



Rapport Neural Speech

Axel Bröns
axel.brons@edu.ece.fr

Valentin Kocijancic
valentin.kocijancic@edu.ece.fr

Hugo Rivière
hugo.riviere@edu.ece.fr

Ethan Petain
ethan.petain@edu.ece.fr

Lyon, le 6 avril 2025

Ce projet est un travail réalisé par des étudiants sans objectif de commercialisation. Nous attestons que ce travail est original, qu'il est le fruit d'un travail commun au groupe et qu'il a été rédigé de manière autonome.

Abstract

Le projet Neural Speech sous le nom de *Couleur Master* vise à développer un système embarqué de reconnaissance vocale capable d'identifier des mots simples, tels que des noms de couleurs, pour faciliter l'apprentissage des enfants ou des personnes en rééducation linguistique. Basé sur une carte Arduino Due, notre solution combine un traitement avancé du signal audio et un réseau de neurones pour analyser et classer les entrées vocales en temps réel. L'utilisation d'algorithmes performant et avancés comme par exemple l'algorithme MFC ont permis de réduire la complexité de la capacité du micro-contrôleur tout en conservant les informations essentielles.

Un réseau de neurones simple mais efficace, entraîné avec TensorFlow, permet une classification robuste des mots enregistrés, avec une précision dépassant 98%. Le système est intégré dans une boîte autonome alimentée par batterie, comprenant un microphone, des LEDs indicatrices et un bouton de commande. Des fonctionnalités bonus, comme la détection d'erreurs ou l'ajout d'une nouvelle couleur, ont été ajoutées pour améliorer l'expérience utilisateur.

Ce projet démontre avec succès l'application pratique de l'IA embarquée, tout en soulignant les défis liés aux contraintes matérielles. Les résultats obtenus ouvrent des perspectives d'amélioration, notamment avec une détection en temps réel ou alors avec un dataset élargi. Couleur Master illustre ainsi la puissance de l'électronique et de l'informatique combinées pour créer des outils pédagogiques innovants.

Contents

1 Objectifs	4
2 Glossaire	4
2.1 Termes	4
2.2 Acronymes	4
3 L'équipe	4
3.1 Présentation de l'équipe	4
3.2 Organisation de l'équipe	5
3.3 Diagramme de Gantt	5
4 Contexte et problématique	6
4.1 Contexte	6
4.2 Problématique	6
4.3 Spécifications techniques	6
5 Conception	6
5.1 Architecture fonctionnelle	6
5.2 Architecture matérielle	6
5.3 Architecture logicielle	7
6 Développement	7
6.1 Traitement du signal	7
6.1.1 Numériser le signal audio	7
6.1.2 Filtre RIF	8
6.1.3 Down-sampling	9
6.1.4 Buffer circulaire	9
6.2 Extraction des caractéristiques sonores	9
6.2.1 Enregistrement des données d'entraînement	11
6.3 Réseau de neurones	12
6.3.1 Complexité des convolutions	12
6.3.2 Entraînement	13
6.4 Prédictions	14
6.5 Bonus	14
6.5.1 Ajout d'une couleur	15
6.5.2 Reconnaissance du locuteur	15
6.5.3 Optimisation	15
7 Tests et validation	16
7.1 Traitement du signal	16
7.1.1 Numérisation	16
7.1.2 Filtre RIF	16
7.1.3 Buffer circulaire	18
7.2 Validation du bon fonctionnement du micro	18
7.3 Visualisation des caractéristiques sonores	18
7.4 Réseau de neurones	20
7.4.1 Réseau simple	20
7.4.2 Réseau complexe	21
7.4.3 Prédictions	24

8 Bilan	24
8.1 État d'avancement	24
8.2 Axe d'évolution et limite de notre solution	24
8.3 Bilan sur le travail d'équipe	25
9 Sources	26
10 Annexes	27

1 Objectifs

Ce document montre notre implication et les détails techniques du projet **Neural Speech** et atteste de la sincérité de notre travail. Vous pourrez trouver dans ce document les attendus du projet comme l'étude documentaire ou encore les différentes exigences techniques. On rappel que ce document est un travail réalisé par des étudiants sans objectif de commercialisation.

2 Glossaire

2.1 Termes

Nous listons les termes que nous allons fréquemment utiliser dans ce document :

Terme technique	Définition
Flash ADC	Type de convertisseur qui transforme un signal analogique en valeur numérique
Aliasing	Phénomène qui se produit lorsqu'un signal est échantillonné à une fréquence trop faible, il y a repliement de spectre
Down-sampling	Réduction la résolution spatiale ou temporelle du signal
Buffer	Zone de mémoire temporaire utilisée pour stocker des données en transit
Padding	Préserve la dimension des signaux lors du filtrage
Pooling	Technique qui consiste à réduire la taille des données tout en conservant l'information essentielle
Flatten	Opération qui permet de réduire en une dimension une structure de donnée multidimensionnelle
Epochs	Passage complet sur l'ensemble des données d'entraînement
Kernels	Petites matrices utilisées pour l'extraction de caractéristiques à partir de données d'entrée telles que des images

2.2 Acronymes

Le tableau des acronymes est volumineux alors nous l'avons mis en annexe, à la table 6.

3 L'équipe

3.1 Présentation de l'équipe

L'équipe est composée de 4 étudiants :

- **BRONS Axel** : Le technicien, il manie la pratique et la théorie à la perfection. Il a une grande compréhension du cours théorique et une expertise pour la réalisation pratique.
- **KOIJANCIC Valentin** : L'ingénieur, il sait résoudre les problèmes que ce soit au niveau de la maquette qu'au niveau réflexion dans le cours théorique. Au moindre problème, on fait appel à lui.
- **PETAIN Ethan** : L'informaticien, il peut faire n'importe quoi avec un ordinateur. Quand on n'arrive pas trouver la solution d'un problème de code ou d'informatique, on se tourne vers lui.

- **RIVIERE Hugo** : Rédacteur professionnel, il sait bien embellir la situation, de plus il apporte un soutien moral à toute l'équipe.



Axel Bröns



Valentin Kocijancic



Ethan Petain

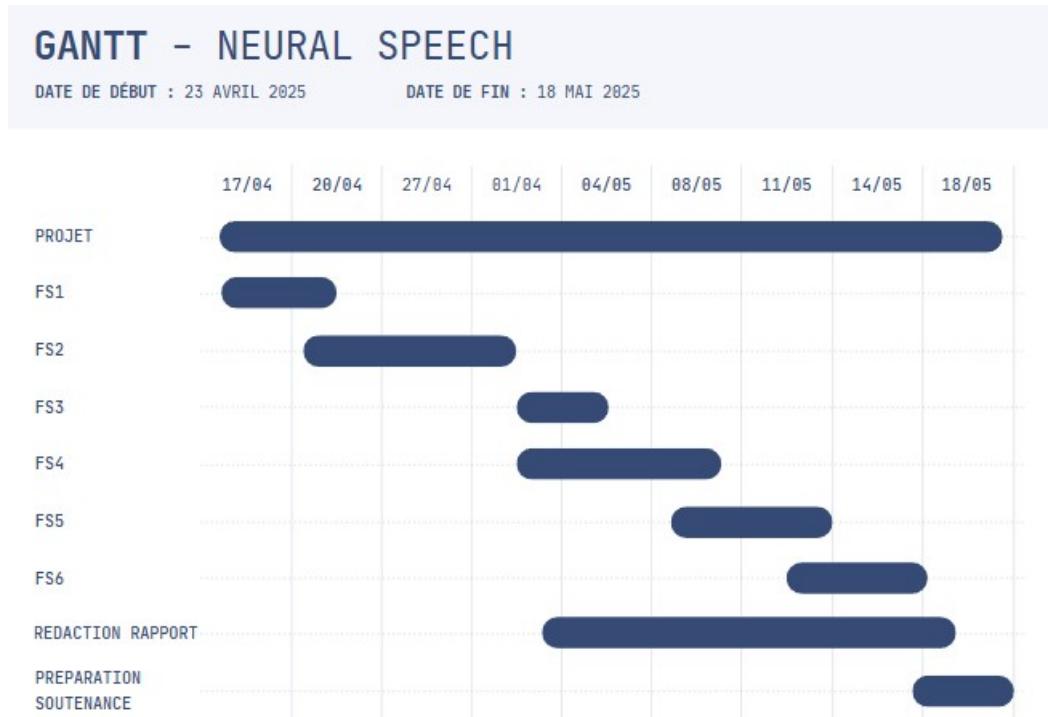


Hugo Rivi  re

3.2 Organisation de l'équipe

Notre organisation du travail dans l'équipe a été faite de manière très simple et efficace. Nous nous connaissons bien, ce qui a grandement aidé à trouver un équilibre et une ambiance de travail. Pour ce faire nous avons divisé les tâches entre nous. Hugo contribue au rapport, Valentin au FS3 et FS4, Ethan au FS5. Axel quand à lui a réalisé l'ensemble des tâches pour ne pas perdre le fil conducteur du projet et pouvoir rassembler les deux parties (IA et traitement du signal) plus efficacement.

3.3 Diagramme de Gantt



Le diagramme de Gantt nous a permis de ne pas se perdre dans toutes les exigences techniques du projet et ainsi pouvoir rendre un livrable à temps et fonctionnel.

4 Contexte et problématique

4.1 Contexte

Pour le projet d'électronique du second semestre de la troisième année du cycle préparatoire, nous avons choisi de faire la différence de trois couleurs. Nous avons convenu que ce projet servirai à aider les enfants pour apprendre les couleurs. Pour ce faire nous pensons pouvoir mettre en place toute les couleurs différentes dans le réseau de neurone. Ce qui sera un outil très utile et ludique pour les enfants. Cela leur permettra d'apprendre les couleurs mais aussi à prononcer correctement ces mots-là. En amélioration supplémentaire nous pourrons développer ce concept pour non plus seulement des couleurs mais bien plus de mots utiles pour les petits enfants qui apprennent à parler. De plus ce projet pourra être étendu au personne qui veulent apprendre le français, ou qui doivent suivre une rééducation linguistique.

