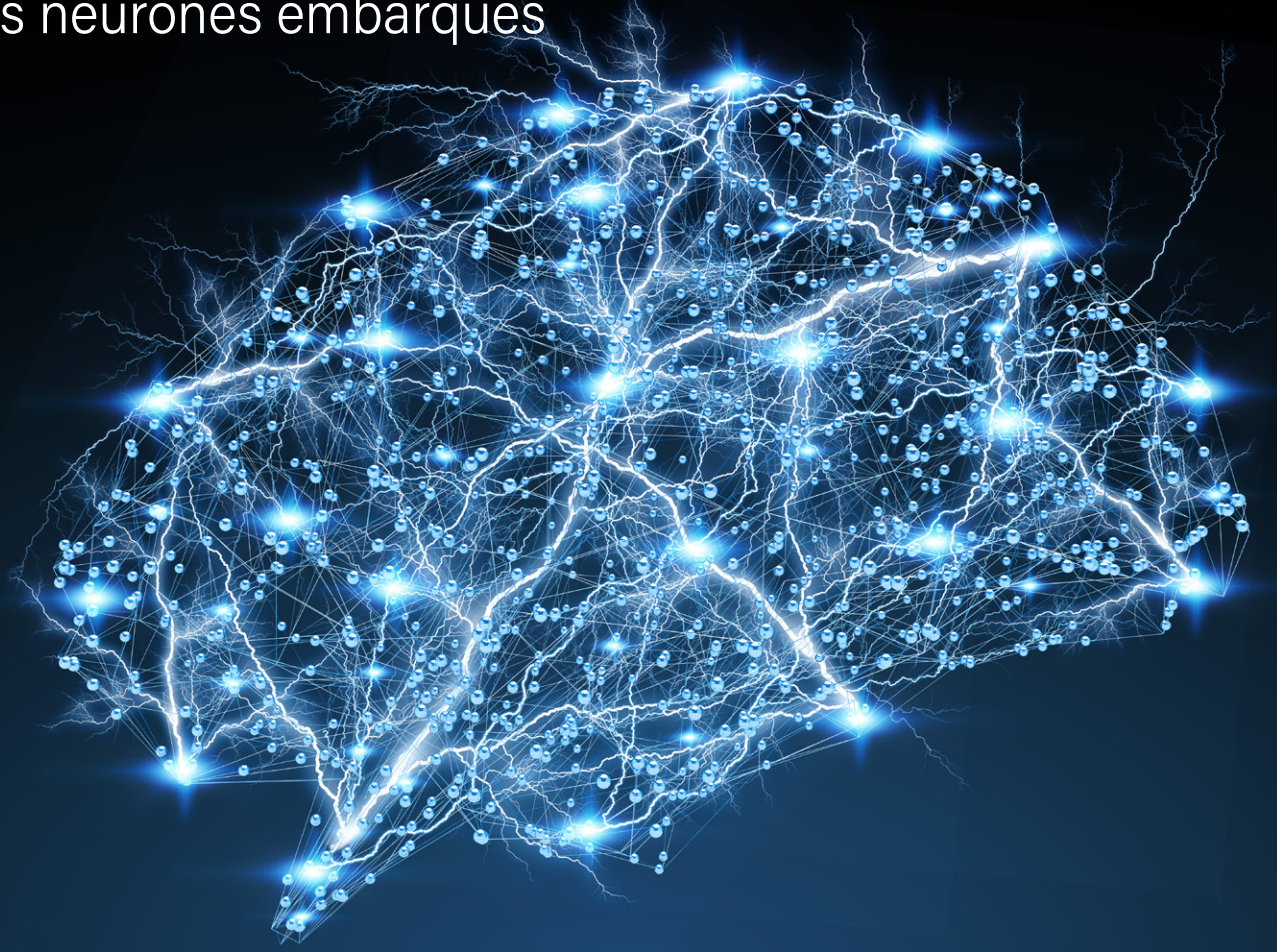
voyage dans les réseaux neuronaux (4e partie)

Les neurones embarqués



Stuart Cording (Elektor)

Notre réseau neuronal fonctionne parfaitement sur les PC et les ordinateurs portables et nous pouvons mettre en œuvre en toute confiance notre perceptron multicouche. Nombre d'applications ont besoin des capacités de faible consommation et de latence minimale qu'offre un microcontrôleur. Certaines ne souhaitent tout simplement pas partager leurs données privées avec des services d'IA basés sur des systèmes tiers dans le nuage. Ici, nous rendrons notre réseau neuronal compatible avec Arduino et transférerons notre code de classification des feux de signalisation dans l'univers des systèmes embarqués.

