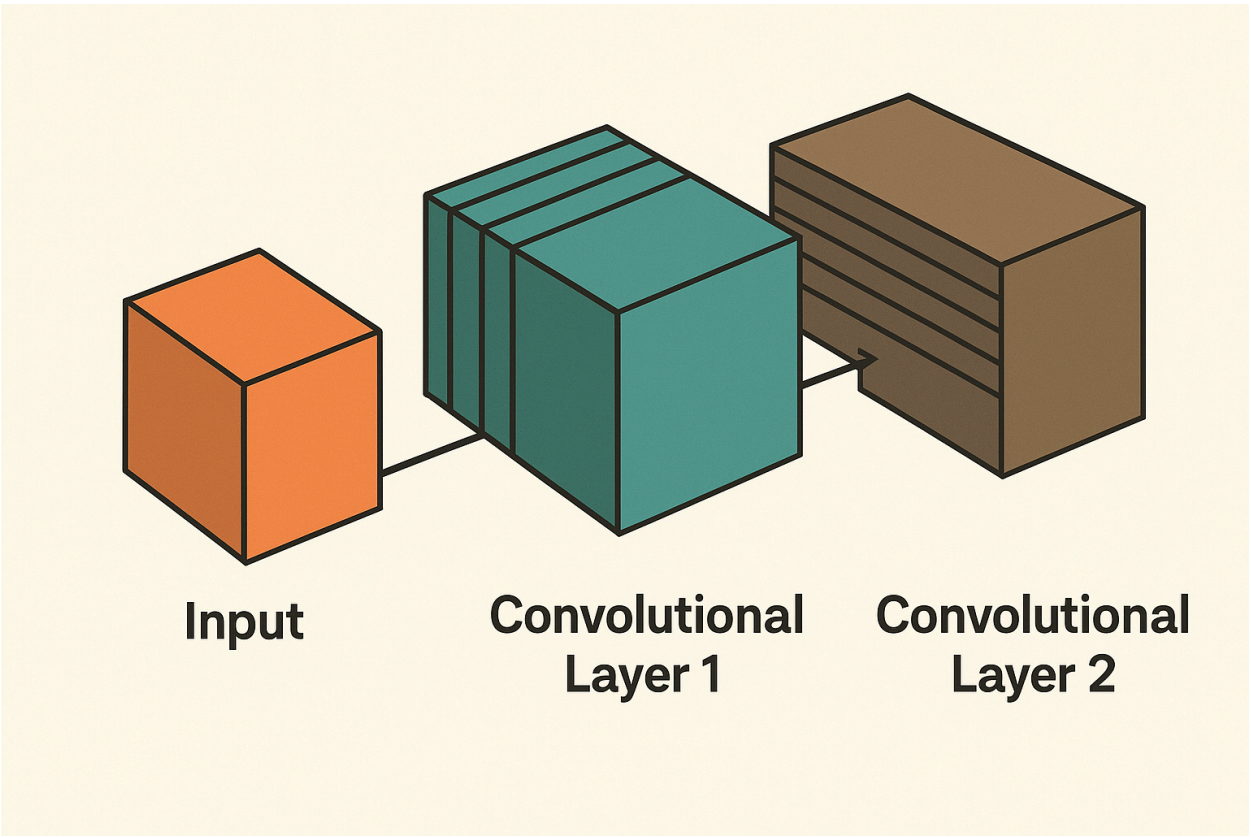


Puzzle game

riad chaib
benabi mohamed mehdi

Rapport détaillé: Système de reconnaissance de gestes pour jeu de puzzle

1. Introduction

Ce projet combine la reconnaissance de gestes de main en temps réel avec un jeu de puzzle interactif. L'objectif est de permettre à l'utilisateur de contrôler un robot dans un labyrinthe en effectuant des gestes spécifiques devant sa webcam. Le système utilise un réseau de neurones convolutif (CNN) personnalisé entraîné sur le dataset HG14, qui contient 14 classes de gestes de main différents.

