Reconnaissance d'Émotions Faciales en Temps Réel

Rapport Technique - Deep Learning

Cyril

Novembre 2025

Résumé

Ce rapport présente le développement d'un système de reconnaissance d'émotions faciales en temps réel utilisant des réseaux de neurones convolutifs (CNN). Nous détaillons l'architecture initiale, les optimisations apportées, ainsi que les choix techniques effectués pour améliorer les performances de détection. Le système est capable de classifier huit émotions (colère, dégoût, peur, joie, tristesse, surprise, neutralité, et mépris) à partir d'un flux vidéo webcam, en utilisant le dataset Balanced AffectNet.

Table des matières

1 Introduction

La reconnaissance automatique des émotions faciales est un domaine en pleine expansion avec des applications dans l'interaction homme-machine, la santé mentale, le marketing, et l'éducation. Ce projet implémente un système complet de reconnaissance d'émotions en temps réel, depuis l'entraînement du modèle jusqu'à l'inférence via webcam.

1.1 Objectifs

— Développer un modèle CNN capable de classifier 8 émotions faciales (7 de FER2013 + Contempt de FER+)
— Implémenter une application temps réel avec détection de visage
— Optimiser les performances pour une utilisation fluide
— Améliorer la précision par des techniques avancées de deep learning

2 Architecture Initiale

2.1 Dataset : Balanced AffectNet

Le dataset Balanced AffectNet est une version équilibrée et pré-traitée du dataset AffectNet original. Il contient :

— 41 008 images de visages en couleur (RGB)
— Résolution : 75×75 pixels
— 8 classes d'émotions parfaitement équilibrées (∼5 126 images par classe)
— Split prédéfini : train (29 526), val (7 382), test (4 100)

Avantages par rapport à FER2013 :

— Images RGB offrant plus d'informations (couleur de peau, rougeurs)
— Résolution supérieure (75×75 vs 48×48)
— Dataset parfaitement équilibré (pas de biais de classe)
— Classe Contempt (mépris) incluse nativement

2.2 Modèle Initial (`model.py`)

L'architecture initiale est un CNN simple avec 3 blocs convolutifs :

```
class FaceEmotionCNN(nn.Module):
    def __init__(self, num_classes=8, in_channels=3, input_size=75):
        super(FaceEmotionCNN, self).__init__()
        # Bloc Convolutif 1 (75x75 -> 37x37)
        self.conv1a = nn.Conv2d(in_channels, 64, kernel_size=3, padding
    =1)
        self.bn1a = nn.BatchNorm2d(64)
        self.conv1b = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn1b = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(0.1)

        # ... Blocs 2, 3, 4 similaires ...

        # Global Average Pooling
        self.global_avg_pool = nn.AdaptiveAvgPool2d(1)

```

```
        # Couches Fully Connected
        self.fc1 = nn.Linear(512, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, num_classes)  # 8 emotions
```

Listing 1 – Architecture CNN pour Balanced AffectNet

2.2.1 Analyse de l'architecture initiale

TABLE 1 – Dimensions à travers le réseau (Balanced AffectNet)

Couche	**Entrée**	**Sortie**	**Paramètres**
Block1 (2×Conv) + Pool	$3 \times 75 \times 75$	$64 \times 37 \times 37$	38,592
Block2 (2×Conv) + Pool	$64 \times 37 \times 37$	$128 \times 18 \times 18$	147,712
Block3 (2×Conv) + Pool	$128 \times 18 \times 18$	$256 \times 9 \times 9$	590,336
Block4 (2×Conv) + Pool	$256 \times 9 \times 9$	$512 \times 4 \times 4$	2,360,320
Global Avg Pool	$512 \times 4 \times 4$	512	0
FC1 + BN	512	256	131,584
FC2 + BN	256	128	33,024
FC3	128	8	1,032
Total			**≈ 3.3M**

Points positifs :
— Utilisation de Batch Normalization pour stabiliser l'entraînement
— Dropout (50%) pour régulariser et éviter l'overfitting
— Architecture simple et rapide à entraîner
Limitations :
— Peu de couches convolutives (faible capacité d'extraction de features)
— Nombre de filtres limité (32-64-128)
— Pas de régularisation dans les couches convolutives
— Couche FC1 très large (2.3M paramètres) créant un goulot d'étranglement

2.3 Script d'entraînement initial (`train.py`)

```
# Hyperparametres
BATCH_SIZE = 64
LEARNING_RATE = 0.001
EPOCHS = 25

# Transformations simples
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.ToTensor(),
])

# Chargement du dataset
dataset = FER2013Dataset('./data/fer2013/fer2013.csv', transform=
    transform)
train_loader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

```

```python
# Modele et optimiseur
model = FaceEmotionCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

# Boucle d'entrainement
for epoch in range(EPOCHS):
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}, Loss: {running_loss/len(train_loader)}")

torch.save(model.state_dict(), 'emotion_model.pth')
```

Listing 2 – Script d'entraînement initial

Limitations de l'entraînement initial :

— Pas de data augmentation (risque d'overfitting)
— Pas de séparation train/validation (impossible de détecter l'overfitting)
— Learning rate fixe (convergence sous-optimale)
— Pas d'early stopping (gaspillage de ressources)
— Pas de gestion du déséquilibre des classes

2.4 Application initiale (app.py)

L'application initiale effectue :

