Vision par ordinateur et détection d'objets
Rapport de projet

Colin BOUREZ, Gabriel GONTIER

10 avril 2025

Introduction

Ce projet consiste à nous permettre de jouer à **Space Invaders** par **détection de mouvements** face à une caméra. Pour cela, il va s'agir d'entraîner un **réseau de neurones** convolutionnel afin de réaliser une tâche de **classification** à partir d'échantillons de mouvements créés nous-mêmes.

Voici les commandes attendues :
— **Lever la main gauche** : déplacement vers la gauche
— **Lever la main droite** : déplacement vers la droite
— **Lever les deux mains** : démarrer le jeu
— **Lever un poing** : tirer

Nous avons donc commencé par capturer un total d'environ **5000 images** des différents mouvements que nous avons réalisés face à la caméra. Précisons que nous avons capturé des images à part pour former un échantillon de **test** bien distinct de celui d'**entraînement**.

Ensuite, nous avons créé un modèle simple de **CNN** puis un modèle similaire mais plus complexe. Nous les avons entraînés et testés à partir de nos données.

Enfin, nous avons modifié le script Python qui sert de **module de contrôle** afin d'adapter les commandes du jeu à la reconnaissance de mouvements face caméra à partir du CNN.

Table des matières

1 Capture d'images **3**

2 Création d'un modèle **4**
2.1 Mise en place 4
2.2 Chargement des données 4
2.3 Traitement et analyse des données 5
2.4 Augmentation des données 6
2.5 Construction du modèle 6
2.6 Entraînement du modèle 7

3 Module de contrôle **8**

4 Jouez ! **8**

1 Capture d'images

Afin d'effectuer une capture d'images de nos mouvements, nous avons écrit un **script Python Capture.ipynb** qui accède à la webcam de l'ordinateur sur lequel il est exécuté.

Ce script capture des images sur une fréquence de **20 par secondes** jusqu'à arrêt par l'utilisateur via la touche **q**.

Le programme permet ensuite d'**enregistrer les différentes images** ainsi récupérées dans le dossier où Jupyter Notebook est ouvert et avec le titre souhaité.

Enfin, le script affiche le nombre d'images qui ont été capturées et enregistrées afin de suivre la taille des différentes **classes** de l'échantillon, à savoir :

— **LEFT** : main gauche levée
— **RIGHT** : main droite levée
— **ENTER** : deux mains levées
— **FIRE** : un poing levé

.

Pour répartir les photographies dans leurs classes respectives, nous les avons manuellement placées dans un dossier portant le nom de la classe en question. Nous avons finalement obtenu un dossier **Train** et un dossier **Test** pour les échantillons d'**entraînement** et de **test** respectivement, chacun contenant un dossier pour chaque classe.

Voici quelques exemples d'images ainsi récupérées :

FIGURE 1 – **'Enter'**

FIGURE 2 – **'Fire'**

Les dossiers des échantillons sont hébergés aux liens suivants :

— **Entraînement :** https ://drive.google.com/drive/folders/1S5nBNDMvk647DkojJQlNKy6zgln6Pe0u ?usp=sharing
— **Test :** https ://drive.google.com/drive/folders/1DuO82yhIdM51kMhKFZFUmBgFS_vFzXIX ?usp=drive_link

2 Création d'un modèle

2.1 Mise en place

Dans le script **model.ipynb**, nous commençons par importer les modules nécessaires tels que **PyTorch** pour programmer un réseau de neurones, ou **MediaPipe** afin de détecter nos mains sur les jeux de données.

Nous fixons ensuite des seeds aléatoires pour permettre une reproductibilité de nos expériences. Enfin, nous définissons le périphérique qui sera utilisé pour les calculs : un **GPU** si disponible ou, à défaut, le **CPU** de l'ordinateur.

Précisons que pour faire fonctionner le script qui suit, il est nécessaire de télécharger les dossiers **Train** et **Test** aux liens ci-dessus dans le répertoire **space-invaders** du projet.

2.2 Chargement des données

Nous initialisons **MediaPipe** afin de récupérer 63 coordonnées de mains (ou landmarks) par image. Le nombre maximal de mains détectables est fixé à 2 en raison de la commande **ENTER**.

