RAPPORT TECHNIQUE

Système de classification de commandes vocales pour drone

Basé sur le corpus VoiceStick (Henry et al., 2025)

Commanditaire : Solange ROSSATO

Nikita DUZHENKO | Aggnia MARINA
Master 2 Industrie de la Langue | Traitement Automatique du Langage Écrit et Parlé

Table de matières

1. Contexte

1.1 Cadre du projet

Le projet VoiceStick s'inscrit dans le cadre d'un projet académique de Master 2 Sciences du langage, parcours Industrie de la langue, à l'Université Grenoble Alpes. Il porte sur le développement d'un système de classification de commandes vocales pour drone, base sur le corpus VoiceStick.

Le corpus VoiceStick a été collecté lors d'une expérience de guidage vocal en 3D impliquant 20 étudiants en psychologie (âge moyen : 22 ans) organisés en 10 binômes. Chaque binôme comprenait un « guide » qui donne des instructions vocales et un « pilote » qui manipule un marqueur en salle de capture de mouvement. La communication était unidirectionnelle : le guide donnait des instructions vocales tandis que le pilote ne pouvait pas répondre. La tâche consistait à guider le pilote vers une cible 3D virtuelle affichée sur un écran. Chaque binôme devait atteindre 12 cibles au total reparties en 2 blocs de 6 cibles, avec un timeout de 120 secondes par cible.

1.2 Données disponibles

Le corpus contient 423 enregistrements audio au total, dont 195 ont été vérifiés manuellement. Les fichiers audio sont au format WAV à 48 kHz. Les annotations sont fournies sous forme de fichiers TextGrid (format Praat) contenant les transcriptions, un découpage par mots et phonèmes, et la distance a la cible. Des fichiers TextGrid _commands documentent les impulsions joystick synchronisées.

Pour notre travail, nous utilisons 347 paires TextGrid/WAV correspondant à environ 58 participants uniques (chacun avec ~6 tentatives). Les annotations de commandes sont stockées dans un tier « commands » des fichiers TextGrid.

1.3 Objectif

L'objectif est de développer un système de classification audio-to-command qui prend en entrée un enregistrement audio segmenté et produit en sortie une commande de drone parmi neuf classes possibles, sans passer par une étape de transcription textuelle intermédiaire. Les neuf classes sont : « forward », « backward », « left », « right », « up », « down », « yawleft », « yawright » et « none » (regroupant les énoncés non directifs ou contextuels comme « encore », « ok », « vas-y »).

1.4 Défis

Ce projet présente plusieurs défis que nous allons détailler. Tout d'abord, la classe « none » représente environ 57 % du jeu d'entrainement (4265 sur 7539 échantillons), tandis que certaines classes directionnelles ont un support très faible (« backward » : 152 échantillons). Ce déséquilibre peut biaiser le modèle vers la prédiction de la classe majoritaire.

Un autre gros défi est celui de l'ambiguïté gauche/droite vs rotation/translation. Lorsqu'un locuteur dit « à gauche », il peut signifier une translation latérale ou une rotation. La résolution de cette ambiguïté nécessite la synchronisation avec les impulsions joystick du fichier _commands.TextGrid.

La diversité lexicale représente également une difficulté : 42 % des énoncés sont uniques dans le corpus, reflétant la spontanéité du langage naturel. Cette diversité rend la généralisation plus difficile.

Nous avons également noté que les commandes joystick sont réalisées par impulsions brèves (200-300 ms). Les segments audio correspondants sont très courts, ce qui pose des défis pour l'extraction de représentations robustes avec wav2vec2. Nous pouvons ajouter à cela le nombre limité de locuteurs qui réduit la capacité du modèle à généraliser à de nouvelles voix.

2. Méthodologie

2.1 Annotation des données

Les annotations sont réalisées dans des fichiers TextGrid contenant un tier « commands ». Chaque intervalle de ce tier associe un segment temporel (début, fin) à une des 9 commandes possibles. Le parser (classe SimpleCommandParser) gère automatiquement la détection d'encodage UTF-8 / UTF-16 avec BOM.