1. Capture vidéo via webcam
2. Détection de visage avec Haar Cascade
3. Prétraitement de l'image (redimensionnement 48×48, grayscale)
4. Inférence avec le modèle CNN
5. Affichage de l'émotion détectée avec emoji

```python
while True:
    ret, frame = cap.read()
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray_frame,
                                          scaleFactor=1.3,
                                          minNeighbors=5)

    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
        roi_gray = gray_frame[y:y+h, x:x+w]

        # Preprocessing
        roi_tensor = data_transform(roi_gray).unsqueeze(0).to(device)

        # Prediction
        with torch.no_grad():
```

```
            outputs = model(roi_tensor)
            probabilities = torch.nn.functional.softmax(outputs, dim=1)
            max_prob, predicted_idx = torch.max(probabilities, 1)

            idx = predicted_idx.item()
            confidence = max_prob.item() * 100
            emotion_text = emotion_dict[idx]
```

Listing 3 – Boucle principale de l'application initiale

Limitations :

— Prédictions instables (oscillations entre frames)
— Pas de normalisation de l'éclairage
— Paramètres de détection de visage sous-optimaux
— Interface utilisateur basique

3 Optimisations de l'Architecture

3.1 Nouvelle Architecture CNN

L'architecture améliorée comporte plusieurs optimisations majeures :

```
class FaceEmotionCNN(nn.Module):
    def __init__(self, num_classes=7):
        super(FaceEmotionCNN, self).__init__()

        # Bloc 1 (48x48 -> 24x24) - Double convolution
        self.conv1a = nn.Conv2d(1, 64, kernel_size=3, padding=1)
        self.bn1a = nn.BatchNorm2d(64)
        self.conv1b = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn1b = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(0.25)

        # Bloc 2 (24x24 -> 12x12)
        self.conv2a = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2a = nn.BatchNorm2d(128)
        self.conv2b = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn2b = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.dropout2 = nn.Dropout(0.25)

        # Bloc 3 (12x12 -> 6x6)
        self.conv3a = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn3a = nn.BatchNorm2d(256)
        self.conv3b = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn3b = nn.BatchNorm2d(256)
        self.pool3 = nn.MaxPool2d(2, 2)
        self.dropout3 = nn.Dropout(0.25)

        # Bloc 4 (6x6 -> 3x3)
        self.conv4a = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.bn4a = nn.BatchNorm2d(512)
        self.conv4b = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.bn4b = nn.BatchNorm2d(512)
        self.pool4 = nn.MaxPool2d(2, 2)
        self.dropout4 = nn.Dropout(0.25)
```

```
36
37          # Global Average Pooling
38          self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
39
40          # Fully Connected avec dropout progressif
41          self.fc1 = nn.Linear(512, 256)
42          self.bn_fc1 = nn.BatchNorm1d(256)
43          self.dropout_fc1 = nn.Dropout(0.5)
44
45          self.fc2 = nn.Linear(256, 128)
46          self.bn_fc2 = nn.BatchNorm1d(128)
47          self.dropout_fc2 = nn.Dropout(0.4)
48
49          self.fc3 = nn.Linear(128, num_classes)
```

Listing 4 – Architecture CNN améliorée

3.1.1 Justification des choix architecturaux

1. **Double convolution par bloc (style VGG)**
   Inspiré de VGGNet, deux convolutions 3×3 successives permettent un champ réceptif de 5×5 avec moins de paramètres qu'une convolution 5×5 unique :

$$\text{Paramètres } 3 \times 3 \times 2 = 2 \times (3^2 \times C^2) = 18C^2 \tag{1}$$

$$\text{Paramètres } 5 \times 5 = 5^2 \times C^2 = 25C^2 \tag{2}$$

2. **Augmentation progressive des filtres (64→128→256→512)**
   Suit le principe que les premières couches extraient des features simples (bords, textures) tandis que les couches profondes capturent des concepts complexes (parties du visage, expressions).

3. **Dropout progressif (0.25 dans conv, 0.5→0.4 dans FC)**
   Le dropout dans les couches convolutives (0.25) régularise sans trop perturber l'apprentissage spatial. Un dropout plus fort dans les FC (0.5) combat l'overfitting où le risque est le plus élevé.

4. **Global Average Pooling (GAP)**
   Remplace le flatten traditionnel. Avantages :
   — Réduit drastiquement les paramètres ($512 \times 3 \times 3 = 4608 \rightarrow 512$)
   — Agit comme régularisateur
   — Invariance spatiale accrue

5. **Initialisation Kaiming**
   Initialisation adaptée aux fonctions d'activation ReLU :

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right) \tag{3}$$

TABLE 2 – Comparaison des architectures

Caractéristique	**Initial**	**Amélioré**
Blocs convolutifs	3	4
Convolutions par bloc	1	2
Filtres max	128	512
Paramètres totaux	≈2.46M	≈4.8M
Global Average Pooling	Non	Oui
Dropout conv	Non	Oui (0.25)
Batch Norm FC	Non	Oui

4 Optimisations de l'Entraînement

4.1 Data Augmentation

La data augmentation est cruciale pour un petit dataset comme FER2013. Elle augmente artificiellement la diversité des données d'entraînement.

```
train_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(10),
    transforms.RandomAffine(
        degrees=0,
        translate=(0.1, 0.1),
        scale=(0.9, 1.1)
    ),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
])
```

Listing 5 – Transformations de data augmentation

TABLE 3 – Transformations de data augmentation

Transformation	**Justification**
RandomHorizontalFlip (p=0.5)	Les expressions sont symétriques. Double effectivement le dataset.
RandomRotation (±10°)	Simule les légères inclinaisons de tête naturelles.
RandomAffine (translate)	Compense les variations de position du visage dans le cadre.
RandomAffine (scale 0.9-1.1)	Simule différentes distances caméra-visage.
ColorJitter (brightness, contrast)	Robustesse aux variations d'éclairage.

