Rendu Projet microIA (Imene Yahiaoui _ Sourour Gazzeh)

Description du projet

Notre projet vise à identifier les chants d'oiseaux à partir de fichiers de test en utilisant une carte Nucleo-64 STM32L476 qui possède une faible quantité de RAM et de stockage (1 MB Flash, 128 KB SRAM) pour jouer les enregistrements, et à afficher les résultats dans la console. Nous avons construit notre base de données en récupérant des enregistrement grâce à l'API du site Xeno-canto, présentée pendant le cours, pour atteindre cet objectif.

Description du flux du travail

Nous avons choisi d'avoir quatre classes dans notre base de données pour quatre espèces (Bruant jaune, Bruant zizi, Coucou gris et Gobemouche gris)

Nous avons récupéré tous les enregistrements de chants de qualité A et B pour les toutes les espèces et les chants de qualité C pour le Gobemouche gris car il n'a pas autant d'enregistrements de bonne qualité que les autres espèces.

Pour cela, Nous avons utilisé la librairie python xeno-canto qui est un wrapper d'api désigner afin d'aider les utilisateurs de récupérer les données souhaitées de xeno-canto.org.

Cette api nous a permis de récupérer des enregistrements de tailles différentes en format mp3. Nous avons donc utilisé ffmpeg pour découper ses derniers en segments de 10 secondes et les convertir en wav afin qu'on puisse extraire les données avec la librairie python wave.

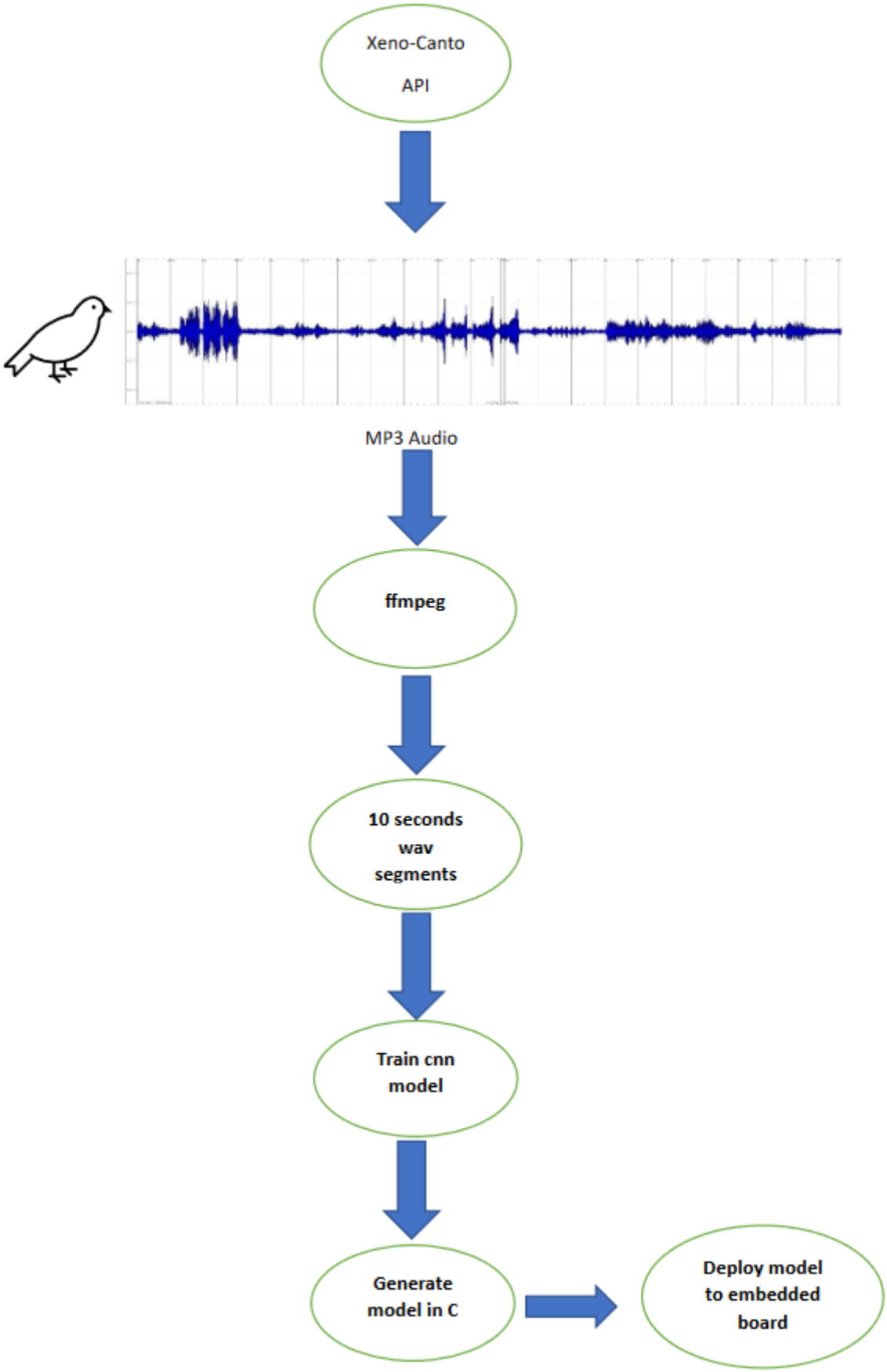
Une fois que nos données sont prêts, on fait en sort que toutes les classes ont le même nombre d'enregistrement et on met aléatoirement les noms de 10% des enregistrements de chaque classe dans le fichier testing_list.txt et 10% dans validatio_list.txt. on aura donc un modèle d'apprentissage de 90% train 10% test 10% validation.