Les outils et les plateformes d'apprentissage automatique (ML) semblent aujourd'hui pousser comme des champignons. Quelle que soit la complexité de votre tâche ou encore le volume de vos données, il existe un service dans le nuage (*cloud*) pour les traiter. Cependant, dans de nombreux cas, la mise en œuvre de l'apprentissage automatique dans le cloud est inadaptée. Si vous traitez des données sensibles ou personnelles, vous ne souhaitez peut-être pas les transférer via l'internet pour les faire analyser par un outil de ML. De même par ex. pour les applications automobiles ou d'autres applications embarquées en temps réel. Ces systèmes exigent une décision immédiate. Une solution de ML locale est donc nécessaire du fait de la latence d'une connexion internet ou de l'impossibilité éventuelle d'une connexion. Cette approche est également nommée « apprentissage automatique en périphérie de réseau » [1].

Avec un réseau neuronal fonctionnant en périphérie, même de simples microcontrôleurs peuvent exécuter des algorithmes de ML. Cependant, en raison des exigences relatives au processeur pour l'entraînement, le réseau est souvent entraîné dans le nuage ou à l'aide d'un PC puissant. Les poids résultants sont ensuite téléchargés sur le microcontrôleur pour un fonctionnement déconnecté d'internet et à faible latence.

Ce dernier article de cette série concerne le portage de notre réseau neuronal de perceptron multicouche sur un Arduino. Couplé à un capteur RVB, nous lui apprenons à classer les couleurs des feux de signalisation comme nous l'avons fait avec Processing sur PC ou ordinateur portable. En utilisant un Arduino à 32 bits (DUE ou Mo Pro), nous pouvons entreprendre la phase d'apprentissage sur le microcontrôleur. Cependant, nous allons constater qu'il est assez lent. Ainsi, nous scinderons également la tâche de manière à ce que le réseau soit entraîné en s'appuyant sur la puissance d'un PC, puis exécuté dans l'environnement à faible latence et basse consommation d'une application de microcontrôleur embarqué.

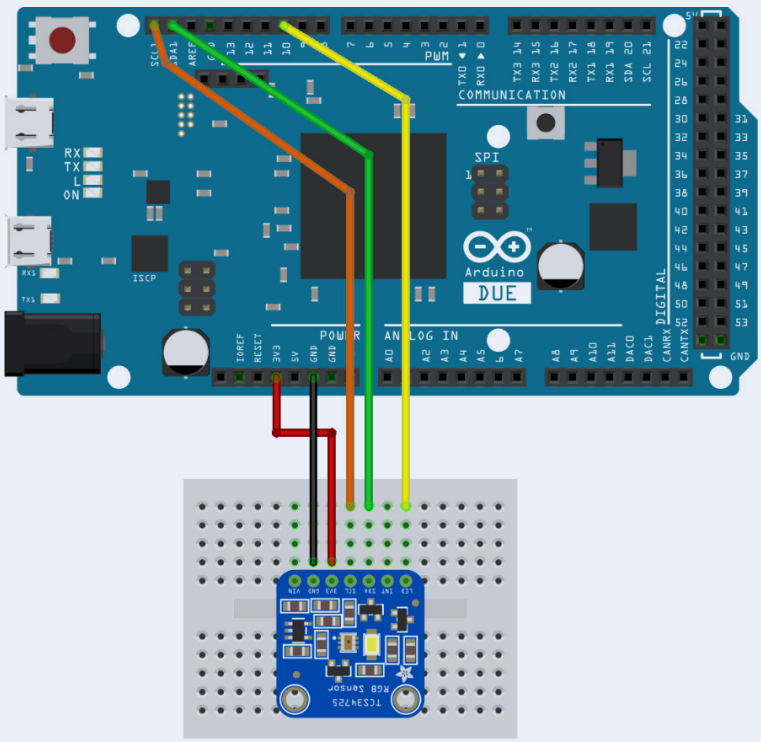


Figure 1. Schéma de câblage du capteur RVB TCS34725 avec un Arduino DUE.

