Compte-rendu PJT IA
Classification de profils de trajectoires

Titouan Audry et Thomas Sounack

Sommaire

Mise en contexte

Les réseaux de neurones ont deux utilités majeures, la création de programme permettant d'associer à des entrées complexes des résultats clairs (par exemple la présence ou non d'une cellule cancéreuse sur une photo) mais aussi la mise en place de programme de complexité plus faible accomplissant la même tâche que des programmes traditionnels.

Nous nous intéressons ici au second cas : nous cherchons à diminuer le temps de calcul d'un programme afin de pouvoir contrôler en temps réel la trajectoire d'un robot et limiter le Jerk (dérivée de l'accélération) afin de réduire le comportement vibratoire du robot.

Malgré le fait qu'il n'existe dans notre cas que 16 profils de trajectoire possibles, la détermination de la classe par des méthodes conventionnelles implique l'utilisation de nombreux tests conditionnels ; qui présentent une importante variance en temps de calcul et un temps calcul moyen relativement élevé.

L'objectif de ce projet est donc de mettre en place une IA afin de diminuer le temps de calcul de la classe de trajectoire et pouvoir ainsi changer de trajectoire en temps réel (en cas de présence d'obstacle, etc…).

1. Mise en œuvre du réseau de neurones

a. Mise en forme des données

Dans un premier temps, nous utilisons un set de données dit "well-balanced", c'est-à-dire que les différentes classes possibles y sont réparties de façon homogène (nous reviendrons plus tard sur l'influence de ce choix). Cette base de données est scindée en deux parties, l'une servant à entraîner l'intelligence artificielle et l'autre permettant de la tester.

Il convient alors de mettre en forme ces données pour qu'elles puissent être renseignées dans l'algorithme. Soulignons que cette partie peut s'avérer être assez laborieuse dans certaines applications où les données à disposition seraient complexes, mais elle est nécessaire au bon fonctionnement de l'IA.

b. Génération du modèle

Après avoir séparé les données en listes d'entrées et listes sorties, nous pouvons générer les couches de neurones. L'API utilisé est Keras, qui est utilisé dans un large panel d'applications.

Choix du modèle

Le modèle choisi est un modèle séquentiel, qui permet de visualiser très clairement la structure du réseau de neurone et de faire le lien entre la théorie du réseau et sa mise en pratique. On notera toutefois que dans la majorité des cas, les applications professionnelles des intelligences artificielles sur Keras se font avec le modèle functional API.

```
model = Sequential()
model.add(Dense(6, input_dim=6, activation='elu', name='Entree'))
model.add(Dense(32, activation='elu', name='Cachee1'))
model.add(Dense(24, activation='elu', name='Cachee2'))
model.add(Dense(16, activation='softmax', name='Sortie'))
```

Structure du réseau de neurones

Paramétrage des couches de neurones

Le nombre de neurones en entrée et en sortie est fixé par rapport aux données du problème. En effet, le nombre de neurones en entrée (resp. en sortie) doit être égal à la dimension du vecteur d'entrée (resp. de sortie). Concrètement, on met autant de neurones en entrée que de variables d'entrée, et autant de sorties que de nombre de classes qu'on souhaite obtenir.

Entre ces deux couches on crée deux couches cachées, de respectivement 32 et 24 neurones. Nous reviendrons sur l'influence du nombre de neurones dans ces couches en dernière partie.

Pour le choix des fonctions d'activation, on assigne à la couche de sortie la fonction 'softmax'. Il n'y a pas d'autre possibilité ici car c'est la fonction de sortie de référence pour les problèmes probabilistes de détermination de classes. Concernant les autres couches, le sujet impose des fonctions 'elu'.

c. Entraînement du modèle

L'entraînement du modèle se fait avec la fonction model.compile de Keras.

Loss functions

Un premier paramètre à déterminer pour utiliser cette fonction est la fonction de pertes, qui correspond à une fonction d'erreur : l'algorithme d'optimisation de l'IA repose en effet sur une estimation à chaque étape de l'erreur de l'algorithme. Les différentes loss functions proposent chacune une méthode pour déterminer cette erreur.

Le choix de la fonction de loss est limité, car nous souhaitons obtenir en sortie du modèle plus de deux classes. La fonction cross-entropy est souvent choisie, d'autant plus que le nombre de classes que l'on souhaite obtenir est limité (16). Nous utiliserons donc cette fonction de loss pour l'entraînement de l'IA.

Optimizer

Il faut également déterminer l'optimizer de l'entraînement. Ce paramètre correspond à l'algorithme utilisé pour ajuster les poids des nœuds en fonction des différences prédiction - résultat réel. L'optimizer Adam permet d'offrir de bonnes performances pour la plupart des modèles, et nous l'utiliserons ici comme suggéré dans l'énoncé.

Metrics

Nous cherchons ici à optimiser l'accuracy de notre modèle. Le paramètre metrics permet de définir l'accuracy comme échelle d'évaluation des performances de l'IA.

Batch size

Il faut également choisir le batch size. Suite aux premières mises en place de réseaux de neurones, nous avons observé que le temps de calcul était inversement proportionnel au batch size. Ce paramètre correspond à la taille de découpe des sous-listes dans la base de données. L'optimizer ajuste les poids des nœuds entre chaque traitement de sous-liste.

