dans ds .

Regardons les performances du réseau de neurones et de l'intelligence artificielle. Commençons avec le réseau de neurones. L'erreur absolue moyenne est désormais de 0,145 au lieu de 0,119 et l'erreur quadratique moyenne de 0,071 plutôt que de 0,062. La [Figure 27](#) contient la répartition des erreurs. Nous constatons que les erreurs inférieures à 0,1 sont moins nombreuses, environ une baisse de 6% est à déplorer. Toutefois, uniquement 4% des erreurs sont supérieures à 0,6, représentant une diminution de 2%. Également, nous voyons que beaucoup plus d'erreurs sont proches de 0,5. Le diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées est montré à la [Figure 28](#). Nous remarquons que les erreurs sont encore plus concentrées vers 0,5 quand la quantité d'informations est faible. Cela pourrait améliorer les performances de l'intelligence artificielle. Vérifions cela avec la répartition des scores et le taux de victoires. Concernant la répartition, la moyenne et les deux derniers quartiles sont toujours de 76 et de 90 respectivement et le premier quartile est maintenant de 80 à la place de 79. Quant au taux de victoires, celui-ci passe de 58,6% à 62,5%. Comme les résultats sont globalement meilleurs, nous décidons de garder cette modification.

4.7 Intelligence artificielle utilisant des drapeaux

Dans cette section, nous allons concevoir une nouvelle intelligence artificielle utilisant des drapeaux. Nommons-la *intelligence artificielle avec drapeaux* et appelons l'intelligence

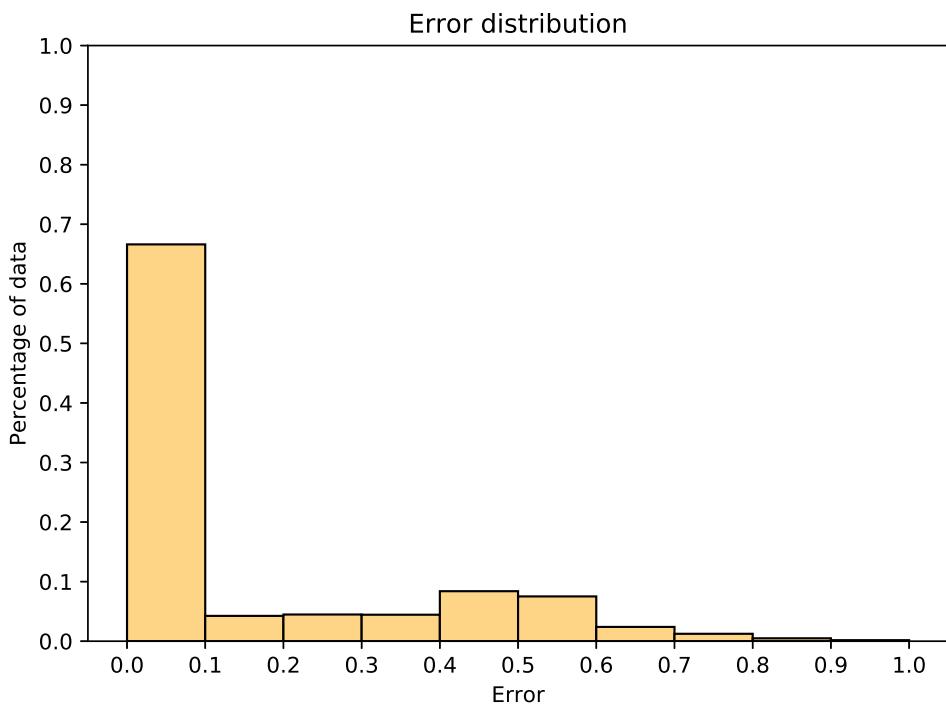


FIGURE 27 – Répartition des erreurs avec l'utilisation de jeux de données sans doublons

