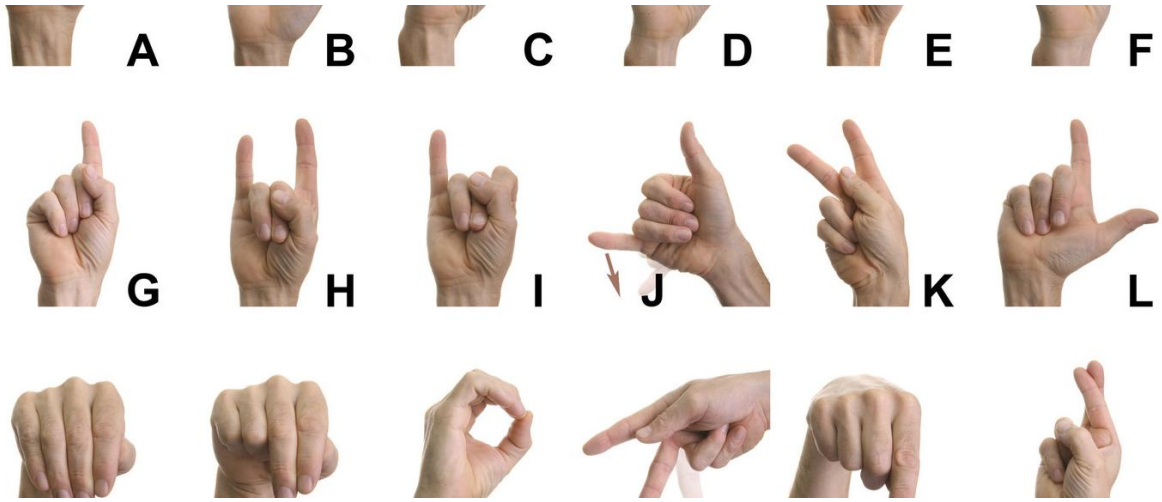
Reconnaissance de la langue des signes

Céline, Jamal, Jérémy



Contexte du projet

Beaucoup de progrès et de recherches en IA ont été faites pour aider les personnes sourdes et muettes. L'apprentissage profond et la vision par ordinateur peuvent également être utilisés pour avoir un impact sur cette cause.

Cela peut être très utile pour les personnes sourdes et muettes dans la communication avec les autres car la connaissance de la langue des signes n'est pas quelque chose qui est commun à tous, de plus, cela peut être étendu à la création des éditeurs automatiques, où la personne peut facilement écrire par ses simples gestes.

Modalités d'évaluation

- Un rapport sur le projet réalisé qui explique les différentes étapes du code
- Description des données
- Présentation de l'architecture utilisée
- Conclusion (avantages et inconvénients, concurrents, recommandations...)
- Revue de code avec le formateur.

Création du Dataset

Dans cette première partie, nous constituons un Dataset avec les différents signes de l'alphabet. Nous avons choisi de collecter des images traitées en amont via un "filtre" (Thresholded) qui permettra de différencier au mieux chaque signe effectué avec la main.