Epochs

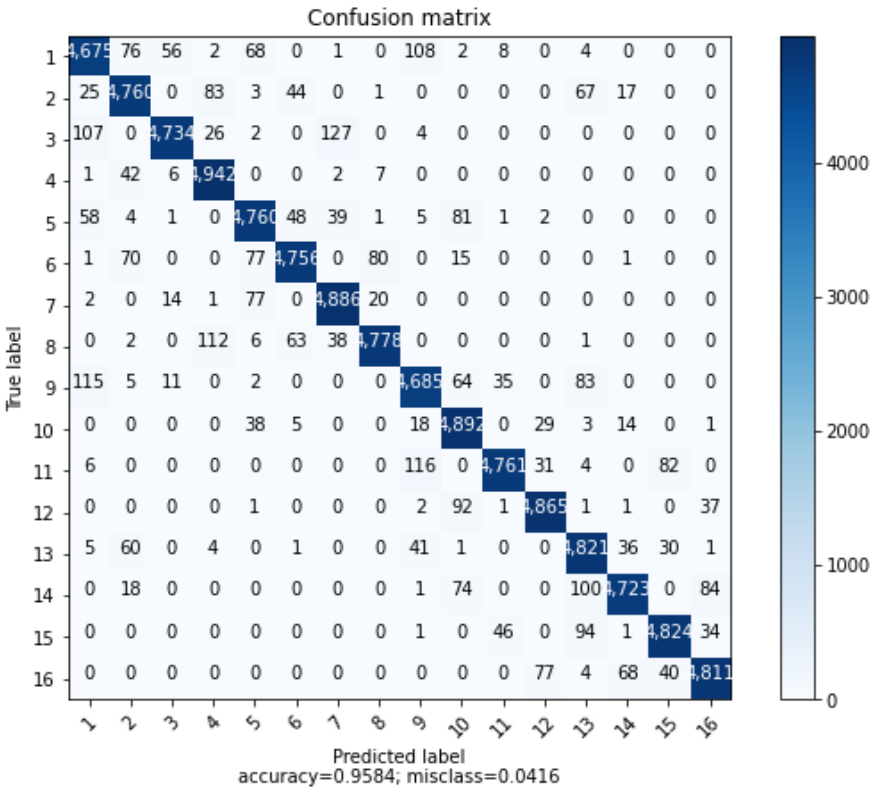
L'epochs (ou époques) correspond au nombre d'entraînements du réseau de neurones avec la liste de train. Cela permet à la précision du modèle de converger vers sa valeur finale, car un seul traitement de la liste de train ne suffit pas à l'IA pour atteindre sa valeur « palier » :

```
Epoch 1/400
235/235 [==============================] - 1s 3ms/step - loss: 1.9327 - accuracy: 0.3799
Epoch 2/400
235/235 [==============================] - 1s 3ms/step - loss: 1.0331 - accuracy: 0.6389
Epoch 3/400
235/235 [==============================] - 1s 3ms/step - loss: 0.8191 - accuracy: 0.7076
```

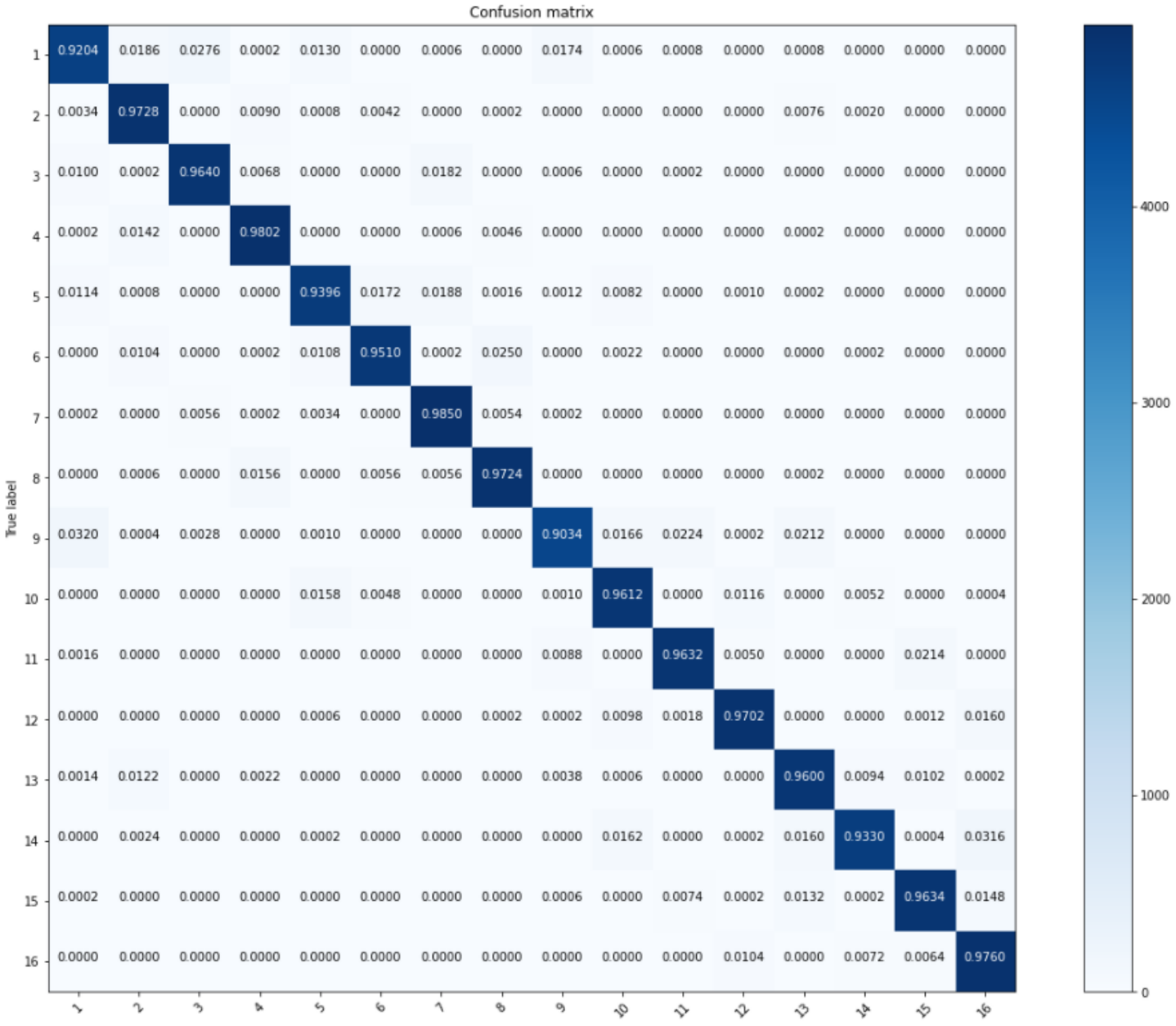
d. Test de l'IA

On utilise ensuite les données de test pour valider les performances de l'IA. Pour cela, on compare les 'prédictions' de l'IA sur la liste d'entrée avec la liste de sortie.

L'affichage des résultats se fait sur une matrice de confusion, qui présente l'intérêt de donner les résultats sous forme très compacte (on travaille sur des listes de test de plusieurs dizaines de milliers d'éléments). Le principe est d'afficher sur les colonnes les prédictions de l'IA, et sur les lignes les classes réelles. Les coefficients diagonaux (i,i) sont donc incrémentés de 1 pour chaque prédiction correcte (de classe i).



Pour pouvoir interpréter plus facilement ces résultats, on peut aussi donner la matrice de confusion sous sa forme normalisée (les coefficients de la matrice sont alors affichés en pourcentages).



On constate que pour une IA entraînée et testée avec un Well Balanced dataset, on obtient une accuracy de 95,84%.

En remettant en contexte ce résultat, cela signifie également qu'un robot équipé avec cette solution donnerait des prédictions fausses dans 4,16% des cas. Les applications du robot ne sont pas précisées dans le sujet (et il est donc difficile de déterminer si le résultat est satisfaisant), mais on comprend bien qu'une erreur dans la détermination du profil de vitesse dans le domaine chirurgical pourrait être très grave.