Choix d'un capteur

Pour identifier les couleurs des feux de signalisation, l'Arduino aura besoin d'un capteur approprié en entrée. Le TCS34725 d'Adafruit est une solution de détection RVB appropriée, disponible et prête à l'emploi sous la forme d'une carte au format 2 × 2 cm [2]. Elle permet d'acquérir des données RVB, comme nous l'avons fait précédemment avec la caméra, et de les appliquer à la même configuration MLP utilisée pour les exemples donnés sur PC. La carte peut être utilisée avec les Arduino 5 V et 3,3 V, et les données du capteur sont

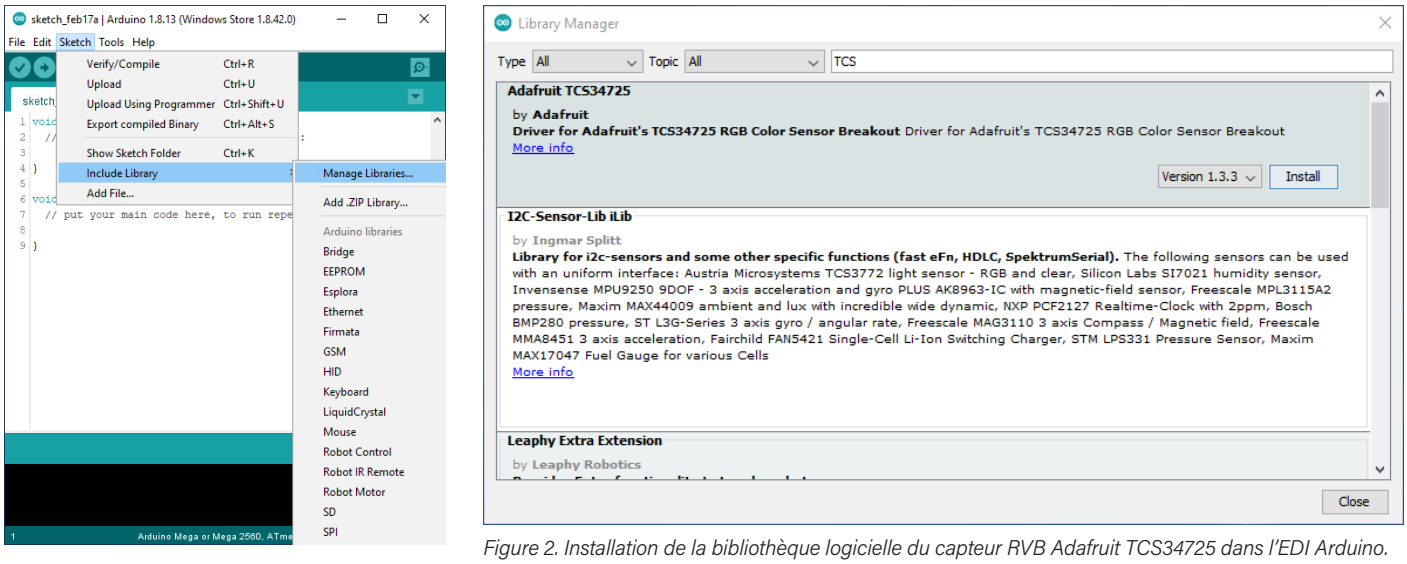


Figure 2. Installation de la bibliothèque logicielle du capteur RVB Adafruit TCS34725 dans l'EDI Arduino.

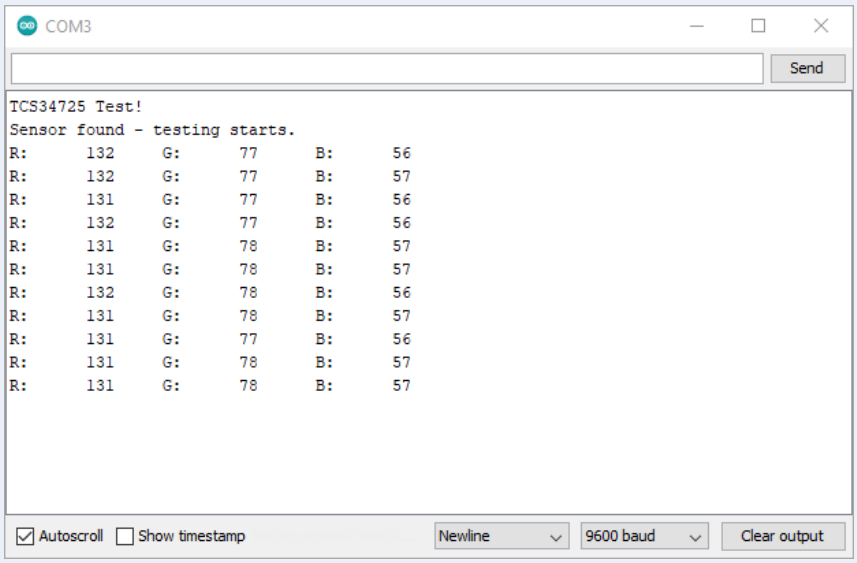


Figure 3. Sorties de valeurs RVB fournies par le croquis tcsrgbsensor.ino Arduino.

acquises à l'aide de l'interface I2C, ce qui nécessite la bibliothèque Wire.