Une fonction **extract_landmarks** est définie pour cette extraction de landmarks et va boucler sur la totalité des dossiers d'images d'entraînement puis de test via une fonction **process_dataset** afin d'enregistrer les coordonnées dans deux fichiers **CSV** distincts : **train_landmarks.csv** et **test_landmarks.csv**.

Enfin, les données des deux fichiers CSV sont lues et chargées en tant que **DataFrames Pandas**. Nous en affichons à chaque fois les 5 premières lignes pour vérifier que ce chargement s'est bien déroulé. Nous observons alors que chaque ligne correspond à une image : son label se trouve en première colonne puis nous retrouvons ses 63 landmarks pour un total de 64 colonnes.

	label	x_0	y_0	z_0	x_1	y_1	z_1
0	ENTER	0.313832	0.764376	2.474214e-07	0.358308	0.734033	-0.026857
1	ENTER	0.650379	0.380406	2.110198e-07	0.603407	0.368063	-0.015487
2	ENTER	0.663164	0.614228	1.384285e-07	0.619237	0.591525	-0.012178
3	ENTER	0.338837	0.690563	3.175797e-07	0.293740	0.682436	-0.011694
4	ENTER	0.716060	0.874130	3.947711e-07	0.661055	0.861623	-0.025433

FIGURE 3 – **Set d'entraînement** : labels et 7 premiers landmarks des 5 premières images

2.3 Traitement et analyse des données

Il s'agit ici de préparer les données d'entraînement et de test des deux fichiers CSV en les normalisant à l'aide d'un scaler **MinMax**. Cela fait en sorte que tous les landmarks de chaque image soient compris **entre 0 et 1** : nous n'avons plus de valeurs négatives. Pour comparaison, voici le même aperçu du jeu d'entraînement que ci-dessus :

	label	x_0	y_0	z_0	x_1	y_1	z_1
0	ENTER	0.329598	0.679027	0.780843	0.373168	0.673895	0.312188
1	ENTER	0.662875	0.273039	0.747177	0.631281	0.284959	0.485181
2	ENTER	0.675536	0.520269	0.680040	0.647952	0.522444	0.535528
3	ENTER	0.354360	0.600981	0.845729	0.305172	0.619060	0.542895
4	ENTER	0.727917	0.795074	0.917120	0.691990	0.809492	0.333856

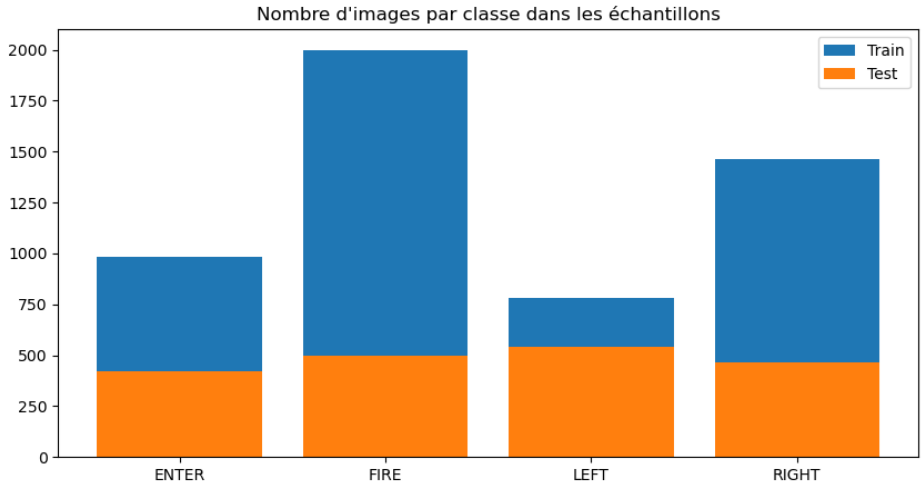
FIGURE 4 – **Set d'entraînement normalisé** : 8 premières colonnes et 5 premières lignes

Nous appliquons cette même transformation sur **l'échantillon de test**.