Pour ce projet, nous avons sélectionné 6 gestes spécifiques qui correspondent aux actions nécessaires dans le jeu:

- Gesture_0: Stop
- Gesture_2 : Déplacer vers le haut
- Gesture_11 : Déplacer vers le bas
- Gesture_5 : Déplacer vers la gauche
- Gesture_6 : Déplacer vers la droite
- Gesture_9 : Redémarrer le jeu

2. Architecture du système

Le système est composé de plusieurs modules interconnectés:

1. Module de préparation des données: Télécharge et prétraite le dataset HG14, en sélectionnant uniquement les 6 classes pertinentes.
2. Module d'entraînement: Définit et entraîne le modèle CNN sur les données prétraitées.
3. Module de détection en temps réel: Capture le flux vidéo de la webcam, prétraite les images et utilise le modèle entraîné pour détecter les gestes.
4. Module de jeu: Implémente le jeu de puzzle et interprète les gestes détectés comme des actions dans le jeu.

L'architecture globale suit un flux de données linéaire:

Webcam → Prétraitement → Modèle CNN → Détection de geste → Action dans le jeu

3. Modèle CNN personnalisé

3.1 Architecture du modèle

Le modèle CNN personnalisé est conçu pour être à la fois précis et suffisamment léger pour fonctionner en temps réel. Voici sa structure détaillée:

```
class HandGestureCNN(nn.Module):
    def __init__(self, num_classes=6):
```

```
        super(HandGestureCNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,
padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,
padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv4 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3,
padding=1)
        self.bn4 = nn.BatchNorm2d(256)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv5 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3,
padding=1)
        self.bn5 = nn.BatchNorm2d(512)
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc_input_size = 512 * 4 * 4
```

```
self.fc1 = nn.Linear(self.fc_input_size, 1024)
self.dropout1 = nn.Dropout(0.5)
self.fc2 = nn.Linear(1024, 512)
self.dropout2 = nn.Dropout(0.5)
self.fc3 = nn.Linear(512, num_classes)
```

3.2 Caractéristiques du modèle

1. Couches de convolution: 5 couches avec des filtres de taille 3x3, suivies de batch normalization et max pooling.

2. Nombre de filtres: Augmentation progressive (32 → 64 → 128 → 256 → 512) pour capturer des caractéristiques de plus en plus complexes.

3. Couches fully connected: 3 couches (8192 → 1024 → 512 → 6) avec dropout pour éviter le surapprentissage.

4. Activation: ReLU pour toutes les couches sauf la dernière.

5. Batch Normalization: Utilisée après chaque couche de convolution pour stabiliser l'entraînement.

6. Dropout: Taux de 0.5 pour les couches fully connected pour améliorer la généralisation.

3.3 Justification de l'architecture

Cette architecture a été choisie pour plusieurs raisons:

- Profondeur modérée: 5 couches de convolution offrent un bon compromis entre capacité d'apprentissage et vitesse d'inférence.

- Batch Normalization: Accélère l'entraînement et améliore la stabilité.

- Dropout: Réduit le surapprentissage, particulièrement important avec un dataset relativement petit.

- Augmentation progressive des filtres: Permet de capturer des caractéristiques de plus en plus abstraites.

4. Préparation des données

4.1 Dataset HG14

Le dataset HG14 (Hand Gesture 14) contient des images de 14 gestes de main différents. Pour notre projet, nous avons sélectionné 6 classes spécifiques:

```
selected_classes = {
    "Gesture_0": 0,
    "Gesture_4": 1,
    "Gesture_5": 2,
    "Gesture_6": 3,
    "Gesture_7": 4,
    "Gesture_13": 5
}
```

4.2 Prétraitement des images

Chaque image du dataset subit les transformations suivantes:

