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 sept émotions (colère, dégoût, peur, joie, tristesse, surprise, neutralité) à partir d'un flux vidéo webcam.

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 7 émotions faciales
— 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 : FER2013

Le dataset FER2013 (Facial Expression Recognition 2013) est le dataset de référence pour cette tâche. Il contient :

— 35 887 images de visages en niveaux de gris
— Résolution : 48×48 pixels
— 7 classes d'émotions équilibrées (sauf "Disgust")

2.2 Modèle Initial (`model.py`)

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

```
class FaceEmotionCNN(nn.Module):
    def __init__(self):
        super(FaceEmotionCNN, self).__init__()
        # Bloc Convolutif 1
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        # Bloc Convolutif 2
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        # Bloc Convolutif 3
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.5)

        # Couches Fully Connected
        # Image 48x48, apres 3 pooling: 48->24->12->6
        self.fc1 = nn.Linear(128 * 6 * 6, 512)
        self.fc2 = nn.Linear(512, 7)  # 7 emotions

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
```

```
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))

        x = x.view(-1, 128 * 6 * 6)  # Flatten
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Listing 1 – Architecture CNN initiale

2.2.1 Analyse de l'architecture initiale

TABLE 1 – Dimensions à travers le réseau initial

Couche	Entrée	Sortie	Paramètres
Conv1 + BN + Pool	$1 \times 48 \times 48$	$32 \times 24 \times 24$	352
Conv2 + BN + Pool	$32 \times 24 \times 24$	$64 \times 12 \times 12$	18,560
Conv3 + BN + Pool	$64 \times 12 \times 12$	$128 \times 6 \times 6$	73,984
Flatten	$128 \times 6 \times 6$	4,608	0
FC1	4,608	512	2,359,808
FC2	512	7	3,591
Total			$\approx$ **2.46M**

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.csv', transform=transform)
train_loader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

```
15
16 # Modele et optimiseur
17 model = FaceEmotionCNN().to(device)
18 criterion = nn.CrossEntropyLoss()
19 optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
20
21 # Boucle d'entrainement
22 for epoch in range(EPOCHS):
23     running_loss = 0.0
24     for inputs, labels in train_loader:
25         inputs, labels = inputs.to(device), labels.to(device)
26         optimizer.zero_grad()
27         outputs = model(inputs)
28         loss = criterion(outputs, labels)
29         loss.backward()
30         optimizer.step()
31         running_loss += loss.item()
32
33     print(f"Epoch {epoch+1}, Loss: {running_loss/len(train_loader)}")
34
35 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

```
1 while True:
2     ret, frame = cap.read()
3     gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
4     faces = face_cascade.detectMultiScale(gray_frame,
5                                           scaleFactor=1.3,
6                                           minNeighbors=5)
7
8     for (x, y, w, h) in faces:
9         cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
10        roi_gray = gray_frame[y:y+h, x:x+w]
11
12        # Preprocessing
13        roi_tensor = data_transform(roi_gray).unsqueeze(0).to(device)
14
15        # 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)

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

        # Fully Connected avec dropout progressif
        self.fc1 = nn.Linear(512, 256)
        self.bn_fc1 = nn.BatchNorm1d(256)
        self.dropout_fc1 = nn.Dropout(0.5)

        self.fc2 = nn.Linear(256, 128)
        self.bn_fc2 = nn.BatchNorm1d(128)
        self.dropout_fc2 = nn.Dropout(0.4)

        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.

[draw, minimum width=3cm, minimum height=2cm] at (0,0) Image sombre ; at (2.5,0) $\rightarrow$; [draw, minimum width=3cm, minimum height=2cm] at (5,0) Histogramme équalisé ;

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 Limitations de FER2013

Le dataset FER2013 présente plusieurs limitations connues :

1. **Basse résolution** : Images de seulement 48×48 pixels
2. **Annotations bruitées** : Labellisation par crowd-sourcing avec erreurs
3. **Déséquilibre des classes** :

Émotion	Nombre d'images	Pourcentage
Angry	4,953	13.8%
Disgust	**547**	**1.5%**
Fear	5,121	14.3%
Happy	8,989	25.0%
Sad	6,077	16.9%
Surprise	4,002	11.2%
Neutral	6,198	17.3%

TABLE 5 – Distribution des classes dans FER2013. "Disgust" est très sous-représentée.

4. **Images en niveaux de gris uniquement**
5. **Conditions non-contrôlées** : Poses, éclairages, occlusions variables

6.2 FER+ : Annotations Corrigées

Microsoft Research a publié FER+ qui améliore FER2013 avec :
— 10 annotateurs par image (au lieu de 1)
— Vote majoritaire pour le label final
— Possibilité d'utiliser des "soft labels" (distribution de probabilités)
— Ajout de la classe "Contempt" (mépris)
— Filtrage des images ambiguës