4.2 Séparation Train/Validation

```
VALIDATION_SPLIT = 0.15  # 15% pour validation

```

```
val_size = int(len(full_dataset) * VALIDATION_SPLIT)
train_size = len(full_dataset) - val_size

train_dataset, val_dataset = random_split(
    full_dataset,
    [train_size, val_size],
    generator=torch.Generator().manual_seed(42)
)
```

Listing 6 – Séparation du dataset

Importance : La validation permet de :
— Détecter l'overfitting (train loss ↓ mais val loss ↑)
— Sélectionner le meilleur modèle
— Ajuster les hyperparamètres

4.3 Early Stopping

L'early stopping arrête l'entraînement quand la validation ne s'améliore plus :

```
PATIENCE = 7
best_val_acc = 0.0
patience_counter = 0

for epoch in range(EPOCHS):
    # ... entrainement ...
    val_loss, val_acc = validate(model, val_loader, criterion, device)

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        patience_counter = 0
        torch.save(model.state_dict(), 'emotion_model_best.pth')
    else:
        patience_counter += 1
        if patience_counter >= PATIENCE:
            print("Early stopping triggered!")
            break
```

Listing 7 – Implémentation de l'early stopping

4.4 Learning Rate Scheduler

Le scheduler réduit le learning rate quand la validation stagne :

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min',       # Surveille la loss
    factor=0.5,       # Divise LR par 2
    patience=3        # Attend 3 epochs sans amelioration
)

# Dans la boucle d'entrainement
scheduler.step(val_loss)
```

Listing 8 – Learning rate scheduler

Principe : Un LR élevé au début permet une convergence rapide, puis un LR plus faible permet un affinage précis.

4.5 Optimiseur AdamW

```
optimizer = optim.AdamW(
    model.parameters(),
    lr=LEARNING_RATE,
    weight_decay=1e-4
)
```

Listing 9 – Optimiseur AdamW avec weight decay

AdamW corrige un problème de Adam : le weight decay est appliqué directement aux poids plutôt qu'au gradient, ce qui donne une meilleure régularisation L2.

4.6 Gradient Clipping

```
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()
```

Listing 10 – Gradient clipping pour la stabilité

Empêche les gradients explosifs qui peuvent déstabiliser l'entraînement.

5 Optimisations de l'Application Temps Réel

5.1 Lissage Temporel des Prédictions

Le problème principal des prédictions frame-par-frame est l'instabilité : l'émotion affichée peut changer rapidement entre frames successives, créant un effet de *flickering* désagréable.

```
from collections import deque

SMOOTHING_WINDOW = 5
prediction_history = deque(maxlen=SMOOTHING_WINDOW)

def get_smoothed_prediction(current_probs):
    prediction_history.append(current_probs.cpu().numpy())

    if len(prediction_history) < 2:
        return current_probs

    # Moyenne ponderee (frames recentes = plus de poids)
    weights = np.linspace(0.5, 1.0, len(prediction_history))
    weights = weights / weights.sum()

    smoothed = np.zeros(7)
    for i, probs in enumerate(prediction_history):
        smoothed += weights[i] * probs

    return torch.tensor(smoothed)
```

Listing 11 – Lissage temporel avec moyenne pondérée

Principe : Au lieu d'utiliser uniquement la prédiction de la frame courante, on calcule une moyenne pondérée sur les N dernières frames. Les frames récentes ont plus de poids pour maintenir la réactivité.

5.2 Égalisation d'Histogramme

```
# Avant preprocessing
roi_gray = cv2.equalizeHist(roi_gray)
```

Listing 12 – Normalisation de l'éclairage

L'égalisation d'histogramme normalise la distribution des niveaux de gris, rendant le modèle plus robuste aux variations d'éclairage.

Image sombre ⟶ Histogramme égalisé

FIGURE 1 – L'égalisation redistribue les intensités sur toute la plage [0, 255]

5.3 Paramètres de Détection Optimisés

```
faces = face_cascade.detectMultiScale(
    gray_frame,
    scaleFactor=1.1,   # Plus precis (etait 1.3)
    minNeighbors=5,
    minSize=(48, 48),  # Taille minimum
    flags=cv2.CASCADE_SCALE_IMAGE
)
```

Listing 13 – Paramètres de détection de visage optimisés

TABLE 4 – Impact des paramètres de détection

Paramètre	Valeur basse	Valeur haute
scaleFactor	Plus précis, plus lent	Moins précis, plus rapide
minNeighbors	Plus de faux positifs	Moins de détections
minSize	Détecte petits visages	Ignore petits visages

5.4 Interface Utilisateur Améliorée

- **Couleurs par émotion** : Chaque émotion a une couleur distinctive
- **Barres de progression** : Affichage des probabilités de toutes les classes
- **Fond pour le texte** : Améliore la lisibilité

```
emotion_colors = {
    0: (0, 0, 255),      # Rouge - Angry
    1: (0, 128, 0),      # Vert fonce - Disgust
    2: (128, 0, 128),    # Violet - Fear
    3: (0, 255, 255),    # Jaune - Happy
    4: (255, 0, 0),      # Bleu - Sad
    5: (0, 165, 255),    # Orange - Surprise
    6: (128, 128, 128)   # Gris - Neutral
}
```

Listing 14 – Couleurs par émotion

6 Amélioration du Dataset

6.1 Avantages du Balanced AffectNet

Par rapport à FER2013, le dataset Balanced AffectNet présente plusieurs avantages majeurs :

1. **Résolution supérieure** : Images 75×75 pixels (vs 48×48)
2. **Images RGB** : 3 canaux de couleur offrant plus d'informations

Émotion	Nombre d'images	Pourcentage
Angry	5,126	12.5%
Contempt	5,126	12.5%
Disgust	5,126	12.5%
Fear	5,126	12.5%
Happy	5,126	12.5%
Neutral	5,126	12.5%
Sad	5,126	12.5%
Surprise	5,126	12.5%

TABLE 5 – Distribution parfaitement équilibrée des classes dans Balanced AffectNet.

3. **Dataset équilibré** : Pas de biais de classe, contrairement à FER2013
4. **8 classes natives** : Inclut Contempt (mépris) dès le départ
5. **Meilleure qualité** : Annotations plus fiables

6.2 Chargement du Dataset

Le dataset Balanced AffectNet est organisé en dossiers par émotion :