La résolution de l'ambiguïté translation/rotation est effectuée par synchronisation entre le fichier TextGrid principal (transcriptions vocales) et le fichier _commands.TextGrid (impulsions joystick enregistrées). Cette synchronisation permet de déterminer si « à gauche » correspond à « left » (translation) ou « yawleft » (rotation) en se référant à l'action réellement exécutée par le pilote.

Le pipeline de préparation (prepare_data.py) exécute 5 étapes séquentielles : (1) parsing des TextGrid vers un dataset.csv, (2) séparation des participants en ensembles train/test (85/15), (3) segmentation audio des segments d'entrainement en clips WAV 16 kHz, (4) extraction des embeddings wav2vec2 pour l'entrainement, (5) copie des fichiers audio bruts pour les participants de test.

2.2 Extraction des embeddings (wav2vec2)

Les représentations acoustiques sont extraites à l'aide du modèle LeBenchmark/wav2vec2-FR-7K-large, un modèle wav2vec2 pré-entrainé sur 7000 heures de français parlé. Ce modèle est utilisé en mode gelé (frozen features) : aucun fine-tuning n'est effectué, seules les représentations de la dernière couche cachée (couche 12) sont extraites.

Pour chaque segment audio, le signal est d'abord rééchantillonné de 48 kHz à 16 kHz (fréquence attendue par wav2vec2). Le modèle produit une séquence de vecteurs (un par trame temporelle d'environ 20 ms), qui est ensuite réduite à un vecteur unique de dimension 1024 par mean pooling (moyenne temporelle sur toute la séquence). Ce vecteur constitue l'embedding du segment.

Avec environ 7500 échantillons d'entrainement et 9 classes, le jeu de données est trop petit pour fine-tuner un modèle de 300M+ paramètres sans risque de surapprentissage sévère. L'approche frozen features permet de bénéficier des représentations riches apprises par le modèle sur 7000 h de français tout en évitant l'overfitting. De plus, cette approche est beaucoup plus rapide et ne nécessite pas de GPU pour l'entrainement des classifieurs en aval.

2.3 Séparation des données et validation croisée GroupKFold

Les données sont d'abord séparées en un ensemble d'entrainement (85 %) et un ensemble de test (15 %) au niveau des participants (et non des segments individuels). Cette séparation par locuteur garantit une indépendance totale : aucun segment d'un locuteur de test n'apparait dans l'entrainement. Dans notre cas : 49 participants en entrainement (7539 segments), 9 participants en test (1408 segments).

Sur l'ensemble d'entrainement, une validation croisée à 5 plis (5-fold) est appliquée en utilisant GroupKFold de Scikit-Learn, avec le participant_id comme variable de groupement. Cela signifie que dans chaque fold, tous les segments d'un même locuteur sont soit dans l'ensemble d'entrainement, soit dans l'ensemble de validation — jamais dans les deux. Cette stratégie simule les conditions réelles d'utilisation où le modèle rencontrera de nouveaux locuteurs inconnus.

À chaque fold, les 49 participants sont repartis en ~39-40 pour l'entrainement et ~9-10 pour la validation. Après la validation croisée, un modèle final est entrainé sur l'ensemble des 49 participants d'entrainement et évalué sur les 9 participants de test.

Il est important de noter qu'avant chaque entrainement (par fold et pour le modèle final), la classe « none » est sous-échantillonnée pour limiter sa surreprésentation. Le ratio est contrôlé par le paramètre none_ratio : le nombre d'échantillons « none » est plafonné à none_ratio × taille de la plus grande classe non-none. Avec none_ratio = 1.5, on passe typiquement de ~6000 a ~4000-5000 échantillons d'entrainement par fold.