Ce constat met en évidence une particularité du domaine de l'IA : le panel d'application d'une intelligence artificielle étant extrêmement large, l'analyse des résultats se fait de manière très différente selon chaque cas.

2. Etude : Well-Balanced vs Representative dataset

Notons que l'utilisation d'un well-balanced dataset implique une étape de préparation supplémentaire des données au préalable. En effet, il faut s'assurer que toutes les classes sont représentées de façon homogène en construisant la liste des données.

Il est donc intéressant de comparer les résultats obtenus en utilisant un set de données représentatif, dit representative dataset comme set d'entraînement et / ou de test.

On considérera pour notre étude quatre cas, dont les résultats sont récapitulés dans le tableau ci-dessous.

Accuracy		Entraînement	
		Well-Balanced	Representative
Test	Well-Balanced	95.72%	90.30%
	Representative	94.97%	97.52%

Dans les trois cas non-traités jusqu'à présent, les matrices de confusion normalisées sont données en annexe.

Les résultats obtenus sont cohérents. En effet, dans le cas du well-balanced dataset, les différentes classes ont été représentées de manière uniforme lors de l'entraînement de l'IA. Il est donc normal que les résultats du test soient du même ordre de grandeur pour le well-balanced dataset et pour le representative dataset, puisque l'IA présente globalement les mêmes capacités de prédiction pour chaque classe.

De même, en entraînant le modèle avec un representative dataset, certaines classes sont plus représentées que d'autres (notamment les classes 1 et 9). L'IA est donc mieux entraînée sur ces classes : les capacités de prédiction de l'IA ne sont plus homogènes selon les classes. En testant ce modèle sur un dataset représentatif l'IA est amenée à tester plus souvent les classes sur lesquelles elle est plus performante, et l'on obtient donc la meilleure accuracy des quatre configurations.

A l'inverse, en testant un well-balanced dataset cette IA teste aussi souvent les classes pour lesquelles elle est performante que les classes pour lesquelles elle est moins performante. L'accuracy s'en trouve donc réduite.

Une fois de plus, le type de dataset à utiliser pour l'entraînement et le test de l'IA dépend fortement du contexte. Entraîner une IA avec un well-balanced dataset permet d'assurer des résultats identiques d'une classe à l'autre ; tandis qu'un entraînement avec un representative dataset donnera une meilleure accuracy pour des classes observées plus fréquemment, mais une accuracy plus mauvaise pour des classes moins observées.

3. Etude : optimisation du nombre de neurones sur les couches cachées

a. Mise en place du GridSearch

Pour faire cette optimisation, nous utilisons la fonction GridSearchCV de la bibliothèque Scikit-learn.

Contrairement à ce que nous avons fait jusqu'ici, nous devons définir le modèle sous la forme d'une fonction, prenant en argument les paramètres que l'on cherche à optimiser : ici neurons1, le nombre de neurones sur la première couche cachée, et neurons2, le nombre de neurones sur la seconde couche cachée.

Nous devons ensuite définir le dictionnaire, qui impose à l'algorithme d'optimisation les différentes valeurs à tester. Il est conseillé d'utiliser des nombre de neurones multiples du vecteur d'entrée (6), et d'avoir une structure dite 'en entonnoir', où le nombre de neurones de la première couche cachée à la sortie est décroissant (nous reviendrons sur ce point lors de l'analyse des résultats). La couche de sortie ayant 16 neurones, la première valeur à tester sera donc 18 neurones.

Notons également que le temps de calcul du gridsearch n'augmente pas de manière linéaire selon le nombre de combinaisons à tester (utilisation de cross-verification, epochs élevé). De plus le temps de prédiction de l'IA augmente avec le nombre de neurones : nous nous limiterons donc à 54 neurones par couche cachée, puisque le modèle doit avoir un temps de prédiction court au vu de son application.

Une des difficultés à cette étape était de parvenir à maintenir la session google colab ouverte pendant toute la durée du calcul (environ 3 à 4h selon les hyperparamètres). L'utilisation du paramètre verbose permet d'avoir un retour régulier sur la console, et donc de maintenir ouverte notre session.

b. Analyse des résultats

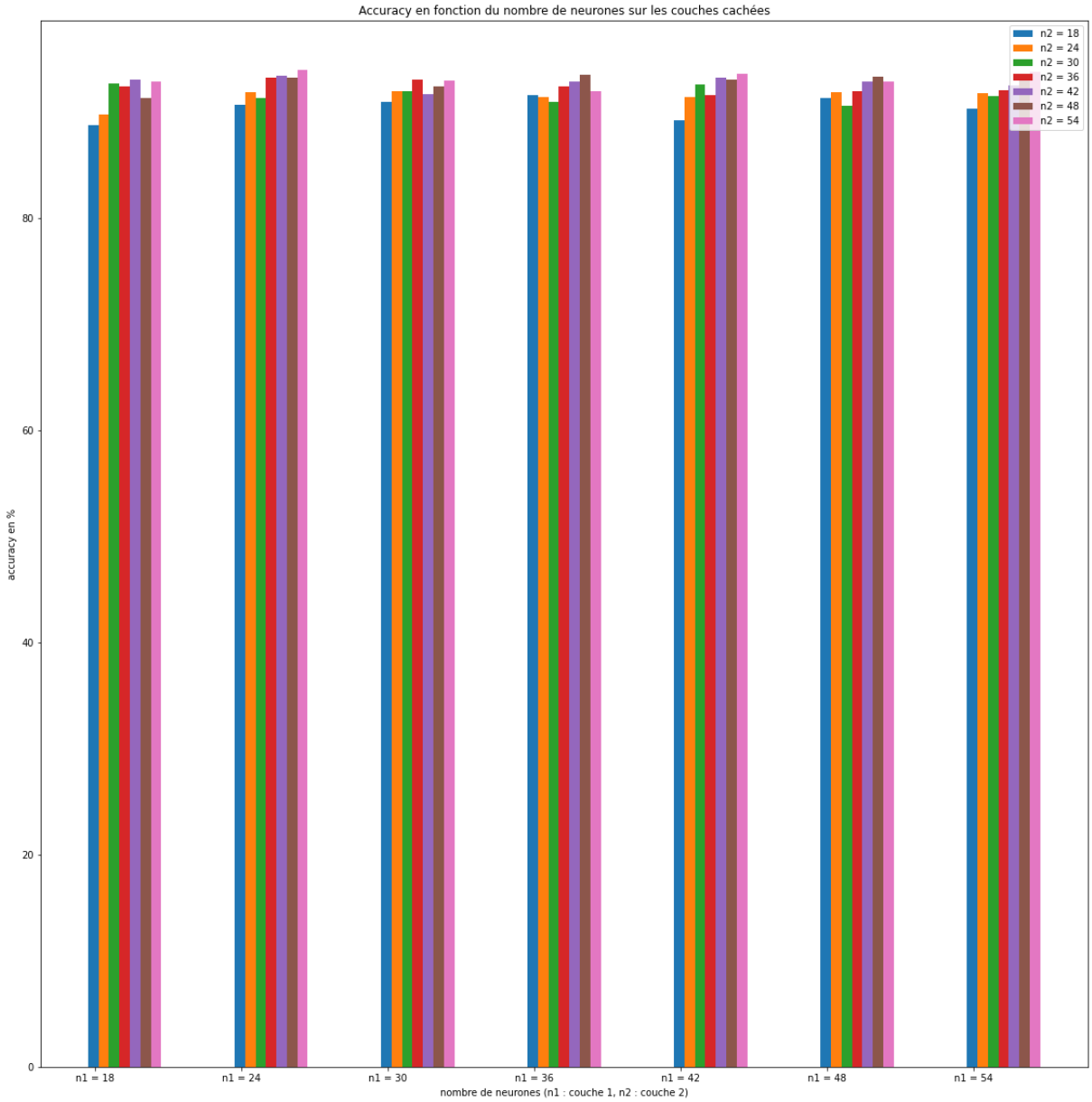
Après calcul (avec un well-balanced dataset), la meilleure solution retenue par l'algorithme est la suivante :

```
0.935954 (0.011500) with: {'neurons1': 42, 'neurons2': 54}
```