```
class BalancedAffectNetDataset(Dataset):
    EMOTION_CLASSES = {
        'Anger': 0, 'Disgust': 1, 'Fear': 2, 'Happy': 3,
        'Sad': 4, 'Surprise': 5, 'Neutral': 6, 'Contempt': 7,
    }

    def __init__(self, root_dir, split='train', transform=None):
        self.images = []
        self.labels = []

        for emotion_name, emotion_idx in self.EMOTION_CLASSES.items():
            emotion_dir = os.path.join(root_dir, split, emotion_name)
            for img_name in os.listdir(emotion_dir):
                self.images.append(os.path.join(emotion_dir, img_name))
                self.labels.append(emotion_idx)

    def __getitem__(self, idx):
        image = Image.open(self.images[idx]).convert('RGB')
        image = np.array(image)  # 75x75x3
        label = self.labels[idx]

        if self.transform:
```

```
            image = self.transform(image=image)['image']

        return image, label
```

Listing 15 – Utilisation du dataset AffectNet

6.3 Équilibrage des Classes

Pour compenser le déséquilibre (notamment le manque de *Disgust*), deux techniques sont implémentées :

6.3.1 Poids de classe dans la loss

```
def get_class_weights(dataset):
    class_counts = np.bincount(labels, minlength=7)
    weights = 1.0 / class_counts
    weights = weights / weights.sum() * len(weights)
    return torch.FloatTensor(weights)

class_weights = get_class_weights(train_dataset)
criterion = nn.CrossEntropyLoss(weight=class_weights)
```

Listing 16 – CrossEntropyLoss avec poids de classe

Le poids de chaque classe est inversement proportionnel à sa fréquence :

$$w_c = \frac{N}{N_c \times C} \tag{4}$$

où N est le nombre total d'échantillons, N_c le nombre d'échantillons de la classe c, et C le nombre de classes.

6.3.2 WeightedRandomSampler

```
def get_balanced_sampler(dataset):
    class_counts = np.bincount(labels, minlength=7)
    weights = 1.0 / class_counts
    sample_weights = weights[labels]

    sampler = WeightedRandomSampler(
        weights=sample_weights,
        num_samples=len(sample_weights),
        replacement=True
    )
    return sampler
```

Listing 17 – Sampler équilibré

Le sampler sur-échantillonne les classes minoritaires pendant l'entraînement.

7 Techniques d'Entraînement Avancées

Pour améliorer significativement les performances, notamment sur les classes difficiles (Disgust, Contempt, Fear, Angry), nous avons développé un script d'entraînement avancé (`train_advanced.py`) intégrant les techniques les plus récentes du deep learning.

7.1 Focal Loss pour le Déséquilibre des Classes

La Cross-Entropy standard traite tous les exemples de manière égale. La **Focal Loss** [?] ajoute un facteur de modulation qui réduit la contribution des exemples faciles et se concentre sur les exemples difficiles :

$$\text{FL}(p_t) = -\alpha_t(1-p_t)^\gamma \log(p_t) \tag{5}$$

où p_t est la probabilité prédite pour la vraie classe, α_t est le poids de la classe, et γ (gamma) contrôle le focus sur les exemples difficiles.

```python
class FocalLoss(nn.Module):
    def __init__(self, alpha=None, gamma=2.0, reduction='mean'):
        super().__init__()
        self.alpha = alpha  # Poids par classe
        self.gamma = gamma  # Focus parameter (2.0 recommande)
        self.reduction = reduction

    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(inputs, targets,
                                  weight=self.alpha,
                                  reduction='none')
        pt = torch.exp(-ce_loss)  # p_t
        focal_loss = ((1 - pt) ** self.gamma) * ce_loss

        if self.reduction == 'mean':
            return focal_loss.mean()
        return focal_loss
```

Listing 18 – Implémentation de la Focal Loss

Avantage : Avec $\gamma = 2$, un exemple bien classifié ($p_t = 0.9$) contribue $100\times$ moins qu'un exemple difficile ($p_t = 0.1$), permettant au modèle de se concentrer sur les émotions sous-représentées comme Disgust.

7.2 Mixup : Régularisation par Interpolation

Mixup [?] crée de nouveaux exemples d'entraînement en interpolant linéairement des paires d'images et leurs labels :

$$\tilde{x} = \lambda x_i + (1-\lambda)x_j \tag{6}$$

$$\tilde{y} = \lambda y_i + (1-\lambda)y_j \tag{7}$$

où $\lambda \sim \text{Beta}(\alpha, \alpha)$ avec $\alpha = 0.2$.

```python
def mixup_data(x, y, alpha=0.2):
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1

    batch_size = x.size(0)
    index = torch.randperm(batch_size).to(x.device)

    mixed_x = lam * x + (1 - lam) * x[index, :]
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam
```

```

def mixup_criterion(criterion, pred, y_a, y_b, lam):
    return lam * criterion(pred, y_a) + (1 - lam) * criterion(pred, y_b)
```

Listing 19 – Implémentation de Mixup

7.3 CutMix : Augmentation par Découpage

CutMix [?] est une variante de Mixup qui découpe et colle des régions rectangulaires entre images :

```
def cutmix_data(x, y, alpha=1.0):
    lam = np.random.beta(alpha, alpha)
    batch_size = x.size(0)
    index = torch.randperm(batch_size).to(x.device)