4.2 Problématique

Ce projet répond à la problématique suivante :

Comment faciliter l'apprentissage des couleurs pour les personnes à bas âge ou sous rééducation linguistique ?

4.3 Spécifications techniques

Les attendus techniques sont de faire un projet de reconnaissance vocale basé sur une IA embarquée. Pour ce faire nous devrons réussir à numériser un signal audio, puis à le traiter numériquement pour pouvoir le passer dans un réseau de neurone. Ce projet doit être fait impérativement sur la carte Arduino Due pour que le système puisse être de l'embarqué.

5 Conception

5.1 Architecture fonctionnelle

L'architecture fonctionnelle suit les fonctions suivantes :

- Enregistrer un mot à l'aide d'un microphone
- Transformer l'audio pour avoir des valeurs
- Convertir ces valeurs en données exploitables
- Reconnaître le mot
- Afficher la lumière correspondant au mot dit

5.2 Architecture matérielle

Le matériel utilisé lors de ce projet est la carte Arduino DUE (Datasheet de l'Arduino Due), un microphone Arduino (Datasheet du MAX9814), d'un bouton poussoir, de LED rouges, verts, bleus et jaune, ainsi qu'un câble de connexion pour connecter la carte à un ordinateur. Pour les validations des différentes parties nous avons aussi utilisé du matériel en plus comme un oscilloscope et un GBF. Nous avons aussi utilisé une boîte en carton pour réaliser notre maquette ainsi qu'une batterie externe que nous connecterons à la carte pour que le système soit totalement indépendant et 100% embarqué. Voici le schéma de notre circuit sur KiCad :

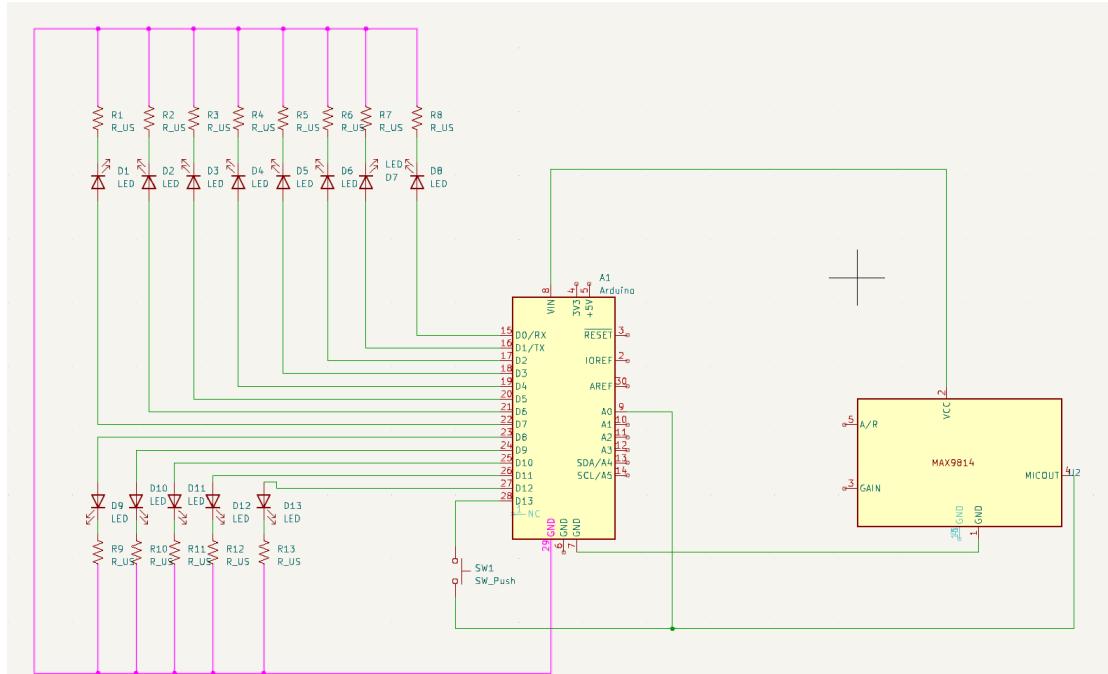


Figure 1: Schéma électrique **KiCad** de notre montage final

5.3 Architecture logicielle

Nous avons utilisé plusieurs langages pour mettre à bien ce projet, en effet, nous avons mis en place un code permettant d'enregistrer les différentes données d'entraînement en appliquant le traitement du signal adéquate, ensuite nous avons un code Python pour entraîner notre réseau de neurones et pour finir nous avons un dernier code C++ qui rassemble ces deux parties et qui concerne la prédictions avec les valeurs qu'on enregistre en direct via le microphone tout en faisant le traitement du signal comme dans le premier code. L'algorithme du code pour enregistrer les données est disponible à la figure 4, l'algorithme pour notre code Python est quant à lui disponible à la figure 6 et pour finir l'algorithme de prédictions est disponible à la figure 7.

6 Développement

6.1 Traitement du signal

6.1.1 Numériser le signal audio

Le but de cette partie est de passer notre signal à 32KHz et par la suite appliquer un filtre numérique. D'abord, nous devons absolument avoir l'Arduino Due en mode d'interruption sur timer pour pouvoir avoir un échantillonnage fixe et précis. Nous voulons avoir un signal avec une fréquence d'acquisition de 32 MHz. Pour cela, nous voulons trouver la valeur théorique du registre *RC*. Pour cela, nous utilisons le *Timer_Clock1*, ainsi on peut calculer la valeur de *RC* grâce à la formule suivante :

$$RC = \frac{MCK}{PF} = \frac{84000000}{2 \times 32000} \approx 1312 \quad (1)$$

Avec $MCK = 84MHz$, la valeur de la Master Clock, F la fréquence voulue et P le prescaler utilisé (dans notre cas notre prescaler vaut 2, parce qu'on utilise la *Clock1*). La raison de pourquoi nous voulons 32 KHz, est lié à une règle de base en traitement du signal : le théorème

de **Nyquist-Shannon**. Ce théorème nous dit que pour représenter un signal sans perte, il faut échantillonner à au moins 2 fois la fréquence maximale qu'on veut capter. Ainsi, si on veut capter toutes les fréquences jusqu'à 16 kHz, il faut au moins 32 kHz d'échantillonnage. Nous faisons alors un compromis entre la qualité du signal et charge de traitement. En effet, nous pourrons laisser à 44KHz, mais nous aurons une plus grande plage de valeurs coûteuse pour notre Arduino Due. Ainsi, nous décidons de choisir 32KHz puisque la fréquence seuil qu'un humain peut entendre est de 20Khz. Ainsi, ce théorème peut se représenter par la formule suivante :

$$f_s \geq 2f_m \quad (2)$$

Avec f_s la fréquence d'échantillonnage et f_m la fréquence maximale du signal. Dans notre cas, $f_s = 32000$ et $f_s \leq 16000$.

6.1.2 Filtre RIF

Après avoir échantillonné notre signal à 32Khz nous devons filtrer les fréquences supérieurs à 4Khz et les atténuer d'au moins 30dB. Nous devons les filtrer pour éviter l'effet de repliement (aliasing en anglais) ce qui déformerait notre enregistrement audio. Nous devons appliquer un filtre, pour rappel, un filtre est un système qui modifie certaines composantes d'un signal, généralement en fréquence. Il peut atténuer le bruit, supprimer les hautes ou basses fréquences, extraire un signal utile... Dans notre cas nous voulons atténuer les fréquences hautes fréquences. Nous choisissons alors le filtre numérique RIF pour plusieurs raisons. Pour commencer, le filtre RIF est un filtre avec une phase linéaire, qui préserve la forme d'onde (les voix sonnent naturellement) et qui a aucune distorsion de phase entre les fréquences. En revanche concernant le filtre RII, les phases sont non linéaire (différentes fréquences décalées différemment), provoque un "bavurage" des signaux vocaux et rend les consonnes "t", "k", "s" floues. On peut récapituler la différence entre les deux filtres à l'aide d'un tableau :

Critères	Filtre RIF	Filtre RII
Stabilité	Stable (pas de pôles)	Peut être instable
Réponse impulsionnelle	Finie (s'arrête)	Infinie
Phase	Linéaire	Non linéaire
Temps de calcul	Élevé	Plus rapide
Mémoire	Nécessite plus de mémoire	Nécessite moins de mémoire
Conception	Simple	Plus complexe

Table 1: Différence entre filtre RIF et filtre RII

Le filtre RIF [5] (Réponse Impulsionnelle Finie) est caractérisé par une réponse uniquement basée sur un nombre fini de valeurs du signal d'entrée. Ainsi, la sortie ne dépend pas des sorties précédentes. La sortie de ce filtre peut se définir grâce à la formule suivante :

$$y[n] = \sum_{k=0}^{N-1} b_k x[n - k] \quad (3)$$

Avec $y[n]$ les valeurs successives du signal de sortie, $x[n]$ les valeurs successives du signal d'entrée, b_k les coefficients de la fonction de transfert du filtre et N le nombre de coefficients (ordre du filtre). Nous distinguons deux filtres RIF différents uniquement par leurs coefficients. C'est ainsi que le choix de ces coefficients sont très important. Par ailleurs, la quantité de coefficients doit être aussi importante du fait de notre mémoire SRAM réduite sur l'Arduino Due.

6.1.3 Down-sampling

Du mot en français, sous-échantillonnage, consiste à réduire le nombre d'échantillons de données dans un jeu de données. En d'autres termes, ça permet dans notre cas de passer de 32KHz à un signal de 8KHz. Ainsi, on va prendre 1 donnée sur 4 données pour essentiellement réduire et économiser la mémoire de la carte Arduino Due. Pour implémenter ceci, on va simplement mettre un compteur dans notre code, si le compteur est à 4 dans la boucle, alors nous allons prendre la valeur et remettre à 0 le compteur et ainsi de suite.