Il est intéressant de noter que l'accuracy obtenue est plus faible que pour les tests précédents entraînés avec un well-balanced dataset. En ré-entraînant un modèle avec ces paramètres, nous obtenons toutefois une accuracy de 97%. Les différents paramètres (dont l'algorithme d'optimisation) et le seed étant fixés et le nombre d'époques étant élevé, il est difficile de voir d'où peut venir cette différence. Une possibilité serait que l'on utilise uniquement la liste WB_Train, il est donc vraisemblable que le modèle s'entraîne et se teste sur une seule liste au lieu de deux. Il y aurait donc moins de données d'entrée.

Nous faisons donc l'hypothèse que l'utilisation d'un gridsearch ne permet pas d'obtenir les mêmes résultats qu'un modèle à part, mais que le classement des résultats reste valide.

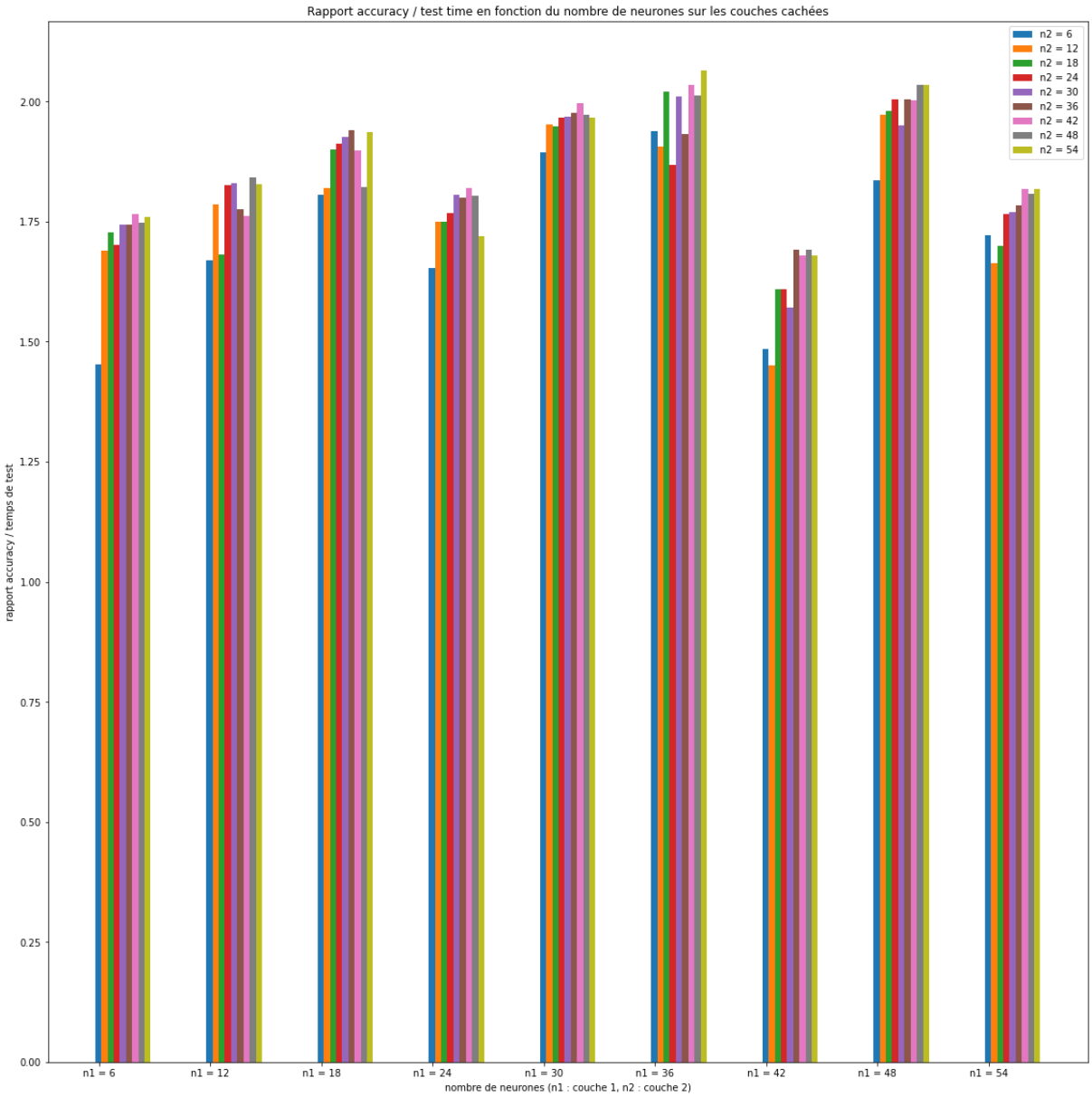
On peut alors afficher les résultats sous la forme d'un graphe :



On observe que l'accuracy converge rapidement, et (hormis pour les premières combinaisons) augmente marginalement avec le nombre de neurones par couche.