2.4 SVM — Support Vector Machine

Le SVM (Support Vector Machine) est un algorithme de classification qui cherche l'hyperplan optimal séparant les classes dans un espace de haute dimension. Avec un noyau RBF (Radial Basis Function), le SVM projette implicitement les données dans un espace de dimension infinie ou les classes deviennent linéairement séparables. Voici à quoi ressemble la classe SVC (C-Support Vector Classification) de Scikit-Learn que nous avons utilisé pour l'entrainement du modèle SVM :

```
class sklearn.svm.SVC (
  *, C=1.0 kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True,
  probability=False, tol=0.001, cache_size=200, class_weight=None,
  verbose=False, max_iter=-1, decision_function_shape='ovr',
  break_ties=False, random_state=None)[1]
```

Pour notre objectif, nous avons choisi les hyperparamètres suivants :

```
svm = SVC (
  kernel='rbf', C=10, gamma='scale', class_weight='balanced', random_state=42)
```

Le noyau gaussien (RBF) est le choix standard pour des données de haute dimension non linéairement séparables. Il mesure la similarité entre deux points via une fonction gaussienne, ce

[1] *SVC*. (s. d.). Scikit-Learn. Consulté 19 février 2026, à l'adresse https://scikit-learn/stable/modules/generated/sklearn.svm.SVC.html

qui est adapté aux embeddings wav2vec2 de dimension 1024 ou les frontières de décision entre commandes ne sont pas linéaires.

Le paramètre de régularisation C contrôle le compromis entre maximiser la marge et minimiser les erreurs de classification. Une valeur de 10 (supérieure au défaut de 1) permet au modèle d'être plus strict sur les erreurs d'entrainement, ce qui est approprié car nos embeddings wav2vec2 sont des features de haute qualité avec un bon rapport signal/bruit. Une valeur C = 10 a été validée empiriquement par GridSearchCV (recherche sur C = {0.1, 1, 10, 100} et gamma = {'scale', 'auto', 0.001, 0.01}) qui a confirmé C = 10 / gamma = 'scale' comme la combinaison optimale.

Le gamma contrôle la portée de l'influence de chaque point d'entrainement. La valeur 'scale' calcule gamma = 1 / (n_features × var(X)), ce qui adapte automatiquement la largeur du noyau à la dimension et à la variance des données. C'est le choix recommandé pour des features normalisées de haute dimension.

Le poids de classe (class_weight) pondère automatiquement chaque classe inversement proportionnellement à sa fréquence. Cela compense le déséquilibre résiduel après le sous-échantillonnage de « none », en pénalisant davantage les erreurs sur les classes minoritaires comme « backward » ou « yawleft ».

Les features sont centrées et réduites (moyenne = 0, variance = 1) avant entrainement. Le SVM avec noyau RBF est sensible à l'échelle des features car le noyau calcule des distances euclidiennes. Sans normalisation, les dimensions à grande variance domineraient le calcul de similarité. De plus, une graine aléatoire a été fixée pour assurer la reproductibilité de l'initialisation des poids et du mélange des données.

2.5 MLP — Multi-Layer Perceptron

Le MLP (Multi-Layer Perceptron) est un réseau de neurones à propagation avant composé de couches entièrement connectées. Contrairement au SVM qui définit des frontières de décision basées sur les vecteurs de support, le MLP apprend des représentations intermédiaires via ses couches cachées, ce qui lui confère potentiellement une plus grande flexibilité pour capturer des motifs complexes dans les données. Voici à quoi ressemble la classe MLPClassifier de Scikit-Learn que nous avons utilisé pour l'entrainement du modèle MLP :

```
class sklearn.neural_network.MLPClassifier (
  hidden_layer_sizes=(100,), activation='relu', *, solver='adam',
  alpha=0.0001, batch_size='auto', learning_rate='constant',
  learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True,
  random_state=None, tol=0.0001, verbose=False, warm_start=False,
  momentum=0.9, nesterovs_momentum=True, early_stopping=False,
  validation_fraction=0.1, beta_1=0.9, beta_2=0.999,
  epsilon=1e-08, n_iter_no_change=10, max_fun=15000)[2]
```