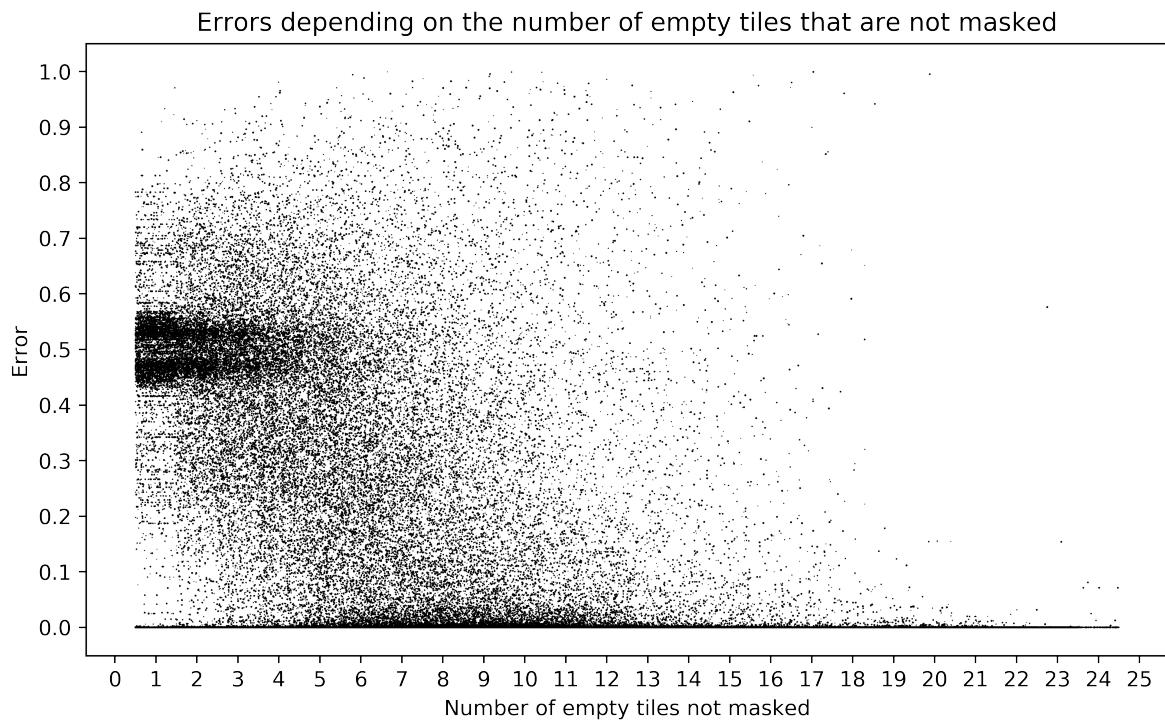


FIGURE 28 – Diagramme de dispersion des erreurs en fonction du nombre de cases vides démasquées avec l'utilisation de jeux de données sans doublons

artificielle utilisée jusqu'à présent *intelligence artificielle sans drapeaux*. Pour rappel, cette dernière procède en trois étapes. Premièrement, elle extrait les sous-grilles de 5×5 où la case centrale est une case masquée. Deuxièmement, elle les envoie au réseau de neurones et récupère les vecteurs de sortie prédicts. Et troisièmement, elle démasque la case dont la première composante du vecteur de sortie prédict correspondant est la plus faible. Cette intelligence artificielle ainsi que le réseau de neurones lié n'utilisaient donc pas de drapeaux.

Pour réaliser l'intelligence artificielle avec drapeaux, il est nécessaire de créer un nouveau réseau de neurones adapté pour celle-ci. La structure ainsi que les paramètres de ce dernier sont identiques au réseau de neurones de l'intelligence artificielle sans drapeaux. Seules les sous-grilles avec masque contenues dans le jeu d'entraînement sont modifiées (les jeux de données, l'ordre des sous-grilles, *etc.* restent inchangés). À partir d'une sous-grille *sg* sans masque, nous générerons n sous-grilles avec masque ($n = 10$ dans notre cas). Une sous-grille *msg* avec masque est créée en copiant le contenu de *sg* dans *msg* et en masquant certaines cases libres (cases ne contenant pas de mur). Plus précisément, nous masquons tout d'abord la case centrale de *msg*. Ensuite, nous masquons m cases libres aléatoires, où m est un nombre aléatoire. Et enfin, parmi toutes les cases démasquées restantes, nous masquons celles qui sont des cases bombes et plaçons un drapeau dessus. De cette façon, nous obtenons des sous-grilles vraisemblables. En effet, grâce à cela, le réseau de neurones rencontrera des situations où certaines cases bombes sont repérées et des situations où les autres cases bombes ne le sont pas, ce qui correspond à la réalité.

Maintenant que nous avons un réseau de neurones conçu et entraîné pour l'intelligence artificielle avec drapeaux, nous pouvons parler de cette dernière. Au premier tour, celle-ci sélectionne une case aléatoire de la grille du jeu et la démasque. Pour chacun des tours suivants, pour chaque case masquée t ne contenant pas de drapeau dans la grille du jeu courante, elle extrait la sous-grille *sg* de 5×5 où t est la case centrale de *sg*, l'envoie au réseau de neurones et récupère le vecteur de sortie prédict. Considérons la valeur v_{min} et l'index i_{min} du vecteur dont la première composante est la plus petite et la valeur v_{max} et l'index i_{max} du vecteur dont la première composante est la plus grande. Si $v_{min} < (1 - v_{max})$, c'est-à-dire que v_{min} est plus proche de 0 que v_{max} est proche de 1, alors l'intelligence artificielle démasque la i_{min} ^e case masquée ne contenant pas de drapeau. Dans le cas contraire, elle pose un drapeau sur la i_{max} ^e case masquée ne contenant pas de drapeau. En faisant cela, nous réalisons l'action la plus sécurisée, car, comme expliqué dans une section antérieure, le réseau prédit 0 quand il « pense » que la case centrale d'une sous-grille est une case vide et 1 quand il s'agit d'une case bombe. Nous avons cependant un cas particulier à traiter. Effectivement, si au tour courant un drapeau est posé sur toutes les cases masquées, alors l'intelligence artificielle retire tous les drapeaux, récupère l'index i_{min} comme décrit ci-dessus et démasque la i_{min} ^e case masquée. De cette façon, nous forçons l'intelligence artificielle à démasquer une case et cela permet également d'éviter que le programme ne tourne en rond indéfiniment.

Passons à présent à l'évaluation de cette nouvelle intelligence artificielle. En ce qui concerne la répartition des scores, la moyenne baisse de 7, passant de 76 à 69, le premier quartile diminue de 18, passant de 80 à 62 et les deux autres quartiles sont toujours de 90. Malgré une diminution des scores, le taux de victoires augmente; il passe de 62,5% à 67%. Nous conservons par conséquent cette intelligence artificielle.