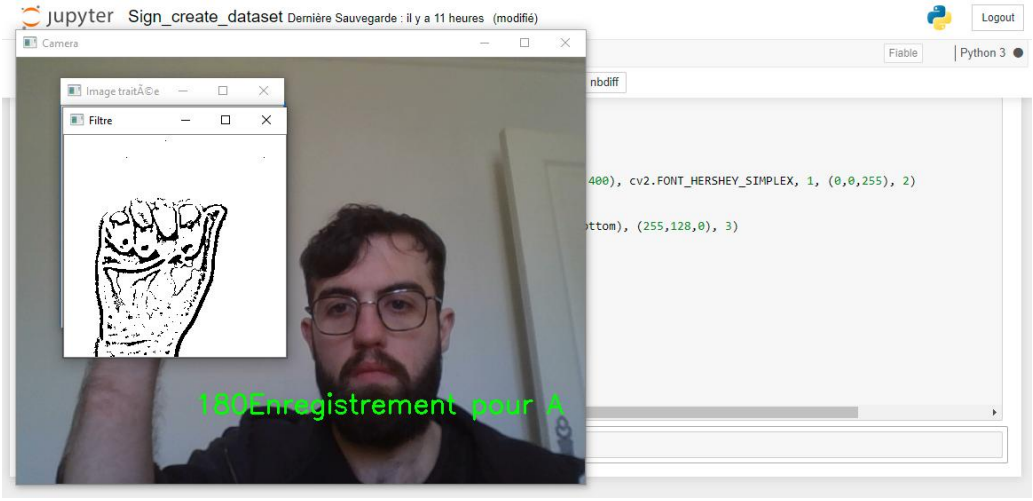
```
# Fonction pour l'application du filtre (Thresholded)
def segment_hand(frame, threshold=20):
    global background

    sdiff = cv2.absdiff(background.astype("uint8"), frame)
    thresholded = cv2.adaptiveThreshold(diff,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,11,2)

    return (thresholded)
```

Nous appliquons donc un seuil adapté à la portion de l'image qui recevra notre main avec deux paramètres à partir de la fonction *segment_hand()* :

- cv2.ADAPTIVE_THRESH_GAUSSIAN_C : la valeur de seuil est la somme pondérée des valeurs de voisinage où les poids sont une fenêtre gaussienne.
- cv2.THRESH_BINARY : Fixe la couleur en 0 ou 255 en fonction du seuil attribué.



Durant cette acquisition de données, nous chargeons préalablement le background, qui ne sera pas pris en compte lorsque nous positionnerons la main. Une deuxième fenêtre apparait et nous pouvons placer notre main pendant les 299 premières frames. Une fois fait, 300 images (250x250 pixels) sont enregistrées sous format .jpg dans un dossier conçu à cet effet. L'opération sera reproduite pour chaque catégorie (chaque signe de l'alphabet).

Le Dataset constitué (près de 8100 images) sera compressé pour l'importer sur *Colab* afin d'entrainer notre modèle.

Création du modèle

Nous créons ensuite le modèle qui servira pour répondre à l'objectif. Nous avons donc travaillé sur *Colab* afin d'entrainer rapidement notre modèle sous Exécution GPU. Le dataset2.zip a été importé dans Colab et dézippé, contenant le dossier data_sign2.

Nous avons défini nos paths et créé une liste répertoriant les différentes classes de signes sous le nom de leur dossier (‘A’, ‘B’, ‘C’, ‘D’...). Une liste a été créée avec la fonction *create_data_list()* regroupant des sous listes qui contiennent elle-même une image (redimensionnée en 128x128 pixels) associée à l’index de leur label.

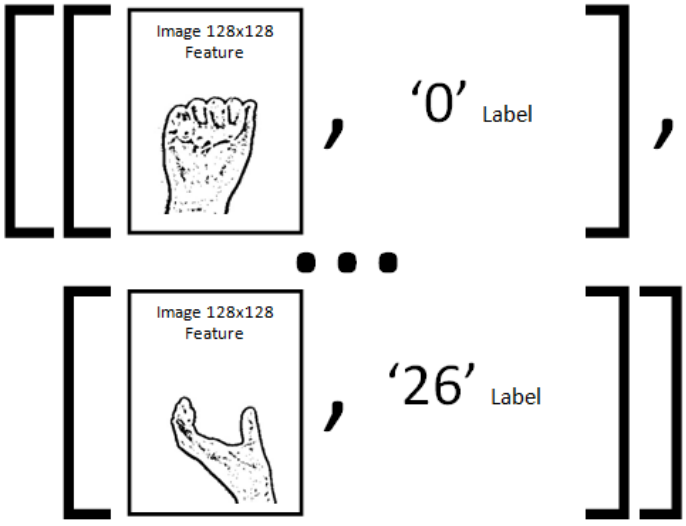


Figure 1. Schéma de la liste Dataset nouvelle crée

Après avoir randomisé les éléments de la liste, nous avons réparti les Features (images) et leur Label associé respectivement dans une liste x et y. Nous avons ensuite séparé les données prévues pour l'entraînement et l'évaluation avec la fonction *TrainTestSplit()* de *sklearn.model_selection* avec 20% de données initiales pour l'évaluation du modèle.