Pour assurer un éclairage constant de l'échantillon à analyser, la carte comprend également une LED blanche commandée par une sortie numérique Arduino ; ici c'est la broche 10 (**fig. 1**). Fort heureusement, la carte est associée à une bibliothèque bien construite, ce qui permet une mise en place et un fonctionnement simples et directs. Comme précédemment, nous utiliserons le code du référentiel GitHub préparé pour cette série d'articles [3].

Avant d'exécuter du code, nous devons installer la bibliothèque pour le capteur RVB. Cela devrait être facile, car elle est accessible via le gestionnaire de bibliothèque dans l'EDI Arduino. Dans la barre de menu, il suffit de sélectionner *Croquis -> Inclure une bibliothèque -> Gérer les bibliothèques* puis d'entrer « tcs » dans le champ de recherche du Gestionnaire de bibliothèque. La bibliothèque « Adafruit TCS34725 » devrait apparaître. Cliquez simplement sur « Install » pour l'installer (**fig. 2**). En cas de difficultés, le code source et le fichier d'en-tête peuvent être téléchargés depuis le référentiel Adafruit GitHub [4] et ajoutés manuellement aux projets.

Pour s'assurer que le capteur fonctionne et disposer de la méthode d'acquisition des valeurs RVB dont nous avons besoin pour entraîner le dispositif de ML, nous allons commencer par le croquis *arduino/tcsrgbsensor/tcsrgbsensor.ino*. Après avoir initialisé le capteur RVB, le code allume la LED et commence à renvoyer les valeurs RVB via la sortie série (**fig. 3**). Ouvrez *Outils -> Moniteur série* pour les visualiser. Le réglage du débit est de 9600 bauds. Pour améliorer la qualité des relevés et réduire l'impact des autres sources de lumière, il est utile de fabriquer un écran de protection autour de la carte du capteur. Une bande de carton noir d'environ 3 cm de haut et 10 cm de long est idéale (**fig. 4**). L'écran permet également de maintenir l'image des feux de signalisation à une distance constante du capteur RVB.

Le capteur RVB produisant des résultats fiables, nous pouvons maintenant obtenir l'évaluation des couleurs rouge, orange et vert à l'aide de notre image de feux de circulation imprimée (à partir de *trafficlight/resources*). Les résultats obtenus par l'auteur sont présentés dans le **tableau 1**.

Tableau 1. Valeurs RVB capturées à l'aide du capteur TCS34725 pour les trois couleurs du feu de signalisation.

Feu tricolore	R	V	B
Rouge	149	56	61
Orange	123	77	61
Vert	67	100	90

Une bibliothèque MLP pour Arduino

Les valeurs RVB étant déterminées, nous avons besoin de l'implémentation du réseau neuronal pour la plateforme Arduino. Comme Processing utilise Java, nous ne pouvons pas simplement prendre le code de la classe Neural et l'ajouter à un projet Arduino. Nous avons donc légèrement modifié la classe Neural Java pour la transformer en classe C++. Dans l'univers Arduino, de tels objets de code réutilisables peuvent être transformés en « bibliothèques ». Celles-ci se composent d'une classe C++ et d'un fichier supplémentaire pour mettre en évidence les mots-clés de la classe. Si vous souhaitez écrire votre propre bibliothèque ou mieux comprendre le fonctionnement des classes C++ sur Arduino, il existe un excellent tutoriel sur le site web Arduino [5].

Le code de notre bibliothèque Neural se trouve dans *arduino/neural*. Les projets Arduino suivants incluent simplement le code source de la classe Neural dans le croquis pour simplifier l'écriture. Le fichier d'en-tête neural.h doit également être ajouté au dossier dans lequel est enregistré le projet.