    # Calcul de la boite de decoupe
    W, H = x.size(2), x.size(3)
    cut_rat = np.sqrt(1. - lam)
    cut_w = int(W * cut_rat)
    cut_h = int(H * cut_rat)

    cx = np.random.randint(W)
    cy = np.random.randint(H)

    bbx1 = np.clip(cx - cut_w // 2, 0, W)
    bby1 = np.clip(cy - cut_h // 2, 0, H)
    bbx2 = np.clip(cx + cut_w // 2, 0, W)
    bby2 = np.clip(cy + cut_h // 2, 0, H)

    x[:, :, bbx1:bbx2, bby1:bby2] = x[index, :, bbx1:bbx2, bby1:bby2]

    # Ajuster lambda selon la surface decoupee
    lam = 1 - ((bbx2-bbx1)*(bby2-bby1) / (W*H))
    return x, y, y[index], lam
```

Listing 20 – Implémentation de CutMix

Avantage : CutMix force le modèle à utiliser des indices partiels du visage (un œil, la bouche seule) plutôt que de dépendre de l'image entière, améliorant la robustesse.

7.4 Label Smoothing

Le label smoothing régularise en remplaçant les labels hard (one-hot) par des labels soft :

$$y_{\text{smooth}} = (1 - \epsilon) \cdot y_{\text{hard}} + \frac{\epsilon}{K} \tag{8}$$

avec $\epsilon = 0.1$ et $K = 8$ classes.

```
class LabelSmoothingLoss(nn.Module):
    def __init__(self, classes, smoothing=0.1):
        super().__init__()
        self.smoothing = smoothing
        self.cls = classes

    def forward(self, pred, target):
```

```
        confidence = 1.0 - self.smoothing
        smooth_val = self.smoothing / (self.cls - 1)

        one_hot = torch.zeros_like(pred)
        one_hot.fill_(smooth_val)
        one_hot.scatter_(1, target.unsqueeze(1), confidence)

        log_prob = F.log_softmax(pred, dim=1)
        return -(one_hot * log_prob).sum(dim=1).mean()
```

Listing 21 – Application du Label Smoothing

7.5 Augmentation Avancée avec Albumentations

Nous utilisons la bibliothèque Albumentations pour des transformations plus sophistiquées :

```
import albumentations as A

train_transform = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.ShiftScaleRotate(
        shift_limit=0.1,
        scale_limit=0.15,
        rotate_limit=15,
        p=0.5
    ),
    A.OneOf([
        A.MotionBlur(blur_limit=3, p=1.0),
        A.GaussianBlur(blur_limit=3, p=1.0),
        A.GaussNoise(var_limit=(10, 50), p=1.0),
    ], p=0.3),
    A.OneOf([
        A.RandomBrightnessContrast(
            brightness_limit=0.2,
            contrast_limit=0.2,
            p=1.0
        ),
        A.CLAHE(clip_limit=2.0, p=1.0),
    ], p=0.5),
    A.CoarseDropout(
        max_holes=1,
        max_height=12,
        max_width=12,
        fill_value=0,
        p=0.25
    ),
])
```

Listing 22 – Transformations Albumentations

TABLE 6 – Nouvelles transformations Albumentations

Transformation	**Justification**
MotionBlur / GaussianBlur	Simule le flou de mouvement en conditions réelles
GaussNoise	Robustesse au bruit de capteur
CLAHE	Améliore le contraste local, meilleur que l'égalisation standard
CoarseDropout	Similaire à Cutout, force la redondance des features

7.6 Gradient Accumulation

Pour simuler des batch sizes plus grands sur GPU avec mémoire limitée :

```
ACCUMULATION_STEPS = 4  # Effective batch = 32 * 4 = 128

for i, (inputs, targets) in enumerate(train_loader):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss = loss / ACCUMULATION_STEPS
    loss.backward()

    if (i + 1) % ACCUMULATION_STEPS == 0:
        torch.nn.utils.clip_grad_norm_(
            model.parameters(), max_norm=1.0
        )
        optimizer.step()
        optimizer.zero_grad()
```

Listing 23 – Gradient Accumulation

Avantage : Un batch effectif de 128 images stabilise les gradients et améliore la convergence.

7.7 Cosine Annealing avec Warm Restarts

Scheduler qui suit une courbe cosinus avec redémarrages périodiques :

```
scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer,
    T_0=10,      # Premiere periode de 10 epochs
    T_mult=2,    # Periodes suivantes x2 (10, 20, 40...)
    eta_min=1e-6
)
```

Listing 24 – CosineAnnealingWarmRestarts

Learning Rate suit une courbe cosinus :
$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\frac{T_{\text{cur}}}{T_i}\pi))$

FIGURE 2 – Le learning rate diminue selon une courbe cosinus puis remonte à chaque restart

8 Application Étendue avec MediaPipe

Pour améliorer la détection des émotions difficiles et enrichir le panel d'emojis affichables, nous avons intégré **MediaPipe**, la bibliothèque de Google pour l'analyse faciale et gestuelle en temps réel.

8.1 MediaPipe Face Mesh : Analyse des Landmarks Faciaux

Face Mesh détecte 468 points de repère 3D sur le visage, permettant une analyse fine des caractéristiques faciales :

```
import mediapipe as mp

mp_face_mesh = mp.solutions.face_mesh
face_mesh = mp_face_mesh.FaceMesh(
    max_num_faces=1,
    refine_landmarks=True,  # Points supplementaires iris
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5
)
```

Listing 25 – Initialisation de Face Mesh

8.1.1 Features Extraites