Les données ont été redimensionnées et normalisées. En parallèle les labels ont été encodés avec la fonction *keras.utils.to_categorical()* :

```
x_app.shape :     (6480, 128, 128, 3)
x_val.shape :     (1620, 128, 128, 3)
y_app.shape :     (6480,)              y_app.shape :     (6480, 27)
y_val.shape :     (1620,)              y_val.shape :     (1620, 27)
```

Construction du Réseau

Le modèle CNN a été créés de cette manière :

```
model = tf.keras.Sequential()

model.add(tf.keras.layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE, 3)))
model.add(tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
model.add(Dropout(0.8))

model.add(tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), activation='relu'))
model.add(Dropout(0.8))

model.add(tf.keras.layers.Flatten())
model.add(Dropout(0.8))

model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(27, activation='softmax'))

model.compile(optimizer ='adam',
            loss = 'categorical_crossentropy',
            metrics = ['accuracy'])
```

- Pour l'initialisation de notre modèle, les couches *Conv2D* sont utilisées sur les traitements d'objets bidimensionnelles, et contiennent différentes classes comme les filtres (dont le nombre est multiplié par 2 à la couche Conv2D suivante), avec une définition de *kernel* de dimension (3, 3) idéal pour les images ayant une taille inférieure ou égale à 128x128 pixels.

  L'input_shape reprend les dimensions des images que nous fournissons à notre modèle (128x128 pixels en 3 couleurs RGB). Enfin, nous avons définis une activation *reLU*, l'activation linéaire standard qui permet de fixer les valeurs négatives de nos matrices à 0.

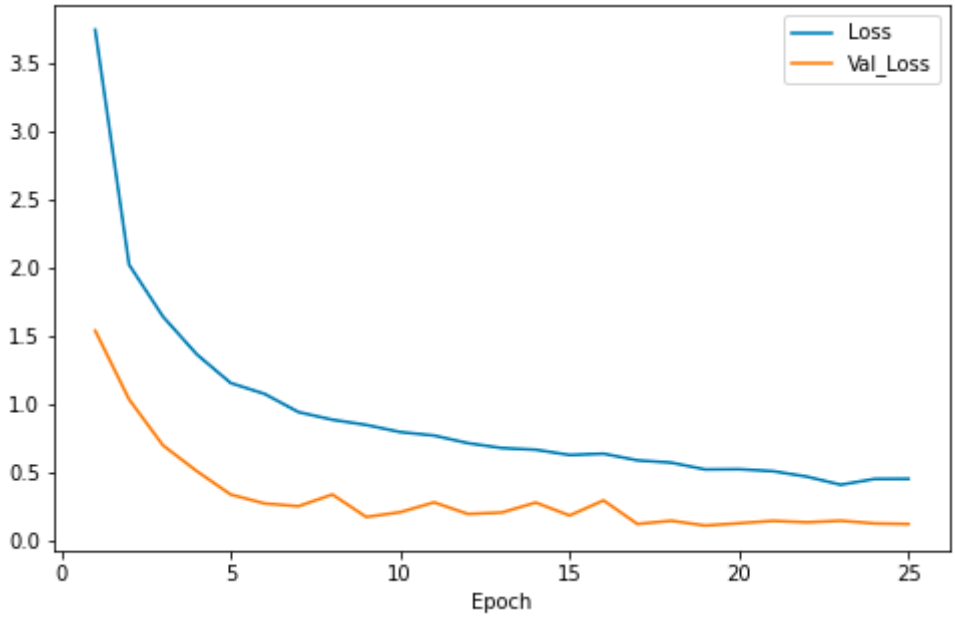
- Les dernières couches *Flatten* et *Dense* font la liaison entre les couches précédentes et convertissent les données en matrice à 1 dimension. Nous utilisons enfin l'activation *Softmax* et la fonction *categorical_crossentropy*, performant pour la classification multiple.