6.1.4 Buffer circulaire

Pour pouvoir enregistrer en continue des valeurs sans faire déborder notre mémoire SRAM, nous allons implémenter un buffer circulaire [1]. Comme le nom l'indique, le buffer circulaire va stocker des valeurs avec une durée de vie puis va écraser les dernières valeurs en continue et ainsi de suite. Du coup, nous allons avoir des valeurs qui peuvent être traité à la suite et en permanence suivant des fenêtres. La taille de notre buffer doit être la même que le nombre de nos coefficients du filtre RIF de la section 6.1.2. En effet, d'après l'équation 3, le filtre calcule chaque échantillons de sortie, ainsi, il aura besoin d'accéder à l'échantillon en cours $x[n]$, et des $N - 1$ échantillons précédents. En d'autre termes, on doit donc toujours avoir N échantillons disponibles. Donc si notre buffer est trop grand, ce ne sera pas optimisé pour la mémoire de l'Arduino Due et à l'inverse si nous le mettons trop petit on va perdre des échantillons.

6.2 Extraction des caractéristiques sonores

L'extraction des caractéristique du signal est un moment clé dans notre projet, en effet c'est grâce à cette partie que nous pourrons communiquer les données de notre voix à notre réseau de neurones et ainsi notre machine pourra apprendre des "patterns" pour pouvoir reconnaître un mot plutôt qu'un autre. Pour ce faire nous allons appliquer la méthode MFCC (Mel-Frequency Cepstral Coefficients) pour réduire la taille de l'audio sans perdre les informations importantes qui composent notre voix. Nous allons utiliser le lien Github (ArduinoMFCC de ElectroniqueECE [2]) disponible où nous aurons toutes les fonctions à porté de main pour pouvoir les utiliser comme il le faut. Concernant le principe, il est divisé en 7 grandes parties :

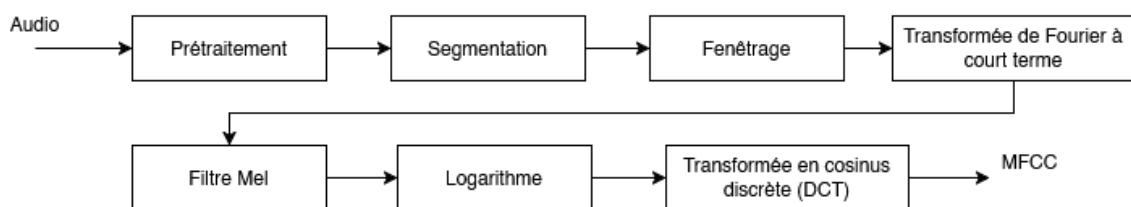


Figure 2: Algorithme d'extraction des MFCC

Pré-traitement Plus précisément, le pré-traitement consiste à prétraité le signal audio en appliquant un pré-emphase pour augmenter les hautes fréquences. Cela permet de réduire les problèmes liés à l'atténuation des hautes fréquences dans les systèmes de transmission. La formule de pré-emphase est la suivante :

$$y[n] = x[n] - \alpha x[n - 1]$$

où $x(t)$ est le signal d'entrée, $y(t)$ est le signal de sortie et α est le coefficient de pré-emphase (généralement compris entre 0,95 et 0,97).

La segmentation La segmentation consiste, elle, à divisé en trames le signal audio de taille fixe (généralement 20 à 40 ms) avec un certain chevauchement entre les trames adjacentes. Cette étape est nécessaire pour rendre compte de la nature non stationnaire des signaux audio.

Le fenêtrage Souvent nommé **Hamming** ou **Hanning** est appliquée à chaque trame pour minimiser les discontinuités aux bords de la trame. La fenêtre est définie comme suit :

$$w[n] = \frac{1}{2}(1 - \cos(\frac{2\pi n}{N-1}))$$

où $w(n)$ est la valeur de la fenêtre à l'échantillon n et N est la taille de la trame.

FFT La Transformée de Fourier à court terme (STFT) est appliquée à chaque trame pour obtenir le spectre de puissance. Cette étape permet de passer du domaine temporel au domaine fréquentiel.

Les filtres Mel Ils sont calculés à l'aide d'un banc de filtres en triangle espacés selon l'échelle de fréquence Mel. L'échelle Mel est une échelle de fréquence perceptuelle qui tient compte de la manière dont l'oreille humaine perçoit les fréquences. La relation entre la fréquence Mel m et la fréquence linéaire f est la suivante :

$$m = 2595 \times \log_{10}\left(1 + \frac{f}{700}\right)$$

Le **logarithme** de l'énergie de chaque filtre Mel est calculé. Cette étape permet de compresser les données en réduisant l'échelle dynamique.

DCT Pour finir, la Transformée en cosinus discrète (DCT) est appliquée au spectre log-Mel pour obtenir les coefficients MFCC.

Avant de passer à l'algorithme nous devons découper notre audio en frame de 256 échantillons. Mais pour ne pas perdre le maximum d'informations nous devons les superposer pour en quelques sorte garder l'effet "continu" de l'audio. On sait que notre enregistrement fait au total 8000 échantillons puisque nous voulons enregistrer 1 seconde (avec une fréquence de 8KHz). Nous avons vu sur les données que le professeur avait déjà enregistrer (`training.txt` de ElectroniqueECE) qu'il y aura 48 lignes de 13 coefficients. Nous voulons garder et avoir la même logique pour pouvoir entraîner et obtenir des résultat cohérents par la suite. Désormais nous pouvons calculer combien d'échantillons nous devons superposer. Voici le schéma de la situation :

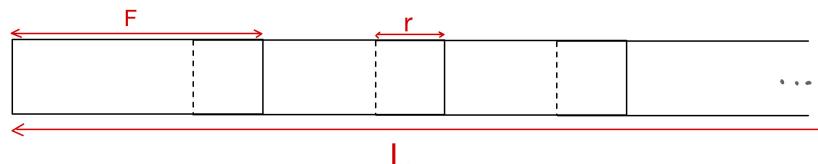


Figure 3: Schéma montrant le recouvrement, avec F le nombre d'échantillons dans une frame, L le nombre total d'échantillons dans l'enregistrement, r le recouvrement qu'on cherche.

D'après le schéma suivant nous pouvons déterminer plus facilement la superposition :

$$L = FN - r(N - 1)$$

Alors on isole r le "hop-length" et on obtient :

$$r = \frac{FN - L}{N - 1} \quad (4)$$

Ainsi, on sait que dans notre cas on a $F = 256$, $N = 48$ et $L = 8000$. Donc, on a :

$$r = \frac{256 \times 48 - 8000}{47} \approx 91.2$$

Ainsi le nombre d'échantillons entre une frame à une autre peut se trouver facilement suivant la formule suivante :

$$F - r \approx 256 - 91.2 \approx 164$$

On connaît maintenant le recouvrement, nous pourrons le mettre dans l'algorithme afin de garder seulement les frames qui nous intéressent. Nous appliquons alors l'algorithme MFCC qui va nous transformer chacune de nos 48 frames de 256 échantillons en 48 frames de 13 coefficients, ce qui nous réduit considérablement la taille de notre enregistrement. De plus, grâce à cette technique, nous pourrons en quelque sorte transformer notre audio en une image reconnaissable (voir figure 14). Nous avons le choix alors d'appliquer la dernière étape DCT, pour notre cas, ça allait réduire considérablement le nombre d'entrée ($624 \rightarrow 288$ car matrice de 48×6). Nous avons fait les tests entre les deux et nous avons pas vu de réelle différence. Alors nous avons fait le choix de garder les coefficients MFCC au lieu de DCT pour qu'on arrive mieux à les visualiser.

6.2.1 Enregistrement des données d'entraînement

Désormais, nous voulons prendre toutes les données après l'algorithme MFCC pour pouvoir les traiter dans le réseau de neurones pour nous donner une prédiction. Pour simplifier la mémoire de la carte nous avons déjà décider d'entrainer notre réseau de neurones directement sur TensorFlow ce qui nous éviterai d'avoir un très petit modèle qui ne prend pas beaucoup d'espace sur la carte. De plus nous avons vu et manipulé les réseaux de neurones durant les TP et nous avons vu que pour améliorer la performance il y avait beaucoup d'optimisateur via le Github de la bibliothèque **NeuralNetworks** de Giorgos Xou [3]. Par exemple, passer pour la mémoire EEPROM, utiliser la mémoire PROGMEM au lieu de RAM etc... Ainsi notre choix se focalisera principalement sur TensorFlow via Python, mais pour cela nous devons extraire toutes les données dans des fichiers textes pour pouvoir l'exporter directement dans notre programme Python. Nous allons les inscrire dans le Serial que nous allons récupérer grâce au logiciel PuTTY. Nous gardons les techniques vues dans la section 6.1 pour enregistrer nos données puisque nous devons utiliser le microphone. Nous implémenterons un bouton poussoir pour indiquer que le locuteur est prêt à parler et ensuite nous utiliserons une LED qui restera allumé pendant toute la durée de l'enregistrement. Ce choix nous évitera d'avoir le bruit du bouton ou alors la différence de temps lorsque le locuteur appuie et parle. Nous gagnerons en précision notamment parce que le temps de réaction d'un humain est très similaire (environ 200 - 300ms) ce qui signifie que si l'enregistrement dure 1 seconde, le son du mot sera parfaitement au centre de l'enregistrement total. Voici l'algorigramme complet de notre code pour enregistrer les valeurs :

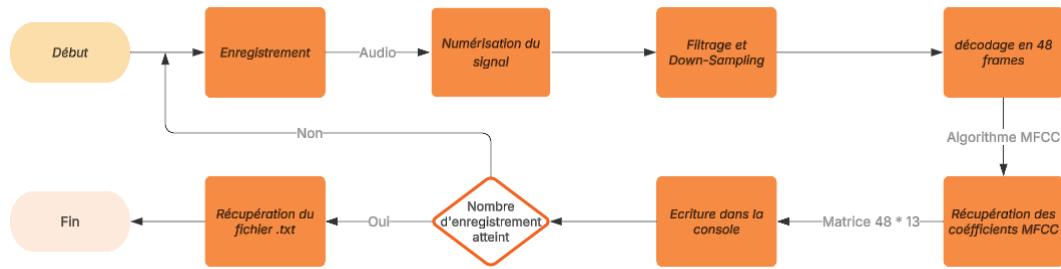


Figure 4: Algorigramme pour enregistrer nos données

6.3 Réseau de neurones

Dans cette partie nous allons voir comment on a mis en place notre réseau de neurones. Nous avons commencé directement à coder sur Python à l'aide de la librairie **TensorFlow** [4], parce que nous nous sommes dit que ça allait être plus pertinent et plus rapide d'entraîner sur une machine externe. Nous avons aussi eu l'ambition de mettre plus que deux couleurs et d'améliorer la précision au maximum pour être robuste et pouvoir prédire des couleurs avec n'importe quelles voix.