Il est également intéressant d'afficher les résultats sur le rapport accuracy / temps de prédiction : le meilleur résultat correspond à la solution offrant le meilleur compromis entre ces deux paramètres. Pour cet essai nous avons également testé les combinaisons contenant 6 et 12 neurones, afin de mettre en évidence l'impact de la simplicité d'un modèle sur le temps de prédiction. On constate que les résultats ne suivent pas la règle 'de l'entonnoir'. On peut supposer qu'il s'agit d'une règle générale qui ne s'applique pas dans notre cas.

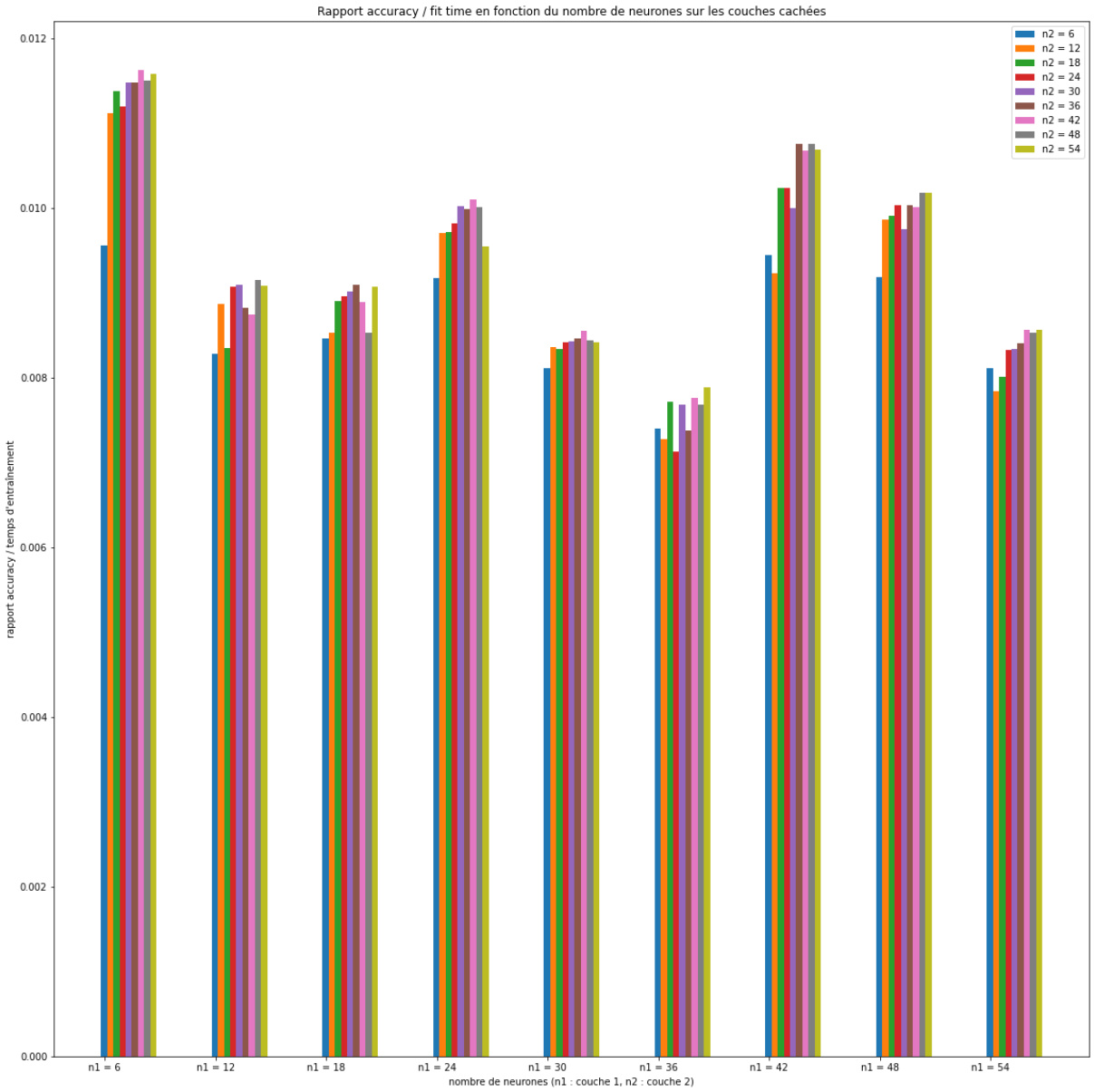
Notons que nous avons utilisé pour cela le résultat 'mean_score_time', qui représente le temps total de calcul moins le temps d'entraînement : ce n'est donc pas exactement le temps de prédiction. Nous ferons ici l'approximation qu'il s'agit du temps de prédiction en utilisant ce résultat comme borne haute du temps réel.



On observe cette fois des différences nettement plus marquées dans les résultats.

Contrairement à ce qu'on pouvait imaginer, le meilleur rapport n'est pas obtenu pour les combinaisons présentant le moins de neurones. Le meilleur résultat est obtenu par le modèle (n1 = 36, n2 = 54). De plus, il est intéressant de remarquer que certaines configurations présentent des rapports relativement faibles, et sont donc peu adaptées aux paramètres que l'on cherche à optimiser (par exemple toutes les combinaisons pour n1 = 42).

Il est également possible d'afficher le rapport accuracy / temps d'entraînement. Plus ce rapport est élevé, plus le modèle offre un bon compromis entre ces deux paramètres.