[2] *MLPClassifier*. (s. d.). Scikit-Learn. Consulté 19 janvier 2026, à l'adresse https://scikit-learn/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Pour notre objectif, nous avons choisi les hyperparamètres suivants :

```
mlp = MLPClassifier(
   hidden_layer_sizes=(512, 256), activation='relu', solver='adam',
   alpha=1e-4, batch_size=64, learning_rate_init=1e-3, max_iter=200,
   early_stopping=True, n_iter_no_change=10, random_state=42, verbose=False)
```

Nous avons opté pour deux couches cachées avec 512 et 256 neurones respectivement. La première couche réduit la dimension de 1024 à 512 (facteur 2), puis à 256. Cette architecture en « entonnoir » force le réseau à apprendre des représentations de plus en plus abstraites. Deux couches suffisent pour notre problème — ajouter des couches supplémentaires augmenterait le risque de surapprentissage avec ~5000 échantillons d'entrainement par fold.

La fonction d'activation ReLU (Rectified Linear Unit) est le standard pour les couches cachées des réseaux modernes. Elle est simple à calculer, évite le problème de la disparition des gradients (vanishing gradients) inhérent aux fonctions sigmoide/tanh, et introduit la non-linéarité nécessaire pour apprendre des frontières de décision complexes.

Adam (Adaptive Moment Estimation) est un optimiseur à taux d'apprentissage adaptatif qui combine les avantages de RMSprop et du momentum. Il est robuste aux hyperparamètres et converge rapidement, ce qui en fait le choix standard pour les MLP.

Le paramètre de régularisation L2, alpha, pénalise les poids trop grands. Une valeur de 0.0001 offre une régularisation légère, suffisante pour prévenir le surapprentissage sans restreindre excessivement la capacité du modèle.

Les mini-lots de 64 échantillons offrent un bon compromis entre stabilité du gradient (vs. batch_size = 1) et vitesse de convergence (vs batch complet). Avec ~4000-5000 échantillons par fold, cela donne ~62-78 itérations par époque.

Le taux d'apprentissage initial de 0.001 est la valeur standard pour Adam. Un taux trop élevé causerait des oscillations, trop faible ralentirait la convergence.

Le nombre maximum d'époques n'a pas été modifié, car, en pratique, l'early stopping arrête l'entrainement bien avant si la performance sur un sous-ensemble de validation interne ne s'améliore pas pendant 10 époques consécutives. Cela prévient le surapprentissage et réduit le temps de calcul. Enfin, la même graine aléatoire, que pour le modèle SVM, a été utilisé.

2.6 Métriques d'évaluation

Le choix des métriques est crucial pour un problème multi-classe déséquilibré comme le nôtre. L'exactitude seule serait trompeuse : un modèle qui prédit toujours « none » obtiendrait ~60 % d'exactitude sur le jeu de test sans avoir appris aucune commande utile. Pour compléter cela, nous avons intégré plusieurs autres métriques que nous allons détailler ci-dessous.

Tout d'abord, la précision, qui est une proportion de prédictions correctes parmi toutes les prédictions d'une classe. Dans notre contexte, une précision faible signifie que le drone recevrait souvent une commande erronée — par exemple, exécuter « right » alors que le pilote a dit autre

chose. C'est un enjeu de sécurité directe. Puis, classiquement, nous avons le rappel, la proportion d'échantillons réellement positifs correctement identifiés. Un rappel faible signifie que le drone ignore des commandes du pilote — il ne réagît pas quand il le devrait. Pour un drone en vol, ne pas détecter un « backward » d'urgence peut être critique.