- Quelques essaies ont été réalisés, mais rencontrant un *overfitting*, nous avons décidé d'ajouter des Dropout, certes très élevés, mais qui ont permis d'obtenir un modèle présentant des résultats satisfaisants lors des tests vidéos.

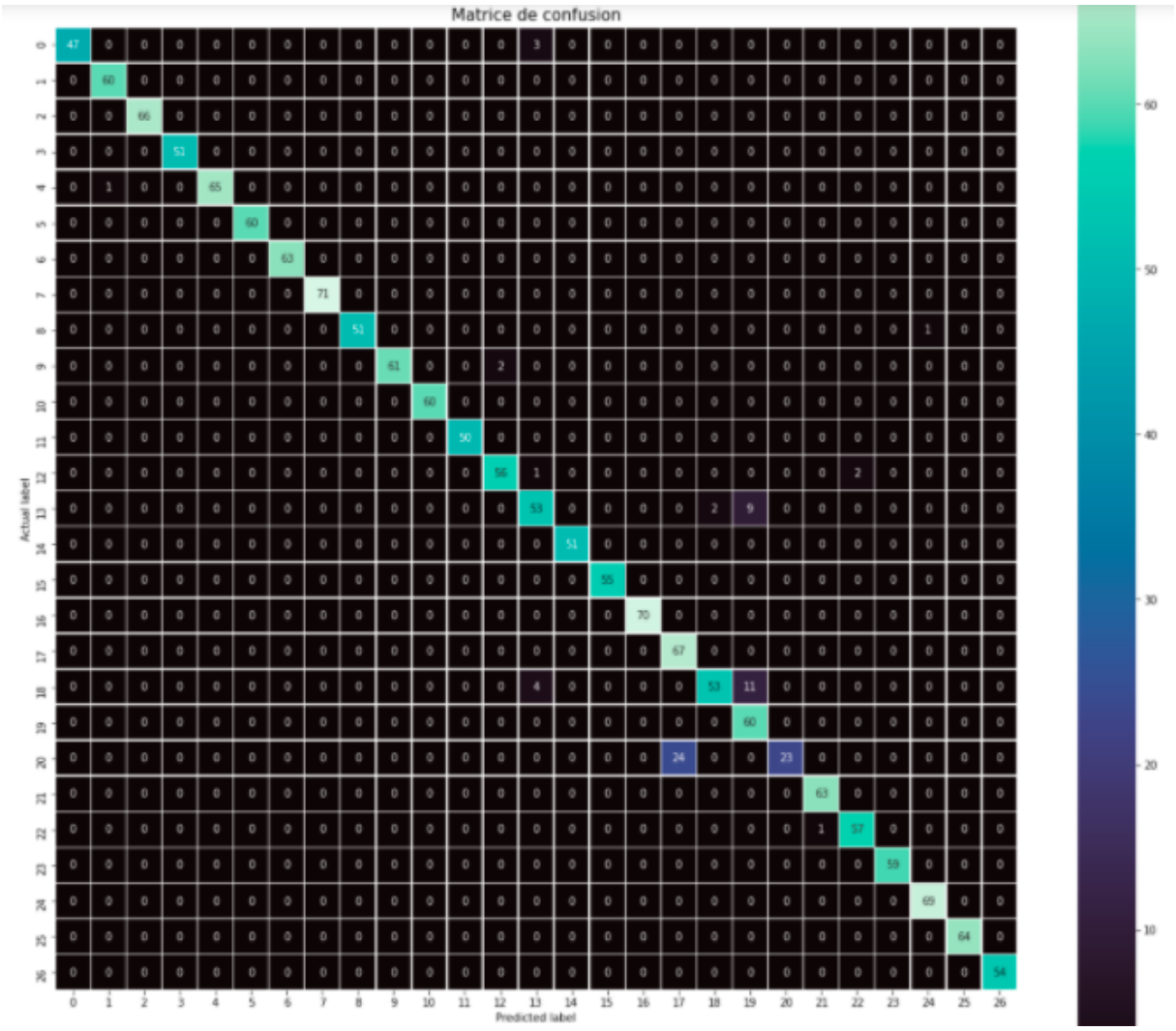
Une data-augmentation a aussi été réalisée en ajoutant un zoom aux images, une rotation. Enfin, nous avons entrainé le modèle sur 25 Epochs avec une évaluation sur les données tests à chacune d'entre-elles.