4.8 Modification de l'intelligence artificielle avec drapeaux

Dans cette section, nous allons modifier l'intelligence artificielle avec drapeaux, présentée dans la section précédente, en ajoutant deux paramètres à celle-ci. À chaque tour, elle a deux choix: soit démasquer une case, soit poser un drapeau sur une case. Les deux paramètres ajoutés à l'intelligence artificielle permettent d'influencer cette décision.

Commençons avec le premier paramètre nommé *playful level*. Plus la valeur pour ce paramètre est élevée et plus l'intelligence artificielle « préférera » démasquer des cases plutôt que de poser des drapeaux. Inversement, plus cette valeur est basse et plus elle « préférera » poser des drapeaux au lieu de démasquer des cases. La valeur neutre est de 1 signifiant que l'intelligence artificielle n'a aucune préférence, tandis que la valeur minimum est de 0 (des drapeaux seront toujours posés) et la valeur maximum est de 2 (aucun drapeau ne sera utilisé). Bien entendu, cela n'a pas beaucoup de sens de fixer une de ces deux valeurs extrêmes pour ce paramètre dans le cadre d'une intelligence artificielle qui utilise des drapeaux.

Le second paramètre s'appelle *flag threshold*. Comme son nom l'indique, il s'agit du seuil qu'une valeur prédite par le réseau de neurones doit dépasser pour que l'intelligence artificielle soit autorisée à poser un drapeau. De ce fait, les valeurs possibles pour ce paramètre s'étendent de 0 (les drapeaux seront toujours autorisés) à 1 (aucun drapeau ne sera autorisé).

Le fonctionnement de l'intelligence artificielle avec drapeaux utilisant ces deux paramètres est très similaire à ce qui a été expliqué dans la section précédente. La seule différence se situe au niveau du choix fait par celle-ci juste après avoir calculé les deux valeurs v_{min} et v_{max} et les deux index i_{min} et i_{max} . Désormais, celle-ci démasquera la i_{min}^e case masquée ne contenant pas de drapeau si $(v_{min} < (pl - v_{max})) \vee (v_{max} \leq ft)$, où pl est la valeur du paramètre *playful level* et ft la valeur du paramètre *flag threshold*, et posera un drapeau sur la i_{max}^e case masquée ne contenant pas de drapeau dans le cas contraire (si $(v_{min} \geq (pl - v_{max})) \wedge (v_{max} > ft)$).

Utilisons différentes combinaisons des deux paramètres *playful level* et *flag threshold* et découvrons celle qui permet d'obtenir le meilleur taux de victoires. Les Figures 29 et 30 montrent l'évolution du taux de victoires en fonction de ceux-ci. Ce sont des graphiques à trois dimensions dont la première correspond aux valeurs du paramètre *playful level*, la seconde aux valeurs du paramètre *flag threshold* et la troisième aux taux de victoires représentés par une couleur. Plus cette couleur est verte et plus le taux de victoires est élevé, alors que plus celle-ci est rouge et plus le taux de victoires est faible. Nous constatons que les taux de victoires les plus élevés sont obtenus lorsque l'intelligence artificielle « préfère » démasquer des cases plutôt que de poser des drapeaux (valeur du paramètre *playful level* supérieure à 1). Également, une valeur élevée pour le paramètre *flag threshold* (supérieure à 0,9) permet d'obtenir un meilleur taux de victoires. Cependant, si celle-ci atteint 1, c'est-à-dire qu'aucun drapeau n'est utilisé, le taux de victoires est faible. En d'autres mots, cela signifie que l'usage des drapeaux apporte un gain de performances non négligeable pour cette intelligence artificielle. Le plus haut taux de victoires enregistré, obtenu avec une valeur de 1,15 pour le paramètre *playful level* et une valeur de 0,96 pour le paramètre *flag threshold*, est de 86,6%. Ces deux paramètres ont donc permis d'augmenter le taux de victoires de 19,6%. Nous gardons par conséquent cette modification.

Cette intelligence artificielle est la meilleure que nous ayons conçue. À titre de comparaison, l'intelligence artificielle aléatoire discutée dans la Section 3.4 permet d'obtenir