Dans un second temps, nous avons calculé la moyenne harmonique de la précision et du rappel : F1 = 2 x P x R / (P + R). Le F1 est notre métrique principale par classe car il pénalise sévèrement les déséquilibres entre précision et rappel — un modèle doit être bon sur les deux aspects pour obtenir un F1 élevé. La métrique principale globale a été la moyenne arithmétique des F1 de chaque classe, sans pondération par le support. Chaque commande compte autant, ce qui est essentiel : on veut que « backward » (152 échantillons) soit aussi bien reconnu que « none » (4265 échantillons). L'objectif était un F1-macro d'environ 75 %. De plus, nous avons utilisé une métrique complémentaire, la moyenne des F1 pondérée par le support de chaque classe, qui reflète la performance globale ressentie par l'utilisateur (les classes fréquentes pèsent plus), mais peut masquer les faiblesses sur les classes rares.

Nous avons produit deux matrices de confusion (*Figure 1*) montrant pour chaque classe réelle combien d'échantillons ont été prédits dans chaque classe. Indispensable pour identifier les paires de classes confondues (ex. « left » vs « yawleft ») et guider les améliorations futures.

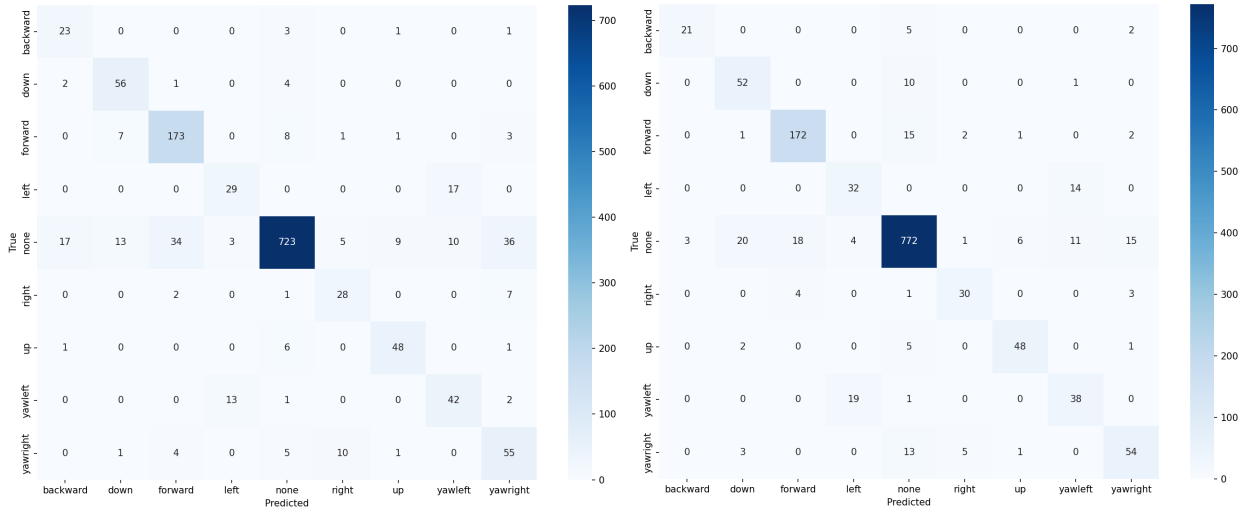

Figure 1. Matrices de confusion du jeu test : (haut) modèle SVM, (bas) modèle MLP.

3. Résultats

3.1 Vue d'ensemble

Les deux modèles sont évalués sur un jeu de test indépendant de 9 locuteurs (1408 segments) jamais vus pendant l'entrainement ni la validation croisée (*Table 1*).

Le MLP surpasse le SVM sur toutes les métriques : +3.0 points d'exactitude, +3.9 points de F1-macro, et +2.5 points de F1-weighted. L'objectif de ~75 % de F1-macro est atteint par le MLP (0.774) et approche par le SVM (0.735).

Métrique	SVM	MLP
Exactitude	0.836	0.866
F1-macro	0.735	0.774
F1-weighted	0.843	0.868

Table 1. Résultats des performances de modèles.