Evaluation du modèle

La val_accuracy calculé atteint 96%. Le modèle semble suffisamment entraîné au vu du graphique de l'évolution du Loss et Val_Loss qui atteignent un plateau :



Une matrice de confusion a été créée pour observer les classes qui ont posé problème. La plupart des lettres sont bien classées sauf pour la lettre U dont la moitié des images se retrouve dans la classe R. Nous avons tout de même constaté que le modèle différenciait bien ces deux lettres lors des tests live.



Test Vidéo

L'objectif est de maintenant tester notre modèle en temps réel. Nous reprenons la même structure que pour l'acquisition des données pour l'entrainement du modèle. A la différence que nous ajoutons la prédiction du signe effectué avec la main et affichons la lettre pour former un mot. Un dictionnaire (word_dict) a été réalisé associant la lettre à son label.

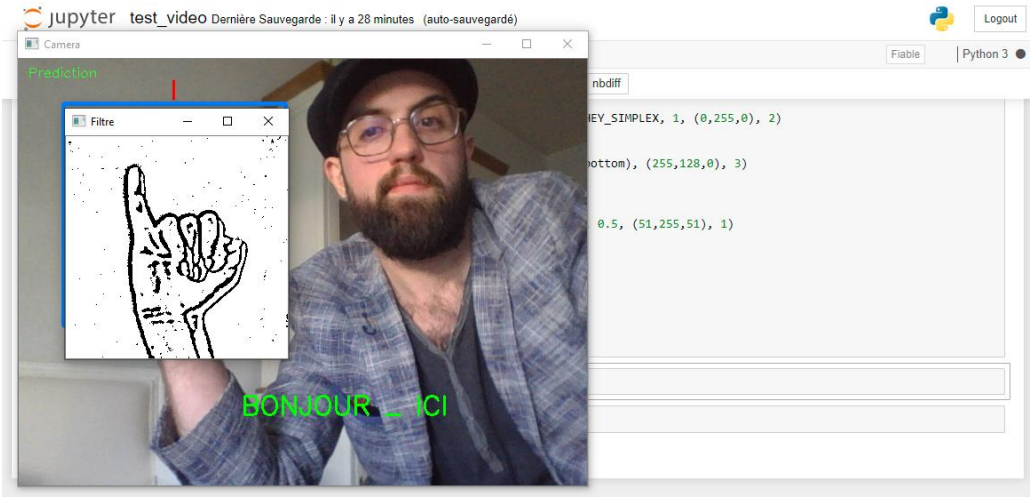
Afin d'avoir un résultat satisfaisant, le test doit être réalisé sur un fond uniforme pour éviter les artefacts parasites.