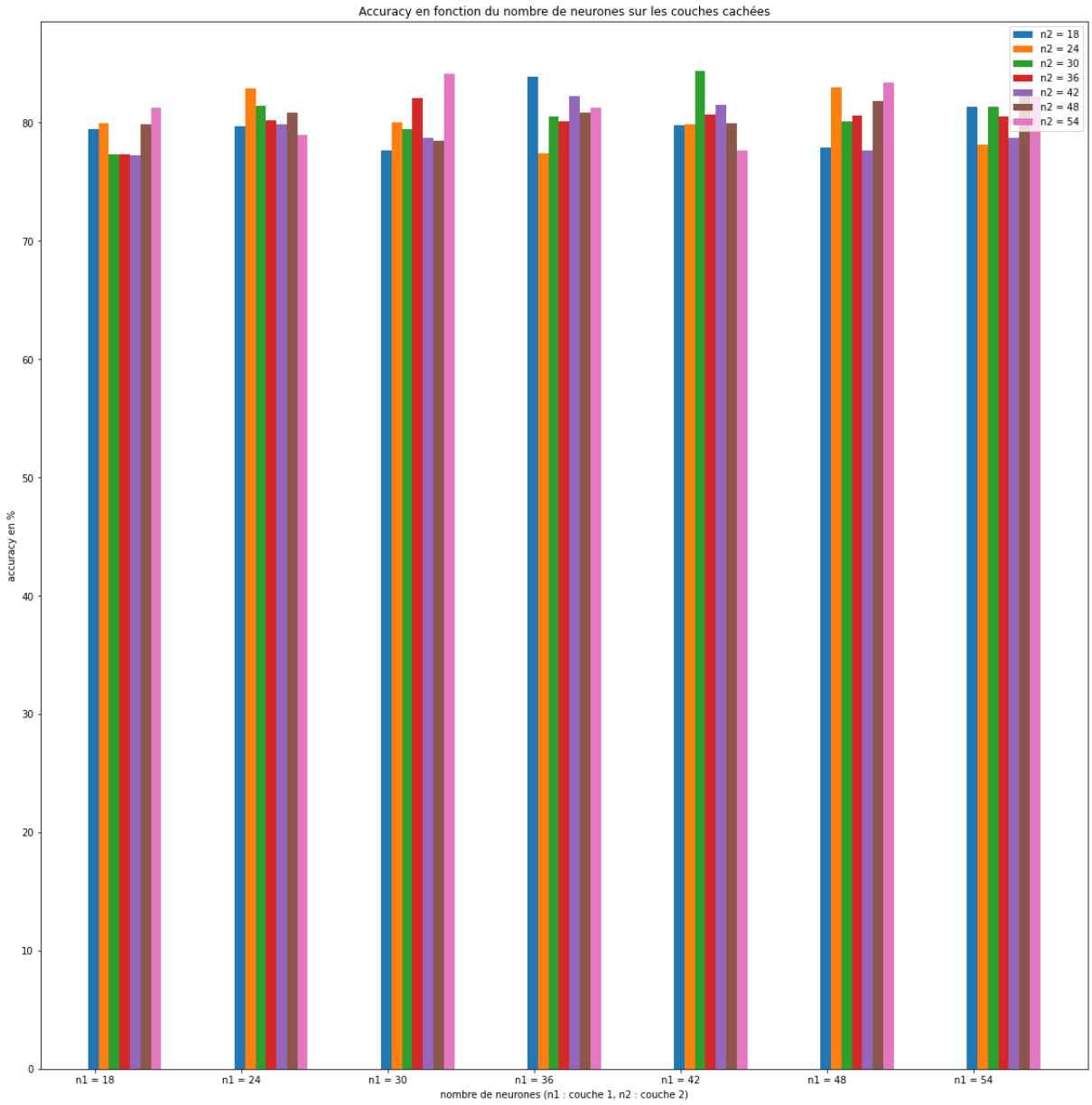


Dans notre application, on ne cherche pas nécessairement à optimiser ce paramètre puisqu'il s'agit d'un calcul réalisé une seule fois en amont de l'utilisation du modèle. Il est toutefois intéressant de noter que ce paramètre peut avoir son importance dans des cas d'utilisations où les données d'entrées de l'algorithme sont très nombreuses et où le temps / la puissance de calcul à disposition sont limités.

c. Optimisation pour un representative dataset

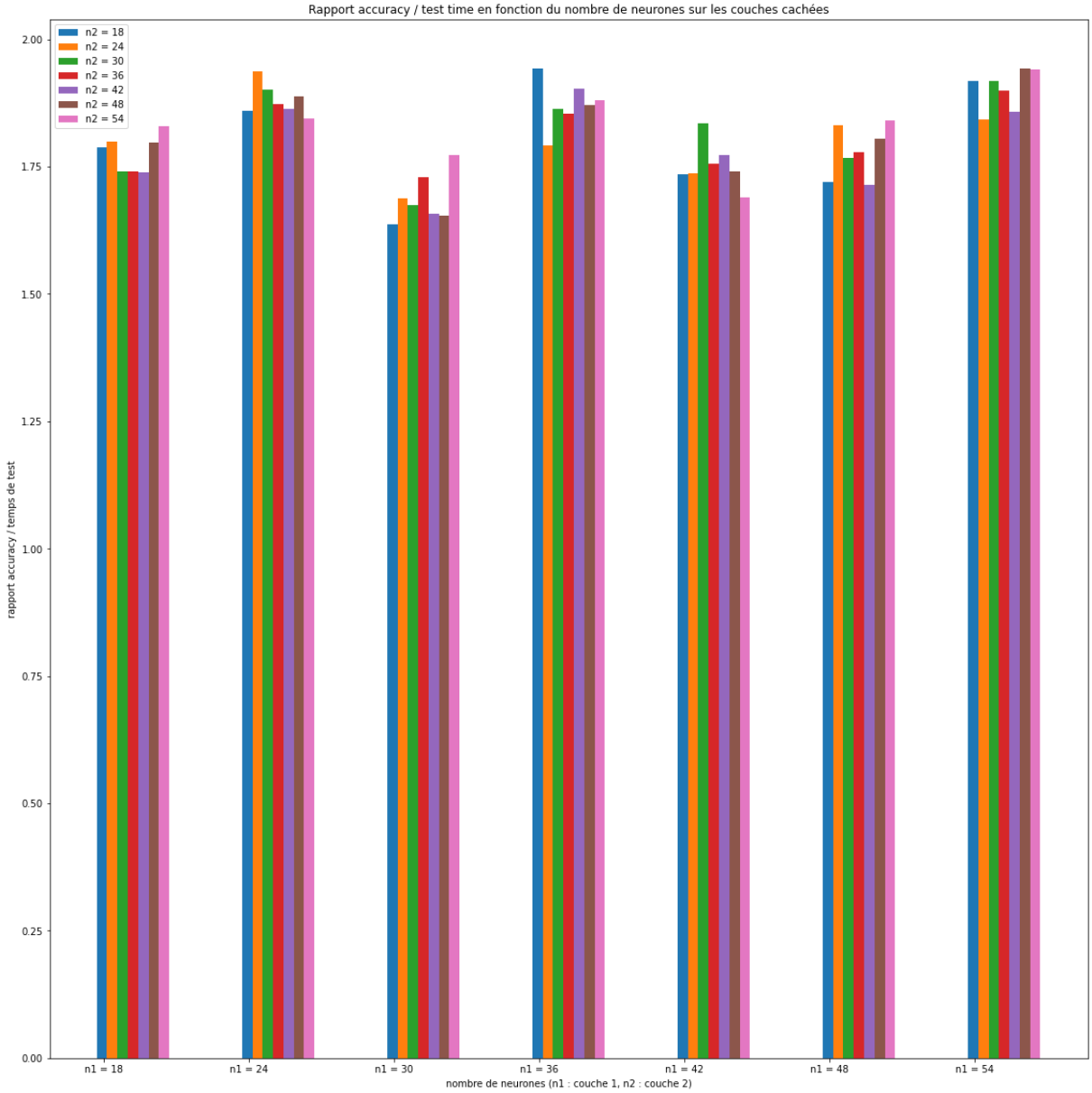
En suivant la logique de notre étude, on peut s'intéresser à la configuration obtenue par le gridsearch en utilisant un representative dataset.

```
0.843622 (0.058973) with: {'neurons1': 54, 'neurons2': 30}
```



On observe comme pour le well-balanced dataset que l'accuracy est sous-estimée.