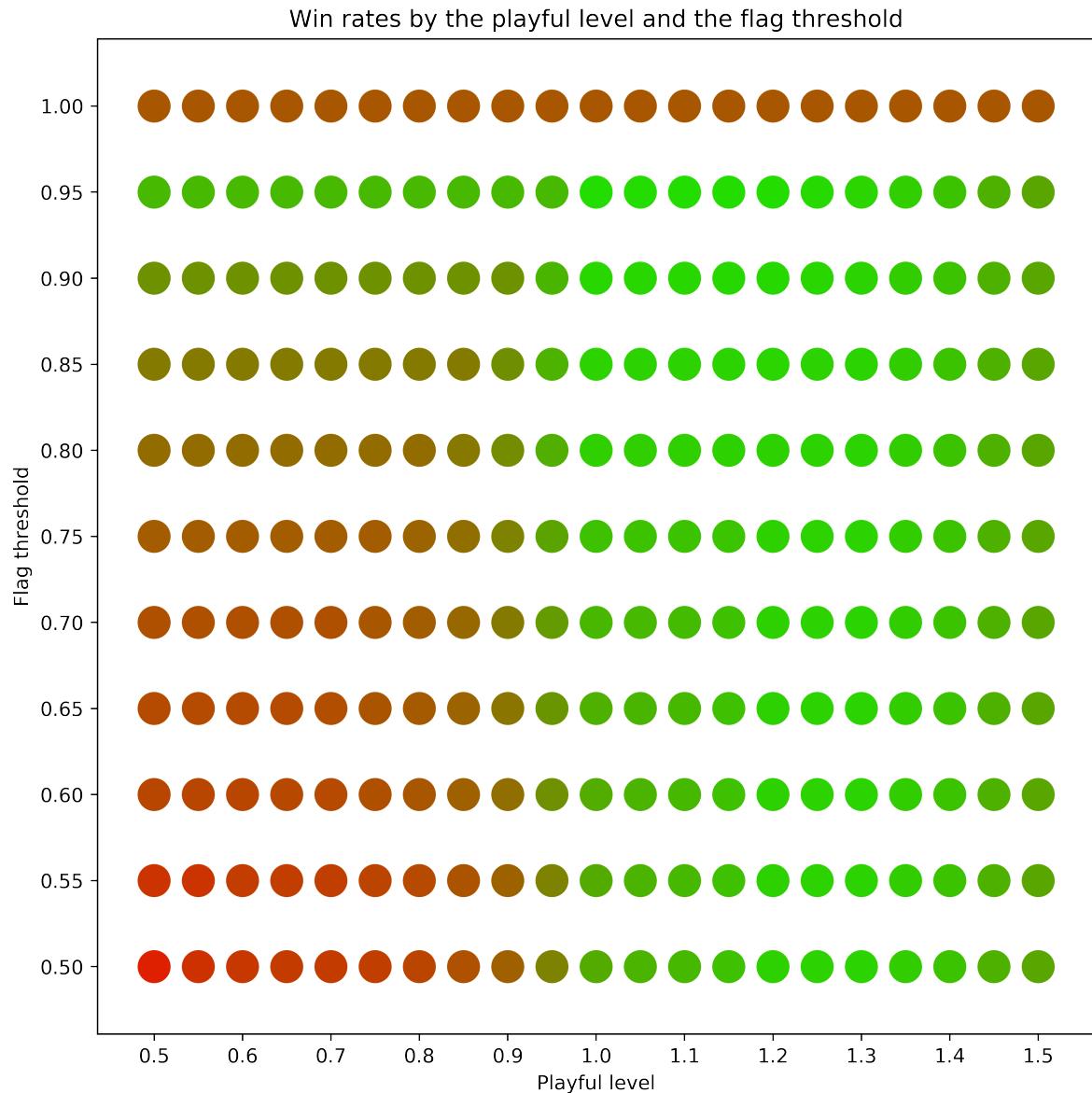


FIGURE 29 – Taux de victoires en fonction des paramètres *playful level* et *flag threshold*