Nous séparons ensuite les features et les labels dans les deux jeux de données : la première colonne de chacun des deux échantillons correspond aux labels **y_train** et **y_test**, tandis que les 63 autres colonnes donnent les features **X_train** et **X_test** :

- **X_train** : 5158 lignes (images) avec 63 colonnes (landmarks) à chaque fois
- **y_train** : on a bien 5158 lignes (labels) aussi
- **X_test** : 1833 lignes (images) avec 63 colonnes (landmarks) à chaque fois
- **y_test** : on a bien 1833 lignes (labels) aussi

Pour conclure cette section, nous définissons une fonction **analyze_dataset** afin d'analyser les échantillons d'entraînement et de test. Nous l'appelons donc sur **X_train** et **X_test** et obtenons de nouvelles informations : le nombre d'images par classe et par échantillon. La représentation graphique de ces infomations apparaît ci-dessous. Notez que nous avions à l'origine 500 images supplémentaires dans la classe **FIRE** mais nous les avons supprimées pour rééquilibrer légèrement les classes.



2.4 Augmentation des données

Pour rendre le futur modèle plus robuste, nous construisons ici une fonction afin d'augmenter les données. En l'occurrence, la fonction **add_noise** permet d'ajouter un bruit gaussien aux landmarks des différentes images afin de simuler des qualités visuelles variées.

Cette augmentation est alors passée sur **X_train** et **X_test** afin d'obtenir **X_train_augmented** ainsi que **X_test_augmented** dont nous vérifions la forme : nous avons bien conservé nos 5158 et 1833 lignes respectivement avec 63 landmarks chacune.

2.5 Construction du modèle

Pour créer un modèle, nous avons déjà besoin que les labels soient des valeurs **numériques**. Nous commençons donc par appliquer un codage des classes :

- **ENTER** $\rightarrow 0$
- **FIRE** $\rightarrow 1$
- **LEFT** $\rightarrow 2$
- **RIGHT** $\rightarrow 3$

Pour créer un modèle avec **PyTorch**, nous avons également besoin de données au format tensoriel. Nous convertissons ainsi **X_train_augmented**, **y_train**, **X_test_augmented** et **y_test** en tenseurs **X_train_augmented_tensor**, **y_train_tensor**, **X_test_augmented_tensor** et **y_test_tensor**.

Les tenseurs sont ensuite regroupés 2 à 2 pour former le dataset d'entraînement ainsi que celui de test : **train_dataset** et **test_dataset**.

Enfin, des dataloaders **train_loader** et **test_loader** sont créés en répartissant les données des 2 datasets en batches de taille 32. Cette répartition se fait aléatoirement pour le dataset d'entraînement afin de rendre encore une fois le modèle plus robuste en réduisant les risques de **sur-apprentissage** (ou overfitting).

Le modèle **MLP** (MultiLayer Perceptron) est alors prêt à être défini avec plusieurs couches linéaires de nombres de neurones décroissants et suivies de fonctions d'activation **ReLU** $\big($c'est-à-dire $x \mapsto \max(0, x)\big)$. Le modèle sort alors une prédiction pour chacune des 4 classes du modèle et conserve la plus probable.

Le modèle est finalement **initialisé** pour en vérifier la structure, puis **testé** sur une image (sur ses 63 landmarks, pour être exact) : on obtient bien une sortie !