```
class FacialAnalyzer:
    # Indices des landmarks cles
    LEFT_EYE = [33, 160, 158, 133, 153, 144]
    RIGHT_EYE = [362, 385, 387, 263, 373, 380]
    MOUTH = [61, 291, 0, 17, 78, 308]
    LEFT_EYEBROW = [70, 63, 105, 66, 107]
    RIGHT_EYEBROW = [336, 296, 334, 293, 300]

    def analyze(self, landmarks):
        features = {}

        # Ouverture des yeux (Eye Aspect Ratio)
        features['left_eye_open'] = self._eye_aspect_ratio(
            [landmarks[i] for i in self.LEFT_EYE]
        )
        features['right_eye_open'] = self._eye_aspect_ratio(
            [landmarks[i] for i in self.RIGHT_EYE]
        )

        # Ouverture de la bouche
        features['mouth_open'] = self._mouth_aspect_ratio(landmarks)

        # Position des sourcils
        features['brow_raise'] = self._brow_raise(landmarks)
        features['brow_squeeze'] = self._brow_squeeze(landmarks)

        # Sourire (ratio largeur/hauteur bouche)
        features['smile'] = self._smile_ratio(landmarks)

        return features

```

```
    def _eye_aspect_ratio(self, eye_points):
        # Ratio vertical/horizontal pour detecter clignement
        vertical = np.linalg.norm(
            np.array(eye_points[1]) - np.array(eye_points[5])
        )
        horizontal = np.linalg.norm(
            np.array(eye_points[0]) - np.array(eye_points[3])
        )
        return vertical / (horizontal + 1e-6)
```

Listing 26 – Classe FacialAnalyzer pour l'extraction de features

8.1.2 Boost des Émotions Difficiles

Les features faciales sont utilisées pour renforcer la détection des émotions que le CNN a du mal à identifier :

```
EMOTION_BOOST = {
    'Angry': 1.4,      # Boost de 40%
    'Disgust': 1.5,    # Boost de 50%
    'Fear': 1.3,       # Boost de 30%
    'Contempt': 1.4,   # Boost de 40%
    'Neutral': 0.85    # Reduction de 15%
}

def boost_emotions(probs, features):
    boosted = probs.copy()

    # Boost Angry si sourcils fronces
    if features.get('brow_squeeze', 0) > 0.6:
        boosted[ANGRY_IDX] *= 1.3

    # Boost Disgust si nez fronce
    if features.get('nose_wrinkle', 0) > 0.5:
        boosted[DISGUST_IDX] *= 1.4

    # Boost Fear si yeux grands ouverts
    avg_eye = (features['left_eye_open'] +
               features['right_eye_open']) / 2
    if avg_eye > 0.4:
        boosted[FEAR_IDX] *= 1.2

    return boosted / boosted.sum()  # Re-normaliser
```

Listing 27 – Système de boost des émotions

8.2 Extension du Mapping Emoji

Au-delà des 8 émotions de base, les features faciales permettent d'afficher des emojis plus nuancés :

```
EXTENDED_EMOJI_MAP = {
    # Emotions de base
    'Happy': '    ',
    'Sad': '    ',
    'Angry': '    ',
    'Surprise': '    ',
```

```
    'Fear': '    ',
    'Disgust': '    ',
    'Neutral': '    ',
    'Contempt': '    ',

    # Combinaisons basees sur features
    'very_happy': '    ',      # Sourire large
    'laughing': '    ',        # Bouche ouverte + sourire
    'thinking': '    ',        # Un sourcil leve
    'sleepy': '    ',          # Yeux fermes
    'wink': '    ',            # Un oeil ferme
    'kiss': '    ',            # Bouche en O
    'love': '    ',            # Coeurs si smile fort
    'cool': '    ',            # Neutre + confiant
    'skeptical': '    ',       # Sourcil asymetrique
}

def get_extended_emoji(emotion, features):
    # Verifier conditions speciales
    if features['left_eye_open'] < 0.15:
        if features['right_eye_open'] > 0.25:
            return EXTENDED_EMOJI_MAP['wink']

    if features['smile'] > 0.7 and features['mouth_open'] > 0.4:
        return EXTENDED_EMOJI_MAP['laughing']

    if emotion == 'Happy' and features['smile'] > 0.8:
        return EXTENDED_EMOJI_MAP['very_happy']

    return EXTENDED_EMOJI_MAP.get(emotion, '   ')
```

Listing 28 – Mapping emoji étendu

8.3 MediaPipe Hands : Détection des Gestes

Pour enrichir l'interaction, nous avons ajouté la détection des gestes de la main :

```
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(
    static_image_mode=False,
    max_num_hands=2,
    min_detection_confidence=0.7,
    min_tracking_confidence=0.5
)
```

Listing 29 – Initialisation de MediaPipe Hands

8.3.1 Gestes Reconnus

```
class HandRecognizer:
    def recognize_gesture(self, landmarks):
        # Extraire les etats des doigts (leve/baisse)
        fingers = self._get_finger_states(landmarks)
        thumb, index, middle, ring, pinky = fingers

        # Pouce leve
        if thumb and not any([index, middle, ring, pinky]):
```

```
            return 'thumbs_up', '    '

        # Signe de paix (V)
        if index and middle and not ring and not pinky:
            return 'peace', '      '

        # Signe OK
        if self._is_ok_gesture(landmarks):
            return 'ok', '    '

        # Rock (cornes)
        if index and pinky and not middle and not ring:
            return 'rock', '    '

        # Shaka (pouce + auriculaire)
        if thumb and pinky and not index and not middle:
            return 'shaka', '    '