L'utilisation de la classe Neural sur un Arduino est pour l'essentiel la même que dans Processing. La création de notre objet `network` à titre de variable globale est légèrement différente et réalisée comme suit :

```
Neural network;
```

Pour construire l'objet avec le nombre souhaité de nœuds d'entrée, cachés et de sortie, nous entrons les lignes suivantes :

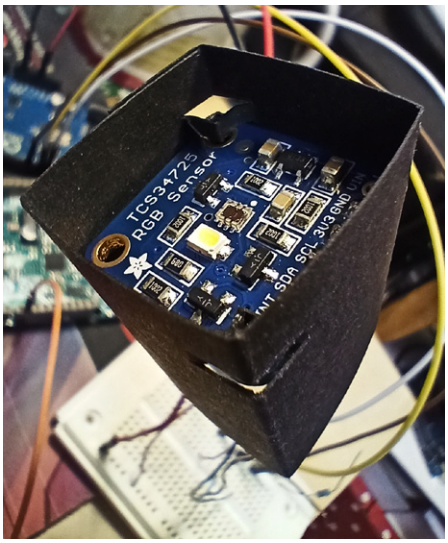


Figure 4. Capteur RVB TCS34725 avec son écran de protection noir.

```
network = new Neural(3,6,4);
```

La configuration de base des valeurs de biais et du taux d'apprentissage est ensuite codée comme précédemment dans Processing :

```
network.setLearningRate(0.5);
network.
setBiasInputToHidden(0.35);
network.
setBiasHiddenToOutput(0.60);
```

À partir de là, nous pouvons continuer à utiliser les méthodes appliquées précédemment dans Processing.

Détection des feux de circulation avec la carte Arduino

Nous pouvons maintenant commencer à explorer le MLP sur Arduino. Le croquis */arduino/tlight_detect/tlight_detect.ino* suit la même structure que le projet *Processing tlight_detect.pde*. Le réseau neuronal (3/6/4) est configuré à partir de la ligne 40, et il est entraîné avec les données RVB dans une boucle à partir de la ligne 51. Avant d'exécuter le code, les valeurs RVB pour « Rouge », « Orange » et « Vert » acquises précédemment doivent être saisies à partir de la ligne 56 :

```
teachRed(220, 56, 8);
teachAmber(216, 130, 11);
teachGreen(123, 150, 128);
```

Téléchargez le code et ouvrez le moniteur série pour visualiser la sortie (**fig. 5**). Comme précédemment, la vitesse de transmission doit être réglée sur de 9600 bauds. Le projet vérifie que le capteur RVB est fonctionnel avant d'éteindre la LED blanche pendant l'apprentissage. Une fois le MLP configuré, il effectue le cycle d'apprentissage 30.000 fois, améliorant à chaque fois sa capacité à classer chacune des trois couleurs.

Pour assurer un certain retour d'information pendant l'apprentissage, un point est émis tous les 1.000 cycles de boucle (3.000 époques d'apprentissage). Comparé au PC, l'apprentissage est un processus lent. Chaque appel à une fonction d'apprentissage (`learnRed()`, etc.) nécessite environ 5,55 ms. L'apprentissage des trois couleurs nécessite environ 8,5 min sur un Arduino M0 Pro doté d'un microcontrôleur SAMD21. Si vous souhaitez examiner le temps d'exécution de votre plateforme, cochez la case « Show Timestamp » dans le moniteur série et remplacez la ligne 66 par :

```
Serial.println(".");
```

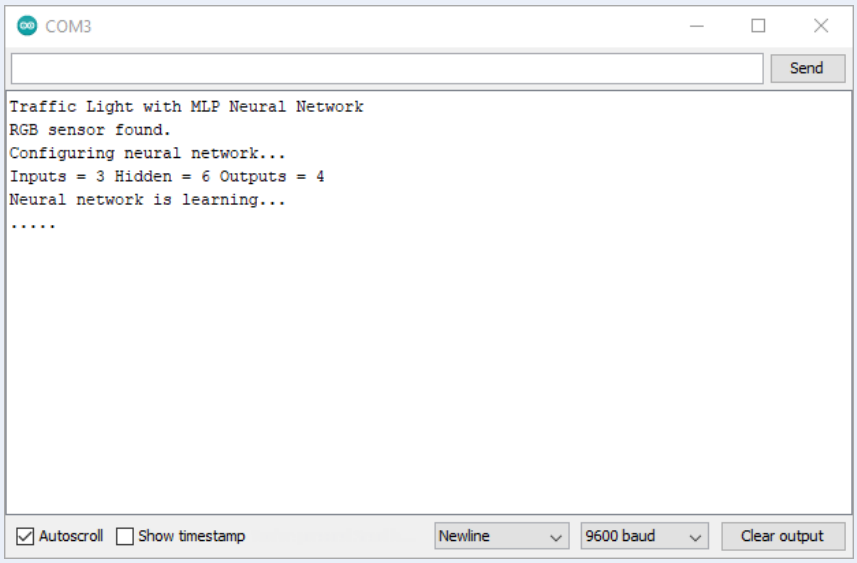


Figure 5. Sortie de tlight_detect.ino pendant l'apprentissage.