En affichant le rapport accuracy / temps de prédiction, on observe également que les résultats ne sont pas les mêmes que pour le well-balanced dataset.



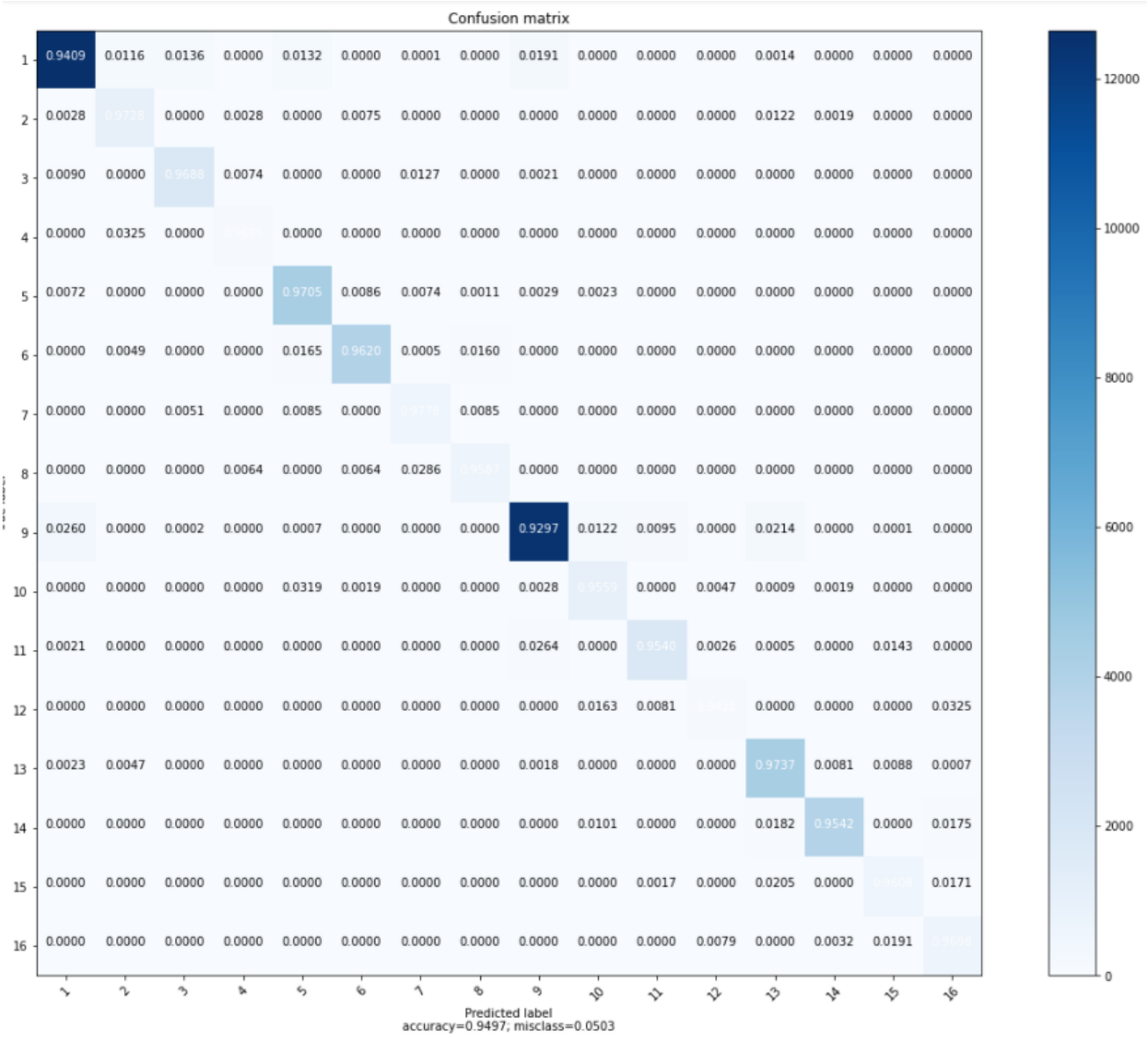
La configuration optimale proposée par l'algorithme étant différente de celle trouvée précédemment, il faudra choisir la configuration en fonction du type de données rencontrées et prendre le parti d'optimiser l'IA pour certaines classes ou non.