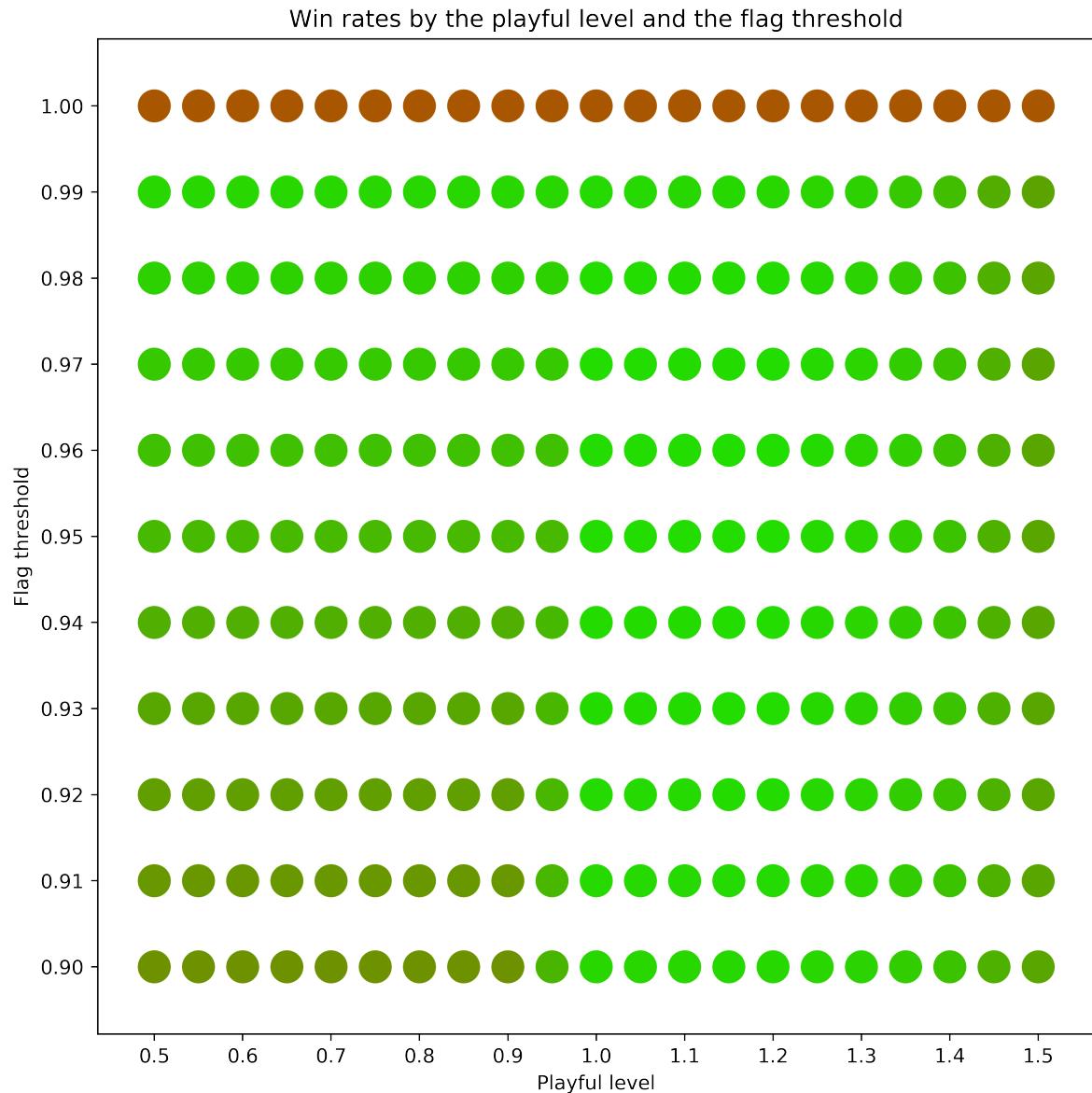


FIGURE 30 – Taux de victoires en fonction des paramètres *playful level* et *flag threshold*

un taux de victoires de 0,1% et l'intelligence artificielle sans drapeaux obtenue à la [Section 4.6](#) un taux de victoires de 62,5%. La répartition des scores de ces trois intelligences artificielles est montrée dans la [Figure 31](#). Sans surprise, nous y voyons que l'intelligence artificielle avec drapeaux possède les meilleurs scores avec une moyenne de 82. À titre informatif, les auteurs de la référence [9], ayant utilisé une approche similaire à la nôtre, et l'auteur de la référence [10] ont atteint un taux de victoires d'un peu plus de 75% et de 92,5% respectivement dans des conditions similaires aux nôtres (taille de la grille du jeu et densité de bombes très proches).

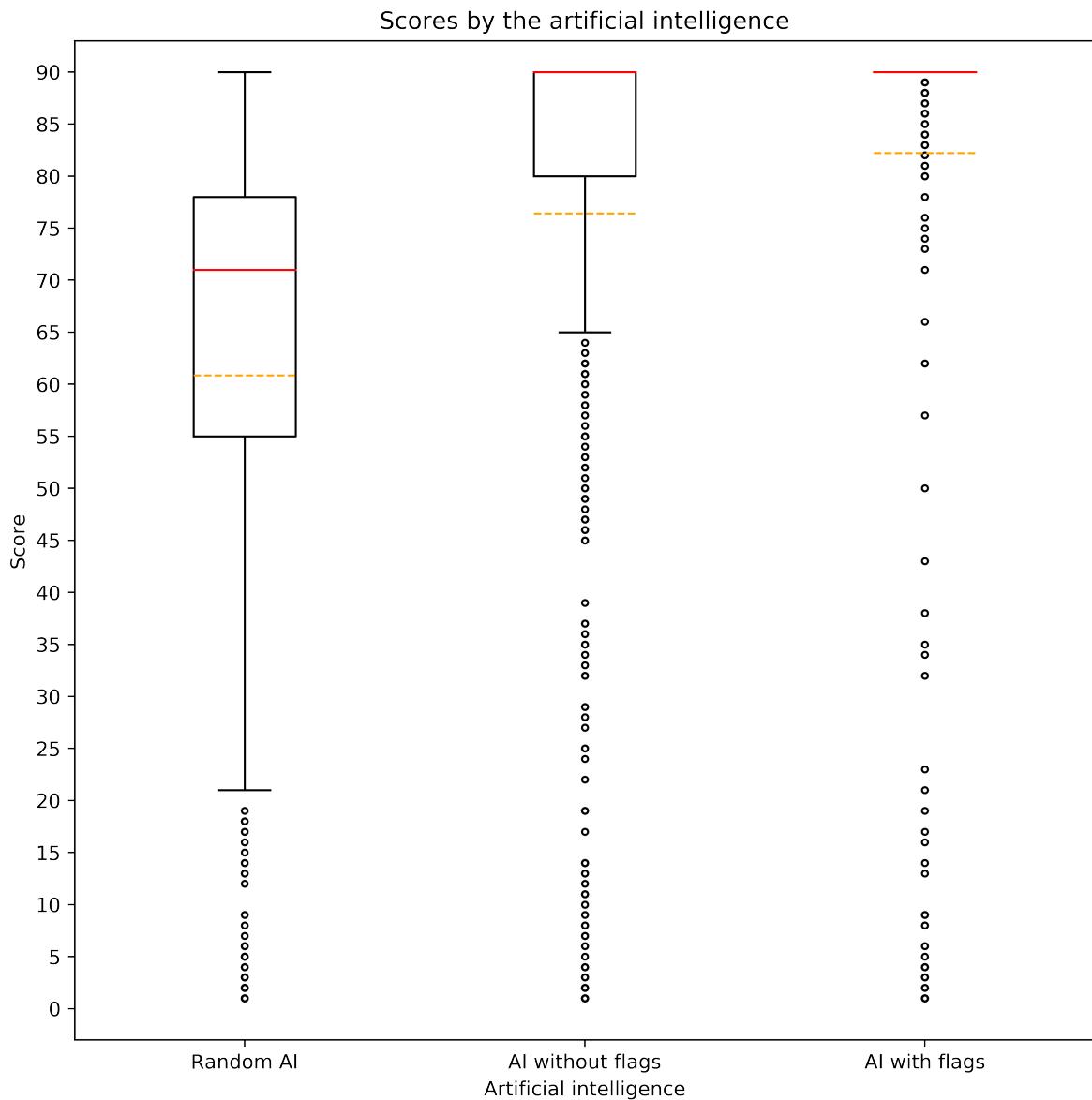


FIGURE 31 – Répartition des scores des trois intelligences artificielles réalisées