        # Poing ferme
        if not any(fingers):
            return 'fist', '   '

        # Main ouverte (tous les doigts)
        if all(fingers):
            return 'wave', '    '

        # Pointer
        if index and not middle and not ring and not pinky:
            return 'point', '    '

        return None, None

    def _get_finger_states(self, landmarks):
        # Comparer position des bouts vs articulations
        thumb = landmarks[4].y < landmarks[3].y
        index = landmarks[8].y < landmarks[6].y
        middle = landmarks[12].y < landmarks[10].y
        ring = landmarks[16].y < landmarks[14].y
        pinky = landmarks[20].y < landmarks[18].y
        return [thumb, index, middle, ring, pinky]
```

Listing 30 – Classe HandRecognizer

TABLE 7 – Gestes reconnus et emojis correspondants

Geste	**Description**	**Emoji**
Thumbs Up	Pouce levé seul	
Peace	Index et majeur levés	
OK	Pouce et index formant un cercle	
Rock	Index et auriculaire levés	
Shaka	Pouce et auriculaire	
Wave	Tous les doigts levés	
Fist	Poing fermé	
Point	Index seul levé	

9 Optimisations de l'Inférence (app_v3.py)

La version 3 de l'application intègre plusieurs optimisations pour améliorer la précision et la stabilité des prédictions.

9.1 Test-Time Augmentation (TTA)

Le TTA applique plusieurs transformations à l'image d'entrée et moyenne les prédictions :

```python
class EmotionClassifier:
    def predict_with_tta(self, face_roi, num_augmentations=3):
        predictions = []

        # Prediction originale
        predictions.append(self._predict_single(face_roi))

        # Flip horizontal
        flipped = cv2.flip(face_roi, 1)
        predictions.append(self._predict_single(flipped))

        # Legeres variations de luminosite
        for _ in range(num_augmentations - 2):
            factor = np.random.uniform(0.9, 1.1)
            adjusted = np.clip(face_roi * factor, 0, 255)
            predictions.append(
                self._predict_single(adjusted.astype(np.uint8))
            )

        # Moyenne des predictions
        return np.mean(predictions, axis=0)
```

Listing 31 – Implémentation du TTA

Avantage : Le TTA réduit la variance des prédictions et améliore la robustesse aux petites perturbations.

9.2 Prétraitement CLAHE

Contrairement à l'égalisation d'histogramme globale, CLAHE (Contrast Limited Adaptive Histogram Equalization) travaille sur des régions locales :

```python
def preprocess_face(self, face_roi):
    # Redimensionner
    face = cv2.resize(face_roi, (48, 48))

    # CLAHE pour le contraste local
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(4, 4))
    face = clahe.apply(face)

    # Normalisation
    face = face.astype(np.float32) / 255.0

    return face
```

Listing 32 – Application de CLAHE

TABLE 8 – Comparaison des méthodes d'égalisation

Méthode	Avantages	Inconvénients
Histogramme global	Simple, rapide	Peut sur-amplifier le bruit
CLAHE	Préserve les détails locaux, contrôle le contraste	Légèrement plus lent

9.3 Lissage Temporel Amélioré

La version 3 utilise un lissage exponentiel plus sophistiqué :

```
SMOOTHING_WINDOW = 7

def get_smoothed_prediction(self, current_probs):
    self.prediction_history.append(current_probs)

    if len(self.prediction_history) < 2:
        return current_probs

    # Poids exponentiels (frames recentes plus importantes)
    n = len(self.prediction_history)
    weights = np.exp(np.linspace(-1, 0, n))
    weights = weights / weights.sum()

    smoothed = np.zeros(8)
    for i, probs in enumerate(self.prediction_history):
        smoothed += weights[i] * probs

    return smoothed
```

Listing 33 – Lissage temporel avec pondération exponentielle

9.4 Architecture Modulaire

L'application V3 sépare clairement les responsabilités :

```
class EmotionClassifier:
    """Gestion du modele CNN et predictions"""
    pass

class FacialAnalyzer:
    """Analyse des landmarks faciaux avec MediaPipe"""
    pass

class HandRecognizer:
    """Detection des gestes de la main"""
    pass

class EmotionApp:
    """Application principale orchestrant les composants"""
    def __init__(self):
        self.classifier = EmotionClassifier()
        self.facial_analyzer = FacialAnalyzer()
        self.hand_recognizer = HandRecognizer()

    def process_frame(self, frame):
```

```
        # 1. Detecter visage
        # 2. Analyser features faciales
        # 3. Predire emotion avec CNN + TTA
        # 4. Boost avec features
        # 5. Detecter gestes
        # 6. Afficher resultats
        pass
```

Listing 34 – Architecture modulaire de app_v3.py

9.5 Datasets Alternatifs

Pour des performances encore meilleures, d'autres datasets sont supportés :

TABLE 9 – Comparaison des datasets d'émotions faciales

Dataset	Images	Résolution	Classes	Accès
FER2013	35,887	48×48 (Gray)	7	Gratuit (Kaggle)
FER+	35,887	48×48 (Gray)	8	Gratuit (GitHub)
Balanced AffectNet	**41,008**	**75×75 (RGB)**	**8**	**Gratuit (Kaggle)**
AffectNet (original)	450,000	Variable	8	Demande requise
RAF-DB	30,000	100×100	7	Demande requise

```
class AffectNetDataset(Dataset):
    # Mapping AffectNet vers les 7 classes FER
    AFFECTNET_TO_FER = {
        0: 6,  # Neutral
        1: 3,  # Happy
        2: 4,  # Sad
        3: 5,  # Surprise
        4: 2,  # Fear
        5: 1,  # Disgust
        6: 0,  # Anger
        7: 6,  # Contempt -> Neutral
    }