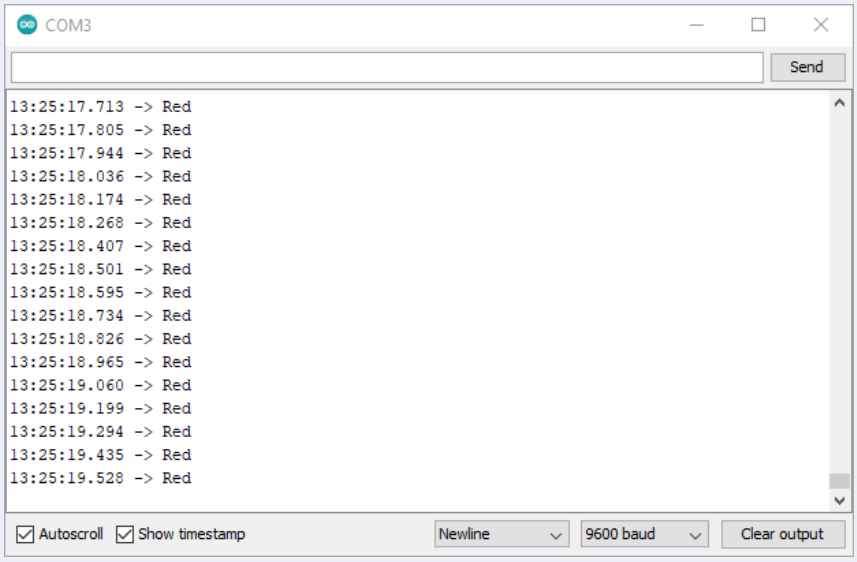


Figure 6. Résultat de tlight_detect.ino après apprentissage et détection des couleurs.

Vous ajoutez ainsi le caractère retour chariot et un horodatage est généré pour chaque caractère « . » produit.

Une fois l'apprentissage terminé, l'Arduino est capable de montrer immédiatement sa nouvelle compétence. Chaque fois qu'une couleur de feu de circulation est détectée, elle est envoyée au moniteur série (**fig. 6**). Au cours de l'expérimentation de l'auteur, le réseau neuronal a également détecté la couleur « orange » même si rien n'était placé devant le capteur. Bien que ce phénomène semble être lié à l'éclairage ambiant, il met en évidence une faiblesse de fonctionnement.

Pour améliorer le code, nous pouvons faire connaître au MLP les couleurs « autres », comme nous l'avons fait précédemment dans Processing. L'approche peut également servir à écarter la classification « orange » en l'absence d'image. Le croquis *tcsrgbsensor.ino* peut être utilisé pour acquérir les lectures du capteur pour l'environnement du feu de signalisation, le cadre et l'arrière-plan de l'image. Ces valeurs peuvent ensuite être entrées aux lignes 60 et suivantes du croquis *tlight_detect.ino* dans les appels de fonction `teachOther()`.

Les valeurs indiquées dans le **tableau 2** ont été atteintes et testées avec le croquis *tlight_detect.ino*. La classification s'est améliorée, mais la classification erronée « orange » en l'absence d'image n'a pas été totalement résolue. Comme toujours, des améliorations sont possibles !