5 Performances de l'intelligence artificielle après suppression de certaines hypothèses

Dans cette section, nous allons supprimer l'hypothèse faite sur la taille des grilles du jeu et sur le nombre de bombes et observerons l'effet de cette suppression sur les performances de notre meilleure intelligence artificielle. Effectivement, cette dernière a été conçue pour des grilles de taille 10×10 contenant 10 bombes et également évaluée sur ce type de grilles (pour rappel, un taux de victoires de 86,6% a pu être atteint). Nous allons donc ici l'évaluer sur d'autres types de grilles; plus précisément sur:

- des grilles de taille différente, mais avec une densité de bombes très proche (par exemple, des grilles de 20×20 contenant 40 bombes);
- des grilles de même taille, mais dont la densité de bombes varie (par exemple des grilles de 10×10 contenant 14 bombes).

Pour connaître la performance de l'intelligence artificielle pour un certain type de grilles, nous utiliserons le taux de victoires et le rapport entre le score moyen obtenu par celle-ci et le score maximum atteignable. Comme le score est défini comme le nombre de cases vides démasquées, ce rapport correspond en réalité au pourcentage du nombre moyen de cases vides que l'intelligence artificielle a pu démasquer.

Regardons tout d'abord l'évolution des performances de l'intelligence artificielle en fonction de la taille de la grille du jeu (avec une densité de bombes proche de 10%). Cela est montré dans le [Tableau 3](#). Nous constatons que plus la taille de la grille est élevée et plus le taux de victoires diminue de manière modérée. Jusqu'à une taille de 20×20 , ce taux est décent (supérieur à 60%). Cette diminution est probablement due au fait que l'intelligence artificielle doit identifier beaucoup plus de cases bombes sur des grilles plus grandes (pour une grille de 22×22 avec une densité de bombes de 9,9%, le nombre de cases bombes est égal à 48). Également, le rapport entre le score moyen et le score maximum baisse en fonction de la taille de la grille. Cependant, cette baisse est légère et ce rapport reste élevé pour toutes les tailles de grille essayées (supérieur à 80%).

Observons maintenant l'évolution des performances de l'intelligence artificielle en fonction de la densité de bombes pour des grilles de taille 10×10 ¹. Le [Tableau 4](#) montre cette évolution. Nous remarquons que le taux de victoires et le rapport entre le score moyen et le score maximum diminuent drastiquement. Cela peut s'expliquer par deux choses. Premièrement, plus la densité de bombes est élevée et plus la difficulté du jeu augmente de manière importante. Deuxièmement, le réseau de neurones a été entraîné spécifiquement pour des grilles de 10×10 contenant 10 bombes (avec de ce fait une densité de bombes de 10%). Cela a pour effet que les marquages élevés seront rares, bien plus qu'avec une densité de bombes plus élevée. Par conséquent, cela signifie le réseau s'est entraîné sur peu, voire trop peu, d'exemples contenant des marquages élevés. Par exemple, dans notre cas, parmi les 2.000.000 de sous-grilles de 5×5 contenues dans nos deux jeux de données, seule une contient une case possédant un marquage de 8. Cela est bien trop peu pour utiliser l'intelligence artificielle avec ce réseau de neurones sur des grilles dont la densité de bombes est élevée.

1. Comme les grilles ont une taille de 10×10 , une densité de bombes de $d\%$ signifie que celles-ci contiendront exactement d bombes.

Taille de la grille (densité de bombes)	Taux de victoires	Rapport entre le score moyen et le score maximum
6 × 6 (11,1%)	83,5%	88,2%
8 × 8 (9,4%)	86%	90,9%
10 × 10 (10%)	86,6%	91,3%
12 × 12 (9,7%)	80%	89,7%
14 × 14 (10,2%)	73,7%	88,2%
16 × 16 (10,2%)	69,3%	86,7%
18 × 18 (9,9%)	67,3%	87,4%
20 × 20 (10%)	61,5%	86,2%
22 × 22 (9,9%)	54,9%	83,8%
24 × 24 (10,1%)	48,8%	81,7%

Tableau 3 – Performances de l'intelligence artificielle en fonction de la taille de la grille du jeu

Densité de bombes	Taux de victoires	Rapport entre le score moyen et le score maximum
4%	98%	98%
6%	95,2%	95,8%
8%	90%	92,8%
10%	86,6%	91,3%
12%	68,3%	81,8%
14%	51,9%	74%
16%	35%	66,7%
18%	18,8%	56,1%
20%	7,9%	43,9%
22%	3,4%	32,3%
24%	0,4%	28,1%

Tableau 4 – Performances de l'intelligence artificielle en fonction de la densité de bombes

6 Conclusion

Dans ce rapport, nous avons parlé de la résolution du problème du jeu du démineur par l'apprentissage automatique. Plus précisément, nous avons utilisé les réseaux de neurones artificiels comme technique. Nous avons développé une première intelligence artificielle que nous avons améliorée par la suite. Pour simplifier le problème, nous avons fait l'hypothèse que toutes les grilles du jeu rencontrées par l'intelligence artificielle sont des grilles de taille 10×10 contenant 10 bombes.