6.3.1 Complexité des convolutions

Nous avons d'abord pensé à mettre plusieurs couches de convolutions pour que la machine puisse reconnaître des caractéristiques uniques dans les MFCC. Mais le problème avec cette méthode est qu'il y aura besoin de beaucoup de couches de convolutions ce qui entraînera un plus lourd traitement sur la carte. En effet, les formules qu'on utilise le padding) 'same' ce qui ne réduit pas l'image. Par exemple si nous appliquons deux couches de convolutions avec deux max-pooling on aura la taille suivante :

Couche	Dimension de la sortie
Convolution 2D	(48, 13, 64)
Max-pooling 2D	(24, 6, 64)
Convolution 2D	(24, 6, 32)
Max-pooling 2D	(12, 3, 32)
Flatten	(1152)
Sortie	(1)

Table 2: Application des différentes couches de convolutions

On peut voir que pour chaque couches de convolutions on applique 64 kernels (ou 32 pour la deuxième couches) ce qui signifie que nous devrons manipuler dans le code Arduino des tableaux de $48 \times 13 \times 64$ ce qui dépasse complètement la mémoire de la carte. Si nous mettons une couche de convolutions avec un filtre appliqué, lorsque nous le testons sur TensorFlow, nos résultats ne sont pas concluant et mettent beaucoup trop longtemps à converger, voir complètement diverger. Pour trouver la bonne combinaison nous avons besoin de faire du fine-tuning et faire des sacrifices pour avoir une bonne rapidité tout en ayant des résultats performant. C'est ainsi que nous avons fait le choix de ne pas utiliser de convolutions.

6.3.2 Entraînement

Normalisation Pour entraîner notre réseau de neurones, nous avons d'abord dû normaliser nos données. En effet les données varient entre 1.5 et à peu près 4.5. Pour cela nous allons simplement normaliser grâce à la formule suivante :

$$x_{\text{norm}} = \frac{x_{\text{brut}} - \mu}{\sigma} \quad (5)$$

Avec σ l'écart-type de toutes les valeurs du tableau, μ la moyenne, x_{norm} les nouvelles valeurs après normalisation et x_{brut} les anciennes valeurs non normalisées. Ainsi nous allons appliquer cette formule pour l'entraînement mais aussi pour la prédictions car nous devons respecter le même format de nos données.

Shuffle En plus de la normalisation des données nous avons aussi mis une fonction `shuffle` pour mélanger l'ordre de nos données d'entraînement. En effet, si nous entraînons notre réseau de neurones avec un ordre bien précis notre réseau pourrait commencer à apprendre l'ordre des réponses. Même si nous savons que nous avons un petit dataset avec un réseau de neurones simple, si nous ajoutons cette fonctionnalité nous pourrons améliorer notre précision.

Les couches de convolution du réseau de neurones Après que les données ont été préparé et normalisé, nous pourrons les mettre dans le réseau de neurones mais nous devons d'abord réfléchir sur la structure du réseau de neurones. Pour cela nous avons pensé à plusieurs architecture incluant les convolutions, les couches denses et d'autres techniques comme par exemple les pooling. Au départ nous sommes partis sur double couches de convolution avec des couches profondes comme le présente le schéma ci-dessous:

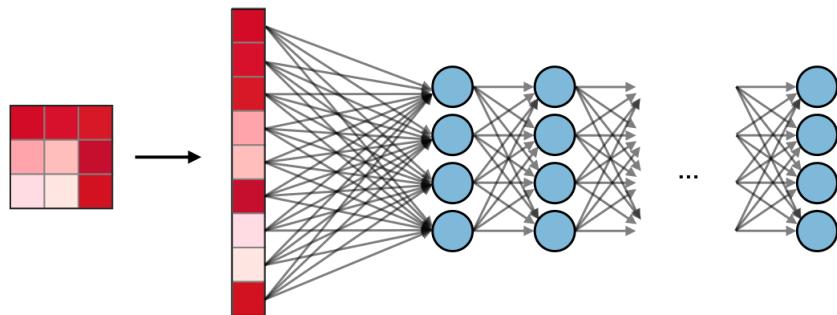


Figure 5: Exemple d'un réseau de neurones convolutionnel profond

Mais comparé à notre application que nous devons faire et ainsi le problème vu dans la section 6.3.1, nous avons pensé que notre réseau de neurones n'avait pas besoin d'être autant complexe sachant qu'on a une carte avec une mémoire limité et que nous avons pas beaucoup de données à entraîner (beaucoup moins de 5000). C'est ainsi que notre choix était sur un réseau de neurones simple.

Les couches de notre réseau de neurones Comme vu à la section 6.2, nous avons fait le choix de garder seulement les coefficients des MFCC, on aura alors en entrée $48 \times 13 = 624$ entrées, et 4 sorties car nous aurons une led par couleurs (bleu, rouge et verte) et une led en plus pour indiquer l'erreur (on développe cette partie dans la section 6.5.3). Nous devons à présent choisir l'architecture de notre réseau de neurones en faisant des essais. C'est ainsi que nous allons le développer le choix du nombre de couches profondes et tous les choix concernant l'architecture

de notre modèle dans la partie 7.4. Par ailleurs nous pouvons déjà construire l'algorigramme de notre code Python nous permettant d'entraîner notre réseau de neurones :

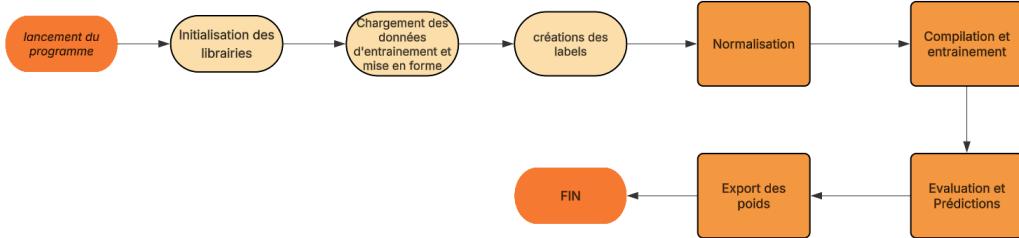


Figure 6: Algorigramme de notre code pour entraîner notre réseau de neurones

Liaison avec l'Arduino Due La liaison est très basique, nous affichons tout simplement les poids de notre réseau de neurones déjà entraîné sur la console de notre programme Python à l'aide de la fonction TensorFlow `model.get_weights()`. Grâce à cette fonction nous pourrons récupérer les poids déjà entraînés de notre réseau de neurones sur TensorFlow puis les coller dans notre réseau de neurones sur la carte Arduino Due.

6.4 Prédictions

Cette partie est la dernière de notre projet, elle concerne la mise en application de notre réseau de neurones. Nous allons rassembler toutes les parties de ce projet pour l'implémenter dans un seul code Arduino pour faire les prédictions directement sur celle-ci. Pour cela nous allons reprendre bloc par bloc pour former un code final qui va permettre de faire la prédictions de plusieurs couleurs. Voici l'algorigramme simplifié de notre code de prédictions final :

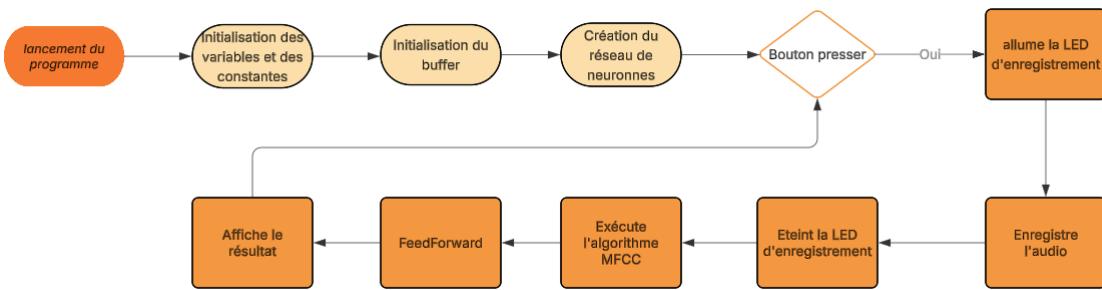


Figure 7: Algorigramme de notre code de prédictions

6.5 Bonus

Pour que notre projet soit 100% en embarqué nous avons décidé de mettre tout notre système dans une boîte fermée avec seulement les leds et le boutons qui sont apparents puis le tout sera branché à une batterie externe qui délivra 5V à notre carte avec le code de la prédictions déjà implémentée sur celle-ci. Voici un croquis de notre construction finale :

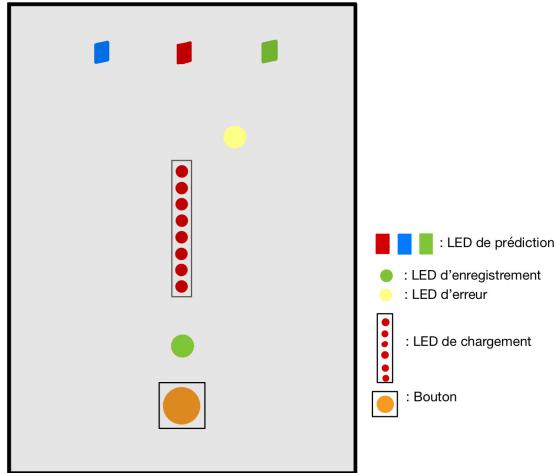


Figure 8: Croquis de notre boîte final avec l'alimentation à l'intérieur. (Vue de dessus)

6.5.1 Ajout d'une couleur

Comme on l'a dit dans la section 6.3 nous voulions rajouté plus d'une couleurs et de pouvoir reconnaître le locuteur qui est en train de parler. Pour cela, nous devons plus de données enregistrer par chacun de nous et les stocker bien comme il faut. Pour reconnaître les couleurs, nous avons rajouté des sorties, ainsi, si nous voulons ajouter n couleurs, il nous faudra rajouter n sorties avec au moins $50n$ enregistrements en plus.