```python
class FERPlusDataset(Dataset):
    EMOTIONS = ['neutral', 'happiness', 'surprise', 'sadness',
                'anger', 'disgust', 'fear', 'contempt',
                'unknown', 'NF']

    def __getitem__(self, idx):
        votes = self.votes[idx]  # 10 votes par image

        if self.use_soft_labels:
            # Distribution de probabilites
            soft_label = np.zeros(7, dtype=np.float32)
            for i, emotion in enumerate(self.EMOTIONS[:8]):
                if emotion in self.FERPLUS_TO_FER:
                    soft_label[self.FERPLUS_TO_FER[emotion]] += votes[i]
            soft_label = soft_label / soft_label.sum()
            return pixels, torch.tensor(soft_label)
        else:
            # Vote majoritaire
            label = np.argmax(votes[:7])
```

```
            return pixels, label
```

Listing 15 – Utilisation de FER+ avec soft labels

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.

6.4 Datasets Alternatifs

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

TABLE 6 – Comparaison des datasets d'émotions faciales

Dataset	Images	Résolution	Classes	Accès
FER2013	35,887	48×48	7	Gratuit (Kaggle)
FER+	35,887	48×48	8	Gratuit (GitHub)
AffectNet	450,000	Variable	8	Demande requise
RAF-DB	30,000	100×100	7	Demande requise
ExpW	91,793	Variable	7	Gratuit

```python
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 18 – Support multi-datasets

7 Résultats Attendus

7.1 Amélioration de la Précision

TABLE 7 – Amélioration attendue de la précision

Configuration	Précision estimée	Gain
Architecture initiale + FER2013	60-65%	-
Architecture améliorée + FER2013	65-70%	+5%
+ Data augmentation	68-72%	+3%
+ FER+ annotations	70-75%	+3%
+ Class balancing	72-76%	+2%
Total avec optimisations	**72-76%**	**+12-16%**

Note : Ces chiffres sont des estimations basées sur la littérature. Les résultats réels dépendent de l'entraînement.

7.2 Amélioration de l'Expérience Utilisateur

— **Stabilité** : Le lissage temporel élimine le "flickering"
— **Robustesse** : L'égalisation d'histogramme gère les variations d'éclairage
— **Lisibilité** : Les couleurs et barres de progression améliorent la compréhension
— **Réactivité** : Les paramètres de détection optimisés améliorent la fluidité

8 Conclusion

Ce projet a permis de développer un système complet 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 robuste** incluant data augmentation, early stopping, et learning rate scheduling
3. **Gestion du déséquilibre des classes** via weighted loss et balanced sampling
4. **Application temps réel stable** grâce au lissage temporel et à la normalisation d'éclairage
5. **Support multi-datasets** permettant d'utiliser FER+, AffectNet, ou RAF-DB

8.1 Perspectives

Pour aller plus loin, les améliorations suivantes pourraient être envisagées :
— Utilisation de transfer learning (VGGFace, ResNet pré-entraîné)
— Architecture avec attention mechanism
— Détection multi-tâches (émotion + âge + genre)
— Déploiement sur edge devices (quantification, pruning)

A Structure du Projet

```
Final_project/
 app.py                 # Application temps réel
 model.py               # Architecture CNN
 train.py               # Script d'entraînement
 dataset.py             # Dataset FER2013 original
 dataset_improved.py    # Datasets améliorés (FER+, AffectNet, RAF-DB)
 download_datasets.py   # Script de téléchargement des datasets
 emotion_model.pth      # Poids du modèle entraîné
 data/
    fer2013.csv        # Dataset FER2013
    fer2013new.csv     # Labels FER+ (optionnel)
 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*.