Parlons de cette première intelligence artificielle. Celle-ci repose sur un système de sous-grilles de 5×5 dont chacune est extraite à partir d'une case de la grille du jeu.

Par conséquent, le réseau de neurones que l'intelligence artificielle utilise a été conçu pour prédire si la case centrale d'une sous-grille est une case bombe ou une case vide. Le jeu de données réalisé est constitué de paires (sg_i, yt_i) , où sg_i représente une sous-grille sans masque (où aucune case n'est masquée) et yt_i vaut 1 si la case centrale de sg_i est une case bombe, 0 dans le cas contraire. Il est nécessaire que sg_i soit une sous-grille vraisemblable, c'est-à-dire qui représente une sous-grille possible, réaliste. De ce fait, les sous-grilles ont été conçues pour correspondre à une extraction réelle d'une sous-grille dans une grille du jeu de 10×10 contenant 10 bombes. Quant au jeu d'entraînement, il est très similaire au jeu de données; il est constitué de paires (msg_i, yt_i) , où msg_i est une sous-grille avec masque (où certaines cases sont masquées) et yt_i représente la même chose que dans le jeu de données. Une telle paire est générée à partir d'une paire du jeu de données. Dans notre cas, nous avons généré 50.000 sous-grilles sans masque où la case centrale est une case vide et 50.000 autres où la case centrale est une case bombe. À partir de chacune de celles-ci, 10 sous-grilles avec masque ont été générées. Le jeu d'entraînement est donc composé de 1.000.000 de paires au total. En ce qui concerne le réseau de neurones, celui-ci comprend trois couches cachées contenant 300, 256 et 128 neurones respectivement avec la fonction unité de rectification linéaire comme fonction d'activation. La couche de sortie ne contient qu'un seul neurone et utilise la fonction sigmoïde comme fonction d'activation. Quant aux paramètres d'entraînement du réseau, le nombre d'itérations a été fixé à 1, la taille du *batch* à 10, l'*optimizer* à RMSProp et la fonction de perte à l'erreur quadratique moyenne.

L'intelligence artificielle à proprement parler fonctionne différemment en fonction du premier tour et des tours suivants. Au premier tour, elle ne fait que démasquer une case au hasard. Pour chacun des prochains tours, elle procède en trois étapes. Tour d'abord, elle extrait les cases masquées de la grille du jeu courante et les sous-grilles correspondantes. Ensuite, elle envoie ces dernières au réseau de neurones, récupère les vecteurs de sortie prédicts et prend le vecteur dont la première composante est la plus petite. Disons qu'il s'agit du i^e vecteur. Alors, comme dernière étape, l'intelligence artificielle démasque la i^e case masquée.

Avec cette première intelligence artificielle, nous avons obtenu un taux de victoire de 18,2%. Concernant le réseau de neurones, l'erreur moyenne absolue commise par celui-ci était de 0,182 et l'erreur quadratique moyenne de 0,115.

Rappelons maintenant les améliorations faites sur cette intelligence artificielle.

La première amélioration consistait à modifier le ratio de sous-grilles où la case centrale est une case bombe. L'objectif était de voir si fournir plus de sous-grilles où la case centrale est une case bombe que de sous-grilles où la case centrale est une case vide (et

inversement) améliorait les performances. Malheureusement, les erreurs commises par le réseau de neurones étaient similaires à précédemment.

Le but de la seconde amélioration était de changer l'ordre des sous-grilles dans le jeu d'entraînement. Nous avons utilisé différents ordres comme l'alternante entre des sous-grilles où la case centrale est une case bombe et des sous-grilles où la case centrale est une case vide ou encore un ordre aléatoire. Cependant, aucun ordre testé n'a permis de diminuer les erreurs faites par le réseau de neurones.

La troisième amélioration permettait d'observer l'influence du nombre de sous-grilles contenues dans le jeu d'entraînement. Des performances plus élevées ont pu être obtenues en augmentant le nombre de sous-grilles, passant de 1.000.000 à 10.000.000. Concernant le réseau de neurones, l'erreur absolue moyenne est passée à 0,179 et l'erreur quadratique moyenne à 0,09. Le taux de victoires obtenu par l'intelligence artificielle a augmenté jusqu'à 19,3%.

La quatrième amélioration consistait à modifier les paramètres de l'entraînement du réseau de neurones. Nous avons pu déterminer qu'un nombre d'itérations de 6 et une taille du *batch* de 2.000 permettant d'améliorer significativement les performances. En effet, l'erreur moyenne absolue commise par le réseau est descendue à 0,119 et l'erreur quadratique moyenne à 0,062. Quant au taux de victoires, il est passé à 58,6%.

L'objectif de la cinquième amélioration était de modifier la structure du réseau de neurones par une structure plus complexe. Le nombre de neurones de toutes les couches cachées a été doublé et une couche supplémentaire, contenant 1.024 neurones, a été ajoutée entre la première et la deuxième couche cachée. Malheureusement, les performances obtenues étaient très semblables aux performances obtenues avant cette modification.

La sixième modification concerne le jeu de données. À ce moment-là, celui-ci contenait des doublons; plusieurs paires identiques étaient présentes. Nous avons donc créé un nouveau jeu de données ne contenant pas de doublons. Un modeste accroissement de performances a pu être constaté; le taux de victoires atteint était de 62,5%.