6.5.2 Reconnaissance du locuteur

Concernant cette partie, le but ici sera de reconnaître le locuteur qui est en train de parler. Pour se faire nous avons rajouter un autre réseau de neurones parallèle à celui des couleurs et nous l'avons entraîné de la même manière sauf qu'au lieu de lui apprendre les couleurs nous lui avons appris à différencier entre les voix. Ainsi, en sortie du réseau de neurones nous aurons n neurones pour n locuteurs différent. Comme nous avons rajouté un neurones nous devrons logiquement entraîner la voix de chacun pour que la machine puisse apprendre la voix de chacun.

6.5.3 Optimisation

Nous avons aussi implémenter la fonctionnalité du mot qui n'existe pas dans le dataset de base. Pour ce faire nous avons décider d'enregistrer plus de mots à entrainer en accentuant sur les mots connus de la langue française. Et ensuite nous allons les connecter à un neurones qui sera dédié aux "erreurs". Par exemple sur 50 mots inconnus, nous allons enregistrer 10 mots différents finissant par le son -er comme par exemple les verbes du premiers groupe, puis les 10 prochains les verbes du deuxième groupe en finissant par -ir, puis les 10 prochains sur des adverbes qui finissent en -ement puis les 10 prochains par des mots ordinaire, aléatoire et pour finir les 10 derniers par du vide ou alors du bruits de fond. C'est avec cette technique que nous sommes performant sur tous les mots. De plus nous pourrons être très robuste sur des bruits de fond ou des mots qui ne sont pas des couleurs.

7 Tests et validation

7.1 Traitement du signal

7.1.1 Numérisation

D'après le développement de la section 6.1.1, et d'après la formule 1, nous savons que notre $RC = 1312$, ainsi, nous ajoutons cette ligne :

```
1 TC_SetRC(TC0, 0, 1312);
```

En faisant bien attention de sélectionner la *Clock1*. A l'issue de cette partie nous devons théoriquement obtenir un signal d'acquisition à 32 kHz. Pour le montrer nous allons le montrer grâce à l'oscilloscope et nous obtenons ces courbes :

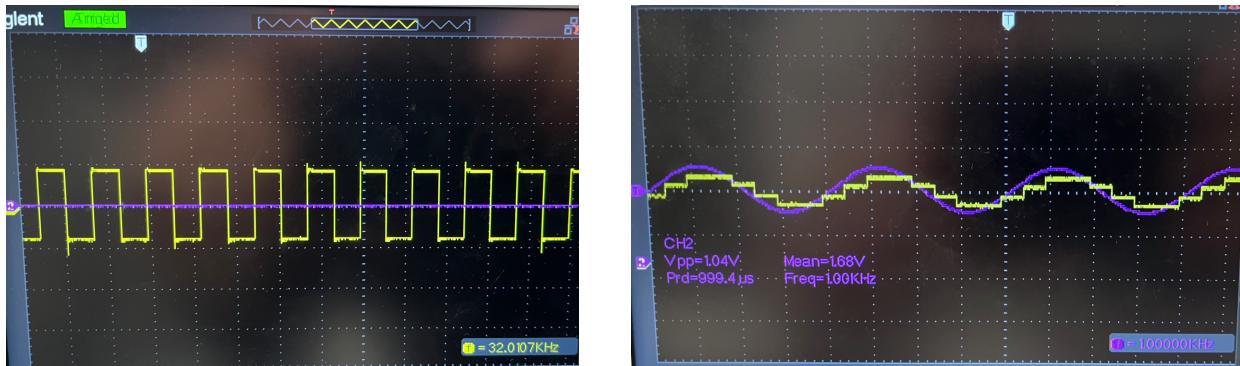


Figure 9: Validation du bon fonctionnement de notre signal d'acquisition 32 kHz

D'après la figure 9, on peut voir le bon fonctionnement et l'obtention de la courbe à 32 KHz.

7.1.2 Filtre RIF

Concernant le filtre, d'après la section 6.1.2 nous voulons implémenter un filtre RIF. Pour cela, nous allons nous aider du site web <http://t-filter.engineerjs.com/> [6] pour trouver les coefficients nécessaires à l'implémentation du filtre. Sur ce site nous pouvons implémenter un filtre et récupérer directement les coefficients pour les transvaser directement dans notre code. Sachant que nous travaillons sur l'Arduino Due, nous avons une mémoire très limitée, c'est ainsi que nous avons réduit volontairement le nombre de coefficient qui étaient trop important à stocker dans notre tableau. Nous avons alors agrandi l'espace de la bande de transition du filtre. Ainsi nous nous retrouvons avec la courbe suivante pour valider l'efficacité de nos coefficients :

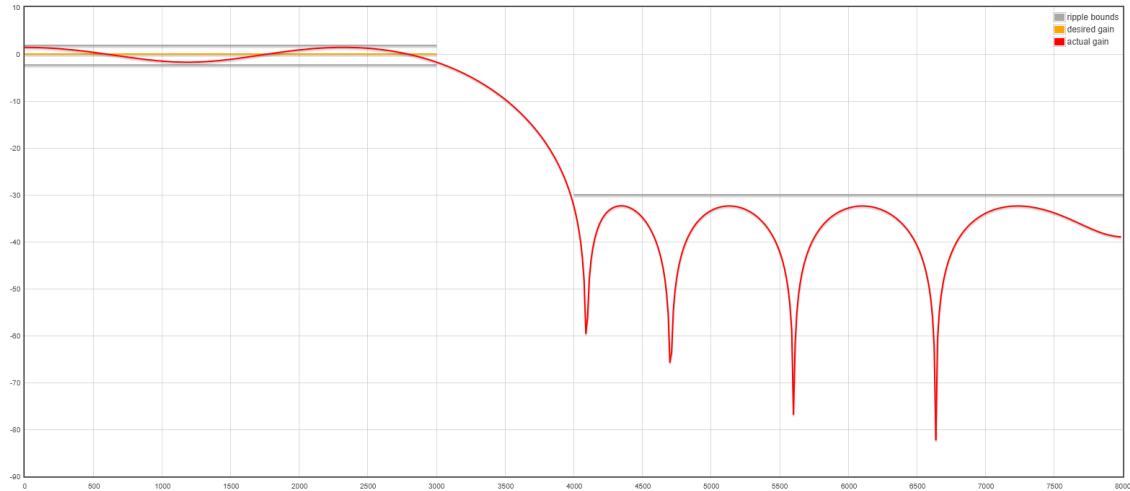


Figure 10: Courbe du filtre passe-bas RIF, qui atténue à partir de 4 kHz

On peut voir sur la figure 10, qu'entre 3000 Hz et 4000 Hz, le signal est beaucoup atténué. Pour valider, nous allons tester grâce à l'oscilloscope et un GBF, pour cela, nous allons envoyer un signal de moins de 4 kHz et nous sommes censé obtenir en sortie notre signal non changé, et à l'inverse si nous mettons un signal de plus de 4 kHz nous devons obtenir notre signal atténué.

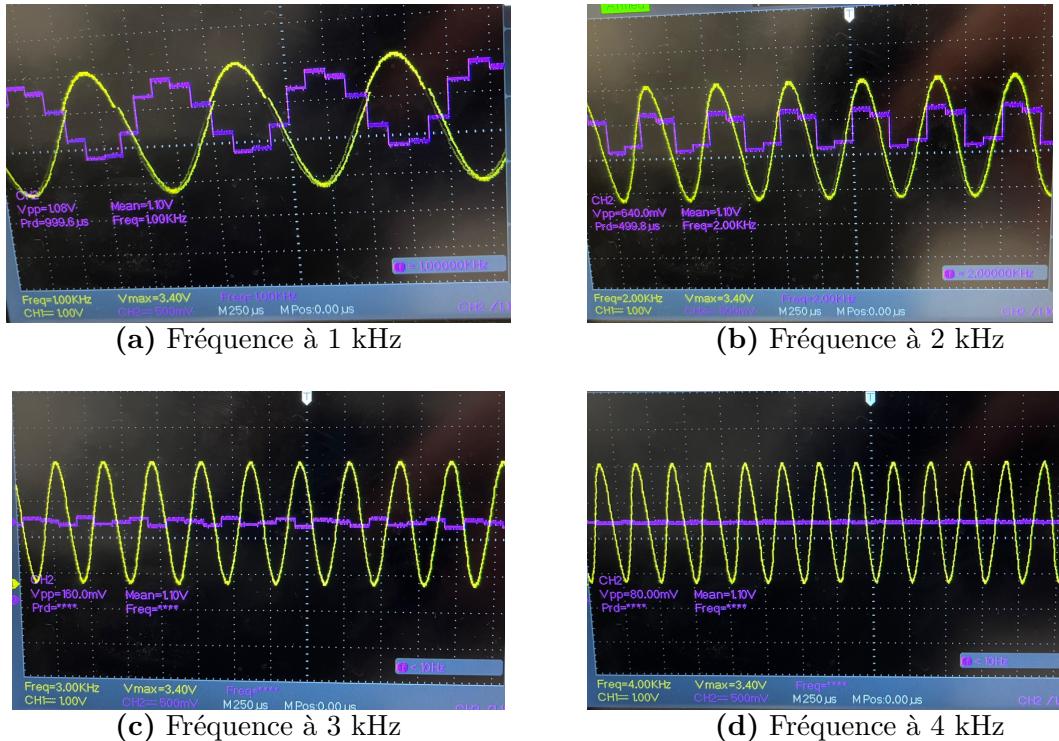


Figure 11: Visualisation du filtre ainsi que le down-sampling

On voit bien sur la photo (a) de la figure 11, le signal du GBF est à 1 kHz et que notre signal est échantillonné et avec qui on a appliqué un **down-sampling** (d'où le nombre de points inférieur à 32 kHz), on aperçoit qu'elle suit bien la courbe jaune du GBF. Cela signifie qu'elle n'est pas filtré. Par ailleurs, plus on se rapproche de la fréquence 4 kHz, on aperçoit que la courbe est

totalemenfiltré (voir la photo **(d)** de la figure 11. Avec la photo **(c)**, on voit que le filtrage se fait très tôt car on a un signal très atténue (mais pas 0 !) à 3 kHz, ce phénomène vient du fait qu'on a pris une distance de transition élevée (voir figure 10) pour éviter d'avoir trop de coefficient dans notre tableau Arduino. On peut conclure que notre filtre a bien été implémenté et notre down-sampling de $\frac{1}{4}$ de même. Grâce à ce filtre nous aurons plus de problème d'aliasing.