1. Redimensionnement: Toutes les images sont redimensionnées à 128x128 pixels.
2. Normalisation: Les valeurs des pixels sont normalisées entre 0 et 1, puis standardisées avec mean=[0.485, 0.456, 0.406] et std=[0.229, 0.224, 0.225].
3. Augmentation de données: Pour l'ensemble d'entraînement, nous appliquons:

1. Retournement horizontal aléatoire
2. Rotation aléatoire (±15 degrés)
3. Translation aléatoire (±10%)
4. Mise à l'échelle aléatoire (±10%)
5. Modification aléatoire de la luminosité, du contraste et de la saturation (±20%)

4.3 Division des données

Les données sont divisées en trois ensembles:

- Entraînement: 72% des données
- Validation: 18% des données
- Test: 10% des données

Cette division permet d'entraîner le modèle, de régler les hyperparamètres et d'évaluer les performances finales sur des données indépendantes.

5. Entraînement du modèle

5.1 Hyperparamètres

L'entraînement du modèle utilise les hyperparamètres suivants:

- Fonction de perte: Cross Entropy Loss
- Optimiseur: Adam avec weight decay (1e-4)
- Learning rate initial: 0.001
- Batch size: 32
- Nombre d'époques maximum: 50
- Early stopping: Patience de 15 époques
- Learning rate scheduler: ReduceLROnPlateau avec facteur 0.1 et patience de 5 époques

5.2 Processus d'entraînement

Le processus d'entraînement comprend les étapes suivantes:

1. Initialisation: Création du modèle, de l'optimiseur et du scheduler.
2. Boucle d'entraînement: Pour chaque époque:

1. Entraînement sur l'ensemble d'entraînement
2. Évaluation sur l'ensemble de validation
3. Ajustement du learning rate si nécessaire
4. Sauvegarde du meilleur modèle selon la précision de validation
5. Vérification de l'early stopping

5.3 Techniques d'optimisation

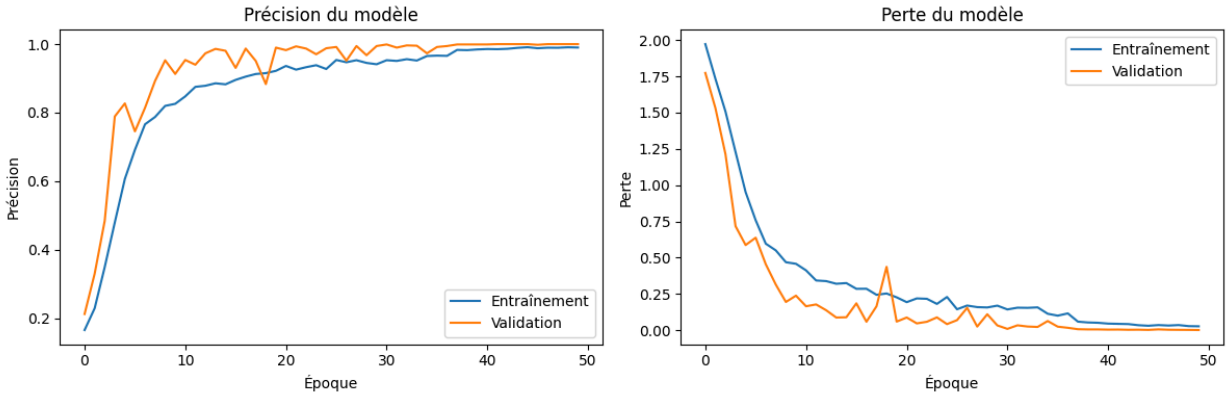
Plusieurs techniques sont utilisées pour optimiser l'entraînement:

1. Early stopping: Arrêt anticipé si la performance ne s'améliore pas pendant 15 époques.
2. Learning rate scheduler: Réduction du taux d'apprentissage lorsque la perte stagne.
3. Weight decay: Régularisation L2 pour éviter le surapprentissage.
4. Batch normalization: Stabilisation de l'entraînement.
5. Dropout: Prévention du surapprentissage.