La septième amélioration consistait à créer une nouvelle intelligence artificielle faisant usage des drapeaux. Pour ce faire, il a été nécessaire d'entraîner un nouveau réseau de neurones adapté à celle-ci. Uniquement les sous-grilles avec masque présentes dans le jeu d'entraînement ont été modifiées. Cela a été fait juste en ajoutant des drapeaux sur quelques cases. En ce qui concerne le fonctionnement de la nouvelle intelligence artificielle, le premier tour est toujours réalisé en démasquant une case aléatoire. Pour chacun des tours suivants, l'intelligence artificielle commence par récupérer toutes les cases masquées et à en extraire les sous-grilles correspondantes. Celles-ci sont ensuite envoyées au réseau de neurones et les vecteurs de sortie prédicts sont récupérés. L'intelligence artificielle calcule v_{min} comme étant la valeur du vecteur dont la première composante est la plus petite et i_{min} comme étant l'index de ce vecteur. De même, elle calcule v_{max} comme étant la valeur du vecteur dont la première composante est la plus élevée et i_{max} comme étant l'index de ce vecteur. Si $v_{min} < (1 - v_{max})$, alors l'intelligence artificielle démasque la i_{min}^e case masquée ne contenant pas de drapeau. Sinon, elle pose un drapeau sur la i_{max}^e case masquée ne contenant pas de drapeau. Dans le cas où toutes les cases contiennent un drapeau, l'intelligence artificielle les retire tous et démasque la i_{min}^e case masquée. Avec cette nous intelligence artificielle, nous avons été en mesure d'atteindre un taux de victoire de 67%.

La dernière amélioration a été réalisée en ajoutant deux paramètres à l'intelligence ar-

tificielle utilisant des drapeaux. Le premier paramètre se nomme *playful level*. En fonction de la valeur de ce paramètre, l'intelligence artificielle aura plus tendance à démasquer des cases ou inversement davantage tendance à poser des drapeaux. Le second paramètre est appelé *flag threshold*. Il s'agit du seuil à partir duquel l'intelligence artificielle est autorisée à poser des drapeaux. Après avoir déterminé la meilleure valeur pour ces deux paramètres, nous avons pu obtenir un taux de victoires de 86,6%.

D'autres améliorations sont envisageables. En particulier, pour rendre l'intelligence artificielle plus performante sur d'autres types de grilles du jeu. Effectivement, celle-ci a été entraînée pour des grilles de taille 10×10 contenant 10 bombes. Cela a pour effet que l'intelligence artificielle éprouve des difficultés lorsque nous l'utilisons sur des grilles dont la densité de bombes est plus élevée par exemple. Comme amélioration possible, nous pouvons donc entraîner le réseau de neurones avec des sous-grilles contenant beaucoup de cases bombes. Également, nous pouvons l'entraîner avec des sous-grilles dont la quantité de cases bombes varie fortement. Dans ce cas, le jeu de données contiendrait par exemple des sous-grilles avec seulement une case bombe et des sous-grilles avec 20 cases bombes. Réaliser une de ces deux modifications devrait ainsi rendre l'intelligence artificielle plus efficiente sur des grilles dont la densité de bombes est élevée et ne devrait pas beaucoup affecter les performances pour des grilles de 10×10 contenant 10 bombes. Une autre amélioration potentielle est de générer de « meilleures » sous-grilles avec masque. Comme nous l'avons vu, les cases sont masquées aléatoirement. Cela a pour conséquence que les sous-grilles avec masque ne ressemblent pas forcément à celles que l'intelligence artificielle rencontrera en pratique. Cela pourrait donc affecter négativement les performances.

En conclusion, à partir de l'intelligence artificielle de départ ainsi que toutes les modifications qui ont été apportées, nous sommes parvenus à un taux de victoires de 86,6%. En d'autres mots, notre intelligence artificielle gagne plus de quatre parties sur cinq en moyenne. À titre de comparaison, les auteurs de la référence [9] ont obtenu un taux de victoire légèrement plus élevé que 75% et l'auteur de la référence [10] un taux de victoire de 92,5%.

Références

- [1] Documentation de keras. <https://keras.io>.
- [2] Méthodes d'évaluation des performances de keras (*metrics*). <https://keras.io/metrics/>.
- [3] Ethem Alpaydin. Introduction to Machine Learning. The MIT Press, 2nd edition, 2010.
- [4] Kriesel David. A Brief Introduction to Neural Networks. 2007. URL: http://www.dkriesel.com/en/science/neural_networks.
- [5] Kevin P. Murphy. Machine Learning: A Probabilistic Perspective. The MIT Press, 2012.
- [6] Michael A. Nielsen. Neural networks and deep learning. <http://neuralnetworksanddeeplearning.com>.
- [7] Mina Niknafs. Neural network optimization. <http://courses.mai.liu.se/FU/MAI0083/>, 1996.
- [8] Nils J. Nilsson. Introduction to machine learning: An early draft of a proposed textbook. <http://robotics.stanford.edu/people/nilsson/MLBOOK.pdf>, 1998.
- [9] Yurchenko Oleg and Tortay Alisher. Minesweeper solver using nn. 2016. URL: <https://github.com/Chizhik/minesweeper/blob/master/MLFinalReport.pdf>.
- [10] Kasper Pedersen. The complexity of minesweeper and strategies for game playing. <http://www.minesweeper.info/articles/ComplexityMinesweeperStrategiesForGamePlaying.pdf>, 01 2004.
- [11] Sebastian Raschka and Vahid Mirjalili. Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-learn, and TensorFlow, 2Nd Edition. Packt Publishing, 2nd edition, 2017.
- [12] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.