3.2 Performances par classe sur le jeu de test

La *Table 2* présente en détail les métriques de performances des deux modèles testés. Les deux reconnaissent correctement les classes majoritaires « none » et « forward ». Les commandes verticales « up » et « down » présentent des performances cohérentes. Les principales difficultés concernent toujours les commandes latérales et de rotation, en raison de l'approche individuelle à la réalisation de tâche par chaque pilote et guide. Toutefois, le MLP améliore nettement certaines classes minoritaires, notamment « backward » et « yawright », ce qui contribue à l'augmentation du F1-macro global.

3.3 Validation croisée

La validation croisée 5-fold GroupKFold fournit une estimation de la robustesse des modèles face à différents sous-ensembles de locuteurs.

Le SVM présente une validation croisée (CV) stable. Il obtient un F1-macro moyen de 0.759 (+/- 0.021) sur les 5 folds, avec des valeurs individuelles entre 0.725 et 0.786. L'écart-type très faible (0.021) indique une excellente stabilité : le modèle généralise de manière consistante indépendamment de la composition des locuteurs dans chaque fold.

Classe	Support	Préc. SVM	Rap. SVM	F1 SVM	Préc. MLP	Rap. MLP	F1 MLP
backward	28	0.53	0.82	0.65	0.88	0.75	0.81
down	63	0.73	0.89	0.80	0.67	0.83	0.74
forward	193	0.81	0.90	0.85	0.89	0.89	0.89
left	46	0.64	0.63	0.64	0.58	0.70	0.63
none	850	0.96	0.85	0.90	0.94	0.91	0.92
right	38	0.64	0.74	0.68	0.79	0.79	0.79
up	56	0.80	0.86	0.83	0.86	0.86	0.86
yawleft	58	0.61	0.72	0.66	0.59	0.66	0.62
yawright	76	0.52	0.72	0.61	0.70	0.71	0.71

Table 2. Performances de modèles par classe sur le jeu de test.

Le MLP présente un problème majeur de stabilité en CV. Les deux premiers folds obtiennent des F1-macro corrects (0.709 et 0.650), mais les folds 3, 4 et 5 s'effondrent à des valeurs proches de 0.02 — le modèle ne fait quasiment aucune prédiction correcte (*Table 3*).

L'effondrement des folds 3-5 du MLP s'explique par plusieurs facteurs convergents :

1. La sensibilité du GroupKFold à la composition des folds. Les 49 participants sont répartis en 5 groupes. Or certaines combinaisons de participants d'entrainement peuvent créer une distribution déséquilibrée ou atypique qui empêche le MLP de converger. Le SVM, résolvant un problème d'optimisation convexe, n'est pas affecté par ce phénomène.
2. Contrairement au SVM qui possède une solution unique garantie, le MLP utilise la descente de gradient stochastique (Adam). L'initialisation aléatoire des poids combinée à certaines distributions de données peut conduire à la disparition des gradients (vanishing gradients), empêchant toute mise à jour utile du réseau.
3. Malgre l'utilisation de early_stopping et d'une graine fixe (random_state = 42), le MLP de Scikit-Learn ne dispose pas de learning rate scheduling (réduction progressive du taux d'apprentissage) ni de gradient clipping, deux techniques qui auraient pu stabiliser la convergence sur les folds problématiques.

Malgré cette instabilité en CV, le modèle final MLP (entraine sur tous les 49 participants) obtient les meilleures performances sur le jeu de test, ce qui confirme que le problème est lié à la convergence sur certains sous-ensembles et non à la capacité intrinsèque de l'architecture.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
SVM	0.773	0.725	0.786	0.747	0.765
MLP	0.709	0.650	0.020	0.024	0.024

Table 3. Les valeurs de F1-macro lors de la CV par fold par modèle.