On charge après les données des enregistrements dans des tableaux train et test. est on entraine notre model cnn inspiré du modèle M5 et modifié pour qu'il entre dans la carte.

On génère après un code c pour le modèle avec des points fixes représentés sur 16 bits, On le compile et évalue une petite base de données afin de vérifiés si les résultats obtenus avec le modèle généré en c sont les même.

En fin, on déploie notre modèle sur la carte en lançant le code .ino et on teste notre ia embarquée en jouant des audios et en observant les résultats sur le "Moniteur série" d'arduino.

Schéma illustratif du workflow


Xeno-Canto
API
MP3 Audio
ffmpeg
10 seconds
wav
segments
Train cnn
model
Generate
model in C
Deploy model
to embedded
board

Modèle CNN

```
model = Sequential()

model.add(Input(shape=(10000, 2)))
model.add(Conv1D(filters=8, kernel_size=30, activation='relu',strides=10))
model.add(MaxPool1D(pool_size=10))
model.add(Conv1D(filters=8, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=4))
model.add(Conv1D(filters=16, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=4))
model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=1))
model.add(AvgPool1D())
model.add(Flatten())
model.add(Dense(units=3))
model.add(Activation('softmax')) # SoftMax activation needs to be separate from Dense to remove it later on
# EXPLORE Learning Rate
opt = tf.keras.optimizers.Adam(learning_rate=10e-3)
model.summary()
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['categorical_accuracy'])
```

Afin de pouvoir déployer notre modèle sur la carte, il était crucial de choisir un modèle de réseau de neurones convolutifs (CNN) qui n'utilise pas trop de ressources, notamment en RAM et en ROM. En effet, les modèles de CNN peuvent être très lourds et complexes, ce qui les rend difficiles à intégrer sur des systèmes embarqués ayant des contraintes de ressources. Nous avons donc pris soin de sélectionner un modèle adapté à ces contraintes afin de pouvoir obtenir des performances raisonnables tout en restant dans les limites de la carte.

Nous avons pris le modèle M5 vu en lab5, et nous avons changé les paramètres de nos données pour que l'input soit de dimension (10000,2). Donc nos données ont été redimensionnées à 10khz et 2 secondes.

Pour les couches de convolution, nous avons diminué les tailles des kernels et les nombres de filtres, et nous avons augmenté le nombre de strides pour la premiere couche de convolution

conv layer	filters	kernel size	strides
first	8	30	10
second	8	3	default
third	8	3	default
fourth	32	3	default

Architecture du processing des données par le capteur

On récupere les données de l'audio avec le micro du capteur et on les stocke dans un tableau inputs. on passe ce tableau dans la fonction "cnn" qui est le modèle que nous avons entrainé. Cette fonction nous retourne un tableau outputs qui contient les probabilités que l'audio appartient à chaque classe. On prend la classe qui a la plus grande probabilité et on l'affiche dans le moniteur série.

DMA

Le controlleur DMA est un périphérique qui permet de transférer des données entre la mémoire et le microcontrolleur avec une utilisation minimale du CPU. le DMA est répartie en canaux, chaque canal peut être

configuré pour transférer des données entre une source et une destination spécifique. Dans notre cas, nous avons utilisé le canal 1 pour transférer les données du micro vers la mémoire.

ADC

Le convertisseur analogique-numérique (ADC) est un périphérique qui permet de convertir un signal analogique en un signal numérique. Dans notre cas, nous avons utilisé l'ADC pour convertir le signal analogique du micro en un signal numérique.

Résultats obtenus

Dans un premier temps, Nous avons utilisé seulement les premiers 10 secondes de chaque enregistrement pour l'apprentissage et le test. Cette stratégie nous faisait perdre beaucoup d'informations et nous donnait une accuracy de 44%.

Cependant, après avoir utilisé le logiciel ffmpeg pour découper chaque audio en segments de 10 secondes et en utilisant la totalité, nous avons observé une amélioration significative de l'accuracy, qui est passée à 60%. Cette modification a permis d'inclure une quantité de données plus importante dans l'apprentissage du modèle, ce qui a contribué à améliorer ses performances.

En faisant un premier essaie avec toutes les classe, Nous avons remarqué que le model confondait le plus le bruant jaune avec le bruant zizi.