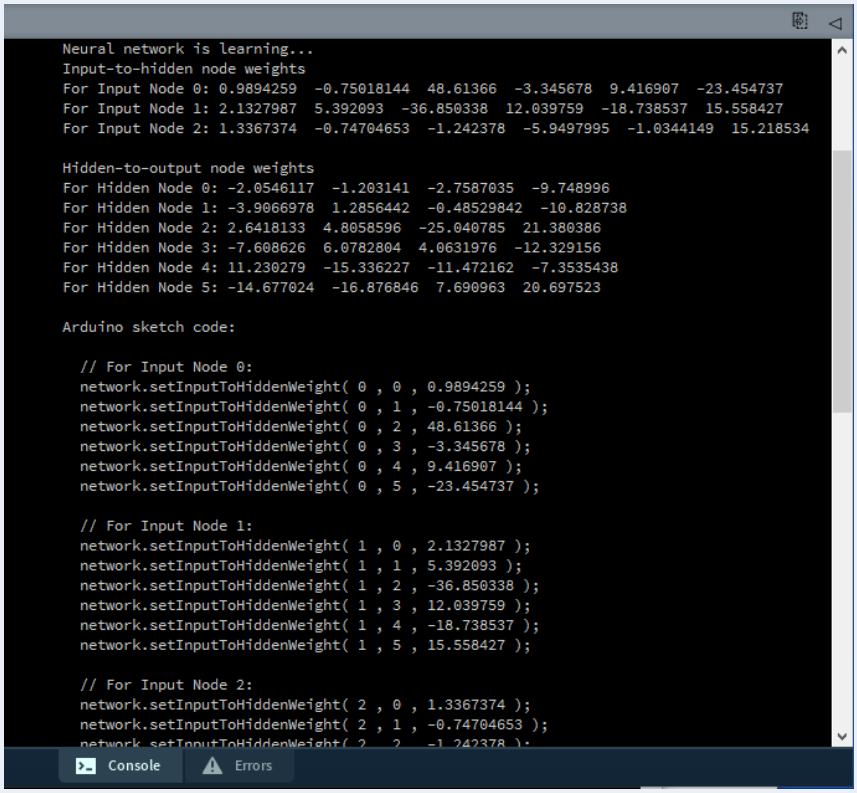


Figure 7. Un PC permet de calculer rapidement les poids en utilisant learnfast.pde dans Processing. La sortie de la console texte peut ensuite être collée dans tlight_weights.ino.

Tableau 2. Valeurs RVB capturées à l'aide du capteur TCS34725 pour les couleurs « indésirables ».

Couleur	R	V	B
Feu tricolore dans une ambiance sombre	92	90	82
Cadre blanc	92	90	75
Fond bleu	73	93	89

Apprentissage accéléré

Si vous ajoutez l'apprentissage des « autres » couleurs au croquis Arduino, l'attente sera d'environ 18 min pour qu'un Mo Pro assimile les classifications souhaitées. Il y a donc certainement la possibilité d'optimiser le processus. Puisque nous disposons d'un PC puissant capable de calculer les poids en moins d'une seconde, il serait logique de faire l'apprentissage avec, puis de transférer les résultats vers l'Arduino. Avec ces poids, nous avons également un moyen de programmer plusieurs microcontrôleurs avec les mêmes « connaissances ». Si les capteurs RVB fonctionnent tous de manière similaire par rapport à notre entrée, chaque microcontrôleur devrait classer correctement l'image du feu de signalisation. C'est donc ce que nous allons réaliser ensuite.

Nous revenons brièvement à Processing pour ouvrir le projet */arduino/learnfast/learnfast.pde*. L'ensemble de l'application s'exécute dans la fonction `setup()`. Le réseau neuronal est configuré avec les mêmes nœuds d'entrée, cachés et de sortie que ceux utilisés sur la carte Arduino (3/6/4). Dans la boucle d'apprentissage (ligne 36), les valeurs utilisées sont celles acquises par l'Arduino à l'aide du capteur RVB et du croquis *tcsrgbsensor.ino*. Lorsque le code est exécuté, il envoie du texte sur sa console. La dernière section contient le code qui configure tous les poids entrée-caché et caché-sortie (**fig. 7**). Il suffit de copier le code généré à partir de la ligne `// For Input Node =0` jusqu'à la fin de la sortie de texte.

De retour dans l'IDE Arduino, nous pouvons maintenant ouvrir le croquis */arduino/tlight_weights/tlight_weights.ino*. Il s'agit de la même chose que le croquis *tlight_detect.ino* mais, avec une préprogrammation des poids au lieu d'une phase d'entraînement du réseau neuronal. C'est le cas à la ligne 51 avec la fonction `importWeights()`. Collez simplement le code de la sortie de *learnfast.pde* dans la fonction `importWeights()` à la ligne 86 dans *tlight_weights.ino*. Programmez la carte Arduino, et elle devrait détecter avec précision les couleurs des feux de signalisation comme auparavant.

En fait, maintenant que nous avons ce processus d'apprentissage accéléré, nous pouvons aussi programmer le même croquis *tlight_weights.ino* dans un Arduino UNO. Il suffit de brancher le capteur RVB à la carte, d'ouvrir le moniteur série, et le fonctionnement est tout aussi précis qu'avec un Arduino Mo Pro ou DUE. À titre de comparaison, vous pouvez scruter la broche 9 pour voir combien de temps il faut à la méthode `calculateOutput()` pour effectuer les calculs.