Conclusion

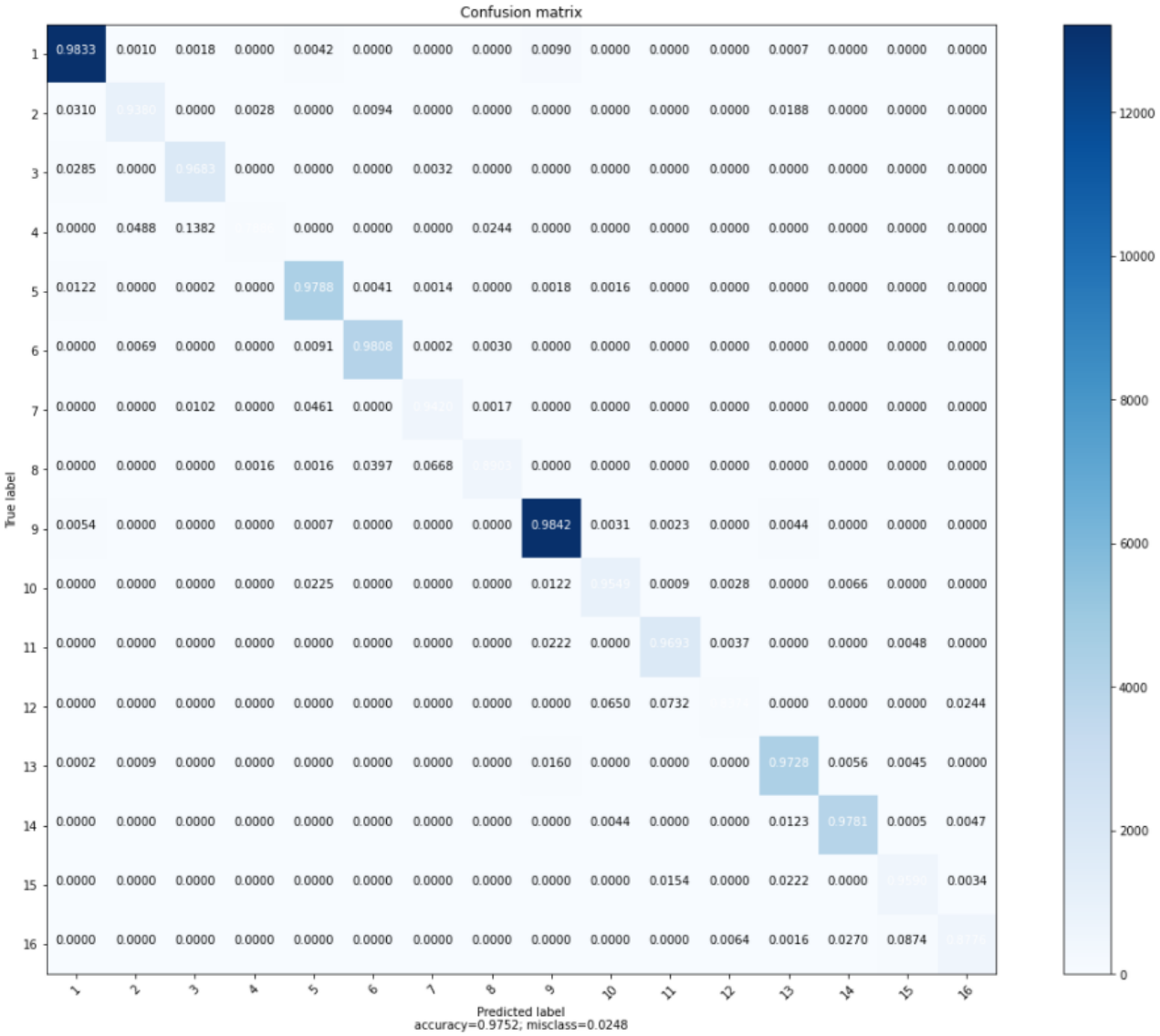
N'ayant pas de cahier des charges défini, il est difficile de retenir une configuration définitive pour notre modèle. En effet, ce choix dépendrait de l'application de notre IA. Après avoir déterminé si l'on souhaite entraîner le modèle avec un well-balanced ou un representative dataset, le choix du nombre de neurones sur les couches cachées se fera de manière à respecter l'accuracy du cahier des charges tout en offrant le temps de prédiction le plus faible possible.

Ce projet nous aura permis de nous familiariser avec le concept d'intelligence artificielle, tout en nous sensibilisant aux différents choix à faire pour mettre en place un modèle. C'est en effet l'un des points les plus importants dans ce domaine : l'IA étant un outil très polyvalent, un ingénieur doit garder à l'esprit l'application de son modèle afin de faire les choix les plus pertinents pour l'optimiser.

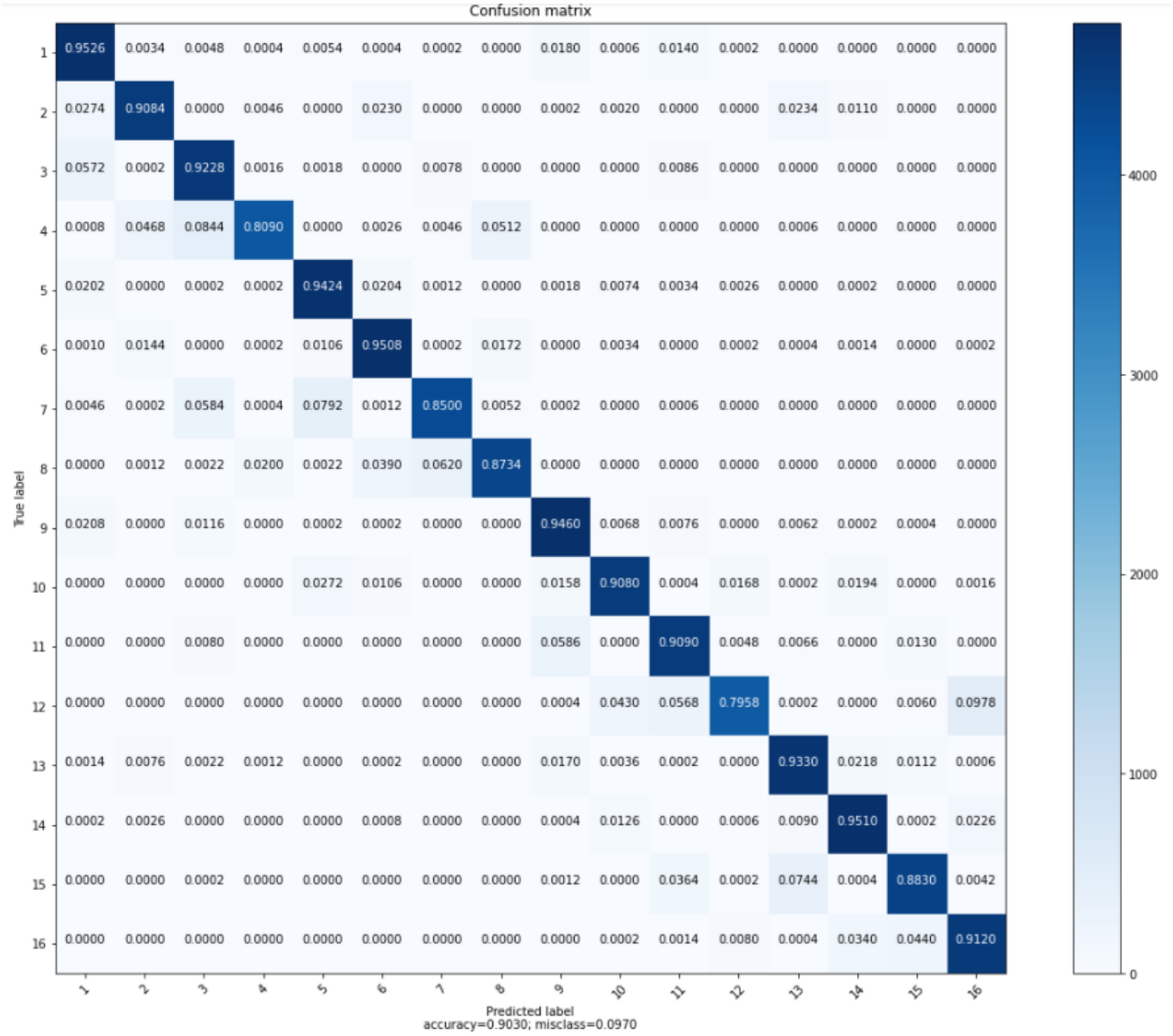
Annexe



Annexe 1 : Matrice de confusion – Well-balanced train ; representative test



Annexe 2 : Matrice de confusion – Representative train ; representative test



Annexe 3 : Matrice de confusion – Representative train ; well-balanced test