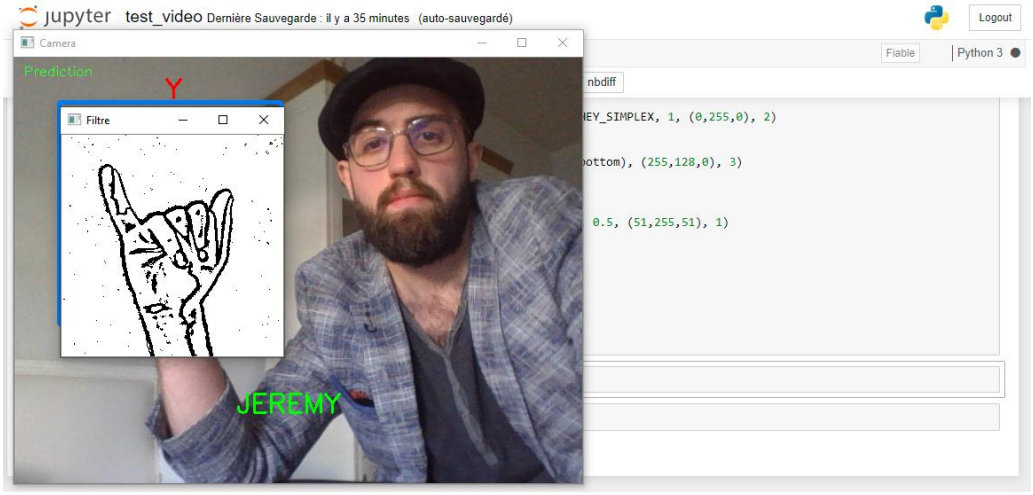
Bien qu'à l'ouverture de la caméra, la main n'est pas encore placée dans le cadre de détection, le modèle prédit de façon chaotique des lettres du fait d'une trop grande sensibilité.

Une prédiction est affichée en rouge en fonction du signe effectué. Cette prédiction renvoie la lettre ayant la plus grande probabilité (word_dict [np.argmax(pred)]).

Un algorithme a été ajouté afin d'afficher un mot. Toutes les prédictions sont stockées dans une liste si la probabilité de cette prédiction est supérieure à 60-70%. Si la lettre enregistrée à chaque frame est présente plus de 35 fois dans la liste, elle est ajoutée au mot qu'on veut former. La liste est réinitialisée à l'ajout de cette lettre

Ce processus est toutefois long (35 frames doivent être analysées) mais permet de s'assurer une détection du bon signe. En cas d'erreur une ligne de code a été ajoutée pour supprimer la dernière lettre ajoutée au mot.





Certaines lettres sont plus difficilement déchiffrables que d'autres (notamment le F, K, M, N, T). Aussi, la lettre A ne semble pas avoir une probabilité prédictive élevé, et ne s'ajoute pas au mot. Nous avons pu malgré tout construire des mots et phrases. Pour des raisons de visibilité, nous avons changé l'espace par un underscore.

Conclusion

A partir d'un Dataset d'images modifiées, nous avons pu créer un modèle de classification. Bien que l'accuracy sur les données tests fût satisfaisant, le test vidéo a été plus compliqué à mettre en place. La méthode choisie n'est clairement pas la plus performante. Beaucoup de facteurs peuvent altérer les prédictions du modèle et l'expérience en générale :

- Bien qu'un filtre a été ajouté sur notre ROI afin de supprimer le background, d'accentuer les traits du contour de la main, certains artefacts restent présents notamment en présence de variation de luminosité.

- Certaines lettres sont plus compliquées à former avec ce genre de méthode qui peuvent être aussi dû à leur ressemblance avec d'autres lettres, et se confondent si nous positionnons pas notre main correctement dans le cadre de détection. Le modèle compare les frames du ROI avec les images sur lesquelles il s'est entrainé, de trop grande variation de position de la main induit ce dernier en erreur.

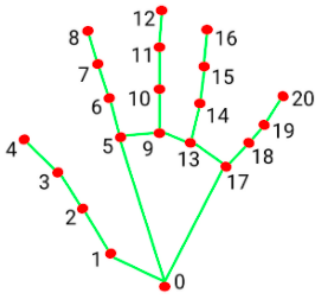
- La formation des mots dans le langage des signes sollicite généralement les deux mains, ce qui n'a pas été fait ici, seule la main gauche est prise en compte par le modèle.

- La vidéo n'est pas fluide du au calcule de probabilité de la prédiction. L'intégration de la lettre au mot qu'on veut former est un processus assez long.