    def __getitem__(self, idx):
        image = Image.open(self.images[idx]).convert('L')
        image = image.resize((48, 48))
        label = self.AFFECTNET_TO_FER[self.labels[idx]]
        return image, label
```

Listing 35 – Support multi-datasets

10 Résultats Attendus

10.1 Amélioration de la Précision

TABLE 10 – Amélioration attendue de la précision avec Balanced AffectNet

Configuration	**Précision estimée**	**Gain**
FER2013 baseline (grayscale 48×48)	60-65%	-
Balanced AffectNet (RGB 75×75)	70-75%	+10%
+ Data augmentation RGB	73-78%	+3%
+ Mixup/CutMix	76-80%	+3%
+ Label Smoothing	78-82%	+2%
+ TTA à l'inférence	80-84%	+2%
Total avec optimisations	**80-84%**	**+20-24%**

10.2 Amélioration des Classes Difficiles

TABLE 11 – Impact attendu sur les classes (avec dataset équilibré)

Émotion	**FER2013 Baseline**	**Balanced AffectNet**	**Gain**
Angry	55%	75%	+20%
Disgust	35%	72%	+37%
Fear	45%	70%	+25%
Contempt	40%	68%	+28%

Note : L'amélioration majeure sur Disgust et Contempt est due au dataset parfaitement équilibré et aux images RGB de meilleure qualité.

10.3 Amélioration de l'Expérience Utilisateur

- **Stabilité** : Le lissage temporel amélioré élimine le *flickering*
- **Robustesse** : CLAHE gère mieux les variations d'éclairage que l'égalisation standard
- **Précision TTA** : Le test-time augmentation réduit la variance des prédictions
- **Emojis étendus** : Plus de 15 emojis possibles grâce à l'analyse des features faciales
- **Gestes de la main** : 8 gestes reconnus avec emojis correspondants
- **Lisibilité** : Les couleurs et barres de progression améliorent la compréhension
- **Réactivité** : Les paramètres de détection optimisés améliorent la fluidité

11 Conclusion

Ce projet a permis de développer un système complet et avancé de reconnaissance d'émotions faciales en temps réel. Les principales contributions sont :

1. **Architecture CNN optimisée** avec double convolution, dropout progressif, et Global Average Pooling

2. **Pipeline d'entraînement avancé** incluant :
   — Focal Loss pour le déséquilibre des classes
   — Mixup et CutMix pour la régularisation
   — Label Smoothing pour la généralisation
   — Augmentations avancées avec Albumentations
3. **Intégration MediaPipe** pour :
   — Analyse des 468 landmarks faciaux (Face Mesh)
   — Détection de 8 gestes de la main
   — Boost contextuel des émotions difficiles
4. **Application temps réel optimisée (V3)** avec :
   — Test-Time Augmentation (TTA)
   — Prétraitement CLAHE
   — Lissage temporel exponentiel
   — Mapping emoji étendu (15+ emojis)
5. **Support multi-datasets** permettant d'utiliser FER+, AffectNet, ou RAF-DB

11.1 Perspectives

Pour aller plus loin, les améliorations suivantes pourraient être envisagées :
— Utilisation de transfer learning (VGGFace, ResNet pré-entraîné sur visages)
— Architecture avec attention mechanism (Transformer, CBAM)
— Détection multi-tâches (émotion + âge + genre)
— Analyse de la valence et de l'arousal (modèle dimensionnel)
— Déploiement sur edge devices (quantification INT8, pruning)
— Intégration d'un modèle audio pour l'analyse multimodale

A Structure du Projet

```
Final_project/
|-- app.py                    # Application temps reel (version initiale)
|-- app_extended.py           # Version avec MediaPipe Face Mesh
|-- app_extended_v2.py        # Version avec gestes de la main
|-- app_v3.py                 # Version optimisee (TTA, CLAHE, modulaire)
|-- model.py                  # Architecture CNN (RGB 75x75)
|-- train_affectnet.py        # Entrainement sur Balanced AffectNet
|-- train.py                  # Script d'entrainement (legacy FER2013)
|-- dataset_affectnet.py      # Dataset Balanced AffectNet
|-- dataset.py                # Dataset FER2013 (legacy)
|-- download_datasets.py      # Script de telechargement des datasets
|-- emotion_model.pth         # Poids du modele entraine
|-- emotion_model_best.pth    # Meilleur modele (early stopping)
|-- data/
|   +-- affectnet/            # Dataset Balanced AffectNet
|       |-- train/
|       |   |-- Anger/
|       |   |-- Contempt/
|       |   |-- Disgust/
```

```
|       |   |-- Fear/
|       |   |-- Happy/
|       |   |-- Neutral/
|       |   |-- Sad/
|       |   +-- Surprise/
|       |-- val/
|       +-- test/
|-- report/
|   +-- report.tex           # Ce rapport
+-- README.md
```

B Références

1. Goodfellow, I. J., et al. (2013). "Challenges in representation learning : A report on three machine learning contests." *ICML Workshop*.
2. Barsoum, E., et al. (2016). "Training Deep Networks for Facial Expression Recognition with Crowd-Sourced Label Distribution." *ACM ICMI*.
3. Mollahosseini, A., et al. (2017). "AffectNet : A Database for Facial Expression, Valence, and Arousal Computing in the Wild." *IEEE Trans. Affective Computing*.
4. Li, S., et al. (2017). "Reliable Crowdsourcing and Deep Locality-Preserving Learning for Expression Recognition in the Wild." *CVPR*.
5. He, K., et al. (2015). "Delving Deep into Rectifiers : Surpassing Human-Level Performance on ImageNet Classification." *ICCV*.
6. Lin, T. Y., et al. (2017). "Focal Loss for Dense Object Detection." *ICCV*.
7. Zhang, H., et al. (2018). "mixup : Beyond Empirical Risk Minimization." *ICLR*.
8. Yun, S., et al. (2019). "CutMix : Regularization Strategy to Train Strong Classifiers with Localizable Features." *ICCV*.
9. Lugaresi, C., et al. (2019). "MediaPipe : A Framework for Building Perception Pipelines." *arXiv preprint arXiv :1906.08172*.
10. Buslaev, A., et al. (2020). "Albumentations : Fast and Flexible Image Augmentations." *Information*, 11(2), 125.