Et ensuite ? D'autres systèmes embarqués ?

Alors, que pouvons-nous faire à partir de là ? Eh bien, nous disposons d'un réseau neuronal MLP opérationnel qui fonctionne à la fois sur PC et sur microcontrôleur. Nous disposons également d'un processus d'apprentissage accéléré pour générer les poids nécessaires aux applications sur microcontrôleur. Vous pouvez essayer d'appliquer ce MLP à des tâches trop difficiles à résoudre à l'aide d'instructions *if-else* et de limites fixes. Il pourrait même être possible de mettre en œuvre un projet simple de reconnaissance vocale pour identifier quelques mots. Vous pourriez également explorer les éléments suivants :

- La classe Neural utilise le type de données double. Peut-elle être accélérée en utilisant le type *float* tout en conservant sa précision ? Quelle vitesse d'exécution peut-elle atteindre ?
- La fonction sigmoïde utilise la fonction mathématique *exp()*, qui calcule l'exponentielle élevée à l'argument transmis. La fonction d'activation peut-elle être simplifiée par une approximation tout en assurant une classification précise ?
- Si vous vouliez tenter la reconnaissance vocale, comment prépareriez-vous un échantillon de voix pour le présenter à ce dispositif MLP ?
- Et si vous apportiez des modifications importantes ? Comment implémenter une deuxième couche de nœuds

cachés ? Pouvez-vous implémenter une fonction d'activation différente ?

Dans cette série d'articles, nous avons couvert un large domaine et découvert les premières recherches sur les neurones artificiels. À partir de là, nous avons examiné comment les MLP utilisent la rétropropagation pour apprendre, puis étudié leur fonctionnement en utilisant les puissantes capacités de traitement d'un PC. Nous avons ensuite porté le MLP sur un microcontrôleur à titre d'exemple de dispositif d'apprentissage automatique en périphérie de réseau. Si vous décidez de développer ces exemples plus avant, n'hésitez pas à partager vos résultats avec nous, chez Elektor.

210160-D-04

Des questions, des commentaires ?

Envoyez un courriel à l'auteur (stuart.cording@elektor.com).

Contributeurs

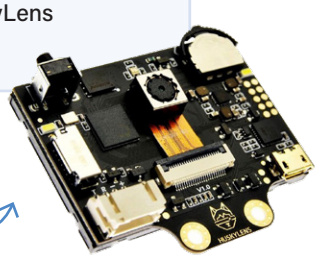
Idée, texte et images : **Stuart Cording**
Rédaction : **Jens Nickel, C. J. Abate**
Illustrations: **Patrick Wielders**
Mise en page : **Harmen Heida**
Traduction : **Pascal Godart**

Une agriculture plus performante

L'agriculture s'est toujours appuyée sur la nature et les saisons afin de déterminer le meilleur moment pour récolter et semer. La tradition a toujours dicté la date optimale pour planter en bénéficiant de la mousson. Cependant, avec l'évolution des conditions météorologiques, les rendements des cultures ont souffert. Tout cela est en train de changer grâce à l'IA. Des agriculteurs indiens participant à un projet pilote ont effectué leurs plantations d'arachide fin juin, soit trois semaines plus tard que la normale. C'est la suite Microsoft Cortana Intelligence qui a produit ces conseils. Grâce aux historiques de données climatiques, les recommandations reçues par les agriculteurs ont permis d'obtenir un rendement supérieur de 30 % en moyenne [6].

PRODUITS

- Livre en anglais, « Artificial Intelligence » de B. van Dam
  www.elektor.fr/18090
- Kit AIY Vision pour Raspberry Pi de Google
  www.elektor.fr/19365
- Caméra AI avec boîtier en silicone de HuskyLens
  www.elektor.fr/19248



LIENS

[1] M. Patrick, « ML at the Edge: a Practical Example », codemotion, 09/2020 : https://bit.ly/2ZQYipU
[2] Capteur de couleur RVB avec filtre IR et LED blanche – TCS34725 : http://bit.ly/2NKFS7T
[3] Dépôt GitHub – simple-neural-network : https://bit.ly/2ZHLv9p
[4] Dépôt GitHub – Adafruit_TCS34725 : http://bit.ly/37RJg81
[5] « Writing a Library for Arduino », Arduino : http://bit.ly/3aWeNaP
[6] « Digital Agriculture: Farmers in India are using AI to increase crop yields », Microsoft News Center, 11/2017 : http://bit.ly/2PacsQZ