4. Discussion

Les résultats révèlent un compromis classique en apprentissage automatique. Le SVM offre une stabilité remarquable (écart-type CV de 0.021) grâce à son optimisation convexe, mais sa capacité de représentation est limitée par le choix du noyau. Le MLP, plus flexible avec ses deux couches cachées (512, 256 neurones), capture des motifs plus subtils dans les embeddings — d'où ses meilleures performances sur le test (+3.9 points de F1-macro) — mais au prix d'une sensibilité à l'initialisation et à la composition des données d'entrainement.

Malgré le sous-échantillonnage de « none » et la pondération des classes, le déséquilibre reste un facteur limitant. En test, « backward » (28 échantillons), « right » (38) et « left » (46) ont des supports très faibles, rendant leurs métriques statistiquement volatiles. Un seul échantillon mal classé peut modifier le F1 de « backward » de +/- 0.03. Les performances sur ces classes doivent être interprétées avec prudence.

Les principales erreurs de classification concernent les paires « left »/« yawleft » et « right »/« yawright ». Ces confusions sont phonétiquement prévisibles : les commandes de lacet (yaw) partagent la même composante directionnelle que les translations simples. Le mean pooling sur l'ensemble du segment peut perdre l'information séquentielle qui distingue les deux types de commande.

L'utilisation de wav2vec2 en mode gelé impose un plafond de performance. Le modèle pré-entrainé a appris des représentations générales du français parlé, mais n'a pas été optimisé pour distinguer des commandes courtes et similaires dans un contexte de pilotage. Un fine-tuning, même partiel (par exemple sur les dernières couches du transformer), pourrait améliorer la séparation entre classes proches. Cependant, avec seulement ~7500 échantillons, le risque de surapprentissage est réel.

5. Perspectives

En ce qui concerne le modèle MLP, il pourrait être envisageable d'implémenter un learning rate scheduling (réduction du taux d'apprentissage à chaque plateau), du gradient clipping, et expérimenter avec différentes architectures (3 couches, batch normalization, dropout plus agressif). L'objectif est d'obtenir une CV stable comparable à celle du SVM tout en conservant les performances supérieures du MLP sur le test.

Un autre point qui a du potentiel dans l'amélioration de la généralisation est l'ajout de nouveaux locuteurs (au-delà des 20 actuels), ce qui donnerait plus de stabilité à la CV. Les classes à faible support (« backward », « right », « left ») bénéficieraient particulièrement de données supplémentaires. L'augmentation de données (variation de vitesse, pitch shifting, ajout de bruit) pourrait être explorée comme alternative moins couteuse.

De plus, un fine-tuning des dernières couches du modèle wav2vec2 sur le corpus VoiceStick pourrait améliorer la séparation entre classes phonétiquement proches (left/yawleft, right/yawright). Cela nécessite cependant une stratégie anti-surapprentissage rigoureuse (faible taux d'apprentissage, régularisation forte, augmentation de données).

Il serait également intéressant de remplacer le mean pooling par une approche préservant l'information temporelle (par exemple, un classifieur LSTM ou un attention pooling sur les trames wav2vec2) pourrait aider à distinguer « left » de « yawleft » en capturant le préfixe discriminant. Cela est particulièrement pertinent pour les segments courts (200-300 ms) ou chaque trame compte.

Le système actuel fonctionne en mode asynchrone (analyse d'un enregistrement complet). Pour une utilisation réelle en vol, il faudrait développer un pipeline streaming capable de détecter et classifier les commandes en temps réel, avec une latence inférieure à 500 ms, cela implique une segmentation VAD (Voice Activity Detection) en continu et une inférence optimisée.

Enfin, la classe « none » regroupe actuellement des énoncés très divers : retours verbaux (« ok », « c'est bien ») et commandes contextuelles (« encore », « continue »). Une sous-classification de « none » permettrait d'extraire les commandes contextuelles et de les résoudre en utilisant l'historique des commandes précédentes (ex. « encore » après « forward » = « forward »).