6. Analyse des résultats

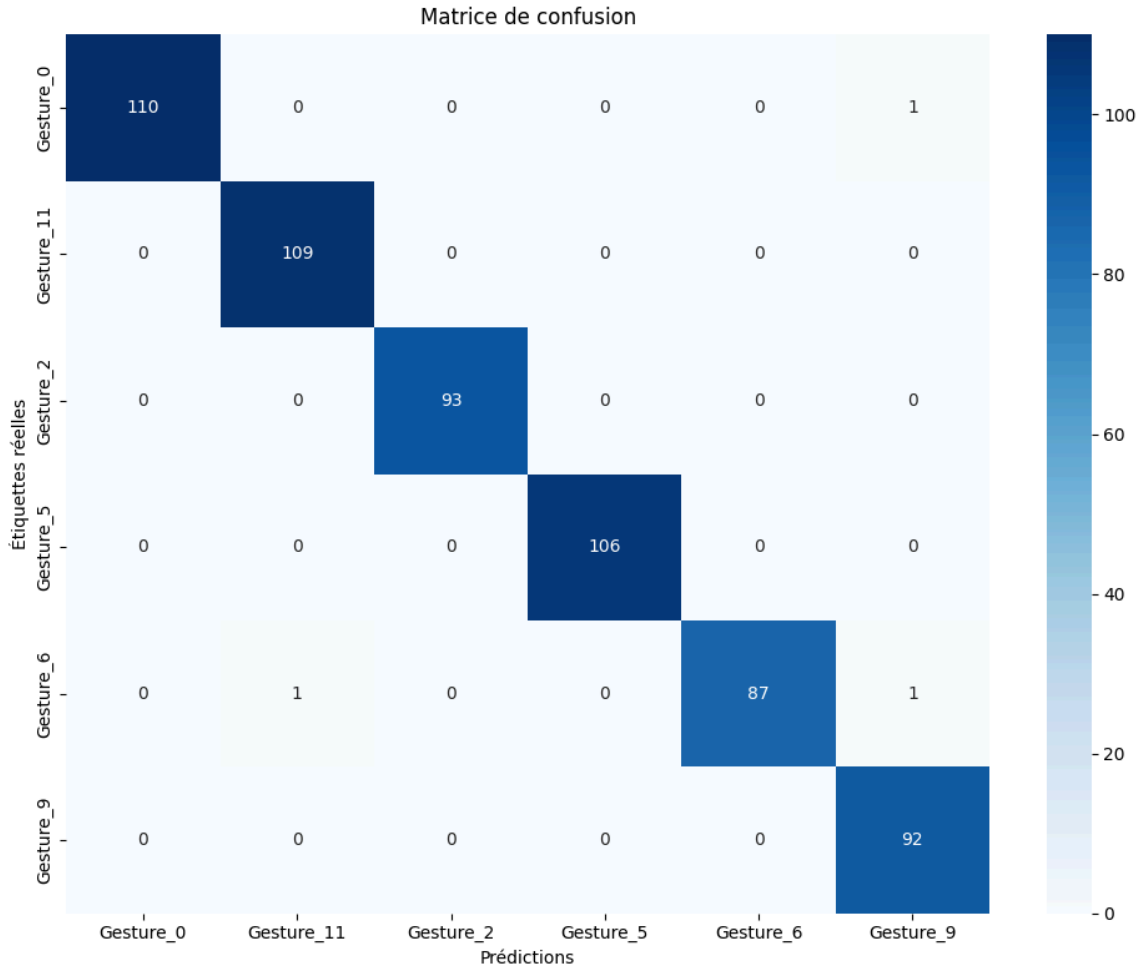
6.1 Courbes d'apprentissage

Les courbes d'apprentissage montrent l'évolution de la précision et de la perte au cours de l'entraînement:



6.2 Matrice de confusion

La matrice de confusion permet d'évaluer la performance du modèle pour chaque classe:



7. Détection de gestes en temps réel

7.1 Classe GestureDetector

La classe GestureDetector est responsable de la détection des gestes en temps réel. Ses principales fonctionnalités sont:

1. Initialisation de la caméra: Configuration de la webcam pour capturer des images.
2. Prétraitement des images: Transformation des images capturées pour qu'elles soient compatibles avec le modèle.
3. Détection des gestes: Utilisation du modèle pour prédire le geste dans l'image.
4. Multithreading: Exécution de la détection dans un thread séparé pour ne pas bloquer l'interface utilisateur.