7.1.3 Buffer circulaire

Concernant le buffer-circulaire, comme expliqué dans la section 6.1.4, nous voulons stocker en continu des valeurs sans avoir à déplacer manuellement les éléments (ce qui économise du temps de traitement). Pour ce faire, pour valider que le buffer fonctionne correctement nous allons tout simplement afficher les valeurs de celui-ci dans la console.

Buffer Contents:							
3	6	9	4	4	9	9	7
9	10	9	10	7	8	9	7
8	7	8	9	8	7	7	8
9	11	10	8	6	9	11	7

Figure 12: Affichage dans la console le bon fonctionnement du buffer circulaire

On peut alors valider le bon fonctionnement du buffer.

7.2 Validation du bon fonctionnement du micro

Pour cette partie, il était demandé de valider le bon fonctionnement du micro après les filtres, nous allons utiliser le logiciel **PuTTY** pour extraire toutes les données dans un fichier texte puis après nous allons, à l'aide d'un script Python, convertir les données **txt** en fichier **bin** pour que Audacity puisse lire les données en brute. Nous enregistrons notre voix en disant le mot électronique, et nous devons obtenir un audio où on l'entend correctement.



Figure 13: Exemple d'audio obtenu sur le logiciel Audacity

Pour conclure, on arrive bien à reconnaître notre voix en disant électronique même si celle-ci semble un petit peu modifiée causés par les filtres et le down-sampling ajouté à notre signal audio. On peut conclure du bon fonctionnement des parties précédentes.

7.3 Visualisation des caractéristiques sonores

Découpage Comme on l'a dit dans la section 6.2, on souhaite avoir à la fin de cette partie un tableau avec dimension 48×13 pour pouvoir ensuite le mettre dans notre réseau de neurones. Pour cela, nous découpons d'abord notre enregistrement en frame de 256 échantillons chacun. Puis nous savons le nombre de frame que nous désirons et ainsi grâce à l'équation 6.2 nous savons que nous voulons recouvrir nos frames avec 164 échantillons. Pour commencer, nous devons avoir un enregistrement d'une seconde, pour cela nous décidons de prendre seulement 8000 échantillons

(parce que notre fréquence est à 8 kHz après le down-sampling). Nous validons cette opération grâce au logiciel **PuTTY** puis nous visualisons les données sur un logiciel d'éditeur de texte comme par exemple **Notepad++** et nous confirmons le nombres de lignes totales qui est de 8000. Après cette étapes, nous devons les découper en N frames par 256 échantillons (48 dans notre cas), nous stockons les valeurs dans un tableau à 2 dimensions en faisant attention à respecter le bon nombre d'échantillons que nous recouvrons et nous vérifions maintenant que nous avons un double tableau avec 48 lignes et 256 échantillons par lignes. Nous vérifions les 91 échantillons qui se recouvrent et se partagent entre les frames. Nous effectuons le même teste que précédemment avec **PuTTY** et **Notepad++** et nous validons le bon fonctionnement du découpage.

Algorithm MFCC Désormais, nous voulons appliquer l'algorithme des MFCC pour passer d'un tableau de dimension 48×256 en un tableau de 48×13 . Pour cela, à chaque fois que nous allons calculer la frame actuelle nous allons exécuter cette commande dans notre boucle :

```
1 mymfcc.compute(frames[f], mfcc);
```

Avec **frames[f]** la frame actuelle et **mfcc** le tableau de sortie. Pour valider ce passage, nous allons afficher dans la console toutes les frames avec leur coefficients, on valide la bonne dimension de celle-ci. Maintenant, pour valider le bon fonctionnement de l'algorithme MFCC, nous allons visualiser les MFCC à l'aide du script Python qui a été donné lors des TP. On enregistre 2 fois les mots bleus, rouges et verts pour les comparer entre eux. Voici les enregistrement que nous obtenons:

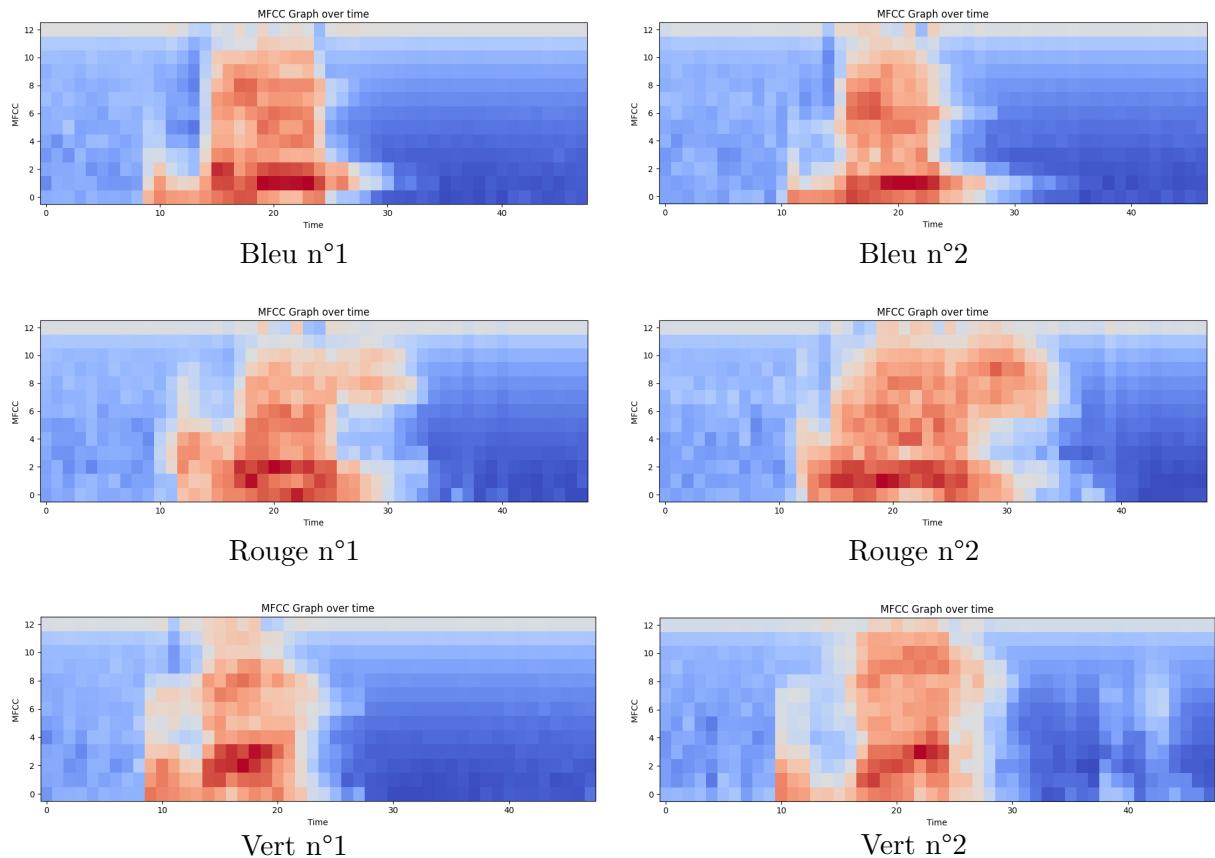


Figure 14: Visualisation des MFCC enregistrés

On peut constater que les motifs sont bien issus d'enregistrements différents mais on arrive

quand même à extraire certaines particularités entre les enregistrements bleus, rouges et verts, notamment par la forme de celle-ci. Même si nous les avons pas utilisé lors de notre réseau de neurones nous avons aussi visualisé la DCT sur un de nos enregistrement pour valider le bon fonctionnement de notre module et nous avons eu cette visualisation :

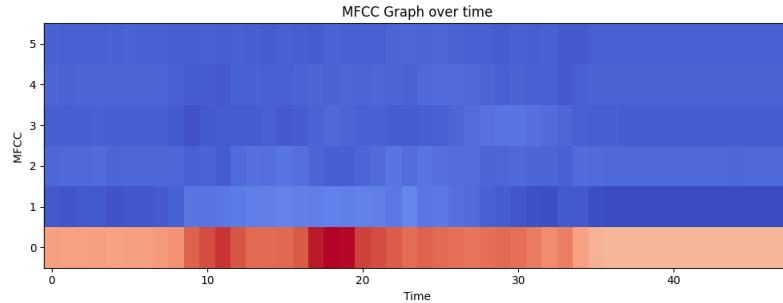


Figure 15: Visualisation des coefficients après DCT. Ici il y a 6 coefficients.

Ainsi notre algorithme est bien prêt pour enregistrer nos données d'entraînement pour notre réseau de neurones.