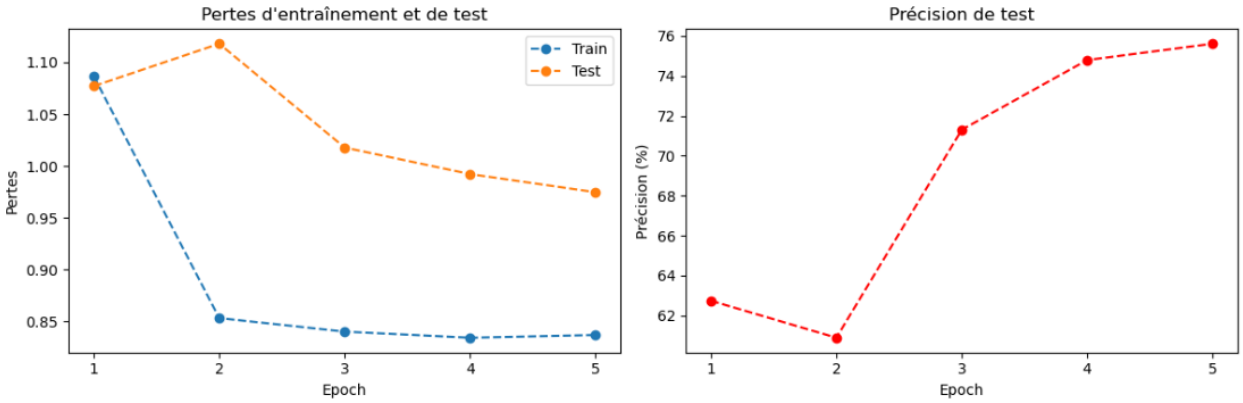
2.6 Entraînement du modèle

Il ne reste maintenant plus qu'à entraîner notre modèle sur le jeu de données. Pour cela, nous définissons une fonction **train_model** qui va boucler sur un certain nombres d'epochs (en l'occurrence 5).

Pour chaque epoch, le modèle est placé en mode **entraînement** et la fonction boucle sur les batches qui composent l'échantillon d'entraînement : on récupère les labels de chaque batch et on en passe les features dans le modèle pour obtenir des prédictions. Les labels sont alors comparés aux prédictions afin d'obtenir une perte et de réaliser une **backpropagation**. Enfin, l'optimiseur met à jour les poids et la loss est incrémentée. On réinitialise le gradient à 0 à chaque étape.

Toujours au sein d'une même epoch, le modèle passe ensuite en mode **évaluation** afin de tester son efficacité. De façon similaire à précédemment, il s'agit de boucler sur les batches de l'échantillon de test afin de comparer les labels aux prédictions et d'obtenir finalement une perte moyenne par batch sur l'epoch actuelle.

Le module **tqdm** permet d'afficher des barres de progression des 2 phases de chaque epoch et les différentes métriques produites au cours des boucles (pertes d'entraînement, de test et précision / accuracy de test) sont enregistrées dans des listes pour être tracées :



L'accuracy à une epoch correspond au pourcentage de prédictions correctes sur l'ensemble des batches. Avec comme perte la **CrossEntropyLoss** (adaptée à la classification multiclasses comme ici) et un optimiseur **Adam** de **taux d'apprentissage** 0.001, nous atteignons facilement les 75 à 80 % d'accuracy sur l'échantillon de test.

Les poids du modèle ainsi entraîné sont finalement enregistrés dans un fichier **model.pth** afin de pouvoir être rappelés dans ce qui va suivre !

3 Module de contrôle

Il s'agit à présent d'utiliser le modèle précédent pour détecter nos mouvements face caméra au cours d'une partie de **Space Invaders**.

Pour ce faire, nous avons adapté le script **control_module.py** en y rajoutant l'architecture de notre modèle **MLP** et en important les poids de la version entraînée stockés dans le fichier **model.pth**.

Nous utilisons à nouveau **MediaPipe** pour détecter les mains sur le flux vidéo et d'en récupérer les coordonnées. Nous avons donc initialisé le module et créé une fonction **extract_landmarks** qui remplit ce rôle à partir d'une **frame** passée en paramètre.

Le codage des classes évoqué précédemment est également rappelé et une fonction **show_video** permet d'afficher pendant le jeu le retour caméra de l'utilisateur.

Enfin, des **boucles** permettent d'allumer la caméra afin de récupérer 2 frames par seconde qui sont passées dans la fonction d'extraction de landmarks. Ces derniers sont passés dans le modèle afin de prédire la commande associée et de l'envoyer au jeu. A noter que les commandes prédites par le modèle sont affichées en temps réel sur le retour vidéo et que la touche **q** permet de mettre fin au jeu en arrêtant la capture.

4 Jouez !

Il ne reste plus qu'à jouer. Pour cela, il suffit de se rendre dans le répertoire **space-invaders** du projet via un terminal et de suivre les instructions du repo **GitHub** :

1. Démarrer le serveur **HTTP** pour le jeu :

```
python -m http.server 8000
```

2. Démarrer le serveur **WebSocket** :

```
node server.js
```

Se rendre ensuite sur l'**URL** du jeu : http ://localhost :8000

3. Démarrer le **module de contrôle** :

```
python control_module.py
```

Une vidéo de démonstration est disponible juste ici :
https ://drive.google.com/file/d/1POUiXCxvYLY9IdonZZHKTGHfaqgf0VCf/view ?usp=sharing