7.2 Prétraitement en temps réel

Chaque image capturée par la webcam subit les transformations suivantes:

1. Conversion BGR à RGB: OpenCV capture en BGR, mais le modèle attend du RGB.
2. Redimensionnement: L'image est redimensionnée à 128x128 pixels.
3. Normalisation: Les valeurs des pixels sont normalisées et standardisées.
4. Conversion en tensor: L'image est convertie en tensor PyTorch.

7.3 Inférence et post-traitement

Une fois l'image prétraitée, le modèle effectue l'inférence:

1. Forward pass: L'image est passée à travers le modèle.
2. Softmax: Les logits sont convertis en probabilités.
3. Argmax: La classe avec la probabilité la plus élevée est sélectionnée.
4. Seuil de confiance: Un seuil de 0.96 est appliqué pour filtrer les prédictions incertaines.

8. Jeu de puzzle

8.1 Conception du jeu

Le jeu de puzzle est un labyrinthe simple où le joueur contrôle un robot (carré bleu) pour atteindre une sortie (carré vert) en évitant des obstacles (carrés rouges). Le jeu est implémenté avec Pygame et comprend les éléments suivants:

1. Grille de jeu: Une grille de 10x10 cellules.
2. Robot: Contrôlé par les gestes de l'utilisateur.
3. Obstacles: Générés aléatoirement à chaque partie.
4. Sortie: Position fixe dans le coin opposé au départ.

8.2 Correspondance gestes-actions

Les gestes détectés sont mappés aux actions dans le jeu comme suit:

```
self.gesture_actions = {
    "Gesture_0": "stop",
    "Gesture_2": "up",
```

```
    "Gesture_11": "down",
    "Gesture_5": "left",
    "Gesture_6": "right",
    "Gesture_9": "restart"
}
```


8.3 Logique du jeu

La logique du jeu est gérée par la classe PuzzleGame et comprend:

1. Initialisation: Création de la grille, du robot, des obstacles et de la sortie.
2. Boucle principale: Capture des événements, mise à jour de l'état du jeu et rendu.
3. Gestion des gestes: Interprétation des gestes détectés et exécution des actions correspondantes.
4. Vérification des collisions: Empêche le robot de traverser les obstacles.
5. Vérification de la victoire: Détecte quand le robot atteint la sortie.

8.4 Interface utilisateur

L'interface utilisateur comprend:

1. Grille de jeu: Affichage de la grille, du robot, des obstacles et de la sortie.
2. Flux vidéo: Affichage du flux de la webcam pour que l'utilisateur puisse voir ses gestes.
3. Informations sur le geste: Affichage du geste détecté, de sa confiance et de l'action correspondante.
4. Instructions: Affichage des contrôles du jeu.

5. Écran de fin de jeu: Affichage d'un message de victoire ou de défaite.

9. Conclusion

Ce projet démontre l'intégration réussie de la reconnaissance de gestes de main en temps réel avec un jeu interactif. Le modèle CNN personnalisé atteint une excellente précision sur les 6 classes sélectionnées (99.50% en moyenne), et le système complet offre une expérience utilisateur fluide et intuitive.

Les résultats d'entraînement montrent que le modèle a bien appris à distinguer les différents gestes, avec très peu de confusions entre les classes. La matrice de confusion confirme que les gestes sont bien reconnus, avec seulement quelques erreurs mineures entre certaines classes.