7.4 Réseau de neurones

7.4.1 Réseau simple

Maintenant que nous avons les MFCC prêt, nous avons enregistré d'abord 165 enregistrements (50 bleus, 50 rouges, 50 verts, et 5 de chaque pour les données de testes). Nous avons d'abord regardé comment réagirai nos prédictions si nous mettons un réseau de neurones très basique, c'est à dire, les couches d'entrées et les couches de sortie comme sur ce schéma :

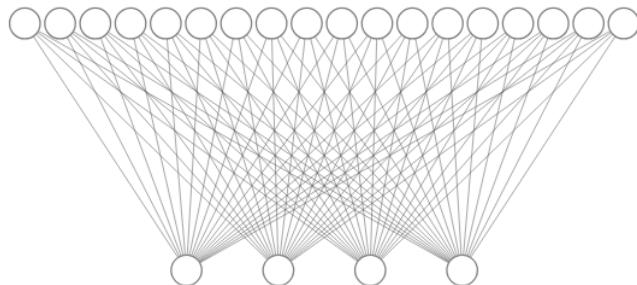


Figure 16: Réseau de neurones basiques (624 entrées et 4 sorties) tourné horizontalement

Et nous entraînons ce modèle sur 64 epochs et nous faisons nos prédictions sur nos données de tests et nous obtenons les précisions suivantes :

	Mot testé	Bleu	Rouge	Vert
1	Bleu	0.9808915	0.0099078	0.0092007
2	Rouge	0.0282434	0.9690930	0.0026637
3	Vert	0.0012404	0.0000397	0.9987199
:	:	:	:	:
13	Bleu	0.8619314	0.0719171	0.0661516
14	Rouge	0.0321044	0.9488044	0.0190913
15	Vert	0.0157210	0.0104149	0.9738640

Table 3: Prédictions pour un réseau de neurones simple

Les prédictions sont correctes, pour évaluer l'erreur de notre réseau nous avons pensé que ça allait être plus optimale de regarder la précision sur nos données de tests (**accuracy** sur TensorFlow) et non la MSE car elle est linéaire. Durant l'entraînement, nous avons fait de la classification, du coup notre fonction de perte est la **categorical_crossentropy** [4], qui fonctionne très bien pour des classification comme dans notre cas. Or dans ce cas nous avons seulement 5 données de tests par couleurs ce qui n'est pas assez pour avoir une précisions significative (notre précision est de 100%, avec une perte en fin d'entraînement à 0.01).

7.4.2 Réseau complexe

Pour pallier au problème de la partie 7.4.1, nous avons enregistrer plus de valeurs pour nos entraînements pour espérer être plus précis. Nous avons aussi décider d'enregistrer les valeurs pour la LED d'erreur développé à la section 6.5.3. Nous avons réparti ainsi les enregistrement suivant ce tableau :

	Axel			Ethan			Valentin			Hugo			Erreur	Total
	Bleu	Rouge	Vert	Bleu	Rouge	Vert	Bleu	Rouge	Vert	Bleu	Rouge	Vert		
Training	50	50	50	50	50	50	50	50	50	50	50	50	150	750
Test	5	5	5	5	5	5	5	5	5	5	5	5	15	75

Table 4: Répartition de nos enregistrements

Avec ceci, nous prenons le même réseau de neurones que la figure 16 et nous l'entraînons avec 50 epochs. Nous voyons que nous avons nos premières erreurs. En effet, 50 epochs n'étant pas assez pour 750 données en tout, alors nous décidons d'en mettre plus. Mais sur Python, nous obtenons de très bon résultat, par ailleurs, quand nous les essayons sur nos cartes, pour tester avec de nouvelles valeurs on voit que notre modèle n'est pas performant. Etant conscient du grand nombre d'epochs, nous avons décidé de complexifier notre réseau de neurones pour qu'il apprenne plus rapidement et avec plus de précisions. Ainsi nous avons ajouté deux couches de neurones profondes pour être avec $\{624, 32, 12, 4\}$.

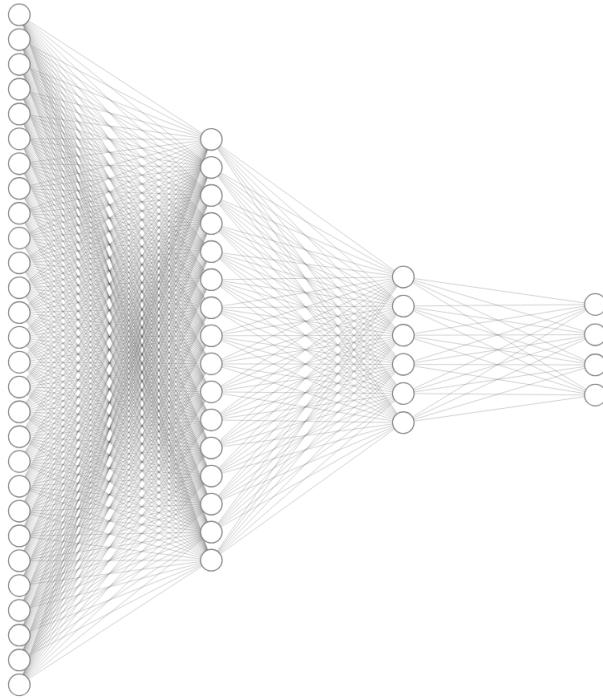


Figure 17: Réseau de neurones plus complexe avec comme architecture $\{624, 32, 12, 4\}$ (nous avons volontairement mis moins de neurones sur le schéma pour que ça soit compréhensible)

Or ici, lorsque nous faisons nos prédictions nous obtenons des résultats très convaincant pour la suite :

	Mot testé	Bleu	Rouge	Vert	Erreur
1	Bleu	0.9994659	0.0000060	0.0005232	0.0000049
2	Rouge	0.0000307	0.9990181	0.0009508	0.0000005
3	Vert	0.0000019	0.0001382	0.9998598	0.0000000
4	Erreur	0.0000342	0.0024410	0.0000426	0.9974822
:	:	:	:	:	:
72	Bleu	0.9995578	0.0004174	0.0000135	0.0000113
73	Rouge	0.0011244	0.9988153	0.0000603	0.0000000
74	Vert	0.0001200	0.0002301	0.9996244	0.0000255
75	Erreur	0.0000175	0.0023316	0.0000843	0.9975666

Table 5: Prédictions pour un réseau de neurones complexe

Notre précision ici est de **98.46%** avec une perte en fin d'entraînement de < 0.001 . Or ici, d'être trop précis nous questionne sur la possibilité d'être en sur-apprentissage et donc de ne pas être performant sur de nouvelles données. Pour être sûr nous avons décidé de tracer les courbes d'apprentissage pour suivre la tendance du `train_loss`.

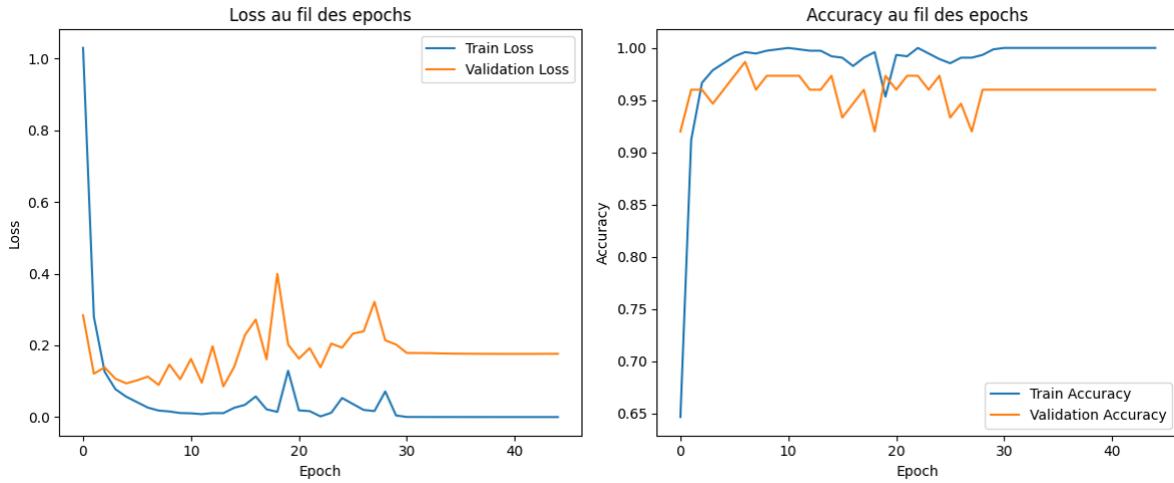


Figure 18: Courbes d'apprentissage nous montrant un sur-apprentissage

Pour pallier ce problème nous avons décidé de d'abord ajouter la fonction TensorFlow Dropout [4] qui permet de réduire le nombre de neurones volontairement en leur ajoutant du "bruit" pour que le réseau apprenne les vrais motifs et pas le motif précisément. De plus nous avons décidé de réduire la complexité tout en gardant une couche profonde tel le schéma suivant :

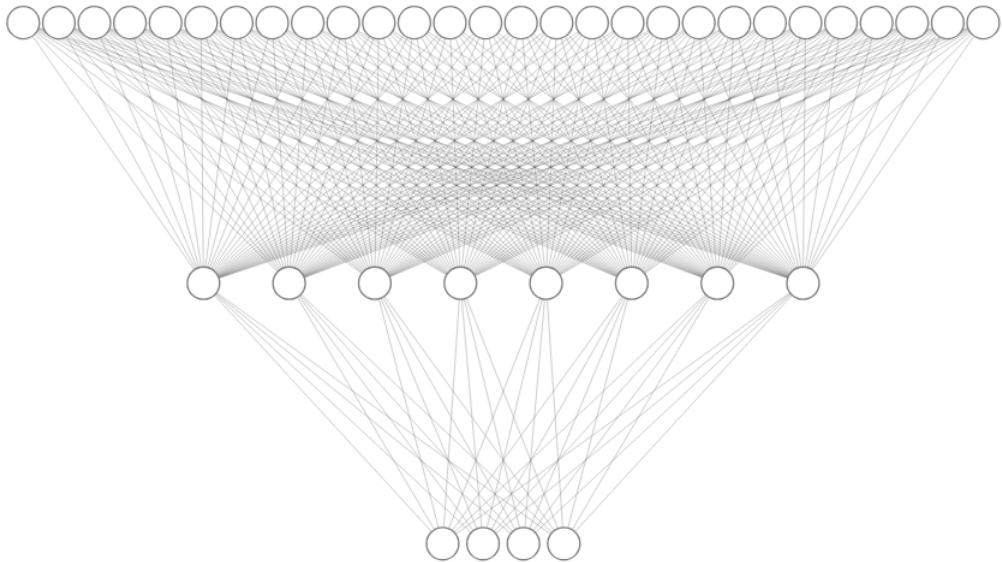


Figure 19: Architecture du réseau de neurones avec seulement une couche profonde (réduite pour meilleure compréhension)