En écoutant quelques enregistrements, on peut remarquer qu'ils ont un chant trés similaire

```
19/19 - 0s - loss: 0.9575 - categorical_accuracy: 0.5918 - 92ms/epoch - 5ms/step
19/19 [==============================] - 0s 5ms/step
tf.Tensor(
[[ 84  18  13  32]
 [ 17 103  13  14]
 [ 15  18  87  27]
 [ 20  20  33  74]], shape=(4, 4), dtype=int32)
```

Nous avant donc fait un autre essaie sans la classe du bruant jaune et l'accuracy est monté à 70%.

```
14/14 - 0s - loss: 0.6614 - categorical_accuracy: 0.7098 - 72ms/epoch - 5ms/step
14/14 [==============================] - 0s 4ms/step
tf.Tensor(
[[103  28  16]
 [ 18 115  14]
 [ 25  27  95]], shape=(3, 3), dtype=int32)
```

En effet, un des difficultés de faire apprendre une ia à reconnaitre les chants des oiseaux est la similarité de chants de plusieurs espèces.

Une autre difficulté est le fait que les audios sont souvent "pollués" par du bruit de font ou des chants d'autres espèces. Il y a aussi le fait que les oiseaux ont souvent plusieurs types de chants différents (le "si yut-tee yut-tee" joyeux, le "te tuuiii" gazouillant, le "yun-yun-yun" alarmant, etc).

Le dernier problème que nous avons rencontré est le déséquilibre du nombre d'enregistrements entre les espèces. Cela est du au fait que certaines sont plus populaires que d'autres, et certaines sont plus rare que d'autre. Pour donner un exemple, nous avons décidé de ne pas inclure le Faucon crécerelle dans notre base de données car nous avons trouvé seulement une soixantaine d'enregistrements de chants pour cette espèce.

Le meilleur résultat que nous avons obtenu est une accuracy de 80% pour le modèle sans la classe du bruant jaune en ayant un input de (10000, 10) qui concervait le plus d'informations.

```python
model = Sequential()

model.add(Input(shape=(10000, 10)))
model.add(Conv1D(filters=8, kernel_size=30, activation='relu',strides=20))
model.add(MaxPool1D(pool_size=10))
model.add(Conv1D(filters=8, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=3))
model.add(Conv1D(filters=16, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=3))
model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=1))
model.add(AvgPool1D())
model.add(Flatten())
model.add(Dense(units=3))
model.add(Activation('softmax')) # SoftMax activation needs to be separate from Dense to remove it later on
# EXPLORE Learning Rate
opt = tf.keras.optimizers.Adam(learning_rate=10e-3)
model.summary()
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['categorical_accuracy'])
```

✓ 0.1s Python

Mais malheureusement même en diminuant le nombre de filtres et la taille des kernels et en augmentant le nombre des strides, le modèle restait trop lourd pour être déployé sur la carte.

Nous sommes conscients que ces résultats ne sont pas encore à la hauteur de nos attentes. Toutefois, compte tenu du nombre limité de filtres que nous avons utilisés dans les couches de convolution et la baisse de la frequence et des secondes dans l'entrée pour pouvoir déployer le modèle sur la carte, cette performance reste tout à fait satisfaisante.

Analyse des performances pour le modèle avec toutes les classes

	unit of measurment	**value**
ROM Memory	octets	46088
Mem Rom	octets	46094
Calculated latency	ms	100

Estimation de la consomatation d'énergie

Consomation en mode active : 62 mW

On suppoose que la carte recupere les données pendant tous les 2 secondes et on les envoie au serveur pour les traiter.

Donc la consomation moyenne par periode : L*P/T = 62*0.1/2.= 3.1 mW

Energie de la batterie : 3700 mWh

Temps de vie de la batterie avec le sleep mode : 3700/3.1 = 1193.55 heures = 49.73 jours

Temps de vie de la batterie sans le sleep mode : 3700/62 = 59.67 heures = 2.48 jours

Temps de vie de la batterie en cas se communication sans fil : si on suppose que la carte consomme 100 mW tous les 50 ms pendant la connectio, la consomation moyenne par periode : L*P/T = 100*0.05 + 62*0.1/2 = 5.6 mW.

Donc le temps de vie de la batterie : 3700/5.1 = 660.71 heures = 27.53 jours.

Conclusion

Pour conclure, notre projet a été une réussite et nous a permis d'acquérir de nouvelles compétences en traitement de données. Toutefois, pour améliorer encore davantage la précision de notre modèle d'identification des chants d'oiseaux, il serait intéressant de mettre en place un mécanisme de seuil de détection de niveau sonore. Ce dernier permettrait de détecter les parties du signal audio inutiles, telles que les silences entre les enregistrements ou les bruits de fond, et de les supprimer automatiquement. En éliminant ces parties superflues du signal, nous pourrions optimiser notre modèle et ainsi améliorer l'identification des chants d'oiseaux.

En 2020, une équipe polonaise avait réussit à implémenter une solution pour ce problème qui avait une accuracy de 87%. Il ont utilsé **MFCC** (Mel-frequency cepstrum) pour filtrer les audios et **EfficientNetB3** pour implémenter leurs modèle. Il sera intérresant d'explorer cette solution est voir s'il est possible de l'implémenter dans une carte embarquée du type Nucleo-L476RG.