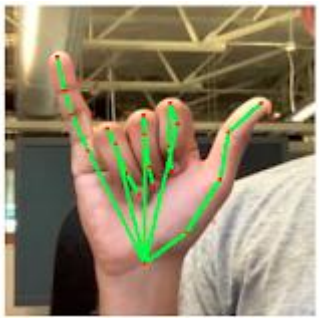
De nombreuses questions se sont posées tout au long du projet. Le premier obstacle auquel nous nous sommes confrontés est la réalisation de notre Dataset. Nous étions partis vers la capture d'images en couleur de notre main faisant les différents signes, mais nous avons estimé que trop de paramètres rentreraient en compte et fausseraient les prédictions de notre modèle sur les tests vidéo.

Nous nous sommes donc inspirer de la méthode de classification des signes avec des images traitées en amont. Comme nous l'avons constaté, cette méthode n'est pas la plus optimale.

D’autres solutions peuvent être mentionnées et qui ont été testées dans les autres groupes comme MediaPipe Hands qui détecte et repère les différentes articulations de la main.



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP



Enfin une autre méthode est d’effectuer une feature-extraction (SURF ou BRISK) comme évoqué dans la publication d’Abhiruchi Bhattacharya *et al.* (2019) par exemple et qui montre de bons résultats avec des modèles de Machine Learning SVM, Régression Logistique, KNN, Classification Naïve Bayésienne.

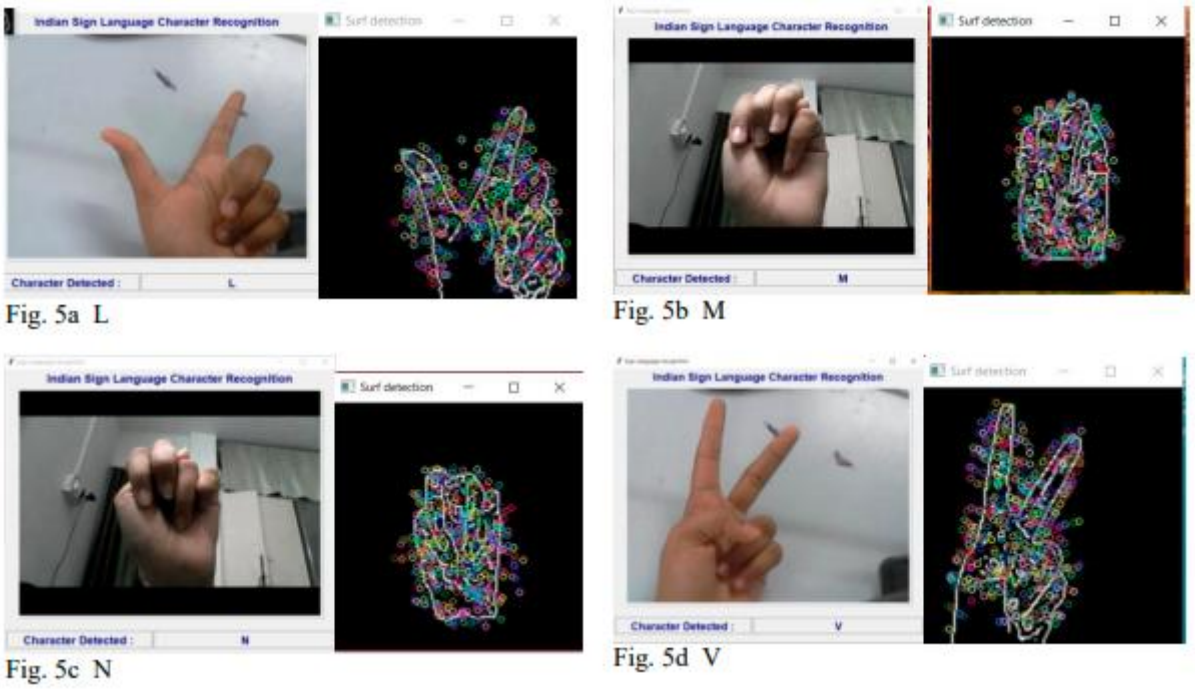


Figure 2. Exemple de détection de lettre sur vidéo avec pre-processing et SURF detection d'après Abhiruchi Bhattacharya et al. 2019.