Nous retracions les courbes d'apprentissage et nous voyons que les résultats sont bien meilleurs:

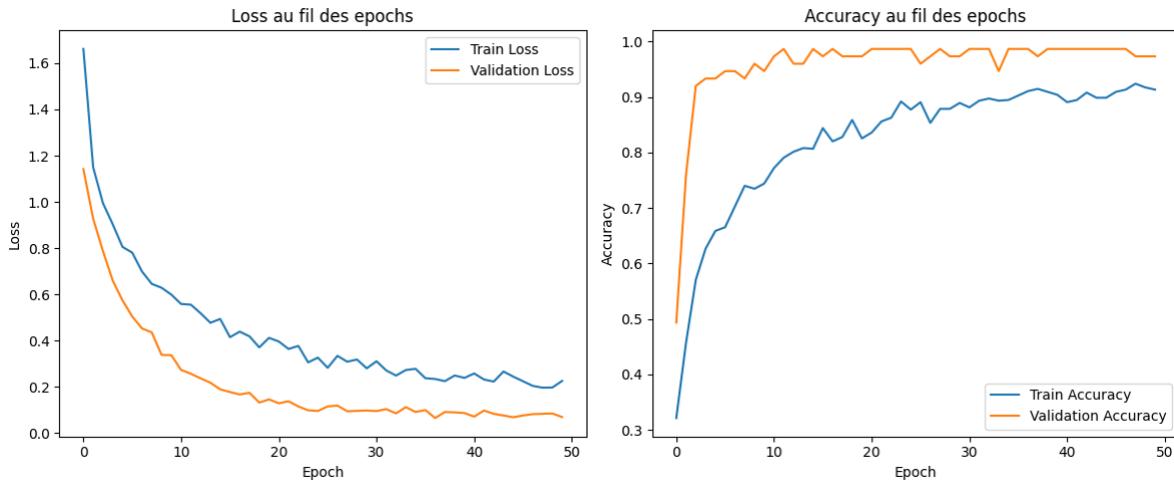


Figure 20: Nouvelle courbe d'apprentissage associé à l'architecture de la figure 19

Ainsi nous gardons ce réseau de neurones pour faire la prédictions sur la carte Arduino. Pour cela, on a juste à copier les poids ainsi que les biais sur la carte Arduino et utiliser seulement la fonction `FeedForward` de la bibliothèque `NeuralNetworks` de Giorgos Xou [3]. (voir l'algorigramme 7). Nous obtenons ainsi une très bonne précision.

7.4.3 Prédictions

Après avoir notre réseau de neurones prêt et ensuite implémenter les poids sur la carte, nous avons fait 10 essais et les 10 essais ont été concluant et ainsi pu prédire chaque couleurs ainsi que la led d'erreur si nous disons rien dans le micro ou faisant du bruit de fond.

8 Bilan

8.1 État d'avancement

Pour conclure, chaque partie de notre projet fonctionne parfaitement. Ce projet nous a passionné, ainsi nous sommes allé plus loin que ceux qui était demandé (ajout de couleur, ajout d'une led d'erreur, optimisation dans le traitement du signal en utilisant PROGMEM, optimisation de la performance en améliorant la précision de notre réseau de neurones).

8.2 Axe d'évolution et limite de notre solution

Par ailleurs, nous avons constaté des petits points d'améliorations dans notre projet.

Plus d'enregistrements Premièrement on pensait ajouter plus de données d'entraînement et aussi enregistrer des données de tests indépendamment des données d'entraînement, en effet, lorsque nous enregistrons les données d'entraînement nous enregistrons les données de test juste après, ce qui fait qu'on était dans la même pièce avec les mêmes conditions. Ainsi, si nous sommes dans une pièce avec beaucoup de résonance ce n'est pas la même que si nous sommes dans une pièce sans petit bruit de fond. Ainsi pour pallier ce problème, nous avons deux solutions, soit améliorer et faire une étude encore plus poussé du traitement du signal et / ou entraîner avec d'autres personnes. Le nombre de personnes qui a enregistré est aussi un problème pour

nous, étant seulement 4 personnes, le modèle est très sensible au changement de voix, malgré les variations de voix qu'on a fait.

Reconnaissance du locuteur Dans la section 6.5.2, nous savons comment faire reconnaître le locuteur, mais nous n'avons volontairement pas implémenté ce système et nous voulions nous concentrer sur l'optimisation de la performance de celui-ci. De plus, puisque notre modèle est aussi sensible dans les milieux dans lequel on enregistre, il fallait que chacune des personnes s'enregistre dans des pièces différentes pour que la machine apprenne vraiment la tonalité de voix et non le milieu.

Amélioration poussée du traitement du signal Pour pallier au problèmes ci-dessus, il serait intéressant pour nous de faire une étude plus poussé du traitement de signal pour vraiment atténuer au maximum les bruits parasites et vraiment se focaliser sur la voix de la personne. Mais pour cela, nous devons nous y pencher plus longtemps.

Ajout des couches de convolutions Pour résoudre les différents problème vu au dessus, il serait intéressant de pouvoir ajouter des couches de convolutions avec plusieurs filtres, mais comme expliqué dans la section 6.3.1 la problème vient de la mémoire de la carte, ainsi, nous pourrons penser à économiser vraiment en profondeur grâce à une étude plus poussé sur les registres de la carte.

Réduction du temps de calcul Même si notre système est très rapide (environ 3 secondes pour détecter le mot) nous pourrons éventuellement améliorer le temps de calcul notamment en faisant tournant le buffer indéfiniment en faisant des traitements du signal en continu sur celui-ci permettant de reconnaître presque en temps réel les mots qu'il arrive à reconnaître. Pour cela nous devrons utiliser le DMA (Direct Memory Access).

8.3 Bilan sur le travail d'équipe

Ce travail en équipe a encore une fois prouver notre mise au travail rapide et efficace ensemble. En effet, ce projet est le cinquième projet que l'on fait ensemble alors notre organisation n'est plus à prouver. Nous avons encore beaucoup de chose à apprendre individuellement comme certaines parties plus exigeantes techniquement que les autres mais notre cohésion reste à son maximum. Nous avons fini le projet en avance sans autre problème apparent. Notre mise au travail bien en amont nous a permis de ne pas être sous l'eau pour la dernière semaine puisque notre projet avait pratiquement été fini en 3 semaines.

9 Sources

References

- [1] Bertrand Vandepoortaele, Hugues Gilliard. Architecture pour le traitement numérique du signal (tns), 2018. Consulté le 3 mai 2025. URL: https://bvdp.inetdoc.net/files/iut/tp_tns/cours_tns.pdf.
- [2] Foued Derraz. arduinomfcc, 2023. Algorithme MFCC. URL: <https://github.com/FouedDrz/arduinoMFCC/tree/main>.
- [3] Giorgos Xou. Neural networks github, 2025. URL: <https://github.com/GiorgosXou/NeuralNetworks>.
- [4] Google. Tensorflow, 2025. Consulté le 10 mai 2025. URL: <https://www.tensorflow.org/>.
- [5] INSA Rouen. Traitement du signal - cours 9, 2024. URL: https://moodle.insa-rouen.fr/pluginfile.php/7255/mod_resource/content/1/cours9.pdf.
- [6] TFilter. Filter fir, 2025. URL: <http://t-filter.engineerjs.com/>.

10 Annexes

Acronyme	Signification	Explication
RIF	Réseau à Impulsions Finies	Filtre numérique dont la réponse impulsionnelle est de durée finie. Ce filtre est utiliser pour traiter les fréquence indésirable.
RII	Réseau à Impulsions Infinites	Filtre numérique avec une réponse impulsionnelle théoriquement infinie. Ce filtre est plus efficace que le RIF mais plus instable.
CNN	Convolutional Neural Network	Type d'architecture de deep learning spécialisé dans le traitement d'images ou de signaux.
LED	Light Emitting Diode	Composant électronique qui émet de la lumière lorsqu'il est parcouru par un courant.
MFCC	Mel Frequency Cepstral Coefficients	Nombre qui permet l'analyse du son, comme la reconnaissance vocale.
RAM	Random Access Memory	Mémoire rapide et volatile utilisée pour stocker temporairement les données.
SRAM	Static Random Access Memory	Mémoire rapide et volatile utilisée pour stocker temporairement les données, plus rapide que la RAM.
MCK	Master Clock	Signal d'horloge utilisé pour synchroniser les composants numériques
RC	Résistance - Capacité	filtre analogique composée d'une résistance et d'un condensateur
GBF	Générateur de Basse Fréquence	générateur de signal permettant de produire des ondes sinusoïdales, carrées ou triangulaires à basse fréquence
DCT	Discrete Cosine Transform	Transformation d'un signal en une somme de cosinus, mettant en valeur les fréquences dominantes.
FFT	Fast Fourier Transform	Transformation d'un signal temporel en spectre de fréquences.
MEL	Échelle de Mel	échelle perceptuelle de fréquences qui imite la façon dont l'oreille humaine perçoit les sons.
EEPROM	Electrically Erasable Programmable Read-Only Memory	Mémoire non volatile qu'on peut écrire et effacer électriquement, utilisée pour stocker des données même après mise hors tension.
PROGMEM	Program Memory	Mot-clé utilisé pour stocker des données constantes dans la mémoire flash du programme au lieu de la RAM.

Table 6: Acronyme de la partie 2.2